Import Data

It's simple: given an image, classify it as a digit.

Each image in the MNIST dataset is 28x28 and contains a centered, grayscale digit. We'll flatten e
which we'll use as input to our neural network. Our output will be one of 10 possible classes: one

```
1 from keras.datasets import mnist
```

```
1 # the data, split between train and test sets
2 (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
1 x_train.shape
```

```
(60000, 28, 28)
```

```
1 x_train[0].shape
```

```
(28, 28)
```

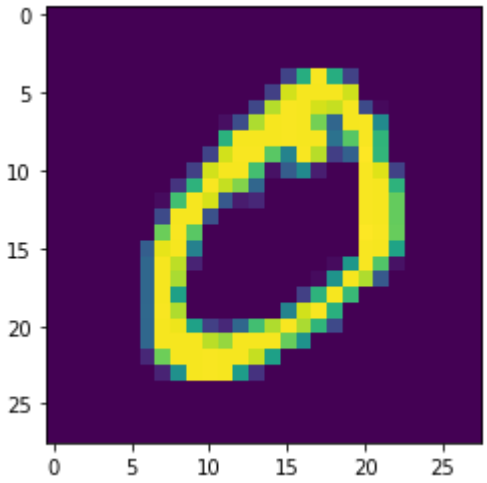
```
1 x_train[0]
```

```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
        253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
        205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
         90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
        190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
        253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
        241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
        148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
```

```
 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0]], dtype=uint8)
```

```
1 from matplotlib import pyplot as plt
2 plt.imshow(x_train[1])
```

```
<matplotlib.image.AxesImage at 0x7fc8ed20bb70>
```




```
1 y_train.shape
```

```
(60000,)
```

```
1 y_train[1]
```

```
0
```

```
1 x_train = x_train.reshape(-1, 784)
2 x_test = x_test.reshape(-1, 784)
```

```
1 x_train.shape
```

```
(60000, 784)
```

As mentioned earlier, we need to flatten each image before we can pass it into our neural network

We'll also normalize the pixel values from [0, 255] to [0, 1] to make our network easier to train (usi

```
# Normalize images
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

```
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
60000 train samples
10000 test samples
```

```
# convert class vectors to binary class matrices
from keras.utils import to_categorical
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
y_train[1]
```

```
array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

Create NN model

Every Keras model is either built using the Sequential class, which represents a linear stack of lay more customizeable.

We'll be using the simpler Sequential model, since our network is indeed a linear stack of layers.

The Sequential constructor takes an array of Keras Layers. Since we're just building a standard fe layer, which is your regular fully-connected (dense) network layer.

The last thing we always need to do is tell Keras what our network's input will look like. We can do layer in the Sequential model:

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

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
```

```
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

Once the input shape is specified, Keras will automatically infer the shapes of inputs for later laye

We've finished defining our model! Here's where we're at:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

Compile NN model

Before we can begin training, we need to configure the training process. We decide 3 key factors c

- The **optimizer**. We'll stick with a pretty good default: the Adam gradient-based optimizer. Ke into as well
- The **loss function.** Since we're using a Softmax output layer, we'll use the Cross-Entropy loss binary_crossentropy (2 classes) and categorical_crossentropy (>2 classes), so we'll use the
- A list of metrics. Since this is a classification problem, we'll just have Keras report on the ac

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

Train NN model

Training a model in Keras literally consists only of calling fit() and specifying some parameters. Th only manually supply a few:

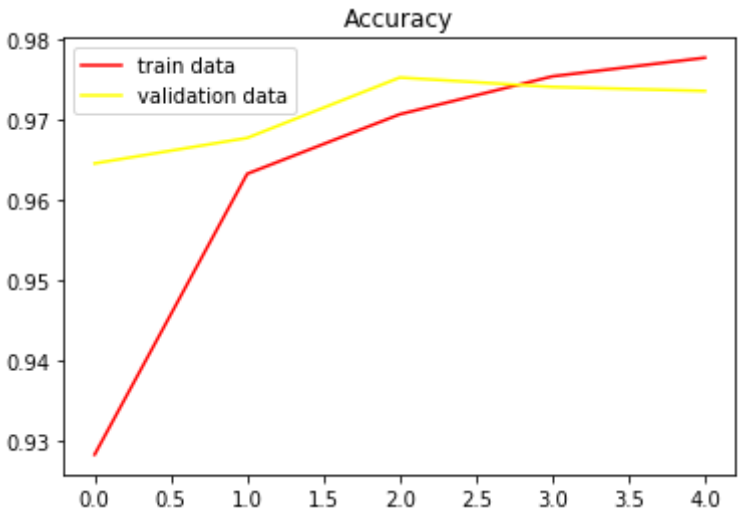
- The training data (images and labels), commonly known as X and Y, respectively.
- The number of epochs (iterations over the entire dataset) to train for.
- The batch size (number of samples per gradient update) to use when training.

