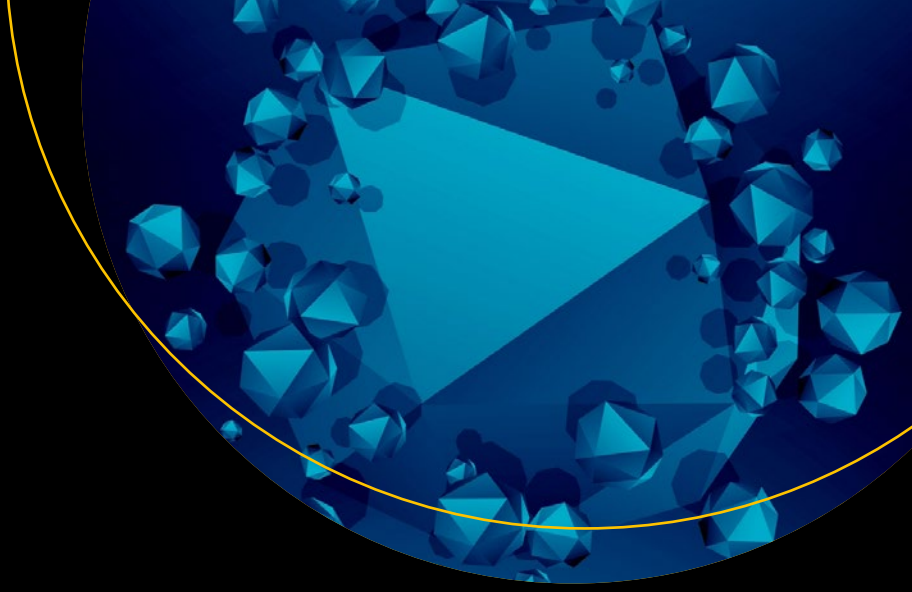


PyTorch Recipes

A Problem-Solution Approach

Pradeepta Mishra

Apress®

PyTorch Recipes

A Problem-Solution Approach

Pradeepta Mishra

Apress®

PyTorch Recipes

Pradeepta Mishra
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-4257-5 ISBN-13 (electronic): 978-1-4842-4258-2
https://doi.org/10.1007/978-1-4842-4258-2

Library of Congress Control Number: 2018968538

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Matthew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4257-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I would like to dedicate this book to my dear parents, my lovely wife, Prajna, and my daughter, Priyanshi (Aarya). This work would not have been possible without their inspiration, support, and encouragement.

Table of Contents

About the Author

Pradeepta Mishra is a data scientist and artificial intelligence architect. He currently heads NLP, ML, and AI initiatives at Lymbyc, a leading-edge innovator in AI and machine learning based out of Bangalore, India. He has expertise in designing artificial intelligence systems for performing tasks such as understanding natural language and recommendations based on natural language processing. He has filed three patents as an inventor and has authored and co-authored two books: *R Data Mining Blueprints* (Packt Publishing, 2016) and *R: Mining Spatial, Text, Web, and Social Media Data* (Packt Publishing, 2017). There are two courses available on Udemy based on these books.

Pradeepta presented a keynote talk on the application of bidirectional LSTM for time series forecasting at the 2018 Global Data Science Conference 2018. He delivered a TEDx talk titled "Can Machines Think?", a session on the power of artificial intelligence in transforming industries and changing job roles across industries. He has also delivered more than 150 tech talks on data science, machine learning, and artificial intelligence at various meetups, technical institutions, universities, and community forums.

He is on LinkedIn at www.linkedin.com/in/pradeepta/.

About the Technical Reviewer

Shivendra Upadhyay has more than eight years of experience working for consulting and software firms. He has worked in data science with KPMG for more than three years, and has a firm grasp of machine learning and data science tools and technologies.

Acknowledgments

I would like to thank my wife, Prajna, for her continuous inspiration and support, and sacrificing her weekends just to sit alongside me to help me in completing the book; my daughter, Aarya, for being patient all through my writing time; my father, for his eagerness to know how many chapters I had completed.

A big thank you to Nikhil, Celestin, and Divya, for fast-tracking the whole process and helping me and guiding me in the right direction.

I would like to thank my bosses, Ashish and Saty, for always being supportive of my initiatives in the AI and ML journey, and their continuous motivation and inspiration in writing in the AI space.

Introduction

Development of artificial intelligent products and solutions has recently become a norm; hence, the demand for graph theory–based computational frameworks is on the rise. Making the deep learning models work in real-life applications is possible when the modeling framework is dynamic, flexible, and adaptable to other frameworks.

PyTorch is a recent entrant to the league of graph computation tools/programming languages. Addressing the limitations of previous frameworks, PyTorch promises a better user experience in the deployment of deep learning models, and the creation of advanced models using a combination of convolutional neural networks, recurrent neural networks, LSTMs, and deep neural networks.

PyTorch was created by Facebook's Artificial Intelligence Research division, which seeks to make the model development process simple, straightforward, and dynamic, so that developers do not have to worry about declaring objects before compiling and executing the model. It is based on the Torch framework and is an extension of Python.

This book is intended for data scientists, natural language processing engineers, artificial intelligence solution developers, existing practitioners working on graph computation frameworks, and researchers of graph theory. This book will get you started with understanding tensor basics, computation, performing arithmetic-based operations, matrix algebra, and statistical distribution-based operations using the PyTorch framework.

Chapters 3 and 4 provide detailed descriptions on neural network basics. Advanced neural networks, such as convolutional neural networks, recurrent neural networks, and LSTMs are explored. Readers will be able to implement these models using PyTorch functions.

Chapters 5 and 6 discuss fine-tuning the models, hyper parameter tuning, and the refinement of existing PyTorch models in production. Readers learn how to choose the hyper parameters to fine-tune the model.

In Chapter 7, natural language processing is explained. The deep learning models and their applications in natural language processing and artificial intelligence is one of the most demanding skill sets in the industry. Readers will be able to benchmark the execution and performance of PyTorch implementation in deep learning models to execute and process natural language. They will be able to compare PyTorch with other graph computation–based deep learning programming tools.

CHAPTER 1

Introduction to PyTorch, Tensors, and Tensor Operations

PyTorch has been evolving as a larger framework for writing dynamic models. Because of that, it is very popular among data scientists and data engineers deploying large-scale deep learning frameworks. This book provides a structure for the experts in terms of handling activities while working on a practical data science problem. As evident from applications that we use in our day-to-day lives, there are layers of intelligence embedded with the product features. Those features are enabled to provide a better experience and better services to the user.

The world is moving toward artificial intelligence. There are two main components of it: deep learning and machine learning. Without deep learning and machine learning, it is impossible to visualize artificial intelligence.

PyTorch is the most optimized high-performance tensor library for computation of deep learning tasks on GPUs (graphics processing units) and CPUs (central processing units). The main purpose of PyTorch is to enhance the performance of algorithms in large-scale computing

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_1

environments. PyTorch is a library based on Python and the Torch tool provided by Facebook's Artificial Intelligence Research group, which performs scientific computing.

NumPy-based operations on a GPU are not efficient enough to process heavy computations. Static deep learning libraries are a bottleneck for bringing flexibility to computations and speed. From a practitioner's point of view, PyTorch tensors are very similar to the N-dimensional arrays of a NumPy library based on Python. The PyTorch library provides bridge options for moving a NumPy array to a tensor array, and vice versa, in order to make the library flexible across different computing environments.

The use cases where it is most frequently used include natural language processing, image processing, computer vision, social media data analysis, and sensor data processing. Although PyTorch provides a large collection of libraries and modules for computation, three modules are very prominent.

- *Autograd*. This module provides functionality for automatic differentiation of tensors. A recorder class in the program remembers the operations and retrieves those operations with a trigger called *backward* to compute the gradients. This is immensely helpful in the implementation of neural network models.
- *Optim*. This module provides optimization techniques that can be used to minimize the error function for a specific model. Currently, PyTorch supports various advanced optimization methods, which includes Adam, stochastic gradient descent (SGD), and more.

- *NN*. NN stands for *neural network* model. Manually defining the functions, layers, and further computations using complete tensor operations is very difficult to remember and execute. We need functions that automate the layers, activation functions, loss functions, and optimization functions and provides a layer defined by the user so that manual intervention can be reduced. The NN module has a set of built-in functions that automates the manual process of running a tensor operation.

Industries in which artificial intelligence is applied include banking, financial services, insurance, health care, manufacturing, retail, clinical trials, and drug testing. Artificial intelligence involves classifying objects, recognizing the objects to detecting fraud, and so forth. Every learning system requires three things: input data, processing, and an output layer. Figure 1-1 explains the relationship between these three topics. If the performance of any learning system improves over time by learning from new examples or data, it is called a *machine learning system*. When a machine learning system becomes too difficult to reflect reality, it requires a deep learning system.

In a deep learning system, more than one layer of a learning algorithm is deployed. In machine learning, we think of supervised, unsupervised, semisupervised, and reinforcement learning systems. A supervised machine-learning algorithm is one where the data is labeled with classes or tagged with outcomes. We show the machine the input data with corresponding tags or labels. The machine identifies the relationship with a function. Please note that this function connects the input to the labels or tags.

In unsupervised learning, we show the machine only the input data and ask the machine to group the inputs based on association, similarities or dissimilarities, and so forth.

In semisupervised learning, we show the machine input features and labeled data or tags; however we ask the machine to predict the untagged outcomes or labels.

In reinforcement learning, we introduce a reward and penalty mechanism, where every correct action is rewarded and every incorrect action is penalized.

In all of these examples of machine learning algorithms, we assume that the dataset is small, because getting massive amounts of tagged data is a challenge, and it takes a lot of time for machine learning algorithms to process large-scale matrix computations. Since machine learning algorithms are not scalable for massive datasets, we need deep learning algorithms.

Figure 1-1 shows the relationships among artificial intelligence, machine learning, and deep learning. Natural language is an important part of artificial intelligence. We need to develop systems that understand natural language and provide responses to the agent. Let's take an example of machine translation, where a sentence in language 1 (French) can be converted to language 2 (English), and vice versa. To develop such a system, we need a large collection of English-French bilingual sentences. The corpus requirement is very large, as all the language nuances need to be covered by the model.

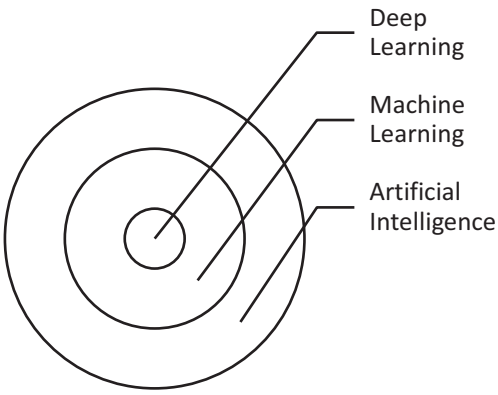


Figure 1-1. *Relationships among ML, DL, and AI*

After preprocessing and feature creation, you can observe hundreds of thousands of features that need to be computed to produce output. If we train a machine learning supervised model, it would take months to run and to produce output. To achieve scalability in this task, we need deep learning algorithms, such as a recurrent neural network. This is how the artificial intelligence is connected to deep learning and machine learning.

There are various challenges in deploying deep learning models that require large volumes of labeled data, faster computing machines, and intelligent algorithms. The success of any deep learning system requires good labeled data and better computing machines because the smart algorithms are already available.

The following are various use cases that require deep learning implementation:

- Speech recognition
- Video analysis
- Anomaly detection from videos
- Natural language processing
- Machine translation
- Speech-to-text conversion

The development of the NVIDIA GPU computing for processing large-scale data is another path-breaking innovation. The programming language that is required to run in a GPU environment requires a different programming framework. Two major frameworks are very popular for implementing graphical computing: TensorFlow and PyTorch. In this book, we discuss PyTorch as a framework to implement data science algorithms and make inferences.

The major frameworks for graph computations include PyTorch, TensorFlow, and MXNet. PyTorch and TensorFlow compete with each other in neurocomputations. TensorFlow and PyTorch are equally good

in terms of performance; however, the real differences are known only when we benchmark a particular task. Concept-wise there are certain differences.

- In TensorFlow, we have to define the tensors, initialize the session, and keep placeholders for the tensor objects; however, we do not have to do these operations in PyTorch.
- In TensorFlow, let's consider sentiment analysis as an example. Input sentences are tagged with positive or negative tags. If the input sentence's length is not equal, then we set the maximum sentence length and add zero to make the length of other sentences equal, so that the recurrent neural network can function; however, this is a built-in functionality in PyTorch, so we do not have to define the length of the sentences.
- In PyTorch, the debugging is much easier and simpler, but it is a difficult task in TensorFlow.
- In terms of data visualization, model deployment definitely better in TensorFlow; however, PyTorch is evolving and we expect to eventually see the same functionality in the future.

TensorFlow has definitely undergone many changes to reach a stable state. PyTorch is just entering the game, so it will take some time to realize the full potential of this tool.

What Is PyTorch?

PyTorch is a machine learning and deep learning tool developed by Facebook's artificial intelligence division to process large-scale image analysis, including object detection, segmentation and classification. It is

not limited to these tasks, however. It can be used with other frameworks to implement complex algorithms. It is written using Python and the C++ language. To process large-scale computations in a GPU environment, the programming languages should be modified accordingly. PyTorch provides a great framework to write functions that automatically run in a GPU environment.

PyTorch Installation

Installing PyTorch is quite simple. In Windows, Linux, or macOS, it is very simple to install if you are familiar with the Anaconda and Conda environments for managing packages. The following steps describe how to install PyTorch in Windows/macOS/Linux environments.

1. Open the Anaconda navigator and go to the environment page, as displayed in the screenshot shown in Figure 1-2.

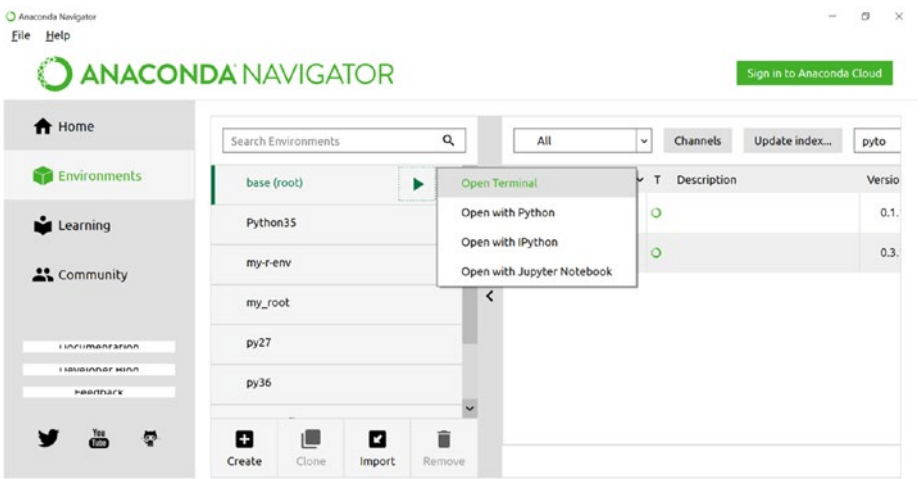


Figure 1-2. *Relationships among ML, DL, and AI*

2. Open the terminal and terminal and type the following:

   ```
   conda install -c peterjc123 pytorch
   ```

3. Launch Jupyter and open the IPython Notebook.

4. Type the following command to check whether the PyTorch is installed or not.

   ```
   from __future__ import print_function
   import torch
   ```

5. Check the version of the PyTorch.

```
In [1]: from __future__ import print_function
        import torch

In [2]: torch.version.__version__
Out[2]: '0.4.1'
```

This installation process was done using a Microsoft Windows machine. The process may vary by operating system, so please use the following URLs for any issue regarding installation and errors.

There are two ways to install it: Conda (Anaconda) library management or the Pip3 package management framework. Also, installations for a local system (such as macOS, Windows, or Linux) and a cloud machine (such as Microsoft Azure, AWS, and GCP) are different. To set up according to your platform, please follow the official PyTorch installation documents at https://PyTorch.org/get-started/cloud-partners/.

PyTorch has various components.

- Torch has functionalities similar to NumPy with GPU support.
- Autograd's torch.autograd provides classes, methods, and functions for implementing automatic differentiation of arbitrary scalar valued functions. It requires minimal changes to the existing code. You only need to declare `class:'Tensor's`, for which gradients should be computed with the `requires_grad=True` keyword.
- NN is a neural network library in PyTorch.
- Optim provides optimization algorithms that are used for the minimization and maximization of functions.
- Multiprocessing is a useful library for memory sharing between multiple tensors.
- Utils has utility functions to load data; it also has other functions.

Now we are ready to proceed with the chapter.

Recipe 1-1. Using Tensors

Problem

The data structure used in PyTorch is graph based and tensor based, therefore, it is important to understand basic operations and defining tensors.

Solution

The solution to this problem is practicing on the tensors and its operations, which includes many examples that use various operations. Though it is assumed that the user is familiar with PyTorch and Python basics, a refresher on PyTorch is essential to create interest among new users.

How It Works

Let's have a look at the following examples of tensors and tensor operation basics, including mathematical operations.

The `x` object is a list. We can check whether an object in Python is a tensor object by using the following syntax. Typically, the `is_tensor` function checks and the `is_storage` function checks whether the object is stored as tensor object.

```
In [3]: x = [12,23,34,45,56,67,78]

In [4]: torch.is_tensor(x)
Out[4]: False

In [5]: torch.is_storage(x)
Out[5]: False
```

Now, let's create an object that contains random numbers from Torch, similar to NumPy library. We can check the tensor and storage type.

```
In [6]: y = torch.randn(1,2,3,4,5)

In [7]: torch.is_tensor(y)
Out[7]: True

In [8]: torch.is_storage(y)
Out[8]: False

In [9]: torch.numel(y) # the total number of elements in the input Tensor
Out[9]: 120
```

The y object is a tensor; however, it is not stored. To check the total number of elements in the input tensor object, the numerical element function can be used. The following script is another example of creating zero values in a 2D tensor and counting the numerical elements in it.

```
In [10]: torch.zeros(4,4)
Out[10]: tensor([[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]])

In [11]: torch.numel(torch.zeros(4,4))
Out[11]: 16
```

Like NumPy operations, the eye function creates a diagonal matrix, of which the diagonal elements have ones, and off diagonal elements have zeros. The eye function can be manipulated by providing the shape option. The following example shows how to provide the shape parameter.

```
In [14]: torch.eye(3,4)

Out[14]: tensor([[1., 0., 0., 0.],
                 [0., 1., 0., 0.],
                 [0., 0., 1., 0.]])

In [15]: torch.eye(5,4)

Out[15]: tensor([[1., 0., 0., 0.],
                 [0., 1., 0., 0.],
                 [0., 0., 1., 0.],
                 [0., 0., 0., 1.],
                 [0., 0., 0., 0.]])
```

Linear space and points between the linear space can be created using tensor operations. Let's use an example of creating 25 points in a linear space starting from value 2 and ending with 10. Torch can read from a NumPy array format.

```
In [20]: import numpy as np
         x1 = np.array(x)

In [21]: x1

Out[21]: array([12, 23, 34, 45, 56, 67, 78])

In [22]: torch.from_numpy(x1)

Out[22]: tensor([12, 23, 34, 45, 56, 67, 78])

In [18]: x1

Out[18]: array([12, 23, 34, 45, 56, 67, 78])

In [19]: torch.from_numpy(x1)

Out[19]: tensor([12, 23, 34, 45, 56, 67, 78], dtype=torch.int32)

In [20]: torch.linspace(2, 10, steps=25) #linear spacing

Out[20]: tensor([ 2.0000,  2.3333,  2.6667,  3.0000,  3.3333,  3.6667,
         4.0000,  4.3333,
                  4.6667,  5.0000,  5.3333,  5.6667,  6.0000,  6.3333,
         6.6667,  7.0000,
                  7.3333,  7.6667,  8.0000,  8.3333,  8.6667,  9.0000,
         9.3333,  9.6667,
                 10.0000])
```

Like linear spacing, logarithmic spacing can be created.

```
In [22]: torch.logspace(start=-10, end=10, steps=15) #Logarithmic spacing
Out[22]: tensor([1.0000e-10, 2.6827e-09, 7.1969e-08, 1.9307e-06, 5.1795e-
         05, 1.3895e-03,
                 3.7276e-02, 1.0000e+00, 2.6827e+01, 7.1968e+02, 1.9307e+
         04, 5.1795e+05,
                 1.3895e+07, 3.7276e+08, 1.0000e+10])
```

Random number generation is a common process in data science to generate or gather sample data points in a space to simulate structure in the data. Random numbers can be generated from a statistical distribution, any two values, or a predefined distribution. Like NumPy functions, the random number can be generated using the following example. Uniform distribution is defined as a distribution where each outcome has equal probability of happening; hence, the event probabilities are constant.

```
In [25]: # random numbers from a uniform distribution between the values
         # 0 and 1
         torch.rand(10)
Out[25]: tensor([0.9721, 0.3732, 0.9673, 0.7479, 0.0599, 0.7082, 0.2308,
         0.4880, 0.6320,
                 0.8672])
```

The following script shows how the random number from two values, 0 and 1, are selected. The result tensor can be reshaped to create a (4,5) matrix. The random numbers from a normal distribution with arithmetic mean 0 and standard deviation 1 can also be created, as follows.

```
In [26]: torch.rand(4, 5)
         # random values between 0 and 1 and fillied with a matrix of
         # size rows 4 and columns 5

Out[26]: tensor([[0.1100, 0.6942, 0.6697, 0.4465, 0.8691],
                 [0.0219, 0.9273, 0.5040, 0.0069, 0.8226],
                 [0.5838, 0.4148, 0.9051, 0.6725, 0.7106],
                 [0.0938, 0.6354, 0.8358, 0.8852, 0.1933]])
```

```
In [27]: #random numbers from a normal distribution,
         #with mean =0 and standard deviation =1
         torch.randn(10)

Out[27]: tensor([-0.3417,  0.0228, -0.1101, -0.3394,  1.0303,  0.9248, -
         0.1369,  1.2947,
                 -0.1683,  0.2832])
```

```
In [28]: torch.randn(4, 5)

Out[28]: tensor([[ 1.0414, -0.3132,  0.4765, -0.0957, -0.5839],
                 [ 0.8857, -0.9681, -0.9982, -0.6281,  0.3062],
                 [ 0.7603,  0.6687, -1.3832, -0.6559,  1.5712],
                 [ 0.0675,  1.2403, -0.0631,  0.0151,  0.3821]])
```

To select random values from a range of values using random permutation requires defining the range first. This range can be created by using the arrange function. When using the arrange function, you must define the step size, which places all the values in an equal distance space. By default, the step size is 1.

```
In [29]: #selecting values from a range, this is called random permutation
         torch.randperm(10)

Out[29]: tensor([6, 0, 8, 9, 4, 2, 3, 5, 1, 7])
```

```
In [30]: #usage of range function
         torch.arange(10, 40,2) #step size 2

Out[30]: tensor([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
         38])
```

```
In [31]: torch.arange(10,40) #step size 1

Out[31]: tensor([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
         24, 25, 26, 27,
                 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
```

To find the minimum and maximum values in a 1D tensor, argmin and argmax can be used. The dimension needs to be mentioned if the input is a matrix in order to search minimum values along rows or columns.

```
In [32]: d = torch.randn(4, 5)
         d
Out[32]: tensor([[-1.6702,  0.7889, -1.1928,  1.3117,  0.2354],
                 [ 0.8243, -0.4847,  1.4079,  0.5033,  0.1626],
                 [-1.0932, -1.0021, -0.8570,  0.3783,  1.0392],
                 [ 0.1769, -0.6660,  1.9855,  0.9098, -0.9700]])

In [33]: torch.argmin(d,dim=1)
Out[33]: tensor([0, 1, 0, 4])

In [34]: torch.argmax(d,dim=1)
Out[34]: tensor([3, 2, 4, 2])
```

If it is either a row or column, it is a single dimension and is called a *1D tensor*. If the input is a matrix, in which rows and columns are present, it is called a *2D tensor*. If it is more than two-dimensional, it is called a *multidimensional tensor*.

```
In [35]: # create a 2dtensor filled with values as 0
         torch.zeros(4,5)
Out[35]: tensor([[0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.]])

In [36]: # create a 1d tensor filled with values as 0
         torch.zeros(10)
Out[36]: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Now, let's create a sample 2D tensor and perform indexing and concatenation by using the concat operation on the tensors.

```
In [37]: #indexing and performing operation on the tensors
         x = torch.randn(4,5)

In [38]: x

Out[38]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])

In [39]: #concatenate two tensors
         torch.cat((x,x))

Out[39]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785],
                 [-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])
```

The sample x tensor can be used in 3D as well. Again, there are two different options to create three-dimensional tensors; the third dimension can be extended over rows or columns.

```
In [41]: #concatenate n times based on array size, over column
         torch.cat((x,x,x),1)

Out[41]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023, -0.2021,  0.9669, -0.9
         755,
                  -0.3235,  0.7023, -0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603, -0.2164,  0.3371,  0.3
         157,
                  -0.6900,  1.2603, -0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861,  0.9157,  0.4921, -0.9
         229,
                   0.9070,  0.3861,  0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785,  1.2092,  0.8473, -0.5
         978,
                   0.3835,  0.4785,  1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])

In [42]: #concatenate n times based on array size, over rows
         torch.cat((x,x),0)

Out[42]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785],
                 [-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])
```

A tensor can be split between multiple chunks. Those small chunks can be created along dim rows and dim columns. The following example shows a sample tensor of size (4,4). The chunk is created using the third argument in the function, as 0 or 1.

```
In [45]: a = torch.randn(4, 4)
         print(a)
         torch.chunk(a,2)

         tensor([[-0.4715, -0.4157,  0.2914,  1.5323],
                 [ 0.5085,  0.5978, -0.1680, -0.8484],
                 [ 0.1818,  0.3133,  0.1674,  1.3080],
                 [-1.8344, -1.0157,  1.2815,  0.1155]])

Out[45]: (tensor([[-0.4715, -0.4157,  0.2914,  1.5323],
                  [ 0.5085,  0.5978, -0.1680, -0.8484]]),
          tensor([[ 0.1818,  0.3133,  0.1674,  1.3080],
                  [-1.8344, -1.0157,  1.2815,  0.1155]]))

In [46]: torch.chunk(a,2,0)

Out[46]: (tensor([[-0.4715, -0.4157,  0.2914,  1.5323],
                  [ 0.5085,  0.5978, -0.1680, -0.8484]]),
          tensor([[ 0.1818,  0.3133,  0.1674,  1.3080],
                  [-1.8344, -1.0157,  1.2815,  0.1155]]))

In [47]: torch.chunk(a,2,1)

Out[47]: (tensor([[-0.4715, -0.4157],
                  [ 0.5085,  0.5978],
                  [ 0.1818,  0.3133],
                  [-1.8344, -1.0157]]), tensor([[ 0.2914,  1.5323],
                  [-0.1680, -0.8484],
                  [ 0.1674,  1.3080],
                  [ 1.2815,  0.1155]]))
```

The gather function collects elements from a tensor and places it in another tensor using an index argument. The index position is determined by the LongTensor function in PyTorch.

```
In [48]: torch.Tensor([[11,12],[23,24]])

Out[48]: tensor([[11., 12.],
                 [23., 24.]])

In [49]: torch.gather(torch.Tensor([[11,12],[23,24]]), 1,
                      torch.LongTensor([[0,0],[1,0]]))

Out[49]: tensor([[11., 11.],
                 [24., 23.]])

In [50]:  torch.LongTensor([[0,0],[1,0]])
         #the 1D tensor containing the indices to index

Out[50]: tensor([[0, 0],
                 [1, 0]])
```

The LongTensor function or the index select function can be used to fetch relevant values from a tensor. The following sample code shows two options: selection along rows and selection along columns. If the second argument is 0, it is for rows. If it is 1, then it is along the columns.

```
In [51]: a = torch.randn(4, 4)
         print(a)

         tensor([[-0.1124, -0.8486,  0.0494,  0.8509],
                 [ 0.8989, -0.0790, -0.1527,  0.6677],
                 [-0.0801, -0.4841, -0.3250, -0.0913],
                 [ 0.0106,  1.3410,  1.3337, -0.3133]])

In [52]: indices = torch.LongTensor([0, 2])

In [53]: torch.index_select(a, 0, indices)

Out[53]: tensor([[-0.1124, -0.8486,  0.0494,  0.8509],
                 [-0.0801, -0.4841, -0.3250, -0.0913]])

In [54]: torch.index_select(a, 1, indices)

Out[54]: tensor([[-0.1124,  0.0494],
                 [ 0.8989, -0.1527],
                 [-0.0801, -0.3250],
                 [ 0.0106,  1.3337]])
```

It is a common practice to check non-missing values in a tensor, the objective is to identify non-zero elements in a large tensor.

```
In [57]: #identify null input tensors using nonzero function
         torch.nonzero(torch.tensor([10,00,23,0,0.0]))

Out[57]: tensor([[0],
                 [2]])

In [58]: torch.nonzero(torch.Tensor([10,00,23,0,0.0]))

Out[58]: tensor([[0],
                 [2]])
```

Restructuring the input tensors into smaller tensors not only fastens the calculation process, but also helps in distributed computing. The split function splits a long tensor into smaller tensors.

```
In [57]: # splitting the tensor into small chunks
         torch.split(torch.tensor([12,21,34,32,45,54,56,65]),2)

Out[57]: (tensor([12, 21]), tensor([34, 32]), tensor([45, 54]), tensor([56, 65]))

In [58]: # splitting the tensor into small chunks
         torch.split(torch.tensor([12,21,34,32,45,54,56,65]),3)

Out[58]: (tensor([12, 21, 34]), tensor([32, 45, 54]), tensor([56, 65]))
```

Now, let's have a look at examples of how the input tensor can be resized given the computational difficulty. The transpose function is primarily used to reshape tensors. There are two ways of writing the transpose function: `.t` and `.transpose`.

```
In [61]: #how to reshape the tensors along a new dimension

In [62]: x

Out[62]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])

In [63]: x.t() #transpose is one option to change the shape of the tensor

Out[63]: tensor([[-0.2021, -0.2164,  0.9157,  1.2092],
                 [ 0.9669,  0.3371,  0.4921,  0.8473],
                 [-0.9755,  0.3157, -0.9229, -0.5978],
                 [-0.3235, -0.6900,  0.9070,  0.3835],
                 [ 0.7023,  1.2603,  0.3861,  0.4785]])


In [65]: x.transpose(1,0)

Out[65]: tensor([[-0.2021, -0.2164,  0.9157,  1.2092],
                 [ 0.9669,  0.3371,  0.4921,  0.8473],
                 [-0.9755,  0.3157, -0.9229, -0.5978],
                 [-0.3235, -0.6900,  0.9070,  0.3835],
                 [ 0.7023,  1.2603,  0.3861,  0.4785]])
```

The unbind function removes a dimension from a tensor. To remove the dimension row, the 0 value needs to be passed. To remove a column, the 1 value needs to be passed.

```
In [67]: x

Out[67]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])

In [68]: torch.unbind(x,1) #dim=1 removing a column

Out[68]: (tensor([-0.2021, -0.2164,  0.9157,  1.2092]),
          tensor([0.9669, 0.3371, 0.4921, 0.8473]),
          tensor([-0.9755,  0.3157, -0.9229, -0.5978]),
          tensor([-0.3235, -0.6900,  0.9070,  0.3835]),
          tensor([0.7023, 1.2603, 0.3861, 0.4785]))

In [69]: torch.unbind(x) #dim=0 removing a row

Out[69]: (tensor([-0.2021,  0.9669, -0.9755, -0.3235,  0.7023]),
          tensor([-0.2164,  0.3371,  0.3157, -0.6900,  1.2603]),
          tensor([ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861]),
          tensor([ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]))
```

Mathematical functions are the backbone of implementing any algorithm in PyTorch; therefore, it is needed to go through functions that help perform arithmetic-based operations. A scalar is a single value, and a tensor 1D is a row, like NumPy. The scalar multiplication and addition with a 1D tensor are done using the add and mul functions.

```
In [70]: x

Out[70]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])
```

The following script shows scalar addition and multiplication with a tensor.

```
In [73]: #adding value to the existing tensor, scalar addition
         torch.add(x,20)

Out[73]: tensor([[19.7979, 20.9669, 19.0245, 19.6765, 20.7023],
                 [19.7836, 20.3371, 20.3157, 19.3100, 21.2603],
                 [20.9157, 20.4921, 19.0771, 20.9070, 20.3861],
                 [21.2092, 20.8473, 19.4022, 20.3835, 20.4785]])

In [74]: x

Out[74]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])

In [75]: # scalar multiplication
         torch.mul(x,2)

Out[75]: tensor([[-0.4043,  1.9337, -1.9509, -0.6470,  1.4047],
                 [-0.4327,  0.6743,  0.6314, -1.3800,  2.5205],
                 [ 1.8315,  0.9843, -1.8458,  1.8140,  0.7723],
                 [ 2.4184,  1.6946, -1.1956,  0.7671,  0.9570]])

In [76]: x

Out[76]: tensor([[-0.2021,  0.9669, -0.9755, -0.3235,  0.7023],
                 [-0.2164,  0.3371,  0.3157, -0.6900,  1.2603],
                 [ 0.9157,  0.4921, -0.9229,  0.9070,  0.3861],
                 [ 1.2092,  0.8473, -0.5978,  0.3835,  0.4785]])
```

Combined mathematical operations, such as expressing linear equations as tensor operations can be done using the following sample script. Here we express the outcome y object as a linear combination of beta values times the independent x object, plus the constant term.

```
In [79]: intercept = torch.randn(1)
         intercept

Out[79]: tensor([-0.4292])

In [80]: x = torch.randn(2, 2)
         x

Out[80]: tensor([[-0.7945,  0.7653],
                 [ 0.5497,  0.8854]])

In [81]: beta = 0.7456
         beta

Out[81]: 0.7456
```

Output = Constant + (beta * Independent)

```
In [84]: torch.mul(intercept,x)

Out[84]: tensor([[ 0.3410, -0.3284],
                 [-0.2359, -0.3800]])

In [85]: torch.mul(x,beta)

Out[85]: tensor([[-0.5924,  0.5706],
                 [ 0.4098,  0.6602]])

In [86]: ## y = intercept + (beta * x)
         torch.add(torch.mul(intercept,x),torch.mul(x,beta)) # tensor y

Out[86]: tensor([[-0.2514,  0.2422],
                 [ 0.1739,  0.2802]])
```

Like NumPy operations, the tensor values must be rounded up by using either the ceiling or the flooring function, which is done using the following syntax.

```
In [87]: # how to round up tensor values
         torch.manual_seed(1234)
         torch.randn(5,5)

Out[87]: tensor([[-0.1117, -0.4966,  0.1631, -0.8817,  0.0539],
                 [ 0.6684, -0.0597, -0.4675, -0.2153, -0.7141],
                 [-1.0831, -0.5547,  0.9717, -0.5150,  1.4255],
                 [ 0.7987, -1.4949,  1.4778, -0.1696, -0.9919],
                 [-1.4569,  0.2563, -0.4030,  0.4195,  0.9380]])

In [88]: torch.manual_seed(1234)
         torch.ceil(torch.randn(5,5))

Out[88]: tensor([[-0., -0.,  1., -0.,  1.],
                 [ 1., -0., -0., -0., -0.],
                 [-1., -0.,  1., -0.,  2.],
                 [ 1., -1.,  2., -0., -0.],
                 [-1.,  1., -0.,  1.,  1.]])

In [89]: torch.manual_seed(1234)
         torch.floor(torch.randn(5,5))

Out[89]: tensor([[-1., -1.,  0., -1.,  0.],
                 [ 0., -1., -1., -1., -1.],
                 [-2., -1.,  0., -1.,  1.],
                 [ 0., -2.,  1., -1., -1.],
                 [-2.,  0., -1.,  0.,  0.]])
```

Limiting the values of any tensor within a certain range can be done using the minimum and maximum argument and using the clamp function. The same function can apply minimum and maximum in parallel or any one of them to any tensor, be it 1D or 2D; 1D is the far simpler version. The following example shows the implementation in a 2D scenario.

```
In [90]: # truncate the values in a range say 0,1
         torch.manual_seed(1234)
         torch.clamp(torch.floor(torch.randn(5,5)), min=-0.3, max=0.4)

Out[90]: tensor([[-0.3000, -0.3000,  0.0000, -0.3000,  0.0000],
                 [ 0.0000, -0.3000, -0.3000, -0.3000, -0.3000],
                 [-0.3000, -0.3000,  0.0000, -0.3000,  0.4000],
                 [ 0.0000, -0.3000,  0.4000, -0.3000, -0.3000],
                 [-0.3000,  0.0000, -0.3000,  0.0000,  0.0000]])
```

```
In [91]: #truncate with only lower limit
         torch.manual_seed(1234)
         torch.clamp(torch.floor(torch.randn(5,5)), min=-0.3)

Out[91]: tensor([[-0.3000, -0.3000,  0.0000, -0.3000,  0.0000],
                 [ 0.0000, -0.3000, -0.3000, -0.3000, -0.3000],
                 [-0.3000, -0.3000,  0.0000, -0.3000,  1.0000],
                 [ 0.0000, -0.3000,  1.0000, -0.3000, -0.3000],
                 [-0.3000,  0.0000, -0.3000,  0.0000,  0.0000]])

In [92]: #truncate with only upper limit
         torch.manual_seed(1234)
         torch.clamp(torch.floor(torch.randn(5,5)), max=0.3)

Out[92]: tensor([[-1.0000, -1.0000,  0.0000, -1.0000,  0.0000],
                 [ 0.0000, -1.0000, -1.0000, -1.0000, -1.0000],
                 [-2.0000, -1.0000,  0.0000, -1.0000,  0.3000],
                 [ 0.0000, -2.0000,  0.3000, -1.0000, -1.0000],
                 [-2.0000,  0.0000, -1.0000,  0.0000,  0.0000]])
```

How do we get the exponential of a tensor? How do we get the fractional portion of the tensor if it has decimal places and is defined as a floating data type?

```
In [94]: #compute the exponential of a tensor
         torch.exp(x)

Out[94]: tensor([[0.4518, 2.1496],
                 [1.7327, 2.4241]])

In [95]: np.exp(x)

Out[95]: tensor([[0.4518, 2.1496],
                 [1.7327, 2.4241]])

In [96]: #how to get the fractional portion of each tensor

In [97]: torch.add(x,10)

Out[97]: tensor([[ 9.2055, 10.7653],
                 [10.5497, 10.8854]])

In [98]: torch.frac(torch.add(x,10))

Out[98]: tensor([[0.2055, 0.7653],
                 [0.5497, 0.8854]])
```

The following syntax explains the logarithmic values in a tensor. The values with a negative sign are converted to nan. The power function computes the exponential of any value in a tensor.

```
In [99]: # compute the log of the values in a tensor

In [100]: x

Out[100]: tensor([[-0.7945,  0.7653],
                  [ 0.5497,  0.8854]])

In [101]: torch.log(x) #log of negatives are nan

Out[101]: tensor([[    nan, -0.2675],
                  [-0.5984, -0.1217]])

In [102]: # to rectify the negative values do a power tranforamtion
          torch.pow(x,2)

Out[102]: tensor([[0.6312, 0.5857],
                  [0.3021, 0.7840]])
```

To compute the transformation functions (i.e., sigmoid, hyperbolic tangent, radial basis function, and so forth, which are the most commonly used transfer functions in deep learning), you must construct the tensors. The following sample script shows how to create a sigmoid function and apply it on a tensor.

```
In [107]: # how to compute the sigmoid of the input tensor

In [108]: x

Out[108]: tensor([[-0.7945,  0.7653],
                  [ 0.5497,  0.8854]])

In [109]: torch.sigmoid(x)

Out[109]: tensor([[0.3112, 0.6825],
                  [0.6341, 0.7080]])

In [110]: # finding the square root of the values

In [111]: x

Out[111]: tensor([[-0.7945,  0.7653],
                  [ 0.5497,  0.8854]])

In [112]: torch.sqrt(x)

Out[112]: tensor([[   nan, 0.8748],
                  [0.7414, 0.9410]])
```

Conclusion

This chapter is a refresher for people who have prior experience in PyTorch and Python. It is a basic building block for people who are new to the PyTorch framework. Before starting the advanced topics, it is important to become familiar with the terminology and basic syntaxes. The next chapter is on using PyTorch to implement probabilistic models, which includes the creation of random variables, the application of statistical distributions, and making statistical inferences.

CHAPTER 2

Probability Distributions Using PyTorch

Probability and random variables are an integral part of computation in a graph-computing platform like PyTorch. Understanding probability and associated concepts are essential. This chapter covers probability distributions and implementation using PyTorch, and interpreting the results from tests.

In probability and statistics, a random variable is also known as a *stochastic variable*, whose outcome is dependent on a purely stochastic phenomenon, or random phenomenon. There are different types of probability distributions, including normal distribution, binomial distribution, multinomial distribution, and Bernoulli distribution. Each statistical distribution has its own properties.

The torch.distributions module contains probability distributions and sampling functions. Each distribution type has its own importance in a computational graph. The distributions module contains binomial, Bernoulli, beta, categorical, exponential, normal, and Poisson distributions.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_2

Recipe 2-1. Sampling Tensors

Problem

Weight initialization is an important task in training a neural network and any kind of deep learning model, such as a convolutional neural network (CNN), a deep neural network (DNN), and a recurrent neural network (RNN). The question always remains on how to initialize the weights.

Solution

Weight initialization can be done by using various methods, including random weight initialization. Weight initialization based on a distribution is done using uniform distribution, Bernoulli distribution, multinomial distribution, and normal distribution. How to do it using PyTorch is explained next.

How It Works

To execute a neural network, a set of initial weights needs to be passed to the backpropagation layer to compute the loss function (and hence, the accuracy can be calculated). The selection of a method depends on the data type, the task, and the optimization required for the model. Here we are going to look at all types of approaches to initialize weights.

If the use case requires reproducing the same set of results to maintain consistency, then a manual seed needs to be set.

```
In [2]: import torch

In [3]: # how to perform random sampling of the tensors

In [4]: torch.manual_seed(1234)

Out[4]: <torch._C.Generator at 0x2852c8a1b30>

In [5]: torch.manual_seed(1234)
        torch.randn(4,4)

Out[5]: tensor([[-0.1117, -0.4966,  0.1631, -0.8817],
                [ 0.0539,  0.6684, -0.0597, -0.4675],
                [-0.2153,  0.8840, -0.7584, -0.3689],
                [-0.3424, -1.4020,  0.3206, -1.0219]])
```

The seed value can be customized. The random number is generated purely by chance. Random numbers can also be generated from a statistical distribution. The probability density function of the *continuous uniform distribution* is defined by the following formula.

$$f(x)=\begin{cases}\dfrac{1}{b-a} & \text{for } a\le x\le b,\\ 0 & \text{for } x<a \text{ or } x>b\end{cases}$$

The function of x has two points, a and b, in which a is the starting point and b is the end. In a continuous uniform distribution, each number has an equal chance of being selected. In the following example, the start is 0 and the end is 1; between those two digits, all 16 elements are selected randomly.

```
In [8]: torch.Tensor(4, 4).uniform_(0, 1) #random number from uniform distribution

Out[8]: tensor([[0.2837, 0.6567, 0.2388, 0.7313],
                [0.6012, 0.3043, 0.2548, 0.6294],
                [0.9665, 0.7399, 0.4517, 0.4757],
                [0.7842, 0.1525, 0.6662, 0.3343]])
```

In statistics, the *Bernoulli distribution* is considered as the discrete probability distribution, which has two possible outcomes. If the event happens, then the value is 1, and if the event does not happen, then the value is 0.

For *discrete probability distribution*, we calculate probability mass function instead of probability density function. The probability mass function looks like the following formula.

$$\begin{cases} q = (1-p) & \text{for} k = 0 \\ p & \text{for} k = 1 \end{cases}$$

From the Bernoulli distribution, we create sample tensors by considering the uniform distribution of size 4 and 4 in a matrix format, as follows.

```
In [10]: torch.bernoulli(torch.Tensor(4, 4).uniform_(0, 1))

Out[10]: tensor([[1., 1., 1., 1.],
                 [0., 0., 0., 0.],
                 [0., 0., 1., 0.],
                 [1., 1., 0., 0.]])
```

The generation of sample random values from a *multinomial distribution* is defined by the following script. In a multinomial distribution, we can choose with a replacement or without a replacement. By default, the multinomial function picks up without a replacement and returns the result as an index position for the tensors. If we need to run it with a replacement, then we need to specify that while sampling.

```
In [12]: torch.Tensor([10, 10, 13, 10,34,45,65,67,87,89,87,34])

Out[12]: tensor([10., 10., 13., 10., 34., 45., 65., 67., 87., 89., 87., 34.])

In [13]: torch.multinomial(torch.tensor([10., 10., 13., 10.,
                                         34., 45., 65., 67.,
                                         87., 89., 87., 34.]),
                           3)

Out[13]: tensor([ 8,  7, 11])
```

Sampling from multinomial distribution with a replacement returns the tensors' index values.

```
In [14]: torch.multinomial(torch.tensor([10., 10., 13., 10.,
                                         34., 45., 65., 67.,
                                         87., 89., 87., 34.]),
                           5, replacement=True)

Out[14]: tensor([ 5, 10,  5,  9, 10])
```

The weight initialization from the normal distribution is a method that is used in fitting a neural network, fitting a deep neural network, and CNN and RNN. Let's have a look at the process of creating a set of random weights generated from a normal distribution.

```
In [16]: torch.normal(mean=torch.arange(1., 11.),
                      std=torch.arange(1, 0, -0.1))

Out[16]: tensor([1.2111, 2.3034, 3.5310, 4.7278, 6.1060, 6.3294, 6.9060, 7.9908,
         9.3492,
                 9.9928])

In [17]: torch.normal(mean=0.5,
                      std=torch.arange(1., 6.))

Out[17]: tensor([-1.1794, -2.9019,  2.4459,  7.5613,  5.9058])

In [18]: torch.normal(mean=0.5,
                      std=torch.arange(0.2,0.6))

Out[18]: tensor([0.7487])
```

Recipe 2-2. Variable Tensors

Problem

What is a variable in PyTorch and how is it defined? What is a random variable in PyTorch?

Solution

In PyTorch, the algorithms are represented as a computational graph. A variable is considered as a representation around the tensor object, corresponding gradients, and a reference to the function from where it was created. For simplicity, gradients are considered as slope of the function. The slope of the function can be computed by the derivative of the function with respect to the parameters that are present in the function. For example, in linear regression (Y = W*X + alpha), representation of the variable would look like the one shown in Figure 2-2.

Basically, a PyTorch variable is a node in a computational graph, which stores data and gradients. When training a neural network model, after each iteration, we need to compute the gradient of the loss function with respect to the parameters of the model, such as weights and biases. After that, we usually update the weights using the gradient descent algorithm. Figure 2-1 explains how the linear regression equation is deployed under the hood using a neural network model in the PyTorch framework.

In a computational graph structure, the sequencing and ordering of tasks is very important. The one-dimensional tensors are X, Y, W, and alpha in Figure 2-2. The direction of the arrows change when we implement backpropagation to update the weights to match with Y, so that the error or loss function between Y and predicted Y can be minimized.

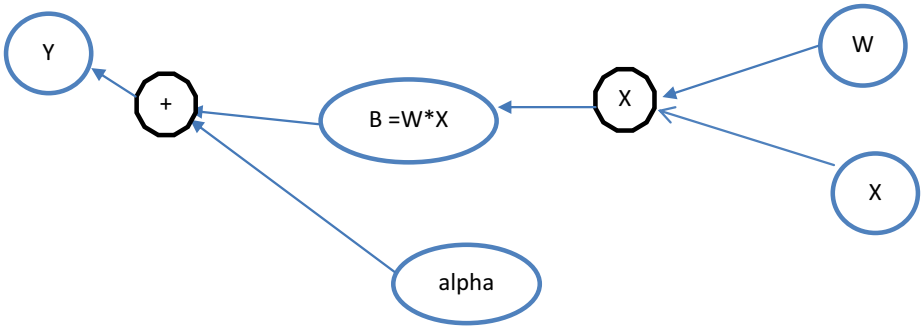


Figure 2-1. *A sample computational graph of a PyTorch implementation*

How It Works

An example of how a variable is used to create a computational graph is displayed in the following script. There are three variable objects around tensors— x1, x2, and x3—with random points generated from $a = 12$ and $b = 23$. The graph computation involves only multiplication and addition, and the final result with the gradient is shown.

The partial derivative of the loss function with respect to the weights and biases in a neural network model is achieved in PyTorch using the Autograd module. Variables are specifically designed to hold the changed values while running a backpropagation in a neural network model when the parameters of the model change. The variable type is just a wrapper around the tensor. It has three properties: data, grad, and function.

```
In [45]: from torch.autograd import Variable

In [47]: Variable(torch.ones(2,2),requires_grad=True)

Out[47]: tensor([[1., 1.],
                 [1., 1.]], requires_grad=True)

In [53]: a, b = 12,23
         x1 = Variable(torch.randn(a,b),
                       requires_grad=True)
         x2 = Variable(torch.randn(a,b),
                       requires_grad=True)
         x3 =Variable(torch.randn(a,b),
                       requires_grad=True)

In [66]: c = x1 * x2
         d = a + x3
         e = torch.sum(d)

         e.backward()

         print(e)

         tensor(3326.2632, grad_fn=<SumBackward0>)
```

Recipe 2-3. Basic Statistics

Problem

How do we compute basic statistics, such as mean, median, mode, and so forth, from a Torch tensor?

Solution

Computation of basic statistics using PyTorch enables the user to apply probability distributions and statistical tests to make inferences from data. Though the Torch functionality is like that of Numpy, Torch functions have GPU acceleration. Let's have a look at the functions to create basic statistics.

How It Works

The mean computation is simple to write for a 1D tensor; however, for a 2D tensor, an extra argument needs to be passed as a mean, median, or mode computation, across which the dimension needs to be specified.

```
In [19]: #computing the descriptive statistics: mean
         torch.mean(torch.tensor([10., 10., 13., 10., 34.,
                                  45., 65., 67., 87., 89., 87., 34.]))
Out[19]: tensor(45.9167)

In [20]: # mean across rows and across columns
         d = torch.randn(4, 5)
         d
Out[20]: tensor([[-0.2632, -0.5432, -1.6406,  0.9295, -1.2777],
                 [-0.7428,  0.9711,  0.3551,  0.8562, -0.3635],
                 [-0.1552, -1.2282, -0.8039, -0.4530, -0.2217],
                 [-2.0901, -1.2658, -1.8761, -0.6066,  0.7470]])

In [21]: torch.mean(d,dim=0)
Out[21]: tensor([-0.8128, -0.5165, -0.9914,  0.1815, -0.2789])

In [22]: torch.mean(d,dim=1)
Out[22]: tensor([-0.5590,  0.2152, -0.5724, -1.0183])
```

Median, mode, and standard deviation computation can be written in the same way.

```
In [23]: #compute median
         torch.median(d,dim=0)

Out[23]: (tensor([-0.7428, -1.2282, -1.6406, -0.4530, -0.3635]),
          tensor([1, 2, 0, 2, 1]))

In [24]: torch.median(d,dim=1)

Out[24]: (tensor([-0.5432,  0.3551, -0.4530, -1.2658]), tensor([1, 2, 3, 1]))

In [25]: # compute the mode
         torch.mode(d)

Out[25]: (tensor([-1.6406, -0.7428, -1.2282, -2.0901]), tensor([2, 0, 1, 0]))

In [26]: torch.mode(d,dim=0)

Out[26]: (tensor([-2.0901, -1.2658, -1.8761, -0.6066, -1.2777]),
          tensor([3, 3, 3, 3, 0]))

In [27]: torch.mode(d,dim=1)

Out[27]: (tensor([-1.6406, -0.7428, -1.2282, -2.0901]), tensor([2, 0, 1, 0]))
```

Standard deviation shows the deviation from the measures of central tendency, which indicates the consistency of the data/variable. It shows whether there is enough fluctuation in data or not.

```
In [28]: #compute the standard deviation
         torch.std(d)
Out[28]: tensor(0.9237)

In [29]: torch.std(d,dim=0)
Out[29]: tensor([0.8890, 1.0459, 1.0087, 0.8243, 0.8287])

In [30]: torch.std(d,dim=1)
Out[30]: tensor([0.9987, 0.7507, 0.4458, 1.1436])

In [31]: #compute variance
         torch.var(d)
Out[31]: tensor(0.8532)

In [32]: torch.var(d,dim=0)
Out[32]: tensor([0.7903, 1.0939, 1.0175, 0.6795, 0.6868])

In [33]: torch.var(d,dim=1)
Out[33]: tensor([0.9974, 0.5636, 0.1987, 1.3079])
```

Recipe 2-4. Gradient Computation

Problem

How do we compute basic gradients from the sample tensors using PyTorch?

Solution

We are going to consider a sample datase0074, where two variables (x and y) are present. With the initial weight given, can we computationally get the gradients after each iteration? Let's take a look at the example.

How It Works

`x_data` and `y_data` both are lists. To compute the gradient of the two data lists requires computation of a loss function, a forward pass, and running the stuff in a loop.

The forward function computes the matrix multiplication of the weight tensor with the input tensor.

```
In [50]: # Using forward pass
         def forward(x):
             return x * w
```

```
In [76]: import torch
         from torch.autograd import Variable

         x_data = [11.0, 22.0, 33.0]
         y_data = [21.0, 14.0, 64.0]

         w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value

         # Before training
         print("predict (before training)",  4, forward(4).data[0])
```

```
predict (before training) 4 tensor(4.)
```

```
In [77]: # Using forward pass
         def forward(x):
             return x * w
```

```
In [78]: # define the Loss function
         def loss(x, y):
             y_pred = forward(x)
             return (y_pred - y) * (y_pred - y)
```

```
In [79]: # Run the Training Loop
         for epoch in range(10):
             for x_val, y_val in zip(x_data, y_data):
                 l = loss(x_val, y_val)
                 l.backward()
                 print("\tgrad: ", x_val, y_val, w.grad.data[0])
                 w.data = w.data - 0.01 * w.grad.data

                 # Manually set the gradients to zero after updating weights
                 w.grad.data.zero_()

             print("progress:", epoch, l.data[0])
```

```
        grad:  11.0 21.0 tensor(-220.)
        grad:  22.0 14.0 tensor(2481.6001)
        grad:  33.0 64.0 tensor(-51303.6484)
progress: 0 tensor(604238.8125)
        grad:  11.0 21.0 tensor(118461.7578)
        grad:  22.0 14.0 tensor(-671630.6875)
        grad:  33.0 64.0 tensor(13114108.)
progress: 1 tensor(39481139200.)
        grad:  11.0 21.0 tensor(-30279010.)
        grad:  22.0 14.0 tensor(171986000.)
        grad:  33.0 64.0 tensor(-3358889472.)
progress: 2 tensor(2590022582665216.)
        grad:  11.0 21.0 tensor(7755301376.)
        grad:  22.0 14.0 tensor(-44050112512.)
        grad:  33.0 64.0 tensor(860298674176.)
progress: 3 tensor(169906757784039325696.)
        grad:  11.0 21.0 tensor(-1986333900800.)
        grad:  22.0 14.0 tensor(11282376818688.)
        grad:  33.0 64.0 tensor(-220344807849984.)
progress: 4 tensor(11145967797036089329319936.)
        grad:  11.0 21.0 tensor(508751660449792.)
        grad:  22.0 14.0 tensor(-2889709562888192.)
        grad:  33.0 64.0 tensor(56436029183229952.)
progress: 5 tensor(731181229735636676902291243008.)
        grad:  11.0 21.0 tensor(-130304505987203072.)
        grad:  22.0 14.0 tensor(740129586448171008.)
```

```
In [80]: # After training
         print("predict (after training)",  4, forward(4).data[0])

         predict (after training) 4 tensor(-92686910753578617489326080.)
```

The following program shows how to compute the gradients from a loss function using the variable method on the tensor.

```
In [102]: from torch import FloatTensor
          from torch.autograd import Variable

          a = Variable(FloatTensor([5]))

          weights = [Variable(FloatTensor([i]), requires_grad=True) for i in (12, 53, 91, 73)]

          w1, w2, w3, w4 = weights

          b = w1 * a
          c = w2 * a
          d = w3 * b + w4 * c
          Loss = (10 - d)

          Loss.backward()

          for index, weight in enumerate(weights, start=1):
              gradient, *_ = weight.grad.data
              print(f"Gradient of w{index} w.r.t to Loss: {gradient}")

          Gradient of w1 w.r.t to Loss: -455.0
          Gradient of w2 w.r.t to Loss: -365.0
          Gradient of w3 w.r.t to Loss: -60.0
          Gradient of w4 w.r.t to Loss: -265.0
```

Recipe 2-5. Tensor Operations

Problem

How do we compute or perform operations based on variables such as matrix multiplication?

Solution

Tensors are wrapped within the variable, which has three properties: grad, volatile, and gradient.

How It Works

Let's create a variable and extract the properties of the variable. This is required to weight update process requires gradient computation. By using the mm module, we can perform matrix multiplication.

```
In [88]: x = Variable(torch.Tensor(4, 4).uniform_(-4, 5))
         y = Variable(torch.Tensor(4, 4).uniform_(-3, 2))
         # matrix multiplication
         z = torch.mm(x, y)
         print(z.size())

         torch.Size([4, 4])
```

The following program shows the properties of the variable, which is a wrapper around the tensor.

```
In [85]: z = Variable(torch.Tensor(4, 4).uniform_(-5, 5))
         print(z)

         tensor([[-2.6830,  2.0509, -2.6185, -3.6709],
                 [-4.9271,  3.6834,  4.0502,  3.4515],
                 [ 3.4576,  4.5814,  0.0632, -4.6377],
                 [-3.1634,  2.9827, -3.9532, -0.6395]])

In [86]: print('Requires Gradient : %s ' % (z.requires_grad))
         print('Volatile : %s ' % (z.volatile))
         print('Gradient : %s ' % (z.grad))
         print(z.data)

         Requires Gradient : False
         Volatile : False
         Gradient : None
         tensor([[-2.6830,  2.0509, -2.6185, -3.6709],
                 [-4.9271,  3.6834,  4.0502,  3.4515],
                 [ 3.4576,  4.5814,  0.0632, -4.6377],
                 [-3.1634,  2.9827, -3.9532, -0.6395]])
```

Recipe 2-6. Tensor Operations

Problem

How do we compute or perform operations based on variables such as matrix-vector computation, and matrix-matrix and vector-vector calculation?

Solution

One of the necessary conditions for the success of matrix-based operations is that the length of the tensor needs to match or be compatible for the execution of algebraic expressions.

How It Works

The tensor definition of a scalar is just one number. A 1D tensor is a vector, and a 2D tensor is a matrix. When it extends to an *n* dimensional level, it can be generalized to only tensors. When performing algebraic computations in PyTorch, the dimension of a matrix and a vector or scalar should be compatible.

```
In [103]: #tensor operations

In [109]: mat1 = torch.FloatTensor(4,4).uniform_(0,1)
          mat1

Out[109]: tensor([[0.7748, 0.5287, 0.8794, 0.0527],
                  [0.2824, 0.0036, 0.4439, 0.1425],
                  [0.7889, 0.8024, 0.2917, 0.9638],
                  [0.1534, 0.1216, 0.4023, 0.6097]])

In [110]: mat2 = torch.FloatTensor(5,4).uniform_(0,1)
          mat2

Out[110]: tensor([[0.9493, 0.1600, 0.8915, 0.7011],
                  [0.2557, 0.1726, 0.6665, 0.4602],
                  [0.9003, 0.5600, 0.0388, 0.3431],
                  [0.3064, 0.2131, 0.4333, 0.7120],
                  [0.1883, 0.4072, 0.2834, 0.8132]])

In [111]: vec1 = torch.FloatTensor(4).uniform_(0,1)
          vec1

Out[111]: tensor([0.8854, 0.9122, 0.9897, 0.2618])

In [112]: # scalar addition

In [113]: mat1 + 10.5

Out[113]: tensor([[11.2748, 11.0287, 11.3794, 10.5527],
                  [10.7824, 10.5036, 10.9439, 10.6425],
                  [11.2889, 11.3024, 10.7917, 11.4638],
                  [10.6534, 10.6216, 10.9023, 11.1097]])
```

```
In [114]: # scalar subtraction

In [115]: mat2 - 0.20

Out[115]: tensor([[ 0.7493, -0.0400,  0.6915,  0.5011],
                  [ 0.0557, -0.0274,  0.4665,  0.2602],
                  [ 0.7003,  0.3600, -0.1612,  0.1431],
                  [ 0.1064,  0.0131,  0.2333,  0.5120],
                  [-0.0117,  0.2072,  0.0834,  0.6132]])

In [116]: # vector and matrix addition

In [117]: mat1 + vec1

Out[117]: tensor([[1.6603, 1.4409, 1.8691, 0.3145],
                  [1.1678, 0.9158, 1.4336, 0.4042],
                  [1.6743, 1.7145, 1.2814, 1.2255],
                  [1.0388, 1.0337, 1.3920, 0.8715]])

In [118]: mat2 + vec1

Out[118]: tensor([[1.8348, 1.0722, 1.8812, 0.9629],
                  [1.1412, 1.0848, 1.6562, 0.7220],
                  [1.7857, 1.4721, 1.0285, 0.6049],
                  [1.1918, 1.1252, 1.4230, 0.9738],
                  [1.0738, 1.3193, 1.2731, 1.0750]])
```

Since the mat1 and the mat2 dimensions are different, they are not compatible for matrix addition or multiplication. If the dimension remains the same, we can multiply them. In the following script, the matrix addition throws an error when we multiply similar dimensions—mat1 with mat1. We get relevant results.

```
In [119]: # matrix-matrix addition

In [123]: mat1 + mat2

          ---------------------------------------------------------------------------
          --
          RuntimeError                              Traceback (most recent call las
          t)
          <ipython-input-123-bb4bbf5b2f84> in <module>()
          ----> 1 mat1 + mat2

          RuntimeError: The size of tensor a (4) must match the size of tensor b
           (5) at non-singleton dimension 0


In [131]: mat1 * mat1

Out[131]: tensor([[0.6004, 0.2795, 0.7733, 0.0028],
                  [0.0797, 0.0000, 0.1970, 0.0203],
                  [0.6224, 0.6438, 0.0851, 0.9288],
                  [0.0235, 0.0148, 0.1618, 0.3717]])
```

Recipe 2-7. Distributions

Problem

Knowledge of statistical distributions is essential for weight normalization, weight initialization, and computation of gradients in neural network–based operations using PyTorch. How do we know which distributions to use and when to use them?

Solution

Each statistical distribution follows a pre-established mathematical formula. We are going to use the most commonly used statistical distributions, their arguments in scenarios of problems.

How It Works

Bernoulli distribution is a special case of *binomial distribution*, in which the number of trials can be more than one; but in a Bernoulli distribution, the number of experiment or trial remains one. It is a discrete probability distribution of a random variable, which takes a value of 1 when there is probability that an event is a success, and takes a value of 0 when there is probability that an event is a failure. A perfect example of this is tossing a coin, where 1 is heads and 0 is tails. Let's look at the program.

```
In [ ]: # about Bernoulli distribution

In [138]: from torch.distributions.bernoulli import Bernoulli

In [139]: dist = Bernoulli(torch.tensor([0.3,0.6,0.9]))

In [140]: dist.sample() #sample is binary, it takes 1 with p and 0 with 1-p
Out[140]: tensor([1., 1., 1.])

In [132]: #Creates a Bernoulli distribution parameterized by probs
          #Samples are binary (0 or 1). They take the value 1 with probability p
          #and 0 with probability 1 - p.
```

The *beta distribution* is a family of continuous random variables defined in the range of 0 and 1. This distribution is typically used for Bayesian inference analysis.

```
In [133]: from torch.distributions.beta import Beta

In [141]: dist = Beta(torch.tensor([0.5]), torch.tensor([0.5]))
          dist
Out[141]: Beta()

In [142]: dist.sample()
Out[142]: tensor([0.3067])
```

The binomial distribution is applicable when the outcome is twofold and the experiment is repetitive. It belongs to the family of discrete probability distribution, where the probability of success is defined as 1 and the probability of failure is 0. The binomial distribution is used to model the number of successful events over many trials.

```
In [143]: from torch.distributions.binomial import Binomial

In [144]: dist = Binomial(100, torch.tensor([0 , .2, .8, 1]))

In [147]: dist.sample()
Out[147]: tensor([  0.,  29.,  85., 100.])

In [148]: # 100- count of trials
          # 0, 0.2, 0.8 and 1 are event probabilities
```

In probability and statistics, a categorical distribution can be defined as a generalized Bernoulli distribution, which is a discrete probability distribution that explains the possible results of any random variable that may take on one of the possible categories, with the probability of each category exclusively specified in the tensor.

```
In [174]: from torch.distributions.categorical import Categorical

In [175]: dist = Categorical(torch.tensor([ 0.20, 0.20, 0.20, 0.20, 0.20 ]))
          dist
Out[175]: Categorical()

In [176]: dist.sample()
Out[176]: tensor(4)

In [177]: # 0.20, 0.20, 0.20, 0.20,0.20 event probabilities
```

A *Laplacian distribution* is a continuous probability distribution function that is otherwise known as a *double exponential distribution*. A Laplacian distribution is used in speech recognition systems to understand prior probabilities. It is also useful in Bayesian regression for deciding prior probabilities.

```
In [155]: # Laplace distribution parameterized by loc and 'scale'.

In [157]: from torch.distributions.laplace import Laplace

In [161]: dist = Laplace(torch.tensor([10.0]), torch.tensor([0.990]))
          dist
Out[161]: Laplace()

In [162]: dist.sample()
Out[162]: tensor([10.2167])
```

A *normal distribution* is very useful because of the property of central limit theorem. It is defined by mean and standard deviations. If we know the mean and standard deviation of the distribution, we can estimate the event probabilities.

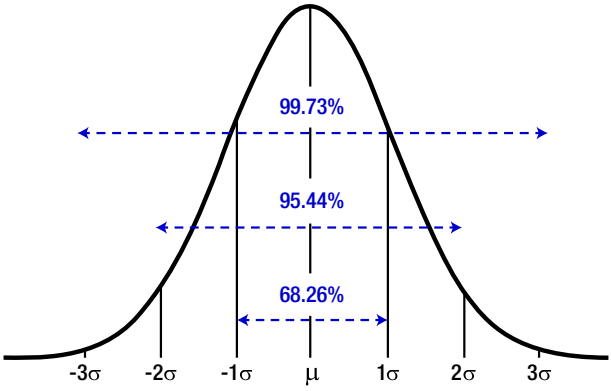


Figure 2-2. *Normal probability distribution*

```
In [163]: #Normal (Gaussian) distribution parameterized by loc and 'scale'.

In [165]: from torch.distributions.normal import Normal

In [166]: dist = Normal(torch.tensor([100.0]), torch.tensor([10.0]))
          dist
Out[166]: Normal()

In [167]: dist.sample()
Out[167]: tensor([95.2452])
```

Conclusion

This chapter discussed sampling distribution and generating random numbers from distributions. Neural networks are the primary focus in tensor-based operations. Any sort of machine learning or deep learning model implementation requires gradient computation, updating weight, computing bias, and continuously updating the bias.

This chapter also discussed the statistical distributions supported by PyTorch and the situations where each type of distribution can be applied.

The next chapter discusses deep learning models in detail. Those deep learning models include convolutional neural networks, recurrent neural networks, deep neural networks, and autoencoder models.

CHAPTER 3

CNN and RNN Using PyTorch

Probability and random variables are an integral part of computation in a graph-computing platform like PyTorch. Understanding probability and the associated concepts are essential. This chapter covers probability distributions and implementation using PyTorch, as well as how to interpret the results of a test. In probability and statistics, a random variable is also known as a *stochastic variable*, whose outcome is dependent on a purely stochastic phenomenon, or random phenomenon. There are different types of probability distribution, including normal distribution, binomial distribution, multinomial distribution, and the Bernoulli distribution. Each statistical distribution has its own properties.

Recipe 3-1. Setting Up a Loss Function

Problem

How do we set up a loss function and optimize it? Choosing the right loss function increases the chances of model convergence.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_3

Solution

In this recipe, we use another tensor as the update variable, and introduce the tensors to the sample model and compute the error or loss. Then we compute the rate of change in the loss function to measure the choice of loss function in model convergence.

How It Works

In the following example, t_c and t_u are two tensors. This can be constructed from any NumPy array.

```
In [1]: import torch

In [2]: torch.__version__
Out[2]: '0.4.1'

In [3]: torch.tensor
Out[3]: <function _VariableFunctions.tensor>

In [4]: t_c = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0])
        t_u = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4])
```

The sample model is just a linear equation to make the calculation happen and the loss function defined if the mean square error (MSE) shown next. Going forward in this chapter, we will increase the complexity of the model. For now, this is just a simple linear equation computation.

```
In [165]: t_c
Out[165]: tensor([ 0.5000, 14.0000, 15.0000, 28.0000, 11.0000,  8.0000,  3.0000, -4.0000,
                    6.0000, 13.0000, 21.0000])

In [166]: t_u
Out[166]: tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000, 21.8000,
                  48.4000, 60.4000, 68.4000])
```

Let's now define the model. The w parameter is the weight tensor, which is multiplied with the t_u tensor. The result is added with a constant tensor, b, and the loss function chosen is a custom-built one; it is also

available in PyTorch. In the following example, t_u is the tensor used, t_p is the tensor predicted, and t_c is the precomputed tensor, with which the predicted tensor needs to be compared to calculate the loss function.

```
In [5]: def model(t_u, w, b):
            return w * t_u + b
```

```
In [6]: def loss_fn(t_p, t_c):
            squared_diffs = (t_p - t_c)**2
            return squared_diffs.mean()
```

The formula w * t_u + b is the linear equation representation of a tensor-based computation.

```
In [7]: w = torch.ones(1)
        b = torch.zeros(1)

        t_p = model(t_u, w, b)
        t_p
Out[7]: tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000, 21.8000,
                48.4000, 60.4000, 68.4000])
```

```
In [8]: loss = loss_fn(t_p, t_c)
        loss
Out[8]: tensor(1763.8846)
```

The initial loss value is 1763.88, which is too high because of the initial round of weights chosen. The error in the first round of iteration is backpropagated to reduce the errors in the second round, for which the initial set of weights needs to be updated. Therefore, the rate of change in the loss function is essential in updating the weights in the estimation process.

```
In [9]: delta = 0.1

        loss_rate_of_change_w = (loss_fn(model(t_u,
                                               w + delta, b),
                                         t_c) - loss_fn(model(t_u, w - delta, b),
                                                        t_c)) / (2.0 * delta)
```

```
In [10]: learning_rate = 1e-2

         w = w - learning_rate * loss_rate_of_change_w
```

There are two parameters to update the rate of loss function: the learning rate at the current iteration and the learning rate at the previous iteration. If the delta between the two iterations exceeds a certain threshold, then the weight tensor needs to be updated, else model convergence could happen. The preceding script shows the delta and learning rate values. Currently, these are static values that the user has the option to change.

```
In [9]: delta = 0.1

        loss_rate_of_change_w = (loss_fn(model(t_u,
                                               w + delta, b),
                                         t_c) - loss_fn(model(t_u, w - delta, b),
                                                        t_c)) / (2.0 * delta)

In [10]: learning_rate = 1e-2

         w = w - learning_rate * loss_rate_of_change_w

In [11]: loss_rate_of_change_b = (loss_fn(model(t_u, w, b + delta), t_c) -
                                  loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)

         b = b - learning_rate * loss_rate_of_change_b
```

```
In [15]: b
Out[15]: tensor([46.0250])
```

This is how a simple mean square loss function works in a two-dimensional tensor example, with a tensor size of 10,5.

Let's look at the following example. The MSELoss function is within the neural network module of PyTorch.

```
In [26]: from torch import nn
         loss = nn.MSELoss()
         input = torch.randn(10, 5, requires_grad=True)
         target = torch.randn(10, 5)
         output = loss(input, target)
         output.backward()

In [27]: output
Out[27]: tensor(2.1542, grad_fn=<MseLossBackward>)
```

When we look at the gradient calculation that is used for backpropagation, it is shown as MSELoss.

```
In [30]: output.grad_fn
Out[30]: <MseLossBackward at 0x11db95978>
```

Recipe 3-2. Estimating the Derivative of the Loss Function

Problem

How do we estimate the derivative of a loss function?

Solution

Using the following example, we change the loss function to two times the differences between the input and the output tensors, instead of MSELoss function. The following grad_fn, which is defined as a custom function, shows the user how the final output retrieves the derivative of the loss function.

How It Works

Let's look at the following example. In the previous recipe, the last line of the script shows the grad_fn as an object embedded in the output object tensor. In this recipe, we explain how this is computed. grad_fn is a derivative of the loss function with respect to the parameters of the model. This is exactly what we do in the following grad_fn.

```
In [12]: def dloss_fn(t_p, t_c):
             dsq_diffs = 2 * (t_p - t_c)
             return dsq_diffs

In [13]: def model(t_u, w, b):
             return w * t_u + b

In [14]: def dmodel_dw(t_u, w, b):
             return t_u

In [15]: def dmodel_db(t_u, w, b):
             return 1.0

In [16]: def grad_fn(t_u, t_c, t_p, w, b):
             dloss_dw = dloss_fn(t_p, t_c) * dmodel_dw(t_u, w, b)
             dloss_db = dloss_fn(t_p, t_c) * dmodel_db(t_u, w, b)
             return torch.stack([dloss_dw.mean(), dloss_db.mean()])
```

The parameters are the input, bias settings, and the learning rate, and the number of epochs for the model training. The estimation of these parameters provides values to the equation.

```
In [17]: params = torch.tensor([1.0, 0.0])

         nepochs = 100

         learning_rate = 1e-2

         for epoch in range(nepochs):
             # forward pass
             w, b = params
             t_p = model(t_u, w, b)

             loss = loss_fn(t_p, t_c)
             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             grad = grad_fn(t_u, t_c, t_p, w, b)

             print('Params:', params)
             print('Grad:', grad)

             params = params - learning_rate * grad

         params
```

This is what the initial result looks like. Epoch is an iteration that produces a loss value from the loss function defined earlier. The params vector is about coefficients and constants that need to be changed to minimize the loss function. The grad function computes the feedback value to the next epoch. This is just an example. The number of epochs chosen is an iterative task depending on the input data, output data, and choice of loss and optimization functions.

```
Epoch 0, Loss 1763.884644
Params: tensor([1., 0.])
Grad: tensor([4517.2964,   82.6000])
Epoch 1, Loss 5802484.500000
Params: tensor([-44.1730,  -0.8260])
Grad: tensor([-261257.4062,   -4598.9707])
Epoch 2, Loss 19408031744.000000
Params: tensor([2568.4011,   45.1637])
Grad: tensor([15109615.0000,   266155.7188])
Epoch 3, Loss 64915909902336.000000
Params: tensor([-148527.7344,   -2616.3933])
Grad: tensor([-873852544.,  -15392727.])
Epoch 4, Loss 217130439561707520.000000
Params: tensor([8589997.0000,  151310.8750])
Grad: tensor([50538569728.,   890226304.])
Epoch 5, Loss 726257020202974707712.000000
Params: tensor([-496795712.,   -8750952.])
Grad: tensor([-2922858414080.,   -51485540352.])
Epoch 6, Loss 2429181687085405986357248.000000
```

If we reduce the learning rate, we are able to pass relevant values to the gradient, the parameter updates in a better way, and model convergence becomes quicker.

```
In [18]: params = torch.tensor([1.0, 0.0])

         nepochs = 100

         learning_rate = 1e-4

         for epoch in range(nepochs):
             # forward pass
             w, b = params
             t_p = model(t_u, w, b)

             loss = loss_fn(t_p, t_c)
             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             grad = grad_fn(t_u, t_c, t_p, w, b)

             print('Params:', params)
             print('Grad:', grad)

             params = params - learning_rate * grad

         params
```

The initial results look like as the following. The results are at epoch 5 and the loss value is 29.35, which is much lower than 1763.88 at epoch 0, and corresponding to the epoch, the estimated parameters are 0.24 and –.01, at epoch 100. These parameter values are optimal.

```
Epoch 0, Loss 1763.884644
Params: tensor([1., 0.])
Grad: tensor([4517.2964,   82.6000])
Epoch 1, Loss 323.090546
Params: tensor([ 0.5483, -0.0083])
Grad: tensor([1859.5493,   35.7843])
Epoch 2, Loss 78.929634
Params: tensor([ 0.3623, -0.0118])
Grad: tensor([765.4666,  16.5122])
Epoch 3, Loss 37.552845
Params: tensor([ 0.2858, -0.0135])
Grad: tensor([315.0790,   8.5787])
Epoch 4, Loss 30.540285
Params: tensor([ 0.2543, -0.0143])
Grad: tensor([129.6733,   5.3127])
Epoch 5, Loss 29.351152
Params: tensor([ 0.2413, -0.0149])
Grad: tensor([53.3496,  3.9682])
Epoch 6, Loss 29.148882
```

If we reduce the learning rate a bit, then the process of weight updating will be a little slower, which means that the epoch number needs to be increased in order to find a stable state for the model.

```
In [19]: t_un = 0.1 * t_u
```

```
In [20]: params = torch.tensor([1.0, 0.0])

nepochs = 100

learning_rate = 1e-2

for epoch in range(nepochs):
    # forward pass
    w, b = params
    t_p = model(t_un, w, b)

    loss = loss_fn(t_p, t_c)
    print('Epoch %d, Loss %f' % (epoch, float(loss)))

    # backward pass
    grad = grad_fn(t_un, t_c, t_p, w, b)

    print('Params:', params)
    print('Grad:', grad)

    params = params - learning_rate * grad

params
```

The following are the results that we observe.

```
Epoch 0, Loss 80.364342
Params: tensor([1., 0.])
Grad: tensor([-77.6140, -10.6400])
Epoch 1, Loss 37.574917
Params: tensor([1.7761, 0.1064])
Grad: tensor([-30.8623,  -2.3864])
Epoch 2, Loss 30.871077
Params: tensor([2.0848, 0.1303])
Grad: tensor([-12.4631,   0.8587])
Epoch 3, Loss 29.756193
Params: tensor([2.2094, 0.1217])
Grad: tensor([-5.2218,  2.1327])
Epoch 4, Loss 29.507149
Params: tensor([2.2616, 0.1004])
Grad: tensor([-2.3715,  2.6310])
Epoch 5, Loss 29.392458
Params: tensor([2.2853, 0.0740])
Grad: tensor([-1.2492,  2.8241])
Epoch 6, Loss 29.298828
```

If we increase the number of epochs, then what happens to the loss function and parameter tensor can be viewed in the following script, in which we print the loss value to find the minimum loss corresponding to the epoch. Then we can extract the best parameters from the model.

```
In [21]: params = torch.tensor([1.0, 0.0])

         nepochs = 5000

         learning_rate = 1e-2

         for epoch in range(nepochs):
             # forward pass
             w, b = params
             t_p = model(t_un, w, b)

             loss = loss_fn(t_p, t_c)
             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             grad = grad_fn(t_un, t_c, t_p, w, b)

             params = params - learning_rate * grad

         params
```

The following are the results.

```
Epoch 0, Loss 80.364342
Epoch 1, Loss 37.574917
Epoch 2, Loss 30.871077
Epoch 3, Loss 29.756193
Epoch 4, Loss 29.507149
Epoch 5, Loss 29.392458
Epoch 6, Loss 29.298828
Epoch 7, Loss 29.208717
Epoch 8, Loss 29.119417
Epoch 9, Loss 29.030487
Epoch 10, Loss 28.941875
Epoch 11, Loss 28.853565
Epoch 12, Loss 28.765556
Epoch 13, Loss 28.677851
Epoch 14, Loss 28.590431
Epoch 15, Loss 28.503321
Epoch 16, Loss 28.416496
Epoch 17, Loss 28.329975
Epoch 18, Loss 28.243738
```

The following is the final loss value at the final epoch level.

```
Epoch 4983, Loss 2.927648
Epoch 4984, Loss 2.927646
Epoch 4985, Loss 2.927648
Epoch 4986, Loss 2.927647
Epoch 4987, Loss 2.927647
Epoch 4988, Loss 2.927647
Epoch 4989, Loss 2.927648
Epoch 4990, Loss 2.927646
Epoch 4991, Loss 2.927648
Epoch 4992, Loss 2.927647
Epoch 4993, Loss 2.927646
Epoch 4994, Loss 2.927647
Epoch 4995, Loss 2.927648
Epoch 4996, Loss 2.927647
Epoch 4997, Loss 2.927647
Epoch 4998, Loss 2.927647
Epoch 4999, Loss 2.927648

Out[45]: tensor([  5.3671, -17.3012], requires_grad=True)
```

At epoch 5000, the loss value is 2.92, which is not going down further; hence, at this iteration level, the tensor output displays 5.36 as the final weight and –17.30 as the final bias. These are the final parameters from the model.

To fine-tune this model in estimating parameters, we can redefine the model and the loss function and apply it to the same example.

```
In [22]: def model(t_u, w, b):
             return w * t_u + b
```

```
In [23]: def loss_fn(t_p, t_c):
             sq_diffs = (t_p - t_c)**2
             return sq_diffs.mean()
```

Set up the parameters. After completing the training process, we should reset the grad function to None.

```
In [24]: params = torch.tensor([1.0, 0.0], requires_grad=True)

         loss = loss_fn(model(t_u, *params), t_c)
```

```
In [25]: params.grad is None
Out[25]: True
```

Recipe 3-3. Fine-Tuning a Model

Problem

How do we find the gradients of the loss function by applying an optimization function to optimize the loss function?

Solution

We'll use the backward() function.

How It Works

Let's look at the following example. The backward() function calculates the gradients of a function with respect to its parameters. In this section, we retrain the model with new set of hyperparameters.

```
In [26]: loss.backward()

In [27]: params.grad
Out[27]: tensor([4517.2969,   82.6000])
```

Reset the parameter grid. If we do not o reset the parameters in an existing session, the error values accumulated from any other session become mixed, so it is important to reset the parameter grid.

```
In [28]: if params.grad is not None:
             params.grad.zero_()

In [29]: def model(t_u, w, b):
             return w * t_u + b

In [30]: def loss_fn(t_p, t_c):
             sq_diffs = (t_p - t_c)**2
             return sq_diffs.mean()
```

After redefining the model and the loss function, let's retrain the model.

```
In [31]: params = torch.tensor([1.0, 0.0], requires_grad=True)

         nepochs = 5000

         learning_rate = 1e-2

In [32]: for epoch in range(nepochs):
             # forward pass
             t_p = model(t_un, *params)
             loss = loss_fn(t_p, t_c)

             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             if params.grad is not None:
                 params.grad.zero_()

             loss.backward()

             #params.grad.clamp_(-1.0, 1.0)
             #print(params, params.grad)

             params = (params - learning_rate * params.grad).detach().requires_grad_()

         params
```

We have taken 5000 epochs. We train the parameters in a backward propagation method and get the following results. At epoch 0, the loss value is 80.36. We try to minimize the loss value as we proceed with the next iteration by adjusting the learning rate. At the final epoch, we observe that the loss value is 2.92, which is same result as before but with a different loss function and using backpropagation.

```
Epoch 0, Loss 80.364342
Epoch 1, Loss 37.574917
Epoch 2, Loss 30.871077
Epoch 3, Loss 29.756193
Epoch 4, Loss 29.507149
Epoch 5, Loss 29.392458
Epoch 6, Loss 29.298828
Epoch 7, Loss 29.208717
Epoch 8, Loss 29.119417
Epoch 9, Loss 29.030487
Epoch 10, Loss 28.941875
Epoch 11, Loss 28.853565
Epoch 12, Loss 28.765556
Epoch 13, Loss 28.677851
Epoch 14, Loss 28.590431
Epoch 15, Loss 28.503321
Epoch 16, Loss 28.416496
Epoch 17, Loss 28.329973
Epoch 18, Loss 28.243738
```

```
Epoch 4983, Loss 2.927646
Epoch 4984, Loss 2.927648
Epoch 4985, Loss 2.927647
Epoch 4986, Loss 2.927648
Epoch 4987, Loss 2.927648
Epoch 4988, Loss 2.927648
Epoch 4989, Loss 2.927646
Epoch 4990, Loss 2.927648
Epoch 4991, Loss 2.927647
Epoch 4992, Loss 2.927647
Epoch 4993, Loss 2.927647
Epoch 4994, Loss 2.927648
Epoch 4995, Loss 2.927647
Epoch 4996, Loss 2.927646
Epoch 4997, Loss 2.927647
Epoch 4998, Loss 2.927648
Epoch 4999, Loss 2.927647

Out[37]: tensor([  5.3671, -17.3012], requires_grad=True)
```

The final model parameters are 5.3671 with a bias of –17.3012.

Recipe 3-4. Selecting an Optimization Function

Problem

How do we optimize the gradients with the function in Recipe 3-3?

Solution

There are certain functions that are embedded in PyTorch, and there are certain optimization functions that the user has to create.

How It Works

Let's look at the following example.

```
In [33]: import torch.optim as optim

         dir(optim)

Out[33]: ['ASGD',
          'Adadelta',
          'Adagrad',
          'Adam',
          'Adamax',
          'LBFGS',
          'Optimizer',
          'RMSprop',
          'Rprop',
          'SGD',
          'SparseAdam',
          '__builtins__',
          '__cached__',
          '__doc__',
          '__file__',
          '__loader__',
          '__name__',
          '__package__',
          '__path__',
          '__spec__',
          'lr_scheduler']
```

Each optimization method is unique in solving a problem. We will describe it later.

The Adam optimizer is a first-order, gradient-based optimization of stochastic objective functions. It is based on adaptive estimation of lower-order moments. This is computationally efficient enough for deployment on large datasets. To use torch.optim, we have to construct an optimizer object in our code that will hold the current state of the parameters and will update the parameters based on the computed gradients, moments, and learning rate. To construct an optimizer, we have to give it an iterable containing the parameters and ensure that all the parameters are variables to optimize. Then, we can specify optimizer-specific options, such as the learning rate, weight decay, moments, and so forth.

Adadelta is another optimizer that is fast enough to work on large datasets. This method does not require manual fine-tuning of the learning rate; the algorithm takes care of it internally.

```
In [34]: params = torch.tensor([1.0, 0.0], requires_grad=True)
         learning_rate = 1e-5
         optimizer = optim.SGD([params], lr=learning_rate)
```

```
In [35]: t_p = model(t_u, *params)
         loss = loss_fn(t_p, t_c)
         loss.backward()
         optimizer.step()
         params
Out[35]: tensor([ 0.9548, -0.0008], requires_grad=True)
```

```
In [36]: params = torch.tensor([1.0, 0.0], requires_grad=True)
         learning_rate = 1e-2
         optimizer = optim.SGD([params], lr=learning_rate)
         t_p = model(t_un, *params)
         loss = loss_fn(t_p, t_c)
         optimizer.zero_grad()
         loss.backward()
         optimizer.step()
         params
Out[36]: tensor([1.7761, 0.1064], requires_grad=True)
```

Now let's call the model and loss function out once again and apply them along with the optimization function.

```
In [37]: def model(t_u, w, b):
             return w * t_u + b
```

```
In [38]: def loss_fn(t_p, t_c):
             sq_diffs = (t_p - t_c)**2
             return sq_diffs.mean()
```

```
In [39]: params = torch.tensor([1.0, 0.0], requires_grad=True)

         nepochs = 5000
         learning_rate = 1e-2

         optimizer = optim.SGD([params], lr=learning_rate)
```

```
In [40]: for epoch in range(nepochs):

             # forward pass
             t_p = model(t_un, *params)
             loss = loss_fn(t_p, t_c)

             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             optimizer.zero_grad()
             loss.backward()
             optimizer.step()

         t_p = model(t_un, *params)

         params
```

Let's look at the gradient in a loss function. Using the optimization library, we can try to find the best value of the loss function.

```
Epoch 0, Loss 80.364342
Epoch 1, Loss 37.574917
Epoch 2, Loss 30.871077
Epoch 3, Loss 29.756193
Epoch 4, Loss 29.507149
Epoch 5, Loss 29.392458
Epoch 6, Loss 29.298828
Epoch 7, Loss 29.208717
Epoch 8, Loss 29.119417
Epoch 9, Loss 29.030487
Epoch 10, Loss 28.941875
Epoch 11, Loss 28.853565
Epoch 12, Loss 28.765556
Epoch 13, Loss 28.677851
Epoch 14, Loss 28.590431
Epoch 15, Loss 28.503321
Epoch 16, Loss 28.416496
Epoch 17, Loss 28.329975
Epoch 18, Loss 28.243738
```

The example has two custom functions and a loss function. We have taken two small tensor values. The new thing is that we have taken the optimizer to find the minimum value.

In the following example, we have chosen Adam as the optimizer.

```
In [41]: def model(t_u, w, b):
             return w * t_u + b

In [42]: def loss_fn(t_p, t_c):
             sq_diffs = (t_p - t_c)**2
             return sq_diffs.mean()

In [43]: params = torch.tensor([1.0, 0.0], requires_grad=True)

         nepochs = 5000
         learning_rate = 1e-1

         optimizer = optim.Adam([params], lr=learning_rate)

In [44]: for epoch in range(nepochs):
             # forward pass
             t_p = model(t_u, *params)
             loss = loss_fn(t_p, t_c)

             print('Epoch %d, Loss %f' % (epoch, float(loss)))

             # backward pass
             optimizer.zero_grad()
             loss.backward()
             optimizer.step()

         t_p = model(t_u, *params)

         params
```

```
Epoch 0, Loss 1763.884644
Epoch 1, Loss 1334.349121
Epoch 2, Loss 967.815857
Epoch 3, Loss 664.756348
Epoch 4, Loss 424.630096
Epoch 5, Loss 245.539536
Epoch 6, Loss 123.854935
Epoch 7, Loss 53.885216
Epoch 8, Loss 27.729158
Epoch 9, Loss 35.472927
Epoch 10, Loss 65.865623
Epoch 11, Loss 107.448486
Epoch 12, Loss 149.893204
Epoch 13, Loss 185.160660
Epoch 14, Loss 208.152191
Epoch 15, Loss 216.751862
Epoch 16, Loss 211.386810
Epoch 17, Loss 194.333359
Epoch 18, Loss 168.973526
```

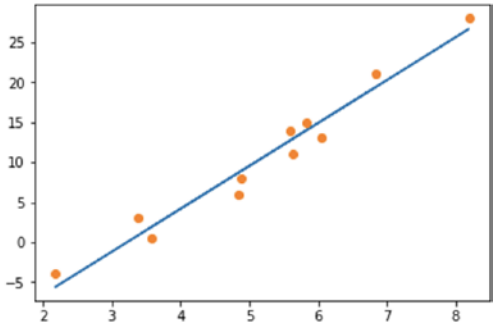
In the preceding code, we computed the optimized parameters and computed the predicted tensors using the actual and predicted tensors. We can display a graph that has a line shown as a regression line.

```
Epoch 4983, Loss 2.927648
Epoch 4984, Loss 2.927646
Epoch 4985, Loss 2.927648
Epoch 4986, Loss 2.927647
Epoch 4987, Loss 2.927647
Epoch 4988, Loss 2.927647
Epoch 4989, Loss 2.927648
Epoch 4990, Loss 2.927646
Epoch 4991, Loss 2.927648
Epoch 4992, Loss 2.927647
Epoch 4993, Loss 2.927646
Epoch 4994, Loss 2.927647
Epoch 4995, Loss 2.927648
Epoch 4996, Loss 2.927647
Epoch 4997, Loss 2.927647
Epoch 4998, Loss 2.927647
Epoch 4999, Loss 2.927648

Out[45]: tensor([  5.3671, -17.3012], requires_grad=True)
```

Let's visualize the sample data in graphical form using the actual and predicted tensors.

```
In [45]: from matplotlib import pyplot as plt
         %matplotlib inline
         plt.plot(0.1 * t_u.numpy(), t_p.detach().numpy())
         plt.plot(0.1 * t_u.numpy(), t_c.numpy(), 'o')
```

```
Out[45]: [<matplotlib.lines.Line2D at 0x115d13d68>]
```



Recipe 3-5. Further Optimizing the Function

Problem

How do we optimize the training set and test it with a validation set using random samples?

Solution

We'll go through the process of further optimization.

How It Works

Let's look at the following example. Here we set the number of samples, then we take 20% of the data as validation samples using shuffled_indices. We took random samples of all the records. The objective of the train and validation set is to build a model in a training set, make the prediction on the validation set, and check the accuracy of the model.

```
In [46]: n_samples = t_u.shape[0]
         n_val = int(0.2 * n_samples)

         shuffled_indices = torch.randperm(n_samples)

         train_indices = shuffled_indices[:-n_val]
         val_indices = shuffled_indices[-n_val:]

         train_indices, val_indices
Out[46]: (tensor([ 4,  6, 10,  7,  2,  1,  0,  9,  8]), tensor([3, 5]))
```

```
In [47]: t_u_train = t_u[train_indices]
         t_c_train = t_c[train_indices]

         t_u_val = t_u[val_indices]
         t_c_val = t_c[val_indices]
```

```
In [48]: def model(t_u, w, b):
             return w * t_u + b
```

```
In [49]: def loss_fn(t_p, t_c):
             sq_diffs = (t_p - t_c)**2
             return sq_diffs.mean()
```

```
In [50]: params = torch.tensor([1.0, 0.0], requires_grad=True)

         nepochs = 5000
         learning_rate = 1e-2

         optimizer = optim.SGD([params], lr=learning_rate)

         t_un_train = 0.1 * t_u_train
         t_un_val = 0.1 * t_u_val
```

Now let's run the train and validation process. We first take the training input data and multiply it by the parameter's next line. We make a prediction and compute the loss function. Using the same model in third line, we make predictions and then we evaluate the loss function for the validation dataset. In the backpropagation process, we calculate the gradient of the loss function for the training set, and using the optimizer, we update the parameters.

```
In [51]: for epoch in range(nepochs):

             # forward pass
             t_p_train = model(t_un_train, *params)
             loss_train = loss_fn(t_p_train, t_c_train)

             t_p_val = model(t_un_val, *params)
             loss_val = loss_fn(t_p_val, t_c_val)

             print('Epoch %d, Training loss %f, Validation loss %f' % (epoch, float(loss_train),
                                                                       float(loss_val)))

             # backward pass
             optimizer.zero_grad()
             loss_train.backward()
             optimizer.step()

         t_p = model(t_un, *params)

         params
```

```
Epoch 0, Training loss 53.544399, Validation loss 201.054077
Epoch 1, Training loss 30.566721, Validation loss 115.439827
Epoch 2, Training loss 25.551018, Validation loss 85.457054
Epoch 3, Training loss 24.416952, Validation loss 73.581451
Epoch 4, Training loss 24.121834, Validation loss 68.444214
Epoch 5, Training loss 24.008158, Validation loss 66.075096
Epoch 6, Training loss 23.933849, Validation loss 64.913399
Epoch 7, Training loss 23.868191, Validation loss 64.295708
Epoch 8, Training loss 23.804564, Validation loss 63.927151
Epoch 9, Training loss 23.741520, Validation loss 63.673698
Epoch 10, Training loss 23.678753, Validation loss 63.473740
Epoch 11, Training loss 23.616192, Validation loss 63.298817
Epoch 12, Training loss 23.553831, Validation loss 63.135735
Epoch 13, Training loss 23.491661, Validation loss 62.978333
Epoch 14, Training loss 23.429680, Validation loss 62.823845
Epoch 15, Training loss 23.367884, Validation loss 62.670914
Epoch 16, Training loss 23.306282, Validation loss 62.518906
Epoch 17, Training loss 23.244867, Validation loss 62.367603
Epoch 18, Training loss 23.183636, Validation loss 62.216805
```

The following are the last 10 epochs and their results.

