Image Colorization

The SoC 2021 project.

Useful Links

A Machine Learning course Here
Notes on Machine Learning here

Linear Regression

The main aim is to estimate a linear equation representing the given set of data. There are two approaches to this.

1. A closed form solution.
   This can be directly obtained by solving the linear differential equation.
2. An iterative approach.
   This is similar to **Gradient Descent**. We try to obtain the minima ($L1$, $L2$ norm etc) by calculating the gradient at each point and moving in small steps along the gradient vector.
   Refer to this video for more details.

Logistic Regression

Refer to the following link to see an example of logistic regression.

Gradient Descent

Here is a useful video.
An article about Gradient Descent here
A useful post on GeeksForGeeks here

Deep Learning

A book on deep learning here

Chapter 1 - Using neural nets to recognize handwritten digits

Perceptrons

So how do perceptrons work?

A perceptron takes several binary inputs, $x_1, x_2, \ldots, x_n$ and produces a single binary output.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

A way you can think about the perceptron is that it's a device that makes decisions by weighing up the evidence. By varying the weights and the threshold, we can get different models of decision-making. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$output = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND.

Sigmoid Neurons

A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.
Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \ldots$ But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1. So, for instance, $0.638\ldots$ is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, $w_1, w_2, \ldots,$ and an overall bias, b. But the output is not 0 or 1. Instead, it's $\sigma(w \cdot x + b)$, where σ is called the sigmoid function.
To understand the similarity to the perceptron model, suppose $z \equiv w \cdot x+b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. In other words, when $z=w \cdot x+b$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that $z=w \cdot x+b$ is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$. So when $z=w \cdot x+b$ is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron.

Neural Networks Architecture

A typical neural network consists of an input layer, an output layer and 0 or more hidden layers. *Hidden Layers* are just layers which are neither the input layer nor the output layer. Each layer consists neurons whose input is taken from the previous layer and the output acts like the input to the next layer.
Each layer in the neural network captures an abstraction/feature of the object/data we are trying to learn. For example, the digit recognition neural network has 3 layers. The input layer consists of 784 (28 * 28) neurons and the output layer has 10 neurons each corresponding to an identity of a digit. The hidden layer has 20 neurons. One may think of the hidden layer as capturing the main features of a digit such as loops, curves and straight lines.
The aim of a neural network is to learn the input data and output the corresponding label to a particular input. To do this, we define a **cost function**. For example, in the digit recognition neural network, we use a *least squared error* norm to minimise the error in the output. This function is particularly useful because it is convex and it is minimized when the output of the neural network matches the correct output.

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

To minimise the cost function, we use a calculus approach. Notice that the above cost function is a function of all the weights and biases in the network. We aim to minimise the cost function for a given input dataset by tuning the weights and biases in the network. We tweak each of these variables (weights and biases) and measure the change in the cost function for *all* the inputs. This way, we identify the **gradient of the function**. We move in the negative direction of the gradient to minimise the cost function. We repeat this until convergence (A threshold accuracy). This algorithm is called as **Gradient Descent**.

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Here, η refers to the **learning rate** and is one of the *hyperparameters*.

In the above algorithm, we calculated the cost function for all the inputs in each iteration. This turns out to be a computationally expensive step. Therefore, we select a **mini-batch** from the input and evaluate the cost function over this input. We assume that this cost function is a representative of the real cost function. We change the mini-batch in each iteration so as to cover all the inputs. The change in the cost function looks like this:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

Note The above cost function is taken as an average to maintain consistency in the mini-batches method.

In summary, we take a random set of weights and biases and perform gradient descent on a subset of inputs to obtain the optimal set of parameters. This method is called as **Stochastic Gradient Descent**. (Stochastic refers to the random initial start and random mini-batches).

Implementation of Neural Networks

The following code is taken from the book. We implement Neural Networks in Python for the rest of the document.

The Network Class

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

The FeedForward Mechanism

```
def feedforward(self, a):
        """Return the output of the network if "a" is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

The Stochastic Gradient Descent Method

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The "training_data" is a list of tuples
        "(x, y)" representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If "test_data" is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)
```

The Back Propagation Algorithm

```
  def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

Note. Back Propagation was not discussed until here. It will be discussed in the next chapter.

Observations

The weights and biases are called as *parameters* whereas the size of a mini-batch, number of epochs (The number of times SGD is repeated over the whole input) and learning rate are called as *hyperparameters*. These hyperparameters play a crucial role in the time taken for training the network. They must be chosen with care and they can be tuned based on the observation made from the output.

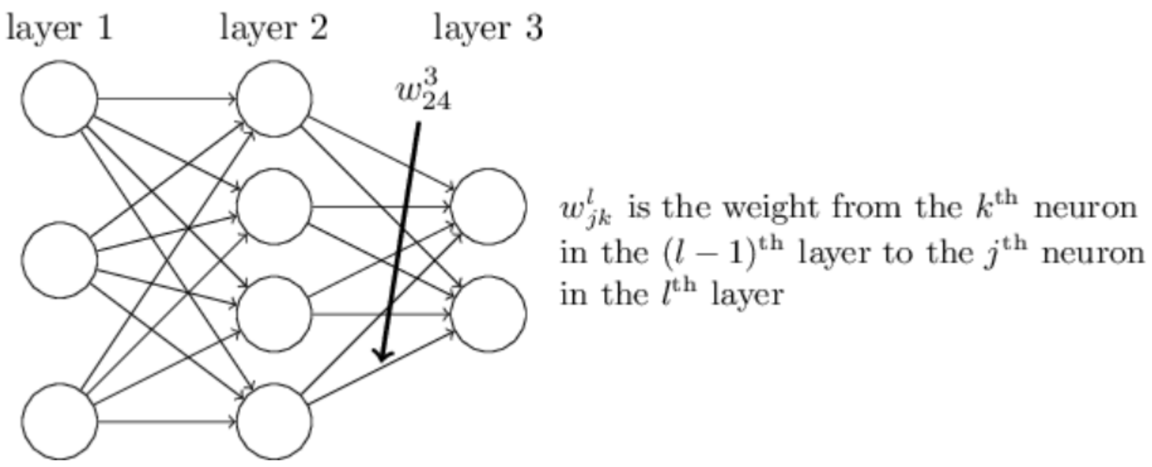
A deep neural network is able to learn complex data due to its hierarchical structure. Each layer captures a level of abstraction which is used by the next layer to determine a higher level of features in the data.

Chapter 2 - How the backpropagation algorithm works

A Matrix based approach for faster calculations

All the above equations in the network are written in terms of summations. We can easily replace them using matrix representation. This allows faster calculations and does away with indices. The notation is w^l_{jk} to denote the weight for the connection from the kth neuron in the (l−1)th layer to the jth neuron in the lth layer.

Note. Pay attention to the reverse notation.


layer 1
layer 2
layer 3
w^3_{24}
w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer