SS 2025 Course Coordinator: Andre Herz
© Prof. Georgia Koppe `andre.herz@iwr.uni-heidelberg.de`

Machine Learning Essentials

Exercise Sheet 05

Due: 09.06.2025 11:15

This sheet transitions from manual backpropagation to more advanced **training techniques** and introduces the **PyTorch library**. We'll explore different gradient descent variants, the importance of proper dataset splitting for hyperparameter tuning and model evaluation, and common practices for optimizing neural network training.

Regulations

Please submit your solutions via Moodle in teams of **3** students. The coding tasks must be completed using the template `MLE25_sheet05.ipynb`. Each submission must include **exactly** one file:

- Upload a `.pdf` file containing both your Jupyter notebook and solutions to analytical exercises.

The Jupyter notebook can be exported to pdf by selecting **File** → **Download as** → **pdf** in JupyterLab. If this method does not work, you may print the notebook as a pdf instead. Your analytical solutions can be either scanned handwritten solutions or created using LaTeX.

Exercise 1: Optimization and Training Tricks

In the previous sheet, you've implemented **batch gradient descent (BGD)**, where the entire training set is used to compute the gradient for each parameter update. While BGD gives an accurate estimate of the gradient, it can be computationally infeasible for large datasets. This exercise explores alternative **gradient descent methods** and other **optimization concepts**[1].

[1]Training modern ML/deep learning model architectures often involves the application of many different "training tricks", some of which are not covered here, like e.g. batch normalization, dropout or specific initialization schemes for parameters. For a concise overview of some important ones, see `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks`.

Tasks

1. **Comparing Gradient Descent Methods**
   Choosing the right GD method is crucial for efficient training of ML models. Explain how the following optimization algorithms are defined: **Batch Gradient Descent (BGD)**, **Stochastic Gradient Descent (SGD)**, and **Mini-batch SGD**. Also, for each of them, briefly mention some of the advantages and disadvantages in terms of computational cost per update, speed of convergence (consider both number of epochs and time on the clock) and the potential effect of noise in the gradient estimate on navigating the loss landscape.

   (3 pts.)

2. **Learning Rate Schedules**
   Using a fixed learning rate for GD is not always ideal for navigating rough loss landscapes. A common method is to apply **learning rate schedules**, where the learning rate is dynamically adapted during training.

   (a) Why might a fixed learning rate not be optimal throughout the entire training process of a neural net (you may do a sketch to explain this)?

   (b) Describe the general concept of a learning rate schedule. Give a formal example of one type of schedules (e.g. exponential decay or cosine annealing) and explain how it modifies the learning rate over time. How does it help improve training?

   (2 pts.)

3. **Validation and Hyperparameter Tuning**
   As we've already seen, effectively training ML models involves more than just minimizing the training loss. Instead, our primary goal is that our models **generalize** well to unseen data. Often, this involves selecting appropriate **hyperparameters**, like number of epochs, batch size, or regularization strength. So far, we have only considered splitting our data into a **training set** we use to inform our estimation of the model's parameters, and a **test set** we use to evaluate the model's generalization performance on. However, during model development, when we **iteratively refine our approach**, e.g. by

   - Hyperparameter tuning,
   - Selecting between different models,
   - Data preprocessing (e.g. feature scaling), or
   - Trying out different regularization strategies,

   then it is **crucial** to also hold out a **validation set**.

   (a) Explain the distinct roles of the **training set**, **validation set**, and **test set** in the typical machine learning workflow.

(b) Explain why it is crucial **not** to use the test set for iterative model refinement. Relate your discussion to the **bias** and **variance** of the estimate for the model's generalization performance (see Exercise 3.1).

(c) Briefly describe the common strategy of **grid search** for hyperparameter tuning, explaining how the validation set is utilized in this process.

(3 pts.)

4. **Advanced Optimizers**
While SGD provides a good approach to training neural nets, modern ML heavily relies on **advanced optimizers** like **Adam, RMSProp and Adagrad**. These have become standard for training modern deep learning models more efficient than with "vanilla SGD". For these complex models, the right choice of optimizer can provide significant boosts in training speed and performance.

(a) On a conceptual level, explain the core mechanisms behind the

i. RMSProp and the

ii. Momentum

optimizer.

(b) One of the most popular optimizers is **Adam (Adaptive Moment Estimation)**. It's widely used due to its efficiency and adaptive learning rate capabilities. It achieves this by basically maintaining an estimation of the exponentially decaying average of past gradients (**first moment** m_t) and past squared gradients (**second moment** v_t). The core update of parameters using Adam is

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates for the moments[2], and ϵ is a small constant added to avoid numerical stability issues. Let's dissect a single update step to understand its components. Consider a specific parameter w at timestep t. Suppose that its first/second moment estimate from the previous step is $m_{t-1} = 0.5$ and $v_{t-1} = 0.2$, respectively. The current gradient for w is $g_t = 2.0$. We apply Adam's standard update rule:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

Set $\beta_1 = 0.9$, $\beta_2 = 0.99$. Let the learning rate $\alpha = 0.01$ and $\epsilon = 10^{-8}$. For simplicity in this exercise, assume we are many steps into training, so the bias correction factors are close to 1, meaning we can approximate $\hat{m}_t \approx m_t$ and $\hat{v}_t \approx v_t$.

[2]The complete update procedure for Adam is nicely described in Algorithm 1 of the original paper from ICLR 2015: `https://arxiv.org/pdf/1412.6980`.

i. Calculate the updated first moment m_t and the updated second moment v_t for parameter w.

ii. Using these m_t and v_t, calculate the update step $\Delta w_t = -\alpha \frac{m_t}{\sqrt{v_t + \epsilon}}$ for parameter w.

iii. Now, imagine another parameter, w', had the same $m'_{t-1} = 0.5$ and current gradient $g'_t = 2.0$ as w. However, w' has a history of much larger squared gradients, resulting in $v'_{t-1} = 20$. Briefly explain:
   - How would v'_t qualitatively compare to v_t?
   - Consequently, how would $|\Delta w'_t|$ compare to $|\Delta w_t|$?
   - What does this comparison imply about Adam's behavior?

(5 pts.)

Exercise 2: Getting Familiar with PyTorch

In this exercise (and following ones), we'll be using the **PyTorch** library to build, train, and evaluate our models. PyTorch is a widely-used, powerful framework for machine/deep learning. Its fundamental datastructure is the `tensor`, which functions similar as NumPy's `ndarray`, but with the crucial property of tracking computational operations for **automatic differentiation (AD)**, which automates backpropagation. To get started with the core concepts and tensor operations, it is **highly recommended** to explore the "Learn the Basics" tutorial on the official website. A guide to installing PyTorch in your environment can be found here (don't forget to `activate` your environment before running the command).
In this exercise, we'll apply PyTorch to train an MLP for classification of the **FashionMNIST** dataset, which consists of 28x28 grayscale images of 10 different 'classes' of clothing. We'll also apply the concepts introduced in Exercise 1. Thus, we'll now approach a much more "realistic" representation of the typical ML workflow than the one we have illustrated e.g. on Sheet 0.

Tasks

1. First, we need to load the data and do the required preprocessing.

   (a) For the preprocessing, define a sequence of transformations using `torchvision.transforms`. This sequence should convert the images to PyTorch Tensors and standardize them, using $\mu = 0.286$ and $\sigma = 0.353$.

   (b) Appyling your transformations, load the FashionMNIST training and test sets using `torchvision.datasets.FashionMNIST`. Create a 5x2 subplot grid that shows example images of each class, alongside the corresponding label.

(c) As discussed in Exercise 1, the loaded training set must be split further. Create a **validation set** using 10.000 “random” samples from the original 60.000 training samples.

(3 pts.)

2. Now, let’s define the model architecture using PyTorch.

   (a) Create a PyTorch model class for your MLP, inheriting from `torch.nn.Module`.

   (b) Define the **layers** in the constructor (`__init__`):

   - Input layer: FashionMNIST images are 28x28 pixels, so how many input features (i.e. input units) do we need?
   - *k* Hidden Layers: We’ll use fully connected feedforward layers with ReLU activation here. The number of hidden layers *k* and respective units for each layer is for you to design.
   - Output Layer: Use one output unit for each of the classes.

   Use `torch.nn.Linear` for linear layers (i.e. layers with linear activation function). You can find other activation functions in the docs of `torch.nn`.

   (c) Implement the `forward` method of your model. Think about the correct input and output shapes for each layer.

   (d) In the pytorch docs for `torch.nn`, look up and set an appropriate loss function for this multi-class classification model.

(4 pts.)

3. Next, implement training and validation.

   (a) Create `DataLoader` objects for your training, validation, and test sets. Use appropriate batch sizes (this will be a hyperparameter you tune). Think about which dataset(s) need `shuffle=True`.

   (b) Implement a training loop that iterates for a defined number of epochs. **For each epoch**:

   i. Set your model to training mode (`model.train()`). Iterate through the mini-batches from the training DataLoader. For each mini-batch:

      A. Perform a forward pass,

      B. compute the loss,

      C. zero gradients (`optimizer.zero_grad`),

      D. perform backpropagation (`loss.backward`), and

      E. update parameters (`optimizer.step`).

      Compute and record the average training loss and training accuracy for the epoch.

ii. For validation, set your model to **evaluation mode** (`model.eval`). Disable gradient calculations by using `torch.no_grad` while iterating through the validation DataLoader. Calculate and record the average validation loss and validation accuracy for the epoch.

(5 pts.)

4. Now we **train the model**, **tune its hyperparameters** and finally **evaluate its performance**.

   (a) Let's compare this to the Adam optimizer. Using your MLP architecture and batch size, train the model with `torch.optim.SGD` and then again with `torch.optim.Adam` (you can take its default learning rate of 0.001). Train for a sufficient number of epochs to observe convergence behavior.

   (b) Lastly, we evaluate the performances of the trained models.

   - For both your best SGD run and your Adam run: Plot a graph that shows the evolution of the training loss and validation loss during training. Also plot another graph for the evolution of the training and validation accuracy.
   - Compare the behavior (speed, stability, final validation loss/accuracy) of SGD and Adam based on your plots. Which optimizer performed better for this task and dataset based on validation metrics?
   - Select your overall best model. Report this model's accuracy on the test set.
   - Briefly discuss your findings. Did you observe any signs of overfitting or underfitting? Are you satisfied with your best model's performance?

(5 pts.)