Module 21: Deep Neural Networks (Part 1)

Quick Reference Guide

Learning Outcomes:

1. Articulate the basics and evolution of neural networks.
2. Compute the output of a neural network given parameters and data.
3. Train a simple neural network.
4. Apply the basics of Keras for developing machine learning models.
5. Use Keras to create, compile, fit, and evaluate a model.
6. Select the appropriate output layer and loss function for multiclass classification.
7. Adjust hyperparameters and model architecture to maximize model generalizability.

Introduction to Neural Networks

Another technique for machine learning is known as a **neural network**. Neural networks were invented decades ago, but their popularity only exploded in the 2010s when researchers found great success applying them initially to **image classification**.

Consider the problem of trying to classify whether an image contains a horse or not. An image is just a set of features like any other observation. For example, if you have a picture that is 640 pixels wide by 480 pixels tall, and with three color channels for red, green, and blue. Then the image is just a list of 921,600 features. In other words, its dimensionality is 921,600. In principle, you could train a decision tree, logistic regression model, or an SVM on these 921,600 features. But with so many features, you run the risk of overfitting. Unless you had some absolutely gigantic corpus of example

images. And this input set would need to potentially cover all kinds of different lighting conditions, rotations, et cetera.

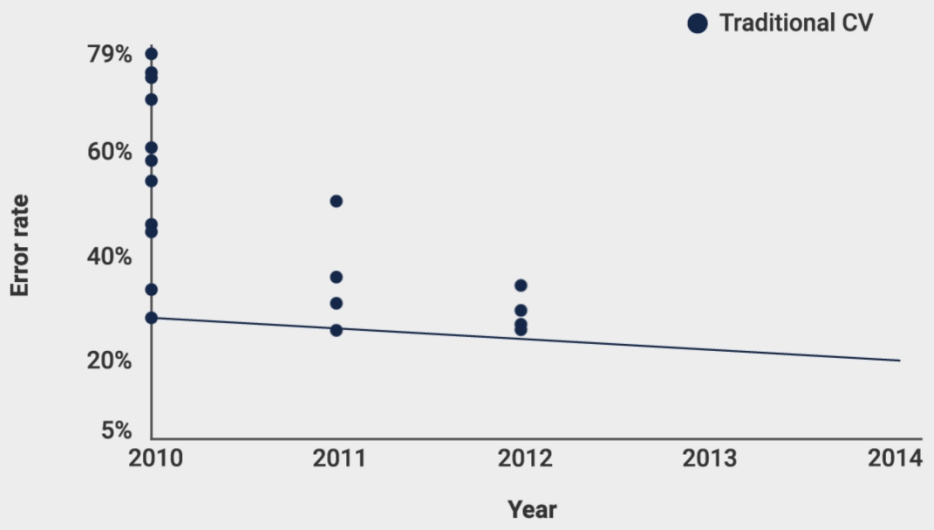
One approach to improve the training process is to first **preprocess** the images, converting them into some lower-dimensional form that captures the most important aspects. The focus with this approach is to find the best possible preprocessing technique. Researchers all around the world spend huge amounts of time and effort trying to come up with newer, better techniques for preprocessing images.

Image classification is an incredibly important problem. Automated classification of images has near limitless applications, from social media to archaeology, from national defense to medicine. With so much potential impact, it is important to be able to compare the performance difference between different image classification techniques. To that end, researchers worked together to create a standard benchmark to compare their algorithms and preprocessing techniques. The most famous such benchmark is called **ImageNet** and was created by Fei-Fei Li at Stanford University. ImageNet consists of a massive database of labeled images. In the 2010s, this dataset made an enormous impact on the machine learning field and the broader world.

To participate in the ImageNet challenge, teams would come up with a new approach to image classification, often a new preprocessing technique. They would train their classifier using the labeled training set and then generate their best predictions for the unlabeled test set. Each year, the test set accuracy rates were published along with papers describing each team's approach. When ImageNet was first released as a contest, machine learning algorithms were far inferior to humans on this dataset. This

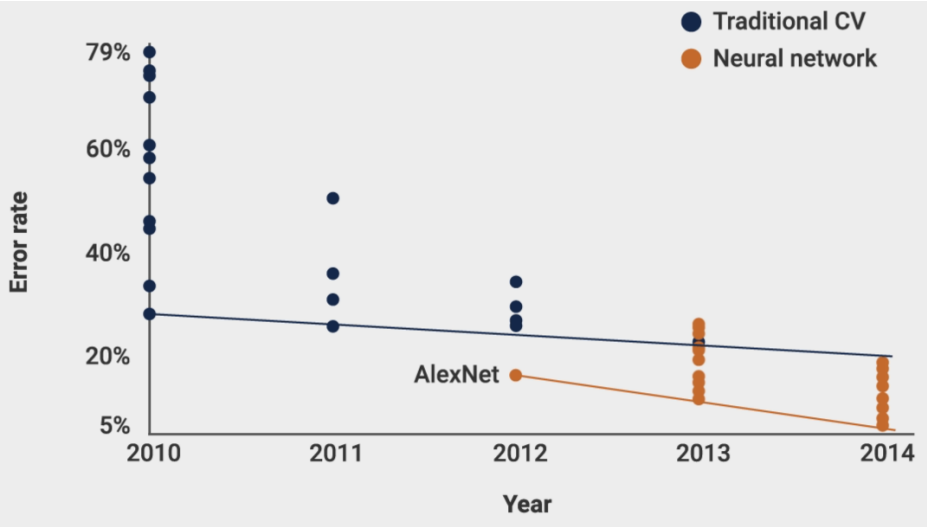
dataset acted as a grand challenge, attracting researchers from around the world to try build an algorithm that exceeded human capability.

Here is a plot of the participants in the ImageNet challenge from 2010 to 2012:



In the year 2010, the spread of entrants was fairly wide, and no team was anywhere near human-level correctness, which is roughly around 5%. By 2012, the spread of the teams' performance was narrower, but human-level image classification still seemed far off. A rough line provides a crude sense of the seemingly slow progress of the field on this challenge.

In 2012, **AlexNet** was the first neural network to succeed in the challenge. Neural networks were not new, but AlexNet was the first time a convolutional network was trained on a dataset large enough and on hardware fast enough to yield a neural network that generalized well on an important real-world benchmark. The improvement was immense. Note the massive gulf between AlexNet and the best traditional computer vision approach.



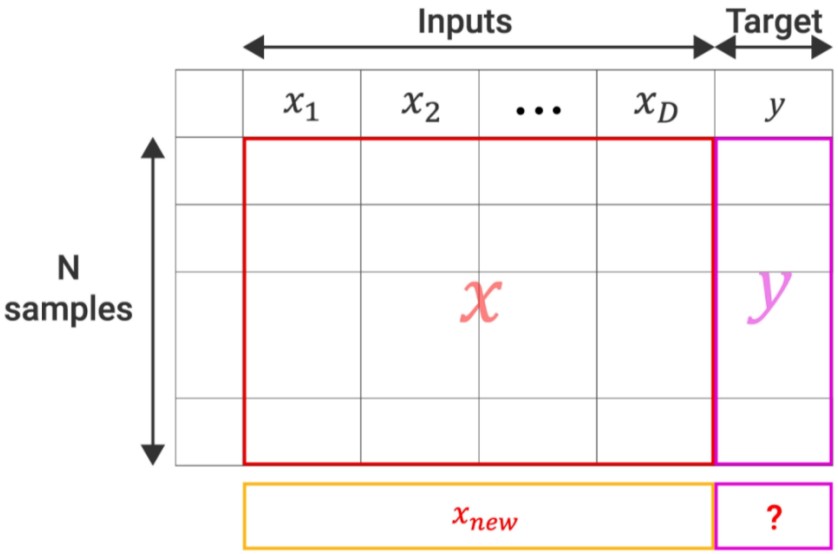
AlexNet was an incredible breakthrough, inspiring teams around the world to tweak the approach and do even better. By the following year, almost all entries were neural network based, and many of them performed better than AlexNet had in 2012. By 2014, human equivalent classification was well within reach. A line is plotted to show how rapid progress was once neural networks entered the scene.

By 2015, neural networks had reached almost human levels of performance. And by 2017, the last year of the contest, the vast majority of teams participating had algorithms that could perform at or slightly better than humans. So in less than a decade, ImageNet was solved and off-the-shelf software was capable of beating humans at this important task.

It is hard to overstate how important this moment in history was. AlexNet kicked off a revolution that affected everything. Today, neural networks have been applied in many domain areas, resulting in vastly better performance in a wide variety of fields, including speech recognition and artificial face generation. They have even beaten humans at the ancient game of Go, an AI benchmark that was thought to be decades away.

Foundations

To understand the basic structure of neural networks as a generalization of linear regression, you can first recap your setup. You start with a dataset that consists of measurements from some system:

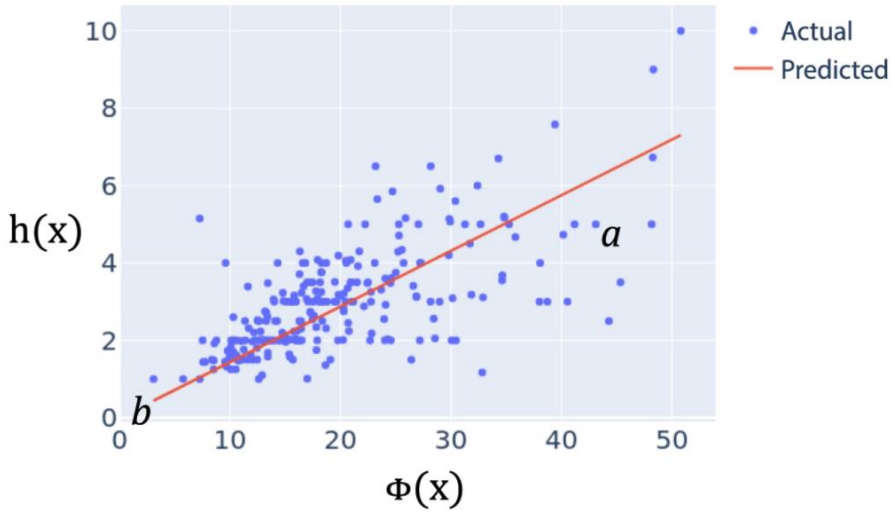


There are D inputs or independent variables, x_1 through x_D, and one dependent or target variable y. Those are the columns in the DataFrame. You have N samples for x and y, which are arranged into rows. Your goal is to construct a model that, given some previously unseen input value x_{new}, will make a good prediction for its corresponding target value y.

It has been found useful to define a set of features, ϕ_0 through ϕ_{m-1}, based on the original inputs x. These are nonlinear transformations of x that are designed to improve the performance of the model. You have seen examples of this in linear regression and logistic regression, where nonlinear features introduced curvature into your regression lines and decision boundaries. Recall that in linear regression, a simple model, $h(x)$,

was proposed to predict the output as a linear combination of the inputs. In the one-dimensional case, this is simply a line with intercept b and slope a:

$$h(x) = a\phi(x) + b$$



The task of linear regression was to find the values of a and b that minimized the loss function.

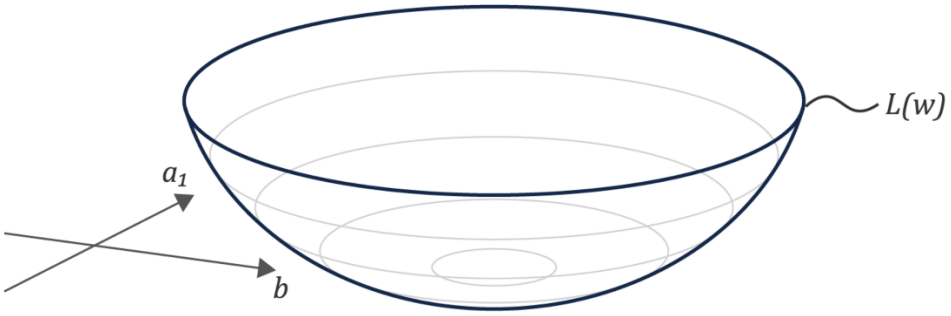
In general, if you have m features instead of just one, then the model is a sum of m terms. Those are $b + a_1\phi_1(x)$ through $a_{m-1}\phi_{m-1}(x)$. You can express this compactly in a vector notation by defining a weight vector A, a feature vector $\phi(x)$, and an offset b. The model then becomes $h(x) = A^T\phi(x) + b$. The offset is sometimes called the **bias**, but this really has nothing to do with the statistical bias of the model.

