PyTorch Artificial Intelligence Fundamentals

A recipe-based approach to design, build and deploy your own AI models with PyTorch 1.x

Jibin Mathew
PyTorch Artificial Intelligence Fundamentals

A recipe-based approach to design, build and deploy your own AI models with PyTorch 1.x

Jibin Mathew
"I can do all things through Christ who strengthens me."

- Philippians 4:13
Contributors

About the author

Jibin Mathew is a senior data scientist and machine learning researcher who has worked in the AI domain for more than 7 years. He is a serial entrepreneur and has founded multiple AI start-ups. He has a strong software engineering background and understands the complete workflow, from research to scalable production deployment. He has built solutions in the fields of healthcare, environment, finance, industrial monitoring, and retail. He has been an adviser to various companies in their AI endeavors. He was the winner of Singularity University’s Global Impact Challenge 2018 and has been part of various global platforms. He is an active contributor to the community and shares his knowledge by authoring content and through blog posts.
About the reviewers

Ganesh Naik is an author, consultant and corporate trainer in the artificial intelligence, data science, machine learning, and embedded Linux product development fields. He is a graduate in computer engineering and has industry experience. He has authored books such as Mastering Python Scripting for System Administrators and Learning Linux Shell Scripting and is the coauthor of Bash Cookbook, Packt Publishing. Ganesh has a passion for teaching. He has trained thousands of engineers in AI, ML, and Linux product development. He has worked as a corporate trainer for corporates such as the Indian Space Research Organisation, Intel, GE, Samsung, Motorola, and various other corporates in India and various other countries.

Rafal Pronko has been a data scientist for 6 years. In his professional life, he has worked on projects such as ad categorization, extracting information from short descriptions; object detection (Smart Shelf and CleanAI projects); and recognition, face, and anti-spoofing recognition mechanisms (the SmartBar anti-spoofing app). Now, he is working for CV Timeline on extracting information from unstructured text such as resumes and social media profiles, categorizing the data, and helping recruiters make better decisions.

Packt is searching for authors like you

If you’re interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.
Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>Chapter 1: Working with Tensors Using PyTorch</strong></td>
<td>6</td>
</tr>
<tr>
<td>Technical requirements</td>
<td>7</td>
</tr>
<tr>
<td>Installing PyTorch</td>
<td>7</td>
</tr>
<tr>
<td>Creating tensors in PyTorch</td>
<td>8</td>
</tr>
<tr>
<td>How to do it...</td>
<td>9</td>
</tr>
<tr>
<td>How it works...</td>
<td>13</td>
</tr>
<tr>
<td>There's more...</td>
<td>14</td>
</tr>
<tr>
<td>See also</td>
<td>14</td>
</tr>
<tr>
<td>Exploring the NumPy bridge</td>
<td>14</td>
</tr>
<tr>
<td>How to do it...</td>
<td>15</td>
</tr>
<tr>
<td>How it works...</td>
<td>16</td>
</tr>
<tr>
<td>There's more...</td>
<td>16</td>
</tr>
<tr>
<td>See also</td>
<td>17</td>
</tr>
<tr>
<td>Exploring gradients</td>
<td>17</td>
</tr>
<tr>
<td>How to do it...</td>
<td>18</td>
</tr>
<tr>
<td>How it works...</td>
<td>20</td>
</tr>
<tr>
<td>There's more...</td>
<td>20</td>
</tr>
<tr>
<td>See also</td>
<td>20</td>
</tr>
<tr>
<td>Viewing tensors in PyTorch</td>
<td>21</td>
</tr>
<tr>
<td>How to do it...</td>
<td>21</td>
</tr>
<tr>
<td>How it works...</td>
<td>22</td>
</tr>
<tr>
<td>There's more...</td>
<td>23</td>
</tr>
<tr>
<td>See also</td>
<td>23</td>
</tr>
<tr>
<td><strong>Chapter 2: Dealing with Neural Networks</strong></td>
<td>24</td>
</tr>
<tr>
<td>Technical requirements</td>
<td>25</td>
</tr>
<tr>
<td>Defining the neural network class</td>
<td>26</td>
</tr>
<tr>
<td>How to do it...</td>
<td>27</td>
</tr>
<tr>
<td>How it works...</td>
<td>28</td>
</tr>
<tr>
<td>There's more...</td>
<td>29</td>
</tr>
<tr>
<td>See also</td>
<td>29</td>
</tr>
<tr>
<td>Creating a fully connected network</td>
<td>30</td>
</tr>
<tr>
<td>How to do it...</td>
<td>30</td>
</tr>
<tr>
<td>How it works...</td>
<td>31</td>
</tr>
<tr>
<td>There's more...</td>
<td>32</td>
</tr>
<tr>
<td>See also</td>
<td>32</td>
</tr>
<tr>
<td>Defining the loss function</td>
<td>32</td>
</tr>
<tr>
<td>How to do it...</td>
<td>33</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>How it works...</td>
<td>34</td>
</tr>
<tr>
<td>There's more...</td>
<td>34</td>
</tr>
<tr>
<td>See also</td>
<td>34</td>
</tr>
<tr>
<td><strong>Implementing optimizers</strong></td>
<td>35</td>
</tr>
<tr>
<td>How to do it...</td>
<td>36</td>
</tr>
<tr>
<td>How it works...</td>
<td>38</td>
</tr>
<tr>
<td>There's more...</td>
<td>38</td>
</tr>
<tr>
<td>See also</td>
<td>38</td>
</tr>
<tr>
<td><strong>Implementing dropouts</strong></td>
<td>39</td>
</tr>
<tr>
<td>How to do it...</td>
<td>39</td>
</tr>
<tr>
<td>How it works...</td>
<td>40</td>
</tr>
<tr>
<td>There's more...</td>
<td>41</td>
</tr>
<tr>
<td>See also</td>
<td>41</td>
</tr>
<tr>
<td><strong>Implementing functional APIs</strong></td>
<td>41</td>
</tr>
<tr>
<td>How to do it...</td>
<td>41</td>
</tr>
<tr>
<td>How it works...</td>
<td>42</td>
</tr>
<tr>
<td>There's more...</td>
<td>42</td>
</tr>
<tr>
<td>See also</td>
<td>42</td>
</tr>
<tr>
<td><strong>Chapter 3: Convolutional Neural Networks for Computer Vision</strong></td>
<td>43</td>
</tr>
<tr>
<td><strong>Technical requirements</strong></td>
<td>44</td>
</tr>
<tr>
<td><strong>Exploring convolutions</strong></td>
<td>45</td>
</tr>
<tr>
<td>How to do it...</td>
<td>46</td>
</tr>
<tr>
<td>How it works...</td>
<td>47</td>
</tr>
<tr>
<td>There's more...</td>
<td>48</td>
</tr>
<tr>
<td>See also</td>
<td>48</td>
</tr>
<tr>
<td><strong>Exploring pooling</strong></td>
<td>48</td>
</tr>
<tr>
<td>How to do it...</td>
<td>49</td>
</tr>
<tr>
<td>How it works...</td>
<td>51</td>
</tr>
<tr>
<td>There's more...</td>
<td>51</td>
</tr>
<tr>
<td>See also</td>
<td>51</td>
</tr>
<tr>
<td><strong>Exploring transforms</strong></td>
<td>51</td>
</tr>
<tr>
<td>How to do it...</td>
<td>51</td>
</tr>
<tr>
<td>How it works...</td>
<td>53</td>
</tr>
<tr>
<td>There's more...</td>
<td>54</td>
</tr>
<tr>
<td>See also</td>
<td>54</td>
</tr>
<tr>
<td><strong>Performing data augmentation</strong></td>
<td>54</td>
</tr>
<tr>
<td>How to do it...</td>
<td>55</td>
</tr>
<tr>
<td>How it works...</td>
<td>56</td>
</tr>
<tr>
<td>There's more...</td>
<td>56</td>
</tr>
<tr>
<td>See also</td>
<td>56</td>
</tr>
<tr>
<td><strong>Loading image data</strong></td>
<td>57</td>
</tr>
<tr>
<td>Getting ready</td>
<td>57</td>
</tr>
<tr>
<td>How to do it...</td>
<td>57</td>
</tr>
<tr>
<td>How it works...</td>
<td>59</td>
</tr>
</tbody>
</table>
# Table of Contents

