Neural Networks Tutorial

November 18, 2019

1 A Classic Neural Networks Tutorial

For this tutorial, we'll be using Keras. Keras is like a higher-level abstraction of Tensorflow – a popular ML library – and will allow us to do some pretty cool stuff without knowing a lot of linear algebra/calculus. :)

[1]:
```
# import fashion dataaset and model
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
```

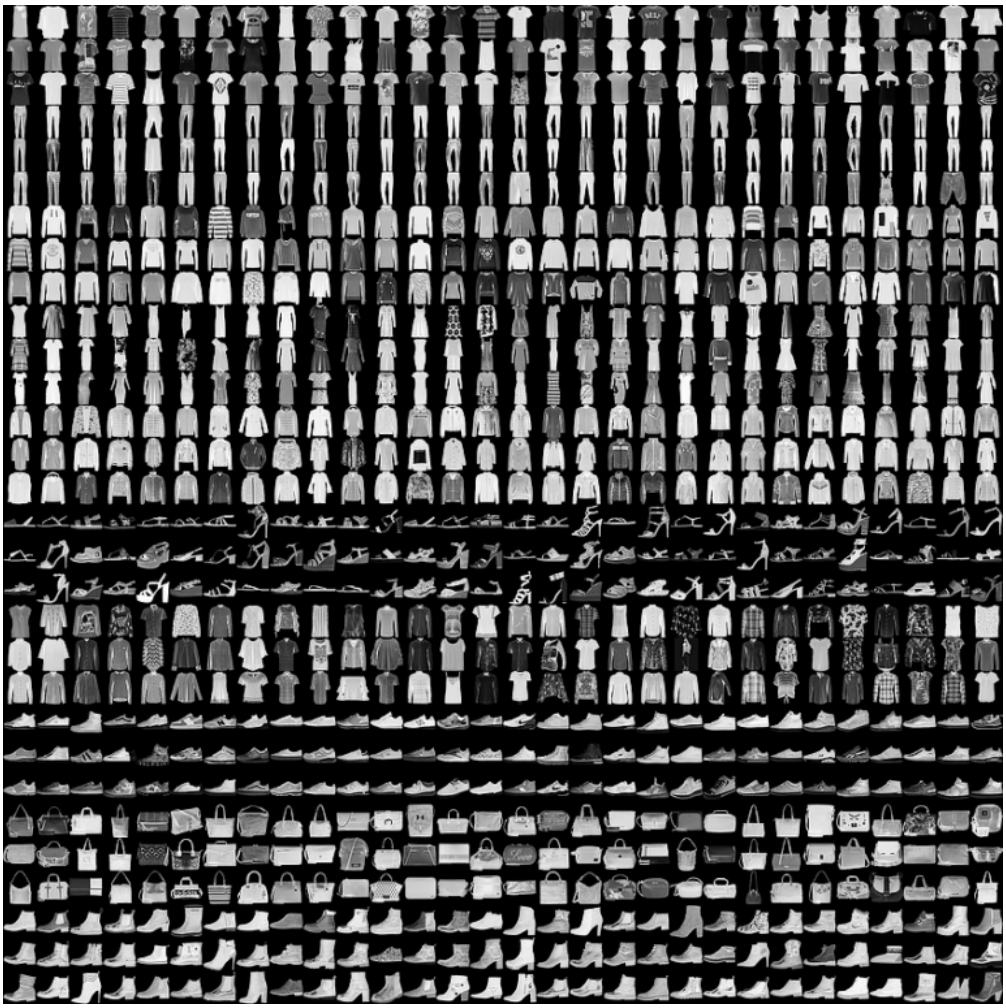
1.1 The dataset - Fashion MNIST

We'll be using labeled images of different types of clothing from MNIST (Modified National Institute of Standards and Technology database). They have a large database of handwritten digit images that I'm sure you've seen if you've gone through neural net examples in the past, but we'll be analyzing Fashion-MNIST for the following reasons:

1. The MNIST digit dataset is too easy, and even classic ML algorithms can achieve 97% accuracy on this dataset.

2. MNIST is overused.

3. MNIST is not very representative of stuff you might actually do; i.e. bad ideas might work well on the digit dataset, but not on most other datasets.



title

1.2 The Goal

We want to turn each of these 28x28 pixel, grayscale images into one of 10 classifications:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle Boot

The dataset is already presplit into 60,000 training images and 10,000 testing images. These

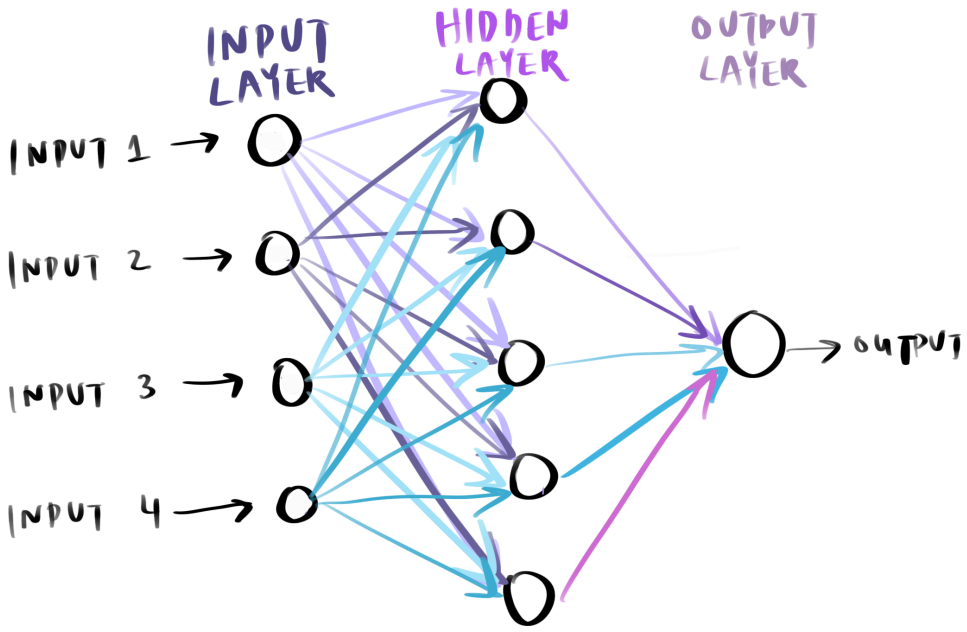
images are all labeled with their corresponding category (a number between 0 and 9). Let's get started by splitting the data into training, testing, and validation data!

1.3 But before we start... what even *is* a neural net?

title

For many people, this is what a neural net is: a black box that you feed an input (like an image, sequence of numbers, audio, etc), and it performs some kind of magic to correctly classify it or make a prediction.

From a high level, that's actually very accurate. A simple neural network looks something like this:



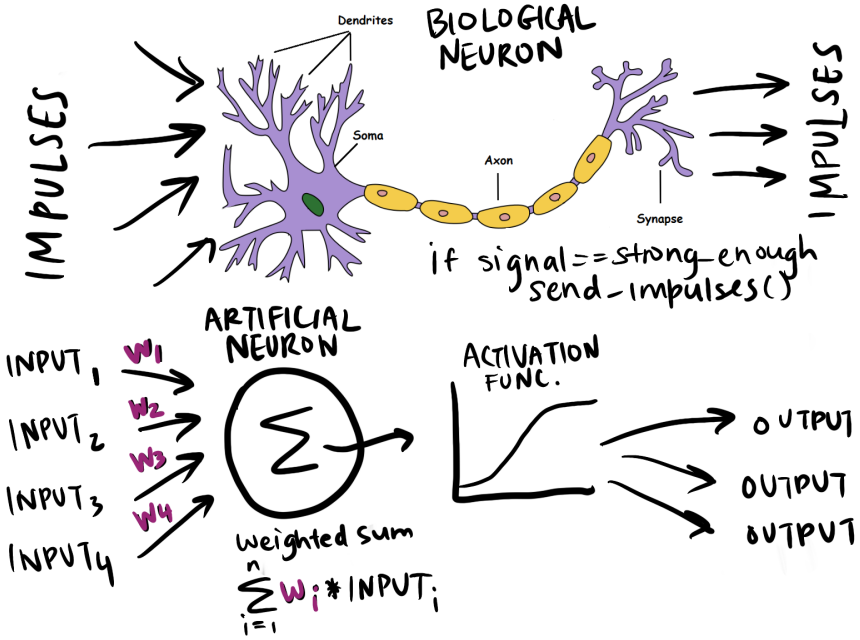
title

There are three parts:

1. **Input Layer**: This is the actual data that you pass in. Usually this is a vector of numbers. For us, it would be some kind of numerical data that represents each pixel in the 28x28 images of clothing.
2. **Hidden Layer**: This is an intermediate layer between the input and output layers. It is also where all the computation is done. There can be multiple hidden layers.
3. **Output Layer**: The actual output of a neural network. For a classification problem, we want it to output a vector where one of the values is 1 (aka the neuron is lit up).

1.3.1 Single Neuron

To better understand what kind of computation goes on inside hidden layers, let's take a look at a single neuron and contrast it to a biological neuron.



title

In biology, a neuron receives some kind of electrical signal from other neurons. If the signal is strong enough, it might be passed down the axon, and the neuron might "fire", meaning that it signals other neurons.

Similarly, an artificial neuron gets a bunch of inputs (aka signals) from other neurons, calculates a weighted sum, adds a bias value, and passes that number into an activation function to determine whether how much the artificial neuron "fires."

Each connection between two neurons has a different weight. This is similar to biological neurons; paths between neurons that are fired more often together are "stronger." In an artificial neural network, the higher the weight, the stronger the connection between the neurons. The bias value is similar to an intercept in a regression. It allows you to shift the activation function left or right.

Essentially, a neural network consists of many connected neurons. Thus, it is comprised of a series of these calculations where outputs are passed and altered from neuron to neuron until you reach the output layer.

1.3.2 Activation Function

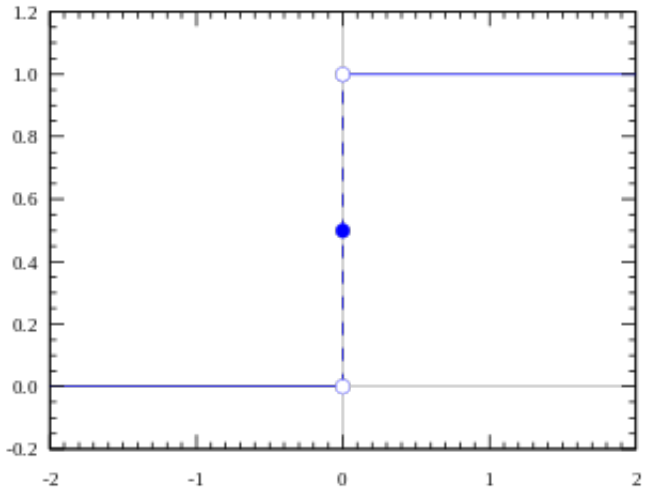
An activation function is what determines whether the neuron "fires" or not. But what does that mean? Like we covered, a neuron does this calculation:

$$Y = \sum (weight * input) + bias$$

title

Y is a weighted sum plus a bias term. We pass Y into an activation function to determine whether it means the neuron should "fire" or not. But Y can be any value between negative and positive infinity. How does the neuron know what value means that it should fire? This is where an activation function comes in. Let's try several possibilities.

Step Function The simplest example is a step function. If the input value is above a threshold, output 1 ("fired"). Else, output 0. At first, this seems great. We can transform values from (-inf, inf) into values {0, 1}.



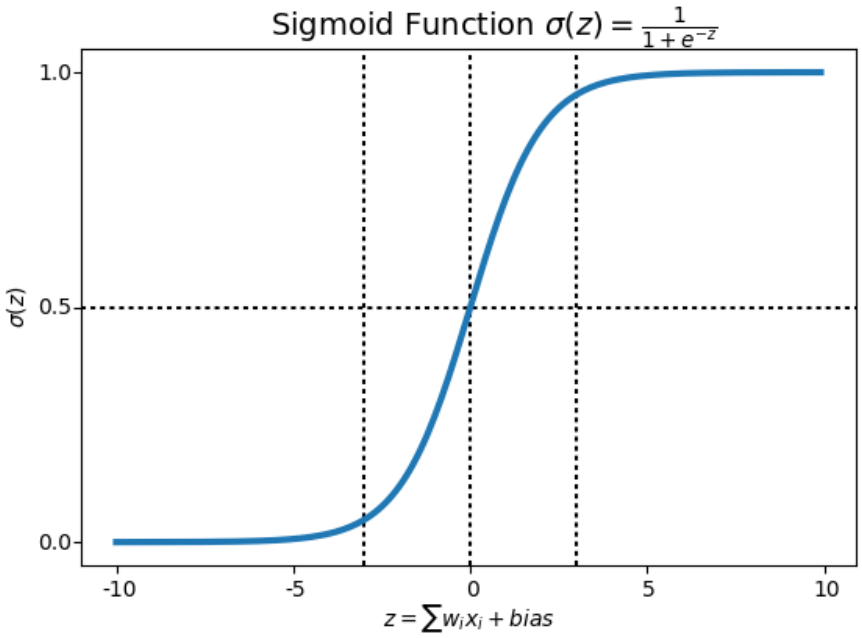
title

However, this runs into the issue where you may have many neurons with the value of 1 at the same time. For example, if we get a vector with five 1s for our output layer, where each 1 refers to a type of clothing, how do we know which type it is? Thus, we need a function where activations aren't binary (ex: 20% activated, 66% activated).

Linear Function Since the range of a linear function is between (-inf, inf), this doesn't confine our output values. In addition, the goal of an activation function is to introduce some non-linearity into our neural network.

