Assignment 2: Fashion Dataset and Architecture Comparisons

In this assignment, you are to perform the following:

1. Study the `FashionMNIST` dataset, and understand the images and their respective labels.
2. Implement a `Trainer` class in the file `trainer_lib.py` for reusable and reproducible training loops.
3. Implement the following architectures:

- Linear model
- MLP with one hidden layer
- Simple convolution with pooling
- Deep convolution with pooling, followed by MLP with one hidden layer.

In [2]:
```
"🔒"
import torch
from torch import nn
from torch import optim
from torch.utils.data import DataLoader, random_split
import torchvision
from torchsummaryX import summary
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from importlib import reload

import warnings
warnings.filterwarnings('ignore')
```

Loading the data

In this section, you will simply run the following data loading cells to familiarize yourself with the layout and semantics of the training dataset.

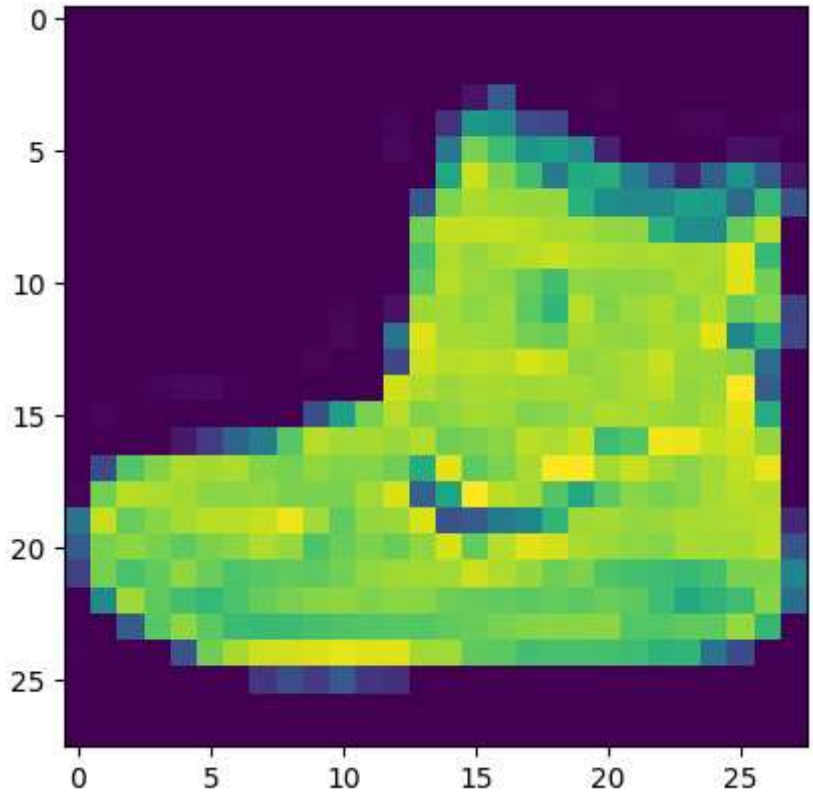
In [3]:
```
"🔒"
#
# Loading data
#
home = os.environ.get('HOME')
root = os.path.join(home, 'public/data')
dataset = torchvision.datasets.FashionMNIST(
    root,
    train=True,
```

```
    transform=torchvision.transforms.ToTensor())

dataset
```

Out[3]:
```
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: /home/jovyan/public/data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

In [4]:
```
"🔒"
#
# Print out important stats about the dataset
#
image_tensor, label = dataset[0]
print("The first image is of shape:", image_tensor.shape)
print("The first label is:", label)

plt.imshow(np.transpose(image_tensor, (1, 2, 0)));
```

```
The first image is of shape: torch.Size([1, 28, 28])
The first label is: 9
```



The dataset object has a `.classes` field that contains the names of the different labels. It has 10 classes ranging from **T-shirt** to **Ankle boot**.

In [5]:
```
"🔒"
#
# Print the Lookup
#
```

```
lookup = pd.Series({x: i for (i,x) in enumerate(dataset.classes)})
lookup
```

Out[5]:
```
T-shirt/top    0
Trouser        1
Pullover       2
Dress          3
Coat           4
Sandal         5
Shirt          6
Sneaker        7
Bag            8
Ankle boot     9
dtype: int64
```

In fact, we can print the first 36 entries in the dataset, and plot them as a grid. Below is the result of that.

In [6]:
```
"🔒"
xs = dataset.data[:36]
xs = xs.reshape(36, 1, 28, 28)
mosiac = torchvision.utils.make_grid(xs, nrow=6)
mosiac = np.transpose(mosiac, (1, 2, 0))
plt.imshow(mosiac)
plt.xticks([])
plt.yticks([]);
```

Getting ready for training and validation

We will be using the same training data for all three neural network architectures.

The `train_dataloader` is a dataloader for the training dataset, and `val_dataloader` is the dataloader for the validation dataset.

In [7]:
```
"🔒"
train_dataset, val_dataset = random_split(dataset, (0.8, 0.2))
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=len(val_dataset), shuffle=False
```

Linear classifier

In this section, you will implement a simple linear model. The linear model is to be implemented as a `nn.Module` and should have a field `model` which is a `nn.Sequential` module.

You are to complete the implementation by creating two layers in the `nn.Sequence(...)`:

- A layer to flatten the input from `(1,28,28)` to a vector.
- The second layer perform linear classification to 10 dimensional logit. You are to use the `nn.LazyLinear` layer to implement the linear layer. The advantage of the `LazyLinear` layer is that you do not need to compute the input dimension explicitly.

In [8]:
```
"✍"
# @workUnit

class MyLinear(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # complete the architecture
            nn.Flatten(),
            nn.LazyLinear(10),
        )
    def forward(self, image):
        return self.model(image)
```

In [9]:
```
"🔒"
# @check
# @title: check linear model architecture

m = MyLinear()
xs, ys = next(iter(train_dataloader))
summary(m, xs);
```

```
==========================================================================
                    Kernel Shape Output Shape Params Mult-Adds
Layer
0_model.Flatten_0               -    [32, 784]      -         -
1_model.LazyLinear_1    [784, 10]     [32, 10]  7.85k     7.84k
--------------------------------------------------------------------------
                     Totals
Total params          7.85k
Trainable params      7.85k
Non-trainable params    0.0
Mult-Adds             7.84k
==========================================================================
```

A trainer class

In this section, you will be implementing a trainer class that will be used throughout the assignment.

The trainer class has the following methods:

Constructor:

- Input:
  - `model` : a `nn.Module` that is the neural network model.
  - `train_dataloader` : a `DataLoader` that is the training data
  - `val_dataloader` : is the `Dataloader` for validation
  - The constructor uses the `optim.Adam(...)` as the optimizer.
  - The constructor uses the `nn.CrossEntropy` as the loss function.

`Trainer.train_one_epoch`:

It performs training of `model` for **one epoch** by iterating over the batches from `train_dataloader`.

- Input:
  - `max_batches` is the maximum batches taken from the dataloader. **For performance reasons, we will only sample 10 batches.**
- Output: it returns two float numbers:
  - mean loss over the batches
  - mean accuracy over the batches

`Trainer.val_one_epoch`:

It performs validation by iterating over all batches from the validation dataloader.

- Output: it returns two float numbers:
  - mean loss

- mean accuracy

`Trainer.train`

This method performs training.

- Input:
  - `epochs`: the number of epochs.
  - `max_batches`: the maximum batches to take per epoch.
- Output: returns a `DataFrame` containing training loss, training accuracy, validation loss, validation accuracy, and the time per epoch.

`Trainer.reset`

This method resets the trainable parameters of the model.

A skeleton implementation is given in `trainer_lib.py`. You are to complete the implementation.

In [10]:
```
"🔒"
import trainer_lib
```

Below are some basic sanity checks of the trainer implementation.

In [11]:
```
"🔒"
# @check
# @title: check trainer construction
reload(trainer_lib)
trainer = trainer_lib.Trainer(m, train_dataloader, val_dataloader)
print(trainer is not None)
```

```
True
```

In [12]:
```
"🔒"
# @check
# @title: check trainer methods
for method in ['train_one_epoch', 'val_one_epoch', 'train', 'reset']:
    print(f"trainer.{method}?", hasattr(trainer, method))
```

```
trainer.train_one_epoch? True
trainer.val_one_epoch? True
trainer.train? True
trainer.reset? True
```

In [13]:
```
"🔒"
# @check
# @title: check trainer.train_one_epoch

(loss, acc) = trainer.train_one_epoch(max_batches=1)
print('loss is numeric', isinstance(loss, float))
print('acc is numeric', isinstance(acc, float))
```

```
loss is numeric True
acc is numeric True
```

In [14]:
```
"🔒"
# @check
# @title: check trainer.val_one_epoch

(loss, acc) = trainer.val_one_epoch()
print('loss is numeric', isinstance(loss, float))
print('acc is numeric', isinstance(acc, float))
```

```
loss is numeric True
acc is numeric True
```

In [15]:
```
"🔒"
# @check
# @title: check trainer.reset

trainer.reset()
print("Ok")
```

```
Ok
```

Training the linear model

In this section, we will train the linear model. We will take 10 batches per epoch, and train for 20 epochs.

The training history is stored in `model_linear_log`.

This will take approximately **70 seconds**.

In [16]:
```
"🔒"
reload(trainer_lib)

model_linear = MyLinear()
trainer = trainer_lib.Trainer(model_linear, train_dataloader, val_dataloader)
trainer.reset()
model_linear_log = trainer.train(epochs=20, max_batches=10)
model_linear_log.round(2)
```

```
[0 (3.13s)]: train_loss=2.07 train_acc=0.32, val_loss=1.81 val_acc=0.42
[1 (3.25s)]: train_loss=1.63 train_acc=0.55, val_loss=1.45 val_acc=0.62
[2 (3.09s)]: train_loss=1.38 train_acc=0.61, val_loss=1.25 val_acc=0.66
[3 (3.49s)]: train_loss=1.20 train_acc=0.67, val_loss=1.15 val_acc=0.66
[4 (3.07s)]: train_loss=1.14 train_acc=0.65, val_loss=1.04 val_acc=0.67
[5 (3.48s)]: train_loss=1.04 train_acc=0.67, val_loss=0.98 val_acc=0.69
[6 (3.61s)]: train_loss=1.01 train_acc=0.68, val_loss=0.93 val_acc=0.70
[7 (2.64s)]: train_loss=0.87 train_acc=0.74, val_loss=0.91 val_acc=0.69
[8 (3.40s)]: train_loss=0.91 train_acc=0.69, val_loss=0.88 val_acc=0.72
[9 (3.37s)]: train_loss=0.91 train_acc=0.71, val_loss=0.86 val_acc=0.72
[10 (3.44s)]: train_loss=0.80 train_acc=0.73, val_loss=0.84 val_acc=0.72
[11 (3.41s)]: train_loss=0.83 train_acc=0.76, val_loss=0.82 val_acc=0.72
[12 (3.39s)]: train_loss=0.80 train_acc=0.75, val_loss=0.81 val_acc=0.74
[13 (3.39s)]: train_loss=0.80 train_acc=0.74, val_loss=0.79 val_acc=0.75
[14 (3.38s)]: train_loss=0.74 train_acc=0.76, val_loss=0.79 val_acc=0.72
[15 (2.72s)]: train_loss=0.76 train_acc=0.75, val_loss=0.76 val_acc=0.73
[16 (3.11s)]: train_loss=0.77 train_acc=0.75, val_loss=0.75 val_acc=0.74
[17 (3.79s)]: train_loss=0.79 train_acc=0.74, val_loss=0.74 val_acc=0.76
[18 (3.42s)]: train_loss=0.74 train_acc=0.73, val_loss=0.73 val_acc=0.76
[19 (3.30s)]: train_loss=0.68 train_acc=0.78, val_loss=0.72 val_acc=0.76
== Total training time 65.92 seconds ==
```

Out[16]:

	train_loss	train_accuracy	val_loss	val_accuracy	epoch_duration
0	2.07	0.32	1.81	0.42	3.13
1	1.63	0.55	1.45	0.62	3.25
2	1.38	0.61	1.25	0.66	3.09
3	1.20	0.67	1.15	0.66	3.49
4	1.14	0.65	1.04	0.67	3.07
5	1.04	0.67	0.98	0.69	3.48
6	1.01	0.68	0.93	0.70	3.61
7	0.87	0.74	0.91	0.69	2.64
8	0.91	0.69	0.88	0.72	3.40
9	0.91	0.71	0.86	0.72	3.37
10	0.80	0.73	0.84	0.72	3.44
11	0.83	0.76	0.82	0.72	3.41
12	0.80	0.75	0.81	0.74	3.39
13	0.80	0.74	0.79	0.75	3.39
14	0.74	0.76	0.79	0.72	3.38
15	0.76	0.75	0.76	0.73	2.72
16	0.77	0.75	0.75	0.74	3.11
17	0.79	0.74	0.74	0.76	3.79
18	0.74	0.73	0.73	0.76	3.42
19	0.68	0.78	0.72	0.76	3.30

MLP Models

In this section, you are to construct a MLP model. It uses `LazyLinear` to create a hidden dimension of 50, with activation `nn.ReLU` for the hidden layer.

In [17]:
```
"✍"
# @workUnit

class MLPModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # complete the following
            nn.Flatten(),
            nn.LazyLinear(50),
            nn.ReLU(),
            nn.LazyLinear(10),
        )
```

```
    def forward(self, x):
        return self.model(x)
```

In [18]:
```
"🔒"
# @check
# @title: MLP model architecture

model_mlp = MLPModel()
summary(model_mlp, xs);
```

```
==========================================================================
                     Kernel Shape Output Shape  Params Mult-Adds
Layer
0_model.Flatten_0               -    [32, 784]       -         -
1_model.LazyLinear_1    [784, 50]     [32, 50]  39.25k     39.2k
2_model.ReLU_2                  -     [32, 50]       -         -
3_model.LazyLinear_3     [50, 10]     [32, 10]   510.0     500.0
--------------------------------------------------------------------------
                      Totals
Total params          39.76k
Trainable params      39.76k
Non-trainable params     0.0
Mult-Adds             39.7k
==========================================================================
```

Training MLP

We will train the MLP model in 20 epochs with 10 batches per epoch.

You should expect:

- Total training time: 80 seconds
- Final validation accuracy approximately 75% or higher.

In [19]:
```
"🔒"
model_mlp = MLPModel()
reload(trainer_lib)
trainer = trainer_lib.Trainer(model_mlp, train_dataloader, val_dataloader)
trainer.reset()
model_mlp_log = trainer.train(epochs=20, max_batches=10)
model_mlp_log.round(2)
```

```
[0 (4.87s)]: train_loss=2.19 train_acc=0.20, val_loss=2.06 val_acc=0.35
[1 (4.30s)]: train_loss=1.90 train_acc=0.42, val_loss=1.74 val_acc=0.48
[2 (4.21s)]: train_loss=1.68 train_acc=0.53, val_loss=1.49 val_acc=0.65
[3 (4.19s)]: train_loss=1.41 train_acc=0.62, val_loss=1.28 val_acc=0.65
[4 (4.91s)]: train_loss=1.17 train_acc=0.63, val_loss=1.13 val_acc=0.63
[5 (5.20s)]: train_loss=1.05 train_acc=0.68, val_loss=1.03 val_acc=0.67
[6 (3.78s)]: train_loss=1.01 train_acc=0.65, val_loss=0.94 val_acc=0.68
[7 (4.41s)]: train_loss=0.90 train_acc=0.65, val_loss=0.89 val_acc=0.67
[8 (5.69s)]: train_loss=0.87 train_acc=0.71, val_loss=0.87 val_acc=0.70
[9 (5.09s)]: train_loss=0.83 train_acc=0.72, val_loss=0.81 val_acc=0.72
[10 (4.10s)]: train_loss=0.77 train_acc=0.71, val_loss=0.78 val_acc=0.73
[11 (6.10s)]: train_loss=0.78 train_acc=0.75, val_loss=0.76 val_acc=0.74
[12 (6.31s)]: train_loss=0.73 train_acc=0.72, val_loss=0.75 val_acc=0.74
[13 (4.29s)]: train_loss=0.81 train_acc=0.71, val_loss=0.73 val_acc=0.75
[14 (4.20s)]: train_loss=0.69 train_acc=0.75, val_loss=0.73 val_acc=0.75
[15 (3.91s)]: train_loss=0.70 train_acc=0.76, val_loss=0.73 val_acc=0.75
[16 (5.09s)]: train_loss=0.75 train_acc=0.72, val_loss=0.72 val_acc=0.75
[17 (4.91s)]: train_loss=0.67 train_acc=0.76, val_loss=0.70 val_acc=0.76
[18 (5.39s)]: train_loss=0.72 train_acc=0.78, val_loss=0.67 val_acc=0.78
[19 (4.60s)]: train_loss=0.70 train_acc=0.76, val_loss=0.67 val_acc=0.77
== Total training time 95.67 seconds ==
```

Out[19]:

	train_loss	train_accuracy	val_loss	val_accuracy	epoch_duration
0	2.19	0.20	2.06	0.35	4.87
1	1.90	0.42	1.74	0.48	4.30
2	1.68	0.53	1.49	0.65	4.21
3	1.41	0.62	1.28	0.65	4.19
4	1.17	0.63	1.13	0.63	4.91
5	1.05	0.68	1.03	0.67	5.20
6	1.01	0.65	0.94	0.68	3.78
7	0.90	0.65	0.89	0.67	4.41
8	0.87	0.71	0.87	0.70	5.69
9	0.83	0.72	0.81	0.72	5.09
10	0.77	0.71	0.78	0.73	4.10
11	0.78	0.75	0.76	0.74	6.10
12	0.73	0.72	0.75	0.74	6.31
13	0.81	0.71	0.73	0.75	4.29
14	0.69	0.75	0.73	0.75	4.20
15	0.70	0.76	0.73	0.75	3.91
16	0.75	0.72	0.72	0.75	5.09
17	0.67	0.76	0.70	0.76	4.91
18	0.72	0.78	0.67	0.78	5.39
19	0.70	0.76	0.67	0.77	4.60

Convolution based networks

In this section, we will construct a simple convolutional network. It consists of the following layers:

- 2D convolution with the `num_kernels` kernels of size `kernel_size`.
- Max pooling with pooling size `pool_size`.
- ReLU activation
- Flatten the max pooling output to a vector
- A (lazy) linear layer that maps to 10 dimensional logits vector

In [20]:
```
"✍"
# @workUnit

class LinearCNNModel(nn.Module):
    def __init__(self, num_kernels, kernel_size, pool_size):
```

```
        super().__init__()
        self.model = nn.Sequential(
            # complete the following
            nn.Conv2d(1, num_kernels, kernel_size, padding='same'),
            nn.MaxPool2d(pool_size),
            nn.ReLU(),
            nn.Flatten(),
            nn.LazyLinear(10)
        )
    def forward(self, x):
        return self.model(x)
```

We will construct a convolutional network with 5 kernels, with kernel size 3, and pooling size of 2.

In [21]:
```
"🔒"
# @check
# @title: CNN Linear architecture

model_cnn_linear = LinearCNNModel(5, 3, 2)
summary(model_cnn_linear, xs);
```

```
==========================================================================
                      Kernel Shape     Output Shape Params Mult-Adds
Layer
0_model.Conv2d_0      [1, 5, 3, 3]  [32, 5, 28, 28]   50.0    35.28k
1_model.MaxPool2d_1              -  [32, 5, 14, 14]      -         -
2_model.ReLU_2                   -  [32, 5, 14, 14]      -         -
3_model.Flatten_3                -        [32, 980]      -         -
4_model.LazyLinear_4     [980, 10]         [32, 10]  9.81k      9.8k
--------------------------------------------------------------------------
                      Totals
Total params           9.86k
Trainable params       9.86k
Non-trainable params     0.0
Mult-Adds             45.08k
==========================================================================
```

Training of Convolutional Network

We will train a convolutional network using 20 epochs, and 10 batches per epoch.

You should expect:

- Total training time: 120 seconds
- Validation accuracy approximately 76% or higher.

In [22]:
```
"🔒"
reload(trainer_lib)
trainer = trainer_lib.Trainer(model_cnn_linear, train_dataloader, val_dataloader)
trainer.reset()
model_cnn_linear_log = trainer.train(epochs=20, max_batches=10)
model_cnn_linear_log.round(2)
```

```
[0 (7.78s)]: train_loss=2.15 train_acc=0.29, val_loss=2.00 val_acc=0.44
[1 (11.91s)]: train_loss=1.83 train_acc=0.49, val_loss=1.65 val_acc=0.64
[2 (8.00s)]: train_loss=1.53 train_acc=0.63, val_loss=1.40 val_acc=0.57
[3 (8.69s)]: train_loss=1.32 train_acc=0.57, val_loss=1.19 val_acc=0.66
[4 (8.71s)]: train_loss=1.19 train_acc=0.64, val_loss=1.06 val_acc=0.68
[5 (9.54s)]: train_loss=1.02 train_acc=0.67, val_loss=0.96 val_acc=0.72
[6 (8.75s)]: train_loss=0.95 train_acc=0.72, val_loss=0.91 val_acc=0.72
[7 (7.91s)]: train_loss=0.87 train_acc=0.73, val_loss=0.86 val_acc=0.71
[8 (8.00s)]: train_loss=0.87 train_acc=0.69, val_loss=0.82 val_acc=0.72
[9 (7.39s)]: train_loss=0.76 train_acc=0.75, val_loss=0.81 val_acc=0.72
[10 (6.70s)]: train_loss=0.71 train_acc=0.76, val_loss=0.76 val_acc=0.74
[11 (6.90s)]: train_loss=0.82 train_acc=0.71, val_loss=0.76 val_acc=0.73
[12 (7.70s)]: train_loss=0.66 train_acc=0.76, val_loss=0.75 val_acc=0.74
[13 (8.50s)]: train_loss=0.74 train_acc=0.76, val_loss=0.72 val_acc=0.76
[14 (7.60s)]: train_loss=0.78 train_acc=0.73, val_loss=0.70 val_acc=0.76
[15 (8.10s)]: train_loss=0.69 train_acc=0.74, val_loss=0.74 val_acc=0.73
[16 (7.31s)]: train_loss=0.71 train_acc=0.74, val_loss=0.67 val_acc=0.77
[17 (7.42s)]: train_loss=0.76 train_acc=0.70, val_loss=0.67 val_acc=0.77
[18 (8.07s)]: train_loss=0.60 train_acc=0.80, val_loss=0.67 val_acc=0.76
[19 (8.00s)]: train_loss=0.62 train_acc=0.78, val_loss=0.65 val_acc=0.77
== Total training time 162.99 seconds ==
```

Out[22]:

	train_loss	train_accuracy	val_loss	val_accuracy	epoch_duration
0	2.15	0.29	2.00	0.44	7.78
1	1.83	0.49	1.65	0.64	11.91
2	1.53	0.63	1.40	0.57	8.00
3	1.32	0.57	1.19	0.66	8.69
4	1.19	0.64	1.06	0.68	8.71
5	1.02	0.67	0.96	0.72	9.54
6	0.95	0.72	0.91	0.72	8.75
7	0.87	0.73	0.86	0.71	7.91
8	0.87	0.69	0.82	0.72	8.00
9	0.76	0.75	0.81	0.72	7.39
10	0.71	0.76	0.76	0.74	6.70
11	0.82	0.71	0.76	0.73	6.90
12	0.66	0.76	0.75	0.74	7.70
13	0.74	0.76	0.72	0.76	8.50
14	0.78	0.73	0.70	0.76	7.60
15	0.69	0.74	0.74	0.73	8.10
16	0.71	0.74	0.67	0.77	7.31
17	0.76	0.70	0.67	0.77	7.42
18	0.60	0.80	0.67	0.76	8.07
19	0.62	0.78	0.65	0.77	8.00

Deep Conv model

We will construct a **deep convolutional** network with the following layers:

- Conv2d with 16 kernels of size 5, with `padding='same'`
- MaxPool2d with pooling size of 2
- Conv2d with 32 kernels of size 3, with padding
- MaxPool2d with pooling size of 2
- Flatten the maxpooling output to a vector
- a MLP with hidden layer of 50, and ReLU is used as the activation function, and its output layer produces the 10 dimensional logits vector

In [23]:
```
"✍"
# @workUnit

class DeepCNNModel(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # complete the following
            nn.Conv2d(1, 16, 5, padding='same'),
            nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.MaxPool2d(2),
            nn.Flatten(),
            nn.LazyLinear(50),
            nn.ReLU(),
            nn.LazyLinear(10)

        )
    def forward(self, x):
        return self.model(x)
```

In [24]:
```
"🔒"
# @check
# @title: deep cnn architecture

model_cnn_deep = DeepCNNModel()
summary(model_cnn_deep, xs);
```

```
==========================================================================
                        Kernel Shape      Output Shape  Params Mult-Adds
Layer
0_model.Conv2d_0       [1, 16, 5, 5]  [32, 16, 28, 28]   416.0    313.6k
1_model.MaxPool2d_1                -  [32, 16, 14, 14]       -         -
2_model.Conv2d_2      [16, 32, 3, 3]  [32, 32, 14, 14]   4.64k  903.168k
3_model.MaxPool2d_3                -    [32, 32, 7, 7]       -         -
4_model.Flatten_4                  -        [32, 1568]       -         -
5_model.LazyLinear_5      [1568, 50]          [32, 50]  78.45k     78.4k
6_model.ReLU_6                     -          [32, 50]       -         -
7_model.LazyLinear_7        [50, 10]          [32, 10]   510.0     500.0
--------------------------------------------------------------------------
                         Totals
Total params            84.016k
Trainable params        84.016k
Non-trainable params        0.0
Mult-Adds             1.295668M
==========================================================================
```

Training the deep CNN network

We train the deep convolutional network.

You should expect:

- Total training time: 250 seconds
- Validation accuracy should be close to or exceeds 80%.

In [25]:
```
"🔒"
reload(trainer_lib)
```

```
trainer = trainer_lib.Trainer(model_cnn_deep, train_dataloader, val_dataloader)
trainer.reset()
model_cnn_deep_log = trainer.train(epochs=20, max_batches=10)
model_cnn_deep_log.round(2)
```

