FMAN45 Machine Learning, spring 2022
Assignment 3

Instructions *Read this assignment description carefully. Start by downloading the provided data and skeleton code, then complete the tasks and typeset your solutions in LaTeX.*

Report

- *Summarize your solutions and results concisely in a detailed report.*
- *All solutions, plots and figures should be in* ***one*** *self-contained pdf-file.*
- *It should thus be possible to understand all material presented in the report without running any code.*

Submission

- *You should submit the following via Canvas before the deadline:*
  - *one pdf-document, containing your report; and*
  - *one zip-archive, containing the data you've created and all the code.*
- *Do* ***not*** *include the original data in your archive.*
- *Do* ***not*** *include the pdf-document in the zip-archive.*

Also note that there are strict rules about collaboration and plagiarism, see information on assignment page.

1 Common Layers and Backpropagation

In this section you will implement forward and backward steps for a few common neural network layers.

Recall that the loss L is a function of the network input, the ground truth value and the network parameters. When you train a neural net you have to evaluate the gradient with respect to all the parameters in the network. The algorithm to do this is called backpropagation.

1.1 Dense Layer

Introduction. *Let a fixed, given layer in the network $f = \sigma \circ \ell : \mathbb{R}^m \to \mathbb{R}^n$ consist of a linear part $\ell : \mathbb{R}^m \to \mathbb{R}^n$ and an activation function $\sigma : \mathbb{R}^n \to \mathbb{R}^n$. We represent the linear part ℓ by a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$. Let $\mathbf{x} \in \mathbb{R}^m$ be a data point that enters the given layer and let $\mathbf{y} = \ell(\mathbf{x}) \in \mathbb{R}^n$*

contain the values of the data point after having passed through the linear part of the layer. Then for all i,

$$y_i = \sum_{j=1}^{m} W_{ij} x_j + b_i. \tag{1}$$

Using matrix notation we have $\mathbf{y} = \mathbf{W}\mathbf{x}+\mathbf{b}$. Note that $\mathbf{W}$ and $\mathbf{b}$ are parameters that are trainable. Several such mappings, or layers*, are concatenated to form a full network.*

Computing the gradients. *To derive the equations for backpropagation we want to express $\frac{\partial L}{\partial \mathbf{x}}$, as an expression containing the gradient $\frac{\partial L}{\partial \mathbf{y}}$ with respect to the pre-activation value $\mathbf{y}$. Using the chain rule we get*

$$\frac{\partial L}{\partial x_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial x_i}. \tag{2}$$

We also need to compute the gradient with respect to the parameters $\mathbf{W}$ and $\mathbf{b}$. To do this, again use the chain rule,

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial W_{ij}} \tag{3}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{n} \frac{\partial L}{\partial y_l} \frac{\partial y_l}{\partial b_i} \tag{4}$$

> ***Exercise 1 (10 points):*** *Derive expressions for $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{b}}$ in terms of $\frac{\partial L}{\partial \mathbf{y}}, \mathbf{W}$ and $\mathbf{x}$. Include a full derivation of your results. The answers should all be given as matrix expressions without any explicit sums.*

Checking the gradients. *When you have code that uses gradients, it is very important to check that the gradients are correct. A bad way to conclude that the gradient is correct is to manually look at the code and convince yourself that it is correct or just to see if the function seems to decrease when you run optimization; it might still be a descent direction even if it is not the gradient. A much better way is to check finite differences using the formula*

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x} - \epsilon \mathbf{e}_i)}{2\epsilon}, \tag{5}$$

where $\mathbf{e}_i$ is a vector that is all zero except for position i where it is 1, and ϵ is a small number. In the code there is a file `tests/test_fully_connected.m` where you can test your implementation.

Computing the gradients of batches. *When you are training a neural network it is common to evaluate the network and compute gradients not with respect to just one element but N elements in a batch. We use superscripts to denote the elements in the batch. For the dense layer above, we have multiple inputs $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(N)}$ and we wish to compute $\mathbf{y}^{(1)} = \mathbf{W}\mathbf{x}^{(1)} + \mathbf{b}$, $\mathbf{y}^{(2)} = \mathbf{W}\mathbf{x}^{(2)} + \mathbf{b}$, $\cdots$, $\mathbf{y}^{(N)} = \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b}$. In the code for the forward pass you can see that the input array first is reshaped to a matrix $\mathbf{X}$ where each column contains all values for a single batch, that is,*

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(N)} \end{pmatrix}. \tag{6}$$

For instance, if $\mathbf{x}^{(1)}$ an image of size $5 \times 5 \times 3$ it is reshaped to a long vector with length 75. We wish to compute

$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}^{(1)} & \mathbf{y}^{(2)} & \dots & \mathbf{y}^{(N)} \end{pmatrix} = \begin{pmatrix} \mathbf{W}\mathbf{x}^{(1)} + \mathbf{b} & \mathbf{W}\mathbf{x}^{(2)} + \mathbf{b} & \dots & \mathbf{W}\mathbf{x}^{(N)} + \mathbf{b} \end{pmatrix}. \tag{7}$$

When we are backpropagating to $\mathbf{X}$ we have to compute

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{x}^{(1)}} & \frac{\partial L}{\partial \mathbf{x}^{(2)}} \cdots \frac{\partial L}{\partial \mathbf{x}^{(N)}} \end{pmatrix} \tag{8}$$

using

$$\frac{\partial L}{\partial \mathbf{Y}} = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{y}^{(1)}} & \frac{\partial L}{\partial \mathbf{y}^{(2)}} \cdots \frac{\partial L}{\partial \mathbf{y}^{(N)}} \end{pmatrix}. \tag{9}$$

Use your expression obtained in Exercise 1 and simplify to matrix operations. For the parameters, we have that both $\mathbf{W}$ and $\mathbf{b}$ influence all elements $\mathbf{y}^{(i)}$, so for the parameters we compute $\frac{\partial L}{\partial \mathbf{W}}$ using the chain rule with respect to each element in each $\mathbf{y}^{(i)}$ and just sum them up. More formally,

$$\frac{\partial L}{\partial W_{ij}} = \sum_{l=1}^{N} \sum_{k=1}^{n} \frac{\partial L}{\partial y_k^{(l)}} \frac{\partial y_k^{(l)}}{\partial W_{ij}} \tag{10}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{N} \sum_{j=1}^{n} \frac{\partial L}{\partial y_j^{(l)}} \frac{\partial y_j^{(l)}}{\partial b_i}. \tag{11}$$

Note that the inner sum is the same as you have computed in the previous exercise, so you just sum the expression you obtained in the previous exercise over all elements in the batch. You can implement the forward and backward passes with for-loops, but it is also possible to use matrix operations to vectorize the code and for full credit you should vectorize it. Potentially useful functions in Matlab include `bsxfun`, `sub2ind`, `ind2sub`, `reshape` *and* `repmat`.

Using for-loops in programming can slow down your programm significantly.

This is because in a for-loop, the data is being processed sequentially and not in parallel. Our aim is to parallelise the tasks in backpropagation by vectorising our Matlab commands and to avoid a for-loop running over all the data points.

Exercise 2 (10 points): *Derive expressions for* $\mathbf{Y}$, $\frac{\partial L}{\partial \mathbf{X}}$, $\frac{\partial L}{\partial \mathbf{W}}$ *and* $\frac{\partial L}{\partial \mathbf{b}}$ *in terms of* $\frac{\partial L}{\partial \mathbf{Y}}, \mathbf{W}, \mathbf{X}$ *and* $\mathbf{b}$. *Include a full derivation of your results. Also add Matlab code that implements these vectorised expressions that you have just derived. In the code there are two files* `layers/fully_connected_forward.m` *and* `layers/fully_connected_backward.m`. *Implement these functions and check that your implementation is correct by running* `tests/test_fully_connected.m`. *For full credit your code should be vectorized over the batch. Include the relevant code in the report.*

1.2 ReLU

The most commonly used function as a nonlinearity is the rectified linear unit (ReLU). It is defined by

$$ReLU : \mathbb{R} \to \mathbb{R}; \quad x_i \mapsto y_i := \max(x_i, 0). \tag{12}$$

Exercise 3 (10 points): *Derive the backpropagation expression for* $\frac{\partial L}{\partial x_i}$ *in terms of* $\frac{\partial L}{\partial y_i}$. *Include a full derivation of your results. Implement the layer in* `layers/relu_forward.m` *and* `layers/relu_backward.m`. *Test it with* `tests/test_relu.m`. *For full credit you must use built in indexing and not use any for-loops. Include the relevant code in the report.*

1.3 Softmax Loss

Suppose we have a vector $\mathbf{x} = [x_1, \ldots, x_m]^\top \in \mathbb{R}^m$. *The goal is to classify the input as one out of* m *classes and* x_i *is a score for class* i *and a larger* x_i *is a higher score. By using the softmax function we can interpret the scores* x_i *as probabilities. Let*

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} \tag{13}$$

be the probability for class i. *Now suppose that the ground truth class is* c *and denote the natural logarithm with base* e *by* $\log$. *Since now* y_i *are probabilities we can define the loss* L *to be minimized as the negative log likelihood,*

$$L(\mathbf{x}, c) = -\log(y_c) = -\log\left(\frac{e^{x_c}}{\sum_{j=1}^{m} e^{x_j}}\right) = -x_c + \log\left(\sum_{j=1}^{m} e^{x_j}\right). \tag{14}$$

Exercise 4 (10 points): *Compute the expression for* $\frac{\partial L}{\partial x_i}$ *in terms of* y_i*. Include a full derivation of your results. Note that this is the final layer, so there are no gradients to backpropagate. Implement the layer in* `layers/softmaxloss_forward.m` *and* `layers/softmaxloss_backward.m`*. Test it with* `tests/test_softmaxloss.m`*. Make sure that* `tests/test_gradient_whole_net.m` *runs correctly when you have implemented all layers. For full credit you should use built in functions for summation and indexing and not use any for-loops. Include the relevant code in the report.*

2 Training a Neural Network

The function we are trying to minimize when we are training a neural net is

$$L(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, z^{(i)}; \mathbf{w}) \tag{15}$$

where $\mathbf{w}$ *are all the parameters of the network,* $\mathbf{x}^{(i)}$ *is the input and* $z^{(i)}$ *the corresponding ground truth and* $L(\mathbf{x}^{(i)}, z^{(i)}; \mathbf{w})$ *is the loss for a single example, given for instance by a neural net with a softmax loss in the last layer. In practice,* N *is very large and we never evaluate the gradient over the entire sum. Instead we evaluate it over a batch with* $n \ll N$ *elements because the network can be trained much faster if you use batches and update the parameters in every step.*

If you are using gradient descent to train the network, the way the parameters are updated is

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \frac{\partial L}{\partial \mathbf{w}} \tag{16}$$

where α *is a hyperparameter called the learning rate. Since we only evaluate the gradient over a few examples, the estimated gradient in (16) might be very noisy. The idea behind gradient descent with momentum is to average the gradient estimations over time and use the smoothed gradient to update the parameters. The update in (16) is modified as*

$$\mathbf{m}_n = \mu \mathbf{m}_{n-1} + (1-\mu)\frac{\partial L}{\partial \mathbf{w}} \tag{17}$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \mathbf{m}_n \tag{18}$$

where $\mathbf{m}_n$ *is a moving average of the gradient estimations and* μ *is a hyperparameter in the range* $0 < \mu < 1$ *controlling the smoothness.*

Exercise 5 (10 points): *Implement gradient descent with momentum in* `training.m`*. Remember to include weight decay. Include the relevant code in the report.*

3 Classifying Handwritten Digits

You must have solved all previous problems before you can start working on this and the next problem.
In `mnist_starter.m` *there is a simple baseline for the MNIST dataset. Read the comments in the code carefully. It reaches about 98 % accuracy on the test set (this of course varies a bit from time to time). Validate this to make sure that the code you have written so far is correct.*

Exercise 6 (25 points): *Plot the kernels the first convolutional layer learns. Plot a few images that are misclassified. Plot the confusion matrix for the predictions on the test set and compute the precision and the recall for all digits. Write down the number of parameters for all layers in the network. Write comments about all plots and figures.*

4 Classifying Tiny Images

In `cifar10_starter.m` *there is a simple network that can be trained on the CIFAR10 dataset. This baseline give an accuracy of about 48 % after training for 5000 iterations. Note that it is much harder to get good classification results on this dataset than MNIST, mainly due to significant intraclass variation.*

Exercise 7 (25 points): *Do whatever you want to improve the accuracy of the baseline network. Write what you have tried in the report, even experiments that did not work out. For your final model, compute and report all plots and numbers that were required in the previous exercise and comment on it.*