Sigmoid Function The sigmoid function is commonly used when you want to classify/output probabilities. It's range is between 0 and 1, and it commonly forces activations of the neuron to the extremes of the function (aka close to 0 or 1). However, this is also one of the problems of the sigmoid function: notice that toward the ends of the function, Y changes less as X changes. Thus,

if your inputs get larger and larger, your output values may not change very much from layer to layer, which leads to slow learning. This is called **vanishing gradient.**

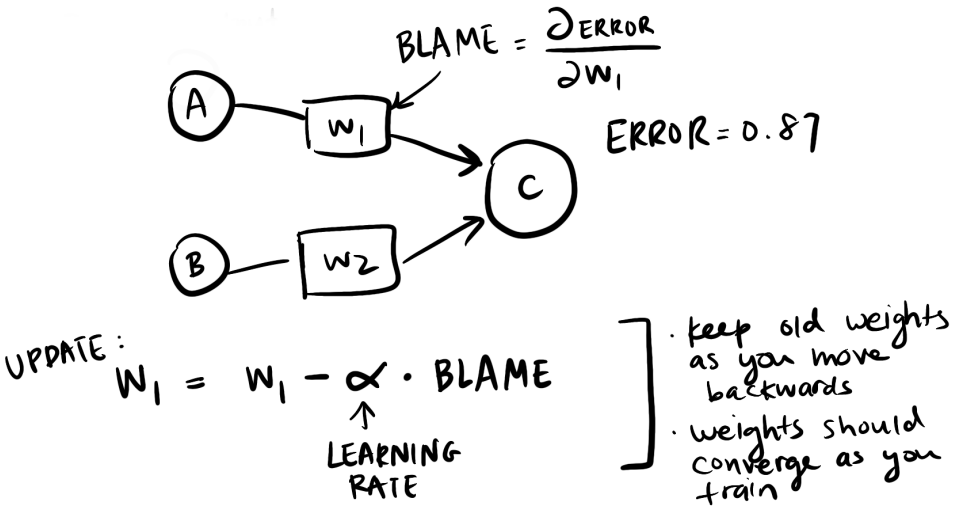


title

Because each activation function has pros/cons and best use cases, there are a lot that are used in practice (ReLu, tanh). Here's more reading if you're interested.

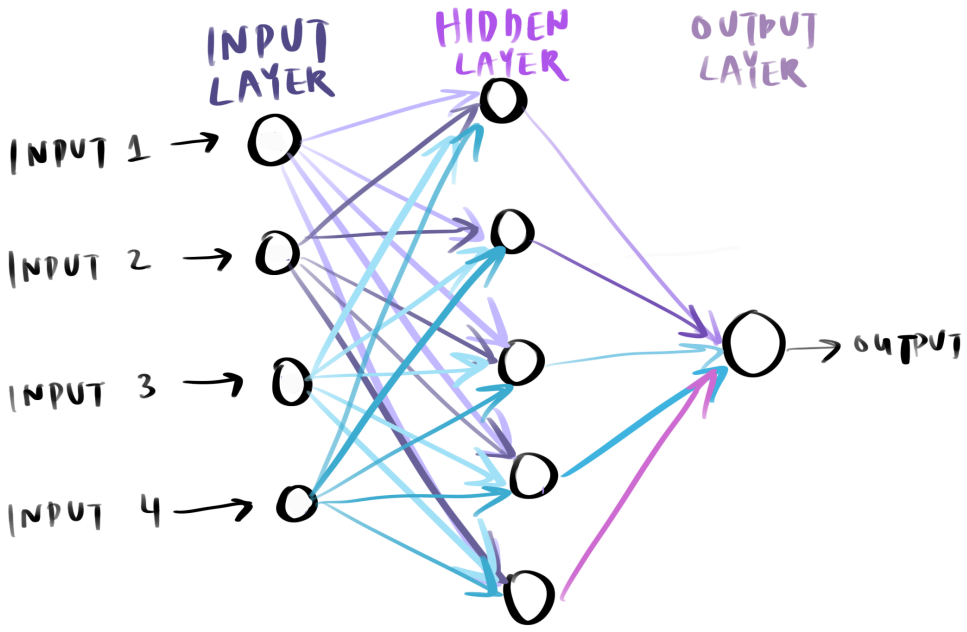
1.3.3 Backpropagation

Backpropagation is basically gradient descent, but for neural networks. It is a way that we alter the weights, or the values associated with every connection between two neurons based on a cost function, like the squared error between desired and predicted output. It's called backpropagation because you move backward through the neural network as you update the weights.



title

1.3.4 The almighty perceptron



title

This is a single layer perceptron. Notice the structure is a series of connected, feed-forward neurons (directed forward toward the output layer). We feed in data into the input layer, and the hidden layer neurons fire accordingly and communicate with the output layer.

