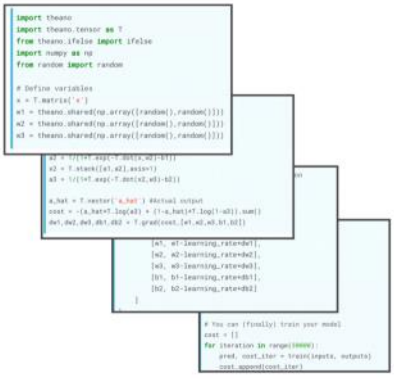
Introduction to Deep Learning with Keras

Monday, 17 August 2020 7:30 PM

Introducing keras

Theano vs Keras



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

# Define model and add layers
model = Sequential()
model.add(Dense(2,input_shape=(2,),activation='sigmoid'))
model.add(Dense(1,activation='sigmoid'))

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

# Train model
model.fit(inputs,outputs)
```

Keras

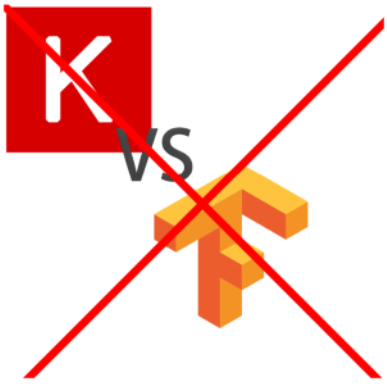
- Deep Learning Framework
- Enables fast experimentation
- Runs on top of other frameworks
- Written by François Chollet

Why use Keras?

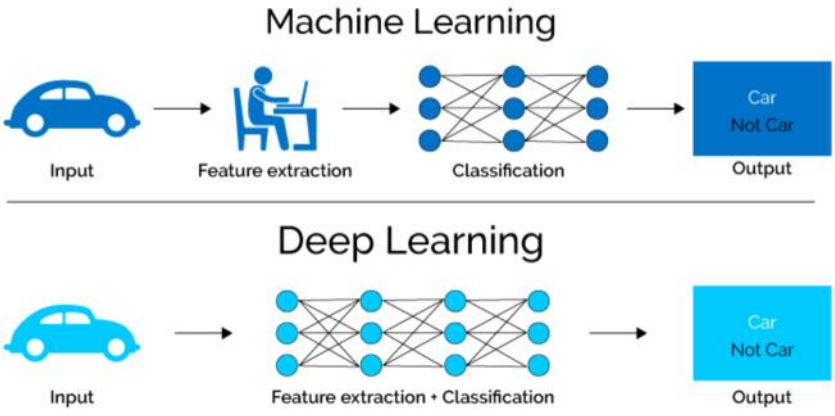
- Fast industry-ready models
- For beginners and experts
- Less code
- Build any architecture
- Deploy models in multiple platforms

Keras + TensorFlow

- TensorFlow's high level framework of choice
- Keras is complementary to TensorFlow
- You can use TensorFlow for low level features



Feature Engineering



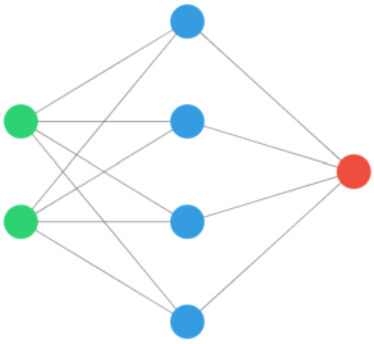
Unstructured data

So, when to use neural networks?

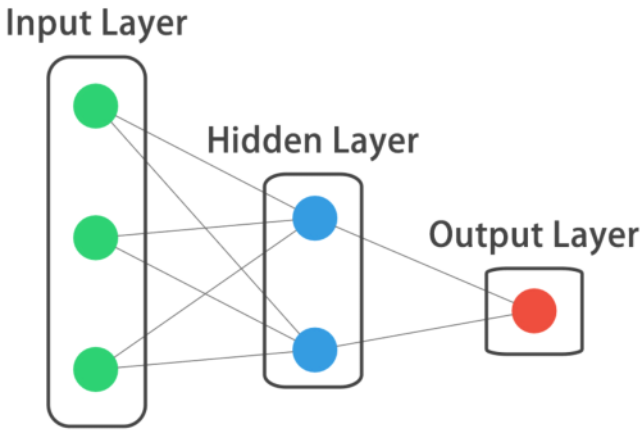
- Dealing with unstructured data
- Don't need easily interpretable results
- You can benefit from a known architecture

Example: Classify images of cats and dogs

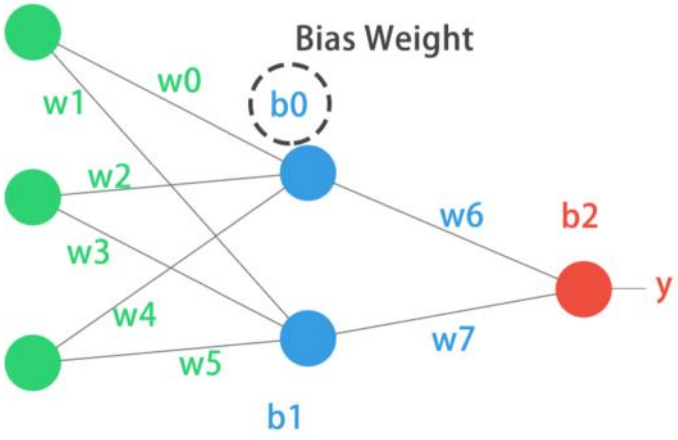
- **Images -> Unstructured data**
- You don't care about why the network knows it's a cat or a dog
- You can benefit from convolutional neural networks



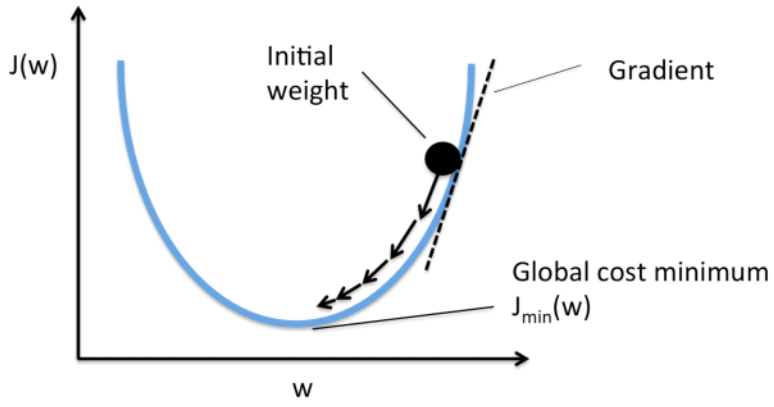
A neural network?



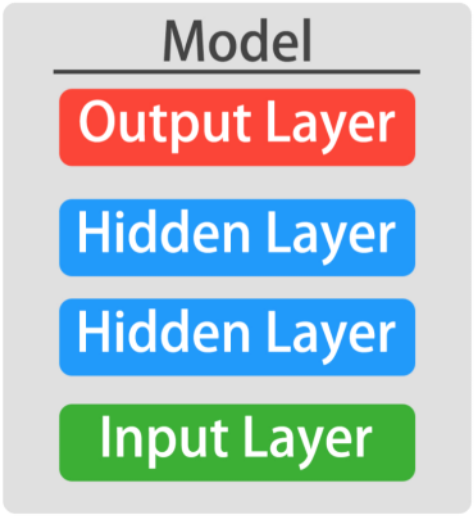
Parameters



Gradient descent

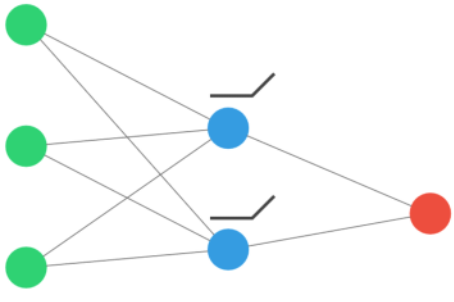


The sequential API

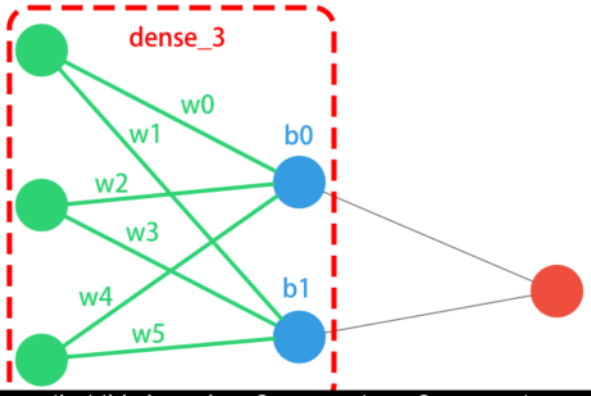


Adding activations

```
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))
```



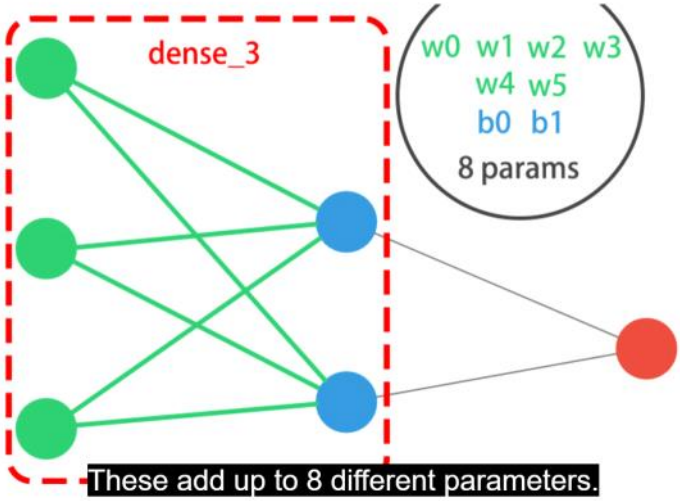
Visualize parameters



That's why we see that this layer has 8 parameters: 6 parameters or weights come

missing 2 parameters come from the bias weights, b0 and b1, 1 per each neuron in the hidden layer.

Visualize parameters



These add up to 8 different parameters.

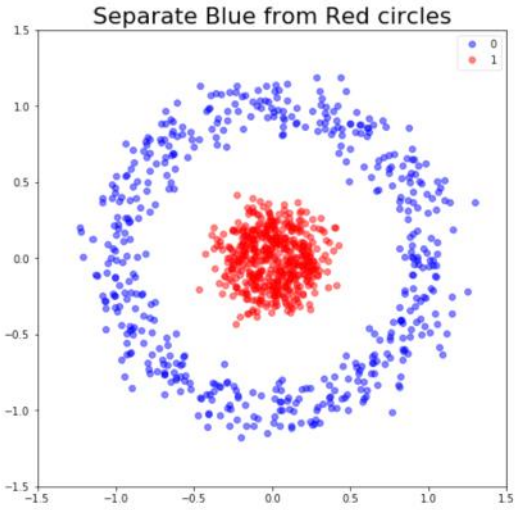
Summarize your model!

```
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
```

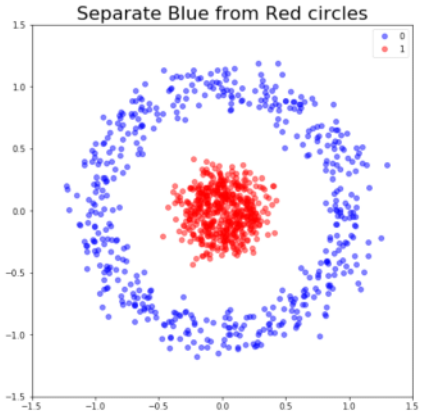
Going Deeper

When to use binary classification?



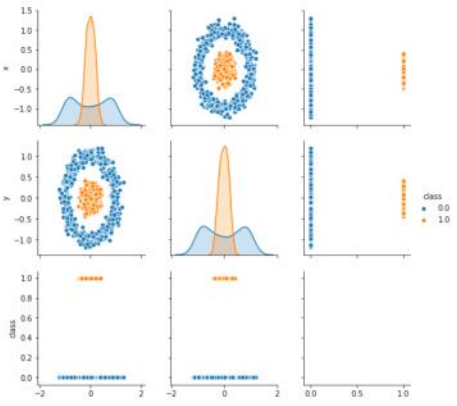
Our dataset

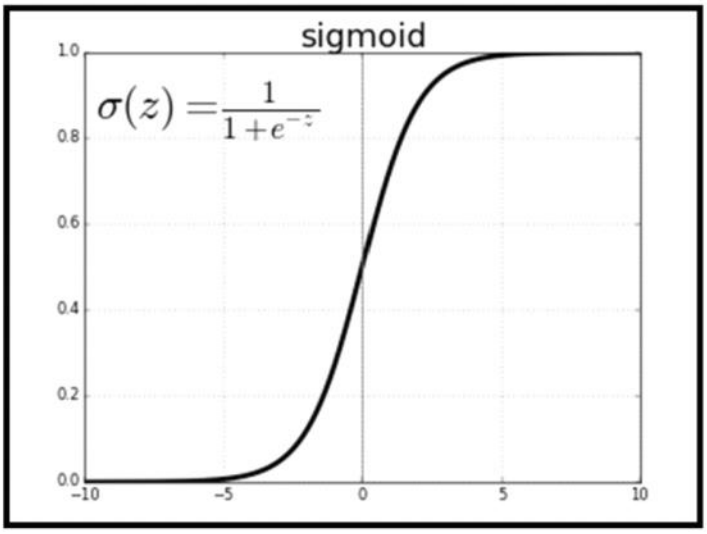
coordinates	labels
[0.242, 0.038]	1
[0.044, -0.057]	1
[-0.787, -0.076]	0



Pairplots

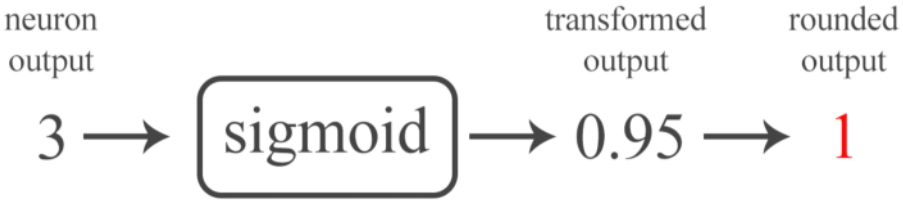
```
import seaborn as sns

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





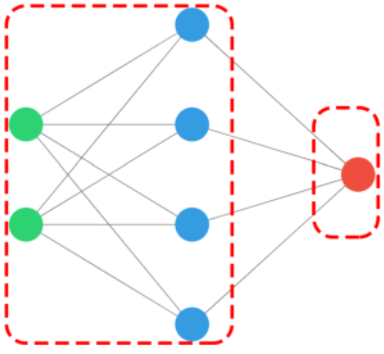
Output Layer

The sigmoid function



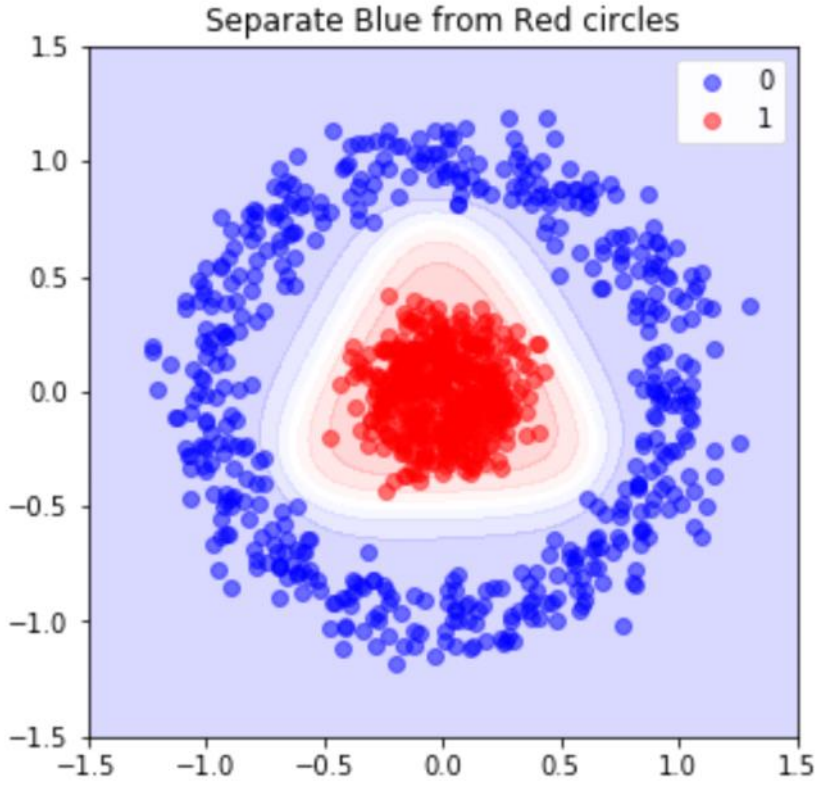
Let's build it

```
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'))
```

