Gaurav Singhal

Image Classification with PyTorch

Gaurav Singhal

Apr 1, 2020 · 19 Min read · 44,648 Views

Apr 1, 2020 · 19 Min read · 44,648 Views

Data Pytorch

Introduction

184

- Introduction
- Importing Library and Data
- Image Pre-processing
- Normalization
- Splitting the Dataset
- Designing a Convolution Neural Network (CNN)
- Convolution Layer
- Pooling Layer
- Activation Layer
- Loss
- Optimization
- Conclusion
- Top ^

Introduction

PyTorch has revolutionized the approach to computer vision or NLP problems. It's a dynamic deep-learning framework, which makes it easy to learn and use.

In this guide, we will build an image classification model from start to finish, beginning with exploratory data analysis (EDA), which will help you understand the shape of an image and the distribution of classes. You'll learn to prepare data for optimum modeling results and then build a convolutional neural network (CNN) that will classify images according to whether they contain a cactus or not.

Click here to download the aerial cactus dataset from an ongoing Kaggle competition. Instead of MNIST B/W images, this dataset contains RGB image channels. Hence, it is perfect for beginners to use to explore and play with CNN. It's also a chance to classify something other than cats and dogs.

Importing Library and Data

To begin, import the `torch` and `torchvision` frameworks and their libraries with `numpy`, `pandas`, and `sklearn`. Libraries and functions used in the code below include:

- `transforms`, for basic image transformations
- `torch.nn.functional`, which contains useful activation functions
- `Dataset` and `Dataloader`, PyTorch's data loading utility

python

```python
import pandas as pd
import matplotlib.pyplot as plt
```

```
import torchvision
import torchvision.transforms as transforms

from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split

%matplotlib inline
```

python

```
import os
os.getcwd()
# place the files in your IDE working dicrectory .
labels = pd.read_csv(r'/aerialcactus/train.csv')
submission = pd.read_csv(r'/aerialcactus/sample_submission.csv)

train_path = r'/aerialcactus/train/train/'
test_path = r'/aerialcactus/test/test/'
```

python

```
labels.head()
```

	id	has_cactus
0	0004be2cfeaba1c0361d39e2b000257b.jpg	1
1	000c8a36845c0208e833c79c1bffedd1.jpg	1
2	000d1e9a533f62e55c289303b072733d.jpg	1
3	0011485b40695e9138e92d0b3fb55128.jpg	1
4	0014d7a11e90b62848904c1418fc8cf2.jpg	1

python

```
labels.tail()
```

	id	has_cactus
17495	ffede47a74e47a5930f81c0b6896479e.jpg	0
17496	ffef6382a50d23251d4bc05519c91037.jpg	1
17497	fff059ecc91b30be5745e8b81111dc7b.jpg	1
17498	fff43acb3b7a23edcc4ae937be2b7522.jpg	0
17499	fffd9e9b990eba07c836745d8aef1a3a.jpg	1

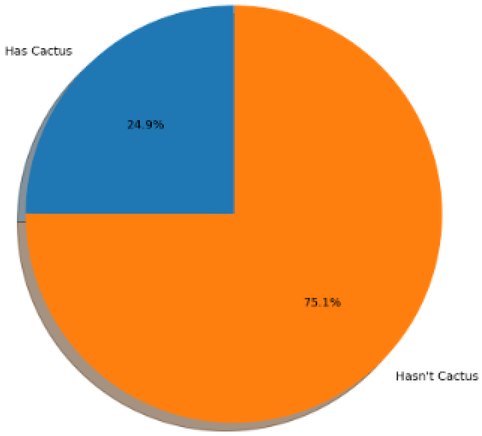
python

```
labels['has_cactus'].value_counts()
```

```
1    13136
0     4364
Name: has_cactus, dtype: int64
```

python

```
label = 'Has Cactus', 'Hasn\'t Cactus'
plt.figure(figsize = (8,8))
plt.pie(labels.groupby('has_cactus').size(), labels = label, autopct='%
plt.show()
```

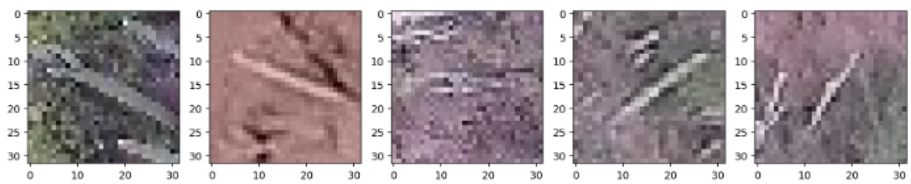


As per the pie chart, the data is biased towards one class. Imbalanced data will affect the final results. We already have enough data for CNN to produce results, so there is no need for any data sampling or augmentation.