There's more...  
See also  
**Defining the CNN architecture**  
How to do it...  
How it works...  
There's more...  
See also  
**Training an image classifier**  
How to do it...  
How it works...  
There's more...  
See also

Chapter 4: Recurrent Neural Networks for NLP  
**Introducing RNNs**  
Technical requirements  
**Tokenization**  
How to do it...  
How it works...  
There's more...  
See also  
**Creating fields**  
How to do it...  
How it works...  
There's more...  
See also  
**Developing a dataset**  
Getting ready  
How to do it...  
How it works...  
There's more...  
See also  
**Developing iterators**  
How to do it...  
How it works...  
There's more...  
See also  
**Exploring word embeddings**  
How to do it...  
How it works...  
There's more...  
See also  
**Building an LSTM network**  
How to do it...  
How it works...
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>There's more...</td>
<td>84</td>
</tr>
<tr>
<td>See also</td>
<td>84</td>
</tr>
<tr>
<td><strong>Multilayer LSTMs</strong></td>
<td>84</td>
</tr>
<tr>
<td>How to do it...</td>
<td>84</td>
</tr>
<tr>
<td>How it works...</td>
<td>86</td>
</tr>
<tr>
<td>There's more...</td>
<td>86</td>
</tr>
<tr>
<td>See also</td>
<td>86</td>
</tr>
<tr>
<td><strong>Bidirectional LSTMs</strong></td>
<td>86</td>
</tr>
<tr>
<td>Getting ready</td>
<td>86</td>
</tr>
<tr>
<td>How to do it...</td>
<td>87</td>
</tr>
<tr>
<td>How it works...</td>
<td>88</td>
</tr>
<tr>
<td>There's more...</td>
<td>88</td>
</tr>
<tr>
<td>See also</td>
<td>88</td>
</tr>
<tr>
<td><strong>Chapter 5: Transfer Learning and TensorBoard</strong></td>
<td>89</td>
</tr>
<tr>
<td>Technical requirements</td>
<td>90</td>
</tr>
<tr>
<td>Adapting a pretrained model</td>
<td>90</td>
</tr>
<tr>
<td>Getting ready</td>
<td>90</td>
</tr>
<tr>
<td>How to do it...</td>
<td>91</td>
</tr>
<tr>
<td>How it works...</td>
<td>92</td>
</tr>
<tr>
<td><strong>Implementing model training</strong></td>
<td>92</td>
</tr>
<tr>
<td>How to do it...</td>
<td>92</td>
</tr>
<tr>
<td>How it works...</td>
<td>93</td>
</tr>
<tr>
<td><strong>Implementing model testing</strong></td>
<td>94</td>
</tr>
<tr>
<td>How to do it...</td>
<td>94</td>
</tr>
<tr>
<td>How it works...</td>
<td>95</td>
</tr>
<tr>
<td><strong>Loading the dataset</strong></td>
<td>95</td>
</tr>
<tr>
<td>How to do it...</td>
<td>96</td>
</tr>
<tr>
<td>How it works...</td>
<td>98</td>
</tr>
<tr>
<td><strong>Defining the TensorBoard writer</strong></td>
<td>98</td>
</tr>
<tr>
<td>Getting ready</td>
<td>98</td>
</tr>
<tr>
<td>How to do it...</td>
<td>99</td>
</tr>
<tr>
<td>How it works...</td>
<td>99</td>
</tr>
<tr>
<td><strong>Training the model and unfreezing layers</strong></td>
<td>100</td>
</tr>
<tr>
<td>How to do it...</td>
<td>100</td>
</tr>
<tr>
<td>How it works...</td>
<td>107</td>
</tr>
<tr>
<td>There's more...</td>
<td>107</td>
</tr>
<tr>
<td>See also</td>
<td>107</td>
</tr>
<tr>
<td><strong>Chapter 6: Exploring Generative Adversarial Networks</strong></td>
<td>108</td>
</tr>
<tr>
<td>Technical requirements</td>
<td>110</td>
</tr>
<tr>
<td>Creating a DCGAN generator</td>
<td>110</td>
</tr>
<tr>
<td>How to do it...</td>
<td>111</td>
</tr>
<tr>
<td>How it works...</td>
<td>112</td>
</tr>
<tr>
<td>See also</td>
<td>113</td>
</tr>
<tr>
<td>Creating a DCGAN discriminator</td>
<td>113</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>Getting Ready</td>
<td>114</td>
</tr>
<tr>
<td>How to do it...</td>
<td>114</td>
</tr>
<tr>
<td>How it works...</td>
<td>115</td>
</tr>
<tr>
<td>See also</td>
<td>115</td>
</tr>
<tr>
<td><strong>Training a DCGAN model</strong></td>
<td>116</td>
</tr>
<tr>
<td>Getting Ready</td>
<td>116</td>
</tr>
<tr>
<td>How to do it...</td>
<td>116</td>
</tr>
<tr>
<td>How it works...</td>
<td>122</td>
</tr>
<tr>
<td>There's more...</td>
<td>123</td>
</tr>
<tr>
<td>See also</td>
<td>123</td>
</tr>
<tr>
<td><strong>Visualizing DCGAN results</strong></td>
<td>123</td>
</tr>
<tr>
<td>Getting Ready</td>
<td>123</td>
</tr>
<tr>
<td>How to do it...</td>
<td>124</td>
</tr>
<tr>
<td>How it works...</td>
<td>125</td>
</tr>
<tr>
<td>There's more...</td>
<td>126</td>
</tr>
<tr>
<td>See also</td>
<td>126</td>
</tr>
<tr>
<td><strong>Running PGGAN with PyTorch hub</strong></td>
<td>126</td>
</tr>
<tr>
<td>Getting ready</td>
<td>128</td>
</tr>
<tr>
<td>How to do it...</td>
<td>128</td>
</tr>
<tr>
<td>How it works...</td>
<td>129</td>
</tr>
<tr>
<td>There's more...</td>
<td>129</td>
</tr>
<tr>
<td>See also</td>
<td>129</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7: Deep Reinforcement Learning</th>
<th>130</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introducing deep RL</strong></td>
<td>131</td>
</tr>
<tr>
<td><strong>Introducing OpenAI gym – CartPole</strong></td>
<td>131</td>
</tr>
<tr>
<td>Getting ready</td>
<td>132</td>
</tr>
<tr>
<td>How to do it...</td>
<td>133</td>
</tr>
<tr>
<td>How it works...</td>
<td>134</td>
</tr>
<tr>
<td>There's more...</td>
<td>135</td>
</tr>
<tr>
<td>See also</td>
<td>135</td>
</tr>
<tr>
<td><strong>Introducing DQNs</strong></td>
<td>135</td>
</tr>
<tr>
<td>How to do it...</td>
<td>136</td>
</tr>
<tr>
<td>How it works...</td>
<td>136</td>
</tr>
<tr>
<td>There's more...</td>
<td>136</td>
</tr>
<tr>
<td>See also</td>
<td>137</td>
</tr>
<tr>
<td><strong>Implementing the DQN class</strong></td>
<td>137</td>
</tr>
<tr>
<td>Getting ready</td>
<td>137</td>
</tr>
<tr>
<td>How to do it...</td>
<td>137</td>
</tr>
<tr>
<td>How it works...</td>
<td>139</td>
</tr>
<tr>
<td>There's more...</td>
<td>140</td>
</tr>
<tr>
<td>See also</td>
<td>140</td>
</tr>
<tr>
<td><strong>Training DQN</strong></td>
<td>140</td>
</tr>
<tr>
<td>How to do it...</td>
<td>141</td>
</tr>
</tbody>
</table>
# Table of Contents

How it works... 143  
There's more... 144  
See also 144  
**Introduction to Deep GA** 144  
How to do it... 145  
How it works... 145  
There's more... 145  
See also 145  
**Generating agents** 146  
How to do it... 146  
How it works... 147  
See also 147  
**Selecting agents** 147  
How to do it... 147  
How it works... 148  
**Mutating agents** 149  
How to do it... 149  
How it works... 150  
**Training Deep GA** 151  
How to do it... 151  
How it works... 153  
There's more... 154  
See also 154  
**Chapter 8: Productionizing AI Models in PyTorch** 155  
**Technical requirements** 155  
**Deploying models using Flask** 156  
Getting ready 156  
How to do it... 156  
How it works... 159  
There's more... 160  
See also 160  
**Creating a TorchScript** 161  
How to do it... 161  
How it works... 165  
There's more... 166  
See also 166  
**Exporting to ONNX** 166  
Getting ready 167  
How to do it... 167  
How it works... 169  
There's more... 170  
See also 170  
**Other Books You May Enjoy** 171
<table>
<thead>
<tr>
<th>Index</th>
<th>174</th>
</tr>
</thead>
</table>
Artificial Intelligence (AI) continues to grow in popularity and disrupt a wide range of domains, but it is a complex and daunting topic. In this book, you’ll get to grips with building deep learning apps, and how you can use PyTorch for research and solving real-world problems.

This book uses a recipe-based approach, starting with the basics of tensor manipulation, before covering Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) in PyTorch. Once you are well-versed with these basic networks, you’ll build a medical image classifier using deep learning. Next, you’ll use TensorBoard for visualizations. You’ll also delve into Generative Adversarial Networks (GANs) and Deep Reinforcement Learning (DRL) before finally deploying your models to production at scale. You’ll discover solutions to common problems faced in machine learning, deep learning, and reinforcement learning. You’ll learn to implement AI tasks and tackle real-world problems in computer vision, natural language processing (NLP), and other real-world domains.

By the end of this book, you’ll have the foundations of the most important and widely used techniques in AI using the PyTorch framework.

Who this book is for

This PyTorch book is for AI engineers who are just getting started and machine learning engineers, data scientists, and deep learning enthusiasts who are looking for a guide to help them solve AI problems effectively. Working knowledge of the Python programming language and a basic understanding of machine learning are expected.

What this book covers

Chapter 1, Working with Tensors Using PyTorch, introduces PyTorch and its installation and then jumps on to working with tensors using PyTorch.

Chapter 2, Dealing with Neural Networks, goes through all of the requirements to get started and train a fully connected neural network, providing a thorough explanation of all the components of a basic neural network: layers, feedforward network, backpropagation, loss functions, gradients, weight updates, and using a CPU/GPU.
Chapter 3, *Convolutional Neural Networks for Computer Vision*, starts by looking at a class of neural networks for more advanced tasks, called convolutional neural networks. Here, we will explore TorchVision alongside PyTorch, train a CNN model, and visualize its progress using TensorBoard. We will also cover various tasks related to the building blocks of convolutional networks. A convolutional neural network (CNN or ConvNet) is a class of DNN that is most commonly applied to analyze images.

Chapter 4, *Recurrent Neural Networks for NLP*, explores recurrent neural networks and looks at various modifications within RNNs, as well as best practices.

Chapter 5, *Transfer Learning and TensorBoard*, shows how to train an image classifier to distinguish normal and pneumonia chest X-rays, using a trained ResNet-50 model to perform transfer learning. We will replace the classifier and have two output units to represent the Normal and Pneumonia classes.

Chapter 6, *Exploring Generative Adversarial Networks*, explores generative adversarial networks and how to implement the components of PyTorch and train an end-to-end network. We will explore DCGANs and further improve the limitations of DCGANs with a progressive GAN network.

Chapter 7, *Deep Reinforcement Learning*, helps you to gain an understanding of deep RL with various recipes. This chapter is a series of recipes and tasks where you’ll utilize the abilities and architectures you need to turn into a deep reinforcement learning expert.

Chapter 8, *Productizing AI models in PyTorch*, looks at productizing PyTorch applications in two ways. Firstly, productizing an already trained model, and secondly, performing distributed training on large datasets. Finally, we will look at portability between various frameworks.

**To get the most out of this book**

Working knowledge of Python is required.

**Download the example code files**

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipec/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/PyTorch-Artificial-Intelligence-Fundamentals. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

### Download the color images


### Conventions used

There are a number of text conventions used throughout this book.

