Deep Learning — Assignment 3

Third assignment for the 2024 Deep Learning course (NWI-IMC070) of the Radboud University.

Names:

Group:

Instructions:

- Fill in your names and the name of your group.
- Answer the questions and complete the code where necessary.
- Keep your answers brief, one or two sentences is usually enough.
- Re-run the whole notebook before you submit your work.
- Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file.
- The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Objectives

This assignment is a continuation of assignment 2. We will work on the same dataset with a similar network architecture. In this assignment you will

1. Experiment with weight decay
2. Experiment with dropout
3. Experiment with early stopping

In [ ]:
```
%config InlineBackend.figure_formats = ['png']
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
import time
import torchvision
import tqdm.notebook as tqdm
import collections
import IPython
import pandas as pd

plt.style.use('ggplot')
```

```
# Fix the seed, so outputs are exactly reproducible
torch.manual_seed(12345);

# Use the GPU if available
def detect_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")
device = detect_device()
```

3.1 FashionMNIST again

We will work with the same code as last week, with some slight modifications. So, we will again do experiments with the Fashion-MNIST dataset.

In [ ]:
```
fashionmnist = torchvision.datasets.FashionMNIST(
    root=".", download=True,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        lambda img: img.flatten()
    ]))

# use 1000 samples for training, 500 for validation, ignore the rest
fashion_train, fashion_validation, _ = torch.utils.data.random_split(
    fashionmnist, [1000, 500, len(fashionmnist) - (1000 + 500)])
```

The neural network is the same as in assignment 2. We include our implementation of `build_net` here. You may substitute your own implementation instead, if you prefer.

In [ ]:
```
def build_net():
    return torch.nn.Sequential(
        torch.nn.Linear(784, 128),
        torch.nn.ReLU(),
        torch.nn.Linear(128, 64),
        torch.nn.ReLU(),
        torch.nn.Linear(64, 10)
    )
```

The training code is also almost the same as last time, except that we add an extra parameter to `fit`.

In [ ]:
```
def fit(net, train, validation, optimizer, epochs=25, batch_size=10, after_epoch=None, device=d
    """
    Train and evaluate a network.
     - net:              the network to optimize
     - train, validation: the training and validation sets
```

```
    - optimizer:        the optimizer (such as torch.optim.SGD())
    - epochs:           the number of epochs to train
    - batch_size:       the batch size
    - after_epoch:      optional function to call after every epoch
    - device:           whether to use a gpu ('cuda') or the cpu ('cpu')

    Returns a dictionary of training and validation statistics.
    """

    # move the network parameters to the gpu, if necessary
    net = net.to(device)

    # initialize the loss and accuracy history
    history = collections.defaultdict(list)
    epoch_stats, phase = None, None

    # initialize the data loaders
    data_loader = {
        'train':      torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=True),
        'validation': torch.utils.data.DataLoader(validation, batch_size=batch_size)
    }

    # measure the length of the experiment
    start_time = time.time()

    # some advanced PyTorch to look inside the network and log the outputs
    # you don't normally need this, but we use it here for our analysis
    def register_measure_hook(idx, module):
        def hook(module, input, output):
            with torch.no_grad():
                # store the mean output values
                epoch_stats['%s %d: %s output mean' % (phase, idx, type(module).__name__)] += \
                    output.mean().detach().cpu().numpy()
                # store the mean absolute output values
                epoch_stats['%s %d: %s output abs mean' % (phase, idx, type(module).__name__)] +
                    output.abs().mean().detach().cpu().numpy()
                # store the std of the output values
                epoch_stats['%s %d: %s output std' % (phase, idx, type(module).__name__)] += \
                    output.std().detach().cpu().numpy()
        module.register_forward_hook(hook)

    # store the output for all layers in the network
    for layer_idx, layer in enumerate(net):
        register_measure_hook(layer_idx, layer)
    # end of the advanced PyTorch code

    for epoch in tqdm.tqdm(range(epochs), desc='Epoch', leave=False):
        # initialize the loss and accuracy for this epoch
        epoch_stats = collections.defaultdict(float)
        epoch_stats['train steps'] = 0
        epoch_stats['validation steps'] = 0
        epoch_outputs = {'train': [], 'validation': []}
```

```
    # first train on training data, then evaluate on the validation data
    for phase in ('train', 'validation'):
        # switch between train and validation settings
        net.train(phase == 'train')

        epoch_steps = 0
        epoch_loss = 0
        epoch_accuracy = 0

        # loop over all minibatches
        for x, y in data_loader[phase]:
            # move data to gpu, if necessary
            x = x.to(device)
            y = y.to(device)

            # compute the forward pass through the network
            pred_y = net(x)

            # compute the current loss and accuracy
            loss = torch.nn.functional.cross_entropy(pred_y, y)
            pred_class = torch.argmax(pred_y, dim=1)
            accuracy = torch.mean((pred_class == y).float())

            # add to epoch loss and accuracy
            epoch_stats[f'{phase} loss'] += loss.detach().cpu().item()
            epoch_stats[f'{phase} accuracy'] += accuracy.detach().cpu().item()

            # store outputs for later analysis
            epoch_outputs[phase].append(pred_y.detach().cpu().numpy())

            # only update the network in the training phase
            if phase == 'train':
                # set gradients to zero
                optimizer.zero_grad()

                # backpropagate the gradient through the network
                loss.backward()

                # track the gradient and weight of the first layer
                # (not standard; we only need this for the assignment)
                epoch_stats['train mean abs grad'] += \
                    torch.mean(torch.abs(net[0].weight.grad)).detach().cpu().item()
                epoch_stats['train mean abs weight'] += \
                    torch.mean(torch.abs(net[0].weight)).detach().cpu().item()

                # update the weights
                optimizer.step()

            epoch_stats[f'{phase} steps'] += 1

        # compute the mean loss and accuracy over all minibatches
        for key in epoch_stats:
            if phase in key and not 'steps' in key:
```

```
                epoch_stats[key] = epoch_stats[key] / epoch_stats[f'{phase} steps']
                history[key].append(epoch_stats[key])

            # count the number of update steps
            history[f'{phase} steps'].append((epoch + 1) * epoch_stats[f'{phase} steps'])

            # store the outputs
            history[f'{phase} outputs'].append(np.concatenate(epoch_outputs[phase]).flatten())

        history['epochs'].append(epoch)
        history['time'].append(time.time() - start_time)

        # call the after_epoch function
        if after_epoch is not None:
            stop = after_epoch(net, epoch, epoch_stats)
            if stop is Stop:
                break

    return history

# marker to indicate stopping
Stop = "stop"
```

In [ ]:

```
# helper code to plot our results
class HistoryPlotter:
    def __init__(self, plots, table, rows, cols, param_names=[]):
        self.plots = plots
        self.table = table
        self.rows = rows
        self.cols = cols
        self.histories = {}
        self.results = []
        self.params = []
        self.param_names = []

        self.fig, self.axs = plt.subplots(ncols=cols * len(plots), nrows=rows,
                                          sharex='col', sharey='none',
                                          figsize=(3.5 * cols * len(plots), 3 * rows))
        plt.tight_layout()
        IPython.display.display(self.fig)
        IPython.display.clear_output(wait=True)

    # add the results of an experiment to the plot
    def add(self, title, history, row, col, epoch=-1, param=None):
        self.histories[title] = history
        self.results.append((title, {key: history[key][epoch] for key in self.table}))
        self.params.append(param)

        for plot_idx, plot_xy in enumerate(self.plots):
            ax = self.axs[row, col * len(self.plots) + plot_idx]
            for key in plot_xy['y']:
                lines = ax.plot(history[plot_xy['x']], history[key], label=key)
                if epoch >= 0:
```

```
                ax.plot([history[plot_xy['x']][epoch]], [history[key][epoch]], marker='*', color='black')
            if 'accuracy' in plot_xy['y'][0]:
                ax.set_ylim([0, 1.01])
            ax.legend()
            ax.set_xlabel(plot_xy['x'])
            ax.set_title(title)
        plt.tight_layout()
        IPython.display.clear_output(wait=True)
        IPython.display.display(self.fig)

    # print a table of the results for all experiments
    def print_table(self):
        df = pd.DataFrame([
            { 'experiment': title, **{key: row[key] for key in self.table} }
            for title, row in self.results
        ])
        IPython.display.display(df)

    def done(self):
        plt.close()
        self.print_table()
```

3.2 Weight decay (6 points)

The training can be regularized using weight decay. This option is built-in in many of the PyTorch optimizers (documentation).

We will set up an experiment to investigate how this affects the training of the model.

We use the good settings from last week:

- Optimizer: Adam
- Learning rate: 0.0001
- Minibatch size: 32
- 150 epochs

and apply L2 weight decay with a factor 0, 0.0001, 0.001, 0.01, or 0.1.

(a) Complete the code below and run the experiment. **(1 point)**

In [ ]:
```
plotter_weight_decay = \
    HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation loss']},
                          {'x': 'epochs', 'y': ['train accuracy', 'validation accuracy']},
                          {'x': 'epochs', 'y': ['train mean abs weight']},],
                   table=['train accuracy', 'validation accuracy', 'train mean abs weight'],
                   param_names=['weight_decay'],
                   rows=5, cols=1)

epochs = 150
lr = 0.0001
```

```
batch_size = 32
weight_decays = [0, 0.0001, 0.001, 0.01, 0.1]

for row, weight_decay in enumerate(weight_decays):
    net = build_net()
    # TODO: Set up optimizer with the given weight_decay
    history_weight_decay = fit(net, fashion_train, fashion_validation, optimizer=optimizer, epochs=
    plotter_weight_decay.add(f'weight_decay={weight_decay}', history_weight_decay, row=row, c

plotter_weight_decay.done()
```

(b) How can you observe the amount of overfitting in the plots? **(1 point)**

TODO Your answer here.

(c) How does weight decay affect the performance of the model in the above experiments? Give an explanation in terms of the amount of overfitting. **(1 point)**

TODO Your answer here.

In these experiments you have implemented weight decay using the built in weight decay feature of the optimizer.

An alternative is to add an L2 penaltiy to the loss:

$$L_{\text{regularized}}(\theta) = L(\theta) + \lambda \frac{1}{2} |\theta|_2^2,$$

which results in a gradient

$$\nabla_\theta L_{\text{regularized}}(\theta) = \nabla_\theta L(\theta) + \lambda\theta.$$

Using gradient descend will then decrease the weights.

(d) Are the two ways of implementing weight decay equivalent when using the Adam optimizer? **(1 point)**

Hint: look at the torch documentation for Adam.

TODO Your answer here.

Another way to implement weight decay, and where the name comes from, is to scale or decay the weights directly:

$$\theta \leftarrow (1 - \lambda)\theta$$

Combined with Adam, this gives the `AdamW` optimizer.

(e) Are Adam and AdamW equivalent? Explain your answer. **(1 point)**

TODO Your answer here.

Learning curves

So far the only learning curves we have looked at have the number of epochs on the horizontal axis. We can also make learning curves putting another parameter on that axis.

(f) Run the code below to plot a learning curve **(no points)**

In [ ]:
```
weight_decays = torch.tensor(plotter_weight_decay.params) + 1e-7
  # Note: add 1e-7 to prevent log(0)

fig, axs = plt.subplots(ncols=2, nrows=1, figsize=(4.5 * 2, 3))
for i, stat in enumerate(['loss', 'accuracy']):
    keys = [f'train {stat}', f'validation {stat}']
    values = {key: [r[1][key][-1] for r in plotter_weight_decay.histories.items()] for key in keys}

    ax = axs[i]
    for key in keys:
        ax.plot(weight_decays, values[key], '.-', label=key)
    ax.set_xscale('log');
    ax.set_xlabel('weight decay');
    ax.set_ylabel(stat);
    ax.legend();
```

(g) Looking a this learning curve, how do you see signs of overfitting or underfitting? **(1 point)**

TODO Your answer here.

3.3 Dropout (9 points)

Next, we will do experiments with dropout. This gives another way of regularizing the training.

(a) Make a copy of the network architecture below, and add dropout. **(1 point)**

Add dropout layers after each linear layer, except for the last.

Hint: see torch.nn.Dropout.

In [ ]:
```
def build_net_with_dropout(p):
    # TODO: network with dropout layers, with dropout probability p
```

(b) Should you put dropout layers before or after ReLU activation functions? Does it matter? **(1 point)**

TODO Your answer here.

(c) What would happen if you put a dropout layer after the last linear layer? (1 point)

TODO Your answer here.

(d) Set up an experiment to see how dropout affects our results. (1 point)

In [ ]:
```
plotter_dropout = \
    HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation loss']},
                          {'x': 'epochs', 'y': ['train accuracy', 'validation accuracy']},],
                   table=['train accuracy', 'validation accuracy', 'train mean abs weight'],
                   param_names=['dropout'],
                   rows=3, cols=2)

epochs = 150
lr = 0.0001
batch_size = 32
dropouts = [0, 0.1, 0.3, 0.5, 0.7, 0.9]

for row, dropout in enumerate(dropouts):
    # TODO: Set up a network with the right dropout, and an optimizer
    history_dropout = fit(net, fashion_train, fashion_validation, optimizer=optimizer, epochs=epoc
    plotter_dropout.add(f'dropout={dropout}', history_dropout, row=row//2, col=row%2, param=dr

plotter_dropout.done()
```

(e) How does dropout affect the results? (1 point)

TODO Your answer here.

(f) How does dropout affect training speed? Has the training converged in all runs? (1 point)

TODO Your answer here.

(g) With a large amount of dropout, the training loss can be worse than the validation loss. How is this possible? (1 point)

TODO Your answer here.

(h) Plot a learning curve for the dropout parameter (1 point)

You should use a linear scale for the x-axis. Be sure to use the right histories, and to label your axes.

In [ ]:
```
# TODO: Plot a learning curve
```

(i) Does the above learning curve show a clear optimum? (1 point)

TODO Your answer here.

3.4 Early stopping (7 points)

If you look at the learning curves of the unregularised models, you can see that the validation loss starts to go up after a certain amount of training. It would be good to stop at that point, which is called early stopping.

There are two ways to implement early stopping:

1. Run the training for a fixed number of epochs, but keep track of the best result on the validation set.
2. Run until the validation loss does not decrease for a certain number of epochs.

Only the second option is actually early *stopping*, but the first option can be easier to implement.

(a) Implement the first style of early stopping, and run the experiment below. (2 points)

You can pass a function to the `after_epoch` parameter of `fit`. This function is called after every epoch.

The `epoch` parameter to `plotter.add` highlights a specific epoch in the results with a star, and selects it for the table.

In [ ]:
```
plotter_early_stop = \
    HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation loss']},
                          {'x': 'epochs', 'y': ['train accuracy', 'validation accuracy']}],
                   table=['train loss', 'validation loss', 'train accuracy', 'validation accuracy', 'epochs'],
                   rows=4, cols=1)

epochs = 150
lr = 0.001
lrs = [0.1, 0.01, 0.001, 0.0001]
batch_size = 32

for row, lr in enumerate(lrs):
    # the best network, epoch at which we found it, and stats
    best_net = None
    best_epoch = 0
    best_stats = {'train loss': torch.inf, 'validation loss': torch.inf, 'train accuracy': 0, 'validation accu

    def track_best(net, epoch, epoch_stats):
        global best_net, best_epoch, best_stats
        # TODO: keep track of the best model

    net = build_net()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history_early_stop = fit(net, fashion_train, fashion_validation, optimizer=optimizer, epochs=ep
    plotter_early_stop.add(f'lr={lr}', history_early_stop, row=row, col=0, epoch=best_epoch)
```

```
plotter_early_stop.done()
```

(b) Looking at the results, does early stopping prevent overfitting? (1 point)

TODO: Your answer here.

(c) Is it fair to compare the validation loss you get with early stopping to the loss without early stopping? Briefly explain your answer. (1 point)

TODO Your answer here.

Copying a neural network with `net2 = net1` makes a shallow copy, that is, the two variables refer to the same network in memory.

(d) If you used `best_net = net` in `track_best`, would `best_net` contain the optimal early stopping parameters after training? If not, how could you get access to them? (1 point)

TODO Your answer here.

Actual early *stopping*

It is wasteful to keep training if we know that the loss is only getting worse. So we might as well stop at that point.

(e) Implement the second variant of early stopping: stop training if the validation loss does not decrease for 5 epochs. (1 point)

Hint: The `fit` function will stop the training if the `after_epoch` function returns `Stop`.

In [ ]:
```
plotter_early_stop2 = \
    HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation loss']},
                          {'x': 'epochs', 'y': ['train accuracy', 'validation accuracy']}],
                   table=['train loss', 'validation loss', 'train accuracy', 'validation accuracy', 'epochs'],
                   rows=4, cols=1)

epochs = 150
lr = 0.001
lrs = [0.1, 0.01, 0.001, 0.0001]
batch_size = 32

for row, lr in enumerate(lrs):
    best_net = None
    best_epoch = 0
    best_stats = {'train loss': torch.inf, 'validation loss': torch.inf, 'train accuracy': 0, 'validation accu

    def stop_if_no_loss_decrease(net, epoch, epoch_stats):
        global best_net, best_epoch, best_stats
```

```
        # TODO: return Stop if the loss does not go down for 5 epochs

    net = build_net()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history_early_stop2 = fit(net, fashion_train, fashion_validation, optimizer=optimizer, epochs=e
    plotter_early_stop2.add(f'lr={lr}', history_early_stop2, row=row, col=0)

plotter_early_stop2.done()
```

(f) With the second variant of early stopping, is the network after training the optimal one? **(1 point)**

TODO: Your answer here.

3.5 Hyperparameter optimization (3 points)

We have seen quite a few hyperparameters this week and last week. To pick the optimal parameters, one strategy would be to do what we have done, and run an experiment for each parameter individually.

(a) Look at the previous experiments, and pick the best hyperparameter values. **(1 point)**

Optionally: look at the experiments from assignment 2 and also pick the optimal network width.

Batch size: TODO
Optimizer: TODO
Learning rate: TODO
Dropout: TODO
Weight decay: TODO

(b) If you select the hyperparameters this way, will you get the best results? Explain your answer. **(1 point)**

TODO: Your answer here.

An alternative is to use a grid search, and try all possible combinations of hyperparameters.

(c) How many experiments would you need to do to explore all combinations of learning rate, weight decay, and dropout that we used in this assignment? **(1 point)**

TODO: Your answer here.

The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 25 points.

Version 9571342 / 2024-09-18