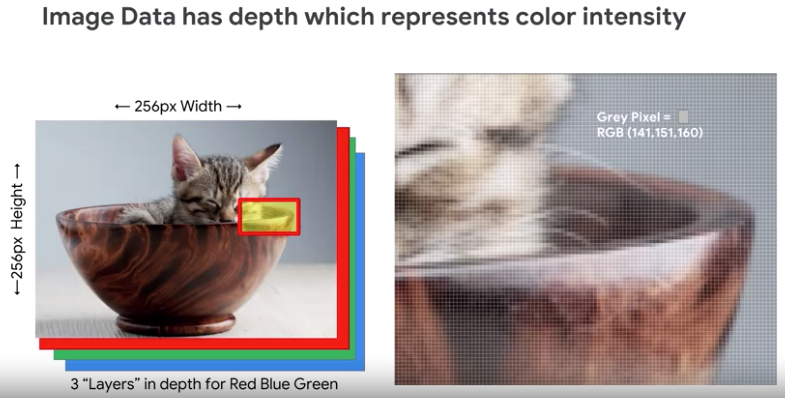
Structured vs Unstructured Data:

A single feature vector with a small number of components or elements in that array. Again, if you're already familiar with this structure because these vectors are the tensors that you've been feeding into your early models tensor flow.

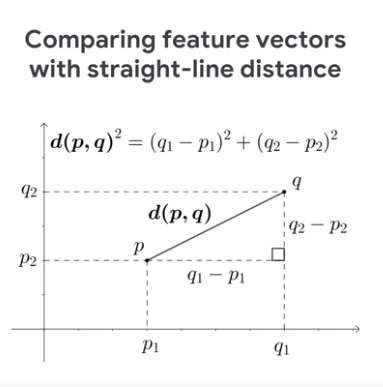


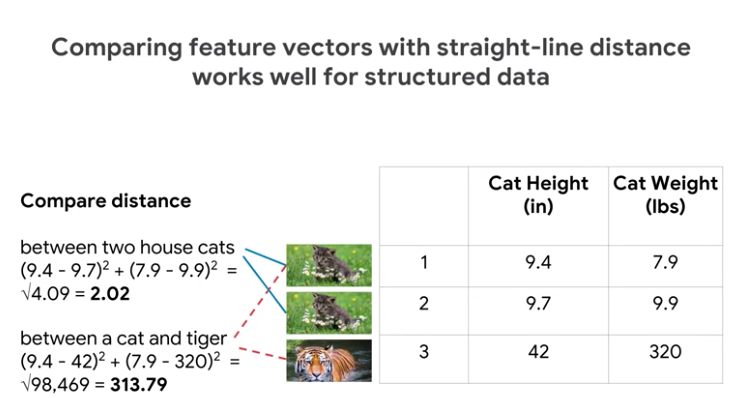


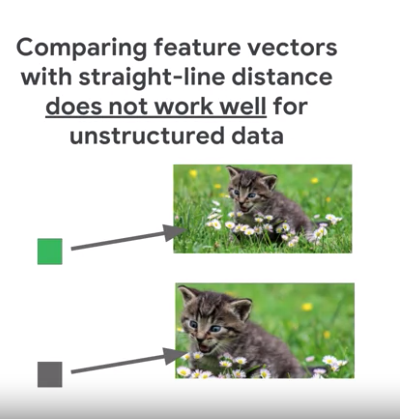


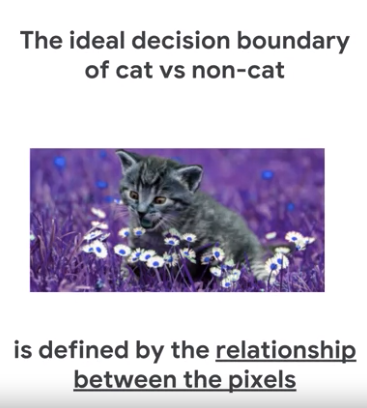
The real challenge though, in which unstructured data like images is different from structured data has to actually do with **locality**.

At a differentiating point that'll make this all clear is how we compare one vector of data with the other. One of the most common ways of comparing vectors is the simple euclidean distance. In euclidean distance, we consider the element wise difference between each of the vectors components and the distances between them grows as the sum increases. Lower the distance, the more similar those vectors are. With structured data, euclidean distances behave exactly as you might expect, euclidean distance often correlates with the semantic space.









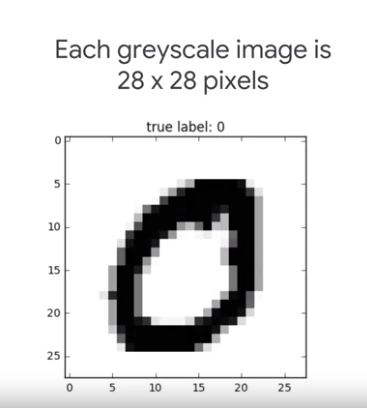
Linear Models for Image Classification:

we'll cover the MNIST image dataset that you'll be using for part of this course. Then, we'll tackle an image classification problem, with a linear model in TensorFlow. After that, we'll move on to tackling the same problem using a deep neural network. Lastly, we'll close out the discussion and the application of dropout, which is the regularization technique for neural networks. To help them prevent from overfitting or memorizing our training dataset.

understand how image data is represented as floating point numbers, that can then be flattened. You can compare functions from model confidence and image classification with the focus on Softmax. you need to train and evaluate a linear model for image classification using TensorFlow. After that as you might guess, you got to do the same thing except with a Deep Neural Network. Lastly, you're going to understand and how to actually apply dropout as a regularization technique for Deep Neural Networks.

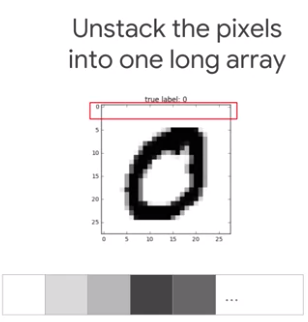


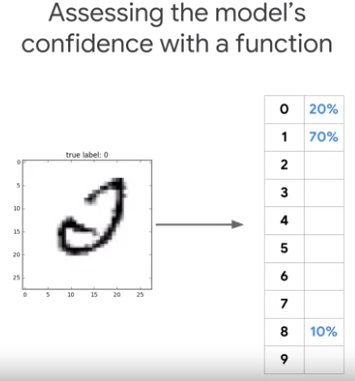


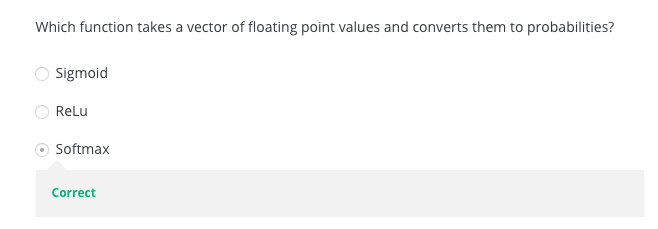


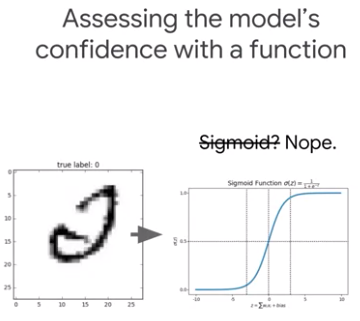
If we want to convert this 2-D image into a single dimensional vector. What can we do? Well, we could flatten the image. It'll take each row of the pixel data and line it up and along single row end to end.

If we want to convert this 2-D image into a single dimensional vector. What can we do? Well, we could flatten the image. It'll take each row of the pixel data and line it up and along single row end to end.

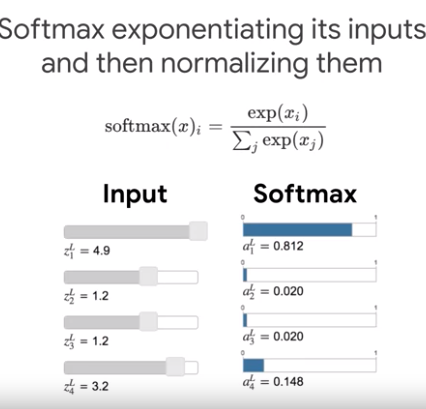








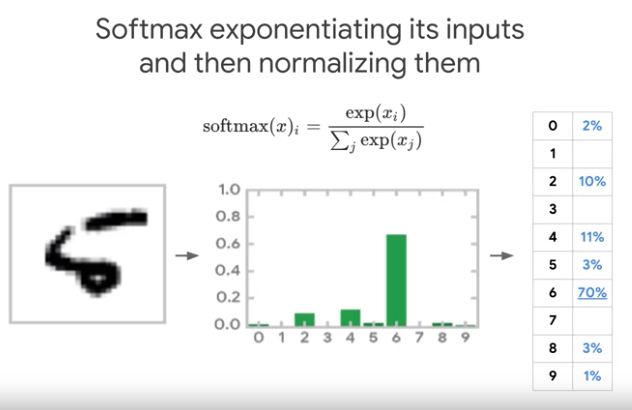
This will work if there are only two classes. Here we've got 10 numbers and the total of these probabilities has to be equal to one, which means the classes are exclusive. One of the most common ways and the correct answer, is for using a Softmax function. What does the Softmax function does, is it exponentiates its inputs and then it normalizes them. The exponentiation means that one more unit of evidence, increases the weight given to any hypothesis multiplicatively. Conversely, having one less unit of evidence means that hypothesis gets a fraction of its earlier weight. Basically, it makes the high value is higher and the lower value is lower, but keeps the relative order the same. In addition, as you can see here, the Softmax function normalizes the weights. So, they all add up to one when they're summed up together and this forms that valid probability distribution.



Getting the probabilities each class / total value. (Normalized to one)

A close up of text on a white background

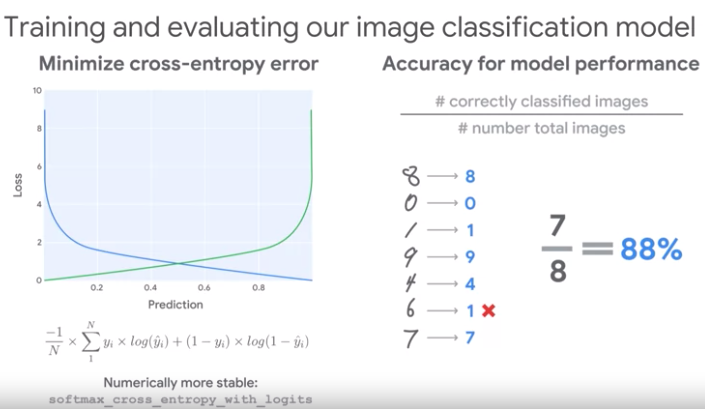
Description automatically generated



Loss Function for Logistic Regression:

A close up of text on a whiteboard

Description automatically generated



We will avoid numerical issues that come with taking the log of really really tiny tiny numbers, we'll use an optimized TensorFlow function called Softmax cross-entropy with logits version number two.

**Linear Models:**

Simplier model to overfit and maintain.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

**DNN:**

What is a linear model good for?

A screenshot of a cell phone

Description automatically generated

A close up of a piece of paper

Description automatically generated

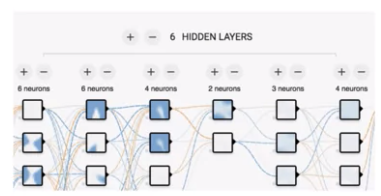
What you've figured out by now is using the feature option available to us in this type of model, this dataset is not linearly solvable. The best model that I could train had loss of about 0.6. However, this qualifier of the feature options available to us is crucial, because in fact, there is a feature that can make learning this relationship trivial. You get a magical feature that can unswirl the data and create one through feature engineering but, in absence of that, what we can do in our attempts to engineer new features for linear models fail, what do we do? Well, we could use more complex or complicated models. There are many types of models that are able to learn non-linear decision boundaries.

A close up of a logo

Description automatically generated

Activation function separates linear models from neural networks.

Hidden Layers.z



Consequently, you can think of a neural network as a hierarchy of features. Before neural networks, data scientists spent much of their time doing feature engineering. Now, the model itself is taking some of that responsibility and you can think of these layers as being a form of feature engineering onto themselves.

DNN Model Code:

A screenshot of a cell phone

Description automatically generated

**DNN Dropout Layer:**

Why not add more layers and increase the size of the neural network?

Large means that it would take up more memory, meaning a smaller batch sizes which will slow down optimization. Large also means slower to decide. One of the worst things about really large neural networks and really complex models in general, they're extremely good at overfitting on your training data, and that's something that we don't want.

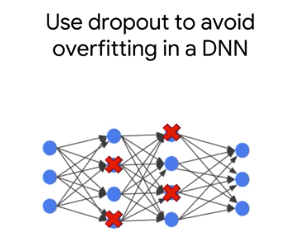
A screenshot of a map

Description automatically generated

Regularization:

A screenshot of a cell phone

Description automatically generated



Dropout's a technique or parts of a neural network, a randomly dropped during training with a probability that's determined by a hyperparameter. At every training step, each neuron has a probability p of temporarily being dropped out. Thus for each training step, a unique neural network architecture is generated.

Accordingly, the final neural network can be described as an average, an ensemble of many different networks.

A screenshot of a cell phone

Description automatically generated

L1 a.k.a, lasso regularization is our regularizer for sparsity. L2 regularization is likely to introduce weight values that are normally distributed about zero, which means that most model weights becomes small but not zero.

A screenshot of a cell phone

Description automatically generated

Rate layer: which probability to drop the neurons in that layer. (Which probability to drop neurons in that layer). Because dropout is only used during training, we pass in a parameter that encodes whether or not the model is in training mode.

A picture containing clock, object

Description automatically generated

So what's the primary driver in neural network that updates the weights during training? If you said back propagation, that's exactly right. So back propagation process that minimizes your loss function, and ultimately gives better weights and reduces your loss function for a better result. But what happens if the neurons in your network start to memorize or over fit to that trained dataset? Like they became too dependent on other neurons and learn complex relationships in your training dataset that just aren't there in the real world?

A close up of a map

Description automatically generated

So again, this only happens during training because this is the only time when we want something like dropout to prevent over fitting. It's not as much. It's not a problem in evaluating and certainly not our problem in testing or predicting. So we need to create a final dropout layer that targets one of our layers for dropout with a certain probability, and we only want that to take effect in training.

A screenshot of a cell phone

Description automatically generated

def dnn\_dropout\_model(img, mode, hparams):

dprob = hparams.get("dprob", 0.1)

X = tf.reshape(tensor = img, shape = [-1, HEIGHT \* WIDTH]) #flatten

h1 = tf.layers.dense(inputs = X, units = 300, activation = tf.nn.relu)

h2 = tf.layers.dense(inputs = h1, units = 100, activation = tf.nn.relu)

h3 = tf.layers.dense(inputs = h2, units = 30, activation = tf.nn.relu)

h3d = tf.layers.dropout(inputs = h3, rate = dprob, training = (mode == tf.estimator.ModeKeys.TRAIN)) #only dropout when training

ylogits = tf.layers.dense(inputs = h3d, units = NCLASSES, activation = None)

return ylogits, NCLASSES

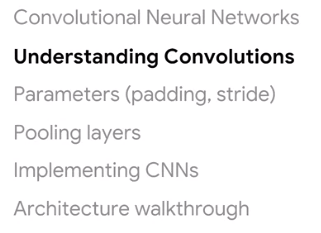
Softmax.

18 Million

More pixels

Image Classifier in later modules

**CNN:**



You should consider when setting up your neural net including padding, stride length, activation functions, and the number of channels. In addition to convolutions, you will see that many CNN architectures use pooling layers to help detect patterns regardless of their location in an image, while reducing the computational requirements for training CNNs.

A close up of text on a white background

Description automatically generated

A close up of text on a white background

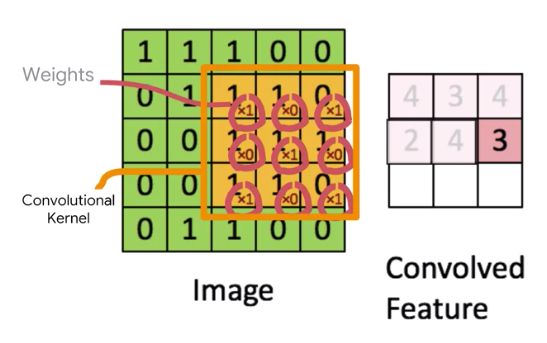
Description automatically generated

A close up of a map

Description automatically generated

A screenshot of a cell phone

Description automatically generated



Output will be smaller size.

Input : 5 x 5

Kernel: 3 x 3

Reduced by : Kernel size -1 = 2

Output: 5 -2 = 3

The right side of the animation shows an example, the size of the input is five, the size of the kernel is three. So, the output shape is reduced by kernel size minus one, which means by two. In general, if you use square kernels you should expect that the output shape will be smaller by kernel size minus one along each dimension of the input.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

TensorFlow processing units or GPUs, graphical processing units. These types of processors can be much faster than traditional CPUs at working with CNNs, because they can compute the values of the convolution operation for all the different positions of the convolution kernel in parallel.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

The kernel depth, the cube you see convolving is always equal to the number of input channels, three in this case for RGB. Now as output from the convolutional layer, the output layer has depth equal to the number of kernels applied.

**CNN Model Parameters:**

Why the input and output sizes different? Why is the input of a 300 by 300 image, but each filter output is only 296 by 296? If you said, this was because of the pixels at the edge of the image for which there are no valid neighbors to compute the full convolution, then you're correct.

A close up of a keyboard

Description automatically generated

If you're using square kernels and the size of the kernel side is an odd number, just take the length of the kernel side, subtract one, and then divide that in half. You will get the required thickness of the border to preserve the shape of the input.

Padding thickness = (Kernal side – 1) / 2

that unless you're building your own implementation of the convolutional layer, you won't have to do these calculations yourself. Most machine learning frameworks, like TensorFlow, provide built-in support for padding

A screenshot of a cell phone

Description automatically generated

Recent research has shown that it's better to use smaller kernel sizes and add more convolutional layers. In other words, instead of using a nine by nine filter, try sequencing two layers of three by three filters.

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

When this type of maxpooling operation is applied to the grasshopper image, the image size is cut in half along both the horizontal and vertical dimensions.

A screenshot of a cell phone

Description automatically generated

Implementing CNNs with TensorFlow:

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

A screenshot of a cell phone

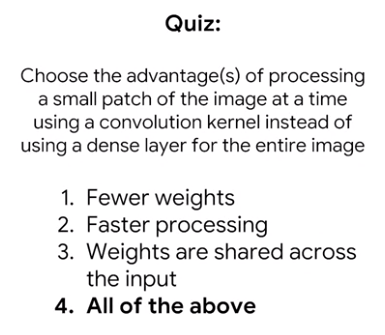
Description automatically generated

A close up of text on a white background

Description automatically generated

You saw that using a dense layer as the first hidden layer of a deep neural network meant that every pixel of the input image had to be connected to every neuron on the first layer. (Which is bad, we need to define weights for each edge.)

You also learned how kernels enable convolution layers to detect patterns, and how pixels are placed next to each other so that the convolution layer can be trained to find edges, corners, textures and other visual patterns. Lastly, you saw that convolutional layers are just collections of these filters which are sometimes used together with pooling layers. Remember that the goal of the convolutional and pooling layers is to recognize patterns and reduce the dimensionality of the image before ultimately passing it off as a flattened feature vector into a fully connected to deep neural network like the one you see here.



Weights are the kernel weights.

A screenshot of a cell phone

Description automatically generated