**CodeInText**: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "For Linux, we will use the following pip manager."

A block of code is set as follows:

```python
a = np.ones((2, 3))
a
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```python
b.shape
torch.Size([2, 3])
```
Any command-line input or output is written as follows:

```
pip3 install
https://download.pytorch.org/whl/cu90/torch-1.1.0-cp36-cp36m-win_amd64.whl
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "A **scalar** is a single independent value."

**Warnings or important notes** appear like this.

**Tips and tricks** appear like this.

**Sections**

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There’s more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

**Getting ready**

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

**How to do it...**

This section contains the steps required to follow the recipe.

**How it works...**

This section usually consists of a detailed explanation of what happened in the previous section.
There's more…
This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also
This section provides helpful links to other useful information for the recipe.

Get in touch
Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews
Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.
Deep learning is a subfield within the parent field of machine learning, which is the study and application of a class of algorithms inspired by the working of the brain. Given enough data and iteration through it, these algorithms can approximate any function that describes the data, and are rightly called universal function approximators. So where does PyTorch come into this ecosystem?

PyTorch is an open-source deep learning framework in Python that lets us start by examining a problem, come up with a prototype solution, and progress in our development of this solution all the way up to the creation of a distributed compute cluster. It keeps you covered from research to production. PyTorch is adapted from Torch, which is a scientific computing framework with wide support for machine learning algorithms that provides you with a great ability (using GPUs) and is written in Lua. So why PyTorch?

PyTorch is deeply integrated with Python, has an imperative style, uses a Python-like syntax, and is easy to use and flexible in Eager mode. It has a very shallow learning curve and lets you focus on the functionality rather than the boilerplate and syntax of the framework. A pure imperative execution of Python code would miss a lot of optimization opportunities, and so, with the introduction of just-in-time (JIT) compilers, PyTorch allows a transition to graph mode for speed, functionality, and optimization in C++ runtime environments. It has great community support from professionals from different domains, and plays well with libraries. It has native Open Neural Network Exchange (ONNX) support for interoperability with frameworks. It is distributed, scales to production, integrates with TensorBoard, and has great documentation and APIs, and you can easily write custom extensions for CPUs and GPUs. We will explore these and more in upcoming chapters.
In this chapter, we will cover the following recipes:

- Installing PyTorch
- Creating tensors in PyTorch
- Interoperating NumPy bridge
- Gradients and no gradients
- Viewing tensors in PyTorch

**Technical requirements**

To work through this chapter, you need to have Python3 installed. You will also require any modern machine, but you don’t need a GPU-enabled device for this chapter. If you want to leverage GPU capabilities, you can use NVIDIA CUDA-enabled GPUs.

**Installing PyTorch**

We will install PyTorch in this section.

NumPy is an essential library for this chapter and will be automatically installed for you while you install PyTorch as part of its dependency. This means that we need not explicitly install NumPy.

You can install PyTorch with other package managers, such as conda, as described at https://pytorch.org/.

To install PyTorch for Python3 CPU, we can use the following commands:

- For Linux, we will use the following pip manager:

  ```bash
  pip3 install torch==1.4.0+cpu -f https://download.pytorch.org/whl/torch_stable.html
  ```

- For Windows, we will use the following pip manager:

  ```bash
  pip3 install torch==1.4.0+cpu -f https://download.pytorch.org/whl/torch_stable.html
  ```
• For MacOS, we will use the following `pip` manager:

  `pip3 install torch`

To install PyTorch for the Python3 CUDA-enabled GPU version, we can use the following:

• For Linux, we will use the following `pip` manager:

  `pip3 install torch`

• For Windows, we will use the following `pip` manager:

  `pip3 install https://download.pytorch.org.whl/cu90/torch-1.1.0-cp36-cp36m-win_amd64.whl`

MacOS Binaries don't support CUDA, so you should install it from the source if you need CUDA. You can alternatively install it using other package managers and even build it from the source. For other package managers and Python versions, visit https://pytorch.org/.

You can quickly verify the installation works OK by going to the Python terminal and typing in the following commands:

```python
import torch
import numpy
```

If these imports worked fine, you are good to go!

## Creating tensors in PyTorch

Let's first understand what a tensor is. A **scalar** is a single independent value, a 1D array of values is called a **vector**, a 2D array of values is called a **matrix**, and any array of values that is more than 2D is simply called a **tensor**. A tensor is a generalized term that encompasses scalars, vectors, and matrices.

A scalar is a $0^\text{th}$ order tensor, a vector is a $1^\text{st}$ order tensor and a matrix is a $2^\text{nd}$ order tensor.
The following are the various tensors:

- **Scalar**: This is a zeroth-order tensor. An example of a scalar is $x_1$.
- **Vector**: This is a first-order tensor; the following is an example of a vector:
  
  \[
  \begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3
  \end{bmatrix}
  \]

- **Matrix**: This is a second-order tensor. The following is an example of a matrix:
  
  \[
  \begin{bmatrix}
  x_1 & x_2 \\
  x_3 & x_4
  \end{bmatrix}
  \]

- **Tensors**: These are anything above a second-order tensor, as shown by the following example:
  
  \[
  \begin{bmatrix}
  x_1 & x_2 & x_1 & x_2 \\
  x_3 & x_4 & x_3 & x_4 \\
  x_1 & x_2 & x_1 & x_2 \\
  x_3 & x_4 & x_3 & x_4
  \end{bmatrix}
  \]

With this, we will move on to our recipe showing how to work with tensors.

**How to do it...**

There are multiple ways to create a tensor in PyTorch. We will look at a few of them in this section:

- We can create a tensor with all ones as follows:

  1. Let's start by importing the library:

     ```
     import torch
     ```

  2. We will use the `ones()` method:

     ```
     torch.ones((2,3))
     ```
This will return a tensor that contains ones and has a default float datatype as follows:

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

- Now, we will create a tensor consisting of only integer ones:
  1. We will do exactly the same as in the previous recipe, but we will add `datatype (dtype)` as a parameter:
    ```
torch.ones((2,3), dtype=torch.int8)
    ``
  2. This will return a tensor consisting only of integer ones:
    ```
tensor([[1, 1, 1],
            [1, 1, 1]], dtype=torch.int8)
```
- Next, we will create a tensor consisting of only integer zeros:
  1. We will do exactly the same as before, but using the `zeros()` method:
    ```
torch.zeros((2,3), dtype=torch.int8)
    ``
  2. This will return a tensor consisting of only integer zeros:
    ```
tensor([[0, 0, 0],
            [0, 0, 0]], dtype=torch.int8)
```
- We will now create a tensor filled with a specific value:
  1. We will use the `full()` method and pass in the required fill value along with the shape:
    ```
torch.full((2, 3), 3.141592)
    ``
  2. This will return a tensor with the given value:
    ```
tensor([[3.1416, 3.1416, 3.1416],
            [3.1416, 3.1416, 3.1416]])
```

