A. What is training?

In Chapter 3, we discussed the weights associated with connections between neurons. These weights determine what a neural network outputs based on the input data. However, these weights are what we call *trainable variables*, meaning that we need to train our neural network to find the optimal weights for each connection.

For any neural network, training involves setting up a *loss function*. The loss function tells us how bad the neural network's output is compared to the actual labels.

Since a larger loss means a worse model, we want to train the model to output values that minimize the loss function. The model does this by *learning* the optimal weight settings. Remember, the weights are just real numbers, so the model is essentially just figuring out the best numbers to set the weights to.

B. Loss as error

In regression problems, common loss functions are the L1 norm:

$$\sum_i |actual_i - predicted_i|$$

and the L2 norm:

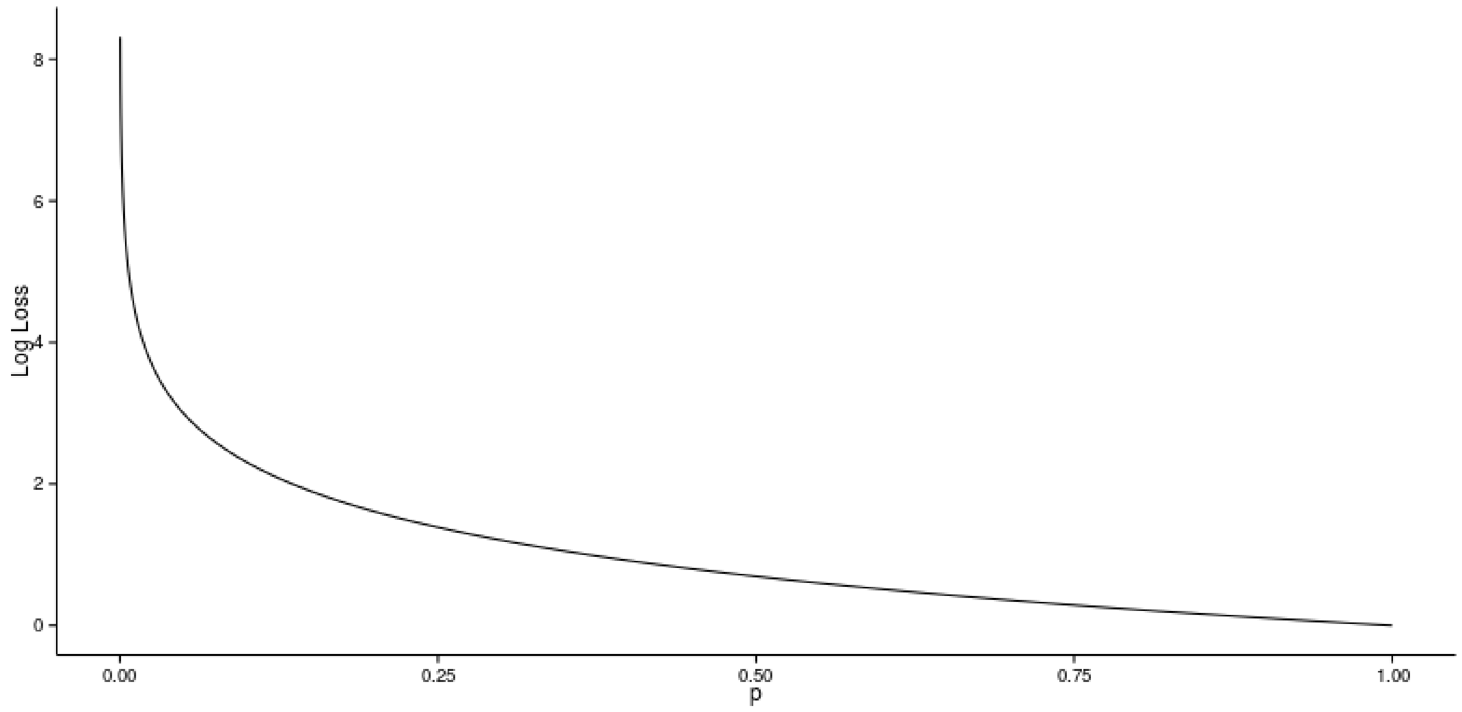
$$\sum_i (actual_i - predicted_i)^2$$

These provide an error metric for how far the predictions are from the labels, so the goal is to minimize the prediction error by minimizing the L1 and L2 norm.

In classification problems there's no good error measurement between predictions and labels, since the labels are discrete values. For example, in regression if we predict a stock's price was $99 but the actual value was $100, our prediction is still really good even though it was incorrect. However, in classification a prediction is either right or wrong, without any sense of how close it is to the actual label.

C. Cross entropy

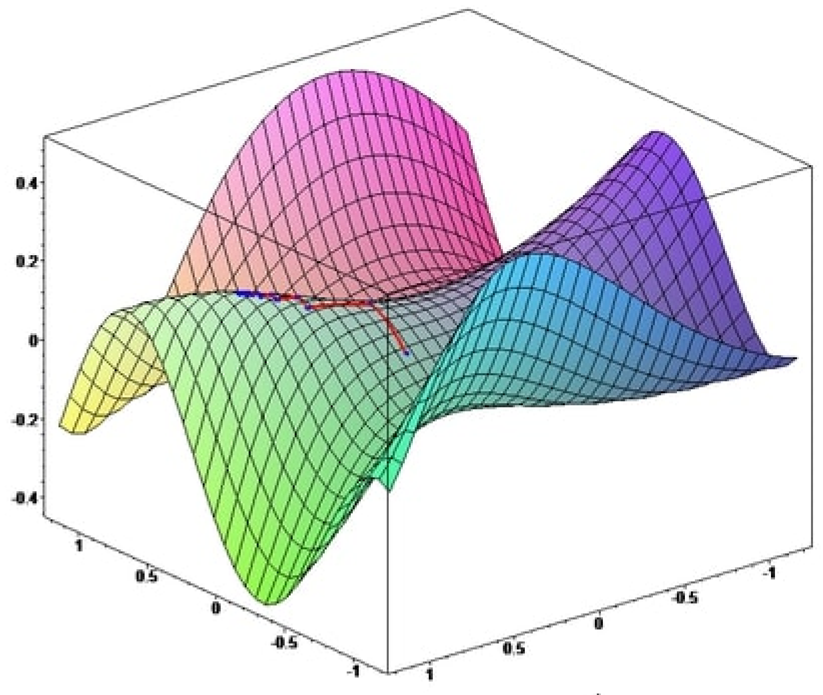
Rather than defining error as being right or wrong in our prediction, we can instead define it in terms of probability. Therefore, we want a loss function that is small when the probability is close to the label (i.e. a probability of 0.99 for a label of 1) and large when the probability is far from the label (i.e. a probability of 0.99 for a label of 0). The loss function that achieves this is known as cross entropy, also referred to as *log loss*.



Cross entropy (log loss) for a label of 1. Thex-axis represents the probability and the y-axis represents the log loss.

D. Optimization

Now we can just minimize the cross entropy based on the model's logits and labels to get our optimal weights. We do this through gradient descent, where the model updates its weights based on a *gradient* of the loss function until it reaches the minimum loss (at which point the weights converge to the optimum). We use backpropagation to find the optimal gradient for the model to follow. Gradient descent is implemented as an object in TensorFlow, called `tf.train.GradientDescentOptimizer`.



In the above graph, the colored shape represents values of the loss function, and the *x* and *y* axes represent weight values in the model. The model follows a gradient (red line) towards the minimum of the loss function.

The size of the gradient depends on something called the *learning rate*. A larger learning rate means the model could potentially reach the minimum loss quicker, but could also overshoot the minimum. Smaller learning rates are more likely to reach the minimum, but may take longer. Usually we test out learning rates between 0.001 to 0.1 to find the best one for model training. You can set the learning rate via the `learning_rate` argument when initializing a TensorFlow `Optimizer` (e.g. `GradientDescentOptimizer`).

Regular gradient descent has trouble minimizing complex loss functions, so we usually use better optimization methods for training. A popular and effective optimization method is Adam, which is implemented in TensorFlow as `tf.train.AdamOptimizer`. It has default values already set for its parameters (e.g. `learning_rate`), so in our code we initialize the object with no arguments.