dl-assignment-7

November 1, 2023

1 Deep Learning — Assignment 7

Seventh assignment for the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Daan Brugmans, Maximilian Pohl

Group: 31

1.1 Introduction

For this assignment we are doing things a bit different. * Your task is to reproduce the paper The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks; Jonathan Frankle, Michael Carbin. * Try to follow the experimental settings in that paper, unless there is a reason to deviate. * If the paper is not clear on some details, make a reasonable choice yourself, and motivate that choice. * You will have 3 weeks to work on this assignment. * Be aware that this assignment will take more time than previous ones. It is ok if you do not completely finish it. * We will *not* be providing you with much code. You will have to implement many things yourself. * You may freely use code from earlier weeks, and from the d2l books. Please add a comment to reference the original source. * You may *not* use implementations of the paper you find online.

Tips and hints * It is allowed and recommended to use more than just this notebook. Make separate python files for a library of functions, and for training and analyses. * If you like working with jupyter notebooks: make a separate notebook for trying things out, and keep this one clean. * Use checkpoint files before and during training. * In the notebook only display and discuss these results. * You may add new cells to this notebook as needed. * While the task is to reproduce parts of a paper, the big picture is more important than the exact details. * It is allowed to discuss the assignment with other groups, but try not to spoil too much. * If you get stuck, contact the teachers via discord.

1.2 Required software

If you need to import any additional libraries, add them below.

[1]:
```
%config InlineBackend.figure_formats =['png']
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
%reload_ext autoreload
import torch
import torch.nn
import torch.nn.utils.prune
import torchvision
import matplotlib.pyplot as plt
from d2l import torch as d2l
from dl_assignment_7_common import *  # Your functions should go here if you
 ↪want to use them from scripts
import simple_training

device = d2l.try_gpu()
```

1.3 7.1 The paper (2 points)

(a) Read sections 1, 2, and 5 of the paper The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks; by Jonathan Frankle, Michael Carbin

We will refer to this as the "LTH paper" from now on, or just "the paper". To answer some later questions you will also need to look at other sections, and search through the appendices.

(b) In your own words, briefly explain the key message of the paper. (2 points)

Note: "briefly" means: a few sentences at most.

The key idea of this paper is that in fully connected and convolutional networks, there exist sub-networks that, when trained with fortunate initialization, achieve a comparable performance with a lot less parameters.

1.4 7.2 Models and datasets (9 points)

(a) What neural network architectures are used in the paper? (1 point)

Convolutional and fully connected neural networks. These are combined to Lenet, Conv-2, Conv-4, Conv-6, Resnet-18, and VGG-19.

To keep things simple, we will start with a simple architecture, corresponding to what the paper calls `Lenet`.

(b) Define a function that constructs a `Lenet` network using PyTorch. (2 points)

Hint: see Figure 2.

Note: the LTH paper is not entirely clear about this, but the convolution layers use `padding='same'`.

(c) Define a function that can construct a network given the architecture name.

To keep the code as generic as possible, we can make function.

We will do all our experiments with two datasets: MNIST and FashionMNIST

(d) Are these datasets also used in the papers? (1 point)

MNIST is used in the paper, but FashionMNIST is not used in the paper.

(e) Define a function that loads a dataset given the dataset name. (3 points)

Hint: Standard datasets such as MNIST and CIFAR10 are available in the torchvision library.

(f) Most of these datasets come with a predefined train/test split. Is this used in the LTH paper? If so, update the dataset loader to return a pair `(trainset, testset)`. (1 point)

Yes, the paper uses the prededined train/test splits.

(g) Does the LTH paper use a validation set? If so, update the dataset loader to return `(train_dataset, validation_dataset, test_dataset)`. (1 point)

Hint: `random_split`, and/or see assignment 2.

Yes, the paper uses ~8.3% of training data as validation data.

1.5 7.3 Training (12 points)

(a) What optimization algorithm is used in the paper? What values are used for the hyperparameters? (1.5 points)

If you are unable to find the values used for some of the hyperparameters, use reasonable default values.

For the LeNet architecture, the paper uses the Adam optimizer with a learning rate of 1.2e-3 (0.0012) and a batch size of 60 per iteration, with 50.000 iterations total.

(b) Implement an evaluation function, that evaluates a model on a validation or test set (passed as an argument). (2 points)

The function should return loss and accuracy.

Hints: the book defines a function for this that you may use (see assignment 3).

(c) Implement a training loop. (4 points)

Make sure that the network parameters are saved to a file before and during training.

Because you will be doing many experiments, it would be a shame to have to re-run them when you reload the notebook. A better solution is to save model checkpoints. See the tutorial on saving and loading model parameters for how to implement this in PyTorch.

(d) Change the training function so that it saves the model at the start and at the end of training. (1 point)

Hint: Saving a model requires a filename. Because you will be running many experiments, come up with a descriptive naming convention and/or directory structure. Example: `path = f"checkpoints/model-{arch}-{dataset}-{run}-{phase_of_the_moon}-{iteration}.pth"`.

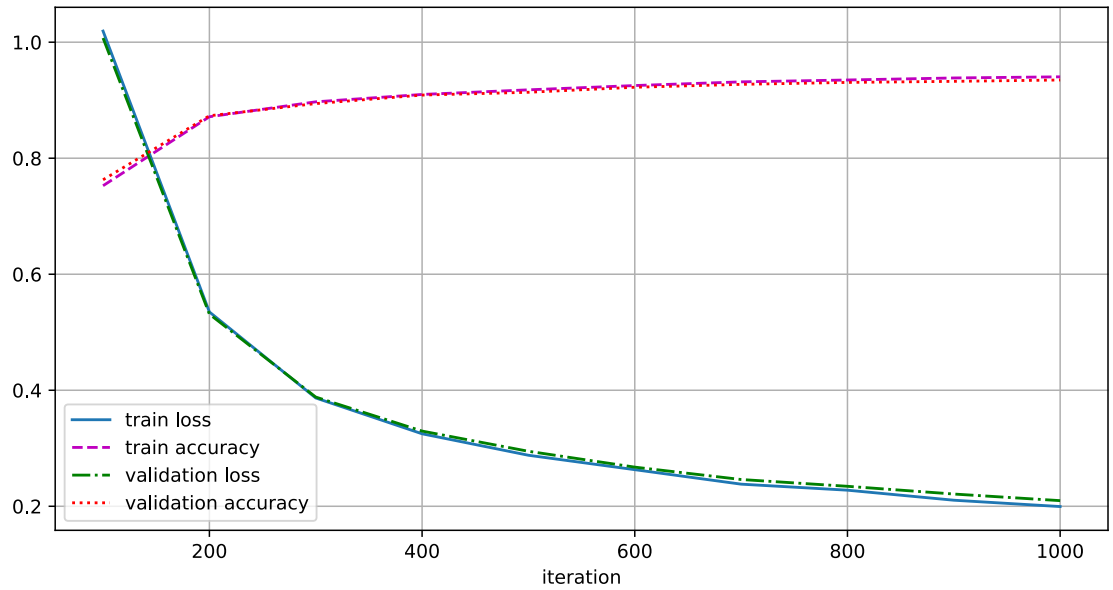
Hint 2: it is easier to save the whole model, see the bottom of the tutorial.

(e) Train a simple network on a simple dataset. (1 point)

You may want to create a new python script (`simple_training.py`), and just load the trained network here instead.

[6]:
```
torch.manual_seed(12345)
simple_training.train()
simple_model = simple_training.load('./checkpoints/
 ↪model-lenet-mnist-1-waning_gibbous-final.pth')
```

```
train loss 0.198, train accuracy 0.941, val loss 0.209, val accuracy 0.934, test
loss 0.198, test accuracy 0.941, min val loss at iteration 1000
```



(f) Does the training converge? How well does your network perform? (1 point)

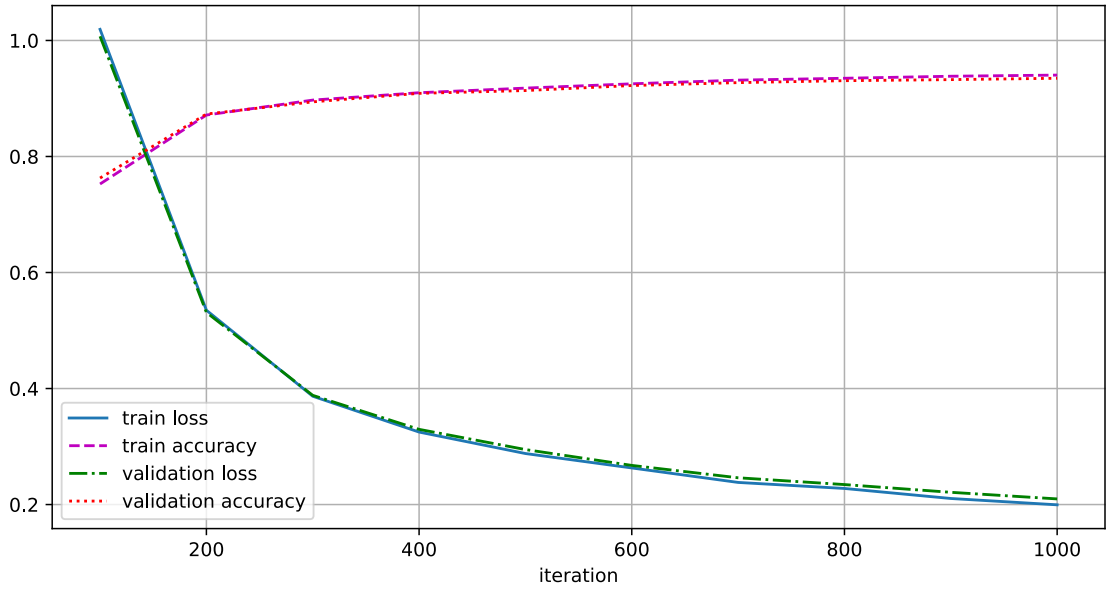
The training did not converge yet, but we already have a test accuracy of about 94%.

(g) Re-train the same network, with the *same* initial weights. Are the results *exactly* the same? (2 points)

[8]:
```
torch.manual_seed(12345)
simple_training.train()
```

```
train loss 0.198, train accuracy 0.941, val loss 0.209, val accuracy 0.934, test
loss 0.198, test accuracy 0.941, min val loss at iteration 1000
```

[8]:
```
Sequential(
  (0): Linear(in_features=784, out_features=300, bias=True)
  (1): Sigmoid()
  (2): Linear(in_features=300, out_features=100, bias=True)
  (3): Sigmoid()
  (4): Linear(in_features=100, out_features=10, bias=True)
)
```



Yes, when fixing the seed, and therefore the initialization between multiple runs, the results are exactly the same. We could verify this by making sure all checkpoints stay exactly the same. Even though it seems deterministic in this specific case, CUDA may be a source of randomness even if the seed it fixed, see here

(h) The LTH paper uses a variant of 'early-stopping'. How is this done? Implement it in your training loop. (1 point)

Hint: A simple way to keep track of the best model is to create a model checkpoint in a file `"checkpoints/model-...-best.pth"`.

Hint 2: It is okay to compute the validation scores less often, this can speed up training.

The paper uses the minimum validation loss as an early-stopping criterion, i.e., uses the model parameter that resulted in the minimal validation loss during training.

1.6 7.4 Pruning (13 points)

Next up, you should implement pruning. Starting with the one-shot pruning method.

Hint: Pruning is implemented already in PyTorch, in the module torch.utils.prune. The pruning method used in the LTH paper corresponds is called l1_unstructured in PyTorch.

(a) The PyTorch pruning function accepts an amount to prune. Is that the amount of weights to set to 0 or the amount to keep nonzero? Is the paper using the same? (1 point)

The pytorch implementation uses a fraction to represent how much to prune. The paper uses two notations. The table on page 3, for example uses the same notation as PyTorch, while the figures use the number of connections to keep, i.e., `1 - the connections to prune`.

(b) Should all parameters be pruned or only weights? Is the pruning rate the same for all types of layers? (1 point)

Only weights are pruned, no other paramters.

The output layers use half of the pruning rate of the other layers.

(c) Define a function to prune a network as used in the LTH paper. It should take an amount to prune as an argument. (3 points)

Hint: for a `Sequential` layer, you can access the layers as `net.children()`. For a layer, you can use `isinstance(layer, torch.nn.Linear)` to check if it is a linear layer.

(d) Check your pruning function on a very simple neural network. Print the layer weights before and after pruning to make sure you understand what is going on. (1 point)

Hint: the PyTorch pruning functions do not change the trainable parameters, rather they set `module.weight` to `weight_orig * weight_mask`.

[9]:
```
layer = next(simple_model.children())
layer.weight.detach().numpy()
```

[9]:
```
array([[ 0.03440844,  0.02711475,  0.03515311, …, -0.00918605,
         0.02984412,  0.01577999],
       [ 0.01094779, -0.02576708, -0.02192686, …, -0.02304294,
         0.01285846,  0.02556786],
       [ 0.03481854,  0.0045896 , -0.01619756, …,  0.00643216,
         0.02418092, -0.02184306],
       …,
       [-0.01016665, -0.00815832, -0.02156981, …,  0.01457045,
         0.03356247,  0.00566124],
       [ 0.01635601, -0.02943694,  0.02661679, …,  0.01629939,
        -0.01500897,  0.03353949],
       [ 0.00193773,  0.02985419, -0.02870209, …, -0.02147981,
        -0.00264075,  0.03220505]], dtype=float32)
```

[10]:
```
prune(simple_model, 0.5)
layer = next(simple_model.children())
layer.weight.detach().numpy()
```

[10]:
```
array([[ 0.,  0.,  0., …, -0.,  0.,  0.],
       [ 0., -0., -0., …, -0.,  0.,  0.],
       [ 0.,  0., -0., …,  0.,  0., -0.],
       …,
       [-0., -0., -0., …,  0.,  0.,  0.],
       [ 0., -0.,  0., …,  0., -0.,  0.],
       [ 0.,  0., -0., …, -0., -0.,  0.]], dtype=float32)
```

(e) Define a function that applies the pruning mask from a pruned network to another network of the same architecture. (2 points)

This function should only do pruning (so some weights become 0), other weights should remain the same.

Hint: the pruning functions already generate and store pruning masks. You should be able to extract these from a pruned network.

Hint 2: `custom_from_mask`

(f) Check your mask copy function on a very simple neural network. Check that only the pruning mask is copied. (1 point)

[12]:
```
torch.manual_seed(12345)
net1 = simple_training.train(graph=False)
prune(net1, 0.4)
layer = next(net1.children())
layer.weight.cpu().detach().numpy()
```

```
train loss 0.198, train accuracy 0.941, val loss 0.209, val accuracy 0.934, test
loss 0.198, test accuracy 0.941, min val loss at iteration 1000
```

[12]:
```
array([[ 0.,  0.,  0., …, -0.,  0.,  0.],
       [ 0., -0., -0., …, -0.,  0.,  0.],
       [ 0.,  0., -0., …,  0.,  0., -0.],
       …,
       [-0., -0., -0., …,  0.,  0.,  0.],
       [ 0., -0.,  0., …,  0., -0.,  0.],
       [ 0.,  0., -0., …, -0., -0.,  0.]], dtype=float32)
```

[13]:
```
torch.manual_seed(1)
net2 = simple_training.train(graph=False)
copy_prune(net1, net2)
layer = next(net2.children())
layer.weight.cpu().detach().numpy()
```

```
train loss 0.195, train accuracy 0.943, val loss 0.212, val accuracy 0.939, test
loss 0.201, test accuracy 0.940, min val loss at iteration 1000
```

[13]:
```
array([[ 0., -0., -0., …,  0., -0.,  0.],
       [-0., -0., -0., …, -0.,  0., -0.],
       [ 0., -0., -0., …, -0.,  0., -0.],
       …,
       [ 0.,  0.,  0., …,  0., -0.,  0.],
       [ 0., -0., -0., …, -0., -0., -0.],
       [ 0.,  0., -0., …, -0.,  0., -0.]], dtype=float32)
```

(g) Define a function that randomly prunes a network. (1 point)

(h) Check the above function. (1 point)

[14]:
```
torch.manual_seed(1)
net2 = simple_training.train(graph=False)
```

```
prune_random(net2, 0.5)
layer = next(net2.children())
layer.weight.cpu().detach().numpy()
```

```
train loss 0.195, train accuracy 0.943, val loss 0.212, val accuracy 0.939, test
loss 0.201, test accuracy 0.940, min val loss at iteration 1000
```

[14]:
```
array([[ 0.        , -0.        , -0.        , …,  0.        ,
        -0.00409383,  0.00250415],
       [-0.        , -0.02243793, -0.03091859, …, -0.0029185 ,
         0.        , -0.        ],
       [ 0.        , -0.        , -0.        , …, -0.        ,
         0.00562053, -0.        ],
       …,
       [ 0.        ,  0.03346273,  0.02398633, …,  0.01418912,
        -0.        ,  0.        ],
       [ 0.01004672, -0.02992319, -0.01731488, …, -0.        ,
        -0.        , -0.        ],
       [ 0.        ,  0.        , -0.01331014, …, -0.        ,
         0.032425  , -0.01334369]], dtype=float32)
```

(i) Define a function that performs the experiment described in Section 1 of the LTH paper on a given dataset and with a given architecture. (2 points)

Save all needed results to a file, such as test loss and accuracy. This will make your job easier later on.

1.7 7.5 Confirming the Lottery Ticket Hypothesis (10 points)

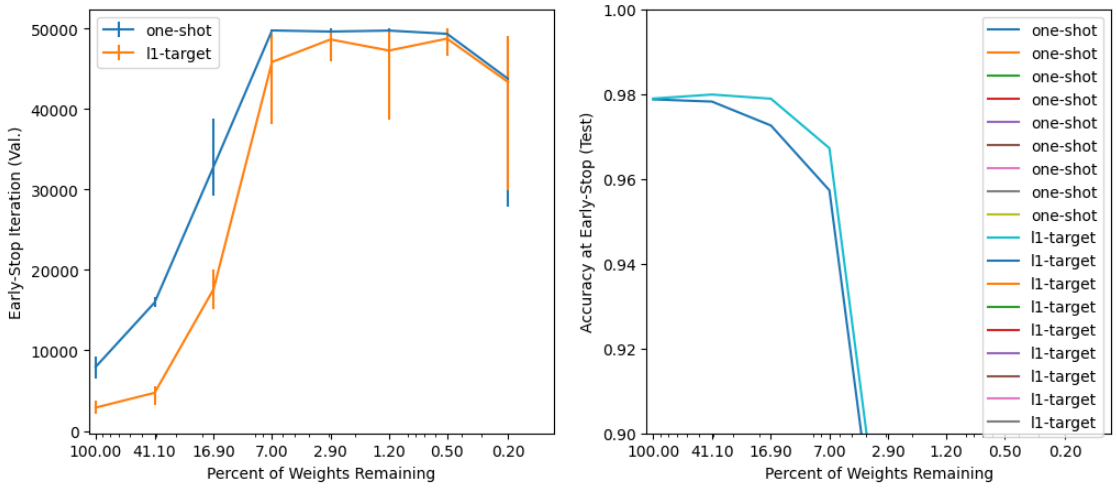
(a) Perform the experiments needed to reproduce the red lines in Figure 1 from the LTH paper. (6 points)

- It is ok to ignore the error bars for now, and focus on doing one series of experiments.
- You may also reduce the number of points in the plot to keep the computation time managable.
- You do not have to match the visual style of the figure.

Hint: create a python script (`experiment1-{dataset}-{method}.py`) that does all the training as needed. Then load the checkpoint files and do your analysis here. You may also want to define more helper functions.

Hint 2: look at previous assignments for how to plot. If you do want to include error bars, see also documentation for `plt.errorbar`.

[3]:
```
plot_experiment_1()
```



(b) Do your results match the paper? Discuss similarities and differences. (2 points)

The left picture does show the same idea as in the paper, namely that a lucky initialization can lead to a faster training, but it is by far not as extreme as shown in the paper.

The left picture does not match the figure in the paper, as apparently the correct initialization did not help with increasing the accuracy.

(c) What can you conclude from this experiment? (2 points)

We managed to successfully reproduce parts of the papers results, which can be seen as an indication the winning-ticket-hypothesis is correct.

There might still be optimizations possible that improve our results such that they match the results of the paper even more.

1.8 7.6 Experiments from Section 2 (12 points)

(a) What is the difference between the experiment in Figure 1 and Figure 3 of the paper. (1 point)

Hint: are there differences in the method, the architecture, or the dataset?

Figure 1 shows how the winning ticket trained network performs compared to a randomly pruned network in terms of speed, i.e., the number of iterations until the early stopping criterion.

In contrast to that, Figure 3 compares the accuracy of the networks for different pruning rates, using iterative pruning.

In Figure 1, multiple network architectures are compared, namely Lenet, Conv-6, Conv-4, and Con-2, while Figure 3 only shows the performance of Lenet.

The training for Figure 1 was for 50,000 iterations, while the training for Figure 3 only lasted for 20,000 iterations.

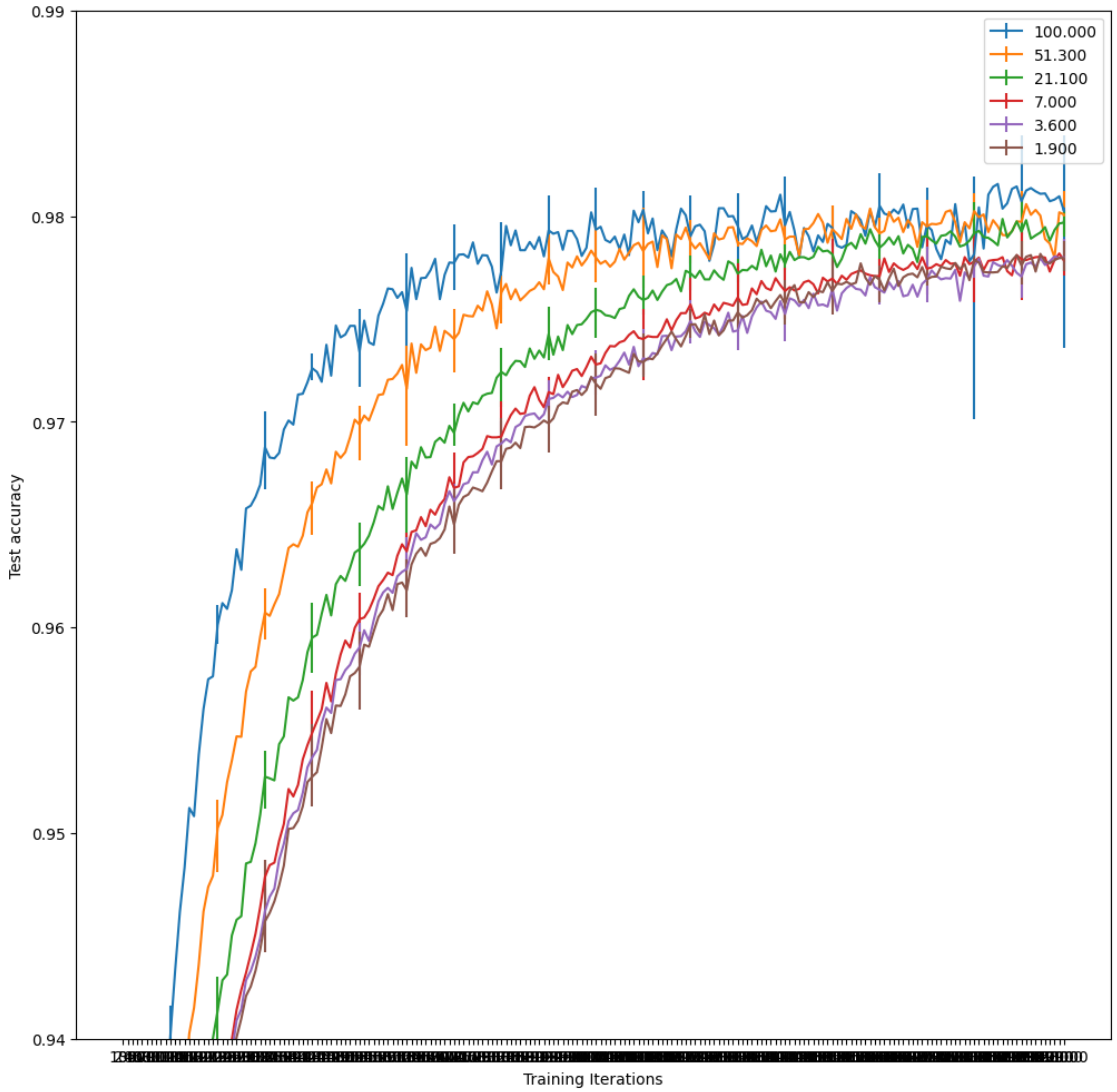
(b) Implement the iterative pruning method from the paper. (3 points)

As Strategy 1 from Appendix B has the better performance, we expect that this approach was implemented in the paper, and thus, we are going to implement it as well.

(c) Perform the experiments needed to reproduce Figure 4a from the paper. (4 points)

Hint: see previous section

[4]:
```
plot_iterative_pruning()
```



(d) Do your results match the paper? Discuss similarities and differences. (2 points)

No, unfortunately, they don't. The paper shows increasing test accuracy when pruning more weights. Our graph shows an increasingly worse performance, in terms of test accuracy, when pruning more weights.

(e) What can you conclude from this experiment? (2 points)

There exists a realistic chance that our implementation does not match the implementation used in the paper exactly. We can therefore neither prove that the results of the paper are wrong nor that they are correct.

1.9 7.7 Experiments from Section 4 (7 points)

Section 3 and 4 deal with convolutional neural networks. We are going to skip the networks in section 3, and move on to Figure 8.

(a) Section 4 of the paper describes a slightly different pruning method. Implement that method. (2 points)

Hint: look at `torch.nn.utils.prune.global_unstructured` and at the examples on that page.

If you get stuck on this step, you can continue with the same pruning methods as before.

TODO: Function implemented as: `TODO` in `dl_assignment_7_common.py`

(b) Implement a function that constructs the network architecture used in Figure 8. (1 point)

Extend the `create_network` function defined earlier.

Hint: VGG16 and Resnet18 are predefined in torchvision.

(c) Perform the experiments needed to reproduce Figure 8 from the LTH paper. (2 points)

- Again: you do not need to include error bars.
- You may limit yourself to one of the figures.

TODO: Training implemented in `TODO.py`

```
[ ]: # TODO: your code here
```

(d) Do your results match the paper? Discuss similarities and differences. (2 points)

TODO: your answer here.

1.10 The end

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

This assignment has 65 points. Version 97b3d19 / 2023-10-19