Note that the values are rounded off.
• Now, we will create an empty tensor:
  
  1. We will use the `empty()` method for this:
     
     ```python
     torch.empty((2,3))
     ```

  2. This will return a tensor filled with uninitialized data, varying each time and for each machine:
     
     ```
     tensor([[2.5620e-01, 4.5773e-41, 2.5620e-01],
             [4.5773e-41, 4.4842e-44, 0.0000e+00]])
     ```

• We will next create a tensor from a uniform distribution:
  
  1. We will use the `rand()` method:
     
     ```python
     torch.rand((2,3))
     ```

  2. This will draw a tensor with random values from a uniform distribution from [0, 1]:
     
     ```
     tensor([[0.6714, 0.0930, 0.4395],
             [0.5943, 0.6582, 0.6573]])
     ```

• We will create a tensor with mean 0 and variance 1:
  
  1. We will use the `randn()` method:
     
     ```python
     torch.randn((2,3))
     ```

  2. This will draw a tensor with random values with mean 0 and variance 1 from a normal distribution, also called the **standard normal distribution**:
     
     ```
     tensor([[ 0.3470, -0.4741, 1.2870],
             [ 0.8544, 0.9717, -0.2017]])
     ```

• Next, we will create a tensor from a given range of values
  
  1. We will use the `rand_int()` method, passing in the lower limit, the upper limit, and the shape:
     
     ```python
     torch.randint(10, 100, (2,3))
     ```

  2. This will return a tensor between 10 and 100, similar to the following:
     
     ```
     tensor([[63, 93, 68],
             [93, 58, 29]])
     ```
• We will now create a tensor from existing data:

1. We will use the `torch.tensor` class for this:

```python
torch.tensor([[1, 2, 3], [4, 5, 6]])
```

This will create a copy of the data and create a tensor. If you want to avoid making a copy, you could use `torch.as_tensor([[1, 2, 3], [4, 5, 6]])`.

2. This returns a tensor with the same datatype as that of the data, which in this case is an integer tensor:

```python
tensor([[1, 2, 3],
        [4, 5, 6]])
```

Also, note that, if one of the data values is a float, then all of the values would be converted into a float; however, if one of the values is a string, then an error is thrown.

• Next we will create a tensor with the attributes from another tensor:

1. Let’s first create a reference tensor for this:

```python
a = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

2. Let’s see the datatype of tensor `a`:

```python
a.dtype
torch.int64
```

3. Now let’s look at the shape of the tensor:

```python
a.shape
torch.Size([2, 3])
```

4. The datatype and shape meet our expectation, so now let’s create a tensor `b` so that it matches the attributes of `a` and use the `torch.*_like` format for this:

```python
b = torch.ones_like(a)
b
```
This results in the following output:

```python
tensor([[1, 1, 1],
        [1, 1, 1]])
```

5. Let's see the datatype of tensor b:

```python
b.dtype
```
```
torch.int64
```

6. Let's also look at the shape of tensor b:

```python
b.shape
```
```
torch.Size([2, 3])
```

- Next, we will create a tensor with a similar type to another tensor, but of a different size:

1. We will use the same tensor a, from the previous step and use the `torch.new_*` format for this:

    ```python
    a.new_full((2, 2), 3.)
    ```

2. This returns the following output:

    ```python
tensor([[3, 3],
            [3, 3]])
    ```

These are the different methods to create tensors in PyTorch.

How it works...

In this recipe, we had a look at the various methods for creating tensors from various data sources. Before we start exploring the concept of deep learning with PyTorch and how it works, it is essential to understand some of the most commonly used functionalities for dealing with the basic unit of data, tensors. We can use the `torch.tensor()` method to create tensors with various kinds of values and shapes. We could even draw a tensor from a uniform distribution or standard normal distribution, which are essential in initializing a neural network for optimal performance and training time, and all these tensors have a default `torch.FloatTensor` datatype and update the datatype using the `dtype` parameter.
The `.ones()` method creates a tensor containing 1 in the given shape, `.zeros()` fills the tensors with all zeros, and the `full()` method fills the tensors with the given value. The `.empty()` method creates an empty tensor, `.rand()` draws a tensor with random values from a uniform distribution from [0, 1), and `.randn()` draws a tensor with random values with mean 0 and variance 1 from a normal distribution, also called the standard normal distribution.

The `rand_int()` method draws random integers from a given range and creates a tensor in a given shape. We can create tensors with the shape of another tensor, a tensor with all ones, but the shapes and datatype of another tensor can be created using the `ones_like()` method. We can use the `torch.new_*` format to create a tensor with a similar type to another tensor, but a different size.

We can also take data from an existing source and convert it into tensors, and there are advanced tensor creation techniques that reduce the memory footprint and use the shape of an existing tensor and/or the datatype of a tensor.

There's more...

You can find the shape of a tensor using the `shape` attribute or `size()` method and the datatype using the `dtype` attributes of a tensor. You can also use `torch.numel()` to get the total number of elements in a tensor.

See also


Exploring the NumPy bridge

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object and various derived objects. Beyond this, NumPy is used as an efficient container for generic multidimensional data. NumPy allows for seamless and speedy integration with a wide variety of databases.
NumPy is the standard Python library and is used to deal with numerical data. Many well-known ML/DS libraries in Python, such as pandas (a library that is used to read data from many sources) and scikit-learn (one of the most important ML libraries, used to read and write images) use NumPy under the hood. You will deal with numpy a lot, for example, while dealing with tabular data, loading it using the pandas library and getting numpy arrays out of the dataframe; reading images, where many existing libraries have in-built APIs for reading them as numpy arrays; and also converting numpy arrays into images, as well as text and other forms of data. Also, these all support numpy arrays using scikit-learn, which is a machine learning library. As you can see, it is important to have a bridge between numpy arrays and PyTorch tensors.

How to do it...

Let’s start by importing numpy:

1. We will start by creating a numpy array; for this, let's import numpy:

   ```python
   import numpy as np
   ```

2. We will create a numpy array consisting of only ones:

   ```python
   a = np.ones((2, 3))
   ```

   This results in the following output:

   ```output
   array([[1., 1., 1.],
          [1., 1., 1.]])
   ```

3. Now we will convert it into a PyTorch tensor:

   ```python
   b = torch.from_numpy(a)
   ```

   This results in the following output:

   ```output
   tensor([[1., 1., 1.],
            [1., 1., 1.]], dtype=torch.float64)
   ```
4. Now we will convert a tensor to a `numpy` array:

   ```python
   b.numpy()
   ```

   This results in the following output:

   ```
   array([[1., 1., 1.],
         [1., 1., 1.]])
   ```

   With this recipe, we've now got the hang of moving back and forth between NumPy and Torch tensors.

**How it works...**

We started by importing `numpy` to create a `numpy` array. Then, we created a `numpy` array consisting of only ones using `np.ones()`, which converted it into a PyTorch tensor using the `from_numpy()` method. Then we converted the tensor to a `numpy` array using the `.numpy()` method.

It is extremely easy to switch between a PyTorch tensor and NumPy; in fact, it can be achieved with just two methods. This makes it possible to take a predicted tensor and convert into an image from NumPy (using a library that supports NumPy-to-image conversion) and similarly back from NumPy to a tensor.

**There's more...**

The underlying memory is shared between a NumPy array and a PyTorch tensor, and hence any change made in one would affect the other.

Let's have a look at how this is presented in the following code block:

```python
>>a
array([[1., 1., 1.],
       [1., 1., 1.]])

>>b = torch.from_numpy(a)
>>b
tensor([[1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)

>>a*=2
>>a
array([[2., 2., 2.],
       [2., 2., 2.]])
```
[2., 2., 2.])

>> b = tensor([[2., 2., 2.], [2., 2., 2.]], dtype=torch.float64)

We can see that the changes from the numpy are reflected in the tensor as well.

**See also**

To learn more, click on PyTorch's official documentation link for the NumPy bridge at https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#numpy-bridge.

**Exploring gradients**

Let's briefly go through what gradients are. For this, we need to first understand what a gradient descent is. In machine learning problems, we provide an input and desired output pair and ask our model to generalize the relationship between the given input and output pair. But sometimes the model learns that its predictions would be way off from the desired output (this difference is known as a loss). So what is gradient descent?

**Gradient descent** is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. We use it while training a model so that it minimizes the loss. It is used to find the values of a function's parameters (coefficients or weights in machine learning) that minimize the cost or loss function.

So what is a gradient? A gradient measures how much the output of the given function varies when varying the inputs by a small factor, which is the same as the concept of derivatives in calculus. A gradient calculates the variation in all weights with respect to the change in error. Gradients are the slope of a function. A higher gradient means a steeper slope and that a model can learn more rapidly. The gradient points toward the direction of steepest slope. The Autograd module in PyTorch performs all gradient calculations in PyTorch. It is the core Torch package for automatic differentiation. It uses a tape-based system for automatic differentiation. In the forward phase, the Autograd tape will remember all the operations it executed, and in the backward phase it will replay them.
How to do it...

Let's start by creating a tensor.

1. Unlike the tensors that we have created so far, we will add a new key that lets PyTorch know that it needs to perform gradient calculations on the following tensor:

   ```python
   x = torch.full((2,3), 4, requires_grad=True)
   x
   ```

   This results in the following output:

   ```
   tensor([[4., 4., 4.],
           [4., 4., 4.]], requires_grad=True)
   ```

2. Let's create another tensor, y, that is derived out of tensor a; we will see the difference in the output of this new tensor, as it has a gradient function attached to it:

   ```python
   y = 2*x+3
   y
   ```

   This results in the following output:

   ```
   tensor([[11., 11., 11.],
           [11., 11., 11.]], grad_fn=<AddBackward0>)
   ```

3. Let's further explore gradients in PyTorch, starting with the original x:

   ```python
   x
   ```

   This results in the following output:

   ```
   tensor([[4., 4., 4.],
           [4., 4., 4.]], requires_grad=True)
   ```

4. We will then define y, a slightly more complex tensor than the previous example:

   ```python
   y = (2*x**2+3)
   y
   ```

   This results in the following output:

   ```
   tensor([[35., 35., 35.],
           [35., 35., 35.]], grad_fn=<AddBackward0>)
   ```
5. Next, we will calculate gradients with respect to $x$ on $y$, since $y$ is a tensor, and we want to calculate the gradient with respect to this tensor. To do this, we will pass the shape of $x$, which is the same as $y$:

   ```
y.backward(torch.ones_like(x))
   ```

6. Now, let's see the value of the gradient of $x$ using the `grad` attribute:

   ```
x.grad
   ```

This results in the following output:

   ```
tensor([[16., 16., 16.],
   [16., 16., 16.]])
   ```

7. Moving on to the no-gradient part of this section, we can turn off the gradient calculation at a certain point in the code by going through the following steps, first using the `requires_grad_()` method on the tensor, if we revisit tensor $x$:

   ```
>> x.requires_grad
True
>> x.requires_grad_(False) # turning off gradient
>> x.requires_grad
False
   ```

8. We can turn off tracking the gradient calculation by using the `.no_grad()` method, starting with $x$:

   ```
>> x = torch.full((2,3), 4,requires_grad=True)
>> x

