dl-assignment-2

September 18, 2023

1 Deep Learning — Assignment 2

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

Names: Daan Brugmans, Maximilian Pohl

Group: 31

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.

1.1 Objectives

In this assignment you will 1. Implement a neural network in PyTorch; 2. Use automatic differentiation to compute gradients; 3. Experiment with SGD and Adam; 4. Experiment with hyperparameter optimization; 5. Experiment with regularization techniques.

Before we start, if PyTorch or pandas is not installed, install it now with `pip install`.

[1]:
```
# !pip install torch
# !pip install torchvision
# !pip install pandas
# !pip install matplotlib
# !pip install tqdm
```

[2]:
```
%config InlineBackend.figure_formats =['png']
%matplotlib inline
import numpy as np
import sklearn.datasets
import matplotlib.pyplot as plt
import torch
import time
import torchvision
import tqdm.notebook as tqdm
import collections
import IPython
import pandas as pd

np.set_printoptions(suppress=True, precision=6, linewidth=200)
plt.style.use('ggplot')

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

1.2 2.1 Implementing a model with PyTorch

In the first assignment, you implemented a neural network from scratch in NumPy. In practice, it is more convenient to use a deep learning framework. In this course, we use PyTorch.

In this example, we will use PyTorch to implement and train a simple neural network.

1.2.1 PyTorch tensors

Similar to NumPy, PyTorch works with multi-dimensional tensors. These can be scalars, vectors, matrices, or have an even higher dimension.

Tensors can be created by converting NumPy arrays, or directly in PyTorch:

[3]:
```
# create a 10x10 matrix filled with zeros
x = np.zeros([10, 10])
x = torch.tensor(x)
print(x)

# create a 3x5 matrix of ones
x = torch.ones([3, 5])
print(x)
```

```
tensor([[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., 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., 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=torch.float64)
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

Note that the tensors have a datatype (`dtype`) that defines the type of number in the matrix. This can be 32-bit or 64-bit floating point numbers, or various types of integers. (See the PyTorch documentation for a complete list.)

[4]:
```python
print('float32:', torch.tensor([1., 2., 3.], dtype=torch.float32))
print('float64:', torch.tensor([1., 2., 3.], dtype=torch.float64))
print('int:', torch.tensor([1, 2, 3], dtype=torch.int))
print('long:', torch.tensor([1, 2, 3], dtype=torch.long))
```

```
float32: tensor([1., 2., 3.])
float64: tensor([1., 2., 3.], dtype=torch.float64)
int: tensor([1, 2, 3], dtype=torch.int32)
long: tensor([1, 2, 3])
```

It can be important to choose the correct datatype for your tensors, because this influences the precision, the computational cost, or the memory requirements. Some functions also specifically require integers.

1.2.2 Moving data to and from the GPU

If you have a GPU, you can use it to increase the speed of PyTorch computations. To do this, you have to move your data to the GPU by calling `to('cuda')`. Afterwards, you can move the results back to the CPU by calling `to('cpu')` or `cpu()`.

Note: in MacOS you can use the `'mps'` backend instead of `'cuda'`.

[5]:
```python
# this only works if you have a GPU available
if torch.cuda.is_available():
    # define a variable on the CPU
    x = torch.ones((3,))
    # move the variable to the GPU
    x = x.to('cuda')
    print('x is now on the GPU:', x)
    # move the variable back to the CPU
    x = x.to('cpu')
    print('x is now on the CPU:', x)
elif torch.backends.mps.is_available():
    # define a variable on the CPU
    x = torch.ones((3,))
    # move the variable to the GPU
    x = x.to('mps')
    print('x is now on the GPU:', x)
    # move the variable back to the CPU
    x = x.to('cpu')
    print('x is now on the CPU:', x)
else:
    print('It looks like you don\'t have a GPU available.')
```

```
x is now on the GPU: tensor([1., 1., 1.], device='cuda:0')
x is now on the CPU: tensor([1., 1., 1.])
```

Note: If you want to run a computation on the GPU, all variables of that function should be moved to the GPU first. If some variables are still on the CPU, PyTorch will throw an error.

If you have a GPU, make the following code run without errors.

[6]:
```
if torch.cuda.is_available():
    x = torch.tensor([1, 2, 3], device='cuda')
    y = torch.tensor([1, 2, 3], device='cuda')
    print('x is on the CPU:', x)
    print('y is on the GPU:', y)
    z = x * y
    print(z)
```

```
x is on the CPU: tensor([1, 2, 3], device='cuda:0')
y is on the GPU: tensor([1, 2, 3], device='cuda:0')
tensor([1, 4, 9], device='cuda:0')
```

To use the GPU when it is available, and fall back to the CPU otherwise, a common trick is to define a global `device` constant. You can then use `tensor.to(device)` and `torch.tensor(device=device)`.

[7]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "mps" if torch.
  ↪backends.mps.is_available() else "cpu")
```

1.2.3 Converting back to NumPy

Sometimes, it is useful to convert PyTorch tensors back to NumPy arrays, for example, if you want to plot the performance of your network.

Call `detach().cpu().numpy()` on the tensor variable to convert the variable to NumPy:

[8]:
```
x = torch.tensor(5.)
x_in_numpy = x.detach().cpu().numpy()
x_in_numpy
```

[8]:
```
array(5., dtype=float32)
```

Note: `detach` detaches the tensor from the other computation, by among other things, removing gradient information. `cpu` transfers the tensor from GPU to CPU if needed. Finally `numpy` converts from a pytorch tensor to a numpy array, these function very similarly, they just come from different libraries.

1.2.4 Computing the gradients of a simple model

You can use the PyTorch tensors to perform computations, such as the function $y = x \cdot w + b$:

[9]:
```
w = torch.tensor(2.)
b = torch.tensor(1.)
x = torch.tensor(5.)
print('w:', w)
print('b:', b)
print('x:', x)
```

```
y = x * w + b
print(y)
```

```
w: tensor(2.)
b: tensor(1.)
x: tensor(5.)
tensor(11.)
```

If we would like to compute the gradient for the parameters `w` and `b`, we could derive and compute them manually (as you did last week):

[10]:
```
y_grad = 1
w_grad = y_grad * x
b_grad = y_grad
print('w_grad:', w_grad)
print('b_grad:', b_grad)
```

```
w_grad: tensor(5.)
b_grad: 1
```

This can be a lot of work, and it is easy to make mistakes. Fortunately, PyTorch (and other deep learning libraries) can compute these gradients automatically using automatic differentiation.

1.2.5 Computing the gradient automatically

You can compute an automatic gradient as follows:

1. Tell PyTorch which variables need a gradient. You can do this by setting `requires_grad=True` when you define the variable.
2. Perform the computation.
3. Use the `backward()` function on the result to compute the gradients using backpropagation.
4. The `grad` property of your variables will now contain the gradient.

Have a look at this example, and compare the gradients with the gradients we computed manually:

[11]:
```
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(1., requires_grad=True)
x = torch.tensor(5.)

# compute the function
y = x * w + b

# compute the gradients, given dy = 1
y.backward()

print('w.grad:', w.grad)
print('b.grad:', b.grad)
# x did not have requires_grad, so no gradient was computed
print('x.grad:', x.grad)
```

```
w.grad: tensor(5.)
b.grad: tensor(1.)
x.grad: None
```

This also works for much more complicated functions (and even entire neural networks):

[12]:
```python
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(1., requires_grad=True)
x = torch.tensor(5.)

y = torch.exp(torch.sin(x * w) + b)
y.backward()

print('w.grad:', w.grad)
print('b.grad:', b.grad)
```

```
w.grad: tensor(-6.6191)
b.grad: tensor(1.5777)
```

1.2.6 Inspect the automatic differentiation history

PyTorch can compute these gradients automatically because it keeps track of the operations that generated the result.

While you don't normally need to do this, you can look inside `y.grad_fn` to see how it works:

[13]:
```python
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(1., requires_grad=True)
x = torch.tensor(5.)

y = x * w + b

print('y.grad_fn:', y.grad_fn)
print('  \-->', y.grad_fn.next_functions)
print('           \-->', y.grad_fn.next_functions[0][0].next_functions)
```

```
y.grad_fn: <AddBackward0 object at 0x7f1d2f316fe0>
  \--> ((<MulBackward0 object at 0x7f1d2f317c70>, 0), (<AccumulateGrad object at
0x7f1d2f317cd0>, 0))
          \--> ((None, 0), (<AccumulateGrad object at 0x7f1d2f315990>, 0))
```

The `grad_fn` of `y` contains a tree that reflects how `y` was computed: * the last operation was an addition (x * w) **plus** b: `AddBackward0` knows how to compute the gradient of that; * one of the inputs to the addition was a multiplication x **times** w: `MulBackward0` computes the gradient; * eventually, the backpropagation reaches the input variables: `AccumulateGrad` is used to store the gradient in the `grad` property of each variable.

As long as you use operations for which PyTorch knows the gradient, the `backward()` function can perform automatic backpropagation and the chain rule to compute the gradients. If you want, can read more about this in the PyTorch autograd tutorial.

1.3 2.2 PyTorch neural network with torch.nn (1 point)

The `torch.nn` module of PyTorch contains a large number of building blocks to construct your own neural network architectures. You will need this in this and future assignments. Have a look at the documentation for `torch.nn` to see what is available. In this assignment, we will use `torch.nn.Linear` to build networks with linear layers, as well as some activation and loss functions.

1.3.1 A network module

As a first example, the two-layer network from last week can be implemented like this:

[14]:
```
class Week1Net(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.layer1 = torch.nn.Linear(64, 32)
        self.relu = torch.nn.ReLU()
        self.layer2 = torch.nn.Linear(32, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x


net = Week1Net()
```

Observe the following: * A network in PyTorch is usually implemented as a subclass of `torch.nn.Module`, as it is here. * The `__init__` function defines the layers that are used in the network. * The `forward` function computes the output of the network given one or more inputs (in this case: `x`).

Notice that there is no final activation function (such as a sigmoid or softmax) at the end of the network. This is not a mistake: we will do this later, by using a loss function that combines the sigmoid/softmax and the loss in one computation, because this is more numerically stable.

1.3.2 Network parameters

Some of the components of the network have parameters, such as the weight and bias in the Linear layers. We can list them using the `parameters` or `named_parameters` function:

[15]:
```
for name, param in net.named_parameters():
    print(name)
    print(param)
    print()
```

