

In [1]:

```
import keras
import numpy as np
```

Using TensorFlow backend.

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 \(https://keras.io\)](https://keras.io)

Datasets

Keras has many datasets conveniently 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 \(https://keras.io/datasets\)](https://keras.io/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()
```

```
Downloading data from https://s3.amazonaws.com/keras-datasets/boston_housing.npz (https://s3.amazonaws.com/keras-datasets/boston_housing.npz)
57344/57026 [=====] - 0s 2us/step
65536/57026 [=====] - 0s 1us/step
```

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)
```

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 \(https://keras.io/activations/\)](https://keras.io/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 architecture with others.

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 \(https://keras.io/optimizers\)](https://keras.io/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 567us/step - loss: 222
.0887
Epoch 2/100
455/455 [=====] - 0s 59us/step - loss: 70.9
517
Epoch 3/100
455/455 [=====] - 0s 62us/step - loss: 68.6
262
Epoch 4/100
455/455 [=====] - 0s 53us/step - loss: 68.1
205
Epoch 5/100
455/455 [=====] - 0s 61us/step - loss: 65.0
651
Epoch 6/100
455/455 [=====] - 0s 63us/step - loss: 62.7
591
Epoch 7/100
455/455 [=====] - 0s 60us/step - loss: 60.7
517
```

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: ', 77.772160548789827)
```

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)
```

```
[[ 22.47864723]
 [ 17.21179581]
 [ 21.25464058]
 [ 24.37393761]
 [ 24.12023926]
 [ 17.37979698]
 [ 24.73394585]
 [ 27.53306198]
 [ 23.72990036]
 [ 17.78282166]
 [ 17.4911499 ]
 [ 13.67580032]
 [ 20.99804688]
 [ 23.24216461]
 [ 28.07510376]
 [ 15.97411633]
 [ 26.91640663]
 [ 15.5116539 ]
 [ 17.02495956]
 [ 17.75162607]
```

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.

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 \(https://keras.io/datasets\)](https://keras.io/datasets). This might take a little while to download.

In [15]:

```
from keras.datasets import cifar10
(cifar_x_train, cifar_y_train), (cifar_x_test, cifar_y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
170500096/170498071 [=====] - 227s 1us/step
- ETA: 2:14 - ETA: 1:41
170508288/170498071 [=====] - 227s 1us/step
```

(b) Initialize a Sequential model

In [32]:

```
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](https://keras.io/layers/convolutional/#conv2d) (https://keras.io/layers/convolutional/#conv2d) 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 [33]:

```
print(cifar_x_train.shape)
```

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

In [34]:

```
from keras.layers.convolutional import Conv2D
print(cifar_x_train.shape)
cifar_model.add(Conv2D(filters = 32,
                        kernel_size = (5, 5),
                        activation = 'relu',
                        padding = 'same',
                        input_shape = (32,32,3)))
```

```
(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](https://keras.io/layers/pooling/) (https://keras.io/layers/pooling/).

In [35]:

```
from keras.layers.pooling import MaxPooling2D
cifar_model.add(MaxPooling2D(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 [36]:

```
##YOUR CODE HERE
cifar_model.add(Conv2D(filters = 32,
                        kernel_size = (5, 5),
                        activation = 'relu',
                        padding = 'same'))
```

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

In [37]:

```
##YOUR CODE HERE
cifar_model.add(MaxPooling2D(2,2))
```

(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` (<https://keras.io/layers/core/#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 [39]:

```
from keras.layers import Flatten
##YOUR CODE HERE
cifar_model.add(Flatten())
cifar_model.add(Dense(64, activation = 'relu'))
cifar_model.add(Dense(10, 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 [40]:

```
##YOUR CODE HERE
cifar_model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 32, 32, 32)	2432
max_pooling2d_4 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_9 (Conv2D)	(None, 16, 16, 32)	25632
max_pooling2d_5 (MaxPooling2D)	(None, 8, 8, 32)	0
flatten_2 (Flatten)	(None, 2048)	0
dense_3 (Dense)	(None, 64)	131136
dense_4 (Dense)	(None, 10)	650
Total params: 159,850		
Trainable params: 159,850		
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](https://keras.io/utils/#to_categorical) (https://keras.io/utils/#to_categorical).

In [42]:

```
from keras.utils import to_categorical
y_train_cat = to_categorical(cifar_y_train,num_classes=10)
y_test_cat = to_categorical(cifar_y_test,num_classes=10)
```

(ii) 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 too much about accuracy, just train for at least 1 epoch.

In [45]:

```
##YOUR COMPILING CODE HERE
cifar_model.compile(optimizer = 'SGD', loss = 'categorical_crossentropy', metrics
```

In [47]:

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

##YOUR TRAINING CODE HERE
history = cifar_model.fit(cifar_x_train, y_train_cat, batch_size = 32, validation
```

Train on 50000 samples, validate on 10000 samples

Epoch 1/3

50000/50000 [=====] - 209s 4ms/step - loss: 0.0025 - acc: 0.1108 - val_loss: 9.8674 - val_acc: 0.1402 - ETA: 59s - loss: 0.0026 - acc: 0.1016

Epoch 2/3

50000/50000 [=====] - 218s 4ms/step - loss: 0.0016 - acc: 0.1588 - val_loss: 6.7438 - val_acc: 0.1635 - ETA: 2:41 - loss: 0.0018 - acc: 0.1478

Epoch 3/3

50000/50000 [=====] - 246s 5ms/step - loss: 9.1540e-04 - acc: 0.1460 - val_loss: 3.5700 - val_acc: 0.1363 - ETA: 31s - loss: 9.4789e-04 - acc: 0.1459

Now we can evaluate on our test set.

In [48]:

```
cifar_model.evaluate(cifar_x_test, y_test_cat)
```

10000/10000 [=====] - 14s 1ms/step

Out[48]:

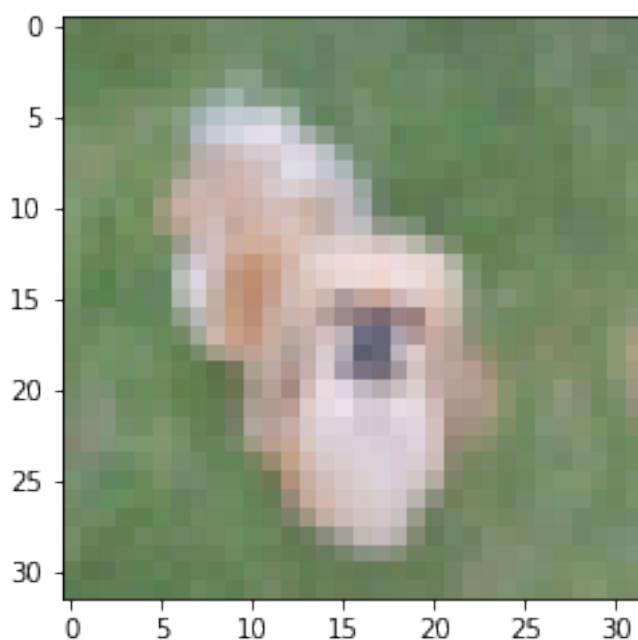
```
[3.570005874633789, 0.1363]
```

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

In [52]:

```
y_pred = cifar_model.predict(cifar_x_test)
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(cifar_x_test[1235])
print("Predicted label: ", np.argmax(y_pred[1235]))
print("True label: ", cifar_y_test[1235])
```

```
('Predicted label: ', 0)
('True label: ', array([5]))
```



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 [51]:

```
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
z (https://s3.amazonaws.com/text-datasets/imdb.npz)
17465344/17464789 [=====] - 6s 0us/step
17473536/17464789 [=====] - 6s 0us/step
```

In [53]:

```
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 [54]:

```
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 [55]:

```
imdb_model = Sequential()
```

In [56]:

```
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 [61]:

```
##YOUR CODE HERE
from keras.layers import LSTM
imdb_model.add(LSTM(32))
imdb_model.add(Dense(16, activation='relu'))
imdb_model.add(Dense(1, 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

dense_6 (Dense)	(None, 1)	17
=====		
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 [63]:

```
##YOUR CODE HERE
imdb_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
imdb_model.fit(x_train_proc, y_train,
               validation_data = (x_test_proc, y_test),
               epochs=3,
               batch_size=64)
```

```
Train on 25000 samples, validate on 3913 samples
Epoch 1/3
25000/25000 [=====] - 92s 4ms/step - loss: 0.4400 - acc: 0.7830 - val_loss: 0.3214 - val_acc: 0.8594
Epoch 2/3
25000/25000 [=====] - 94s 4ms/step - loss: 0.3098 - acc: 0.8688 - val_loss: 0.3105 - val_acc: 0.8612
Epoch 3/3
25000/25000 [=====] - 93s 4ms/step - loss: 0.2952 - acc: 0.8762 - val_loss: 0.2984 - val_acc: 0.8740
```

Out[63]:

```
<keras.callbacks.History at 0x11ccda750>
```

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

In [64]:

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

```
3913/3913 [=====] - 6s 1ms/step  
( 'Accuracy: ', 0.87400971126471094)
```

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

In [65]:

```
y_pred = imdb_model.predict(x_test_proc)
```

In [66]:

```
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[12
```

Downloading data from https://s3.amazonaws.com/text-datasets/imdb_word_index.json (https://s3.amazonaws.com/text-datasets/imdb_word_index.json)

```
1646592/1641221 [=====] - 1s 1us/step  
1654784/1641221 [=====] - 1s 1us/step  
[u'the', u'as', u'on', u'and', u'we', u'br', u'of', u'person', u'and',  
, u'br', u'of', u'and', u'his', u'just', u'of', u'and', u'really',  
u'turn', u'up', u'br', u'is', u'and', u'been', u'last', u'fan', u'no  
t', u'is', u'wasn't', u'and', u'lives', u'for', u'it', u'give', u'an  
d', u'to', u'between', u'and', u'and', u'and', u'it', u'up', u'been',  
, u'and', u'movie', u'of', u'and', u'and', u'our', u'so', u'and', u'  
film', u'about', u'and', u'anyone', u'who', u'and', u'and', u'to', u'  
'and', u'and', u'it', u'small', u'movie', u'about', u'and', u'and',  
u'about', u'sort', u'of', u'and', u'and', u'was', u'although', u'exc  
ept', u'and', u'but', u'were', u'and', u'in', u'credits', u'what', u'  
'see', u'most', u'from', u'few', u'her', u'very', u'american', u'as',  
, u'on', u'it', u'events', u'fans', u'are', u'is', u'wasn't', u'thei  
r', u'to', u'if', u'is', u'and', u'in', u'at', u'and', u'in', u'two',  
, u'has', u'now', u'and', u'but', u'is', u'taking', u'i', u'i', u'vi  
ewer', u'who', u'and', u'like', u'other', u'off', u'of', u'and', u'b  
r', u'and', u'and', u'film', u'and', u'to', u'of', u'and', u'and', u'  
'original', u'as', u'it', u'is', u'on', u'film', u'and', u'to', u'gi  
ves', u'to', u'it', u'boy', u'main', u'wife']
```

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 \(https://github.com/fchollet/keras/issues/5912\)](https://github.com/fchollet/keras/issues/5912)). So we can't actually see what the sentences look like.

