Assignment 1
Computer Vision and Deep Learning

Delivery Deadline: **Friday, February 3rd, 23:59PM.**
You are allowed to work in groups of 1 or 2 people.

Introduction

In this assignment, we will explore the power of gradient descent to perform binary classification with Logistic Regression. Then, we will explore multi-class classification with Softmax Regression on the MNIST dataset. Further, you will experiment with weight regularization through L2 norm, simple visualization of weights and basic visualization to present your result.

With this assignment, we provide you starter code for the programming tasks. You can download this from:
https://github.com/TDT4265-tutorial/TDT4265_StarterCode.

Python files: For this assignment, we require you to use the provided files, and you are not allowed to create any additional files for your code (except for task 4).

Report outline: We've included a jupyter notebook to help you create your report. You do not have to use this file. Remember to export the jupyter notebook to PDF before submitting it to the blackboard.

To set up your environment, follow the guide in the GitHub repo:
https://github.com/TDT4265-tutorial/TDT4265_StarterCode/blob/main/python_setup_instructions.md

Recommended readings

1. See "Recommended Resources" on the blackboard.
2. 3Blue1Brown: What is a Neural Network?
3. 3Blue1Brown Gradient Descent.
4. Neural Networks and Deep Learning: Chapter 1.

Delivery

We ask you to follow these guidelines:

- **Report:** Deliver your answers as a **single PDF file**. Include all tasks in the report, and mark it clearly with the task you are answering (Task 1.a, Task 1.b, Task 2.c etc.). There is no need to include your code in the report.
- **Plots in report:** For the plots in the report, ensure that they are large and easily readable. You might want to use the "ylim" function in the matplotlib package to "zoom" in on your plots. Label the different graphs such that it is easy for us to see which graphs correspond to the train, validation and test set.
- **Source code:** Upload your code as a zip file. In the assignment starter code, we have included a script (`create_submission_zip.py`) to create your delivery zip. **Please use this**, as this will structure the zip file as we expect. (Run this from the same folder as all the python files).

  To use the script, simply run: `python create_submission_zip.py`
- **Upload to blackboard:** Upload the ZIP file with your source code and the report to blackboard before the delivery deadline.

Any group that does not follow these guidelines will be subtracted in points.

Task 1: Regression

Notation: We use index k to represent a node in the output layer, index j to represent a node in the hidden layer, and index i to represent an input unit x_i. Hence, the weight from node i in the input layer to node k in the output layer is w_{ki}. We write the activation of output unit k as $\hat{y}_k = f(z_k)$, where f represents the output unit activation function (sigmoid for logistic regression or softmax for softmax regression). In equations where multiple training examples are used (for example, summing over samples), we will use n to specify which training example we are referring to. Hence, y_k^n is the output of node k for training example n. If we do not specify n by writing y_k, we implicitly refer to y_k^n. Capital letters N, I, J or K refer to the total number of nodes in a layer. For logistic regression, we will not specify which output node we're referring to, as there is only a single output node k. Therefore weight $w_{k,i}$ can be written as w_i.

Logistic Regression

Logistic regression is a simple tool to perform binary classification. Logistic regression can be modelled as using a single neuron reading an input vector $x \in \mathbb{R}^{I+1}$ and parameterized by a weight vector $w \in \mathbb{R}^{I+1}$. I is the number of input nodes, and we add a 1 at the beginning for a bias parameter (known as the "bias trick"). The neuron outputs the probability that x is a member of class C_1. This can be written as,

$$P(x \in C_1|x) = f(x) = \frac{1}{1+e^{-w^Tx}}, \quad w^Tx = \sum_i^I w_i \cdot x_i \tag{1}$$

$$P(x \in C_2|x) = 1 - P(x \in C_1|x) = 1 - f(x) \tag{2}$$

where $f(x)$ returns the probability of x being a member of class C_1; $f \in [0, 1]$ [1]. By defining the output of our network as $\hat{y}$, we have $\hat{y} = f(x)$.

We use the **cross entropy loss** function (Equation 3) for two categories to measure how well our function performs over our dataset. This loss function measures how well our hypothesis function f does over the N data points.

$$C(w) = \frac{1}{N}\sum_{n=1}^{N} C^n, \quad \text{where } C^n(w) = -(y^n \ln(\hat{y}^n) + (1-y^n)\ln(1-\hat{y}^n)) \tag{3}$$

Here, y^n is the target value (also known as the label of the image). Note that we are computing the average cost function such that the magnitude of our cost function is not dependent on the number of training examples. Our goal is to minimize this cost function through gradient descent, such that the cost function reaches a minimum of 0. This happens when $y^n = \hat{y}^n$ for all n.

