Neural Networks in Keras

Sequential Model

In [1]:

```
from keras.models import Sequential
```

```
Using TensorFlow backend.
```

In [2]:

```
# Create a Sequential model
model = Sequential()
```

The keras.models.Sequential class is a wrapper for the neural network model that treats the network as a sequence of layers. It implements the Keras model interface with common methods like compile(), fit(), and evaluate() that are used to train and run the model. We'll cover these functions soon, but first let's start looking at the layers of the model.

Layers

The Keras Layer class provides a common interface for a variety of standard neural network layers. There are fully connected layers, max pool layers, activation layers, and more. You can add a layer to a model using the model's add() method. For example, a simple model with a single hidden layer might look like this:

In [3]:

```
import numpy as np
from keras.layers.core import Dense, Activation
```

In [4]:

```
# X has shape (num_rows, num_cols), where the training data are store
# as row vectors
X = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=np.float32)

# y must have an output vector for each input vector
y = np.array([[0], [0], [0], [1]], dtype=np.float32)

# 1st Layer - Add an input layer of 32 nodes with the same input shape as
# the training samples in X
model.add(Dense(32, input_dim=X.shape[1]))

# Add a softmax activation layer
model.add(Activation('softmax'))

# 2nd layer - Add a fully connected output layer
model.add(Dense(1))

# Add a sigmoid activation layer
model.add(Activation('sigmoid'))
```

Keras requires the input shape to be specified in the first layer, but it will automatically infer the shape of all other layers. This means you only have to explicitly set the input dimensions for the first layer.

The first (hidden) layer from above, **model.add(Dense(32, input_dim=X.shape[1]))**, creates 32 nodes which each expect to receive 2-element vectors as inputs. Each layer takes the outputs from the previous layer as inputs and pipes through to the next layer. This chain of passing output to the next layer continues until the last layer, which is the output of the model. We can see that the output has dimension 1.

The activation "layers" in Keras are equivalent to specifying an activation function in the Dense layers (e.g., **model.add(Dense(128)); model.add(Activation('softmax'))** is computationally equivalent to model.add(Dense(128, activation="softmax")))), but it is common to explicitly separate the activation layers because it allows direct access to the outputs of each layer before the activation is applied (which is useful in some model architectures).

Once we have our model built, we need to compile it before it can be run. Compiling the Keras model calls the backend (tensorflow, theano, etc.) and binds the optimizer, loss function, and other parameters required before the model can be run on any input data. We'll specify the loss function to be **binary_crossentropy** which can be used when there are only two classes, and specify **adam** as the optimizer (which is a reasonable default when speed is a priority). And finally, we can specify what metrics we want to evaluate the model with. Here we'll use accuracy.

In [19]:
```
model.compile(loss="binary_crossentropy", optimizer="adam", metrics = ["accuracy"])
```

We can see the resulting model architecture with the following command:

In [20]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                96
_________________________________________________________________
activation_1 (Activation)    (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 129.0
Trainable params: 129.0
Non-trainable params: 0.0
_________________________________________________________________
```

The model is trained with the **fit()** method, through the following command that specifies the number of training epochs and the message level (how much information we want displayed on the screen during training).

Note: In Keras 1, **nb_epoch** sets the number of epochs, but in Keras 2 this changes to the keyword **epochs**.

In [21]:
```
model.fit(X, y, epochs=1000, verbose=0)
```

Out[21]:
```
<keras.callbacks.History at 0x1d87169a908>
```

Finally, we can use the following command to evaluate the model:

In [23]:
```
model.evaluate(X,y)
```
```
4/4 [==============================] - 0s
```

Out[23]:
```
[0.28967487812042236, 1.0]
```

Quiz

Let's start with the simplest example. In this quiz you will build a simple multi-layer feedforward neural network to solve the XOR problem.

1. Set the first layer to a Dense() layer with an output width of 8 nodes and the input_dim set to the size of the training samples (in this case 2).
2. Add a tanh activation function.
3. Set the output layer width to 1, since the output has only two classes. (We can use 0 for one class an 1 for the other)
4. Use a sigmoid activation function after the output layer.
5. Run the model for 50 epochs.

In [28]:

```
import numpy as np
from keras.utils import np_utils
import tensorflow as tf
#tf.python.control_flow_ops = tf
np.random.seed(42)
```

In [29]:

```
# Our data
X = np.array([[0,0],[0,1],[1,0],[1,1]]).astype('float32')
y = np.array([[0],[1],[1],[0]]).astype('float32')
```

In [31]:

```
# Building the model
xor = Sequential()
```

In [32]:

```
# Add required layers
xor.add(Dense(8, input_dim=X.shape[1]))
xor.add(Activation('tanh'))
xor.add(Dense(1))
xor.add(Activation('sigmoid'))

# Specify loss as "binary_crossentropy", optimizer as "adam"
# and add the accuracy metric
xor.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

# Print the model architecture
xor.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 8)                 24
_________________________________________________________________
activation_3 (Activation)    (None, 8)                 0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 9
_________________________________________________________________
activation_4 (Activation)    (None, 1)                 0
=================================================================
Total params: 33.0
Trainable params: 33.0
Non-trainable params: 0.0
```

This should give you an accuracy of 50%. That's ok, but certainly not great. Out of 4 input points, we're correctly classifying only 2 of them. Let's try to change some parameters around to improve. For example, you can increase the number of epochs. You'll pass this quiz if you get 75% accuracy. Can you reach 100%?

To get started, review the Keras documentation about models and layers. The Keras example of a **Multi-Layer Perceptron**network is similar to what you need to do here. Use that as a guide, but keep in mind that there will be a number of differences.

In [64]:

```
# Fitting the model
history = xor.fit(X,y,epochs=2355,verbose=0)
```

In [65]:

```
# Scoring the model
score = xor.evaluate(X,y)
```

```
print("/nAccuracy: ", score[-1])
```

```
4/4 [==============================] - 0s
/nAccuracy:  1.0
```

In [66]:

```
# Checking the predictions
print("\nPredictions:")
print(xor.predict_proba(X))
```

```
Predictions:
4/4 [==============================] - 0s
[[  1.05174536e-06]
 [  9.99997020e-01]
 [  9.99998808e-01]
 [  2.13772728e-06]]
```