TDT4265: Computer Vision and Deep Learning

Assignment 1

Håkon Hukkelås
hakon.hukkelas@ntnu.no
Department of Computer Science
Norwegian University of Science and Technology

January 16, 2020

- **Delivery deadline: Friday, January 31, 2020, by 23:59.**
- **This project count towards 6% of your final grade.**
- You can work on your own or in groups of up to 2 people.
- Upload your code as a single ZIP file.
- Upload your report as a single PDF file to blackboard.
- You are required to use python3 to finish the programming assignments.
- The delivered code is taken into account with the evaluation. Ensure your code is well documented and as readable as possible.

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/hukkelas/TDT4265-StarterCode.

To set up your environment, follow the guide in the Github repo:
https://github.com/hukkelas/TDT4265-StarterCode/blob/master/python_setup_instructions.md

Recommended readings

1. 3Blue1Brown: What is a Neural Network?
2. 3Blue1Brown Gradient Descent
3. 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 1.1, Task1.2, Task 2.1a etc).
- **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 corresponds 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 zipfile as we expect. (Run this from the same folder as all the python files).

  To use the script, simply run: `python3 create_submission_zip.py`
- **Upload to blackboard:** Upload the ZIP file with your source code and the report to blackboard before the delivery deadline.

Late delivery & Other Penalties

We have a strict policy on late deliveries. Any delivery submitted after the deadline will be limited in maximum score which you can achieve on the assignment. See "Course Information" on blacbkoard for more information.

Also, if you fail to follow our delivery guidelines we will **subtract 0.2 points** from your final score. This includes if you do not deliver your report as a PDF file, if you don't deliver your code as .ZIP file, or if you don't use the `create_submission_zip.py` to generate your .zip file.

Task 1: Theory

Logistic Regression

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

$$P(x \in C_1|x) = f_w(x) = \frac{1}{1 + e^{-w^T x}} \tag{1}$$

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

where $f_w(x)$ simply notes that f_w is parameterized by w. $f_w(x)$ returns the probability of x being a member of class C_1; $f_w \in [0, 1]$ [1]. By defining the output of our network as $\hat{y}$, we have $\hat{y} = f_w(x)$. With this hypothesis function, 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_w does over the N data points.

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

Here, y^n is the target value (also known as the label of the image), for example n. Note that we are taking the mean over all training samples N; this is to ensure that our cost function is not dependent on 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.

Task 1a: Derive the gradient for Logistic Regression (0.6 points)

To minimize the cost function with gradient descent, we require the gradient of the cost function in Equation 3. Show that for the logistic regression cost function, the gradient is:

$$\frac{\partial C^n(w)}{\partial w_j} = -(y^n - \hat{y}^n) x_j^n \tag{4}$$

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:

$$\frac{\partial f_w(x^n)}{\partial w_j} = x_j^n f_w(x^n)(1 - f_w(x^n)) \tag{5}$$

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

Softmax Regression

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

$$\hat{y}_k^n = \frac{e^{z_k^n}}{\sum_{k'}^K e^{z_{k'}^n}} \tag{6}$$

where

$$z_k^n = w_k^T x^n \tag{7}$$

Here z_k^n is called the weighted input to unit $\hat{y}_k$. Equation 6 is known as the Softmax function and it has the attribute that $\sum_k^K \hat{y}_k^n = 1$. Note that each output has its own weight vector w_k.

With our model defined, we now define the cross-entropy cost function for multiple classes in Equation 8.

$$C(w) = -\frac{1}{N \cdot K} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k^n \ln(\hat{y}_k^n) \tag{8}$$

Again, taking the average of this number over training samples, N, makes the function independent of batch size. Also, averaging this over number of categories, K, makes it independent over the number of categories.

Task 1b: Derive the gradient for Softmax Regression (0.8 points)

For the softmax function in Equation 8, show that the gradient is:

$$\frac{\partial C^n(w)}{\partial w_{kj}} = -\frac{1}{K} x_j^n (y_k^n - \hat{y}_k^n) \tag{9}$$

w_{kj} is the weight from unit j to unit k. A few hints if you get stuck:

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

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.

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 $x \in \mathbb{R}^{785}$ (this is the bias trick). For this assignment, we will use a subset of the MNIST dataset[2].

Tip: it is recommended to use a small number of images while developing your code (for example, 100 images) for debugging purposes.

Logistic Regression through gradient descent (1.6 points)

For this task, we will use mini-batch gradient descent to train a logistic regression model. 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 to classify $x \in \mathbb{R}^{785}$ for the two categories 2's and 3's. The target is 1 if the the input is from the "2" category, and 0 otherwise. Remove all numbers that are not a 2 or 3 (this pre-processing is already implemented in the starter code).

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 4, you can compute it in one go instead of iterating over the number of examples and weights.

Listing 1 The mini-batch gradient descent algorithm for m batches and a single epoch. N is the batch size and α is the learning rate.

```
2. $w_0 \leftarrow 0$
3. for t = 0, ..., m do
4.     $w_{t+1} = w_t - \alpha \frac{1}{N} \sum_{n=1}^{N} \frac{\partial C^n(w)}{\partial w_{kj}}$
5. return $w$
```

Early Stopping: Early stopping is a tool to stop the training before your model overfits on the training dataset. Therefore, we often split the training dataset into two sets; training and validation (also known as hold-out set). The first set is to train our model with gradient descent, and the other set is to validate that our model is able to generalize to unseen data. As you train your network, regularly test your network on the validation set. If the cost function on the validation set begins to increase consistently, stop the training and return the weights at the minimum validation loss. If early stopping kicks in, you want to save the model weights where the validation loss started to increase.

For this task, please:

(a) [0.6*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 $[0, 1]$, and it should apply the bias trick.

[2] We use the first 20,000 images in the training set and the **last** 2,000 images in the test set (these are the hardest images in the test set). Note that the first half of these images are written by postal workers, representing "clean" images, while the second half is written by high school students.

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`. The backward pass computes the gradient for each parameter in our network. To find the gradient for our weight, we can use the equation derived in task 1 (Equation 4).

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!

(b) [0.6*pt*] **The rest of the task 2 subtasks should be implemented in `task2.py`.**

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.

During training, track the training loss for each gradient step in a tracking variable (this is outlined 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).

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, 0.4])` worked fine).

(c) [0.2*pt*] Implement a function that computes the binary classification accuracy[3] over a dataset. Implement this in the function `calculate_accuracy`.

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).

Finally, report the accuracy of your trained model on the training, validation and test set.

(d) [0.2*pt*] Implement early stopping into your training loop. Explain what your early stop criteria is. For example, in our reference implementation we stop training if the validation loss increases consistently after passing through 20% of the train dataset 5 times.

Increase the number of epochs to 500. After how many epochs does early stopping kick in? Do you notice any signs of overfitting?

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

Task 3: Regularization (1 point)

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. 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.

$$J(w) = C(w) + \lambda R(w), \tag{10}$$

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

$$R(w) = ||w||^2 = \sum_{i,j} w_{i,j}^2, \tag{11}$$

where w is the weight vector of our model.

For your report, please:

(a) [0.2*pt*] Derive the update term for softmax regression for gradient descent in J with respect to w, for L_2 regularization. All you have to do is figure out $\frac{\partial R}{\partial W}$.

(b) [0.4*pt*] Implement L_2 regularization, and train logistic regression. Do this for 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 (during training). What do you observe? Which λ value works best for you?

Tip: It should only be necessary to change your `backward` function and the hyperparameter `l2_reg_lambda`.

(c) [0.2*pt*] Plot the length (L_2 norm, $||w||^2$) of the weight vector for the each λ value in task 3b. What do you observe? Have 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.

(d) [0.2*pt*] Plot the weights for each value of λ as an image. Your weight has 785 elements (784+bias), plot the first 784 elements as a 28×28 image. What do you observe?

Figure 1 shows an example of how the weights might look for different λ values. Note that your result might differ slightly.

Tip: You can use `plt.imshow(...)` to show an image.

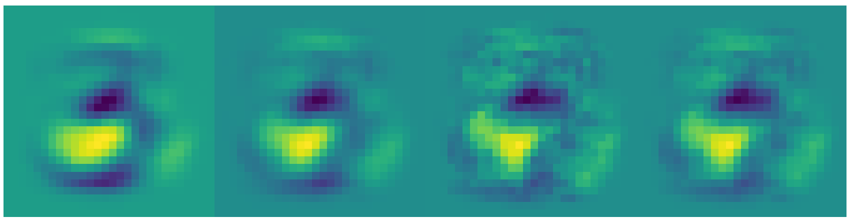

Figure 1: Task 3 weights for $\lambda = 1.0$ (leftmost image), to $\lambda = 0.001$ (rightmost image).

Task 4: Softmax Regression through Gradient Descent (2.0 points)

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 your report, please:

(a) [0.6*pt*] Before implementing our gradient descent training loop, we will implement a couple of essential functions. Implement four functions in the file `task4a.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 6.

Implement a function that performs the backward pass through our single layer neural network. Implement this in the function `backward`. The backward pass computes the gradient for each parameter in our network. To find the gradient for our weight, we can use the equation derived in task 1 (Equation 9).

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 8.

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

(b) [0.6*pt*] **The rest of the task 4 subtasks should be implemented in `task4.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.

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([0.01, 0.2])` worked fine).

(c) [0.2*pt*] Implement a function that computes the multi-class classification accuracy over a dataset. Implement this in the function `calculate_accuracy`. Include in your report a plot of the training and validation accuracy over training.

Finally, report the accuracy of your trained model on the training, validation and test set.

(d) [0.2*pt*] Implement L_2 regularization for your softmax model. Again, we have included the hyperparameter `l2_reg_lambda` to set the L_2 regularization.

(e) [0.4*pt*] Take your weight vector for each digit, remove the weight representing the bias, giving you a vector of size 784, and reshape your weight to $28x28$. Visualize the weight as a grayscale image. Repeat this for each digit (0-9).

Then, train two different models with different λ values for the L_2 regularization. Use $\lambda = 0.0$ and $\lambda = 0.1$. Visualize the weight for each digit for the two models. Why is the weights for the model with $\lambda = .1$ less noisy?

The visualization should be similar to Figure 2.

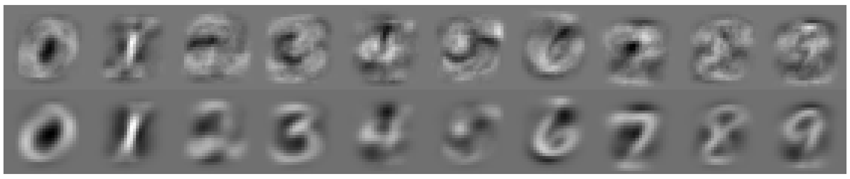

Figure 2: The visualization of the weights for a model with $\lambda = 0.0$ (top row), and $\lambda = 0.1$ (bottom row).

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:

$$\frac{\partial C^n}{\partial w_{ji}} = \frac{C^n(w_{ji} + \epsilon) - C^n(w_{ji} - \epsilon)}{2\epsilon}, \tag{12}$$

where ϵ is a small constant (e.g. 10^{-2}), and $C(w_{w_{ij}} + \epsilon)$ 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!