CS 189 Introduction to Machine Learning
Fall 2017

HW7

This homework is due **Saturday, October 14 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

2 Backpropogation Algorithm for Neural Networks

In this problem, we will be implementing the backprop algorithm to train a neural network to approximate the function

$$f(x) = \sin(x)$$

To establish notation for this problem, the output of layer i given the input $\vec{a}$ in the neural network is given by

$$l_i(\vec{a}) = \sigma(W_i\vec{a} + \vec{b}_i).$$

In this equation, W is a $n_{i+1} \times m_i$ matrix that maps the input $\vec{a}_i$ of dimension m_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i+1$ and we have that $m_i = n_{i-1}$. The vector $\vec{b}_i$ is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives of the weights W_i and the biases $\vec{b}_i$ of each layer, we use the chain rule starting with the output of the network and work our way backwards through the layers, which is where the backprop algorithm gets its name.

You are given starter code with incomplete function implementations. For this problem, you will fill in the missing code so that we can train a neural network to learn the function $f(x) = \sin(x)$. The code currently trains a network with two hidden layers with 100 nodes per layer. Later in this problem, you will be exploring how the number of layers and the number of nodes per layer affects the approximation.

(a) **Start by drawing a small example network with three computational layers, where the last layer has a single scalar output.** The first layer should have a single external input corresponding to the input x. The computational layers should have widths of 5, 3, and 1 respectively. The final "output" layer's "nonlinearity" should be a linear unit that just returns its input. The earlier "hidden" layers should have ReLU units. Label all the n_i and m_i as well as all the as and W_i and b weights. You can consider the bias terms to be weights connected to a dummy unit whose output is always 1 for the purpose of labeling. You can also draw and label the loss function that will be important during training — use a squared-error loss.

Here, the important thing is for you to understand your own clear ways to illustrate neural nets. You can follow conventions seen online or in lecture or discussion, or you can modify those conventions to pick something that makes the most sense to you. The important thing is to have your illustration be unambiguous so you can use it to help understand the forward flow of information during evaluation and the backward flow during gradient computations. Since you're going to be implementing all this during this problem, it is good to be clear.

(b) Let's start by implementing the cost function of the network. This function is used to assign an error for each prediction made by the network during training. The implementation will be using the mean squared error cost function, which is given by

$$\text{MSE}(\hat{\vec{y}}) = \frac{1}{2}\sum_{i=1}^{n} ||y_i - \hat{y}_i||_2^2$$

where y_i is the observation that we want the neural network to output and $\hat{y}_i$ is the prediction from the network.

Write the derivative of the mean squared error cost function with respect to the predicted outputs $\hat{\vec{y}}$. In `backprop.py` implement the functions `QuadraticCost.fx` and `QuadraticCost.dx`

(c) Now, let's take the derivatives of the nonlinear activation functions used in the network. **Implement the following nonlinear functions in the code and their derivatives**

$$\sigma_{\text{linear}}(z) = z$$

$$\sigma_{\text{ReLU}}(z) = \begin{cases} 0 & z < 0 \\ z & \text{otherwise} \end{cases}$$

$$\sigma_{\tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

For the tanh function, feel free to use the tanh function in `numpy`. We have provided the sigmoid function as an example activation function.

(d) Now that we have the cost function and the nonlinear functions implemented, let's implement the network evaluation. **Implement the function `Model.evaluate` that takes the input to the network and outputs the predicted values $\hat{\vec{y}}$ as well as the outputs of each layer $l_i(\vec{a})$ and the values in each layer before applying the nonlinear function σ. See the comments in the code for further details.**

(e) We now need to compute the derivative of the cost function with respect to the weights W and the biases $\vec{b}$ of each layer in the network. We will be using all of the code we previously implemented to help us compute these gradients. **Assume that $\frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}}$ is given, where $\vec{a}_{i+1}$ is the input to layer $i+1$. Write the expression for $\frac{\partial \text{MSE}}{\partial \vec{a}_i}$ in terms of $\frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}}$. Then implement these derivative calcualtions in the function `Model.train`.**

(f) Finally, we use these gradients to update the model parameters using gradient descent. **Implement the function `GDOptimizer.update` to update the parameters in each layer of the network.** You will need to use the derivatives $\frac{\partial \text{MSE}}{\partial \vec{a}_i}$ and the outputs of each layer $\vec{a}_i$ to compute the derivates $\frac{\partial \text{MSE}}{\partial W_i}$ and $\frac{\partial \text{MSE}}{\partial \vec{b}_i}$. Use the learning rate η, given by `self.eta` in the function, to scale the gradients when using them to update the model parameters.

(g) **Show your results by reporting the mean squared error of the network when using the** *ReLU* **nonlinearity, the** tanh **nonlinearity, and the linear activation function. Additionally, make a plot of the approximated** sin **function in the range** $[-\pi, \pi]$**.** Use the model parameters, the learning rate, and the training iterations provided in the code to train the models. When you have all of the above parts implemented, you will just need to copy the output of the script when you run `backprop.py`.

(h) Let's now explore how the number of layers and the number of hidden nodes per layer affects the approximation. **Train a models using the tanh and the ReLU activation functions with** 5, 10, 25, **and** 50 **hidden nodes per layer (width) and** 1, 2, **and** 3 **hidden layers (depth)**. Use the same training iterations and learning rate from the starter code. **Report the resulting error on the training set after training for each combination of parameters.**

(i) **Implement a shortcut-training approach that doesn't bother to update any of the weights that are inputs to the hidden layers and just leaves them at the starting random values. All it does is treat the outputs of the hidden layers as random features, and does OLS+Ridge to set the final weights. Compare the resulting approximation by both plots and mean-squared-error values for the 24 cases above (2 nonlinearities times 4 widths times 3 depths). Comment on what you see.**

(j) **Bonus:** Modify the code to implement stochastic gradient descent where the batch size is given as a parameter to the function `Model.train`. **Choose several different batch sizes and report the final error on the training set given the batch sizes.**

(k) **Bonus:** Experiment with using different learning rates. Try both different constant learning rates and rates that change with the iteration number such as one that is proportional to 1/iteration. Feel free to change the number of training iterations. **Report your results with the number of training iterations and the final error on the training set.**

3 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.