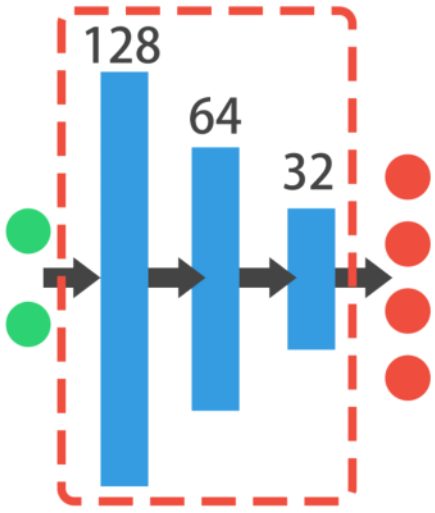


Compiling, training, predicting

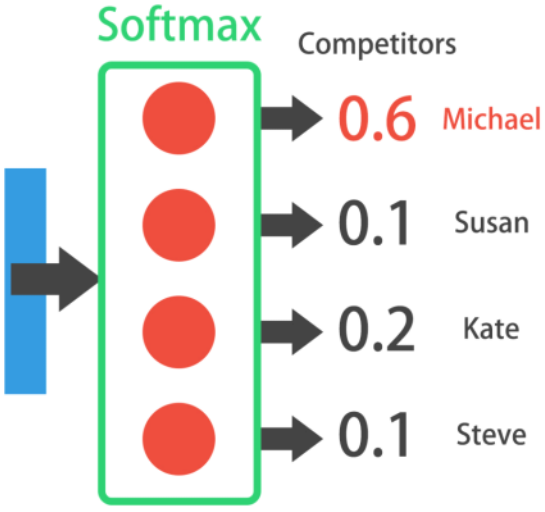
```
# Compile model
model.compile(optimizer='sgd',
              loss='binary_crossentropy')

# Train model
model.train(coordinates, labels, epochs=20)

# Predict with trained model
preds = model.predict(coordinates)
```



The architecture



The output layer



Multi-class model

```
# Instantiate a sequential model
# ...

# Add an input and hidden layer
# ...

# Add more hidden layers
# ...

# Add your output layer
model.add(Dense(4, activation='softmax')
```

Preparing a dataset

```
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)
```

One-hot encoding

Label Encoding

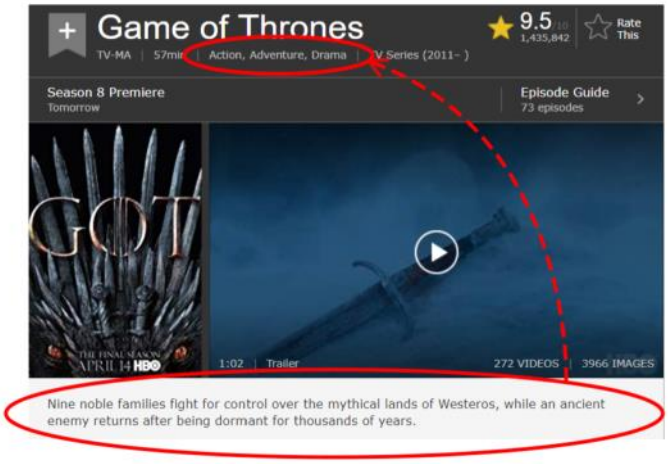
Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

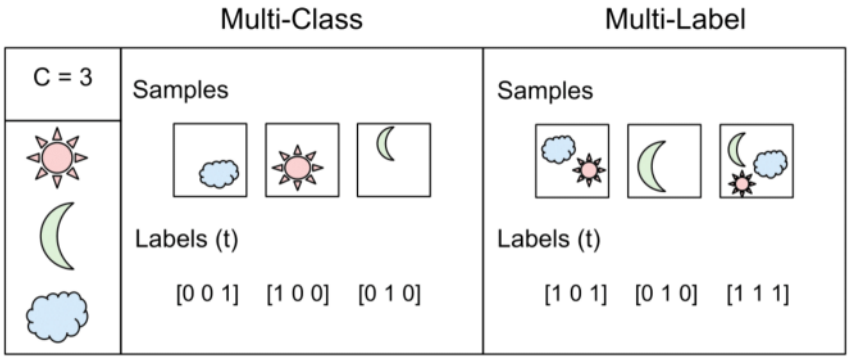
→

One Hot Encoding

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

Real world examples





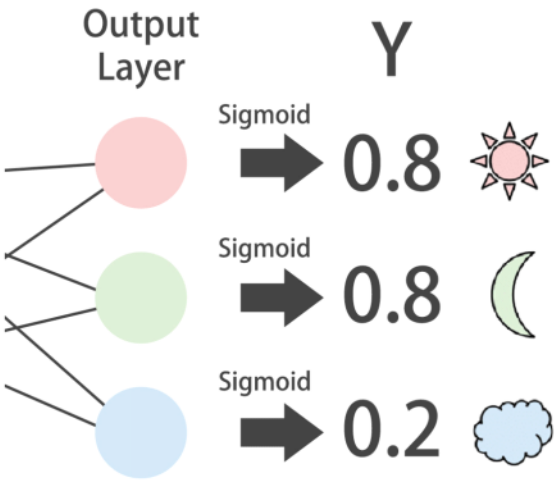
However in a multi-label problem each individual in the sample can have all, none or a subset of the available classes.

https://gombru.github.io/2018/05/23/

The architecture

```
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'))
```

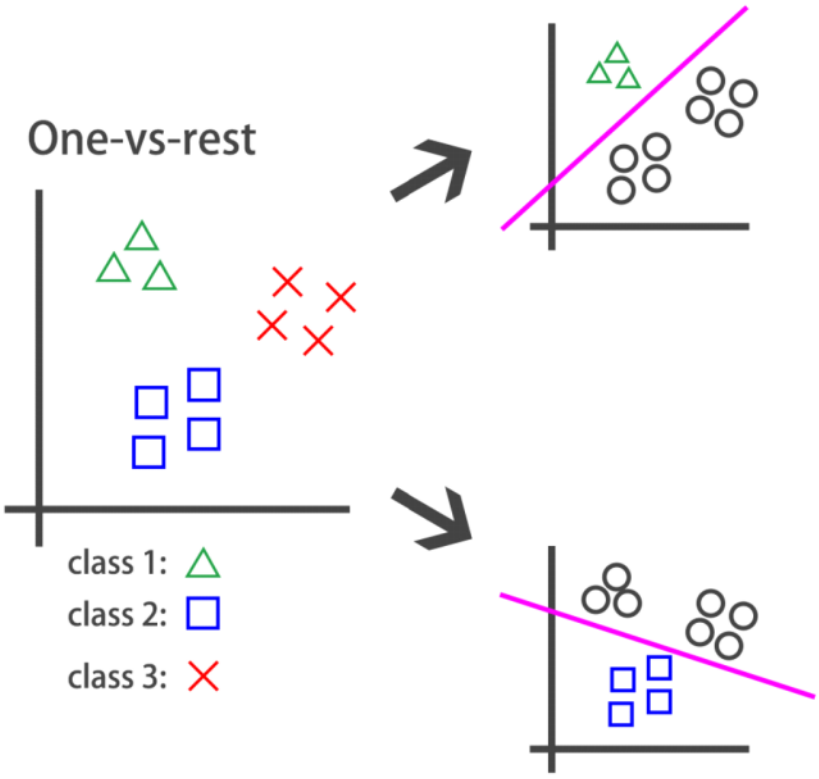
Sigmoid outputs



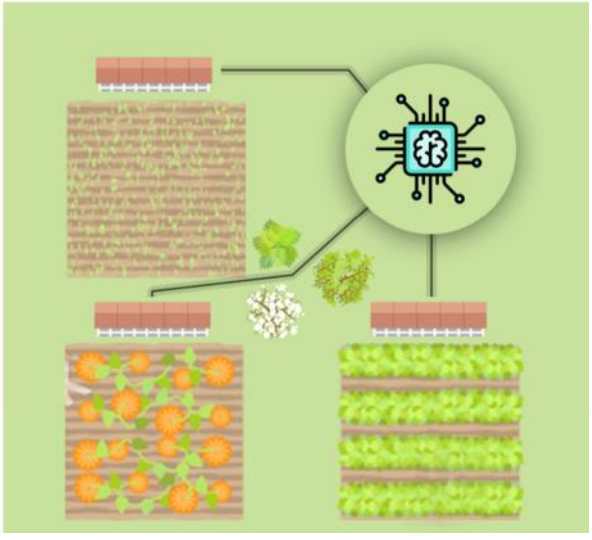

```
# Compile the model with binary crossentropy
model.compile(optimizer='adam', loss='binary_crossentropy')
```

```
# 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



An irrigation machine

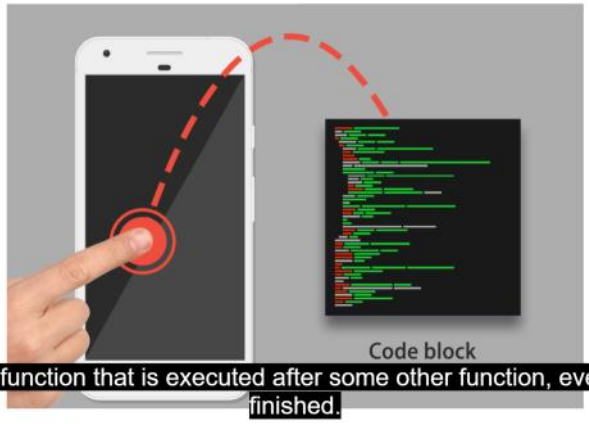
Sensor measurements

	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

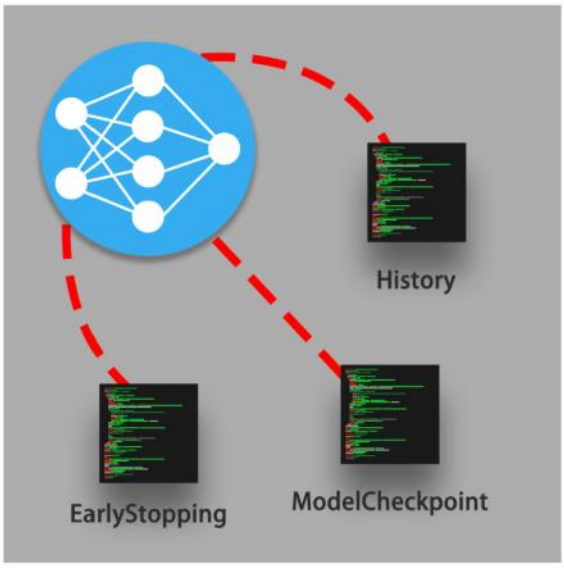
What is a callback?



A callback is a function that is executed after some other function, event, or task has finished.

For instance, when you touch your phone screen, a block of code that identifies the type of gesture will be triggered.

Callbacks in Keras



A callback you've been missing

