Keras is cool!

November 13, 2017

```
In [1]: import keras
        import numpy as np
        import ssl
        ssl._create_default_https_context = ssl._create_unverified_context
```

```
Using TensorFlow backend.
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/importlib/_bootstrap.py:205: Ru
  return f(*args, **kwds)
```

1 Keras

Keras is a neural network framework that wraps tensorflow (if you haven't heard of tensorflow it's another neural network framework) and makes it really simple to implement common neural networks. It's philosophy is to make simple things easy (but beware trying to implement uncommon, custom neural networks can be pretty challenging in Keras, for the purposes of this course you will never have to that though so don't worry about it). If you are ever confused during this homework, Keras has really good documentation, so just go to Keras Docs

2 Datasets

Keras has many datasets conviently builtin to the library. We can access them from the `keras.datasets` module. For this homework, we will be using their housing price dataset, their image classification dataset and their movie review sentiment dataset. To get a full list of their datasets, you can go to this link. Keras Datasets. To use their datasets, we just import them and then call `load_data()`, load_data returns two tuples, the first one is training data, and the second one is testing data. See the example below

```
In [2]: from keras.datasets import boston_housing
        (x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

You can also choose the proportion of training data you would like.

```
In [3]: print("Size of training set before: ", x_train.shape)
        (x_train, y_train), (x_test, y_test) = boston_housing.load_data(test_split=0.10)
        print("Size of training set after: ", x_train.shape)
```

```
Size of training set before:  (404, 13)
Size of training set after:  (455, 13)
```

```
In [4]: from keras.utils import normalize
        x_train = normalize(x_train, axis=1)
        x_test = normalize(x_test, axis=1)
```

3 Models

Every thing in Keras starts out with a model. From an initial model, we can add layers, train the model on data, evaluate the model on test sets, etc. We initialize a model with `Sequential()`. Sequential refers to the fact that the model has a sequence of layers. Personally, I have very rarely used anything other than sequential, so I think its all you really need to worry about.

```
In [5]: from keras.models import Sequential
        model = Sequential()
```

Once we have a model, we can add layers to it with `model.add`. Keras has a really good range of layers we can use. For example, if we want a basic fully connected layer we can use `Dense`. I will now run through an example of using Keras to build and train a fully connected neural network for the purposes of regressing on housing prices for the dataset we loaded earlier.

```
In [6]: from keras.layers import Dense
        model.add(Dense(16, input_shape=(13,)))
```

This line of code adds a fully connected layer with 32 neurons. For the first layer of any model we always have to specify the input shape. In our case we will be training a fully connected network on the boston housing data, so each data point has 13 features. That's why we use an input_shape of (13,). The nice part about Keras is other than the input_shape for the first layer, we don't have to worry about shapes the rest of the time, Keras takes care of it. This can be really useful when you are doing complicated convolutions and things like that where working out the input shape to the next layer can be non-trivial.

Now let's add an Activation function to our network after our first fully connected layer.

```
In [7]: from keras.layers import Activation
        model.add(Activation('relu'))
```

Simple as that. We just added a relu activation to the whole layer. To see a list of activation functions available in Keras go to Keras Activations. Now let's add the final layer in our model.

```
In [8]: model.add(Dense(1))
```

Now we can use a handy utility in Keras to print out what our model looks like so far.

```
In [9]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                224
_________________________________________________________________
activation_1 (Activation)    (None, 16)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 241
Trainable params: 241
Non-trainable params: 0
_________________________________________________________________
```

You can see it shows us what layers we have, the output shapes of each layer, and how many parameters there are for each layer. All this information can be really useful when trying to debug a model, or even for sharing your model architechture with others.

4 Training

Now for actually training the model. Before we train a model we have to compile it. `model.compile` is how you specify which optimizer to use and what loss function to use. Sometimes choosing the right optimizer can have a significant effect on model performance. For a list of optimizers look at Keras Optimizers. Choosing the right optimizer is mostly just trying each one to see which works better, there is some general advice for when to use each one but its basically just another hyperparameter. We also have to choose a loss function. Choosing the right loss function is really important since the loss function basically decides what the goal of the model is. Since we are doing regression we want to choose mean squared error, to get our output to be as close as possible to the label.