If you add more hidden layers, it becomes a **multilayer perceptron**, or MLP. Let's try using perceptrons to classify images.

Flattening the data Notice the input layer is flat, whereas our data is 2D. How can we solve this?

[2]:
```
# Load in data
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

[3]:
```
# 60000 rows, where each row is a 2D picture (28 by 28)
x_train.shape
```

[3]:
```
(60000, 28, 28)
```

[4]:
```
# Flatten data from 2D to 1D:
# reshape(__, -1) means that we're telling numpy that there are
# two dimensions, and to "infer" the second dimension

# we pass in x_train.shape[0], which means there will be 60000 rows
# thus, numpy will "infer" that the length of each row will be 28x28 = 784
x_train = x_train.reshape(x_train.shape[0], -1) / 255.0 # rgb is (0, 255)
x_test = x_test.reshape(x_test.shape[0], -1) / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

[5]:
```
# note 28 * 28 = 784!
# we've turned each picture into a 1D array of pixels
x_train.shape
```

[5]:
```
(60000, 784)
```

[6]:
```
x_train[0][100:110]
```

[6]:
```
array([0.28627451, 0.        , 0.        , 0.00392157, 0.01568627,
       0.        , 0.        , 0.        , 0.        , 0.00392157])
```

[7]:
```
# 0s for 9 categories that are not true for that picture, 1 for the right␣
 ↪category
y_train[0]
```

[7]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.], dtype=float32)
```

Creating the Model We'll be using the `Sequential` model with `Dense` layers. This is used to create a perceptron-based neural network model. If you're curious, a `Dense` layer is just a regular NN layer for a multilayer perceptron that does: `output = activation(weighted sum of inputs + bias)`.

This means that it takes the dot product of your input vector and a weight vector for each neuron, and adds a bias. The dot product is a scalar value, meaning that it's not a vector (no direction). Finally, after taking the dot product, the Sequential Neural Network feeds this into an activation function.

Single Layer Perceptron

[8]:
```
# Literally the simplest neural net model that Keras has
# Sequential is a linear stack of layers
model = Sequential()
```

[9]:
```
# add layers to model

# this is the hidden layer, which has 10 neurons, and 784 input values per neuron
model.add(Dense(10, input_dim=784, activation='sigmoid'))

