Neural Networks

Neural networks are a family of model architectures designed to find **nonlinear** patterns in data. During training of a neural network, the **model** automatically learns the optimal feature crosses to perform on the input data to minimize loss.

Nodes and Hidden Layers

To build a **neural network** that learns **nonlinearities**, begin with the following familiar model structure: a **linear model** of the form

$$y' = b + w_1x_1 + w_2x_2 + w_3x_3$$

Adding Layers to the Network

In neural network terminology, additional layers between the input layer and the output layer are called **hidden layers**, and the nodes in these layers are called **neurons**.

The value of each neuron in the hidden layer is calculated the same way as the output of a linear model: take the sum of the product of each of its inputs (the neurons in the previous network layer) and a unique weight parameter, plus the bias. Similarly, the neurons in the next layer (here, the output layer) are calculated using the hidden layer's neuron values as inputs

Activation Functions

A function that enables neural networks to learn nonlinear relationships between features and the label

Adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs. In brief, each layer is effectively learning a more complex, higher-level function over the raw inputs

Training using Backpropagation

Backpropagation is the most common training algorithm for neural networks. It makes gradient descent feasible for multi-layer neural networks. Many machine learning code libraries (such as Keras) handle backpropagation automatically, so you don't need to perform any of the underlying calculations yourself
/h3

Vanishing Gradients

The **gradients** for the lower neural network layers (those closer to the input layer) can become very small. In **deep networks** (networks with more than one hidden layer), computing these gradients can involve taking the product of many small terms.

When the gradient values approach 0 for the lower layers, the gradients are said to "vanish". Layers with vanishing gradients train very slowly, or not at all.

The ReLU activation function can help prevent vanishing gradients.

Exploding Gradients

If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

Batch normalization can help prevent exploding gradients, as can lowering the learning rate.

Dead ReLU Units

Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.

Lowering the learning rate can help keep ReLU units from dying.

Dropout Regularisation

Yet another form of regularization, called **dropout regularization**, is useful for neural networks. It works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularisation:

- 0.0 = No dropout regularisation.

- 1.0 = Drop out all nodes. The model learns nothing.
- Values between 0.0 and 1.0 = More useful.

Multi-Class Classification

Multi-Class classification models can pick up on multiple possibilities as opposed to Binary Classifiers

Some real-world multi-class problems entail choosing from *millions* of separate classes. For example, consider a multi-class classification model that can identify the image of just about anything.

This section details the two main variants of multi-class classification:

- **one-vs.-all**
- **one-vs.-one**, which is usually known as **softmax**

One versus all

Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome. During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question.

For example, given a picture of a piece of fruit, four different recognizers might be trained, each answering a different yes/no question:

1. Is this image an apple?
2. Is this image an orange?
3. Is this image a banana?
4. Is this image a grape?

One versus one (softmax)

What if we want to predict the probabilities of each fruit relative to each others? In this case, instead of predicting "apple" versus "not apple", we want to predict "apple" versus "orange" versus "pear" versus "grape". This type of multi-class classification is called *one-vs.-one classification*.

We can implement a one-vs.-one classification using the same type of neural network architecture used for one-vs.-all classification, with one key change. We need to apply a different transform to the output layer.

For one-vs.-all, we applied the sigmoid activation function to each output node independently, which resulted in an output value between 0 and 1 for each node, but did not guarantee that these values summed to exactly 1.

For one-vs.-one, we can instead apply a function called *softmax*, which assigns decimal probabilities to each class in a multi-class problem such that all probabilities add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

Completion

You earned the **Machine Learning Crash Course: Neural networks** badge!

The badge has been added to your profile.