```
# 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]
```

A callback you've been missing

```
# 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'])
```

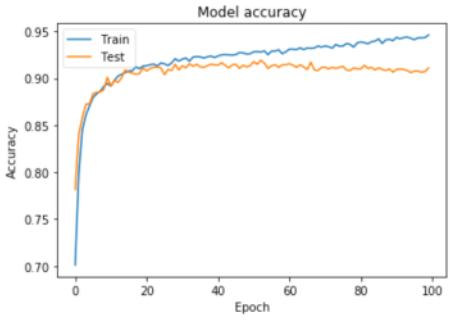
```
[0.7753975939750672, ..., 0.4155936544282096]
```

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

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

History plots

```
# 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()
```



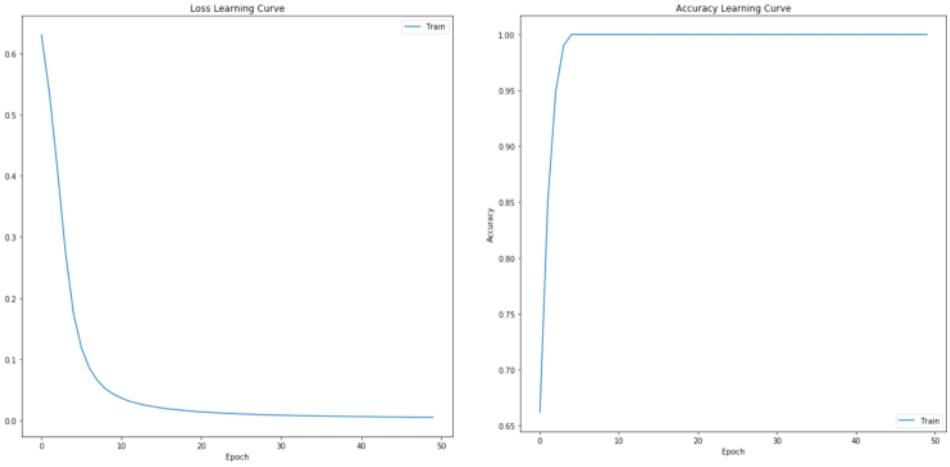
Early stopping

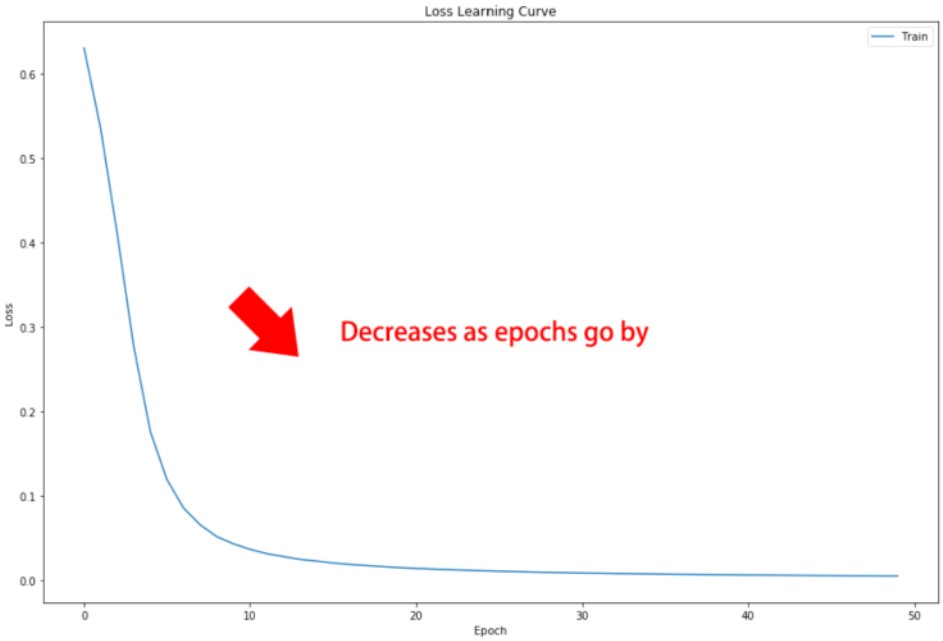
```
# Import early stopping from keras callbacks
from keras.callbacks import EarlyStopping

# 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])
```

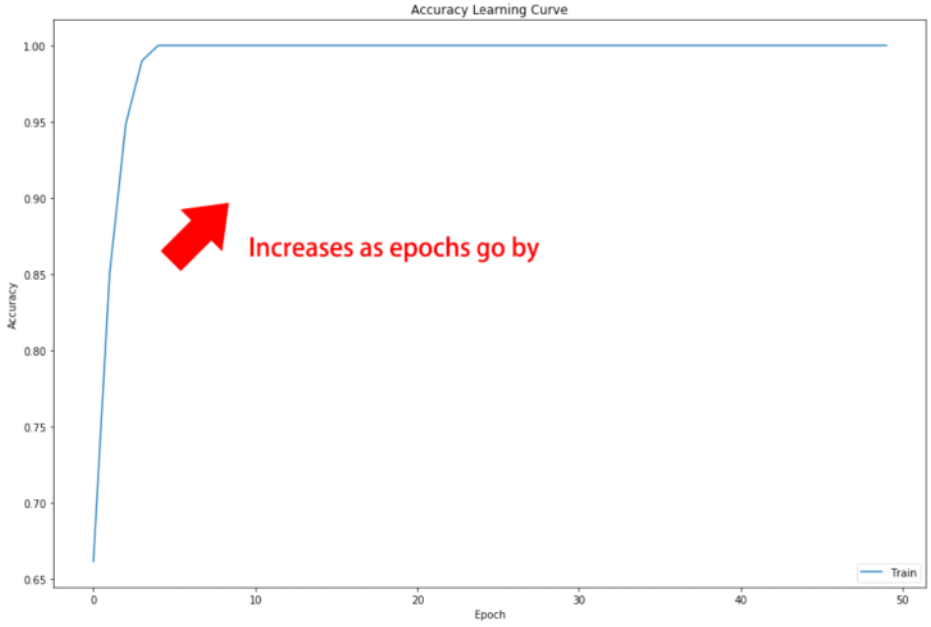
Model checkpoint

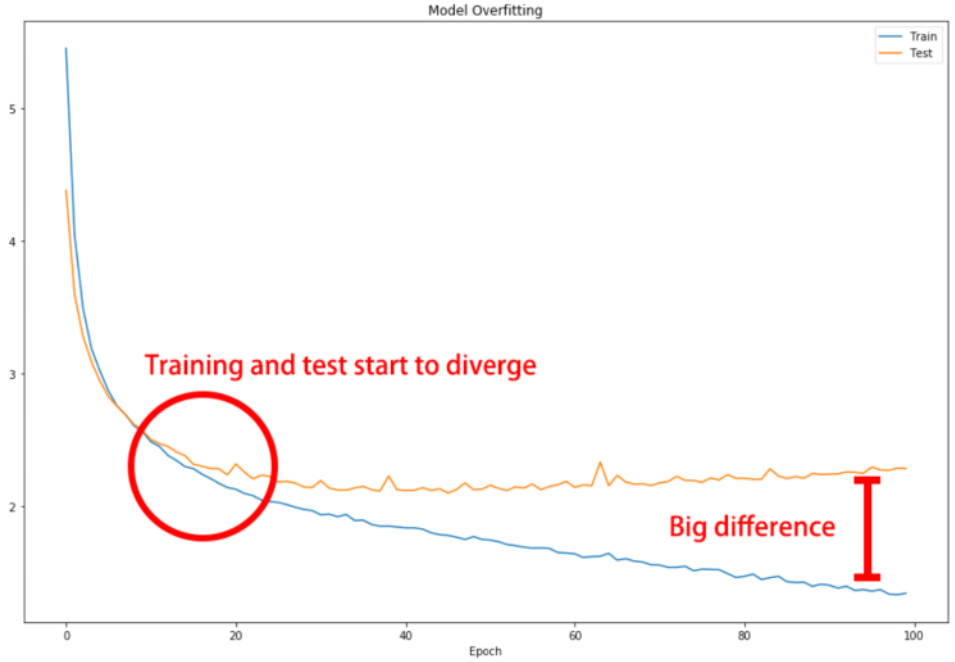
```
# Import model checkpoint from keras callbacks
from keras.callbacks import ModelCheckpoint

