Week 7 Homework

Bookmark this page

Week 7 Homework

(0.3076923076923077 / 1.0 points)

A code and data folder that will be useful for doing this lab can be found **here**. Download this to your computer, or alternatively, use the **colab notebook**.

This homework continues the exploration and implementation of neural networks as discussed in the notes.

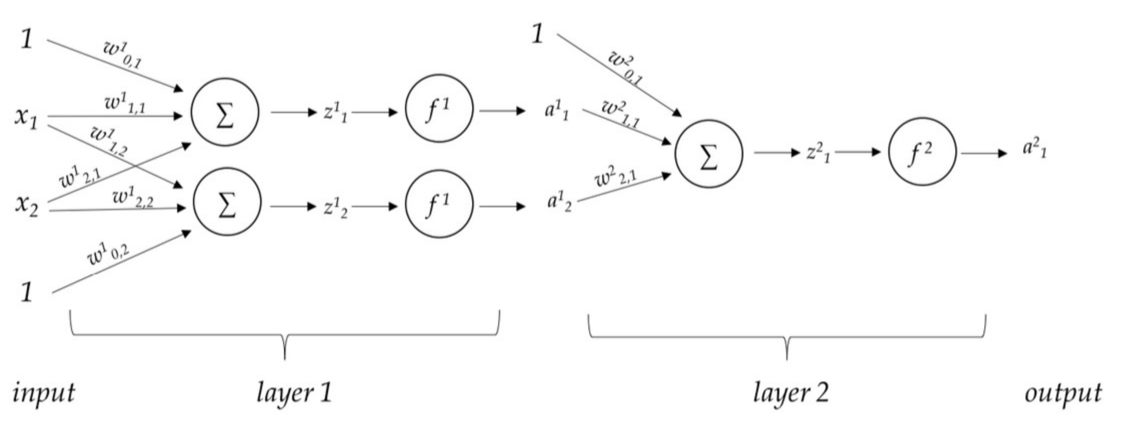
In particular, this homework considers neural networks with multiple layers. Each layer has multiple inputs and outputs, and can be broken down into two parts:

- A **linear** module that implements a linear transformation: $z_j = (\sum_{i=1}^m x_i W_{i,j}) + W_{0j}$ specified by a weight matrix W and a bias vector W_0. The output is $[z_1, \ldots, z_n]^T$.
- An **activation** module that applies an activation function to the outputs of the linear module for some activation function f, such as Tanh or ReLU in the hidden layers or Softmax (see below) at the output layer. We write the output as: : $[f(z_1), \ldots, f(z_m)]^T$, although technically, for some activation functions such as softmax, each output will depend on all the z_i, not just one.

We will use the following notation for quantities in a network:

- Inputs to the network are $x_1, \ldots, x_d$.
- Number of layers is L
- There are m^l inputs to layer l
- There are $n^l = m^{l+1}$ outputs from layer l
- The weight matrix for layer l is W^l, an $m^l \times n^l$ matrix, and the bias vector (offset) is W_0^l, an $n^l \times 1$ vector
- The outputs of the linear module for layer l are known as **pre-activation** values and denoted z^l
- The activation function at layer l is $f^l(\cdot)$
- Layer l activations are $a^l = [f^l(z_1^l), \ldots, f^l(z_{n^l}^l)]^T$
- The output of the network is the values $a^L = [f^L(z_1^L), \ldots, f^L(z_{n^L}^L)]^T$
- Loss function $Loss(a, y)$ measures the loss of output values a when the target is y

Here is an illustrative picture:



1) Backpropagation

The materials for week 6 and week 7 will be helpful here, including the week6 lecture and week7 lecture.

We have seen in the lecture notes how to train multi-layer neural networks as classifiers using stochastic gradient descent (SGD). One of the key steps in the SGD method is the evaluation of the gradient of the loss function with respect to the model parameters. In this problem, you will derive the backpropagation method for a general L-layer neural network. We'll exploit the decomposition of the network into *linear* and *activation* modules that we introduced at the start of this homework. Remember that we've defined the shapes of the various quantities at the start of the homework.

- Each linear module has a `forward` method that takes in a column vector of activations A (from the previous layer) and returns a column vector Z of pre-activations; it can also store its input or output vectors for use by other methods (e.g., for subsequent backpropagation).

- Each activation module has a `forward` method that takes in a column vector of pre-activations Z and returns a column vector A of activations; it can also store its input or output vectors for use by other methods (e.g., for subsequent backpropagation).