# output layer
model.add(Dense(10, activation='sigmoid'))
model.compile(loss='categorical_crossentropy', optimizer='sgd',␣
 ↪metrics=['accuracy'])
```

[10]:
```
# Train
model.fit(x_train, y_train, epochs=10, validation_split=0.1)
```

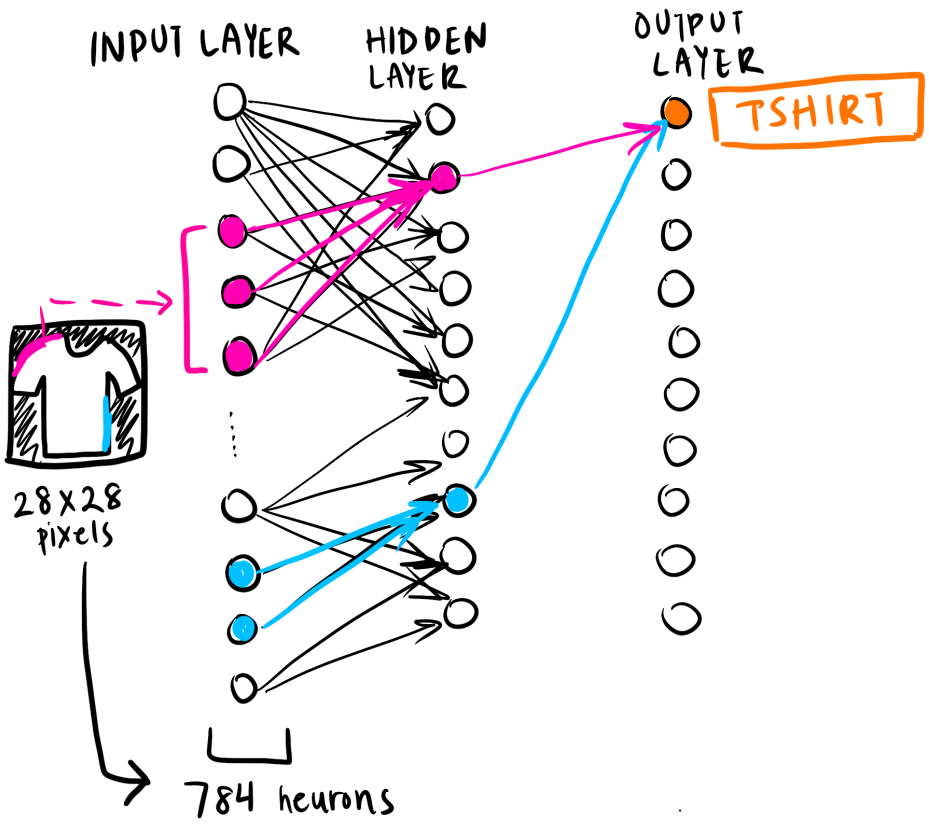
```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 2s 46us/sample - loss: 2.1314 -
accuracy: 0.2876 - val_loss: 1.9468 - val_accuracy: 0.4535
Epoch 2/10
54000/54000 [==============================] - 2s 36us/sample - loss: 1.7773 -
accuracy: 0.5831 - val_loss: 1.6096 - val_accuracy: 0.6443
Epoch 3/10
54000/54000 [==============================] - 2s 38us/sample - loss: 1.4785 -
accuracy: 0.6521 - val_loss: 1.3500 - val_accuracy: 0.6718
Epoch 4/10
54000/54000 [==============================] - 2s 39us/sample - loss: 1.2589 -
accuracy: 0.6728 - val_loss: 1.1626 - val_accuracy: 0.6873
Epoch 5/10
54000/54000 [==============================] - 2s 46us/sample - loss: 1.1002 -
accuracy: 0.6911 - val_loss: 1.0259 - val_accuracy: 0.7105
Epoch 6/10
54000/54000 [==============================] - 2s 42us/sample - loss: 0.9830 -
accuracy: 0.7114 - val_loss: 0.9254 - val_accuracy: 0.7397
Epoch 7/10
54000/54000 [==============================] - 3s 49us/sample - loss: 0.8939 -
accuracy: 0.7334 - val_loss: 0.8467 - val_accuracy: 0.7493
Epoch 8/10
54000/54000 [==============================] - 2s 36us/sample - loss: 0.8239 -
accuracy: 0.7518 - val_loss: 0.7854 - val_accuracy: 0.7645
Epoch 9/10
54000/54000 [==============================] - 2s 36us/sample - loss: 0.7681 -
accuracy: 0.7664 - val_loss: 0.7376 - val_accuracy: 0.7757
Epoch 10/10
54000/54000 [==============================] - 2s 36us/sample - loss: 0.7236 -
accuracy: 0.7776 - val_loss: 0.6980 - val_accuracy: 0.7892
```

[10]:
```
<tensorflow.python.keras.callbacks.History at 0x149c53f50>
```

[11]:
```
%%capture
_, test_acc1 = model.evaluate(x_test, y_test)
```

[12]:
```
print(test_acc1)
```

```
0.7743
```



title

The Intuition Behind the Model Suppose we have a picture of a tshirt and a fully trained single layer perceptron neural network model. We feed the flattened 784 numbers of the image into the input layer.

The goal is to have groups of neurons be able to "recognize edges" by firing when there is an edge in a certain area. For example, the group of pink neurons "fire" when the pink edge is present in the image. The group of blue neurons fire when the blue edge is present in the image. Then, in the output layer, a group of certain edges firing together should allow the neural network to determine what kind of clothing it is.

Single Layer Perceptron: Wider Hidden Layer

[13]:
```
# Widen network and use better activation
model2 = Sequential()
```

```
# 50 neurons in hidden layer
model2.add(Dense(50, input_dim=784, activation='relu')) # relu is good for␣
 ↪hidden layers

# still 10 neurons in output layer
model2.add(Dense(10, activation='softmax')) # softmax is good for output layers
model2.compile(loss='categorical_crossentropy', optimizer='sgd',␣
 ↪metrics=['accuracy'])
model2.fit(x_train, y_train, epochs=10, validation_split=0.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 3s 53us/sample - loss: 0.8380 -
accuracy: 0.7301 - val_loss: 0.6021 - val_accuracy: 0.7845
Epoch 2/10
54000/54000 [==============================] - 2s 42us/sample - loss: 0.5507 -
accuracy: 0.8139 - val_loss: 0.5204 - val_accuracy: 0.8177
Epoch 3/10
54000/54000 [==============================] - 2s 43us/sample - loss: 0.4988 -
accuracy: 0.8288 - val_loss: 0.4816 - val_accuracy: 0.8307
Epoch 4/10
54000/54000 [==============================] - 2s 46us/sample - loss: 0.4731 -
accuracy: 0.8371 - val_loss: 0.4625 - val_accuracy: 0.8368
Epoch 5/10
54000/54000 [==============================] - 2s 39us/sample - loss: 0.4557 -
accuracy: 0.8424 - val_loss: 0.4493 - val_accuracy: 0.8410
Epoch 6/10
54000/54000 [==============================] - 3s 59us/sample - loss: 0.4417 -
accuracy: 0.8463 - val_loss: 0.4437 - val_accuracy: 0.8472
Epoch 7/10
54000/54000 [==============================] - 2s 43us/sample - loss: 0.4317 -
accuracy: 0.8503 - val_loss: 0.4309 - val_accuracy: 0.8508
Epoch 8/10
54000/54000 [==============================] - 2s 39us/sample - loss: 0.4217 -
accuracy: 0.8548 - val_loss: 0.4324 - val_accuracy: 0.8505
Epoch 9/10
54000/54000 [==============================] - 2s 39us/sample - loss: 0.4138 -
accuracy: 0.8558 - val_loss: 0.4292 - val_accuracy: 0.8512
Epoch 10/10
54000/54000 [==============================] - 2s 39us/sample - loss: 0.4061 -
accuracy: 0.8589 - val_loss: 0.4445 - val_accuracy: 0.8447
```

[13]: `<tensorflow.python.keras.callbacks.History at 0x12c179750>`

[14]:
```
%%capture
_, test_acc2 = model2.evaluate(x_test, y_test)
```

[15]:
```
print(test_acc2)
```

0.8325

Multilayer Perceptron

[16]:
```
# Deeper
model3 = Sequential()

# 2 hidden layers
model3.add(Dense(50, input_dim=784, activation='relu'))
model3.add(Dense(50, activation='relu'))

# 1 output layer
model3.add(Dense(10, activation='softmax'))

# stochastic gradient descent. categorical_crossentropy is a loss function often␣
 ↪used when
# you have values that can only be in one category.
model3.compile(loss='categorical_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
model3.fit(x_train, y_train, epochs=10, validation_split=0.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 3s 61us/sample - loss: 0.5282 -
accuracy: 0.8129 - val_loss: 0.4007 - val_accuracy: 0.8563
Epoch 2/10
54000/54000 [==============================] - 3s 60us/sample - loss: 0.3879 -
accuracy: 0.8609 - val_loss: 0.3763 - val_accuracy: 0.8615
Epoch 3/10
54000/54000 [==============================] - 3s 52us/sample - loss: 0.3502 -
accuracy: 0.8714 - val_loss: 0.3574 - val_accuracy: 0.8633
Epoch 4/10
54000/54000 [==============================] - 3s 58us/sample - loss: 0.3286 -
accuracy: 0.8808 - val_loss: 0.3466 - val_accuracy: 0.8708
Epoch 5/10
54000/54000 [==============================] - 3s 52us/sample - loss: 0.3120 -
accuracy: 0.8848 - val_loss: 0.3391 - val_accuracy: 0.8758
Epoch 6/10
54000/54000 [==============================] - 3s 46us/sample - loss: 0.2987 -
accuracy: 0.8900 - val_loss: 0.3467 - val_accuracy: 0.8750
Epoch 7/10
54000/54000 [==============================] - 3s 46us/sample - loss: 0.2879 -
accuracy: 0.8923 - val_loss: 0.3451 - val_accuracy: 0.8737
Epoch 8/10
54000/54000 [==============================] - 3s 48us/sample - loss: 0.2785 -
accuracy: 0.8965 - val_loss: 0.3236 - val_accuracy: 0.8838
Epoch 9/10
54000/54000 [==============================] - 3s 48us/sample - loss: 0.2690 -
```

```
accuracy: 0.9000 - val_loss: 0.3712 - val_accuracy: 0.8730
Epoch 10/10
54000/54000 [==============================] - 3s 47us/sample - loss: 0.2615 -
accuracy: 0.9021 - val_loss: 0.3258 - val_accuracy: 0.8843
```

[16]:
```
<tensorflow.python.keras.callbacks.History at 0x12ca947d0>
```

[17]:
```
%%capture
_, test_acc3 = model3.evaluate(x_test, y_test)
```

[18]:
```
print(test_acc3)
```

