Programming Assignment 1

CSE 151B/251B: Deep Learning

Winter 2024

Instructions

Due on Thursday, January 25, 2024 (11:59 PM)

1. Please submit your assignment on Gradescope. The instructions for this will be coming soon. There are two components to this assignment: written homework (Part I). For the programming assignment portion of the homework (Part II), you will be writing a report in a conference paper format, reporting your findings. All parts of the assignments must be typeset, including figures. You must use NeurIPS format for your report (link below; NeurIPS 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!). We strongly recommend that you use some dialect of TEXor LATEX. You may also use Word if you so choose. The link below has both LATEXand Word NeurIPS formats. Figures may be generated with Excel or Python, so long as they are computer generated. **We will not be accepting any handwritten work - this includes the "written part."** NeurIPS templates in LATEXand Word are available from the 2023 NeurIPS format site (or you can use the given TEXfile in Piazza resources). The page limits mentioned there don't apply.

2. You are expected to use Python (usually with NumPy). You also need to submit all of the source code files and a *README.md* file that includes detailed instructions on how to run your code.

   You should write clean code with consistent format, as well as explanatory comments. Do not submit any of your output plot files or .pyc files, just the .py files and a README.md that explains how to run your code.

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

4. ***If you end up dropping the class and your teammate does not, you are expected to help your teammate anyway!*** Please don't leave your teammate(s) without your assistance. Being on a team means just that: teamwork! When you join a team, you have made a commitment. Please honor it.

5. Any form of copying, plagiarizing, grabbing code from the web, having someone else write your code for you, etc., is cheating. As stated in the syllabus, you may use chatgpt or copilot, but you must indicate in the text or code where you used it, and include an appendix with the prompt you used, the output of the AI model, and how you modified it, if you did. Furthermore, you must **understand your code**, so that you can explain it in a code review. 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

Problems to be solved 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 - unfortunately, it is not compatible with recent updates by Microsoft, so you will have to use the equation editor in Word, which kind of sucks. This might be a good time to learn LaTeX! If you decide to use LaTeX, I highly recommend using Overleaf, a free online editor. Alternatively, 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. **Perceptrons. (7 points)**

   In class, we showed how to learn the "OR" function using the perceptron learning rule. Now we want to learn the "NAND" function using four patterns, as shown in Table 1.

Input	Output
0 0	1
0 1	1
1 0	1
1 1	0

Table 1: The "NAND" function

   (a) (1 pt) Write down the perceptron learning rule as an update equation.

   (b) (4 pts) Draw the rest of this table, as we did in class, as the network learns NAND. Initialize w_0, w_1, and w_2 to be 0 and fix the learning rate to 1. Add one row for each randomly selected pattern (training example) for the perceptron to learn. Stop when the learning converges. Make sure you show the final learned weights and bias. You may pick a "random" order to make the learning converge faster, if you can (you may not need all of these rows).

x_1	x_2	Output	Teacher	w_0	w_1	w_2
1	1	1	0	0	0	0
0	0	0	1	-1	-1	-1
0	1	0	1	0	-1	-1

   (c) (2 pts) Is the solution unique? Why or why not? Justify your answer.

2. (15 pts) **For multi-class classification on the MNIST dataset, we will use the cross-entropy error 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.

   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) **(CSE 251B Required; CSE 151B Extra Credit) Derivation. (5 pts)** In the following discussion, n denotes the nth input pattern.(see "Notation and Nomenclature" under General Resources). Recall that the definition of δ is $\delta_j^n = -\frac{\partial E^n}{\partial a_j^n}$, where a_j^n is the weighted sum of the inputs to unit j for pattern n. Show that if our output activation function is softmax and the hidden layer activation function is $tanh$ then, for the output layer, $\delta_k^n = t_k^n - y_k^n$ and for the hidden layer, $\delta_j^n = (1 - tanh^2(a_j^n)) \sum_k (t_k^n - y_k^n)\, w_{jk}$. Use the cross-entropy cost function in your derivation:

$$E^n = -\sum_{i=1}^{c} t_i^n \ln y_i^n \tag{1}$$

Hint: 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 different than we defined it above, and differently than the PDP book chapter 8.

(b) (5 pts) **Update rule.** Write the update rule for w's in terms of the δ's (given 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

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

and E^n represents the error for example n.

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.) When we say "generalized form," we mean that here, you can leave the output delta, i.e., "$-\frac{\partial E}{\partial a_k^n}$". as simply δ_k^n. I.e., that derivation is the extra credit above. Recall you start with this:

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

(c) (5pts) **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.

