**15. Intro to Deep Learning**

**15.1 A Single Neuron**

Using Keras and Tensorflow you'll learn how to:

* create a **fully-connected** neural network architecture
* apply neural nets to two classic ML problems: **regression** and **classification**
* train neural nets with **stochastic gradient descent**, and
* improve performance with **dropout**, **batch normalization**, and other techniques

**What is Deep Learning?**

Some of the most impressive advances in artificial intelligence in recent years have been in the field of *deep learning*. Natural language translation, image recognition, and game playing are all tasks where deep learning models have neared or even exceeded human-level performance.

So what is deep learning? **Deep learning** is an approach to machine learning characterized by deep stacks of computations. This depth of computation is what has enabled deep learning models to disentangle the kinds of complex and hierarchical patterns found in the most challenging real-world datasets.

Through their power and scalability **neural networks** have become the defining model of deep learning. Neural networks are composed of neurons, where each neuron individually performs only a simple computation. The power of a neural network comes instead from the complexity of the connections these neurons can form.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**15.2 Deep Neural Networks**

In this lesson we're going to see how we can build neural networks capable of learning the complex kinds of relationships deep neural nets are famous for.

The key idea here is *modularity*, building up a complex network from simpler functional units. We've seen how a linear unit computes a linear function -- now we'll see how to combine and modify these single units to model more complex relationships.

A screenshot of a computer

Description automatically generated

**Many Kinds of Layers**  
A "layer" in Keras is a very general kind of thing. A layer can be, essentially, any kind of *data transformation*. Many layers, like the [convolutional](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) and [recurrent](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RNN) layers, transform data through use of neurons and differ primarily in the pattern of connections they form. Others though are used for [feature engineering](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) or just [simple arithmetic](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Add). There's a whole world of layers to discover -- [check them out](https://www.tensorflow.org/api_docs/python/tf/keras/layers)!

A graph with red dots

Description automatically generated

A graph and diagram on a white background

Description automatically generatedReLU

A diagram of a network

Description automatically generated

**15.3 Stochastic Gradient Descent**

A graph on a white background

Description automatically generated

A white background with black text

Description automatically generated

A white paper with black text

Description automatically generated



A close up of a text

Description automatically generated

A screenshot of a web page

Description automatically generated

**15.4 Overfitting and Underfitting**

A graph on a white background

Description automatically generated

A screenshot of a graph

Description automatically generated

A white background with black text

Description automatically generated

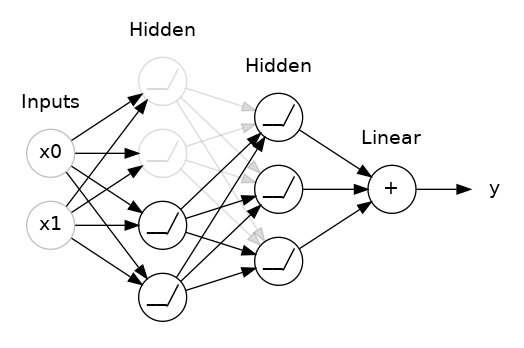
A diagram of a graph

Description automatically generated with medium confidence

**15.5 Dropout and Batch Normalization**

A close-up of a text

Description automatically generated



A close-up of a white background

Description automatically generated

**Batch Normalization**

The next special layer we'll look at performs "batch normalization" (or "batchnorm"), which can help correct training that is slow or unstable.

With neural networks, it's generally a good idea to put all of your data on a common scale, perhaps with something like scikit-learn's [StandardScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) or [MinMaxScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html). The reason is that SGD will shift the network weights in proportion to how large an activation the data produces. Features that tend to produce activations of very different sizes can make for unstable training behavior.

Now, if it's good to normalize the data before it goes into the network, maybe also normalizing inside the network would be better! In fact, we have a special kind of layer that can do this, the **batch normalization layer**. A batch normalization layer looks at each batch as it comes in, first normalizing the batch with its own mean and standard deviation, and then also putting the data on a new scale with two trainable rescaling parameters. Batchnorm, in effect, performs a kind of coordinated rescaling of its inputs.

Most often, batchnorm is added as an aid to the optimization process (though it can sometimes also help prediction performance). Models with batchnorm tend to need fewer epochs to complete training. Moreover, batchnorm can also fix various problems that can cause the training to get "stuck". Consider adding batch normalization to your models, especially if you're having trouble during training.

**15.6 Binary Classification**

Classification into one of two classes is a common machine learning problem. You might want to predict whether or not a customer is likely to make a purchase, whether or not a credit card transaction was fraudulent, whether deep space signals show evidence of a new planet, or a medical test evidence of a disease. These are all **binary classification** problems.

In your raw data, the classes might be represented by strings like "Yes" and "No", or "Dog" and "Cat". Before using this data we'll assign a **class label**: one class will be 0 and the other will be 1. Assigning numeric labels puts the data in a form a neural network can use.

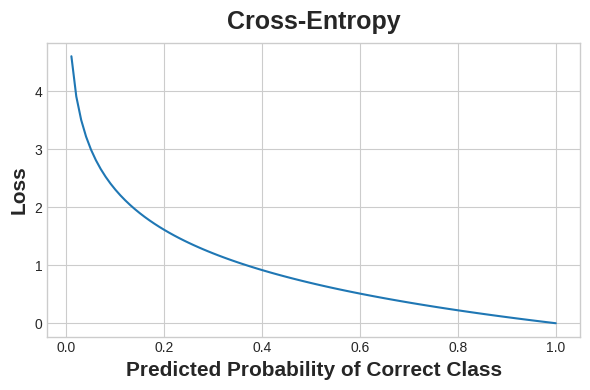
**Accuracy and Cross-Entropy**

**Accuracy** is one of the many metrics in use for measuring success on a classification problem. Accuracy is the ratio of correct predictions to total predictions: accuracy = number\_correct / total. A model that always predicted correctly would have an accuracy score of 1.0. All else being equal, accuracy is a reasonable metric to use whenever the classes in the dataset occur with about the same frequency.

The problem with accuracy (and most other classification metrics) is that it can't be used as a loss function. SGD needs a loss function that changes smoothly, but accuracy, being a ratio of counts, changes in "jumps". So, we have to choose a substitute to act as the loss function. This substitute is the *cross-entropy* function.

Now, recall that the loss function defines the *objective* of the network during training. With regression, our goal was to minimize the distance between the expected outcome and the predicted outcome. We chose MAE to measure this distance.

For classification, what we want instead is a distance between *probabilities*, and this is what cross-entropy provides. **Cross-entropy** is a sort of measure for the distance from one probability distribution to another.



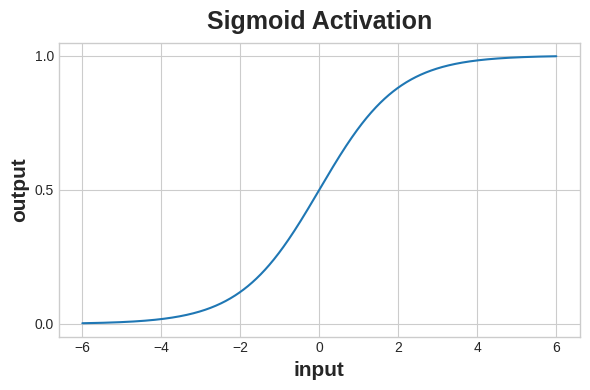
Cross-entropy penalizes incorrect probability predictions.

The idea is that we want our network to predict the correct class with probability 1.0. The further away the predicted probability is from 1.0, the greater will be the cross-entropy loss.

The technical reasons we use cross-entropy are a bit subtle, but the main thing to take away from this section is just this: use cross-entropy for a classification loss; other metrics you might care about (like accuracy) will tend to improve along with it.

**Making Probabilities with the Sigmoid Function**

The cross-entropy and accuracy functions both require probabilities as inputs, meaning, numbers from 0 to 1. To covert the real-valued outputs produced by a dense layer into probabilities, we attach a new kind of activation function, the **sigmoid activation**.



The sigmoid function maps real numbers into the interval [0,1]

.

To get the final class prediction, we define a *threshold* probability. Typically this will be 0.5, so that rounding will give us the correct class: below 0.5 means the class with label 0 and 0.5 or above means the class with label 1. A 0.5 threshold is what Keras uses by default with its [accuracy metric](https://www.tensorflow.org/api_docs/python/tf/keras/metrics/BinaryAccuracy).

Diagram of a binary classifier

A screenshot of a computer

Description automatically generated

A screen shot of a computer code

Description automatically generated