Lab Assignment 2

Overview

Welcome to the second lab assignment for Neural Networks and Deep Learning! Please be sure to **read this document in its entirety**. In this assignment, we study the challenges of training *deep* neural networks. Networks with many layers are much more difficult to train than the shallow 1-2 layer models studied in the first lab, and therefore require many careful design choices. We will explore various techniques for training deep networks including momentum-based optimizers, non-saturating activation functions, and residual connections. This assignment is worth **50 points** distributed across 3 parts, and an extra credit opportunity is provided which is worth **5 points**.

Submission Instructions

Your submission must include two parts submitted as a **single .pdf file**:

1. **A lab report**: In contrast to lab 1, your report must follow the methods/results/analysis structure outlined in the report guidance document on Canvas. Important items to include in the report are highlighted in **blue** throughout this manual. The report should be about 3-5 pages long, but you may exceed this page limit if desired.

2. **Appendix with your code**: All your code must be copy-pasted as **plain text** into the appendix. Submissions of code screenshots will result in a penalty to your grade.

Programming Requirements

For this assignment, you will design your own models and implement everything necessary to train them. Implementations of existing architectures are **NOT PERMITTED**. You are however permitted to use the following PyTorch modules:

1. `torch.nn.Conv2d()` and `torch.nn.MaxPool2d()`

2. `torch.nn.Conv1d()` and `torch.nn.MaxPool1d()`

3. `torch.nn.RNNCell()` and `torch.nn.Embedding()`

4. `torch.nn.Linear()` and `torch.nn.Bilinear()`

You are required to implement the **forward pass** for all other model components, including activation functions. You may freely use autograd to compute the backward passes for all models. However, the use of PyTorch optimizers is **not allowed**. Specifically, you are permitted to use `loss.backward()` to update the gradients of all parameter tensors, but all optimizer classes from `torch.optim` are not permitted. These programming requirements are summarized below:

1. The forward pass for all models must be implemented from scratch. You may use as many of the permitted modules as you wish.

   (a) You may freely use arithmetic and tensor utility operations like `flatten()` and `matmul()` to implement the forward pass.

2. Training loops, mini-batch logic, and loss computation must be implemented from scratch.

3. Gradient resetting and parameter update steps must be implemented from scratch.

Programming Hints

1. Most PyTorch modules have `module.weight` and `module.bias` attributes. You can use these to directly access the parameter tensors used in the permitted building blocks.

2. All tensors which have `tensor.requires_grad` set to `True` will store the gradients computed by autograd in the `tensor.grad` attribute.

   (a) Remember to reset all parameter gradients before computing backward passes with `loss.backward()`. This can be accomplished by setting `tensor.grad = None` for all relevant parameter tensors.

   (b) `loss.backward()` does **not** apply any parameter updates automatically. You must use each `tensor.grad` to implement the desired update rule.

Part 1: Baseline Models [10 Points]

To conduct meaningful scientific experiments, you will need a solid baseline model to compare against. This will enable analysis of the various techniques we will explore in parts 2 and 3. To accomplish this goal, select a dataset and design a simple classification model for the task.

Selecting a Difficult Dataset

Choose a real-world dataset not covered in the programming tutorials that is designed for the multi-class classification problem. You can use datasets from any existing library (e.g. PyTorch, Tensorflow), website (e.g. Kaggle, Hugging Face), or even use your own data! However, there are several requirements of the dataset:

1. The dataset must be for the multi-class classification task.

2. The dataset must contain at least 1000 total samples.

3. The dataset must be too difficult for shallow models, i.e., a 1-2 layer networks should not achieve more than 50% accuracy.

The programming tutorials are a good place to look for inspiration.

Data Preprocessing Preprocessing the dataset samples is a critical step in ensuring deep neural networks can successfully learn the task. Therefore, it is essential that you preprocess your chosen dataset, if it does not already come preprocessed. Some common preprocessing techniques are listed here:

1. Normalization. These techniques center the data distribution by ensuring the dataset has zero mean and unit variance.

2. Dimension reduction. These techniques leverage tools like principal component analysis (PCA) to reduce the input dimensionality of the data.