```
Epoch 4982, Training loss 2.983931, Validation loss 3.215126
Epoch 4983, Training loss 2.983923, Validation loss 3.215217
Epoch 4984, Training loss 2.983915, Validation loss 3.215302
Epoch 4985, Training loss 2.983905, Validation loss 3.215386
Epoch 4986, Training loss 2.983897, Validation loss 3.215471
Epoch 4987, Training loss 2.983890, Validation loss 3.215561
Epoch 4988, Training loss 2.983881, Validation loss 3.215646
Epoch 4989, Training loss 2.983873, Validation loss 3.215738
Epoch 4990, Training loss 2.983866, Validation loss 3.215823
Epoch 4991, Training loss 2.983858, Validation loss 3.215914
Epoch 4992, Training loss 2.983848, Validation loss 3.215999
Epoch 4993, Training loss 2.983841, Validation loss 3.216077
Epoch 4994, Training loss 2.983834, Validation loss 3.216171
Epoch 4995, Training loss 2.983825, Validation loss 3.216249
Epoch 4996, Training loss 2.983817, Validation loss 3.216341
Epoch 4997, Training loss 2.983809, Validation loss 3.216427
Epoch 4998, Training loss 2.983802, Validation loss 3.216522
Epoch 4999, Training loss 2.983793, Validation loss 3.216608

Out[56]: tensor([  5.3768, -17.6654], requires_grad=True)
```

In the previous step, the gradient was set to true. In the following set, we disable gradient calculation by using the torch.no_grad() function. The rest of the syntax remains same. Disabling gradient calculation is useful for drawing inferences, when we are sure that we will not call `Tensor.backward()`. This reduces memory consumption for computations that would otherwise be `requires_grad=True`.

```
In [52]: for epoch in range(nepochs):

             # forward pass
             t_p_train = model(t_un_train, *params)
             loss_train = loss_fn(t_p_train, t_c_train)

             with torch.no_grad():
                 t_p_val = model(t_un_val, *params)
                 loss_val = loss_fn(t_p_val, t_c_val)

             print('Epoch %d, Training loss %f, Validation loss %f' % (epoch, float(loss_train),
                                                                       float(loss_val)))

             # backward pass
             optimizer.zero_grad()
             loss_train.backward()
             optimizer.step()
```

```
Epoch 0, Training loss 2.978740, Validation loss 3.361475
Epoch 1, Training loss 2.978740, Validation loss 3.361475
Epoch 2, Training loss 2.978740, Validation loss 3.361475
Epoch 3, Training loss 2.978740, Validation loss 3.361475
Epoch 4, Training loss 2.978740, Validation loss 3.361475
Epoch 5, Training loss 2.978740, Validation loss 3.361475
Epoch 6, Training loss 2.978740, Validation loss 3.361475
Epoch 7, Training loss 2.978740, Validation loss 3.361475
Epoch 8, Training loss 2.978740, Validation loss 3.361475
Epoch 9, Training loss 2.978740, Validation loss 3.361475
Epoch 10, Training loss 2.978740, Validation loss 3.361475
Epoch 11, Training loss 2.978740, Validation loss 3.361475
Epoch 12, Training loss 2.978740, Validation loss 3.361475
Epoch 13, Training loss 2.978740, Validation loss 3.361475
Epoch 14, Training loss 2.978740, Validation loss 3.361475
Epoch 15, Training loss 2.978740, Validation loss 3.361475
Epoch 16, Training loss 2.978740, Validation loss 3.361475
Epoch 17, Training loss 2.978740, Validation loss 3.361475
Epoch 18, Training loss 2.978740, Validation loss 3.361475
```

The last rounds of epochs are displayed in other lines of code, as follows.

```
Epoch 4983, Training loss 2.978740, Validation loss 3.361475
Epoch 4984, Training loss 2.978740, Validation loss 3.361475
Epoch 4985, Training loss 2.978740, Validation loss 3.361475
Epoch 4986, Training loss 2.978740, Validation loss 3.361475
Epoch 4987, Training loss 2.978740, Validation loss 3.361475
Epoch 4988, Training loss 2.978740, Validation loss 3.361475
Epoch 4989, Training loss 2.978740, Validation loss 3.361475
Epoch 4990, Training loss 2.978740, Validation loss 3.361475
Epoch 4991, Training loss 2.978740, Validation loss 3.361475
Epoch 4992, Training loss 2.978740, Validation loss 3.361475
Epoch 4993, Training loss 2.978740, Validation loss 3.361475
Epoch 4994, Training loss 2.978740, Validation loss 3.361475
Epoch 4995, Training loss 2.978740, Validation loss 3.361475
Epoch 4996, Training loss 2.978740, Validation loss 3.361475
Epoch 4997, Training loss 2.978740, Validation loss 3.361475
Epoch 4998, Training loss 2.978740, Validation loss 3.361475
Epoch 4999, Training loss 2.978740, Validation loss 3.361475

Out[153]: tensor([  5.4408, -18.0102], requires_grad=True)
```

The final parameters are 5.44 and –18.012.

Recipe 3-6. Implementing a Convolutional Neural Network (CNN)

Problem

How do we implement a convolutional neural network using PyTorch?

Solution

There are various built-in datasets available on torchvision. We are considering the MNIST dataset and trying to build a CNN model.

How It Works

Let's look at the following example. As a first step, we set up the hyperparameters. The second step is to set up the architecture. The last step is to train the model and make predictions.

```
In [54]: import torch
         import torch.nn as nn
         from torch.autograd import Variable
         import torch.utils.data as Data
         import torchvision
         import matplotlib.pyplot as plt
         %matplotlib inline

In [55]: torch.manual_seed(1)    # reproducible

Out[55]: <torch._C.Generator at 0x1156e82f0>
```

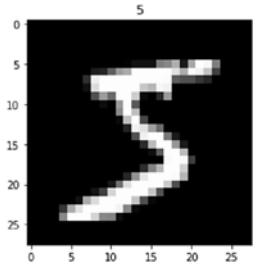
In the preceding code, we are importing the necessary libraries for deploying the convolutional neural network model using the digits dataset. The MNIST digits dataset is the most popular dataset in deep learning for computer vision and image processing.

In [61]:
```
# Hyper Parameters
EPOCH = 1
# train the input data n times, to save time, we just train 1 epoch
BATCH_SIZE = 50
# 50 samples at a time to pass through the epoch
LR = 0.001
# learning rate
DOWNLOAD_MNIST = True
# set to False if you have downloaded
```

In [62]:
```
# Mnist digits dataset
train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,
    # this is training data
    transform=torchvision.transforms.ToTensor(),
    # torch.FloatTensor of shape (Color x Height x Width) and
    #normalize in the range [0.0, 1.0]
    download=DOWNLOAD_MNIST,
    # download it if you don't have it
)
```

In [58]:
```
# plot one example
print(train_data.train_data.size())                 # (60000, 28, 28)
print(train_data.train_labels.size())               # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()
```

```
torch.Size([60000, 28, 28])
torch.Size([60000])
```



Let's load the dataset using the loader functionality.

```
In [59]: # Data Loader for easy mini-batch return in training, the image batch shape will be
         #(50, 1, 28, 28)
         train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
```

```
In [60]: # convert test data into Variable, pick 2000 samples to speed up testing
         test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)
         test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1)).
         type(torch.FloatTensor)[:2000]/255.
         # shape from (2000, 28, 28) to (2000, 1, 28, 28), value in range(0,1)
         test_y = test_data.test_labels[:2000]
```

In convolutional neural network architecture, the input image is converted to a feature set as set by color times height and width of the image. Because of the dimensionality of the dataset, we cannot model it to predict the output. The output layer in the preceding graph has classes such as car, truck, van, and bicycle. The input bicycle image has features that the CNN model should make use of and predict it correctly. The convolution layer is always accompanied by the pooling layer, which can be max pooling and average pooling. The different layers of pooling and convolution continue until the dimensionality is reduced to a level where we can use fully connected simple neural networks to predict the correct classes.

```
In [61]: class CNN(nn.Module):
             def __init__(self):
                 super(CNN, self).__init__()
                 self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
                     nn.Conv2d(
                         in_channels=1,              # input height
                         out_channels=16,            # n_filters
                         kernel_size=5,              # filter size
                         stride=1,                   # filter movement/step
                         padding=2,
                         # if want same width and length of this image after con2d,
                         #padding=(kernel_size-1)/2 if stride=1
                     ),                              # output shape (16, 28, 28)
                     nn.ReLU(),                      # activation
                     nn.MaxPool2d(kernel_size=2),
                     # choose max value in 2x2 area, output shape (16, 14, 14)
                 )
                 self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
                     nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
                     nn.ReLU(),                      # activation
                     nn.MaxPool2d(2),                # output shape (32, 7, 7)
                 )
                 self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

             def forward(self, x):
                 x = self.conv1(x)
                 x = self.conv2(x)
                 x = x.view(x.size(0), -1)
                 # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
                 output = self.out(x)
                 return output, x    # return x for visualization
```

In [62]:
```
cnn = CNN()
print(cnn)  # net architecture
```

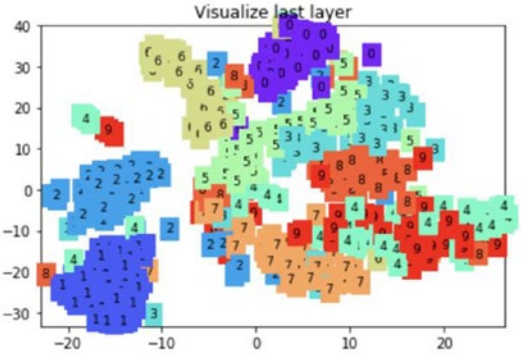
```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

In [63]:
```
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

In [65]:
```
from matplotlib import cm
try: from sklearn.manifold import TSNE; HAS_SK = True
except: HAS_SK = False; print('Please install sklearn for layer visualization, if not there')
def plot_with_labels(lowDWeights, labels):
    plt.cla()
    X, Y = lowDWeights[:, 0], lowDWeights[:, 1]
    for x, y, s in zip(X, Y, labels):
        c = cm.rainbow(int(255 * s / 9)); plt.text(x, y, s, backgroundcolor=c, fontsize=9)
    plt.xlim(X.min(), X.max()); plt.ylim(Y.min(), Y.max()); plt.title('Visualize last layer');
    #plt.pause(0.01)

plt.ion()
```

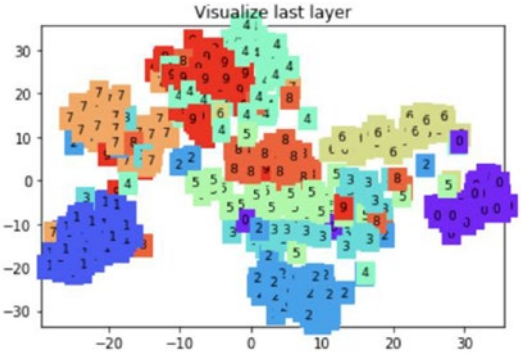
```
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):
        # gives batch data, normalize x when iterate train_loader
        b_x = Variable(x)   # batch x
        b_y = Variable(y)   # batch y

        output = cnn(b_x)[0]               # cnn output
        loss = loss_func(output, b_y)   # cross entropy loss
        optimizer.zero_grad()           # clear gradients for this training step
        loss.backward()                 # backpropagation, compute gradients
        optimizer.step()                # apply gradients

        if step % 100 == 0:
            test_output, last_layer = cnn(test_x)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == test_y).sum().item() / float(test_y.size(0))
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data[0],
                  '| test accuracy: %.2f' % accuracy)
            if HAS_SK:
                # Visualization of trained flatten layer (T-SNE)
                tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
                plot_only = 500
                low_dim_embs = tsne.fit_transform(last_layer.data.numpy()[:plot_only, :])
                labels = test_y.numpy()[:plot_only]
                plot_with_labels(low_dim_embs, labels)
plt.ioff()
```

```
Epoch:  0 | train loss: 2.2964 | test accuracy: 0.16
```




```
Epoch:  0 | train loss: 0.4369 | test accuracy: 0.88
```

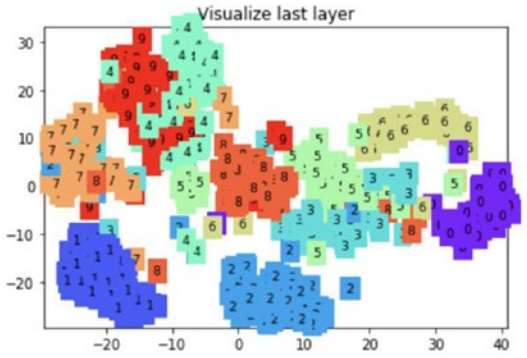
In the preceding graph, if we look at the number 4, it is scattered throughout the graph. Ideally, all of the 4s are closer to each other. This is because the test accuracy was very low.




```
Epoch:  0 | train loss: 0.1482 | test accuracy: 0.94
```

Visualize last layer

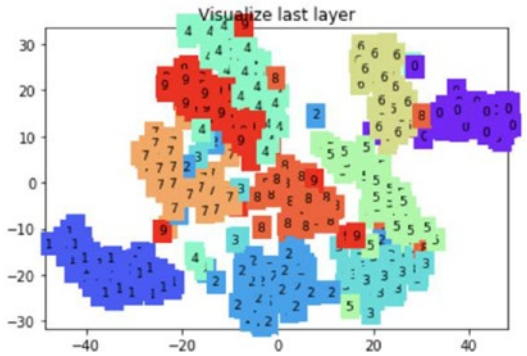
In this iteration, the training loss is reduced from 0.4369 to 0.1482 and the test accuracy improves from 16% to 94%. The digits with the same color are placed closely on the graph.




```
Epoch:  0 | train loss: 0.1407 | test accuracy: 0.95
```

In the next epoch, the test accuracy on the MNIST digits dataset the accuracy increases to 95%.

```
Epoch:  0 | train loss: 0.0450 | test accuracy: 0.98
```



In the final step/epoch, the digits with similar numbers are placed together. After training a model successfully, the next step is to make use of the model to predict. The following code explains the predictions process. The output object is numbered as 0, 1, 2, and so forth. The following shows the real and predicted numbers.

```
In [66]: # print 10 predictions from test data
         test_output, _ = cnn(test_x[:10])
         pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
         print(pred_y, 'prediction number')
         print(test_y[:10].numpy(), 'real number')

         [7 2 1 0 4 1 4 9 5 9] prediction number
         [7 2 1 0 4 1 4 9 5 9] real number
```

Recipe 3-7. Reloading a Model

Problem

How do we store and re-upload a model that has already been trained? Given the nature of deep learning models, which typically require a larger training time, the computational process creates a huge cost to the company. Can we retrain the model with new inputs and store the model?

Solution

In the production environment, we typically cannot train and predict at the same time because the training process takes a very long time. The prediction services cannot be applied until the training process using epoch is completed, the prediction services cannot be applied. Disassociating the training process from the prediction process is required; therefore, we need to store the application's trained model and continue until the next phase of training is done.

How It Works

Let's look at the following example, where we are creating the save function, which uses the Torch neural network module to create the model and the restore_net() function to get back the neural network model that was trained earlier.

```
In [68]: import torch
         from torch.autograd import Variable
         import matplotlib.pyplot as plt
         %matplotlib inline

         torch.manual_seed(1)    # reproducible

Out[68]: <torch._C.Generator at 0x1156e82f0>

In [69]: #sample data
         x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)  # x data (tensor), shape=(100, 1)
         y = x.pow(2) + 0.2*torch.rand(x.size())  # noisy y data (tensor), shape=(100, 1)
         x, y = Variable(x, requires_grad=False), Variable(y, requires_grad=False)
```

The preceding script contains a dependent Y variable and an independent X variable as sample data points to create a neural network model. The following `save` function stores the model. The `net1` object is the trained neural network model, which can be stored using two different protocols: (1) save the entire neural network model with all the weights and biases, and (2) save the model using only the weights. If the trained model object is very heavy in terms of size, we should save only the parameters that are weights; if the trained object size is low, then the entire model can be stored.

```
In [70]: def save():
             # save net1
             net1 = torch.nn.Sequential(
                 torch.nn.Linear(1, 10),
                 torch.nn.ReLU(),
                 torch.nn.Linear(10, 1)
             )
             optimizer = torch.optim.SGD(net1.parameters(), lr=0.5)
             loss_func = torch.nn.MSELoss()

             for t in range(100):
                 prediction = net1(x)
                 loss = loss_func(prediction, y)
                 optimizer.zero_grad()
                 loss.backward()
                 optimizer.step()

             # plot result
             plt.figure(1, figsize=(10, 3))
             plt.subplot(131)
             plt.title('Net1')
             plt.scatter(x.data.numpy(), y.data.numpy())
             plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)

             # 2 ways to save the net
             torch.save(net1, 'net.pkl')  # save entire net
             torch.save(net1.state_dict(), 'net_params.pkl')   # save only the parameters
```

The prebuilt neural network model can be reloaded to the existing PyTorch session by using the `load` function. To test the `net1` object and make predictions, we load the `net1` object and store the model as `net2`. By using the `net2` object, we can predict the outcome variable. The following script generates the graph as a dependent and an independent variable. `prediction.data.numpy()` in the last line of the code shows the predicted result.

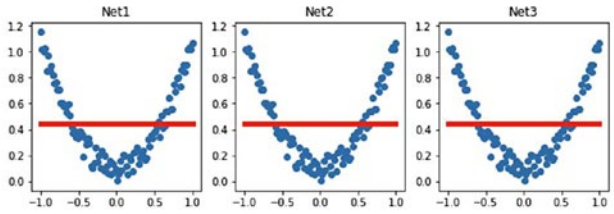
```
In [71]: def restore_net():
             # restore entire net1 to net2
             net2 = torch.load('net.pkl')
             prediction = net2(x)

             # plot result
             plt.subplot(132)
             plt.title('Net2')
             plt.scatter(x.data.numpy(), y.data.numpy())
             plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
```

Loading the pickle file format of the entire neural network is relatively slow; however, if we are only making predictions for a new dataset, we can only load the parameters of the model in a pickle format rather than the whole network.

```
In [72]: def restore_params():
             # restore only the parameters in net1 to net3
             net3 = torch.nn.Sequential(
                 torch.nn.Linear(1, 10),
                 torch.nn.ReLU(),
                 torch.nn.Linear(10, 1)
             )

             # copy net1's parameters into net3
             net3.load_state_dict(torch.load('net_params.pkl'))
             prediction = net3(x)

             # plot result
             plt.subplot(133)
             plt.title('Net3')
             plt.scatter(x.data.numpy(), y.data.numpy())
             plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
             plt.show()
```

Reuse the model. The `restore` function makes sure that the trained parameters can be reused by the model. To restore the model, we can use the load_state_dict() function to load the parameters of the model. If we see the following three models in the graph, they are identical, because net2 and net3 are copies of net1.

```
In [73]: # save net1
         save()
         # restore entire net (may slow)
         restore_net()
         # restore only the net parameters
         restore_params()
```



Recipe 3-8. Implementing a Recurrent Neural Network (RNN)

Problem

How do we set up a recurrent neural network using the MNIST dataset?

Solution

The recurrent neural network is considered as a memory network. We will use the epoch as 1 and a batch size of 64 samples at a time to establish the connection between the input and the output. Using the RNN model, we can predict the digits present in the images.

How It Works

Let's look at the following example. The recurrent neural network takes a sequence of vectors in the input layer and produces a sequence of vectors in the output layer. The information sequence is processed through the internal state transfer in the recurrent layer. Sometimes the output values have a long dependency in past historical values. This is another variant of the RNN model: the long short-term memory (LSTM) model. This is applicable for

any sort of domain where the information is consumed in a sequential manner; for example, in a time series where the current stock price is decided by the historical stock price, where the dependency can be short or long. Similarly, the context prediction using the long and short range of textual input vectors. There are other industry use cases, such as noise classification, where noise is also a sequence of information.

The following piece of code explains the execution of RNN model using PyTorch module.

There are three sets of weights: U, V and W. The set of weights vector, represented by W, is for passing information among the memory cells in the network that display communication among the hidden state. RNN uses an embedding layer using the Word2vec representation. The embedding matrix is the size of the number of words by the number of neurons in the hidden layer. If you have 20,000 words and 1000 hidden units, for example, the matrix has a 20,000×1000 size of the embedding layer. The new representations are passed to LSTM cells, which go to a sigmoid output layer.

```
In [75]: import torch
         from torch import nn
         from torch.autograd import Variable
         import torchvision.datasets as dsets
         import torchvision.transforms as transforms
         import matplotlib.pyplot as plt
         %matplotlib inline

In [76]: torch.manual_seed(1)    # reproducible

Out[76]: <torch._C.Generator at 0x1156e82f0>

In [77]: # Hyper Parameters
         EPOCH = 1               # train the training data n times, to save time, we just train 1 epoch
         BATCH_SIZE = 64
         TIME_STEP = 28          # rnn time step / image height
         INPUT_SIZE = 28         # rnn input size / image width
         LR = 0.01               # learning rate
         DOWNLOAD_MNIST = True   # set to True if haven't download the data
```

The RNN models have hyperparameters, such as the number of iterations (EPOCH); batch size dependent on the memory available in a single machine; a time step to remember the sequence of information; input size, which shows the vector size; and learning rate. The selection of

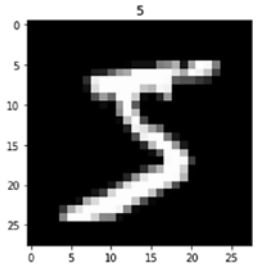
these values is indicative; we cannot depend on them for other use cases. The value selection for hyperparameter tuning is an iterative process; either you can choose multiple parameters and decide which one is working, or do parallel training of the model and decide which one is working fine.

```
In [78]: # Mnist digital dataset
train_data = dsets.MNIST(
    root='./mnist/',
    train=True,                         # this is training data
    transform=transforms.ToTensor(),    # Converts a PIL.Image or numpy.ndarray to
                                        # torch.FloatTensor of shape (C x H x W) and normalize
    download=DOWNLOAD_MNIST,            # download it if you don't have it
)
```

Using the dsets.MINIST() function, we can load the dataset to the current session. If you need to store the dataset, then download it locally.

```
In [79]: # plot one example
print(train_data.train_data.size())     # (60000, 28, 28)
print(train_data.train_labels.size())   # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()
```

```
torch.Size([60000, 28, 28])
torch.Size([60000])
```



The preceding script shows what the sample image dataset would look like. To train the deep learning model, we need to convert the whole training dataset into mini batches, which help us with averaging the final accuracy of the model. By using the data loader function, we can load the training data and prepare the mini batches. The purpose of the shuffle selection in mini batches is to ensure that the model captures all the variations in the actual dataset.

```
In [80]: # Data Loader for easy mini-batch return in training
         train_loader = torch.utils.data.DataLoader(dataset=train_data,
                                                    batch_size=BATCH_SIZE, shuffle=True)

In [81]: # convert test data into Variable, pick 2000 samples to speed up testing
         test_data = dsets.MNIST(root='./mnist/', train=False, transform=transforms.ToTensor())
         test_x = Variable(test_data.test_data, volatile=True).type(torch.FloatTensor)[:2000]/255.
         # shape (2000, 28, 28) value in range(0,1)
         test_y = test_data.test_labels.numpy().squeeze()[:2000]    # covert to numpy array
```

The preceding script prepares the training dataset. The test data is captured with the flag `train=False`. It is transformed to a tensor using the test data random sample of 2000 each at a time is picked up for testing the model. The test features set is converted to a variable format and the test label vector is represented in a NumPy array format.

```
In [82]: class RNN(nn.Module):
             def __init__(self):
                 super(RNN, self).__init__()

                 self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
                     input_size=INPUT_SIZE,
                     hidden_size=64,         # rnn hidden unit
                     num_layers=1,           # number of rnn layer
                     batch_first=True,       # input & output will has batch size as 1s dimension. e.g.
                 )

                 self.out = nn.Linear(64, 10)

             def forward(self, x):
                 # x shape (batch, time_step, input_size)
                 # r_out shape (batch, time_step, output_size)
                 # h_n shape (n_layers, batch, hidden_size)
                 # h_c shape (n_layers, batch, hidden_size)
                 r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

                 # choose r_out at the last time step
                 out = self.out(r_out[:, -1, :])
                 return out
```

In the preceding RNN class, we are training an LSTM network, which is proven effective for holding memory for a long time, and thus helps in learning. If we use the nn.RNN() model, it hardly learns the parameters, because the vanilla implementation of RNN cannot hold or remember the information for a long period of time. In the LSTM network, the image width is considered the input size, hidden size is decided as the number of neurons in the hidden layer, `num_layers` shows the number of RNN layers in the network.

The RNN module, within the LSTM module, produces the output as a vector size of 64×10 because the output layer has digits to be classified as 0 to 9. The last forward function shows how to proceed with forward propagation in an RNN network.

The following script shows how the LSTM model is processed under the RNN class. In the LSTM function, we pass the input length as 28 and the number of neurons in the hidden layer as 64, and from the hidden 64 neurons to the output 10 neurons.

```
In [83]: rnn = RNN()
         print(rnn)

         RNN(
           (rnn): LSTM(28, 64, batch_first=True)
           (out): Linear(in_features=64, out_features=10, bias=True)
         )
```

```
In [84]: optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)   # optimize all cnn parameters
         loss_func = nn.CrossEntropyLoss()                      # the target label is not one-hotted
```

To optimize all RNN parameters, we use the Adam optimizer. Inside the function, we use the learning rate as well. The loss function used in this example is the cross-entropy loss function. We need to provide multiple epochs to get the best parameters.

In the following script, we are printing the training loss and the test accuracy. After one epoch, the test accuracy increases to 95% and the training loss reduces to 0.24.

```
In [85]: # training and testing
         for epoch in range(EPOCH):
             for step, (x, y) in enumerate(train_loader):        # gives batch data
                 b_x = Variable(x.view(-1, 28, 28))              # reshape x to (batch, time_step, input_
                 b_y = Variable(y)                               # batch y

                 output = rnn(b_x)                               # rnn output
                 loss = loss_func(output, b_y)                   # cross entropy loss
                 optimizer.zero_grad()                           # clear gradients for this training ste
                 loss.backward()                                 # backpropagation, compute gradients
                 optimizer.step()                                # apply gradients

                 if step % 50 == 0:
                     test_output = rnn(test_x)                   # (samples, time_step, input_size)
                     pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
                     accuracy = sum(pred_y == test_y) / float(test_y.size)
                     print('Epoch: ', epoch, '| train loss: %.4f' % loss.data[0], '| test accuracy: %.2f'
```

```
Epoch:  0 | train loss: 2.2883 | test accuracy: 0.10
Epoch:  0 | train loss: 0.8138 | test accuracy: 0.62
Epoch:  0 | train loss: 0.9010 | test accuracy: 0.78
Epoch:  0 | train loss: 0.6608 | test accuracy: 0.83
Epoch:  0 | train loss: 0.3150 | test accuracy: 0.85
Epoch:  0 | train loss: 0.2186 | test accuracy: 0.91
Epoch:  0 | train loss: 0.4511 | test accuracy: 0.90
Epoch:  0 | train loss: 0.4673 | test accuracy: 0.90
Epoch:  0 | train loss: 0.2014 | test accuracy: 0.93
Epoch:  0 | train loss: 0.2198 | test accuracy: 0.93
Epoch:  0 | train loss: 0.0439 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1979 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0518 | test accuracy: 0.95
Epoch:  0 | train loss: 0.1723 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1908 | test accuracy: 0.94
Epoch:  0 | train loss: 0.0576 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0414 | test accuracy: 0.96
Epoch:  0 | train loss: 0.3591 | test accuracy: 0.95
Epoch:  0 | train loss: 0.2465 | test accuracy: 0.95
```

Once the model is trained, then the next step is to make predictions using the RNN model. Then we compare the actual vs. real output to assess how the model is performing.

```
In [86]: # print 10 predictions from test data
         test_output = rnn(test_x[:10].view(-1, 28, 28))
         pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
         print(pred_y, 'prediction number')
         print(test_y[:10], 'real number')

         [7 2 1 0 4 1 4 9 5 9] prediction number
         [7 2 1 0 4 1 4 9 5 9] real number
```

Recipe 3-9. Implementing a RNN for Regression Problems

Problem

How do we set up a recurrent neural network for regression-based problems?

Solution

The regression model requires a target function and a feature set, and then a function to establish the relationship between the input and the output. In this example, we are going to use the recurrent neural network (RNN) for a regression task. Regression problems seem to be very simple; they do work best but are limited to data that shows clear linear relationships. They are quite complex when predicting nonlinear relationships between the input and the output.

How It Works

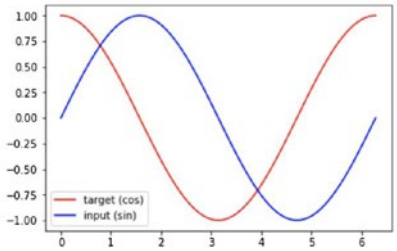
Let's look at the following example that shows a nonlinear cyclical pattern between input and output data. In the previous recipe, we looked at an example of RNN in general for classification-related problems, where predicted the class of the input image. In regression, however, the architecture of RNN would change, because the objective is to predict the real valued output. The output layer would have one neuron in regression-related problems.

```
In [88]: import torch
         from torch import nn
         from torch.autograd import Variable
         import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline

In [89]: torch.manual_seed(1)    # reproducible

Out[89]: <torch._C.Generator at 0x1156e82f0>

In [90]: # Hyper Parameters
         TIME_STEP = 10      # rnn time step
         INPUT_SIZE = 1      # rnn input size
         LR = 0.02           # learning rate
```

RNN time step implies that the last 10 values predict the current value, and the rolling happens after that.

The following script shows some sample series in which the target cos function is approximated by the sin function.

```
In [91]: # show data
         steps = np.linspace(0, np.pi*2, 100, dtype=np.float32)
         x_np = np.sin(steps)    # float32 for converting torch FloatTensor
         y_np = np.cos(steps)
         plt.plot(steps, y_np, 'r-', label='target (cos)')
         plt.plot(steps, x_np, 'b-', label='input (sin)')
         plt.legend(loc='best')
         plt.show()
```



Recipe 3-10. Using PyTorch Built-in Functions

Problem

How do we set up an RNN module and call the RNN function using PyTorch?

Solution

By using the built-in function available in the neural network module, we can implement an RNN model.

How It Works

Let's look at the following example. The neural network module in the PyTorch library contains the RNN function. In the following script, we use the input matrix size, the number of neurons in the hidden layer, and the number of hidden layers in the network.

```
In [92]: class RNN(nn.Module):
             def __init__(self):
                 super(RNN, self).__init__()

                 self.rnn = nn.RNN(
                     input_size=INPUT_SIZE,
                     hidden_size=32,     # rnn hidden unit
                     num_layers=1,       # number of rnn layer
                     batch_first=True,   # input & output will has batch size as 1s dimension. e.g.
                     #(batch, time_step, input_size)
                 )
                 self.out = nn.Linear(32, 1)

             def forward(self, x, h_state):
                 # x (batch, time_step, input_size)
                 # h_state (n_layers, batch, hidden_size)
                 # r_out (batch, time_step, hidden_size)
                 r_out, h_state = self.rnn(x, h_state)

                 outs = []    # save all predictions
                 for time_step in range(r_out.size(1)):    # calculate output for each time step
                     outs.append(self.out(r_out[:, time_step, :]))
                 return torch.stack(outs, dim=1), h_state
```

After creating the RNN class function, we need to provide the optimization function, which is Adam, and this time, the loss function is the mean square loss function. Since the objective is to make predictions of a continuous variable, we use MSELoss function in the optimization layer.

