Part III

Multilayer Perceptrons

Chapter 6

Crash Course In Multilayer Perceptrons

Artificial neural networks are a fascinating area of study, although they can be intimidating when just getting started. There is a lot of specialized terminology used when describing the data structures and algorithms used in the field. In this lesson you will get a crash course in the terminology and processes used in the field of Multilayer Perceptron artificial neural networks. After completing this lesson you will know:

- The building blocks of neural networks including neurons, weights and activation functions.
- How the building blocks are used in layers to create networks.
- How networks are trained from example data.

Let's get started.

6.1 Crash Course Overview

We are going to cover a lot of ground in this lesson. Here is an idea of what is ahead:

1. Multilayer Perceptrons.
2. Neurons, Weights and Activations.
3. Networks of Neurons.
4. Training Networks.

We will start off with an overview of Multilayer Perceptrons.

6.2 Multilayer Perceptrons

The field of artificial neural networks is often just called *Neural Networks* or *Multilayer Perceptrons* after perhaps the most useful type of neural network. A Perceptron is a single neuron model that was a precursor to larger neural networks. It is a field of study that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that we can use to model difficult problems.

The power of neural networks come from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multilayered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

6.3 Neurons

The building block for neural networks are artificial neurons. These are simple computational units that have weighted input signals and produce an output signal using an activation function.

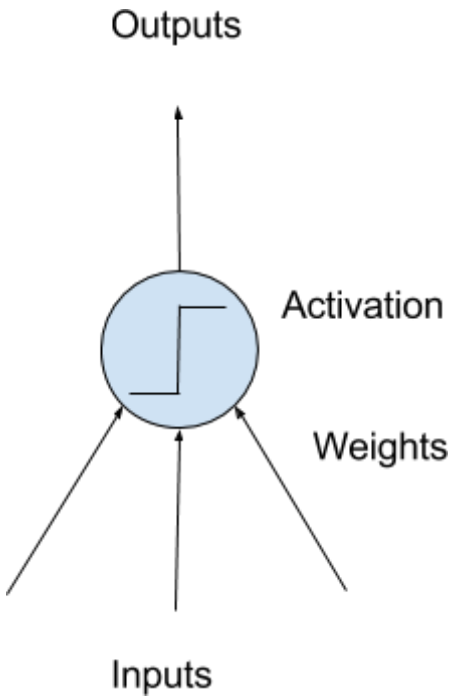

Figure 6.1: Model of a Simple Neuron

6.3.1 Neuron Weights

You may be familiar with linear regression, in which case the weights on the inputs are very much like the coefficients used in a regression equation. Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0 and it too must be weighted. For example, a neuron may have two inputs in which case it requires three weights. One for each input and one for the bias.

Weights are often initialized to small random values, such as values in the range 0 to 0.3, although more complex initialization schemes can be used. Like linear regression, larger weights indicate increased complexity and fragility of the model. It is desirable to keep weights in the network small and regularization techniques can be used.

6.3.2 Activation

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function. An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal. Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

Traditionally nonlinear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Nonlinear functions like the logistic function also called the sigmoid function were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called Tanh that outputs the same distribution over the range -1 to +1. More recently the rectifier activation function has been shown to provide better results.

6.4 Networks of Neurons

Neurons are arranged into networks of neurons. A row of neurons is called a layer and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.

