Feedforward Neural Networks

By: Collin Brown

Contents:

1. High Level Overview

The algorithm to create and train a Feedforward Neural Network consists of the following steps:

```
1. Initialize the Neural Network


while (number_epochs < max_epochs)

do

    2. Forward pass
    3. Calculate Error
    4. Backpropagate Error Through the Network
    5. Update the Model Parameters
    6. Perform Gradient Descent (Batch, Stochastic, or Mini-batch)
```

1. Understanding Dimensions at Initialization

```
    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 model parameters consist of the following:

$$\theta = \Big[\big[\mathbf{W^{0,1}}, \mathbf{W^{1,2}}, \ldots, \mathbf{W^{L-1,L}}\big], \big[\mathbf{b^1}, \mathbf{b^2}, \ldots, \mathbf{b^L}\big]\Big]$$

- When creating an object of type NeuralNetwork, we initialize weights and biases with random numbers generated from a gaussian standard normal distribution. Note that there are many assumptions you can make when initializing the parameters of a network.
- The biases are a list of 1D numpy arrays and the weights are a list of 2D numpy arrays.
- Note that the biases only enter the network after the 0th layer (i.e. the input layer does not get a bias term).
- Note that the row dimension of the weight matrix depends on the number of neurons in the next layer, while the column dimension of the weight matrix depends on the number of neurons in the current layer.
- We have $z_1^{l+1} = w_{1,1}h_1^l + w_{2,1}h_2^l + w_{3,1}h_3^l$, and $z_2^{l+1} = w_{1,2}h_2^l + w_{2,2}h_2^l + w_{3,2}h_3^l$ where z_i^{l+1} refers to the "net input" for layer $l+1$ for the ith neuron.
- We can think of this as a matrix multiplication problem.

$$\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix} \begin{pmatrix} h_1^l \\ h_2^l \\ h_3^l \end{pmatrix}$$

2. Feedforward

```
    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a) + b)
        return a
```

- Zip the lists containing the numpy arrays of biases and weights.
- For each weight matrix sitting between two layers and the corresponding bias vector, apply the activation function (in this example, the activation function is sigmoid).
- This method doesn't keep track of activations in each layer; this happens in the backpropagation method
- `np.dot(a,b)` is just matrix multiplication if a and/or b are 2D numpy arrays (matricies).
- This code is equivalent to the below line of math:

$$\begin{pmatrix} a_1^l \\ a_2^l \\ \ldots \\ a_N^l \end{pmatrix} := \begin{pmatrix} \sigma\big(w_{1,1}a_1^{l-1} + \ldots + w_{1,M}a_M^{l-1} + b_1^l\big) \\ \sigma\big(w_{2,1}a_1^{l-1} + \ldots + w_{2,M}a_M^{l-1} + b_2^l\big) \\ \ldots \\ \sigma\big(w_{N,1}a_1^{l-1} + \ldots + w_{N,M}a_M^{l-1} + b_N^l\big) \end{pmatrix}$$

3. Stochastic Gradient Descent

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data = None):
        if test_data:
            n_test = len(test_data)
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                    training_data[k:k + mini_batch_size]
                    for k in range(0, n, mini_batch_size)
                    ]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data is not None:
                print("Epoch {0}: {1} / {2}".format(j,
                      self.evaluate(test_data), n_test))
            else:
                print("Epoch {0} complete".format(j))
```

Digression on Gradient Descent

- We start by defining a loss function $\mathscr{L}$ which depends on both the data and the model's parameters. Specifically,

$$\mathscr{L}\Big(\big(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}\big)_{i=1}^{N}; \big[\mathbf{W}^{\mathbf{0,1}}, \mathbf{W}^{\mathbf{1,2}}, \ldots, \mathbf{W}^{\mathbf{L-1,L}}\big], \big[\mathbf{b}^{\mathbf{1}}, \mathbf{b}^{\mathbf{2}}, \ldots, \mathbf{b}^{\mathbf{L}}\big]\Big) = \frac{1}{2}\sum_{i=1}^{N}\big(\mathbf{y}_{(i)} - \hat{\mathbf{y}}_{(i)}\big)^2$$

$$\hat{\mathbf{y}}_{(i)} = \text{Classification}\Big(\sigma\big(\mathbf{W}^{\mathbf{L-1,L}}\mathbf{a}^{L-1} + \mathbf{b}^{L}\big)\Big)$$

$$\mathbf{a}^{L-1} = \sigma\big(\mathbf{W}^{\mathbf{L-2,L-1}}\mathbf{a}^{L-2} + \mathbf{b}^{L-1}\big)$$

$$\ldots$$

$$\mathbf{a}^{1} = \sigma\big(\mathbf{W}^{\mathbf{0,1}}\mathbf{x}_{\mathbf{(i)}} + \mathbf{b}^{1}\big)$$

- Our aim is to **choose the parameters** of $\mathscr{L}$ in such a way to minimize the value that $\mathscr{L}$ takes on (i.e. "minimize the loss function").

$$\min_{\theta} \mathscr{L}\Big(\big(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}\big)_{i=1}^{N}; \theta\Big)$$

- The surface of $\mathscr{L}$ exists in some high dimensional parameter space.

--> Insert figure or draw on board

- When weights are initialized randomly, we show up at some point on the surface of $\mathscr{L}$.
- The gradient points in the direction of steepest **ascent**; if we multiply the gradient by -1, it points in the direction of steepest **descent**.
- Starting from a random point on $\mathscr{L}$ (i.e. given a random set of parameters θ), for each parameter $\theta_i \in \theta$, we can calculate $\frac{\partial \mathscr{L}}{\partial \theta_i}$.
- Since $(-1)\nabla\mathscr{L}$ points in the direction of steepest descent, we're "pushing" each parameter θ_i in such a way that after applying the following update rule:

$$\theta_i' := \theta_i - \eta \frac{\partial \mathscr{L}}{\partial \theta_i}$$

- we are guaranteed the following for an arbitrarily small change in θ_i':

$$\mathscr{L}\Big(\big(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}\big)_{i=1}^{N}; \theta'\Big) \leq \mathscr{L}\Big(\big(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}\big)_{i=1}^{N}; \theta\Big)$$

- Note that the above is only true for arbitrarily small changes in a continuous space; in practice, we compute these parameters numerically, so the above condition is by no means guaranteed if $\mathscr{L}$ is not well-behaved.

training_data: A set of (x, y) tuples representing the input-label pairs. Note that x has the dimensions of the input layer and y has the dimensions of the output layer. A tuple (x, y) represents a single observation. **epochs**: The number of times that we want to pass over the entire training set.

mini_batch_size: The size of mini-batches to be used.

Digression on Types of Gradient Descent

1. There are three types of gradient descent: (1) Batch (2) Stochastic (3) Mini-batch
    1. With **batch** gradient descent, the entire training set (i.e. every single (x, y) pair that is contained in the training set) is passed through the network before any update to the weights/biases happen. This can make model updates computationally slow with large datasets (b/c it takes much longer to perform a signle update on the network) and can sometimes lead to convergence to non-optimal local minima due to the gradient being very stable.
    2. With **stochastic** gradient descent, the network parameters are updated every time a single (x, y) pair is passed through the network. This process makes the gradient very noisy, making it hard for the model to converge to an optimum.

3. With **mini-batch** gradient descent, a subsample of the training set is used to calculate the gradient (e.g. 32 or 64 (x, y) tuples). This type of gradient descent is most commonly used in practice, as it achieves some gradient stability (but still allows the gradient to explore so as to not get stuck in a non-optimal minima) and provides frequent updates to the network parameters without performing an update calculation after every single (x, y) tuple gets passed through the network.

eta: The learning rate parameter that tells the update rule how much emphasis it should place on the gradient's update with respect to a particular weight or bias.

- For each epoch in the number of epochs the user chooses, the code above shuffles the ordering of the training data, breaks the training data into mini batches.
- For every mini batch in the set of mini batches calculated above, pass the mini batch to the **update_mini_batch()** method.
- If the user passes test data to the network, this method prints progress updates to the console.

4. Update Mini Batches

```
    def update_mini_batch(self, mini_batch, eta):
        # Initialze the gradients for network biases and weights to
contain
        # zeros for the conformable number of dimensions
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # For each (x, y) pair in the mini-batch, calculate the gradient
        # associated with each pair, then add them together
        for x, y in mini_batch:
            # For a single (x, y) pair, return the weight/bias gradients
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
```

- The code above initializes a list of 1D numpy arrays for each layer's bias and a list of 2D numpy arrays for the weight matrices sitting between each layer.
- Then, for every (x, y) tuple (observation) contained in the mini batch, calculate the gradient with respect to the biases and weights for each layer. That is,

$$\nabla b^l = \begin{pmatrix} \frac{\partial L}{\partial b_1^l} \\ \frac{\partial L}{\partial b_2^l} \\ \cdots \\ \frac{\partial L}{\partial b_{N_l}^l} \end{pmatrix}$$

and

$$\nabla W^{l,l+1} = \begin{pmatrix} \frac{\partial L}{\partial w_{1,1}}, \frac{\partial L}{\partial w_{1,2}}, \cdots, \frac{\partial L}{\partial w_{1,N_l}} \\ \frac{\partial L}{\partial w_{2,1}}, \frac{\partial L}{\partial w_{2,2}}, \cdots, \frac{\partial L}{\partial w_{2,N_l}} \\ \cdots \\ \frac{\partial L}{\partial w_{N_{l+1},1}}, \frac{\partial L}{\partial w_{N_{l+1},2}}, \cdots, \frac{\partial L}{\partial w_{N_{l+1},N_l}} \end{pmatrix}$$

- The rest of the function follows.

```
        # After the entire minibatch has been iterated over, update the
        # network's weights and biases with the average nabla_w/nabla_b
        # in the mini-batch
        self.weights = [w - (eta / len(mini_batch)) * nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b - (eta / len(mini_batch)) * nb
                       for b, nb in zip(self.biases, nabla_b)]
```

- Once all of the gradients are calculated, update the lists of weights and biases by applying the following update rules.

$W^{l,l+1} := W^{l,l+1} - \eta \nabla W^{l,l+1}$

$b^l := b^l - \eta \nabla b^l$

5. Backpropagation

```
    def backprop(self, x, y):
        # Initialize numpy arrays that store the gradients
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # Feedforward step
        # Note that the first activation is simply the input, x
        activation = x
```

```
        activations = [x]  # list to store all of the activations layer
by layer
        zs = []  # List to store the net input vectors, layer by layer
        # For all weights and biases, feed the input forward through the
        # network
        for b, w in zip(self.biases, self.weigths):
            # Compute net input for the current layer
            z = np.dot(w, activation) + b
            # Add most recent net input to the list of net inputs
            zs.append(z)
            # Pass net input through the activation function
            activation = sigmoid(z)
            # Add most recent activation to the list of activations
            activations.append(activation)
```

- The above code starts with the input data 'x' and feeds it forward through the network.
- Intermediate activations and net inputs are stored in variables 'activation' and 'z'.
- Next, there is a backward pass through the network, where the network's parameter gradients are calculated.

```
        # Backward pass through the network
        # Calculate error associated with the weights and biases of the
final
        # layer of the network (This is equation BP1 from MN book)
        delta = self.cost_derivative(
                activations[-1], y) * sigmoid_prime(zs[-1])
        # Gradient of cost wrt bias is just delta (eq BP3)
        nabla_b[-1] = delta
        # Gradient of cost wrt weights is the errors of layer l
multiplied by
        # the activations of layer l-1
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # For every additional layer in the network, compute nabla_b and
        # nabla_w
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(activations[-l].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l - 1].transpose())
```

```
        # Return a tuple of gradients for the biases and weights of the
network
        return (nabla_b, nabla_w)
```

Motivation for Backpropagation

- We need to find $\frac{\partial \mathscr{L}}{\partial w_{i,j}^{l-1,l}}$ and $\frac{\partial \mathscr{L}}{\partial b_i^l}$ for all $i, j, \text{and} l$ to compute our gradient vectors.
- From the chain rule in calculus, we know the following:

$$\frac{\partial \mathscr{L}}{\partial w_{i,j}^{l-1,l}} = \frac{\partial \mathscr{L}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{i,j}^{l-1,l}}$$

and

$$\frac{\partial \mathscr{L}}{\partial b_i^l} = \frac{\partial \mathscr{L}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} \frac{\partial z_i^l}{\partial b_i^l} \$\$$$

INSERT DIAGRAM

- We know that $\frac{\partial z_i^l}{\partial w_{i,j}^{l-1,l}} = a_j^{l-1}$ because

$$\frac{\partial}{\partial w_{i,j}^{l-1,l}} \left(\sum_{k=1}^{N_l} w_{k,i}^{l-1,l} a_k^{l-1,l} + b_i^l \right) = a_k^{l-1} \forall k \in N_l$$

- Intuitively, the way that the above expression changes with respect to weight $w_{i,j}^{l-1,l}$ depends only on the activation in the previous layer that is connected to the net input through this weight. Therefore, the way that the net input changes with respect to this weight is simply equal to the activation in the previous layer, a_j^{l-1}.
- Further, we also know that $\frac{\partial z_i^l}{\partial b_i^l} = 1$ because

$$\frac{\partial}{\partial b_i^l} \left(\sum_{k=1}^{N_l} w_{k,i}^{l-1,l} a_k^{l-1,l} + b_i^l \right) = 1 \forall k \in N_l$$

- So, we can simplify the original expressions to the following:

$$\frac{\partial \mathscr{L}}{\partial w_{i,j}^{l-1,l}} = \frac{\partial \mathscr{L}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} (a_j^{l-1})$$

and

$$\frac{\partial \mathscr{L}}{\partial b_i^l} = \frac{\partial \mathscr{L}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} (1).$$

- However, it is still not clear how we calculate $\frac{\partial \mathscr{L}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l}$ for each i and l.

Solution: Create an artificial variable "error" variable called δ^L, and backpropagate it through the network.

- We define an artificial "error" variable in the following way:

$$\delta^L = (\sigma(z^L) - y) \odot \sigma'(z^L)$$

- where $(\sigma(z^L) - y)$ plays the role of a feedback signal, $\sigma'(z^L)$ is the contribution that the net input of the final layer made to the "decision" of the network, and $\odot$ is the hadamard product operator, which means to take the element-wise product of the two above vectors.
- We "flip" the network around, so that the output layer becomes the input layer, and treat δ^L (i.e. the error for the output layer) as the input to this reversed network.

INSERT DIAGRAM

- To feed δ^L back one layer, we do the following matrix multiplication: $(W^{L-1,L})^T \delta^L$

- Note that transposing the weight matrices lets us move activations in the opposite direction in the network.
- At layer $L-1$, we want to know how δ^L is impacted by a small change in the net input of layer $L-1$. We can get this through the following equation:

$$\delta^{L-1} = (W^{L-1,L})^T \delta^L \odot \sigma'(z^{L-1})$$

- In general, for any layer l, we have

$$\delta^l = (W^{l,l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

- So, we have now computed $\delta^l \forall l = 1...L$, now what?
- Notice that

$$\frac{\partial \mathscr{L}}{\partial a_i^L} = \sigma(z_i^L - y^{(i)})$$

- and

$$\frac{\partial a_i^L}{\partial z_i^L} = \sigma'(z_i^L)$$

- so

$$\delta_i^L = \sigma(z_i^L - y^{(i)})\sigma'(z_i^L) = \frac{\partial \mathscr{L}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L}$$

- The first equality comes from our definition of δ_i^L, but is for a single element of the vector δ^L. The second equality comes from our observation of what each of these partial derivatives is equal to.

- Similarly, for layer $L-1$,

$$\frac{\partial \mathscr{L}}{\partial a_i^{L-1}} = W_{i,j}^{L-1,L}\sigma(z_i^L - y^{(i)})\sigma'(z_i^L)$$

- which, from our definition of δ_i^L, can be rewritten as

$$\frac{\partial \mathscr{L}}{\partial a_i^{L-1}} = W_{i,j}^{L-1,L}\delta_i^L$$

- and we also have

$$\frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} = \sigma'(z_i^{L-1})$$

- so

$$\delta_i^{L-1} = W_{i,j}^{L-1,L}\delta_i^L\sigma'(z_i^{L-1}) = \frac{\partial \mathscr{L}}{\partial a_i^{L-1}}\frac{\partial a_i^{L-1}}{\partial z_i^{L-1}}$$

- If we keep moving through the layers of our network, we observe the following:

$$\delta_i^{L-2} = W_{i,j}^{L-2,L-1}\delta_i^{L-1}\sigma'(z_i^{L-2}) = \frac{\partial \mathscr{L}}{\partial a_i^{L-2}}\frac{\partial a_i^{L-2}}{\partial z_i^{L-2}}$$

$$\dots$$

$$\delta_i^{l} = W_{i,j}^{l,l+1}\delta_i^{l+1}\sigma'(z_i^{l}) = \frac{\partial \mathscr{L}}{\partial a_i^{l}}\frac{\partial a_i^{l}}{\partial z_i^{l}}$$

Bottom Line: since we have computed the "error" variable δ^l for each layer, we can calculate $\frac{\partial \mathscr{L}}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_i^l}$ for each i and l, which were the missing pieces of information that we need to compute every element of our gradient vectors. Once we have our gradient vectors, we can update the parameters of our model through gradient descent.

Digression on Delta

- We can rewrite $\delta^L = \nabla_a \mathscr{L} \odot \sigma ' (z^L)$ as follows:

$$\delta^L = \mathbf{\Sigma}'(z^L)\nabla \mathscr{L}$$

where

$$\mathbf{\Sigma}'(z^L) = \begin{pmatrix} \sigma'(z_1^L), 0, \dots, 0 \\ 0, \sigma'(z_2^L), \dots, 0 \\ \dots \\ 0, 0, \dots, \sigma'(z_N^L) \end{pmatrix}$$

- We can also rewrite equation (2) as follows:

$$\delta^l = \mathbf{\Sigma}'(z^l)\big(\mathbf{W}^{l,l+1}\big)^T\delta^{l+1} \forall l = 1, 2, \dots, L-1$$

- By using repeated substitution of the above equation for each l , and also the equation for δ^L, we can represent δ^l as follows for any l:

$$\delta^l = \mathbf{\Sigma}'(z^l)\big(\mathbf{W}^{l,l+1}\big)^T\mathbf{\Sigma}'(z^{l+1})\big(\mathbf{W}^{l+1,l+2}\big)^T \dots \mathbf{\Sigma}'(z^{L-1})\big(\mathbf{W}^{L-1,L}\big)^T\mathbf{\Sigma}'(z^L)\nabla \mathscr{L}$$

$$\delta^l = \Big[\prod_{j=l}^{L-1} \mathbf{\Sigma}'(z^j)\big(\mathbf{W}^{j,j+1}\big)^T\Big]\mathbf{\Sigma}'(z^L)\nabla \mathscr{L}$$

- Note that the range of σ is $(0, 1)$, and the range of σ' is a subset of $(0, 1)$ (Note that the logistic pdf is a class of probability distributions, so its range depends on how a specific pdf belonging to the logistic distribution class is parameterized).
- This means that in the above chain product, if the number of layers gets large, there is a large chain product involving numbers in the set $(0, 1)$.
- This means that as the number of layers increases, each entry in δ^l is getting multiplied by an increasingly small number.
- So, if there are too many layers, $\delta^l \to \vec{0}$. Each element of $\nabla \mathscr{L}$ depends on δ_j^l (see equations below), so if each $\delta_j^l \to 0$, then $\nabla \mathscr{L} \to \vec{0}$.
- The above problem is called "gradient dampening", and it makes it hard for deep networks to learn because the paramters θ never update in gradient descent.