```
In [93]: rnn = RNN()
         print(rnn)
```

```
RNN(
  (rnn): RNN(1, 32, batch_first=True)
  (out): Linear(in_features=32, out_features=1, bias=True)
)
```

```
In [94]: optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)   # optimize all cnn parameters
         loss_func = nn.MSELoss()
```

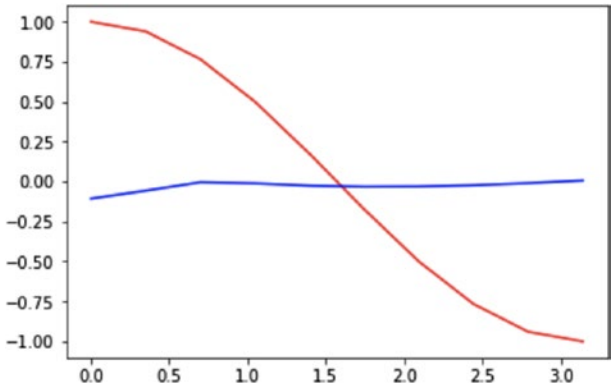
```
In [95]: h_state = None      # for initial hidden state
```

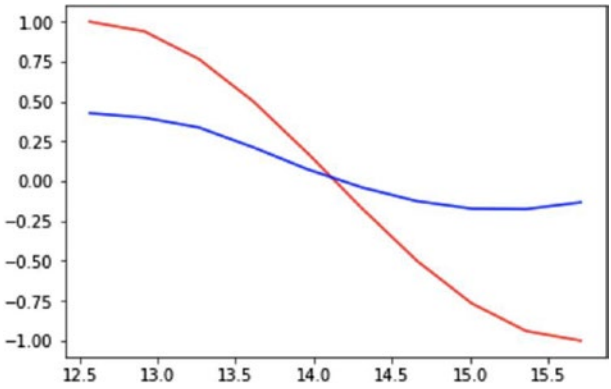
```
In [96]: plt.figure(1, figsize=(12, 5))
         plt.ion()           # continuously plot
```

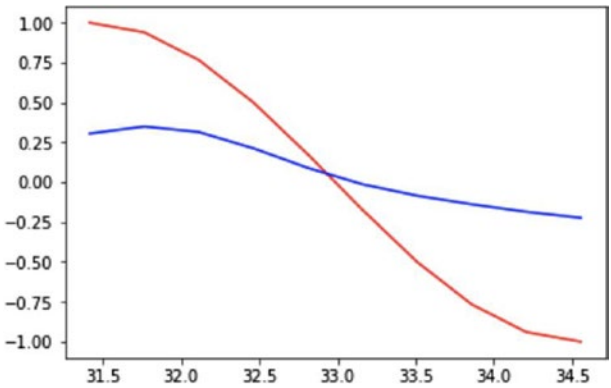
```
<matplotlib.figure.Figure at 0x121b59a58>
```

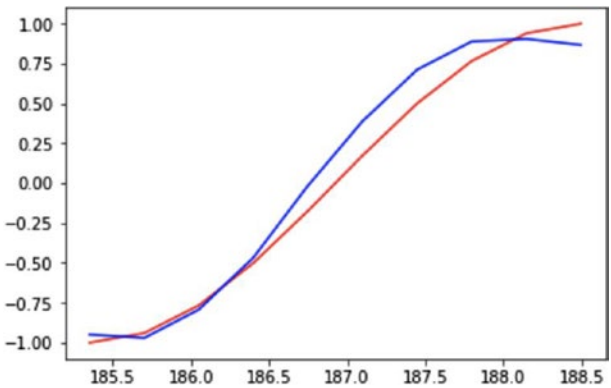
```
In [97]: for step in range(60):
             start, end = step * np.pi, (step+1)*np.pi   # time range
             # use sin predicts cos
             steps = np.linspace(start, end, TIME_STEP, dtype=np.float32)
             x_np = np.sin(steps)    # float32 for converting torch FloatTensor
             y_np = np.cos(steps)

             x = Variable(torch.from_numpy(x_np[np.newaxis, :, np.newaxis]))
             # shape (batch, time_step, input_size)
             y = Variable(torch.from_numpy(y_np[np.newaxis, :, np.newaxis]))

             prediction, h_state = rnn(x, h_state)   # rnn output
             # !! next step is important !!
             h_state = Variable(h_state.data)
             # repack the hidden state, break the connection from last iteration

             loss = loss_func(prediction, y)         # cross entropy loss
             optimizer.zero_grad()                   # clear gradients for this training step
             loss.backward()                         # backpropagation, compute gradients
             optimizer.step()                        # apply gradients

             # plotting
             plt.plot(steps, y_np.flatten(), 'r-')
             plt.plot(steps, prediction.data.numpy().flatten(), 'b-')
             plt.draw(); plt.pause(0.05)
```

Now we iterate over 60 steps to predict the cos function generated from the sample space, and have it predicted by a sin function. The iterations take the learning rate defined as before, and backpropagate the error to reduce the MSE and improve the prediction.




1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
12.5
13.0
13.5
14.0
14.5
15.0
15.5


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
31.5
32.0
32.5
33.0
33.5
34.0
34.5


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
185.5
186.0
186.5
187.0
187.5
188.0
188.5

Recipe 3-11. Working with Autoencoders

Problem

How do we perform clustering using the autoencoders function?

Solution

Unsupervised learning is a branch of machine learning that does not have a target column or the output is not defined. We only need to understand the unique patterns existing in the data. Let's look at the autoencoder architecture in Figure 3-1. The input feature space is transformed into a lower dimensional tensor representation using a hidden layer and mapped back to the same input space. The layer that is precisely in the middle holds the autoencoder's values.

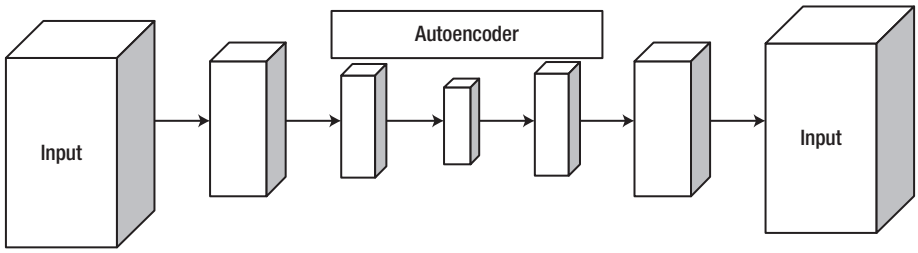


Figure 3-1. *Autoencoder architecture*

How It Works

Let's look at the following example. The torchvision library contains popular datasets, model architectures, and frameworks. Autoencoder is a process of identifying latent features from the dataset; it is used for classification, prediction, and clustering. If we put the input data in the input layer and the same dataset in the output layer, then we add multiple

layers of hidden layers with many neurons, and then we pass through a series of epochs. We get a set of latent features in the innermost hidden layer. The weights or parameters in the central hidden layer are known as the *autoencoder layer*.

```
In [99]: import torch
         import torch.nn as nn
         from torch.autograd import Variable
         import torch.utils.data as Data
         import torchvision
         import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D
         from matplotlib import cm
         import numpy as np
         %matplotlib inline

In [100]: torch.manual_seed(1)    # reproducible

Out[100]: <torch._C.Generator at 0x1156e82f0>

In [101]: # Hyper Parameters
          EPOCH = 10
          BATCH_SIZE = 64
          LR = 0.005          # learning rate
          DOWNLOAD_MNIST = False
          N_TEST_IMG = 5
```

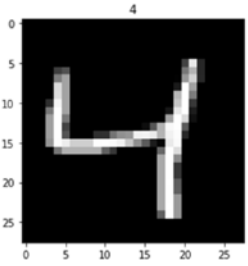
We again use the MNIST dataset to experiment with autoencoder functionality. This time we are taking 10 epochs, a batch size 64 to be passed to the network, a learning rate of 0.005, and 5 images for testing.

```
In [102]: # Mnist digits dataset
          train_data = torchvision.datasets.MNIST(
              root='./mnist/',
              train=True,
              # this is training data
              transform=torchvision.transforms.ToTensor(),
              # Converts a PIL.Image or numpy.ndarray to

              # torch.FloatTensor of shape (C x H x W) and normalize in the range [0.0, 1.0]
              download=DOWNLOAD_MNIST,
              # download it if you don't have it
          )
```

The following plot shows the dataset uploaded from the torchvision library and displayed as an image.

In [103]:
```
# plot one example
print(train_data.train_data.size())     # (60000, 28, 28)
print(train_data.train_labels.size())   # (60000)
plt.imshow(train_data.train_data[2].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[2])
plt.show()
```

```
torch.Size([60000, 28, 28])
torch.Size([60000])
```



In [104]:
```
# Data Loader for easy mini-batch return in training, the image batch shape will be (50, 1, 28,
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
```

In [105]:
```
class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.Tanh(),
            nn.Linear(128, 64),
            nn.Tanh(),
            nn.Linear(64, 12),
            nn.Tanh(),
            nn.Linear(12, 3),   # compress to 3 features which can be visualized in plt
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.Tanh(),
            nn.Linear(12, 64),
            nn.Tanh(),
            nn.Linear(64, 128),
            nn.Tanh(),
            nn.Linear(128, 28*28),
            nn.Sigmoid(),       # compress to a range (0, 1)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded
```

Let's discuss the autoencoder architecture. The input has 784 features. It has a height of 28 and a width of 28. We pass the 784 neurons from the input layer to the first hidden layer, which has 128 neurons in it. Then we apply the hyperbolic tangent function to pass the information to the next hidden layer. The second hidden layer contains 128 input neurons and transforms it into 64 neurons. In the third hidden layer, we apply the hyperbolic tangent function to pass the information to the next hidden layer. The innermost layer contains three neurons, which are considered as three features, which is the end of the encoder layer. Then the decoder function expands the layer back to the 784 features in the output layer.

```
In [106]: autoencoder = AutoEncoder()
          print(autoencoder)

          optimizer = torch.optim.Adam(autoencoder.parameters(), lr=LR)
          loss_func = nn.MSELoss()

          # original data (first row) for viewing
          view_data = Variable(train_data.train_data[:N_TEST_IMG].view(-1, 28*28).type(torch.FloatTensor)/

          AutoEncoder(
            (encoder): Sequential(
              (0): Linear(in_features=784, out_features=128, bias=True)
              (1): Tanh()
              (2): Linear(in_features=128, out_features=64, bias=True)
              (3): Tanh()
              (4): Linear(in_features=64, out_features=12, bias=True)
              (5): Tanh()
              (6): Linear(in_features=12, out_features=3, bias=True)
            )
            (decoder): Sequential(
              (0): Linear(in_features=3, out_features=12, bias=True)
              (1): Tanh()
              (2): Linear(in_features=12, out_features=64, bias=True)
              (3): Tanh()
              (4): Linear(in_features=64, out_features=128, bias=True)
              (5): Tanh()
              (6): Linear(in_features=128, out_features=784, bias=True)
              (7): Sigmoid()
            )
          )
```

Once we set the architecture, then the normal process of making the loss function minimize corresponding to a learning rate and optimization function happens. The entire architecture passes through a series of epochs in order to reach the target output.

Recipe 3-12. Fine-Tuning Results Using Autoencoder

Problem

How do we set up iterations to fine-tune the results?

Solution

Conceptually, an autoencoder works the same as the clustering model. In unsupervised learning, the machine learns patterns from data and generalizes it to the new dataset. The learning happens by taking a set of input features. Autoencoder functions are also used for feature engineering.

How It Works

Let's look at the following example. The same MNIST dataset is used as an example, and the objective is to understand the role of the epoch in achieving a better autoencoder layer. We increase the epoch size to reduce errors to a minimum; however, in practice, increasing the epoch has many challenges, including memory constraints.

In [111]:
```
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):
        b_x = Variable(x.view(-1, 28*28))   # batch x, shape (batch, 28*28)
        b_y = Variable(x.view(-1, 28*28))   # batch y, shape (batch, 28*28)
        b_label = Variable(y)               # batch label

        encoded, decoded = autoencoder(b_x)

        loss = loss_func(decoded, b_y)      # mean square error
        optimizer.zero_grad()               # clear gradients for this training step
        loss.backward()                     # backpropagation, compute gradients
        optimizer.step()                    # apply gradients

        if step % 500 == 0 and epoch in [0, 5, EPOCH-1]:
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data[0])

            # plotting decoded image (second row)
            _, decoded_data = autoencoder(view_data)

            # initialize figure
            f, a = plt.subplots(2, N_TEST_IMG, figsize=(5, 2))

            for i in range(N_TEST_IMG):
                a[0][i].imshow(np.reshape(view_data.data.numpy()[i],
                                          (28, 28)), cmap='gray');
                a[0][i].set_xticks(()); a[0][i].set_yticks(())

            for i in range(N_TEST_IMG):
                a[1][i].clear()
                a[1][i].imshow(np.reshape(decoded_data.data.numpy()[i],
                                          (28, 28)), cmap='gray')
                a[1][i].set_xticks(()); a[1][i].set_yticks(())
            plt.show(); #plt.pause(0.05)
```

```
Epoch:  0 | train loss: 0.2213
```

```
Epoch:  0 | train loss: 0.0678
```

```
Epoch:  5 | train loss: 0.0375
```

Epoch: 9 | train loss: 0.0378

Epoch: 9 | train loss: 0.0382

By using the encoder function, we can represent the input features into a set of latent features. By using the decoder function, however, we can reconstruct the image. Then we can match how image reconstruction is done by using the autoencoder functions. From the preceding set of graphs, it is clear that as we increase the epoch, the image recognition becomes transparent.

Recipe 3-13. Visualizing the Encoded Data in a 3D Plot

Problem

How do we visualize the MNIST data in a 3D plot?

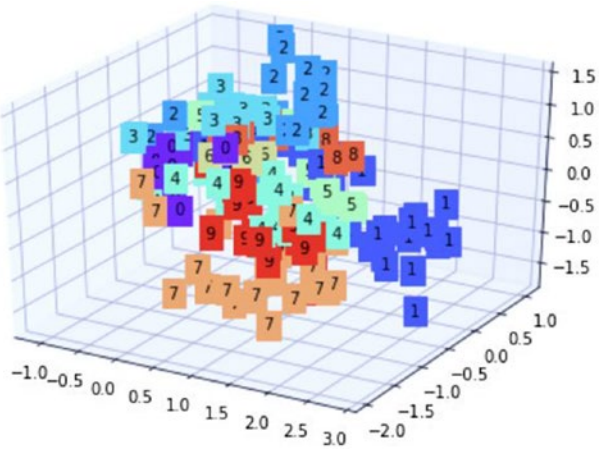
Solution

We use the autoencoder function to get the encoded features and then use the dataset to represent it in a 3D plane.

How It Works

Let's look at the following example. This recipe is about how to represent the autoencoder function derived from the preceding recipe in the three-dimensional space, because we have three neurons in the innermost hidden layer. The following display shows a three-dimensional neuron.

```
In [112]: # visualize in 3D plot
          view_data = Variable(train_data.train_data[:200].view(-1, 28*28).type(torch.FloatTensor)/255.)
          encoded_data, _ = autoencoder(view_data)
          fig = plt.figure(2); ax = Axes3D(fig)
          X, Y, Z = encoded_data.data[:, 0].numpy(), encoded_data.data[:, 1].numpy(),
          encoded_data.data[:, 2].numpy()
          values = train_data.train_labels[:200].numpy()
          for x, y, z, s in zip(X, Y, Z, values):
              c = cm.rainbow(int(255*s/9)); ax.text(x, y, z, s, backgroundcolor=c)
          ax.set_xlim(X.min(), X.max()); ax.set_ylim(Y.min(), Y.max()); ax.set_zlim(Z.min(), Z.max())
          plt.show()
```



Recipe 3-14. Restricting Model Overfitting

Problem

When we fit many neurons and layers to predict the target class or output variable, the function usually overfits the training dataset. Because of model overfitting, we cannot make a good prediction on the test set. The test accuracy is not the same as training accuracy. There would be deviations in training and test accuracy.

Solution

To restrict model overfitting, we consciously introduce dropout rate, which means randomly delete (let's say) 10% or 20% of the weights in the network, and check the model accuracy at the same time. If we are able to match the same model accuracy after deleting the 10% or 20% of the weights, then our model is good.

How It Works

Let's look at the following example. Model overfitting is occurs when the trained model does not generalize to other test case scenarios. It is identified when the training accuracy becomes significantly different from the test accuracy. To avoid model overfitting, we can introduce the dropout rate in the model.

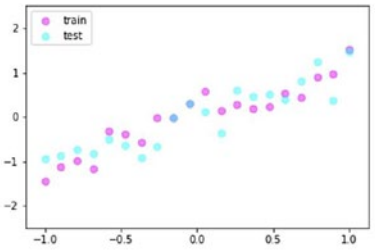
```
In [121]: import torch
          from torch.autograd import Variable
          import matplotlib.pyplot as plt
          %matplotlib inline

          torch.manual_seed(1)    # reproducible

Out[121]: <torch._C.Generator at 0x1156e82f0>
```

```
In [122]: N_SAMPLES = 20
          N_HIDDEN = 300
```

```
In [123]: # training data
          x = torch.unsqueeze(torch.linspace(-1, 1, N_SAMPLES), 1)
          y = x + 0.3*torch.normal(torch.zeros(N_SAMPLES, 1), torch.ones(N_SAMPLES, 1))
          x, y = Variable(x), Variable(y)
```

```
In [124]: # test data
          test_x = torch.unsqueeze(torch.linspace(-1, 1, N_SAMPLES), 1)
          test_y = test_x + 0.3*torch.normal(torch.zeros(N_SAMPLES, 1), torch.ones(N_SAMPLES, 1))
          test_x, test_y = Variable(test_x, volatile=True), Variable(test_y, volatile=True)
```

```
In [125]: # show data
          plt.scatter(x.data.numpy(), y.data.numpy(), c='magenta', s=50, alpha=0.5, label='train')
          plt.scatter(test_x.data.numpy(), test_y.data.numpy(), c='cyan', s=50, alpha=0.5, label='test')
          plt.legend(loc='upper left')
          plt.ylim((-2.5, 2.5))
          plt.show()
```




```
In [126]: net_overfitting = torch.nn.Sequential(
              torch.nn.Linear(1, N_HIDDEN),
              torch.nn.ReLU(),
              torch.nn.Linear(N_HIDDEN, N_HIDDEN),
              torch.nn.ReLU(),
              torch.nn.Linear(N_HIDDEN, 1),
          )
```

The dropout rate introduction to the hidden layer ensures that weights less than the threshold defined are removed from the architecture. A typical threshold for an application's dropout rate is 20% to 50%. A 20% dropout rate implies a smaller degree of penalization; however, the 50% threshold implies heavy penalization of the model weights.

In the following script, we apply a 50% dropout rate to drop the weights from the model. We applied the dropout rate twice.

```
In [127]: net_dropped = torch.nn.Sequential(
              torch.nn.Linear(1, N_HIDDEN),
              torch.nn.Dropout(0.5),  # drop 50% of the neuron
              torch.nn.ReLU(),
              torch.nn.Linear(N_HIDDEN, N_HIDDEN),
              torch.nn.Dropout(0.5),  # drop 50% of the neuron
              torch.nn.ReLU(),
              torch.nn.Linear(N_HIDDEN, 1),
          )
```

```
In [128]: print(net_overfitting)  # net architecture
          print(net_dropped)

          Sequential(
            (0): Linear(in_features=1, out_features=300, bias=True)
            (1): ReLU()
            (2): Linear(in_features=300, out_features=300, bias=True)
            (3): ReLU()
            (4): Linear(in_features=300, out_features=1, bias=True)
          )
          Sequential(
            (0): Linear(in_features=1, out_features=300, bias=True)
            (1): Dropout(p=0.5)
            (2): ReLU()
            (3): Linear(in_features=300, out_features=300, bias=True)
            (4): Dropout(p=0.5)
            (5): ReLU()
            (6): Linear(in_features=300, out_features=1, bias=True)
          )
```

```
In [129]: optimizer_ofit = torch.optim.Adam(net_overfitting.parameters(), lr=0.01)
          optimizer_drop = torch.optim.Adam(net_dropped.parameters(), lr=0.01)
          loss_func = torch.nn.MSELoss()
```

The selection of right dropout rate requires a fair idea about the business and domain.

Recipe 3-15. Visualizing the Model Overfit

Problem

Assess model overfitting.

Solution

We change the model hyperparameters and iteratively see if the model is overfitting data or not.

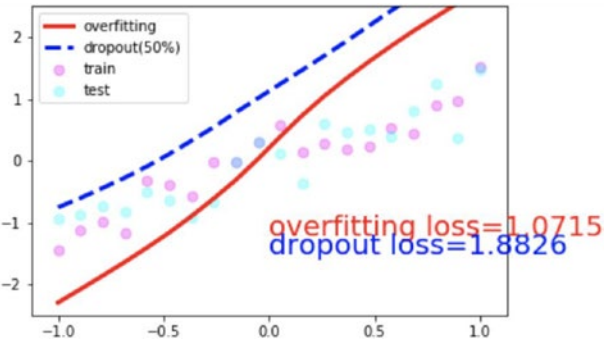
How It Works

Let's look at the following example. The previous recipe covered two types of neural networks: overfitting and dropout rate. When the model parameters estimated from the data come closer to the actual data, for the training dataset and the same models differs from the test set, it is a clear sign of model overfit. To restrict model overfit, we can introduce the dropout rate, which deletes a certain percentage of connections (as in weights from the network) to allow the trained model to come to the real data.

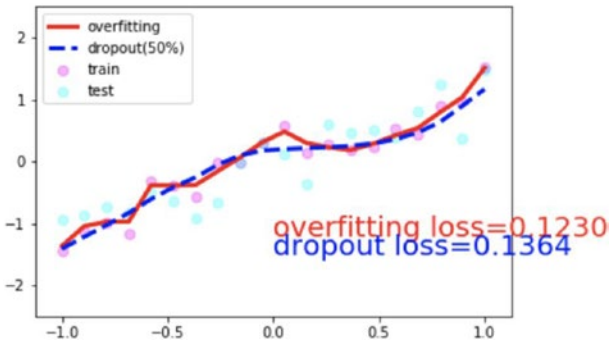
In the following script, the iterations were taken 500 times. The predicted values are generated from the base model, which shows overfitting, and from the dropout model, which shows the deletion of some weights. In the same fashion, we create the two loss functions, backpropagation, and implementation of the optimizer.

In [130]:
```
for t in range(500):
    pred_ofit = net_overfitting(x)
    pred_drop = net_dropped(x)
    loss_ofit = loss_func(pred_ofit, y)
    loss_drop = loss_func(pred_drop, y)

    optimizer_ofit.zero_grad()
    optimizer_drop.zero_grad()
    loss_ofit.backward()
    loss_drop.backward()
    optimizer_ofit.step()
    optimizer_drop.step()

    if t % 100 == 0:
        # change to eval mode in order to fix drop out effect
        net_overfitting.eval()
        net_dropped.eval()  # parameters for dropout differ from train mode

        # plotting
        plt.cla()
        test_pred_ofit = net_overfitting(test_x)
        test_pred_drop = net_dropped(test_x)
        plt.scatter(x.data.numpy(), y.data.numpy(), c='magenta', s=50,
                    alpha=0.3, label='train')
        plt.scatter(test_x.data.numpy(), test_y.data.numpy(), c='cyan', s=50,
                    alpha=0.3, label='test')
        plt.plot(test_x.data.numpy(), test_pred_ofit.data.numpy(), 'r-',
                 lw=3, label='overfitting')
        plt.plot(test_x.data.numpy(), test_pred_drop.data.numpy(), 'b--',
                 lw=3, label='dropout(50%)')
        plt.text(0, -1.2, 'overfitting loss=%.4f' % loss_func(test_pred_ofit, test_y).data[0],
                 fontdict={'size': 20, 'color':  'red'})
        plt.text(0, -1.5, 'dropout loss=%.4f' % loss_func(test_pred_drop, test_y).data[0],
                 fontdict={'size': 20, 'color': 'blue'})
        plt.legend(loc='upper left'); plt.ylim((-2.5, 2.5));plt.pause(0.1)
```

```
# change back to train mode
net_overfitting.train()
net_dropped.train()
plt.show()
```



The initial round of plotting includes the overfitting loss and dropout loss and how it is different from the actual training and test data points from the preceding graph.



After many iterations, the preceding graph was generated by using the two functions with the actual model and with the dropout rate. The takeaway from this graph is that actual training data may get closer to the overfit model; however, the dropout model fits the data really well.

Recipe 3-16. Initializing Weights in the Dropout Rate

Problem

How do we delete the weights in a network? Should we delete randomly or by using any distribution?

Solution

We should delete the weights in the dropout layer based on probability distribution, rather than randomly.

How It Works

Let's look at the following example. In the previous recipe, three layers of a dropout rate were introduced: one after the first hidden layer and two after the second hidden layer. The probability percentage was 0.50, which meant randomly delete 50% of the weights. Sometimes, random selection of weights from the network deletes relevant weights, so an alternative idea is to delete the weights in the network generated from statistical distribution.

The following script shows how to generate the weights from a uniform distribution, then we can use the set of weights in the network architecture.

```
In [132]: import numpy as np
          from __future__ import print_function

In [133]: import torch

In [134]: torch.Tensor(5, 3)

Out[134]: tensor([[ 0.0000, -0.0000,  0.0927],
                  [ 0.0000,  0.0000,  0.0000],
                  [ 0.0000,  0.0000,  0.0000],
                  [ 0.0000,  0.0000,  0.0000],
                  [ 0.0000, -0.0000,  0.0925]])

In [135]: #From a uniform distribution

In [136]: torch.Tensor(5, 3).uniform_(-1, 1)

Out[136]: tensor([[ 0.8790,  0.7375,  0.1182],
                  [ 0.3652,  0.1322,  0.8587],
                  [ 0.3682, -0.2907,  0.0051],
                  [ 0.0886, -0.7588, -0.5371],
                  [ 0.0085,  0.0812, -0.6360]])


In [138]: x = torch.Tensor(5, 3).uniform_(-1, 1)
          print(x.size())

          torch.Size([5, 3])

In [139]: #Creation from lists & numpy

In [140]: z = torch.LongTensor([[1, 3], [2, 9]])
          print(z.type())
          # Cast to numpy ndarray
          print(z.numpy().dtype)

          torch.LongTensor
          int64

In [141]: # Data type inferred from numpy
          print(torch.from_numpy(np.random.rand(5, 3)).type())
          print(torch.from_numpy(np.random.rand(5, 3).astype(np.float32)).type())

          torch.DoubleTensor
          torch.FloatTensor
```

Recipe 3-17. Adding Math Operations

Problem

How do we set up the broadcasting function and optimize the convolution function?

Solution

The script snippet shows how to introduce batch normalization when setting up a convolutional neural network model, and then further setting up a pooling layer.

How It Works

Let's look at the following example. To introduce batch normalization in the convolutional layer of the neural network model, we need to perform tensor-based mathematical operations that are functionally different from other methods of computation.

```
In [142]: #Simple mathematical operations

In [143]: y = x * torch.randn(5, 3)
          print(y)

          tensor([[ 0.1587,  0.4137, -0.4801],
                  [-0.2706,  0.0411, -0.8954],
                  [ 0.3616, -0.0245, -0.3401],
                  [-0.6478, -0.1207, -0.1698],
                  [ 0.2107, -0.2128,  0.1017]])

In [144]: y = x / torch.sqrt(torch.randn(5, 3) ** 2)
          print(y)

          tensor([[ 2.1697, -1.1561, -7.4875],
                  [-0.5094,  0.4193, -4.4016],
                  [ 0.4308,  0.0421,  0.6234],
                  [ 2.3634,  2.1020, -0.2185],
                  [ 4.8023,  0.4352,  0.4892]])
```

In [145]:
```
#Broadcasting
```

In [146]:
```
print (x.size())
y = x + torch.randn(5, 1)
print(y)
```

```
torch.Size([5, 3])
tensor([[ 1.0416, -0.1192, -0.1256],
        [ 0.0484,  1.5687,  0.0468],
        [ 0.1000, -0.4971,  0.3657],
        [ 0.3893,  0.5367, -0.2656],
        [ 2.1538,  1.9121,  2.0349]])
```

In [147]:
```
#Reshape
```

In [148]:
```
y = torch.randn(5, 10, 15)
print(y.size())
print(y.view(-1, 15).size())  # Same as doing y.view(50, 15)
print(y.view(-1, 15).unsqueeze(1).size()) # Adds a dimension at index 1.
print(y.view(-1, 15).unsqueeze(1).squeeze().size())
print()
print(y.transpose(0, 1).size())
print(y.transpose(1, 2).size())
print(y.transpose(0, 1).transpose(1, 2).size())
print(y.permute(1, 2, 0).size())
```

```
torch.Size([5, 10, 15])
torch.Size([50, 15])
torch.Size([50, 1, 15])
torch.Size([50, 15])

torch.Size([10, 5, 15])
torch.Size([5, 15, 10])
torch.Size([10, 15, 5])
torch.Size([10, 15, 5])
```

In [149]:
```
#Repeat
```

In [150]:
```
print(y.view(-1, 15).unsqueeze(1).expand(50, 100, 15).size())
print(y.view(-1, 15).unsqueeze(1).expand_as(torch.randn(50, 100, 15)).size())
```

```
torch.Size([50, 100, 15])
torch.Size([50, 100, 15])
```

In [151]:
```
#Concatenate tensors
```

In [152]:
```
# 2 is the dimension over which the tensors are concatenated
print(torch.cat([y, y], 2).size())
# stack concatenates the sequence of tensors along a new dimension.
print(torch.stack([y, y], 0).size())
```

```
torch.Size([5, 10, 30])
torch.Size([2, 5, 10, 15])
```

In [153]:
```
#Advanced Indexing
```

In [154]:
```
y = torch.randn(2, 3, 4)
print(y[[1, 0, 1, 1]].size())

# PyTorch doesn't support negative strides yet so ::-1 does not work.
rev_idx = torch.arange(1, -1, -1).long()
print(y[rev_idx].size())
```

```
torch.Size([4, 3, 4])
torch.Size([2, 3, 4])
```

The following piece of script shows how the batch normalization using a 2D layer is resolved before entering into the 2D max pooling layer.

```
In [157]: #Convolution, BatchNorm & Pooling Layers

In [158]: x = Variable(torch.randn(10, 3, 28, 28))

          conv = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3, 3), stride=1,
                           padding=1, bias=True)
          bn = nn.BatchNorm2d(num_features=32)
          pool = nn.MaxPool2d(kernel_size=(2, 2), stride=2)

          output_conv = bn(conv(x))
          outpout_pool = pool(conv(x))

          print('Conv output size : ', output_conv.size())
          print('Pool output size : ', outpout_pool.size())

          Conv output size :  torch.Size([10, 32, 28, 28])
          Pool output size :  torch.Size([10, 32, 14, 14])
```

Recipe 3-18. Embedding Layers in RNN

Problem

The recurrent neural network is used mostly for text processing. An embedded feature offers more accuracy on a standard RNN model than raw features. How do we create embedded features in an RNN?

Solution

The first step is to create an embedding layer, which is a fixed dictionary and fixed-size lookup table, and then introduce the dropout rate after than create gated recurrent unit.

How It Works

Let's look at the following example. When textual data comes in as a sequence, the information is processed in a sequential way; for example, when we describe something, we use a set of words in sequence to convey the meaning. If we use the individual words as vectors to represent the data, the resulting dataset would be very sparse. But if we use a phrase-based approach or a combination of words to represent as feature vector,

then the vectors become a dense layer. Dense vector layers are called *word embeddings*, as the embedding layer conveys a context or meaning as the result. It is definitely better than the bag-of-words approach.

```
In [159]: #Recurrent, Embedding & Dropout Layers

In [160]: inputs = [[1, 2, 3], [1, 0, 4], [1, 2, 4], [1, 4, 0], [1, 3, 3]]
          x = Variable(torch.LongTensor(inputs))

          embedding = nn.Embedding(num_embeddings=5, embedding_dim=20, padding_idx=1)
          drop = nn.Dropout(p=0.5)
          gru = nn.GRU(input_size=20, hidden_size=50, num_layers=2, batch_first=True,
                       bidirectional=True, dropout=0.3)

          emb = drop(embedding(x))
          gru_h, gru_h_t = gru(emb)

          print('Embedding size : ', emb.size())
          print('GRU hidden states size : ', gru_h.size())
          print('GRU last hidden state size : ', gru_h_t.size())

          Embedding size :  torch.Size([5, 3, 20])
          GRU hidden states size :  torch.Size([5, 3, 100])
          GRU last hidden state size :  torch.Size([4, 5, 50])

In [161]: #The functional API provides users a way to use these classes in a functional way.