```
In [10]: model.compile(optimizer='SGD', loss='mean_squared_error')
```

Now we have to actually train our model on the data. This is really easy in Keras, in fact it only takes one line of code.

```
In [11]: model.fit(x_train, y_train, epochs=100)
```

```
Epoch 1/100
455/455 [==============================] - 0s 216us/step - loss: 222.0887
Epoch 2/100
455/455 [==============================] - 0s 19us/step - loss: 70.9517
Epoch 3/100
455/455 [==============================] - 0s 19us/step - loss: 68.6262
Epoch 4/100
455/455 [==============================] - 0s 18us/step - loss: 68.1205
Epoch 5/100
455/455 [==============================] - 0s 20us/step - loss: 65.0651
```

```
Epoch 6/100
455/455 [==============================] - 0s 19us/step - loss: 62.7591
Epoch 7/100
455/455 [==============================] - 0s 19us/step - loss: 69.7109
Epoch 8/100
455/455 [==============================] - 0s 18us/step - loss: 63.9435
Epoch 9/100
455/455 [==============================] - 0s 20us/step - loss: 68.6849
Epoch 10/100
455/455 [==============================] - 0s 19us/step - loss: 67.2664
Epoch 11/100
455/455 [==============================] - 0s 18us/step - loss: 68.4528
Epoch 12/100
455/455 [==============================] - 0s 20us/step - loss: 63.9322
Epoch 13/100
455/455 [==============================] - 0s 18us/step - loss: 64.7034
Epoch 14/100
455/455 [==============================] - 0s 20us/step - loss: 62.9615
Epoch 15/100
455/455 [==============================] - 0s 19us/step - loss: 65.0559
Epoch 16/100
455/455 [==============================] - 0s 21us/step - loss: 62.3066
Epoch 17/100
455/455 [==============================] - 0s 19us/step - loss: 64.3174
Epoch 18/100
455/455 [==============================] - 0s 21us/step - loss: 63.5996
Epoch 19/100
455/455 [==============================] - 0s 21us/step - loss: 63.2130
Epoch 20/100
455/455 [==============================] - 0s 19us/step - loss: 65.5239
Epoch 21/100
455/455 [==============================] - 0s 19us/step - loss: 63.4187
Epoch 22/100
455/455 [==============================] - 0s 19us/step - loss: 61.5135
Epoch 23/100
455/455 [==============================] - 0s 20us/step - loss: 63.8363
Epoch 24/100
455/455 [==============================] - 0s 19us/step - loss: 61.2746
Epoch 25/100
455/455 [==============================] - 0s 21us/step - loss: 60.7953
Epoch 26/100
455/455 [==============================] - 0s 20us/step - loss: 61.5134
Epoch 27/100
455/455 [==============================] - 0s 21us/step - loss: 62.9285
Epoch 28/100
455/455 [==============================] - 0s 19us/step - loss: 63.2626
Epoch 29/100
455/455 [==============================] - 0s 19us/step - loss: 59.2703
```

```
Epoch 30/100
455/455 [==============================] - 0s 21us/step - loss: 62.6400
Epoch 31/100
455/455 [==============================] - 0s 20us/step - loss: 58.5801
Epoch 32/100
455/455 [==============================] - 0s 19us/step - loss: 64.4447
Epoch 33/100
455/455 [==============================] - 0s 19us/step - loss: 62.9598
Epoch 34/100
455/455 [==============================] - 0s 19us/step - loss: 61.8156
Epoch 35/100
455/455 [==============================] - 0s 19us/step - loss: 61.7386
Epoch 36/100
455/455 [==============================] - 0s 19us/step - loss: 62.8147
Epoch 37/100
455/455 [==============================] - 0s 20us/step - loss: 59.2237
Epoch 38/100
455/455 [==============================] - 0s 20us/step - loss: 58.9978
Epoch 39/100
455/455 [==============================] - ETA: 0s - loss: 60.26 - 0s 18us/step - loss: 60.431
Epoch 40/100
455/455 [==============================] - 0s 20us/step - loss: 60.3835
Epoch 41/100
455/455 [==============================] - 0s 18us/step - loss: 59.2592
Epoch 42/100
455/455 [==============================] - 0s 20us/step - loss: 58.5429
Epoch 43/100
455/455 [==============================] - 0s 19us/step - loss: 59.2077
Epoch 44/100
455/455 [==============================] - 0s 18us/step - loss: 59.3693
Epoch 45/100
455/455 [==============================] - 0s 19us/step - loss: 59.5425
Epoch 46/100
455/455 [==============================] - 0s 20us/step - loss: 60.6734
Epoch 47/100
455/455 [==============================] - 0s 19us/step - loss: 61.3588
Epoch 48/100
455/455 [==============================] - 0s 22us/step - loss: 57.9099
Epoch 49/100
455/455 [==============================] - 0s 19us/step - loss: 59.4303
Epoch 50/100
455/455 [==============================] - 0s 19us/step - loss: 57.5272
Epoch 51/100
455/455 [==============================] - 0s 20us/step - loss: 64.3194
Epoch 52/100
455/455 [==============================] - 0s 19us/step - loss: 61.4115
Epoch 53/100
455/455 [==============================] - 0s 20us/step - loss: 61.1940
```

```
Epoch 54/100
455/455 [==============================] - 0s 20us/step - loss: 59.2822
Epoch 55/100
455/455 [==============================] - 0s 19us/step - loss: 59.3558
Epoch 56/100
455/455 [==============================] - 0s 20us/step - loss: 59.1468
Epoch 57/100
455/455 [==============================] - 0s 20us/step - loss: 65.6341
Epoch 58/100
455/455 [==============================] - 0s 18us/step - loss: 58.1385
Epoch 59/100
455/455 [==============================] - 0s 19us/step - loss: 58.9885
Epoch 60/100
455/455 [==============================] - 0s 20us/step - loss: 56.6659
Epoch 61/100
455/455 [==============================] - 0s 20us/step - loss: 59.8593
Epoch 62/100
455/455 [==============================] - 0s 18us/step - loss: 57.0311
Epoch 63/100
455/455 [==============================] - 0s 19us/step - loss: 61.5732
Epoch 64/100
455/455 [==============================] - 0s 19us/step - loss: 59.3234
Epoch 65/100
455/455 [==============================] - 0s 18us/step - loss: 57.4042
Epoch 66/100
455/455 [==============================] - 0s 19us/step - loss: 56.9180
Epoch 67/100
455/455 [==============================] - 0s 18us/step - loss: 57.3458
Epoch 68/100
455/455 [==============================] - 0s 19us/step - loss: 57.5109
Epoch 69/100
455/455 [==============================] - 0s 19us/step - loss: 59.6554
Epoch 70/100
455/455 [==============================] - 0s 18us/step - loss: 62.2634
Epoch 71/100
455/455 [==============================] - 0s 19us/step - loss: 60.5034
Epoch 72/100
455/455 [==============================] - 0s 19us/step - loss: 61.1353
Epoch 73/100
455/455 [==============================] - 0s 19us/step - loss: 61.3979
Epoch 74/100
455/455 [==============================] - 0s 19us/step - loss: 57.5183
Epoch 75/100
455/455 [==============================] - 0s 19us/step - loss: 61.2670
Epoch 76/100
455/455 [==============================] - 0s 19us/step - loss: 65.5827
Epoch 77/100
455/455 [==============================] - 0s 19us/step - loss: 63.1127
```

```
Epoch 78/100
455/455 [==============================] - 0s 19us/step - loss: 55.3479
Epoch 79/100
455/455 [==============================] - 0s 18us/step - loss: 76.1321
Epoch 80/100
455/455 [==============================] - 0s 19us/step - loss: 57.7357
Epoch 81/100
455/455 [==============================] - 0s 19us/step - loss: 56.0391
Epoch 82/100
455/455 [==============================] - 0s 20us/step - loss: 57.5794
Epoch 83/100
455/455 [==============================] - 0s 19us/step - loss: 58.8991
Epoch 84/100
455/455 [==============================] - 0s 19us/step - loss: 59.0633
Epoch 85/100
455/455 [==============================] - 0s 19us/step - loss: 56.4551
Epoch 86/100
455/455 [==============================] - 0s 19us/step - loss: 57.6249
Epoch 87/100
455/455 [==============================] - 0s 20us/step - loss: 56.8427
Epoch 88/100
455/455 [==============================] - 0s 18us/step - loss: 57.3884
Epoch 89/100
455/455 [==============================] - 0s 19us/step - loss: 56.8882
Epoch 90/100
455/455 [==============================] - 0s 20us/step - loss: 56.9333
Epoch 91/100
455/455 [==============================] - 0s 19us/step - loss: 61.6671
Epoch 92/100
455/455 [==============================] - 0s 19us/step - loss: 55.1219
Epoch 93/100
455/455 [==============================] - 0s 19us/step - loss: 56.7051
Epoch 94/100
455/455 [==============================] - 0s 19us/step - loss: 56.8940
Epoch 95/100
455/455 [==============================] - 0s 18us/step - loss: 57.3250
Epoch 96/100
455/455 [==============================] - 0s 19us/step - loss: 59.0964
Epoch 97/100
455/455 [==============================] - 0s 19us/step - loss: 61.7038
Epoch 98/100
455/455 [==============================] - 0s 19us/step - loss: 56.6185
Epoch 99/100
455/455 [==============================] - 0s 18us/step - loss: 57.6715
Epoch 100/100
455/455 [==============================] - 0s 22us/step - loss: 60.0144
```

