Introduction to Deep Learning with Keras

Deeper machine learning using the Keras library

Black Raven (James Ng)
28 Feb 2021 · 51 min read

This is a memo to share what I have learnt in Introduction to Deep Learning with Keras, capturing the learning objectives as well as my personal notes. The course is taught by Miguel Esteban from DataCamp, and it includes 4 chapters:

Chapter 1: Introducing Keras
Chapter 2: Going Deeper
Chapter 3: Improving Your Model Performance
Chapter 4: Advanced Model Architectures

Deep learning is here to stay! It's the go-to technique to solve complex problems that arise with unstructured data and an incredible tool for innovation. Keras is one of the frameworks that make it easier to start developing deep learning models, and it's versatile enough to build industry-ready models in no time.

In this course, you will learn regression and save the earth by predicting asteroid trajectories, apply binary classification to distinguish between real and fake dollar bills, use multiclass classification to decide who threw which dart at a dart board, learn to use neural networks to reconstruct noisy images and much more. Additionally, you will learn how to better control your models during training and how to tune them to boost their performance.

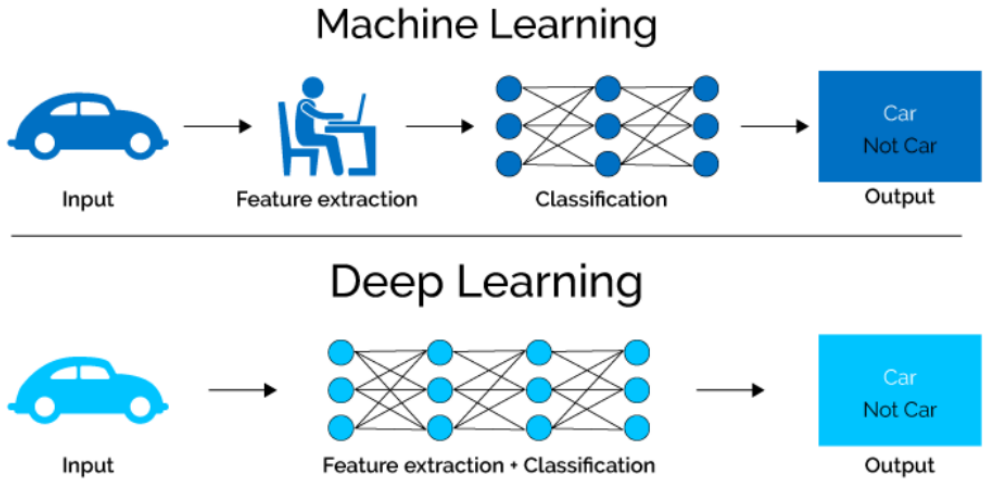
Chapter 1. Introducing Keras

In this first chapter, you will get introduced to neural networks, understand what kind of problems they can solve, and when to use them. You will also build several networks and save the earth by training a regression model that approximates the orbit of a meteor that is approaching us!

What is Keras?

Theano = low level deep learning framework, require many lines of codes and algorithm understanding
Keras = high level deep learning framework, few lines of codes

- open source, fast model building for experientation
- runs on top of other frameworks eg Theano, TensorFlow, CNTK
- build simple/complex architecture eg auto-encoders, convolutional, recurrent neural networks
- can be deployed across multiple platforms eg Android, iOS, web-apps
- Keras is fully integrated into TensorFlow 2,
- choice to use TensorFlow low level feature eg control of how model applies gradients



Domain expert needs to set rules based on experimentation and heuristics to extract relevant features of data.

Neural networks can learn the best features and their combination, and can perform feature engineering within the network.

Unstructured data is info that cannot be put into a table, eg. sound, video, images. Such data pose a big challenge to feature engineering, thus use neural network.

When to use neural networks?

- Dealing with unstructured data
- No need to interpret/explain how the results are achieved
- Benefit from using a know architecture, eg. convolutional neural network
- Eg. classification images of cats and dogs

Describing Keras

Which of the following statements about Keras is **false**?

- Keras is integrated into TensorFlow, that means you can call Keras from within TensorFlow and get the best of both worlds.
- Keras can work well on its own without using a backend, like TensorFlow.
- Keras is an open source project started by François Chollet.

Keras is a wrapper around a backend library, so a backend like TensorFlow, Theano, CNTK, etc must be provided.

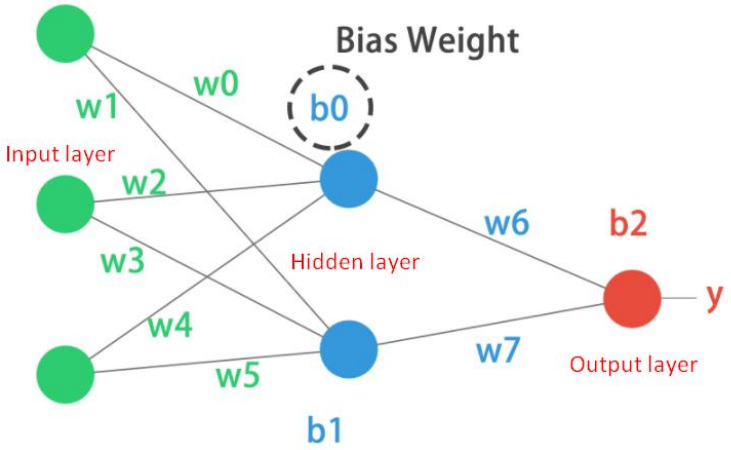
Would you use deep learning?

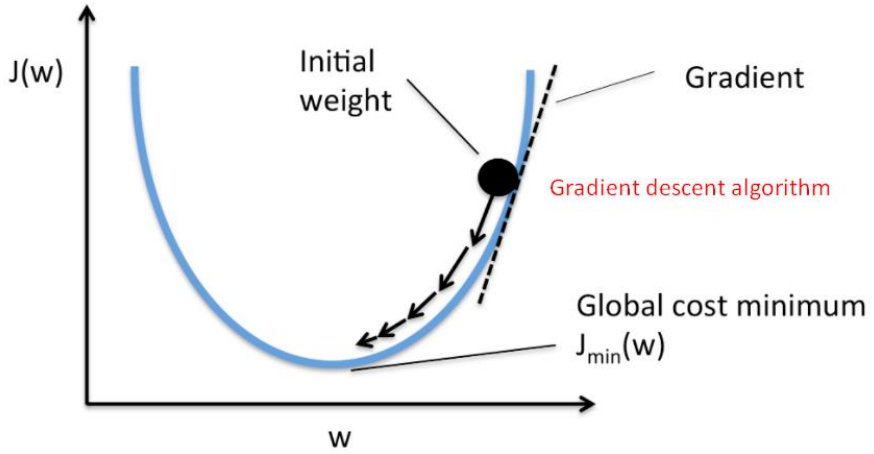
Imagine you're building an app that allows you to take a picture of your clothes and then shows you a pair of shoes that would match well. This app needs a machine learning module that's in charge of identifying the type of clothes you are wearing, as well as their color and texture. Would you use deep learning to accomplish this task?

- I'd use deep learning, since we are dealing with tabular data and neural networks work well with images.
- I'd use deep learning since we are dealing with unstructured data and neural networks work well with images.
- This task can be easily accomplished with other machine learning algorithms, so deep learning is not required.

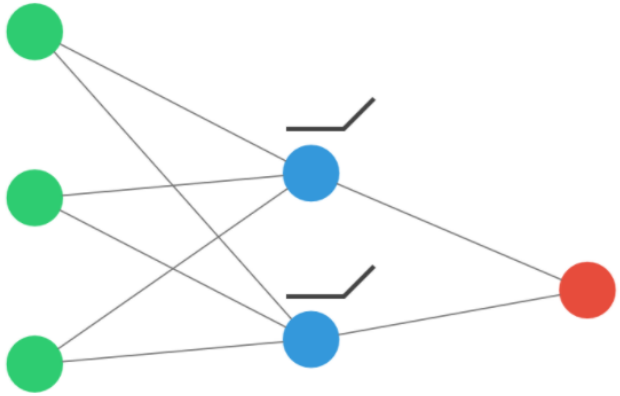
Using deep learning would be the easiest way. The model would generalize well if enough clothing images are provided.

Your first neural network






```
from keras.models import Sequential
from keras.layers import Dense
# Create a new sequential model
model = Sequential()
# Add and input and dense layer
model.add(Dense(2, input_shape=(3,),
                activation="relu"))
# Add a final 1 neuron layer
model.add(Dense(1))
```



```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 2)                 8
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 3
=================================================================
Total params: 11
Trainable params: 11
Non-trainable params: 0
_________________________________________________________________
```

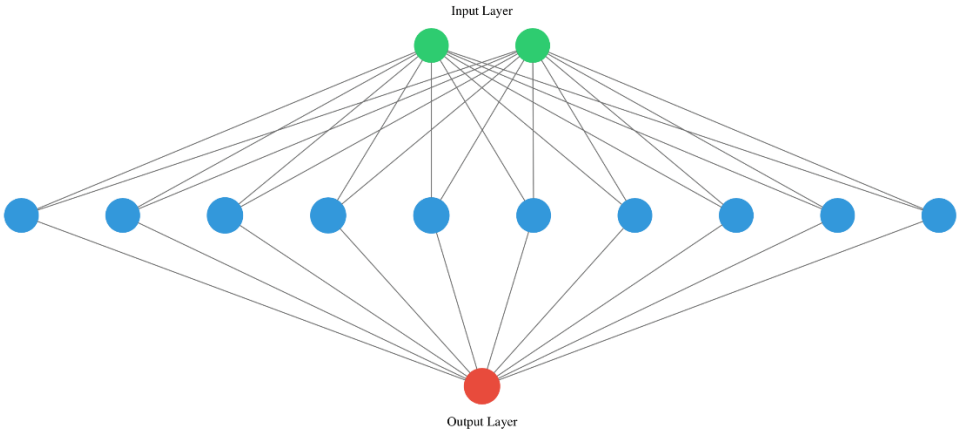
dense_3 with 8 parameters:
w0, w1, w2, w3, w4 w5, b0, b1

dense_4 with 3 parameters:
w6, w7, b2

Hello nets!

You're going to build a simple neural network to get a feeling of how quickly it is to accomplish this in Keras. You will build a network that **takes two numbers as an input**, passes them through **a hidden layer of 10 neurons**, and finally **outputs a single non-constrained number**.

A **non-constrained output can be obtained by avoiding setting an activation function in the output layer**. This is useful for problems like regression, when we want our output to be able to take any non-constrained value.



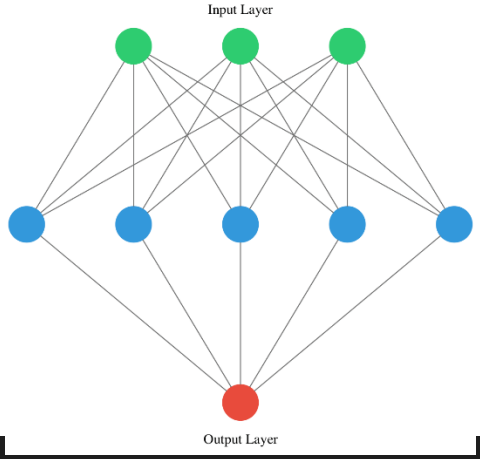

```
# Import the Sequential model and Dense layer
from keras.models import Sequential
from keras.layers import Dense

# Create a Sequential model
model = Sequential()

# Add an input layer and a hidden layer with 10 neurons
model.add(Dense(10, input_shape=(2,), activation="relu"))

# Add a 1-neuron output layer
model.add(Dense(1))

# Summarise your model
model.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 10)                30
    _________________________________________________________________
    dense_2 (Dense)              (None, 1)                 11
    =================================================================
    Total params: 41
    Trainable params: 41
    Non-trainable params: 0
    _________________________________________________________________
```

Counting parameters

You've just created a neural network. But you're going to create a new one now, taking some time to think about the weights of each layer. The Keras `Dense` layer and the `Sequential` model are already loaded for you to use.

This is the network you will be creating:




```
# Instantiate a new Sequential model
model = Sequential()

# Add a Dense layer with five neurons and three inputs
model.add(Dense(5, input_shape=(3,), activation="relu"))

# Add a final Dense layer with one neuron and no activation
model.add(Dense(1))

# Summarize your model
model.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 5)                 20
    _________________________________________________________________
    dense_2 (Dense)              (None, 1)                 6
    =================================================================
    Total params: 26
    Trainable params: 26
    Non-trainable params: 0
    _________________________________________________________________
```

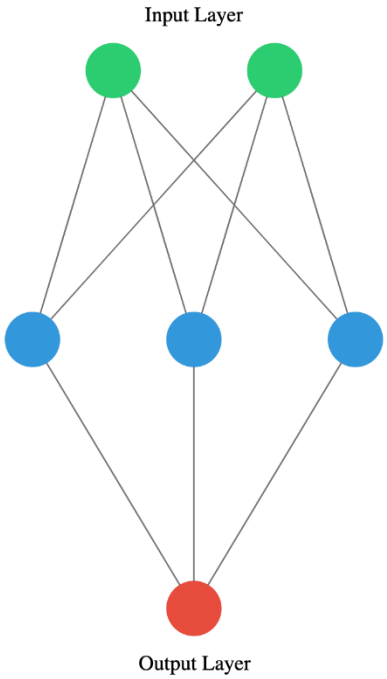
Given the `model` you just built, which answer is correct regarding the number of weights (parameters) in the **hidden layer**?

- ○ There are 15 parameters, 3 for every neuron in the hidden layer.
- ◉ There are 20 parameters, 15 from the connections of our inputs to our hidden layer and 5 from the bias weight of each neuron in the hidden layer.
- ○ There are 20 parameters, no bias weights were needed in this simple model.

Build as shown!

You will take on a final challenge before moving on to the next lesson.

Build the network shown in the picture below. Prove your mastered Keras basics in no time!




```
from keras.models import Sequential
from keras.layers import Dense

# Instantiate a Sequential model
model = Sequential()

# Build the input and hidden layer
model.add(Dense(3, input_shape=(2,)))

# Add the ouput layer
model.add(Dense(1))
```

You've shown you can already translate a visual representation of a neural network into Keras code. Let's keep going!

Surviving a meteor strike

Compiling model

```
# Compiling your previously built model
model.compile(optimizer="adam", loss="mse")
```

Training model

```
# Train your model
model.fit(X_train, y_train, epochs=5)
```

```
Epoch 1/5
1000/1000 [==============================] - 0s 242us/step - loss: 0.4090
Epoch 2/5
1000/1000 [==============================] - 0s 34us/step - loss: 0.3602
Epoch 3/5
1000/1000 [==============================] - 0s 37us/step - loss: 0.3223
Epoch 4/5
1000/1000 [==============================] - 0s 34us/step - loss: 0.2958
Epoch 5/5
1000/1000 [==============================] - 0s 33us/step - loss: 0.2795
```

Predicting

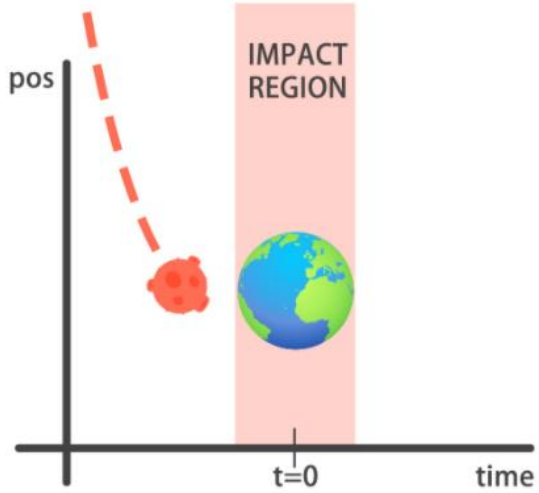
```
# Predict on new data
preds = model.predict(X_test)

# Look at the predictions
print(preds)
```

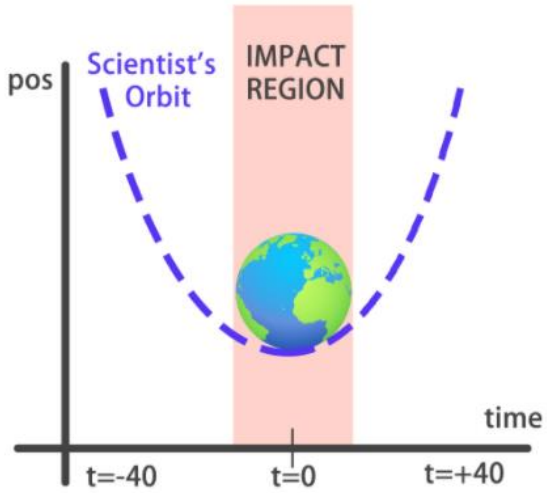
```
array([[0.6131608 ],
       [0.5175948 ],
       [0.60209155],
       ...,
       [0.55633   ],
       [0.5305591 ],
       [0.50682044]])
```

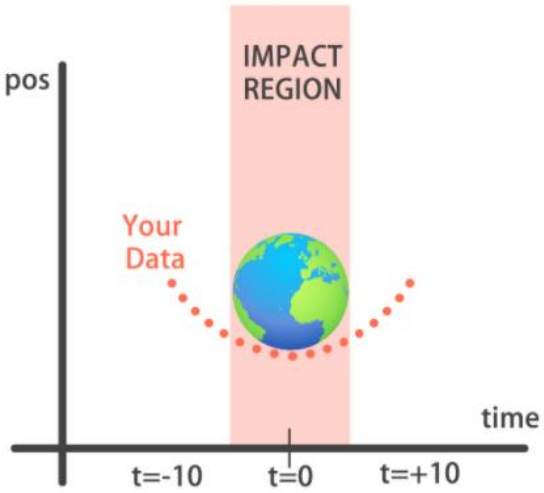
Evaluating

```
# Evaluate your results
model.evaluate(X_test, y_test)
```

```
1000/1000 [==============================] - 0s 53us/step
0.25
```

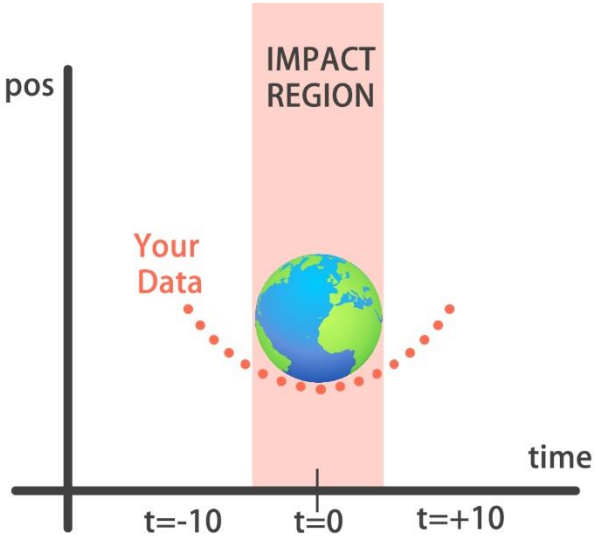






Specifying a model

You will build a simple regression model to predict the orbit of the meteor!



Your training data consist of measurements taken at time steps from **-10 minutes before the impact region to +10 minutes after**. Each time step can be viewed as an X coordinate in our graph, which has an associated position Y for the meteor orbit at that time step.

Note that you can view this problem as approximating a quadratic function via the use of neural networks.

This data is stored in two numpy arrays: one called `time_steps` , what we call *features*, and another called `y_positions`, with the *labels*. Go on and build your model! It should be able to predict the y positions for the meteor orbit at future time steps.

Keras `Sequential` model and `Dense` layers are available for you to use.

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

# Add a Dense layer with 50 neurons and an input of 1 neuron
model.add(Dense(50, input_shape=(1,), activation='relu'))

# Add two Dense layers with 50 neurons and relu activation
model.add(Dense(50, activation='relu'))
model.add(Dense(50, activation='relu'))

# End your model with a Dense layer and no activation
model.add(Dense(1))
```

You are closer to forecasting the meteor orbit! It's important to note we aren't using an activation function in our output layer since `y_positions` aren't bounded and they can take any value. Your model is built to perform a regression task.

Training

You're going to train your first model in this course, and for a good cause!

Remember that **before training your Keras models you need to compile them**. This can be done with the `.compile()` method.
The `.compile()` method takes arguments such as the `optimizer`, used for weight updating, and the `loss` function, which is what we want to minimize. Training your model is as easy as calling the `.fit()` method, passing on the *features*, *labels* and a number of *epochs* to train for.

The regression `model` you built in the previous exercise is loaded for you to use, along with the `time_steps` and `y_positions` data. Train it and evaluate it on this very same data, let's see if your model can learn the meteor's trajectory.

```
# Compile your model
model.compile(optimizer = 'adam', loss = 'mse')

print("Training started..., this can take a while:")

# Fit your model on your data for 30 epochs
model.fit(time_steps, y_positions, epochs = 30)

# Evaluate your model
print("Final loss value:",model.evaluate(time_steps, y_positions))
```

```
<script.py> output:
    Training started..., this can take a while:
    Epoch 1/30

      32/2000 [..............................] - ETA: 12s - loss: 2465.2439
     928/2000 [============>.................] - ETA: 0s - loss: 1819.5736
    1696/2000 [========================>.....] - ETA: 0s - loss: 1510.8248
    2000/2000 [==============================] - 0s 163us/step - loss: 1369.0031
    Epoch 2/30

      32/2000 [..............................] - ETA: 0s - loss: 514.3895
    2000/2000 [==============================] - 0s 68us/step - loss: 0.2241
    Epoch 30/30

      32/2000 [..............................] - ETA: 0s - loss: 0.1710
     896/2000 [============>.................] - ETA: 0s - loss: 0.2499
    1888/2000 [===========================>..] - ETA: 0s - loss: 0.2371
    2000/2000 [==============================] - 0s 56us/step - loss: 0.2347

      32/2000 [..............................] - ETA: 1s
    2000/2000 [==============================] - 0s 31us/step
    Final loss value: 0.15582470980100332
```

You can check the console to see how the loss function decreased as epochs went by. Your model is now ready to make predictions on unseen data.

Predicting the orbit!

You've already trained a `model` that approximates the orbit of the meteor approaching Earth and it's loaded for you to use.

Since you trained your model for values between -10 and 10 minutes, your model hasn't yet seen any other values for different time steps. You will now visualize how your model behaves on unseen data.

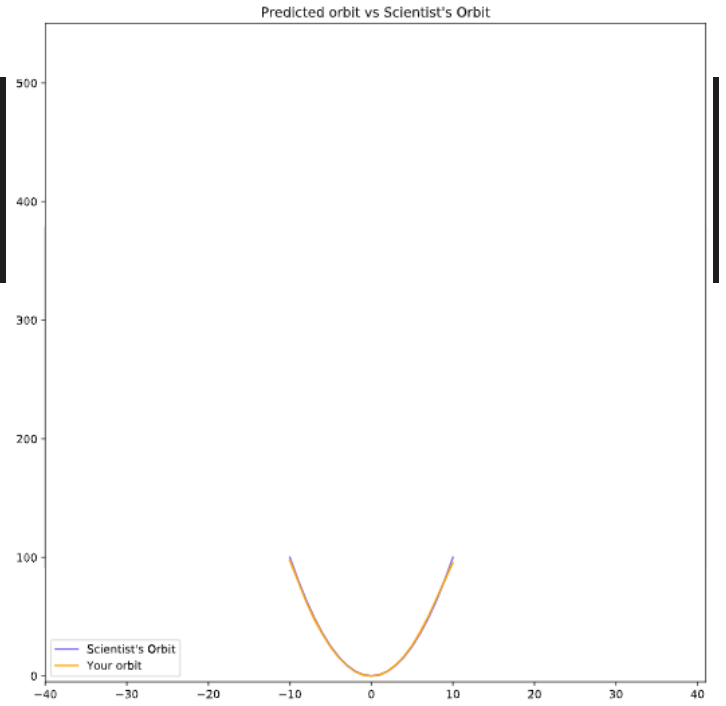
If you want to check the source code of `plot_orbit`, paste `show_code(plot_orbit)` into the console.

Hurry up, the Earth is running out of time!

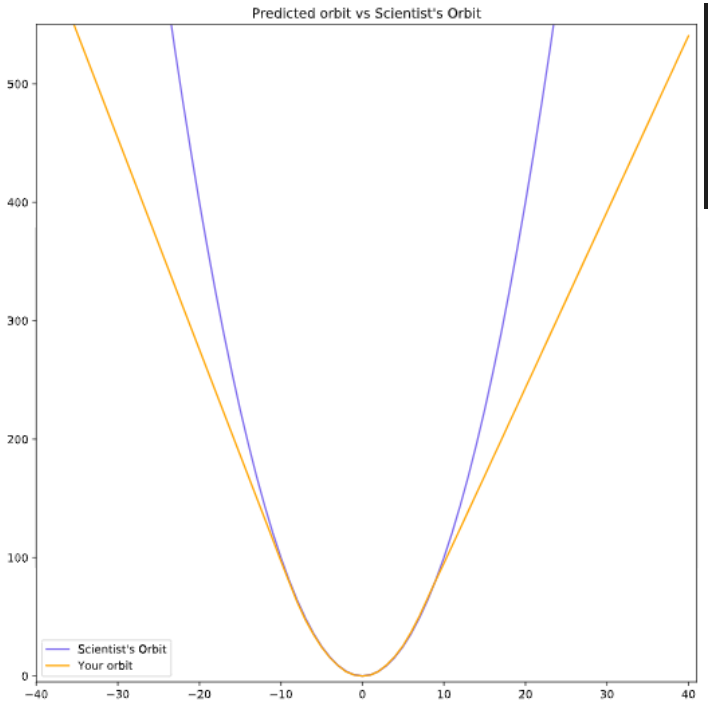
*Remember `np.arange(x,y)` produces a range of values from **x** to **y-1**. That is the `[x, y)` interval.*

```
In [1]: show_code(plot_orbit)
def plot_orbit(model_preds):
  axeslim = int(len(model_preds)/2)
  plt.plot(np.arange(-axeslim, axeslim + 1),np.arange(-axeslim, axeslim + 1)**2,color="mediumslateblue")
  plt.plot(np.arange(-axeslim, axeslim + 1),model_preds,color="orange")
  plt.axis([-40, 41, -5, 550])
  plt.legend(["Scientist's Orbit", 'Your orbit'],loc="lower left")
  plt.title("Predicted orbit vs Scientist's Orbit")
  plt.show()
```

```
# Predict the twenty minutes orbit
twenty_min_orbit = model.predict(np.arange(-10, 11))

# Plot the twenty minute orbit
plot_orbit(twenty_min_orbit)
```



```
# Predict the eighty minute orbit
eighty_min_orbit = model.predict(np.arange(-40, 41))

# Plot the eighty minute orbit
plot_orbit(eighty_min_orbit)
```



Your model fits perfectly to the scientists trajectory for time values between -10 to +10, the region where the meteor crosses the impact region, so we won't be hit! However, it starts to diverge when predicting for new values we haven't trained for. This shows neural networks learn according to the data they are fed with. Data quality and diversity are very important. You've barely scratched the surface of what neural networks can do.

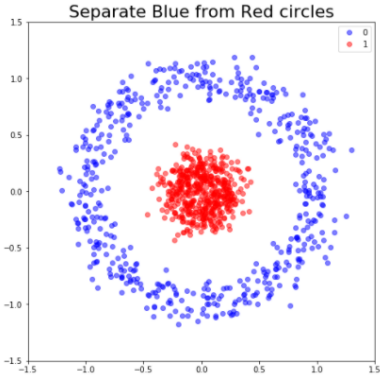
Chapter 2. Going Deeper

By the end of this chapter, you will know how to solve binary, multi-class, and multi-label problems with neural networks. All of this by solving problems like detecting fake dollar bills, deciding who threw which dart at a board, and building an intelligent system to water your farm. You will also be able to plot model training metrics and to stop training and save your models when they no longer improve.

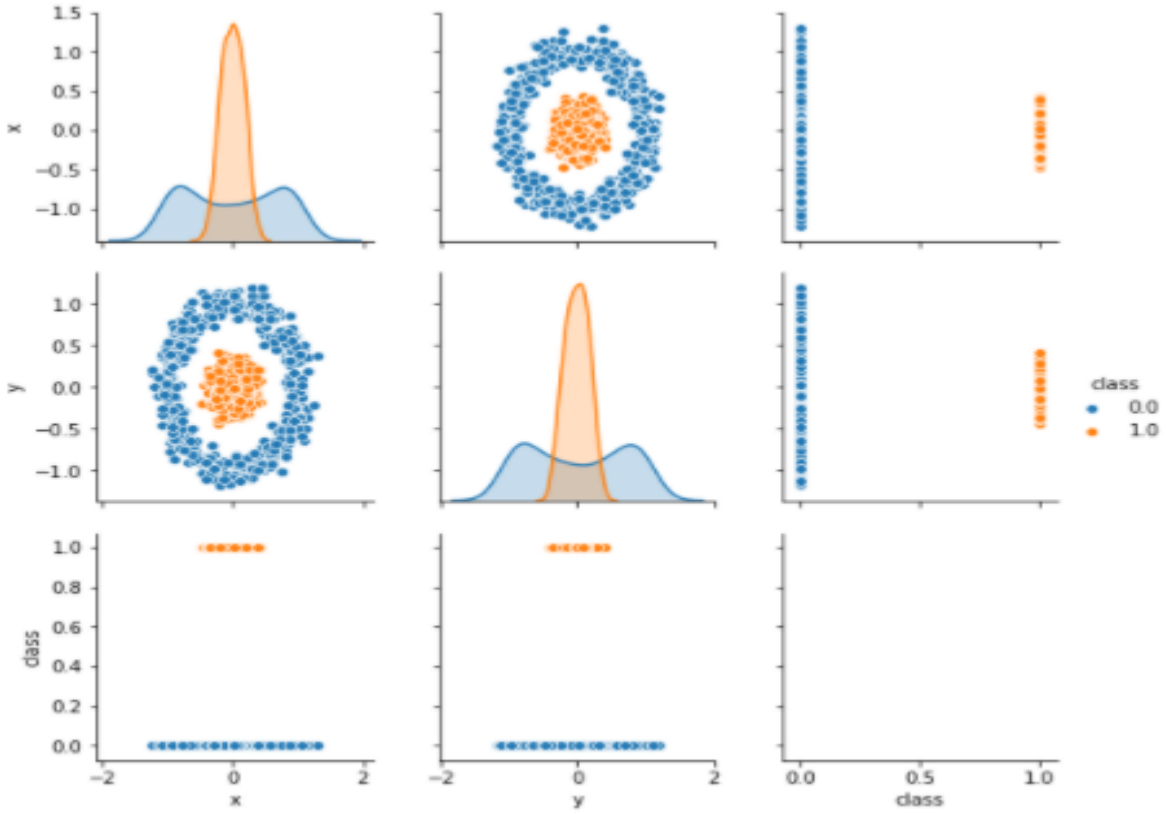
Binary classification

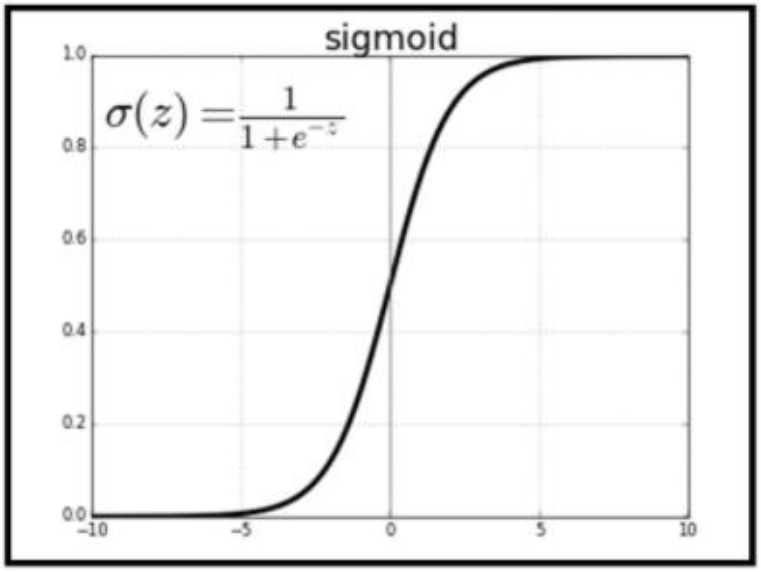
To predict whether an observation belongs to one of 2 possible classes

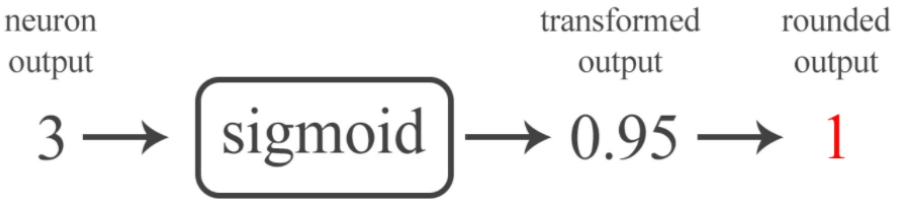
coordinates	labels
[0.242, 0.038]	1
[0.044, -0.057]	1
[-0.787, -0.076]	0



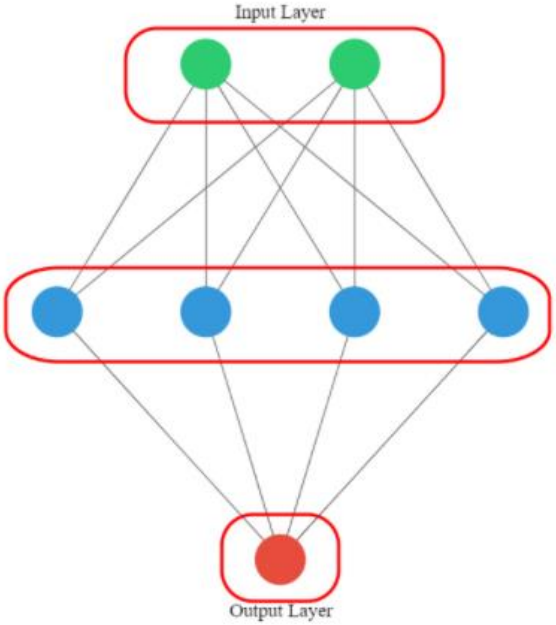

```
import seaborn as sns

# Plot a pairplot
sns.pairplot(circles, hue="target")
```



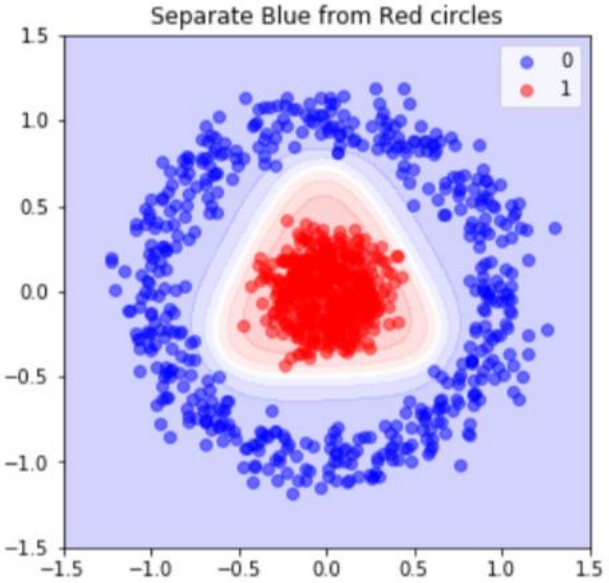





```
from keras.models import Sequential
from keras.layers import Dense
# Instantiate a sequential model
model = Sequential()
# Add input and hidden layer
model.add(Dense(4, input_shape=(2,),
                activation='tanh'))
# Add output layer, use sigmoid
model.add(Dense(1,
                activation='sigmoid'))
```




```
# Compile model
model.compile(optimizer='sgd',
              loss='binary_crossentropy')
# Train model
model.train(coordinates, labels, epochs=20)
# Predict with trained model
preds = model.predic
```

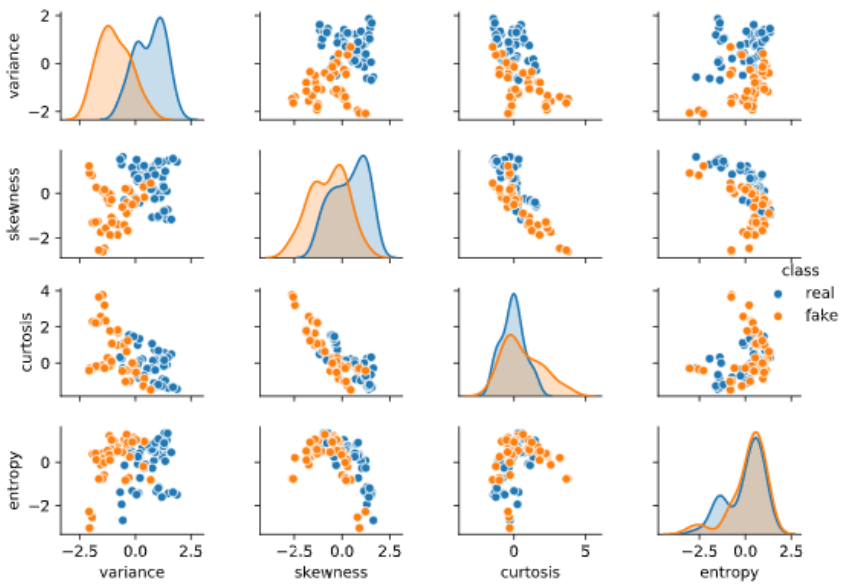


Exploring dollar bills

You will practice building classification models in Keras with the **Banknote Authentication** dataset. Your goal is to distinguish between real and fake dollar bills. In order to do this, the dataset comes with 4 features: `variance`,`skewness`,`kurtosis` and `entropy`. These features are calculated by applying mathematical operations over the dollar bill images. The labels are found in the dataframe's `class` column. A pandas DataFrame named `banknotes` is ready to use, let's do some data exploration!

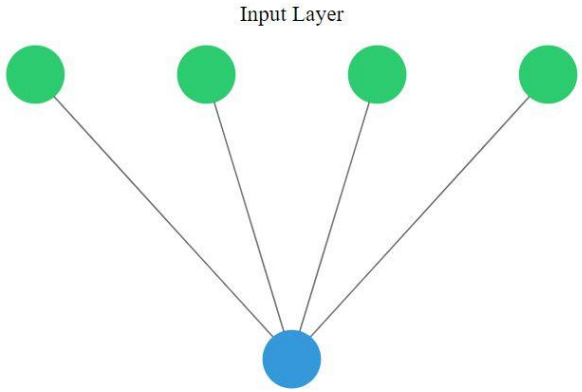
```
# Import seaborn
import seaborn as sns

# Use pairplot and set the hue to be our class column
sns.pairplot(banknotes, hue='class')

# Show the plot
plt.show()

# Describe the data
print('Dataset stats: \n', banknotes.describe())

# Count the number of observations per class
print('Observations per class: \n', banknotes['class'].value_counts())
```

```
<script.py> output:
    Dataset stats:
              variance    skewness    curtosis     entropy
    count   96.000000   96.000000   96.000000   96.000000
    mean    -0.057791   -0.102829    0.230412    0.081497
    std      1.044960    1.059236    1.128972    0.975565
    min     -2.084590   -2.621646   -1.482300   -3.034187
    25%     -0.839124   -0.916152   -0.415294   -0.262668
    50%     -0.026748   -0.037559   -0.033603    0.394888
    75%      0.871034    0.813601    0.978766    0.745212
    max      1.869239    1.634072    3.759017    1.343345
    Observations per class:
     real    53
    fake    43
    Name: class, dtype: int64
```



Your pairplot shows that there are features for which the classes spread out noticeably. This gives us an intuition about our classes being easily separable. Let's build a model to find out what it can do!

A binary classification model



Now that you know what the **Banknote Authentication** dataset looks like, we'll build a simple model to distinguish between real and fake bills.

You will perform binary classification by using a single neuron as an output. The input layer will have 4 neurons since we have 4 features in our dataset. The model's output will be a value constrained between 0 and 1.

We will interpret this output number as the probability of our input variables coming from a fake dollar bill, with 1 meaning we are certain it's a fake bill.

```
# Import the sequential model and dense layer
from keras.models import Sequential
from keras.layers import Dense

# Create a sequential model
model = Sequential()

# Add a dense layer
model.add(Dense(1, input_shape=(4,), activation='sigmoid'))

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

# Display a summary of your model
model.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 1)                 5
    =================================================================
    Total params: 5
    Trainable params: 5
    Non-trainable params: 0
    _________________________________________________________________
```

Let's use this classification model to detect fake dollar bills!

Is this dollar bill fake ?

You are now ready to train your `model` and check how well it performs when classifying new bills! The dataset has already been partitioned into features: `X_train` & `X_test`, and labels: `y_train` & `y_test`.

```
# Train your model for 20 epochs
model.fit(X_train, y_train, epochs = 20)

# Evaluate your model accuracy on the test set
accuracy = model.evaluate(X_test, y_test)[1]

# Print accuracy
print('Accuracy:', accuracy)
```

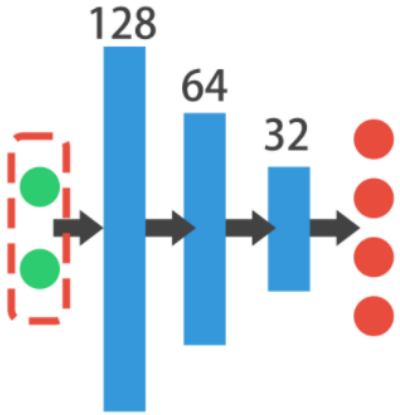
```
<script.py> output:
    Epoch 1/20

     32/960 [>.............................] - ETA: 3s - loss: 0.7641 - acc: 0.5000
    960/960 [==============================] - 0s 162us/step - loss: 0.6655 - acc: 0.6531
    Epoch 2/20

     32/960 [>.............................] - ETA: 0s - loss: 0.6201 - acc: 0.5938
    960/960 [==============================] - 0s 50us/step - loss: 0.6429 - acc: 0.6698
    Epoch 3/20

     32/960 [>                            ] - ETA: 0s - loss: 0.7369 - acc: 0.5312
    Epoch 19/20

     32/960 [>.............................] - ETA: 0s - loss: 0.5203 - acc: 0.7500
    960/960 [==============================] - 0s 38us/step - loss: 0.4341 - acc: 0.8240
    Epoch 20/20

     32/960 [>.............................] - ETA: 0s - loss: 0.4655 - acc: 0.7812
    960/960 [==============================] - 0s 38us/step - loss: 0.4266 - acc: 0.8250

     32/412 [=>............................] - ETA: 0s
    412/412 [==============================] - 0s 52us/step
    Accuracy: 0.8252427167105443
```

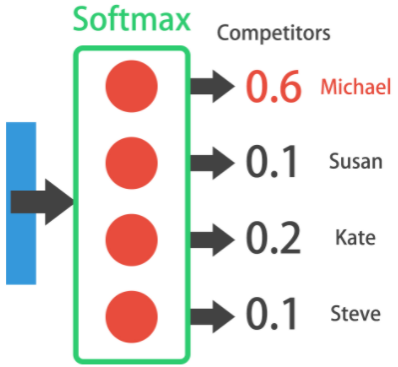
It looks like you are getting a high accuracy even with this simple model!

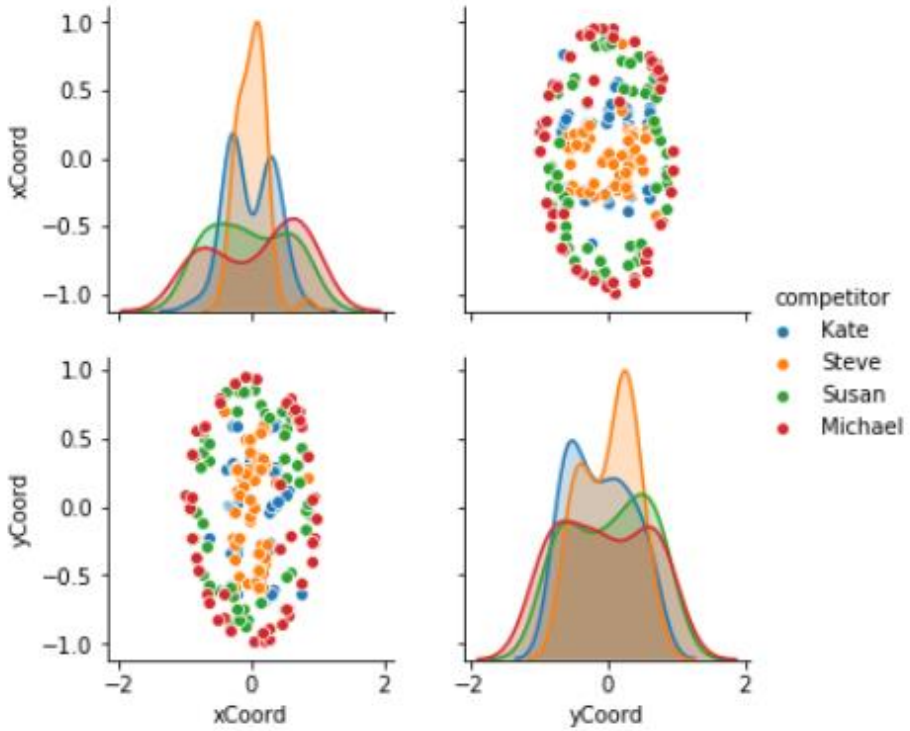
Multi-class classification

Classes are mutually exclusive

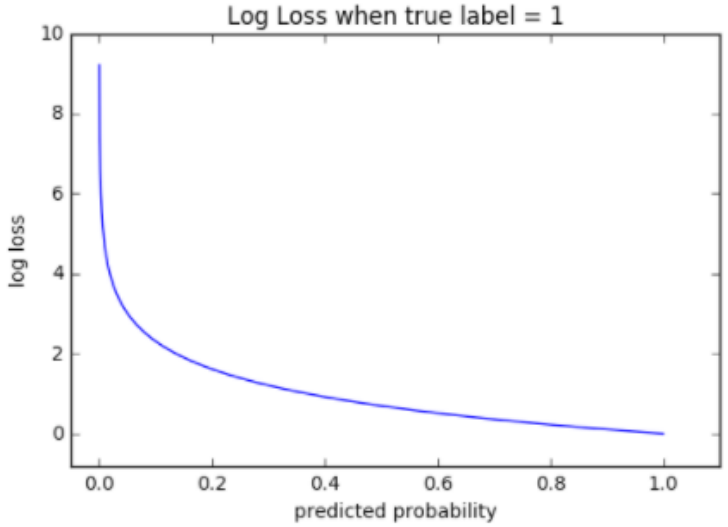
xCoord	yCoord	competitor
-0.037673	0.057402	Steve
-0.331021	-0.585035	Susan
-0.123567	0.839730	Susan
-0.086160	0.959787	Michael
-0.902632	0.078753	Michael








```
# Instantiate a sequential model
# ...
# Add an input and hidden layer
# ...
# Add more hidden layers
# ...
# Add your output layer
model.add(Dense(4, activation='softmax')

model.compile(optimizer='adam', loss='categorical_crossentropy')
```



```
import pandas as pd
from keras.utils import to_categorical

# Load dataset
df = pd.read_csv('data.csv')
# Turn response variable into labeled codes
df.response = pd.Categorical(df.response)
df.response = df.response.cat.codes
# Turn response variable into one-hot response vector
y = to_categorical(df.response)
```

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

A multi-class model

You're going to build a model that predicts who threw which dart only based on where that dart landed! (That is the dart's x and y coordinates on the board.)

This problem is a multi-class classification problem since each dart can only be thrown by one of 4 competitors. So classes/labels are mutually exclusive, and therefore we can build a neuron with as many output as competitors and use the `softmax` activation function to achieve a total sum of probabilities of 1 over all competitors.

Keras `Sequential` model and `Dense` layer are already loaded for you to use.

```
# Instantiate a sequential model
model = Sequential()

# Add 3 dense layers of 128, 64 and 32 neurons each
model.add(Dense(128, input_shape=(2,), activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))

# Add a dense layer with as many neurons as competitors
model.add(Dense(4, activation='softmax'))

# Compile your model using categorical_crossentropy loss
```

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

Your models are increasing in depth, just as your knowledge on neural networks!

Prepare your dataset

In the console you can check that your labels, `darts.competitor` are not yet in a format to be understood by your network. They contain the names of the competitors as strings. You will first turn these competitors into unique numbers,then use the `to_categorical()` function from `keras.utils` to turn these numbers into their one-hot encoded representation.

This is useful for multi-class classification problems, since there are as many output neurons as classes and for every observation in our dataset we just want one of the neurons to be activated.

The dart's dataset is loaded as `darts`. Pandas is imported as `pd`. Let's prepare this dataset!

```
# Transform into a categorical variable
darts.competitor = pd.Categorical(darts.competitor)

# Assign a number to each category (label encoding)
darts.competitor = darts.competitor.cat.codes

# Print the label encoded competitors
print('Label encoded competitors: \n',darts.competitor.head())
```

```
<script.py> output:
    Label encoded competitors:
     0    2
    1    3
    2    1
    3    0
    4    2
    Name: competitor, dtype: int8
```

```
# Import to_categorical from keras utils module
from keras.utils import to_categorical

coordinates = darts.drop(['competitor'], axis=1)
# Use to_categorical on your labels
competitors = to_categorical(darts.competitor)

# Now print the one-hot encoded labels
print('One-hot encoded competitors: \n',competitors)
```

```
<script.py> output:
    One-hot encoded competitors:
     [[0. 0. 1. 0.]
     [0. 0. 0. 1.]
     [0. 1. 0. 0.]
     ...
     [0. 1. 0. 0.]
     [0. 1. 0. 0.]
     [0. 0. 0. 1.]]
```

Each competitor is now a vector of length 4, full of zeroes except for the position representing her or himself.

Training on dart throwers

Your model is now ready, just as your dataset. It's time to train! The `coordinates` features and `competitors` labels you just transformed have been partitioned into `coord_train`,`coord_test` and `competitors_train`,`competitors_test`.

Your `model` is also loaded. Feel free to visualize your training data or `model.summary()` in the console. Let's find out who threw which dart just by looking at the board!

```
# Fit your model to the training data for 200 epochs
model.fit(coord_train, competitors_train, epochs=200)

# Evaluate your model accuracy on the test data
accuracy = model.evaluate(coord_test, competitors_test)[1]

# Print accuracy
print('Accuracy:', accuracy)
```

Your model just trained for 200 epochs! The accuracy on the test set is quite high. How are the predictions looking? Let's find out!

```
<script.py> output:
    Epoch 1/200

     32/640 [>.............................] - ETA: 4s - loss: 1.3952 - acc: 0.3750
    640/640 [==============================] - 0s 392us/step - loss: 1.3863 - acc: 0.3047
    Epoch 2/200

     32/640 [>.............................] - ETA: 0s - loss: 1.3664 - acc: 0.5000
    640/640 [==============================] - 0s 56us/step - loss: 1.3537 - acc: 0.4109
    Epoch 3/200
    Epoch 200/200

     32/640 [>.............................] - ETA: 0s - loss: 0.3567 - acc: 0.8125
    640/640 [==============================] - 0s 55us/step - loss: 0.5137 - acc: 0.8000

     32/160 [=====>........................] - ETA: 0s
    160/160 [==============================] - 0s 168us/step
    Accuracy: 0.84375
```

Softmax predictions

Your recently trained `model` is loaded for you. This model is generalizing well!, that's why you got a high accuracy on the test set.

Since you used the `softmax` activation function, for every input of 2 coordinates provided to your model there's an output vector of 4 numbers. Each of these numbers encodes the probability of a given dart being thrown by one of the 4 possible competitors.

When computing accuracy with the model's `.evaluate()` method, your model takes the class with the highest probability as the prediction. `np.argmax()` can help you do this since it returns the index with the highest value in an array.

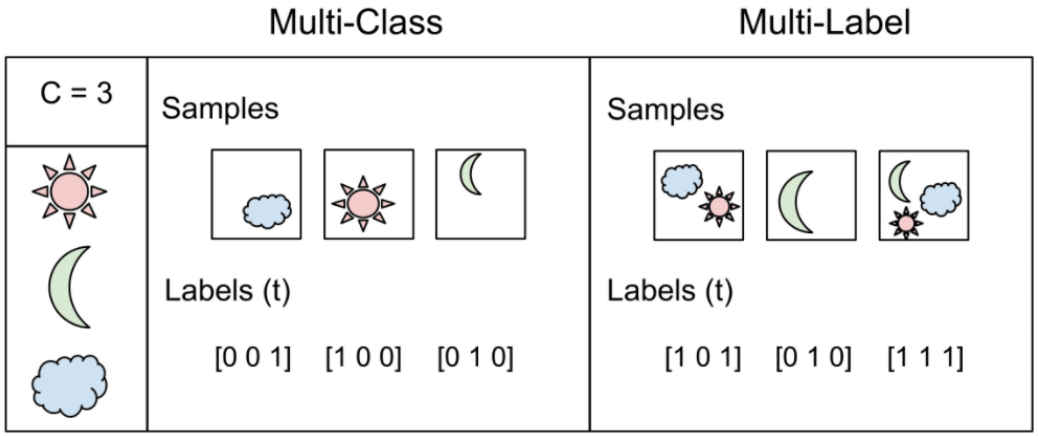
Use the collection of test throws stored in `coords_small_test` and `np.argmax()` to check this out!

```
# Predict on coords_small_test
preds = model.predict(coords_small_test)

# Print preds vs true values
print("{:45} | {}".format('Raw Model Predictions','True labels'))
for i,pred in enumerate(preds):
  print("{} | {}".format(pred,competitors_small_test[i]))

# Extract the position of highest probability from each pred vector
preds_chosen = [np.argmax(pred) for pred in preds]

# Print preds vs true values
print("{:10} | {}".format('Rounded Model Predictions','True labels'))
for i,pred in enumerate(preds_chosen):
  print("{:25} | {}".format(pred,competitors_small_test[i]))
```

```
<script.py> output:
    Raw Model Predictions                          | True labels
    [0.34438723 0.00842557 0.63167274 0.01551455] | [0. 0. 1. 0.]
    [0.0989717  0.00530467 0.07537904 0.8203446 ] | [0. 0. 0. 1.]
    [0.33512568 0.00785374 0.28132284 0.37569773] | [0. 0. 0. 1.]
    [0.8547263  0.01328656 0.11279515 0.01919206] | [1. 0. 0. 0.]
    [0.3540977  0.00867271 0.6223853  0.01484426] | [0. 0. 1. 0.]
    Rounded Model Predictions | True labels
                            2 | [0. 0. 1. 0.]
                            3 | [0. 0. 0. 1.]
                            3 | [0. 0. 0. 1.]
                            0 | [1. 0. 0. 0.]
                            2 | [0. 0. 1. 0.]
```

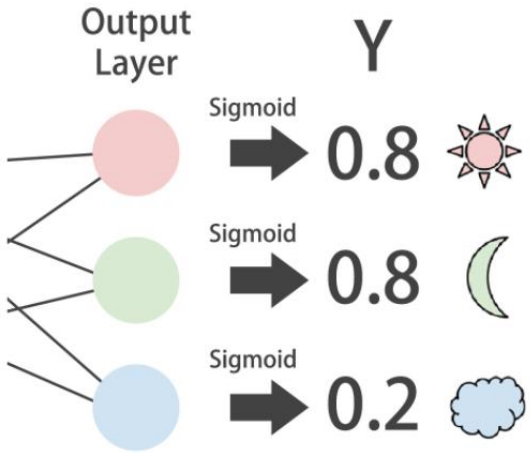
As you've seen you can easily interpret the softmax output. This can also help you spot those observations where your network is less certain on which class to predict, since you can see the probability distribution among classes per prediction. Let's learn how to solve new problems with neural networks!

Multi-label classification



Multi-Label vectors are also one-hot encoded

```
from keras.models import Sequential
from keras.layers import Dense

# Instantiate model
model = Sequential()
# Add input and hidden layers
model.add(Dense(2, input_shape=(1,)))
# Add an output layer for the 3 classes and sigmoid activation
model.add(Dense(3, activation='sigmoid'))
```



Use sigmoid outputs because we no longer care about the sum of probabilities. We want each output neuron to be able to individually take a value between 0 and 1.

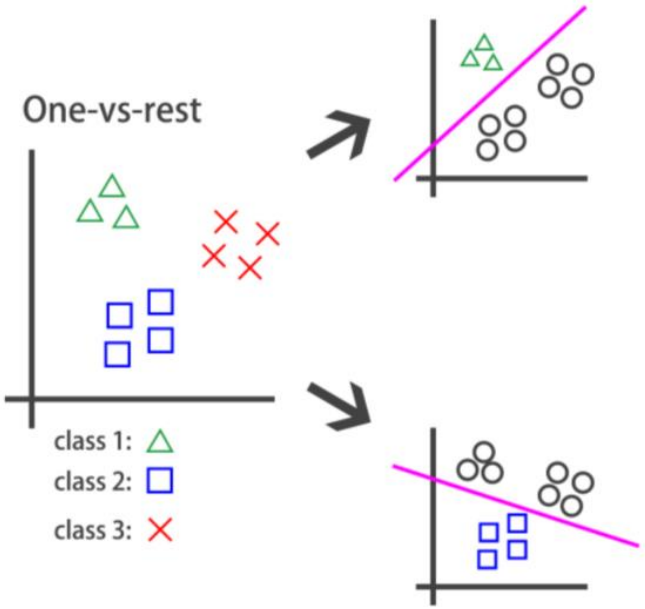
Use binary_crossentropy as the loss function when compiling the model, as if you were performing several binary classification problems: for each output we are deciding whether or not its corresponding lebel is present.

Use validation_split to reserve a percentage of training data for testing at each epoch.

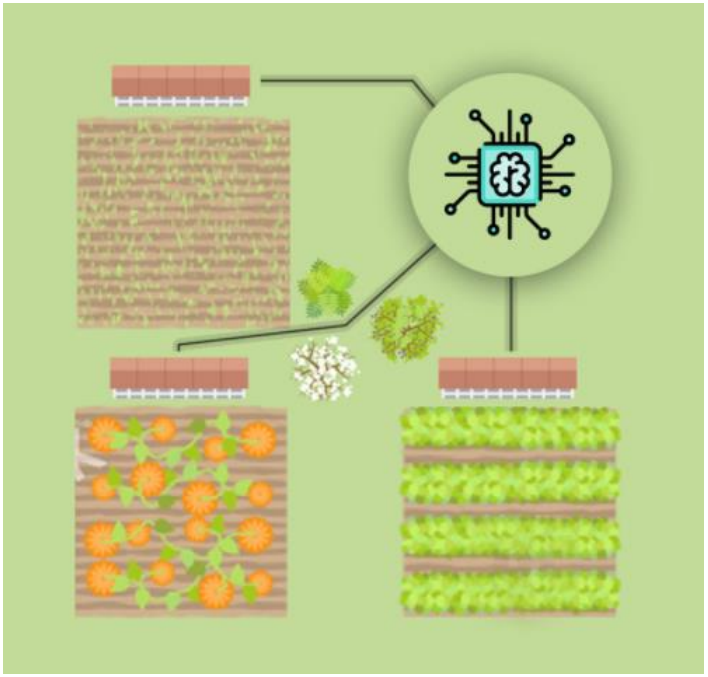
```
# Compile the model with binary crossentropy
model.compile(optimizer='adam', loss='binary_crossentropy')
```

```
# Train your model, recall validation_split
model.fit(X_train, y_train,
          epochs=100,
          validation_split=0.2)
```

```
Train on 1260 samples, validate on 280 samples
Epoch 1/100
1260/1260 [==============================] - 0s 285us/step
- loss: 0.7035 - acc: 0.6690 - val_loss: 0.5178 - val_acc: 0.7714
...
```



An irrigation machine
A farm field has an array of 20 sensors distributed along 3 crop fields. These sensors measure, among other things, the humidity of the soil, radiation of the sun, etc. Your task is to use the combination of measurements of these sensors to decide which parcels to water, given each parcel has different environmental requirements.



Sensor measurements

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	4.0	3.0	4.0	4.0	1.0	1.0	4.0	0.0	3.0	3.0	2.0	2.0	3.0	1.0	4.0	2.0	2.0	2.0	1.0	4.0
1	1.0	1.0	6.0	3.0	2.0	3.0	4.0	5.0	1.0	3.0	3.0	2.0	3.0	2.0	2.0	4.0	0.0	1.0	2.0	4.0
2	1.0	5.0	7.0	6.0	4.0	0.0	0.0	6.0	0.0	1.0	1.0	3.0	4.0	2.0	1.0	0.0	4.0	1.0	1.0	3.0
3	0.0	1.0	3.0	3.0	7.0	1.0	2.0	2.0	0.0	4.0	3.0	2.0	4.0	2.0	0.0	2.0	3.0	4.0	1.0	2.0
4	1.0	5.0	2.0	2.0	1.0	0.0	3.0	3.0	1.0	1.0	5.0	1.0	1.0	2.0	2.0	4.0	3.0	3.0	0.0	5.0

Parcels to water

	0	1	2
0	0	0	0
1	1	1	1
2	1	1	0
3	1	1	1
4	0	0	0

An irrigation machine

You're going to automate the watering of farm parcels by making an intelligent irrigation machine. Multi-label classification problems differ from multi-class problems in that each observation can be labeled with zero or more classes. So classes/labels are not mutually exclusive, you could water all, none or any combination of farm parcels based on the inputs.

To account for this behavior what we do is have an output layer with as many neurons as classes but this time, unlike in multi-class problems, each output neuron has a `sigmoid` activation function. This makes each neuron in the output layer able to output a number between 0 and 1 independently.

Keras `Sequential()` model and `Dense()` layers are preloaded. It's time to build an intelligent irrigation machine!

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

# Add a hidden layer of 64 neurons and a 20 neuron's input
model.add(Dense(64, input_shape=(20,), activation='relu'))

# Add an output layer of 3 neurons with sigmoid activation
model.add(Dense(3, activation='sigmoid'))

# Compile your model with binary crossentropy loss
model.compile(optimizer = 'adam',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])

model.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 64)                1344
    _________________________________________________________________
    dense_2 (Dense)              (None, 3)                 195
    =================================================================
    Total params: 1,539
    Trainable params: 1,539
    Non-trainable params: 0
    _________________________________________________________________
```

You've already built 3 models for 3 different problems! Hopefully you're starting to get a feel for how different problems can be modeled in the neural network realm.

Training with multiple labels

An output of your multi-label `model` could look like this: `[0.76 , 0.99 , 0.66 ]`. If we round up probabilities higher than 0.5, this observation will be classified as containing all 3 possible labels `[1,1,1]`. For this particular problem, this would mean watering all 3 parcels in your farm is the right thing to do, according to the network, given the input sensor measurements.

You will now train and predict with the `model` you just built. `sensors_train`, `parcels_train`, `sensors_test` and `parcels_test` are already loaded for you to use. Let's see how well your intelligent machine performs!

```
# Train for 100 epochs using a validation split of 0.2
model.fit(sensors_train, parcels_train, epochs=100, validation_split=0.2)

# Predict on sensors_test and round up the predictions
preds = model.predict(sensors_test)
preds_rounded = np.round(preds)

# Print rounded preds
print('Rounded Predictions: \n', preds_rounded)

# Evaluate your model's accuracy on the test data
accuracy = model.evaluate(sensors_test, parcels_test)[1]

# Print accuracy
print('Accuracy:', accuracy)
```

```
<script.py> output:
    Train on 1120 samples, validate on 280 samples
    Epoch 1/100

      32/1120 [..............................] - ETA: 7s - loss: 1.2917 - acc: 0.3750
    1024/1120 [==========================>...] - ETA: 0s - loss: 0.7495 - acc: 0.6058
    1120/1120 [==============================] - 0s 276us/step - loss: 0.7333 - acc: 0.6164 - val_loss: 0.5356 - val_acc: 0.7369
    Epoch 2/100

      32/1120 [..............................] - ETA: 0s - loss: 0.5277 - acc: 0.7604
    1024/1120 [==========================>...] - ETA: 0s - loss: 0.4896 - acc: 0.7572
    1120/1120 [==============================] - 0s 61us/step - loss: 0.4883 - acc: 0.7589 - val_loss: 0.4290 - val_acc: 0.7881
    Epoch 3/100
```

```
1120/1120 [==============================] - 0s 63us/step - loss: 0.1245 - acc: 0.9482 - val_loss: 0.2710 - val_acc: 0.8845
Epoch 100/100

  32/1120 [..............................] - ETA: 0s - loss: 0.1866 - acc: 0.9375
 640/1120 [================>.............] - ETA: 0s - loss: 0.1241 - acc: 0.9490
1120/1120 [==============================] - 0s 99us/step - loss: 0.1244 - acc: 0.9491 - val_loss: 0.2738 - val_acc: 0.8905
Rounded Predictions:
 [[1. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 ...
 [1. 1. 0.]
 [0. 1. 0.]
 [1. 1. 1.]]

 32/600 [>.............................] - ETA: 0s
600/600 [==============================] - 0s 39us/step
Accuracy: 0.9055555701255799
```

You can see how the `validation_split` argument is useful for evaluating how your model performs as it trains. Let's move on and improve your model training by using callbacks!