```
layer1.weight
Parameter containing:
tensor([[ 0.1204,  0.0949,  0.1230,  …,  0.0031,  0.0987,  0.0778],
         [-0.0368, -0.0247, -0.0457,  …,  0.1129, -0.1046, -0.0674],
         [-0.0711,  0.0584, -0.1189,  …, -0.0951, -0.0886, -0.1063],
         …,
         [ 0.1221,  0.0225,  0.0512,  …,  0.1025, -0.0922,  0.0637],
         [ 0.0418,  0.0997, -0.1024,  …,  0.0736,  0.0114, -0.0886],
         [-0.0612, -0.1228, -0.0385,  …, -0.0722, -0.0586,  0.0396]],
       requires_grad=True)

layer1.bias
Parameter containing:
tensor([ 0.1221, -0.0313,  0.0030,  0.1030, -0.0493,  0.1041, -0.1215,  0.1121,
         0.0570,  0.0154,  0.0428,  0.1016,  0.0611, -0.0624,  0.0602,  0.0817,
        -0.0005, -0.0573,  0.0241,  0.0720, -0.0800,  0.0652,  0.0580, -0.1215,
         0.0597,  0.1039, -0.0357,  0.0801, -0.0139,  0.0949,  0.0461,  0.0790],
       requires_grad=True)

layer2.weight
Parameter containing:
tensor([[ 0.1005, -0.0351, -0.1699,  0.0777, -0.0036,  0.0751,  0.1679,  0.1333,
         -0.0726,  0.1432, -0.0506, -0.1634,  0.1369, -0.0367, -0.1742, -0.0405,
         -0.1532,  0.0290,  0.0143, -0.1697,  0.0045,  0.0300,  0.0904, -0.1308,
         -0.0617, -0.1161,  0.1292,  0.1582,  0.0042, -0.1346,  0.1013,
-0.0748],
        [-0.0751,  0.0436, -0.1357, -0.0882,  0.0188, -0.0927,  0.0721, -0.1600,
          0.0224, -0.1211, -0.0192,  0.0276, -0.0662,  0.0555,  0.1753,  0.1378,
          0.1409,  0.0317, -0.0507, -0.0087,  0.1636,  0.1673, -0.0612,  0.0134,
         -0.0218,  0.1729, -0.0871, -0.0755,  0.1557,  0.1195, -0.0141,
0.0056],
        [ 0.0592,  0.1288,  0.1240, -0.1467,  0.0861,  0.1423, -0.1154, -0.0852,
          0.1663, -0.1341, -0.1102, -0.1478, -0.0226,  0.0725, -0.0525, -0.0006,
          0.0511, -0.0813,  0.0825,  0.0361, -0.1439, -0.1459, -0.0694, -0.0278,
          0.0586, -0.1597, -0.0381,  0.0511, -0.0302,  0.1037,  0.1678,
-0.1638],
        [-0.1242, -0.0819,  0.0434, -0.0663, -0.1559,  0.0854,  0.0812,  0.0954,
         -0.0056,  0.1144,  0.1415, -0.0693, -0.0386, -0.1554, -0.1176, -0.0675,
         -0.1691, -0.0852, -0.0780, -0.0113,  0.1593, -0.1508,  0.0891, -0.0631,
         -0.0919,  0.1456, -0.0824,  0.0032, -0.1406, -0.1423,  0.1550,
0.0381],
        [-0.1145,  0.0736, -0.0803,  0.1550, -0.0515,  0.0311, -0.1039,  0.0592,
         -0.0226,  0.0356,  0.1043, -0.1755, -0.1055, -0.0358, -0.1417,  0.1595,
          0.1576,  0.0066, -0.1628,  0.0522,  0.1662, -0.1428, -0.0405, -0.1686,
          0.1727, -0.1234,  0.0946, -0.1440,  0.1661,  0.0724, -0.1472,
-0.0787],
        [-0.0453, -0.0660, -0.1202, -0.0731,  0.0487, -0.1479,  0.0916,  0.0244,
          0.0995, -0.0514, -0.1416,  0.0963,  0.1590, -0.1648, -0.0693,  0.0358,
          0.0075,  0.0890, -0.0311,  0.1225,  0.1644,  0.0101, -0.1744,  0.0538,
          0.1388, -0.0937,  0.1376,  0.1069,  0.1303,  0.0128,  0.1552,
-0.0848],
        [-0.1508,  0.1222,  0.0316,  0.0804, -0.0110,  0.1416, -0.0443,  0.0694,
          0.1205, -0.0790,  0.0266,  0.1356, -0.0878, -0.0796, -0.1237, -0.1016,
         -0.0188, -0.0912,  0.1273, -0.0179,  0.0519,  0.0676,  0.0304, -0.0700,
          0.0929, -0.1287, -0.0336, -0.1470,  0.1354,  0.1603, -0.0828,
0.0509],
        [ 0.0032, -0.1590, -0.1421, -0.0455,  0.0651,  0.0982, -0.0494, -0.1568,
         -0.0634,  0.0235,  0.0327, -0.0177,  0.0686, -0.1389, -0.1247, -0.0117,
         -0.0403,  0.0905,  0.0059, -0.1593, -0.0617,  0.1646, -0.0644,  0.0848,
         -0.1262,  0.0425,  0.0559,  0.1635, -0.0367, -0.0321, -0.0527,
0.0354],
        [-0.0517, -0.1339,  0.1701, -0.0203, -0.1142, -0.0950, -0.0854, -0.0799,
         -0.0870,  0.0674,  0.1186, -0.1028, -0.1255,  0.0318,  0.1197, -0.1081,
         -0.1753, -0.0694,  0.0787, -0.1544,  0.1407,  0.0042,  0.1198,  0.0164,
         -0.1663,  0.0190, -0.0645, -0.0124,  0.0778,  0.1020, -0.1467,
-0.0847],
        [-0.0396,  0.0901, -0.0129, -0.0929, -0.1193, -0.1335, -0.0279, -0.0114,
          0.1294, -0.0672, -0.0905,  0.1248,  0.0537,  0.0499, -0.0461, -0.0606,
         -0.0840,  0.0231, -0.0480,  0.0691, -0.0595,  0.1688,  0.1069,  0.0172,
         -0.0058,  0.0096,  0.1693,  0.0193,  0.0524, -0.1406,  0.1590,
-0.1754]],
       requires_grad=True)

layer2.bias
Parameter containing:
tensor([-0.0152,  0.1297,  0.0527,  0.1201, -0.1444,  0.1721,  0.0794,  0.0805,
         0.1084, -0.1591], requires_grad=True)
```

As you can see, these parameters have been initialized to non-zero values.

(a) Why are these weights not zero? (1 point)

If we initialize the network with zero weights, we could not compute reasonable gradients to search for the gradient decent, as the multiplication with an all-zero weight matrix will cause all derivatives to become zero.

1.3.3 Shortcut: use `torch.nn.Sequential`

Quite often, as in our network above, a network architecture consists of a number of layers that are computed one after the other. PyTorch has a special `torch.nn.Sequential` function to quickly define these networks, without having to define a new class.

For example, the network we implemented earlier can also be written like this:

[16]:
```python
def build_net():
    return torch.nn.Sequential(
        torch.nn.Linear(64, 32),
        torch.nn.ReLU(),
        torch.nn.Linear(32, 10)
    )
```

```
net = build_net()
print(net)
```

```
Sequential(
  (0): Linear(in_features=64, out_features=32, bias=True)
  (1): ReLU()
  (2): Linear(in_features=32, out_features=10, bias=True)
)
```

1.4 2.3 A neural network for Fashion-MNIST (12 points)

In this assignment, we will do experiments with the Fashion-MNIST dataset. First, we download the dataset, and create a random training set with 1000 images and a validation set with 500 images:

[17]:
```
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)])
```

1.4.1 Plot some images

The Fashion-MNIST contains images of 28 by 28 pixels, from 10 different classes. In our experiments we flatten the images to a vector with 28 x 28 = 784 features.

[18]:
```
# plot some of the images
plt.figure(figsize=(10, 2))
plt.imshow(np.hstack([fashion_train[i][0].reshape(28, 28) for i in range(9)]),␣
  ↪cmap='gray')
plt.grid(False)
plt.tight_layout()
plt.axis('off')
plt.title('labels: ' + str([fashion_train[i][1] for i in range(9)]));
```

labels: [7, 9, 6, 0, 5, 3, 9, 8, 4]

(a) Can you, as a human, distinguish the different classes? Do you think a neural network should be able to learn to do this as well? (1 point)

As a human, I can easily distinguish things like shoes, trousers, and shirts from each other, but distinguishing a shirt from a hoody, for example, is harder while still possible. As for the numbers, I think a complex enough network can get a high accuracy, but will not reach 100%.

1.4.2 Use the DataLoader to create batches

We will use the `DataLoader` from PyTorch (see the documentation) to automatically create random batches of 10 images:

```
[19]: data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=10,␣
        ↪shuffle=True)
```

We will use the data loader to loop over all batches in the dataset.

For each batch, we get `x`, a tensor containing the images, and `y`, containing the label for each image:

```
[20]: for x, y in data_loader:
          print('x.dtype:', x.dtype, 'x.shape:', x.shape)
          print('y.dtype:', y.dtype, '  y.shape:', y.shape)
          # one batch is enough for now
          break
```

```
x.dtype: torch.float32 x.shape: torch.Size([10, 784])
y.dtype: torch.int64   y.shape: torch.Size([10])
```

1.4.3 Construct a network

We will construct a network that can classify these images.

(b) Implement a network with the following architecture using `torch.nn.Sequential`: (2 points)

- Accept flattened inputs: 28 x 28 images mean an input vector with 784 elements.
- Linear hidden layer 1, ReLU activation, output 128 features.
- Linear hidden layer 2, ReLU activation, output 64 features.
- Linear output layer, to 10 classes.
- No final activation function.

```
[21]: def build_net() -> torch.nn.Module:
          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)
          )
```

```
net = build_net()
print(net)
```

```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
)
```

(c) Test your network by creating a data loader and computing the output for one batch: (3 points)

[22]:
```
# use a data loader to loop over fashion_train,
# with a batch size of 16, shuffle the dataset
data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=16,␣
 ↪shuffle=True)

net = build_net()
for x, y in data_loader:
    print('x.shape:', x.shape)
    print('y.shape:', y.shape)
    output = net.forward(x)
    print('output.shape', output.shape)
    break
```

```
x.shape: torch.Size([16, 784])
y.shape: torch.Size([16])
output.shape torch.Size([16, 10])
```

1.4.4 Train a network with PyTorch

To train the network, we need a number of components: * A network, like the one you just defined. * A DataLoader to loop over the training samples in small batches. * A loss function, such as the cross-entropy loss. See the loss functions in the PyTorch documentation. * An optimizer, such as SGD or Adam: after we use the `backward` function to compute the gradients, the optimizer computes and applies the updates to the weights of the network. See the optimization algorithms in the PyTorch documentation.

As an example, the code below implements all of these components and runs a single update step of the network.

(d) Have a look at the code to understand how it works. Then make the following changes: (4 points) * Set the batch size to 16 * Use Adam as the optimizer and set the learning rate to 0.01 * For each minibatch, compute the output of the network * Compute and optimize the cross-entropy loss

[23]:
```
# initialize a new instance of the network
net = build_net()

# construct a data loader for the training set
data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=16,␣
 ↪shuffle=True)

# initialize the Adam optimizer
# we pass the list of parameters of the network
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)

loss_function = torch.nn.CrossEntropyLoss()

# repeat for multiple epochs
for epoch in range(10):
    # compute the mean loss and accuracy for this epoch
    loss_sum = 0.0
    accuracy_sum = 0.0
    steps = 0

    # loop over all minibatches in the training set
    for x, y in data_loader:
        # compute the prediction given the input x
        output = net.forward(x)

        # compute the loss by comparing with the target output y
        loss = loss_function(output, y)

        # for a one-hot encoding, the output is a score for each class
        # we assign each sample to the class with the highest score
        pred_class = torch.argmax(output, dim=1)
        # compute the mean accuracy
        accuracy = torch.mean((pred_class == y).to(float))

        # reset all gradients to zero before backpropagation
        optimizer.zero_grad()
        # compute the gradient
        loss.backward()
        # use the optimizer to update the parameters
        optimizer.step()

        accuracy_sum += accuracy.detach().cpu().numpy()
        loss_sum += loss.detach().cpu().numpy()
        steps += 1

    # print('y:', y)
    # print('pred_class:', pred_class)
    # print('accuracy:', accuracy)
    print('epoch:', epoch,
          'loss:', loss_sum / steps,
          'accuracy:', accuracy_sum / steps)
```

```
epoch: 0 loss: 1.1801386555982014 accuracy: 0.5525793650793651
epoch: 1 loss: 0.7564850208305177 accuracy: 0.7103174603174603
epoch: 2 loss: 0.6169877696841483 accuracy: 0.7638888888888888
epoch: 3 loss: 0.5665754687100176 accuracy: 0.7886904761904762
epoch: 4 loss: 0.5834637322123088 accuracy: 0.8005952380952381
epoch: 5 loss: 0.5290524505433583 accuracy: 0.8214285714285714
epoch: 6 loss: 0.5228580394907604 accuracy: 0.8134920634920635
epoch: 7 loss: 0.4587841405281945 accuracy: 0.8313492063492064
epoch: 8 loss: 0.3974530810401553 accuracy: 0.8630952380952381
epoch: 9 loss: 0.4477589400041671 accuracy: 0.8601190476190477
```

(e) Run the optimization for a few epochs. Does the loss go down? Has the training converged? (1 point)

Yes, the loss decreased, while the accuracy converged from epoch 6 onwards to around 84 %.

(f) Looking back at the network, we did not include a SoftMax activation function after the last linear layer. But typically you need to use a softmax activation when using cross-entropy loss. Was there a mistake? (1 point)

Hint: Look at the documentation of the cross-entropy loss function. Is the formula there the same as in the slides?

The formula in the documentation shows that it already combines the Categorical CBE loss and the softmax activation function, as we have learned it in the lecture

(optional) Why do you think the developers of PyTorch did it this way?

Probably because of efficiency and/or numerical stability for the floating point arithmetic.

1.5 2.4 Training code for the rest of this assignment

For the rest of this assignment, we will use a slightly more advanced training function. It runs the training loop for multiple epochs, and at the end of each epoch evaluates the network on the validation set.

Feel free to look inside, but keep in mind that some of this code is only needed to generate the plots in this assignment.

[24]:
```
def fit(net: torch.nn.Module, train, validation, optimizer, epochs=25,␣
 ↪batch_size=10, device=device):
    """
    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
      - 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).to(float))

                # add to epoch loss and accuracy
                epoch_stats['%s loss' % phase] += loss.detach().cpu().numpy()
                epoch_stats['%s accuracy' % phase] += accuracy.detach().cpu().
↪numpy()

                # 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().numpy()
                    epoch_stats['train mean abs weight'] += \
                        torch.mean(torch.abs(net[0].weight)).detach().cpu().
↪numpy()

                    # update the weights
                    optimizer.step()

                epoch_stats['%s steps' % phase] += 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['%s␣
↪steps' % phase]
                    history[key].append(epoch_stats[key])

            # count the number of update steps
            history['%s steps' % phase].append((epoch + 1) * epoch_stats['%s␣
↪steps' % phase])

            # store the outputs
            history['%s outputs' % phase].append(np.
↪concatenate(epoch_outputs[phase]).flatten())

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

    return history
```

[25]:
```
# helper code to plot our results
class HistoryPlotter:
    def __init__(self, plots, table, rows, cols):
        self.plots = plots
        self.table = table
        self.rows = rows
        self.cols = cols
        self.histories = {}
        self.results = []

        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):
        self.histories[title] = history
        self.results.append((title, {key: history[key][-1] for key in self.
↪table}))

        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']:
                ax.plot(history[plot_xy['x']], history[key], label=key)
            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()
```

1.6 2.5 Optimization and hyperparameters (10 points)

An important part of training a neural network is hyperparameter optimization: finding good learning rates, minibatch sizes, and other parameters to train an efficient and effective network.

In this part, we will explore some of the most common hyperparameters.

1.6.1 Learning rate with SGD and Adam

First, we will investigate optimizers and learning rates: * The choice of optimizer: SGD and especially Adam are common choices. * The learning rate determines the size of the updates by the optimizer.

Optimizing hyperparameters is often a matter of trial-and-error.

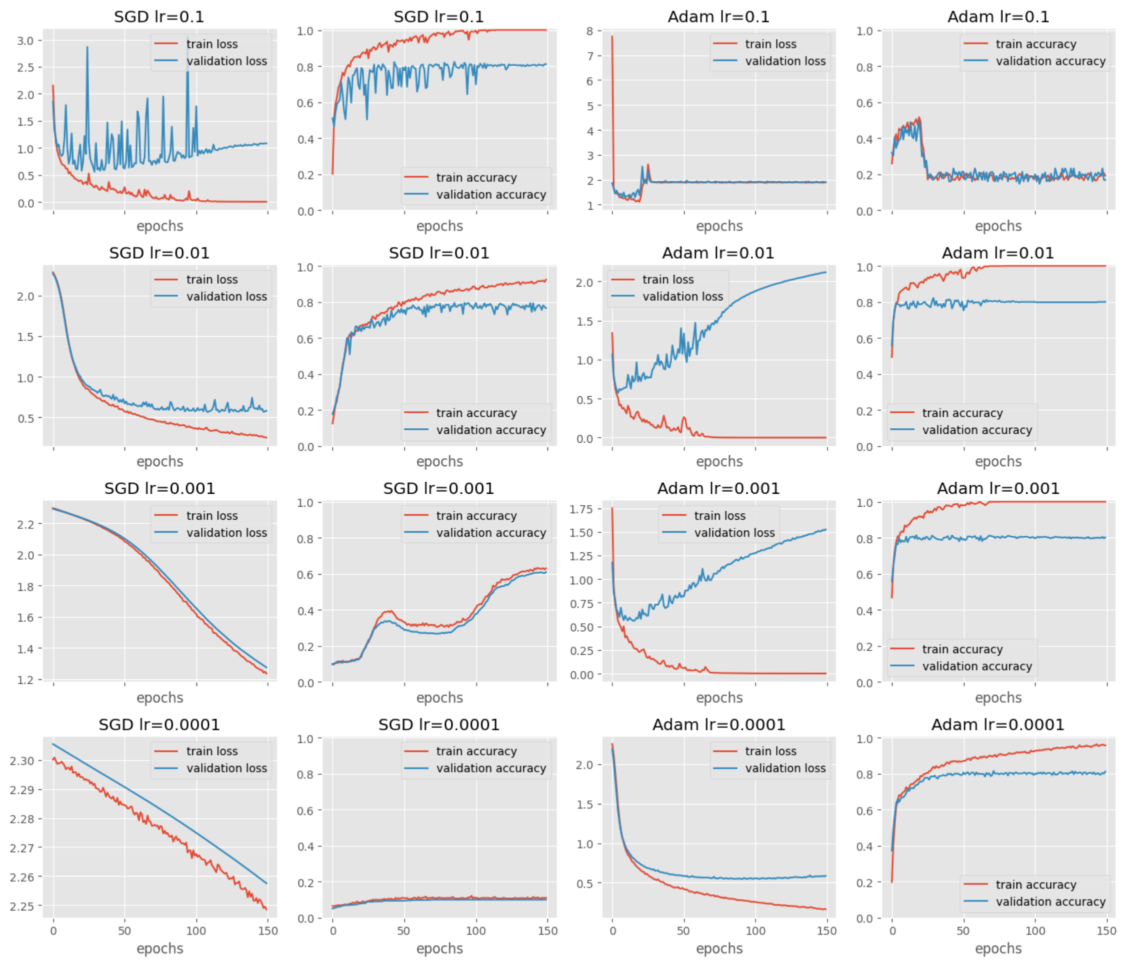
We will run an experiment to train our network with the following settings: * Optimizer: SGD or Adam * Learning rate: 0.1, 0.01, 0.001, 0.0001 * Minibatch size: 32 * 150 epochs

For each setting, we will plot: * The train and validation accuracy * The train and validation loss

We will also print a table with the results of the final epoch.

(a) Run the experiment and have a look at the results.

[26]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation␣
 ↪loss']},
                                {'x': 'epochs', 'y': ['train accuracy',␣
 ↪'validation accuracy']}, ],
                         table=['train accuracy', 'validation accuracy'],
                         rows=4, cols=2)

epochs = 150
batch_size = 32

for row, lr in enumerate((0.1, 0.01, 0.001, 0.0001)):
    net = build_net()
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
 ↪epochs=epochs, batch_size=batch_size)
    plotter.add('SGD lr=%s' % str(lr), history, row=row, col=0)

    net = build_net()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
 ↪epochs=epochs, batch_size=batch_size)
    plotter.add('Adam lr=%s' % str(lr), history, row=row, col=1)

plotter.done()
```



```
       experiment  train accuracy  validation accuracy
0      SGD lr=0.1        1.000000             0.810156
1     Adam lr=0.1        0.191406             0.167187
2     SGD lr=0.01        0.923828             0.764844
3    Adam lr=0.01        1.000000             0.799609
4    SGD lr=0.001        0.629883             0.609375
5   Adam lr=0.001        1.000000             0.801953
6   SGD lr=0.0001        0.110352             0.100000
7  Adam lr=0.0001        0.957031             0.810937
```

As you can see, not every combination of hyperparameters works equally well.

(b) Was 150 epochs long enough to train the network with all settings? List the experiments that have/have not yet converged. (2 points)

Experiments that have converged to a good result: SGD lr=0.1, Adam lr=0.001, Adam lr=0.01

Experiments that need more training: SGD lr=0.001, SGD lr=0.0001, SGD lr=0.01, Adam lr=0.0001

Other experiments: Adam lr=0.1

(c) How does the learning rate affect the speed of convergence? (1 point)

In case the learning rate is not too big to converge at all, a bigger learning rate leads to a faster converges

(d) A larger learning rate does not always lead to better or faster training. What happened to Adam with a learning rate of 0.1? (1 point)

As the learning rate is too high, the loss does not converge at all, but keeps moving around the whole search space.

(e) It seems that Adam works reasonably well with learning rates 0.01, 0.001, and 0.0001. Can you explain the difference between the three learning curves, in terms of performance, stability, and speed? (2 points)

Final performance: The final validation accuracy is equally good for all variants. The training accuracy for 0.0001 is only at 95%, while the other get to 100%

Stability: The smaller the lr is, the more stable the learning curve is.

Speed: The bigger the lr, the faster the accuracy converges

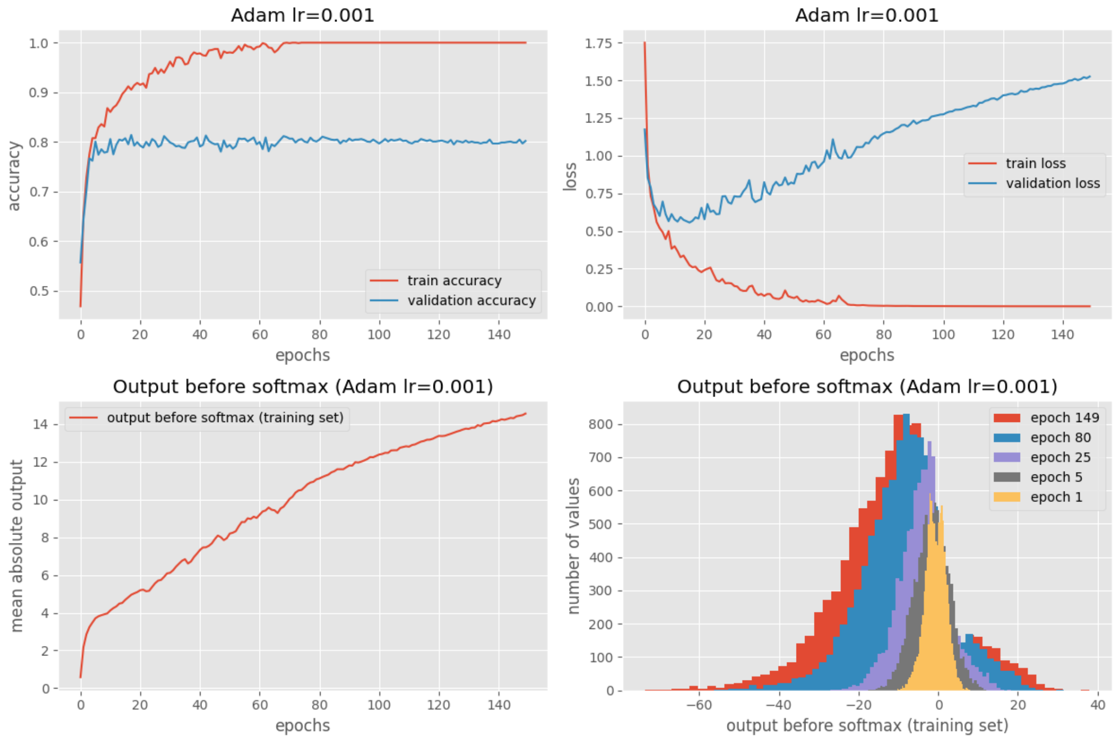
1.6.2 Same accuracy, increasing loss

You may have noticed something interesting in the curves for “Adam lr=0.001”: after 10 to 20 epochs, the loss on the validation set starts increasing again, while the accuracy remains the same. How is this possible?

We can find a clue by looking at the output of the network. We will plot the final outputs: the prediction just before the softmax activation function. These values are also called ‘logits’.

(f) Run the code below to generate the plots.

[27]:
```python
def plot_output_stats():
    fig, axs = plt.subplots(ncols=2, nrows=2,
                            figsize=(6 * 2, 4 * 2))
    # plot train and validation accuracy
    axs[0, 0].plot(plotter.histories['Adam lr=0.001']['epochs'],
                   plotter.histories['Adam lr=0.001']['train accuracy'],
                   label='train accuracy')
    axs[0, 0].plot(plotter.histories['Adam lr=0.001']['epochs'],
                   plotter.histories['Adam lr=0.001']['validation accuracy'],
                   label='validation accuracy')
    axs[0, 0].set_xlabel('epochs')
    axs[0, 0].set_ylabel('accuracy')
    axs[0, 0].set_title('Adam lr=0.001')
    axs[0, 0].legend()

    # plot train and validation loss
    axs[0, 1].plot(plotter.histories['Adam lr=0.001']['epochs'],
                   plotter.histories['Adam lr=0.001']['train loss'],
                   label='train loss')
    axs[0, 1].plot(plotter.histories['Adam lr=0.001']['epochs'],
                   plotter.histories['Adam lr=0.001']['validation loss'],
                   label='validation loss')
    axs[0, 1].set_xlabel('epochs')
    axs[0, 1].set_ylabel('loss')
    axs[0, 1].set_title('Adam lr=0.001')
    axs[0, 1].legend()

    # plot curve of mean absolute output values
    axs[1, 0].plot(plotter.histories['Adam lr=0.001']['epochs'],
                   plotter.histories['Adam lr=0.001']['train 4: Linear output␣
 ↪abs mean'],
                   label='output before softmax (training set)')
    axs[1, 0].set_xlabel('epochs')
    axs[1, 0].set_ylabel('mean absolute output')
    axs[1, 0].set_title('Output before softmax (Adam lr=0.001)')
    axs[1, 0].legend()

    # plot distributions of output values
    for epoch in (149, 80, 25, 5, 1):
        axs[1, 1].hist(plotter.histories['Adam lr=0.001']['train␣
 ↪outputs'][epoch], bins=50,
                       label='epoch %d' % epoch)
    axs[1, 1].set_xlabel('output before softmax (training set)')
    axs[1, 1].set_ylabel('number of values')
    axs[1, 1].set_title('Output before softmax (Adam lr=0.001)')
    axs[1, 1].legend()

    plt.tight_layout()


plot_output_stats()
```



Bottom left: the mean of the final outputs for all training images at different epochs. Bottom right: histograms showing the distribution of the output values at different epochs.

You should now be able to answer this question:

(g) Why does the accuracy remain stable while the loss keeps increasing? (1 points)

Perhaps you can combine these plots with your knowledge of the softmax activation and cross-entropy loss function to explain this curious behaviour.

The network gains more confidence about its predictions over time, which we can see in the bigger values being fed to the softmax function. This results in bigger penalties in case a prediction is wrong. As the model overfits to the training data, the validation set still hits wrong predictions, but with a higher confidence, which results in an increasing loss.

1.6.3 Minibatch size

Another important hyperparameter is the minibatch size. Sometimes your minibatch size is limited by the available memory in your GPU, but you can often choose different values.

We will run an experiment to train our network with different minibatch sizes: * Minibatch size: 4, 16, 32, 64

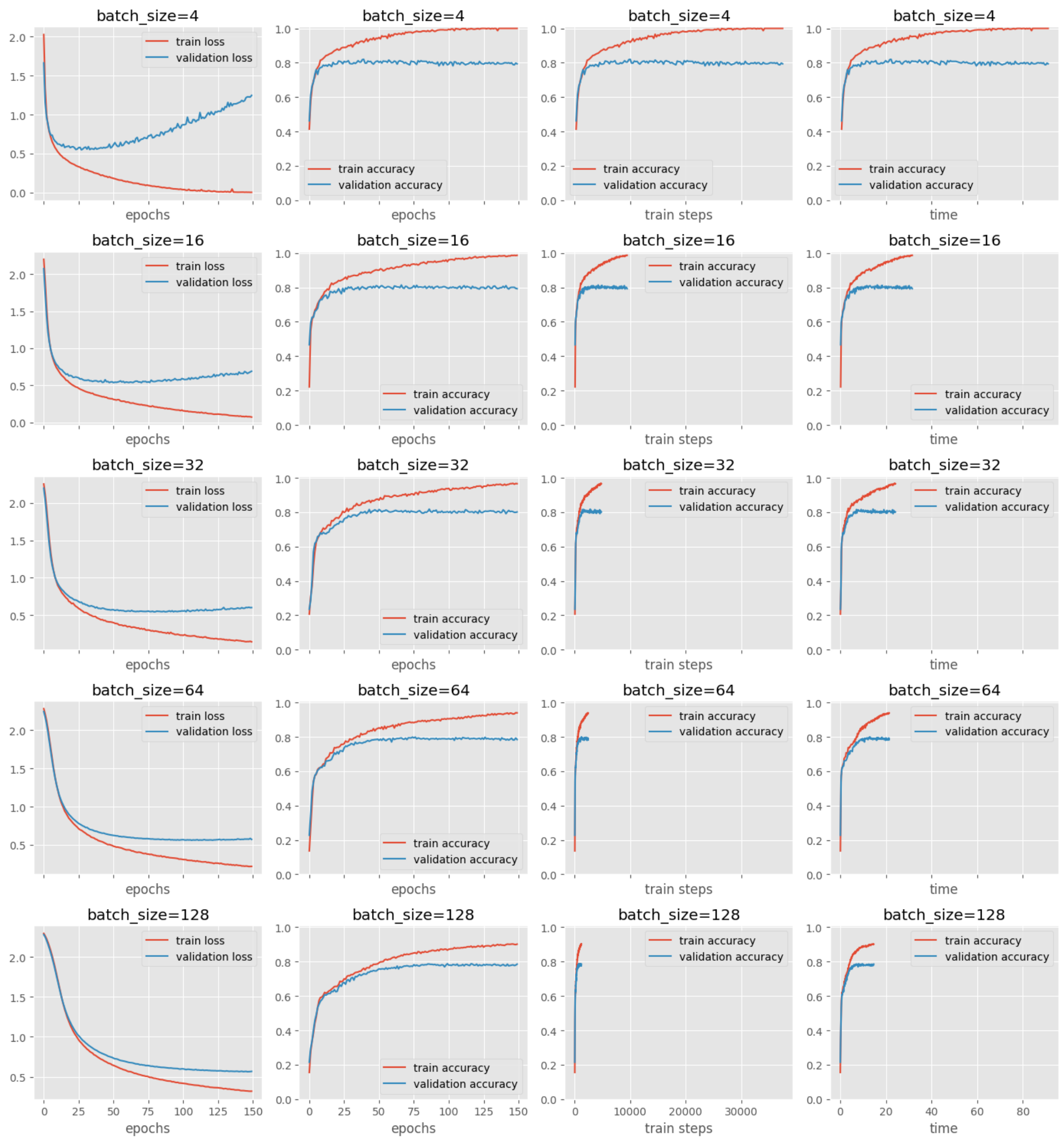
We will fix the other hyperparameters to values that worked well in the previous experiment: * Optimizer: Adam * Learning rate: 0.0001 * 150 epochs

For each setting, we will plot: * The train and validation accuracy vs number of epochs * The train and validation loss vs number of epochs * The train and validation accuracy vs the number of gradient descent update steps * The train and validation accuracy vs the training time

We will also print a table with the results of the final epoch.

(h) Run the experiment and have a look at the results.

[28]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation␣
 ↪loss']},
                                {'x': 'epochs', 'y': ['train accuracy',␣
 ↪'validation accuracy']},
                                {'x': 'train steps', 'y': ['train accuracy',␣
 ↪'validation accuracy']},
                                {'x': 'time', 'y': ['train accuracy',␣
 ↪'validation accuracy']}],
                         table=['train accuracy', 'validation accuracy',␣
 ↪'time'],
                         rows=5, cols=1)

epochs = 150
lr = 0.0001
batch_sizes = [4, 16, 32, 64, 128]

for row, batch_size in enumerate(batch_sizes):
    net = build_net()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
 ↪epochs=epochs, batch_size=batch_size)
    plotter.add('batch_size=%s' % str(batch_size), history, row=row, col=0)

plotter.done()
```



```
        experiment  train accuracy  validation accuracy       time
0     batch_size=4        1.000000             0.794000  91.083919
1    batch_size=16        0.987103             0.792969  31.599946
2    batch_size=32        0.965820             0.801172  24.201279
3    batch_size=64        0.940039             0.784706  21.442080
4   batch_size=128        0.901668             0.785358  14.721876
```

(i) Why is it useful to look at the number of training steps and the training time? How is this visible in the plots? (1 point)

It is useful for not wasting resources, and we can see that in the same time, we get much better solution if we use a bigger batch size, or a comparable accuracy in much less time. We can see this in the right plots, as the graphs get much shorter, i.e., do span the whole x-axis, while producing comparable accuracies.

(j) What are the effects of making the minibatches very small? (1 point)

It takes longer, and the accuracy curves are a bit more noisy.

(k) What are the effects of making the minibatches very large? (1 point)

In these experiments, the time needed gets smaller and the curves are smoother. Generally, a bigger batch size might lead to slower training performance.

1.7 2.6 Regularization (13 points)

Besides choosing the hyperparameters, we can include other components to improve the training of the model.

In this section, we will experiment with batch normalization, weight decay, and data augmentation.

1.7.1 Batch normalization

Batch normalization can be implemented with the batch normalization modules from `torch.nn` (documentation).

For a network with 1D feature vectors, you can use `torch.nn.BatchNorm1d` (documentation).

(a) Construct a network with batch normalization: (1 point)

Use the same structure as before, but include batchnorm after the hidden linear layers. So we have: * A linear layer from 784 to 128 features, followed by batchnorm and a ReLU activation. * A linear layer from 128 to 64 features, followd by batchnorm and ReLU activation. * A final linear layer from 64 features to 10 outputs, no activation.

[29]:
```
def build_net_bn():
    return torch.nn.Sequential(
        torch.nn.Linear(784, 128),
        torch.nn.BatchNorm1d(128),
        torch.nn.ReLU(),
        torch.nn.Linear(128, 64),
        torch.nn.BatchNorm1d(64),
        torch.nn.ReLU(),
        torch.nn.Linear(64, 10)
    )


net = build_net_bn()
print(net)
```

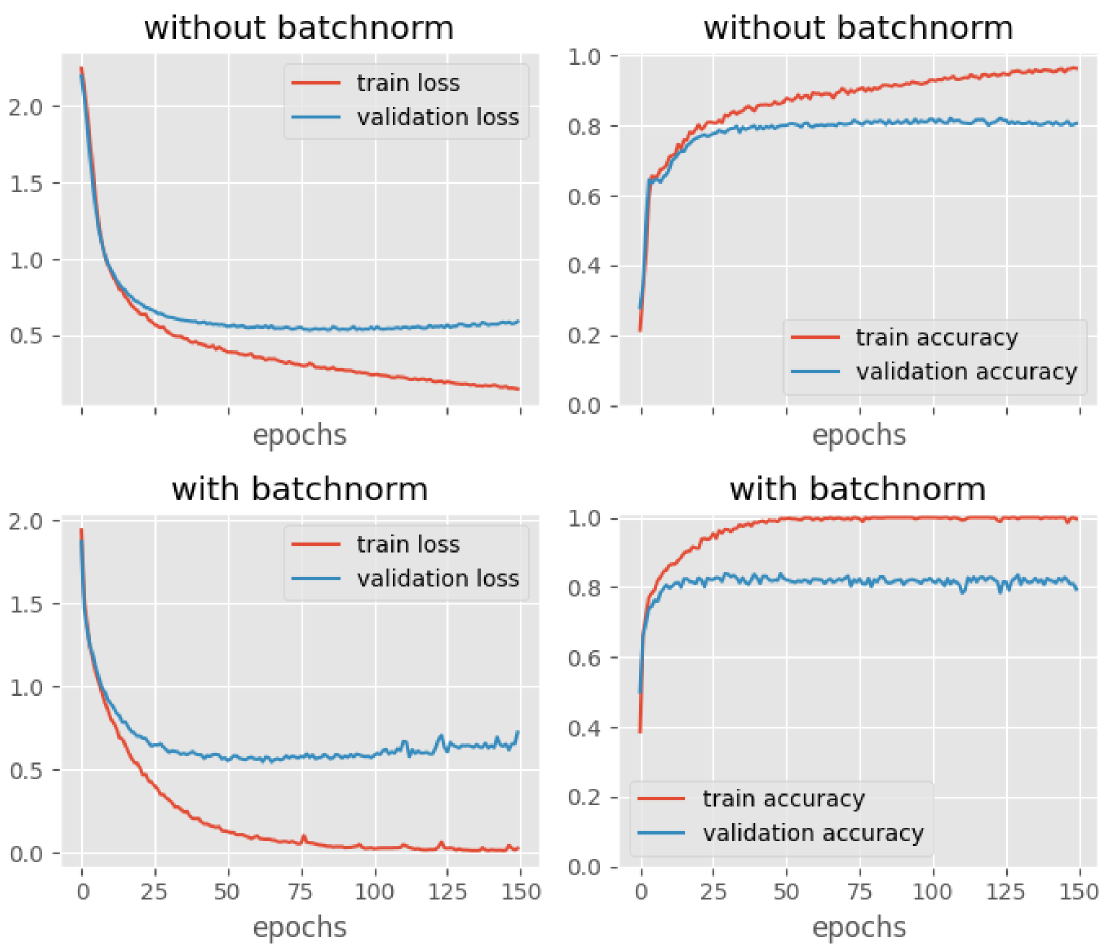
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (2): ReLU()
  (3): Linear(in_features=128, out_features=64, bias=True)
  (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (5): ReLU()
  (6): Linear(in_features=64, out_features=10, bias=True)
)
```

We will run an experiment to compare a network without batch normalization with a network with batch normalization.

We will fix the other hyperparameters to values that worked well in the previous experiment: * Optimizer: Adam * Learning rate: 0.0001 * Minibatch size: 32 * 150 epochs

(b) Run the experiment and have a look at the results.

[30]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation␣
 ↪loss']},
                                {'x': 'epochs', 'y': ['train accuracy',␣
 ↪'validation accuracy']}, ],
                         table=['train accuracy', 'validation accuracy'],
                         rows=2, cols=1)

epochs = 150
lr = 0.0001
batch_size = 32

networks = [
    ('without batchnorm', build_net),
    ('with batchnorm', build_net_bn)
]
histories = {}
for row, (network_name, network_fn) in enumerate(networks):
    net = network_fn()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
 ↪epochs=epochs, batch_size=batch_size)
    plotter.add(network_name, history, row=row, col=0)
    histories[network_name] = history

plotter.done()
```



```
          experiment  train accuracy  validation accuracy
0  without batchnorm        0.963867             0.806641
1     with batchnorm        0.996094             0.794531
```

(c) Does batch normalization improve the performance of the network? (1 point)

Yes, it does. The validation accuracy goes up about 3% points, and reaches a higher level of accuracy faster than without normalization.

(d) Does batch normalization affect the training or convergence speed? (1 point)

Yes, the training and validation accuracy are converging faster when using batch normalization. With batch normalization, the validation accuracy converges after about ten epochs and the train accuracy after about 50 epochs. Without batch normalization, the validation accuracy converges after about 27 epochs and the train accuracy does not converge within the 150 epochs tested.

Let us look a bit closer at how batch normalization changes the network.

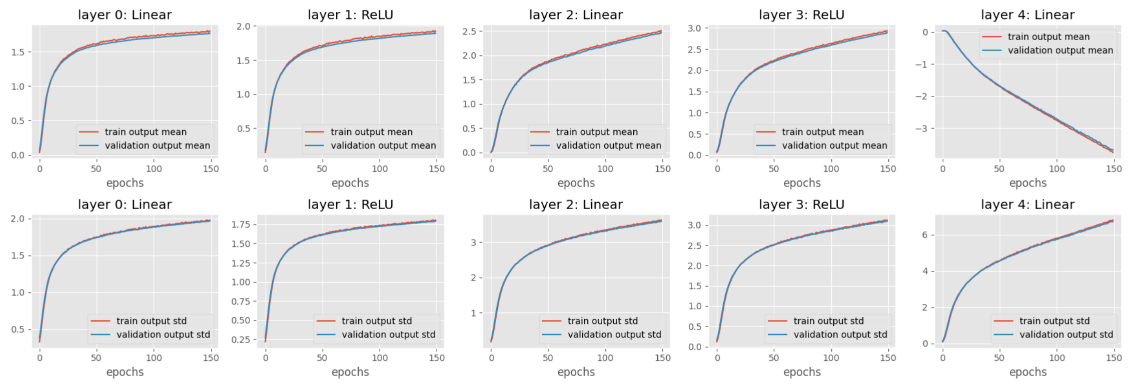
We will plot some statistics about the values inside the network.

[31]:
```
def plot_layer_stats(layers, history):
    fig, axs = plt.subplots(ncols=layers, nrows=2,
                            figsize=(3.5 * layers, 3 * 2))

    for layer in range(layers):
        i = 0
        for stat in ('mean', 'std'):
            for phase in ('train', 'validation'):
                keys = [key for key in history.keys()
                        if '%s %d:' % (phase, layer) in key and 'output %s' %␣
↪stat in key]
                if len(keys) == 1:
                    key = keys[0]
                    ax = axs[i, layer]
                    ax.plot(history['epochs'], history[key], label='%s output␣
↪%s' % (phase, stat))
                    ax.set_xlabel('epochs')
                    ax.legend()
                    ax.set_title(key.replace(' output %s' % stat, '').
↪replace(phase, 'layer'))
                i += 1

    plt.tight_layout()
```

First, we will plot the statistics of the network without batch normalization.

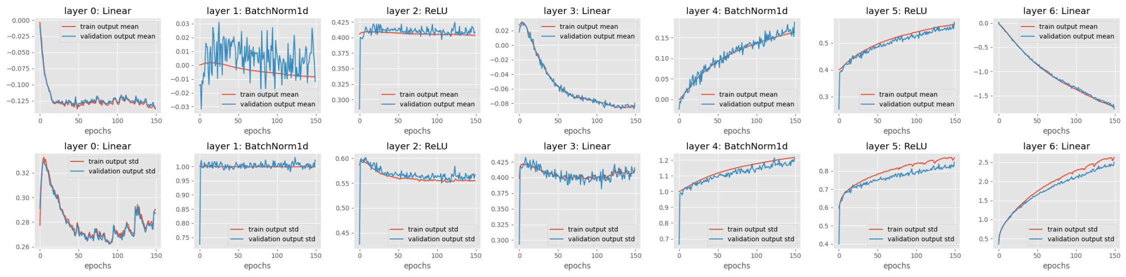
For each layer, the plots show the mean and standard deviation of the output values of that layer:

[32]:
```
plot_layer_stats(5, histories['without batchnorm'])
```



We make similar plots for the network with batch normalization: (Note that the number of layers is slightly larger.)

[33]:
```
plot_layer_stats(7, histories['with batchnorm'])
```



(e) Compare the mean training values in the batch normalization network with those in the network without batch normalization. Can you explain this with how batch normalization works? (1 point)

In the network without normalization, we can see that the mean and standard deviation continuously grow, except for the last layer. On the other hand, in the network with batch normalization, we can observe that layer 1 and 4, those are the normalization layers, try to smooth the train output to a mean of 0 and a standard deviation of 1. As a consequence, the validation output is also in the same area. This also leads to a smaller change in the mean and std deviation of the other layers.

(f) Compare the train and validation curves for the batch normalization layers. The training curves are smooth, but the validation curve is noisy. Why does this happen? (1 point)

This is because the training happens in smoothed batches, while the validation takes place on single, non-smoothed values.

(g) Batch normalization is supposed to normalize the values to $\mu = 0$ and $\sigma = 1$, but in layer 4, the mean and standard deviation are steadily increasing over time. Why and how does this happen? (1 point)

This is because the normalization layer has two values β and γ, which are learnable, i.e., adopted during the backpropagation. This values put an offset to the mean (β) and scale the std deviation (γ).

1.7.2 Weight decay

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

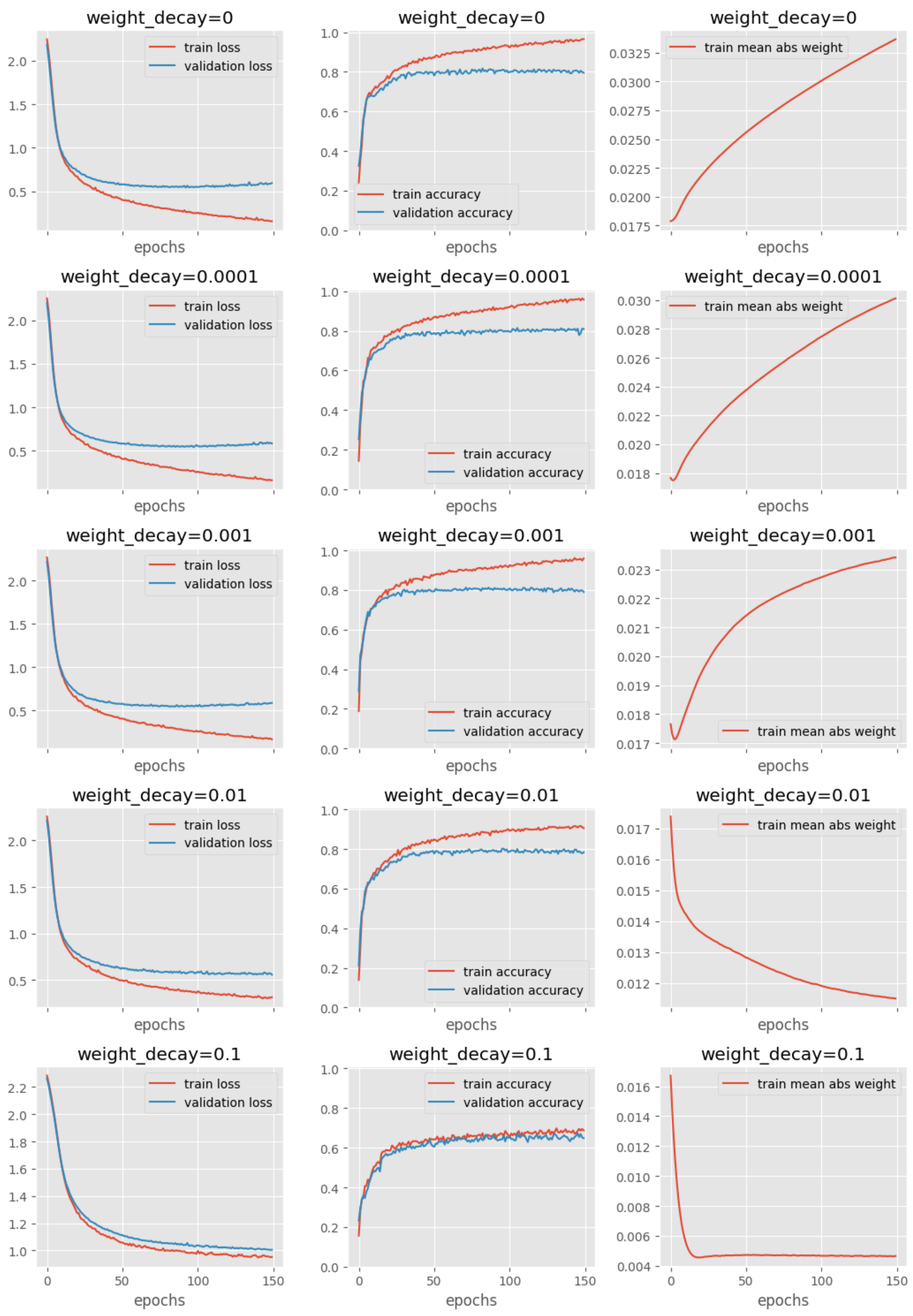
We will set up an experiment to investigate how this affects the training of the model.

We use the good settings from before: * 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.

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

[34]:
```
plotter = 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'],
                         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()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr,␣
↪weight_decay=weight_decay)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
↪epochs=epochs, batch_size=batch_size)
    plotter.add('weight_decay=%s' % str(weight_decay), history, row=row, col=0)

plotter.done()
```


weight_decay=0
train loss
validation loss
train accuracy
validation accuracy
train mean abs weight
weight_decay=0.0001
weight_decay=0.001
weight_decay=0.01
weight_decay=0.1
epochs

```
           experiment  train accuracy  validation accuracy  \
0      weight_decay=0        0.964844             0.795312
1 weight_decay=0.0001        0.958008             0.809766
2  weight_decay=0.001        0.958984             0.790234
3   weight_decay=0.01        0.905273             0.783594
4    weight_decay=0.1        0.687500             0.648438

   train mean abs weight
0               0.033657
1               0.030113
2               0.023411
3               0.011496
4               0.004659
```

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

If the validation accuracy goes down, while the training accuracy continues growing, it would mean that we observe overfitting. Another version of overfitting is the difference between the training and validation loss. In the data given, we cannot see any overfitting in the accuracy, but we can observe a slight overfitting in the loss.

(j) 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)

For a weight decay of 0.0001, 0.001, and 0.01, the weight decay reduces the difference in the training and validation loss with only a small harm in the validation accuracy. If the weight decay is too big, i.e., 0.1 in this case, the difference between the training and validation loss gets even smaller, but the accuracy of the network goes down dramatically.

1.7.3 Data augmentation

Finally, we will look at data augmentation.

We will run experiments with three types of data augmentation: * Adding random noise to the pixels, taken from a normal distribution $\mathcal{N}(0, \sigma)$ with $\sigma = 0$, 0.01, 0.1, or 0.2. * Flipping the image horizontally. * Shifting the image up, down, left, or right by one pixel.

We create a new dataset class that generates noisy images and use this instead of our normal training set.

[35]:
```python
class NoisyDataset(torch.utils.data.Dataset):
    def __init__(self, ds, noise_sigma=0, flip_horizontal=False, shift=False):
        self.ds = ds
        self.noise_sigma = noise_sigma
        self.flip_horizontal = flip_horizontal
        self.shift = shift

    def __len__(self):
        return len(self.ds)

    def __getitem__(self, idx):
        x, y = self.ds[idx]
        # add random noise
        x = x + self.noise_sigma * torch.randn(x.shape)
        # flip the pixels horizontally with probability 0.5
        if self.flip_horizontal and torch.rand(1) > 0.5:
            x = torch.flip(x.reshape(28, 28), dims=(1,)).flatten()
        # shift the image by one pixel in the horizontal or vertical directions
        if self.shift:
            x = x.reshape(28, 28)
            # shift max one pixel
            shifts = [*torch.randint(-1, 2, (2,)).numpy()]
            x = torch.roll(x, shifts=shifts, dims=(0, 1))
            x = x.flatten()
        return x, y
```

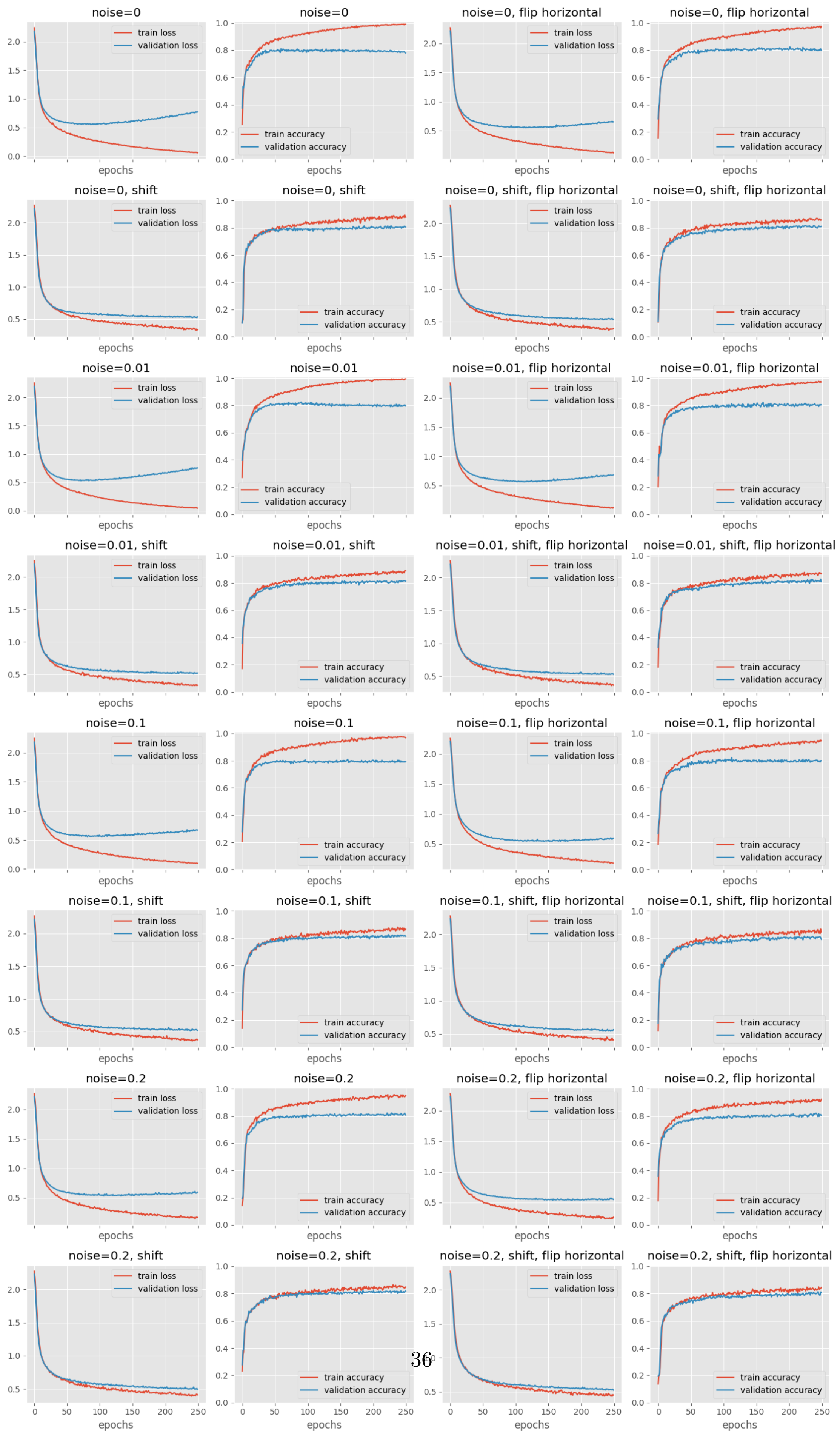
We set up an experiment to see if data augmentation improves our results. We use combinations of the three augmentations: noise, flipping, and shifting.

We will train for 250 epochs.

We keep the other settings as before: * Optimizer: Adam * Learning rate: 0.0001 * Minibatch size: 32

(k) Run the experiment and have a look at the results.

```
[36]: plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation␣
       ↪loss']},
                                     {'x': 'epochs', 'y': ['train accuracy',␣
       ↪'validation accuracy']}, ],
                               table=['train accuracy', 'validation accuracy',␣
       ↪'time'],
                               rows=8, cols=2)

      epochs = 250
      lr = 0.0001
      batch_size = 32

      for row, noise_sigma in enumerate((0, 0.01, 0.1, 0.2)):
          for row2, shift in enumerate([False, True]):
              for col, flip_horizontal in enumerate([False, True]):
                  noisy_fashion_train = NoisyDataset(fashion_train, noise_sigma,␣
       ↪flip_horizontal=flip_horizontal, shift=shift)
                  net = build_net()
                  optimizer = torch.optim.Adam(net.parameters(), lr=lr)
                  history = fit(net, noisy_fashion_train, fashion_validation,␣
       ↪optimizer=optimizer, epochs=epochs,
                                batch_size=batch_size)
                  label = ('noise=%s' % str(noise_sigma)) + \
                          (', shift' if shift else '') + \
                          (', flip horizontal' if flip_horizontal else '')
                  plotter.add(label, history, row=row * 2 + row2, col=col)

      plotter.done()
```


noise=0
noise=0
noise=0, flip horizontal
noise=0, flip horizontal
noise=0, shift
noise=0, shift
noise=0, shift, flip horizontal
noise=0, shift, flip horizontal
noise=0.01
noise=0.01
noise=0.01, flip horizontal
noise=0.01, flip horizontal
noise=0.01, shift
noise=0.01, shift
noise=0.01, shift, flip horizontal
noise=0.01, shift, flip horizontal
noise=0.1
noise=0.1
noise=0.1, flip horizontal
noise=0.1, flip horizontal
noise=0.1, shift
noise=0.1, shift
noise=0.1, shift, flip horizontal
noise=0.1, shift, flip horizontal
noise=0.2
noise=0.2
noise=0.2, flip horizontal
noise=0.2, flip horizontal
noise=0.2, shift
noise=0.2, shift
noise=0.2, shift, flip horizontal
noise=0.2, shift, flip horizontal
train loss
validation loss
train accuracy
validation accuracy
epochs

```
                              experiment  train accuracy  validation accuracy  \
0                                noise=0        0.990234             0.779297   
1              noise=0, flip horizontal        0.972656             0.800000   
2                         noise=0, shift        0.877930             0.805469   
3       noise=0, shift, flip horizontal        0.858398             0.810156   
4                             noise=0.01        0.994141             0.796094   
5           noise=0.01, flip horizontal        0.971680             0.805078   
6                      noise=0.01, shift        0.889648             0.814063   
7    noise=0.01, shift, flip horizontal        0.869141             0.810156   
8                              noise=0.1        0.968750             0.792188   
9            noise=0.1, flip horizontal        0.946289             0.799219   
10                      noise=0.1, shift        0.867188             0.815234   
11    noise=0.1, shift, flip horizontal        0.845703             0.792188   
12                             noise=0.2        0.948242             0.807031   
13           noise=0.2, flip horizontal        0.921875             0.804297   
14                      noise=0.2, shift        0.847656             0.817969   
15    noise=0.2, shift, flip horizontal        0.844727             0.807031   

         time  
0   46.474891  
1   58.374272  
2   57.782119  
3   68.838805  
4   49.611206  
5   58.821444  
6   50.261813  
7   52.544060  
8   71.146262  
9   53.677801  
10  68.330370  
11  60.000288  
12  41.402164  
13  48.999198  
14  54.555535  
15  56.659636  
```

(l) How does data augmentation affect overfitting in the above experiment? Discuss each of the augmentation types. (3 points)

Adding noise: Noise has only a limited effect on the overfitting prevention, but it does have a small effect on generalization, i.e., the validation loss gets closer to training loss.

Horizontal flips: Horizontal flips seem to affect the overfitting as well, but only with a small effect.

Shifting: Shifting has by far the biggest effect on overfitting prevention in this experiment and reduces the difference between the validation and training curves in both, the accuracy and loss.

Combining all the methods seems to lead to the best results in regard to overfitting and even improves the validation accuracy.

(m) Why do we have to train the networks with data augmentation a bit longer than networks without data augmentation? (1 points)

Because modifying the data takes time as well.

1.8 2.7 Network architecture (5 points)

An often overlooked hyperparameter is the architecture of the neural network itself. Here you can think about the width (the size of each hidden layer) or the depth (the number of layers).

(a) Copy the `build_net` function from 2.3b and change it to take a parameter for the width of the first hidden layer. (1 point)

The second hidden layer should have half that width, so the network we have been using so far has `width=128`.

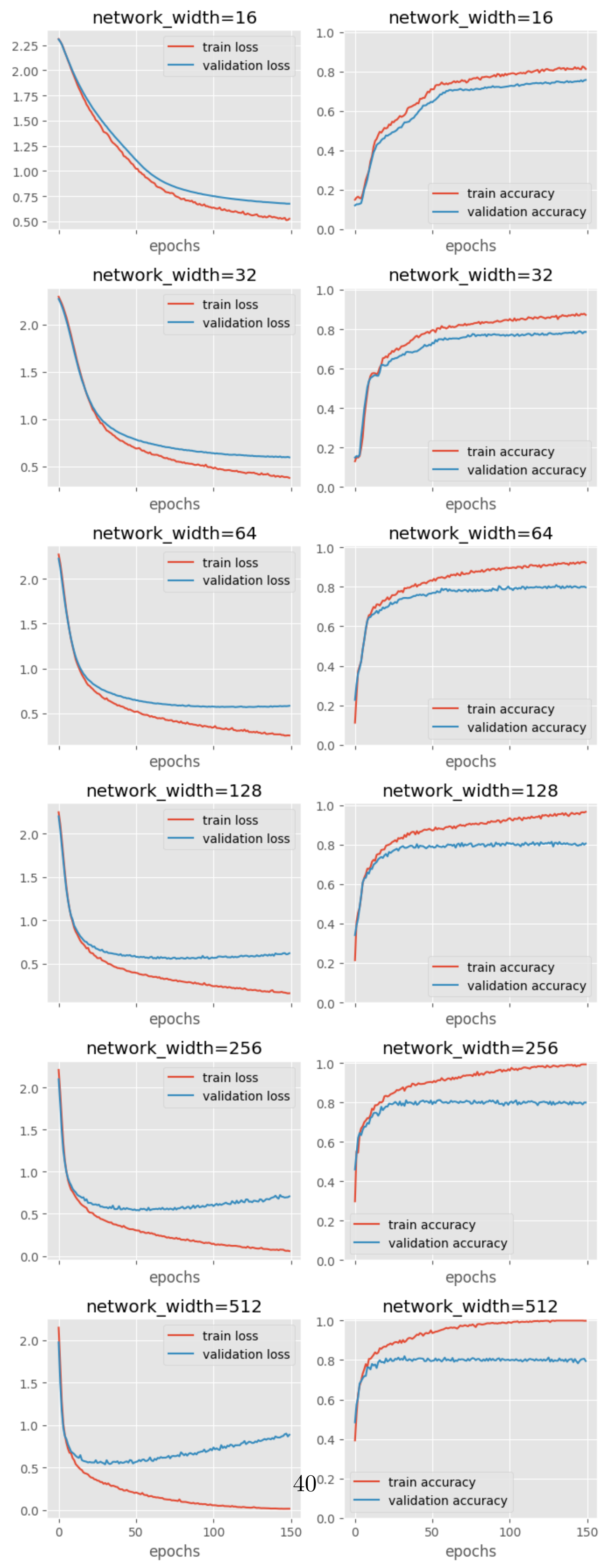
Hint: `a // b` is the Python notation for integer division rounding down.

[37]:
```
def build_net(width=128):
    return torch.nn.Sequential(
        torch.nn.Linear(784, width),
        torch.nn.ReLU(),
        torch.nn.Linear(width, width // 2),
        torch.nn.ReLU(),
        torch.nn.Linear(width // 2, 10)
    )
```

(b) Set up an experiment to see how the size of the network affects our results. (1 point)

We keep the other settings as before: * Optimizer: Adam * Epochs: 150 * Learning rate: 0.0001 * Minibatch size: 32 * Widths: 16, 32, 64, 128, 256, 512

[38]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'validation␣
 ↪loss']},
                                {'x': 'epochs', 'y': ['train accuracy',␣
 ↪'validation accuracy']}, ],
                         table=['train accuracy', 'validation accuracy',␣
 ↪'time'],
                         rows=6, cols=1)

epochs = 150
lr = 0.0001
batch_size = 32
network_widths = [16, 32, 64, 128, 256, 512]

for row, width in enumerate(network_widths):
    net = build_net(width)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    history = fit(net, fashion_train, fashion_validation, optimizer=optimizer,␣
 ↪epochs=epochs, batch_size=batch_size)
    plotter.add('network_width=%s' % str(width), history, row=row, col=0)

plotter.done()
```


network_width=16
train loss
validation loss
train accuracy
validation accuracy
network_width=32
network_width=64
network_width=128
network_width=256
network_width=512
epochs
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
0.5
1.0
1.5
2.0
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25

```
           experiment  train accuracy  validation accuracy       time
0    network_width=16        0.812500             0.756641  21.915163
1    network_width=32        0.872070             0.785547  23.936207
2    network_width=64        0.921875             0.797266  22.508843
3   network_width=128        0.965820             0.805078  32.240287
4   network_width=256        0.992188             0.798828  31.415512
5   network_width=512        0.999023             0.794922  32.723022
```

(c) For what network sizes do you observe underfitting? (1 point)

The 16-neuron network is underfitting, which we can see in a not as good validation accuracy. The 32-neuron network does slightly underfit as well.

(d) Do you see overfitting for the largest networks? How can you see this from the plots? (1 point)

The 256-neuron and 512-neuron networks are clearly overfitting, which we can see at a decreasing training loss while the validation loss goes up a lot. The 128-neuron network seems to slightly overfit as well.

(e) How many parameters are there in a network with width 128? How does that compare to the number of training samples? (1 point)

You don't have to give an exact value, as long as you are in the right order of magnitude, it is okay.

(Feel free to write some python code to do computations.)

[39]:
```python
def network_size(size):
    weights_layer1 = 784 * size
    biases_layer1 = size
    weights_layer2 = size * (size // 2)
    biases_layer2 = size // 2
    weights_layer3 = (size // 2) * 10
    biases_layer3 = 10
    return weights_layer1 + biases_layer1 + weights_layer2 + biases_layer2 +␣
 ↪weights_layer3 + biases_layer3

print("Network params", network_size(128))

print("Data set", len(fashionmnist))

print("Network params / Data set size", network_size(128) / len(fashionmnist))
```

```
Network params 109386
Data set 60000
Network params / Data set size 1.8231
```

The network, with a width of 128, has about 110,000 parameters, while we have only 60,000 samples. This means that we have around 1.8 as many parameters than examples, of which we use only a part for the training and another part only for validation.

1.9 2.8 Discussion (3 points)

(a) Several of the experiments have included a baseline with exactly the same hyperparameters (batch_size=32, weight_decay=0, network_size=128). Are the results exactly the same? What does this tell you about comparing results for picking the best hyperparameters? (2 points)

No, they are not exactly the same, as there is a lot of randomness included. For example, the weight initialization for the linear layers, and the shuffling of the training data. This means that a slight change in the network performance might not be due to a change in the hyperparameters, but just because of the randomness. Therefore, only significant changes in the network performance are relevant to choosing the hyperparameters.

(b) Throughout this assignment we have used a validation set of 500 samples for selecting hyperparameters. Do you think that you will see the same results on an independent test set? Would the best results be obtained with the hyperparameters that are optimal on the validation set? (1 point)

The results of the independent test set probably differ a bit from the validation set. Similarly, the best hyperparameters for the validation set might not be the same as for the independent test set. Still, we do not want to optimize our model for the test set, as it would bias the prediction on how good the network performs on real-world data.

1.10 The end

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

This assignment has 44 points. Version d4d27c6 / 2023-09-08