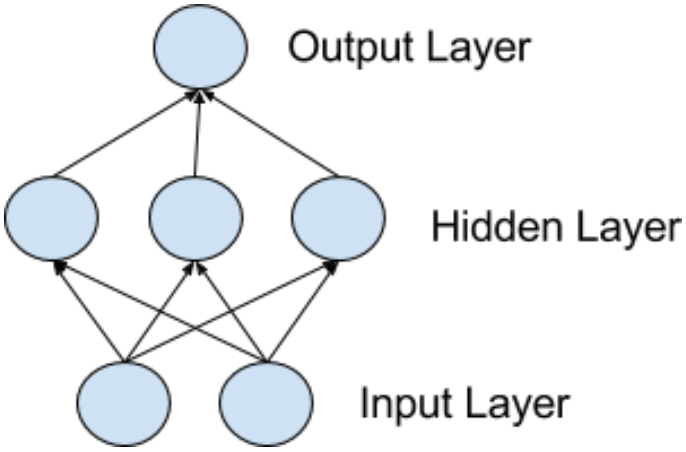

Figure 6.2: Model of a Simple Network

6.4.1 Input or Visible Layers

The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These are not neurons as described above, but simply pass the input value though to the next layer.

6.4.2 Hidden Layers

Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

6.4.3 Output Layer

The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem. The choice of activation function in the output layer is strongly constrained by the type of problem that you are modeling. For example:

- A regression problem may have a single output neuron and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the primary class. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.

- A multiclass classification problem may have multiple neurons in the output layer, one for each class (e.g. three neurons for the three classes in the famous iris flowers classification problem). In this case a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

6.5 Training Networks

Once configured, the neural network needs to be trained on your dataset.

6.5.1 Data Preparation

You must first prepare your data for training on a neural network. Data must be numerical, for example real values. If you have categorical data, such as a `sex` attribute with the values `male` and `female`, you can convert it to a real-valued representation called a one hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female) and a 0 or 1 is added for each row depending on the class value for that row.

This same one hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network's output layer, that as described above, would output one value for each class. Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1 called normalization. Another popular technique is to standardize it so that the distribution of each column has the mean of zero and the standard deviation of 1. Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the frequency rank of the word in the dataset and other encoding techniques.

6.5.2 Stochastic Gradient Descent

The classical and still preferred training algorithm for neural networks is called stochastic gradient descent. This is where one row of data is exposed to the network at a time as input. The network processes the input upward activating neurons as it goes to finally produce an output value. This is called a forward pass on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount that they contributed to the error. This clever bit of math is called the Back Propagation algorithm. The process is repeated for all of the examples in your training data. One round of updating the network for the entire training dataset is called an epoch. A network may be trained for tens, hundreds or many thousands of epochs.

6.5.3 Weight Updates

The weights in the network can be updated from the errors calculated for each training example and this is called online learning. It can result in fast but also chaotic changes to the network.

Alternatively, the errors can be saved up across all of the training examples and the network can be updated at the end. This is called batch learning and is often more stable.

Because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update is often reduced to a small number, such as tens or hundreds of examples. The amount that weights are updated is controlled by a configuration parameter called the learning rate. It is also called the step size and controls the step or change made to network weights for a given error. Often small learning rates are used such as 0.1 or 0.01 or smaller. The update equation can be complemented with additional configuration terms that you can set.

- **Momentum** is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.
- **Learning Rate Decay** is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine tuning changes later in the training schedule.

6.5.4 Prediction

Once a neural network has been trained it can be used to make predictions. You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously. The network topology and the final set of weights is all that you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass allowing it to generate an output that you can use as a prediction.

6.6 Summary

In this lesson you discovered artificial neural networks for machine learning. You learned:

- How neural networks are not models of the brain but are instead computational models for solving complex machine learning problems.
- That neural networks are comprised of neurons that have weights and activation functions.
- The networks are organized into layers of neurons and are trained using stochastic gradient descent.
- That it is a good idea to prepare your data before training a neural network model.

6.6.1 Next

You now know the basics of neural network models. In the next section you will develop your very first Multilayer Perceptron model in Keras.

Chapter 7

Develop Your First Neural Network With Keras

Keras is a powerful and easy-to-use Python library for developing and evaluating deep learning models. It wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in a few short lines of code. In this lesson you will discover how to create your first neural network model in Python using Keras. After completing this lesson you will know:

- How to load a CSV dataset ready for use with Keras.
- How to define and compile a Multilayer Perceptron model in Keras.
- How to evaluate a Keras model on a validation dataset.

