Artificial Neural Networks: Matrix Form (Part 5)

December 14, 2014 in ml primers, neural networks

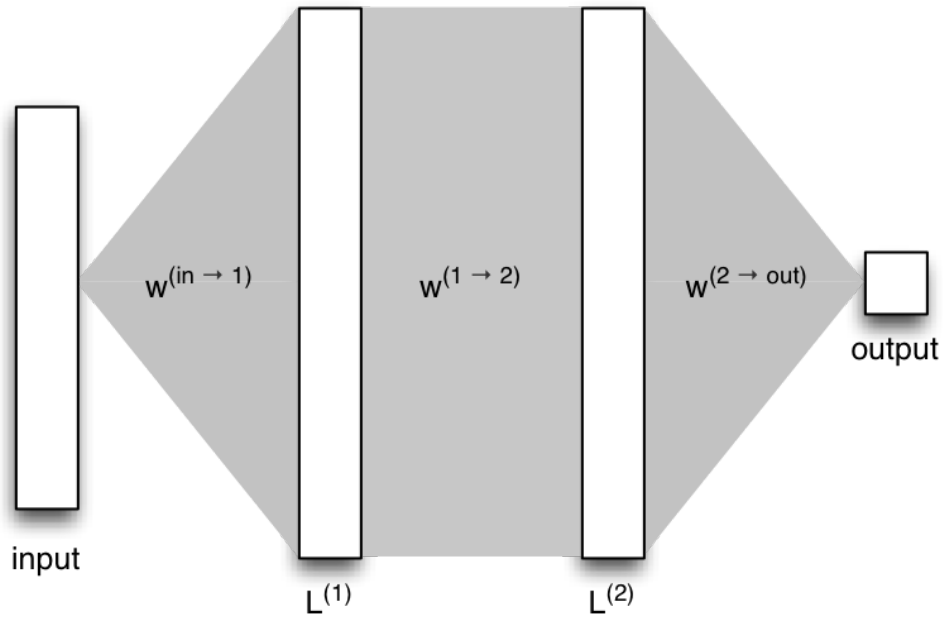
To actually implement a multilayer perceptron learning algorithm, we do not want to hard code the update rules for each weight. Instead, we can formulate both feedforward propagation and backpropagation as a series of matrix multiplies. This is what leads to the impressive performance of neural nets - pushing matrix multiplies to a graphics card allows for massive parallelization and large amounts of data.

This tutorial will cover how to build a matrix-based neural network. It builds heavily off of the theory in Part 4 of this series, so make sure you understand the math there!

Theory

The feedforward step

Consider the following network:



From now on, I'm going to index matrices by using $A^{(i)}$, where A refers to the type of matrix and (i) is an index of the position of the matrix in the network (we can also have $(i \rightarrow j)$ for a weight matrix connected layer i to layer j). The only exceptions are the input data matrix X and the output of the network $\hat{Y}$. I denote the value of an element in row i and column j of some matrix $A^{(k)}$ with $A^{(k)}_{ij}$.

Let's look at the low-level operations needed for the feedforward step as we go from the input X to the first layer $L^{(1)}$. If we want to compute the input for a single unit j in layer L_1 for a single example x, without using matrices, we would carry out the following operation:

$$s^{(1)}_j = \sum_i x_i w^{(in \rightarrow 1)}_{i \rightarrow j}$$

That is, for each component x_i in the example x, we multiply it by the weight that goes from unit i in the input layer to unit j in layer 1. If we want to compute the input to the hidden layer $L^{(2)}$, the only change is that we now include the activation functions of the units in $L^{(1)}$:

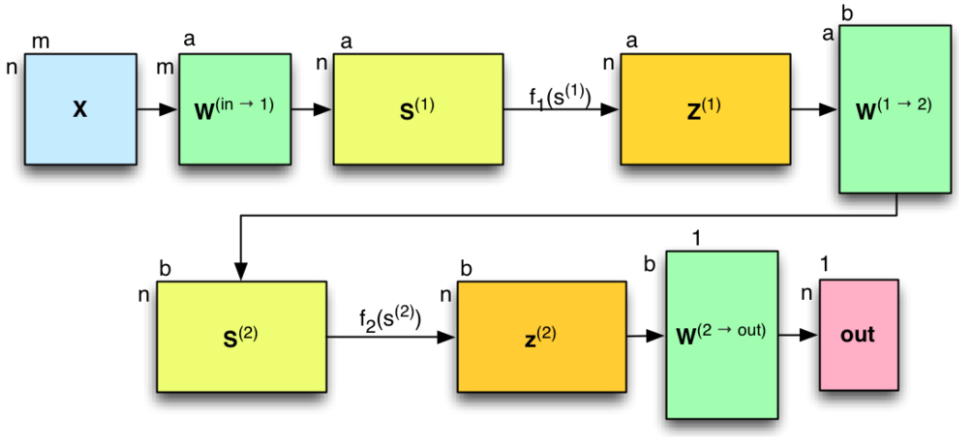
$$s^{(2)}_j = \sum_i f^{(1)}(s^{(1)}_i) w^{(1 \rightarrow 2)}_{i \rightarrow j}$$
$$s^{(2)}_j = \sum_i z^{(1)}_i w^{(1 \rightarrow 2)}_{i \rightarrow j}$$

However, for both of these cases, we must repeat this for every unit j in the layer. In addition, if we have n examples in a minibatch, then we also need to repeat these operations n times, which ultimately leads to a cubic complexity. Nested for loops are generally less efficient than using vectorized code (which can be optimized by handing it off to a BLAS library). A matrix multiply still has cubic complexity, but it is easier to

parallelize with a GPU.

Because of this, we will instead formulate our problem as a series of matrix multiplies, where we stuff all of our values into matrices and let the compiler decide how to optimize it. Typically, the rows in our data matrices correspond to items in a minibatch, while columns correspond to a values of these items (such as the value of s_i for *all* items in the minibatch). Weight matrices have the same number of rows as units in the previous layer, and the same number of columns as units in the next layer.

Consider instead the same network shown above, but with the weights, examples, and unit inputs and outputs all represented in matrix form:



Our input (in blue) is now represented with an $n \times m$ matrix, where we have n examples with m features each. The weight matrices are shown in green. Each row i of a weight matrix corresponds to the weights leading from unit i in the previous layer to all of the units j in the next layer. The layers themselves are shown in yellow and orange, with the S matrices representing the s_i's leading to each of the units i in a layer, and for each of the n examples. Applying the elementwise activation function to the matrix S generates the layer outputs Z.

To compute the input activations for $S^{(1)}$ during the feed forward step, we now use the following operation:

$$S^{(1)} = XW^{(in \to 1)}$$

Let's check the math here. $XW^{(in \to 1)}$ is an $n \times a$ matrix, so it matches the dimensions of $S^{(1)}$. Based on how we defined our matrices, the value of the element $S_{ij}^{(1)}$ corresponds to the activation of unit j in layer 1 for example x_i in our minibatch. In addition, by the definition of matrix multiplication:

$$S_{ij}^{(1)} = \sum_{k=1}^{m} X_{ik} W_{kj}^{(in \to 1)}$$

Now if we were to just examine a single row x in X (a single example in our minibatch), we could rewrite this as:

$$S_{ij}^{(1)} = \sum_{i=1}^{m} x_i W_{ij}^{(in \to 1)}$$

This is equivalent of forward propagation for the non-matrix case:

$$s_j^{(1)} = \sum_i x_i w_{i \to j}^{(in \to 1)}$$

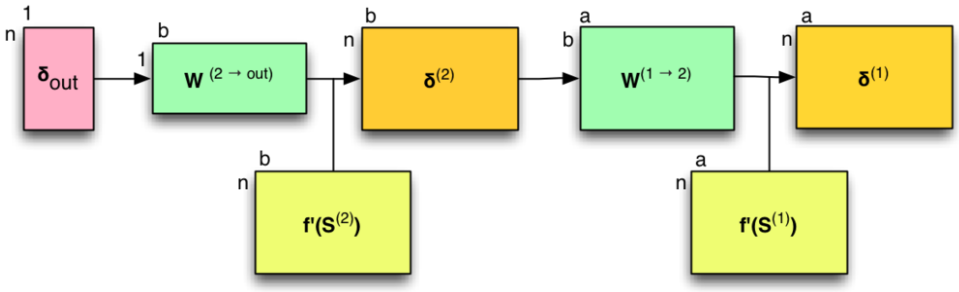
Thus, if we repeat this over all rows in X (i.e. perform a full matrix multiplication), we'll end up with all of the activations for layer 1 for each item in the minibatch. These values are stored in the matrix $S^{(1)}$.

Thus we can propagate the input forward through this network with a series of matrix multiplies:

$$\begin{aligned} S^{(1)} &= XW^{(in \to 1)} \\ Z^{(1)} &= f_1(S^{(1)}) \\ S^{(2)} &= Z^{(1)} W^{(1 \to 2)} \\ Z^{(2)} &= f_2(S^{(2)}) \\ \hat{y} &= f_{out}\left(Z^{(2)} W^{(2 \to out)}\right) \end{aligned}$$

Backpropagation

Recall that for backpropagation, for each of the weight updates, we need to calculate the error signal δ_i for each unit i in the network. Thus, the main thing we'll be concerned with during the matrix form of backpropagation is making sure we correctly compute each δ_i for each unit *and* for each item in the minibatch. We then use these error signals to compute the gradient. A schematic of backpropagation with matrices is shown here:



Recall that we computed (and stored) the inputs S and outputs Z of each layer during forward propagation. During the feedforward step, we should also compute the element-wise derivative of the activation functions for each layer. We create $D^{(out)}$ by computing δ_{out} for each of the output units and for each of the items in the minibatch. We then feed these deltas backward through the network, much like we did in the forward propagation step.

Again we should check the operations specified in the above schematic to make sure this setup is correct. Recall the general definition of δ_j for some hidden unit j:

$$\delta_j = f'_j(s_j) \sum_{k \in \text{outs}(j)} \delta_k w_{j \to k}$$

Now let's look at how we compute the matrix D_1 for the ith example in the minibatch (where $\odot$ denotes element-wise multiplication):

$$\begin{aligned} D^{(1)} &= F'^{(1)} \odot D^{(2)} W^{(1 \to 2)} \\ D^{(1)}_{ij} &= F'^{(1)}_{ij} \sum_{k=1}^{b} D^{(2)}_{ik} W^{(1 \to 2)}_{kj} \\ &= f'^{(1)}_{ij}(s^{(1)}_{ij}) \sum_{k \in \text{outs}(j)} \delta^{(2)}_{ik} w^{(1 \to 2)}_{j \to k} \end{aligned}$$

We see here that the matrix operations are computing the correct values.

Weight Updates

The final step is to compute the weight updates themselves. Recall the weight update rule that we derived in the last section:

$$\Delta w_{i \to j} = -\eta \sum_{x_i} \delta_j^{(x_i)} z_i^{(x_i)}$$

Let's derive the matrix form of the updates for the weight matrix $W^{(1 \to 2)}$. Recall that this matrix has dimensions $a \times b$, so we need to end up with a final matrix with the same dimensions so that we can perform an element-wise update of the weights. In addition, the updates use z_i (which are stacked in the matrix $Z^{(1)}$ with dimensions $n \times a$) and δ_j (which are stacked in matrix $D^{(2)}$ with dimensions $n \times b$). The matrix form of the weight update will be:

$$\begin{aligned} \Delta W^{(1 \to 2)} &= -\eta (Z^{(1)})^T D^{(2)} \\ \Delta W^{(1 \to 2)}_{ij} &= -\eta \sum_{k=1}^{n} (Z^{(1)})^T_{ik} D^{(2)}_{kj} \\ &= -\eta \sum_{x_i} \delta_j^{(x_i)} z_i^{(x_i)} \end{aligned}$$

Here we see that during the matrix mulitply, we multiply each $\delta_j^{(x_i)}$ by $z_i^{(x_i)}$ and sum over all of the examples in the minibatch. This is the exact weight update computed in the previous section.

Algorithm Summary

We've derived the matrix form of forward and backpropagation, as well as the weight updates themselves. The full neural network procedure can be summarized with the following algorithm:

1. Forward propagation:
   a. Propagate the input to the first layer with $S^{(1)} = XW^{(in \to 1)}$.
   b. For each of the hidden layers (that go from index i to j) compute $Z^{(i)} = f_i\left(S^{(i)}\right)$ and $S^{(j)} = Z^{(i)}W^{(i \to j)}$. Store the activation derivatives for the backpropagation step, $F^{(i)} = \left(f_i'(S^{(i)})\right)^T$. Also store $Z^{(i)}$.
   c. At the output, for the softmax activation function, we have $p(y_i = y) = Z_{iy}^{(out)}$.
2. Backpropagation:
   a. Set $D^{(out)} = (Z^{(out)} - Y)^T$.
   b. For each of the hidden layers going from index i to j, set $D^{(i)} = F^{(i)} \odot W^{(i)}D^{(j)}$.
3. Weight updates:
   a. For the weights going from the input to the output, use $\Delta W^{(in \to 1)} = -\eta(D^{(1)}X)^T$.
   b. For the weights leaving a hidden layer i going to layer j, use $\Delta W^{(i \to j)} = -\eta(D^{(j)}Z^{(i)})^T$.

Implementation

In this final section of the series, we'll go over the details on how to implement a fully extensible multilayer perceptron. Formulating the problem as a series of matrix multiplies makes the implementation straightforward - in fact, most of the code here is a direct translation from the theory section. Let's dive in!

Getting the code

You can either check out the latest version of my code from my machine learning Git repository here (the repository is still a bit unorganized), or I've created a stand-alone bundle that you can download here.

The Data

For this tutorial, I'll be using MNIST, which is a standard benchmark in the deep learning community. Most state of the art methods can achieve near-perfect performance on this dataset, so in practice other datasets are used for a fuller comparison, but MNIST is great for a sanity check. We can get pretty good results using a relatively small network.

MNIST is a collection of handwritten digits from 0 to 9, so this is a multiclass classification problem. Each instance is a 28 by 28 image unwrapped into a rows of 784 pixels each. You can download the dataset in a format that this code uses here, courtesy of deeplearning.net. The top-level script unpacks this data and chunks it into minibatches:

```
5  f = gzip.open('/path/to/mnist.pkl.gz')
6  train_set, valid_set, test_set = cPickle.load(f)
7  f.close()
8
9  minibatch_size = 100
10 print "Creating data..."
11 train_data, train_labels = create_minibatches(train_set[0], train_set[1],
12                                                minibatch_size,
13                                                create_bit_vector=True)
14 valid_data, valid_labels = create_minibatches(valid_set[0], valid_set[1],
15                                                minibatch_size,
16                                                create_bit_vector=True)
17 print "Done!"
```

Initializing the network

For this relatively simple example, the only tunable parameters are the network size and the minibatch size (which is typically set to 100). When using MNIST, the input layer must have 784 units and the output layer must have 10 units. You can specify a network size with a list in the form [784, A, B, 10], where A is the number of units in the first hidden layer and B is the number of units in the second. However, you are free to play around with the number of units (or even the number of layers)!

I have purposefully left out a tunable learning rate, because selecting the proper rate is an art in itself. Instead, I've found a learning rate that works for smaller 2-layer networks. The learning rate does not change during training, nor is there any "momentum," which are both techniques used for more complicated networks.

When we initialized our MLP, we create the input, hidden, and output layers in the file `neural_networks.py`:

```
39 class Layer:
40     def __init__(self, size, minibatch_size, is_input=False, is_output=False,
41                  activation=f_sigmoid):
42         self.is_input = is_input
43         self.is_output = is_output
44 
45         # Z is the matrix that holds output values
46         self.Z = np.zeros((minibatch_size, size[0]))
47         # The activation function is an externally defined function (with a
48         # derivative) that is stored here
49         self.activation = activation
50 
51         # W is the outgoing weight matrix for this layer
52         self.W = None
53         # S is the matrix that holds the inputs to this layer
54         self.S = None
55         # D is the matrix that holds the deltas for this layer
56         self.D = None
57         # Fp is the matrix that holds the derivatives of the activation function
58         self.Fp = None
59 
60         if not is_input:
61             self.S = np.zeros((minibatch_size, size[0]))
62             self.D = np.zeros((minibatch_size, size[0]))
63 
64         if not is_output:
65             self.W = np.random.normal(size=size, scale=1E-4)
66 
67         if not is_input and not is_output:
68             self.Fp = np.zeros((size[0], minibatch_size))
```

We have some edge cases to consider during this initialization - there are no error signals propagated to the input layer, and the output layer does not have any outgoing weights. Other than that, we initialize our matrices with the proper dimensions according to the algorithm we derived in the theory section. A technique that often helps is to randomly initialize the weights of our network, which we do when initializing the matrix `W`.

Activation functions

During training, we need to compute the activation function values for each layer in addition to their derivatives. I define global versions of these activation functions so if you wanted to use something other than a sigmoid, like a rectified linear function, it's relatively easy to plug them in.

```
26 def f_sigmoid(X, deriv=False):
27     if not deriv:
28         return 1 / (1 + np.exp(-X))
29     else:
30         return f_sigmoid(X)*(1 - f_sigmoid(X))
31 
32 def f_softmax(X):
33     Z = np.sum(np.exp(X), axis=1)
34     Z = Z.reshape(Z.shape[0], 1)
35     return np.exp(X) / Z
```

Note that we do not need to compute the derivative of the softmax function here. In fact, the error signal at the output is still $\delta_o = (\hat{y} - y)$, the same as in the linear case. This is because we use a 1-of-k coding

scheme, and we specify a different loss function than in the linear case (if you're interested in more information about this, read the discussion here).

Forward propagation

Forward propagation is a straightforward translation of the matrix multiplies we derived in the theory section. To begin, in the `MLP` class, we set the output of the input layer to the input data itself. For each of the successive layers, we perform forward propagation. I did not cover bias values in the theory section, but adding bias values is as simple as adding an additional hidden unit that always outputs 1, and connecting it to the next layer.

```
117 # We need to be sure to add bias values to the input
118 self.layers[0].Z = np.append(data, np.ones((data.shape[0], 1)), axis=1)
119
120 for i in range(self.num_layers-1):
121     self.layers[i+1].S = self.layers[i].forward_propagate()
122 return self.layers[-1].forward_propagate()
```

Within the `Layer` class, we perform the matrix multiply itself:

```
68 def forward_propagate(self):
69     if self.is_input:
70         return self.Z.dot(self.W)
71
72     self.Z = self.activation(self.S)
73     if self.is_output:
74         return self.Z
75     else:
76         # For hidden layers, we add the bias values here
77         self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
78         self.Fp = self.activation(self.S, deriv=True).T
79         return self.Z.dot(self.W)
```

At the input layer, there is no activation function, so we simply propagate the input data forward through the input layer's weights. If the layer is an output layer, we directly return the value of its output. Otherwise, if we are in a hidden layer, we append a bias weight, compute the activation function derivative, and return the next layer's inputs.

Backpropagation

Backpropagation is also fairly straightforward, and is carried out within the `MLP` class:

```
125 def backpropagate(self, yhat, labels):
126     self.layers[-1].D = (yhat - labels).T
127     for i in range(self.num_layers-2, 0, -1):
128         # We do not calculate deltas for the bias values
129         W_nobias = self.layers[i].W[0:-1, :]
130
131         self.layers[i].D = W_nobias.dot(self.layers[i+1].D) * \
132                            self.layers[i].Fp
```

We first compute the error signal at the output, $\delta_o = (\hat{y} - y)$. We then propagate this error signal backwards through each of the hidden layers.

Weight updates

After we've calculated the error signals for each layer, all that's left is to update the weights. Again, we directly use the formula given in the algorithm summary section:

```
134 def update_weights(self, eta):
135     for i in range(0, self.num_layers-1):
136         W_grad = -eta*(self.layers[i+1].D.dot(self.layers[i].Z)).T
137         self.layers[i].W += W_grad
```

Running the code

For the sake of space, I've omitted a large amount of boilerplate and evaluation code. To run the network, simply run the script `mnist_mlp.py`. You can change the layer configuration in the `MLP` initialization on line 20. Here's what the output looks like after an run with a layer configuration of [784, 100, 100, 10]:

```
1 Creating data...
2 Done!
```

```
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
Training for 50 epochs...
[   0]  Training error: 0.57362 Test error: 0.57510
[   1]  Training error: 0.09442 Test error: 0.08670
[   2]  Training error: 0.07088 Test error: 0.06610
[   3]  Training error: 0.04640 Test error: 0.04660
[   4]  Training error: 0.03870 Test error: 0.04260
...
[  46]  Training error: 0.00008 Test error: 0.02670
[  47]  Training error: 0.00006 Test error: 0.02660
[  48]  Training error: 0.00006 Test error: 0.02670
[  49]  Training error: 0.00004 Test error: 0.02690
```

You can see the training and test error slowly decreasing. Around epoch 50, we begin to severely overfit the training data. However, we're only making about 270 errors on the validation set, which isn't too shabby! To compare, Hinton's landmark dropout paper achieved about 130 errors, beating the previous record of 160 errors. More recent techniques have pushed the rate to well below 100 errors. For such a relatively simple network, we achieve good performance.

Conclusion

I hope this series helped your understanding and gave you a solid foundation to begin understanding more recent deep learning papers. Given enough time, I may write an additional bonus post on some technique used in practice to achieve even better performance (such as dropout, momentum, weight decay, etc.). If you have any questions, please leave a comment!

Tags: deep learning, neural network, machine learning, mnist

Prev / Next