The Mathematics of Machine Learning
Homework Set 9

Due 16 May 2025 before 17:00
via Canvas

- You are allowed to work on this homework in pairs. One person per pair submits the answers via Canvas. Make sure to put both names on the submission.
- In this homework, you will work with a Jupyter Python notebook, but you do *not* need to submit it. Only submit your answers to the questions below.

Generalizing from Logistic Regression to Neural Networks

In the 1990s, programming and experimenting with neural networks required carefully writing `C++` and assembly code. Nowadays, there are excellent software libraries that allow us to focus on specifying *what* we want to do, and then most of how this is achieved is handled automatically behind the scenes. This allows us to achieve in minutes what would have taken months of work in the past. One popular library is PyTorch, with which you will experiment in this exercise.

Neural networks will only be introduced in the next lecture, so the exercise is split into two parts. Part I you can already do before the next lecture. In this part you will use PyTorch to train a logistic regression model using stochastic gradient descent, which allows you to get everything running. Then, after the lecture, you are all set to go into more depth and generalize the code to neural networks in Part II. You will see that this requires remarkably little additional code, even though the model is much more sophisticated.

Running the code in this exercise is interesting to get familiar with PyTorch and the actual implementation of neural networks, but, if your computer is not fast enough to do that, then you should still be able to answer all questions without actually running the code.

Part I: Setting up Logistic Regression in PyTorch

The PyTorch Quickstart Tutorial As our starting point we will use the following tutorial from the PyTorch project: `https://pytorch.org/tutorials/`

`beginner/basics/quickstart_tutorial.html`. Download the tutorial as a Jupyter notebook called `quickstart_tutorial.ipynb` from the link 'Download Notebook' at the top of the page.

In order to run the notebook, you will need to have the following Python packages installed: torch, torchvision, matplotlib. Install these using your preferred method; for instance by running

```
pip install torch, torchvision, matplotlib
```

from the command line.

The Data Read through the section 'Working with data' in the notebook. The first time it is run, this part will automatically download the Fashion MNIST classification data set into a subdirectory called 'data'. This data consists of images of fashion items of 28×28 grayscale pixels, for $K = 10$ different classes. Familiarize yourself with the type of images, the classes and the number of data points in the train and test sets by reading the description here: `https://github.com/zalandoresearch/fashion-mnist`.

The Model Read through the section 'Creating Models' in the notebook. The first line checks if your computer has any special hardware device that facilitates faster computations. Don't worry if it does not; in that case you can just use the CPU. The remainder of the 'Creating Models' section sets up a basic neural network, which we will now replace by a linear model in order to switch to logistic regression:

- In the `__init__` function for the NeuralNetwork class, add the definition for a linear function by adding the following line:

  ```
  self.linear = nn.Linear(28*28, 10)
  ```

  This specifies the model consisting of all linear functions

  $$x \mapsto x^\top \beta + \beta_0 \tag{1}$$

  from $\mathbb{R}^{784}$ to $\mathbb{R}^{10}$. Here $784 = 28 \times 28$ is the dimension of the input images after converting them to a big vector, and $10 = K$ is the number of output classes in the data. The matrix β has dimensions 784×10 and the vector β_0 has 10 components, which provide one intercept (also called *bias* term) per class.

- The `forward` function actually specifies how the model calculates its prediction. It starts by flattening its input x, which converts it from a 28×28 matrix to a vector of length 784. Tell it to use the linear function we have just defined by replacing the line `logits = self.linear_relu_stack(x)` by

  ```
  logits = self.linear(x)
  ```

That's it! We have now converted the model to be multiclass logistic regression instead of a neural network. For any input image $x \in \mathbb{R}^{28\times 28}$, the forward function now flattens it to a vector and then computes

$$\texttt{logits} = f_{\beta,\beta_0}(x) := \begin{pmatrix} x^\top \beta_1 + \beta_{0,1} \\ \vdots \\ x^\top \beta_{10} + \beta_{0,10} \end{pmatrix}$$

where β_k is the k-th row of β, corresponding to the parameters for the k-th class, and $\beta_{0,k}$ is the corresponding intercept for that class. If necessary, consult the slides about multiclass logistic regression from the course to remind yourself of the setup. Note that we have now made the intercepts explicit. NB This is the machine learning version of multiclass logistic regression.

Questions

Answer the following question:

1. [1pt] What is the number of parameters in the multiclass logistic regression model you have just implemented?

Part II: From Logistic Regression to Neural Networks

Optimization The section 'Optimizing the Model Parameters' provides the code to train the model. As its loss function, it selects "cross entropy loss", which is another name for the generalization of the logistic loss (see the slides about logistic regression) to the multiclass case:

$$L(y, f_{\beta,\beta_0}(x)) = -\log \frac{e^{f_{\beta,\beta_0}(x)_y}}{\sum_{j=1}^{K} e^{f_{\beta,\beta_0}(x)_j}}.$$

It then selects stochastic gradient descent with minibatches of size 64 and step size $\eta = 10^{-3}$. Answer the following questions about stochastic gradient descent (SGD):

2. [1pt] How many minibatches of size 64 can we construct with $N = 60\,000$ data points in the training set, and how many data points do we then have left over for the last minibatch?

3. [1pt] When performing a gradient update on a minibatch $\mathcal{B} \subset \{1, \dots, N\}$, the implementation in the notebook calculates the corresponding gradient as

$$\sum_{i\in\mathcal{B}} \nabla_{\beta,\beta_0} L\big(y_i, f_{\beta,\beta_0}(x_i)\big),$$

where ∇_{β,β_0} means that we are taking the gradient of $L\big(y_i, f_{\beta,\beta_0}(x_i)\big)$ with respect to all components of β and β_0. Notice that we are summing gradients here without dividing by the number of data points in $\mathcal{B}$. Speculate how this leads to a qualitatively different update for SGD on the last minibatch (which contains fewer data points) compared to all other minibatches.

Remark: To avoid having to think about the size of the last minibatch not matching the size of the other ones, a simple solution is to just throw away the last minibatch. This is easily implemented by adding the option "drop_last=True" to `DataLoader` when constructing `train_dataloader` and `test_dataloader`.

Generalizing to a Fully Connected Neural Network Switch back to using the neural network defined in the original notebook by setting

```
logits = self.linear_relu_stack(x)
```

The function `linear_relu_stack()` specifies a neural network with the following layers: first, a fully connected (i.e. linear) layer that maps the inputs to 512 hidden units, followed by a ReLu activation function. Then, another fully connected layer that maps the output of the previous layer to another 512 hidden units, followed again by a ReLu activiation, and finally a fully connected layer that maps to 10 classes. NB All the layers also have intercepts/bias terms.

4. [1pt] How many parameters does this neural network have?