# 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])
```

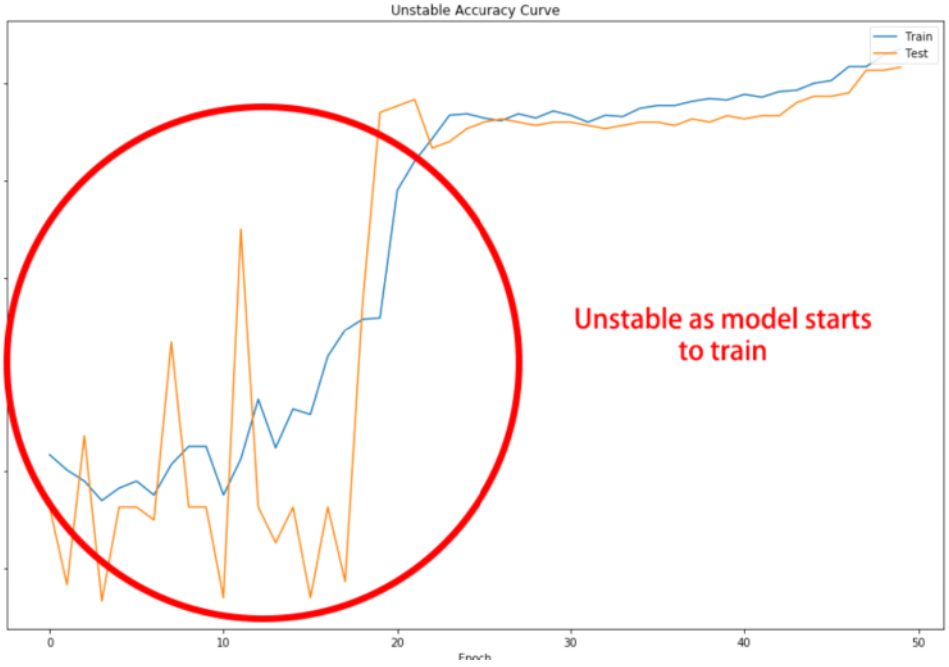
Improving Your Model Performance



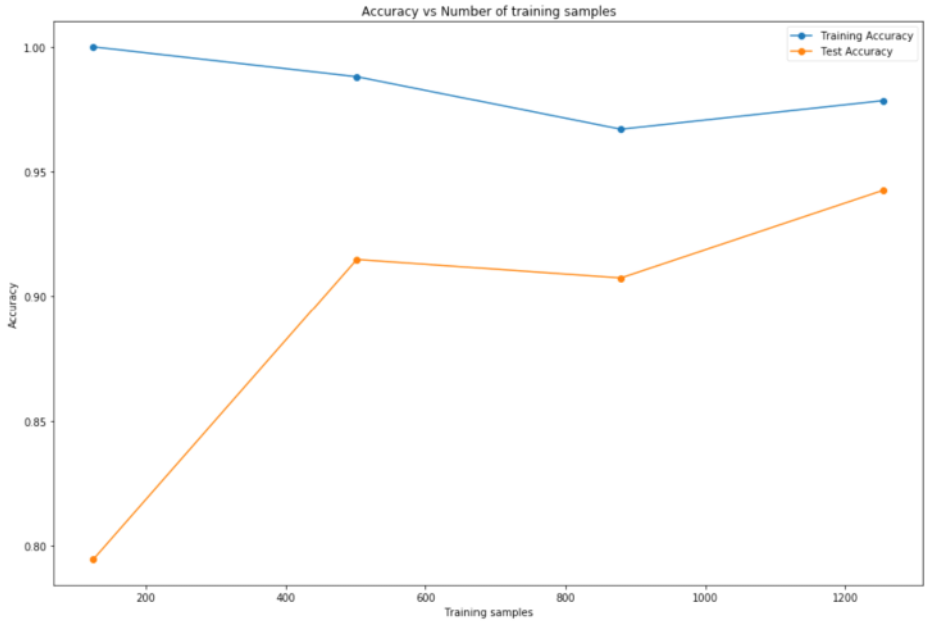

Loss Learning Curve
Train
Loss
Epoch
Decreases as epochs go by

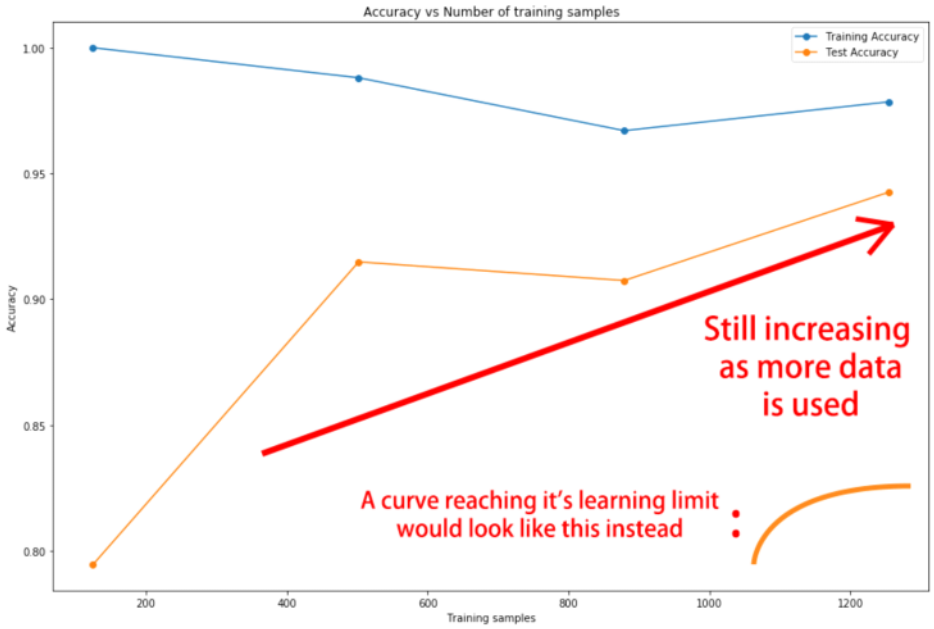

Accuracy Learning Curve
Train
Accuracy
Epoch
Increases as epochs go by


Model Overfitting
Train
Test
Training and test start to diverge
Big difference
Epoch


Unstable Accuracy Curve
Train
Test
Unstable as model starts
to train

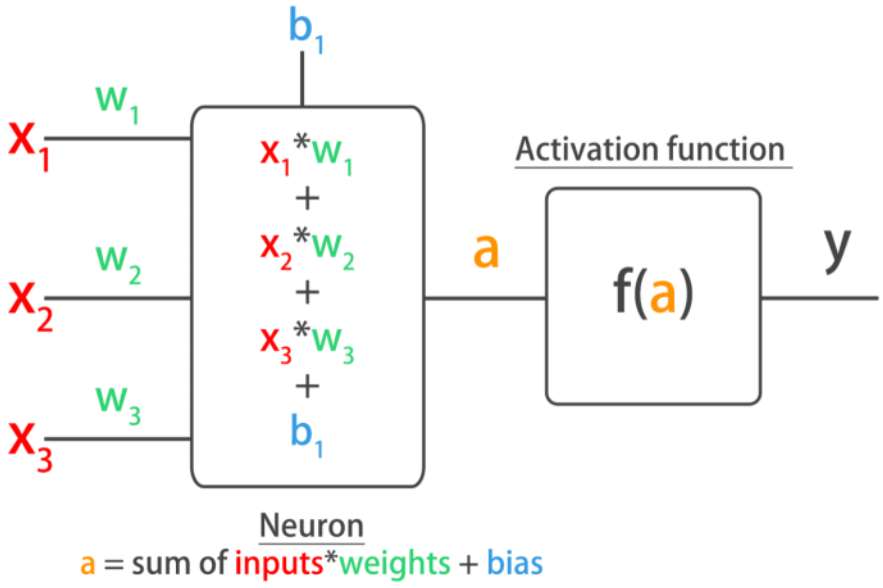

Neural networks like to be fed
with a BIG and VARIED
amount of data.

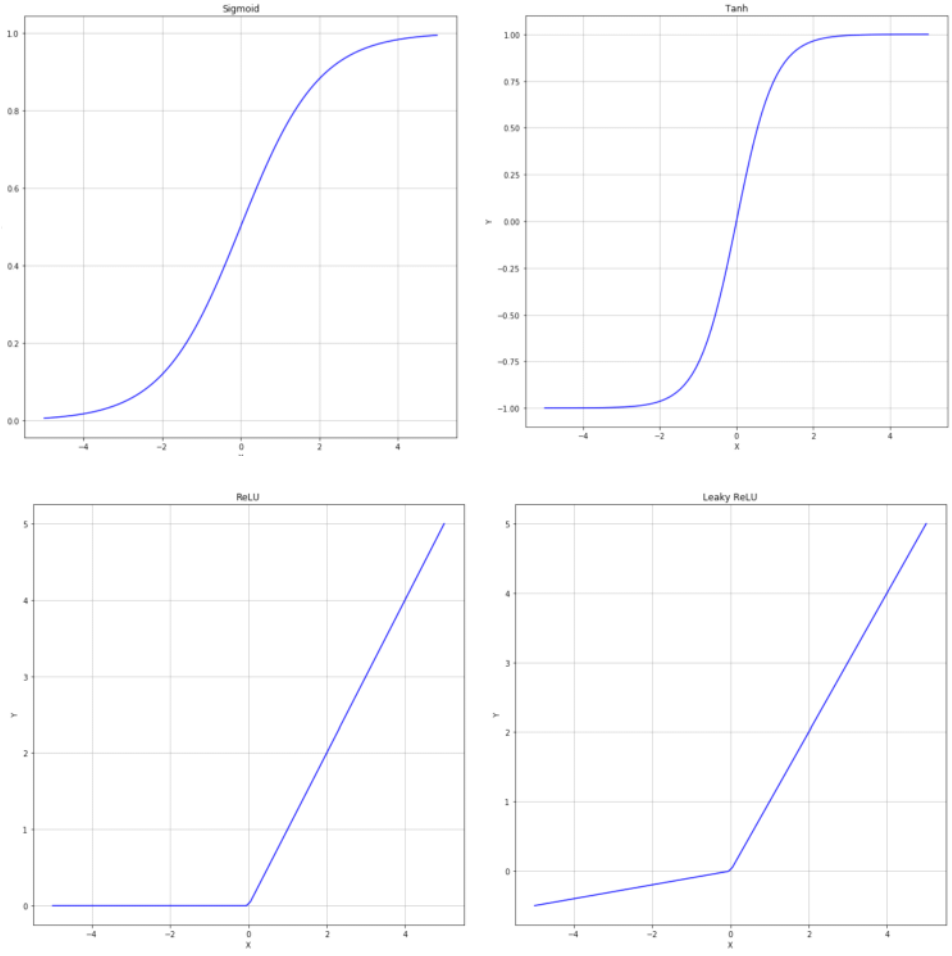

Accuracy vs Number of training samples
Training Accuracy
Test Accuracy
Accuracy
1.00
0.95
0.90
0.85
0.80
200
400
600
800
1000
1200
Training samples



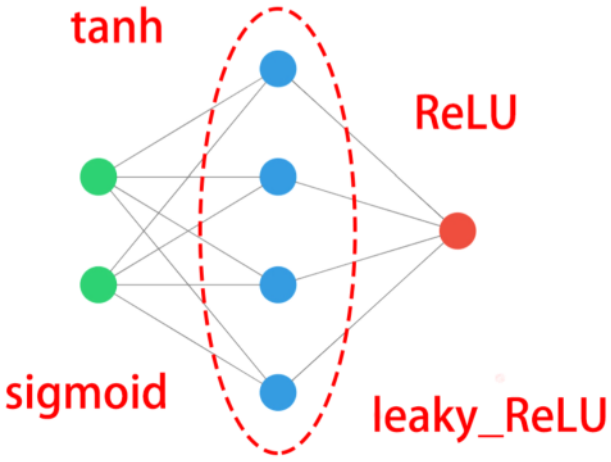