tensor([[4., 4., 4.],
   [4., 4., 4.]], requires_grad=True)
>> x.requires_grad
True
>> with torch.no_grad():
...   print((x**5+3).requires_grad)
False
   ```

With this, we have explored some functionalities of the `Autograd` package.
How it works...

We can see that Autograd keeps track of operations; when we create the tensor $y$ from $x$, $y=2x+3$, we see that a gradient function, $\text{grad}_\text{fn}$, is attached to the tensor.

We started by creating a new type of tensor that has `require_grad` set to `True`, after which we created a tensor $y$, such that, $y = 2x^2 + 3$ and discovered that $y$ has a different gradient function attached to it. We also looked at using `requires_grad_()`, and finally `no_grad()`.

PyTorch has a package called `autograd` that performs all the tracking and automatic differentiation for all operations on tensors. It is a define-by-run framework, which means that your backpropagation is defined by how your code is run and that every single iteration can be different. We utilized the `require_grad` attribute of the `torch.Tensor` class to determine the state of the gradient calculation and, upon calling the `.backward()` method, automatically computed all of the gradients and the gradient of the tensor in its `.grad` attribute.

We can disable the gradient calculation between the code and also temporarily disable the tracking of tensors for the gradient calculation, increasing the speed of computation. This disabling of the calculation is mostly used during evaluation.

There's more...

You can use the `torch.set_grad_enabled()` method to enable and disable gradient calculation and the `detach()` method for the future tracking of computations. Use the `grad_fn` attribute to see the gradient function attached to the tensor.

See also

Viewing tensors in PyTorch

While working with tensors and dealing with neural networks, we often need to go through and rearrange data in the tensors so that the dimensions of the tensors fit the needs of the architecture. In this section, we will explore common rearrangement and reshaping techniques in PyTorch.

In this recipe, we will learn about getting a tensor to look the way we want.

How to do it...

Let’s look at how to change the shape of tensors:

1. First, we will create a tensor, a:

   ```
   >>> a = torch.Tensor([1, 2, 3, 4])
   ```

2. We will then use the `reshape()` method:

   ```
   >>> torch.reshape(a, (2, 2))
   ```

   This results in the following output:

   ```
   tensor([[1., 2.],
           [3., 4.]])
   ```

3. Next, we will look at the `resize_()` method:

   ```
   >>> a = torch.Tensor([1, 2, 3, 4, 5, 6])
   >>> a.shape
   torch.Size([6])
   >>> a.resize_((2, 2))
   ```

   This results in the following output:

   ```
   tensor([[1., 2.],
           [3., 4.]])
   ```
4. The most common method is `view()`:

```python
>> a = torch.Tensor([1, 2, 3, 4, 5, 6])
>> a.view((2, 3))
```

This results in the following output:

```python
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

5. With the `view()` method, you can choose not to mention one of the dimensions and arrange the rest of them, and PyTorch will calculate the missing dimension as follows:

```python
>> a.view((2, -1))
```

This results in the following output:

```python
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

These are the different ways to reshape tensors.

**How it works...**

In the previous recipe, we manipulated tensors to change their shape based on the network architecture, looking at three different methods, each applying to a different use case:

- **The `.reshape()` method**: `.reshape(a, b)` returns a new tensor with the same data as the original tensor with size `(a, b)` as it copies the data to another part of memory; `.reshape()` can operate on both contiguous and noncontiguous tensors, and may return a copy or a view of the original tensor.

- **The `.resize()` method**: `.resize_(a, b)` returns the same tensor without creating a copy with the new given shape. But we should keep in mind that, if the new shape results in fewer elements than the original tensor, then it won't throw any error, and some elements will be removed from the tensor but not from memory. If the new shape results in more elements than the original tensor, new elements will be uninitialized in memory without throwing any error.

- **The `.view()` method**: `.view(a, b)` will return a new tensor with the same data as weights with size `(a, b)`. `.view()` can only operate on a contiguous tensor and returns the same storage as the input.
There's more...

You can use the dimension of another tensor and make a given tensor resemble the dimension of that tensor without affecting the actual dimensions of either of them.

Look at the following code block:

```python
g>>a = torch.Tensor([[1, 2, 3],
                      [4, 5, 6]])
g>>a
tensor([[1., 2., 3.],
        [4., 5., 6.]])

>>b = torch.Tensor([4,5,6,7,8,9])
g>>b
tensor([4., 5., 6., 7., 8., 9.])

>>b.view_as(a)
tensor([[4., 5., 6.],
        [7., 8., 9.]])
```

From this, we can see that the tensor \( b \) takes the shape of tensor \( a \).

See also