Chapter 18

Crash Course In Convolutional Neural Networks

Convolutional Neural Networks are a powerful artificial neural network technique. These networks preserve the spatial structure of the problem and were developed for object recognition tasks such as handwritten digit recognition. They are popular because people are achieving state-of-the-art results on difficult computer vision and natural language processing tasks. In this lesson you will discover Convolutional Neural Networks for deep learning, also called ConvNets or CNNs. After completing this crash course you will know:

- The building blocks used in CNNs such as convolutional layers and pool layers.
- How the building blocks fit together with a short worked example.
- Best practices for configuring CNNs on your own object recognition tasks.

Let's get started.

18.1 The Case for Convolutional Neural Networks

Given a dataset of gray scale images with the standardized size of 32×32 pixels each, a traditional feedforward neural network would require 1,024 input weights (plus one bias). This is fair enough, but the flattening of the image matrix of pixels to a long vector of pixel values looses all of the spatial structure in the image. Unless all of the images are perfectly resized, the neural network will have great difficulty with the problem.

Convolutional Neural Networks expect and preserve the spatial relationship between pixels by learning internal feature representations using small squares of input data. Feature are learned and used across the whole image, allowing for the objects in the images to be shifted or translated in the scene and still detectable by the network. It is this reason why the network is so useful for object recognition in photographs, picking out digits, faces, objects and so on with varying orientation. In summary, below are some of the benefits of using convolutional neural networks:

- They use fewer parameters (weights) to learn than a fully connected network.
- They are designed to be invariant to object position and distortion in the scene.
- They automatically learn and generalize features from the input domain.

18.2 Building Blocks of Convolutional Neural Networks

There are three types of layers in a Convolutional Neural Network:

1. Convolutional Layers.
2. Pooling Layers.
3. Fully-Connected Layers.

18.3 Convolutional Layers

Convolutional layers are comprised of filters and feature maps.

18.3.1 Filters

The filters are essentially the *neurons* of the layer. They have both weighted inputs and generate an output value like a neuron. The input size is a fixed square called a patch or a receptive field. If the convolutional layer is an input layer, then the input patch will be pixel values. If they deeper in the network architecture, then the convolutional layer will take input from a feature map from the previous layer.

18.3.2 Feature Maps

The feature map is the output of one filter applied to the previous layer. A given filter is drawn across the entire previous layer, moved one pixel at a time. Each position results in an activation of the neuron and the output is collected in the feature map. You can see that if the receptive field is moved one pixel from activation to activation, then the field will overlap with the previous activation by (field width - 1) input values.

The distance that filter is moved across the input from the previous layer each activation is referred to as the stride. If the size of the previous layer is not cleanly divisible by the size of the filters receptive field and the size of the stride then it is possible for the receptive field to attempt to read off the edge of the input feature map. In this case, techniques like zero padding can be used to invent mock inputs with zero values for the receptive field to read.

18.4 Pooling Layers

The pooling layers down-sample the previous layers feature map. Pooling layers follow a sequence of one or more convolutional layers and are intended to consolidate the features learned and expressed in the previous layers feature map. As such, pooling may be consider a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

They too have a receptive field, often much smaller than the convolutional layer. Also, the stride or number of inputs that the receptive field is moved for each activation is often equal to the size of the receptive field to avoid any overlap. Pooling layers are often very simple, taking the average or the maximum of the input value in order to create its own feature map.

18.5 Fully Connected Layers

Fully connected layers are the normal flat feedforward neural network layer. These layers may have a nonlinear activation function or a softmax activation in order to output probabilities of class predictions. Fully connected layers are used at the end of the network after feature extraction and consolidation has been performed by the convolutional and pooling layers. They are used to create final nonlinear combinations of features and for making predictions by the network.

18.6 Worked Example

You now know about convolutional, pooling and fully connected layers. Let's make this more concrete by working through how these three layers may be connected together.

18.6.1 Image Input Data

Let's assume we have a dataset of gray scale images. Each image has the same size of 32 pixels wide and 32 pixels high, and pixel values are between 0 and 255, e.g. a matrix of $32 \times 32 \times 1$ or 1,024 pixel values. Image input data is expressed as a 3-dimensional matrix of $width \times height \times channels$. If we were using color images in our example, we would have 3 channels for the red, green and blue pixel values, e.g. $32 \times 32 \times 3$.

18.6.2 Convolutional Layer

We define a convolutional layer with 10 filters and a receptive field 5 pixels wide and 5 pixels high and a stride length of 1. Because each filter can only get input from (i.e. *see*) 5×5 (25) pixels at a time, we can calculate that each will require $25 + 1$ input weights (plus 1 for the bias input). Dragging the 5×5 receptive field across the input image data with a stride width of 1 will result in a feature map of 28×28 output values or 784 distinct activations per image.

We have 10 filters, so that is 10 different 28×28 feature maps or 7,840 outputs that will be created for one image. Finally, we know we have 26 inputs per filter, 10 filters and 28×28 output values to calculate per filter, therefore we have a total of $26 \times 10 \times 28 \times 28$ or 203,840 connections in our convolutional layer, we want to phrase it using traditional neural network nomenclature. Convolutional layers also make use of a nonlinear transfer function as part of activation and the rectifier activation function is the popular default to use.

18.6.3 Pool Layer

We define a pooling layer with a receptive field with a width of 2 inputs and a height of 2 inputs. We also use a stride of 2 to ensure that there is no overlap. This results in feature maps that are one half the size of the input feature maps. From 10 different 28×28 feature maps as input to 10 different 14×14 feature maps as output. We will use a `max()` operation for each receptive field so that the activation is the maximum input value.

18.6.4 Fully Connected Layer

Finally, we can flatten out the square feature maps into a traditional flat fully connected layer. We can define the fully connected layer with 200 hidden neurons, each with $10 \times 14 \times 14$ input connections, or 1,960 + 1 weights per neuron. That is a total of 392,200 connections and weights to learn in this layer. We can use a sigmoid or softmax transfer function to output probabilities of class values directly.

18.7 Convolutional Neural Networks Best Practices

Now that we know about the building blocks for a convolutional neural network and how the layers hang together, we can review some best practices to consider when applying them.

- **Input Receptive Field Dimensions**: The default is 2D for images, but could be 1D such as for words in a sentence or 3D for video that adds a time dimension.
- **Receptive Field Size**: The patch should be as small as possible, but large enough to *see* features in the input data. It is common to use 3×3 on small images and 5×5 or 7×7 and more on larger image sizes.
- **Stride Width**: Use the default stride of 1. It is easy to understand and you don't need padding to handle the receptive field falling off the edge of your images. This could be increased to 2 or larger for larger images.
- **Number of Filters**: Filters are the feature detectors. Generally fewer filters are used at the input layer and increasingly more filters used at deeper layers.
- **Padding**: Set to zero and called zero padding when reading non-input data. This is useful when you cannot or do not want to standardize input image sizes or when you want to use receptive field and stride sizes that do not neatly divide up the input image size.
- **Pooling**: Pooling is a destructive or generalization process to reduce overfitting. Receptive field size is almost always set to 2×2 with a stride of 2 to discard 75% of the activations from the output of the previous layer.
- **Data Preparation**: Consider standardizing input data, both the dimensions of the images and pixel values.
- **Pattern Architecture**: It is common to pattern the layers in your network architecture. This might be one, two or some number of convolutional layers followed by a pooling layer. This structure can then be repeated one or more times. Finally, fully connected layers are often only used at the output end and may be stacked one, two or more deep.
- **Dropout**: CNNs have a habit of overfitting, even with pooling layers. Dropout should be used such as between fully connected layers and perhaps after pooling layers.

18.8 Summary

In this lesson you discovered convolutional neural networks. You learned:

- Why CNNs are needed to preserve spatial structure in your input data and the benefits they provide.
- The building blocks of CNN including convolutional, pooling and fully connected layers.
- How the layers in a CNN hang together.
- Best practices when applying CNN to your own problems.

18.8.1 Next

You now know about convolutional neural networks. In the next section you will discover how to develop your first convolutional neural network in Keras for a handwriting digit recognition problem.

Chapter 19

Project: Handwritten Digit Recognition

A popular demonstration of the capability of deep learning techniques is object recognition in image data. The *hello world* of object recognition for machine learning and deep learning is the MNIST dataset for handwritten digit recognition. In this project you will discover how to develop a deep learning model to achieve near state-of-the-art performance on the MNIST handwritten digit recognition task in Python using the Keras deep learning library. After completing this step-by-step tutorial, you will know:

- How to load the MNIST dataset in Keras and develop a baseline neural network model for the problem.
- How to implement and evaluate a simple Convolutional Neural Network for MNIST.
- How to implement a close to state-of-the-art deep learning model for MNIST.

Let's get started.

Note: You may want to speed up the computation for this tutorial by using GPU rather than CPU hardware, such as the process described in Chapter 5. This is a suggestion, not a requirement. The tutorial will work just fine on the CPU.

19.1 Handwritten Digit Recognition Dataset

The MNIST problem is a dataset developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem[1]. The dataset was constructed from a number of scanned document datasets available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, as the Modified NIST or MNIST dataset.

Images of digits were taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required. Each image is a 28×28 pixel square (784 pixels total). A standard split of the dataset is used to

[1] http://yann.lecun.com/exdb/mnist/

evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.

It is a digit recognition task. As such there are 10 digits (0 to 9) or 10 classes to predict. Results are reported using prediction error, which is nothing more than the inverted classification accuracy. Excellent results achieve a prediction error of less than 1%. State-of-the-art prediction error of approximately 0.2% can be achieved with large Convolutional Neural Networks. There is a listing of the state-of-the-art results and links to the relevant papers on the MNIST and other datasets on Rodrigo Benenson's webpage[2].

19.2 Loading the MNIST dataset in Keras

The Keras deep learning library provides a convenience method for loading the MNIST dataset. The dataset is downloaded automatically the first time this function is called and is stored in your home directory in `~/.keras/datasets/mnist.pkl.gz` as a 15 megabyte file. This is very handy for developing and testing deep learning models. To demonstrate how easy it is to load the MNIST dataset, we will first write a little script to download and visualize the first 4 images in the training dataset.

```
# Plot ad hoc mnist instances
from keras.datasets import mnist
import matplotlib.pyplot as plt
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# show the plot
plt.show()
```

Listing 19.1: Load the MNIST Dataset in Keras.

You can see that downloading and loading the MNIST dataset is as easy as calling the `mnist.load_data()` function. Running the above example, you should see the image below.