The linear regression problem can then be expressed very succinctly as finding the parameters a and b that minimize the squared loss function, which is the sum over all of the data points of the square of the difference between the predicted output $h(x_i)$ and the true output y_i.

$$L(a,b) = \sum_{i-1}^{N} (h(x_i) - y_i)^2$$

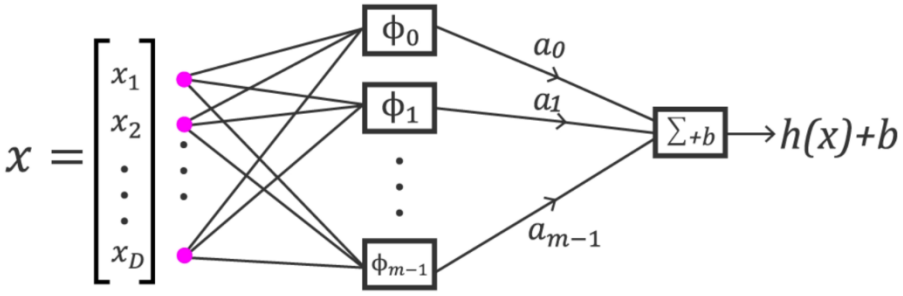
$$= \sum_{i=1}^{N} (A^T \phi(x_i) + b - y_i)^2$$

Plugging the linear regression model into this, you get a summation of $(A^T \phi(x_i) + b - y_i)^2$. In optimization theory, this is called a **convex problem**. The cost function is shaped like a smooth bowl.



The goal is simply to find the point at the bottom of the bowl. Gradient descent and its close cousin, stochastic gradient descent, have very effective methods for finding this minimum.

Here is a representation of the linear regression model as a graph:

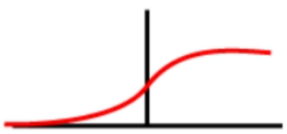


$$h(x) = A^T \phi(x) + b = \sum_{m=0}^{m-1} A_m \phi_m(x) + b$$

Think of each of the lines in this graph as transmitting a single number. You enter the numbers on the left at the input nodes, which are marked x_1 through x_D. These are all of the components of the input vector. These numbers then flow from left to right and enter the boxes in the middle marked with ϕ_0 through ϕ_{m-1}. Each ϕ box applies a nonlinear feature transformation. The outputs of these boxes then go through lines which multiply them by weights a_0 through a_{m-1}. Finally, they are added together with the offset b, and this is the output of the model. Notice how this corresponds with your formula for linear regression. First, the ϕ functions are evaluated, then they are multiplied by weights, and added up. The objective of the training process is then to adjust the coefficients a and b so as to minimize the error.

Neural networks start from the same picture and generalize it in several ways. First, they use very specific feature functions, which in the neural network community are called **activation** functions. A few of the most common activation functions are the **sigmoid** function, which is at the core of logistic regression:

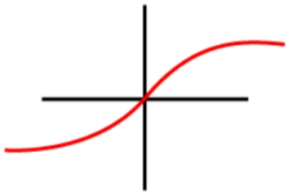
$$\phi(x) = \frac{1}{1 + e^{-x}}$$



In a neural network, it acts like a switch, either opening or closing a channel.

The **tanh** or **hyperbolic tangent** function; this function puts saturation limits on the output of the neuron:

$$\phi(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



It allows small values to go through more or less unchanged, but truncates large positive or negative values.

And the rectified linear unit or **ReLU**:

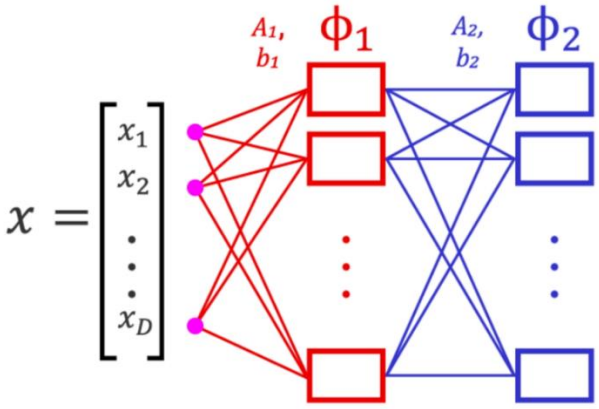
$$\phi(x) = \max(0, x)$$

This takes the maximum between the input value and 0. And thus it prevents any negative numbers from going through.

A second generalization of neural networks is that they add coefficients to all of the lines in the graph, not just the ones on the output side. So whereas before with linear regression, the input to the feature vector was simply x. Now with neural networks, it will be a matrix $A_1 x$ plus an offset vector b_1.

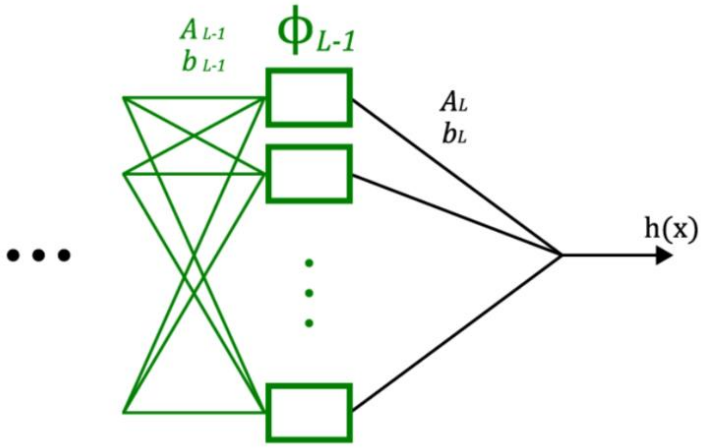
The third generalization of neural networks is that they allow for many layers to be used. These layers are set out sequentially so that each feature

vector ϕ is fed by a matrix A multiplied by the output of the previous layer plus an offset vector b.



$$c_1 = \phi_1(A_1 x + b_1)$$
$$c_2 = \phi_2(A_2 c_1 + b_2)$$

Use capital letter L for the number of layers in the network and count them from left to right.



$$c_{L-1} = \phi_{L-1}(A_{L-1} c_{L-2} + b_{L-1})$$
$$h(x) = A_L c_{L-1} + b_L$$

There are $L - 1$ internal or hidden layers. And the last one is the output layer. Use lowercase c for the outputs from each layer. The final output of

the model $h(x)$ is the transformation of c_{L-1} with coefficients A_L and b_L. Then c_{L-1} is the output of layer $L-1$. This equals the feature ϕ_{L-1} applied to the input of that layer, which is the transformation of the output of the previous layer through coefficients A_{L-1} and b_{L-1}. You continue in this way, feeding each layer with the output of the previous layer, which is transformed through a dense mesh of coefficients A and b until you reach the input.

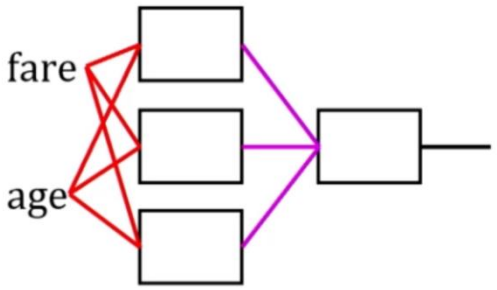
A final generalization has to do with the output layer. By adding a sigmoid activation function to the output node, you immediately transform the regression model into a binary classification model.

The training problem for neural networks is to find all of the A and b coefficients such that the loss function is minimized. This is very similar to linear regression. However, the model is much more complicated since it consists of a set of L-nested functions. This complexity of neural networks allows it to take different forms and adapt to different data sets. However, it also makes neural networks much more difficult to train.

Neural Network Playground

Suppose you are building a model for Titanic passengers. You know the fare that a passenger paid and their age. And you want to know: Did they survive the sinking of the Titanic? So you have A_1, A_2, b_1, and b_2:

$$A_1 = \begin{bmatrix} -0.5016 & -0.5452 \\ 0.0305 & -0.0182 \\ -0.4035 & -0.0335 \end{bmatrix} \qquad A_2 = \begin{bmatrix} -0.0166 \\ 9.374 \\ 0.0185 \end{bmatrix}$$



$$h(x) = \phi(A_2 a_1 + b_2)$$

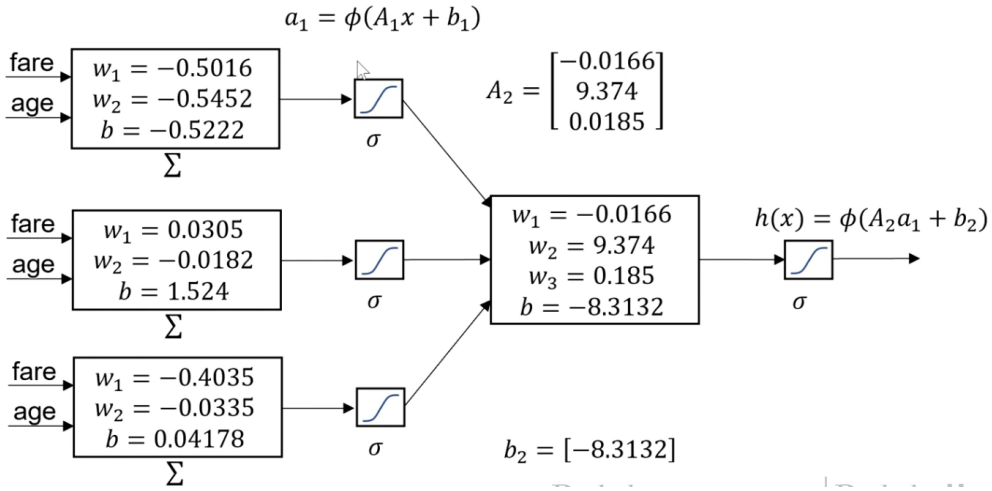
$$b_1 = \begin{bmatrix} -0.5222 \\ 1.524 \\ 0.04178 \end{bmatrix} \qquad b_2 = [-8.3132]$$

$$a_1 = \phi(A_1 x + b_1)$$

And in this case, you have chosen the logistic activation function:

$$\phi(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

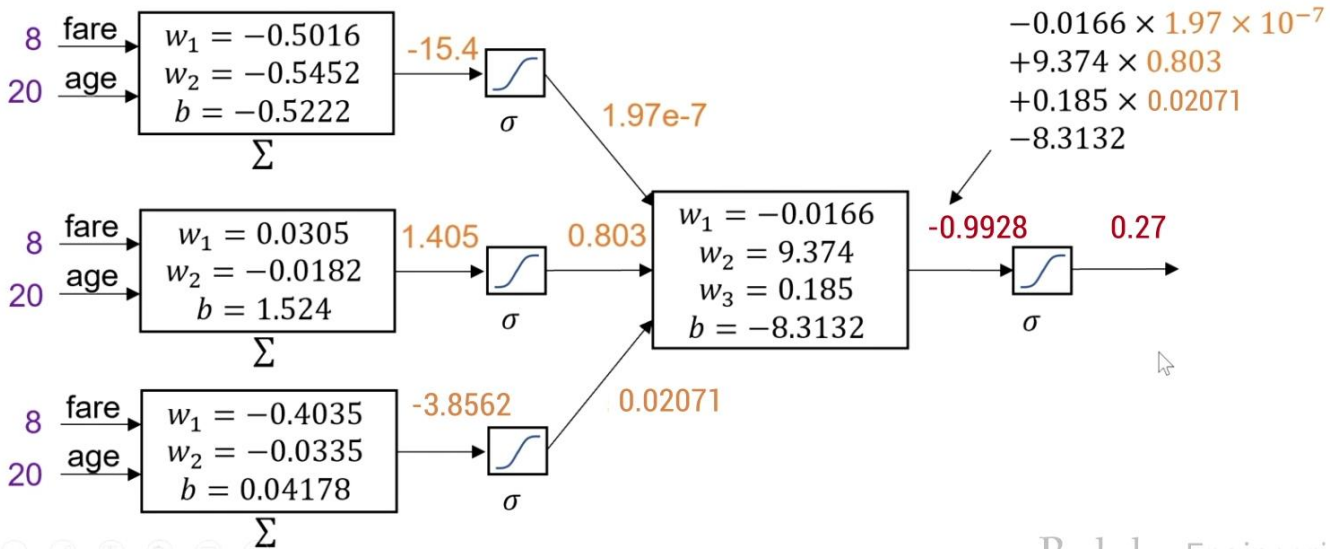
These matrices can be viewed a different way. Here, the resulting weights and intercepts of the A_1 and b_1 matrices are shown for each of the three neurons in the hidden layer:



For example, the top-left entry in A_1 is w_1 for the first neuron. In order to compute the overall output of this network for a specific value, you plug in

the fare and age, compute some weighted sums, put those into the logistic activation function, and then pass those to the next level to repeat the process.

