INFR10086 Machine Learning (MLG) Semester 2, 2025/6

Tutorial 2: Digit Classification

In this tutorial, we will study linear classifiers for digit classification and a few practical considerations when implementing them.

1 Setting things up

We will need python for this tutorial, in particular, the two python packages, `numpy` and `matplotlib`. There are a few optional `bash` commands. These should all be installed if you are on Linux, or any DICE machine.

Action. Download the tarball for this tutorial and untar it.

`https://homepages.inf.ed.ac.uk/htang2/mlg2025/tutorial-2.tar.gz`

If you are on Linux, you can run the following commands in a terminal.

```
$ wget https://homepages.inf.ed.ac.uk/htang2/mlg2025/tutorial-2.tar.gz
$ tar xf tutorial-2.tar.gz
$ cd tutorial-2
```

2 The MNIST data set

MNIST is a data set that comes with hand-written digits in the form of 28×28 matrices and their respective labels, i.e., $0, 1, \ldots, 9$.

Action. Visualize a few digits in the data set by running the following commands.

```
$ mkdir exp0
$ cd exp0
$ ../src/plot-digit.py 0
$ ../src/plot-digit.py 1
$ ../src/plot-digit.py 2
```

Discussion.

- What are the numbers printed out when we run the above commands?
- The above commands also save 3 images, `digit-1.png`, `digit-2.png`, and `digit-3.png`. What are in the images?

We will do what is called **standardization** on the data set. We perform

$$x \leftarrow \frac{x - \mu}{\sigma} \tag{1}$$

for every data point x, where μ is the global mean and σ^2 is the global variance.

Action. Visualize the mean of all digits in the data set by running the following command.

```
$ cd exp0
$ ../src/plot-mean.py
```

Discussion. The above command saves an image, `global-mean.png`. What does the global mean look like and what does it tell us?

3 The implementation of a linear digit classifier

In this section, we will study how to implement stochastic gradient descent for training a linear classifier for digit classification. In particular, we will look at the simplest case where the size of mini-batch is 1. In other words, we use a single sample to estimate the gradient.

Recall that stochastic gradient descent iteratively updates the parameter vector θ using the equation

$$\theta_{t+1} = \theta_t - \eta_t \nabla L(\theta_t). \tag{2}$$

We need an initial parameter vector θ_0 to start the process.

Action. The following command produces a random weight matrix `weight-0.npy` and a random bias vector `bias-0.npy`.

```
$ mkdir exp1
$ cd exp1
$ ../src/init-weight-bias.py weight-0.npy bias-0.npy
```

We haven't really trained anything, but the weight matrix and the bias vector should constitute a valid classifier. Since we have classes from 0, 1, ..., 9, this is a multiclass classification. As a reminder, a multiclass linear classifier can be written as

$$\operatorname*{argmax}_{y=1,\ldots,K} w_y^\top x + b_y, \tag{3}$$

where w_y is the weight vector, b_y is the bias term for the class y, and K is the number of classes (in this case, 10). It is more convenient to stack the weight vectors and bias terms, computing all the values in one go

$$\begin{bmatrix} w_1^\top \\ w_2^\top \\ \vdots \\ w_K^\top \end{bmatrix} x + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}. \tag{4}$$

We can then rewrite the multiclass linear classifier as

$$\operatorname*{argmax}_{y=1,\dots,K} s_y \tag{5}$$

where $s = Wx + b$, $W = \begin{bmatrix} w_1 & w_2 & \cdots & w_K \end{bmatrix}^\top$, and $b = \begin{bmatrix} b_1 & b_2 & \cdots & b_K \end{bmatrix}^\top$. Sometimes the vector s is called a score vector, and s_y, i.e., the y-th coordinate, is the score of the class y.

Action. Evaluate the random classifier by running the following command.

```
$ ../src/eval.py weight-0.npy bias-0.npy
```

We can see that the script `eval.py` prints the misclassification error rate and the averaged log loss of our random classifier.

Discussion. Now open `../src/eval.py`. Which line in `eval.py` computes $Wx + b$ and makes a prediction?

Action. Launch `python` and run the following commands.

```
>>> import numpy
>>> W = numpy.load('weight-0.npy')
>>> W.shape
>>> b = numpy.load('bias-0.npy')
>>> b.shape
```

Discussion. What are the shapes of weight matrix `W` and the bias vector `b`? Why are they in these shapes? In particular, why 784, why 10, and why 784×10?

We are going to use stochastic gradient descent to minimize log loss. Recall that the log loss, defined on a single sample (x, y), for multiclass classification is

$$\ell = -(w_y^\top x + b_y) + \log \sum_{y'=1}^{K} \exp\left(w_{y'}^\top x + b_{y'}\right). \tag{6}$$

Again, this can be more conveniently written as

$$\ell = -s_y + \log \sum_{y'=1}^{K} \exp(s_{y'}), \tag{7}$$

where $s = Wx + b$.

Discussion. In `../src/eval.py`, we can see that the following block of code computes the log loss.

```
def log_loss(W, b, x, y):
    s = x @ W + b
    m = numpy.max(s)
    logZ = m + numpy.log(numpy.sum(numpy.exp(s - m)))
    return -s[y] + logZ
```

What term does `logZ` corresponds to in equation (7)? What is the purpose of computing `m`?

Discussion. In `eval.py`, we see that the loss is averaged using the following block of code

```
loss = log_loss(weight, bias, img, label)
avg_loss = (1.0 / (count + 1.0) * loss + count / (count + 1.0) * avg_loss)
count += 1
```

Explain how the above block computes the average log loss on the entire data set.

The script `train.py` implements stochastic gradient descent with mini-batch of size 1. The parameters include the weight matrix W and the bias vector b. In other words, the gradient updates are

$$W_{t+1} = W_t - \eta_t \nabla_W \ell(W_t; x_t, y_t) \tag{8}$$
$$b_{t+1} = b_t - \eta_t \nabla_b \ell(b_t; x_t, y_t) \tag{9}$$

In particular, the gradient with respect to s_i for some class i can be derived as

$$\nabla_{s_i} \ell = -\mathbb{1}_{y=i} + \frac{\exp(s_i)}{\sum_{y'=1}^{K} \exp(s_{y'})}. \tag{10}$$

Discussion. In `../src/train.py`, the following block of code computes the gradient.

```
def grad_log_loss(W, b, x, y):
    s = score(W, b, x)
    m = numpy.max(s)
    logZ = m + numpy.log(numpy.sum(numpy.exp(s - m)))
    prob = numpy.exp(s - logZ)
    prob[y] -= 1.0
    return (numpy.outer(x, prob), prob)
```

Explain how the above code computes the gradient $\nabla_W \ell$ and $\nabla_b \ell$.

4 Training a linear digit classifier

We are finally ready to experience the training of a multiclass linear classifier. Recall that an optimization algorithm reaches a satisfactory result when

$$L(\theta_t) - L(\theta^*) < \epsilon, \tag{11}$$

where t is the number of gradient updates and ϵ is the desired error.

Action. Run the following commands to train the classifier for 3 epochs.

```
$ ../src/train.py weight-0.npy bias-0.npy 1
$ ../src/train.py weight-1.npy bias-1.npy 2
$ ../src/train.py weight-2.npy bias-2.npy 3
```

If you are more skilled in bash, you can run the following in bash to train a total 20 epochs.

```
for i in {1..20}; do
    ../src/train.py weight-$((i-1)).npy bias-$((i-1)).npy $i;
done
```

Discussion. In `../src/train.py`, where is the step size (or learning rate) specified?

Discussion. How many updates do we have in an epoch?

Discussion. How do we know if we need more epochs?

Discussion. In `train.py`, which line shuffles the order of the data set? Why shuffling?

After training, we can evaluate our trained classifiers using `eval.py`.

Action. Run the following commands to evaluate the classifiers we get from the first 3 epochs.

```
$ ../src/eval.py weight-0.npy bias-0.npy
$ ../src/eval.py weight-1.npy bias-1.npy
$ ../src/eval.py weight-2.npy bias-2.npy
$ ../src/eval.py weight-3.npy bias-3.npy
```

If you are more skilled in bash, you can run the following in bash to evaluate all 20 epochs.

```
for i in {0..20}; do
    ../src/eval.py weight-$i.npy bias-$i.npy;
done
```

Discussion. The script `eval.py` prints both the misclassification error rate and the averaged log loss. Which one should we look at during training?

Discussion. Based on the 20 epochs of results, does stochastic gradient descent always reduce the log loss after every epoch?

Recall that we standardize the data before training.

Discussion. Which line in `eval.py` standardizes the data?

From `eval.py`, we see that the data is standardized before making a prediction. If there is a good linear classifier W and b on the standardized data set, the linear classifier $\tilde{W}$ and $\tilde{b}$ on the original data set can be derived as

$$W\left(\frac{x-\mu}{\sigma}\right)+b=\frac{W}{\sigma}x+b-\frac{W\mu}{\sigma}=\tilde{W}x+\tilde{b} \tag{12}$$

where $\tilde{W}=W/\sigma$ and $\tilde{b}=b-W\mu/\sigma$.

Discussion. Given the above argument, we can find $\tilde{W}$ and $\tilde{b}$ directly on the original data set. If finding $\tilde{W}$ and $\tilde{b}$ on the original data set is equivalent to finding W and b on the standardized data set, what is the point of standardization?