- Each linear module has a `backward` method that takes in a column vector $\frac{\partial Loss}{\partial Z}$ and returns a column vector $\frac{\partial Loss}{\partial A}$. This module also computes and stores $\frac{\partial Loss}{\partial W}$ and $\frac{\partial Loss}{\partial W_0}$, the gradients with respect to the weights.

- Each activation module has a `backward` method that takes in a column vector $\frac{\partial Loss}{\partial A}$ and returns a column vector $\frac{\partial Loss}{\partial Z}$.

The backpropagation algorithm will consist of:

- Calling the `forward` method of each module in turn, feeding the output of one module as the input to the next; starting with the input values of the network. After this pass, we have a predicted value for the final network output.

- Calling the `backward` method of each module in reverse order, using the returned value from one module as the input value of the previous one. The starting value for the backward method is $\partial Loss(a^L, y)/\partial a^L$, where a^L is the activation of the final layer (computed during the forward pass) and y is the desired output (the label).

1.1) Linear Module

The `forward` method, given A from the previous layer, implements:

$$Z = W^T A + W_0$$

and stores the input A to be used by the `backward` method.

Recall that there are $n^l = m^{l+1}$ outputs from layer l. For layer l, W is a $m^l \times n^l$ matrix, W_0 is a $n^l \times 1$ vector, and A from the previous layer is a $n^{l-1} \times 1$ (or $m^l \times 1$) vector. Given these shapes, make sure that you understand why the forward equation has W^T and not W.

The following questions ask for a matrix expression involving any of `A`, `Z`, `dLdA`, `dLdZ`,`W` and `W_0`.

Enter your answers as Python expressions. You can use `transpose(x)` for transpose of an array, and `x@y` to indicate a matrix product of two arrays. Remember that `x*y` denotes component-wise multiplication.

The `backward` method, given $\mathbf{dLdZ} = \partial Loss/\partial Z$ (an $n^l \times 1$ vector), returns $\mathbf{dLdA} = \partial Loss/\partial A$ (an $m^l \times 1$ vector):

$$\frac{\partial Loss}{\partial A} = \frac{\partial Z}{\partial A}\frac{\partial Loss}{\partial Z}$$

1.1.A)

dLdA= W @ dLdZ

Check Syntax | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Your entry was parsed as:

$$\mathbf{WdLdZ}$$

The `backward` method, given $\mathbf{dLdZ} = \partial Loss/\partial Z$, also computes $\mathtt{dLdW}$ (an $m^l \times n^l$ matrix) and $\mathtt{dLdW0}$ (an $n^l \times 1$ vector), and stores them in the module instance.

$$\mathtt{dLdW} = \frac{\partial Loss}{\partial W} = \frac{\partial Z}{\partial W}\frac{\partial Loss}{\partial Z}$$

1.1.B)

dLdW= A @ transpose(dLdZ)

Check Syntax | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Your entry was parsed as:

$$\mathbf{A}\left(\mathbf{dLdZ}\right)^T$$

and

$$\mathtt{dLdW0} = \frac{\partial Loss}{\partial W_0} = \frac{\partial Loss}{\partial Z}\frac{\partial Z}{\partial W_0}$$

1.1.C)

dLdW0= dLdZ

Check Syntax | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Your entry was parsed as:

$$\mathbf{dLdZ}$$

We will use `dLdW` and `dLdW0` as the gradient values in SGD.

1.2) Activation Module

Activation modules don't have any weights and so they are simpler.

The `forward` method for functions like *tanh* or *sigmoid*, given Z, return the function on the vector, componentwise. *Softmax* operates on the whole vector, as described earlier, and will need some special treatment.

The `backward` method, given $\mathbf{dLdA} = \partial Loss/\partial A$, returns:

$$dLdZ = \frac{\partial Loss}{\partial Z} = \frac{\partial Loss}{\partial A}\frac{\partial A}{\partial Z}$$

In this case, $m^l = n^l$ and the quantities are column vectors of that size.

For Softmax $= SM(Z)$ at the output layer and assuming that we are using *NLL* as the $Loss(A, Y)$ function, we have seen that there is a simple form for $\mathbf{dLdZ} = \frac{\partial Loss}{\partial Z}$; namely, it is the prediction error $A - Y$. A similar result holds when using *NLL* with a sigmoid output activation or a quadratic loss with a linear output activation. Note that for hinge loss with a linear activation, the form of `dLdZ` is different (see the lecture notes on the hinge loss).

2) Implementing Neural Networks

Please watch the lecture videos for this week before attempting this problem. Additionally, please review the SGD notes .

Although for "real" applications you want to use one of the many packaged implementations of neural networks (we'll start using one of those soon), there is no substitute for implementing one yourself to get an in-depth understanding. Luckily, that is relatively easy to do if we're not too concerned with maximum efficiency.

We'll use the modular implementation that we guided you through in the previous problem, which leads to clean code. The basic framework for SGD training is given below. We can construct a network and train it as follows:

```
# build a 3-layer network
net = Sequential([Linear(2,3), Tanh(),
                  Linear(3,3), Tanh(),
                  Linear(3,2), SoftMax()])
# train the network on data and labels
net.sgd(X, Y)
```

You will need to fill in the missing code. We encourage you to test in your own Python environment and then paste your answer and verify the results. The test cases are provided in the code distribution linked at the top of the page. **The code distribution includes additional test methods that will test each of the methods in turn, so you can debug incrementally.** Below are some hints for some of the methods:

- `Sequential.sgd`: Implement SGD. Randomly pick a data point Xt, Yt by using `np.random.randint` to choose a random index into the data. Compute the predicted output Ypred for Xt with the forward method. Compute the loss for Ypred relative to Yt. Use the backward method to compute the gradients. Use the sgd_step method to change the weights. Repeat.

- `SoftMax.class_fun`: Given the column vector of class probabilities for each point (computed by Softmax), this returns a vector of the classes (integers) with the highest probability for each point.

- We will (later) be generalizing SGD to operate on a "mini-batch" of data points instead of a single point. You should strive for an implementation of the forward, backward, and class_fun methods that works with batches of data. Whenever b is mentioned as part of the shape of a matrix in the code, this b refers to the number of points.

- **A note on debugging.** We have provided you with a file `code_for_hw7.py` (as well as a colab) that has a copy of the template below and a detailed set of outputs to check your implementation. **Trying to debug directly on MITx will not be a good experience; intermediate tests for each method are available ONLY in the code file/colab.**

```
class Module:
    def sgd_step(self, lrate): pass  # For modules w/o weights


class Linear(Module):
    def __init__(self, m, n):
        self.m, self.n = (m, n)  # (in size, out size)
        self.W0 = np.zeros([self.n, 1])  # (n x 1)
        self.W = np.random.normal(0, 1.0 * m ** (-.5), [m, n])  # (m x n)

    def forward(self, A):
        self.A = A   # (m x b)  Hint: make sure you understand what b stands for
        return np.transpose(A.T @ self.W) + self.W0  # Your code (n x b)

    def backward(self, dLdZ):  # dLdZ is (n x b), uses stored self.A
        self.dLdW = self.A @ np.transpose(dLdZ)       # Your code
        self.dLdW0 = np.sum(dLdZ, axis=1, keepdims=True)       # Your code
        return self.W @ dLdZ             # Your code: return dLdA (m x b)

    def sgd_step(self, lrate):  # Gradient descent step
        self.W -= lrate * self.dLdW           # Your code
        self.W0 -= lrate * self.dLdW0         # Your code


class Tanh(Module):            # Layer activation
    def forward(self, Z):
        self.A = np.tanh(Z)
        return self.A

    def backward(self, dLdA):    # Uses stored self.A
        return dLdA * (1 - self.A**2)             # Your code: return dLdZ


class ReLU(Module):              # Layer activation
    def forward(self, Z):
        self.A = np.where(Z <= 0, 0, Z)             # Your code
        return self.A

    def backward(self, dLdA):    # uses stored self.A
        return dLdA * (self.A != 0)             # Your code: return dLdZ


class SoftMax(Module):           # Output activation
    def forward(self, Z):
        exp_Z = np.exp(Z)
        return exp_Z / np.sum(exp_Z, axis=0)             # Your code

    def backward(self, dLdZ):    # Assume that dLdZ is passed in
        return dLdZ

    def class_fun(self, Ypred):  # Return class indices
        return np.argmax(Ypred, axis=0)             # Your code


class NLL(Module):       # Loss
    def forward(self, Ypred, Y):
        self.Ypred = Ypred
        self.Y = Y
        return -np.sum(Y * np.log(Ypred))      # Your code

    def backward(self):  # Use stored self.Ypred, self.Y
        return self.Ypred - self.Y      # Your code


class Sequential:
    def __init__(self, modules, loss):            # List of modules, loss module
        self.modules = modules
        self.loss = loss

    def sgd(self, X, Y, iters=100, lrate=0.005):  # Train
        D, N = X.shape
        sum_loss = 0
        for it in range(iters):
            i = np.random.randint(N)
            X_i = X[:, i:i+1]
            Y_i = Y[:, i:i+1]
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

Hint: these test cases are provided in the code distribution to help you debug