Softmax Regression

Softmax regression is simply a generalization of logistic regression to multi-class classification. Given an input x which can belong to K different classes, softmax regression will output a vector $\hat{y}$ (with length K), where each element $\hat{y}_k$ represents the probability that x is a member of class k.

$$\hat{y}_k = \frac{e^{z_k}}{\sum_{k'}^{K} e^{z_{k'}}}, \quad \text{where } z_k = w_k^T \cdot x = \sum_i^I w_{k,i} \cdot x_i \tag{4}$$

[1]The function f is known as the sigmoid activation function

Equation 4 is known as the Softmax function, and has the attribute that $\sum_k^K \hat{y}_k = 1$. Now, w is no longer a vector, but a weight matrix, $w \in \mathbf{R}^{K \times I}$.

The cross-entropy cost function for multiple classes is defined as,

$$C(w) = \frac{1}{N}\sum_{n=1}^{N} C^n(w), \quad C^n(w) = -\sum_{k=1}^{K} y_k^n \ln(\hat{y}_k^n) \tag{5}$$

For this task, please:

(a) [7*pt*] Derive the gradient for Logistic Regression. To minimize the cost function with gradient descent, we require the gradient of the cost function. Show that for Equation 3, the gradient is:

$$\frac{\partial C^n(w)}{\partial w_i} = -(y^n - \hat{y^n})x_i^n \tag{6}$$

Show thorough work such that your approach is clear.

Hint: To solve this, you have to use the chain rule. Also, you can use the fact that:

$$\frac{\partial f(x^n)}{\partial w_i} = x_i^n f(x^n)(1 - f(x^n)) \tag{7}$$

Solution 1a:

$$C^n(w) = -(t^n ln(y^n) + (1-t^n) ln(1-y^n)), \tag{8}$$

we use chain rule and derivation of the logarithmic function to get:

$$\frac{dC^n(w)}{dw_j} = -(\frac{t^n}{y^n}\frac{dy^n}{dw_j} - \frac{1-t^n}{1-y^n}\frac{dy^n}{dw_j}) \tag{9}$$

Using

$$\frac{dy^n}{dw_j} = x_j^n y^n (1-y^n), \tag{10}$$

we get

$$\frac{dC^n(w)}{dw_j} = -(\frac{t^n}{y^n}x_j^n y^n(1-y^n) - \frac{1-t^n}{1-y^n}x_j^n y^n(1-y^n)) \tag{11}$$

Simplifying this we get:

$$\frac{dC^n}{dw_j} = -x_j(t^n - y^n) \tag{12}$$

(b) [9*pt*] Derive the gradient for Softmax Regression. For the multi-class cross entropy cost in Equation 5, show that the gradient is:

$$\frac{\partial C^n(w)}{\partial w_{kj}} = -x_j^n(y_k^n - \hat{y}_k^n) \tag{13}$$

A few hints if you get stuck:

- Derivation of the softmax is the hardest part. Break it down into two cases.
- $\sum_{k=1}^K y_k^n = 1$
- $\ln(\frac{a}{b}) = \ln a - \ln b$

Derive the gradient for softmax regression

We have

$$C^n(w) = -\sum_{k=1}^{C} t_k^n ln(y_k^n) \tag{14}$$

By using $ln(a/b) = ln(a) - ln(b)$, we get

$$-\frac{dC^n(w)}{dw_{kj}} = \frac{d}{dw_{kj}} \sum_{k'=1}^{C} t_{k'}^n (a_{k'}^n - ln(\sum_{k''} e^{a_{k''}^n})) \tag{15}$$

$$-\frac{dC^n(w)}{dw_{kj}} = \frac{d}{dw_{kj}} \sum_{k'=1}^{C} t_{k'}^n a_{k'}^n - t_{k'}^n ln(\sum_{k''} e^{a_{k''}^n}) \tag{16}$$

Since $t_{k'}$ is constant w.r.t w_{kj}, and $\frac{d}{dw_{kj}} a_{k'} = 0$ for all $k'! = k$, we get:

$$-\frac{dC^n(w)}{dw_{kj}} = t_k^n x_j^n - \sum_{k'=1}^{C} t_{k'}^n \frac{d}{dw_{kj}} ln(\sum_{k''} e^{a_{k''}^n}) \tag{17}$$

Taking the derivative of the logarithm, we get:

$$-\frac{dC^n(w)}{dw_{kj}} = t_k^n x_j^n - \sum_{k'=1}^{C} t_{k'}^n \frac{1}{\sum_{k''} e^{a_{k''}^n}} \frac{d}{dw_{kj}} \sum_{k''} e^{a_{k''}^n} \tag{18}$$

