Notes on a 2-Layer Feed Forward Neural Network for Regression Tasks

Jonathan Navarrete

2019
January

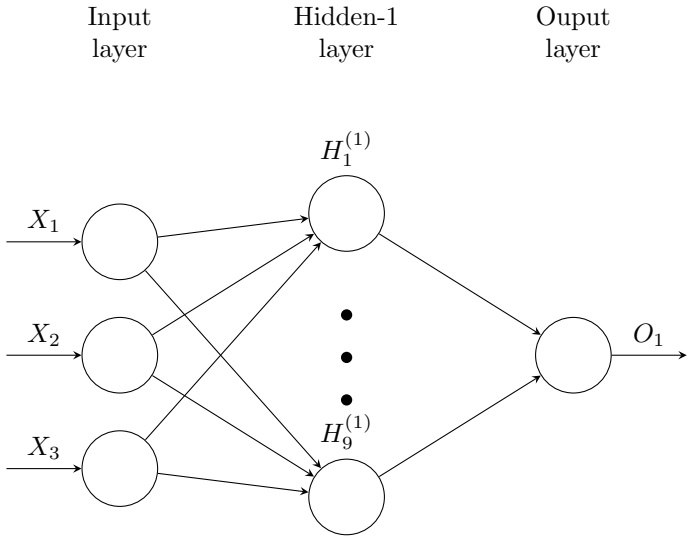
1 Introduction

The following notes relate to a simple feed forward neural network trained for regression tasks. The neural network has an input layer (the inputs) and two hidden layers. The first hidden layer recieves the input data and transforms it into a nonlinear space. The data is feed forward into the second hidden layer for the output. The neural network is trained using vanilla stochastic gradient descent (SGD).

The neural network is implemented in a Python script (nnet.py).

```
def example_data(rows = 20, columns = 3):
    #np.random.seed(123)
    X = np.random.normal(size=(rows, columns))
    X[:, 0] = np.linspace(start=-10, stop=10, num=rows)**4
    X[:, 1] = np.linspace(start=-10, stop=10, num=rows)
    X[:, 2] = np.linspace(start=-10, stop=10, num=rows)**3
    y = 1.5 * X[:, 0] + X[:, 1] + 2 * X[:, 2]
    y = y[:, None]
    ## scale the values ##
    X = X / 1000
    y = (y - 68) / 500
    return (X, y)
```

2 Architecture



2.1 Data

The neural network takes 3 inputs, where the design matrix $\mathbf{X}$ is $N \times 3$; however, this can be generalized to any number of M inputs for a $N \times M$ design matrix. The neural network is then fed one observation at a time, $\mathbf{x}_i^\intercal$ for $i = 1, 2, ..., N$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ & \cdots & \\ x_{N1} & x_{N2} & x_{N3} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\intercal \\ \mathbf{x}_2^\intercal \\ \cdots \\ \mathbf{x}_N^\intercal \end{bmatrix}$$

There is one output target

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_N \end{bmatrix}$$

2.2 First Hidden Layer

In the first hidden layer there are 9 hidden units $H_1^{(1)}, H_2^{(1)}, ..., H_9^{(1)}$. Each hidden unit sees each of the three 3 inputs, aggregates the inputs with weights and passes that hidden output to an activation function. The activation function used in the first hidden layer is logistic,

$$H_k^{(1)} = f(z) = \frac{1}{1 + e^{-z}}$$

where $z = w_1^{(1)} x_i 1 + w_2^{(1)} x_i 2 + w_3^{(1)} x_i 3$.
For each hidden unit, we have three weights $\mathbf{w}^{(1)}$

2.3 Second Hidden Layer

In the second hidden layer there is 1 hidden units (the output unit). Because there is only one output, y_i, there is only the need for one hidden unit.

The second hidden layer expects 9 inputs from the first hidden layer. The activation function is simply the identity function $g(z) = z$, where $z = w_1^{(2)} x_i 1 + w_2^{(2)} x_i 2 + w_9^{(2)} x_i 3$.

3 Feed Forward

3.1 Input to First Hidden Layer

For each observation i, we take input $\mathbf{x}_i^\intercal$ and pass it to the first layer to each hidden unit j

$$h_j = f(z) = \frac{1}{1 + e^{-z_j}}$$

where

$$z_j = \mathbf{x}_i^\intercal \mathbf{w}_j^{(1)} = \begin{bmatrix} x_{i1} & x_{i2} & x_{i3} \end{bmatrix} \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_3^{(1)} \end{bmatrix} = w_1^{(1)} x_1 + w_2^{(1)} x_2 + w_3^{(1)} x_3$$

This can be more efficiently computed by using a 3×9 weights matrix

$$\mathbf{W}^{(1)} = \begin{bmatrix} \mathbf{w}_1^{(1)} & \mathbf{w}_2^{(1)} & \cdots & \mathbf{w}_9^{(1)} \end{bmatrix}$$

Take the input row vector $\mathbf{x}_i^\intercal$ and multiply it with $W^{(1)}$ to obtain $\mathbf{z}^\intercal$

$$\mathbf{z}^\intercal = \mathbf{x}_i^\intercal \mathbf{W}^{(1)} = \begin{bmatrix} z_1 & z_2 & \cdots & z_9 \end{bmatrix}$$

Afterwards, we pass the hidden layer outputs through the activation function,

$$\mathbf{h}^\intercal = f(\mathbf{z}^\intercal) = \begin{bmatrix} h_1 & h_2 & \cdots & h_9 \end{bmatrix}$$

3.2 Hidden to Output Layer

After the first hidden layer is computed, the hidden outputs $\mathbf{h}^\intercal$ is fed to the output layer. The ouptut layer has 1 output and expects 9 inputs. Thus, we have a 9×1 weights matrix

$$\mathbf{W}^{(2)} = \mathbf{w}^{(2)} = \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_9 \end{bmatrix}$$

$$o_i = g(\mathbf{h}^\intercal \mathbf{w}^{(2)}) = \mathbf{h}^\intercal \mathbf{w}^{(2)}$$

4 Backpropogation

> The back-propagation algorithm (Rumelhart et al., 1986a), often simply called backprop, allows the information from the cost to then ow backward throughthe network in order to compute the gradient. [...] The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm,such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specic to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undened).[1]

[1] https://www.deeplearningbook.org/contents/mlp.html

4.1 Calculation Errors

For this regression task we will use the mean squared error (MSE) as the statistic to measure performance. We'd like to create a neural network that has a small as possible MSE. Thus we are trying to minimize MSE with respect to the weights in the neural network.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(o_i - y_i)^2 = \frac{1}{N}\sum_{i=1}^{N} e_i^2$$

To train the neural network, it'll be necessary to differentiate this error response with respect to each weight. This will be accomplished using the backpropogation algorithm.

4.2 Gradient for Output Layer

Using the backprop algorithm we will take the derivative of MSE with respect to (wrt) the weights in the output layer. Let $E = MSE(w^{(2)})$, then by the chain rule we obtain

$$\frac{dE}{dw_k^{(2)}} = \frac{dE}{do_i}\frac{do_i}{dw_k^{(2)}}$$

The derivative of E wrt o_i is

$$\frac{dE}{do_i} = \frac{2}{N}\sum_{i}^{N}(y_i - o_i)$$

and the derivative of o_i wrt $w_k^{(2)}$ is

$$\frac{do_i}{dw_k^{(2)}} = \frac{d}{dw_k^{(2)}}(dw_1^{(2)}h_1^{(1)} + \cdots + w_k^{(2)}h_k^{(1)} + \cdots + +w_9^{(2)}h_9^{(1)}) = h_k^{(1)}$$

Thus, the partial derivate is

$$\frac{2}{N}\sum_{i}^{N}(y_i - o_i) \times h_k^{(1)}$$

However, this process will need to be continued for all weights $w_1^{(2)}, \cdots, w_9^{(2)}$. Taking partial derivatives for all weights gives us the gradient.

$$\frac{dE}{dw_k^{(2)}} = \frac{dE}{do_i} \nabla o_i$$

where

$$\nabla o_i = \begin{bmatrix} \frac{do_i}{dw_1^{(2)}} \\ \cdots \\ \frac{do_i}{dw_9^{(2)}} \end{bmatrix} = \begin{bmatrix} h_1^{(1)} \\ \cdots \\ h_9^{(1)} \end{bmatrix}$$

4.3 Gradient for Hidden Layer

Just as was done for the second hidden layer, the output layer, we have to take the derivative wrt the first hidden layer's weights $w^{(1)}$.

$$\frac{dE}{dw_k^{(1)}} = \frac{dE}{do_i} \frac{do_i}{dh_k^{(1)}} \frac{dh_k}{dw_j^{(1)}}$$

Taking the derivative of E wrt o_i remains the same. However, the second partial derivative is a bit different

$$\frac{do_i}{dh_k^{(1)}} = \frac{d}{dh_k^{(1)}} (dw_1^{(2)} h_1^{(1)} + \cdots + w_k^{(2)} h_k^{(1)} + \cdots + + w_9^{(2)} h_9^{(1)}) = w_k^{(2)}$$

Repeating this process of taking partial derivatives wrt each hidden layer input we obtain the gradient

$$\nabla o_i = \begin{bmatrix} \frac{do_i}{dh_1^{(1)}} \\ \cdots \\ \frac{do_i}{dh_9^{(1)}} \end{bmatrix} = \begin{bmatrix} w_1^{(2)} \\ \cdots \\ w_9^{(2)} \end{bmatrix}$$

The last remaining step is to calculate the derivative of h_k wrt $w_j^{(1)}$

$$\frac{dh_k}{dw_j^{(1)}} = \frac{dh_k}{dz_k} \times \frac{dz_k}{dw_j^{(1)}}$$

Since $h_k = f(z_k) = \textbf{logistic}(z_k)$, uses the logistic activation function, we'll need to take the derivative of the logistic function.

$$f'(z_k) = f(z_k) \times (1 - f(z_k))$$

with the gradient being

$$\nabla \mathbf{h} = f(\mathbf{z})(1 - f(\mathbf{z}))$$

and

$$\frac{dz_k}{dw_j^{(1)}} = \frac{d}{dw_j^{(1)}}(w_k^{(1)} x_k) = x_k$$

with the gradient being

$$\nabla \mathbf{z_k^\intercal} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

and for k = 1, ..., 9

$$\nabla \mathbf{z} = \begin{bmatrix} \mathbf{z_1^\intercal} \\ \cdots \\ \mathbf{z_9^\intercal} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ & \cdots & \\ x_1 & x_2 & x_3 \end{bmatrix}$$

Finally, the gradient of E becomes

$$\nabla E = \frac{dE}{do_i} \times \nabla o_i \times \nabla \mathbf{h} \nabla \mathbf{z}^\intercal = \frac{2}{N} \sum_i^N \left((y_i - o_i) \times \begin{bmatrix} w_1^{(2)} \\ \cdots \\ w_9^{(2)} \end{bmatrix} \times f(\mathbf{z})(1 - f(\mathbf{z})) \right) \begin{bmatrix} x_1 & x_2 & x_3 \\ & \cdots & \\ x_1 & x_2 & x_3 \end{bmatrix}$$

5 Python Implementation

The neural network described above is implemented in the 'NeuralNetwork' class. The neural network class requires the X and y data inputs, the number of hidden units desired (default is 9), and the desired learning rate (default is 0.1). After the object has been initialized, it will have three methods

1. __init__
2. forward
3. backpropogation

4. train

At initialization, the object generates two weights matrices for the first and second (output) hidden layers.

```
class NeuralNetwork:

    def __init__(self, X, y, hidden_units = 9, learning_rate = 0.1):
        """
        Two hidden layer neural network for regression
            Hidden Layer 1: Logistic Activation
            Hidden Layer 2: Linear Activation (for regression)
        """
        self.X = X
        self.y = y
        self.learning_rate = learning_rate
        self.hidden_units = hidden_units
        np.random.seed(123)
        self.weights1 = np.random.normal(loc=0, scale=1,
                                size=(X.shape[1], hidden_units)) # (input units,
        self.weights2 = np.random.normal(loc=0, scale=1,
                                size=(hidden_units, 1)) # for now there will only


    def forward(self, X = None, verbose=False):
                ...


    def backpropogation(self, hidden_output1, hidden_output2, x, target):
                ...


    def train(self, n_epochs = 15):
        ...
```

5.1 Forward Pass

The forward pass takes incoming data $\mathbf{x}_i^\intercal$ and applies a linear operation on it with its respective layer weights. Afterwards, the hidden input is passed through the logistic activation function.