In [162]: x = Variable(torch.randn(10, 3, 28, 28))
          filters = Variable(torch.randn(32, 3, 3, 3))
          conv_out = F.relu(F.dropout(F.conv2d(input=x, weight=filters, padding=1),
                                      p=0.5, training=True))

          print('Conv output size : ', conv_out.size())

          Conv output size :  torch.Size([10, 32, 28, 28])
```

Conclusion

This chapter covered using the PyTorch API, creating a simple neural network mode, and optimizing the parameters by changing the hyperparameters (i.e., learning rate, epochs, gradients drop). We looked at recipes on how to create a convolutional neural network and a recurrent neural network, and introduced the dropout rate in these networks to control model overfitting.

We took small tensors to follow what exactly goes on behind the scenes with calculations and so forth. We only need to define the problem statement, create features, and apply the recipe to get results. In the next chapter, we implement many more examples with PyTorch.

CHAPTER 4

Introduction to Neural Networks Using PyTorch

Deep neural network–based models are gradually becoming the backbone for artificial intelligence and machine learning implementations. The future of data mining will be governed by the usage of artificial neural network–based advanced modeling techniques. One obvious question is why neural networks are only now gaining so much importance, because it was invented in 1950s.

Borrowed from the computer science domain, neural networks can be defined as a parallel information processing system where all the input relates to each other, like neurons in the human brain, to transmit information so that activities like face recognition, image recognition, and so forth, can be performed. In this chapter, you learn about the application of neural network-based methods on various data mining tasks, such as classification, regression, forecasting, and feature reduction. An *artificial neural network* (ANN) functions in a way that is similar to the way that the human brain functions, in which billions of neurons link to each other for information processing and insight generation.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_4

Recipe 4-1. Working with Activation Functions

Problem

What are the activation functions and how do they work in real projects? How do you implement an activation function using PyTorch?

Solution

Activation function is a mathematical formula that transforms a vector available in a binary, float, or integer format to another format based on the type of mathematical transformation function. The neurons are present in different layers—input, hidden, and output, which are interconnected through a mathematical function called an *activation function*. There are different variants of activation functions, which are explained next. Understanding the activation function helps in accurately implementing a neural network model.

How It Works

All the activation functions that are part of a neural network model can be broadly classified as linear functions and nonlinear functions. The PyTorch torch.nn module creates any type of a neural network model. Let's look at some examples of the deployment of activation functions using PyTorch and the torch.nn module.

The core differences between PyTorch and TensorFlow is the way a computational graph is defined, the way the two frameworks perform calculations, and the amount of flexibility we have in changing the script and introducing other Python-based libraries in it. In TensorFlow, we need to define the variables and placeholders before we initialize the model.

We also need to keep track of objects that we need later, and for that we need a placeholder. In TensorFlow, we need to define the model first, and then compile and run; however, in PyTorch, we can define the model as we go—we don't have to keep placeholders in the code. That's why the PyTorch framework is dynamic.

Linear Function

A linear function is a simple functions typically used to transfer information from the demapping layer to the output layer. We use the linear function in places where variations in data are lower. In a deep learning model, practitioners typically use a linear function in the last hidden layer to the output layer. In the linear function, the output is always confined to a specific range; because of that, it is used in the last hidden layer in a deep learning model, or in linear regression–based tasks, or in a deep learning model where the task is to predict the outcome from the input dataset. The following is the formula.

$$y = \alpha + \beta x$$

Bilinear Function

A bilinear function is a simple functions typically used to transfer information. It applies a bilinear transformation to incoming data.

$$y = x_1 * A * x_2 + b$$

```
In [2]: import numpy as np
        from __future__ import print_function
        import torch
        import torch.nn as nn
        import torch.optim as optim
        import torch.nn.init as init
        import torch.nn.functional as F
        from torch.autograd import Variable
```

```
In [3]: #torch.nn: - Neural networks can be constructed using the torch.nn package.
```

```
In [7]: x = Variable(torch.randn(100, 10))
        y = Variable(torch.randn(100, 30))

        linear = nn.Linear(in_features=10, out_features=5, bias=True)
        output_linear = linear(x)
        print('Output size : ', output_linear.size())

        bilinear = nn.Bilinear(in1_features=10, in2_features=30, out_features=5, bias=Tru
        output_bilinear = bilinear(x, y)
        print('Output size : ', output_bilinear.size())

        Output size :  torch.Size([100, 5])
        Output size :  torch.Size([100, 5])
```

Sigmoid Function

A sigmoid function is frequently used by professionals in data mining and analytics because it is easier to explain and implement. It is a nonlinear function. When we pass weights from the input layer to the hidden layer in a neural network, we want our model to capture all sorts of nonlinearity present in the data; hence, using the sigmoid function in the hidden layers of a neural network is recommended. The nonlinear functions help with generalizing the dataset. It is easier to compute the gradient of a function using a nonlinear function.

The sigmoid function is a specific nonlinear activation function. The sigmoid function output is always confined within 0 and 1; therefore, it is mostly used in performing classification-based tasks. One of the limitations of the sigmoid function is that it may get stuck in local minima. An advantage is that it provides probability of belonging to the class. The following is its equation.

$$f(x) = \frac{1}{1 + e^{-\beta x}}$$

```
In [15]: x = Variable(torch.randn(100, 10))
         y = Variable(torch.randn(100, 30))

         sig = nn.Sigmoid()
         output_sig = sig(x)
         output_sigy = sig(y)
         print('Output size : ', output_sig.size())
         print('Output size : ', output_sigy.size())

         Output size :  torch.Size([100, 10])
         Output size :  torch.Size([100, 30])
```

```
In [23]: print(x[0])
         print(output_sig[0])

         tensor([-0.2906, -1.6351,  1.3991,  0.2325,  1.3212, -0.4533, -0.6505, -1.0113,
                 -0.5113,  0.3498])
         tensor([0.1512, 0.6219, 0.7841, 0.5964, 0.6643, 0.3863, 0.5444, 0.6584, 0.6335,
                 0.6603])
```

Hyperbolic Tangent Function

A hyperbolic tangent function is another variant of a transformation function. It is used to transform information from the mapping layer to the hidden layer. It is typically used between the hidden layers of a neural network model. The range of the tanh function is between –1 and +1.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```
In [25]: x = Variable(torch.randn(100, 10))
         y = Variable(torch.randn(100, 30))

         func = nn.Tanh()
         output_x = func(x)
         output_y = func(y)
         print('Output size : ', output_x.size())
         print('Output size : ', output_y.size())

         Output size :  torch.Size([100, 10])
         Output size :  torch.Size([100, 30])
```

```
In [26]: print(x[0])
         print(output_x[0])
         print(y[0])
         print(output_y[0])
```

```
tensor([-0.4240, -0.0526,  0.6183, -1.5435,  0.6646, -0.6500, -0.4578, -0.3698,
        -1.1450, -0.0863])
tensor([-0.4003, -0.0525,  0.5500, -0.9127,  0.5814, -0.5716, -0.4283, -0.3538,
        -0.8161, -0.0861])
tensor([-0.8031, -0.6013, -0.3921, -0.7244, -1.8933,  0.4067,  0.0127,  1.0005,
         0.0717, -0.0426,  1.2915, -1.0941,  0.1721, -1.6982,  1.4946, -0.3869,
         0.1720,  0.2120,  1.1270,  0.6129,  0.6682,  0.7401, -0.5784, -1.9474,
         0.2925, -0.1313, -0.1723, -0.8419, -1.5402,  1.4225])
tensor([-0.6658, -0.5380, -0.3732, -0.6196, -0.9557,  0.3856,  0.0127,  0.7618,
         0.0716, -0.0426,  0.8595, -0.7984,  0.1704, -0.9352,  0.9042, -0.3687,
         0.1703,  0.2089,  0.8100,  0.5461,  0.5838,  0.6292, -0.5215, -0.9601,
         0.2844, -0.1305, -0.1706, -0.6868, -0.9122,  0.8901])
```

Log Sigmoid Transfer Function

The following formula explains the log sigmoid transfer function, which is used in mapping the input layer to the hidden layer. If the data is not binary, and it is a float type with a lot of outliers (as in large numeric values present in the input feature), then we should use the log sigmoid transfer function.

$$f(x)=\log\left(\frac{1}{1+e^{-\beta x}}\right)$$

```
In [27]: x = Variable(torch.randn(100, 10))
         y = Variable(torch.randn(100, 30))

         func = nn.LogSigmoid()
         output_x = func(x)
         output_y = func(y)
         print('Output size : ', output_x.size())
         print('Output size : ', output_y.size())
```

```
Output size :  torch.Size([100, 10])
Output size :  torch.Size([100, 30])
```

```
In [28]: print(x[0])
         print(output_x[0])
         print(y[0])
         print(output_y[0])
```

```
tensor([-1.2316, -1.3268,  1.9000,  1.7787,  0.1831, -1.0017,  0.4925, -1.0220,
        -0.4205, -0.9975])
tensor([-1.4877, -1.5621, -0.1394, -0.1560, -0.6058, -1.3145, -0.4769, -1.3294,
        -0.9253, -1.3114])
tensor([-0.4882, -2.7058, -1.0450,  0.7822,  0.3581,  0.5182, -0.5690,  0.4783,
        -0.8853, -1.3757, -0.4806, -0.6006, -0.5823, -0.6117,  0.9460, -0.1890,
         0.3575, -0.1234,  0.8133, -0.4657,  0.0915,  1.8386,  1.4911,  0.8194,
         0.3151,  0.7654,  0.8646,  0.6649, -0.5962, -0.0363])
tensor([-0.9668, -2.7705, -1.3464, -0.3766, -0.5300, -0.4672, -1.0176, -0.4823,
        -1.2307, -1.6010, -0.9620, -1.0379, -1.0261, -1.0451, -0.3281, -0.7921,
        -0.5303, -0.7567, -0.3670, -0.9528, -0.6484, -0.1476, -0.2030, -0.3651,
        -0.5480, -0.3820, -0.3515, -0.4150, -1.0350, -0.7115])
```

ReLU Function

The rectified linear unit (ReLu) is another activation function. It is used in transferring information from the input layer to the output layer. ReLu is mostly used in a convolutional neural network model. The range in which this activation function operates is from 0 to infinity. It is mostly used between different hidden layers in a neural network model.

```
In [29]: x = Variable(torch.randn(100, 10))
         y = Variable(torch.randn(100, 30))

         func = nn.ReLU()
         output_x = func(x)
         output_y = func(y)
         print('Output size : ', output_x.size())
         print('Output size : ', output_y.size())
```

```
Output size :  torch.Size([100, 10])
Output size :  torch.Size([100, 30])
```

```
In [30]: print(x[0])
         print(output_x[0])
         print(y[0])
         print(output_y[0])
```

```
tensor([-0.9385,  1.2685,  0.8244,  1.4173, -0.5307,  1.2229,  0.4709,  1.0213,
         1.5226,  1.1761])
tensor([0.0000, 1.2685, 0.8244, 1.4173, 0.0000, 1.2229, 0.4709, 1.0213, 1.5226,
        1.1761])
tensor([ 0.5548, -0.4403, -0.8275, -1.5821,  0.4721, -1.9475, -0.6546, -0.3003,
        -0.3691,  0.4592,  0.4883,  1.8626,  2.2027, -1.5628, -1.5948, -0.8585,
         0.4188,  1.0138, -1.1743,  0.7051,  0.7039, -0.8336, -1.3807, -0.4965,
        -0.6433, -0.8506,  0.1166,  0.2624,  1.8957, -1.6594])
tensor([0.5548, 0.0000, 0.0000, 0.0000, 0.4721, 0.0000, 0.0000, 0.0000, 0.0000,
        0.4592, 0.4883, 1.8626, 2.2027, 0.0000, 0.0000, 0.0000, 0.4188, 1.0138,
        0.0000, 0.7051, 0.7039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1166,
        0.2624, 1.8957, 0.0000])
```

The different types of transfer functions are interchangeable in a neural network architecture. They can be used in different stages, such as the input to the hidden layer, the hidden layer to the output layer, and so forth, to improve the model's accuracy.

Leaky ReLU

In a standard neural network model, a dying gradient problem is common. To avoid this issue, leaky ReLU is applied. Leaky ReLU allows a small and non-zero gradient when the unit is not active.

```
In [31]: x = Variable(torch.randn(100, 10))
         y = Variable(torch.randn(100, 30))

         func = nn.LeakyReLU()
         output_x = func(x)
         output_y = func(y)
         print('Output size : ', output_x.size())
         print('Output size : ', output_y.size())
```

```
Output size :  torch.Size([100, 10])
Output size :  torch.Size([100, 30])
```

```
In [32]: print(x[0])
         print(output_x[0])
         print(y[0])
         print(output_y[0])
```

```
tensor([ 0.3985,  1.4977, -1.1393, -0.1416,  0.2813, -0.6829,  1.5117, -0.2203,
        -0.8328,  0.9996])
tensor([ 0.3985,  1.4977, -0.0114, -0.0014,  0.2813, -0.0068,  1.5117, -0.0022,
        -0.0083,  0.9996])
tensor([ 1.2351, -0.1622,  0.5539,  0.2257, -1.1427,  1.3397,  0.3469, -0.9726,
        -0.7446,  0.1247,  0.4870,  0.0022,  0.1420,  0.6089, -0.6707, -1.0513,
        -2.0271, -1.0162, -0.4727, -1.3370, -0.7387, -0.3780, -0.7860,  0.7585,
        -0.2731,  1.2200, -1.3455,  1.0045, -1.3200,  1.6446])
tensor([ 1.2351, -0.0016,  0.5539,  0.2257, -0.0114,  1.3397,  0.3469, -0.0097,
        -0.0074,  0.1247,  0.4870,  0.0022,  0.1420,  0.6089, -0.0067, -0.0105,
        -0.0203, -0.0102, -0.0047, -0.0134, -0.0074, -0.0038, -0.0079,  0.7585,
        -0.0027,  1.2200, -0.0135,  1.0045, -0.0132,  1.6446])
```

Recipe 4-2. Visualizing the Shape of Activation Functions

Problem

How do we visualize the activation functions? The visualization of activation functions is important in correctly building a neural network model.

Solution

The activation functions translate the data from one layer into another layer. The transformed data can be plotted against the actual tensor to visualize the function. We have taken a sample tensor, converted it to a PyTorch variable, applied the function, and stored it as another tensor. Represent the actual tensor and the transformed tensor using matplotlib.

How It Works

The right choice of an activation function will not only provide better accuracy but also help with extracting meaningful information.

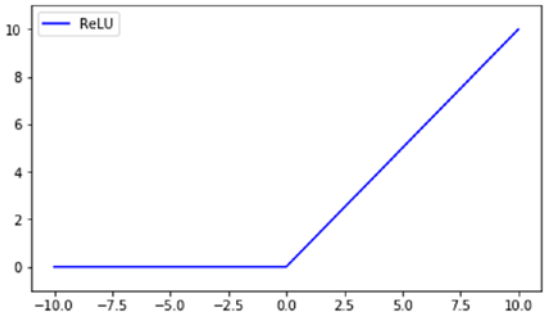
```
In [31]: import torch.nn.functional as F
         from torch.autograd import Variable
         import matplotlib.pyplot as plt
         %matplotlib inline
```

```
In [32]: x = torch.linspace(-10, 10, 1500)
         x = Variable(x)
         x_1 = x.data.numpy()   # tranforming into numpy
```

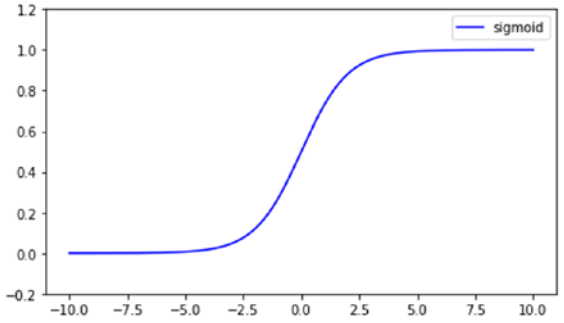
```
In [33]: y_relu = F.relu(x).data.numpy()
         y_sigmoid = torch.sigmoid(x).data.numpy()
         y_tanh = torch.tanh(x).data.numpy()
         y_softplus = F.softplus(x).data.numpy()
```

In this script, we have an array in the linear space between –10 and +10, and we have 1500 sample points. We converted the vector to a Torch variable, and then made a copy as a NumPy variable for plotting the graph. Then, we calculated the activation functions. The following images show the activation functions.

```
In [34]: plt.figure(figsize=(7, 4))
         plt.plot(x_1, y_relu, c='blue', label='ReLU')
         plt.ylim((-1, 11))
         plt.legend(loc='best')
Out[34]: <matplotlib.legend.Legend at 0x1d7b4a48fd0>
```

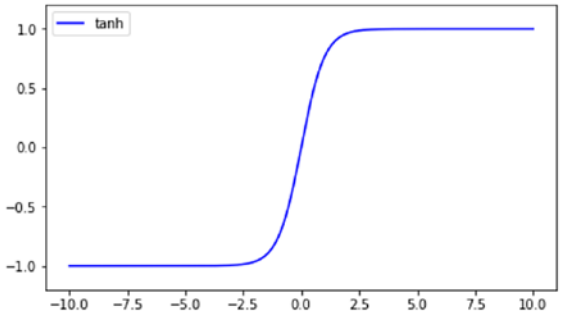



```
In [35]: plt.figure(figsize=(7, 4))
         plt.plot(x_1, y_sigmoid, c='blue', label='sigmoid')
         plt.ylim((-0.2, 1.2))
         plt.legend(loc='best')
Out[35]: <matplotlib.legend.Legend at 0x1d7b47f9780>
```



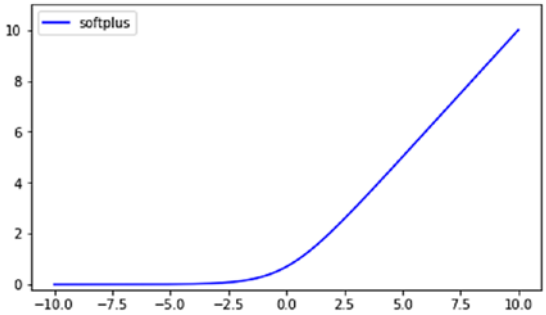
In [36]:
```
plt.figure(figsize=(7, 4))
plt.plot(x_1, y_tanh, c='blue', label='tanh')
plt.ylim((-1.2, 1.2))
plt.legend(loc='best')
```

Out[36]: <matplotlib.legend.Legend at 0x1d7b4b1eef0>



In [37]:
```
plt.figure(figsize=(7, 4))
plt.plot(x_1, y_softplus, c='blue', label='softplus')
plt.ylim((-0.2, 11))
plt.legend(loc='best')
```

Out[37]: <matplotlib.legend.Legend at 0x1d7b4b89ef0>



Recipe 4-3. Basic Neural Network Model

Problem

How do we build a basic neural network model using PyTorch?

Solution

A basic neural network model in PyTorch requires six steps: preparing training data, initializing weights, creating a basic network model, calculating the loss function, selecting the learning rate, and optimizing the loss function with respect to the model's parameters.

How It Works

Let's follow a step-by-step approach to create a basic neural network model.

```
In [185]: def prep_data():
              train_X = np.asarray([13.3,14.4,15.5,16.71,16.93,14.168,19.779,16.182,
                                    17.59,12.167,17.042,10.791,15.313,17.997,15.654,
                                    19.27,13.1])
              train_Y = np.asarray([11.7,12.76,12.09,13.19,11.694,11.573,13.366,12.596,
                                    12.53,11.221,12.827,13.465,11.65,12.904,12.42,12.94,
                                    11.3])
              dtype = torch.FloatTensor
              X = Variable(torch.from_numpy(train_X).type(dtype),
                           requires_grad=False).view(17,1)
              y = Variable(torch.from_numpy(train_Y).type(dtype),requires_grad=False)
              return X,y
```

To show a sample neural network model, we prepare the dataset and change the data type to a float tensor. When we work on a project, data preparation for building it is a separate activity. Data preparation should be done in the proper way. In the preceding step, train x and train y are two NumPy vectors. Next, we change the data type to a float tensor because it is necessary for matrix multiplication. The next step is to convert it to variable, because a variable has three properties that help us fine-tune the object. In the dataset, we have 17 data points on one dimension.

```
In [187]: def set_weights():
              w = Variable(torch.randn(1),requires_grad = True)
              b = Variable(torch.randn(1),requires_grad=True)
              return w,b

In [188]: #deploy neural network model

In [189]: def build_network(x):
              y_pred = torch.matmul(x,w)+b
              return y_pred

In [190]: #implement in PyTorch
          import torch.nn as nn
          f = nn.Linear(17,1) # Much simpler.
          f
Out[190]: Linear(in_features=17, out_features=1, bias=True)
```

The set_weight() function initializes the random weights that the neural network model will use in forward propagation. We need two tensors weights and biases. The build_network() function simply multiplies the weights with input, adds the bias to it, and generates the predicted values. This is a custom function that we built. If we need to implement the same thing in PyTorch, then it is much simpler to use nn.Linear() when we need to use it for linear regression.

```
In [210]: def loss_calc(y,y_pred):
              loss = (y_pred-y).pow(2).sum()
              for param in [w,b]:
                  if not param.grad is None: param.grad.data.zero_()
              loss.backward()
              return loss.data[0]

In [211]: # optimizing results

In [212]: def optimize(learning_rate):
              w.data -= learning_rate * w.grad.data
              b.data -= learning_rate * b.grad.data

In [208]: learning_rate = 1e-4

In [216]: x,y = prep_data()  # x - training data,y - target variables
          w,b = set_weights() # w,b - parameters
          for i in range(5000):
              y_pred = build_network(x) # function which computes wx + b
              loss = loss_calc(y,y_pred) # error calculation
              if i % 1000 == 0:
                  print(loss)
              optimize(learning_rate)    # minimize the loss w.r.t. w, b
```

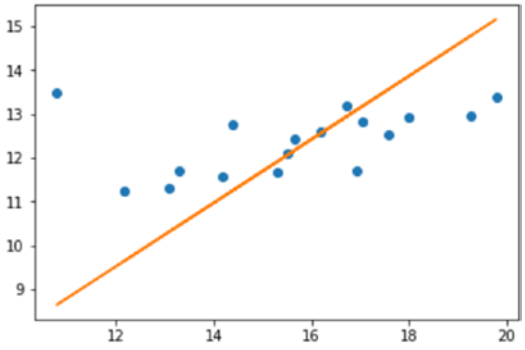
Once we define a network structure, then we need to compare the results with the output to assess the prediction step. The metric that tracks the accuracy of the system is the loss function, which we want to be minimal. The loss function may have a different shape. How do we know exactly where the loss is at a minimum, which corresponds to which iteration is providing the best results? To know this, we need to apply the optimization function on the loss function; it finds the minimum loss value. Then we can extract the parameters corresponding to that iteration.

```
tensor(573.6302)
tensor(38.7160)
tensor(34.2038)
tensor(30.3315)
tensor(27.0084)
```

```
In [197]: import matplotlib.pyplot as plt
          %matplotlib inline
```

```
In [198]: x_numpy = x.data.numpy()
          y_numpy = y.data.numpy()
          y_pred = y_pred.data.numpy()
          plt.plot(x_numpy,y_numpy,'o')
          plt.plot(x_numpy,y_pred,'-')
```

```
Out[198]: [<matplotlib.lines.Line2D at 0x1d7b4da98d0>]
```



Median, mode and standard deviation computation can be written in the sa

Standard deviation shows the deviation from the measures of central tendency, which indicates the consistency of the data/variable. It shows whether there is enough fluctuation in data or not.

Recipe 4-4. Tensor Differentiation

Problem

What is tensor differentiation, and how is it relevant in computational graph execution using the PyTorch framework?

Solution

The computational graph network is represented by nodes and connected through functions. There are two different kinds of nodes: dependent and independent. *Dependent nodes* are waiting for results from other nodes to process the input. *Independent nodes* are connected and are either constants or the results. Tensor differentiation is an efficient method to perform computation in a computational graph environment.

How It Works

In a computational graph, tensor differentiation is very effective because the tensors can be computed as parallel nodes, multiprocess nodes, or multithreading nodes. The major deep learning and neural computation frameworks include this tensor differentiation.

Autograd is the function that helps perform tensor differentiation, which means calculating the gradients or slope of the error function, and backpropagating errors through the neural network to fine-tune the weights and biases. Through the learning rate and iteration, it tries to reduce the error value or loss function.

To apply tensor differentiation, the nn.backward() method needs to be applied. Let's take an example and see how the error gradients are backpropagated. To update the curve of the loss function, or to find where the shape of the loss function is minimum and in which direction it is moving, a derivative calculation is required. Tensor differentiation is a way to compute the slope of the function in a computational graph.

```
In [219]: x = Variable(torch.ones(4, 4) * 12.5, requires_grad=True)

In [220]: x

Out[220]: tensor([[12.5000, 12.5000, 12.5000, 12.5000],
                  [12.5000, 12.5000, 12.5000, 12.5000],
                  [12.5000, 12.5000, 12.5000, 12.5000],
                  [12.5000, 12.5000, 12.5000, 12.5000]], requires_grad=True)

In [221]: fn = 2 * (x * x) + 5 * x + 6

          # 2x^2 + 5x + 6

In [224]: fn.backward(torch.ones(4,4))

In [225]: print(x.grad)

          tensor([[55., 55., 55., 55.],
                  [55., 55., 55., 55.],
                  [55., 55., 55., 55.],
                  [55., 55., 55., 55.]])
```

In this script, the x is a sample tensor, for which automatic gradient calculation needs to happen. The fn is a linear function that is created using the x variable. Using the backward function, we can perform a backpropagation calculation. The .grad() function holds the final output from the tensor differentiation.

Conclusion

This chapter discussed various activation functions and the use of the activation functions in various situations. The method or system to select the best activation function is accuracy driven; the activation function that gives the best results should always be used dynamically in the model. We also created a basic neural network model using small sample tensors, updated the weights using optimization, and generated predictions. In the next chapter, we see more examples.

CHAPTER 5

Supervised Learning Using PyTorch

Supervised machine learning is the most sophisticated branch of machine learning. It is in use in almost all fields, including artificial intelligence, cognitive computing, and language processing. Machine learning literature broadly talks about three types of learning: supervised, unsupervised, and reinforcement learning. In supervised learning, the machine learns to recognize the output; hence, it is task driven and the task can be classification or regression.

In unsupervised learning, the machine learns patterns from data; thus, it generalizes the new dataset and the learning happens by taking a set of input features. In reinforcement learning, the learning happens in response to a system that reacts to situations.

This chapter covers regression techniques in detail with a machine learning approach and interprets the output from regression methods in the context of a business scenario. The algorithmic classification is shown in Figure 5-1.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_5

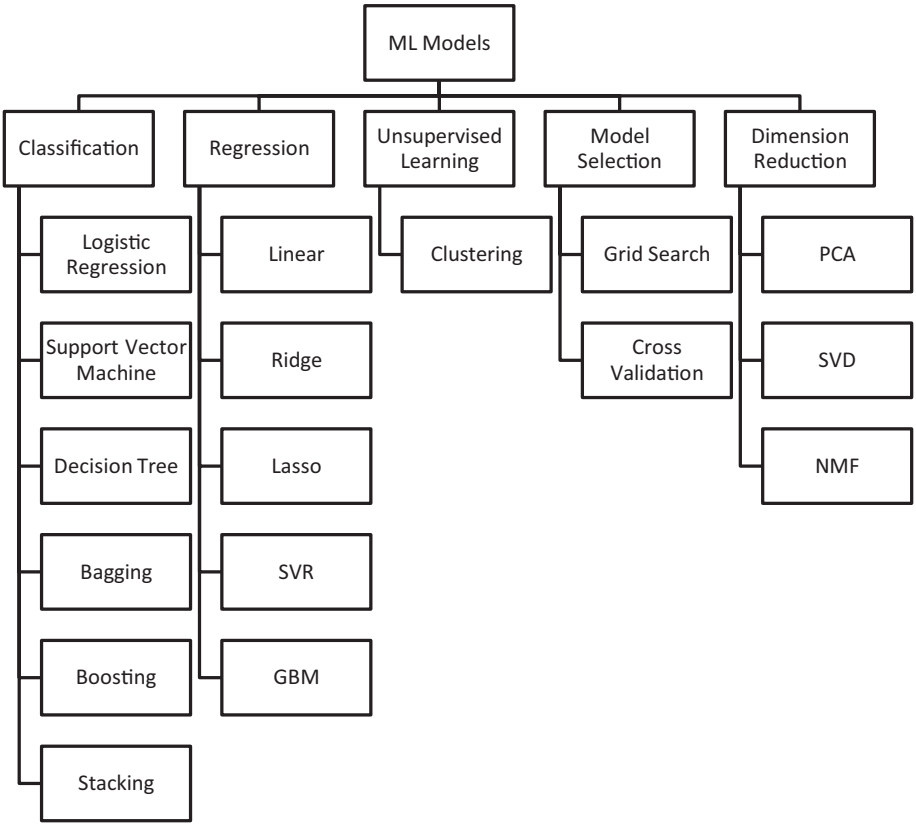


Figure 5-1. *Algorithmic classification*

Each object or row represents one event and each event is categorized into groups. Identifying which level group a record belongs to is called *classification*, in which the target variable has specific labels or tags attached to the events. For example, in a bank database, each customer is tagged as either a loyal customer or not a loyal customer. In a medical records database, each patient's disease is tagged. In the telecom industry, each subscriber is tagged as a churn or non-churn customer. These are examples in which a supervised algorithm performs classification. The word *classification* comes from the classes available in the target column.

In *regression learning*, the objective is to predict the value of a continuous variable; for example, given the features of a property, such as the number of bedrooms, square feet, nearby areas, the township, and so forth, the asking price for the house is determined. In such scenarios, the regression models can be used. Similar examples include predicting stock prices or the sales, revenue, and profit of a business.

In an unsupervised learning algorithm, we do not have an outcome variable, and tagging or labeling is not available. We are interested in knowing the natural grouping of the observations, or records, or rows in a dataset. This natural grouping should be in such a way that within groups, similarity should be at a maximum and between groups similarity should be at a minimum.

In real-world scenarios, there are cases where regression does not help predict the target variable. In supervised regression techniques, the input data is also known as *training data*. For each record, there is a label that has a continuous numerical value. The model is prepared through a training process that predicts the right output, and the process continues until the desired level of accuracy is achieved. We may need advanced regression methods to understand the pattern existing in the dataset.

Introduction to Linear Regression

Linear regression analysis is known as the most reliable, easy to apply, and most widely used among all statistical techniques. This assumes linear, additive relationships between dependent and independent variables. The objective of linear regression is to predict the dependent or target variable through independent variables. The specification of the linear regression model is as follows.

$$Y = \alpha + \beta X$$

This formula has a property in which the prediction for Y is a straight-line function of each of the X variables, keeping all others fixed, and the contributions of different X variables for the predictions are additive.

The slopes of their individual straight-line relationships with Y are the coefficients of the variables. The coefficients and intercept are estimated by least squares (i.e., setting them equal to the unique values that minimize the sum of squared errors within the sample of data to which the model is fitted).

The model's prediction errors are typically assumed to be independently and identically normally distributed. When the beta coefficient becomes zero, the input variable X has no impact on the dependent variable. The OLS method attempts to minimize the sum of the squared residuals. The residuals are defined as the difference between the points on the regression line to the actual data points in the scatterplot. This process seeks to estimate the beta coefficients in a multiple linear regression model.

Let's take a sample dataset of 15 people. We capture the height and weight for each of them. By taking only their heights, can we predict the weight of a person using a linear regression technique? The answer is yes.

Person	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Height	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
Weight	115	117	120	123	126	129	132	135	139	142	146	150	154	159	164

To represent this graphically, we measure height on the x axis, and we measure weight on the y axis. The linear regression equation is on the graph where the intercept is 87.517 and the coefficient is 3.45. The data points are represented by dots and the connecting line shows linear relationship (see Figure 5-2).

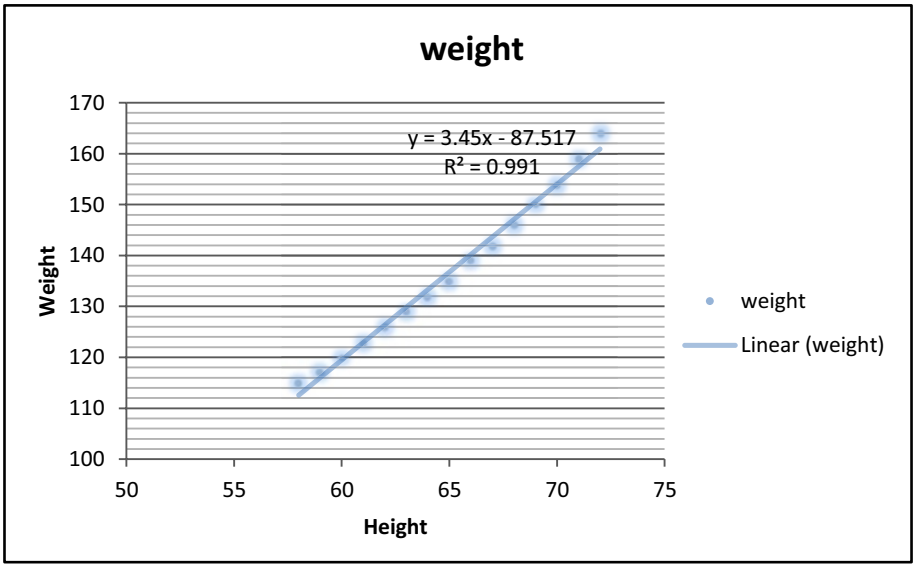


Figure 5-2. *Height and weight relationships*

Why do we assume that a linear relationship exists between the dependent variable and a set of independent variables, when most o real-life scenarios reflect any other type of relationship than a linear relationship? The reasons why we stick to linear relationship are described next.