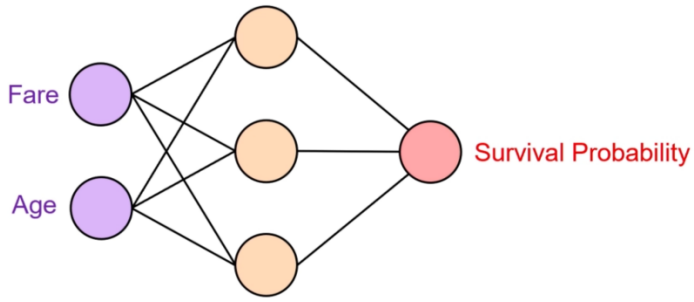
For example, if a passenger aged 20 paid an $8 fare, the network generates the following values:



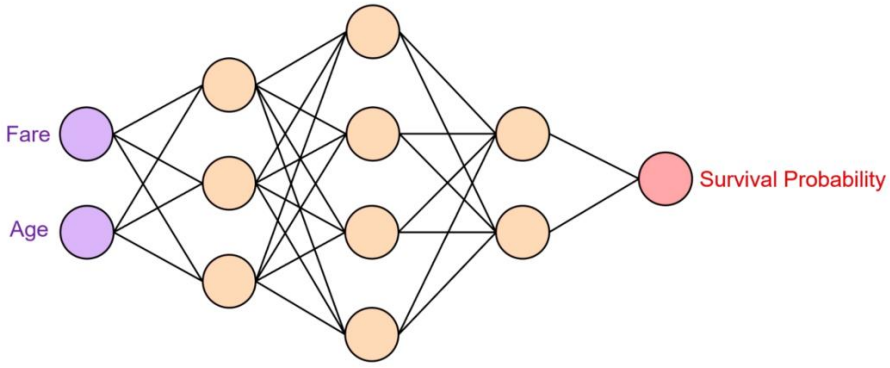
If a passenger paid $8 and they were 20, then the first neuron computes the value −15.4. That is the weighted sum of the fare and the age, including the interceptor. These weights come from the A_1 and b_1 matrices. You then apply the logistic activation function and get 1.97 times 10 to the minus 7th.

Repeating this for the second neuron, you get back 1.405 as the weighted sum and 0.803 as the value after the activation function. Lastly, the third neuron computes its activation, which is 0.02071. These three intermediate features are then fed to the output neuron. It in turn generates an ultimate value of 0.27. The network believes there is a 27% chance that this passenger survives. That is what the A_1, b_1, A_2, and b_2 matrices mean.

Neural networks are often shown in schematic form:

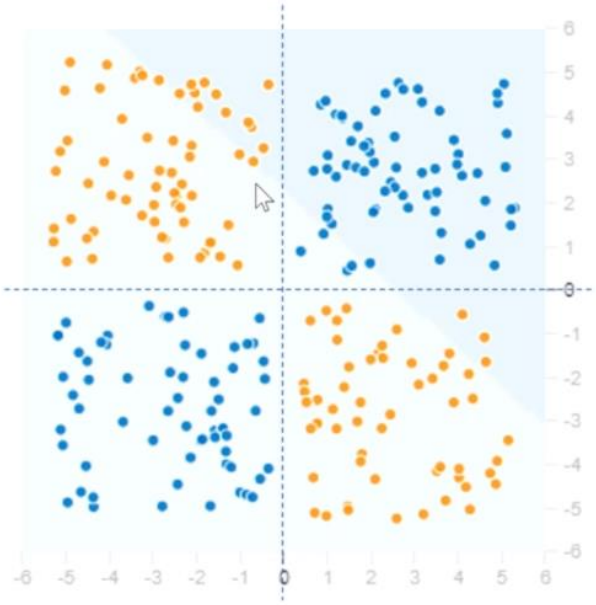


The front layer has the inputs. In this case there are two of them. Next is the hidden layer, and then the output layer, which produces the predictions. You can have multiple hidden layers. For example, this neural network has three, four, and two neurons in three different hidden layers, and a single output, survival probability.



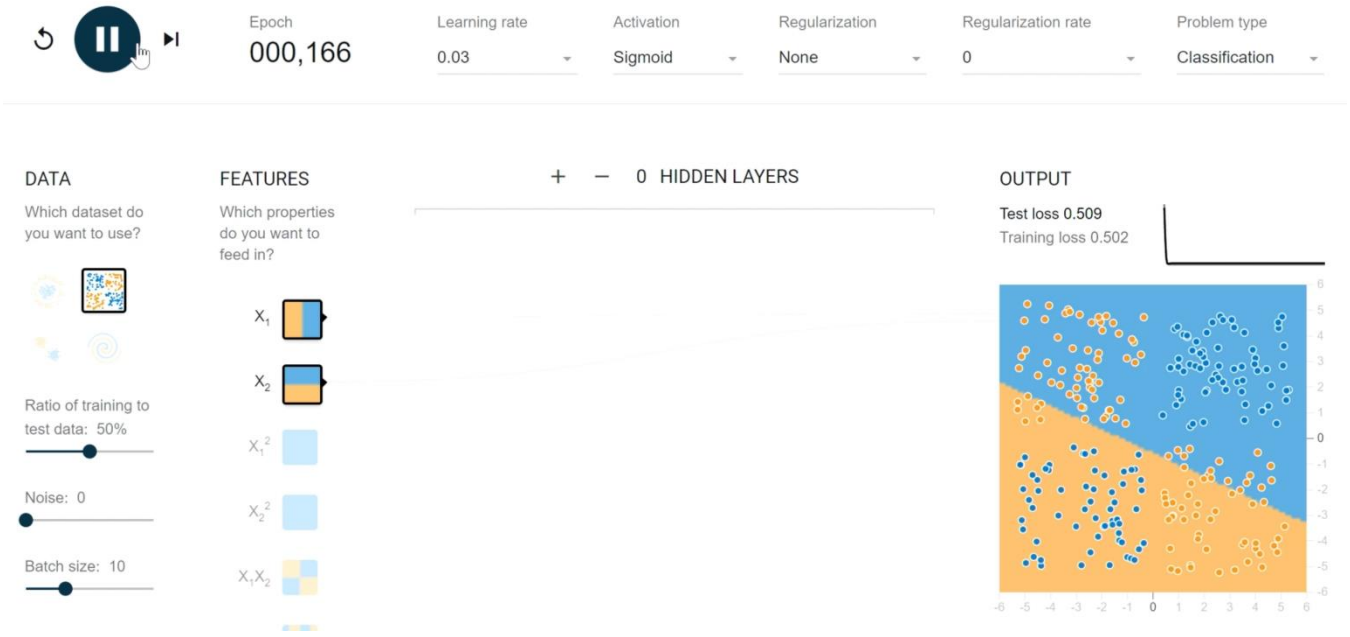
So in this case, the first hidden layer generates three new features, the second uses these to generate four new features, the third uses these to generate two new features. And finally, the output layer uses these two features to give output a prediction.

Consider the data shown, which is clearly not linearly separable in the x, y feature space:

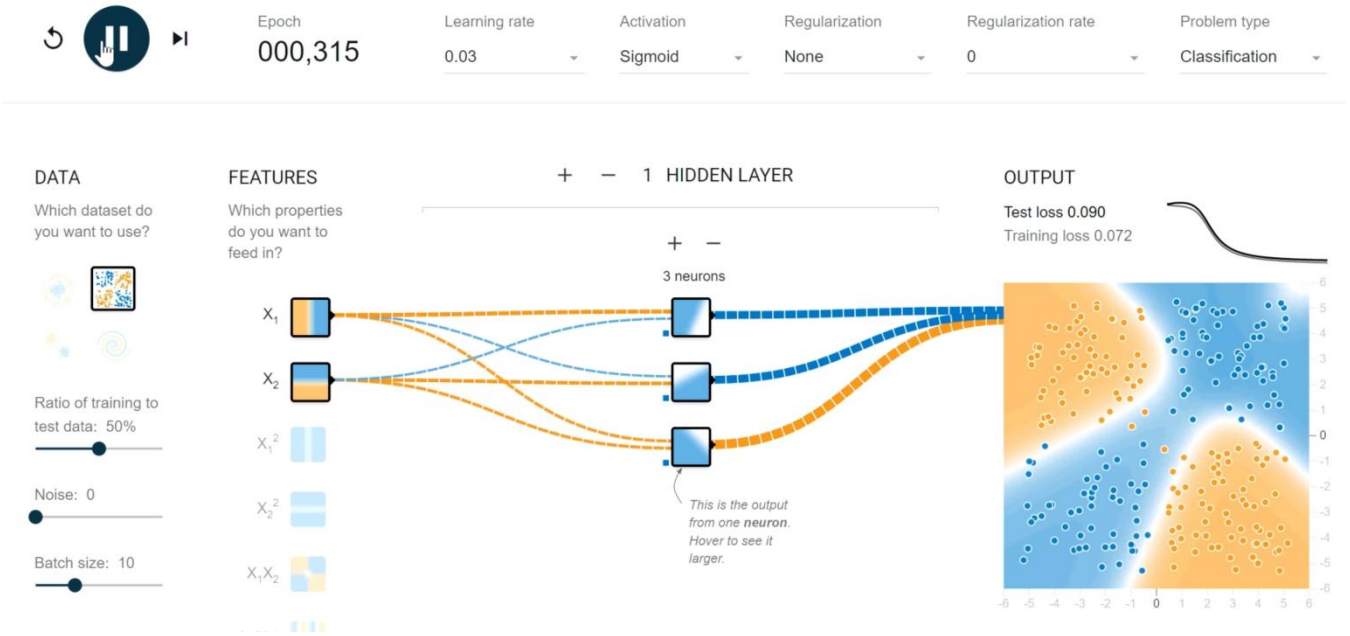


No line can be drawn that separates the orange points from the blue points. However, considering y is X_2 in this example, there is a feature in terms of X_1 and X_2 that can predict the data's classes with high accuracy. If you compute the feature X_1X_2, that is really useful. If $X_1X_2 > 0$, then the data is blue. So if the Xs are both positive, they are in the top-right quadrant, and if the Xs are both negative, then they are in the bottom-left quadrant. So that is a featurization of the data you can use to make it linearly separable.

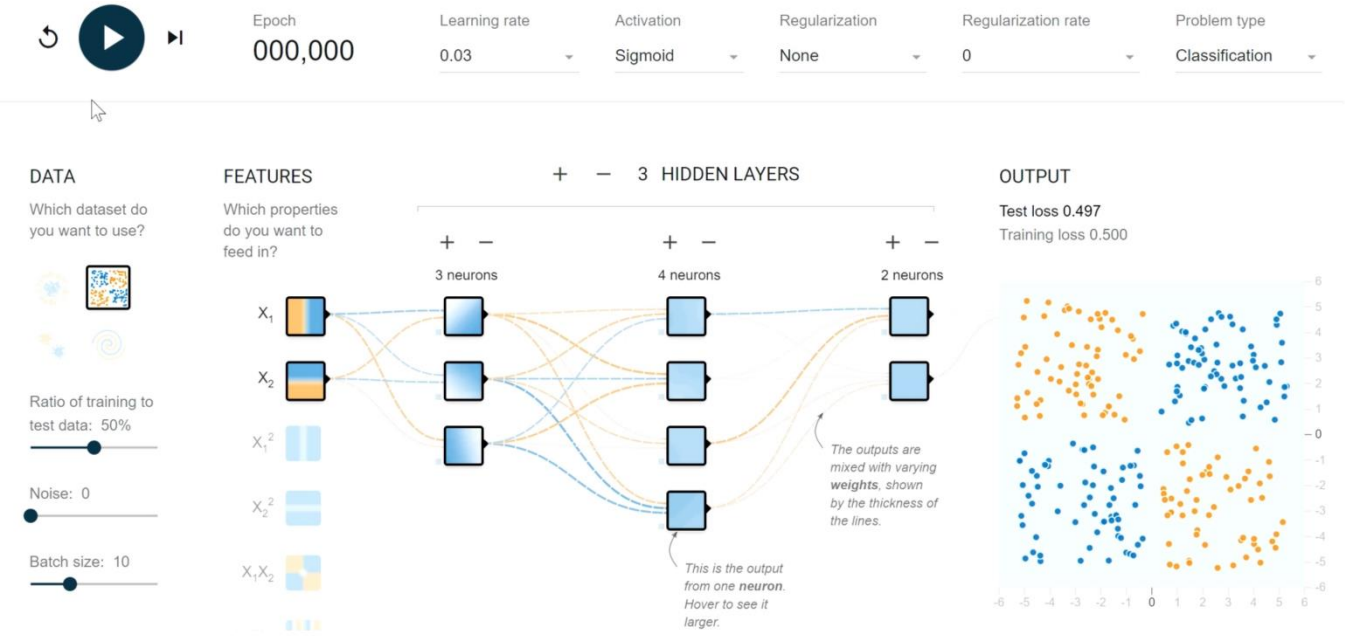
To see how a neural network can come up with this rule on its own, you can use the Neural Network Playground created by TensorFlow. Start with the simplest possible network. In this model, there are no hidden layers. In other words, the classifier is just a logistic regression classifier.