The hidden output is then forwarded to the second hidden layer, the output layer, and passed through the linear function again. The output layer has an identity activation function, so there's no additional work needed there.

```
...

    def forward(self, X = None, verbose=False):
        if isinstance(X, np.ndarray):
            hidden_output1 = linear(X, self.weights1)
            hidden_output1 = sigmoid(hidden_output1) # apply the nonlinear tr

            hidden_output2 = linear(hidden_output1, self.weights2)

            if verbose:
                print("hidden output: ", hidden_output1.round(3))
                print("output: ", hidden_output2.round(3))

            return hidden_output2, hidden_output1
        else:
            raise ValueError("need a vector in the forward function")

    ...
```

The linear function is simply dot product.

```
def linear(X_input, weights):
    """
    out = Xw
    X should be (rows, columns) and w should match (columns, hidden units)
    """
    a = np.dot(X_input, weights)
    return a
```

5.2 Backpropogation

During the training pass, we need to feed back the errors to update the weights. In the backpropogation method, the neural network takes the hidden output, neural network output, x input and the output target values.

First, calculate the output error, the difference between the target and the neural network output. With that error, you multiply it with the weights

of the second hidden layer.

Finally, to get the update for the first layer, we multiply it by the gradient of $\mathbf{z}^\intercal$; this is just the input. Instead of creating a matrix, we can just calculate the product between hidden error and $\mathbf{x}^\intercal$. The vectors multiplied together will create a matrix.

```
...

    def backpropogation(self, hidden_output1, hidden_output2, x, target):
        output_error = (target - hidden_output2) #  -(target - y); 20 x 1

        # update hidden layer weights
        hidden_error = np.dot(self.weights2, output_error) # neccessary for h
        hidden_error = hidden_error * hidden_output1 * (1 - hidden_output1) #
        update1 = hidden_error*x[:, None] # multiply the input units as is pa

        # update output layer weights; linear not logistic
        update2 = output_error * hidden_output1 # use hidden layer outputs to
        return update2[:, None], update1

    ...
```

The output returns a tuple containing the updates for output and first hidden layers' weights. It should be noted that adding '[:, None]' to a 1-D array turns it to a 2-D array (matrix).

In the notes above, we used a matrix

$$X_i = \begin{bmatrix} \mathbf{x}_i^\intercal \\ \cdots \\ \mathbf{x}_i^\intercal \end{bmatrix}$$

however, we can get the same result (and save some work) by using only 1 array, $\mathbf{x}_i^\intercal$.

5.3 Gradient Descent

Gradient descent during the training step occurs with the gradient_descent method.

As with gradient descent, once the update is computed we update the weight with

$$w_{t+1} = w_t - \alpha \nabla F$$

where ∇F is the gradient

```
...

    def gradient_descent(self, delta_weights1, delta_weights2):
        self.weights1 += self.learning_rate * delta_weights1 # hidden layer w
        self.weights2 += self.learning_rate * delta_weights2 # output layer w
        return None

    ...
```

5.4 Training

The training function is fairly straight forward. You train each epoch by looping through each observation in the data set. For each observation, the function passes it through the 'forward' method to calculate the hidden outputs from both hidden layers. Then the outputs and data observations are passed through the backpropogation to obtain an update for the weights. However, the weights don't get updated immediately.

Updating the weights will only occur after we pass all observations through the neural network. Two zero vectors are created, 'delta_weights1' and 'delta_weights2'. These will contain all updates.

After all updates are calculated, they are then aggregated to the current weights to update weights for the next epoch.

```
...

    def train(self, n_epochs = 15):
        n_records = self.X.shape[0]
        for _ in range(n_epochs):
            delta_weights1 = np.zeros_like(self.weights1)
            delta_weights2 = np.zeros_like(self.weights2)
            for x, y in zip(self.X, self.y):
                hidden_output2, hidden_output1 = self.forward(X=x,
                    verbose=False)
                update2, update1 = self.backpropogation(hidden_output1, hidde
                    x=x, target=y)
```

```
                delta_weights1 += update1 / n_records
                delta_weights2 += update2 / n_records
            self.gradient_descent(delta_weights1, delta_weights2)
            yhat, _ = self.forward(X=self.X)
            mse_stat = MSE(y = self.y, yhat = yhat)
            #print("MSE: ", mse_stat)
        return yhat

...
```