Let's get started.

7.1 Tutorial Overview

There is not a lot of code required, but we are going to step over it slowly so that you will know how to create your own models in the future. The steps you are going to cover in this tutorial are as follows:

1. Load Data.
2. Define Model.
3. Compile Model.
4. Fit Model.
5. Evaluate Model.
6. Tie It All Together.

7.2 Pima Indians Onset of Diabetes Dataset

In this tutorial we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset available for free download from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years. It is a binary classification problem (onset of diabetes as 1 or not as 0). The input variables that describe each patient are numerical and have varying scales. Below lists the eight attributes for the dataset:

1. Number of times pregnant.
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skin fold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index.
7. Diabetes pedigree function.
8. Age (years).
9. Class, onset of diabetes within five years.

Given that all attributes are numerical makes it easy to use directly with neural networks that expect numerical inputs and output values, and ideal for our first neural network in Keras. This dataset will also be used for a number of additional lessons coming up in this book, so keep it handy. below is a sample of the dataset showing the first 5 rows of the 768 instances:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Listing 7.1: Sample of the Pima Indians Dataset.

The dataset file is available in your bundle of code recipes provided with this book. Alternatively, you can download the Pima Indian dataset from the UCI Machine Learning repository and place it in your local working directory, the same as your Python file[1]. Save it with the file name:

```
pima-indians-diabetes.csv
```

Listing 7.2: Pima Indians Dataset File.

[1] http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data

The baseline accuracy if all predictions are made as *no onset of diabetes* is 65.1%. Top results on the dataset are in the range of 77.7% accuracy using 10-fold cross validation[2]. You can learn more about the dataset on the dataset home page on the UCI Machine Learning Repository[3].

7.3 Load Data

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to initialize the random number generator with a fixed seed value. This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code. You can initialize the random number generator with any seed you like, for example:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 7.3: Load Libraries and Seed Random Number Generator.

Now we can load our Pima Indians dataset. You can now load the file directly using the NumPy function `loadtxt()`. There are eight input variables and one output variable (the last column). Once loaded we can split the dataset into input variables (X) and the output class variable (Y).

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

Listing 7.4: Load The Dataset Using NumPy.

We have initialized our random number generator to ensure our results are reproducible and loaded our data. We are now ready to define our neural network model.

7.4 Define Model

Models in Keras are defined as a sequence of layers. We create a `Sequential` model and add layers one at a time until we are happy with our network topology. The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the `input_dim` argument and setting it to 8 for the 8 input variables.

How do we know the number of layers to use and their types? This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough

[2] http://www.is.umk.pl/projects/datasets.html#Diabetes
[3] http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes

to capture the structure of the problem if that helps at all. In this example we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the `Dense` class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as `init` and specify the activation function using the `activation` argument. In this case we initialize the network weights to a small random number generated from a uniform distribution (`uniform`), in this case between 0 and 0.05 because that is the default uniform weight initialization in Keras. Another traditional alternative would be `normal` for small random numbers generated from a Gaussian distribution.

We will use the rectifier (`relu`) activation function on the first two layers and the `sigmoid` activation function in the output layer. It used to be the case that sigmoid and tanh activation functions were preferred for all layers. These days, better performance is seen using the rectifier activation function. We use a sigmoid activation function on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5. We can piece it all together by adding each layer. The first hidden layer has 12 neurons and expects 8 input variables. The second hidden layer has 8 neurons and finally the output layer has 1 neuron to predict the class (onset of diabetes or not).

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

Listing 7.5: Define the Neural Network Model in Keras.

Below provides a depiction of the network structure.

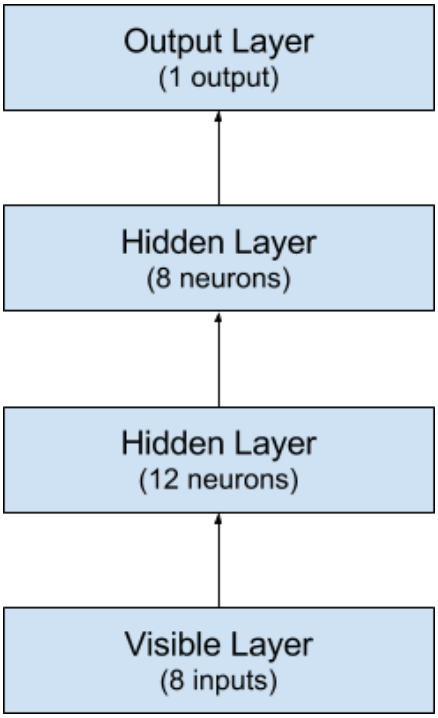

Figure 7.1: Visualization of Neural Network Structure.

7.5 Compile Model

Now that the model is defined, we can compile it. Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware. When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to make predictions for this problem.

We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training. In this case we will use logarithmic loss, which for a binary classification problem is defined in Keras as `binary_crossentropy`. We will also use the efficient gradient descent algorithm `adam` for no other reason that it is an efficient default. Learn more about the Adam optimization algorithm in the paper *Adam: A Method for Stochastic Optimization*[4]. Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 7.6: Compile the Neural Network Model.

[4] http://arxiv.org/abs/1412.6980

7.6 Fit Model

We have defined our model and compiled it ready for efficient computation. Now it is time to execute the model on some data. We can train or fit our model on our loaded data by calling the `fit()` function on the model.

The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the `nb_epoch` argument. We can also set the number of instances that are evaluated before a weight update in the network is performed called the batch size and set using the `batch_size` argument. For this problem we will run for a small number of epochs (150) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

Listing 7.7: Fit the Neural Network Model to the Dataset.

This is where the work happens on your CPU or GPU.

7.7 Evaluate Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset. This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for the training and evaluation of your model.

You can evaluate your model on your training dataset using the `evaluation()` function on your model and pass it the same input and output used to train the model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 7.8: Evaluate the Neural Network Model on the Dataset.

7.8 Tie It All Together

You have just seen how you can easily create your first neural network model in Keras. Let's tie it all together into a complete code example.

```
# Create your first MLP in Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
```

```
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 7.9: Complete Working Example of Your First Neural Network in Keras.

Running this example, you should see a message for each of the 150 epochs printing the loss and accuracy for each, followed by the final evaluation of the trained model on the training dataset. It takes about 10 seconds to execute on my workstation running on the CPU with a Theano backend.

```
...
Epoch 145/150
768/768 [==============================] - 0s - loss: 0.4574 - acc: 0.7786
Epoch 146/150
768/768 [==============================] - 0s - loss: 0.4620 - acc: 0.7734
Epoch 147/150
768/768 [==============================] - 0s - loss: 0.4633 - acc: 0.7760
Epoch 148/150
768/768 [==============================] - 0s - loss: 0.4554 - acc: 0.7812
Epoch 149/150
768/768 [==============================] - 0s - loss: 0.4656 - acc: 0.7643
Epoch 150/150
768/768 [==============================] - 0s - loss: 0.4618 - acc: 0.7878
448/768 [===============>..............] - ETA: 0sa
cc: 78.39%
```

Listing 7.10: Output of Running Your First Neural Network in Keras.

7.9 Summary

In this lesson you discovered how to create your first neural network model using the powerful Keras Python library for deep learning. Specifically you learned the five key steps in using Keras to create a neural network or deep learning model, step-by-step including:

- How to load data.
- How to define a neural network model in Keras.
- How to compile a Keras model using the efficient numerical backend.

- How to train a model on data.
- How to evaluate a model on data.

7.9.1 Next

You now know how to develop a Multilayer Perceptron model in Keras. In the next section you will discover different ways that you can evaluate your models and estimate their performance on unseen data.