[2] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

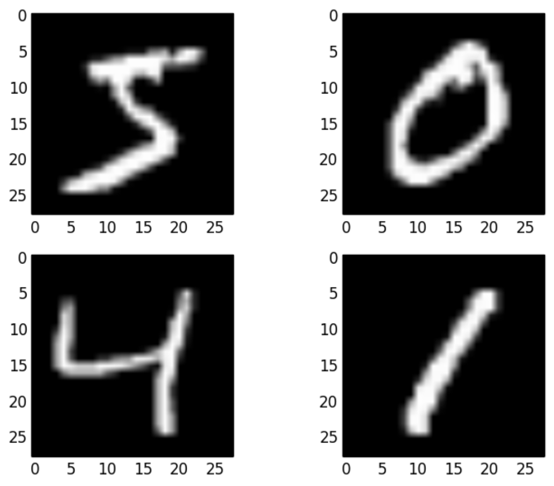

Figure 19.1: Examples from the MNIST dataset

19.3 Baseline Model with Multilayer Perceptrons

Do we really need a complex model like a convolutional neural network to get the best results with MNIST? You can get good results using a very simple neural network model with a single hidden layer. In this section we will create a simple Multilayer Perceptron model that achieves an error rate of 1.74%. We will use this as a baseline for comparison to more complex convolutional neural network models. Let's start off by importing the classes and functions we will need.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
```

Listing 19.2: Import Classes and Functions.

It is always a good idea to initialize the random number generator to a constant to ensure that the results of your script are reproducible.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 19.3: Initialize The Random Number Generator.

Now we can load the MNIST dataset using the Keras helper function.

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Listing 19.4: Load the MNIST Dataset.

The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a Multilayer Perceptron model we must reduce the images down into a vector of pixels. In this case the 28×28 sized images will be 784 pixel input vectors. We can do this transform easily using the `reshape()` function on the NumPy array. The pixel values are integers, so we cast them to floating point values so that we can normalize them easily in the next step.

```
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

Listing 19.5: Prepare MNIST Dataset For Modeling.

The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Listing 19.6: Normalize Pixel Values.

Finally, the output variable is an integer from 0 to 9. This is a multiclass classification problem. As such, it is good practice to use a one hot encoding of the class values, transforming the vector of class integers into a binary matrix. We can easily do this using the built-in `np_utils.to_categorical()` helper function in Keras.

```
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 19.7: One Hot Encode The Output Variable.

We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score.

```
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(num_pixels, input_dim=num_pixels, init='normal', activation='relu'))
  model.add(Dense(num_classes, init='normal', activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 19.8: Define and Compile the Baseline Model.

The model is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). A rectifier activation function is used for the neurons in the hidden layer. A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction. Logarithmic loss is used as the loss function (called `categorical_crossentropy` in Keras) and the efficient ADAM gradient descent algorithm is used to learn the weights. A summary of the network structure is provided below:

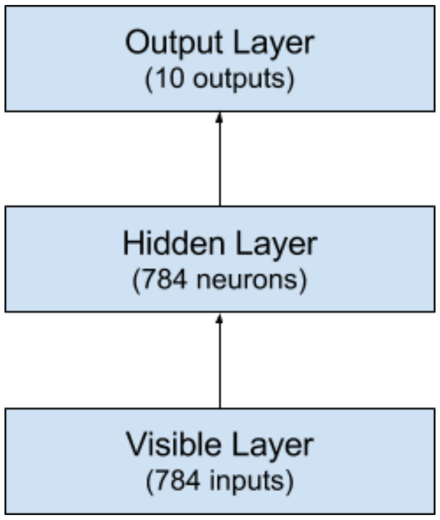

Figure 19.2: Summary of Multilayer Perceptron Network Structure.

We can now fit and evaluate the model. The model is fit over 10 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. A verbose value of 2 is used to reduce the output to one line for each training epoch. Finally, the test dataset is used to evaluate the model and a classification error rate is printed.

```
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.9: Evaluate the Baseline Model.

The full code listing is provided below for completeness.

```
# Baseline MLP for MNIST dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(num_pixels, input_dim=num_pixels, init='normal', activation='relu'))
  model.add(Dense(num_classes, init='normal', activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.10: Multilayer Perceptron Model for MNIST Problem.

Running the example might take a few minutes when run on a CPU. You should see the output below. This simple network defined in very few lines of code achieves a respectable error rate of 1.73%.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
7s - loss: 0.2791 - acc: 0.9203 - val_loss: 0.1420 - val_acc: 0.9579
Epoch 2/10
7s - loss: 0.1122 - acc: 0.9679 - val_loss: 0.0992 - val_acc: 0.9699
Epoch 3/10
8s - loss: 0.0724 - acc: 0.9790 - val_loss: 0.0784 - val_acc: 0.9745
Epoch 4/10
8s - loss: 0.0510 - acc: 0.9853 - val_loss: 0.0777 - val_acc: 0.9774
Epoch 5/10
8s - loss: 0.0367 - acc: 0.9898 - val_loss: 0.0628 - val_acc: 0.9792
Epoch 6/10
8s - loss: 0.0264 - acc: 0.9931 - val_loss: 0.0641 - val_acc: 0.9797
Epoch 7/10
8s - loss: 0.0185 - acc: 0.9958 - val_loss: 0.0604 - val_acc: 0.9810
Epoch 8/10
10s - loss: 0.0148 - acc: 0.9967 - val_loss: 0.0621 - val_acc: 0.9811
Epoch 9/10
8s - loss: 0.0109 - acc: 0.9978 - val_loss: 0.0607 - val_acc: 0.9817
Epoch 10/10
8s - loss: 0.0073 - acc: 0.9987 - val_loss: 0.0595 - val_acc: 0.9827
```

```
Baseline Error: 1.73%
```

Listing 19.11: Sample Output From Evaluating the Baseline Model.

19.4 Simple Convolutional Neural Network for MNIST

Now that we have seen how to load the MNIST dataset and train a simple Multilayer Perceptron model on it, it is time to develop a more sophisticated convolutional neural network or CNN model. Keras does provide a lot of capability for creating convolutional neural networks. In this section we will create a simple CNN for MNIST that demonstrates how to use all of the aspects of a modern CNN implementation, including Convolutional layers, Pooling layers and Dropout layers. The first step is to import the classes and functions needed.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
```

Listing 19.12: Import classes and functions.

Again, we always initialize the random number generator to a constant seed value for reproducibility of results.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 19.13: Seed Random Number Generator.

Next we need to load the MNIST dataset and reshape it so that it is suitable for use training a CNN. In Keras, the layers used for two-dimensional convolutions expect pixel values with the dimensions `[channels][width][height]`. In the case of RGB, the first dimension channels would be 3 for the red, green and blue components and it would be like having 3 image inputs for every color image. In the case of MNIST where the channels values are gray scale, the pixel dimension is set to 1.

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][channels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
```

Listing 19.14: Load Dataset and Separate Into Train and Test Sets.

As before, it is a good idea to normalize the pixel values to the range 0 and 1 and one hot encode the output variable.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
```

```
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 19.15: Normalize and One Hot Encode Data.

Next we define our neural network model. Convolutional neural networks are more complex than standard Multilayer Perceptrons, so we will start by using a simple structure to begin with that uses all of the elements for state-of-the-art results. Below summarizes the network architecture.

1. The first hidden layer is a convolutional layer called a `Convolution2D`. The layer has 32 feature maps, which with the size of 5×5 and a rectifier activation function. This is the input layer, expecting images with the structure outline above.
2. Next we define a pooling layer that takes the maximum value called `MaxPooling2D`. It is configured with a pool size of 2×2.
3. The next layer is a regularization layer using dropout called `Dropout`. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.
4. Next is a layer that converts the 2D matrix data to a vector called `Flatten`. It allows the output to be processed by standard fully connected layers.
5. Next a fully connected layer with 128 neurons and rectifier activation function is used.
6. Finally, the output layer has 10 neurons for the 10 classes and a softmax activation function to output probability-like predictions for each class.

As before, the model is trained using logarithmic loss and the ADAM gradient descent algorithm. A depiction of the network structure is provided below.

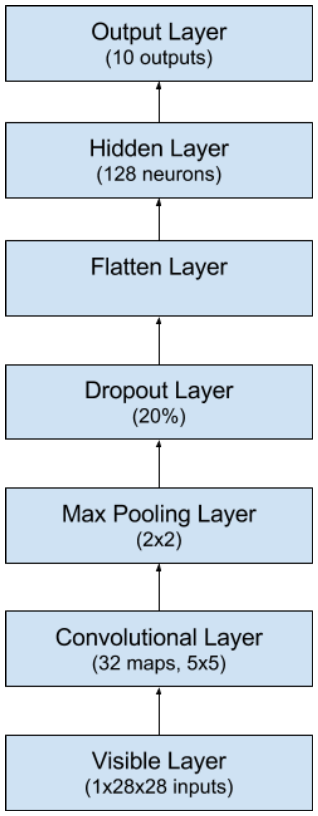

Figure 19.3: Summary of Convolutional Neural Network Structure.

```
def baseline_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(1, 28, 28),
      activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 19.16: Define and Compile CNN Model.

We evaluate the model the same way as before with the Multilayer Perceptron. The CNN is fit over 10 epochs with a batch size of 200.

```
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
```

```
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.17: Fit and Evaluate The CNN Model.

The full code listing is provided below for completeness.

```
# Simple CNN for the MNIST Dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][channels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define a simple CNN model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(32, 5, 5, input_shape=(1, 28, 28), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.18: CNN Model for MNIST Problem.

Running the example, the accuracy on the training and validation test is printed each epoch and at the end of the classification error rate is printed. Epochs may take 60 seconds to run on the CPU, or about 10 minutes in total depending on your hardware. You can see that the network achieves an error rate of 1.00, which is better than our simple Multilayer Perceptron model above.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
52s - loss: 0.2412 - acc: 0.9318 - val_loss: 0.0756 - val_acc: 0.9763
Epoch 2/10
52s - loss: 0.0728 - acc: 0.9781 - val_loss: 0.0527 - val_acc: 0.9825
Epoch 3/10
53s - loss: 0.0497 - acc: 0.9852 - val_loss: 0.0393 - val_acc: 0.9852
Epoch 4/10
53s - loss: 0.0414 - acc: 0.9868 - val_loss: 0.0436 - val_acc: 0.9852
Epoch 5/10
53s - loss: 0.0324 - acc: 0.9898 - val_loss: 0.0376 - val_acc: 0.9871
Epoch 6/10
53s - loss: 0.0281 - acc: 0.9910 - val_loss: 0.0421 - val_acc: 0.9866
Epoch 7/10
52s - loss: 0.0227 - acc: 0.9928 - val_loss: 0.0319 - val_acc: 0.9895
Epoch 8/10
52s - loss: 0.0198 - acc: 0.9938 - val_loss: 0.0357 - val_acc: 0.9885
Epoch 9/10
52s - loss: 0.0158 - acc: 0.9951 - val_loss: 0.0333 - val_acc: 0.9887
Epoch 10/10
52s - loss: 0.0144 - acc: 0.9953 - val_loss: 0.0322 - val_acc: 0.9900
CNN Error: 1.00%
```

Listing 19.19: Sample Output From Evaluating the CNN Model.

19.5 Larger Convolutional Neural Network for MNIST

Now that we have seen how to create a simple CNN, let's take a look at a model capable of close to state-of-the-art results. We import the classes and functions then load and prepare the data the same as in the previous CNN example. This time we define a larger CNN architecture with additional convolutional, max pooling layers and fully connected layers. The network topology can be summarized as follows.

1. Convolutional layer with 30 feature maps of size 5×5.
2. Pooling layer taking the max over 2×2 patches.
3. Convolutional layer with 15 feature maps of size 3×3.
4. Pooling layer taking the max over 2×2 patches.
5. Dropout layer with a probability of 20%.
6. Flatten layer.
7. Fully connected layer with 128 neurons and rectifier activation.

8. Fully connected layer with 50 neurons and rectifier activation.

9. Output layer.

A depictions of this larger network structure is provided below.

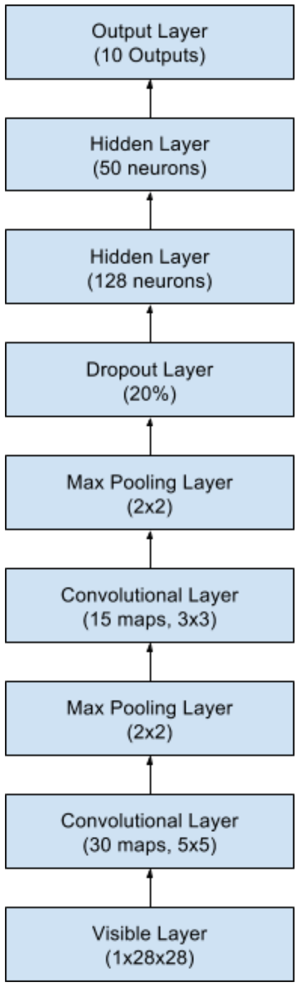

Figure 19.4: Summary of the Larger Convolutional Neural Network Structure.

Like the previous two experiments, the model is fit over 10 epochs with a batch size of 200.

```
# Larger CNN for the MNIST Dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
```

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define the larger model
def larger_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(30, 5, 5, input_shape=(1, 28, 28), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Convolution2D(15, 3, 3, activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(50, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = larger_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.20: Larger CNN for the MNIST Problem.

Running the example prints accuracy on the training and validation datasets each epoch and a final classification error rate. The model takes about 60 seconds to run per epoch on a modern CPU. This slightly larger model achieves the respectable classification error rate of 0.83%.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
59s - loss: 0.3791 - acc: 0.8793 - val_loss: 0.0812 - val_acc: 0.9742
Epoch 2/10
59s - loss: 0.0929 - acc: 0.9708 - val_loss: 0.0467 - val_acc: 0.9846
Epoch 3/10
59s - loss: 0.0679 - acc: 0.9787 - val_loss: 0.0374 - val_acc: 0.9878
Epoch 4/10
59s - loss: 0.0539 - acc: 0.9828 - val_loss: 0.0321 - val_acc: 0.9893
Epoch 5/10
```

```
59s - loss: 0.0462 - acc: 0.9858 - val_loss: 0.0282 - val_acc: 0.9897
Epoch 6/10
60s - loss: 0.0396 - acc: 0.9874 - val_loss: 0.0278 - val_acc: 0.9902
Epoch 7/10
60s - loss: 0.0365 - acc: 0.9884 - val_loss: 0.0229 - val_acc: 0.9921
Epoch 8/10
59s - loss: 0.0328 - acc: 0.9895 - val_loss: 0.0287 - val_acc: 0.9901
Epoch 9/10
60s - loss: 0.0306 - acc: 0.9903 - val_loss: 0.0224 - val_acc: 0.9923
Epoch 10/10
59s - loss: 0.0268 - acc: 0.9915 - val_loss: 0.0240 - val_acc: 0.9917
Large CNN Error: 0.83%
```

Listing 19.21: Sample Output From Evaluating the Larger CNN Model.

This is not an optimized network topology. Nor is this a reproduction of a network topology from a recent paper. There is a lot of opportunity for you to tune and improve upon this model. What is the best classification error rate you can achieve?

19.6 Summary

In this lesson you discovered the MNIST handwritten digit recognition problem and deep learning models developed in Python using the Keras library that are capable of achieving excellent results. Working through this tutorial you learned:

- How to load the MNIST dataset in Keras and generate plots of the dataset.
- How to reshape the MNIST dataset and develop a simple but well performing Multilayer Perceptron model for the problem.
- How to use Keras to create convolutional neural network models for MNIST.
- How to develop and evaluate larger CNN models for MNIST capable of near world class results.

19.6.1 Next

You now know how to develop and improve convolutional neural network models in Keras. A powerful technique for improving the performance of CNN models is to use data augmentation. In the next section you will discover the data augmentation API in Keras and how the different image augmentation techniques affect the MNIST images.