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 continue 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 previously used softmax regression with cross-entropy error as the objective function, and learned the weights of a single-layer network to classify the digits. In this assignment, we will add a hidden layer between the input and output, that consists of J units with the sigmoid 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.** Derive the update rule for w_{ij} and w_{jk} using learning rate α, starting with the gradient descent rule:

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

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

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

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

(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 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.

   Choose any 10 weights at random, and show that the gradient obtained for that weight after backpropagation is within acceptable region ($O(\epsilon^2)$) 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 all N samples and compute the loss. This value is $E(w+\epsilon)$. Then reduce w by the same amount ϵ, do a forward pass for all N samples and compute the loss $E(w-\epsilon)$. Then compute the gradient using equation mentioned above and compare this with gradient obtained by backpropagation.

   (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 $t \in \{0, ..., 9\}$ (but using a one-hot encoding). Choose your own stopping criteria. 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. Report your training procedure and plot your training and testing accuracy vs. number of training iterations of gradient descent. Again, by *accuracy*, we mean the percent correct on the training and testing patterns.

   (d) (3pts) **Experiment with Regularization.** Starting with the network of 3(a), 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 training iterations of gradient descent. Comment on the change of performance, if any.

   (e) (3pts) **Experiment with Momentum.** Start with the network of 3(a). Add a momentum term weighted by γ to the update rule, and set γ to 0.9. Report training and testing accuracy vs. number of training iterations of gradient descent. Comment on the change of performance.

   (f) (4pts) **Experiment with Activations.** Start with the network of 3(a). Try using different activation functions for the hidden units. You are already using the sigmoid, try the other two below. Note that the derivative changes!

      i. Sigmoid. $f(z) = \frac{1}{1+e^{-z}}$

      ii. Tanh. $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

      iii. ReLU. $f(z) = \max(0, z)$

   Derive the update rule for each activation function. Report training and testing accuracy vs. number of training iterations of gradient descent. Comment on the change of performance.

   (g) **Experiment with Network Topology.** Start with the network of 3(a). Now, we will consider how the topology of the neural network changes the performance.

i. Try halving and doubling the number of hidden units. What do you observe if the number of hidden units is too small? What if the number is too large?

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. By that, we mean it should have roughly the same total number of weights and biases. Report training and testing accuracy vs. number of training iterations of gradient descent.

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 of how developing models in PyTorch will be like.

A **config** dictionary is provided which has all the information necessary to build the model. Purpose of each flag is indicated in the comment against it. Use this dictionary to decide the architecture, activation functions, whether you want to use momentum or not, etc.

The class **Activation** has function definitions for all activation functions and gradients through those functions. You need to fill in these function definitions. 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 a separate layer. So to add the activation layer after a fully-connected/ linear layer, a new object of this class needs to be made and added to the model.

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 activation function on the computed weighted sum of inputs since activation is implemented as a separate layer, as mentioned above. Teh function 'backward_pass' takes in gradient from its next layer as input, computes the gradient for its weights (to be saved in 'd_w') and biases (to be saved in 'd_b'), and also computes the gradient with respect to input for that layer (to be saved in 'd_x'). This 'd_x' variable is returned by this function which in turn will be the input for backward pass for its previous layer.

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. 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.
- **Do not** shuffle the data again since it has already been shuffled and we will evaluate your code with values we have obtained on this shuffled data only.

- 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.