```
0.8762
```

Convolutional Neural Nets CNNs are out of the scope of this lecture, but I wanted to include them to show you how that more complicated neural network architectures exist, and you can use them to improve performance. You can think of CNNs as a more complex, deep learning architecture that does an operation called convolution, maxpooling and then before passing in data to a multilayer perceptron.

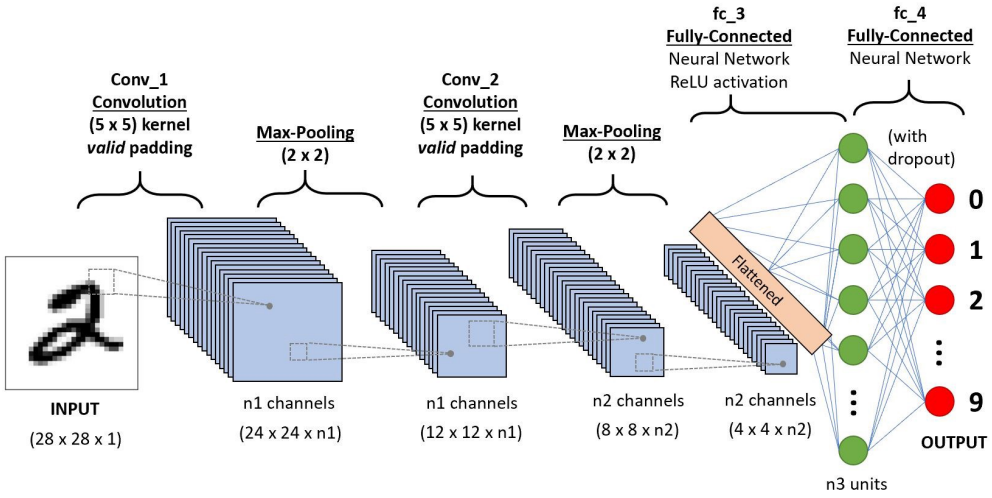
Convolution allows you to extract "hidden," higher-level features that are not immediately obvious from your input.

title

Maxpooling reduces dimensionality, and also extracts the more dominant features.

title

The outputs of these operations are then flattened (like we did before), and passed into a simple neural net for classification.



title

[19]:
```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
import numpy as np
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
x_train = x_train[:,:,:,np.newaxis] / 255.0
x_test = x_test[:,:,:,np.newaxis] / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

[20]:
```
model4 = Sequential()
model4.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu',␣
 ↪input_shape=(28,28, 1)))
model4.add(MaxPooling2D(pool_size=2))
model4.add(Flatten())
model4.add(Dense(10, activation='softmax'))
model4.compile(loss='categorical_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
```

[21]:
```
model4.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 64)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 12544)             0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                125450
=================================================================
Total params: 125,770
Trainable params: 125,770
Non-trainable params: 0
_________________________________________________________________
```

[22]:
```
model4.fit(x_train, y_train, epochs=10, validation_split=0.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [==============================] - 25s 467us/sample - loss: 0.4444 -
accuracy: 0.8457 - val_loss: 0.3367 - val_accuracy: 0.8783
Epoch 2/10
54000/54000 [==============================] - 22s 407us/sample - loss: 0.3156 -
accuracy: 0.8885 - val_loss: 0.3046 - val_accuracy: 0.8913
Epoch 3/10
54000/54000 [==============================] - 23s 419us/sample - loss: 0.2827 -
accuracy: 0.9004 - val_loss: 0.2910 - val_accuracy: 0.8962
```

```
Epoch 4/10
54000/54000 [==============================] - 22s 399us/sample - loss: 0.2610 -
accuracy: 0.9072 - val_loss: 0.2848 - val_accuracy: 0.8970
Epoch 5/10
54000/54000 [==============================] - 21s 383us/sample - loss: 0.2458 -
accuracy: 0.9130 - val_loss: 0.2888 - val_accuracy: 0.8987
Epoch 6/10
54000/54000 [==============================] - 22s 399us/sample - loss: 0.2315 -
accuracy: 0.9171 - val_loss: 0.2753 - val_accuracy: 0.9045
Epoch 7/10
54000/54000 [==============================] - 21s 397us/sample - loss: 0.2195 -
accuracy: 0.9205 - val_loss: 0.2759 - val_accuracy: 0.9018
Epoch 8/10
54000/54000 [==============================] - 21s 392us/sample - loss: 0.2083 -
accuracy: 0.9257 - val_loss: 0.2820 - val_accuracy: 0.9045
Epoch 9/10
54000/54000 [==============================] - 22s 404us/sample - loss: 0.1979 -
accuracy: 0.9289 - val_loss: 0.2730 - val_accuracy: 0.9075
Epoch 10/10
54000/54000 [==============================] - 22s 400us/sample - loss: 0.1881 -
accuracy: 0.9331 - val_loss: 0.2747 - val_accuracy: 0.9032
```

[22]:
```
<tensorflow.python.keras.callbacks.History at 0x12ded2490>
```

[23]:
```
%%capture
_, test_acc4 = model4.evaluate(x_test, y_test)
```

[24]:
```
print(test_acc4)
```

```
0.8984
```