Keras callbacks

A callback is a function that is executed after some other function, event, or task has finished. a keras callback is a block of code that gets executed after each epoch during training or after the training is finished. They are useful to store metrics as the model trains and to make decisions as the training goes by.

After training, the history object (python dictionary) stores the saved metrics of the model at each epoch during training as an array of numbers.

```
# Training a model and saving its history
history = model.fit(X_train, y_train,
                    epochs=100,
                    metrics=['accuracy'])
print(history.history['loss'])
```

```
[0.6753975939750672, ..., 0.3155936544282096]
```

```
print(history.history['acc'])
```

```
[0.7030952412741525, ..., 0.8604761900220599]
```

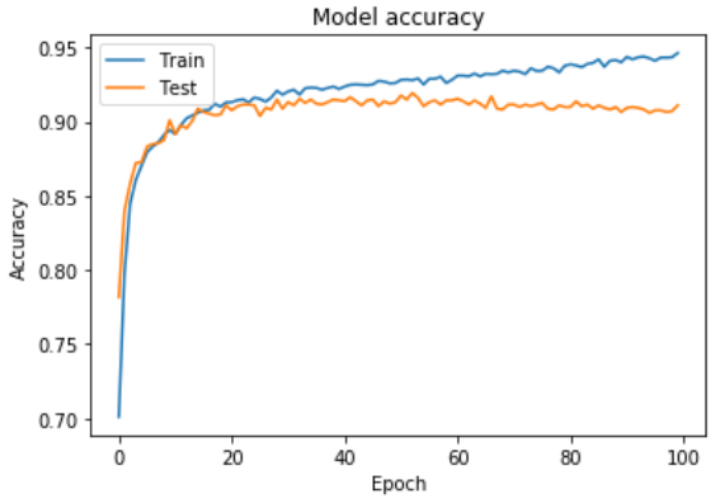
```
# Training a model and saving its history
history = model.fit(X_train, y_train,
                    epochs=100,
                    validation_data=(X_test, y_test),
                    metrics=['accuracy'])
print(history.history['val_loss'])
```

```
[0.7753975939750672, ..., 0.4155936544282096]
```

```
print(history.history['val_acc'])
```

```
[0.6030952412741525, ..., 0.7604761900220599]
```

```
# Plot train vs test accuracy per epoch
plt.figure()
# Use the history metrics
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
# Make it pretty
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'])
plt.show()
```




```
# Import early stopping from keras callbacks
from keras.callbacks import EarlyStopping

# Instantiate an early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5)

# Train your model with the callback
model.fit(X_train, y_train, epochs=100,
          validation_data=(X_test, y_test),
          callbacks = [early_stopping])
```

After epoch 25, overfitting started, training accuracy keeps improving whereas test accuracy does not.
Early stopping can prevent overfitting, since it stops training when validation accuracy no longer improves.

```python
# Import model checkpoint from keras callbacks
from keras.callbacks import ModelCheckpoint

# Instantiate a model checkpoint callback
model_save = ModelCheckpoint('best_model.hdf5',
                             save_best_only=True)

# Train your model with the callback
model.fit(X_train, y_train, epochs=100,
          validation_data=(X_test, y_test),
          callbacks = [model_save])
```

Model checkpoint callback allows us to save model as it trains. By default, validation loss is monitored.

The history callback

The history callback is returned by default every time you train a model with the `.fit()` method. To access these metrics you can access the `history` dictionary parameter inside the returned `h_callback` object with the corresponding keys.

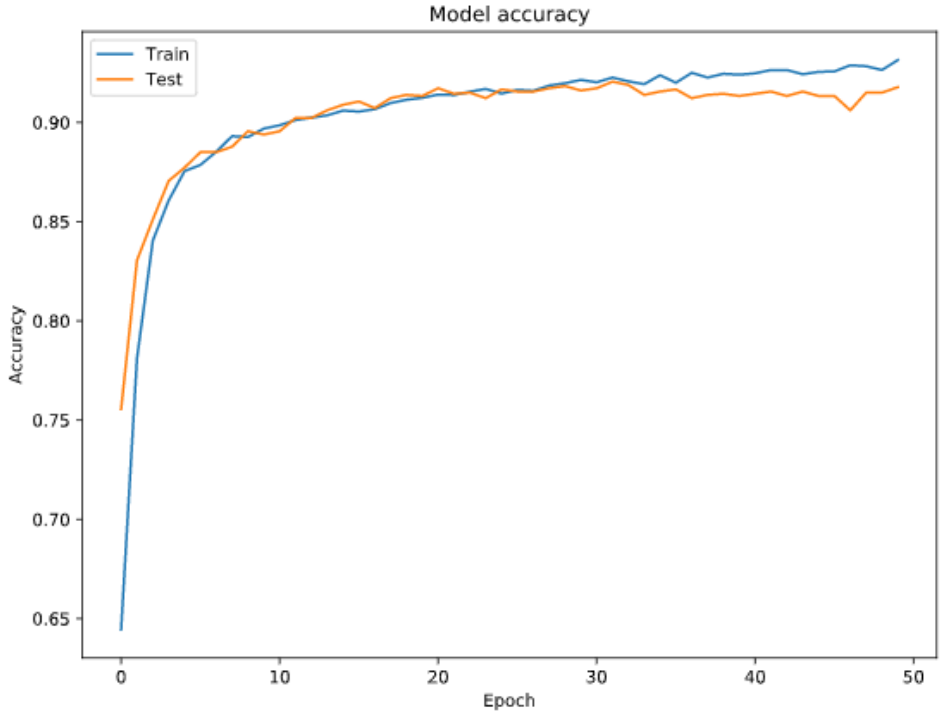
The irrigation machine `model` you built in the previous lesson is loaded for you to train, along with its features and labels now loaded as `X_train`, `y_train`, `X_test`, `y_test`. This time you will store the model's `history`callback and use the `validation_data` parameter as it trains.

You will plot the results stored in `history` with `plot_accuracy()` and `plot_loss()`, two simple matplotlib functions. You can check their code in the console by pasting `show_code(plot_loss)`.

Let's see the behind the scenes of our training!

```python
# Train your model and save its history
h_callback = model.fit(X_train, y_train, epochs = 50,
                    validation_data=(X_test, y_test))

# Plot train vs test loss during training
plot_loss(h_callback.history['loss'], h_callback.history['val_loss'])

# Plot train vs test accuracy during training
plot_accuracy(h_callback.history['acc'], h_callback.history['val_acc'])
```

These graphs are really useful for detecting overfitting and to know if your neural network would benefit from more training data. More on this in the next chapter!



Early stopping your model

The early stopping callback is useful since it allows for you to stop the model training if it no longer improves after a given number of epochs. To make use of this functionality you need to pass the callback inside a list to the model's callback parameter in the `.fit()` method.

The `model` you built to detect fake dollar bills is loaded for you to train, this time with early stopping. `X_train`, `y_train`, `X_test` and `y_test` are also available for you to use.

```
# Import the early stopping callback
from keras.callbacks import EarlyStopping

# Define a callback to monitor val_acc
monitor_val_acc = EarlyStopping(monitor='val_acc',
                                patience=5)

# Train your model using the early stopping callback
model.fit(X_train, y_train,
          epochs=1000, validation_data=(X_test, y_test),
          callbacks=[monitor_val_acc])
```

Now you won't ever fall short of epochs! Your model will stop early if the quantity monitored doesn't improve for the given amount of epochs.

A combination of callbacks

Deep learning models can take a long time to train, especially when you move to deeper architectures and bigger datasets. Saving your model every time it improves as well as stopping it when it no longer does allows you to worry less about choosing the number of epochs to train for. You can also restore a saved model anytime and resume training where you left it.

The model training and validation data are available in your workspace as `X_train`, `X_test`, `y_train`, and `y_test`.

Use the `EarlyStopping()` and the `ModelCheckpoint()` callbacks so that you can go eat a jar of cookies while you leave your computer to work!

```
# Import the EarlyStopping and ModelCheckpoint callbacks
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Early stop on validation accuracy
monitor_val_acc = EarlyStopping(monitor = 'val_acc', patience = 3)

# Save the best model as best_banknote_model.hdf5
modelCheckpoint = ModelCheckpoint('best_banknote_model.hdf5', save_best_only = True)

# Fit your model for a stupid amount of epochs
h_callback = model.fit(X_train, y_train,
                    epochs = 1000000000000,
                    callbacks = [monitor_val_acc, modelCheckpoint],
                    validation_data = (X_test, y_test))
```

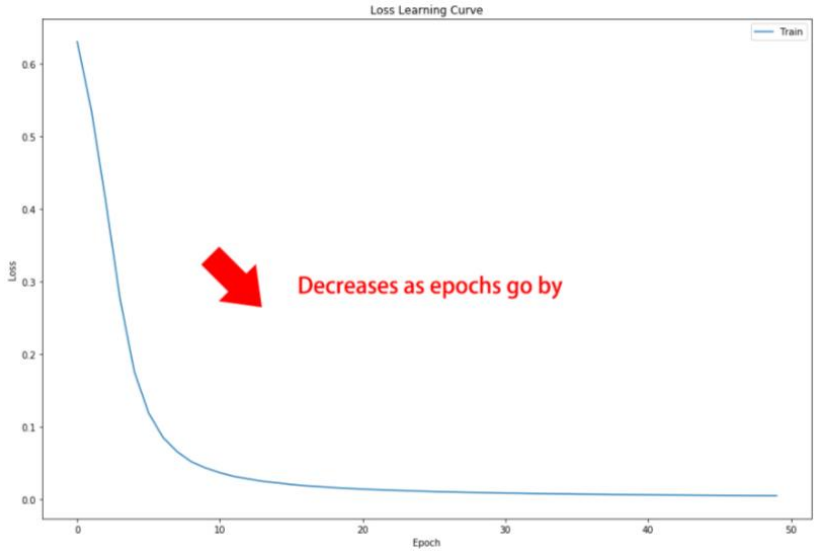
You've learned a powerful callback combo!
Now you always save the model that performed best, even if you early stopped at one that was already performing worse.

Chapter 3. Improving Your Model Performance

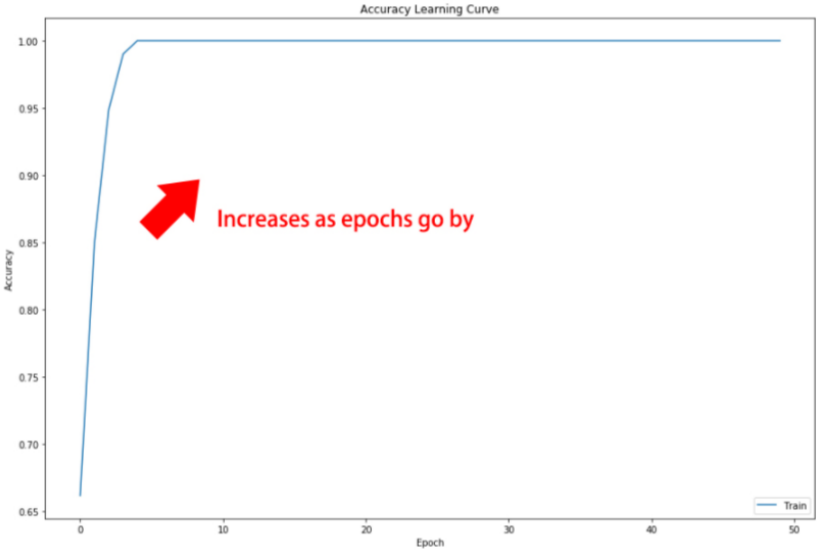
In the previous chapters, you've trained a lot of models! You will now learn how to interpret learning curves to understand your models as they train. You will also visualize the effects of activation functions, batch-sizes, and batch-normalization. Finally, you will learn how to perform automatic hyperparameter optimization to your Keras models using sklearn.

Learning curves

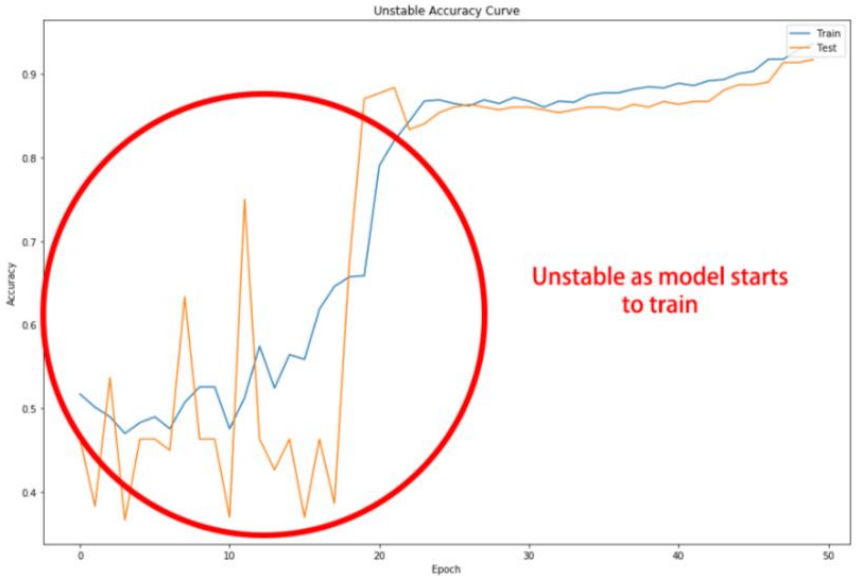
Loss Learning Curve
Loss tends to decrease as epochs go by, since our model is learning to minimise the loss function.



Accuracy Learning Curve
Accuracy tends to increase as epochs go by, since our model makes fewer mistakes as it learns.

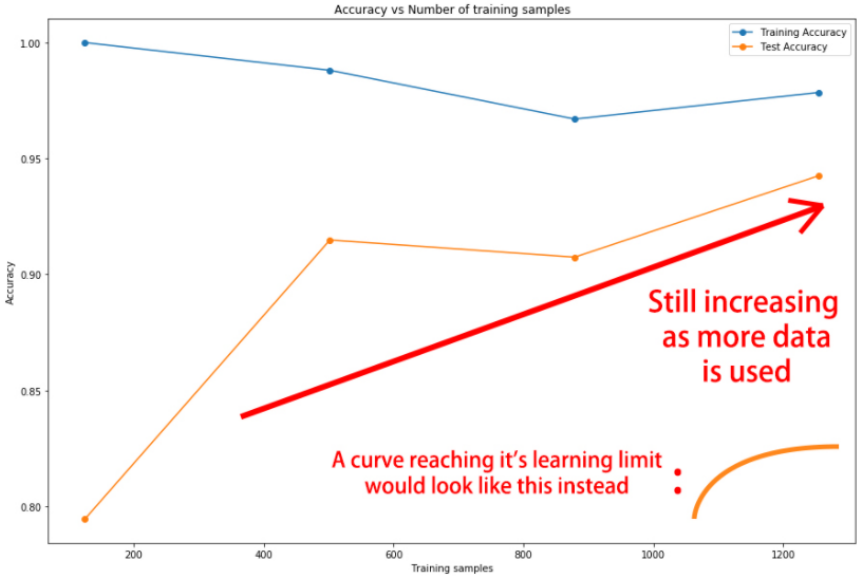


If we plot training and validation curves, we can identify overfitting when the curves start to diverge. Overfitting is when our model starts learning particularities of our training data, which don't generalise well on unseen data. Early stopping and callback is useful to stop training our model before it starts overfitting.



Many reasons can lead to unstable learning curves when models starts learn: the chosen optimizer, learning rate, batch-size, network architecture, weight initialization, etc. These parameters can be tuned to improve our model learning curves, as we aim for better accuracy and generalization power.

Neural networks like to be fed with a big and varied amount of data. If after using all our data we see that our test accuracy still has a tendency to improve, then it is worth to gather more data if possible to allow the model to keep learning.



```
# Store initial model weights
init_weights = model.get_weights()
# Lists for storing accuracies
train_accs = []
tests_accs = []
```

```
for train_size in train_sizes:
    # Split a fraction according to train_size
    X_train_frac, _, y_train_frac, _ =
    train_test_split(X_train, y_train, train_size=train_size)
    # Set model initial weights
    model.set_weights(initial_weights)
    # Fit model on the training set fraction
    model.fit(X_train_frac, y_train_frac, epochs=100,
                  verbose=0,
                  callbacks=[EarlyStopping(monitor='loss', patience=1)])
    # Get the accuracy for this training set fraction
    train_acc = model.evaluate(X_train_frac, y_train_frac, verbose=0)[1]
    train_accs.append(train_acc)
    # Get the accuracy on the whole test set
    test_acc = model.evaluate(X_test, y_test, verbose=0)[1]
    test_accs.append(test_acc)
    print("Done with size: ", train_size)
```

Learning the digits

You're going to build a model on the **digits dataset**, a sample dataset that comes pre-loaded with scikit learn. The **digits dataset** consist of **8x8 pixel handwritten digits from 0 to 9**:

You want to distinguish between each of the 10 possible digits given an image, so we are dealing with **multi-class classification**.

The dataset has already been partitioned into `X_train`, `y_train`, `X_test`, and `y_test`, using 30% of the data as testing data. The labels are already one-hot encoded vectors, so you don't need to use Keras `to_categorical()` function.

Let's build this new `model`!

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

# Input and hidden layer with input_shape, 16 neurons, and relu
model.add(Dense(16, input_shape = (64,), activation = 'relu'))

# Output layer with 10 neurons (one per digit) and softmax
model.add(Dense(10, activation = 'softmax'))

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

# Test if your model is well assembled by predicting before training
print(model.predict(X_train))
```

```
<script.py> output:
    [[1.57801419e-01 3.13342916e-08 1.17609663e-04 ... 2.88670161e-03
      1.75133277e-08 9.27261251e-04]
     [9.17966962e-01 4.87130869e-08 1.09600009e-08 ... 1.81080788e-04
      8.53955407e-06 9.01037129e-05]
     [9.99938369e-01 1.82372684e-09 9.08111347e-12 ... 2.19022222e-05
      2.59289088e-15 4.20937489e-08]
     ...
     [5.37219822e-01 5.52924506e-09 1.57055577e-10 ... 1.38584892e-05
      4.47214532e-09 1.09312405e-05]
     [2.70578653e-01 5.34917831e-07 8.48428527e-08 ... 1.55824000e-05
      4.48651798e-03 3.27920467e-02]
     [4.90147155e-03 2.87994535e-05 1.48348074e-04 ... 1.64761033e-04
      2.08042213e-04 1.32970810e-01]]
```

Predicting on training data inputs before training can help you quickly check that your model works as expected.

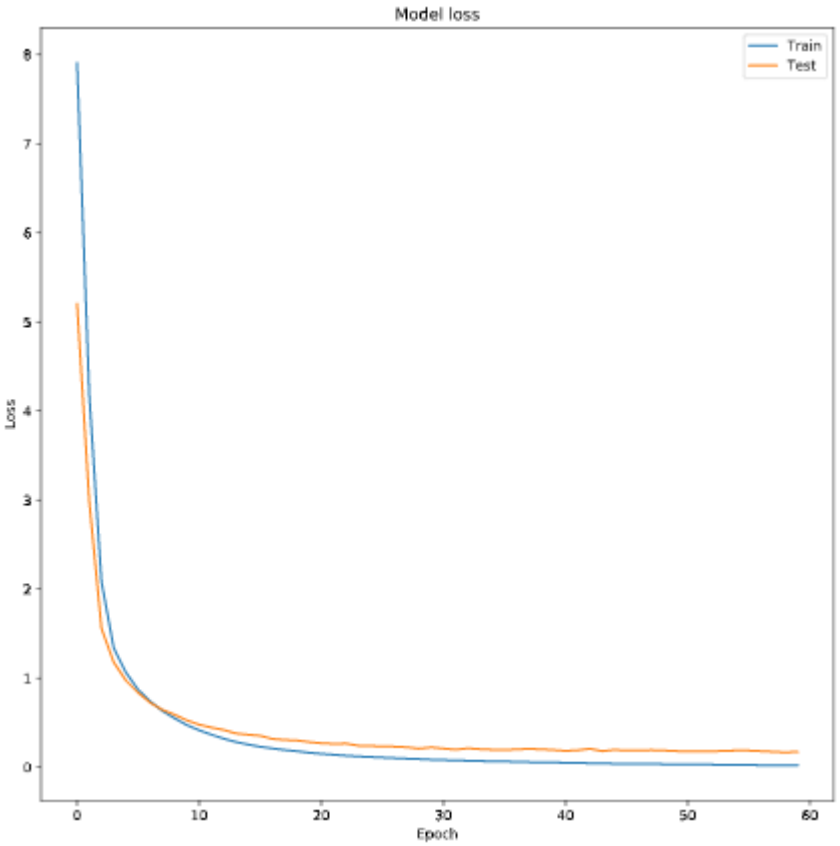
Is the model overfitting?

Let's train the `model` you just built and plot its learning curve to check out if it's overfitting! You can make use of the loaded function `plot_loss()` to plot training loss against validation loss, you can get both from the history callback.

If you want to inspect the `plot_loss()` function code, paste this in the console: `show_code(plot_loss)`

```
# Train your model for 60 epochs, using X_test and y_test as validation data
h_callback = model.fit(X_train, y_train, epochs = 60, validation_data=(X_test, y_test), verbose=0)
```

```
# Extract from the h_callback object loss and val_loss to plot the learning curve
plot_loss(h_callback.history['loss'], h_callback.history['val_loss'])
```



Just by looking at the picture, do you think the learning curve shows this model is overfitting after having trained for 60 epochs?

- ○ Yes, it started to overfit since the test loss is higher than the training loss.
- ◉ No, the test loss is not getting higher as the epochs go by.

This graph doesn't show overfitting but convergence. It looks like your model has learned all it could from the data and it no longer improves. The test loss, although higher than the training loss, is not getting worse, so we aren't overfitting to the training data.

Do we need more data?

It's time to check whether the **digits dataset** `model` you built benefits from more training examples!

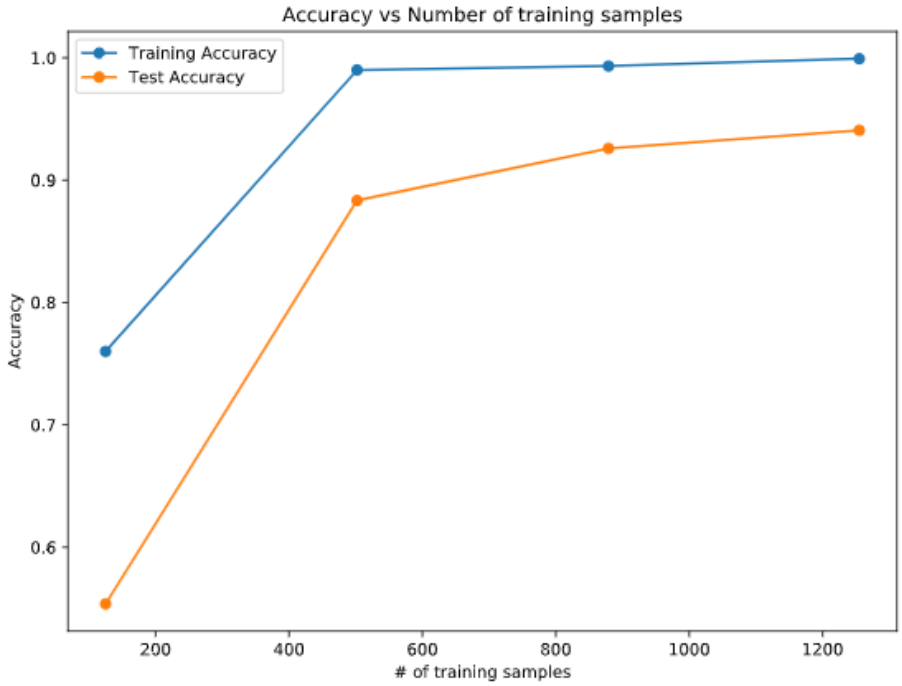
In order to keep code to a minimum, various things are already initialized and ready to use:

- The `model` you just built.
- `X_train`,`y_train`,`X_test`, and `y_test`.
- The `initial_weights` of your model, saved after using `model.get_weights()`.
- A pre-defined list of training sizes: `training_sizes`.
- A pre-defined early stopping callback monitoring loss: `early_stop`.
- Two empty lists to store the evaluation results: `train_accs` and `test_accs`.

Train your model on the different training sizes and evaluate the results on `X_test`. End by plotting the results with `plot_results()`.

