Assignment 1 Part 1

CSE 151B: Deep Learning

Summer Session 1 2026

Instructions

Due on Wednesday, July 8, 2026 (11:59 PM)

1. There are two components to this assignment: architectures & algorithms, and programming. For the programming assignment portion of the homework, you will be running machine learning experiments and answering multiple choice questions based off of your findings. Figures may be generated with Excel or Python, so long as they are computer generated. Your answers will be submitted on PrairieLearn. More details on submissions and autograding will be released soon.

2. You are expected to use Python - this assignment will be done in 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 for anything other than saving models, or any off-the-shelf code such as SciPy, 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. 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. Discussions of course materials and homework solutions are encouraged, but you should work on the final solutions to the architectures & algorithms 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.

Problems

1. **Perceptrons. (7 points)**

   We want to learn the "AND" function using four patterns, as shown in Table 1.

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

Table 1: The "AND" function

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

   (b) (4 pts) Draw the rest of this table, as the network learns AND. 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). If the perceptron output $(w_0 + w_1x_1 + w_2x_2)$ is greater than or equal to zero, its output is 1. Otherwise, its output is 0.

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

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

2. (15 pts) **For multi-class classification on the Fashion-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}. In the following discussion, n denotes the nth input pattern.

   (a) **Derivation. (5 pts)** Take the derivative of the two-class cross entropy function $X_{ent}$with respect to w_i (an arbitrary weight in the model).

$$Xent = -\sum_{n=1}^{N}[t^n \ln y^n + (1 - t^n)\ln(1 - y^n)]$$

   Show that this leads to the delta rule for logistic regression:

$$w_i = w_i + \alpha \sum_{n=1}^{N}(t^n - y^n)x_i^n$$

Recall that y uses the logistic activation function:

$$y^n(a^n) = \frac{1}{1+e^{-a^n}}$$

where a^n is the weighted sum of the inputs for the nth input. This expands into $a^n = \sum_{j=0}^{d} w_j^n x_j^n$. (Note that n is not an exponent! Machine learning math really likes abusing notation.)

(b) (5 pts) **Update rule.** 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 and E^n is the cross entropy error for example n. The output layer's delta is $\delta_k^n = t_k^n - y_k^n$ where t_k^n, and the hidden layer's delta is $\delta_j^n = (1 - tanh^2(a_j^n)) \sum_k (t_k^n - y_k^n)\ w_{jk}$. E^n is defined by the following:

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

Write the update rule for w's in terms of the δ's 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. 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 -1 and 1 respectively.

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.

$h_1 =$

$h_2 =$

$y_1 =$

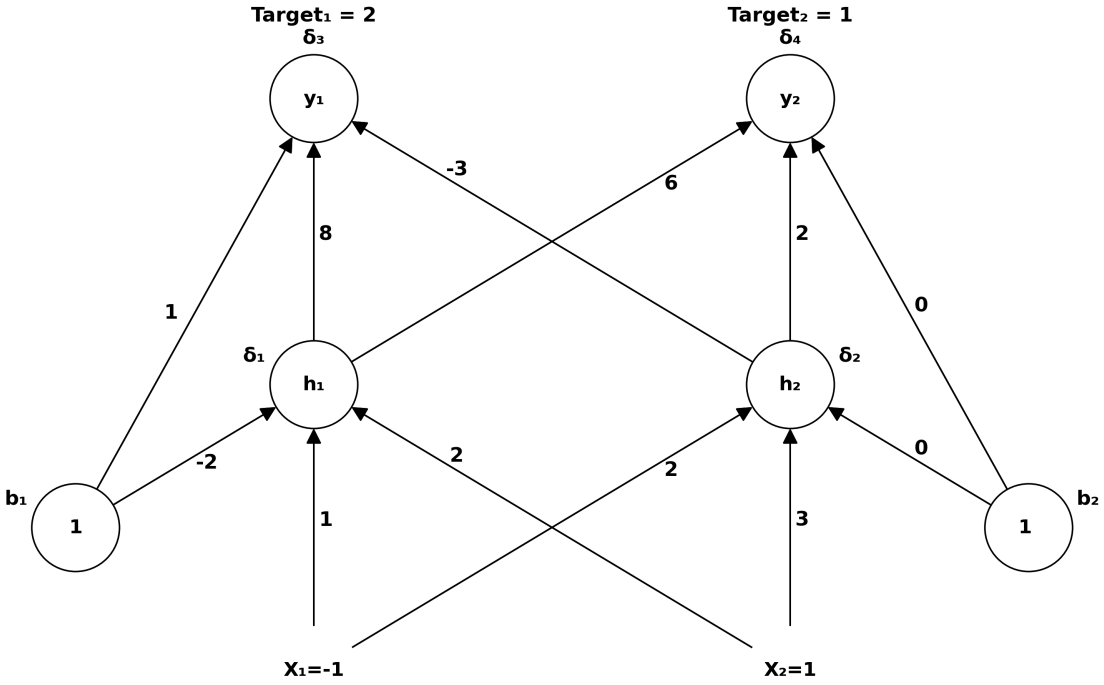
$y_2 =$

Deltas:

$\delta_1 =$

$\delta_2 =$

$\delta_3 =$



$\delta_4 =$

Weights:

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

h_1 to $y_1 =$
h_1 to $y_2 =$
h_2 to $y_1 =$
h_2 to $y_2 =$

Programming Assignment

0.1 Data

In this PA, we will classify images in the FashionMNIST 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 FashionMNIST dataset, and if not, initialize it locally at `./data/numpy/`. You will see it once you start implementing and running code.

0.2 Instructions for Programming Assignment

You must implement all **TODO** sections in the **neuralnet.py, main.py, train.py, gradient.py and util.py** files to complete the assignment. We have provided a skeleton 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. The definitions of *forward* and *backward* have been given for you. The code is structured such that each activation function can be 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. As the name suggests, the *forward* function takes in an input vector `x`, computes the weighted input `a` and then uses its *Activation* class object to calculate the activation `z`. The *backward* function takes the weighted sum of the deltas (`deltaCur` variable in backward method of **Activation** class) from the layer above it, 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. 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 - it uses the configuration specified in one of the **'config.yaml'** files to generate the network. Understand this function carefully. The **forward** function takes in the input dataset `x` and targets (in one hot encoded form), performs a forward pass on the data `x` and returns the loss and accuracy (or number of correct predictions, the implementation is up to you). The **backward** function computes the error between the 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. 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:

- **Run this on Datahub's CPU instances.** Running it on an instance with a GPU increasing training/inference time significantly.
- **Please do not modify the format print statements for the model's test set accuracy and gradient check outputs.** This format is what our autograder looks for to extract your performance.

  For questions 3, 5, 6, and 7, your model test set accuracy results should be printed in the format:

"Final test accuracy: 0.xxxx, Test loss: 0.xxxx"

For question 4, your model gradeint check should be printed in the format:

"Checking gradient using epsilon 0.02
Layer x w[y,z]: numerical=xxxxxx, analytical=xxxxxx, diff=xxxxx, within tolerance = xxxx
Layer x w[y,z]: numerical=xxxxxx, analytical=xxxxxx, diff=xxxxx, within tolerance = xxxx
..."

- **Please save your trained models in a folder named "models" in your submission.** This makes it so we can test your models without having to retrain them.
- **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 (for purposes other than model loading), 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 visualization.** (1pt) In `visualize.ipynb`, run the code already provided for you to see what some images look like.
   To Answer: How many classes are there in the FashionMNIST dataset?
   (a) 10
   (b) 20
   (c) 100
   (d) 5

2. **Data Preprocessing.** (2pts) Read in the FashionMNIST 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. 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. For each image in X_{train}, compute the average pixel value and standard deviation over the single image. Subtract from each pixel this mean and divide by the standard deviation. Now every pixel will roughly run between -1 and 1 or so, and will have mean 0. Do the same for the validation and test sets. Finally, one-hot encode the labels for the train, validation, and test sets.
   To Answer: What is the purpose of z-scoring our data before putting it through the pipeline?
   (a) To make it so our activation functions do not saturate.
   (b) It redefines our data on a standardized scale and allows us to compare different value ranges across features.
   (c) It removes outliers form our data and allows us to separate noise from signal.
   (d) It compresses feature values to be between 0 and 1 for faster training.

3. **Softmax Regression.** (10 pts) We will first perform softmax regression on a single-layer neural network (i.e. input layer directly connects to output layer). 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*. 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:

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

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. This makes the delta rule:

$$w_{jk} = w_{jk} - \alpha \sum_n \frac{\partial E^n(w)}{\partial w_{jk}} \tag{9}$$

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 **86%**. It is acceptable for it to be a few points off but if there is a significant difference (for instance, 70%), then there is probably a bug in your code. The loss plots should look like an exponential decay function. **Please store this model in a folder named "models" and make it loadable with the flag –experiment 'test_softmax' in main.py.**

Algorithm 1 Stochastic Gradient Descent

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

To Answer: What is the purpose of using Softmax instead of just taking values directly from the output layer?

(a) Softmax makes it so that all outputs sum to 1, allowing creation of a probability distribution.

(b) Softmax makes every output either 0 or 1.

(c) Softmax makes negative logits 0.

(d) Softmax caps logits at 1.

4. **Backpropagation.** (10 pts) 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 training samples: 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 0.991752, 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 5 unique weights in the input to hidden layer and 5 unique weights in the hidden to output layer (total of 10 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 according to the print statements in the code package. **If you train a model, please store this model in a folder named "models" and make it loadable with the flag –experiment 'test_gradients' in main.py.**

To Answer: Why do we need to maintain a small epsilon when numerically approximating the gradient?

(a) The further away from we get from the point of approximation, the less accurate our approximation is.

(b) A small epsilon guarantees our numerical approximation is the same as the backprop gradient.

(c) It is faster efficient to approximate the gradient with a small epsilon.

(d) As epsilon gets larger, approximation gets easier, making the information from numerical approximation less useful.

5. **Momentum Experiments.** (10 pts) Using the vectorized update rule you obtained from the architectures & algorithms portion of the 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 **88%**. You should beat your previous model without momentum. It is acceptable for it to be a few points off but if there is a significant difference (for instance, 75%), 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. **Please store this model in a folder named "models" and make it loadable with the flag –experiment 'test_momentum' in main.py.**

   To Answer: A student is developing a machine learning algorithm to classify images, but he complains that it takes a long time for the model to converge and the model often gets stuck in local, nonoptimal minimums. Which of the following could help his situation?

   (a) Implement momentum with a momentum factor of 0.

   (b) Implement momentum with a momentum factor of 0.5.

   (c) Implement L2 regularization.

   (d) Use ReLU.

6. **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). Try using L1 regularization instead of L2 regularization. Save the best performing model from your experiments in this section. **Please store this model in a folder named "models" and make it loadable with the flag –experiment 'test_regularization' in main.py.**

   To Answer: A student is also working on a machine learning model to classify data from the FashionMNIST, but runs into overfitting issues. Which of the following would be most likely to help?

   (a) Implement L2 Regularization.

   (b) Implement L1 Regularization.

   (c) Increase number of images in each training batch.

   (d) Decrease number of images in each training batch.

7. **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. Like question 6, save the best performing model from your experiments in this section. **Please store this model in a folder named "models" and make it loadable with the flag –experiment 'test_activation' in main.py.**

To Answer: When implementing a deep network, a student notices that the later layers of her model are barely being updated. She is currently using a tanh activation function and is thinking about replacing it. Which of the following would best address the problems she observed?

(a) Switch activation function to ReLU.

(b) Switch activation function to sigmoid.

(c) Stick with tanh and decrease the amount of training epochs.

(d) Decrease the learning rate.

Acknowledgments

All parts of this assignment were directly adapted from Professor Gary Cottrell's Winter 2025 offering of the course.