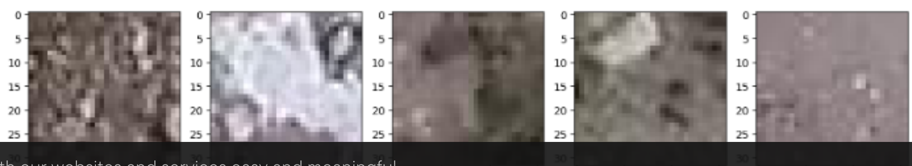
Image Pre-processing

Images in a dataset do not usually have the same pixel intensity and dimensions. In this section, you will pre-process the dataset by standardizing the pixel values. The next required process is transforming raw images into tensors so that the algorithm can process them.

python
```python
import matplotlib.image as img
fig,ax = plt.subplots(1,5,figsize = (15,3))

for i,idx in enumerate(labels[labels['has_cactus'] == 1]['id'][-5:]):
    path = os.path.join(train_path,idx)
    ax[i].imshow(img.imread(path))
```



python
```python
fig,ax = plt.subplots(1,5,figsize = (15,3))
for i,idx in enumerate(labels[labels['has_cactus'] == 0]['id'][:5]):
    path = os.path.join(train_path,idx)
    ax[i].imshow(img.imread(path))
```



Use the below code to standardize the image by defined mean and standard deviation because using raw image data will not give the desired results.

python

```python
import numpy as np
import matplotlib.pyplot as plt

def imshow(image, ax=None, title=None, normalize=True):
    if ax is None:
        fig, ax = plt.subplots()
    image = image.numpy().transpose((1, 2, 0))

    if normalize:
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        image = std * image + mean
        image = np.clip(image, 0, 1)

    ax.imshow(image)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.tick_params(axis='both', length=0)
    ax.set_xticklabels('')
    ax.set_yticklabels('')

    return ax
```

python

```python
class CactiDataset(Dataset):
    def __init__(self, data, path , transform = None):
        super().__init__()
        self.data = data.values
        self.path = path
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self,index):
        img_name,label = self.data[index]
        img_path = os.path.join(self.path, img_name)
        image = img.imread(img_path)
        if self.transform is not None:
            image = self.transform(image)
        return image, label
```

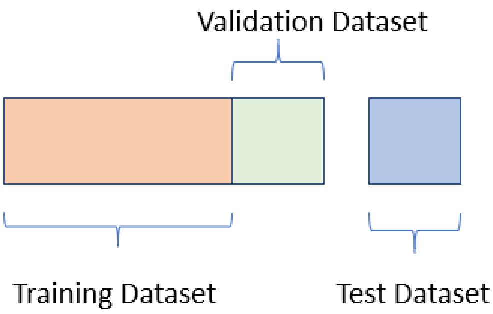
Normalization

You can stack multiple image transformation commands in `transform.Compose`. Normalizing an image is an important step that makes model training stable and fast. In `tranforms.Normalize()` class, a list of means and standard deviations is sent in the form of a list. It uses this formula: x= (x-mean)/std, for x=0 and 1

python
```
train_transform = transforms.Compose([transforms.ToPILImage(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(means,std)])

test_transform = transforms.Compose([transforms.ToPILImage(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(means,std)])

valid_transform = transforms.Compose([transforms.ToPILImage(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(means,std)])
```

Splitting the Dataset

How well the model can learn depends on the variety and volume of the data. We need to divide our data into a training set and a validation set using `train_test_split`.



Training dataset: The model learns from this dataset's examples. It fits a parameter to a classifier.

Validation dataset: The examples in the validation dataset are used to tune the hyperparameters, such as learning rate and epochs. The aim of creating a validation set is to avoid large overfitting of the model. It is a checkpoint to know if the model is fitted well with the training dataset.

Test dataset: This dataset test the final evolution of the model, measuring how well it has learned and predicted the desired output. It contains unseen, real-life data.

python
```
train, valid_data = train_test_split(labels, stratify=labels.has_cactus
```

python
```
train_data = CactiDataset(train, train_path, train_transform )
valid_data = CactiDataset(valid_data, train_path, valid_transform )
test_data = CactiDataset(submission, test_path, test_transform )
```

Define the values of hyperparameters.

```python
# Hyper parameters

num_epochs = 35
num_classes = 2
batch_size = 25
learning_rate = 0.001
```

Whenever you initialize the batch of images, it is on the CPU for computation by default. The function `torch.cuda.is_available()` will check whether a GPU is present. If CUDA is present, `.device("cuda")` will route the tensor to the GPU for computation.

```python
# CPU or GPU

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda', index=0)
```

The device will use CUDA with a single GPU processor. This will make our calculations faster. If you have a CPU in your system, no problem. You can use Google Colab, which provides free GPU.

In the code below, `dataloader` ombines a dataset and a sampler and provides an iterable over the given dataset. `dataset()` indicates which dataset to load form the available data. For details, read this documentation.

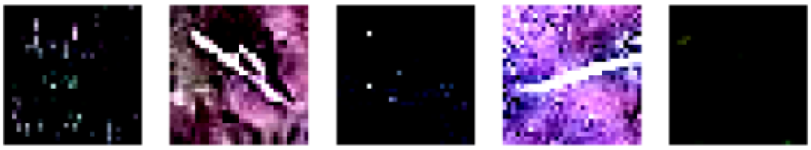
```python
train_loader = DataLoader(dataset = train_data, batch_size = batch_size
valid_loader = DataLoader(dataset = valid_data, batch_size = batch_size
test_loader = DataLoader(dataset = test_data, batch_size = batch_size,
```

```python
import numpy as np
import matplotlib.pyplot as plt

def imshow(image, ax=None, title=None, normalize=True):
    if ax is None:
        fig, ax = plt.subplots()
    image = image.numpy().transpose((1, 2, 0))

    if normalize:
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        image = std * image + mean
        image = np.clip(image, 0, 1)

    ax.imshow(image)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
```

```
    ax.set_xticklabels('')
    ax.set_yticklabels('')

    return ax
```

```python
trainimages, trainlabels = next(iter(train_loader))

fig, axes = plt.subplots(figsize=(12, 12), ncols=5)
print('training images')
for i in range(5):
    axe1 = axes[i]
    imshow(trainimages[i], ax=axe1, normalize=False)

print(trainimages[0].size())
```

```
training images
torch.Size([3, 32, 32])
```



The next step is to make a CNN model that learns ffrom the manipulated training dataset.

Designing a Convolution Neural Network (CNN)

If you try to recognize objects in a given image, you notice features like color, shape, and size that help you identify objects in images. The same technique is used by a CNN. The two main layers in a CNN are the convolution and pooling layer, where the model makes a note of the features in the image, and the fully connected (FC) layer, where classification takes place.

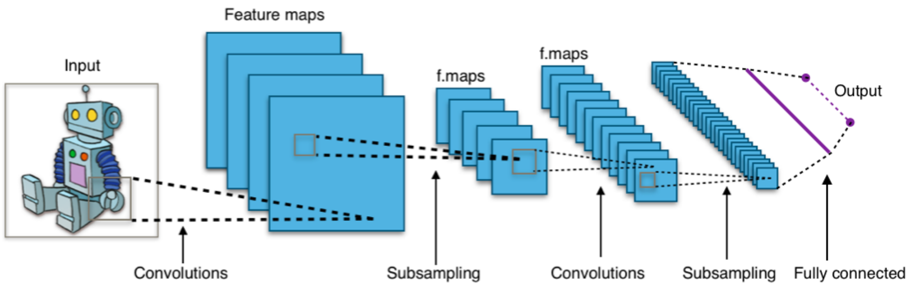


Image Source: https://commons.wikimedia.org/wiki/File:Typical_cnn.png

Convolution Layer

Mathematically, convolution is an operation performed on two functions to produce a third function. Convolution is operating in speech processing (1 dimension), image processing (2 dimensions), and video processing (3 dimensions). The convolution layer forms a thick filter on the image.

The convolutional layer's output shape is affected by the choice of kernel size, input dimensions, padding, and strides (number of pixels by which the window moves).

In this model, a 3x3 kernel size is used. It will have 27 weights and 1 bias.

$$\text{Weights} = \text{W x H x in\_channels} = 3\text{x}3\text{x}3 = 27$$

This is what happens behind the CNN.

