Programming Assignment 2

CSE 251B: Neural Networks for Pattern Recognition

Winter 2022

Instructions

Due on February 3, 2022

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. **The report should be written using LaTeX or Word in NeurIPS format**. The templates, both in **Word** and LaTeX are available from the 2015 NIPS format site.

2. For the programming part, you are allowed to work in groups of 2 or 3. Again, **don't forget to include a paragraph for each team member in your report that describes what each team member contributed to the project.** We expect ***everyone*** to do some of the coding! Just writing the report is not enough!

3. 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. **Do not submit any of your output plot files or .pyc files, just the .py files and a readme that can reproduce your findings.**

4. For the programming assignment, you are expected to write your code using NumPy. Using any machine learning frameworks (PyTorch, Tensorflow, Keras, etc.) which compute the gradients for you is not allowed for this assignment.

5. 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.

Multi-layer Neural Networks

In this assignment, we will be classifying images from the CIFAR-10 Dataset (https://www.cs.toronto.edu/~kriz/cifar.html). In Assignment 1, we classified traffic signs from the German Traffic Sign Recognition Benchmark dataset using stochastic and batch gradient descent to update the weights in logistic and softmax regression. In this assignment, we will use multi-layer neural networks with softmax outputs for classification.

Part I

Homework problems to be solved individually, and turned in individually

For this part we will not be accepting handwritten reports. Please use latex or word for your report. MathType is a handy tool for equations in Word. The free version (MathType Lite) has everything you need. This should be done individually, and each team member should turn in his or her own work separately.

1. (5 pts) **Maximum Likelihood Estimation**. An exponential distribution can be completely characterized by the parameter λ . Given n samples $[x_1, x_2, ..., x_n]$ drawn independently from an exponential distribution, find the maximum likelihood estimate for the parameter λ. Recall that the exponential distribution is given by

$$p(x; \lambda) = \lambda \exp(-\lambda x) \tag{1}$$

2. (15pts) **For multiclass classification on the CIFAR-10 dataset, we will use the cross-entropy loss as our objective function and softmax as the output layer**. In our network, we will have 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. **For the purpose of this homework part of the assignment, you will be using one hidden layer, but the programming part of the assignment (Problem 3) will require you to use more than one hidden 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) (10pts) **Derivation.** In the following discussion, n denotes the nth input pattern. Derive the expression for δ for both the units of output layer (δ_k^n) and the hidden layer (δ_j^n). Recall that the definition of δ is $\delta_i^n = -\frac{\partial E^n}{\partial a_i^n}$, where a_i^n is the weighted sum of the inputs to unit i. There are two "hard parts" to this: 1) taking the derivative of the softmax; and 2) figuring out how to apply the chain rule to get the hidden deltas. Bishop and Chapter 8 of the PDP books both have good hints on the latter, and Bishop on the former. However, crucial steps have been left out of the Bishop derivation (Chapter 6). Our main hint here is: break it up into two parts (see equation 6.161 in Bishop), when $k = k'$ and when it doesn't. Note that Bishop (Equation 4.31) defines δ_j^n without a minus sign, which is the opposite of the way that we defined it above, and different from the PDP book chapter 8.

   (b) (2pts) **Update rule.** Write the update rule for w's in terms of the δ's you derived above using learning rate α, starting with the gradient descent rule:

$$w_{ij} = w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} \tag{2}$$

   where

$$\frac{\partial E}{\partial w_{ij}} = \sum_n \frac{\partial E^n}{\partial w_{ij}} \tag{3}$$

   You have to write both the update rules, the hidden to output layer (w_{jk}) update rule and the input to hidden (w_{ij}) update rule in a generalized form. (Hint: you will have to use chain rule for differentiation.)

$$\frac{\partial E^n}{\partial w_{ij}} = \frac{\partial E^n}{\partial a_j^n} \frac{\partial a_j^n}{\partial w_{ij}} \tag{4}$$

   (c) (3pts) **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 CIFAR-10 datatset. Refer to your derivations from Problem 2.

   (a) (0pts) Implement the `normalize_data` and `one_hot_encoding` methods provided in the starter code and read in the CIFAR-10 data using the `load_data` method. This will load the training and testing data for you. Create a validation split from the training data. **This time, to save time, we will only do one-fold cross-validation.**

   Normalize the data by z-scoring it. That is, we will compute the average image over every image in X_{train}, as well as the standard deviation of each of the pixels, and normalize by subtracting the mean, and dividing by the standard deviation. In this assignment, you are dealing with $32x32$ color images, which have three channels: one for red one for green, and one for blue. When normalizing your data, you should normalize each channel separately. We will describe why this is a good idea in class, but if you want to look ahead, read the lecture on tricks of the trade, and/or read the reading "lecun98efficient.pdf" Now, using the train data's mean and standard deviation, z-score the validation and test sets as well.

   Another way to z-score would be for each image separately. I.e., in this case, you would compute the average and standard deviation of pixels *within* an image. Feel free to try this approach also! Note this is something you can do online, whereas the first version you have to have the whole dataset to do.

   (b) (5pts) Check your code for computing the gradient using a small subset of data. For example, in computing $E(w + \epsilon)$ below, you could use 10 examples, one from each category. 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 $E(w + \epsilon)$ is the loss computed over those 10 examples with the weight incremented by ϵ, and 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 for each hidden layer, 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 the 10 training examples, and compute the loss. This value is $E(w+\epsilon)$. Then subtract ϵ from w, and do a forward pass for the same 10 training examples and compute the loss $E(w - \epsilon)$. Then compute the gradient with respect to w using the approximation given above and compare this with gradient obtained by backpropagation (you will have to add the 10 numbers you get from backprop, one for each example). Report the results in a Table.

   (c) (9pts) Using the vectorized update rule you obtained from 2(c), perform gradient descent to learn a classifier that maps each input image to one of the labels $t \in \{0, ..., 9\}$, using a one-hot encoding. ***Now use 2 hidden layers, each with 64 units, and tanh activation for the hidden layers, instead of one hidden layer. For this programming assignment, use mini-batch 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 the validation set for early stopping of your training: stop the training when the error on the validation set goes up. Use the following criteria - If the validation error goes up for some *patience* number of epochs, stop training and save the weights which resulted in minimum validation error. The validation set error should go up at some point. Another method is to train the network for a certain maximum number of epochs, saving the weights that give the best performance on the holdout set as you go. The *patience* parameter could be 5 for example.

In the Methods section of your report, describe your training procedure. Plot your training and validation accuracy (i.e., percent correct) vs. number of training epochs, as well as training and validation loss vs. number of training epochs. Report accuracy on test set using the best weights obtained through early stopping.

You may experiment with different learning rates, minibatch sizes, etc., but you only need to report your results and plots on the best learning rate you find. For the default setting with learning rate 0.005, minibatch size 128, and two hidden layers with 64 units each and tanh activation, you can expect an accuracy around 37%.

(d) (4pts) **Experiment with Regularization.** Starting with the network you used for part (c), with new initial random weights, add weight decay (L_2 regularization) 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 $1e-2$, $1e-3$, and $1e-4$.) Again, plot training and validation loss, training and validation accuracy, and report final test accuracy. For this problem, train for 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) (6pts) **Experiment with Activations.** Starting with the best network of part (d) (*please explicitly mention learning rate, and amount of regularization* λ) try using different activation functions for the hidden units. Here, the best network means the network architecture and learning rate that gave you the best test accuracy. You are already using tanh, so try the other three below. Note that the derivative changes when the activation rule changes!!

$$sigmoid(z) = \frac{1}{1+e^{-z}} \tag{5}$$

$$ReLU(z) = \max(0, z) \tag{6}$$

$$leakyReLU(z) = \max(0.1z, z) \tag{7}$$

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 δ's. For each activation function you try, plot training and validation loss on one graph, training and validation accuracy on another, and report final test accuracy. Comment on the change of performance. How do these activation functions compare to each other? What works best for your problem?

(f) (6pts) **Experiment with Network Topology.** Starting with the network from part (e), (using the best learning rate, regularization parameter λ, and activation function), consider how your neural network architecture changes the performance in the following scenarios (*please explicitly mention learning rate, amount of regularization* λ*, and the activation function used*):

i. Try halving and doubling the number of hidden units in each layer (try the combinations 64×128, 128×64, 32×64, and 64×32). Plot training and validation loss, training and validation accuracy, and report final test accuracy. How does performance change? Explain your results.

ii. Change the number of hidden layers. Use three hidden layers instead of two, with approximately the same number of parameters as the previous network with two hidden layers of 64 units. By that, we mean it should have roughly the same total number of weights and biases. This is a standard method for comparing different neural network architectures in order to make a fair comparison. You can also experiment with more nodes and layers and see how performance changes. Again, plot training and validation loss, training and validation accuracy, and report final test accuracy. How did the performance change?

iii. Now use one hidden layer instead of two, maintaining approximately the same number of parameters as the network with two hidden layers of 64 units. Plot the training and validation loss, as well as training and validation accuracy, and report final test accuracy. How does the performance change with changes in the width and depth of the network in all three parts?

Instructions for Programming Assignment

The CIFAR-10 dataset and starter code has been provided to you on Piazza.

You need to edit the **neuralnet.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.

Follow instructions in the code on how to install `PyYAML`. The **config.yaml** specifies the configuration for your Neural Network architecture, training hyperparameters, type of activation, etc. The purpose of each flag is indicated by the comment before it. Play around with the parameters here to decide what works best for the problem.

For your convenience, each function that you need to implement has a 'TODO' in its docstring. However, if you need to add new functions, please feel free to do so.

The class **Activation** includes the definitions for all activation functions and their gradients, which you need to fill in. The definitions of `forward` and `backward` 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` and `backward` methods need to be implemented by you. As the name suggests, 'forward' 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 'backward' method 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. If the previous layer is the input layer, it stops there.

The **Neuralnetwork** class defines the entire network. The `__init__()` method has been implemented for you which uses the configuration specified in `config.yaml` to create the network. Make sure to understand this function very carefully since good understanding of this will be needed while implementing `forward` and `backward` for this class. The function 'forward' 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' 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. The `loss` method computes cross-entropy loss using the predictions and targets and returns this loss.

All of the classes above implement a `__call__()` method which calls the class's forward method by default. Recall that the `__call__()` method allows instances of the class to be callable. This will make it really easy to appreciate PyTorch API's later on as they are also very similar.

The **load_data** method has been partially implemented for you. You need to implement the **softmax**, **normalize_data**, **train** and **test** functions. The requirements for these functions and all other functions are given in the code.

Once you have implemented everything, you can run **checker.py** to verify your implementation. The checker.py file uses `sanity.pkl` to verify your results using the default configuration provided (in `config.yaml`). You might want to save a copy of the default configuration before you begin experimenting with it.

Furthermore, a few things to take care of:

- **Do not** add additional keys in **config.yaml**.
- **Do not** rename the classes.
- **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.
- 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.

General hints for this PA: The whole training procedure for all parts of this assignment takes us about 30 minutes on a conventional laptop. We achieve test set accuracies from 37% for the vanilla method to 52%, depending on the activation, regularization, and network topology used. You don't need to exactly match those accuracies. The purpose of this assignment is to help you understand the impact of different techniques, so it is most important to us that you explain your results and what causes them. However, if you think a bug might have caused a significant difference in performance, you should look into it and try to fix it.

1 Project Report Outline with Rubric (32 Points)

1. Title (0 pts): The title *has to be informative* and not generic like '251B PA2 Report'. Additionally, please include a list of authors.
2. Abstract (1 pt): A short description of what you did. Please mention any key findings, interesting insights, and final results (i.e., percent correct, *not* loss numbers)
3. 3b (5 pts): Follow instructions from 3b in the write up.
4. 3c (10 pts): Follow instructions from 3c in the write up.
5. 3d (5 pts): Follow instructions from 3d in the write up.
6. 3e (5 pts): Follow instructions from 3e in the write up.
7. 3f (5 pts): Follow instructions from 3f in the write up.
8. (1 pt) Team contributions: A short paragraph from *each* team member with what they contributed to the project - team members won't necessarily get the same grade if someone slacked off!
9. For the source code - please make sure you submit clean, well-documented code.

2 Submission

Submission for all 3 parts (report, code, individual) will be done through Gradescope. Report and code both only need to be submitted by one team member - all other team members should be added to the submission.