```
# Store initial model weights
init_weights = model.get_weights()

# Lists for storing accuracies
train_accs = []
tests_accs = []
```

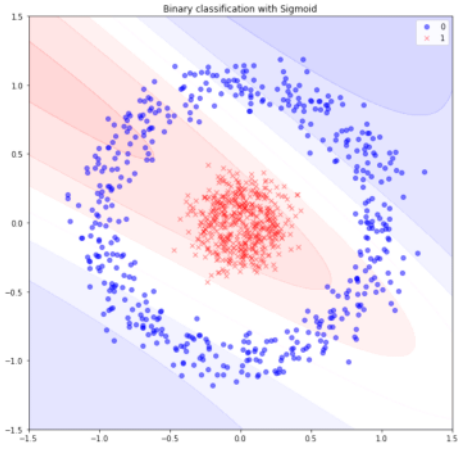
```
for train_size in train_sizes:
    # Split a fraction according to train_size
    X_train_frac, _, y_train_frac, _ =
    train_test_split(X_train, y_train, train_size=train_size)
    # Set model initial weights
    model.set_weights(initial_weights)
    # Fit model on the training set fraction
    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)
```

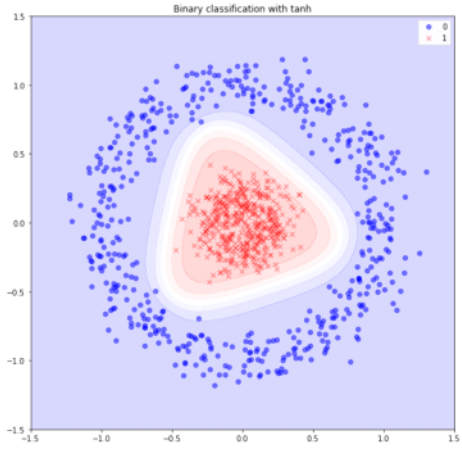

b1
w1
x1
x1*w1
+
x2*w2
+
x3*w3
+
b1
w2
x2
w3
x3
Activation function
a
f(a)
y
Neuron
a = sum of inputs*weights + bias

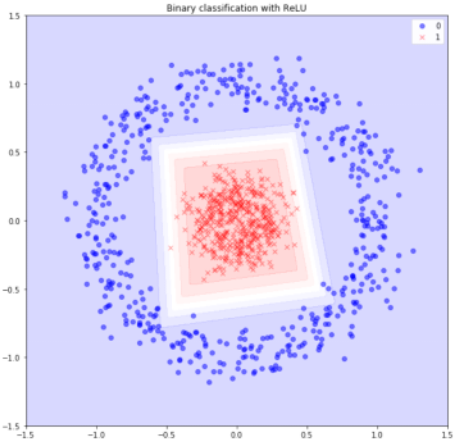

Sigmoid
Tanh
ReLU
Leaky ReLU

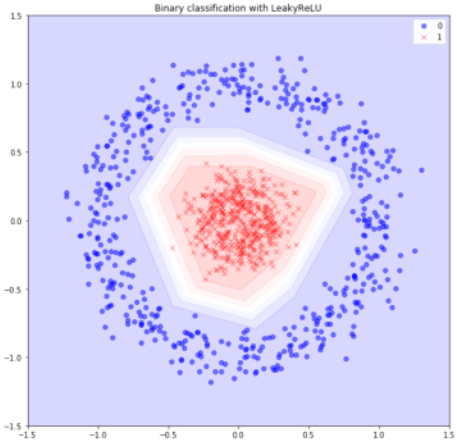
Effects of activation functions











Which activation function to use?

- No magic formula
- Different properties
- Depends on our problem
- Goal to achieve in a given layer
- ReLU are a good first choice
- Sigmoids not recommended for deep models
- Tune with experimentation

Comparing activation functions

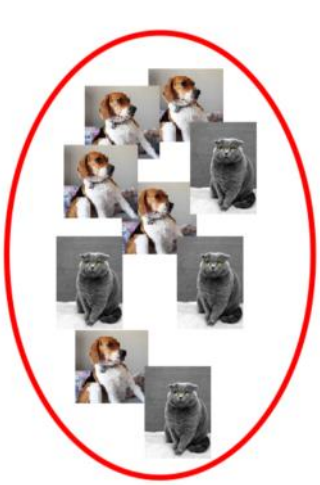
```
# Set a random seed
np.random.seed(1)

# 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
```

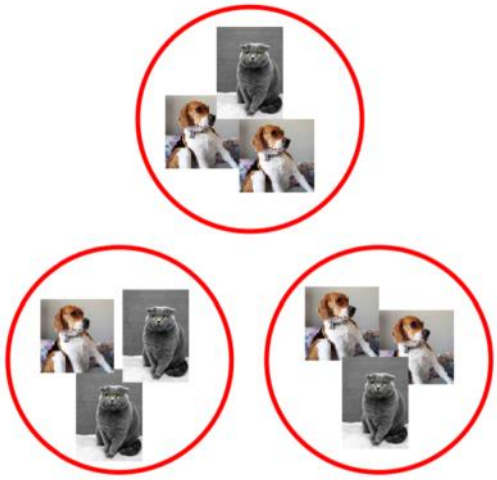
Comparing activation functions

```
# 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
```

Comparing activation functions

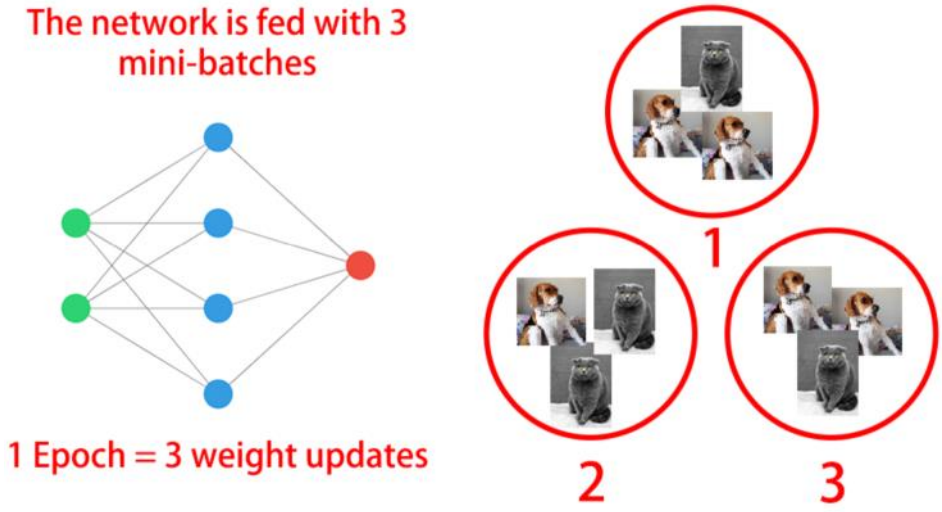
```
import pandas as pd

# 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')
```





Batch

Mini-batches



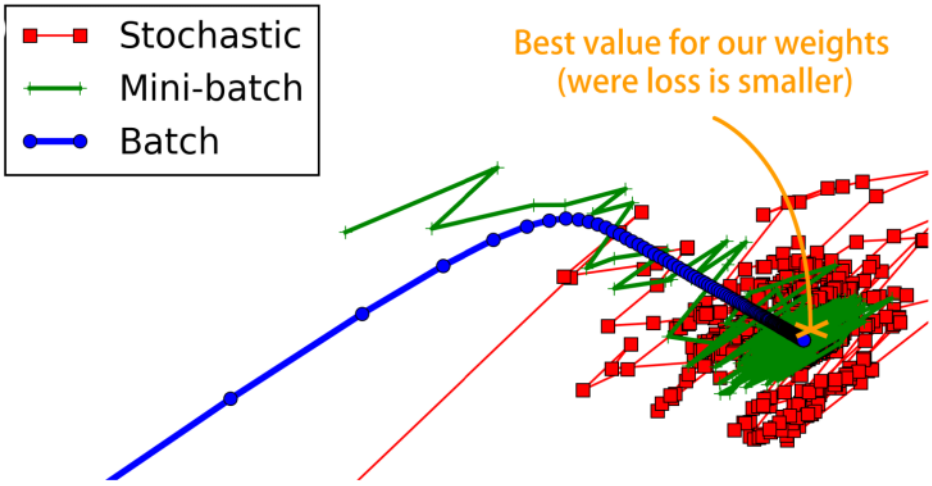
Mini-batches

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

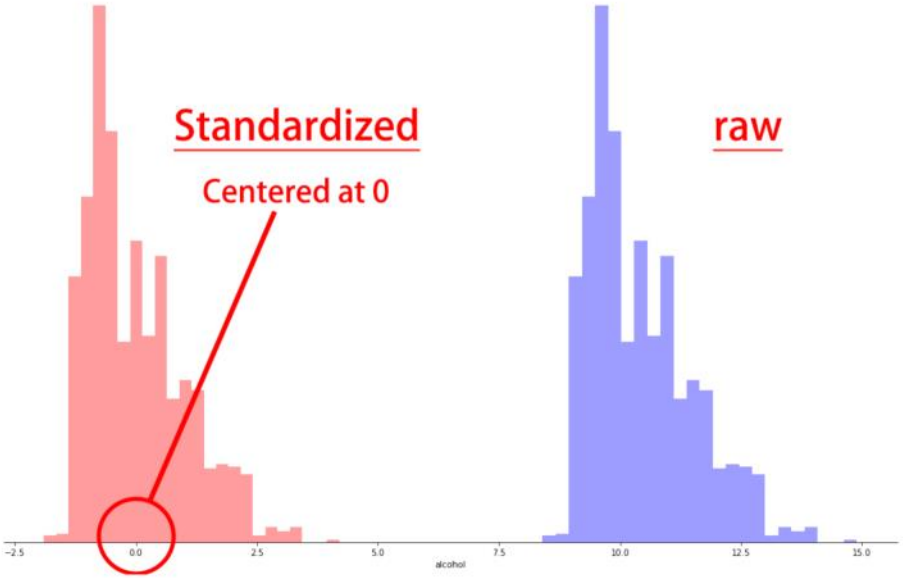
Disadvantages

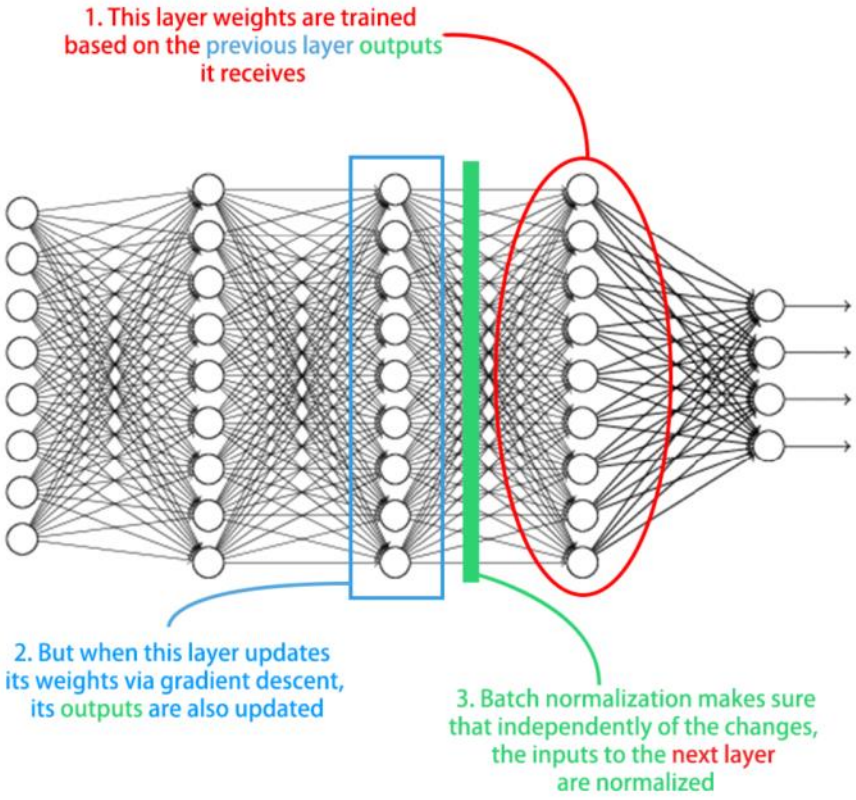
- More iterations need to be run
- Need to be adjusted, we need to find a good batch size



Standardization
(a normalization approach)

$$\frac{\text{data} - \text{mean}}{\text{standard deviation}}$$


Standardized
Centered at 0
raw
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
alcohol


1. This layer weights are trained based on the previous layer outputs it receives
2. But when this layer updates its weights via gradient descent, its outputs are also updated
3. Batch normalization makes sure that independently of the changes, the inputs to the next layer are normalized

Batch normalization advantages

- Improves gradient flow
- Allows higher learning rates
- Reduces dependence on weight initializations
- Acts as an unintended form of regularization
- Limits internal covariate shift

Batch normalization in Keras

```
# 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'))
```

Neural network hyperparameters

- Number of layers
- Number of neurons per layer
- Layer order
- Layer activations
- Batch sizes
- Learning rates
- Optimizers

Sklearn recap

```
# 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}
```

Turn a Keras model into a Sklearn estimator

```
# 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)
```

Cross-validation

```
# 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
- Don't use many epochs
- Use a smaller sample of your dataset
- Play with batch sizes, activations, optimizers and learning rates

Random search on Keras models

```
# 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'}
```

Tuning other hyperparameters

```
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...
```

Tuning other hyperparameters

```
# 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}
```

Advance Model Architecture

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>]
```

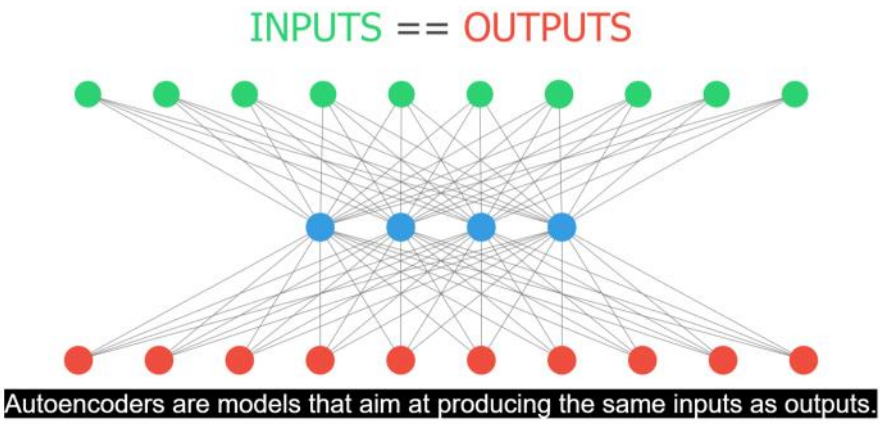
What are tensors?

```
# 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!



Autoencoder use cases

- Dimensionality reduction:
  - 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:
  - A poor reconstruction will result when the model is fed with unseen inputs.

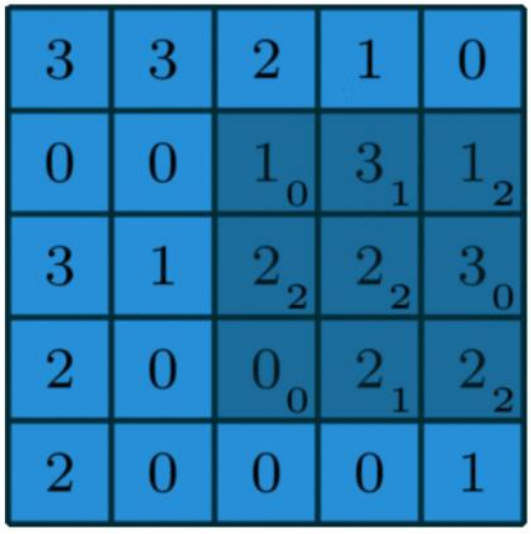
Building a simple autoencoder

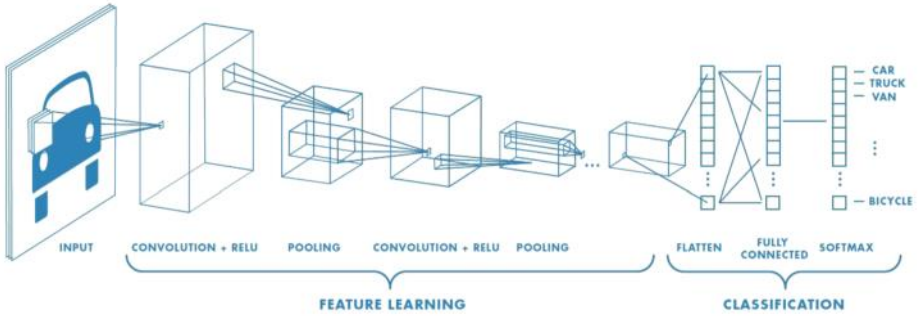
```
# Instantiate a sequential model
autoencoder = Sequential()

# 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')
```

Breaking it into an encoder

```
# 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],...)
```






```
# 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'))
```

Pre-processing images for ResNet50

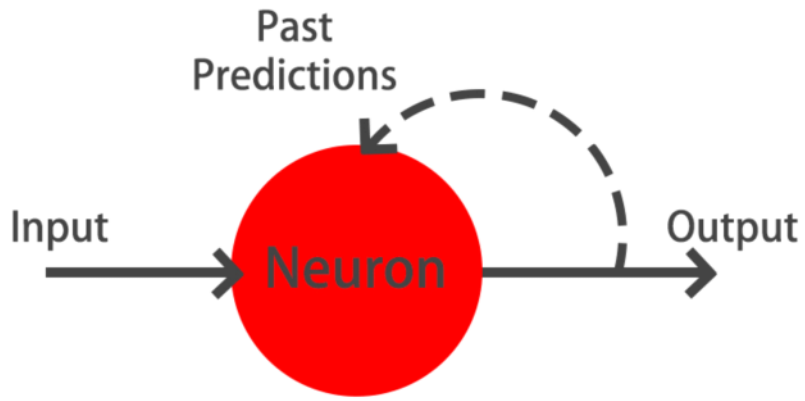
```
# Import image from keras preprocessing
from keras.preprocessing import image

# 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))

# 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)
```

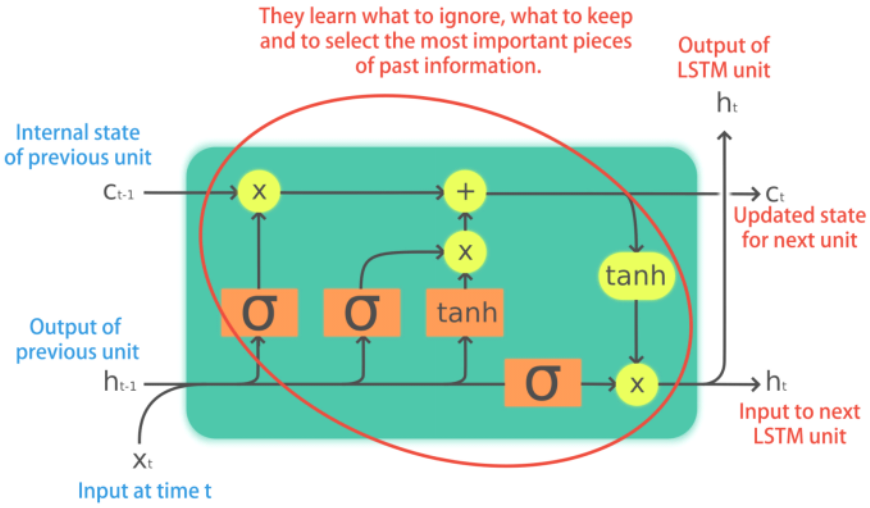
Using the ResNet50 model in Keras

```
# 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)]
```

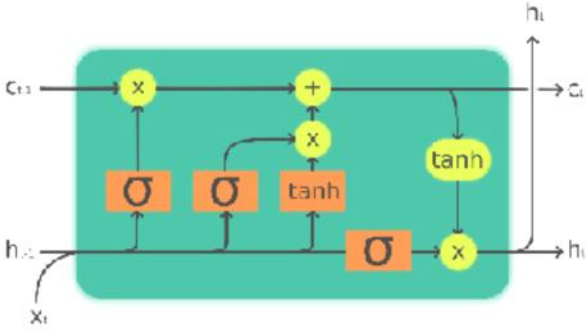
What are RNNs?





When to use LSTMs?

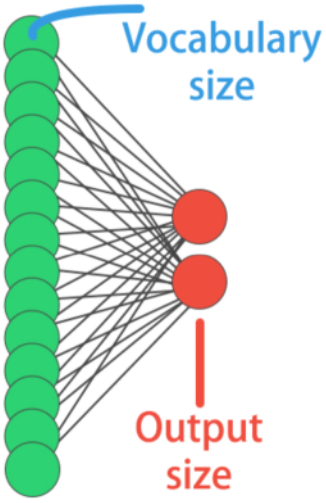
- Image captioning
- Speech to text
- Text translation
- Document summarization
- Text generation
- Musical composition
- ...

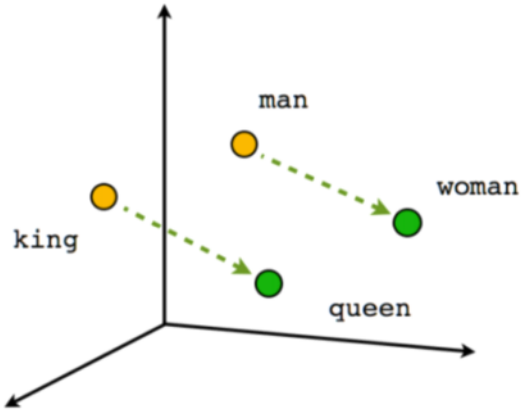


+ Text

this is a sentence

42 11 23 1





```
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']
```

```
# 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'))
```