We know that $\frac{d}{dw_{kj}} e^{a_{k''}^n} = 0$ if $k''! = k$, giving us:

$$-\frac{dC^n(w)}{dw_{kj}} = t_k^n x_j^n - \sum_{k'=1}^{C} t_{k'}^n \frac{x_j^n e^{z_k^n}}{\sum_{k''} e^{a_{k''}^n}} \tag{19}$$

Since $y_k^n = \frac{e^{z_k^n}}{\sum_{k'} e^{a_{k'}^n}}$, we can reduce this to:

$$-\frac{dC^n(w)}{dw_{kj}} = t_k^n x_j^n - x_j^n y_k^n \sum_{k'=1}^{C} t_{k'}^n \tag{20}$$

Knowing that $\sum_{k=1}^{C} t_k = 1$, we get

$$-\frac{dC^n(w)}{dw_{kj}} = x_j^n (t_k^n - y_k^n) \tag{21}$$

Derive the Gradient for Softmax Regression – Alternative Solution

$$\begin{aligned}
-\frac{\partial C^n(w)}{\partial w_{kj}} &= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(y_k^n) \\
&= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(\frac{e^{z_k^n}}{\sum_{k'} e^{a_{k'}^n}}) && ln(a/b) = ln(a) - ln(b) \\
&= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n (ln(e^{z_k^n}) - ln(\sum_{k'} e^{a_{k'}^n})) && ln(a^b) = b \qquad (22) \\
&= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n z_k^n - t_k^n ln(\sum_{k'} e^{a_{k'}^n}) && \text{Split sum at minus sign} \\
&= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n z_k^n - \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(\sum_{k'} e^{a_{k'}^n})
\end{aligned}$$

Let's solve $\frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n z_k^n$:

$$\begin{aligned}
\frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n z_k^n &= \frac{\partial}{\partial w_{kj}} (t_k^n z_k^n + \sum_{k' \neq k}^{C} t_{k'}^n a_{k'}^n) && \text{Used } a_i + \Sigma_{i' \neq i} a_{i'} = \Sigma_i a_i \\
&= \frac{\partial}{\partial w_{kj}} t_k^n z_k^n + \frac{\partial}{\partial w_{kj}} \sum_{k' \neq k}^{C} t_{k'}^n a_{k'}^n && (23) \\
&= t_k^n x_j^n + 0 \\
&= t_k^n x_j^n
\end{aligned}$$

Note that we have to consider both the cases when the k in $\sum_{k=1}^{C}$ is equal to the k in w_{kj} and the case when they are different. Therefore, we extract the k^{th} element (k, as in w_{kj}) from the $\sum_{k=1}^{C}$ sum, leaving us with $t_k^n z_k^n + \sum_{k' \neq k}^{C} t_{k'}^n a_{k'}^n$.

Now let's solve $\frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(\sum_{k'} e^{a_{k'}^n})$ using the same approach:

$$
\begin{aligned}
&\frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(\sum_{k'} e^{a_{k'}^n}) \\
&= \frac{\partial}{\partial w_{kj}} (t_k^n ln(\sum_{k'} e^{a_{k'}^n}) + \sum_{k' \neq k}^{C} t_{k'}^n ln(\sum_{k''} e^{a_{k''}^n})) && \text{Used } a_i + \Sigma_{i' \neq i} a_{i'} = \Sigma_i a_i \\
&= t_k^n \frac{1}{\sum_{k'} e^{a_{k'}^n}} \frac{\partial}{\partial w_{kj}} (\sum_{k'} e^{a_{k'}^n}) + \sum_{k' \neq k}^{C} t_{k'}^n \frac{1}{\sum_{k''} e^{a_{k''}^n}} \frac{\partial}{\partial w_{kj}} (\sum_{k''} e^{a_{k''}^n}) && \text{Computed } \frac{\partial}{\partial w_{kj}} ln(\sum_{k'} e^{a_{k'}^n}) \\
&= t_k^n \frac{e^{a_k^n}}{\sum_{k'} e^{a_{k'}^n}} x_j^n + \sum_{k' \neq k}^{C} t_{k'}^n \frac{e^{a_k^n}}{\sum_{k''} e^{a_{k''}^n}} x_j^n && \text{Computed } \frac{\partial}{\partial w_{kj}} (\sum_{k'} e^{a_{k'}^n}) \\
&= t_k^n y_k^n x_j^n + \sum_{k' \neq k}^{C} t_{k'}^n y_k^n x_j^n && \text{Noticed } \frac{e^{a_k^n}}{\sum_{k'} e^{a_{k'}^n}} = y_k^n \\
&= t_k^n y_k^n x_j^n + y_k^n x_j^n \sum_{k' \neq k}^{C} t_{k'}^n && \text{Noticed } y_k^n \text{ and } x_j^n \text{ are constant in } \Sigma_{k' \neq k}^{C} \\
&= y_k^n x_j^n (t_k^n + \sum_{k' \neq k}^{C} t_{k'}^n) && \text{Rearranged} \\
&= y_k^n x_j^n \sum_{k}^{C} t_k^n && \text{Used } a_i + \Sigma_{i' \neq i} a_{i'} = \Sigma_i a_i \\
&= y_k^n x_j^n && \text{Used } \Sigma_k t_k^n = 1
\end{aligned}
\tag{24}
$$

Now we insert Eq. 23 and Eq. 24 into where we left of in Eq. 22:

$$
\begin{aligned}
-\frac{\partial C^n(w)}{\partial w_{kj}} &= \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n z_k^n - \frac{\partial}{\partial w_{kj}} \sum_{k=1}^{C} t_k^n ln(\sum_{k'} e^{a_{k'}^n}) \\
&= t_k^n x_j^n - y_k^n x_j^n \\
&= x_j^n (t_k^n - y_k^n)
\end{aligned}
\tag{25}
$$

Task 2: Logistic Regression through Gradient Descent

In this assignment you are going to start classifying digits in the well-known dataset MNIST. The MNIST dataset consists of $70,000$ handwritten digits, split into 10 object classes (the numbers 0-9). The images are 28x28 grayscale images, and every image is perfectly labeled. The images are split into two datasets, a training set consisting of $60,000$ images, and a testing set consisting of $10,000$ images. For this assignment, we will use a subset of the MNIST dataset[2].

Bias trick: Each image is 28x28, so the unraveled vector will be $x \in \mathbb{R}^{784}$. For each image, append a '1' to it, giving us $x \in \mathbb{R}^{785}$. With this trick, we don't need to implement the forward and backward pass for the bias.

Logistic Regression through gradient descent

For this task, we will use mini-batch gradient descent to train a logistic regression model to predict if an image is either a 2 or a 3. We will remove all images in the MNIST dataset that are not a 2 or 3 (this pre-processing is already implemented in the starter code). The target is 1 if the the input is from the "2" category, and 0 otherwise.

Mini-batch gradient descent is a method that takes a batch of images to compute an average gradient, then use this gradient to update the weights. Use the gradient derived for logistic regression in Task 1 to classify $x \in \mathbb{R}^{785}$ for the two categories 2's and 3's.

Vectorizing code: We recommend you to vectorize the code with numpy, which will make the runtime of your code significantly faster. Note that vectorizing your code is not required, but highly recommended (it will be required for assignment 2). Vectorizing it simply means that if you want to, for example, compute the gradient in Equation 6, you can compute it in one go instead of iterating over the number of examples and weights. For example, $w^T x$ can be written as `w.dot(x)`.

For this task, please:

(a) [12*pt*] Before implementing our gradient descent training loop, we will implement a couple of essential functions. Implement four functions in the file `task2a.py`.

- Implement a function that pre-processes our images in the function `pre_process_images`. This should normalize our images from the range $[0, 255]$ to $[-1, 1]$ [3], and it should apply the bias trick.
- Implement a function that performs the forward pass through our single layer neural network. Implement this in the function `forward`. This should implement the network outlined by Equation 1.
- Implement a function that performs the backward pass through our single layer neural network. Implement this in the function `backward`. To find the gradient for our weight, we can use the equation derived in task 1 (Equation 6).
- Implement cross entropy loss in the function `cross_entropy_loss`. This should compute the average of the cross entropy loss over all targets/labels and predicted outputs. The cross entropy loss is shown in Equation 3.

We have included a couple of simple tests to help you debug your code. This also includes a gradient approximation test that you should get working. For those interested, this is explained in more detail in the Appendix.

Note that you should not start on the subsequent tasks before all tests are passing!

[2] We will only use $20,000$ images in the training set to reduce computation time.

[3] Normalizing the input to be zero-centered improves convergence for neural networks. Read more about this in Lecun et al. Efficient Backprop Section 4.3.

(b) [9*pt*] Implement logistic regression with mini-batch gradient descent for a single layer neural network. The network should consist of a single weight matrix with $784 + 1$ inputs and a single output (the matrix will have shape 785×1). Initialize the weights (before any training) to all zeros. We've set the default hyperparameters for you, so there is no need to change these.

During training, track the training loss for each gradient step (this is implemented in the starter code). Less frequently, track the validation loss over the whole validation set (in the starter code, this is tracked every time we progress 20% through the training set).

Implement this in the function `train_step` in `task2.py`.

(report) In your report, include a plot of the training and validation loss over training. Have the number of gradient steps on the x-axis, and the loss on the y-axis. Use the `ylim` function to zoom in on the graph (for us, `ylim([0, 0.2])` worked fine).

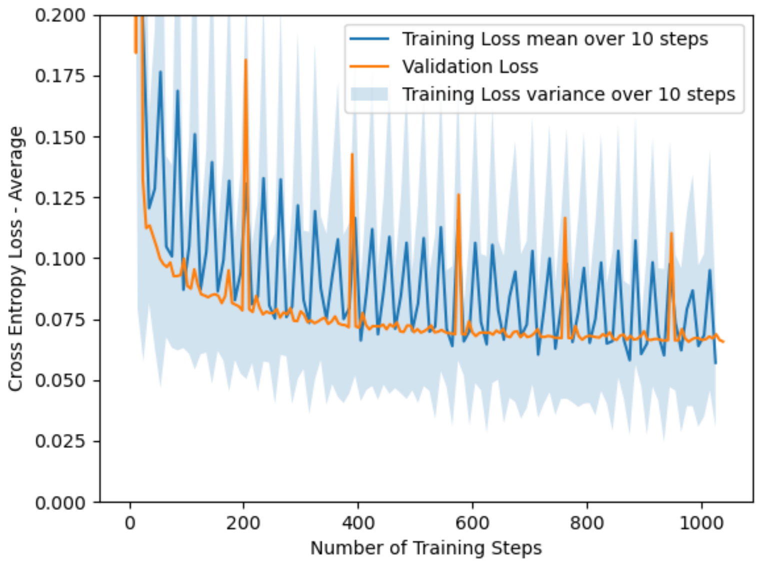

Figure 1: Training graph for task 2b

(c) [4*pt*] Implement a function that computes the binary classification accuracy[4] over a dataset. Implement this in the function `calculate_accuracy`.

(report) Compute the accuracy on the training set and validation set over training. Plot this in a graph (similar to the loss), and include the plot in your report. Use the `ylim` function to zoom in on the graph (for us, `ylim([0.93, 0.99])` worked fine).

Early Stopping: Early stopping is a tool to stop the training before your model overfits on the training dataset. By using a validation set along with our training set[5], we can regularly check if our model is starting to overfit or not. If the cost function on the validation set stops improving, we can stop the training and return the weights at the minimum validation loss.

Dataset shuffling: Shuffling the training dataset between each epoch improves convergence. By using shuffling you present a new batch of examples each time which the network has never seen, which will produce larger errors and improve gradient descent [6].

[4]accuracy $= \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$. The prediction is determined as 1 if $\hat{y} \geq 0.5$ else 0

[5]Note that we never train on the validation set.

[6]For those interested, you can read more about dataset shuffling in Section 4.2 in Efficient Backprop.

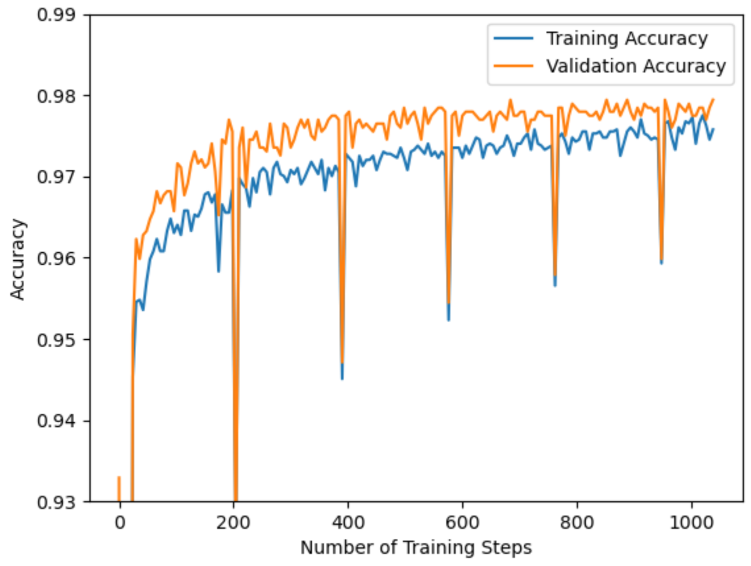

Figure 2: Accuracy graph for task 2c. Final Train accuracy: 0.976, val: 0.9575, test: 0.967

(d) [4*pt*] Implement early stopping into your training loop. Use the following early stop criteria: stop the training if the validation loss does not improve after passing through the validation set 10 times. Increase the number of epochs to 500. **(report)** After how many epochs does early stopping kick in?

You can implement early stopping in the file `trainer.py`.

Solution: Early stop at epoch 33 without shuffling, 16 with shuffling.

(e) [5*pt*] Implement dataset shuffling for your training. Before each epoch, you should shuffle all the samples in the dataset. Implement this in the function `batch_loader` in `utils.py`

(report) Include a plot in your report of the validation accuracy with and without shuffle. You should notice that the validation accuracy has fewer "spikes". Why does this happen?

Solution: Shuffling reduces the chance for each batch containing one batch of only 1s, and the batch should better represent the overall training dataset. If the student struggles to see the difference, try to make them increase or reduce batch size. Decreasing the batch size should worsen the problem without shuffling.

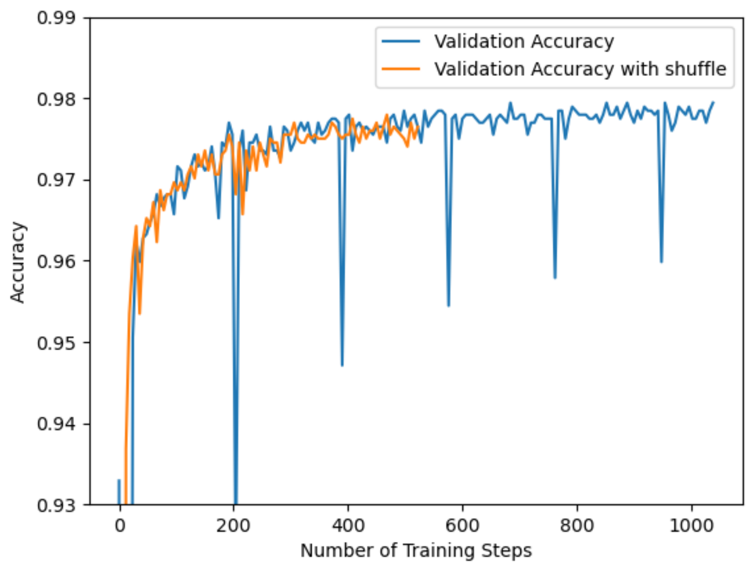

Figure 3: Training graph for task 2e

Task 3: Softmax Regression through Gradient Descent

In this task, we will perform a 10-way classification on the MNIST dataset with softmax regression. Use the gradient derived for softmax regression loss and use mini-batch gradient descent to optimize your model

One-hot encoding: With multi-class classification tasks it is required to one-hot encode the target values. Convert the target values from integer to one-hot vectors. (E.g: $3 \rightarrow [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$). The length of the vector should be equal to the number of classes ($K = 10$ for MNIST, 1 class per digit).

For this task, please:

(a) [14*pt*] Before implementing our gradient descent training loop, we will implement a couple of essential functions. Implement four functions in the file `task3a.py`.

- Implement a function that one-hot encodes our labels in the function `one_hot_encode`. This should return a new vector with one-hot encoded labels.
- Implement a function that performs the forward pass through our single layer softmax model. Implement this in the function `forward`. This should implement the network outlined by Equation 4.
- Implement a function that performs the backward pass through our single layer neural network. Implement this in the function `backward`. To find the gradient for our weight, use Equation 13.
- Implement cross entropy loss in the function `cross_entropy_loss`. This should compute the average of the cross entropy loss over all targets/labels and predicted outputs. The cross entropy loss is defined in Equation 5.

We have included a couple of simple tests to help you debug your code.

(b) [3*pt*] **The rest of the task 3 subtasks should be implemented in `task3.py`.**

Implement softmax regression with mini-batch gradient descent for a single layer neural network. The network should consist of a single weight matrix, with $784 + 1$ inputs and ten outputs (shape 785×10). Initialize the weight (before any training) to all zeros.

Implement this in `train_step` in task3.py. This function should be identical to the task2b, except that you are using a different cross entropy loss function.

(report) In your report, include a plot of the training and validation loss over training. Have the number of gradient steps on the x-axis, and the loss on the y-axis. Use the `ylim` function to zoom in on the graph (for us, `ylim([0.2, .6])` worked fine).

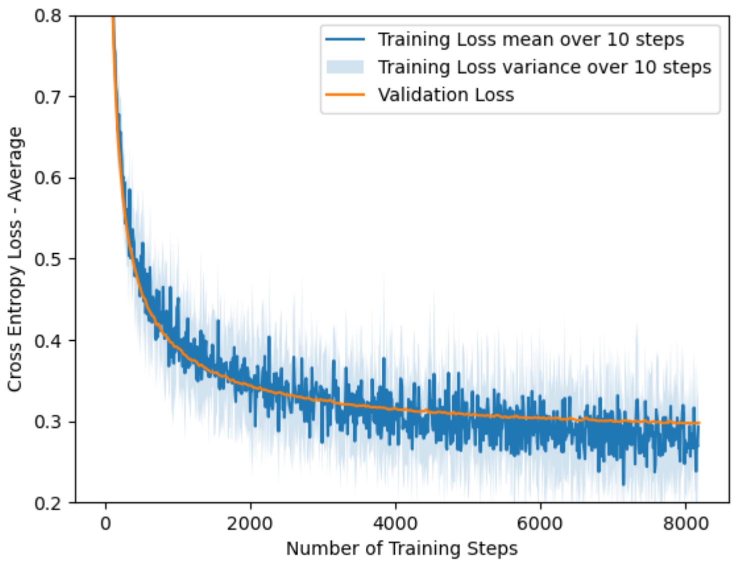

Figure 4: Task 3b

(c) [*4pt*] Implement a function that computes the multi-class classification accuracy over a dataset. Implement this in the function `calculate_accuracy`.

(report) Include in your report a plot of the training and validation accuracy over training.

(d) [*3pt*] **(report)** For your model trained in task 3c, do you notice any signs of overfitting? Explain your reasoning.

Solution: Yes, clear evidence of overfitting. Even though the validation loss does not increase, the model performs much better on the training set.

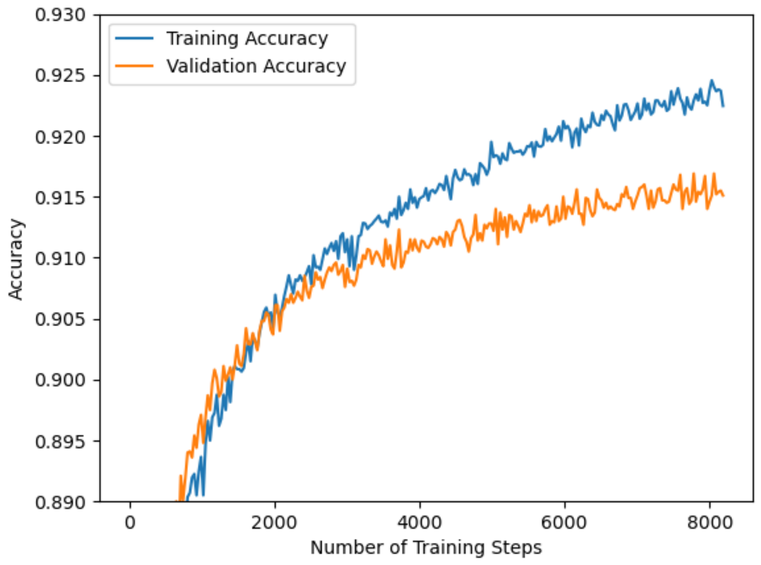

Figure 5: Task 3c

Task 4: Regularization

One way to improve generalization is to use regularization. Regularization is a modification we make to a learning algorithm that is intended to reduce its generalization error [7]. Regularization is the idea that we should penalize the model for being too complex. In this assignment, we will carry this out by introducing a new term in our objective function to make the model "smaller" by minimizing the weights.

$$J(w) = C(w) + \lambda R(w), \tag{26}$$

where $R(w)$ is the complexity penalty and λ is the strength of regularization (constant). There are several forms for R, such as L_2 regularization

$$R(w) = ||w||^2 = \sum_{i,j} w_{i,j}^2, \tag{27}$$

where w is the weight vector of our model.

For your report, please:

(a) [$4pt$] **(report)** Derive the update term for softmax regression with L_2 regularization, that is, find $\frac{\partial J}{\partial w}$, where C is given by Equation 5.

Solution 4a:

$$\frac{\partial J}{\partial w} = \frac{\partial C}{\partial w} + 2\lambda w \tag{28}$$

(b) [$7pt$] Implement L_2 regularization in your backward pass. You can implement the regularization in `backward` in `task3a.py`.

[7]The generalization error can be thought of as training error - validation error.

For the remaining part of the assignment, you can implement the functionality in the file task3.py or create a new file.

(report) Train two different models with different λ values for the L_2 regularization. Use $\lambda = 0.0$ and $\lambda = 1.0$. Visualize the weight for each digit for the two models. Why are the weights for the model with $\lambda = 1.0$ less noisy?

The visualization should be similar to Figure 6.

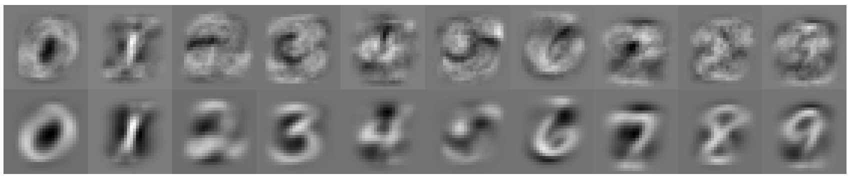

Figure 6: The visualization of the weights for a model with $\lambda = 0.0$ (top row), and $\lambda = 1.0$ (bottom row).

(c) [5*pt*] **(report)** Train your model with different values of λ: 1.0, 0.1, 0.01, 0.001. Note that for each value of λ, you should train a new network from scratch. Plot the validation accuracy for different values of λ on the same graph. Have the accuracy on the y-axis and the number of training steps on the x-axis.

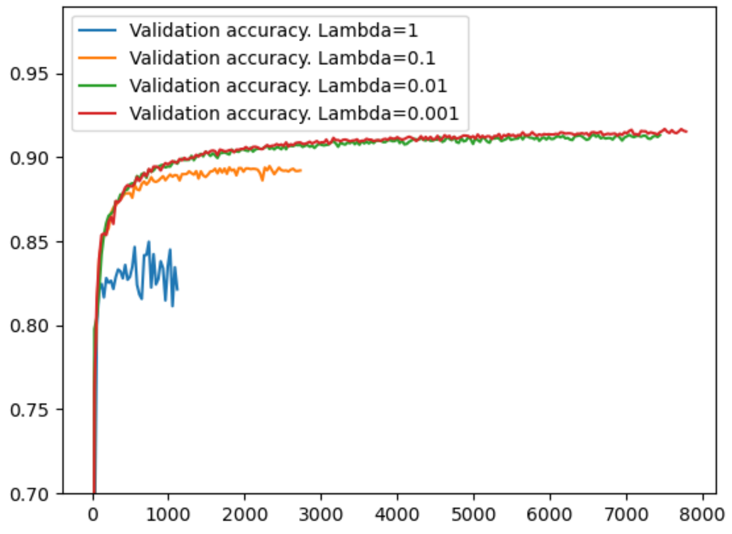

Figure 7: Task 4c.

(d) [5*pt*] **(report)** You will notice that the validation accuracy degrades when applying any amount of regularization. What do you think is the reason for this?

Solution: The model complexity is already quite bad (a single layer network), which struggles to at all completely overfit to the dataset. Making a more general model is hard with a linear network.

(e) [5*pt*] **(report)** Plot the length (L_2 norm, $||w||^2$) of the weight vector for the each λ value in task 4c. What do you observe? Plot the λ value on the x-axis and the L_2 norm on the y-axis.

Note that you should plot the L_2 norm of the weight **after** each network is finished training.

Solution: Higher values of lambda drive the L2 norm i.e. the magnitude of the weights down.

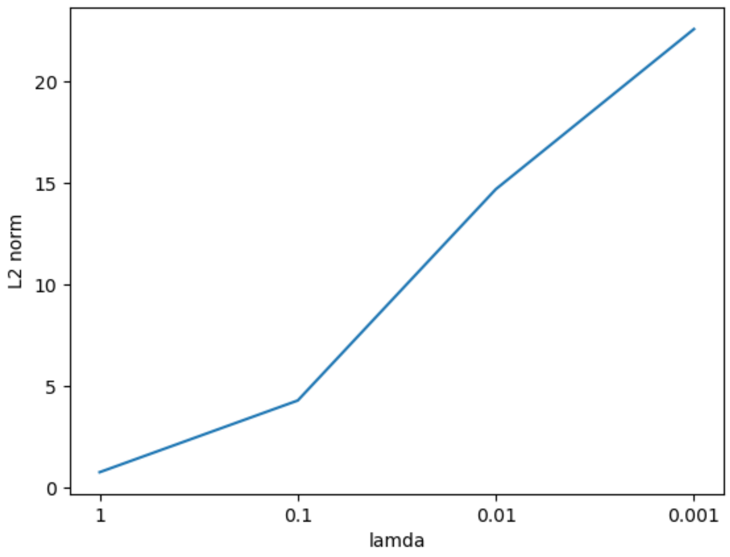

Figure 8: Task 4e

Appendix

Gradient Approximation test

When implementing neural networks from the bottom up, there can occur several minor bugs that completely destroy the training process. Gradient approximation is a method to get a numerical approximation to what the gradient should be, and this is extremely useful when debugging your forward, backward, and cost function. If the test is incorrect, it indicates that there is a bug in one (or more) of these functions.

It is possible to compute the gradient with respect to one weight by using numerical approximation:

$$\frac{\partial C^n}{\partial w_{ji}} = \frac{C^n(w_{ji} + \epsilon) - C^n(w_{ji} - \epsilon)}{2\epsilon}, \tag{29}$$

where ϵ is a small constant (e.g. 10^{-2}), and $C(w_{wij} + \epsilon)$ refers to the error on example x^n when weight w_{ji} is set to $w_{ji} + \epsilon$. The difference between the gradients should be within big-O of ϵ^2, so if $\epsilon = 10^{-2}$, your gradients should agree within $O(10^{-4})$.

If your gradient approximation does not agree with your calculated gradient from backpropagation, there is something wrong with your code!