Deep Learning — Assignment 2

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

Names: Andrew Schroeder and Fynn Gerding

Group: 17

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

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`.

In [ ]:
```
# !pip install torch
# !pip install torchvision
# !pip install pandas
```

In [20]:
```
%config InlineBackend.figure_formats = ['png']
%matplotlib inline
#import numpy as np
import matplotlib.pyplot as plt
import torch
import time
import torchvision
import tqdm.notebook as tqdm
import collections
import IPython
import pandas as pd
import numpy as np

#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);
```

2.1 Implementing a model with PyTorch

In the first assignment, you already worked with PyTorch tensors. Here we will show you some more ways to use them.

Shape and data type

PyTorch tensors are multi-dimensional arrays. So, they can be scalars, vectors, matrices, or have an even higher dimension. You can see that by looking at their shape:

In [2]:
```
# create a 10x10 matrix filled with zeros
x = torch.zeros([10, 10])
print(x)
# The shape shows it is a rank 2 tensor:
print(x.shape)
```

```
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.]])
torch.Size([10, 10])
```

Tensors can store different types of data. The default is 32-bit floating point numbers. The `dtype` property tells you the data type. You can also make tensors with 64-bit floating point numbers, or various types of integers. (See the PyTorch documentation for a complete list.)

In [3]:
```
print(x.dtype)
```

```
torch.float32
```

In [4]:
```
# Some tensors with a different datatype
print((x == x).dtype)
print(torch.ones(5, dtype=torch.int).dtype)
```

```
torch.bool
torch.int32
```

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.

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')` (or `.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'`.

In [5]:
```
# 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='mps: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.

(a) If you have a GPU, make the following code run without errors. (no points)

In [7]:
```
if torch.backends.mps.is_available():
    x = torch.tensor([1, 2, 3])
    y = torch.tensor([1, 2, 3], device='mps')
    print('x is on the CPU:', x)
    print('y is on the GPU:', y)

    # this will not work
    if x.device != y.device:
        x = x.to(y.device)
    z = x * y
    print(z)
```

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

In [8]:
```
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()
print("Running on", device)
```

```
Running on mps
```

In [47]:
```
device = 'cpu'
```

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:

In [9]:
```
x = torch.tensor(5.)
x_in_numpy = x.detach().cpu().numpy()
x_in_numpy
```

Out[9]:
```
array(5., dtype=float32)
```

Here:

- `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. Numpy arrays are very similar to torch tensors, they just come from different libraries.

Computing the gradients of a simple model

We saw last week how to compute gradients. Here is a quick recap:

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:

In [10]:
```
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):

In [11]:
```
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)
```

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:

In [12]:
```
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 0x1068cb9d0>
  \--> ((<MulBackward0 object at 0x1068cbcd0>, 0), (<AccumulateGrad objec
t at 0x1068cb610>, 0))
          \--> ((None, 0), (<AccumulateGrad object at 0x1068c99f0>, 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.

2.2 PyTorch neural network with torch.nn (2 points)

The `torch.nn` module of PyTorch contains a large number of building blocks to construct your own neural network architectures. You will need them 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.

A network module

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

In [15]:
```
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`).
- The layers can be used as if they are functions, for example `self.layer1(x)`. This call will compute the output of the layer given input `x`.
  - You can also call your own network with `net(x)`, internally this will call `net.forward(x)`.
- The `Linear` layer is separate from the activation function, which is done in its own `ReLU` layer.

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:

In [16]:
```
for name, param in net.named_parameters():
    print(name)
    print(param)
    print()
```

```
layer1.weight
Parameter containing:
tensor([[ 0.1232, -0.0560, -0.1089,  ..., -0.0091,  0.0381,  0.0213],
        [-0.0875,  0.0952, -0.0869,  ...,  0.1211, -0.0886,  0.1106],
        [-0.0615, -0.0877,  0.1206,  ...,  0.1179,  0.0436, -0.0989],
        ...,
        [ 0.0734, -0.0998, -0.0433,  ...,  0.0026,  0.0682,  0.0226],
        [ 0.1064,  0.0168, -0.0538,  ...,  0.1106, -0.1021,  0.0957],
        [ 0.0901,  0.0212, -0.0865,  ...,  0.0339,  0.0584,  0.1123]],
       requires_grad=True)

layer1.bias
Parameter containing:
tensor([-0.1017, -0.0645,  0.1144, -0.0167,  0.0711,  0.0685, -0.0711,
0.0765,
        -0.0754, -0.1128,  0.0924, -0.1040,  0.0770, -0.0163, -0.0266,
0.0544,
        -0.0383,  0.1223,  0.0233, -0.0711,  0.0213,  0.0180, -0.0486,
0.1153,
         0.0123, -0.0944, -0.0676,  0.0980, -0.1179, -0.0537,  0.0168, -
0.0573],
       requires_grad=True)

layer2.weight
Parameter containing:
```

```
tensor([[ 0.0443, -0.1104, -0.0371,  0.0976,  0.1535, -0.0890,  0.0691,
0.0154,
          0.1758, -0.0601, -0.1333, -0.0478,  0.0797, -0.0251, -0.1294,
0.1405,
         -0.1026,  0.0580, -0.0351,  0.0858, -0.0655,  0.0851,  0.1459,
0.0442,
         -0.1299,  0.1727,  0.0966,  0.0753,  0.1762, -0.0617, -0.0453,
0.0031],
        [ 0.0561, -0.0167, -0.0052, -0.0981,  0.0751,  0.1341,  0.0486,
0.0879,
         -0.0758, -0.0066,  0.0942, -0.0502, -0.1695, -0.1097,  0.0023, -
0.0109,
         -0.0170,  0.0008,  0.0732,  0.0444, -0.1165,  0.0413, -0.0039, -
0.1623,
         -0.1657,  0.1720, -0.1184,  0.1318, -0.1375, -0.0771, -0.0604,
0.0814],
        [ 0.1379, -0.0160,  0.1175, -0.0937,  0.1262, -0.1299,  0.1146,
0.0822,
          0.0805, -0.0464,  0.1356, -0.0040,  0.1374,  0.0014,  0.0545,
0.1549,
         -0.1615, -0.0964, -0.0124, -0.1038, -0.0083, -0.0452,  0.0775, -
0.1219,
         -0.1184,  0.0099, -0.1320,  0.0891, -0.0954, -0.0925, -0.1310, -
0.0326],
        [ 0.0696,  0.1375, -0.0364, -0.1290, -0.1032,  0.1626, -0.0394,
0.0887,
         -0.1298, -0.1715, -0.1287,  0.0306, -0.1286,  0.1461, -0.0132,
0.1650,
         -0.1274,  0.0347, -0.1385, -0.0582, -0.1321, -0.0444,  0.1465,
0.1414,
         -0.1678, -0.0019, -0.0063,  0.1035,  0.1031,  0.0660,  0.1358,
0.0598],
        [ 0.0294,  0.0020,  0.0675,  0.0957, -0.0175, -0.0994, -0.1231, -
0.0957,
          0.1473, -0.0657, -0.0981, -0.0321,  0.0513, -0.0958, -0.0362, -
0.0520,
          0.0386,  0.0849,  0.1092, -0.0867,  0.1096, -0.0564,  0.0022,
0.0441,
          0.1318, -0.1650, -0.0883, -0.0101,  0.0045, -0.0991, -0.0220, -
0.1732],
        [-0.0033, -0.1670,  0.1600,  0.0884,  0.0866,  0.0535,  0.0729,
0.1229,
          0.0711, -0.0305, -0.0662,  0.1476, -0.0093,  0.0078,  0.1644, -
0.1293,
          0.0833,  0.0884, -0.0645, -0.0564,  0.0708, -0.1181,  0.1246,
0.1642,
         -0.1672, -0.1687,  0.1664,  0.0390,  0.0756, -0.0899, -0.0298,
0.0158],
        [-0.0190, -0.0479,  0.0737, -0.0297, -0.0562,  0.0168, -0.0159, -
0.0489,
         -0.1731,  0.1581,  0.1019, -0.1129,  0.0661,  0.1464, -0.1572, -
0.0180,
          0.1740,  0.0173, -0.0010,  0.0560, -0.0674, -0.1289, -0.1636,
0.1343,
         -0.0097, -0.0875, -0.0497,  0.1275, -0.1620, -0.0084, -0.0025, -
0.0555],
        [ 0.0826,  0.1049,  0.1249, -0.1443,  0.1178,  0.0258, -0.1212, -
```

```
0.0414,
          0.0938,  0.0991, -0.0461,  0.0112,  0.1560, -0.0640, -0.1752, -
0.0720,
          0.0858,  0.1142,  0.0664,  0.1008,  0.0985,  0.1336,  0.0478,
0.0025,
          0.1598, -0.1187,  0.0236,  0.0115,  0.0989,  0.0990, -0.1755, -
0.1379],
        [ 0.0309, -0.0242, -0.0955, -0.1266,  0.0911, -0.0532,  0.0925, -
0.1321,
          0.0408,  0.0186, -0.0338, -0.0723,  0.1222, -0.0907,  0.0596,
0.1336,
          0.0892,  0.0289, -0.0117,  0.0934,  0.0742, -0.0193, -0.1048,
0.0531,
          0.0973, -0.0328,  0.1358, -0.0803,  0.0626, -0.0882, -0.0427,
0.0820],
        [-0.0462, -0.1059, -0.0980,  0.0573,  0.0277,  0.0451, -0.1568,
0.1709,
          0.0787, -0.0696,  0.1498, -0.0135,  0.0844, -0.0603,  0.1693, -
0.0597,
         -0.0252,  0.0631,  0.0699, -0.0489,  0.0498,  0.0188,  0.0440, -
0.0803,
          0.0502,  0.0667, -0.1242, -0.0993,  0.0571, -0.1295,  0.0270, -
0.1158]],
       requires_grad=True)

layer2.bias
Parameter containing:
tensor([-0.0091,  0.1358, -0.1625, -0.1636,  0.0146,  0.1092,  0.1071, -
0.0300,
         0.1731, -0.1546], requires_grad=True)
```

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

(a) Why are these parameters not zero? **(1 point)**

Answer: The default initialisation of weights and biases within **PyTorch** is to use **He initialisation**. This is done to break the symmetry and allow each weight of the network to receive a unique update.

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 class `torch.nn.Sequential` 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:

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

Loss functions

You may have also noticed that there is no activation function (such as a sigmoid or softmax) at the end of the network. This is not a mistake. Instead we will be using a loss function that combines the sigmoid/softmax and the loss in one computation, because this is more numerically stable.

Have a look at `torch.nn.BCEWithLogitsLoss`, which implements binary cross entropy. The documentation mentions the "log-sum-exp trick", which refers to the fact that:

$$\log(\exp(a_1) + \exp(a_2)) = c + \log(\exp(a_1 - c) + \exp(a_2 - c))$$

Then we pick c to be $\max(a_1, a_2)$, so all arguments to $\exp$ are ≤ 1. And in fact this becomes

$$\log(\exp(a_1) + \exp(a_2)) = \max(a_1, a_2) + \log(1 + \exp(-|a_1 - a_2|)).$$

The multi-class equivalent of binary cross entropy loss is categorical cross entropy. In PyTorch this is implemented in `torch.nn.CrossEntropyLoss`

(b) Does `CrossEntropyLoss` expect probabilities or logits? In other words, should you apply a softmax activation function or not? (1 point)

Answer: The documentation for the `torch.nn.CrossEntropyLoss` loss states that the input should be specified as "unnormalized logits for each class". This works, because internally, the CrossEntropyLoss applies a softmax function to the input to convert it into a probability distribution over all the classes. Therefore, it is logically equivalent to "applying LogSoftmax on an input, followed by NLLLoss".

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

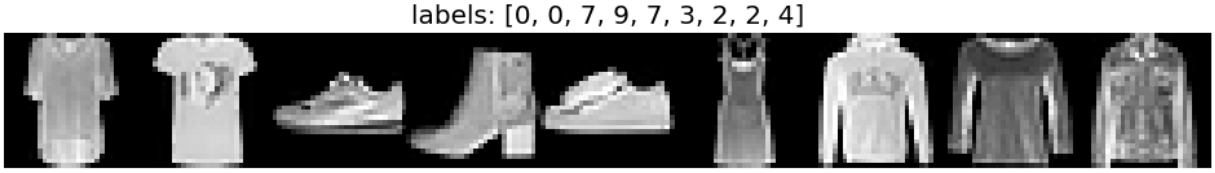
In this assignment, we will do experiments with the Fashion-MNIST dataset. First, we download the dataset, and create a random training set with 1000 images and a validation set with 500 images:

In [39]:
```
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)])
```

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.

In [40]:
```
# 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(
plt.grid(False)
plt.tight_layout()
plt.axis('off')
plt.title('labels: ' + str([fashion_train[i][1] for i in range(9)]));
```



(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? Briefly explain your answer. **(1 point)**

(Briefly means in at most 1 or 2 sentences)

Answer: In this limited sample, the items from the same class appear more similar than items from different classes. Generally, this is an indication that the neural network might be able to pick up on some features that accurately separate the classes.

Use the DataLoader to create batches

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

In [22]:
```
data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=10, s
```

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 labels for each image:

In [23]:
```
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])
```

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 become 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.

In [41]:
```
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)
    )
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)

In [38]:
```
# use a data loader to loop over fashion_train,
# with a batch size of 16, shuffle the dataset
# TODO: construct a DataLoader
data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=16, s

# TODO: construct your network
net = build_net()
for x, y in data_loader:
    print('x.shape:', x.shape)
    print('y.shape:', y.shape)
    # TODO: run your network to compute the output for x
    output = net(x)
    print('output.shape', output.shape)
    assert len(output.shape) == 2
    assert output.shape[0] == x.shape[0], "Expected an output for every i
    assert output.shape[1] == 10, "Expected logit for 10 classes"
    break
```

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

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

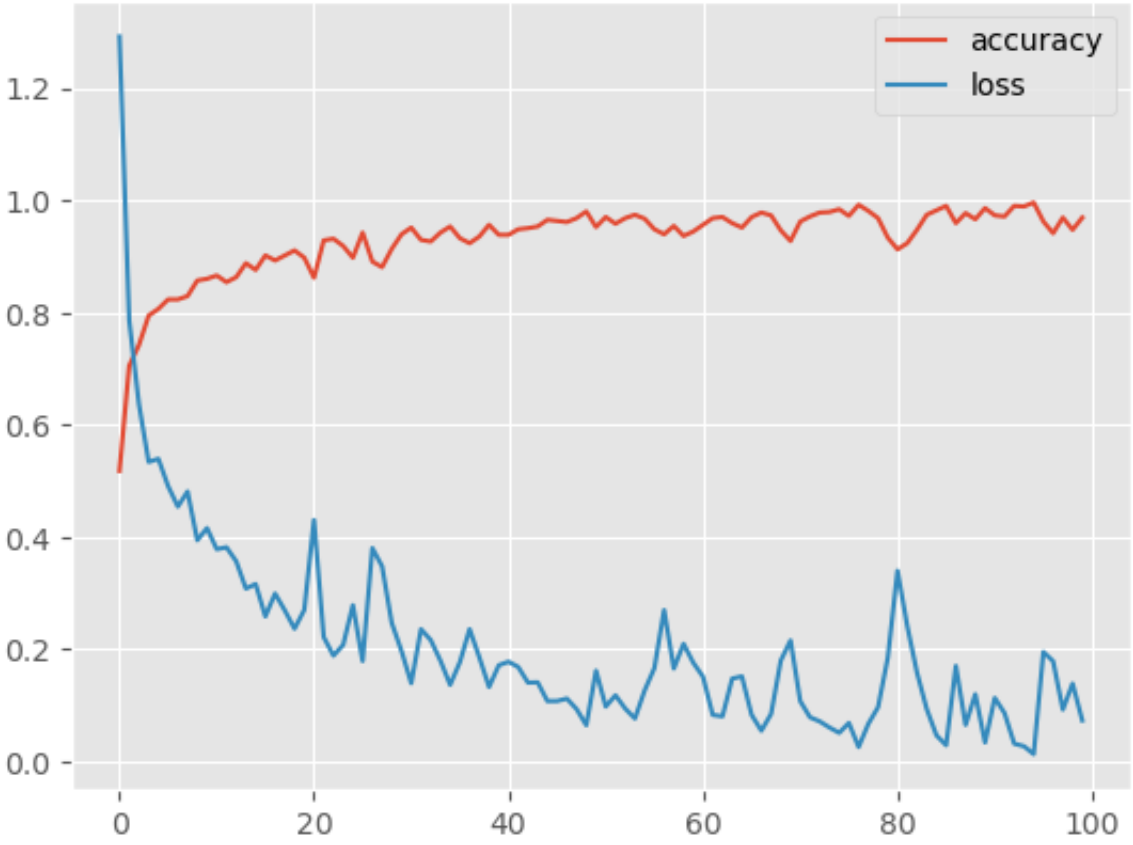
In [42]:
```
# initialize a new instance of the network
net = build_net()

# construct a data loader for the training set
data_loader = torch.utils.data.DataLoader(fashion_train, batch_size=16, s

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

# TODO: Initialize cross-entropy loss function
loss_function = torch.nn.CrossEntropyLoss()

accuracy_history = []
loss_history = []
# repeat for multiple epochs
for epoch in range(100):
    # 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
```

```
        # TODO: compute the output
        output = net(x)

        # compute the loss by comparing with the target output y
        # TODO: use loss_function to compute the loss
        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)
    accuracy_history.append((accuracy_sum / steps))
    loss_history.append((loss_sum / steps))

plt.plot(accuracy_history, label="accuracy")
plt.plot(loss_history, label="loss")
plt.legend()
plt.show()
```

```
epoch: 0 loss: 1.29191255569458 accuracy: 0.5178571428571429
epoch: 1 loss: 0.7861586865924653 accuracy: 0.7053571428571429
epoch: 2 loss: 0.6368071810593681 accuracy: 0.7420634920634921
epoch: 3 loss: 0.5337895010671918 accuracy: 0.7946428571428571
epoch: 4 loss: 0.5393253155643978 accuracy: 0.8065476190476191
epoch: 5 loss: 0.4906629676383639 accuracy: 0.8234126984126984
epoch: 6 loss: 0.4542100863560798 accuracy: 0.8234126984126984
epoch: 7 loss: 0.48108509756506435 accuracy: 0.8293650793650794
epoch: 8 loss: 0.3943061057537321 accuracy: 0.8571428571428571
epoch: 9 loss: 0.4157735971467836 accuracy: 0.8601190476190477
epoch: 10 loss: 0.37826387279681745 accuracy: 0.8660714285714286
epoch: 11 loss: 0.38163708959750475 accuracy: 0.8541666666666666
epoch: 12 loss: 0.3568176586949636 accuracy: 0.8630952380952381
epoch: 13 loss: 0.30808889703263365 accuracy: 0.8878968253968254
epoch: 14 loss: 0.3161555951727288 accuracy: 0.8759920634920635
epoch: 15 loss: 0.2584182714954728 accuracy: 0.9017857142857143
epoch: 16 loss: 0.299220166449982 accuracy: 0.8928571428571429
epoch: 17 loss: 0.2690963123644155 accuracy: 0.9017857142857143
epoch: 18 loss: 0.2366184533706733 accuracy: 0.9107142857142857
```

```
epoch: 19 loss: 0.27005832596489837 accuracy: 0.8978174603174603
epoch: 20 loss: 0.4302471785968731 accuracy: 0.8621031746031746
epoch: 21 loss: 0.2216489732501999 accuracy: 0.9285714285714286
epoch: 22 loss: 0.18864312804200584 accuracy: 0.9315476190476191
epoch: 23 loss: 0.2078077102464343 accuracy: 0.9186507936507936
epoch: 24 loss: 0.2785723843937001 accuracy: 0.8978174603174603
epoch: 25 loss: 0.17900723647621888 accuracy: 0.9424603174603174
epoch: 26 loss: 0.3802770184323428 accuracy: 0.8908730158730159
epoch: 27 loss: 0.34780472032134496 accuracy: 0.8809523809523809
epoch: 28 loss: 0.246608401826095 accuracy: 0.9136904761904762
epoch: 29 loss: 0.19675181365557134 accuracy: 0.939484126984127
epoch: 30 loss: 0.13940141248028903 accuracy: 0.9513888888888888
epoch: 31 loss: 0.23567328222155098 accuracy: 0.9295634920634921
epoch: 32 loss: 0.21646858407332312 accuracy: 0.9265873015873016
epoch: 33 loss: 0.17960255762879987 accuracy: 0.9424603174603174
epoch: 34 loss: 0.13654518803851384 accuracy: 0.9533730158730159
epoch: 35 loss: 0.1769502111359514 accuracy: 0.9325396825396826
epoch: 36 loss: 0.235813238510182 accuracy: 0.9236111111111112
epoch: 37 loss: 0.1855825976291228 accuracy: 0.935515873015873
epoch: 38 loss: 0.13270056934376795 accuracy: 0.9563492063492064
epoch: 39 loss: 0.17143343169508235 accuracy: 0.9384920634920635
epoch: 40 loss: 0.17773270987654252 accuracy: 0.9384920634920635
epoch: 41 loss: 0.1683632256047401 accuracy: 0.9484126984126984
epoch: 42 loss: 0.14050112919334973 accuracy: 0.9503968253968254
epoch: 43 loss: 0.141017514279127 accuracy: 0.9533730158730159
epoch: 44 loss: 0.10753050391813593 accuracy: 0.9652777777777778
epoch: 45 loss: 0.10763578091855235 accuracy: 0.9632936507936508
epoch: 46 loss: 0.11209396936418858 accuracy: 0.9613095238095238
epoch: 47 loss: 0.09332639401873166 accuracy: 0.9682539682539683
epoch: 48 loss: 0.06458175932666996 accuracy: 0.9801587301587301
epoch: 49 loss: 0.1618128459519958 accuracy: 0.9523809523809523
epoch: 50 loss: 0.09782982721096939 accuracy: 0.9702380952380952
epoch: 51 loss: 0.11790273619754728 accuracy: 0.9583333333333334
epoch: 52 loss: 0.09424658331258726 accuracy: 0.9682539682539683
epoch: 53 loss: 0.07653287704673108 accuracy: 0.9742063492063492
epoch: 54 loss: 0.12666169466406463 accuracy: 0.9672619047619048
epoch: 55 loss: 0.16640960189928708 accuracy: 0.9484126984126984
epoch: 56 loss: 0.2697397856751368 accuracy: 0.939484126984127
epoch: 57 loss: 0.16558472703891022 accuracy: 0.9543650793650794
epoch: 58 loss: 0.2095861649362459 accuracy: 0.9365079365079365
epoch: 59 loss: 0.17594213582693585 accuracy: 0.9444444444444444
epoch: 60 loss: 0.15097864540266095 accuracy: 0.9563492063492064
epoch: 61 loss: 0.08292688388887438 accuracy: 0.9682539682539683
epoch: 62 loss: 0.08037544788593161 accuracy: 0.9702380952380952
epoch: 63 loss: 0.14768764943194887 accuracy: 0.9593253968253969
epoch: 64 loss: 0.15162953201474416 accuracy: 0.9513888888888888
epoch: 65 loss: 0.08288552398545786 accuracy: 0.9702380952380952
epoch: 66 loss: 0.05482700602003625 accuracy: 0.9781746031746031
epoch: 67 loss: 0.08470485511805323 accuracy: 0.9732142857142857
epoch: 68 loss: 0.17993178779429328 accuracy: 0.9464285714285714
epoch: 69 loss: 0.21567103727203277 accuracy: 0.9275793650793651
epoch: 70 loss: 0.1074141434556435 accuracy: 0.9623015873015873
epoch: 71 loss: 0.078721916755902 accuracy: 0.9712301587301587
epoch: 72 loss: 0.07137340984338643 accuracy: 0.9781746031746031
epoch: 73 loss: 0.06051604799159534 accuracy: 0.9791666666666666
epoch: 74 loss: 0.05099301625760124 accuracy: 0.9841269841269841
epoch: 75 loss: 0.06858035473652264 accuracy: 0.9722222222222222
```

```
epoch: 76 loss: 0.02577509292487299 accuracy: 0.9920634920634921
epoch: 77 loss: 0.06707125844453306 accuracy: 0.9811507936507936
epoch: 78 loss: 0.09678113782265138 accuracy: 0.9682539682539683
epoch: 79 loss: 0.18480952581739984 accuracy: 0.933531746031746
epoch: 80 loss: 0.3398144297331335 accuracy: 0.9126984126984127
epoch: 81 loss: 0.24158476863383493 accuracy: 0.9236111111111112
epoch: 82 loss: 0.1586628755564258 accuracy: 0.9474206349206349
epoch: 83 loss: 0.09363518428105476 accuracy: 0.9742063492063492
epoch: 84 loss: 0.046637261923197434 accuracy: 0.9821428571428571
epoch: 85 loss: 0.02888386870045141 accuracy: 0.9900793650793651
epoch: 86 loss: 0.17020308905436346 accuracy: 0.9593253968253969
epoch: 87 loss: 0.0651616431931625 accuracy: 0.9771825396825397
epoch: 88 loss: 0.11983791759998966 accuracy: 0.9662698412698413
epoch: 89 loss: 0.03380616733216263 accuracy: 0.9861111111111112
epoch: 90 loss: 0.11318571450621155 accuracy: 0.9732142857142857
epoch: 91 loss: 0.08628110909494498 accuracy: 0.9712301587301587
epoch: 92 loss: 0.031673514248540965 accuracy: 0.9900793650793651
epoch: 93 loss: 0.027098109253168492 accuracy: 0.9890873015873016
epoch: 94 loss: 0.013111303608148153 accuracy: 0.996031746031746
epoch: 95 loss: 0.19513895376844126 accuracy: 0.9623015873015873
epoch: 96 loss: 0.17859710640775683 accuracy: 0.941468253968254
epoch: 97 loss: 0.09243080740617145 accuracy: 0.9692460317460317
epoch: 98 loss: 0.13877802544101586 accuracy: 0.9474206349206349
epoch: 99 loss: 0.07313032888082995 accuracy: 0.9692460317460317
```



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

Answer: While the training shows some instability, the training seems to have converged at around 50 epochs. The accuracy goes up and the loss goes down.

(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?

Answer: In the **PyTorch** implementation of the `CrossEntropyLoss`, the output is normalised internally and the input is not constrained to probability distributions, but can be any "unnormalized logits". See documentation.

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

Answer: This is most likely done to avoid mistakes on user-side.

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.

In [48]:
```
def fit(net, train, validation, optimizer, epochs=25, batch_size=10, devi
    """
    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
        'validation': torch.utils.data.DataLoader(validation, batch_size=
    }

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

    # some advanced PyTorch to look inside the network and log the output
    # 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(m
                    output.mean().detach().cpu().numpy()
                # store the mean absolute output values
                epoch_stats['%s %d: %s output abs mean' % (phase, idx, ty
                    output.abs().mean().detach().cpu().numpy()
                # store the std of the output values
                epoch_stats['%s %d: %s output std' % (phase, idx, type(mo
                    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 d
        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().num
                epoch_stats['%s accuracy' % phase] += accuracy.detach().c

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

                    # 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
                    history[key].append(epoch_stats[key])

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

            # store the outputs
            history['%s outputs' % phase].append(np.concatenate(epoch_out

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

    return history
```

In [49]:
```
# 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=
                                          sharex='col', sharey='none',
                                          figsize=(3.5 * cols * len(plots
        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 sel

        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()
```

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.

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

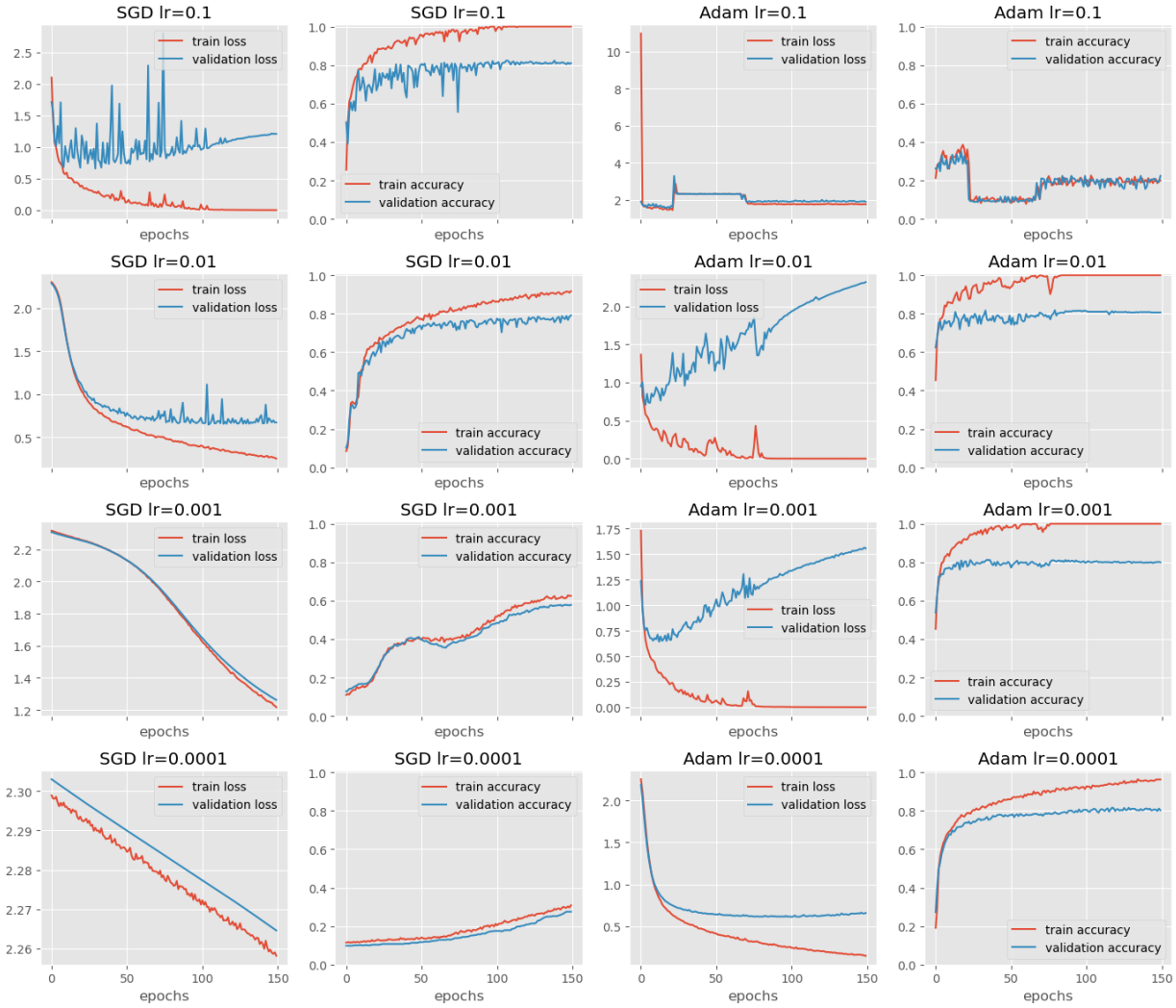
For each setting, we will plot:

- 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.

In [51]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'vali
                                 {'x': 'epochs', 'y': ['train 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=optim
    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=optim
    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	0.199219	0.223828
2	SGD lr=0.01	0.916992	0.791016
3	Adam lr=0.01	1.000000	0.805859
4	SGD lr=0.001	0.625000	0.578125
5	Adam lr=0.001	1.000000	0.799609
6	SGD lr=0.0001	0.308594	0.275000
7	Adam lr=0.0001	0.962891	0.802734

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, SGD lr=0.01, Adam lr=0.01, Adam lr=0.001, Adam lr=0.0001 (when a threshold of 0.75% validation accuracy is applied)

Experiments that need more training: SGD lr=0.001

Other experiments: Adam lr=0.1 (didn't improve much at all), SGD lr=0.0001 (didn't improve much at all)

Conclusion about amount of training epochs: For the networks with a good result, listed above, the 150 episodes seem to be enough to converge. For the networks with optimisers (Adam lr=0.1 and SGD lr=0.0001) even more training time would probably not have improved their performance significantly. Only for SGD with lr=0.001, more epochs would probably have lead to a higher performance.

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

Answer: For larger step sizes (up to 0.1), convergence happened faster, but the models are less stable. For smaller step sizes (up to 0.0001), the networks converged slower or not at all with a larger stability. Everything in between also performed 'in between'.

(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)

Answer: The model's performance, when the Adam optimiser with a learning rate of 0.1 was applied, seems to be unstable and the performance jumps. Additionally, the global optimum does not seem to be found.

(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: While the final performance is comparable, for a step size of 0.0001, the Adam optimiser does not seem to have fully converged yet.

Stability: With decreasing step size, the Adam optimiser gets more and more stable.

Speed: The larger the step size, the faster Adam converges, but as to be seen for 0.1, there seems to be a limit.

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.

In [52]:
```
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 accurac
                   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 ou
                   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'
                       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.

Answer: The model's increasing confidence is expressed as large deviations from 0. The further into the training trajectory, the more confident the model gets. This can be seen in the bottom-right plot. As the outputs are later normalised by the softmax function, the model gravitates to the extremes, which will minimise the loss.

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

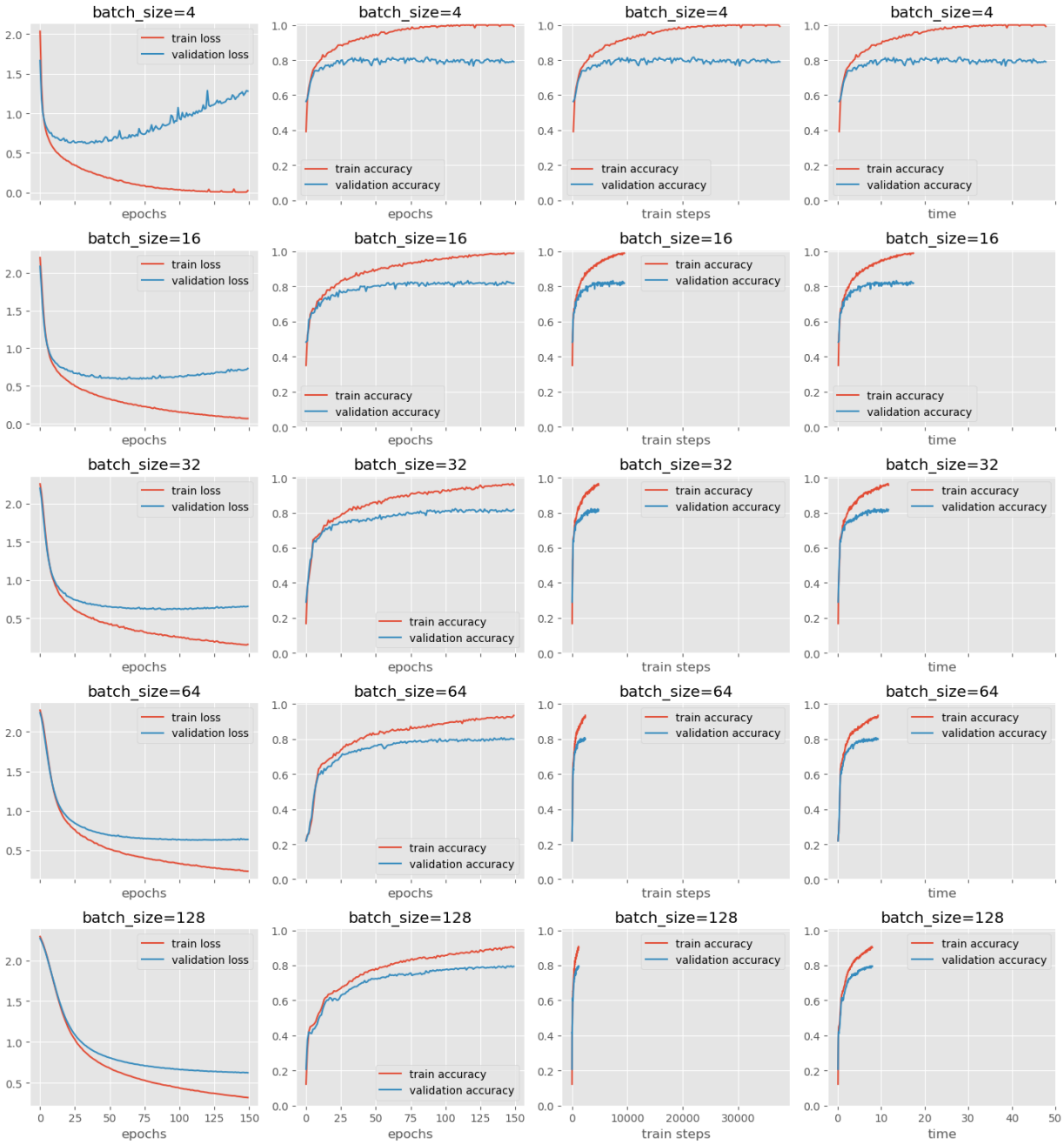
For each setting, we will plot:

- 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.

In [53]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'vali
                                 {'x': 'epochs', 'y': ['train accuracy', '
                                 {'x': 'train steps', 'y': ['train accurac
                                 {'x': 'time', 'y': ['train accuracy', 'va
                         table=['train accuracy', 'validation accuracy',
                         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=optim
    plotter.add('batch_size=%s' % str(batch_size), history, row=row, col=

plotter.done()
```



	experiment	train accuracy	validation accuracy	time
0	batch_size=4	0.992000	0.790000	47.780367
1	batch_size=16	0.989087	0.818359	17.378596
2	batch_size=32	0.957031	0.817187	11.645910
3	batch_size=64	0.934375	0.799579	9.268409
4	batch_size=128	0.900616	0.794181	7.942600

(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)

Answer: More frequent updates (a smaller batch size) will generally lead to a longer training time, as more forwards and backwards passes happen. A reduced number of training steps, caused by large batch sizes, leads to less training steps.

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

Answer: Fast convergence, on the scale of epochs, but longer training times in seconds.

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

Answer: Slower convergence, but fast training time due to fewer training steps.

2.6 Batch normalization (6 points)

Besides choosing the right hyperparameters, we can include other components to improve the training of the model. First we will experiment with batch normalization.

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

For a network with 1D feature vectors, you can use `torch.nn.BatchNorm1d` (documentation). There is also a `BatchNorm2d` and a `BatchNorm3d`.

(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.

In [59]:
```
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_runni
ng_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_runnin
g_stats=True)
  (5): ReLU()
  (6): Linear(in_features=64, out_features=10, bias=True)
)
```

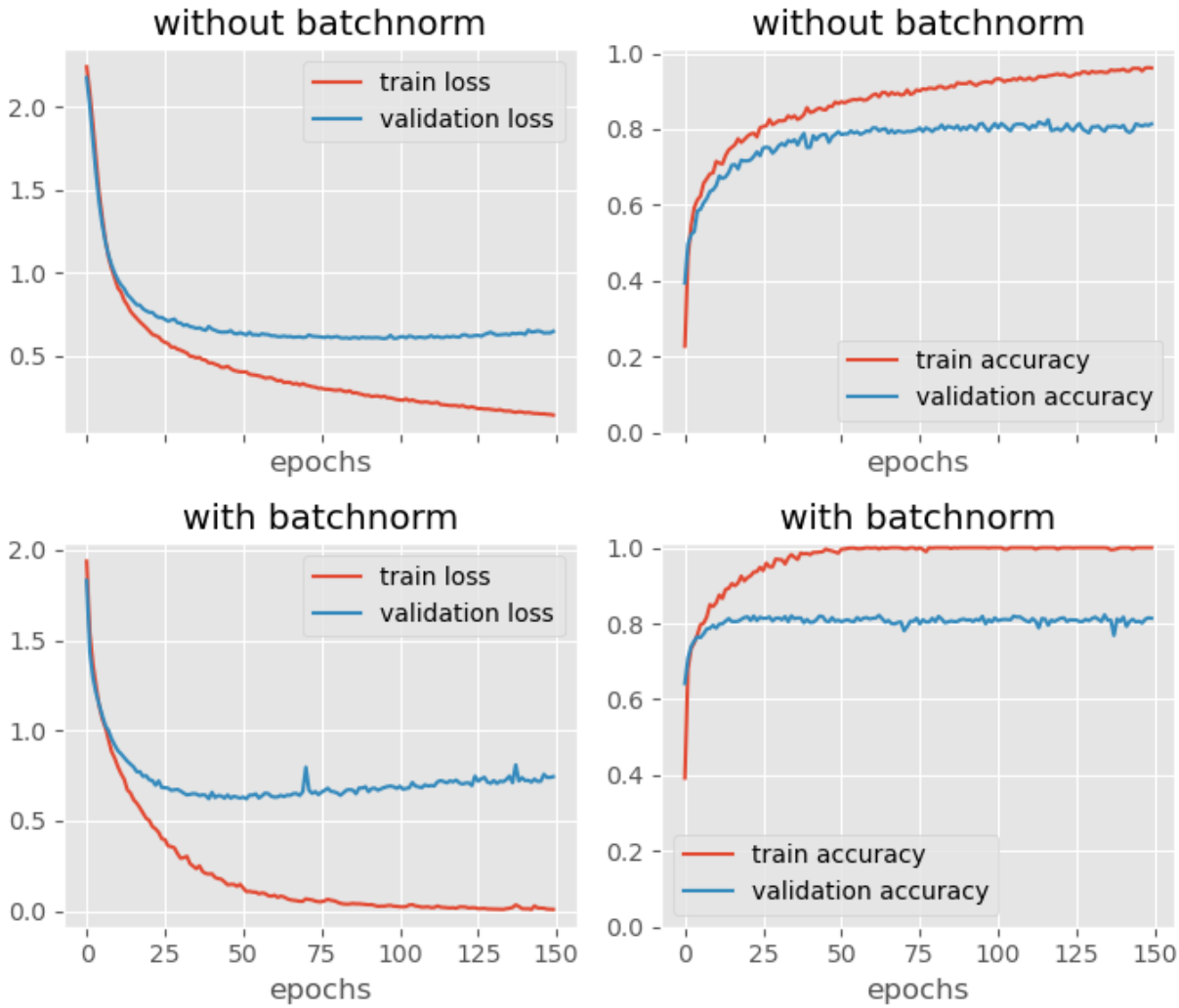
We will run an experiment to compare a network without batch normalization with a network with batch normalization.

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.

In [60]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'vali
                                 {'x': 'epochs', 'y': ['train 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=optim
    plotter.add(network_name, history, row=row, col=0)
    histories[network_name] = history

plotter.done()
```



	experiment	train accuracy	validation accuracy
0	without batchnorm	0.960938	0.813672
1	with batchnorm	1.000000	0.813672

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

Answer: Only partially. Without batch normalisation, the networks training accuracy increases slower and has not completely converged after 150 epochs, but for the final validation accuracy, there does not seem to be a noticeable difference.

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

Answer: The accuracy for both training and validation set increases faster with batch normalisation in place. Convergence is reached at about 50 epochs, while without batch normalisation, the network takes up to 150 epochs to converge.

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

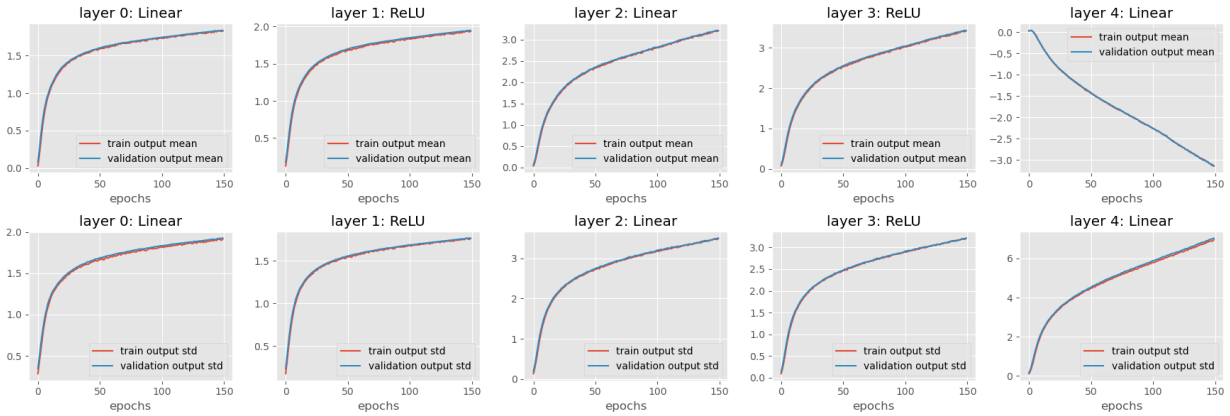
We will plot some statistics about the values inside the network.

In [61]:
```
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 %
                if len(keys) == 1:
                    key = keys[0]
                    ax = axs[i, layer]
                    ax.plot(history['epochs'], history[key], label='%s ou
                    ax.set_xlabel('epochs')
                    ax.legend()
                    ax.set_title(key.replace(' output %s' % stat, '').rep
            i += 1

    plt.tight_layout()
```

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

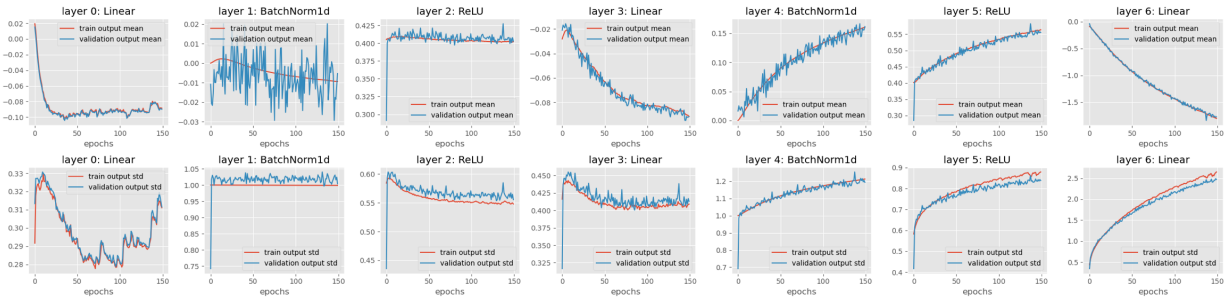
For each layer, the plots show the mean and standard deviation of the output values of that layer:

In [62]:
```
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.)

In [63]:
```
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)

Answer: In the layers from the network *without* batch normalisation, both the training and validation set mean increases with epochs to values up to `3.5`, while in the network *with* batch-normalisation, the mean only ranges up to about `0.5` (the last layer's output excluded). The much more constrained range of values in the batch-normalized network indicates that batch normalisation effectively manages the scale of the outputs across different layers, contributing to stability. (Additionally, when shifting the attention to the standard deviations shown in the 2nd row, the network without batch normalisation has much larger deviations, while the layers with batch normalisation lead to much smaller variances.) All of this is due to the batch normalisation layers, standardising the mean and variance of the activations to be (more) normally distributed within each mini-batch.

(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)

Answer: The validation curves appear more noisy, as the `BatchNorm1d` does not update the running average estimations in testing, more specifically when `track_running_stats` is switched off.

(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)

Answer: Normalisation layers have the learnable parameters γ and β to control the magnitude of the activations adaptively. In the backpropagation process these are optimised to scale the normalised outputs to allow the network to learn most effectively. See documentation. Parameter `$\gamma$` is initialised a `1` and scales the variance as a factor, while `$\beta$` is a constant, contributing to the mean, initialised at `0`. By default, the momentum is set to `0.1`, which allows for the gradual shift shown in layer 4.

2.7 Data augmentation (4 points)

Next, 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 σ = 0, 0.01, 0.1, or 0.2.
- Flipping the image horizontally.
- Shifting the image up, down, left, or right by one pixel.

To implement data augmentation, we create a new dataset class that generates augmented images and use this instead of our normal training set.

In [64]:
```
class NoisyDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, noise_sigma=0, flip_horizontal=False, shi
        self.dataset = dataset
        self.noise_sigma = noise_sigma
        self.flip_horizontal = flip_horizontal
        self.shift = shift

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

    def __getitem__(self, idx):
        x, y = self.dataset[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 dire
        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.

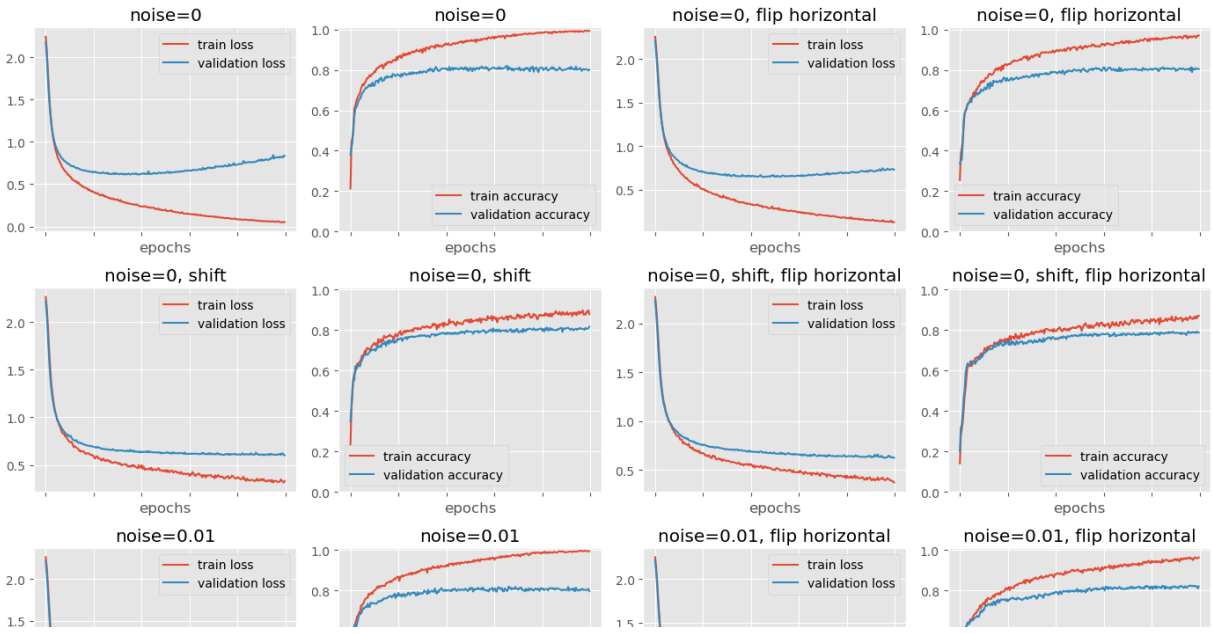
We will train for 250 epochs.

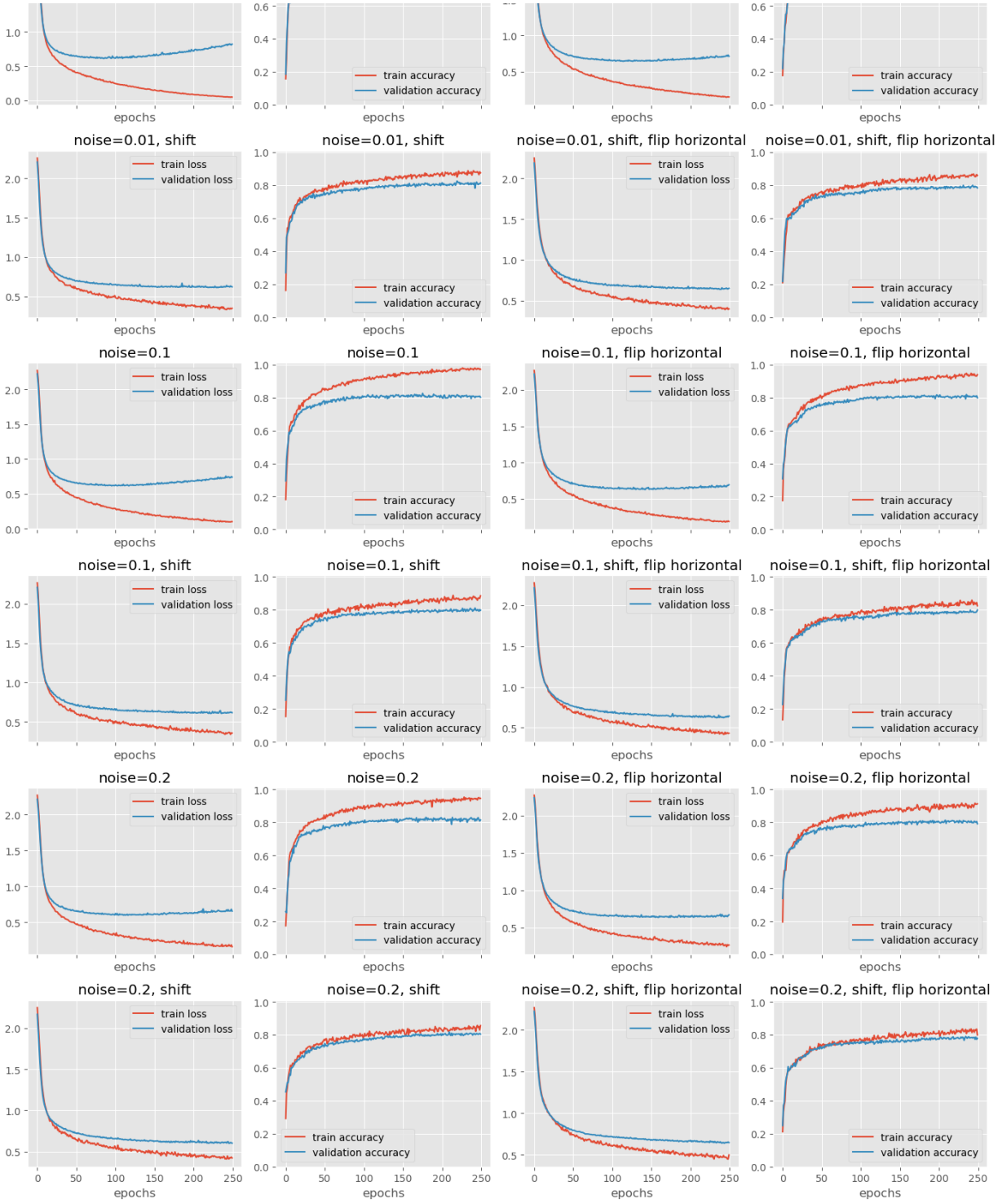
We keep the other settings as before:

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

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

In [65]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'vali
                                 {'x': 'epochs', 'y': ['train accuracy', '
                         table=['train accuracy', 'validation accuracy',
                         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
            net = build_net()
            optimizer = torch.optim.Adam(net.parameters(), lr=lr)
            history = fit(net, noisy_fashion_train, fashion_validation, o
            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=0.01, 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	time
0	noise=0	0.993164	0.800781	23.834541
1	noise=0, flip horizontal	0.970703	0.803906	27.769847
2	noise=0, shift	0.878906	0.816797	28.173541
3	noise=0, shift, flip horizontal	0.871094	0.787500	30.266912
4	noise=0.01	0.994141	0.795703	24.403027
5	noise=0.01, flip horizontal	0.962891	0.819531	28.192669
6	noise=0.01, shift	0.875000	0.812891	29.289789
7	noise=0.01, shift, flip horizontal	0.858398	0.783594	29.661687
8	noise=0.1	0.971680	0.801562	24.815528
9	noise=0.1, flip horizontal	0.938477	0.798828	26.240449
10	noise=0.1, shift	0.885742	0.798047	27.252635
11	noise=0.1, shift, flip horizontal	0.825195	0.799219	28.777783
12	noise=0.2	0.945312	0.814453	23.465903
13	noise=0.2, flip horizontal	0.912109	0.791406	25.250661
14	noise=0.2, shift	0.855469	0.803125	26.927524
15	noise=0.2, shift, flip horizontal	0.798828	0.777734	28.667727

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

Adding noise: Answer: It seems like the validation accuracy is not reduced by the introduction of noise, even up to $\sigma = 0.2$. The training accuracy on the other hand drops down from about `0.993` to `0.945` when looking at the isolated effect of the noise.

Horizontal flips: Answer: Again, the validation accuracy does not change significantly a lot, but the training accuracy seems to drop about `0.3` across all levels of noise. Noteworthy is that the model that was trained on the data with horizontally flipped samples, did not completely converge within the time frame of `250` epochs.

Shifting: Answer: When applied in isolation (no horizontal flips), shifting seems to not significantly change the accuracy achieved on the validation data (across all levels of noise). The training accuracy does decrease a bit and again, the model does not seem to be fully converged after `250` epochs.

When applying noise, shifting and horizontal flips together, the performance during training goes more towards the performance for the validation set, but the accuracy on the validation set does not seem to decrease significantly. This indicates that the model is indeed learning a more robust representation of the data.

Generally, the extremes (e.g. large noise or applying multiple techniques in combination) seem to cut into the performance a bit. At some point it will be too hard for the model to ignore the noise.

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

Answer: Finding a generalisable solution to the classification of images will be generally harder than learning the labels of the training data by heart. With different data augmentation techniques, the variability in the data increases and consequently the problem gets harder. Some data augmentation techniques also increase the size of the training data, which means that there are more patterns to learn.

2.8 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.

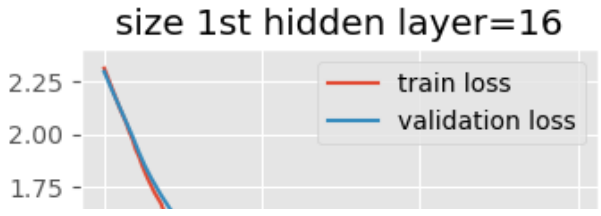
In [68]:
```
def build_net(width = 128):
    return torch.nn.Sequential(
        torch.nn.Linear(784, width),
        torch.nn.ReLU(),
        torch.nn.Linear(width, int(width/2)),
        torch.nn.ReLU(),
        torch.nn.Linear(int(width/2), 10)
    )
```

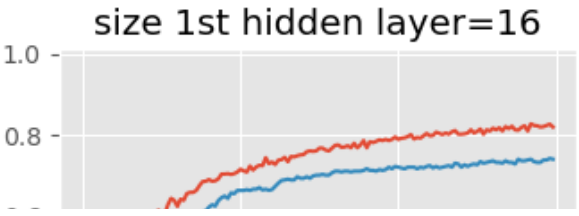
(b) Set up an experiment to see how the size of the network affects our results. (1 point)

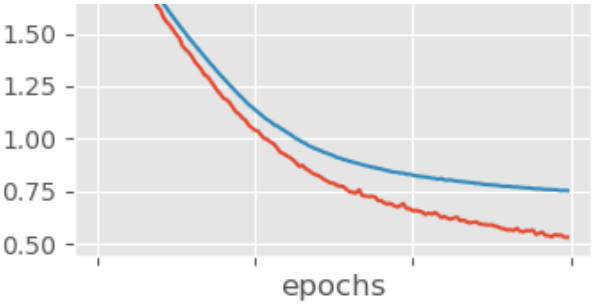
We keep the other settings as before:

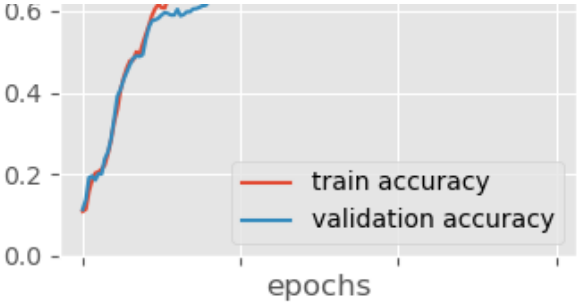
- Optimizer: Adam
- Epochs: 150
- Learning rate: 0.0001
- Minibatch size: 32
- Widths: 16, 32, 64, 128, 256, 512

In [71]:
```
plotter = HistoryPlotter(plots=[{'x': 'epochs', 'y': ['train loss', 'vali
                                 {'x': 'epochs', 'y': ['train accuracy', '
                         table=['train accuracy', 'validation accuracy',
                         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=optim
    label = ('size 1st hidden layer=%s' % str(width))
    plotter.add(label, history, row=row, col=0)

plotter.done()
```

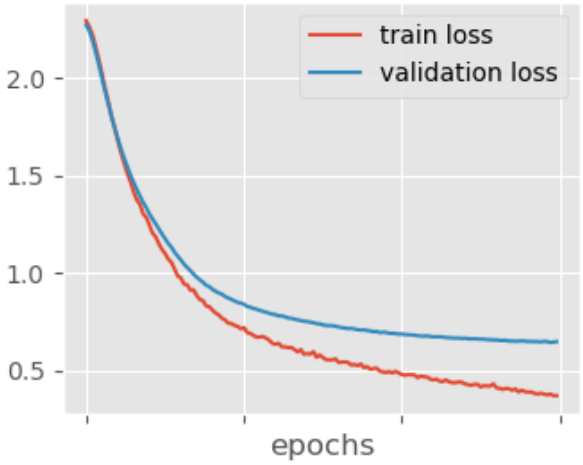




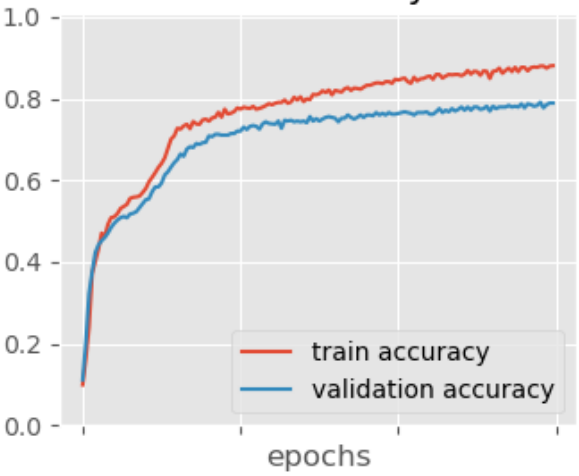




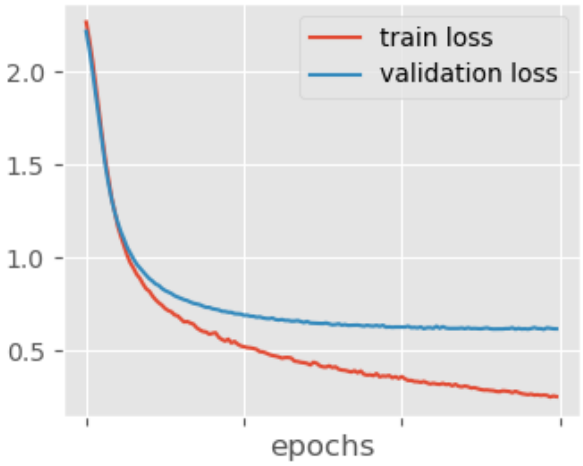
size 1st hidden layer=32



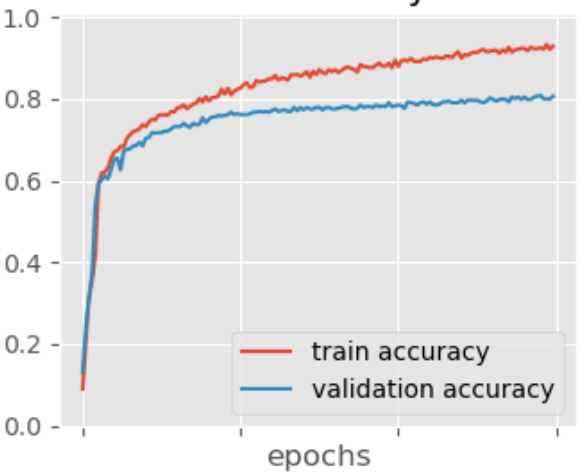
size 1st hidden layer=32



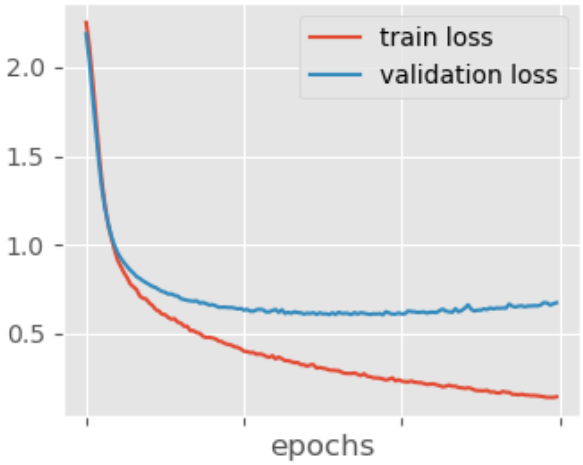
size 1st hidden layer=64



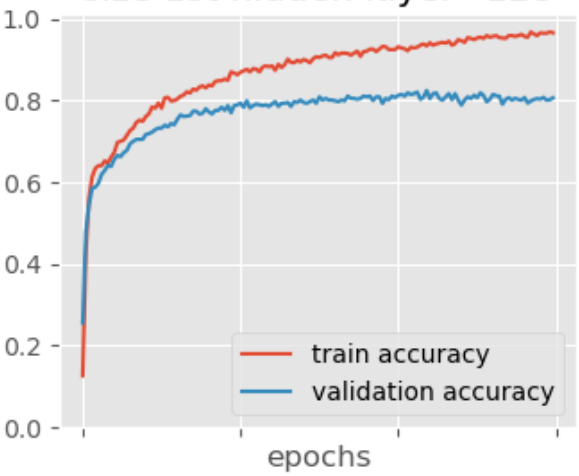
size 1st hidden layer=64



size 1st hidden layer=128

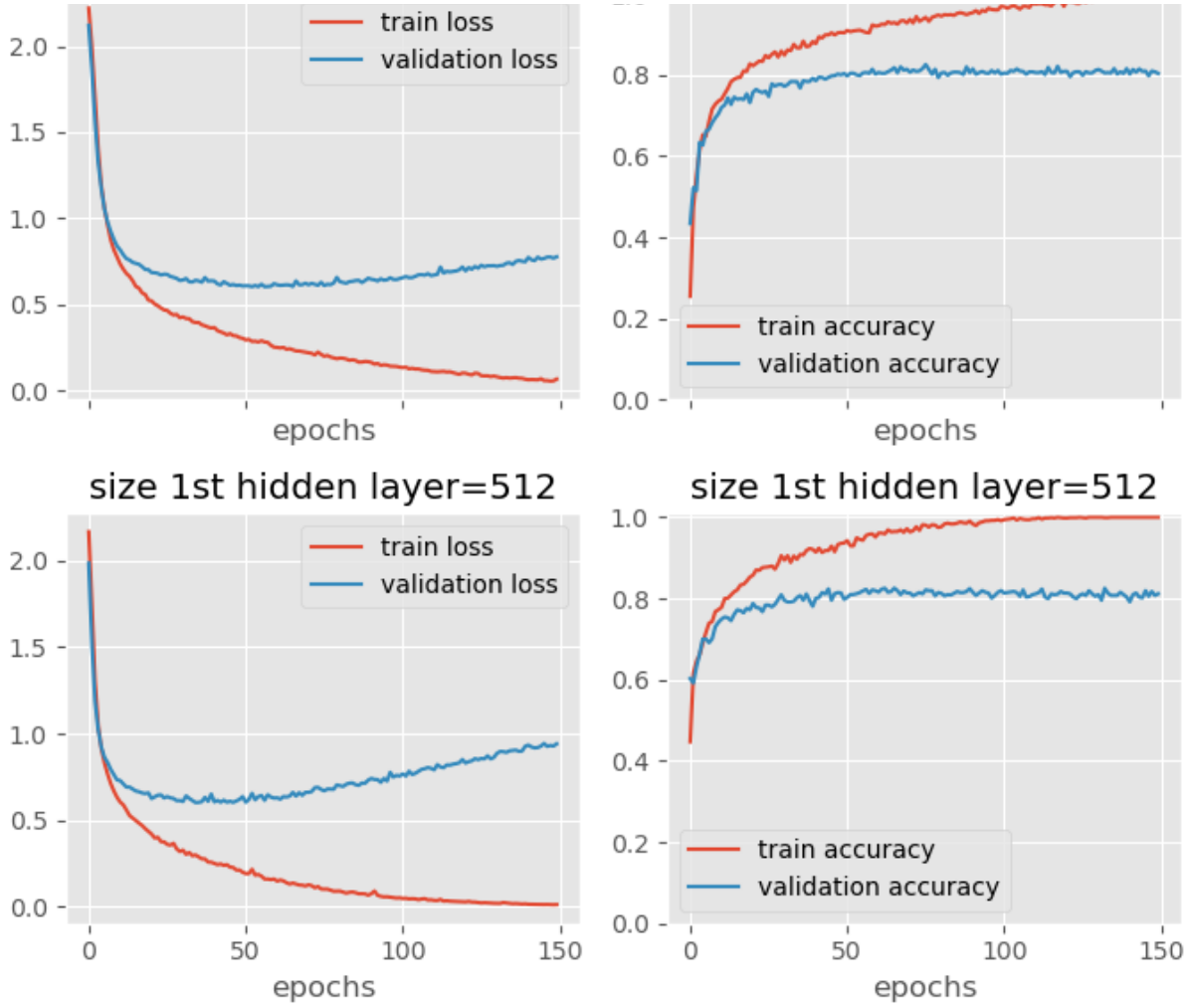


size 1st hidden layer=128



size 1st hidden layer=256

size 1st hidden layer=256



	experiment	train accuracy	validation accuracy	time
0	size 1st hidden layer=16	0.820312	0.741406	9.806598
1	size 1st hidden layer=32	0.880859	0.789453	10.366783
2	size 1st hidden layer=64	0.929688	0.807031	11.430035
3	size 1st hidden layer=128	0.965820	0.806641	11.882502
4	size 1st hidden layer=256	0.989258	0.803516	13.226497
5	size 1st hidden layer=512	1.000000	0.811328	18.233030

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

Answer: Up to a hidden layer size of `64`, the networks converge to a suboptimal solution and the training and validation accuracy is reduced. The missing complexity in the model's architecture leads to an early plato in the model's performance.

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

Answer: Yes, the validation loss is increasing significantly after the initial drop for the networks with more than `128` in the hidden dimension of the first layer.

(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.)

In [84]:
```
hidden_layer_size = 128
net = build_net(hidden_layer_size)
total_params = 0
layer_dict = {0: f'input layer (784x{hidden_layer_size})   ', 1: f'bias i
for i, param in enumerate(net.parameters()):
    current_layer = param.numel()
    total_params += current_layer
    print(f'Layer {i} - {layer_dict[i]}: {current_layer} parameters')
print(f'Total parameters: {total_params}')
```

```
Layer 0 - input layer (784x128)   : 100352 parameters
Layer 1 - bias input layer (128)  : 128 parameters
Layer 2 - hidden layer (128x64)   : 8192 parameters
Layer 3 - bias hidden layer (64)  : 64 parameters
Layer 4 - output layer (64x10)    : 640 parameters
Layer 5 - bias output layer (10)  : 10 parameters
Total parameters: 109386
```

Answer: see above for layer-wise breakdown and total (109386)

2.9 Discussion (3 points)

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

Answer: While the training accuracy was always around `0.96` and the validation accuracy around `0.81`, there were relatively large variations. This is hinting towards the fact that any measurement of a model's performance after `150` epochs will be a *noisy estimate*. Therefore, when choosing hyperparameters, one would need to account for this uncertainty. Solutions could be to take the average of multiple runs, use a larger validation set, or cross-validation.

(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)

Answer: When tuning the hyperparameters on a (small) validation set, it is likely that model is overfitting on this. On an unseen part of the data that was not optimised for, the model will most likely perform not quite as well. If the test set is drawn from an entirely different distribution on the other hand, a larger performance drop should be expected. Then, other hyperparameters would potentially lead to a better performance.

The end

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

This assignment has 42 points. Version 7ac2078 / 2024-09-11