We can also use the testing dataset for validation during training. Keras will evaluate the model or and report the loss and any metrics we asked for. This allows us to monitor our model's progress to identify overfitting and even support early stopping.

```
history = model.fit(x_train, y_train,
                    batch_size=16,
                    epochs=5,
                    verbose=1,
                    validation_split=0.2)
```

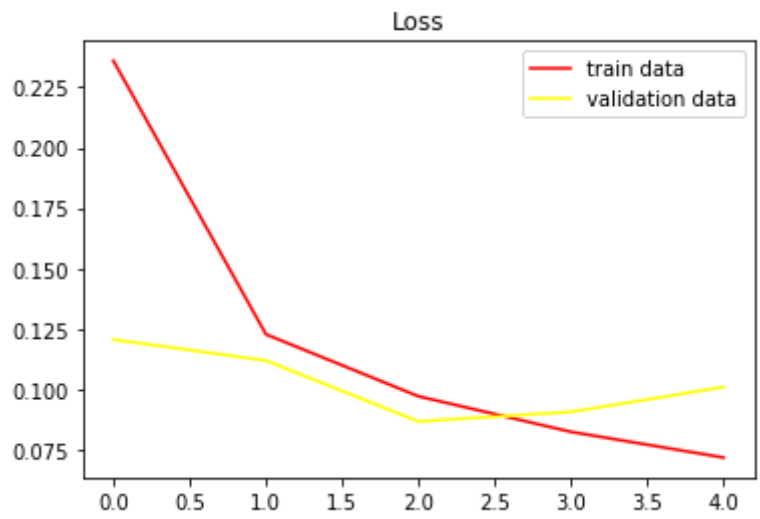
```
Train on 48000 samples, validate on 12000 samples
Epoch 1/5
48000/48000 [==============================] - 12s 255us/step - loss: 0.2359 - acc: 0
Epoch 2/5
48000/48000 [==============================] - 12s 242us/step - loss: 0.1229 - acc: 0
Epoch 3/5
48000/48000 [==============================] - 12s 241us/step - loss: 0.0972 - acc: 0
Epoch 4/5
48000/48000 [==============================] - 12s 246us/step - loss: 0.0826 - acc: 0
Epoch 5/5
48000/48000 [==============================] - 12s 241us/step - loss: 0.0719 - acc: 0
```

```
from matplotlib import pyplot as plt
epochs = history.epoch
acc = history.history['acc']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, color='red', label='train data')
plt.plot(epochs, val_acc, color='yellow', label='validation data')
plt.title('Accuracy')
plt.legend()
plt.show()
```




```
epochs = history.epoch
loss = history.history['loss']
```

```
val_loss = history.history['val_loss']
plt.plot(epochs, loss, color='red', label='train data')
plt.plot(epochs, val_loss, color='yellow', label='validation data')
plt.title('Loss')
plt.legend()
plt.show()
```



Evaluate NN on unseen data

```
model.evaluate(x_test, y_test, verbose=1)
```

```
10000/10000 [==============================] - 0s 37us/step
[0.08824941261925269, 0.9764]
```

Using the Model

Now that we have a working, trained model, let's put it to use.

The first thing we'll do is save it to disk so we can load it back up anytime:

```
model.save_weights('model.h5')
```

```
from keras.models import load_model
model.save('full_model.h5')
```

```
model1 = load_model('./full_model.h5')
```

```
predictions = model1.predict(x_test[:5])
predictions
```

```
array([[9.81469788e-13, 3.95107280e-11, 1.21047501e-08, 1.38923720e-10,
        6.98170634e-13, 1.20128853e-12, 2.25888980e-17, 9.99997854e-01,
        6.04053266e-11, 2.13108319e-06],
       [3.48768386e-21, 2.45340831e-11, 1.00000000e+00, 3.57293659e-15,
        2.66627444e-23, 2.14111148e-24, 2.56077084e-21, 2.75692338e-13,
        4.96200477e-16, 8.83747811e-21],
       [5.94153251e-15, 1.00000000e+00, 2.50776934e-11, 2.53017106e-16,
        2.31111606e-11, 6.20996647e-17, 1.65197056e-14, 1.43673337e-10,
        8.93712274e-12, 9.61253755e-13],
       [9.99976516e-01, 4.43876845e-11, 2.69404965e-09, 1.51615474e-11,
        1.81469915e-08, 4.58895595e-11, 2.33772662e-05, 9.67554301e-11,
        8.93120067e-09, 1.12413254e-07],
       [1.56302471e-09, 9.54811230e-10, 1.23672503e-10, 2.19503105e-12,
        9.99951243e-01, 4.64750391e-11, 2.52363602e-10, 1.48775872e-08,
        5.72255487e-09, 4.87727302e-05]], dtype=float32)
```

```
1 import numpy as np
2 np.argmax(predictions, axis=1)
```

```
array([7, 2, 1, 0, 4])
```

```
1 np.argmax(y_test[:5], axis=1)
```

```
array([7, 2, 1, 0, 4])
```

Furhter improve

What we've covered so far was but a brief introduction - there's much more we can do to experime

Tuning Hyperparameters A good hyperparameters to start with is the learning rate for the Adam o
decrease it?

```
1 from keras.optimizers import Adam
2
3 model.compile(
4   optimizer=Adam(lr=0.005),
5   loss='categorical_crossentropy',
6   metrics=['accuracy'],
7 )
```

What about the **batch size** and number of epochs?

```
1 history = model.fit(x_train, y_train,
2                     batch_size=128,
3                     epochs=20,
4                     verbose=1,
5                     validation_split=0.2)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 2s 45us/step - loss: 0.1274 - acc: 0.9
Epoch 2/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.1127 - acc: 0.9
Epoch 3/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.1073 - acc: 0.9
Epoch 4/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.1086 - acc: 0.9
Epoch 5/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.1020 - acc: 0.9
Epoch 6/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0913 - acc: 0.9
Epoch 7/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0899 - acc: 0.9
Epoch 8/20
48000/48000 [==============================] - 2s 35us/step - loss: 0.1003 - acc: 0.9
Epoch 9/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.0991 - acc: 0.9
Epoch 10/20
48000/48000 [==============================] - 2s 35us/step - loss: 0.0945 - acc: 0.9
Epoch 11/20
48000/48000 [==============================] - 2s 36us/step - loss: 0.1105 - acc: 0.9
Epoch 12/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.0957 - acc: 0.9
Epoch 13/20
48000/48000 [==============================] - 2s 35us/step - loss: 0.0776 - acc: 0.9
Epoch 14/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.0873 - acc: 0.9
Epoch 15/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0831 - acc: 0.9
Epoch 16/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0800 - acc: 0.9
Epoch 17/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.0814 - acc: 0.9
Epoch 18/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0900 - acc: 0.9
Epoch 19/20
48000/48000 [==============================] - 2s 33us/step - loss: 0.0979 - acc: 0.9
Epoch 20/20
48000/48000 [==============================] - 2s 34us/step - loss: 0.0877 - acc: 0.9
```

Network Depth

What happens if we remove or add more fully-connected layers?

How does that affect training and/or the model's final performance?

```
model = Sequential([
  Dense(64, activation='relu', input_shape=(784,)),
  Dense(64, activation='relu'),
  Dense(64, activation='relu'),
  Dense(64, activation='relu'),
  Dense(10, activation='softmax'),
])
```

Activations

What if we use an activation other than ReLU, e.g. sigmoid?

```
model = Sequential([
  Dense(64, activation='sigmoid', input_shape=(784,)),
  Dense(64, activation='sigmoid'),
  Dense(10, activation='softmax'),
])
```

Dropout

What if we tried adding Dropout layers, which are known to prevent overfitting?

```
from keras.layers import Dense, Dropout

model = Sequential([
  Dense(64, activation='relu', input_shape=(784,)),
  Dropout(0.5),
  Dense(64, activation='relu'),
  Dropout(0.5),
  Dense(10, activation='softmax'),
])
```