EECS 445 Homework 6: Neural Networks and Deep Learning

Due: 5:00 pm on December 13, 2016

Valli & Zhao

Homework Policy: Working in groups is fine, but each member must submit their own writeup and write their own code. Over-the-shoulder coding is prohibited. Please write the members of your group on your solutions. There is no strict limit to the size of the group but we may find it a bit suspicious if there are more than 4 to a team. For coding problems, please include your code and report your results (values, plots, etc.) in your PDF submission. You will lose points if your experimental results are only accessible through rerunning your code. Homework will be submitted via Gradescope (https://gradescope.com/). We accept late submissions but your score will be rescaled to $Score_{\text{rescaled}} = Score_{\text{origin}} * max(0, 1 - 0.05 * late_{\text{hour}})$.

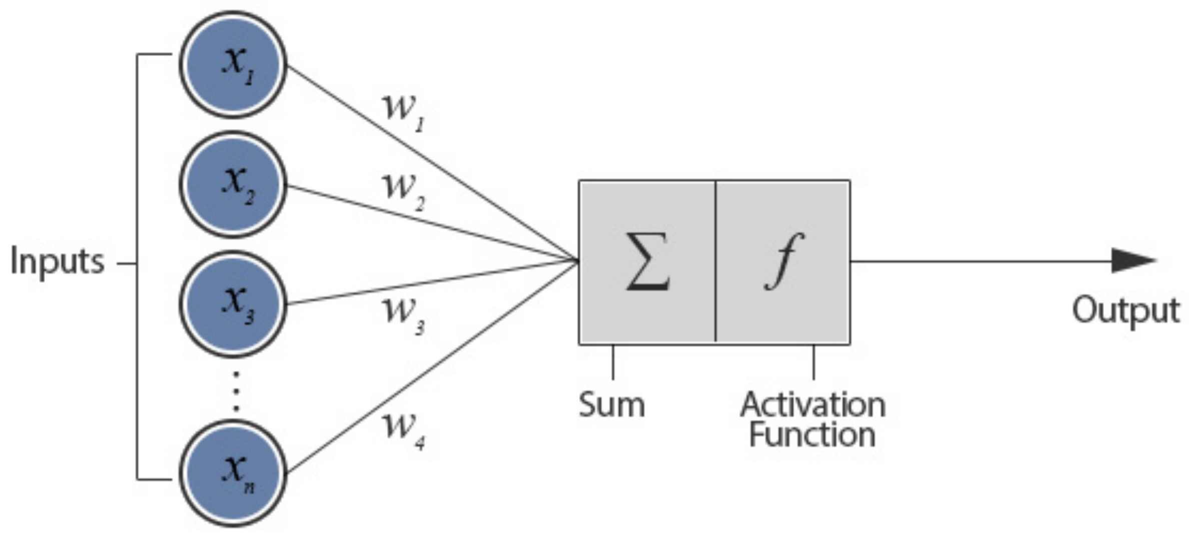
Question 1: Implementing a Fully Connected Layer

In this problem, we will begin by implementing a Fully Connected Layer, also often known as a Dense Layer, which we will then use to learn an averaging function.

An Artificial Neural Network (henceforth abreviated as ANN), and more specfically a Feedforward (i.e., acyclic) ANN, is composed of Layers in a sequence. The first layer is the input, the last layer is the output, and, layers in between the two are commonly termed "hidden" layers.

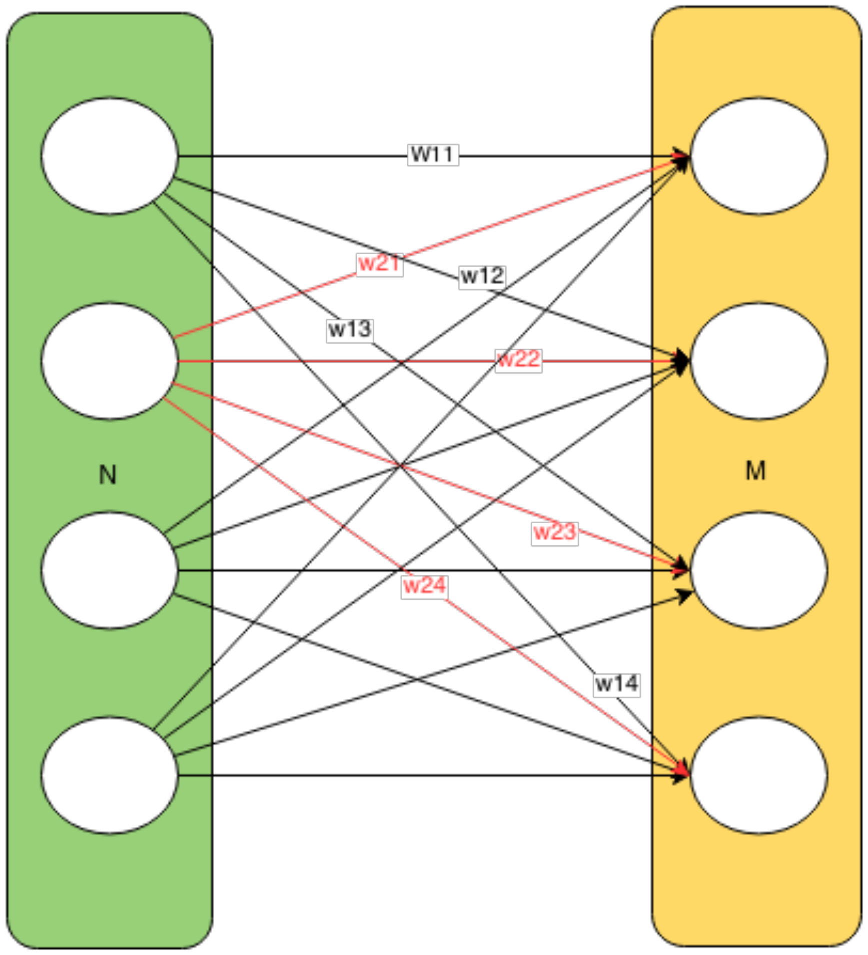
Layers in an ANN are composed of Neurons, which, in their simplest form at least, recieve some inputs, multiply them by some weights and sum the weighted inputs to produce their outputs. These outputs are then sometimes passed through nonlinearities to get "activations." Then finally these "activations" are passed to forthcoming layers as their inputs. That said, for this question, we will only be concerned with linear networks. In other words, the activation function we use is simply the identity (for an input x, the activation function returns x).

The above paragraph's description of a neuron can be illustrated through a diagram:



Task 1 (a) Implement the Linear Activation function and the Linear Activation Gradient. We will need these functions later.

Take a look at the partial implementation of a Fully Connected Layer (the FCLayer class in the code). As you will notice there, the initialization of the parameters has been given for you (we will talk more about this later). From the initialization code and the term "Fully Connected," it should be quite clear what a Fully Connected Layer looks like. A diagram and explanation is given below to make it clear what a Fully Connected Layer looks like:



As shown above, the Fully Connected Layer takes $N = 4$ inputs and produces $M = 4$ outputs. Every neuron is "connected" to every other, and so our weight matrix is of shape $N \times M$. Suppose the input is given by $[x_1, x_2, x_3, x_4]$, then $w_{11} \cdot x_1 + w_{21} \cdot x_2 + w_{31} \cdot x_3 + w_{41} \cdot x_4$ gives the input to the top neuron in the right above (the yellow rectangle). Normally, as in SVM for example, we also add a bias (to allow for modeling a more general class of functions.) So, we then have $w_{11} \cdot x_1 + w_{21} \cdot x_2 + w_{31} \cdot x_3 + w_{41} \cdot x_4 + b_1$. Note that there is an index on the bias as we will have 4 biases in the above diagram corresponding to each of the M neurons in the output, $[b_1, b_2, b_3, b_4]$.

Task 1 (b) Implement the `forward()` function in the FCLayer class. The only argument to this function is the input to the layer. Now, comment out Check 1 and run it. All checks should pass.

Note: The tests given are only simple sanity checks, and more may be used during grading. So, be sure to test your implementation.

Now, we will implement the MSE loss function and it's gradient which together will allow us to measure how good our predictions are with respect to a given target and help us in later updating our weights and learning through backpropagation.

Task 1 (c) Implement the `meanSquaredErrorLoss()` and `meanSquaredErrorLossDerivative()` functions. The arguments to these functions are the predictions of an ANN, y, and the target, t.

You may now run `check2()` to verify that the above functions are correctly implemented. Please include the output in your submission.

Now, we have to define an optimization step, i.e., a gradient descent step in particular, and then we can proceed to actually training a network later using backpropagation.

Given an input θ and some delta by which they should be changed $\Delta\theta$, an optimization step modifies θ in some way using $\Delta\theta$ with the aim of better performance on some loss function $L(\cdot;\theta)$.

We have already discussed gradient descent many times before, however, a brief refresher is given below.

We are concerned with gradient descent to minimize a loss function, and hence, assuming $\Delta\theta$ gives the gradient of the loss with respect to a given layer's weights, we change the weights of the layer in the direction opposite the gradient to reduce the loss.

The step size also matters a lot in how good a solution we find and how quickly we find it. We will control this, as is usual, with a learning rate, say γ. Thus, we have the vanilla gradient descent step:
$\theta \leftarrow \theta - \gamma \cdot \nabla_\theta L(\cdot;\theta)$.

Your first step is to implement this simple optimization step.

Task 1 (d) Implement the `get_parameter_update()` function in the `VanillaGradientDescent` class in `optim.py` where you are given an input, `x`, a delta, `dx`, and a learning rate, `lr`.

Now, to learn weights that will lead us to better model a desired input-output function, we will use backpropagation which was covered in the lectures. If you would like another refresher, take a look at this video: https://www.youtube.com/watch?v=H47Y7pAssTI.

Task 1 (e) Implement `backprop()`.

A few things to note:

- The arguments to this function are the previous layer's delta.
- You can use the member variable `_input` and `_output` to cache inputs or outputs to a layer. You

should not need to add any arguments to the backward function or member variables to the FCLayer class.

- Remember to return a delta for the next layer in the backpropagation step (i.e., for the previous layer when looking forward from the input layer.)
- Also, remember to update your parameters after you have calculated the

$$\delta$$

for the previous layer (when looking forward.)

- Run Check 3.

Question 2: Learning a simple function, Averaging

Task 2:

For this question, train a Fully Connected Layer to learn an averaging function (of 5 inputs.) Most of the explanations required are given in the code (see `train_averaging()` in `train.py`)

Note: For this question and the following questions, not much skeleton code is given and no checks are provided. You have more freedom about choosing how you implement certain functions. That said, you should be checking all your functions and code with your own test cases (we may grade your work later using our own test cases.)

Question 3: Constructing a Multilayer Perceptron

A sequence of Fully Connected Layers is called a Multi-Layer perceptron (deriving from the 1960s introduction of the term perceptron to talk about what we often generally call neurons today).

For this question you have two tasks:

Task 3 (a) Implement the MLP class. See the notes in `__init__` for more details.

Next, we will make training and evaluation easier by introducing another abstraction.

Task 3 (b) Implement the Manager class that makes training and evaluation easier. See the notes in the ` `Manager` class for more details.

Question 4: SoftMax Layer, Activation Functions, Modeling `XOR`

In Question 3, we work with a simple linear regression like problem. However, for classification problems, we normally use a SoftMax Layer. This is what you will implement first.

Task 4 (a) Implement a SoftmaxLayer much like FCLayer in the layers.py file (No skeleton code is given for this question).

Now, we still need a loss function for classification problems.

Task 4 (b) Implement the crossEntropy related functions in `loss_functions.py`

Now, all we have done till now is in the linear regime. To allow for non-linearities, which is very important when modeling non-linear functions or classifying non-linearly separable data, we need non-linear activation functions.

Task 4 (c) Implement all of the remaining activation functions in `activation_functions.py`.

Finally, we can use the above components to learn a classification model for a dataset that is known to be non-linearly separable, namely XOR.

Task 4 (d) Implement `train_XOR()` and try different networks. All networks you try should have two outputs that signify probabilities (of an input belonging in the 0 class or 1 class.)

Make sure to try a simple 1 FCLayer network (i.e., a network without hidden layers) and with a linear activation. Also, try a more complex network with some non-linear activations (and atleast 1 hidden layer.)

State any insights you find. Include plots (of the loss for example).

Question 5: Classifying Digits with a Multilayer Perceptron

To make gradient descent more smooth (i.e., to decrease oscillations) and to make gradient descent faster, it is often useful to use what is known as momentum. This simply means to use a previous update. This can be expressed as two steps shown below.

$$v = \mu v + \gamma \nabla_\theta L(\cdot; \theta)$$

$$\theta = \theta - v$$

The only addition to vanilla gradient descent is the new parameter μ which quantifies how much we are going to stick to the direction (the momentum) and previous update itself (v). If $\mu = 0$, we get vanilla gradient descent. If momentum is used, values of 0.95 through 0.99 are common (though sometimes smaller values or larger ones are used.)

Task 5 (a) Implement `get_parameter_update()` in the MomentumGradientDescent class.

Now, we are going to tackle a more useful, sizeable problem, classification of handwritten digits.

Some code is given for you to get the MNIST dataset with handwritten digit images and labels.

Task 5 (b) Train an MLP (using a Manager as well) to achieve at least 90% accuracy on the test set.

Question 6: Exploration

Explore neural nets and deep learning more!

Task 6:

After getting the Question 5 baseline model trained and working, we want you to do a basic experiment. Explore one of the features listed below and do analysis on the impact it has on your network. What effects are there on training time? Performance? Does it lead to overfitting, or maybe underfitting? There is no shortage of possible questions to ask. Training a neural network well is often described as an “art”, and in order to master this craft it is important to have some intuition of the effects of these parameters. There are so many, we cannot ask you to explore them all, so choose one and try to do some basic quantitative analysis. (This means we expect to see a chart or table with numerical results, not just a paragraph vaguely describing what you have observed.)

Possible options:

- Learning rate, protocols for lowering the learning rate as you train
- Weight decay (regularization)
- Network depth, i.e. the effect of adding or removing layers
- Layer width, i.e. changing the size of the layers
- Weight initialization (see FCLayer’s init.)
- Try implementing extra features (dropout)