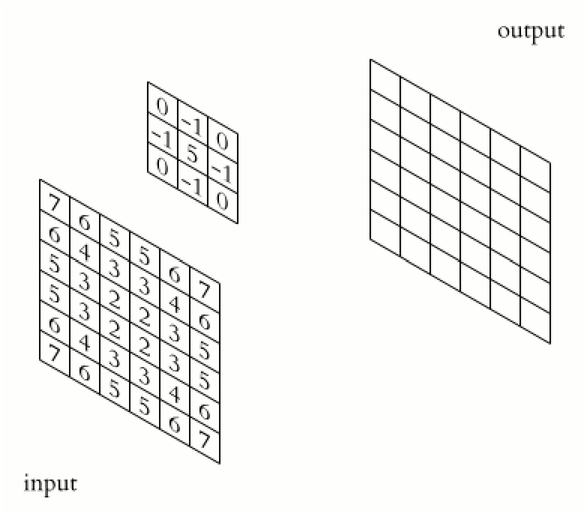


Image Source:
https://upload.wikimedia.org/wikipedia/commons/4/4f/3D_Convolution_Animation.gif

The factors that affect the convolutional layer's output shape are the kernel size, input dimensions, padding and strides (no.of pixel by which the window moves). In this model 3x3 kernel filter is used. It will have 27 weights and 1 bias.

$$W' = \frac{W - F + 2P}{S} + 1$$

W= *Width of Input Image*
F= *Kernel Filter Window Size*
P =*Padding*
S= *Stride*

For, Conv2d, layer 1

Input size = [32x32x3], Filter =3

$$W^{(1)} = \frac{32-3+2(0)}{1} + 1 = 30$$

After applying maxpool layer, S=2

$$W^{(2)} = \frac{32-3+2(0)}{2} + 1 = \frac{30}{2} = 15$$

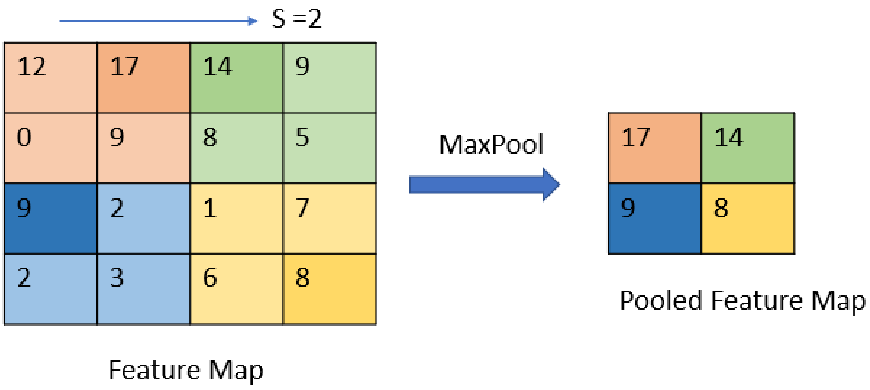
Output size = [15x15x10]

(Output size act as input size for layer 2)

Similarly, carry out the calculation of layer 2.

Pooling Layer

A drawback of a convolution feature map is that it records the exact position of features. Even the smallest development in the feature map will produce different results. This problem is solved by down sampling the feature map. It will be a lower version of the image with important features intact. In this model, max pooling is used. It calculates the maximum value of each patch of the feature map.

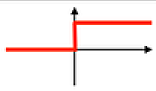
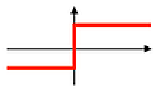
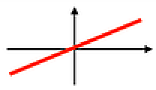
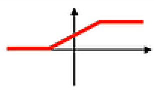
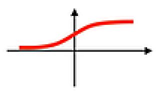
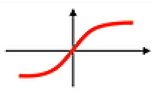
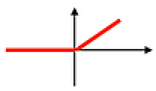
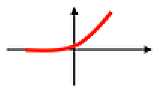


Some brief notes about important parameters of `__init__` model and `forward` are stated below:

Parameters	Use
`_init_()`	**Initializing the layers you want to use.**
`in_channels= 3`	Indicates the colour channels of the input image. B&W= 1 and RGB= 3
`kernel_size= 3`	Size of the filter window (2-d matrix) to apply to the input image. It specifies the height and width of the convolution window.
`out_channels= 10`	It is the number of feature map we want for our convolution layer (how many filters to use?)
`nn.Dropout2d()`	It will prevent overfitting. Parameters of image will be 0 out with the probability to be zeroed is **p**. It is required only during the training mode. It will shut down as soon as the model goes into evaluation mode
`nn.Linear()`	Fully connected layer. According to the documentation, it applies the linear transformation to the input data y $= xW^T + b$ (x= in_feature, y= out_features, b= bias)
`forward()`	**Reuse the same layer for each forward pass, specify the connection of your layer.**
`max_pool2d()`	Selecting maximum element from the feature/activation map. By using stride=2, it compresses the output into half.
`F.relu()`	Adds non-linearity activation function ReLU element wise in the desired convolution layer.
`F.Dropout()`	Equivalent in terms of applying dropout. It is default with **training = False,** you have to set *functional dropout* to **training= True** for an element to be 0.

Activation Layer

During forward propagation, ***activation function*** is used on each layer. The ***non-linearity transformation*** is introduced by the activation function. A neural network without an activation function is just a linear regression model, so it can not be ignored. Below is a list of activation functions.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus 	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Putting it All Together...

python

```
epochs = 35
batch_size = 25
learning_rate = 0.001
```

python

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel
        self.conv2 = nn.Conv2d(10, 20, kernel_size=3)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(720, 1024)
        self.fc2 = nn.Linear(1024, 2)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(x.shape[0],-1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

Create a complete CNN.

python

```
model = CNN()
print(model)
```

```
CNN(
  (conv1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=720, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=2, bias=True)
)
```

Loss

There are different types of losses implemented in machine learning. In this guide, ***cross-entropy*** loss is used. In this context, it is also known as ***log loss***. Notice it has the same formula as that of likelihood, but it contains a log value.

$$logloss_{(\mathrm{N=1})} = y\log(p) + (1-y)\log(1-p)$$

The best thing about this function is that if the prediction is 0, the first half goes away, and if the prediction is 1, the second half drops. With this, you can estimate of where your model can go wrong while predicting the label. Changes are to be made during training to minimize the loss.

Optimization

Select any one optimizer algorithm available in the `torch.optim` package. The optimizers have some elements of the gradient descent. By changing the model parameters, like weights, and adding bias, the model can be optimized. The learning rate will decide how big the steps should be to change the parameters.

1. Calculate what a small change in each weight would do to the loss function (selecting the direction to reach minima).
2. Adjust each weight based on its gradient (i.e., take a small step in the determined direction).
3. Keep doing steps 1 and 2 until the loss function gets as low as possible.

Here, adaptive moment estimation (Adam) is used as an optimizer. It is a blend of RMSprop and stochastic gradient descent.

Loss function and optimization go hand-in-hand. Loss function checks whether the model is moving in the correct direction and making progress, whereas optimization improves the model to deliver accurate results.

```python
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr = learning_rate)
```

```python
%%time
# keeping-track-of-losses
train_losses = []
valid_losses = []

for epoch in range(1, num_epochs + 1):
    # keep-track-of-training-and-validation-loss
    train_loss = 0.0
    valid_loss = 0.0

    # training-the-model
    model.train()
    for data, target in train_loader:
        # move-tensors-to-GPU
        data = data.to(device)
        target = target.to(device)

        # clear-the-gradients-of-all-optimized-variables
        optimizer.zero_grad()
        # forward-pass: compute-predicted-outputs-by-passing-inputs-t
        output = model(data)
        # calculate-the-batch-loss
        loss = criterion(output, target)
        # backward-pass: compute-gradient-of-the-loss-wrt-model-param
        loss.backward()
        # perform-a-ingle-optimization-step (parameter-update)
        optimizer.step()
        # update-training-loss
        train_loss += loss.item() * data.size(0)
```

```python
    # validate-the-model
    model.eval()
    for data, target in valid_loader:

        data = data.to(device)
        target = target.to(device)

        output = model(data)

        loss = criterion(output, target)

        # update-average-validation-loss
        valid_loss += loss.item() * data.size(0)

    # calculate-average-losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    # print-training/validation-statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f
        epoch, train_loss, valid_loss))
```

```
Epoch: 1 	Training Loss: 0.184436 	Validation Loss: 0.092147
Epoch: 2 	Training Loss: 0.114733 	Validation Loss: 0.081786
Epoch: 3 	Training Loss: 0.093309 	Validation Loss: 0.058328
Epoch: 4 	Training Loss: 0.083824 	Validation Loss: 0.059981
Epoch: 5 	Training Loss: 0.075828 	Validation Loss: 0.056827
Epoch: 6 	Training Loss: 0.067742 	Validation Loss: 0.053193
Epoch: 7 	Training Loss: 0.062057 	Validation Loss: 0.043070
Epoch: 8 	Training Loss: 0.060176 	Validation Loss: 0.046709
Epoch: 9 	Training Loss: 0.056599 	Validation Loss: 0.042972
Epoch: 10 	Training Loss: 0.053001 	Validation Loss: 0.043093
Epoch: 11 	Training Loss: 0.049367 	Validation Loss: 0.035351
Epoch: 12 	Training Loss: 0.049309 	Validation Loss: 0.040520
Epoch: 13 	Training Loss: 0.047358 	Validation Loss: 0.072312
Epoch: 14 	Training Loss: 0.045051 	Validation Loss: 0.040706
Epoch: 15 	Training Loss: 0.043941 	Validation Loss: 0.037109
Epoch: 16 	Training Loss: 0.042678 	Validation Loss: 0.042761
Epoch: 17 	Training Loss: 0.038122 	Validation Loss: 0.052519
Epoch: 18 	Training Loss: 0.036785 	Validation Loss: 0.039843
Epoch: 19 	Training Loss: 0.037711 	Validation Loss: 0.038851
Epoch: 20 	Training Loss: 0.039805 	Validation Loss: 0.029329
Epoch: 21 	Training Loss: 0.036795 	Validation Loss: 0.036935
Epoch: 22 	Training Loss: 0.028682 	Validation Loss: 0.042431
Epoch: 23 	Training Loss: 0.036859 	Validation Loss: 0.035401
Epoch: 24 	Training Loss: 0.034109 	Validation Loss: 0.046731
Epoch: 25 	Training Loss: 0.027740 	Validation Loss: 0.037606
Epoch: 26 	Training Loss: 0.032816 	Validation Loss: 0.034929
Epoch: 27 	Training Loss: 0.031394 	Validation Loss: 0.035643
Epoch: 28 	Training Loss: 0.032817 	Validation Loss: 0.034802
Epoch: 29 	Training Loss: 0.029015 	Validation Loss: 0.041272
Epoch: 30 	Training Loss: 0.027359 	Validation Loss: 0.036474
Epoch: 31 	Training Loss: 0.026263 	Validation Loss: 0.037024
Epoch: 32 	Training Loss: 0.024587 	Validation Loss: 0.027725
Epoch: 33 	Training Loss: 0.035328 	Validation Loss: 0.035390
Epoch: 34 	Training Loss: 0.026014 	Validation Loss: 0.047721
Epoch: 35 	Training Loss: 0.026504 	Validation Loss: 0.040448
CPU times: user 7min 30s, sys: 38.2 s, total: 8min 9s
Wall time: 10min 14s
```

python

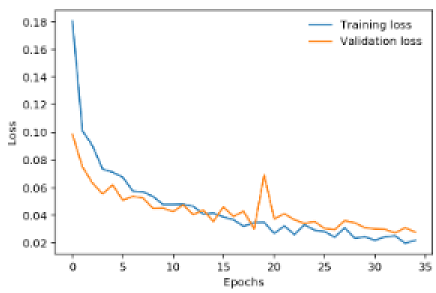
```python
# test-the-model
model.eval()  # it-disables-dropout
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in valid_loader:
        
        labels = labels.to(device)
```

```
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        print('Test Accuracy of the model: {} %'.format(100 * correct / t

# Save
torch.save(model.state_dict(), 'model.ckpt')
```

```
Test Accuracy of the model: 98.91428571428571 %
```

python

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

plt.plot(train_losses, label='Training loss')
plt.plot(valid_losses, label='Validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(frameon=False)
```



Conclusion

Take a deep breath! A CNN-based image classifier is ready, and it gives 98.9% accuracy. As per the graph above, training and validation loss decrease exponentially as the epochs increase. The losses are in line with each other, which proves that the model is reliable and there is no underfitting or overfitting of the model.

Data preparation is the most important and time-intensive process in data science. It is a great skill to know how to play around with data in the initial stage. Getting to know your data is what makes a good data scientist. This guide is not a complete one-stop for pre-processing, but you got a brief overview.

You also learned about the layers involved in designing the CNN model, the role of loss, and optimizer functions.

Building your own neural network is a cumbersome task, and that's why *transfer learning* (taking knowledge from one situation and applying it to another) is used a lot these days. Nevertheless, it is always good to have foundational knowledge.

184