(d) (5pts) Consider the simple network below. The initial weights and biases are as shown (biases are shown as a weight from units with a "1" inside). All units use the ReLU activation function $g(a) = \max(a, 0)$. Assume the δ at the outputs is simply $t - y$. The inputs, X_1 and X_2 are both 1 .

i. Compute the activations of the neurons : h_1, h_2, y_1, y_2 (after applying ReLU)

ii. Compute the deltas of the 4 nodes computed via backprop in the Figure in the space provided. If a ReLU unit's activity is 0 , you can assume the slope is 0.

iii. Now, if any weights change, update it and show the new values, else re-write the original value. Assume the learning rate is 1.

$h_1 =$

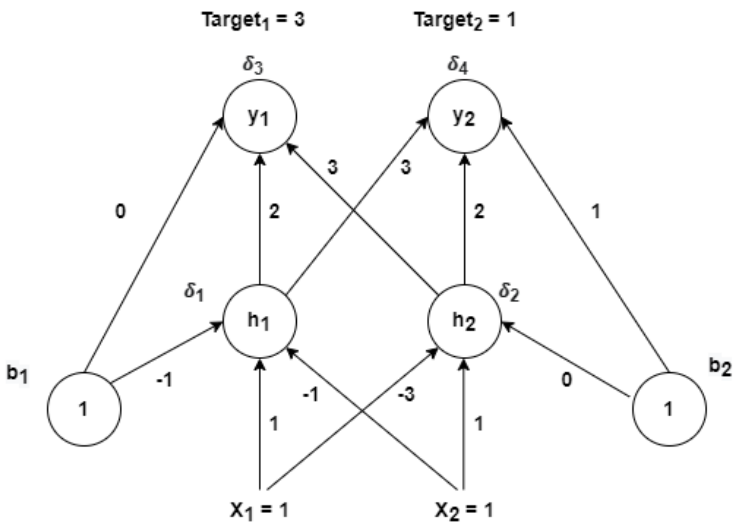
$h_2 =$

$y_1 =$

$y_2 =$

Deltas:

$\delta_1 =$



$\delta_2 =$
$\delta_3 =$
$\delta_4 =$

Weights:

x_1 to $h_1 =$	h_1 to $y_1 =$
x_1 to $h_2 =$	h_1 to $y_2 =$
x_2 to $h_1 =$	h_2 to $y_1 =$
x_2 to $h_2 =$	h_2 to $y_2 =$
b_1 to $h_1 =$	
b_2 to $h_2 =$	
b_1 to $y_1 =$	
b_2 to $y_2 =$	

Part II

Team Programming Assignment

0.1 GitHub Classroom

The link to our GitHub Classroom assignment is `https://classroom.github.com/a/tCO2PRy9`. Following that link will lead to a button that says "Accept this assignment." Accepting this assignment will set up a Git repo with the starter code. You should have admin permissions on this repo, so you can share it with your teammates. *Only one person per team should accept the assignment* and then share the repository with the whole team by adding them as collaborators. On GitHub, this can be done under Settings -> Manage Access -> Add people.

0.2 Data

In this PA, we will classify numbers in the MNIST dataset using a multi-layer neural network. The detailed description of the dataset can be found here.

The code will automatically detect if you have downloaded the MNIST dataset, and if not, initialize it locally at `./data/mnist.pkl`. You will see it once you start implementing and running code.

0.3 Instructions for Programming Assignment

You need to complete the **neuralnet.py, main.py, train.py, gradient.py and util.py** files to complete the assignment. These files are a skeleton code that are designed to guide you to build and implement your neural net in an efficient and modular fashion.

The **config_4.yaml** specifies the configuration for your Neural Network architecture, training hyperparameters, type of activation, etc. for Rubric #4 (below). 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.

In **main.py**, we parse the 'experiment' argument to choose which experiment to run and the config file to be loaded.

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 given for you in this class - you will have to program the implementation. The code is structured in such a way that each activation function can treated as a gate through which we can pass the weighted input sum (a) and get the activated output for that layer. An Activation class object needs to be instantiated while constructing a **Layer** of the Neural Network.

The **Layer** class denotes a standard fully-connected layer of a Neural Network. The *forward* and *backward* functions need to be implemented by you. As the name suggests, *forward* takes in an input vector `x`, computes the weighted input `a` and then uses its *Activation* class object to calculate the activation `z`. The function *backward* takes the weighted sum of the deltas (`deltaCur` variable in backward method of **Activation** class) from the layer above it as input, computes the gradient for its weights (to be saved in `dw`). 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. **Layer** class objects will be instantiated while constructing a **NeuralNetwork** object.

The **NeuralNetwork** class defines the entire network. The `__init__` function has been implemented partially for you which uses the configuration specified in one of the **'config.yaml'** files to generate 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 accuracy (or number of correct predictions can also be returned). 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 (initially there will only be one hidden layer for this project, but we will add more later, and that will lead to more backward passes). The 'loss' function computes cross-entropy loss by taking in the prediction y and targets and returns this loss.

The **load_data** method in **util.py** has been implemented for you, but not the helper functions it calls. You need to implement the other required functions in **util.py** and **train.py**. The requirements for these functions and all other functions are given in the code. Please read the **readme** file for further instructions.

Furthermore, few points to be noted:

- **You can** add additional keys in **config.yaml** files.
- **Do read** the **README.md** file to know about the changes that you are allowed and not allowed to do in the code.
- **You are allowed** to use the following Python libraries and packages: NumPy, PIL, matplotlib, os, and random. If you are unsure about whether you can use a library, please ask on Piazza.
- **You are not allowed** to use any SciPy implementations or any other high-level machine learning packages (including, but not limited to TensorFlow, PyTorch, Keras, scikit-learn, etc.).
- Make sure to include clear instructions in the **README.md** 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.
- You are allowed to use chatgpt and copilot, but you must add proper documentation for using it. Refer to Instructions on Page 1 of this writeup for more information.

To load training, validation and testing data, the given **main.py** will call the function **load_data** in **util.py**.

1. **Data Preprocessing.** (2pts) Read in the MNIST data using the `load_data` function provided in the code. This loads in 60,000 train datapoints and 10,000 test datapoints. The training data needs to be split using an 80-20 ratio to generate the validation data (e.g. for 60,000 training datapoints, we expect 48,000 images in X_{train} and 12,000 images in X_{val}). To save time, we will not use k-fold cross validation. We'll have a fixed training and validation set that will be used for all experiments.

   Normalize the data by z-scoring it. That is, for each image in X_{train}, you compute the average pixel value and their standard deviation over the single image. Then, subtract each pixel with the mean and divide by the standard deviation. The data is already flattened i.e. each pattern has $28 \times 28 = 784$ features. Now every pixel will roughly run between -1 and 1 or so, and will have mean 0. 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 "LeCun-et-al-98-Tricks-of-the-Trade-1998.pdf." Do the same for the validation and test sets. Finally, one-hot encode the labels for the train, validation, and test sets.
   To Report : Dataset used, splitting of training and validation data and normalization procedure used. Report the mean and standard deviation of one of the training images.

2. **Softmax Regression.** (5 pts) We will first perform softmax regression on a single-layer neural network (i.e. input layer directly connects to output layer) and set up code infrastructure for implementing multi-layer later. As a review, softmax regression is the generalization of logistic regression for multi-class classification. For example, given an input x^n and c possible classes, softmax regression will output a vector y^n, where each element, y_k^n represents the probability that x^n is in class k.

$$y_k^n = \frac{exp(a_k^n)}{\sum_{k'} exp(a_{k'}^n)} \tag{5}$$

$$a_k^n = w_k^T x^n \tag{6}$$

   Here, a_k^n is called the *net input* to output unit y_k. Equation 5 is called the *softmax activation function*, and it is a generalization of the logistic activation function. For softmax regression, we use a *one hot encoding* of the targets. That is, the targets are a c-dimensional vector, where the k^{th} element for example n (written t_k^n) is 1 if the input is from category k, and 0 otherwise. Note each output has its own weight vector w_k. With our model defined, we now define the *cross-entropy* cost function for multiple categories in Equation 7:

Algorithm 1 Stochastic Gradient Descent

```
1: procedure STOCHASTIC GRADIENT DESCENT
2:     $w \leftarrow 0$
3:     for t = 1 to $M$ do                                          ▷ Here, t is one epoch.
4:         randomize the order of the indices into the training set
5:         for j = 1 to $N$, in steps of B do          ▷ Here, N is number of examples and B is the batch size
6:             start = j
7:             end = j+B
8:             $w_{t+1} = w_t - learning\_rate * \sum_{n=start}^{end} \nabla E^n(w)$
9:     return $w$
```

$$E = -\sum_n \sum_{k=1}^{c} t_k^n \ln y_k^n \tag{7}$$

Again, taking the average of this over the number of training examples normalizes this loss over different training set sizes. Also averaging over the number of categories c makes it independent of the number of categories. Please take the average over both when reporting results. Surprisingly, it turns out that the learning rule for softmax regression is basically the same as the one for logistic regression! The gradient is:

$$-\frac{\partial E^n(w)}{\partial w_{jk}} = (t_k^n - y_k^n)x_j^n \tag{8}$$

where w_{jk} is the weight from the j^{th} input to the k^{th} output.