It is easy to understand and interpret. There are ways to transform an existing deviation from linearity and make it linear. It is simple to generate prediction.

The field of predictive modeling is mainly concerned with minimizing the errors in a predictive model, or making the most accurate predictions possible. Linear regression was developed in the field of statistics. It is studied as a model for understanding the relationship between the input and the output of numerical variables, but it has been borrowed by machine learning. It is both a statistical algorithm and a machine learning algorithm. The linear regression model depends on the following set of assumptions.

- The linear relationship between dependent and independent variables.
- There should not be any multicollinearity among the predictors. If we have more than two predictors in the input feature space, the input features should not be correlated.
- There should not be any autocorrelation.
- There should not be any heteroscedasticity. The variance of the error term should be constant, along the predictors on another axis, which means the error variance should be constant.
- The error term should be normally distributed. The error term is basically defined as the difference between an actual and a predicted variable.

Within linear regression, there are different variants but in machine learning we consider them as one method. For example, if we are using one explanatory variable to predict the dependent variable, it is called a *simple linear regression model*. If we are using more than one explanatory variable, then the model is called a *multiple linear regression model*. The ordinary least square is a statistical technique to predict the linear regression model; hence, sometimes the linear regression model is also known as an *ordinary least square model*.

Linear regression is very sensitive to missing values and outliers because the statistical method of computing a linear regression depends on the mean, standard deviation, and covariance between the variables. Mean is sensitive to outlier values; therefore, it is expected that we need to clear out the outliers before proceeding toward forming the linear regression model.

In machine learning literature, the method for getting optimum beta coefficients that minimize the error in a regression model is achieved by a method called a *gradient descent algorithm*. How does the gradient descent algorithm work? It starts with an initial value, preferably from zero, and updates the scaling factor by a learning rate regularly iteratively to minimize the error term.

Understanding linear regression based on a machine learning approach requires special data preparation that avoids assumptions by keeping the original data intact. Data transformation is required to make your model more robust.

Recipe 5-1. Data Preparation for the Supervised Model

Problem

How do we perform data preparation for creating a supervised learning model using PyTorch?

Solution

We take an open source dataset, mtcars.csv, which is a regression dataset, to test how to create an input and output tensor.

How It Works

First, the necessary library needs to be imported.

```
In [1]: import torch
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        from torch.autograd import Variable
        import torch.nn.functional as F
        %matplotlib inline

In [2]: torch.__version__
Out[2]: '0.4.1'

In [3]: df = pd.read_csv("C:/Users/Documents/mtcars.csv")

In [4]: df.head()
Out[4]:
```

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

The predictor for the supervised algorithm is qsec, which is used to predict the mileage per gallon provided by the car. What is important here is the data type. First, we import the data, which is in NumPy format, into a PyTorch tensor format. The default tensor format is a float. Using the tensor float format would cause errors when performing the optimization function, so it is important to change the tensor data type. We can reformat the tensor type by using the unsqueeze function and specifying that the dimension is equal to 1.

```
In [5]: torch.manual_seed(1234)    # reproducible
Out[5]: <torch._C.Generator at 0x252fa643b50>

In [6]: x = torch.unsqueeze(torch.from_numpy(np.array(df.qsec)),dim=1)
        y = torch.unsqueeze(torch.from_numpy(np.array(df.mpg)),dim=1)

In [7]: x[0:10]
Out[7]: tensor([[16.4600],
                [17.0200],
                [18.6100],
                [19.4400],
                [17.0200],
                [20.2200],
                [15.8400],
                [20.0000],
                [22.9000],
                [18.3000]], dtype=torch.float64)
```

To reproduce the same result, a manual seed needs to be set; so torch.manual_seed(1234) was used. Although we see that the data type is a tensor, if we check the type function, it will show as double, because a tensor type double is required for the optimization function.

```
In [8]: y[0:10]
Out[8]: tensor([[21.0000],
                [21.0000],
                [22.8000],
                [21.4000],
                [18.7000],
                [18.1000],
                [14.3000],
                [24.4000],
                [22.8000],
                [19.2000]], dtype=torch.float64)
```

Recipe 5-2. Forward and Backward Propagation

Problem

How do we build a neural network torch class function so that we can build a forward propagation method?

Solution

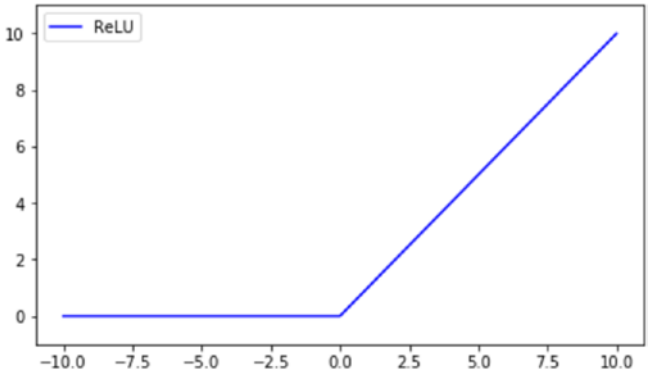
Design the neural network class function, including the hidden layer from the input layer and from the hidden layer to the output layer. In the neural network architecture, the number of neurons in the hidden layer also needs to be specified.

How It Works

In the class Net() function, we first initialize the feature, hidden, and output layers. Then we introduce the back-propagation function using the rectified linear unit as the activation function in the hidden layer.

```
In [9]: class Net(torch.nn.Module):
            def __init__(self, n_feature, n_hidden, n_output):
                super(Net, self).__init__()
                self.hidden = torch.nn.Linear(n_feature, n_hidden)   # hidden layer
                self.predict = torch.nn.Linear(n_hidden, n_output)   # output layer

            def forward(self, x):
                x = F.relu(self.hidden(x))      # activation function for hidden layer
                x = self.predict(x)             # linear output
                return x
```

The following image shows the ReLU activation function. It is popularly used across different neural network models; however, the choice of the activation function should be based on accuracy. If we get more accuracy in a sigmoid function, we should consider that.



Now the network architecture is mentioned in the supervised learning model. The n_feature shows the number of neurons in the input layer. Since we have one input variable, qsec, we will use 1. The number of neurons in the hidden layer can be decided based on the input and the

degree of accuracy required in the learning model. We use the n_hidden equal to 20, which means 20 neurons in the hidden layer 1, and the output neuron is 1.

```
In [10]: net = Net(n_feature=1, n_hidden=20, n_output=1)
         net.double()
         print(net)  # Neural network architecture

         Net(
           (hidden): Linear(in_features=1, out_features=20, bias=True)
           (predict): Linear(in_features=20, out_features=1, bias=True)
         )

In [11]: optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
         loss_func = torch.nn.MSELoss()  # this is for regression mean squared loss
```

The role of the optimization function is to minimize the loss function defined with respect to the parameters and the learning rate. The learning rate chosen here is 0.2. We also pass the neural network parameters into the optimizer. There are various optimization functions.

- *SGD*. Implements stochastic gradient descent (optionally with momentum). The parameters could be momentum, learning rate, and weight decay.
- *Adadelta*. Adaptive learning rate. Has five different arguments, parameters of the network, a coefficient used for computing a running average of the squared gradients, the addition of a term for achieving numerical stability of the model, the learning rate, and a weight decay parameter to apply regularization.
- *Adagrad*. Adaptive subgradient methods for online learning and stochastic optimization. Has arguments such as iterable of parameter to optimize the learning rate and learning rate decay with weight decay.

- *Adam*. A method for stochastic optimization. This function has six different arguments, an iterable of parameters to optimize, learning rate, betas (known as coefficients used for computing running averages of the gradient and its square), a parameter to improve numerical stability, and so forth.
- *ASGD*. Acceleration of stochastic approximation by averaging. It has five different arguments, iterable of parameters to optimize, learning rate, decay term, weight decay, and so forth.
- *RMSprop algorithm*. Uses a magnitude of gradients that are calculated to normalize the gradients.
- *SparseAdam*. Implements a lazy version of the Adam algorithm suitable for sparse tensors. In this variant, only moments that show up in the gradient are updated, and only those portions of the gradient are applied to the parameters.

Apart from the optimization function, a loss function needs to be selected before running the supervised learning model. Again, there are various loss functions; let's look at the error functions.

- *MSELoss*. Creates a criterion that measures the mean squared error between elements in the input variable and target variable. For regression-related problems, this is the best loss function.

```
In [11]: optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
         loss_func = torch.nn.MSELoss()
         # this is for regression mean squared loss

In [23]: optimizer

Out[23]: SGD (
         Parameter Group 0
             dampening: 0
             lr: 0.2
             momentum: 0
             nesterov: False
             weight_decay: 0
         )

In [24]: loss_func

Out[24]: MSELoss()

In [25]: #Turn the interactive mode on
         plt.ion()
```

After running the supervised learning model, which is a regression model, we need to print the actual vs. predicted values and represent them in a graphical format; therefore, we need to turn on the interactive feature of the model.

Recipe 5-3. Optimization and Gradient Computation

Problem

How do we build a basic supervised neural network training model using PyTorch with different iterations?

Solution

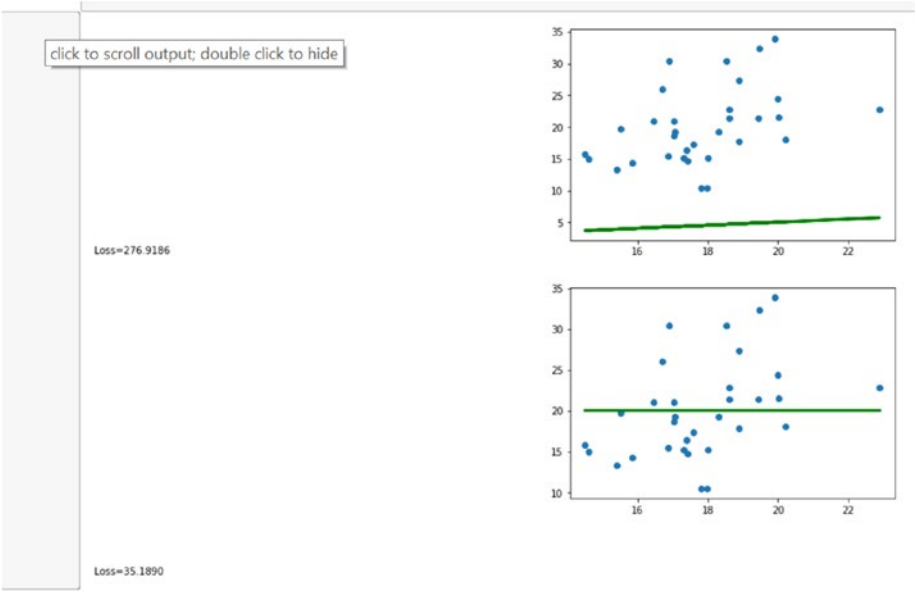
The basic neural network model in PyTorch requires six different steps: preparing training data, initializing weights, creating a basic network model, calculating loss function, selecting the learning rate, and optimizing the loss function with respect to the parameters of the model.

How It Works

Let's follow a step-by-step approach to create a basic neural network model.

```
In [26]: for t in range(100):
             prediction = net(x)     # input x and predict based on x
             loss = loss_func(prediction, y)     # must be (1. nn output, 2. target)
             optimizer.zero_grad()   # clear gradients for next train
             loss.backward()         # backpropagation, compute gradients
             optimizer.step()        # apply gradients

             if t % 50 == 0:
                 # plot and show learning process
                 plt.cla()
                 plt.scatter(x.data.numpy(), y.data.numpy())
                 plt.plot(x.data.numpy(), prediction.data.numpy(), 'g-', lw=3)
                 plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy())
                 plt.show()
         plt.ioff()
```

The final prediction result from the model with the first iteration and the last iteration is now represented in the following graph.



In the initial step, the loss function was 276.91. After optimization, the loss function became 35.1890. The fitted regression line and the way it is fitted to the dataset are represented.

Recipe 5-4. Viewing Predictions

Problem

How do we extract the best results from the PyTorch-based supervised learning model?

Solution

The computational graph network is represented by nodes and connected through functions. Various techniques can be applied to minimize the error function and get the best predictive model. We can increase the iteration numbers, estimate the loss function, optimize the function, print actual and predicted values, and show it in a graph.

How It Works

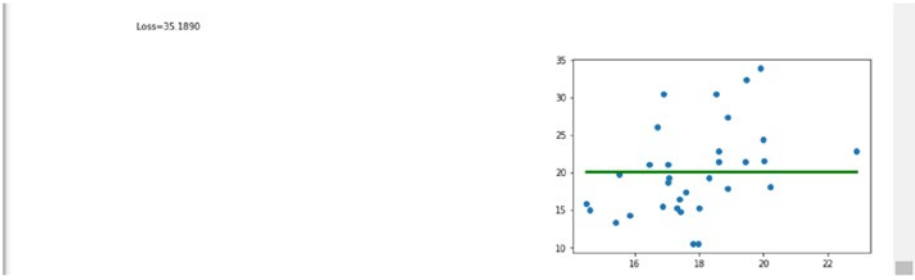
To apply tensor differentiation, the nn.backward() method needs to be applied. Let's take an example to see how the error gradients are backpropagated. The grad() function holds the final output from the tensor differentiation.

```
In [27]: optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
         loss_func = torch.nn.MSELoss()  # this is for regression mean squared loss
```

```
In [28]: for t in range(1000):
             prediction = net(x)     # input x and predict based on x
             loss = loss_func(prediction, y)     # must be (1. nn output, 2. target)
             optimizer.zero_grad()   # clear gradients for next train
             loss.backward()         # backpropagation, compute gradients
             optimizer.step()        # apply gradients

             if t % 100 == 0:
                 # plot and show learning process
                 plt.cla()
                 plt.scatter(x.data.numpy(), y.data.numpy())
                 plt.plot(x.data.numpy(), prediction.data.numpy(), 'g-', lw=3)
                 plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy())
                 plt.show()
         plt.ioff() #Turn the interactive mode off
```

The tuning parameters that can increase the accuracy of the supervised learning model, which is a regression use case, can be achieved with the following methods.

- Number of iterations
- Type of loss function
- Selection of optimization method
- Selection of loss function
- Learning rate
- Decay in the learning rate
- Momentum require for optimization



The real dataset looks like the following.

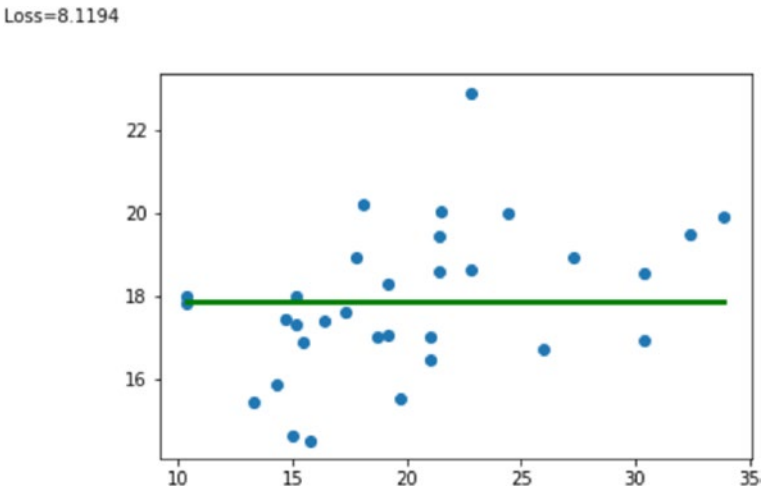
```
In [348]: df.head()
Out[348]:
```

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

The following script explains reading the mpg and qsec columns from the mtcars.csv dataset. It converts those two variables to tensors using the unsqueeze function, and then uses it inside the neural network model for prediction.

```
In [30]: x = torch.unsqueeze(torch.from_numpy(np.array(df.mpg)),dim=1)
         y = torch.unsqueeze(torch.from_numpy(np.array(df.qsec)),dim=1)

In [31]: optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
         loss_func = torch.nn.MSELoss()  # this is for regression mean squared loss

In [32]: plt.ion() #Turn the interactive mode on

In [33]: for t in range(1000):
             prediction = net(x)     # input x and predict based on x
             loss = loss_func(prediction, y)     # must be (1. nn output, 2. target)
             optimizer.zero_grad()   # clear gradients for next train
             loss.backward()         # backpropagation, compute gradients
             optimizer.step()        # apply gradients

             if t % 200 == 0:
                 # plot and show learning process
                 plt.cla()
                 plt.scatter(x.data.numpy(), y.data.numpy())
                 plt.plot(x.data.numpy(), prediction.data.numpy(), 'g-', lw=3)
                 plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy())
                 plt.show()
         plt.ioff() #Turn the interactive mode off
```

After 1000 iterations, the model converges.



The neural networks in the torch library are typically used with the nn module. Let's take a look at that.

Neural networks can be constructed using the torch.nn package, which provides almost all neural network related functionalities, including the following.

- *Linear layers*: nn.Linear, nn.Bilinear
- *Convolution layers*: nn.Conv1d, nn.Conv2d, nn.Conv3d, nn.ConvTranspose2d
- *Nonlinearities*: nn.Sigmoid, nn.Tanh, nn.ReLU, nn.LeakyReLU
- *Pooling layers*: nn.MaxPool1d, nn.AveragePool2d
- *Recurrent networks*: nn.LSTM, nn.GRU
- *Normalization*: nn.BatchNorm2d
- *Dropout*: nn.Dropout, nn.Dropout2d
- *Embedding*: nn.Embedding
- *Loss functions*: nn.MSELoss, nn.CrossEntropyLoss, nn.NLLLoss

The standard classification algorithm is another version of a supervised learning algorithm, in which the target column is a class variable and the features could be numeric and categorical.

Recipe 5-5. Supervised Model Logistic Regression

Problem

How do we deploy a logistic regression model using PyTorch?

Solution

The computational graph network is represented by nodes and connected through functions. Various techniques can be applied to minimize the error function and get the best predictive model. We can increase the iteration numbers, estimate the loss function, optimize the function, print actual and predicted values, and show it in a graph.

How It Works

To apply tensor differentiation, the nn.backward() method needs to be applied. Let's look at an example.

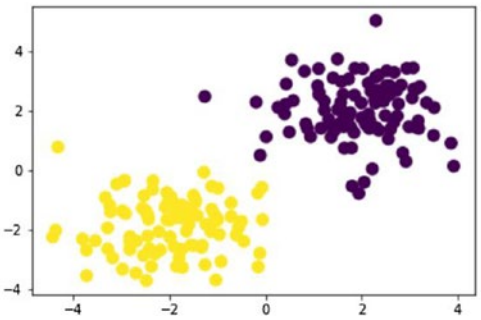
```
In [2]: import torch
        from torch.autograd import Variable
        import torch.nn.functional as F
        import matplotlib.pyplot as plt
        %matplotlib inline

        torch.manual_seed(1)    # reproducible
Out[2]: <torch._C.Generator at 0x2999b184bb0>
```

The following shows data preparation for a logistic regression model.

In [4]:
```
#Sample data preparation
n_data = torch.ones(100, 2)
x0 = torch.normal(2*n_data, 1)
y0 = torch.zeros(100)
x1 = torch.normal(-2*n_data, 1)
y1 = torch.ones(100)
x = torch.cat((x0, x1), 0).type(torch.FloatTensor)
y = torch.cat((y0, y1), ).type(torch.LongTensor)

# torch need to train on Variable, so convert sample features to Variable
x, y = Variable(x), Variable(y)
```

Let's look at the sample dataset for classification.

In [5]:
```
plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=y.data.numpy(), s=100,
plt.show()
```



Set up the neural network module for the logistic regression model.

In [6]:
```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)   # hidden layer
        self.out = torch.nn.Linear(n_hidden, n_output)   # output layer

    def forward(self, x):
        x = F.sigmoid(self.hidden(x))  # activation function for hidden layer
        x = self.out(x)
        return x
```

Check the neural network configuration.

```
In [7]: net = Net(n_feature=2, n_hidden=10, n_output=2)     # define the network
        print(net)  # net architecture

        # Loss and Optimizer
        # Softmax is internally computed.
        # Set parameters to be updated.
        optimizer = torch.optim.SGD(net.parameters(), lr=0.02)
        loss_func = torch.nn.CrossEntropyLoss()  # the target label is NOT an one-hotted

        Net(
          (hidden): Linear(in_features=2, out_features=10, bias=True)
          (out): Linear(in_features=10, out_features=2, bias=True)
        )
```

Run iterations and find the best solution for the sample graph.

```
In [8]: plt.ion() # interactive graph on

In [10]: for t in range(100):
             out = net(x)                 # input x and predict based on x
             loss = loss_func(out, y)     # must be (1. nn output, 2. target), the target

             optimizer.zero_grad()   # clear gradients for next train
             loss.backward()         # backpropagation, compute gradients
             optimizer.step()        # apply gradients

             if t % 10 == 0 or t in [3, 6]:
                 # plot and show learning process
                 plt.cla()
                 _, prediction = torch.max(F.softmax(out), 1)
                 pred_y = prediction.data.numpy().squeeze()
                 target_y = y.data.numpy()
                 plt.scatter(x.data.numpy()[:, 0],
                             x.data.numpy()[:, 1],
                             c=pred_y, s=100, lw=0)
                 accuracy = sum(pred_y == target_y)/200.
                 plt.text(1.5, -4, 'Accuracy=%.2f' % accuracy,
                          fontdict={'size': 20, 'color':  'blue'})
                 plt.show()
         plt.ioff()
```

The first iteration provides almost 99% accuracy, and subsequently, the model provides 100% accuracy on the training data (see Figures 5-3 and 5-4).

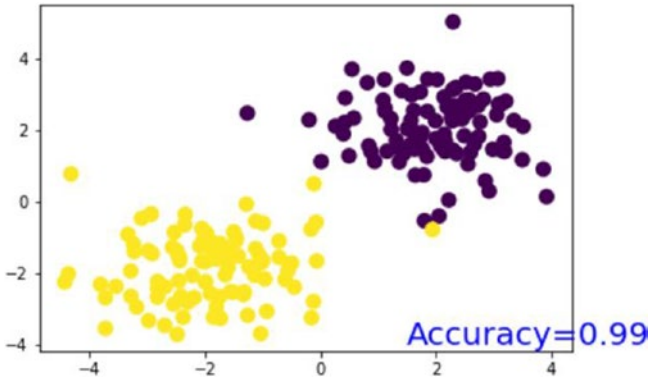


Figure 5-3. *Initial accuracy*

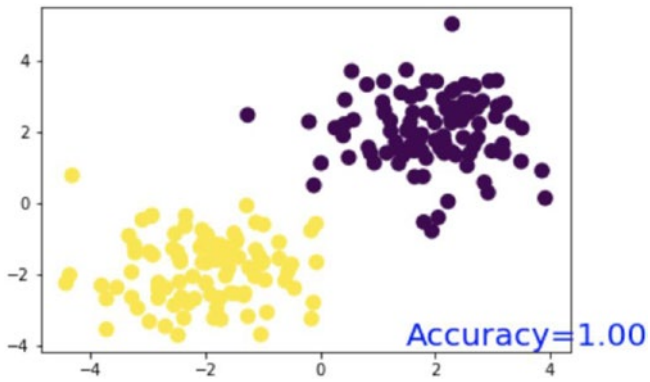


Figure 5-4. *Final accuracy*

Final accuracy shows 100, which is a clear case of overfitting, but we can control this by introducing the dropout rate, which is covered in the next chapter.

Conclusion

This chapter discussed two major types of supervised learning algorithms—linear regression and logistic regression—and their implementation using sample datasets and the PyTorch program. Both algorithms are linear models, one for predicting real valued output and the other for separating one class from another class. Although we considered a two-class classification in the logistic regression example, it can be extended to a multiclass classification model.

CHAPTER 6

Fine-Tuning Deep Learning Models Using PyTorch

Deep learning models are becoming very popular. They have very deep roots in the way biological neurons are connected and the way they transmit information from one node to another node in a network model.

Deep learning has a very specific usage, particularly when the single function–based machine learning techniques fail to approximate real-life challenges. For example, when the data dimension is very large (in the thousands), then standard machine learning algorithms fail to predict or classify the outcome variable. This is also not very efficient computationally. It consumes a lot of resources and model convergence never happens. Most prominent examples are object detection, image classification, and image segmentation.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_6

The most commonly used deep learning algorithms can be classified into three groups.

- *Convolutional neural network*. Mostly suitable for highly sparse datasets, image classification, image recognition, object detection, and so forth.
- *Recurrent neural network*. Applicable to processing sequential information, if there is any internal sequential structure in the way data is generated. This includes music, natural language, audio, and video, where the information is consumed in a sequence.
- *Deep neural network*. Typically applicable when a single layer of a machine learning algorithm cannot classify or predict correctly. There are three variants.
    - *Deep network*, where the number of neurons present in each hidden layer is usually more than the previous layer
    - *Wide network*, where the number of hidden layers are more than a usual neural network model
    - *Both deep and wide network*, where the number of neurons and the number of layers in the network are very high

This chapter discusses how to fine-tune deep learning models using hyperparameters. There is a difference between the parameters and hyperparameters. Usually in the deep learning models, we are not interested in estimating the parameters because they are the weights and keep changing based on the initial values, learning rate, and number of iterations. What is important is deciding on the hyperparameters to fine-tune the models, as discussed in Chapter 3, so that optimum results can be derived.

Recipe 6-1. Building Sequential Neural Networks

Problem

Is there any way to build sequential neural network models, as we do in Keras in PyTorch, instead of declaring the neural network models?

Solution

If we declare the entire neural network model, line by line, with the number of neurons, number of hidden layers and iterations, choice of loss functions, optimization functions, and the selection of weight distribution, and so forth, it will be extremely cumbersome to scale the model. And, it is not foolproof—errors could crop up in the model. To avoid the issues in declaring the entire model line by line, we can use a high-level function that assumes certain default parameters in the back end and returns the result to the user with minimum hyperparameters. Yes, it is possible to not have to declare the neural network model.

How It Works

Let's look at how to create such models. In the Torch library, the neural network module contains a functional API (application programming interface) that contains various activation functions, as discussed in earlier chapters.

```
In [1]: import torch
        import torch.nn.functional as F
```

In the following lines of script, we create a simple neural network model with linear function as the activation function for input to the hidden layer, and the hidden layer to the output layer.

The following function requires declaring `class Net`, declaring the features, hidden neurons, and activation functions, which can be easily replaced by the sequential module.

```
In [2]: # replace following class code with an easy sequential network
        class Net(torch.nn.Module):
            def __init__(self, n_feature, n_hidden, n_output):
                super(Net, self).__init__()
                self.hidden = torch.nn.Linear(n_feature, n_hidden)   # hidden layer
                self.predict = torch.nn.Linear(n_hidden, n_output)   # output layer

            def forward(self, x):
                x = F.relu(self.hidden(x))      # activation function for hidden layer
                x = self.predict(x)             # linear output
                return x
```

Instead of using this script, we can change the class function and replace it with the sequential function. The Keras functions replace the TensorFlow functions, which means that many lines of TensorFlow code can be replaced by a few lines of Keras script. The same thing is possible in PyTorch without requiring any external modules. As an example, in the following, net2 explains the sequential model and net1 explains the preceding script. From a readability perspective, net2 is much better than net1.

```
In [3]: net1 = Net(1, 100, 1)
```

```
In [4]: # easy and fast way to build your network
        net2 = torch.nn.Sequential(
            torch.nn.Linear(1, 100),
            torch.nn.ReLU(),
            torch.nn.Linear(100, 1)
        )
```

If we simply print both the net1 and net2 model architectures, it does the same thing.

```
In [5]: print(net1)     # net1 architecture
        print(net2)     # net2 architecture

        Net(
          (hidden): Linear(in_features=1, out_features=100, bias=True)
          (predict): Linear(in_features=100, out_features=1, bias=True)
        )
        Sequential(
          (0): Linear(in_features=1, out_features=100, bias=True)
          (1): ReLU()
          (2): Linear(in_features=100, out_features=1, bias=True)
        )
```

Recipe 6-2. Deciding the Batch Size

Problem

How do we perform batch data training for a deep learning model using PyTorch?

Solution

Training a deep learning model requires a large amount of labeled data. Typically, it is the process of finding a set of weights and biases in such a way that the loss function becomes minimal with respect to matching the target label. If the training process approximates well to the function, the prediction or classification becomes robust.

How It Works

There are two methods for training a deep learning network: batch training and online training. The choice of training algorithm dictates the method of learning. If the algorithm is backpropagation, then online learning is better. For a deep and wide network model with various layers of backpropagation and forward propagation, then batch training is better.

```
In [6]: import torch
        import torch.utils.data as Data

        torch.manual_seed(1234)    # reproducible

Out[6]: <torch._C.Generator at 0x20fa15ded90>

In [7]: BATCH_SIZE = 5
        # BATCH_SIZE = 8
```

In the training process, the batch size is 5; we can change the batch size to 8 and see the results. In online training process, the weights and biases are updated for every training example based on the variations between predicted result and actual result. However, in the batch training process, the differences between actual and predicted values which is error gets accumulated and computed as a single number over the batch size, and reported at the final layer.

```
In [8]: x = torch.linspace(1, 10, 10)       # this is x data (torch tensor)
        y = torch.linspace(10, 1, 10)       # this is y data (torch tensor)

In [10]: torch_dataset = Data.TensorDataset(x, y)
         loader = Data.DataLoader(
             dataset=torch_dataset,      # torch TensorDataset format
             batch_size=BATCH_SIZE,      # mini batch size
             shuffle=True,               # random shuffle for training
             num_workers=2,              # subprocesses for loading data
         )
```

After training the dataset for five iterations, we can print the batch and step. If we compare the online training and batch training, batch training has many more advantages than online training. When the requirement is to train a huge dataset, there are memory constraints. When we cannot process a huge dataset in a CPU environment, batch training comes to the rescue. In a CPU environment, we can process large amounts of data with a smaller batch size.

In [56]:
```
for epoch in range(5):   # train entire dataset 5 times
    for step, (batch_x, batch_y) in enumerate(loader):  # for each training step
        # train your data...
        print('Epoch: ', epoch, '| Step: ', step, '| batch x: ',
              batch_x.numpy(), '| batch y: ', batch_y.numpy())
```

```
Epoch:  0 | Step:  0 | batch x:  [7. 4. 1. 5. 2.] | batch y:  [ 4.  7. 10.  6.
9.]
Epoch:  0 | Step:  1 | batch x:  [ 6. 10.  8.  3.  9.] | batch y:  [5. 1. 3. 8.
2.]
Epoch:  1 | Step:  0 | batch x:  [3. 2. 9. 8. 5.] | batch y:  [8. 9. 2. 3. 6.]
Epoch:  1 | Step:  1 | batch x:  [ 4.  1.  6.  7. 10.] | batch y:  [ 7. 10.  5.
  4.  1.]
Epoch:  2 | Step:  0 | batch x:  [10.  8.  4.  1.  5.] | batch y:  [ 1.  3.  7.
10.  6.]
Epoch:  2 | Step:  1 | batch x:  [2. 3. 7. 9. 6.] | batch y:  [9. 8. 4. 2. 5.]
Epoch:  3 | Step:  0 | batch x:  [6. 8. 9. 5. 1.] | batch y:  [ 5.  3.  2.  6.
10.]
Epoch:  3 | Step:  1 | batch x:  [ 7. 10.  3.  4.  2.] | batch y:  [4. 1. 8. 7.
9.]
Epoch:  4 | Step:  0 | batch x:  [ 4.  7.  6.  2. 10.] | batch y:  [7. 4. 5. 9.
1.]
Epoch:  4 | Step:  1 | batch x:  [9. 5. 3. 8. 1.] | batch y:  [ 2.  6.  8.  3.
10.]
```

We take the batch size as 8 and retrain the model.

In [57]:
```
BATCH_SIZE = 8
loader = Data.DataLoader(
    dataset=torch_dataset,      # torch TensorDataset format
    batch_size=BATCH_SIZE,      # mini batch size
    shuffle=True,               # random shuffle for training
    num_workers=2,              # subprocesses for loading data
)
```

In [58]:
```
for epoch in range(5):   # train entire dataset 5 times
    for step, (batch_x, batch_y) in enumerate(loader):  # for each training step
        # train your data...
        print('Epoch: ', epoch, '| Step: ', step, '| batch x: ',
              batch_x.numpy(), '| batch y: ', batch_y.numpy())
```

```
Epoch:  0 | Step:  0 | batch x:  [10.  5.  7.  6.  8.  1.  3.  4.] | batch y:
[ 1.  6.  4.  5.  3. 10.  8.  7.]
Epoch:  0 | Step:  1 | batch x:  [2. 9.] | batch y:  [9. 2.]
Epoch:  1 | Step:  0 | batch x:  [ 5.  6.  8.  3. 10.  1.  4.  9.] | batch y:
[ 6.  5.  3.  8.  1. 10.  7.  2.]
Epoch:  1 | Step:  1 | batch x:  [2. 7.] | batch y:  [9. 4.]
Epoch:  2 | Step:  0 | batch x:  [9. 1. 3. 7. 5. 2. 4. 8.] | batch y:  [ 2. 10.
  8.  4.  6.  9.  7.  3.]
Epoch:  2 | Step:  1 | batch x:  [ 6. 10.] | batch y:  [5. 1.]
Epoch:  3 | Step:  0 | batch x:  [ 3.  5. 10.  2.  9.  8.  4.  6.] | batch y:
[8. 6. 1. 9. 2. 3. 7. 5.]
Epoch:  3 | Step:  1 | batch x:  [1. 7.] | batch y:  [10.  4.]
Epoch:  4 | Step:  0 | batch x:  [10.  5.  6.  4.  2.  1.  9.  7.] | batch y:
[ 1.  6.  5.  7.  9. 10.  2.  4.]
Epoch:  4 | Step:  1 | batch x:  [8. 3.] | batch y:  [3. 8.]
```

Recipe 6-3. Deciding the Learning Rate

Problem

How do we identify the best solution based on learning rate and the number of epochs?

Solution

We take a sample tensor and apply various alternative models and print model parameters. The learning rate and epoch number are associated with model accuracy. To reach the global minimum state of the loss function, it is important to keep the learning rate to a minimum and the epoch number to a maximum so that the iteration can take the loss function to the minimum state.

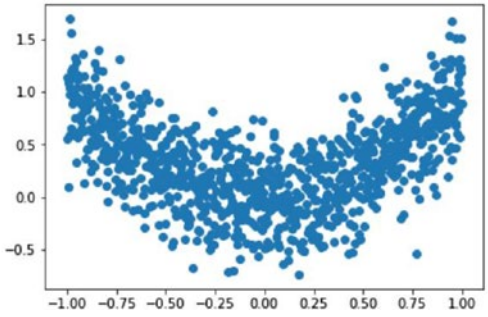
How It Works

First, the necessary library needs to be imported. To find the minimum loss function, gradient descent is typically used as the optimization algorithm, which is an iterative process. The objective is to find the rate of decline of the loss function with respect to the trainable parameters.

```
In [59]: import torch
         import torch.utils.data as Data
         import torch.nn.functional as F
         from torch.autograd import Variable
         import matplotlib.pyplot as plt
         %matplotlib inline

In [60]: torch.manual_seed(12345)    # reproducible
Out[60]: <torch._C.Generator at 0x20fa15ded90>

In [61]: LR = 0.01
         BATCH_SIZE = 32
         EPOCH = 12
```

The sample dataset taken for the experiment includes the following.

```
In [62]: # sample dataset
         x = torch.unsqueeze(torch.linspace(-1, 1, 1000), dim=1)
         y = x.pow(2) + 0.3*torch.normal(torch.zeros(*x.size()))

         # plot dataset
         plt.scatter(x.numpy(), y.numpy())
         plt.show()
```



The sample dataset and the first five records would look like the following.

```
In [71]: x[0:10]
Out[71]: tensor([[-1.0000],
                 [-0.9980],
                 [-0.9960],
                 [-0.9940],
                 [-0.9920],
                 [-0.9900],
                 [-0.9880],
                 [-0.9860],
                 [-0.9840],
                 [-0.9820]])

In [72]: y[0:10]
Out[72]: tensor([[0.5561],
                 [1.1422],
                 [0.0882],
                 [1.1212],
                 [1.0920],
                 [0.9764],
                 [1.0417],
                 [0.5877],
                 [1.6916],
                 [1.5640]])
```

Using the PyTorch utility function, let's load the tensor dataset, introduce the batch size, and test out.

```
In [73]: torch_dataset = Data.TensorDataset(x, y)
         loader = Data.DataLoader(
             dataset=torch_dataset,
             batch_size=BATCH_SIZE,
             shuffle=True, num_workers=2,)

In [74]: torch_dataset
Out[74]: <torch.utils.data.dataset.TensorDataset at 0x20fa410bf60>

In [75]: loader
Out[75]: <torch.utils.data.dataloader.DataLoader at 0x20fa410b2b0>
```

Declare the neural network module.

```
In [76]: class Net(torch.nn.Module):
             def __init__(self):
                 super(Net, self).__init__()
                 self.hidden = torch.nn.Linear(1, 20)   # hidden layer
                 self.predict = torch.nn.Linear(20, 1)   # output layer

             def forward(self, x):
                 x = F.relu(self.hidden(x))      # activation function for hidden layer
                 x = self.predict(x)             # linear output
                 return x

In [77]: net_SGD         = Net()
         net_Momentum    = Net()
         net_RMSprop     = Net()
         net_Adam        = Net()
         nets = [net_SGD, net_Momentum, net_RMSprop, net_Adam]
```

Now, let's look at the network architecture.

```
In [78]: net_Adam
Out[78]: Net(
           (hidden): Linear(in_features=1, out_features=20, bias=True)
           (predict): Linear(in_features=20, out_features=1, bias=True)
         )

In [79]: net_Momentum
Out[79]: Net(
           (hidden): Linear(in_features=1, out_features=20, bias=True)
           (predict): Linear(in_features=20, out_features=1, bias=True)
         )

In [80]: net_RMSprop
Out[80]: Net(
           (hidden): Linear(in_features=1, out_features=20, bias=True)
           (predict): Linear(in_features=20, out_features=1, bias=True)
         )

In [81]: net_SGD
Out[81]: Net(
           (hidden): Linear(in_features=1, out_features=20, bias=True)
           (predict): Linear(in_features=20, out_features=1, bias=True)
         )
```

While performing the optimization, we can include many options; select the best among the best.

```
In [82]: opt_SGD         = torch.optim.SGD(net_SGD.parameters(), lr=LR)
         opt_Momentum    = torch.optim.SGD(net_Momentum.parameters(),
                                           lr=LR, momentum=0.8)
         opt_RMSprop     = torch.optim.RMSprop(net_RMSprop.parameters(),
                                               lr=LR, alpha=0.9)
         opt_Adam        = torch.optim.Adam(net_Adam.parameters(),
                                            lr=LR, betas=(0.9, 0.99))
         optimizers = [opt_SGD, opt_Momentum, opt_RMSprop, opt_Adam]

In [83]: opt_Adam
Out[83]: Adam (
         Parameter Group 0
             amsgrad: False
             betas: (0.9, 0.99)
             eps: 1e-08
             lr: 0.01
             weight_decay: 0
         )
```

```
In [87]: opt_Momentum

Out[87]: SGD (
         Parameter Group 0
             dampening: 0
             lr: 0.01
             momentum: 0.8
             nesterov: False
             weight_decay: 0
         )

In [88]: opt_RMSprop

Out[88]: RMSprop (
         Parameter Group 0
             alpha: 0.9
             centered: False
             eps: 1e-08
             lr: 0.01
             momentum: 0
             weight_decay: 0
         )

In [89]: opt_SGD

Out[89]: SGD (
         Parameter Group 0
             dampening: 0
             lr: 0.01
             momentum: 0
             nesterov: False
             weight_decay: 0
         )

In [67]: loss_func = torch.nn.MSELoss()
         losses_his = [[], [], [], []]   # record loss

In [90]: loss_func

Out[90]: MSELoss()
```

Recipe 6-4. Performing Parallel Training

Problem

How do we perform parallel data training that includes a lot of models using PyTorch?

Solution

The optimizers are really functions that augment the tensor. The process of finding a best model requires parallel training of many models. The choice of learning rate, batch size, and optimization algorithms make models unique and different from other models. The process of selecting the best model requires hyperparameter optimization.

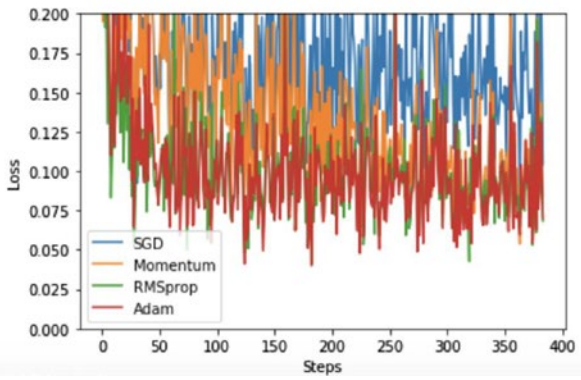
How It Works

First, the right library needs to be imported. The three hyperparameters (learning rate, batch size, and optimization algorithm) make it possible to train multiple models in parallel, and the best model is decided by the accuracy of the test dataset. The following script uses the stochastic gradient descent algorithm, momentum, RMS prop, and Adam as the optimization method.

```
In [68]: # training
for epoch in range(EPOCH):
    print('Epoch: ', epoch)
    for step, (batch_x, batch_y) in enumerate(loader):          # for each traini
        b_x = Variable(batch_x)
        b_y = Variable(batch_y)

        for net, opt, l_his in zip(nets, optimizers, losses_his):
            output = net(b_x)              # get output for every net
            loss = loss_func(output, b_y)  # compute loss for every net
            opt.zero_grad()                # clear gradients for next train
            loss.backward()                # backpropagation, compute gradients
            opt.step()                     # apply gradients
            l_his.append(loss.data[0])     # loss recoder

labels = ['SGD', 'Momentum', 'RMSprop', 'Adam']
for i, l_his in enumerate(losses_his):
    plt.plot(l_his, label=labels[i])
plt.legend(loc='best')
plt.xlabel('Steps')
plt.ylabel('Loss')
plt.ylim((0, 0.2))
plt.show()
```

Let's look at the chart and epochs.

```
Epoch:  0
Epoch:  1
/Applications/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.p
y:14: UserWarning: invalid index of a 0-dim tensor. This will be an error
in PyTorch 0.5. Use tensor.item() to convert a 0-dim tensor to a Python n
umber

Epoch:  2
Epoch:  3
Epoch:  4
Epoch:  5
Epoch:  6
Epoch:  7
Epoch:  8
Epoch:  9
Epoch:  10
Epoch:  11
```



Conclusion

In this chapter, we looked at various ways to make the deep learning model learn from the training dataset. The training process can be made effective by using hyperparameters. The selection of the right hyperparameter is the key. The deep learning models (convolutional neural network, recurrent neural network, and deep neural network) are different in terms of architecture, but the training process and the hyperparameters remain the same. The choice of hyperparameters and selection process is much easier in PyTorch than any other framework.

CHAPTER 7

Natural Language Processing Using PyTorch

Natural language processing is an important branch of computer science. It is the study of human language by computers performing various tasks. Natural language study is also known as *computational linguistics*. There are two different components of natural language processing: natural language understanding and natural language generation. *Natural language understanding* involves analysis and knowledge of the input language and responding to it. *Natural language generation* is the process of creating language from input text. Language can be used in various ways. One word may have different meanings, so removing ambiguity is an important part of natural language understanding.

The ambiguity level can be of three types.

- *Lexical ambiguity* is based on parts of speech; deciding whether a word is a noun, verb, adverb, and so forth.
- *Syntactic ambiguity* is where one sentence can have multiple interpretations; the subject and predicate are neutral.
- *Referential ambiguity* is related to an event or scenario expressed in words.

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2_7

Text analysis is a precursor to natural language processing and understanding. Text analysis means corpus creation creating a collected set of documents, and then removing white spaces, punctuation, stop words, junk values such as symbols, emojis, and so forth, which have no textual meaning. After clean up, the net task is to represent the text in vector form. This is done using the standard Word2vec model, or it can be represented in term frequency and inverse document frequency format (tf-idf). In today's world, we see a lot of applications that use natural language processing; the following are some examples.

- Spell checking applications—online and on smartphones. The user types a particular word and the system checks the meaning of the word and suggests whether the spelling needs to be corrected.
- Keyword search has been an integral part of our lives over the last decade. Whenever we go to a restaurant, buy something, or visit some place, we do an online search. If the keyword typed is wrong, no match is retrieved; however, the search engine systems are so intelligent that they predict the user's intent and suggest pages that user actually wants to search.
- Predictive text is used in various chat applications. The user types a word, and based on the user's writing pattern, a choice of next words appear. The user is prompted to select any word from the list to frame his sentence.
- Question-and-answering systems like Google Home, Amazon Alexa, and so forth, allow users to interact with the system in natural language. The system processes that information, does an intelligent search, and retrieves the best results for the user.

- Alternate data extraction is when actual data is not available to the user, but the user can use the Internet to fetch data that is publicly available, and search for relevant information. For example, if I want to buy a laptop, I want to compare the price of the laptop on various online portals. I have one system scrape the price information from various websites and provide a summary of the prices to me. This process is called *alternate data collection* using web scraping, text processing and natural language processing.
- Sentiment analysis is a process of analyzing the mood of the customer, user, or agent from the text that they express. Customer reviews, movie reviews, and so forth. The text presented needs to be analyzed and tagged as a positive sentiment or a negative sentiment. Similar applications can be built using sentiment analysis.
- Topic modeling is the process of finding distinct topics presented in the corpus. For example, if we take text from science, math, English, and biology, and jumble all the text, then ask the machine to classify the text and tell us how many topics exist in the corpus, and the machine correctly separates the words present in English from biology, biology from science, and so on so forth. This is called a perfect topic modeling system.
- Text summarization is the process of summarizing the text from the corpus in a shorter format. If we have a two-page document that is 1000 words, and we need to summarize it in a 200-word paragraph, then we can achieve that by using text summarization algorithms.

- Language translation is translating one language to another, such as English to French, French to German, and so on so forth. Language translation helps the user understand another language and make the communication process effective.

The study of human language is discrete and very complex. The same sentence may have many meanings, but it is specifically constructed for an intended audience. To understand the complexity of natural language, we not only need tools and programs but also the system and methods. The following five-step approach is followed in natural language processing to understand the text from the user.

- Lexical analysis identifies the structure of the word.
- Syntactic analysis is the study of English grammar and syntax.
- Semantic analysis is the meaning of a word in a context.
- PoS (point of sale) analysis is the understanding and parsing parts of speech.
- Pragmatic analysis is understanding the real meaning of a word in context.

In this chapter, we use PyTorch to implement the steps that are most commonly used in natural language processing tasks.

Recipe 7-1. Word Embedding

Problem

How do we create a word-embedding model using PyTorch?

Solution

Word embedding is the process of representing the words, phrases, and tokens in a meaningful way in a vector structure. The input text is mapped to vectors of real numbers; hence, feature vectors can be used for further computation by machine learning or deep learning models.

How It Works

The words and phrases are represented in real vector format. The words or phrases that have similar meanings in a paragraph or document have similar vector representation. This makes the computation process effective in finding similar words. There are various algorithms for creating embedded vectors from text. Word2vec and GloVe are known frameworks to execute word embeddings. Let's look at the following example.

```
In [1]: import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim

        torch.manual_seed(1234)
Out[1]: <torch._C.Generator at 0x117c8d310>

In [20]: word_to_ix = {"data": 0, "science": 1}

In [21]: word_to_ix
Out[21]: {'data': 0, 'science': 1}

In [22]: embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensional embeddings

In [23]: embeds
Out[23]: Embedding(2, 5)

In [25]: lookup_tensor = torch.tensor([word_to_ix["data"]], dtype=torch.long)
         lookup_tensor
Out[25]: tensor([0])
```

The following sets up an embedding layer.

```
In [26]: hello_embed = embeds(lookup_tensor)
         print(hello_embed)

         tensor([[ 1.1772, -1.2023,  2.2256, -1.9676, -0.2218]], grad_fn=<EmbeddingBackward>)

In [27]: CONTEXT_SIZE = 2

In [28]: EMBEDDING_DIM = 10
```

Let's look at the sample text. The following text has two paragraphs, and each paragraph has several sentences. If we apply word embedding on these two paragraphs, then we will get real vectors as features from the text. Those features can be used for further computation.

```
In [30]: test_sentence = """The popularity of the term "data science" has exploded in
         business environments and academia, as indicated by a jump in job openings.[32]
         However, many critical academics and journalists see no distinction between data
         science and statistics. Writing in Forbes, Gil Press argues that data science is a
         buzzword without a clear definition and has simply replaced "business analytics" in
         contexts such as graduate degree programs.[7] In the question-and-answer section of
         his keynote address at the Joint Statistical Meetings of American Statistical
         Association, noted applied statistician Nate Silver said, "I think data-scientist
         is a sexed up term for a statistician....Statistics is a branch of science.
         Data scientist is slightly redundant in some way and people shouldn't berate the
         term statistician."[9] Similarly, in business sector, multiple researchers and
         analysts state that data scientists alone are far from being sufficient in granting
         companies a real competitive advantage[33] and consider data scientists as only
         one of the four greater job families companies require to leverage big
         data effectively, namely: data analysts, data scientists, big data developers
         and big data engineers.[34]

         On the other hand, responses to criticism are as numerous. In a 2014 Wall Street
         Journal article, Irving Wladawsky-Berger compares the data science enthusiasm with
         the dawn of computer science. He argues data science, like any other interdisciplinary
         field, employs methodologies and practices from across the academia and industry, but
         then it will morph them into a new discipline. He brings to attention the sharp criticisms
         computer science, now a well respected academic discipline, had to once face.[35] Likewise,
         NYU Stern's Vasant Dhar, as do many other academic proponents of data science,[35] argues
         more specifically in December 2013 that data science is different from the existing practice
         of data analysis across all disciplines, which focuses only on explaining data sets.
         Data science seeks actionable and consistent pattern for predictive uses.[1] This practical
         engineering goal takes data science beyond traditional analytics. Now the data in those
         disciplines and applied fields that lacked solid theories, like health science and social
         science, could be sought and utilized to generate powerful predictive models.[1]""".split()
```

Tokenization is the process of splitting sentences into small chunks of tokens, known as *n-grams*. This is called a *unigram* if it is a single word, a *bigram* if it is two words, a *trigram* if it is three words, so on and so forth.

```
# we should tokenize the input, but we will ignore that for now
# build a list of tuples.  Each tuple is ([ word_i-2, word_i-1 ], target word)
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# print the first 3, just so you can see what they look like
print(trigrams[:3])

vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}
```

```
[(['The', 'popularity'], 'of'), (['popularity', 'of'], 'the'), (['of', 'the'], 'term')]
```

The PyTorch n-gram language modeler can extract relevant key words.

```
In [31]: class NGramLanguageModeler(nn.Module):

             def __init__(self, vocab_size, embedding_dim, context_size):
                 super(NGramLanguageModeler, self).__init__()
                 self.embeddings = nn.Embedding(vocab_size, embedding_dim)
                 self.linear1 = nn.Linear(context_size * embedding_dim, 128)
                 self.linear2 = nn.Linear(128, vocab_size)

             def forward(self, inputs):
                 embeds = self.embeddings(inputs).view((1, -1))
                 out = F.relu(self.linear1(embeds))
                 out = self.linear2(out)
                 log_probs = F.log_softmax(out, dim=1)
                 return log_probs


         losses = []
         loss_function = nn.NLLLoss()
         model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
         optimizer = optim.SGD(model.parameters(), lr=0.001)
```

The n-gram extractor has three arguments: the length of the vocabulary to extract, a dimension of embedding vector, and context size. Let's look at the loss function and the model specification.

```
In [32]: losses
Out[32]: []

In [33]: loss_function
Out[33]: NLLLoss()

In [34]: model
Out[34]: NGramLanguageModeler(
           (embeddings): Embedding(228, 10)
           (linear1): Linear(in_features=20, out_features=128, bias=True)
           (linear2): Linear(in_features=128, out_features=228, bias=True)
         )
```

Apply the Adam optimizer.

```
In [35]: optimizer
Out[35]: SGD (
         Parameter Group 0
             dampening: 0
             lr: 0.001
             momentum: 0
             nesterov: False
             weight_decay: 0
         )
```

Context extraction from sentences is also important. Let's look at the following function.

```
In [36]: for epoch in range(10):
             total_loss = 0
             for context, target in trigrams:

                 # Step 1. Prepare the inputs to be passed to the model (i.e, turn the words
                 # into integer indices and wrap them in tensors)
                 context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)

                 # Step 2. Recall that torch *accumulates* gradients. Before passing in a
                 # new instance, you need to zero out the gradients from the old
                 # instance
                 model.zero_grad()

                 # Step 3. Run the forward pass, getting log probabilities over next
                 # words
                 log_probs = model(context_idxs)

                 # Step 4. Compute your loss function. (Again, Torch wants the target
                 # word wrapped in a tensor)
                 loss = loss_function(log_probs, torch.tensor([word_to_ix[target]], dtype=torch.long))

                 # Step 5. Do the backward pass and update the gradient
                 loss.backward()
                 optimizer.step()

                 # Get the Python number from a 1-element Tensor by calling tensor.item()
                 total_loss += loss.item()
             losses.append(total_loss)
         print(losses)  # The loss decreased every iteration over the training data!

         [1869.737211227417, 1855.8300371170044, 1842.2115116119385, 1828.866439819336, 1815.779625892
         6392, 1802.9469690322876, 1790.370189189911, 1778.0546329021454, 1765.9952700138092, 1754.199
         401140213]
```

Recipe 7-2. CBOW Model in PyTorch

Problem

How do we create a CBOW model using PyTorch?

Solution

There are two different methods to represent words and phrases in vectors: *continuous bag of words* (CBOW) and *skip gram*. The bag-of-words approach learns embedding vectors by predicting the word or phrase in context. Context means the words before and after the current word. If we take a context of size 4, this implies that the four words to the left of the current word and the four words to the right of it are considered for context. The model tries to find those eight words in another sentence to predict the current word.

How It Works

Let's look at the following example.

```
In [37]: CONTEXT_SIZE = 2  # 2 words to the left, 2 to the right
```

```
In [38]: raw_text = """For the future of data science, Donoho projects an ever-growing
         environment for open science where data sets used for academic publications are
         accessible to all researchers.[36] US National Institute of Health has already announced
         plans to enhance reproducibility and transparency of research data.[39] Other big journals
         are likewise following suit.[40][41] This way, the future of data science not only exceeds
         the boundary of statistical theories in scale and methodology, but data science will
         revolutionize current academia and research paradigms.[36] As Donoho concludes, "the scope
         and impact of data science will continue to expand enormously in coming decades as scientific
         data and data about science itself become ubiquitously available."[36]""".split()
```

```
In [39]: # By deriving a set from `raw_text`, we deduplicate the array
         vocab = set(raw_text)
         vocab_size = len(vocab)

         word_to_ix = {word: i for i, word in enumerate(vocab)}
         data = []
         for i in range(2, len(raw_text) - 2):
             context = [raw_text[i - 2], raw_text[i - 1],
                        raw_text[i + 1], raw_text[i + 2]]
             target = raw_text[i]
             data.append((context, target))
         print(data[:5])
```

```
[(['For', 'the', 'of', 'data'], 'future'), (['the', 'future', 'data', 'science,'], 'of'),
(['future', 'of', 'science,', 'Donoho'], 'data'), (['of', 'data', 'Donoho', 'projects'], 'sci
ence,'), (['data', 'science,', 'projects', 'an'], 'Donoho')]
```

```
In [40]: class CBOW(nn.Module):

             def __init__(self):
                 pass

             def forward(self, inputs):
                 pass
         # create your model and train.  here are some functions to help you make
         # the data ready for use by your module
         def make_context_vector(context, word_to_ix):
             idxs = [word_to_ix[w] for w in context]
             return torch.tensor(idxs, dtype=torch.long)

         make_context_vector(data[0][0], word_to_ix)  # example

Out[40]: tensor([48, 28, 70, 40])
```

Graphically, the bag-of-words model looks like what is shown in Figure 7-1. It has three layers: input, which are the embedding vectors that take the words and phrases into account; the output vector, which is the relevant word predicted by the model; and the projection layer, which is a computational layer provided by the neural network model.

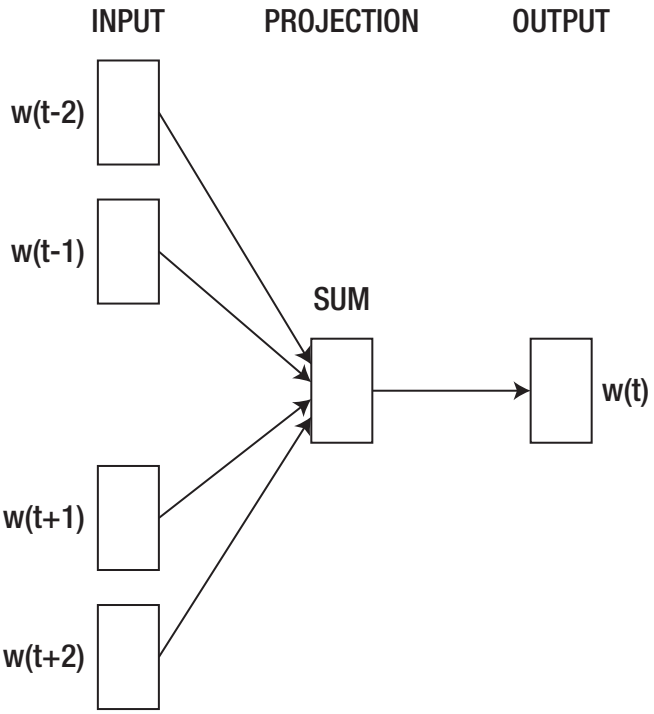


Figure 7-1. *CBOW model representation*

```
In [41]: lin = nn.Linear(5, 3)  # maps from R^5 to R^3, parameters A, b
         # data is 2x5.  A maps from 5 to 3... can we map "data" under A?
         data = torch.randn(2, 5)
         print(lin(data))  # yes

tensor([[-0.3692,  0.5021, -0.2529],
        [-0.1364, -0.1049,  0.1647]], grad_fn=<ThAddmmBackward>)

In [42]: data = torch.randn(2, 2)
         print(data)
         print(F.relu(data))

tensor([[ 0.4833,  0.7295],
        [ 0.4912, -0.4073]])
tensor([[0.4833, 0.7295],
        [0.4912, 0.0000]])

In [43]: # Softmax is also in torch.nn.functional
         data = torch.randn(5)
         print(data)
         print(F.softmax(data, dim=0))
         print(F.softmax(data, dim=0).sum())  # Sums to 1 because it is a distribution!
         print(F.log_softmax(data, dim=0))  # theres also log_softmax

tensor([ 0.2402, -0.5410, -0.4020,  0.5614, -1.6041])
tensor([0.2840, 0.1300, 0.1494, 0.3916, 0.0449])
tensor(1.0000)
tensor([-1.2587, -2.0399, -1.9009, -0.9375, -3.1030])
```

Recipe 7-3. LSTM Model

Problem

How do we create a LSTM model using PyTorch?

Solution

The *long short-term memory* (LSTM) model, also known as the *specific form of recurrent neural network* model, is commonly used in the natural language processing field. Text and sentences come in sequences to make a meaningful sentence, so we need a model that remembers the long and short sequences of text to predict a word or text.

How It Works

Let's look at the following example.

```
In [44]: lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
         inputs = [torch.randn(1, 3) for _ in range(5)]  # make a sequence of length 5

         # initialize the hidden state.
         hidden = (torch.randn(1, 1, 3),
                   torch.randn(1, 1, 3))
         for i in inputs:
             # Step through the sequence one element at a time.
             # after each step, hidden contains the hidden state.
             out, hidden = lstm(i.view(1, 1, -1), hidden)
         inputs = torch.cat(inputs).view(len(inputs), 1, -1)
         hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state
         out, hidden = lstm(inputs, hidden)
         print(out)
         print(hidden)
```

```
tensor([[[ 0.6119,  0.1664, -0.2527]],

        [[ 0.3475,  0.3425, -0.2963]],

        [[ 0.1811,  0.3503, -0.4007]],

        [[ 0.1491,  0.4666, -0.1637]],

        [[ 0.0521,  0.2513, -0.2187]]], grad_fn=<CatBackward>)
(tensor([[[ 0.0521,  0.2513, -0.2187]]], grad_fn=<ViewBackward>), tensor([[[ 0.0788,  0.7995,
-0.3491]]], grad_fn=<ViewBackward>))
```

Prepare a sequence of words as training data to form the LSTM network.

```
In [52]: def prepare_sequence(seq, to_ix):
             idxs = [to_ix[w] for w in seq]
             return torch.tensor(idxs, dtype=torch.long)


         training_data = [
             ("Probability and random variable are integral part of computation ".split(),
              ["DET", "NN", "V", "DET", "NN"]),
             ("Understanding of the probability and associated concepts are essential".split(),
              ["NN", "V", "DET", "NN"])
         ]
```

```
In [53]: training_data
```

```
Out[53]: [(['Probability',
            'and',
            'random',
            'variable',
            'are',
            'integral',
            'part',
            'of',
            'computation'],
           ['DET', 'NN', 'V', 'DET', 'NN']),
          (['Understanding',
            'of',
            'the',
            'probability',
            'and',
            'associated',
            'concepts',
            'are',
            'essential'],
           ['NN', 'V', 'DET', 'NN'])]
```

In [54]:
```
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)
tag_to_ix = {"DET": 0, "NN": 1, "V": 2}

EMBEDDING_DIM = 6
HIDDEN_DIM = 6
```

```
{'Probability': 0, 'and': 1, 'random': 2, 'variable': 3, 'are': 4, 'integral': 5, 'part': 6,
'of': 7, 'computation': 8, 'Understanding': 9, 'the': 10, 'probability': 11, 'associated': 1
2, 'concepts': 13, 'essential': 14}
```

In [46]:
```
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        # Before we've done anything, we dont have any hidden state.
        # Refer to the Pytorch documentation to see exactly
        # why they have this dimensionality.
        # The axes semantics are (num_layers, minibatch_size, hidden_dim)
        return (torch.zeros(1, 1, self.hidden_dim),
                torch.zeros(1, 1, self.hidden_dim))

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(
            embeds.view(len(sentence), 1, -1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

In [55]:
```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
model
loss_function
optimizer
```

Out[55]:
```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.1
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

In [48]:
```
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)
    print(tag_scores)
```

```
tensor([[-1.2305, -1.1328, -0.9526],
        [-1.1960, -1.1592, -0.9575],
        [-1.1702, -1.1421, -0.9928],
        [-1.2641, -1.0862, -0.9675],
        [-1.2992, -1.0806, -0.9471],
        [-1.2178, -1.1114, -0.9808],
        [-1.1838, -1.1092, -1.0104],
        [-1.2382, -1.0778, -0.9949],
        [-1.2360, -1.0898, -0.9858]])
```

In [51]:
```
for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Also, we need to clear out the hidden state of the LSTM,
        # detaching it from its history on the last instance.
        model.hidden = model.init_hidden()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # Step 3. Run our forward pass.
        tag_scores = model(sentence_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(tag_scores, targets)
        loss.backward()
        optimizer.step()
```

In [56]:
```
# See what the scores are after training
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)
    print(tag_scores)
```

```
tensor([[-1.1661, -1.5149, -0.7580],
        [-1.2165, -1.5433, -0.7132],
        [-1.0832, -1.5049, -0.8222],
        [-0.9980, -1.5660, -0.8616],
        [-0.9300, -1.4967, -0.9635],
        [-0.9193, -1.4433, -1.0078],
        [-1.0292, -1.4310, -0.9072],
        [-1.0411, -1.4888, -0.8644],
        [-0.9855, -1.4635, -0.9281]])
```

Index

P. Mishra, *PyTorch Recipes*, https://doi.org/10.1007/978-1-4842-4258-2

D, E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

T

U, V

W, X, Y, Z