Programming Assignment 2 Fall 2018

CSE 190: Deep Learning

Fall 2018

Instructions

Due on Tuesday, October 23rd, 2018

1. Please submit your assignment on Gradescope. There are two components to this assignment: written homework (Problems 1 & 2a-c), and a programming part. You will be writing a report in a conference paper format for the programming part of this assignment, reporting your findings. While we won't enforce this, we prefer the report to be written using LaTeX or Word in NIPS format (NIPS is the top machine learning conference, and it is now dominated by deep nets - it will be good practice for you to write in that format!). You are free to choose an alternate format, but NIPS format is strongly recommended. The templates, both in **Word** and LaTeX are available from the 2015 NIPS format site.

2. You need to submit all of the source codes files and a *readme.txt* file that includes detailed instructions on how to run your code.

   You should write clean code with consistent format, as well as explanatory comments, as this code may be reused in the future.

3. Using PyTorch, or any off-the-shelf code is strictly prohibited.

4. Any form of copying, plagiarizing, grabbing code from the web, having someone else write your code for you, etc., is cheating. We expect you all to do your own work, and when you are on a team, to pull your weight. Team members who do not contribute will not receive the same scores as those who do. Discussions of course materials and homework solutions are encouraged, but you should write the final solutions to the written part alone. Books, notes, and Internet resources can be consulted, but not copied from. Working together on homework must follow the spirit of the **Gilligan's Island Rule** (Dymond, 1986): No notes can be made (or recording of any kind) during a discussion, and you must watch one hour of Gilligan's Island or something equally insipid before writing anything down. Suspected cheating has been and will be reported to the UCSD Academic Integrity office.

Part I

Homework problems to be solved individually, and turned in individually

Multi-layer Neural Networks

In this problem, we will be classifying handwritten digits from Yann LeCun's MNIST Database. In Assignment 1, we classified the faces dataset using a single-layer neural network with different output activation functions. (Logistic and Softmax regression). In this assignment, we are going to classify the MNIST dataset using multi-layer neural networks with softmax outputs.

Problem

1. (5pts) In class we discussed two different error functions: sum-of-squared error (SSE) and cross-entropy error. We learned that SSE is appropriate for linear regression problems where we try to fit data generated from:

$$t = h(x) + \epsilon \tag{1}$$

Here x is a K-dimensional vector, $h(x)$ is a deterministic function of x, where x includes the bias x_0, and ϵ is random noise that has a Gaussian probability distribution with zero mean and variance σ^2, i.e. $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Suppose we want to model this data with a linear function approximation with parameter vector w:

$$y = \sum_{i=0}^{K} w_i x_i \tag{2}$$

Prove that finding the optimal parameter w for the above linear regression problem on the dataset $D = \{(x^1, t^1), ..., (x^N, t^N)\}$ is equal to finding the w^* that minimizes the SSE:

$$w^* = \text{argmin}_w \sum_{n=1}^{N} (t^n - y^n)^2 \tag{3}$$

2. (9pts) For multiclass classification on the MNIST dataset, we will use the cross-entropy error function and softmax as the output layer. Unlike assignment 1, we will add a hidden layer between the input and output, that consists of J units with the tanh activation function. So this network has three layers: an input layer, a hidden layer and a softmax output layer.

   Notation: We use index k to represent a node in output layer and index j to represent a node in hidden layer and index i to represent a node in the input layer. Additionally, the weight from node i in the input layer to node j in the hidden layer is w_{ij}. Similarly, the weight from node j in the hidden layer to node k in the output layer is w_{jk}.

   (a) (5pts) **Derivation** Derive the expression for δ for both the units of output layer (δ_k) and the hidden layer (δ_j). Recall that the definition of δ is $\delta_i = -\frac{\partial E}{\partial a_i}$, where a_i is the weighted sum of the inputs to unit i.

   (b) (2pts) **Update rule.** The update rule for the hidden to output weights (w_{jk}) is the same as it was for softmax regression, except that instead of x_i in the rule, it will be $h(a_j^n)$, the activation of the hidden unit feeding in to w_{jk}. Derive the update rule for w_{ij} using learning rate α, starting with the gradient descent rule:

$$w_{ij} = w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} \tag{4}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial w_{jk}} \tag{5}$$

   The derivative should take into account all of the outputs, so:

$$\frac{\partial E^n}{\partial a_k^n} = \sum_{k'} \frac{\partial E^n}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k^n} \tag{6}$$

   The answer you should get for the w_{ij} weights should be:

$$-\frac{\partial E^n}{\partial a_j^n} = \delta_j = h'(a_j^n) \sum_{k=1}^{c} \delta_k w_{jk} \tag{7}$$

   where $h'(a_j^n)$ is the derivative of the hidden unit activation function at the point a_j^n.

(c) (2pts) **Vectorize computation.** The computation is much faster when you update all w_{ij}s and w_{jk}s at the same time, using matrix multiplications rather than **for** loops. Please show the update rule for the weight matrix from the hidden layer to output layer and the matrix from input layer to hidden layer, using matrix/vector notation.

Part II

Team Programming Assignment

3. **Classification.** Classification on the MNIST datatbase. Refer to your derivations from Problem 2.

   (a) (0pts) Read in the MNIST data using the 'load_data' function provided in the code.

   (b) (3pts) Check your code for computing the gradient using a small subset of data. You can compute the slope with respect to one weight using the numerical approximation:

$$\frac{d}{dw}E(w) \approx \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon}$$

   where ϵ is a small constant, e.g., 10^{-2}. Compare the gradient computed using numerical approximation with the one computed as in backpropagation. The difference of the gradients should be within big-O of ϵ^2, so if you used 10^{-2}, your gradients should agree within 10^{-4}. (See section 4.8.4 in Bishop for more details). Note that w here is *one* weight in the network. You can only check one weight at a time this way - every other weight must stay the same.

   Choose one output bias weight, one hidden bias weight, and two hidden to output weights and two input to hidden weights, and show that the gradient obtained for that weight after backpropagation is within ($O(\epsilon^2)$) of the gradient obtained by numerical approximation. For each selected weight w, first increment the weight by small value ϵ, do a forward pass for one training example, and compute the loss. This value is $E(w+\epsilon)$. Then reduce w by the same amount ϵ, do a forward pass for the same training example and compute the loss $E(w-\epsilon)$. Then compute the gradient using equation mentioned above and compare this with gradient obtained by backpropagation. Report the results in a Table.

   (c) (7pts) Using the update rule you obtained from 2(c), perform gradient descent to learn a classifier that maps each input data to one of the labels $t \in \{0, ..., 9\}$, using a one-hot encoding. Use 50 hidden units. ***For this programming assignment, use stochastic gradient descent throughout, in all problems.***

   You should use momentum in your update rule, i.e., include a momentum term weighted by γ, and set γ to 0.9. You should use cross-validation to decide the stopping criteria: Stop training when the error on the validation set goes up, or better yet, save the weights as you go, keeping the ones from when the validation set error was at a minimum. You should experiment to determine how many epochs are necessary to do this (i.e., the holdout set error should go up at some point - this didn't always happen with the faces). Do this ten times, each time holding out a different 10% of the training data. You now have 10 numbers of epochs where the weights were best. Put those numbers in a Table, along with their average. Use the average number found in this manner to train your final network on all of the data for that number of epochs.

   Describe your training procedure and plot your training and testing accuracy (i.e., percent correct) vs. number of training epochs of gradient descent for the number of epochs determined by cross-validation. You only need to plot this for your final network, the one trained on all of the data.

   (d) (3pts) **Experiment with Regularization.** Starting with the network you used for part c, with new initial random weights, add weight decay to the update rule. (You will have to decide the amount of regularization, i.e., λ, a factor multiplied times the weight decay penalty. Experiment with 0.001 and 0.0001) Report training and testing accuracy vs. number of epochs of gradient descent. For this problem, train about 10% more epochs than you found in part c (i.e., if you found that 100 epochs were best, train for 110 for this problem). Comment on the change of performance, if any.

(e) (4pts) **Experiment with Activations.** Starting with the network of part c, try using different activation functions for the hidden units. You are already using tanh, try the other two below. Note that the derivative changes when the activation rule changes!!

i. Sigmoid. $f(z) = \frac{1}{1+e^{-z}}$
ii. ReLU. $f(z) = \max(0, z)$

The weight update rule is exactly the same for each activation function. The only thing that changes is the derivative of the activation function when computing the hidden unit δs. Report training and testing accuracy vs. number of training iterations of gradient descent. Comment on the change of performance.

(f) **Experiment with Network Topology.** Starting with the network from part c, consider how the topology of the neural network changes the performance.

i. Try halving and doubling the number of hidden units. How does performance change? Consider accuracy on the test set, explain your results.
ii. Change the number of hidden layers. Use two hidden layers instead of one. Create a new architecture that uses two hidden layers of equal size and has approximately the same number of parameters, as the previous network with one hidden layer of 50 units. By that, we mean it should have roughly the same total number of weights and biases. Report training and testing accuracy vs. number of epochs of gradient descent. How did performance change?

Instructions for Programming Assignment

The MNIST dataset has been randomly shuffled, split into training, validation and testing data and uploaded on Resources page in pickle format.

You need to edit the **neuralnet_starter.py** file to complete the assignment. This file is a skeleton code that is designed to guide you to build and implement your neural net in an efficient and modular fashion, and this will give you a feel for what developing models in PyTorch will be like.

A **config** dictionary is provided which has all the information necessary to build the model. The purpose of each flag is indicated in the comment next to it. Use this dictionary to decide the architecture, activation functions, etc. **Please do not add additional keys to it**.

The class **Activation** includes the definitions for all activation functions and their gradients, which you need to fill in. The definitions of 'forward_pass' and 'backward_pass' have been implemented for you in this class. The code is structured in such a way that each activation function is treated as an additional layer on top of a linear layer that computes the net input (a) to the unit. To add an activation layer after a fully-connected or linear layer, a new object of this class needs to be instantiated and added to the model.

The **Layer** class denotes a standard fully-connected/ linear layer. The 'forward_pass' and 'backward_pass' functions need to be implemented by you. As the name suggests, 'forward_pass' takes in an input vector 'x' and outputs the variable 'a'. Do not apply the activation function on the computed weighted sum of inputs since the activation function is implemented as a separate layer, as mentioned above. The function 'backward_pass' takes the weighted sum of the deltas from the layer above it as input, computes the gradient for its weights (to be saved in 'd_w') and biases (to be saved in 'd_b'). If there is another layer below that (multiple hidden layers), it also passes the weighted sum of the deltas back to the previous layer. Otherwise, if the previous layer is the input layer, it stops there.

The **Neuralnetwork** class defines the entire network. The '__init__' function has been implemented for you which uses the 'config' specifications to generate the network. Make sure to understand this function very carefully since good understanding of this will be needed while implementing 'forward_pass' and 'backward_pass' for this class. The function 'forward_pass' takes in the input dataset 'x' and targets (in one hot encoded form) as input, performs a forward pass on the data 'x' and returns the loss and predictions. The 'backward_pass' function computes the error signal from saved predictions and targets and performs a backward pass through all the layers by calling backward pass for each layer of the network, until it reaches the first hidden layer above the input layer (usually there will only be one hidden layer for this project, but when there are more, there will be more backward passes). The 'loss_func' function computes cross-entropy loss by taking in the logits (a fancy term for prediction y) and targets and returns this loss.

Additionally, you need to implement the **softmax**, **load_data**, **trainer** and **test** functions. The requirements for these functions and all other functions are given in the code.

Furthermore, a couple of things to take care of:

- **Do not** add additional keys in **config** dictionary.
- **Do not** modify the main function in the code.
- **Do not** modify the **checker.py** file. This is the file that has test cases for your code. You can download it and use it to verify your results to ensure your implementation is correct.
- You are allowed to write additional functions if you feel the need to do so.
- As such, the code is solvable using numpy and pickle libraries only. However, if you feel the need to use additional libraries, you can do so as long as they don't have implemented functions for backprop, etc. and you mention these dependencies in the Readme file. That said, make sure to include clear instructions in the Readme file to run your code. If you haven't done so and if we are not able to run your code using the instructions you provide, you lose points.