```
Out[11]: <keras.callbacks.History at 0x115049c50>
```

5 Evaluation

Now that we have trained our model we can evaluate it on our testing set. It is also just one line of code.

```
In [12]: print("Loss: ", model.evaluate(x_test, y_test, verbose=0))
```

```
Loss:  77.7485351562
```

This loss might seem very high and it is, mostly because there aren't very many training points in the dataset (also no effort was put into finding the best model).

We can also generate predictions for new data that we don't have labels for. Since we don't have new data, I will just demonstrate the idea with our testing data.

```
In [13]: y_predicted = model.predict(x_test)
         print(y_predicted)
```

```
[[ 22.47613525]
 [ 17.21605873]
 [ 21.25405884]
 [ 24.39192963]
 [ 24.08215523]
 [ 17.38596535]
 [ 24.74367714]
 [ 27.50332832]
 [ 23.69153976]
 [ 17.79429626]
 [ 17.49906158]
 [ 13.70019531]
 [ 20.9994545 ]
 [ 23.24501419]
 [ 28.1033802 ]
 [ 15.98524857]
 [ 26.92928314]
 [ 15.51913261]
 [ 17.02788925]
 [ 17.76092529]
 [ 24.65262032]
 [ 17.83653641]
 [ 16.15238762]
 [ 24.93202019]
 [ 22.77094841]
 [ 23.22046471]
 [ 23.39792442]
 [ 27.21434402]
```

```
[ 16.5770359 ]
[ 22.06186295]
[ 24.40295792]
[ 22.72862244]
[ 17.32306671]
[ 22.2161274 ]
[ 22.20412445]
[ 17.57308769]
[ 21.99674606]
[ 22.88215828]
[ 18.19400215]
[ 23.0478363 ]
[ 25.03045845]
[ 24.63732147]
[ 24.79982567]
[ 27.38162804]
[ 19.71040916]
[ 24.08962631]
[ 17.60988235]
[ 21.99514198]
[ 20.71366882]
[ 22.12634468]
[ 16.91773415]]
```

That's it. We have successfully (depending on your definition of success) built a fully connected neural network and trained that network on a dataset. Now its your turn.

6 Problem 1: Image Classification

We are going to build a convolutional neural network to predict image classes on CIFAR-10, a dataset of images of 10 different things (i.e. 10 classes). Things like aeroplanes, cars, deer, horses, etc.

(a) Load the cifar10 dataset from Keras. If you need a hint go to Keras Datasets. This might take a little while to download.

```
In [14]: from keras.datasets import cifar10

         (cifar_x_train, cifar_y_train), (cifar_x_test, cifar_y_test) = cifar10.load_data()
```

(b) Initialize a Sequential model

```
In [15]: cifar_model = Sequential()
```

(c) Add a `Conv2D` layer to the model. It should have 32 filters, a 5x5 kernel, and a 1x1 stride. The documentation here will be your friend for this problem. **Hint:** This is the first layer of the model so you have to specify the input shape. I recommend printing `cifar_x_train.shape`, to get an idea of what the shape of the data looks like. Then add a `relu` activation layer to the model.

```
In [16]: from keras.layers.convolutional import Conv2D
         print(cifar_x_train.shape)
         ##YOUR CODE HERE
         cifar_model.add(Conv2D(32, 5, strides=(1, 1), input_shape = (32, 32, 3)))
         cifar_model.add(Activation('relu'))

(50000, 32, 32, 3)
```

(d) Add a `MaxPooling2D` layer to the model. The layer should have a 2x2 pool size. The documentation for Max Pooling is here.

```
In [17]: from keras.layers.pooling import MaxPooling2D
         ##YOUR CODE HERE
         cifar_model.add(MaxPooling2D(pool_size=(2, 2)))
```

(e) Add another `Conv2D` identical to last one, then another `relu` activation, then another `MaxPooling2D` layer. **Hint:** You've already written this code

```
In [18]: ##YOUR CODE HERE
         cifar_model.add(Conv2D(32, 5, strides=(1, 1)))#, input_shape = (32, 32, 3)))
         cifar_model.add(Activation('relu'))
         cifar_model.add(MaxPooling2D(pool_size=(2, 2)))
```

(f) Add another `Conv2D` layer identical to the others except with 64 filters instead of 32. Add another `relu` activation layer.

```
In [19]: ##YOUR CODE HERE
         cifar_model.add(Conv2D(64, 5, strides=(1, 1)))#, input_shape = (32, 32, 3)))
         cifar_model.add(Activation('relu'))
```

(g) Now we want to move from 2D data to 1D vectors for classification, to this we have to flatten the data. Keras has a layer for this called Flatten. Then add a `Dense` (fully connected) layer with 64 neurons, a `relu` activation layer, another `Dense` layer with 10 neurons, and a `softmax` activation layer.

```
In [20]: from keras.layers import Flatten
         #from keras.activations import softmax
         ##YOUR CODE HERE
         cifar_model.add(Flatten())
         cifar_model.add(Dense(64))
         cifar_model.add(Activation('relu'))
         cifar_model.add(Dense(10))
         cifar_model.add(Activation('softmax'))
```

Notice that we have constructed a network that takes in an image and outputs a vector of 10 numbers and then we take the softmax of these, which leaves us we a vector of 0s except 1 one and the location of this one in the vector corresponds to which class the network is predicting for that image. This is sort of the canonical way of doing image classification.

(h) Now print a summary of your network.

```
In [21]: ##YOUR CODE HERE
         cifar_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        2432
_________________________________________________________________
activation_2 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 32)        25632
_________________________________________________________________
activation_3 (Activation)    (None, 10, 10, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 1, 64)          51264
_________________________________________________________________
activation_4 (Activation)    (None, 1, 1, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                4160
_________________________________________________________________
activation_5 (Activation)    (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0
=================================================================
Total params: 84,138
Trainable params: 84,138
Non-trainable params: 0
_________________________________________________________________
```

(i) We need to convert our labels from integers to length 10 vectors with 9 zeros and 1 one, where the integer label is the index of the 1 in the vector. Luckily, Keras has a handy function to do this for us. Have a look here

```
In [22]: from keras.utils import to_categorical
         y_train_cat = to_categorical(cifar_y_train)
         y_test_cat = to_categorical(cifar_y_test)
```

(j) Now compile the model with SGD optimizer and categorical_crossentropy loss function, also include `metrics=['accuracy']` as a parameter so we can see the accuracy of the model. Then train the model on the training data. For training we want to weight the classes in the loss function, so set the class_weight parameter of fit to be the class_weights dictionary. Be warned training can take forever, I trained on a cpu for 20 epochs (about 30 minutes) and only got 20% accuracy. For the purposes of this assignment you don't need to worry to much about accuracy, just train for at least 1 epoch.

```
In [23]: ##YOUR COMPILING CODE HERE
         cifar_model.compile(optimizer='SGD', loss='categorical_crossentropy', metrics=['accura
```

```
In [24]: class_weights = {}
         for i in range(10):
             class_weights[i] = 1. / np.where(cifar_y_train==i)[0].size

         ##YOUR TRAINING CODE HERE
         cifar_model.fit(cifar_x_train, y_train_cat, epochs=10, class_weight = class_weights)
```

```
Epoch 1/10
50000/50000 [==============================] - 37s 738us/step - loss: 0.0012 - acc: 0.1107
Epoch 2/10
50000/50000 [==============================] - 36s 728us/step - loss: 7.4196e-04 - acc: 0.1315
Epoch 3/10
50000/50000 [==============================] - 36s 724us/step - loss: 5.9141e-04 - acc: 0.1358
Epoch 4/10
50000/50000 [==============================] - 36s 728us/step - loss: 5.4030e-04 - acc: 0.1405
Epoch 5/10
50000/50000 [==============================] - 37s 737us/step - loss: 5.1425e-04 - acc: 0.1438
Epoch 6/10
50000/50000 [==============================] - 38s 762us/step - loss: 4.9840e-04 - acc: 0.1489
Epoch 7/10
50000/50000 [==============================] - 37s 739us/step - loss: 4.8745e-04 - acc: 0.1530
Epoch 8/10
50000/50000 [==============================] - 36s 718us/step - loss: 4.7928e-04 - acc: 0.1586
Epoch 9/10
50000/50000 [==============================] - 38s 751us/step - loss: 4.7271e-04 - acc: 0.1653
Epoch 10/10
50000/50000 [==============================] - 38s 751us/step - loss: 4.6728e-04 - acc: 0.1721
```

```
Out[24]: <keras.callbacks.History at 0x1155d8d30>
```

Now we can evaluate on our test set.

```
In [25]: cifar_model.evaluate(cifar_x_test, y_test_cat)
```

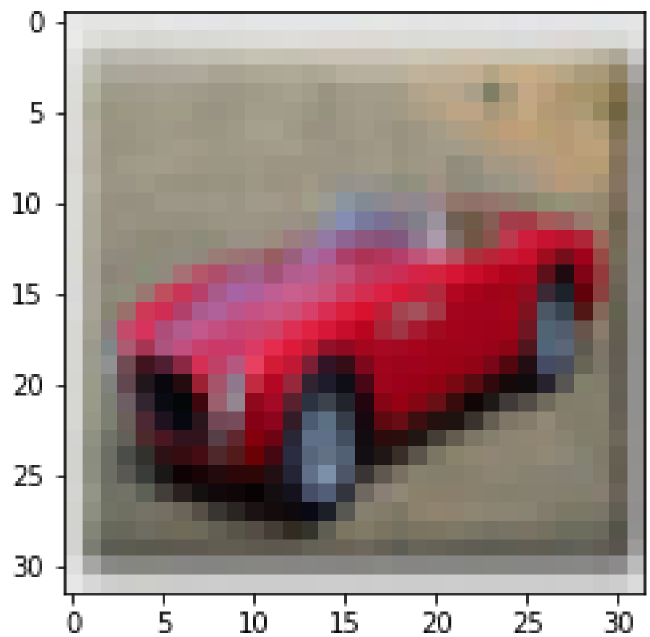
```
10000/10000 [==============================] - 2s 221us/step
```

```
Out[25]: [2.3306676593780518, 0.17199999999999999]
```

We can also get the class labels the network predicts on our test set and look at a few examples.

```
In [26]: y_pred = cifar_model.predict(cifar_x_test)
         import matplotlib.pyplot as plt
         %matplotlib inline
         plt.imshow(cifar_x_test[1234])
         print("Predicted label: ", np.argmax(y_pred[1234]))
         print("True label: ", cifar_y_test[1234])
```

```
Predicted label:  7
True label:  [1]
```



7 Problem 2: Sentiment Classification

In this problem we will use Kera's imdb sentiment dataset. You will take in sequences of words and use an RNN to try to classify the sequences sentiment. First we have to process the data a little bit, so that we have fixed length sequences.

```
In [27]: from keras.datasets import imdb
         (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=1000, maxlen=200)
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/imdb.npz
17465344/17464789 [==============================] - 5s 0us/step
```

```
In [28]: def process_data(data):
             processed = np.zeros(len(data) * 200).reshape((len(data), 200))
```

```
            for i, seq in enumerate(data):
                if len(seq) < 200:
                    processed[i] = np.array(seq + [0 for _ in range(200 - len(seq))])
                else:
                    processed[i] = np.array(seq)
            return processed
```

```
In [29]: x_train_proc = process_data(x_train)
         x_test_proc = process_data(x_test)
         print(x_test_proc.shape)
```

```
(3913, 200)
```

The Embedding Layer is a little bit different from most of the layers, so we have provided that code for you. Basically, the 1000 means that we are using a vocabulary size of 1000, the 32 means we will have a vector of size 32 outputed, and the mask zero means that we don't care about 0, because we are using it for padding.

```
In [30]: imdb_model = Sequential()
```

```
In [31]: from keras.layers.embeddings import Embedding
         imdb_model.add(Embedding(1000, 32, input_length=200, mask_zero=True))
```

(a) For this problem, I won't walk you everything like I did in the last one. What you need to do is as follows. Add an LSTM layer with 32 outputs, then a Dense layer with 16 neurons, then a relu activation, then a dense layer with 1 neuron, then a sigmoid activation. Then you should print out the model summary.

```
In [32]: ##YOUR CODE HERE
         from keras.layers import LSTM
         imdb_model.add(LSTM(32))
         imdb_model.add(Dense(16))
         imdb_model.add(Activation('relu'))
         imdb_model.add(Dense(1))
         imdb_model.add(Activation('sigmoid'))
         imdb_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 200, 32)           32000
_________________________________________________________________
lstm_1 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_5 (Dense)              (None, 16)                528
_________________________________________________________________
activation_7 (Activation)    (None, 16)                0
_________________________________________________________________
```

```
dense_6 (Dense)              (None, 1)                 17
_________________________________________________________________
activation_8 (Activation)    (None, 1)                 0
=================================================================
Total params: 40,865
Trainable params: 40,865
Non-trainable params: 0
_________________________________________________________________
```

(b) Now compile the model with binary cross entropy, and the adam optimizer. Also include accuracy as a metric in the compile. Then train the model on the processed data (no need to worry about class weights this time)

```
In [33]: ##YOUR CODE HERE
         imdb_model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
         imdb_model.fit(x_train_proc, y_train, epochs = 10)
```

```
Epoch 1/10
25000/25000 [==============================] - 82s 3ms/step - loss: 0.4288 - acc: 0.7987
Epoch 2/10
25000/25000 [==============================] - 81s 3ms/step - loss: 0.3230 - acc: 0.8637
Epoch 3/10
25000/25000 [==============================] - 82s 3ms/step - loss: 0.3028 - acc: 0.8730
Epoch 4/10
25000/25000 [==============================] - 81s 3ms/step - loss: 0.2933 - acc: 0.8753
Epoch 5/10
25000/25000 [==============================] - 81s 3ms/step - loss: 0.2733 - acc: 0.8867
Epoch 6/10
25000/25000 [==============================] - 81s 3ms/step - loss: 0.2597 - acc: 0.8925
Epoch 7/10
25000/25000 [==============================] - 81s 3ms/step - loss: 0.2470 - acc: 0.8984
Epoch 8/10
25000/25000 [==============================] - 80s 3ms/step - loss: 0.2362 - acc: 0.9015
Epoch 9/10
25000/25000 [==============================] - 80s 3ms/step - loss: 0.2221 - acc: 0.9108
Epoch 10/10
25000/25000 [==============================] - 80s 3ms/step - loss: 0.2112 - acc: 0.9164
```

```
Out[33]: <keras.callbacks.History at 0x140bae5c0>
```

After training we can evaluate our model on the test set.

```
In [34]: print("Accuracy: ", imdb_model.evaluate(x_test_proc, y_test)[1])
```

```
3913/3913 [==============================] - 3s 693us/step
Accuracy:  0.87682085358
```

Now we can look at our predictions and the sentences they correspond to.

```
In [35]: y_pred = imdb_model.predict(x_test_proc)

In [36]: y_pred = np.vectorize(lambda x: int(x >= 0.5))(y_pred)
         correct = []
         incorrect = []
         for i, pred in enumerate(y_pred):
             if y_test[i] == pred:
                 correct.append(i)
             else:
                 incorrect.append(i)
         word_dict = inv_map = {v: k for k, v in imdb.get_word_index().items()}

         print(list(map(lambda x: word_dict[int(x)] if x != 0 else None, x_test[correct[123]]))

Downloading data from https://s3.amazonaws.com/text-datasets/imdb_word_index.json
1646592/1641221 [==============================] - 1s 0us/step
['the', 'out', 'when', 'of', 'young', 'street', 'funny', 'and', 'and', 'and', 'and', 'and', 't
```

After making this I realized that keras' method for converting from word index back to words is broken right now (see this open github issue). So we can't actually see what the sentences look like.