In **neuralnet.py**, implement the delta rule to update weights according to the gradient. You should use Algorithm 0 to implement mini-batch stochastic gradient descent for a single-layer neural network. In particular, you will want to implement **output** from the **Activation** class, **backward** function of **Layer** class (referring to Line 8), and other relevant functions including **forward** and **backward** functions in **Layer** and **NeuralNetwork** classes. In addition, fill in **train.py** with the procedure outlined in Algorithm 0.

Using the config (config_4.yaml) provided to you, the expected accuracy on the test dataset should be around **92%**. It is acceptable for it to be a few points off but if there is a significant difference (for instance, 80%), then there is probably a bug in your code. The loss plots should look like an exponential decay function.

To Report : Training procedure including choice of hyperparameters, Two Plots - one for train/ validation loss and one for train/validation accuracy, test accuracy

3. **Backpropagation.** (5pts) Extend your previous code and implement backpropagation to include hidden layers. Check your code for computing the gradient using a single training example. You can compute the slope with respect to one weight using the numerical approximation:

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

where ϵ is a small constant, e.g., 10^{-2}, and E^n is the cross-entropy error for one pattern. Do the following for several patterns: 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 a single weight and and check that the gradient obtained for that weight after backpropagation is within ($O(\epsilon^2)$) of the gradient obtained by numerical approximation. For example, if you obtain 0.991790 and 0.991852, the difference is acceptable whereas if you obtain 0.991790 and 0.998552, it indicates that something is wrong. Please perform this procedure ***separately*** for one output bias weight, one hidden bias weight, and two hidden to output weights and two input to hidden weights.

Use the same model for computing the gradient using backpropagation and using numerical approximation. The model can be a trained one or an untrained one - it doesn't matter, as all you are doing is checking your implementation. If you want to store a copy of the model in your code, make sure you store a deep copy.

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.

To Report : A table with the following columns : Type of weight, gradient obtained from numerical approximation, gradient obtained by backpropagation, absolute difference between the two.

4. **Momentum Experiments.** (10 pts) Using the vectorized update rule you obtained from the written homework, 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 128 hidden units. ***For this programming assignment, use mini-batch stochastic gradient descent throughout, in all problems.***

   You should now add 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 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 *patience* parameter could be 5, for example.

   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, but you only need to report your results and plots on the best learning rate you find.

   With the default config (config_6.yaml) provided to you, the expected accuracy on the test dataset should be around **97%**. It is acceptable for it to be a few points off but if there is a significant difference (for instance, 90%), then there is probably a bug in your code, or you have too high or too low a learning rate. The loss plots should look like your standard training plots. You may or may not observe overfitting depending on your choice of hyperparameters.

   To Report : Training procedure including choice of hyperparameters, Two Plots - one for train/ validation loss and one for train/validation accuracy, test accuracy, your observations and inference from the experiments

5. **Regularization Experiments.** (10 pts) Starting with the multi-layer network you used previously, 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 L2 regularization using value of 1e-2 and 1e-4 for λ). Again, plot training and validation loss, training and validation accuracy, and report final test accuracy. 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.
   Try using L1 regularization instead of L2 regularization. Explain any difference in performance, if you see any. In particular, do you see a decrease or increase in performance? Why could that be?

   To Report : Training procedure including regularization method and choices of λ (which worked better? Why do you think?) Two Plots - one for train/ validation loss and one for train/validation accuracy for L2 regularization, test accuracy for both L1 and L2, your observations and inference from the experiments with L2 and L1 regularization.

6. **Activation Experiments.** (10 pts) Starting with the multi-layer network you used previously, 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!!

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

(b) 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.

To Report : 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.

1 Project Report Outline with Rubric (45 Points)

1. Title (1 pt): The title *has to be informative* and not generic like '151B PA1 Report'. Additionally, please include a list of authors.
2. Abstract (2 pts): 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. Data Loading (2 pts): A short description about the dataset loaded, split between training and validation set and how the data was normalized. Mean and std values for any one train image.
4. Softmax Regression (5 pts): Follow instructions **Softmax Regression** section in writeup above.
5. Numerical Approximation of Gradients (5 pts): Follow instructions from **Backpropagation** section in writeup above.
6. Momentum Experiments (10 pts): Follow instructions from **Momentum Experiments** section in writeup above.
7. Regularization Experiments (10 pts): Follow instructions from **Regularization Experiments** section in writeup above.
8. Activation Experiments (10 pts): Follow instructions from **Activation Experiments** section above.
9. (-1pt per missing paragraph) 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! *Everyone should program!*

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. For the source code - please make sure you submit clean, well-documented code.