```
for size in training_sizes:
    # Get a fraction of training data (we only care about the training data)
    X_train_frac, y_train_frac = X_train[:size], y_train[:size]

    # Reset the model to the initial weights and train it on the new training data fraction
    model.set_weights(initial_weights)
    model.fit(X_train_frac, y_train_frac, epochs = 50, callbacks = [early_stop])

    # Evaluate and store both: the training data fraction and the complete test set results
    train_accs.append(model.evaluate(X_train_frac, y_train_frac)[1])
    test_accs.append(model.evaluate(X_test, y_test)[1])

# Plot train vs test accuracies
plot_results(train_accs, test_accs)
```

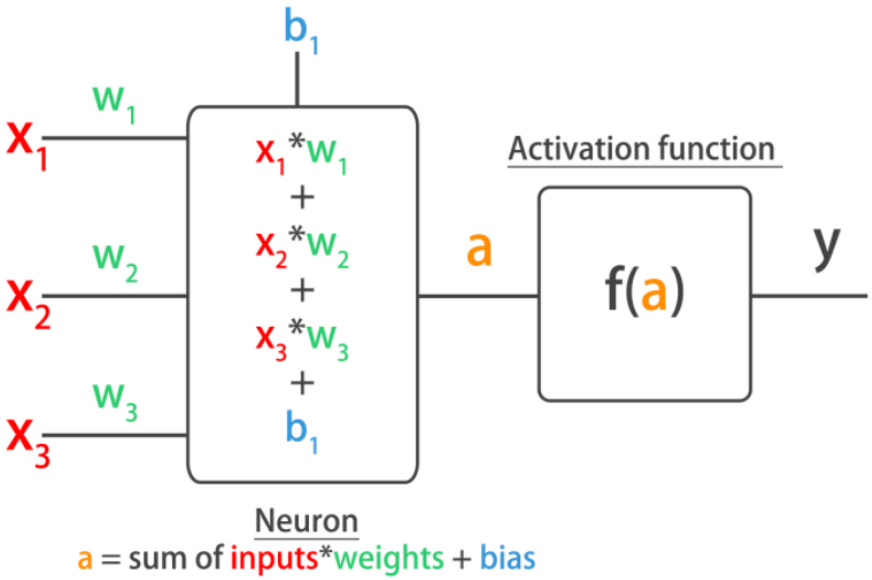
The results shows that your model would not benefit a lot from more training data, since the test set accuracy is already starting to flatten. It's time to look at activation funtions!



Activation functions

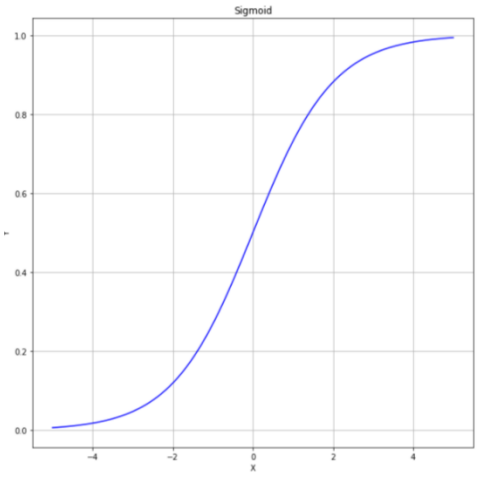
A summation of the inputs reaching the neuron multiplied by the weights of each connection and the addition of the bias weight, resulting in a number: a.

We pass this number into an activation function that essentially takes it as an input and decides how the neuron fires and which output it produces. Activation functions impact learning time, making our model converge faster or slower and achieving lower or higher accuracy. They also allow us to learn more complex functions.

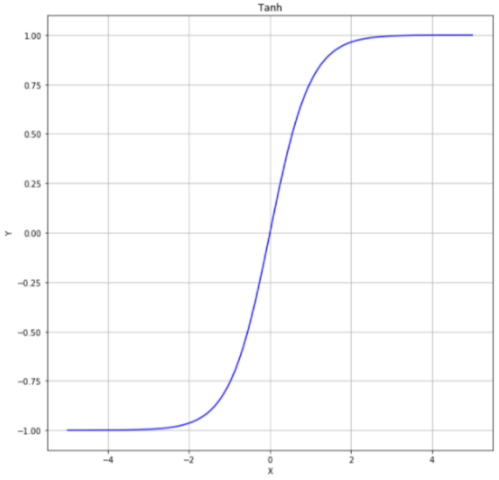


These are 4 popular activation functions

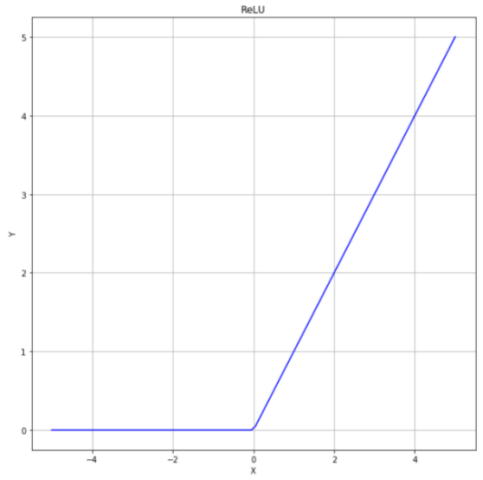
Sigmoid (0,1)



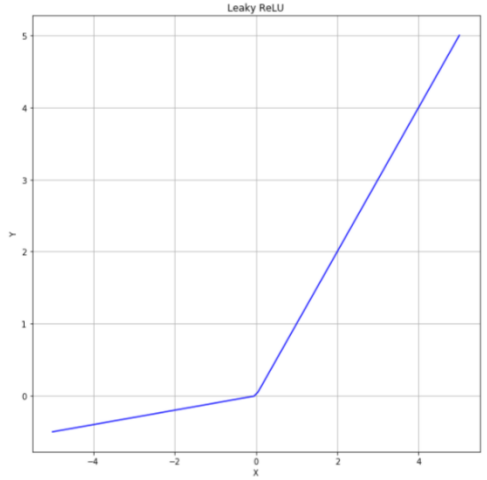
Tanh (-1,1)



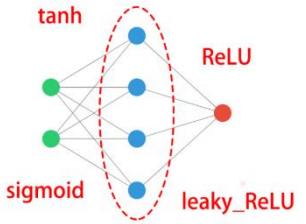
ReLU

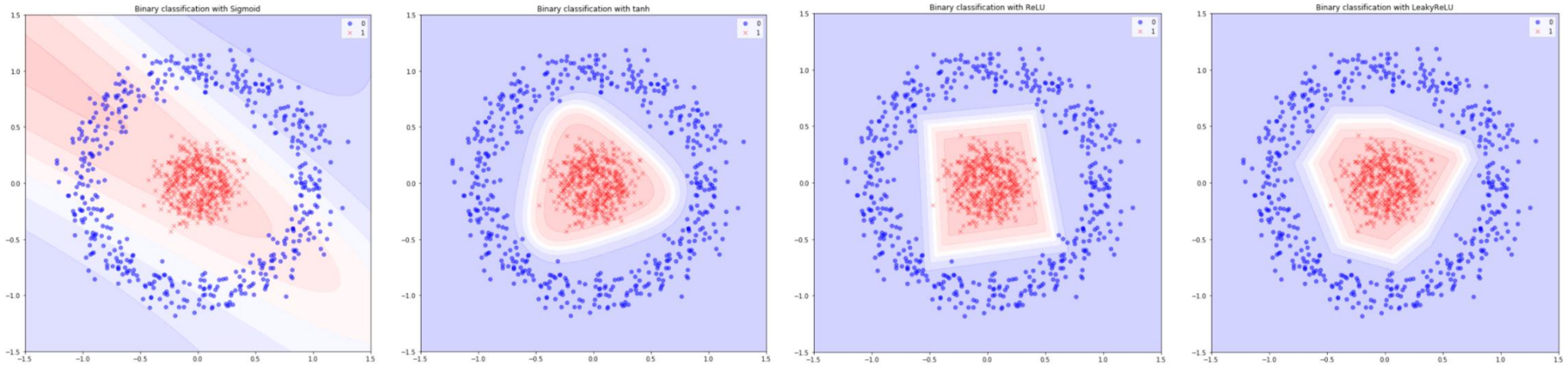


Leaky ReLU



Changing the activation function used in the hidden layer of the model we built for binary classification results in different classification boundaries. All activation functions come with their pros and cons.





It's important to note that these boundaries will be different for every run of the same model because of the random initialization of weights and other random variables that aren't fixed.

Which activation function to use?
A way to go is to start with ReLU as they train fast and will tend to generalize well to most problems, avoid sigmoids, and tune with experimentation.

```
# Set a random seed
np.random.seed(1)
# Return a new model with the given activation
def get_model(act_function):
    model = Sequential()
    model.add(Dense(4, input_shape=(2,), activation=act_function))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

```
# Activation functions to try out
activations = ['relu', 'sigmoid', 'tanh']

# Dictionary to store results
activation_results = {}
for funct in activations:
    model = get_model(act_function=funct)
    history = model.fit(X_train, y_train,
                        validation_data=(X_test, y_test),
                        epochs=100, verbose=0)
    activation_results[funct] = history
```

```
import pandas as pd

# Extract val_loss history of each activation function
val_loss_per_funct = {k:v.history['val_loss'] for k,v in activation_results.items()}
```

```
# Turn the dictionary into a pandas dataframe
val_loss_curves = pd.DataFrame(val_loss_per_funct)

# Plot the curves
val_loss_curves.plot(title='Loss per Activation function')
```

Different activation functions

The `sigmoid()`,`tanh()`, `ReLU()`, and `leaky_ReLU()` functions have been defined and ready for you to use. Each function receives an input number *X* and returns its corresponding *Y* value. Which of the statements below is **false**?

- The `sigmoid()` takes a value of *0.5* when *X = 0* whilst `tanh()` takes a value of *0*.
- The `leaky_ReLU()` takes a value of *-0.01* when *X = -1* whilst `ReLU()` takes a value of *0*.
- **The `sigmoid()` and `tanh()` both take values close to *-1* for big negative numbers.**

For big negative numbers the sigmoid approaches *0* not *-1* whilst the `tanh()` does take values close to *-1*.

Comparing activation functions

Comparing activation functions involves a bit of coding, but nothing you can't do!

You will try out different activation functions on the **multi-label model** you built for your farm irrigation machine in chapter 2. The function `get_model('relu')` returns a copy of this model and applies the `'relu'` activation function to its hidden layer.

You will loop through several activation functions, generate a new model for each and train it. By storing the history callback in a dictionary you will be able to visualize which activation function performed best in the next exercise!

`X_train`, `y_train`, `X_test`, `y_test` are ready for you to use when training your models.

```
import inspect
print(inspect.getsource(get_model))
```

```
def get_model(act_function):
    if act_function not in ['relu', 'leaky_relu', 'sigmoid', 'tanh']:
        raise ValueError('Make sure your activation functions are named correctly!')
    print("Finishing with", act_function, "...")

    return ModelWrapper(act_function)
```

```
# Activation functions to try
activations = ['relu', 'leaky_relu', 'sigmoid', 'tanh']

# Loop over the activation functions
activation_results = {}

for act in activations:
  # Get a new model with the current activation
  model = get_model(act)
  # Fit the model and store the history results
  h_callback = model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=20, verbose=0)
  activation_results[act] = h_callback
```

```
<script.py> output:
    Finishing with relu ...
    Finishing with leaky_relu ...
    Finishing with sigmoid ...
    Finishing with tanh ...
```

You've trained 4 models, each with a different activation function, let's see how well they performed!

Comparing activation functions II

What you coded in the previous exercise has been executed to obtain the `activation_results` variable, this time **100 epochs were used instead of 20**. This way you will have more epochs to further compare how the training evolves per activation function.

For every `h_callback` of each activation function in `activation_results`:

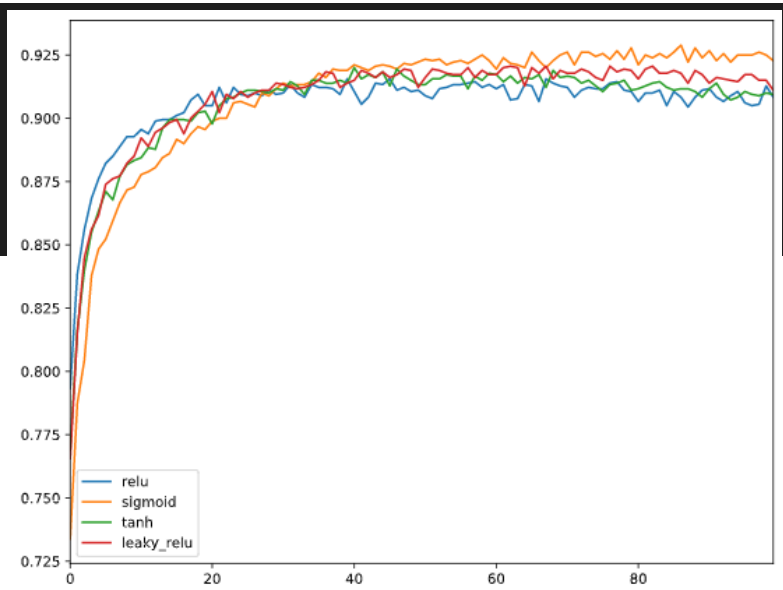
- The `h_callback.history['val_loss']` has been extracted.
- The `h_callback.history['val_acc']` has been extracted.

Both are saved into two dictionaries: `val_loss_per_function` and `val_acc_per_function`.

Pandas is also loaded as `pd` for you to use. Let's plot some quick validation loss and accuracy charts!

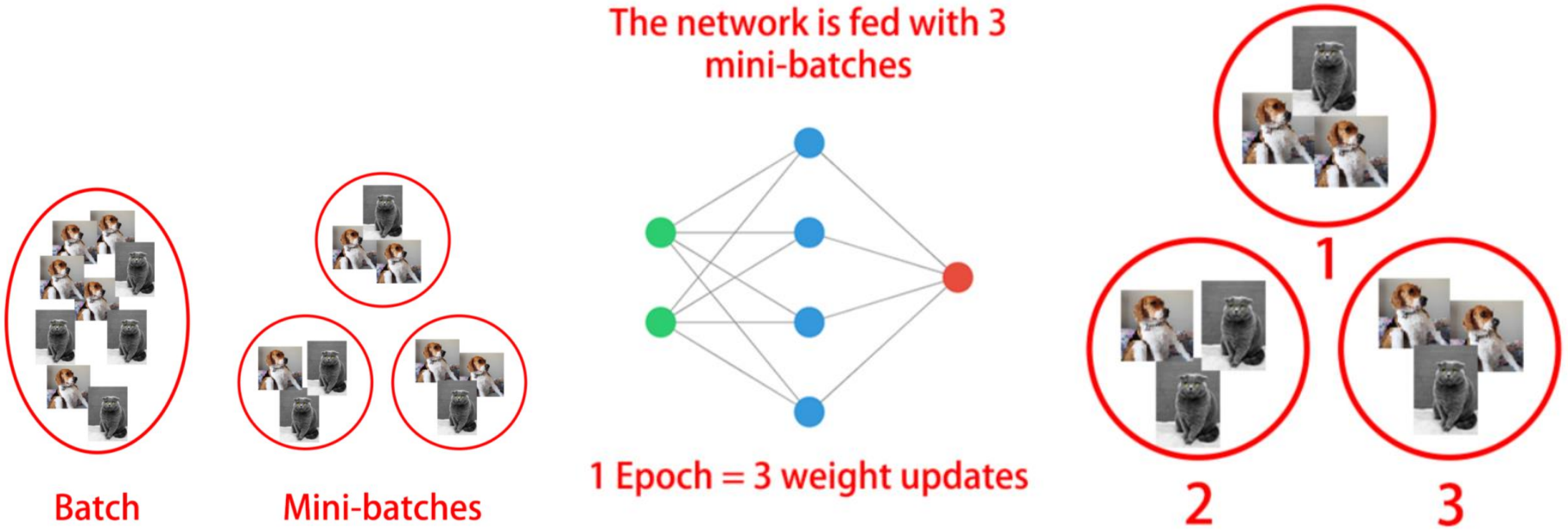
```
# Create a dataframe from val_loss_per_function
val_loss= pd.DataFrame(val_loss_per_function)

# Call plot on the dataframe
val_loss.plot()
plt.show()
```

```
# Create a dataframe from val_acc_per_function
val_acc = pd.DataFrame(val_acc_per_function)

# Call plot on the dataframe
val_acc.plot()
plt.show()
```



You've plotted both: loss and accuracy curves. It looks like sigmoid activation worked best for this particular model as the hidden layer's activation function. It led to a model with lower validation loss and higher accuracy after 100 epochs.

Batch size and batch normalization



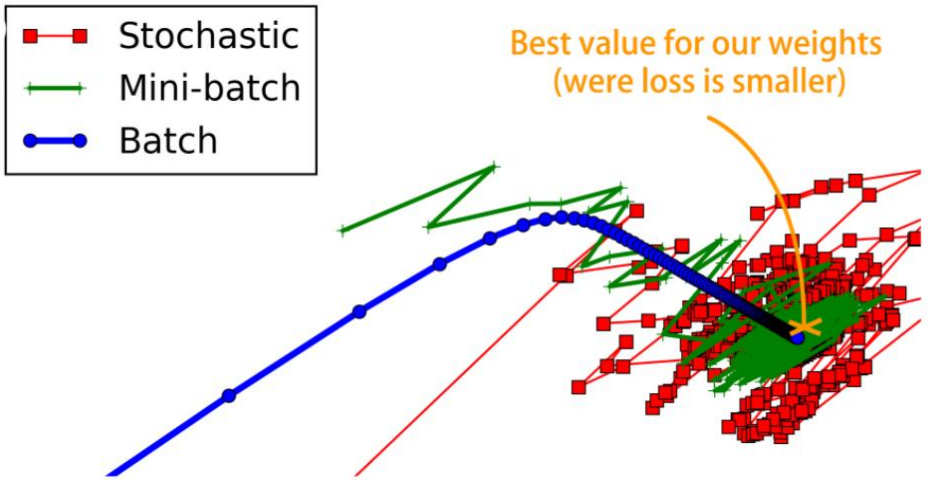
During an epoch we feed our network, calculate the errors and update the network weights. Usually, we take a mini-batch of training samples. That way, if our training set has 9 images and we choose a batch_size of 3, we will perform 3 weight updates per epoch, one per mini-batch.

Mini-batches
Advantages:

- Networks train faster (more weight updates in same amount of time)
- Less RAM memory required, can train on huge datasets
- Noise can help networks reach a lower error, escaping local minima

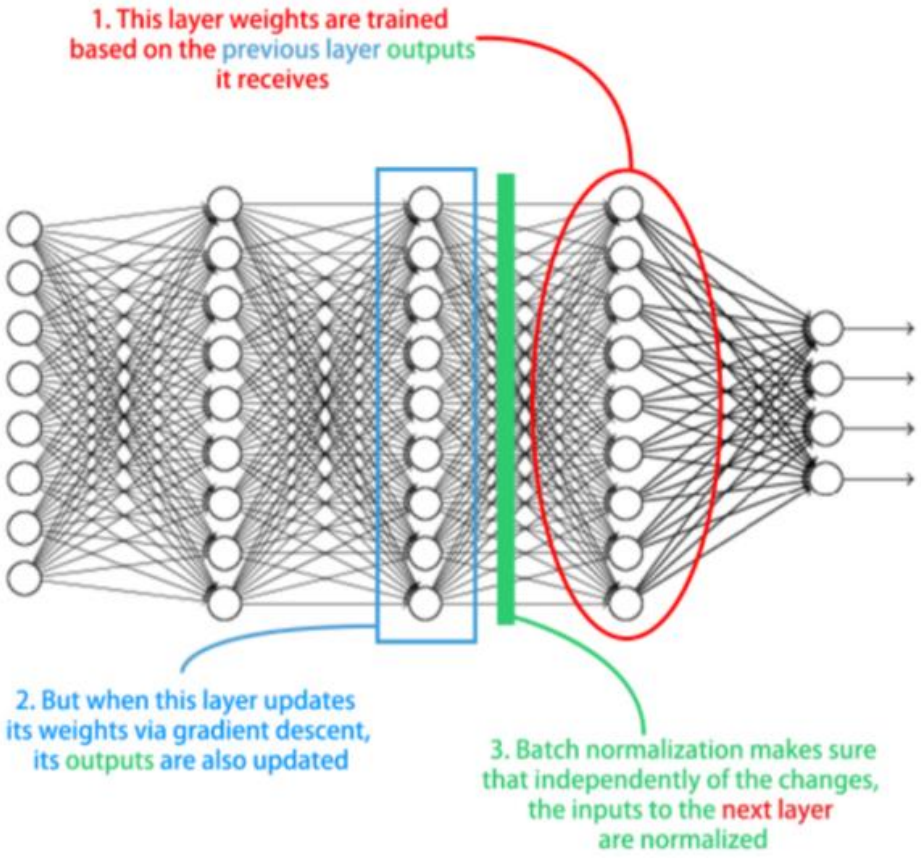
Disadvantages:

- More iterations need to be run
- A hyperparameter that needs to be tuned, we need to find a good batch size



Normalization is a common pre-processing step in machine learning algorithms, especially when features have different scales.

Example:
Standardization = (data - mean) / (standard deviation)



Batch normalization
Batch normalization makes sure that, independently of the weight changes, the inputs to the next layers are normalized. It does this in a smart way, with trainable parameters that also learn how much of this normalization is kept scaling or shifting it.
This improves gradient flow, allows for higher learning rates, reduces weight initializations dependence, adds regularization to our network and limits internal covariate shift.

```
# Import BatchNormalization from keras layers
from keras.layers import BatchNormalization
# Instantiate a Sequential model
model = Sequential()
# Add an input layer
model.add(Dense(3, input_shape=(2,), activation = 'relu'))
# Add batch normalization for the outputs of the layer above
model.add(BatchNormalization())
# Add an output layer
model.add(Dense(1, activation='sigmoid'))
```

Changing batch sizes

You've seen models are usually trained in batches of a fixed size. The smaller a batch size, the more weight updates per epoch, but at a cost of a more unstable gradient descent. Specially if the batch size is too small and it's not representative of the entire training set.

Let's see how different batch sizes affect the accuracy of a simple binary classification model that separates red from blue dots.

You'll use a batch size of one, updating the weights once per sample in your training set for each epoch. Then you will use the entire dataset, updating the weights only once per epoch.

```
# Get a fresh new model with get_model
model = get_model()

# Train your model for 5 epochs with a batch size of 1
model.fit(X_train, y_train, epochs=5, batch_size=1)
```

```
print("\n The accuracy when using a batch of size 1 is: ",
      model.evaluate(X_test, y_test)[1])
```

```
587/700 [========================>.....] - ETA: 0s - loss: 0.2778 - acc: 0.9983
616/700 [=========================>....] - ETA: 0s - loss: 0.2754 - acc: 0.9984
649/700 [==========================>...] - ETA: 0s - loss: 0.2736 - acc: 0.9985
675/700 [===========================>..] - ETA: 0s - loss: 0.2737 - acc: 0.9985
700/700 [==============================] - 1s 2ms/step - loss: 0.2726 - acc: 0.9986

 32/300 [==>...........................] - ETA: 0s
300/300 [==============================] - 0s 136us/step

 The accuracy when using a batch of size 1 is:  0.9966666666666667
```

```
model = get_model()

# Fit your model for 5 epochs with a batch of size the training set
model.fit(X_train, y_train, epochs=5, batch_size=700)
print("\n The accuracy when using the whole training set as batch-size was: ",
      model.evaluate(X_test, y_test)[1])
```

```
700/700 [==============================] - 0s 30us/step - loss: 0.6725 - acc: 0.4786
Epoch 5/5

700/700 [==============================] - 0s 3us/step - loss: 0.6723 - acc: 0.4786

 32/300 [==>...........................] - ETA: 0s
300/300 [==============================] - 0s 92us/step

 The accuracy when using the whole training set as batch-size was:  0.553333334128062
```

You can see that accuracy is lower when using a batch size equal to the training set size. This is not because the network had more trouble learning the optimization function: **Even though the same number of epochs were used for both batch sizes the number of resulting weight updates was very different!**. With a batch of size the training set and 5 epochs we only get 5 updates total, each update computes and averaged gradient descent with all the training set observations. To obtain similar results with this batch size we should increase the number of epochs so that more weight updates take place.

Batch normalizing a familiar model

Remember the **digits dataset** you trained in the first exercise of this chapter?

A multi-class classification problem that you solved using `softmax` and 10 neurons in your output layer.

You will now build a new deeper model consisting of 3 hidden layers of 50 neurons each, using batch normalization in between layers. The `kernel_initializer` parameter is used to initialize weights in a similar way.

```
# Import batch normalization from keras layers
from keras.layers import BatchNormalization

# Build your deep network
batchnorm_model = Sequential()
batchnorm_model.add(Dense(50, input_shape=(64,), activation='relu', kernel_initializer='normal'))
batchnorm_model.add(BatchNormalization())
batchnorm_model.add(Dense(50, activation='relu', kernel_initializer='normal'))
batchnorm_model.add(BatchNormalization())
batchnorm_model.add(Dense(50, activation='relu', kernel_initializer='normal'))
batchnorm_model.add(BatchNormalization())
batchnorm_model.add(Dense(10, activation='softmax', kernel_initializer='normal'))

# Compile your model with sgd
batchnorm_model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
```

That was a deep model indeed. Let's compare how it performs against this very same model without batch normalization!

Batch normalization effects

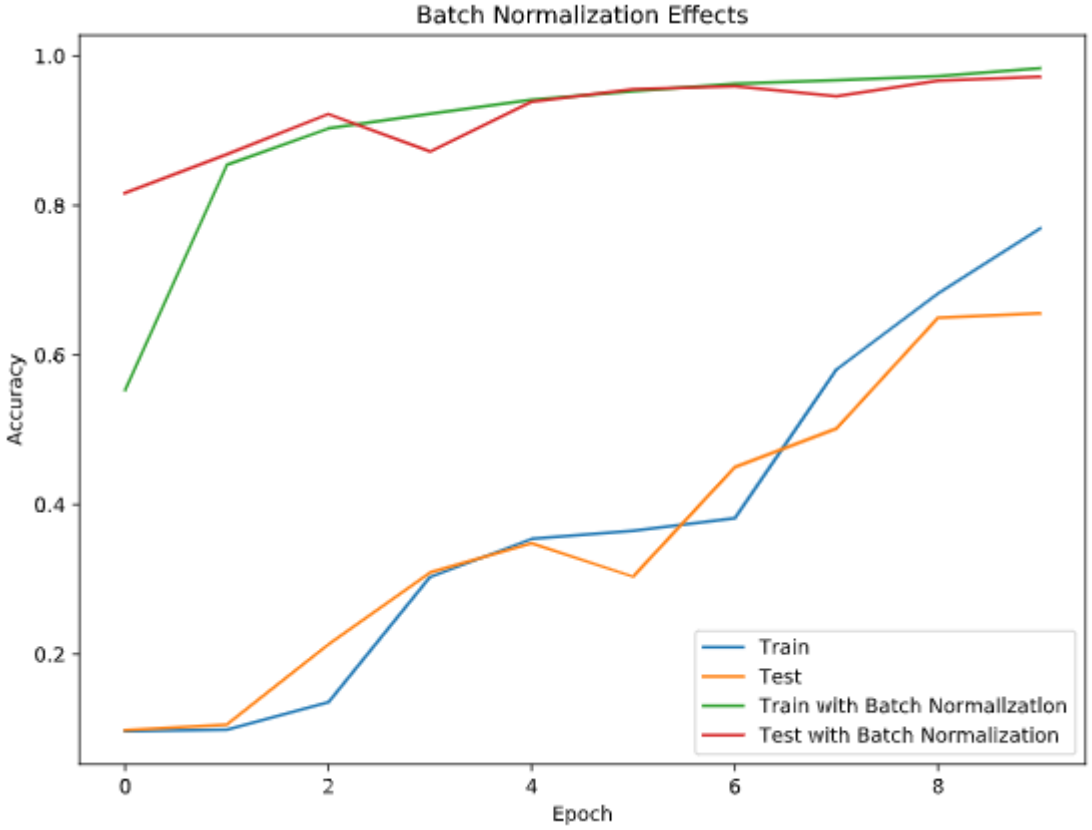
Batch normalization tends to increase the learning speed of our models and make their learning curves more stable. Let's see how two identical models with and without batch normalization compare.

The model you just built `batchnorm_model` is loaded for you to use. An exact copy of it without batch normalization: `standard_model`, is available as well. You can check their `summary()` in the console. `X_train`, `y_train`, `X_test`, and `y_test` are also loaded so that you can train both models.

You will compare the accuracy learning curves for both models plotting them with `compare_histories_acc()`.

You can check the function pasting `show_code(compare_histories_acc)` in the console.

```
# Train your standard model, storing its history callback
h1_callback = standard_model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, verbose=0)

# Train the batch normalized model you recently built, store its history callback
h2_callback = batchnorm_model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, verbose=0)

# Call compare_histories_acc passing in both model histories
compare_histories_acc(h1_callback, h2_callback)
```



You can see that for this deep model batch normalization proved to be useful, helping the model obtain high accuracy values just over the first 10 training epochs.

Hyperparameter tuning

Our aim is to identify those parameters that make our model generalize better.

Neural network hyperparameters:

- number of layers
- number of neurons per layer
- the order of such layers
- layer activation functions
- batch sizes
- learning rates
- optimizers

```
# Import RandomizedSearchCV
from sklearn.model_selection import RandomizedSearchCV
# Instantiate your classifier
tree = DecisionTreeClassifier()
# Define a series of parameters to look over
params = {'max_depth':[3,None], "max_features":range(1,4), 'min_samples_leaf': range(1,4)}
# Perform random search with cross validation
tree_cv = RandomizedSearchCV(tree, params, cv=5)
tree_cv.fit(X,y)

# Print the best parameters
print(tree_cv.best_params_)
```

```
{'min_samples_leaf': 1, 'max_features': 3, 'max_depth': 3}
```

```
# Function that creates our Keras model
def create_model(optimizer='adam', activation='relu'):
    model = Sequential()
    model.add(Dense(16, input_shape=(2,), activation=activation))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=optimizer, loss='binary_crossentropy')
    return model

# Import sklearn wrapper from keras
from keras.wrappers.scikit_learn import KerasClassifier

# Create a model as a sklearn estimator
model = KerasClassifier(build_fn=create_model, epochs=6, batch_size=16)
```

```
# Import cross_val_score
from sklearn.model_selection import cross_val_score

# Check how your keras model performs with 5 fold crossvalidation
kfold = cross_val_score(model, X, y, cv=5)

# Print the mean accuracy per fold
kfold.mean()
```

```
0.913333
```

```
# Print the standard deviation per fold
kfold.std()
```

Tips for neural networks hyperparameter tuning

- random search is preferred over grid search
- do not use too many epochs to check how well your model is performing
- use a smaller sample of your dataset
- experiment with batch sizes, activations, optimizers, learning rates

```
# Define a series of parameters
params = dict(optimizer=['sgd', 'adam'], epochs=3,
              batch_size=[5, 10, 20], activation=['relu','tanh'])

# Create a random search cv object and fit it to the data
random_search = RandomizedSearchCV(model, params_dist=params, cv=3)
random_search_results = random_search.fit(X, y)
# Print results
print("Best: %f using %s".format(random_search_results.best_score_,
random_search_results.best_params_))
```

```
Best: 0.94 using {'optimizer': 'adam', 'epochs': 3, 'batch_size': 10, 'activation': 'relu'}
```

```
def create_model(nl=1,nn=256):
    model = Sequential()
    model.add(Dense(16, input_shape=(2,), activation='relu'))
    # Add as many hidden layers as specified in nl
    for i in range(nl):
        # Layers have nn neurons
        model.add(Dense(nn, activation='relu'))
    # End defining and compiling your model...
```

```
# Define parameters, named just like in create_model()
params = dict(nl=[1, 2, 9], nn=[128,256,1000])

# Repeat the random search...

# Print results...
```

```
Best: 0.87 using {'nl': 2,'nn': 128}
```

Preparing a model for tuning

Let's tune the hyperparameters of a **binary classification** model that does well classifying the **breast cancer dataset**.

You've seen that the first step to turn a model into a sklearn estimator is to build a function that creates it. The definition of this function is important since hyperparameter tuning is carried out by varying the arguments your function receives.

Build a simple `create_model()` function that receives both a learning rate and an activation function as arguments. The `Adam` optimizer has been imported as an object from `keras.optimizers` so that you can also change its learning rate parameter.

```
# Creates a model given an activation and learning rate
def create_model(learning_rate, activation):

    # Create an Adam optimizer with the given learning rate
    opt = Adam(lr = learning_rate)

    # Create your binary classification model
    model = Sequential()
    model.add(Dense(128, input_shape = (30,), activation = activation))
    model.add(Dense(256, activation = activation))
    model.add(Dense(1, activation = 'sigmoid'))

    # Compile your model with your optimizer, loss, and metrics
    model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
    return model
```

With this function ready you can now create a sklearn estimator and generate different models to perform simple hyperparameter tuning on!

Tuning the model parameters

It's time to try out different parameters on your model and see how well it performs! The `create_model()` function you built in the previous exercise is ready for you to use. Since fitting the `RandomizedSearchCV` object would take too long, the results you'd get are printed in the `show_results()` function. You don't need to use the optional `epochs` and `batch_size` parameters when building your `KerasClassifier` object since you are passing them as `params` to the random search and this works already.

```
# Import KerasClassifier from keras scikit learn wrappers
from keras.wrappers.scikit_learn import KerasClassifier

# Create a KerasClassifier
model = KerasClassifier(build_fn = create_model)

# Define the parameters to try out
params = {'activation':['relu', 'tanh'], 'batch_size':[32, 128, 256],
          'epochs':[50, 100, 200], 'learning_rate':[0.1, 0.01, 0.001]}

# Create a randomize search cv object passing in the parameters to try
random_search = RandomizedSearchCV(model, param_distributions = params, cv = KFold(3))

# Running random_search.fit(X,y) would start the search,but it takes too long!
show_results()
```

```
<script.py> output:
    Best:
    0.975395 using {learning_rate: 0.001, epochs: 50, batch_size: 128, activation: relu}
    Other:
    0.956063 (0.013236) with: {learning_rate: 0.1, epochs: 200, batch_size: 32, activation: tanh}
    0.970123 (0.019838) with: {learning_rate: 0.1, epochs: 50, batch_size: 256, activation: tanh}
    0.971880 (0.006524) with: {learning_rate: 0.01, epochs: 100, batch_size: 128, activation: tanh}
    0.724077 (0.072993) with: {learning_rate: 0.1, epochs: 50, batch_size: 32, activation: relu}
    0.588752 (0.281793) with: {learning_rate: 0.1, epochs: 100, batch_size: 256, activation: relu}
    0.966608 (0.004892) with: {learning_rate: 0.001, epochs: 100, batch_size: 128, activation: tanh}
    0.952548 (0.019734) with: {learning_rate: 0.1, epochs: 50, batch_size: 256, activation: relu}
    0.971880 (0.006524) with: {learning_rate: 0.001, epochs: 200, batch_size: 128, activation: relu}
    0.968366 (0.004239) with: {learning_rate: 0.01, epochs: 100, batch_size: 32, activation: relu}
    0.910369 (0.055824) with: {learning_rate: 0.1, epochs: 100, batch_size: 128, activation: relu}
```

Now that we have a better idea of which parameters perform best, lets use them!

Training with cross-validation

Time to train your model with the best parameters found: **0.001** for the **learning rate**, **50 epochs**, **a 128 batch_size** and **relu activations**.

The `create_model()` function from the previous exercise is ready for you to use. `X` and `y` are loaded as features and labels.

Use the best values found for your model when creating your `KerasClassifier` object so that they are used when performing cross_validation.

End this chapter by training an awesome tuned model on the **breast cancer dataset**!

```
# Import KerasClassifier from keras wrappers
from keras.wrappers.scikit_learn import KerasClassifier

# Create a KerasClassifier
model = KerasClassifier(build_fn = create_model(learning_rate = 0.001, activation = 'relu'), epochs = 50,
                        batch_size = 128, verbose = 0)

# Calculate the accuracy score for each fold
kfolds = cross_val_score(model, X, y, cv = 3)

# Print the mean accuracy
print('The mean accuracy was:', kfolds.mean())

# Print the accuracy standard deviation
print('With a standard deviation of:', kfolds.std())
```

```
<script.py> output:
    The mean accuracy was: 0.9718834066666666
    With a standard deviation of: 0.002448915612216046
```

You can now test out different parameters on your networks and find the best models. Congratulations on making it this far, this chapter was quite a challengue! You're now left with a final chapter full of fun models to play with.

Chapter 4. Advanced Model Architectures

It's time to get introduced to more advanced architectures! You will create an autoencoder to reconstruct noisy images, visualize convolutional neural network activations, use deep pre-trained models to classify images and learn more about recurrent neural networks and working with text as you build a network that predicts the next word in a sentence.

Tensors, layers, and autoencoders

Accessing Keras layers

```
# Acessing the first layer of a Keras model
first_layer = model.layers[0]
# Printing the layer, and its input, output and weights
print(first_layer.input)
print(first_layer.output)
print(first_layer.weights)
```

```
<tf.Tensor 'dense_1_input:0' shape=(?, 3) dtype=float32>
```

```
<tf.Tensor 'dense_1/Relu:0' shape=(?, 2) dtype=float32>
```

```
[<tf.Variable 'dense_1/kernel:0' shape=(3, 2) dtype=float32_ref>,
 <tf.Variable 'dense_1/bias:0' shape=(2,) dtype=float32_ref>]
```

Tensors are the main data structures used in deep learning: inputs, outputs, transformations in neural networks are all represented using tensors.
A tensor is a multi-dimensional array of numbers.
A 2-dimensional tensor is a matrix.
A 3-dimensional tensor is an array of matrices.

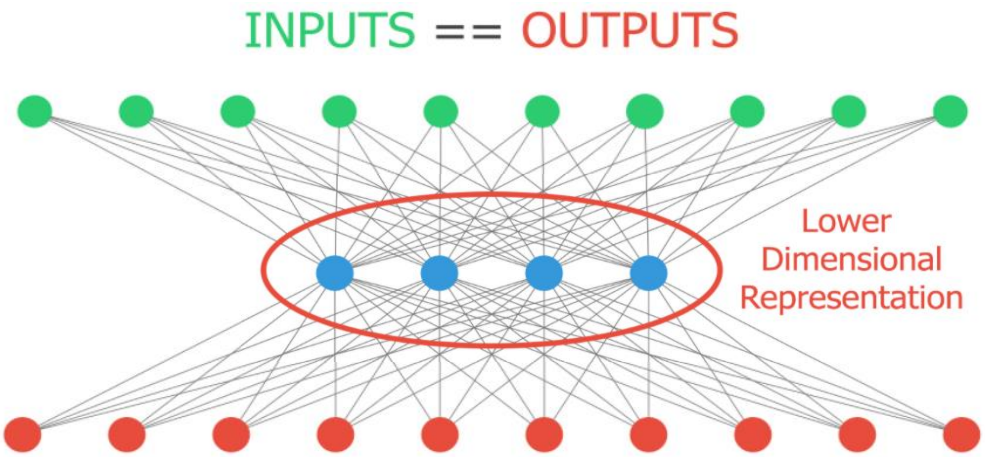
```
# Defining a rank 2 tensor (2 dimensions)
T2 = [[1,2,3],
      [4,5,6],
      [7,8,9]]
# Defining a rank 3 tensor (3 dimensions)
T3 = [[1,2,3],
      [4,5,6],
      [7,8,9],

      [10,11,12],
      [13,14,15],
      [16,17,18],

      [19,20,21],
      [22,23,24],
      [25,26,27]]
```

```
# Import Keras backend
import keras.backend as K
# Get the input and output tensors of a model layer
inp = model.layers[0].input
out = model.layers[0].output
# Function that maps layer inputs to outputs
inp_to_out = K.function([inp], [out])
# We pass and input and get the output we'd get in that first layer
print(inp_to_out([X_train])
```

```
# Outputs of the first layer per sample in X_train
[array([[0.7, 0],...,[0.1, 0.3]])]
```

Autoencoders

Autoencoders are models that aim at producing the same inputs as outputs.



Autoencoder use cases:

- dimensional reduction – obtain a smaller dimensional space representation of our inputs
- de-noising data – if trained with clean data, irrelevant noise will be filtered out during reconstruction
- anomaly detection – when strange values are passed as inputs, the network will give inaccurate output

```
# Instantiate a sequential model
autoencoder = Sequential()
# Add a hidden layer of 4 neurons and an input layer of 100
autoencoder.add(Dense(4, input_shape=(100,), activation='relu'))
# Add an output layer of 100 neurons
autoencoder.add(Dense(100, activation='sigmoid'))
# Compile your model with the appropiate loss
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

input=100, hidden=4, output=100

```
# Building a separate model to encode inputs
encoder = Sequential()
encoder.add(autoencoder.layers[0])
# Predicting returns the four hidden layer neuron outputs
encoder.predict(X_test)
```

```
# Four numbers for each observation in X_test
array([10.0234375, 5.833543, 18.90444, 9.20348],...)
```

It's a flow of tensors

If you have already built a model, you can use the `model.layers` and the `keras.backend` to build functions that, provided with a valid input tensor, return the corresponding output tensor.

This is a useful tool when we want to obtain the output of a network at an intermediate layer.

For instance, if you get the input and output from the first layer of a network, you can build an `inp_to_out` function that returns the result of carrying out forward propagation through only the first layer for a given input tensor.

So that's what you're going to do right now!

`X_test` from the **Banknote Authentication** dataset and its `model` are preloaded. Type `model.summary()` in the console to check it.

```python
# Import keras backend
import keras.backend as K

# Input tensor from the 1st layer of the model
inp = model.layers[0].input

# Output tensor from the 1st layer of the model
out = model.layers[0].output

# Define a function from inputs to outputs
inp_to_out = K.function([inp], [out])

# Print the results of passing X_test through the 1st layer
print(inp_to_out([X_test]))
```

```
<script.py> output:
    [array([[7.77682841e-01, 0.00000000e+00],
            [0.00000000e+00, 0.00000000e+00],
            [0.00000000e+00, 1.50813460e+00],
            [0.00000000e+00, 1.34600031e+00],
            [8.44748616e-01, 0.00000000e+00],
            [5.26736021e-01, 0.00000000e+00],
            [7.72461295e-01, 8.49045813e-01],
            [0.00000000e+00, 1.50589132e+00],
            [1.23799539e+00, 1.65470392e-01],
            [0.00000000e+00, 1.54111814e+00]], dtype=float32)]
```

Let's use this function for something more interesting.

Neural separation

Put on your gloves because you're going to perform brain surgery!

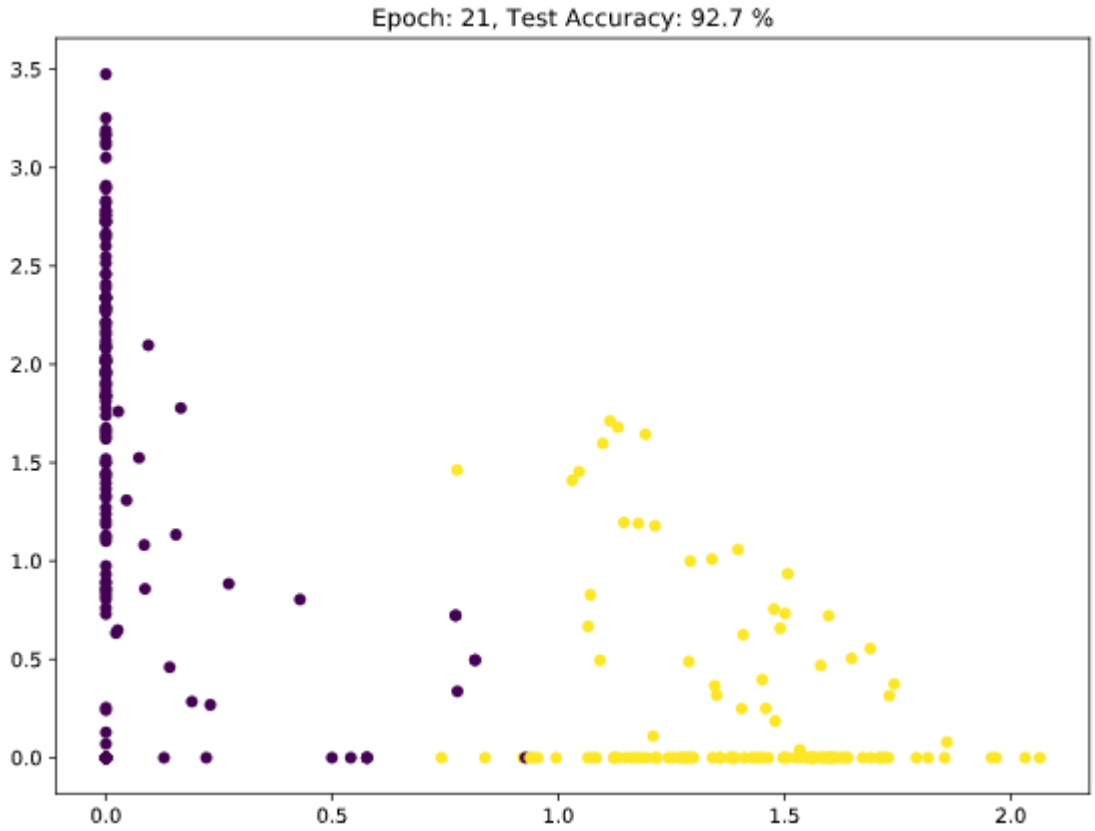
Neurons learn by updating their weights to output values that help them better distinguish between the different output classes in your dataset. You will make use of the `inp_to_out()` function you just built to visualize the output of two neurons in the first layer of the **Banknote Authentication** `model` as it learns.

The `model` you built in chapter 2 is ready for you to use, just like `X_test` and `y_test`. Paste `show_code(plot)` in the console if you want to check `plot()`.

You're performing heavy duty, once all is done, click through the graphs to watch the separation live!

```python
for i in range(0, 21):
    # Train model for 1 epoch
    h = model.fit(X_train, y_train, batch_size = 16, epochs = 1, verbose = 0)
    if i%4==0:
      # Get the output of the first layer
      layer_output = inp_to_out([X_test])[0]
```

```
    # Evaluate model accuracy for this epoch
    test_accuracy = model.evaluate(X_test, y_test)[1]

    # Plot 1st vs 2nd neuron output
    plot()
```



If you take a look at the graphs you can see how the neurons are learning to spread out the inputs based on whether they are fake or legit dollar bills. (A single fake dollar bill is represented as a purple dot in the graph) At the start the outputs are closer to each other, the weights are learned as epochs go by so that fake and legit dollar bills get a different, further and further apart output. Click in between the graphs fast, it's like a movie!

Building an autoencoder

Autoencoders have several interesting applications like anomaly detection or image denoising. They aim at producing an output identical to its inputs. The input will be compressed into a lower dimensional space, **encoded**. The model then learns to **decode** it back to its original form.

You will encode and decode the **MNIST** dataset of handwritten digits, the hidden layer will encode a 32-dimensional representation of the image, which originally consists of 784 pixels (28 x 28). The autoencoder will essentially learn to turn the 784 pixels original image into a compressed 32 pixels image and learn how to use that encoded representation to bring back the original 784 pixels image.

The `Sequential` model and `Dense` layers are ready for you to use. Let's build an autoencoder!

```
# Start with a sequential model
autoencoder = Sequential()

# Add a dense layer with input the original image pixels and neurons the encoded representation
autoencoder.add(Dense(32, input_shape=(784, ), activation="relu"))

# Add an output layer with as many neurons as the orginal image pixels
autoencoder.add(Dense(784, activation = "sigmoid"))

# Compile your model with adadelta
autoencoder.compile(optimizer = 'adadelta', loss = 'binary_crossentropy')

# Summarize your model structure
autoencoder.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 32)                25120
    _________________________________________________________________
    dense_2 (Dense)              (None, 784)               25872
    =================================================================
    Total params: 50,992
    Trainable params: 50,992
    Non-trainable params: 0
    _________________________________________________________________
```

Your autoencoder is now ready. Let's see what you can do with it!

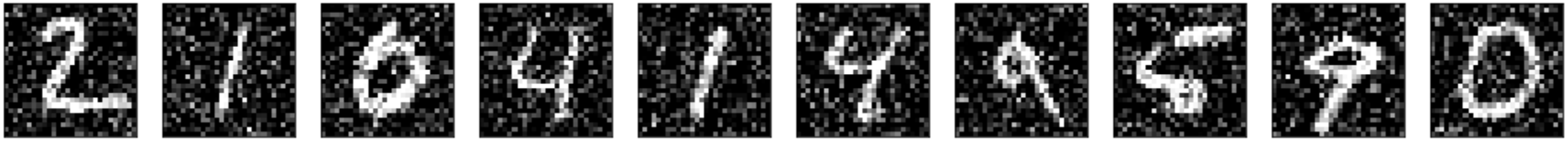
De-noising like an autoencoder

Okay, you have just built an `autoencoder` model. Let's see how it handles a more challenging task.

First, you will build a model that encodes images, and you will check how different digits are represented with `show_encodings()`. To build the encoder you will make use of your `autoencoder`, that has already being trained. You will just use the first half of the network, which contains the input and the bottleneck output. That way, you will obtain a 32 number output which represents the encoded version of the input image.

Then, you will apply your `autoencoder` to noisy images from `MNIST`, it should be able to clean the noisy artifacts.

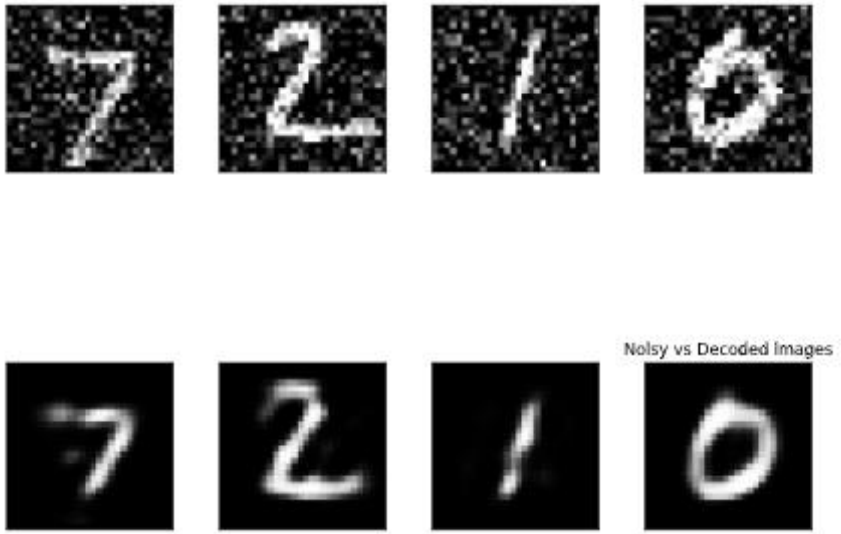
`X_test_noise` is loaded in your workspace. The digits in this noisy dataset look like this:



Apply the power of the autoencoder!

```
# Build your encoder by using the first layer of your autoencoder
encoder = Sequential()
encoder.add(autoencoder.layers[0])

# Encode the noisy images and show the encodings for your favorite number [0-9]
encodings = encoder.predict(X_test_noise)
show_encodings(encodings, number = 1)
```

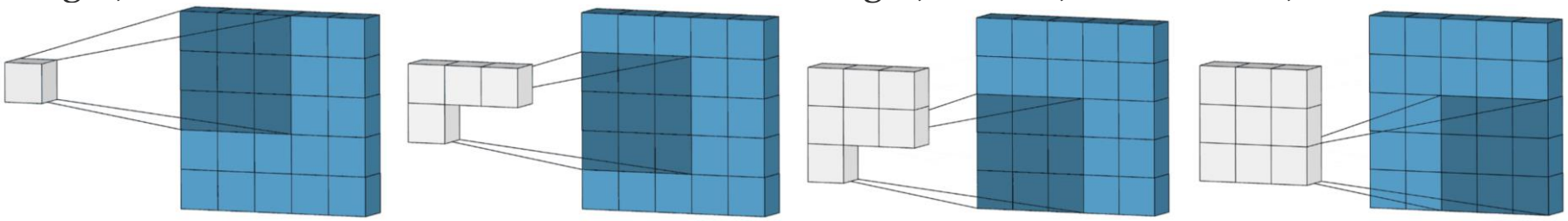
```
# Predict on the noisy images with your autoencoder
decoded_imgs = autoencoder.predict(X_test_noise)

# Plot noisy vs decoded images
compare_plot(X_test_noise, decoded_imgs)
```



The noise is gone now! You could get a better reconstruction by using a convolutional autoencoder. I hope this new model opened up your mind to the many possible architectures and non-classical ML problems that neural networks can solve :)

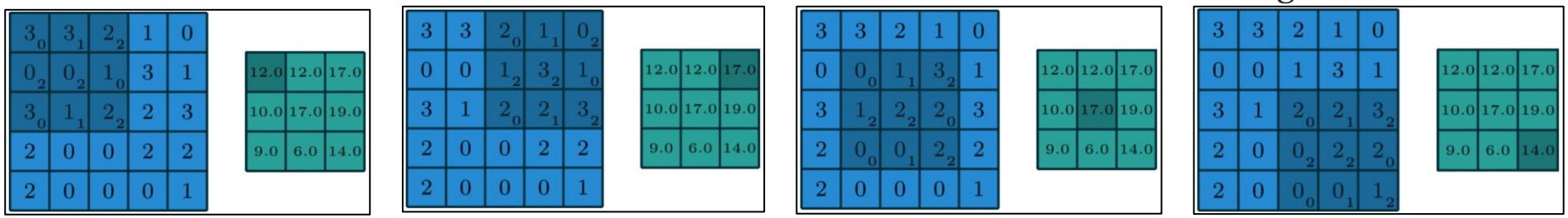
Intro to CNNs

Convolutional Neural Network model uses convolutional layers.
A convolution is a simple mathematical operation that preserves spatial relationships. When applied to images, it can detect relevant areas of interest like edges, corners, vertical lines, etc.

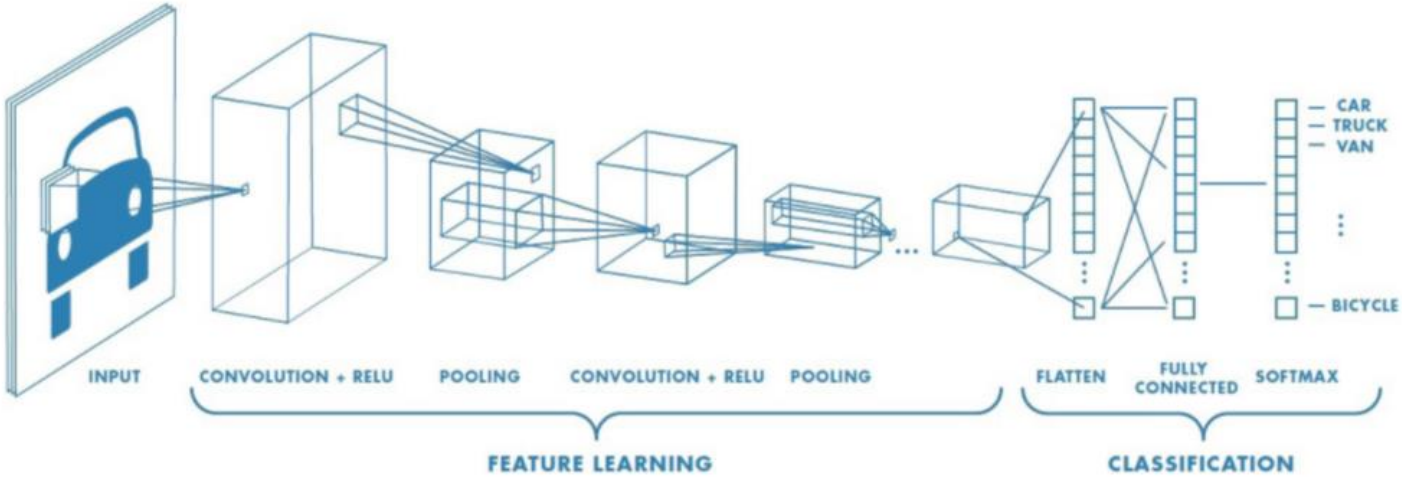


It consists of applying a filter also known as kernel of a given size, eg. 3x3 kernel. We center the kernel matrix of numbers as we slide through each pixel in the image, multiplying the kernel and pixel values at each location and averaging the sum of values obtained.

This effectively computes a new image where certain characteristics are amplified depending on the filter used. The network itself finds the best filter values and combines them to achieve a given task.

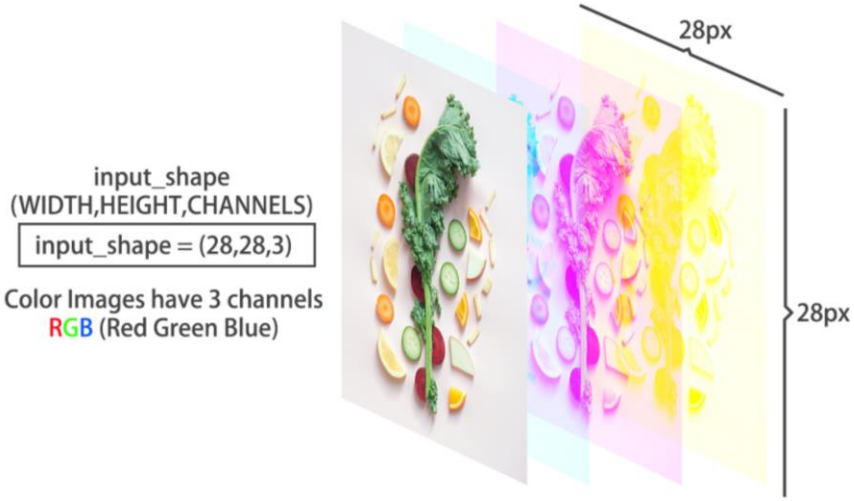


For a classification with many classes, CNNs tend to become very deep. Architecture consist of concatenations of convolutional layers (eg. pooling layers)



Convolutional layers perform feature learning, we then flatten the outputs into a unidimensional vector and pass it to fully connected layers that carry out classification.

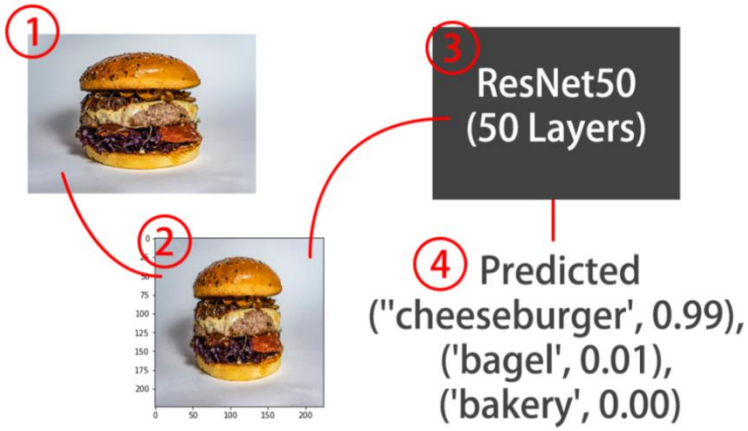
Images are 3-D tensors with width and height (pixels), and depth (color channels).
If images are black and white, the depth is 1 channel.
If images are colored, the depth is 3 channels (RGB).



```
# Import Conv2D layer and Flatten from keras layers
from keras.layers import Dense, Conv2D, Flatten
# Instantiate your model as usual
model = Sequential()

# Add a convolutional layer with 32 filters of size 3x3
model.add(Conv2D(filters=32,
                 kernel_size=3,
                 input_shape=(28, 28, 1),
                 activation='relu'))
# Add another convolutional layer
model.add(Conv2D(8, kernel_size=3, activation='relu'))

# Flatten the output of the previous layer
model.add(Flatten())
# End this multiclass model with 3 outputs and softmax
model.add(Dense(3, activation='softmax'))
```

ResNet50 is a 50 layer-deep model that performs well on Imagenet Dataset, a huge dataset of more than 14 million images. ResNet50 can distinguish between 1000 different classes.

Keras makes it easy for us to use this pre-trained model. We just need to prepare the input image to be understood by the model, predict the processed image, and decode the predictions.




```
# Import image from keras preprocessing
from keras.preprocessing import image
# Import preprocess_input from keras applications resnet50
from keras.applications.resnet50 import preprocess_input
# Load the image with the right target size for your model
img = image.load_img(img_path, target_size=(224, 224))
```

Transfer Learning

```
# Turn it into an array
img = image.img_to_array(img)
# Expand the dimensions so that it's understood by our network:
# img.shape turns from (224, 224, 3) into (1, 224, 224, 3)
img = np.expand_dims(img, axis=0)
# Pre-process the img in the same way training images were
img = preprocess_input(img)
```

```
# Import ResNet50 and decode_predictions from keras.applications.resnet50
from keras.applications.resnet50 import ResNet50, decode_predictions
# Instantiate a ResNet50 model with imagenet weights
model = ResNet50(weights='imagenet')
# Predict with ResNet50 on our img
preds = model.predict(img)
# Decode predictions and print it
print('Predicted:', decode_predictions(preds, top=1)[0])
```

```
Predicted: [('n07697313', 'cheeseburger', 0.9868016)]
```

Building a CNN model

Building a CNN model in Keras isn't much more difficult than building any of the models you've already built throughout the course! You just need to make use of convolutional layers.

You're going to build a shallow convolutional `model` that classifies the **MNIST** digits dataset. The same one you de-noised with your autoencoder! The images are 28 x 28 pixels and **just have one channel**, since they are black and white pictures.

Go ahead and build this small convolutional model!

```
# Import the Conv2D and Flatten layers and instantiate model
from keras.layers import Conv2D,Flatten
model = Sequential()

# Add a convolutional layer of 32 filters of size 3x3
model.add(Conv2D(32, kernel_size = 3, input_shape = (28, 28, 1), activation = 'relu'))
```

```
# Add a convolutional layer of 16 filters of size 3x3
model.add(Conv2D(16, kernel_size = 3, activation = 'relu'))

# Flatten the previous layer output
model.add(Flatten())

# Add as many outputs as classes with softmax activation
model.add(Dense(10, activation = 'softmax'))
```

You can see that the key concepts are the same, you just have to use new layers!

Looking at convolutions

Inspecting the activations of a convolutional layer is a cool thing. You have to do it at least once in your lifetime!

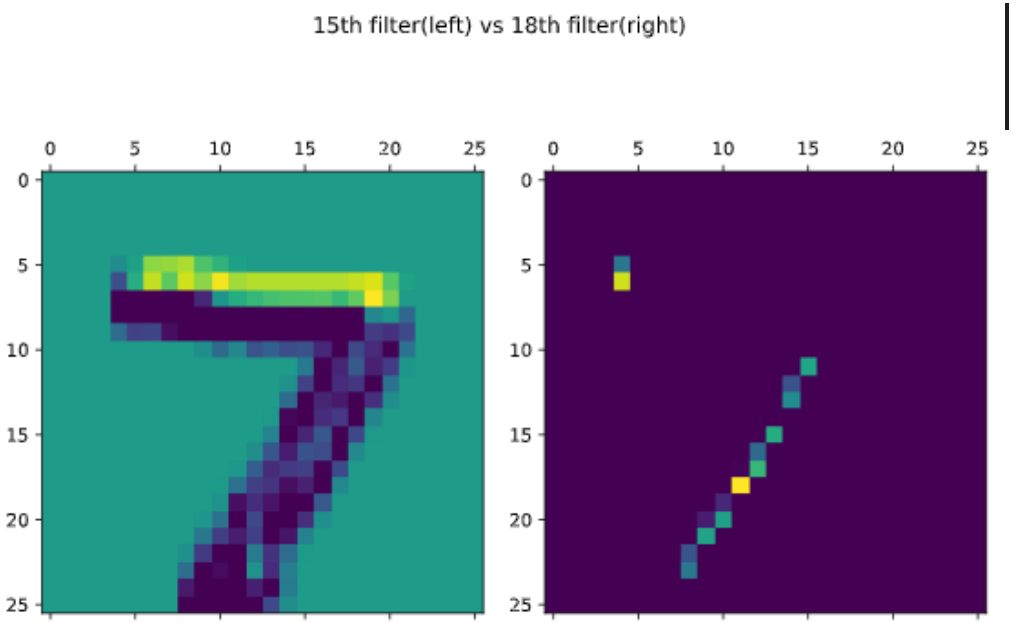
To do so, you will build a new model with the Keras `Model` object, which takes in a list of inputs and a list of outputs. The output you will provide to this new model is the first convolutional layer outputs when given an **MNIST** digit as input image.

The convolutional `model` you built in the previous exercise has already been trained for you. It can now correctly classify **MNIST** handwritten images. You can check it with `model.summary()` in the console.

Let's look at the convolutional masks that were learned in the first convolutional layer of this model!

```
# Obtain a reference to the outputs of the first layer
first_layer_output = model.layers[0].output

# Build a model using the model input and the first layer output
first_layer_model = Model(inputs = model.layers[0].input, outputs = first_layer_output)

# Use this model to predict on X_test
activations = first_layer_model.predict(X_test)

# Plot the first digit of X_test for the 15th filter
axs[0].matshow(activations[0,:,:,14], cmap = 'viridis')
```

```
# Do the same but for the 18th filter now
axs[1].matshow(activations[0,:,:,17], cmap = 'viridis')
plt.show()
```



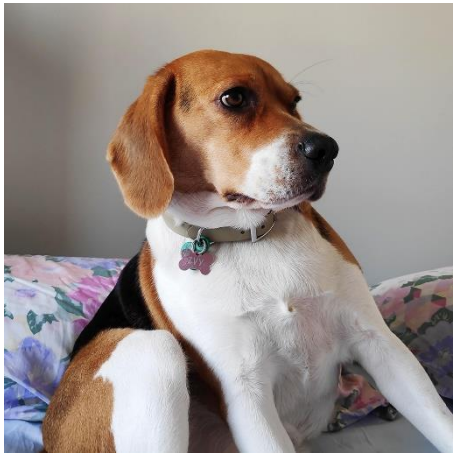
Each neuron filter of the first layer learned a different convolution. The 15th filter (a.k.a convolutional mask) learned to detect horizontal traces in your digits. On the other hand, filter 18th seems to be checking for vertical traces.

Preparing your input image

The original **ResNet50 model** was trained with images of size **224 x 224 pixels** and a number of preprocessing operations; like the subtraction of the mean pixel value in the training set for all training images. You need to pre-process the images you want to predict on in the same way.

When predicting on a single image you need it to fit the model's input shape, which in this case looks like this: (batch-size, width, height, channels),`np.expand_dims` with parameter `axis = 0` adds the batch-size dimension, representing that a single image will be passed to predict. This batch-size dimension value is 1, since we are only predicting on one image.

You will go over these preprocessing steps as you prepare this dog's (named Ivy) image into one that can be classified by **ResNet50**.



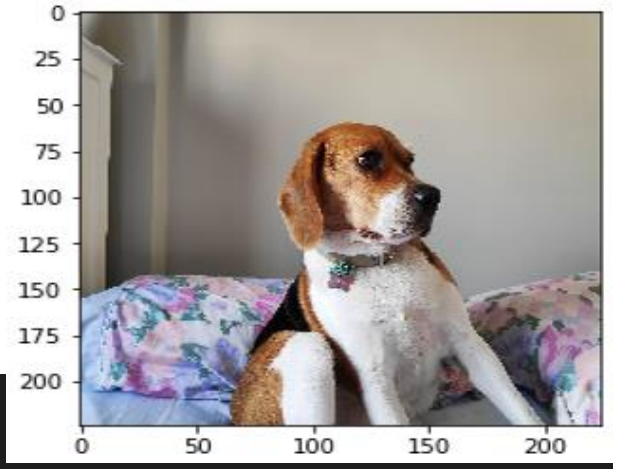
```
# Import image and preprocess_input
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input

# Load the image with the right target size for your model
img = image.load_img(img_path, target_size = (224, 224))
```

```
# Turn it into an array
img_array = image.img_to_array(img)

# Expand the dimensions of the image, this is so that it fits the expected model input format
img_expanded = np.expand_dims(img_array, axis = 0)

# Pre-process the img in the same way original images were
img_ready = preprocess_input(img_expanded)
```

Ivy is now ready for ResNet50. Do you know this dog's breed? Let's see what this model thinks it is!

Using a real world model

Okay, so Ivy's picture is ready to be used by **ResNet50**. It is stored in `img_ready` and now looks like this:



ResNet50 is a model trained on the **Imagenet dataset** that is able to distinguish between 1000 different labeled objects. **ResNet50** is a deep model with 50 layers, you can check it in 3D **here**.

`ResNet50` and `decode_predictions` have both been imported from `keras.applications.resnet50` for you.

It's time to use this trained model to find out Ivy's breed!

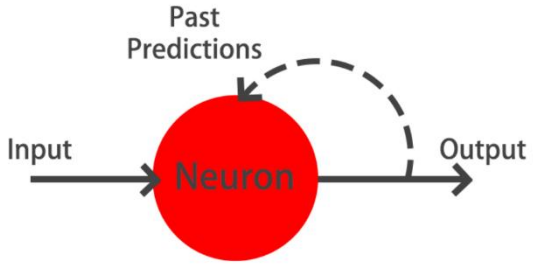
```
# Instantiate a ResNet50 model with 'imagenet' weights
model = ResNet50(weights='imagenet')

# Predict with ResNet50 on your already processed img
preds = model.predict(img_ready)

# Decode the first 3 predictions
print('Predicted:', decode_predictions(preds, top=3)[0])
```

```
<script.py> output:
    Predicted: [('n02088364', 'beagle', 0.8280003), ('n02089867', 'Walker_hound', 0.12915272), ('n02089973', 'English_foxhound', 0.03711732)]
```

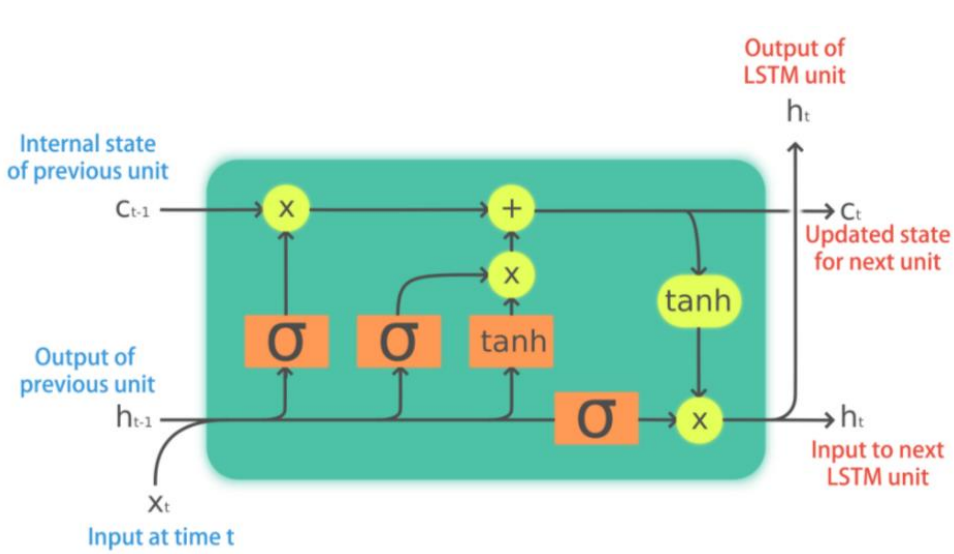
Now you know Ivy is quite probably a Beagle and that deep learning models that have already been trained for you are easy to use!

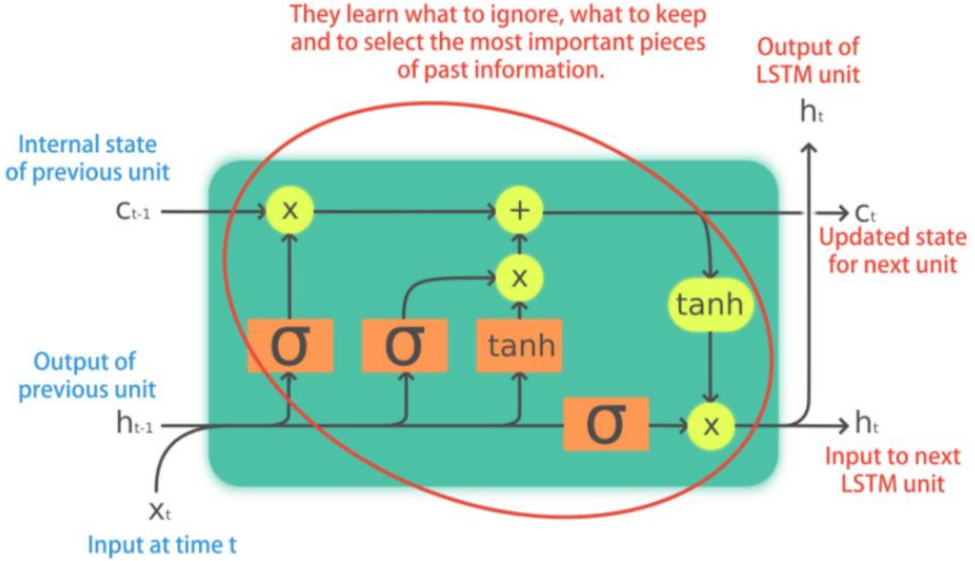
Intro to LSTMs

Recurrent neural network (RNN) can use past predictions in order to infer new ones. This allows to solve problems where there is a dependence on past inputs.



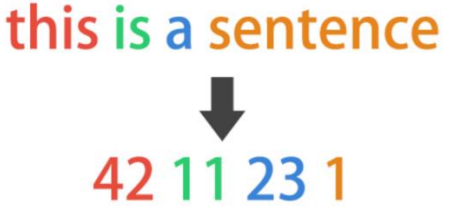
Long Short Term Memory (LSTM) networks are a type of RNN network.
LSTM neurons are pretty complex, they are actually called units or cells. They have an internal state that is passed between units, you can see this as a memory of past steps. A unit receives the internal state, an output from the previous unit, and a new input at time t. Then it updates the state and produces a new output that is returned, as well as passed as an input to the following unit.





LSTMs have been used for image captioning, speech to text, text translation, document summarization, text generation, musical composition,and many more.

Using LSTMs to predict the next word in a sentence
First convert text to numbers, to be used as inputs to an embedding layer. Embedding layers learn to represent words as vectors of a predetermined size. These vectors encode meaning and are used by subsequent layers.

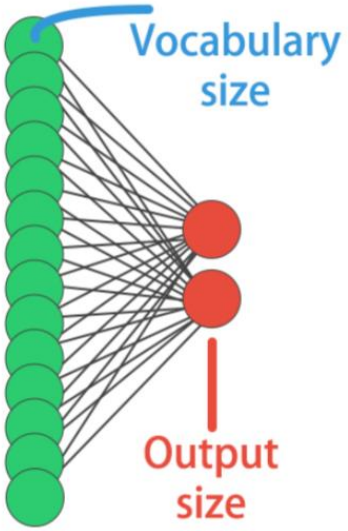


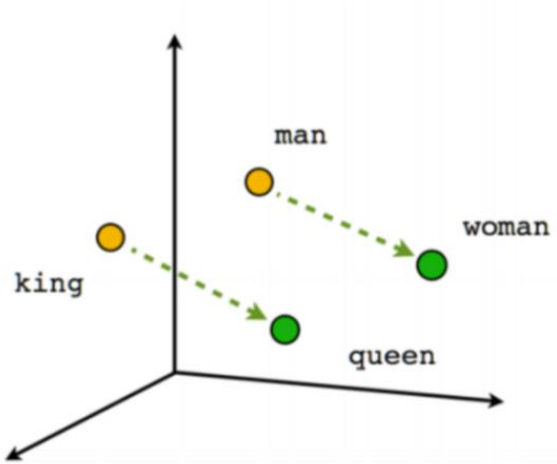
```
text = 'Hi this is a small sentence'

# We choose a sequence length
seq_len = 3

# Split text into a list of words
words = text.split()
```

```
['Hi', 'this', 'is', 'a', 'small', 'sentence']
```

```
# Make lines
lines = []
for i in range(seq_len, len(words) + 1):
  line = ' '.join(words[i-seq_len:i])
  lines.append(line)
```

```
['Hi this is', 'this is a', 'is a small', 'a small sentence']
```





feed model with 2 words and it will predict 3rd one

```
# Import Tokenizer from keras preprocessing text
from keras.preprocessing.text import Tokenizer
# Instantiate Tokenizer
tokenizer = Tokenizer()
# Fit it on the previous lines
tokenizer.fit_on_texts(lines)
# Turn the lines into numeric sequences
sequences = tokenizer.texts_to_sequences(lines)
```

```
array([[5, 3, 1], [3, 1, 2], [1, 2, 4], [2, 4, 6]])
```

```
print(tokenizer.index_word)
```

```
{1: 'is', 2: 'a', 3: 'this', 4: 'small', 5: 'hi', 6: 'sentence'}
```

```
# Import Dense, LSTM and Embedding layers
from keras.layers import Dense, LSTM, Embedding
model = Sequential()
# Vocabulary size
vocab_size = len(tokenizer.index_word) + 1
# Starting with an embedding layer
model.add(Embedding(input_dim=vocab_size, output_dim=8, input_length=2))
# Adding an LSTM layer
model.add(LSTM(8))

# Adding a Dense hidden layer
model.add(Dense(8, activation='relu'))
# Adding an output layer with softmax
model.add(Dense(vocab_size, activation='softmax'))
```

Note: dict starts at 1, not zero

Text prediction with LSTMs

During the following exercises you will build a toy LSTM model that is able to predict the next word using a small text dataset. This dataset consist of cleaned quotes from the **The Lord of the Ring** movies. You can find them in the `text` variable.

```
You're working with this small chunk of The Lord of The Ring quotes stored in the text variable:
==========================================================================================
 It is not the strength of the body but the strength of the spirit.
 It is useless to meet revenge with revenge it will heal nothing.
 Even the smallest person can change the course of history.
 All we have to decide is what to do with the time that is given us.
 The burned hand teaches best. After that, advice about fire goes to the heart.
```

You will turn this `text` into `sequences` of **length 4** and make use of the Keras `Tokenizer` to prepare the features and labels for your model!

The Keras `Tokenizer` is already imported for you to use. It assigns a unique number to each unique word, and stores the mappings in a dictionary. This is important since the model deals with numbers but we later will want to decode the output numbers back into words.

```
# Split text into an array of words
```

```
words = text.split()

# Make sentences of 4 words each, moving one word at a time
sentences = []
for i in range(4, len(words)):
  sentences.append(' '.join(words[i-4:i]))

# Instantiate a Tokenizer, then fit it on the sentences
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sentences)

# Turn sentences into a sequence of numbers
sequences = tokenizer.texts_to_sequences(sentences)
print("Sentences: \n {} \n Sequences: \n {}".format(sentences[:5],sequences[:5]))
```

```
<script.py> output:
    Sentences:
     ['it is not the', 'is not the strength', 'not the strength of', 'the strength of the', 'strength of the body']
     Sequences:
     [[5, 2, 42, 1], [2, 42, 1, 6], [42, 1, 6, 4], [1, 6, 4, 1], [6, 4, 1, 10]]
```

Your sentences are now sequences of numbers, check that identical words are assigned the same number.

Build your LSTM model

You've already prepared your sequences of text. It's time to build your LSTM model!

Remember your sequences had 4 words each, your model will be trained on the first three words of each sequence, predicting the 4th one. You are going to use an `Embedding` layer that will essentially learn to turn words into vectors. These vectors will then be passed to a simple `LSTM` layer. Our output is a `Dense` layer with as many neurons as words in the vocabulary and `softmax` activation. This is because we want to obtain the highest probable next word out of all possible words.

The size of the vocabulary of words (the unique number of words) is stored in `vocab_size`.

```
# Import the Embedding, LSTM and Dense layer
from keras.layers import Embedding, LSTM, Dense

model = Sequential()
```

```
# Add an Embedding layer with the right parameters
model.add(Embedding(input_dim = vocab_size, input_length = 3, output_dim = 8))

# Add a 32 unit LSTM layer
model.add(LSTM(32))

# Add a hidden Dense layer of 32 units and an output layer of vocab_size with softmax
model.add(Dense(32, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
model.summary()
```

```
<script.py> output:
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    embedding_1 (Embedding)      (None, 3, 8)              352
    _________________________________________________________________
    lstm_1 (LSTM)                (None, 32)                5248
    _________________________________________________________________
    dense_1 (Dense)              (None, 32)                1056
    _________________________________________________________________
    dense_2 (Dense)              (None, 44)                1452
    =================================================================
    Total params: 8,108
    Trainable params: 8,108
    Non-trainable params: 0
    _________________________________________________________________
```

That's a nice looking model you've built! You'll see that this model is powerful enough to learn text relationships, we aren't using a lot of text in this tiny example and our sequences are quite short. This model is to be trained as usual, you would just need to compile it with an optimizer like adam and use crossentropy loss. This is because we have modeled this next word prediction task as a classification problem with all the unique words in our vocabulary as candidate classes.

Decode your predictions

Your LSTM `model` has already been trained (details in the previous exercise success message) so that you don't have to wait. It's time to **define a function** that decodes its predictions. The trained `model` will be passed as a default parameter to this function.

Since you are predicting on a model that uses the softmax function, numpy's `argmax()` can be used to obtain the index/position representing the most probable next word out of the output vector of probabilities.

The `tokenizer` you previously created and fitted, is loaded for you. You will be making use of its internal `index_word` dictionary to turn the `model`'s next word prediction (which is an integer) into the actual written word it represents.

You're very close to experimenting with your model!

```
def predict_text(test_text, model = model):
  if len(test_text.split()) != 3:
    print('Text input should be 3 words!')
    return False

  # Turn the test_text into a sequence of numbers
  test_seq = tokenizer.texts_to_sequences([test_text])
  test_seq = np.array(test_seq)

  # Use the model passed as a parameter to predict the next word
  pred = model.predict(test_seq).argmax(axis = 1)[0]

  # Return the word that maps to the prediction
  return tokenizer.index_word[pred]
```

It's finally time to try out your model and see how well it does!

Test your model!

The function you just built, `predict_text()`, is ready to use. *Remember that the model object is already passed by default as the second parameter so you just need to provide the function with your 3 word sentences.*

Try out these strings on your LSTM model:

- 'meet revenge with'
- 'the course of'
- 'strength of the'

Which sentence could be made with the word output from the sentences above?

```
In [2]: predict_text('meet revenge with')
Out[2]: 'revenge'

In [3]: predict_text('the course of')
Out[3]: 'history'

In [4]: predict_text('strength of the')
Out[4]: 'spirit'
```

- ○ A **worthless gnome** is **king**
- ◉ **Revenge** is your **history** and **spirit**
- ○ Take a **sword** and **ride** to **Florida**

Course completed!

You have completed 44 exercises and 14 lessons!

Recap topics covered:

- Basics of neural networks
- Built sequential models and models for regression
- Solving regression, binary classification, multi-class and multi-label problems with neural networks
- Activation functions
- Carried out hyperparameter tuning
- Turning your keras models into sklearn estimators
- Used autoencoders and de-noised images with them
- Key concepts about CNN networks
- Used the pre-trained resnet50 model to classify images
- Visualized convolutions for the MNIST dataset
- Learned about LSTMs concepts and worked with text and embedding layers
- Used a lot of different datasets along the way
- Learned a lot of keras utility functions

Next steps:

- Go deeper into CNNs
- Go deeper into LSTMs
- Check how to work with the Keras functional API which allows you to build more powerful models with shared layers and several branches
- Check the original things that can be created with Generative Adversarial Networks (GAN) and take on some Deeplearning projects of your own.

Happy learning!