```
[0 (19.28s)]: train_loss=2.15 train_acc=0.25, val_loss=1.93 val_acc=0.43
[1 (18.22s)]: train_loss=1.69 train_acc=0.45, val_loss=1.38 val_acc=0.52
[2 (18.51s)]: train_loss=1.17 train_acc=0.55, val_loss=1.02 val_acc=0.65
[3 (19.19s)]: train_loss=0.93 train_acc=0.68, val_loss=0.91 val_acc=0.62
[4 (23.29s)]: train_loss=0.88 train_acc=0.68, val_loss=1.02 val_acc=0.64
[5 (14.40s)]: train_loss=0.96 train_acc=0.65, val_loss=0.84 val_acc=0.71
[6 (15.44s)]: train_loss=0.84 train_acc=0.74, val_loss=0.81 val_acc=0.71
[7 (18.55s)]: train_loss=0.84 train_acc=0.69, val_loss=0.75 val_acc=0.73
[8 (14.30s)]: train_loss=0.75 train_acc=0.72, val_loss=0.74 val_acc=0.73
[9 (16.60s)]: train_loss=0.73 train_acc=0.71, val_loss=0.71 val_acc=0.74
[10 (17.11s)]: train_loss=0.72 train_acc=0.73, val_loss=0.73 val_acc=0.74
[11 (14.49s)]: train_loss=0.65 train_acc=0.74, val_loss=0.67 val_acc=0.76
[12 (15.90s)]: train_loss=0.68 train_acc=0.77, val_loss=0.65 val_acc=0.77
[13 (17.42s)]: train_loss=0.56 train_acc=0.79, val_loss=0.72 val_acc=0.76
[14 (14.74s)]: train_loss=0.65 train_acc=0.79, val_loss=0.66 val_acc=0.76
[15 (13.81s)]: train_loss=0.64 train_acc=0.78, val_loss=0.63 val_acc=0.77
[16 (15.84s)]: train_loss=0.59 train_acc=0.78, val_loss=0.63 val_acc=0.77
[17 (16.01s)]: train_loss=0.63 train_acc=0.75, val_loss=0.63 val_acc=0.78
[18 (15.18s)]: train_loss=0.59 train_acc=0.78, val_loss=0.62 val_acc=0.79
[19 (15.00s)]: train_loss=0.63 train_acc=0.77, val_loss=0.61 val_acc=0.77
== Total training time 333.38 seconds ==
```

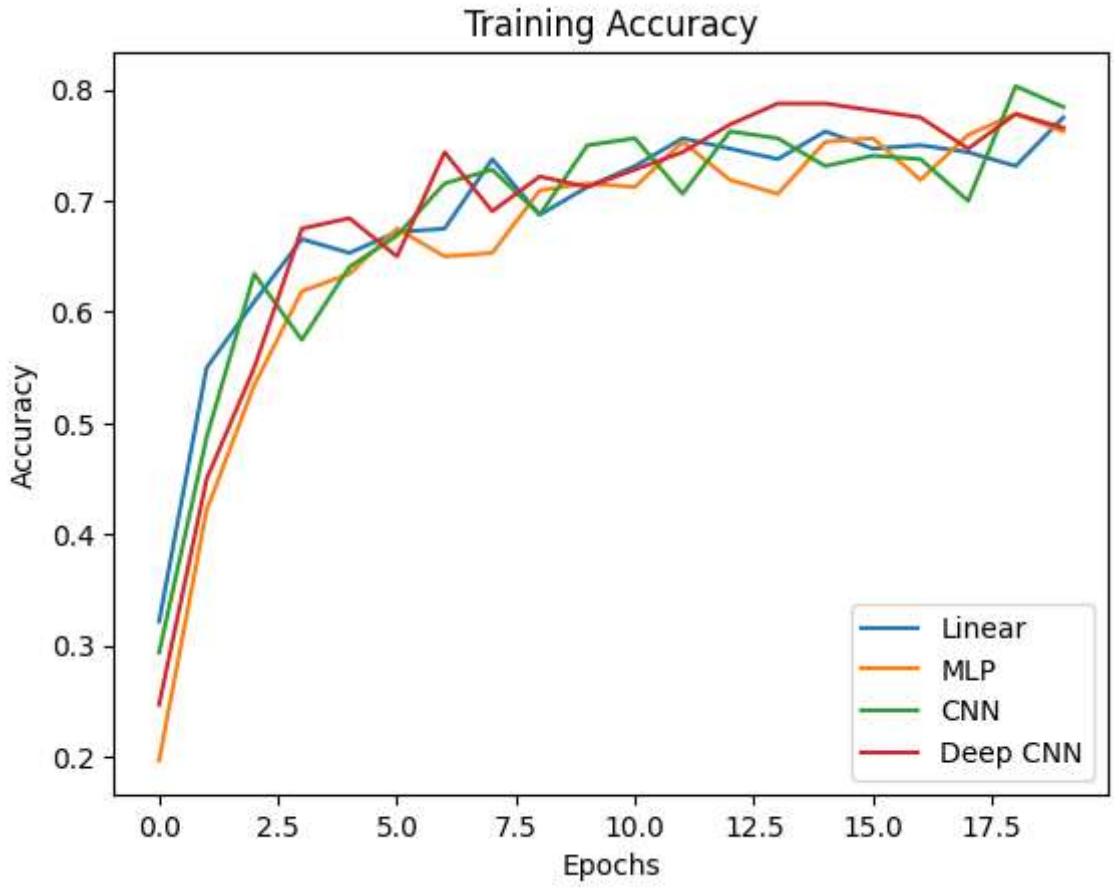
Out[25]:

	train_loss	train_accuracy	val_loss	val_accuracy	epoch_duration
0	2.15	0.25	1.93	0.43	19.28
1	1.69	0.45	1.38	0.52	18.22
2	1.17	0.55	1.02	0.65	18.51
3	0.93	0.68	0.91	0.62	19.19
4	0.88	0.68	1.02	0.64	23.29
5	0.96	0.65	0.84	0.71	14.40
6	0.84	0.74	0.81	0.71	15.44
7	0.84	0.69	0.75	0.73	18.55
8	0.75	0.72	0.74	0.73	14.30
9	0.73	0.71	0.71	0.74	16.60
10	0.72	0.73	0.73	0.74	17.11
11	0.65	0.74	0.67	0.76	14.49
12	0.68	0.77	0.65	0.77	15.90
13	0.56	0.79	0.72	0.76	17.42
14	0.65	0.79	0.66	0.76	14.74
15	0.64	0.78	0.63	0.77	13.81
16	0.59	0.78	0.63	0.77	15.84
17	0.63	0.75	0.63	0.78	16.01
18	0.59	0.78	0.62	0.79	15.18
19	0.63	0.77	0.61	0.77	15.00

Plot

The following plots the training accuracy.

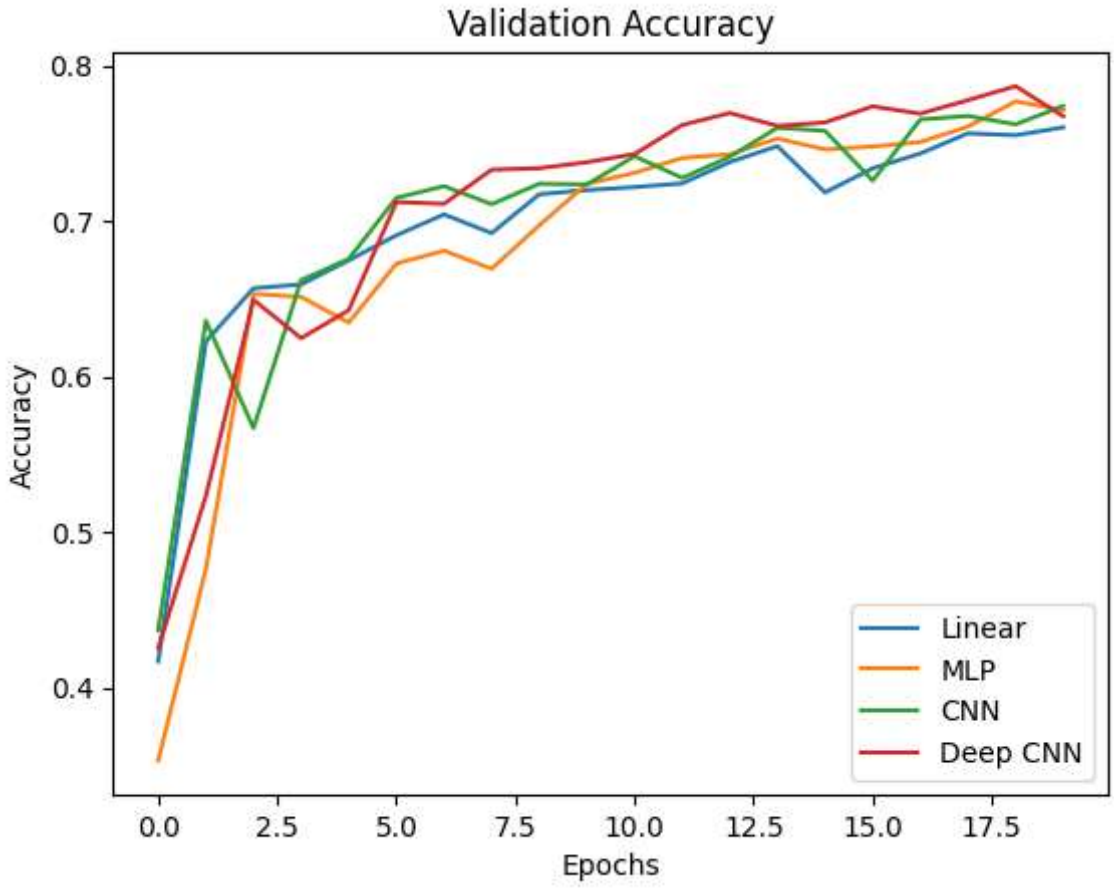
In [26]:
```
"🔒"
plt.figure()
plt.plot(model_linear_log.index, model_linear_log.train_accuracy)
plt.plot(model_mlp_log.index, model_mlp_log.train_accuracy)
plt.plot(model_cnn_linear_log.index, model_cnn_linear_log.train_accuracy)
plt.plot(model_cnn_deep_log.index, model_cnn_deep_log.train_accuracy)

plt.title('Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Linear', 'MLP', 'CNN', 'Deep CNN']);
```



Plot the validation accuracy

Make sure you properly label the plot and the axes.

In [28]:
```
"✍"
# @workUnit

#
# Generate the validation accuracy of the different models
#
plt.figure()
plt.plot(model_linear_log.index, model_linear_log.val_accuracy)
plt.plot(model_mlp_log.index, model_mlp_log.val_accuracy)
plt.plot(model_cnn_linear_log.index, model_cnn_linear_log.val_accuracy)
plt.plot(model_cnn_deep_log.index, model_cnn_deep_log.val_accuracy)

plt.title('Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Linear', 'MLP', 'CNN', 'Deep CNN']);
```


Validation Accuracy
Accuracy
Epochs
0.8
0.7
0.6
0.5
0.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Linear
MLP
CNN
Deep CNN

In [ ]: