Assignment 2
Computer Vision and Deep Learning

Delivery Deadline: **Friday, February 17th, 23:59PM.**
You are allowed to work in groups of 1 or 2 people.

Introduction. In the previous assignment, you implemented a single-layer neural network to classify MNIST digits with softmax regression. In this assignment, we will extend this work to a multi-layer neural network. You will derive update rules for hidden layers by using backpropagation of the cost function. Furthermore, you will experiment with several well-known "tricks of the trade" to improve your network in both accuracy and learning speed. Finally, you will experiment with different network topologies, testing different numbers of hidden units, and hidden layers.

Starter Code. With this assignment, we provide you starter code for the programming tasks. We require you to use the provided files and you are not allowed to create any additional files for your code (except stated otherwise in the task). You can download the starter code from:
https://github.com/TDT4265-tutorial/TDT4265_StarterCode.

Report outline. We've included a jupyter notebook as a skeleton for your report, such that you won't use too much time creating your report. Remember to export the jupyter notebook to PDF before submitting it to the blackboard. It is not mandatory to use this report skeleton, and you can write your report in whatever program you'd like (markdown, latex, word etc), as long as you deliver the report as a PDF file.

Recommended reading.

1. Check "Recommended Resources" on the blackboard for updates.
2. Neural Networks and Deep Learning: Chapter 1 and 2
3. 3Blue1Brown: What is Backpropagation Really Doing?
4. 3Blue1Brown: Backpropagation Calculus

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, Task1.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 so that it is easy 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.
- The delivered code is taken into account with the evaluation. Ensure your code is well-documented and as readable as possible.

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

Task 1. Softmax regression with backpropagation

For multi-class classification on the MNIST dataset, you previously used softmax regression with cross-entropy error as the objective function to train a single-layer neural network. Now, we will extend these derivations to work with multi-layer neural networks. We will extend the network by adding a hidden layer between the input and output that consists of J units with the sigmoid activation function. This network will have two layers: an input layer, a hidden layer, and an output layer [1].

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, i.e. x_i. Hence, the weight from node i in the input layer to node j in the hidden layer is w_{ji}. Similarly, for node j in the hidden layer to node k in the output layer is w_{kj}. We will write the activation of hidden unit j as $a_j = f(z_j)$, where $z_j = \sum_{i=0}^{I} w_{ji}x_i$. f represents the hidden unit activation function (sigmoid in our case), and I is the dimensionality of the input. We write the activation of output unit k as $\hat{y}_k = f(z_k)$, where f represents the output unit activation function (softmax in our case). Note that we use the same symbol f for the hidden and output activation function, even though they are different. However, which f we mean should be clear from the context. This notation enables us to write the slope of the hidden activation function as $f'(z_j)$. Since we are using the bias trick (as we did in assignment 1), you can ignore the bias in your calculations. To avoid too many superscripts, we assume that there is only one data sample ($N = 1$), but extending our update rule to $N > 1$ is straightforward.

In the previous assignment, you derived the gradient descent update rule for the weights w_{kj} of the output layer:

$$w_{kj} := w_{kj} - \alpha \frac{\partial C}{\partial w_{kj}} = w_{kj} - \alpha \delta_k a_j, \tag{1}$$

where $\delta_k = \frac{\partial C}{\partial z_k}$, and ":=" means assignment. For the weights of the hidden layer, the gradient descent rule with learning rate α is:

$$w_{ji} := w_{ji} - \alpha \frac{\partial C}{\partial w_{ji}}, \tag{2}$$

Equation 2 can be written as a recursive update rule that can be applied without computing $\frac{\partial C}{\partial w_{ji}}$ directly by using the definition $\delta_j = \frac{\partial C}{\partial z_j}$.

Task 1a: Backpropagation (4 points)

By using the definition of δ_j, show that Equation 2 can be written as;

$$w_{ji} := w_{ji} - \alpha \delta_j x_i, \tag{3}$$

and show that $\delta_j = f'(z_j) \sum_k w_{kj} \delta_k$.

Hint 1: A good starting point is to try rewriting $\alpha \frac{\partial C}{\partial w_{ji}}$ using the chain rule.

Hint 2: From the previous assignment, we know that $\delta_k = \frac{\partial C}{\partial z_k} = -(y_k - \hat{y}_k)$.

Solution: Note that the notation might not match the assignment properly!

From assignment 1, we know that $\delta_k = -(t_k - y_k)$. Using this, we start by finding $\frac{\partial C}{\partial z_j}$. We take the sum of the error from all the upper layer nodes:

$$\delta_j = \frac{\partial C}{\partial z_j} = \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} \tag{4}$$

[1]Note that we only count the layers with actual learnable parameters (hidden layer and output layer).

This is found by using the chain rule. Since $f'(z_j) = \frac{\partial a_j}{\partial z_j}$ is independent of the sum, we can move this outside.

$$\delta_j = \frac{\partial C}{\partial z_j} = f'(z_j) \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial a_j} \tag{5}$$

We know that $\delta_k = \frac{\partial C}{\partial z_k}$ and $\frac{\partial z_k}{\partial a_j} = w_{kj}$, giving us:

$$\delta_j = \frac{\partial C}{\partial z_j} = f'(z_j) \sum_k \delta_k w_{kj} \tag{6}$$

Finding $\frac{\partial C}{\partial w_{ji}}$ from this, is simply using the chain rule 1 step further.

$$\frac{\partial C}{\partial w_{ji}} = \frac{\partial C}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} = \delta_j x_i \tag{7}$$

Giving us:

$$\alpha \frac{\partial C}{\partial w_{ji}} = \alpha \delta_j x_i \tag{8}$$

Task 2: Softmax Regression with Backpropagation

In this task, we will perform a 10-way classification on the digits in the MNIST dataset with a 2-layer neural network. The network should consist of an input layer, a hidden layer and an output layer. We have set the hyperparameters (learning rate and batch size) that should work fine for each task. If you decide to change them, please state them in your report. We expect you to keep/re-implement the following functions from the last assignment:

- Implementation of `one_hot_encode` and `cross_entropy_loss` (include these in task2a.py).
- Early stopping in the training loop. This is not required; however, early stopping might enable you to stop training early and save computation time. For early stopping, we use the same implementation as the last assignment, except to increase the number of observations of the validation steps to 50 instead of 10.
- Batch shuffling in `batch_loader` in utils.py

Input Normalization: Input normalization is a crucial part of optimizing neural networks efficiently. The convergence is usually faster if the average of each input variable over the training set is close to zero [LeCun et al., 2012] [2]. A simple yet efficient way to normalize your images is

$$X_{norm} = \frac{X - \mu}{\sigma}, \tag{9}$$

where μ and σ are the mean pixel value and the standard deviation over the whole training set, respectively.

For your report, please:

(a) [10*pt*] Find a mean and standard deviation value from the whole training set. Then, implement a function that pre-processes our images in the function `pre_process_images` in `task2a.py`. This should normalize our images with the input normalization trick shown in Equation 9 and apply the bias trick. Report your mean and standard deviation. Note that you should use the same mean and standard deviation value when normalising your training, validation, and test sets!

Solution: Mean: 33.31, STD: 78.57

[2] Section 4.3 in [LeCun et al., 2012] explains in detail the effect of input normalization and presents an extreme case of what can happen if you don't normalize your input data.

(b) [18*pt*] Implement the following in `task2a.py`:

- Implement a function that performs the forward pass through our softmax model. This should compute $\hat{y}$. Implement this in the function `forward`.
- Implement a function that performs the backward pass through our two-layer neural network. Implement this in the function `backward`. The backward pass computes the gradient for each parameter in our network (for both the weights in the hidden layer and the output layer).

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

(c) [6*pt*] **The rest of the task 2 subtasks should be implemented in `task2.py`.**

Implement softmax regression with mini-batch gradient descent for a multi-layer neural network. The network should consist of a single hidden layer with 64 hidden units and an output layer with 10 outputs. Initialize the weight (before any training) to randomly sampled weights between [-1, 1] [3]. You should only require to change `train_step` in `task2.py` to support multi-layer neural networks.

(report) Include a plot of the training and validation loss and accuracy over training. Have the number of gradient steps on the x-axis and the loss/accuracy on the y-axis.

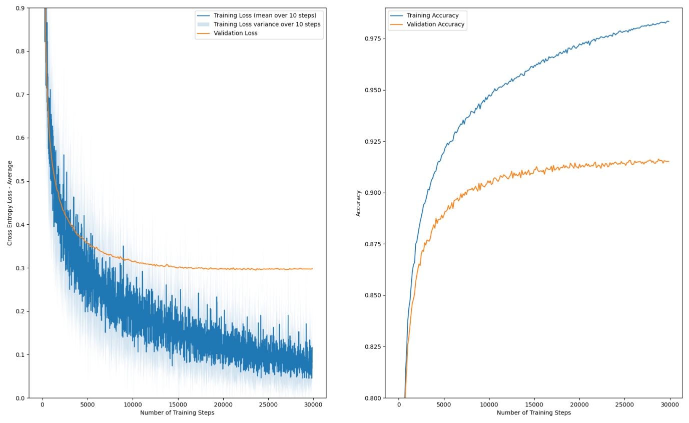

Figure 1: Training graph for task 2c

(d) [5*pt*] **(report)** How many parameters are there in the network defined in task 2c? [4]

Solution: $784 * 64 + 64(bias) + 64 * 10 + 10(bias)$. We can also accept not to include the bias in the last layer, as we are not asking them to implement the bias trick for the last layer.

Task 3: Adding the "Tricks of the Trade"

Read the paper *Efficient Backprop* [LeCun et al., 2012] Section 4.1-4.7. The paper can be found with the assignment files. Implement the following ideas from the paper. Do these changes incrementally, i.e., report your results, then add another trick, and report your result again. This way you can observe what effect the different tricks improve in the network.

Momentum: LeCun *et al.* [LeCun et al., 2012] define the momentum update step as,

$$\Delta w(t+1) = \alpha \cdot \frac{\partial C}{\partial w} + \gamma \Delta w(t) \tag{10}$$

[3] You can use the function `np.random.uniform(-1, 1, (785, 64))` to get a weight with shape $[785, 64]$ sampled from a random uniform distribution.

[4] Number of parameters = number of weights + number of biases

where $\Delta w(t+1)$ is the weight update for step t and γ is the strength of the momentum term. Following this notation, we can now write Equation 2 as $w_{ji} := w_{ji} - \Delta w_{ji}(t)$. Instead of using LeCun's definition, will use the standard way of implementing it in neural network frameworks:

$$\Delta w(t) = \frac{\partial C}{\partial w} + \gamma \cdot \Delta w(t-1) \tag{11}$$

and update our weights with $w_{ji} := w_{ji} - \alpha \cdot \Delta w_{ji}(t)$, where $\Delta w_{ji}(t-1)$ is set to 0 for the first gradient step.

Implement the following:

(a) [6*pt*] Initialize the input weights from a normal distribution, where each weight use a mean of 0 and a standard deviation of $1/\sqrt{\text{fan-in}}$. Fan-in is the number of inputs to the unit/neuron. Implement it in `__init__` for your model in `task2a.py`, where `use_improved_weight_init` can be used to toggle it on/off.

(b) [9*pt*] For the hidden layer only, use the improved sigmoid in Section 4.4. Note that you will need to derive the slope of the activation function again when you are performing backpropagation. Implement it in your model in `task2a.py`, where `use_improved_sigmoid` can be used to toggle it on/off.

(c) [7*pt*] Implement momentum to your gradient update step. Use momentum with $\mu = 0.9$. Note that you will need to reduce your learning rate when applying momentum. For our experiments, a learning rate of 0.02 worked fine. Implement it in `train_step` in `task2.py`, where `use_momentum` can be used to toggle it on/off, and `momentum_gamma` is the momentum strength.

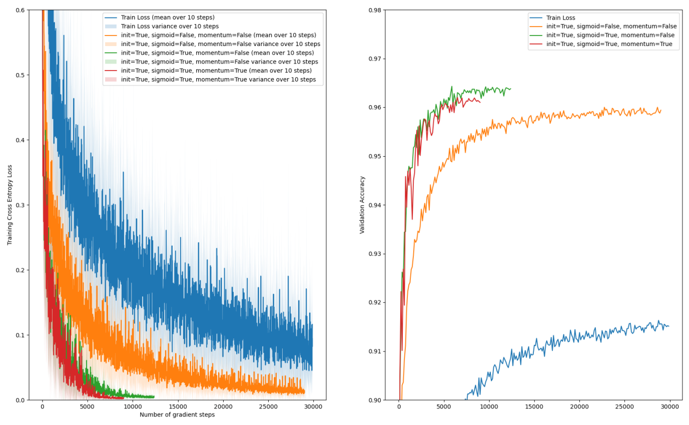

Figure 2: Task 3 solution

In your report, please: Shortly comment on the change in performance, which has hopefully improved with each addition, at least in terms of learning speed. Note that we expect you to comment on convergence speed, number of training steps used, generalization/overfitting, and final accuracy/validation loss of the model. Include a plot where the losses and accuracy after each addition are plotted in the same figure for comparison. We've included `task3.py` as an example on how you can create this comparison plot. You can extend this file to solve this task.

Task 4: Experiment with network topology

Start with your final network from Task 3. Now, we will consider how the network topology changes the performance.

In you report, please answer the following:

(a) [3*pt*] **(report)** Set the number of hidden units to 32. What do you observe if the number of hidden units is too small?

(b) [3*pt*] **(report)** Set the number of hidden units to 128. What do you observe if the number is too large?

For 4a and 4b you can include a plot of either loss or accuracy to support your statements.

(c) [14*pt*] Generalize your implementation of your softmax model to handle a variable number of hidden layers. You can modify your model from task 2a. The variable `neurons_per_layer` specifies the number of layers (length of the list) and the number of units per layer.

To test your implementation you can run the code in `task4c.py`, where we use gradient approximation to test your model on a network with 2 hidden layers.

Hint: When adding a new hidden layer, you can copy the update rule from the previous hidden layer (This is the beautiful part of backpropagation!).

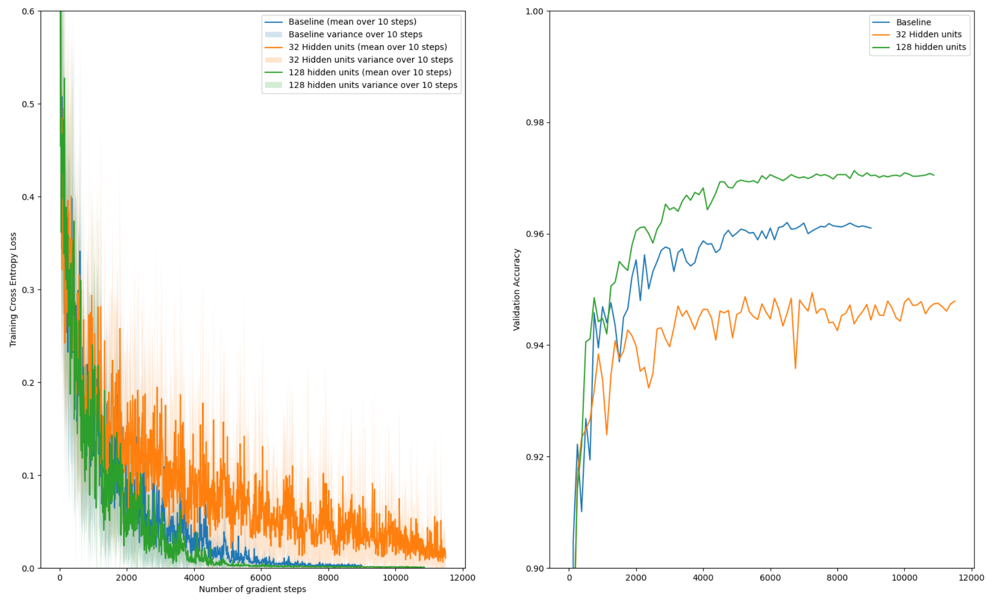

Figure 3: Task 4ab

(d) [5*pt*] Create a new model with two hidden layers of equal size. The model should have approximately the same number of parameters as the network from task 3 [5]. Train your new model (you can use the same hyperparameters as before).

(report) In your report, state the number of parameters there are in your network from task 3, and the number of parameters there are in your new network with multiple hidden layers. Also, state the number of hidden units you use for the new network.

(report) Plot the training, and validation loss over training. Repeat this plot for the accuracy.

(report) How does the network with multiple hidden layers compare to the previous network?

[5]Remember, number of parameters = number of weights + number of biases

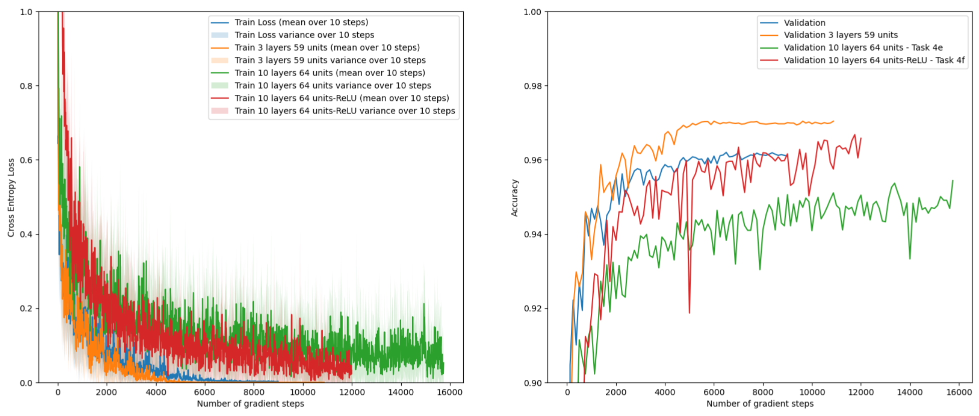

Figure 4: Task 4d

(e) [6*pt*] Train a model with ten hidden layers where each layer has 64 hidden nodes.

(report) Plot the training loss in the same graph as your baseline from task 3 (you can plot it in the same graph as task 4d). What happens with the model? What is the reason for the change in model performance?

Solution: This is the problem of vanishing gradients. If you print out the gradient for each layer, notice that it vanishes the deeper you get.

(f) [4*pt*] Instead of improved sigmoid, implement the ReLU activation function $f(x) = max(0, x)$. What happens to the model performance?

Solution: ReLU solves vanishing gradient problem and allows the model to achieve baseline performance even with deeper network on such a small dataset.

References

[LeCun et al., 2012] LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer.