# The Mathematics of Machine Learning
# Homework Set 9

Due 20 May 2025 before 13:00
via Canvas

- **It is mandatory to submit this homework in groups of three.** One person per group submits the answers via Canvas. Make sure to put all three 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.

In Part I you will use PyTorch to train a logistic regression model using stochastic gradient descent, which allows you to get everything running. In Part II you will generalize the code to a fully connected neural network – you will see that this requires remarkably little additional code, even though the model is much more sophisticated. Start by working on Parts I and II. Part III introduces convolutional and pooling layers, which are covered in the second lecture about neural networks; leave this part until after that lecture. Note that you only have to hand in this homework the week after the second lecture.

*The exercises in this homework should be computationally feasible even on a computer without a GPU to accelerate the training procedure. But, if your group has access to multiple computers, it is recommended to use the fastest one available.*

## 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 $28 \times 28$ 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 $\beta$ has dimensions $784 \times 10$ and the vector $\beta_0$ 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 $28 \times 28$ 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 $\beta_k$ is the $k$-th row of $\beta$, 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 questions:

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

**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 $\nabla_{\beta,\beta_0}$ 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 $\beta$ and $\beta_0$. 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`. Make this change in the notebook.

# Part II: From Logistic Regression 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 activation, 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?

# Part III: A Convolutional Neural Network

So far, all layers we have worked with required flattening each $28 \times 28$ image into a vector of length 784, discarding its spatial structure. A *convolutional layer* instead processes the image while preserving its two-dimensional layout.

**The LeNet Architecture** We will work with LeNet-5, a classical convolutional architecture. Read the introduction to section 7.6 and section 7.6.1 of `https://d2l.ai/chapter_convolutional-neural-networks/lenet.html` for background. Note that LeNet-5 uses *average* pooling rather than the max pooling covered in the lecture: instead of taking the maximum value in each pooling window, it takes the average. Add the following code inside the `__init__` function of `NeuralNetwork`, below the existing definitions:

```
self.LeNet5 = nn.Sequential(
    nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
```

```
    nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.LazyLinear(120), nn.Sigmoid(),
    nn.LazyLinear(84), nn.Sigmoid(),
    nn.LazyLinear(10)
)
```

Then update the `forward` function to remove the flatten step and use the new architecture:

```
def forward(self, x):
    logits = self.LeNet5(x)
    return logits
```

Finally, the prediction cell at the bottom of the notebook tests the model on a single image, but LeNet5 expects a minibatch. Add the line

```
x = x.unsqueeze(0)
```

before the `with torch.no_grad():` line. This inserts an extra dimension at position 0, changing the dimensions of `x` from $(1, 28, 28)$ into $(1, 1, 28, 28)$, which LeNet5 interprets as a minibatch of size 1.

5. [1pt] In Parts I and II, `self.flatten(x)` was the very first line of the `forward` function. In LeNet5, the `Flatten` step appears only halfway through the network. Why must flattening be delayed?

6. [1pt] The input to `LeNet5` has shape $(N, 1, 28, 28)$, where $N$ is the minibatch size, 1 is the number of channels (grayscale), and $28 \times 28$ are the spatial dimensions. What are the dimensions of the output after the first `AvgPool2d`? And after the second `AvgPool2d`?

7. [1pt] How many parameters does LeNet5 have? Provide a breakdown by layer, and compare the total to the fully connected network from Part II.

**Training and Observing a Problem** Run the training for 10 epochs (or 6 if it is taking too long). The optimizer is still SGD with the same settings as before.

8. [1pt] Describe a problem with the training progress of LeNet5 with SGD. How does it compare to the training in Part II?

**The Vanishing Gradient Problem** The poor training behaviour you observed is caused by the *vanishing gradient* problem. Recall that during backpropagation, gradients are multiplied by the derivative of each activation function as they propagate from the output back towards the input. The Sigmoid function $\sigma(z) = 1/(1 + e^{-z})$ has derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, which tends

to zero for inputs with large magnitude. With four Sigmoid activations stacked in LeNet5, the gradient arriving at the convolutional layers is the product of four such small numbers, making it many orders of magnitude smaller than the gradient at the output. As a result, there is no single choice of step size $\eta$ that will work well for all parameters simultaneously, and the network fails to train.

**Fixing the Problem** There are two standard remedies:

- **Switch activations to ReLU**: Replace every `Sigmoid()` in the `LeNet5` definition with `ReLU()`. The ReLU function has a derivative of 0 or 1, so gradients are not systematically shrunk during backpropagation.
- **Switch optimizer to Adam**: Replace the SGD line with

  ```
  optimizer = torch.optim.Adam(params=model.parameters(),
                               lr=0.001)
  ```

  Adam is a popular alternative to SGD, which adapts the effective step size for each parameter individually, allowing it to make progress even when the components of the gradients live on very different scales.

Train LeNet5 with each fix separately (keeping `Sigmoid` activations when testing the Adam fix, and keeping SGD when testing the ReLU fix). Use 10 epochs, or 6 if training takes too long.

9. [2pt] Report the test accuracy after training with (a) `Sigmoid` + Adam and (b) `ReLU` + SGD. Which performs better? Try multiple times and describe the variability.