3. Outlier removal. These techniques leverage statistical measures to remove unrepresentative samples from the dataset.

You **are not** required to implement any of the above preprocessing techniques; they are listed to provide a starting point. However, you **should** preprocess your chosen dataset so that the models can learn effectively. For example, if your dataset does not already come preprocessed, you should at a minimum normalize the samples to have zero mean and unit variance. If your chosen dataset does not have a train/test split, divide it based on a 70/30 split.

Evaluating the Dataset Difficulty Build a simple two-layer network to verify that your chosen dataset requires deep learning techniques. The model should be essentially the same as the two-layer architecture from the previous lab assignment. However, you will need to adjust the input and hidden dimensions according to your chosen dataset, and adjust the output layer for use with the multi-class classification task. You are free to decide the hidden dimension. The model should accept as input flattened versions of the samples from the dataset, e.g., for an image dataset with 32×32 color images, the inputs should be vectors of dimension $3072 = 3 \times 32 \times 32$. Use the sigmoid activation function for the hidden layer, and softmax for the output.

Deliverable: Discuss your chosen dataset and preprocessing steps in your report's **methods** section, and report the performance of the two-layer model in the **results** and **analysis** sections.

Building a Baseline Deep Network

Next, you will need to construct a more complex network to serve as the baseline for future experiments. This network should consist of at least 5 parameterized layers with the sigmoid activation function in-between them, and softmax for output. It is highly recommended to use some of the permitted building blocks based on your chosen dataset's modality. Specifically, you might use 2-d convolutions for images, recurrent layers for text data, and 1-d convolutions for audio. Note that you will likely need to supplement these building blocks with linear layers for the final classification output. This baseline model will form the basis for all experiments in this lab, so you are encouraged to experiment with different architecture structures. However, the final version of the baseline model must consist **only** of sequential parameterized layers and sigmoid/softmax activations.

Training the Baseline Models Implement the stochastic gradient descent (SGD) optimization algorithm, and use **only** this optimizer for the baseline models. Recall that the update rule for SGD is given by:

$$\theta^{i+1} \leftarrow \theta^i - \eta \frac{d\mathcal{L}}{d\theta^i} \tag{1}$$

where η is the learning rate, and $\frac{d\mathcal{L}}{d\theta^i}$ is the loss gradient with respect to the model's parameters θ^i at the i^{th} iteration of training. Process each sample one-at-a-time, i.e., use a batch size of 1. You are free to choose the learning rate, number of epochs, and any other relevant hyper-parameters. Be sure to train the models long enough that they converge properly. Record the performance of the baseline model for future comparisons.

Deliverable: Discuss the design of the baseline model in the **methods** section, and its performance in the **results** and **analysis** sections. Be sure to include all details necessary for a knowledgeable reader to reproduce your results.

Part 2: Activation Functions and Optimizers [20 Points]

In this part, we will experiment with different batch sizes, non-linear activation functions, and optimization algorithms with the goal of improving the performance of the baseline model. Activation functions directly influence the loss landscape of deep networks, thereby affecting their ability to extract patterns from the dataset. On the other hand, optimization algorithms influence the manner in which the model parameters navigate the loss landscape. Together, these techniques can significantly improve the performance of deep networks.

Activation Functions

First, select two activation functions from the list below:

1. Leaky ReLU:
$$f(x) = max(x, 0.1x) \tag{2}$$
2. Hyperbolic tangent:
$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$
3. SiLU:
$$f(x) = \frac{x}{1 + e^{-x}} \tag{4}$$
4. Gaussian:
$$f(x) = e^{-(x^2)} \tag{5}$$

Implement the forward pass for your selected activation functions. Then, create modified versions of the baseline model by replacing sigmoid with each new activation function. Train these modified models with the SGD optimizer and a batch size of 1. Use the same number of epochs used to train the baseline model.

Deliverable: Discuss your chosen activation functions in the **methods** section, and the performance of the resulting models in the **results** and **analysis** sections. Include a hypothesis as to which activation function will perform the best.

Optimizers

So far, we have used a batch size of 1 to train the models. This can lead to high amounts of noise in the sampled gradients, which influences model learning. Therefore, the first modification we make to the optimization algorithm is to incorporate *mini-batches*. Then, we augment mini-batch gradient descent with a learning acceleration technique called *momentum*.

Mini-Batch SGD The idea of mini-batch SGD is straightforward: collect n samples of the gradient and average them before updating the parameters. This reduces the amount of gradient noise and improves model learning. Denote the function defined by the model's parameters at the i^{th} training iteration by f_{θ^i}, and let the dataset samples be indexed as $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)$. The update rule for mini-batch SGD with batch size b is then given by:

$$\begin{gathered}
\hat{y}_1 = f_{\theta_i}(\mathbf{x}_1), \quad \hat{y}_2 = f_{\theta_i}(\mathbf{x}_2), \ ..., \ \hat{y}_b = f_{\theta_i}(\mathbf{x}_b) \\
\mathcal{L}_1 = \mathcal{L}(\hat{y}_1, y_1), \quad \mathcal{L}_2 = \mathcal{L}(\hat{y}_2, y_2), \ ..., \ \mathcal{L}_b = \mathcal{L}(\hat{y}_b, y_b) \\
\bigtriangledown_{\theta^i}\mathcal{L}_1 = \frac{d\mathcal{L}_1}{d\theta^i}, \quad \bigtriangledown_{\theta^i}\mathcal{L}_2 = \frac{d\mathcal{L}_2}{d\theta^i}, \ ..., \ \bigtriangledown_{\theta^i}\mathcal{L}_b = \frac{d\mathcal{L}_b}{d\theta^i} \\
\theta^{i+1} \leftarrow \theta^i - \eta\left[\frac{1}{b}\sum_{k=1}^{b} \bigtriangledown_{\theta^i}\mathcal{L}_k\right]
\end{gathered} \tag{6}$$

Implement the mini-batch SGD optimization algorithm for arbitrary b. Note that because the gradient $\bigtriangledown$ is a linear operator, the average of the gradients is the gradient of the average. Explicitly:

$$\frac{1}{b}\sum_{k=1}^{b} \bigtriangledown_{\theta^i}\mathcal{L}_k = \frac{1}{b}\sum_{k=1}^{b} \frac{d\mathcal{L}_k}{d\theta^i} = \frac{d}{d\theta^i}\left[\frac{1}{b}\sum_{k=1}^{b}\mathcal{L}_k\right] = \bigtriangledown_{\theta^i}\left[\frac{1}{b}\sum_{k=1}^{b}\mathcal{L}_k\right] \tag{7}$$

The right hand side of this equation is much easier to implement.

Next, experiment with different batch sizes to retrain the best performing model. Specifically, use the baseline architecture with the top performing activation function from the previous part. As before, use the same number of epochs as in the previous experiments. Recall that an epoch is defined as a set of training iterations which use every sample from the dataset.

Deliverable: Discuss your implementation of mini-batch SGD in the **methods** section, and its influence on model learning in the **results** and **analysis** sections. Comment on which batch size worked the best for your architecture.

Mini-Batch SGD with Momentum The momentum optimization algorithm exploits the observation that typical deep neural network loss landscapes have a primary minimization direction which is obfuscated by smaller scale noise. Think of hiking down to

Boulder from the town of Nederland (located west in the front range mountains). While there is a primary downhill direction (east), there are many smaller uphill directions produced by the smaller hills. The purpose of momentum is to allow the training process to focus on the primary descent direction. This is accomplished by computing a moving average of the gradients, as follows:

$$\begin{aligned} m^1 &= 0 \\ g^i &= \frac{1}{b}\sum_{k=1}^{b} \nabla_{\theta^i} \mathcal{L}_k \\ m^{i+1} &= \alpha m^i + g^i \\ \theta^{i+1} &\leftarrow \theta^i - \eta m^{i+1} \end{aligned} \tag{8}$$

Here, m is the moving average of the gradients, and $\alpha \in [0, 1]$ is a constant which controls the momentum strength. Initially, m is set to zero, and updated iteratively alongside the model parameters. Note that when $\alpha = 0$ we recover exactly the original mini-batch SGD update rule.

Implement mini-batch with momentum for arbitrary α. Then, experiment with different values of α to retrain the best model so far. That is, add in momentum to the training of the model from the previous section. Continue to use the same number of epochs as the previous experiments. $\alpha = 0.9$ is usually a safe place to start.

Deliverable: Discuss your implementation of mini-batch SGD with momentum in the **methods** section, and its influence on model learning in the **results** and **analysis** sections. Comment on which value of α worked the best for your architecture.

Part 3: Skip Connections [20 Points]

As we observed in the first problem set, skip connections can increase the magnitude of the gradients in deep networks. They can be immensely useful for training larger models as their presence counteracts the vanishing gradient problem. In this final part, we empirically study the effectiveness of skip connections.

Extending the Model

Skip connections are most easily implemented between layers of the same shape, i.e., layers which output tensors with the same number and organization of elements. Therefore, you will need to add layers to your model to ensure there are valid locations for skip connections. To accomplish this, add 10 layers to your architecture which do not change the shape of the output tensors. You may add these extra layers to whichever portion of the model you like, but be sure to pair these with non-linear activation functions after each layer. As an example, suppose we are working with a 2-d convolutional model designed for images. We could add 10 2-d convolution layers in-between pooling layers, and use sufficient zero padding to ensure that the feature shapes remain constant.

Train your extended model using the mini-batch SGD with momentum optimizer. Use the same hyper-parameters from the previous part, i.e., use the best performing batch size and momentum strength from your experiments. Train the model for the same number of epochs. Record the average gradient size (measured with the L_1-norm) over the first epoch of training. Recall that the L_1-norm of a tensor is the sum of the absolute value of its entries. These gradient norms will be used for analysis in the next section.

Deliverable: Discuss the design of your extended architecture in the **methods** section, and its performance relative to the original architecture in the **results** and **analysis** sections. Provide a hypothesis on if the extra layers will improve performance.

Adding Skip Connections

Recall that a skip connection is an additive link in a computational graphs which short-circuits one or more operations. For a tensor $\mathbf{x}$ and functions $f(\cdot), g(\cdot), h(\cdot)$, some possible skip connections are given by:

$$\begin{aligned}\mathbf{y}_1 &= \mathbf{x} \oplus f(\mathbf{x}) \\ \mathbf{y}_2 &= \mathbf{x} \oplus g(f(\mathbf{x})) \\ \mathbf{y}_3 &= \mathbf{x} \oplus h(g(f(\mathbf{x}))) \\ \mathbf{y}_4 &= h(f(\mathbf{x}) \oplus g(f(\mathbf{x})))\end{aligned}$$

where $\oplus$ denotes the element-wise addition operation. Observe that $\mathbf{y}_1$, $\mathbf{y}_2$, and $\mathbf{y}_3$ use skip connections of length 1, 2, and 3, respectively. This is because $\mathbf{x}$ skips one operation (f) in the computation of $\mathbf{y}_1$; $\mathbf{x}$ skips two operations (f, g) in the computation of $\mathbf{y}_2$, etc. $\mathbf{y}_4$ demonstrates how a skip connection of length one might be implemented partway through a computational graph.

Modify your extended architecture by testing at least 2 different skip connection configurations with 3 skip connections each. You may add skip connections between whichever layers you like. Experiment with different configurations of skip connections; try to improve the model's performance as much as you can. As before, use the same number of epochs as previous experiments. Continue to use mini-batch SGD with momentum with the same batch size and momentum strength as the previous parts.

Deliverable: Discuss the design of your two skip connection architectures in the **methods** section, and their performance relative to the extended architecture in the **results** and **analysis** sections. Provide the average gradient size across the layers of all three extended models. Comment on which skip connection architectures produced the best performance and why.

Report Instructions

Key components of the report are summarized here. Please read the report guidance document for more details on writing a good lab report. Minimally, the report should address everything listed in the **deliverables**, be well formatted, clearly understandable, and convey what you found. Details on each section of the report are discussed in the next sections.

Methods

The first section of your report should discuss *what* your experiments aim to study (e.g. which regularization techniques are you using?), *how* it will be conducted (e.g., which variables will be changed? Which left fixed as controls?), and *what* you expect to find (hypothesized outcome). You **MUST** report all architectural and training details (including hyper-parameters!) used in your study. This section should include a discussion of how the baseline architecture was designed. When writing this section, ask yourself "could someone else reproduce this experiment solely based on the report?" If the answer is no, then you need to add detail. You should convey the experimental design in words; copy-pasting code is **NOT** acceptable for this section.

Results

The second section of your report should provide the results of your experiments. This can be accomplished with plots, tables, charts, etc. Your figures **MUST** be clearly formatted, well annotated, and understandable by anyone with a baseline knowledge of deep learning. Remember to include labels for axes, titles/captions for figures, legends, and any other information required for the reader to understand your results. There should be *zero* ambiguity about what information is provided by the figures. You are free to decide how to format the results, but there are a few requirements. First, the test accuracy of all models should be included. Second, report the average recall and precision (across all classes) for each model.

Analysis

The third and final section of your report should discuss what you learned from your experiments. This should be more than a superficial reiteration of what can be observed in your results. Do **NOT** simply state what the results are; rather discuss *what the results say*. What trends did you observe from the experiments? Why do you think the observed trends occur? Which of the techniques explored (activation functions, optimizers, skip connections) had the most significant effect on model performance? Why? You should discuss in depth at least *2 trends* in this section.

Extra Credit: Weight Decay [5 Points]

Weight decay is a form of *regularization*, that is, a technique for adjusting the type of solution found by the training process. Specifically, weight decay penalizes large model weights by modifying the parameter update rule. This encourages the model to find a solution with smaller weights, which can help generalization by preventing excessively large activation values. The update rule for SGD with momentum and weight decay is given by:

$$\begin{aligned} m^1 &= 0 \\ g^i &= \frac{1}{b}\sum_{k=1}^{b} \nabla_{\theta^i} \mathcal{L}_k \\ m^{i+1} &= \alpha m^i + g^i + \beta\theta^i \\ \theta^{i+1} &\leftarrow \theta^i - \eta m^{i+1} \end{aligned} \tag{9}$$

The only difference from SGD with momentum is the inclusion of the $\beta\theta^i$ term in the update rule for m. The purpose of this term is to push the parameters towards zero, weighted by β which is a hyperparameter controlling the strength of the weight decay.

Implement the SGD with momentum and weight decay optimization algorithm. Retrain both the best network without skip connections and the best network with skip connections. Experiment with different values of β to improve the models' performance. Usually, $\beta = 0.0001$ is a safe starting point.

Deliverable: Provide and discuss the results of the weight decay experiments at the end of the report in a section labeled **Extra Credit**. Analyze the influence of weight decay on learning effectiveness and comment on the best value of β.

Collaboration versus Academic Misconduct: Collaboration with other students and AI is permitted, but the work you submit must be your own. Copying/plagiarizing work from another student or AI is not permitted and is considered academic misconduct. For more information about University of Colorado Boulder's Honor Code and academic misconduct, please visit the course syllabus.