No matter how many times you train this model, it cannot predict the two classes with perfect accuracy. If you add a single hidden layer with three neurons, you end up with a more complex decision boundary.

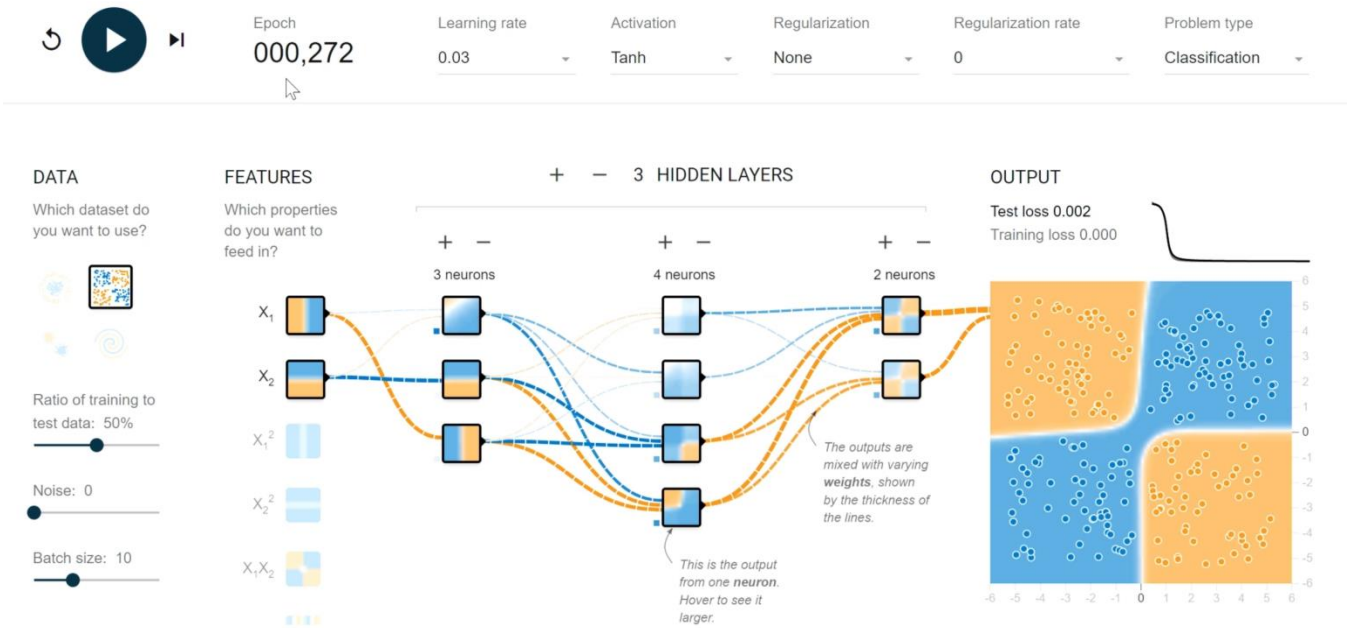


You can also generate the more complicated network from earlier, which has three hidden layers with three neurons, four neurons, and two neurons respectively.

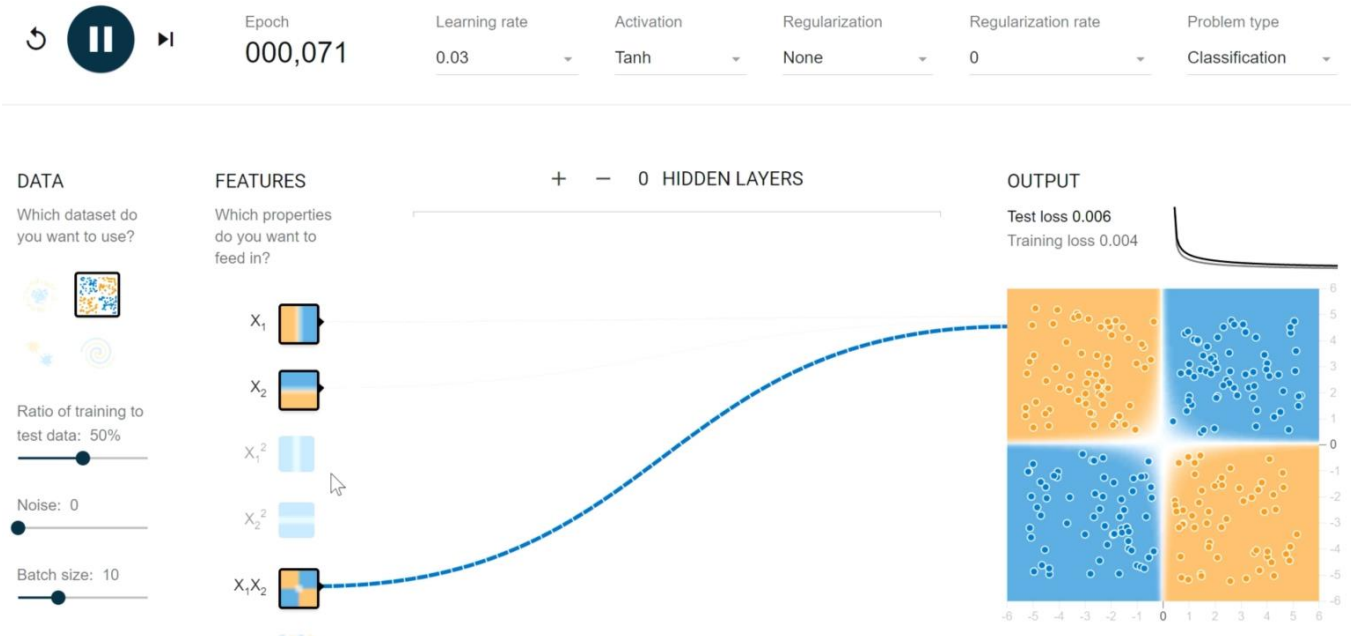


However, if you attempt to train this model, nothing seems to happen. That is because deeper networks are generally harder to train. So instead of using sigmoid, you need to choose a ReLU or Tanh activation function.

In this example, Tanh is used and the network is able to converge:



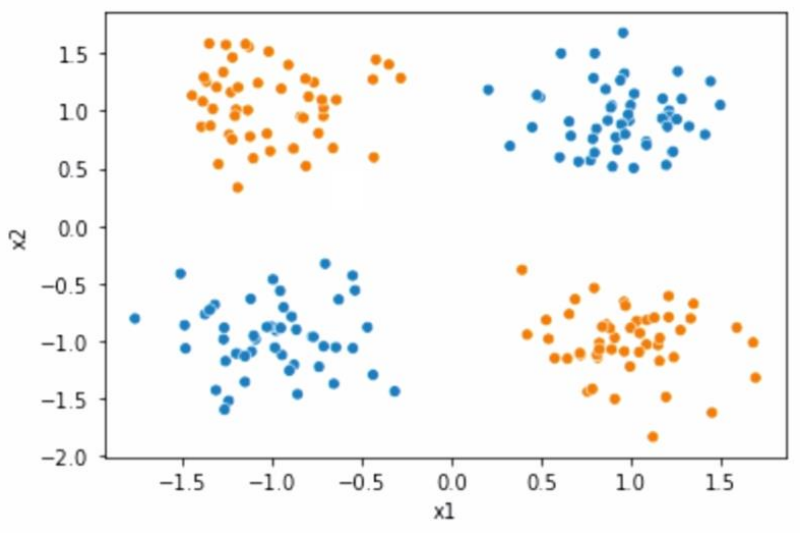
Notice that the two neurons in the final hidden layer have basically discovered the feature $X_1 \times X_2$. It has automatically come up with an output that reflects the data you are trying to optimize very well. Also note that—if you go back to the simplest model with no hidden layers—if you provide access to the $X_1 \times X_2$ feature directly, the model does just fine.



Keras

Using Google Colab, you will explore an example of how to create a neural network in the Keras library. First, ensure that your runtime type hardware accelerator is set to GPU. If set to None, your code will run much slower.

The code used in this example generates the following dataset:



It also includes a function that creates the decision boundaries as a plot.

To start, import the Keras library:

```
from tensorflow import keras
from tensorflow.keras import layers
```

Then create a neural network with three hidden layers, where the first layer has three neurons, the second layer has four, and the third has two:

```
model = keras.Sequential([
    layers.Dense(3, activation="relu"),
    layers.Dense(4, activation="relu"),
```

```
    layers.Dense(2, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Dense is the name for having connections between all neurons from one layer to the next. The ReLU activation function is chosen as opposed to the hyperbolic tangent. It is a common choice in machine learning because it is very rapid to optimize and works well with gradient descent. At the outer layer, you have an activation function that is sigmoidal. That is because you want to produce a probability that is between zero and one.

Once you create this model, it is set up but the weights are not yet initialized. Keras models require you to specify the optimization function, the loss function, and the metrics to track as the model is trained:

```
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

Here, the chosen optimizer is rmsprop. There are other choices, but this one is very popular. For the loss function, since you are doing a binary classification problem, you use the binary cross-entropy. And lastly, you are going to keep track of the accuracy over time, which you can visualize later.

So you compile your model. And again, it is not trained yet. It knows its architecture and the properties it has as it trains itself, but it does not know anything about the world.

Next, you train your model on the data you provided:

```
np.random.seed(12)
history = model.fit(x = df[["x1", "x2"]],
                    y = df["y"],
                    epochs=5,
                    batch_size=8)
```

So given these x and y values, you are telling it to do five rounds of stochastic gradient descent, using a batch size of eight.

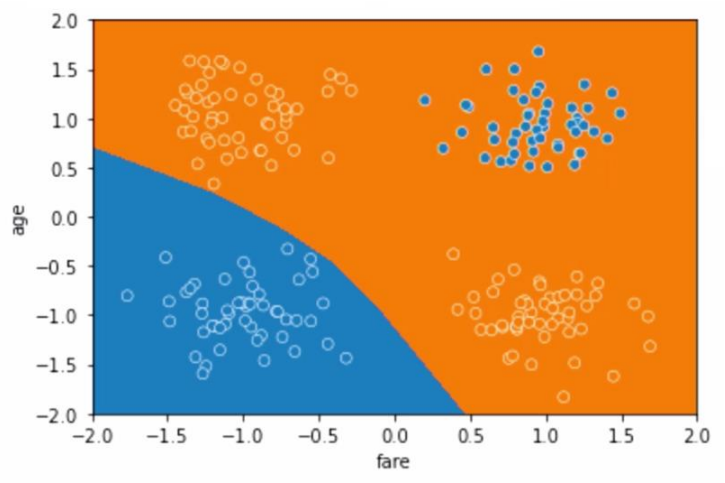
Once you do that, you can see the accuracy evolving over time:

```
Epoch 1/5
25/25 [==============================] 1s 3ms/step - loss: 0.6202
- accuracy: 0.7000
Epoch 2/5
25/25 [==============================] 0s 3ms/step - loss: 0.6053
- accuracy: 0.6800
Epoch 3/5
25/25 [==============================] 0s 3ms/step - loss: 0.5930
- accuracy: 0.6750
```

In this case, even after the first epoch, it has gotten about as good accuracy as it ever will. To see what decision boundary the model has come up with to achieve around 70% accuracy, you run the **visualize_decision_boundaries** function:

```
visualize_decision_boundaries(model, [-2, 2], [-21 2])
sns.scatterplot(data = df, x = "x1", y = "x2", hue = "y", legend = False)
```

Here is the model this network has learned:



It is not a particularly good model because it totally fails to classify the blue data points in the top-right section of the scatterplot.

You can also create a more complex model. For example, this model uses the exact same code but has three layers, with 16 neurons each:

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

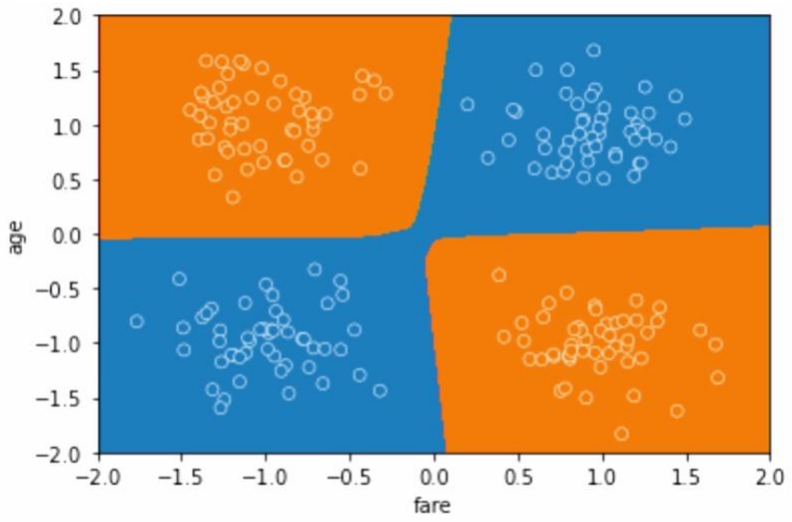
Training this model requires more epochs because it has a more complex network:

```
Epoch 1/20
25/25 [==============================] 1s 3ms/step - loss: 0.6526
- accuracy: 0.9400
Epoch 2/20
25/25 [==============================] 0s 3ms/step - loss: 0.5813
```

- accuracy: 0.9900
Epoch 3/20
25/25 [==============================] 0s 3ms/step - loss: 0.4918
- accuracy: 1.0000

Notice that after the first training epoch, it only has 94% accuracy, but then it is able to get up to 99%. The loss decreases monotonically, but the accuracy bounces around a little bit. You can have a higher cross-entropy loss and a higher accuracy as well. However, the long-term trend is that when the loss goes really low, the accuracy is very good.

If you look at this model, the decision boundary it learned has an extremely low loss and 100% accuracy, as you would expect:



So the glory of the Keras library is that it makes it very simple in code to create arbitrarily complicated neural networks.

Multiclass Classification

So far, you have observed neural networks to solve problems with two classes. Now you will build neural networks for multiclass classification problems.

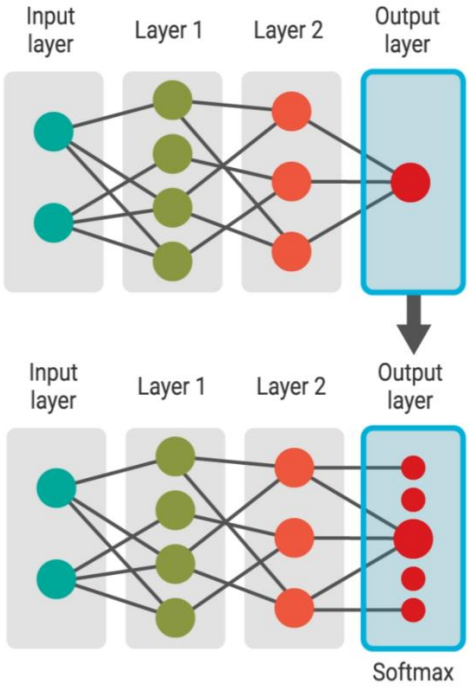
Recall the three methods for doing this in logistic regression:

- One versus rest
- One versus one
- Multinomial logistic regression

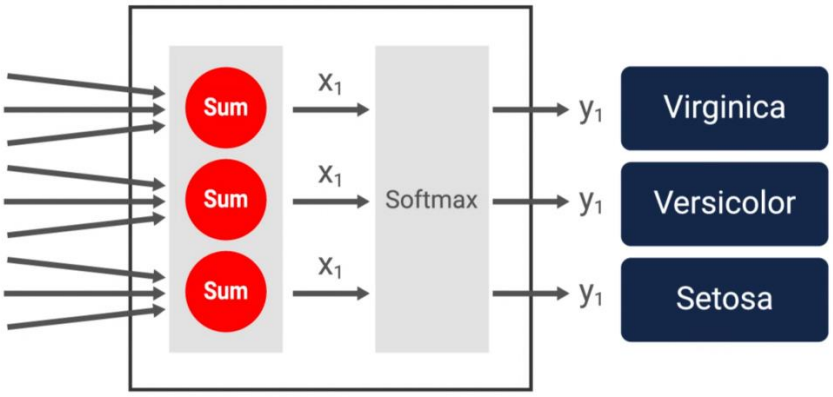
The one versus rest and one versus one methods are generic. And they can be applied to classification models other than logistic regression, including neural networks. They often work well, but both require that you train a set of classification models instead of just one, and then subject them to a majority or soft vote.

Neural networks are often very expensive to train. And so having to create multiple networks can be a drawback. The more common approach to multiclass neural networks is closer to multinomial logistic regression in that it requires only one model, and that model is adapted to produce probabilities for each of the output classes.

Adapting a neural network to generate class probabilities is very simple. All you need to do is to replace the single-node output layer with a multi-node output layer called a softmax layer:



The softmax layer has one output node for each class. So if you are working with the Iris dataset, you will have three output nodes, one for virginica, one for versicolor, and one for setosa:



The softmax layer proceeds in two steps. First, it sums the inputs to produce numbers x_1 through x_m for each of the m classes, in this case three. The model is trained so that the largest of the x's corresponds to the predicted class for the given input sample.

The next stage normalizes the predictions using the softmax formula, which is a generalization of the logistic or sigmoid function to multiple classes:

$$y^1 = \frac{e^{x1}}{e^{x1} + e^{x2} + e^{x3}}$$

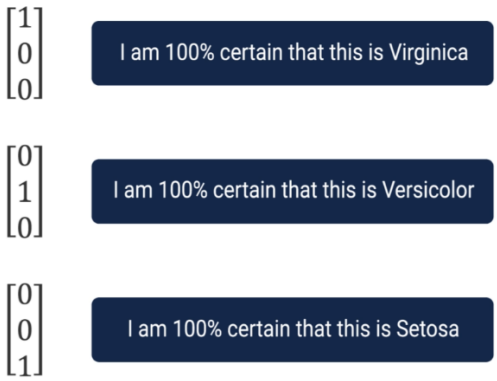
$$y^2 = \frac{e^{x2}}{e^{x1} + e^{x2} + e^{x3}}$$

$$y^3 = \frac{e^{x3}}{e^{x1} + e^{x2} + e^{x3}}$$

The inputs to the softmax function are the x_i's, which can take any value, positive or negative. Softmax then warps these inputs in a way that a) preserves their order so that if x_i is less than x_j, then y_i will be less than y_j, b) the y's are all between zero and one, and c) the y's add up to one.

The real importance of softmax has more to do with the training of the model than with prediction. As with any supervised learning algorithm, a neural network is trained by presenting it with a series of samples for which you are certain of the answer.

For the Iris dataset, you show the model the dimension of the petals and sepals, and say, this is a virginica, setosa or versicolor. You encode this statement with a vector; each entry is the certainty value for that class:



In this case, a 1 for virginica and 0s for setosa and versicolor. Other samples would be represented similarly with a 010 vector if they were versicolor, or a 001 if they were setosa. This way of encoding multiclass variables is called **one-hot encoding**. The softmax function fits perfectly with one-hot encoding because it produces values between zero and one that can also be interpreted as a certainty value for each of the classes.

Notice that this is not possible if you eliminate the softmax function from the output layer and instead train the model using the raw x-values. In that case, you have to encode your output labels with numbers ranging from negative to positive infinity. And it is impossible to express total certainty that a flower is, say, a virginica Iris.

Using softmax also gives the predictions of the neural network a probabilistic interpretation. If your network predicts values of 0.8, 0.05, and 0.15, you can interpret these as the model saying, I am 80% certain that this is a virginica Iris, but also give 5% chance to versicolor, and 15% chance to setosa. But be careful not to take these outputs as the true probabilities about the real world.

Remember that neural networks are difficult to train and often end up in a bad local minimum. So a particular network may not be representative of the true distributions of Iris flowers.

You can try this out using the Petal_length and Petal_width columns of the Iris dataset:

Iris Dataset			
	Petal_length	**Petal_width**	**Species**
0	1.4	0.2	Setosa
1	1.4	0.2	Setosa
2	1.3	0.2	Setosa
...	...	...	...
149	5.1	1.8	virginica

The first step is to one-hot encode the class labels with scikit-learn's **LabelBinarizer** class:

```
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
y_ohe = lb.fit_transform(y)
```

To create an encoder, you first call its constructor and then pass the labels into its **fit_transform** method. Internally, the LabelBinarizer builds and

stores a map between the string labels and a set of one-hot encoded arrays. And it returns the arrays corresponding to the labels you gave it. Here is the one-hot encoded array corresponding to the first ten flowers in the dataset, which are all setosas:

```
y_ohe[:10,:]
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0]])
```

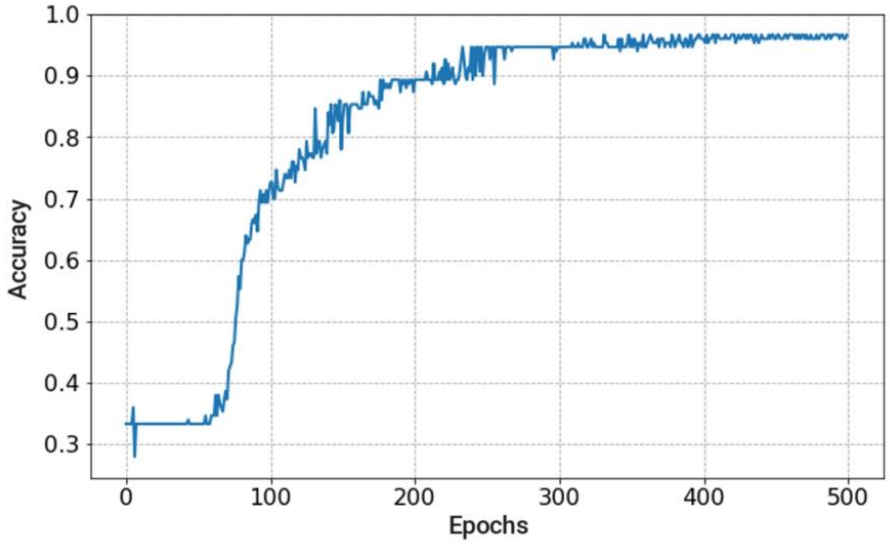
You can also go in the opposite direction from one-hot encoded labels to string labels by using the **inverse_transform** function of the LabelBinarizer. Here you confirm that the first ten entries are setosa Irises:

```
lb.inverse_transform(y_ohe[:10,:])
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
       'setosa', 'setosa', 'setosa', 'setosa'], dtype='<U10')
```

And here, you build a simple neural network with two layers, an internal dense layer with five ReLU units, followed by a softmax output layer with three units for the three output classes:

```
model = Sequential([
        Dense(5, input_dim=2, activation='relu',
        Dense(3, activation="softmax") ])

model.compile(loss="categorical_crossentropy", metrics=["accuracy"])
history = model.fit(X, y_ohe, epochs=500)
```

You then compile and train the model using the one-hot encoded labels for the output. Here, you see that the accuracy reaches about 96% after 200 epochs of training:



Think of softmax as a multinomial logistic regression embedded within a neural network. It is a simple way of adapting the model to produce multiclass predictions. And it is trained along with the rest of the network with stochastic gradient descent.

Two Bits of Syntax

The first Keras feature you will explore next is **integer encoding**, an alternate technique for representing outputs. Instead of one-hot encoding

the outputs, this code assigns each class an integer: zero for setosa, one for versicolor, two for virginica.

```
y_numerical = iris["species"]
y_numerical = y_numerical.replace("setosa", 0)
y_numerical = y_numerical.replace("versicolor", 1)
y_numerical = y_numerical.replace("virginica", 2)
```

Make sure you understand the difference between this integer encoding and the one-hot encoding approach.

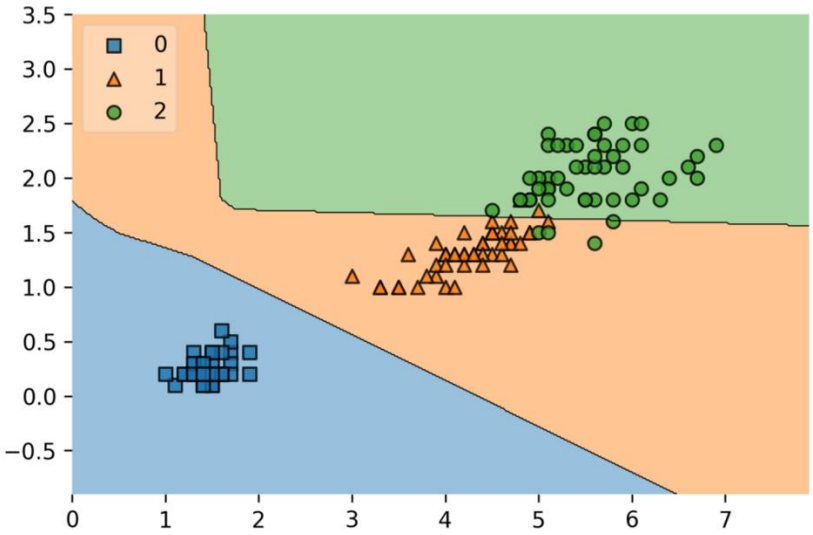
Training a Keras model with this integer encoding is very similar to what you did before. The only difference is that you must select a different loss function, **sparse_categorical_crossentropy**:

```
model_using_numerical_encoding = keras.Sequential([
   layers.Dense(5, activation="relu"),
   layers.Dense(3, activation="softmax")
])

model_using_numerical_encoding.compile(optimizer="rmsprop",
          loss="sparse_categorical_crossentropy",
          metrics=["accuracy"])

history = model_using_numerical_encoding.fit(X,
               y_numerical,
               epochs=500)
```

The choice of these two loss functions makes no difference, in terms of model performance. You simply need to pick the appropriate loss function

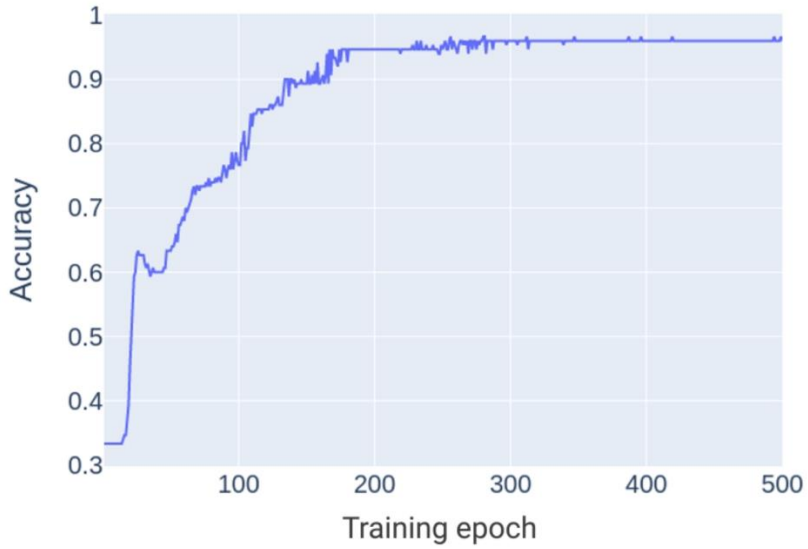
to match the encoding for the observations you are trying to predict: **Categorical_crossentropy** for one-hot encoding and **sparse_categorical_crossentropy** for integer encoding.

The second Keras feature to explore is the ability to plot the loss instead of the accuracy. Here is a neural network with one hidden layer containing five neurons for the Iris dataset:



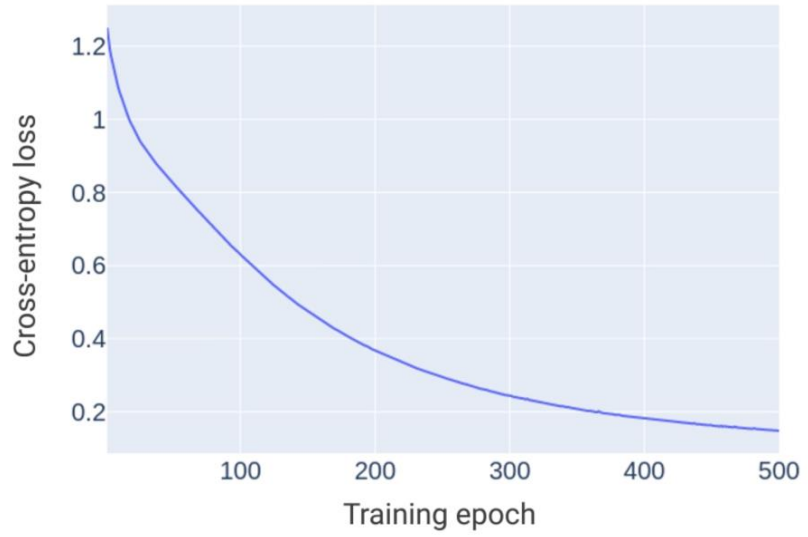
To plot the accuracy for this model, you use the code:

px.line(y = history_using_numerical_encoding.history["accuracy"])



And to plot the loss for this model, you use the code:

px.line(y = history_using_numerical_encoding.history["loss"])



Note that even after 500 epochs, the model is still not entirely stable and is eking out just a bit better loss with each epoch. Also note you are using the entire dataset and have not set aside a separate testing set. In other words, if the model were overfitting, you would not see that here.

Hyperparameter Tuning

Neural networks have a huge number of hyperparameters. These include, for example:

- The number of layers in the network, also known as the depth
- The number of neurons in each layer, the width of each layer
- The activation function for each neuron

In the original working example, the network had a depth of two. There was one hidden layer with a width of five neurons and an output layer. You also know that the hidden layer used ReLU units, and the output layer used softmax:

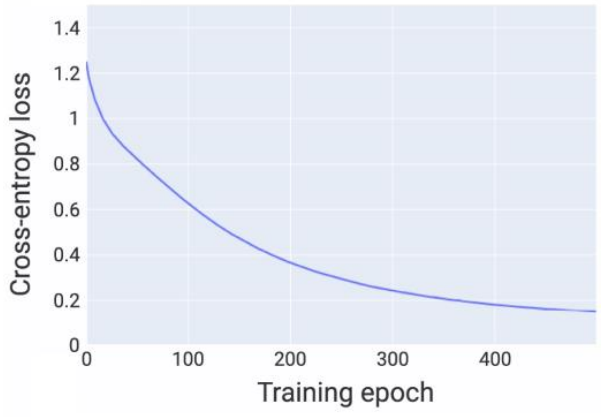
```
model_using_numerical_encoding = keras.Sequential([
        layers.Dense(5, activation="relu"),
        layers.Dense(3, activation="softmax")
])
```

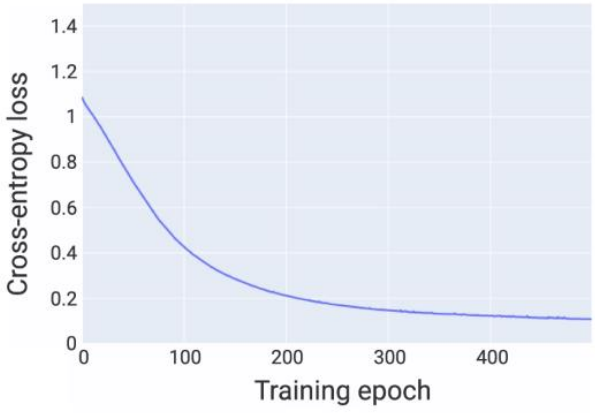
But you could have done something completely different. For example, you could have had 30 layers with 100 neurons each. Or 100 layers, where the first layer has 1 neuron, the second has 2, the third has 3, and so forth. So given this infinitude of possibilities, how do you know which to pick?

Well, see what happens if you tweak the network architecture. Suppose you add another layer of five neurons. You simply add another line to your sequential model definition:

```
model_using_numerical_encoding = keras.Sequential([
    layers.Dense(5, activation="relu"),
    layers.Dense(5, activation="relu"),
    layers.Dense(3, activation="softmax")
])
```

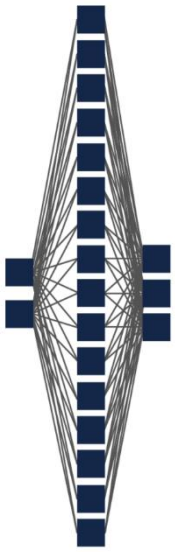
Here, the second hidden layer has five neurons. But it could have been any other number. The number of neurons does not need to match between the first and second hidden layers. If you compare the loss versus the training epoch, you see that the new two-layered architecture seems to yield slightly better results with fewer training epochs:

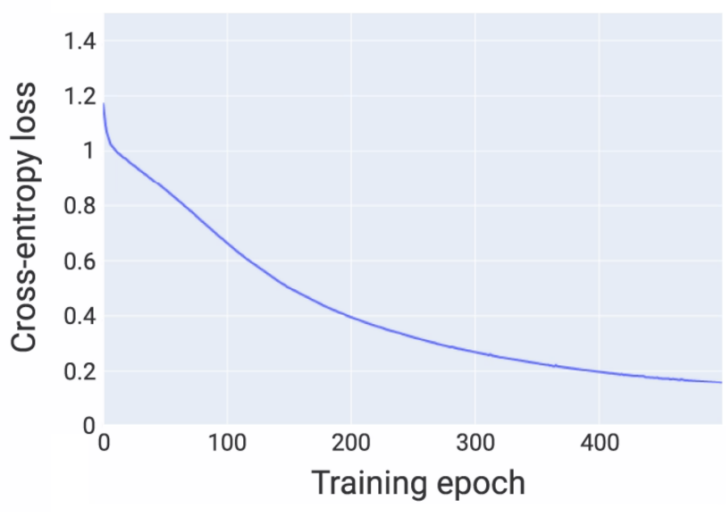




Suppose that, rather than having two hidden layers of 5 neurons, you have just one hidden layer of 16 neurons:

```
model_using_numerical_encoding = keras.Sequential([
        layers.Dense(16, activation="relu"),
        layers.Dense(3, activation="softmax")
])
```



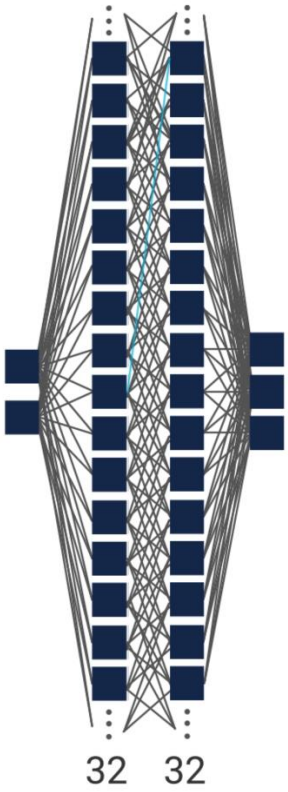


This model's loss-versus-training curve is similar to the original model with only five neurons in the hidden layer. In other words, the extra expressive power of the other 11 neurons in the hidden layer, didn't help much.

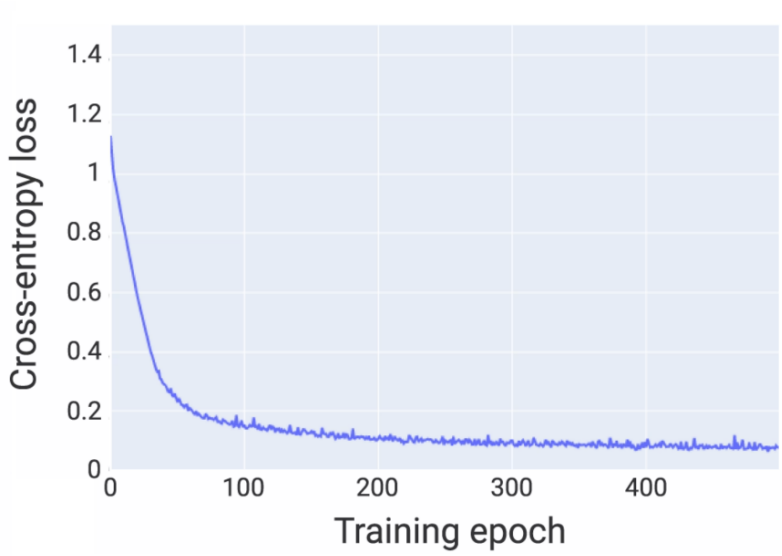
Keras lets you easily create a model that has a large number of parameters. For example, this code is for a model with three hidden layers of 32 neurons each:

```
model_using_numerical_encoding = keras.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32, activation="relu"),
    layers.Dense(32, activation="relu"),
    layers.Dense(3, activation="softmax")
])
```

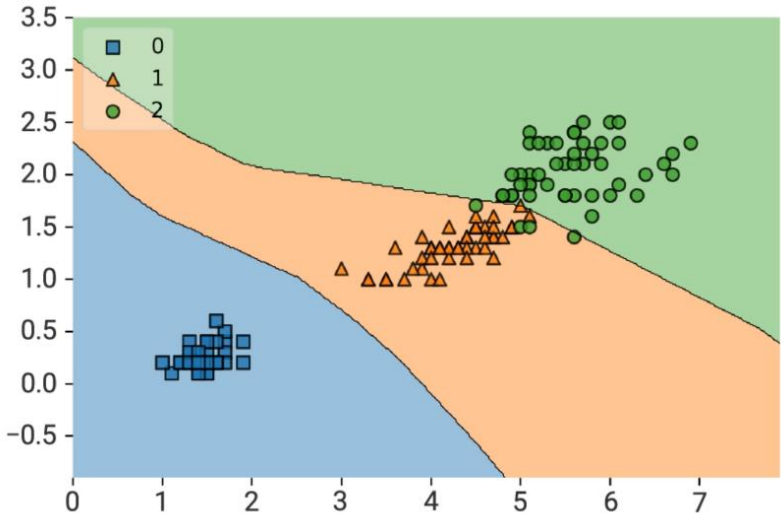
To roughly estimate the number of parameters in this model, consider that each connecting line in the network has its own weight, and each neuron has its own intercept. So the connections between only two of the 32 neuron layers, need 32 squared parameters, or 1,024.



So this model has several thousand parameters and you are only going to be training it on 150 data points. You can see that this model converges in many fewer epochs than the earlier simpler models, though with more noise in the loss between the training epochs:



If you visualize this new model's decision boundaries, you see that it does not appear to be obviously overfit:



So which hyperparameters do you select? Do you want the simple model with just one hidden layer of five neurons for a total of a few dozen parameters? Or at the other extreme, do you want this new complex model with three hidden layers of 32 neurons each, totaling many thousands of parameters? Well, you have already used the most important tool for deciding this question: **Cross-validation**.

Given the high cost to train neural networks, it is much more common to use simple cross-validation rather than k-fold cross-validation. In other words, you typically hold aside a validation set and use that to evaluate the generalizability of your model. For this specific example, you take a random sample of, say, 40 of the 150 flowers and set those aside as a validation or development set:

train, dev = np.split(iris, 150 – 110 = 40 [110])

You then train a bunch of models and see which yields the best loss on the validation set.

So for example, the code shown computes the training and dev set losses versus the epoch for the network with one hidden layer of five neurons:

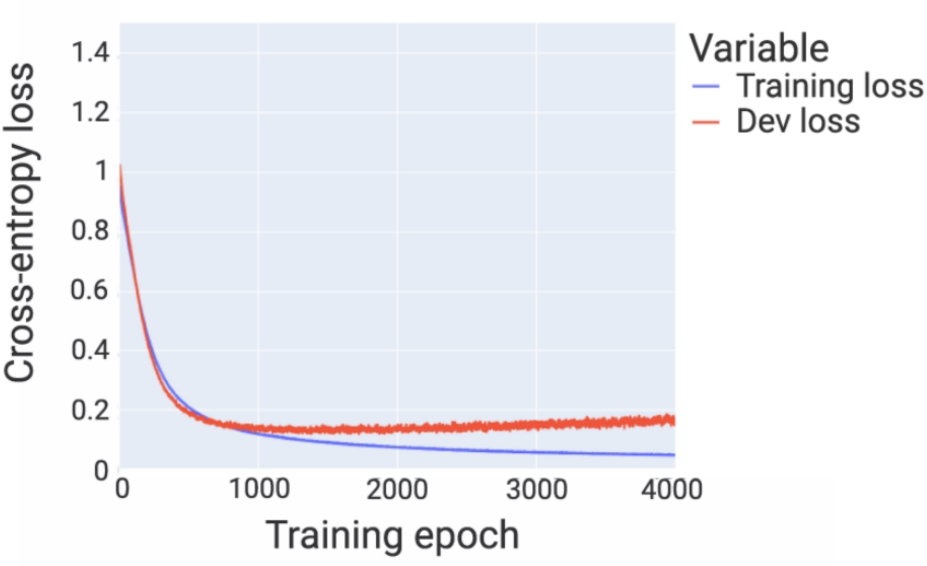
```
network_5.compile(optimizer="rmsprop",
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"])

history_network_5 = network_5.fit(x_train,
            y_train,
            epochs=500,
            verbose = 0,
            validation_data=(x_dev, y_dev))
```

The two main changes in the code are that, first, you shuffle the dataset and then use **np.split** to get a training and dev set:

```
from sklearn.utils import shuffle
iris = sns.load_dataset("iris")
train, dev = np.split(iris, [110])
```

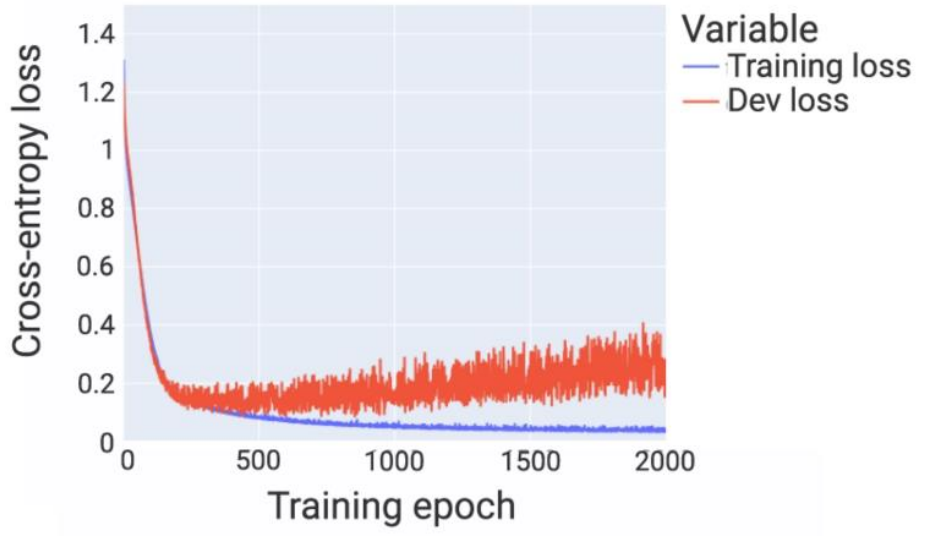
And then, when you call the fit method of your neural network, you provide the validation set so it can keep track of the validation loss over time.

Plotting these two losses together yields the figure shown:



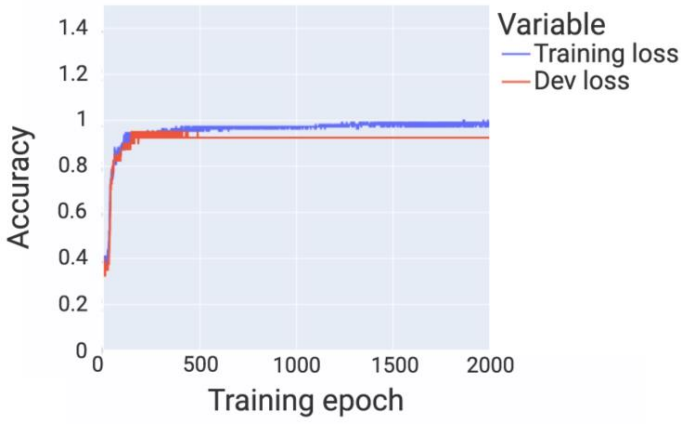
The development set loss reaches its minimum somewhere around 800 epochs in and so it would make sense to stop training at this point to avoid overfitting. The dev loss bottoms out at around 0.15, though that number is fairly noisy.

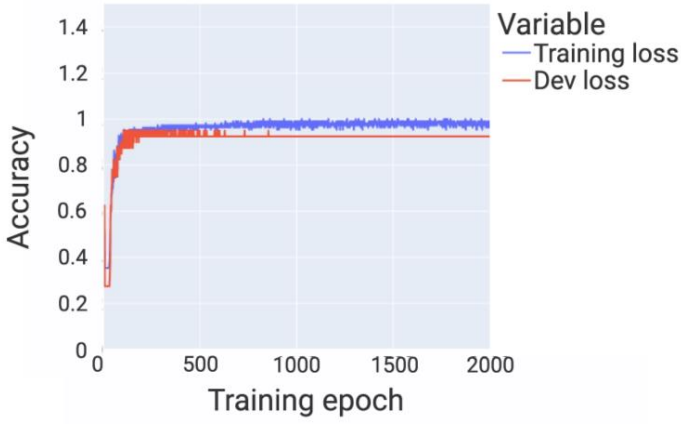
Now by comparison, here the cross-entropy loss for a model with two layers of 16 neurons each is plotted:



Again, the validation set loss decreases to some minimum. But this time, somewhere around 200 epochs in, before increasing back up due to overfitting. For this network, the validation set loss is much noisier. Standardizing your data, using a standard scalar, will reduce this noise and make your models converge more quickly. This model also bottoms out at an error of around 0.15. However, the noisiness of the dev set loss indicates that this model is less likely to generalize well. If you had to pick between these two models, you should choose the simpler one that has just five neurons and a single hidden layer.

Note that if you look at accuracy rather than loss, as shown in these two figures, the models are essentially indistinguishable:





Both level off at 92.5% accuracy so there is no reason to choose one over the other based on the accuracy metric.

In principle, you can repeat that same process for a large number of candidate model architectures. By generating a table of validation set loss values for each of those network architectures, you can choose the one that works best. In other words, the one that has the lowest development set cross-entropy loss. Keep in mind that the resulting value is a bit biased in its estimate of model performance. So you need a totally separate one-time

use test set if you want to give a final, quantitative assessment of the model's loss or quality. You can also, in principle, generate curves showing the development set loss versus the depth of the network. Or the development set loss versus the width of one or more layers of the network.

Note that rather than manually generating a table of validation set loss values, Keras has a tool called KerasTuner that allows you to search a pre-specified set of hyperparameters.

KerasTuner works similarly to GridSearchCV in scikit-learn.

Mini-Lesson: Popular Hyperparameters

With neural networks, there are many hyperparameters, which can be challenging to tune manually. Therefore, KerasTuner is used to make it easy to tune the hyperparameters of neural networks. This is similar to what you have seen in machine learning when using grid search or random search.

Using the KerasTuner, you can choose the right set of parameters for your TensorFlow programs. This is known as hyperparameter tuning or hyper tuning of your machine learning application to select the correct hyperparameters.

In machine learning, hyperparameters determine the model's training process and topology. During the training process, these variables remain constant and directly affect the performance of your ML program. There are two types of hyperparameters:

- Model hyperparameters that affect the model selection, such as the number and width of hidden layers

- Algorithm hyperparameters that affect the quality and speed of the learning algorithm, such as the number of neighbors for a K-nearest neighbors (KNN) classifier and the learning rate for stochastic gradient descent (SGD)

This mini-lesson aims to cover some of the more popular hyperparameters. They include the dropout rate and the regularization parameter.

Dropout Rate

The technique of dropout involves ignoring randomly selected neurons during training. Instead, they are randomly 'dropped out.' The result is that their contribution to downstream activation is removed in the forward propagation, and their weights are not updated in the backpropagation.

Regularization Rate

The term 'regularization' refers to a number of techniques that reduce the complexity of a neural network model during training to reduce overfitting. Two prevalent and efficient regularization techniques are L1 and L2.

L1 regularization penalizes the absolute magnitude of the coefficients. To put it differently, it limits the coefficient size. As a result, L1 can produce sparse models (with few coefficients). Some coefficients can become zero and can be eliminated. Lasso regression uses this technique.

In **L2 regularization**, the square of the magnitude of the coefficients is added to the penalty. As a result, the L2 model does not yield sparse models, and all coefficients are shrunk by the same factor (none are eliminated). Regression and support vector machines use this technique.

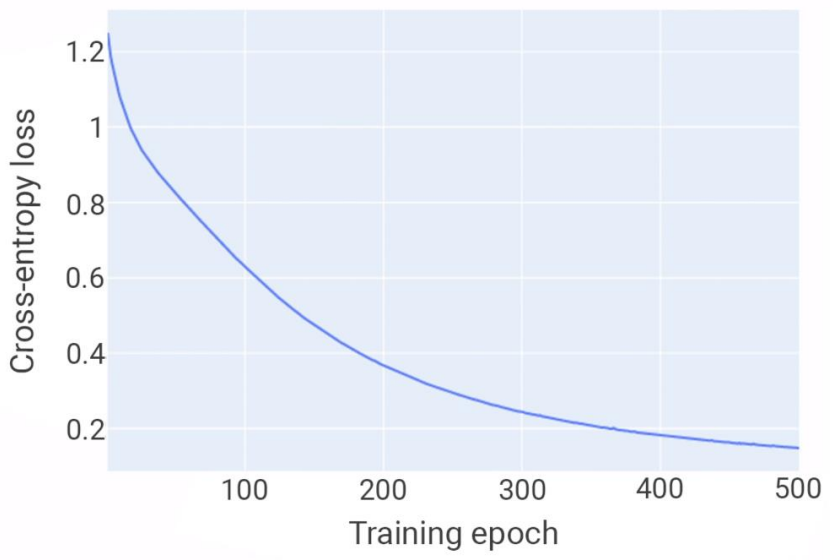
Computing Batch Gradients

Looking at the loss after every mini-batch, rather than after every epoch, helps you understand stochastic gradient descent more deeply.

Keras uses stochastic gradient descent. That means, rather than computing the true gradient on the entire dataset, the algorithm only considers a **mini-batch** at any given time, yielding an approximation of the gradient. If you do not specify the batch size, Keras uses the default of 32. In other words, every time it revises the neural network parameters, it only uses 32 of the 150 flower observations to decide how to revise them.

The loss tracked by the Keras history only gives the error after each training epoch has completed. An epoch consists of a consideration of all data points. Since you had a 150 data points and 32 observations per batch, that means there are five mini-batches per epoch.

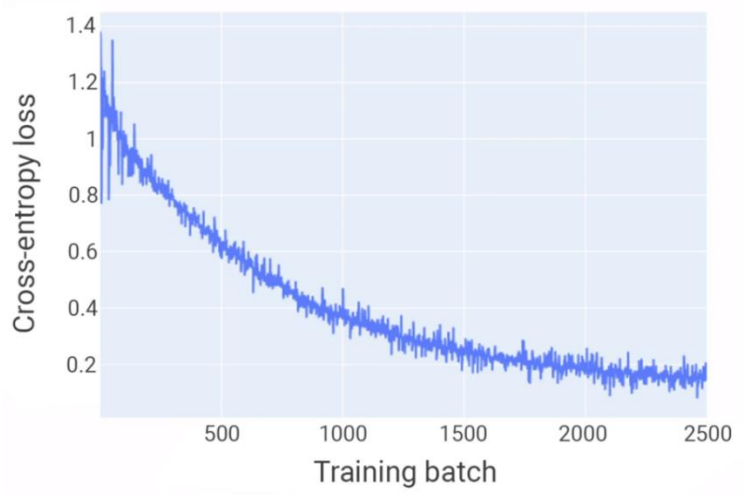
So in a single epoch, the following happens. The stochastic gradient descent algorithm computes the gradient of the cross-entropy loss using 32 data points, say, 0 through 31. After taking a step down, using that approximation of the gradient, it computes another approximation of the gradient on data points 32 through 63. Again, it takes a step down in the direction of the gradient. Then it computes the gradient on data points 64 through 95 and takes another step down. Next, it computes the gradient on data points 96 through 127 and takes another step down. As the last step in the batch, it computes the gradient on data points 128 through 149 and takes one more step down. Only then does Keras record the loss, which yields the cross-entropy plot here:



For practical problems, each epoch of stochastic gradient descent should consider a **random** ordering of the points. So it does not work very well if you do it as described, where the first batch is the first 32 points, and so forth.

You can define a custom callback to track the loss after every batch, yielding the array of values called **batch losses**. The result of the code for this is given here:

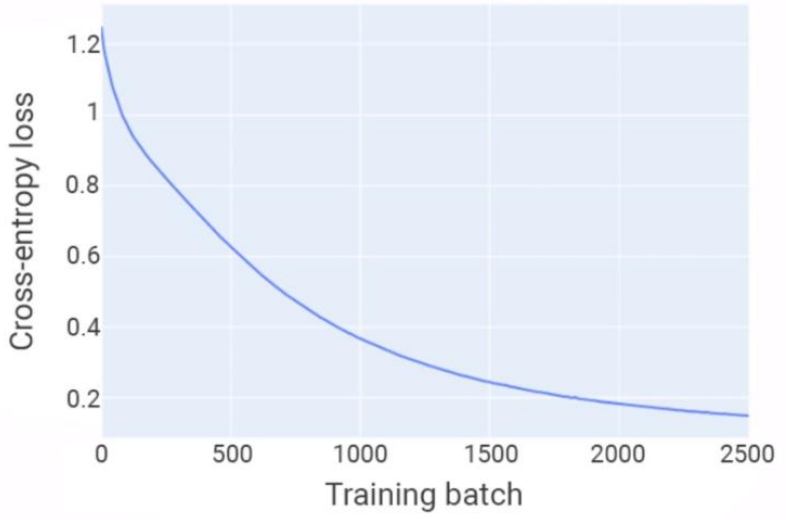
px.line(y=batch_losses)



Here you see the cross-entropy loss after each batch. Since you had 500 epochs, each consisting of five batches, there are 2,500 total training batches on the x-axis. The descent towards lower cross-entropy loss is much noisier.

Note that the epoch loss is the same as every fifth sample of the batch losses. So if you use the code below, you get the exact same figure as when you tracked the loss after every epoch:

```
fig = px.line(x = np.linspace(0, 2500, 500),
              y = batch_losses[4::5])
```

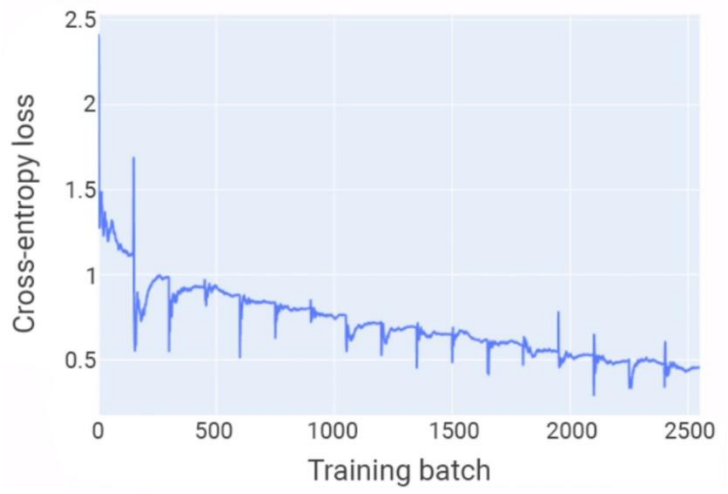


In other words, if you look at the loss after every fifth batch, you are looking at the loss after every entire training epoch, which gives you the smooth curve from before.

By contrast, see what happens if you set the batch size to one:

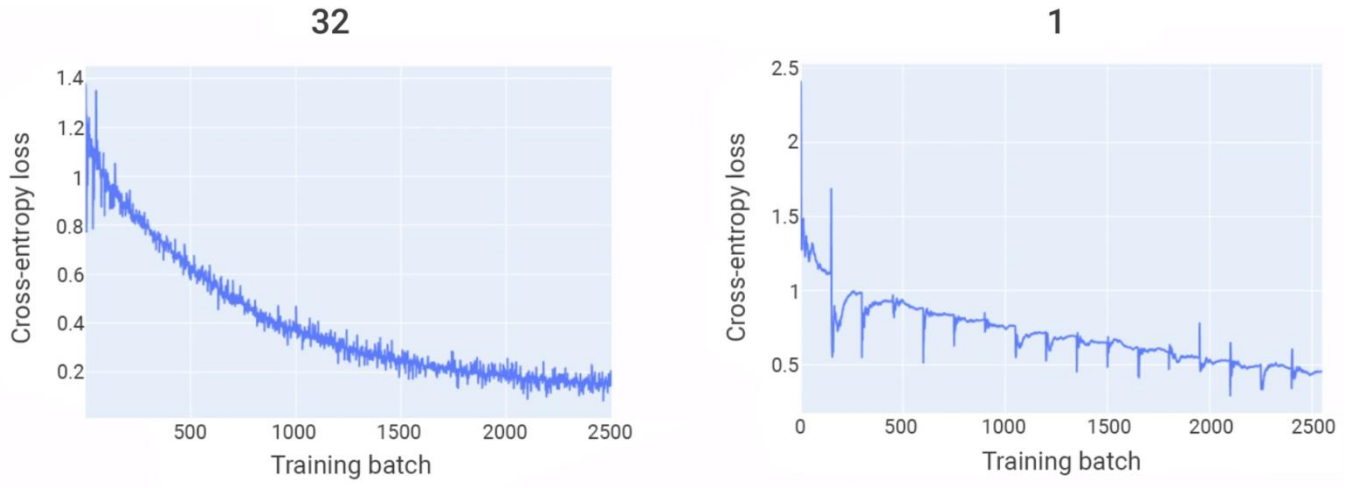
```
history = model_using_numerical_encoding.fit(X,
              y_numerical,
```

batch_size=1,
epochs=17)



If you do this, the loss of your model is much noisier after each batch. That is not surprising since each gradient descent step is now based on only a single data point.

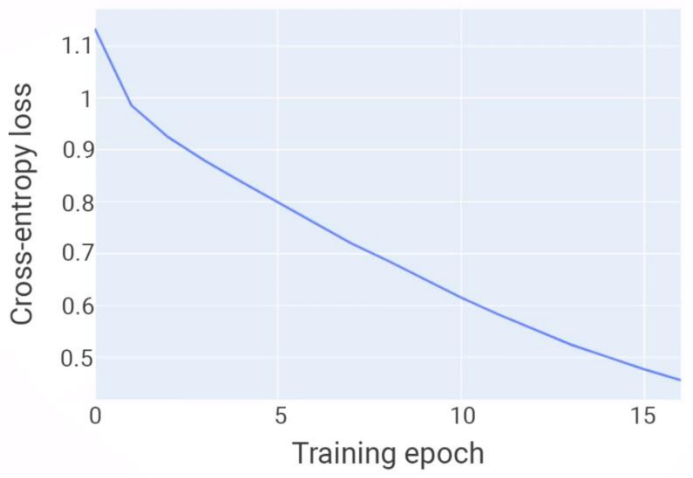
Here, you see the cross-entropy loss versus the batch number for the two choices of batch size, 32 and 1:



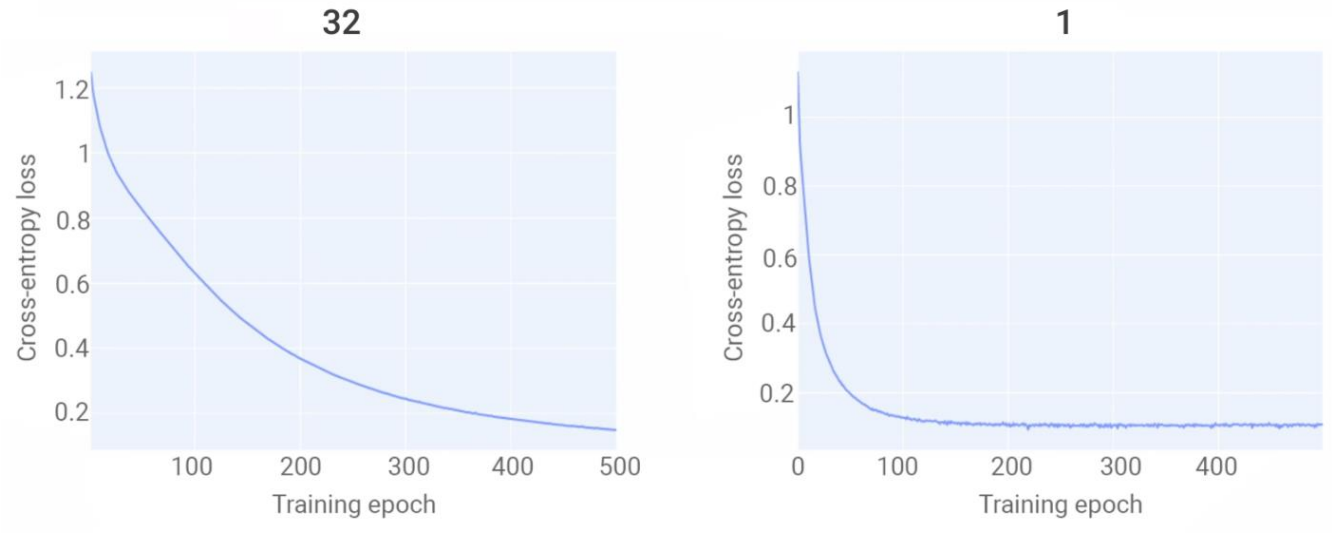
You see that after 2,500 gradient descent steps, the model being trained on only one sample at a time has not made as much progress as the version trained on 32 samples at a time. This is because the quality of the gradient approximation for the one-sample-at-a-time approach is slower.

If you track the time needed to train these models by each step of the process, you see that the gradient descent steps for a batch of size one are much faster. This is because computing the gradient with one data point is faster than using 32 data points. So the batch size controls the tradeoff between the quality of the gradient approximation, and the time needed to compute it.

This brings you to a counterintuitive phenomenon. If you plot the cross-entropy loss versus the epoch number, the trajectory is less noisy than before:



But if you compare the cross-entropy loss versus epoch number for a batch size of 32 versus one, you see that convergence seemed to happen much faster with a batch size of one:



How is the lower-quality approximation of the gradient yielding faster convergence against the epoch number? The answer is that for batch size 1, there are actually 150 different gradient descent steps per epoch. But by contrast, for a batch size of 32, there are only 5 gradient descent steps per epoch. So ultimately that comparison was not fair.