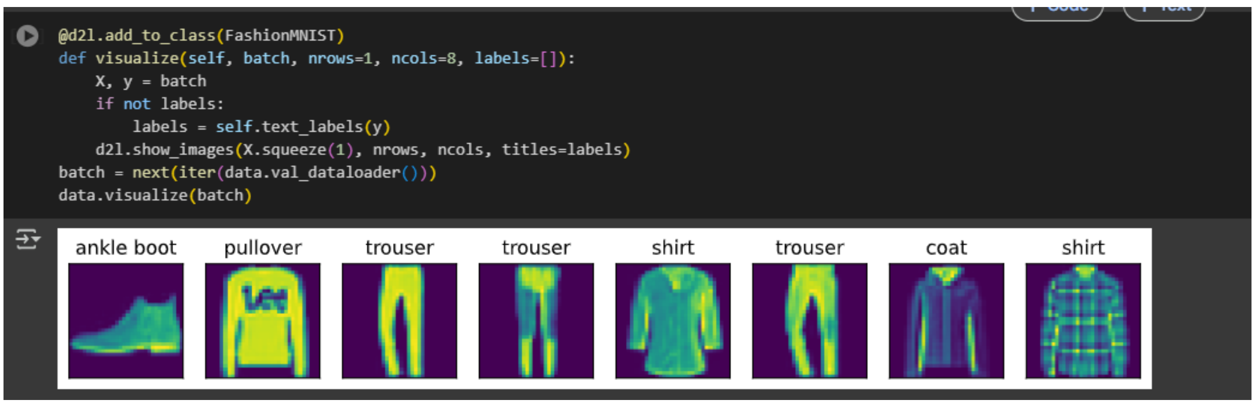
Carson Logas 10979423

Part 1: Linear Neural Networks

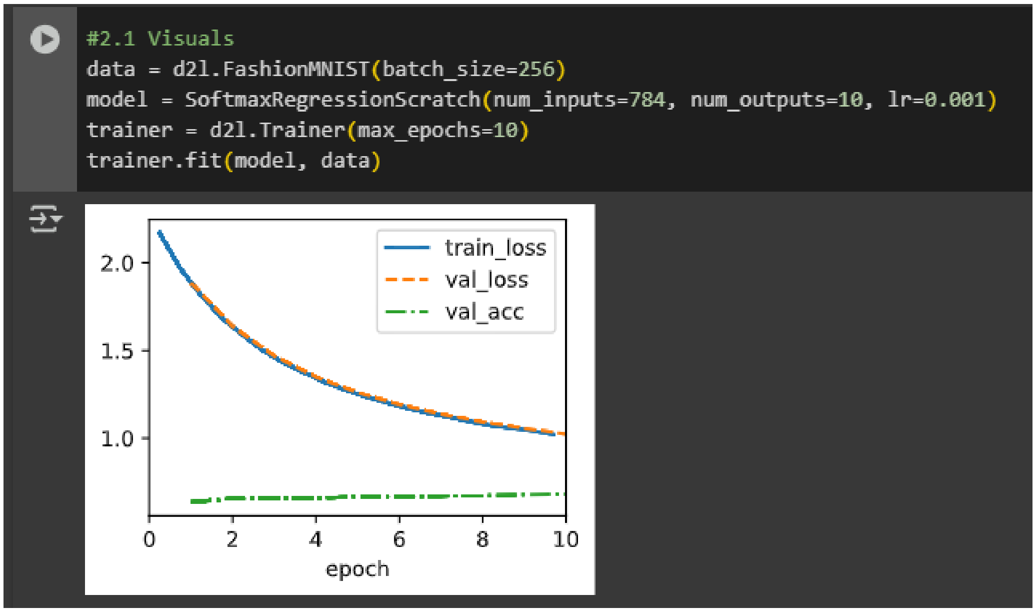
Q1: Visualize 10 images from Fashion-MNIST



```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```

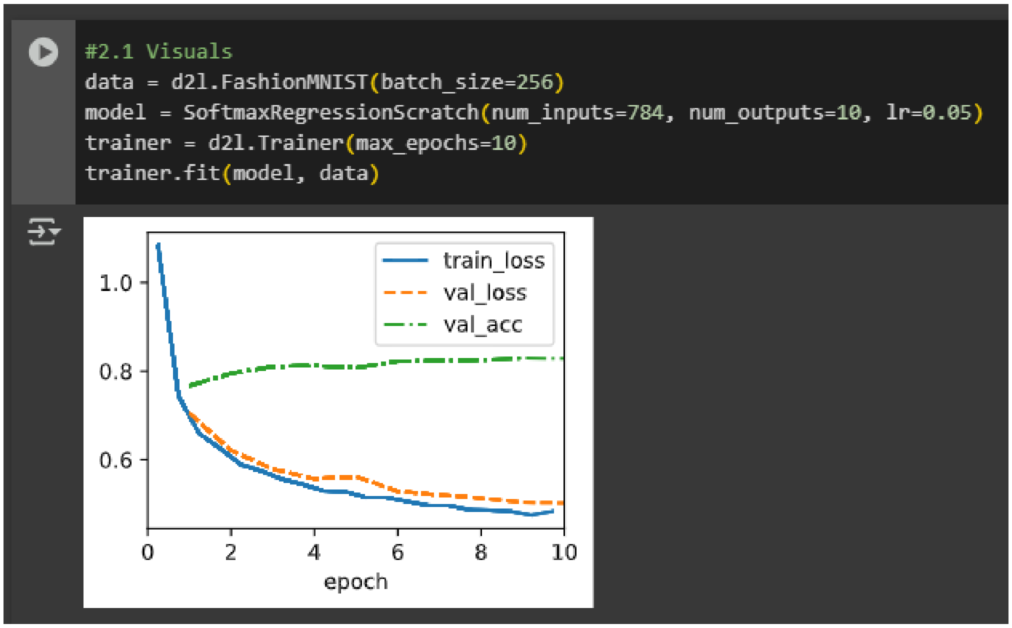
Q2: Hyperparameter Analysis

2.1 Different Learning Rates: - Small LR (0.001): very slow convergence, high validation loss.



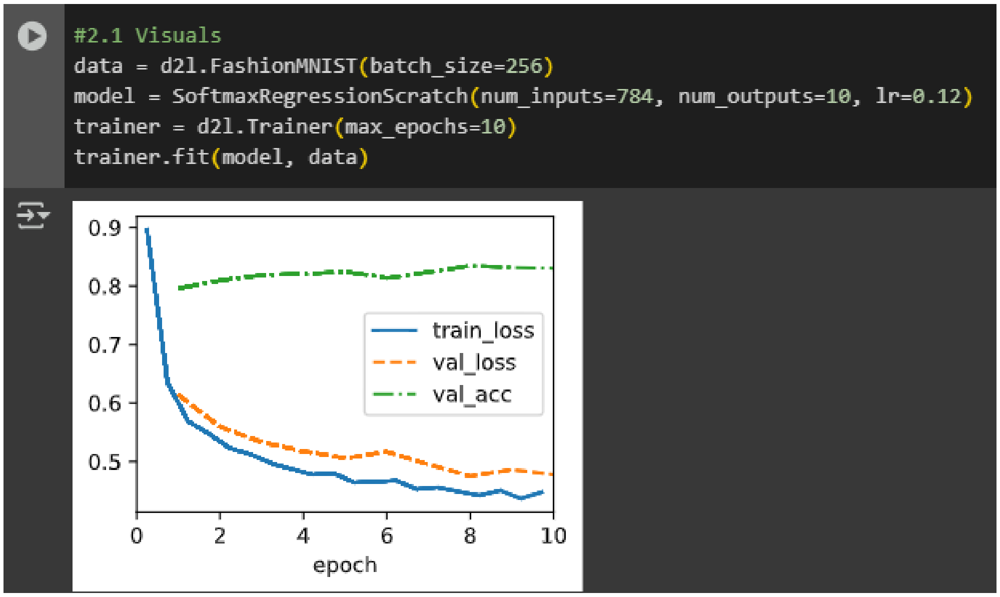
```
#2.1 Visuals
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.001)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

Medium LR (0.05–0.1, .05): fast convergence, lowest validation loss, stable training.



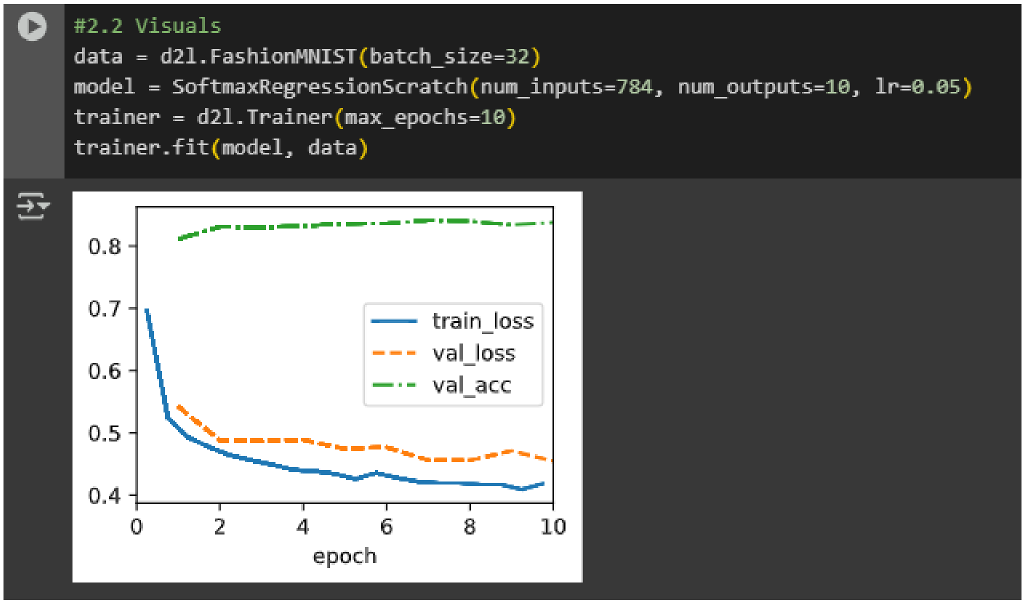
```
#2.1 Visuals
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.05)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

Large LR (>0.1, .12): unstable, swaying validation loss.



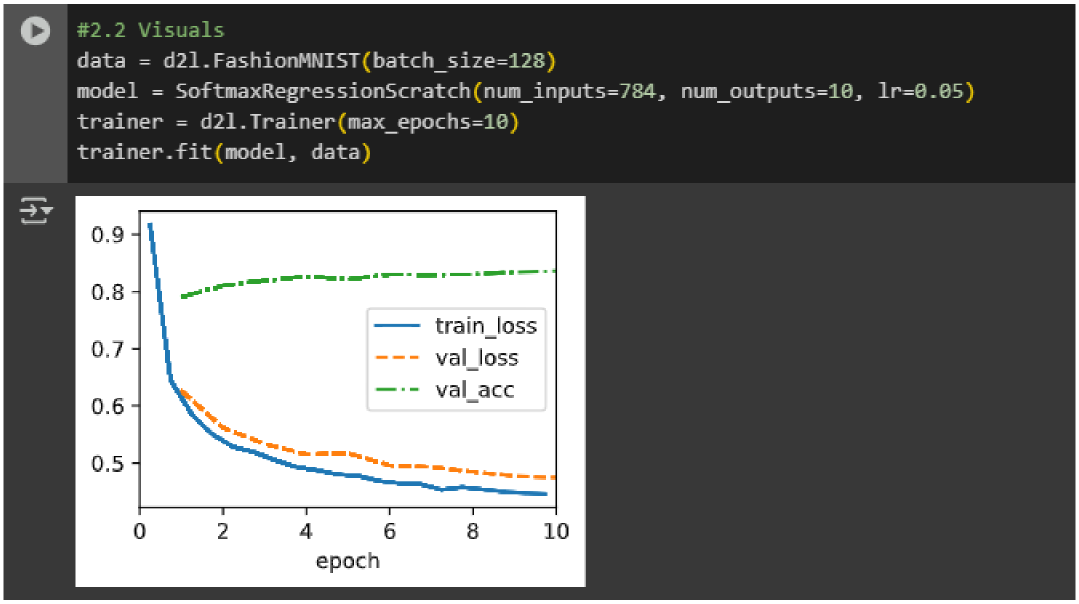
```
#2.1 Visuals
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.12)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

2.2 Different Batch Sizes, Using LR = 0.05: - Small batch (32): faster initial convergence but noisy loss curves; generalization okay.



```
#2.2 Visuals
data = d2l.FashionMNIST(batch_size=32)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.05)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

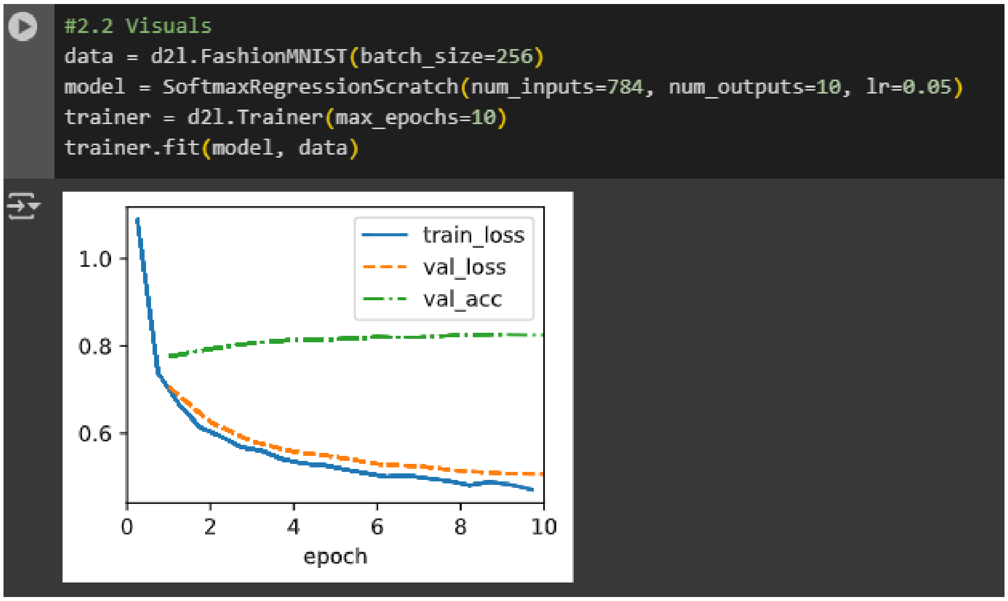
Medium batch (128): smooth training, good balance of speed and accuracy.



```
#2.2 Visuals
data = d2l.FashionMNIST(batch_size=128)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.05)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

Large batch (256): stable curves, but slightly slower early improvement;

sometimes lower final accuracy.



```
#2.2 Visuals
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.05)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

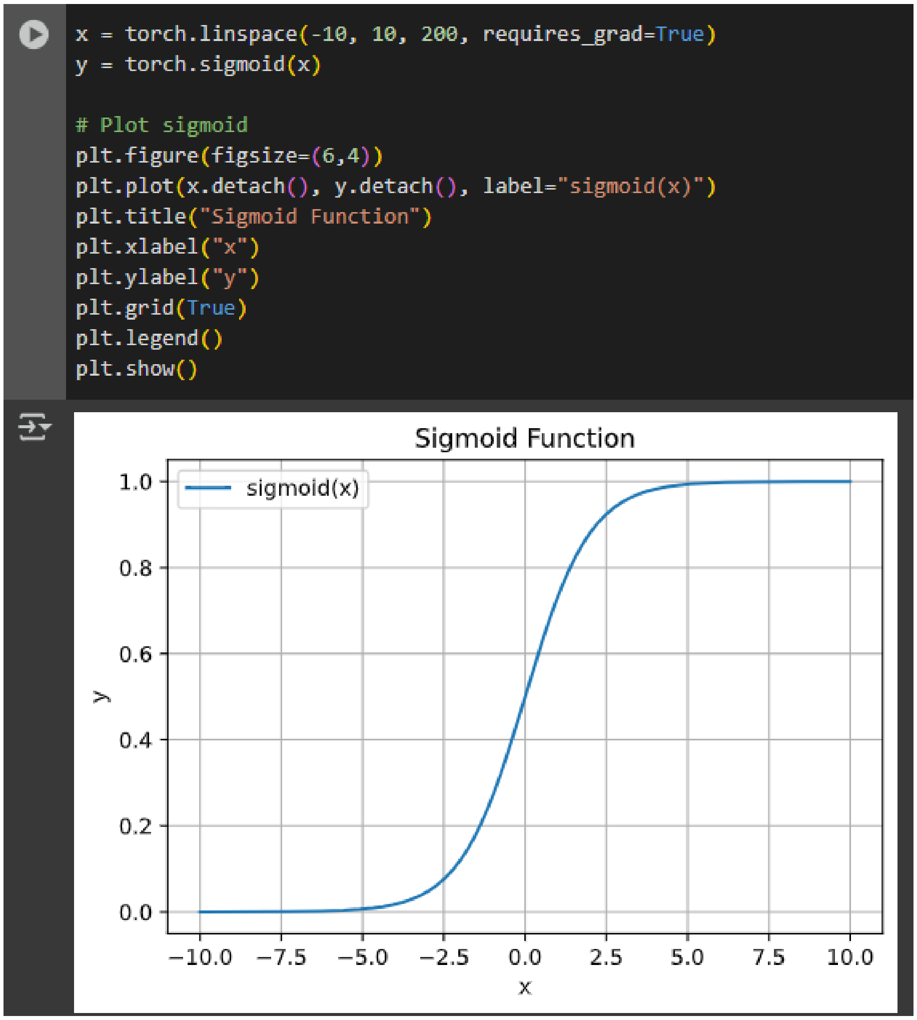
Q3:

The softmax function can do worse from numerical instability because of how exponentials behave. If one of the input values is very large and positive, the exponential of that value can exceed the limits of what floating point numbers can store, causing to overflow and results like “infinity.” Now, if an input value is very large and negative, the exponential becomes so close to zero that it rounds down to zero, which is called underflow. Both of these situations cause the computed probabilities to become unreliable.

A common way to fix this is to subtract the largest input value from all of the inputs before taking exponentials. This works because softmax only depends on the relative differences between the inputs, not their absolute size, so shifting them all down by the same amount does not change the final probabilities. By doing this, the biggest input is always reduced to zero, which keeps its exponential at a safe size of one and prevents overflow. This fix does not completely remove the risk of underflow, since very small values can still shrink down to zero, but those values correspond to classes with almost no probability anyway. For this reason, subtracting the maximum is the standard approach and is considered stable enough for nearly all applications.

Part 2: Multilayer Perceptrons

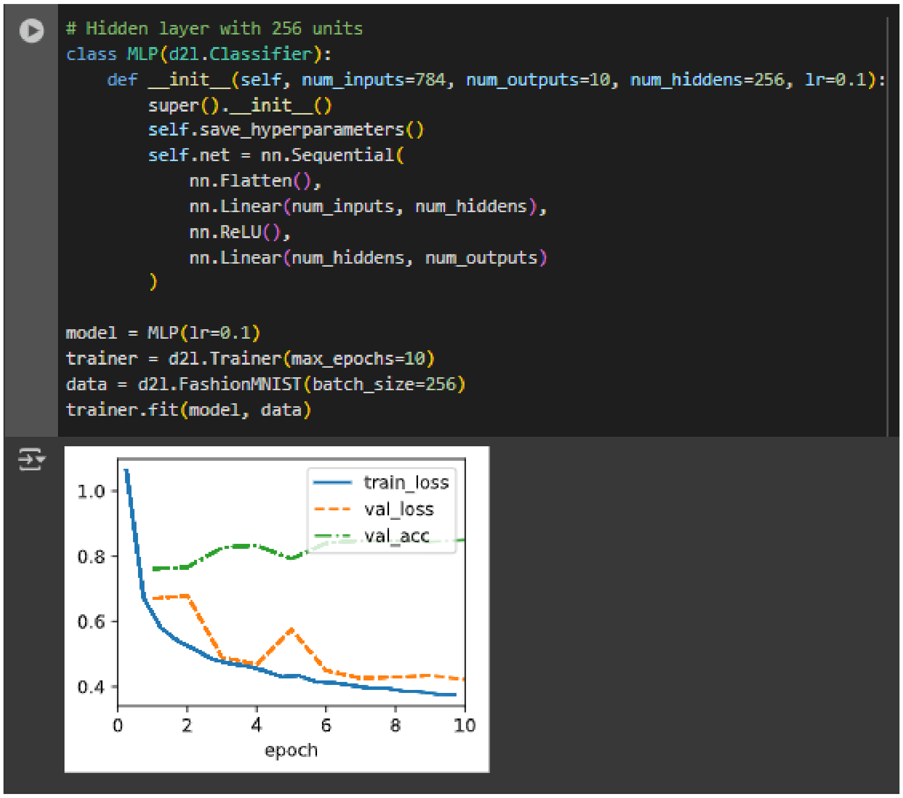
Q1: Vanishing Gradients with Sigmoid



```
x = torch.linspace(-10, 10, 200, requires_grad=True)
y = torch.sigmoid(x)

# Plot sigmoid
plt.figure(figsize=(6,4))
plt.plot(x.detach(), y.detach(), label="sigmoid(x)")
plt.title("Sigmoid Function")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```

Q2: Implementation of MLPs

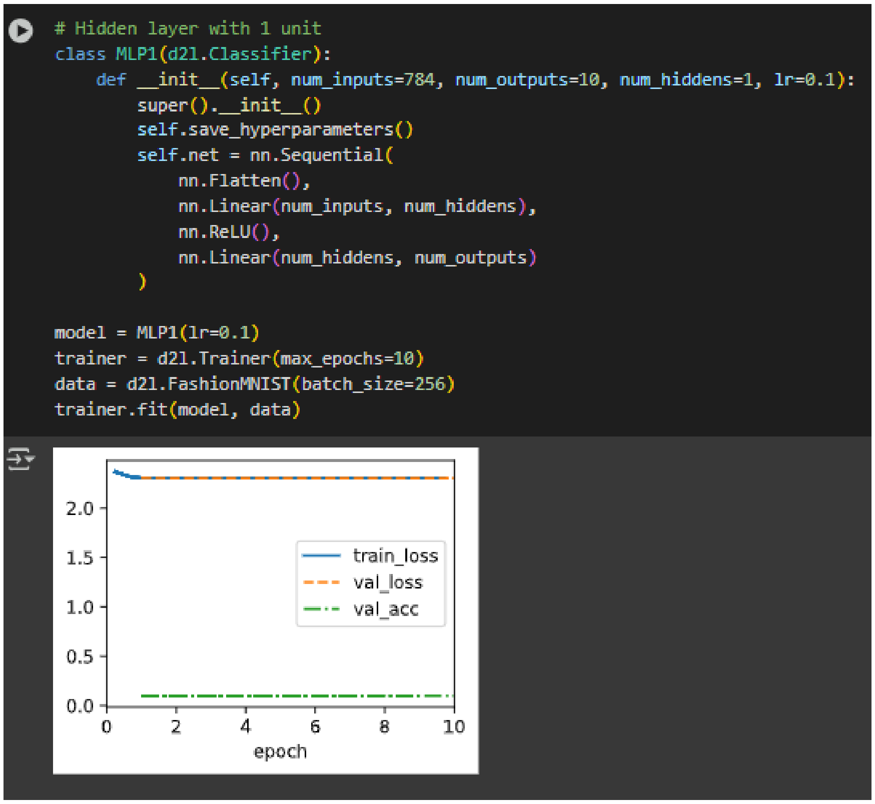


```
# Hidden layer with 256 units
class MLP(d2l.Classifier):
    def __init__(self, num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(num_inputs, num_hiddens),
            nn.ReLU(),
            nn.Linear(num_hiddens, num_outputs)
        )

model = MLP(lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=256)
trainer.fit(model, data)
```

Image Above: Training and validation curves for an MLP with one hidden layer of 256 units. The model shows stable decreasing training and validation loss, along with increasing validation accuracy, showing that the hidden layer provides enough nonlinear space to learn useful patterns from the data.

Image Below: Training and validation curves for an MLP with one hidden layer of only 1 unit. The model fails to improve validation accuracy and shows poor loss behavior, showing that a single hidden neuron does not provide representational power and essentially reduces the network to a near linear classifier.



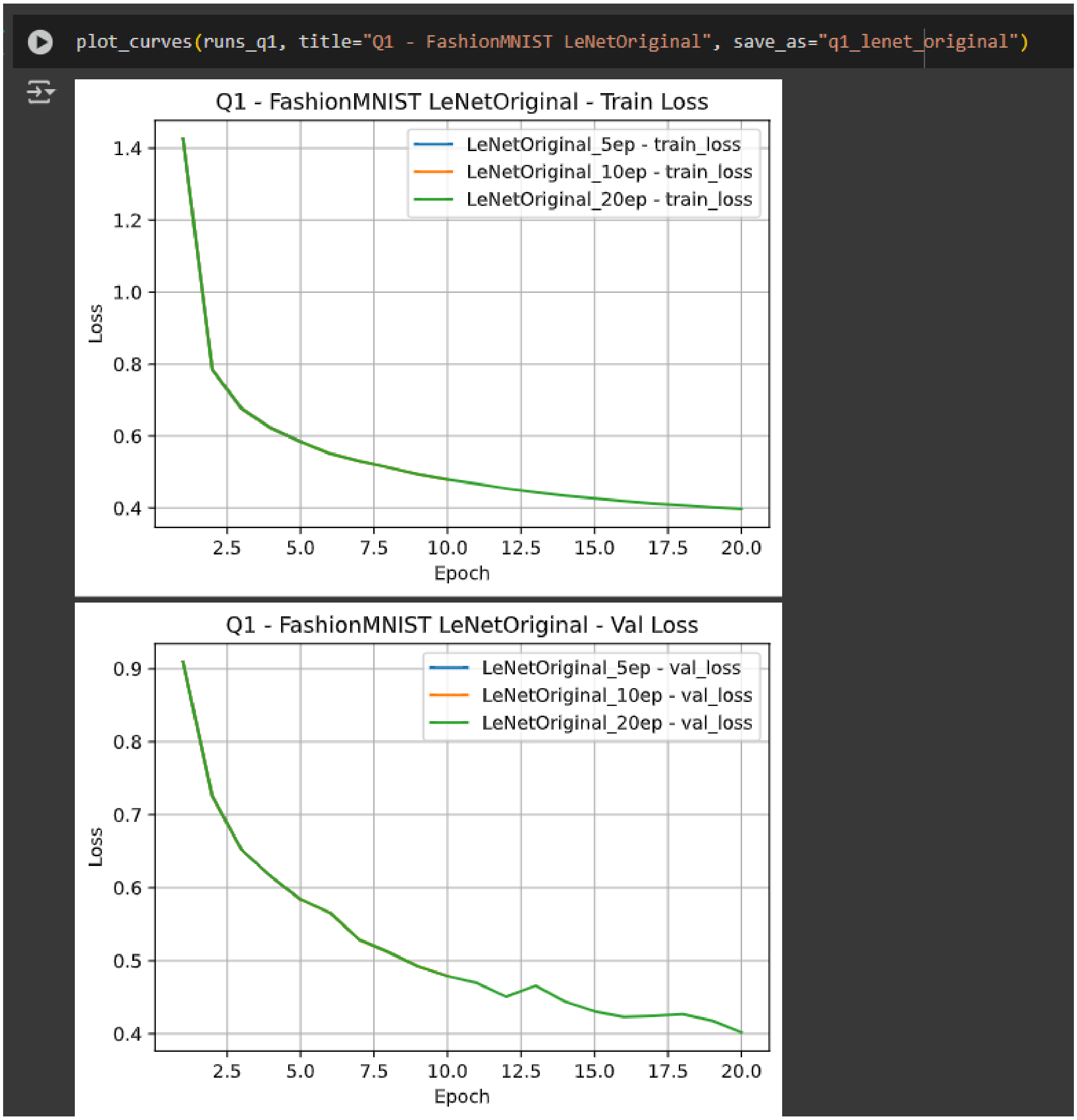
```
# Hidden layer with 1 unit
class MLP1(d2l.Classifier):
    def __init__(self, num_inputs=784, num_outputs=10, num_hiddens=1, lr=0.1):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(num_inputs, num_hiddens),
            nn.ReLU(),
            nn.Linear(num_hiddens, num_outputs)
        )

model = MLP1(lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=256)
trainer.fit(model, data)
```

Q3: When the model is just predicting, memory use is small because it only needs the weights and the output of each layer one at a time, and then it throws those outputs away. Training takes way more memory because the model has to keep all the outputs from every layer to calculate gradients and update weights. It also stores the gradients themselves and, if using adaptive moment estimation, extra copies of the weights. That’s why training can use several times more memory than prediction, especially with deeper networks or bigger batch sizes

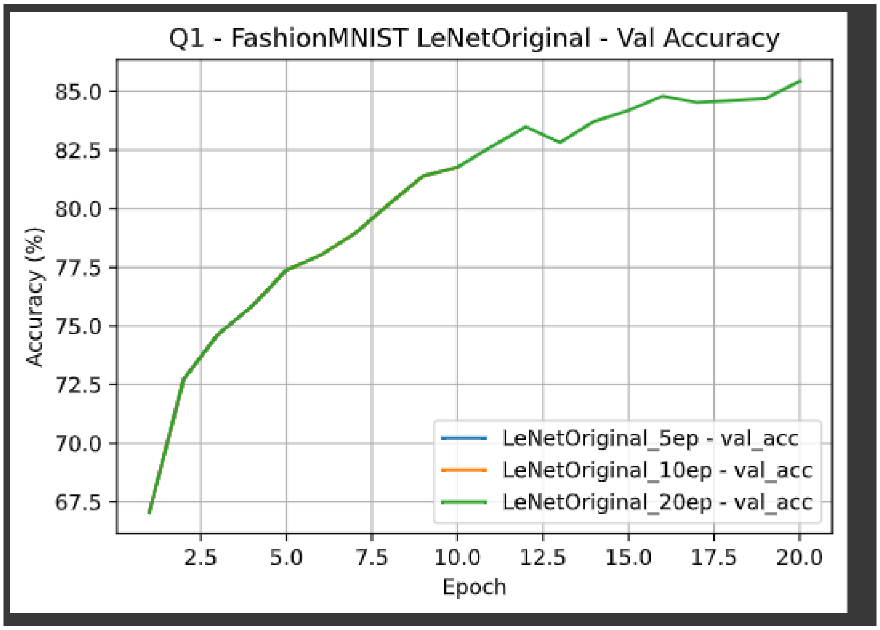
Part 3: LeNet (CNNs)

Q1: Different Epochs (5, 10, 20)



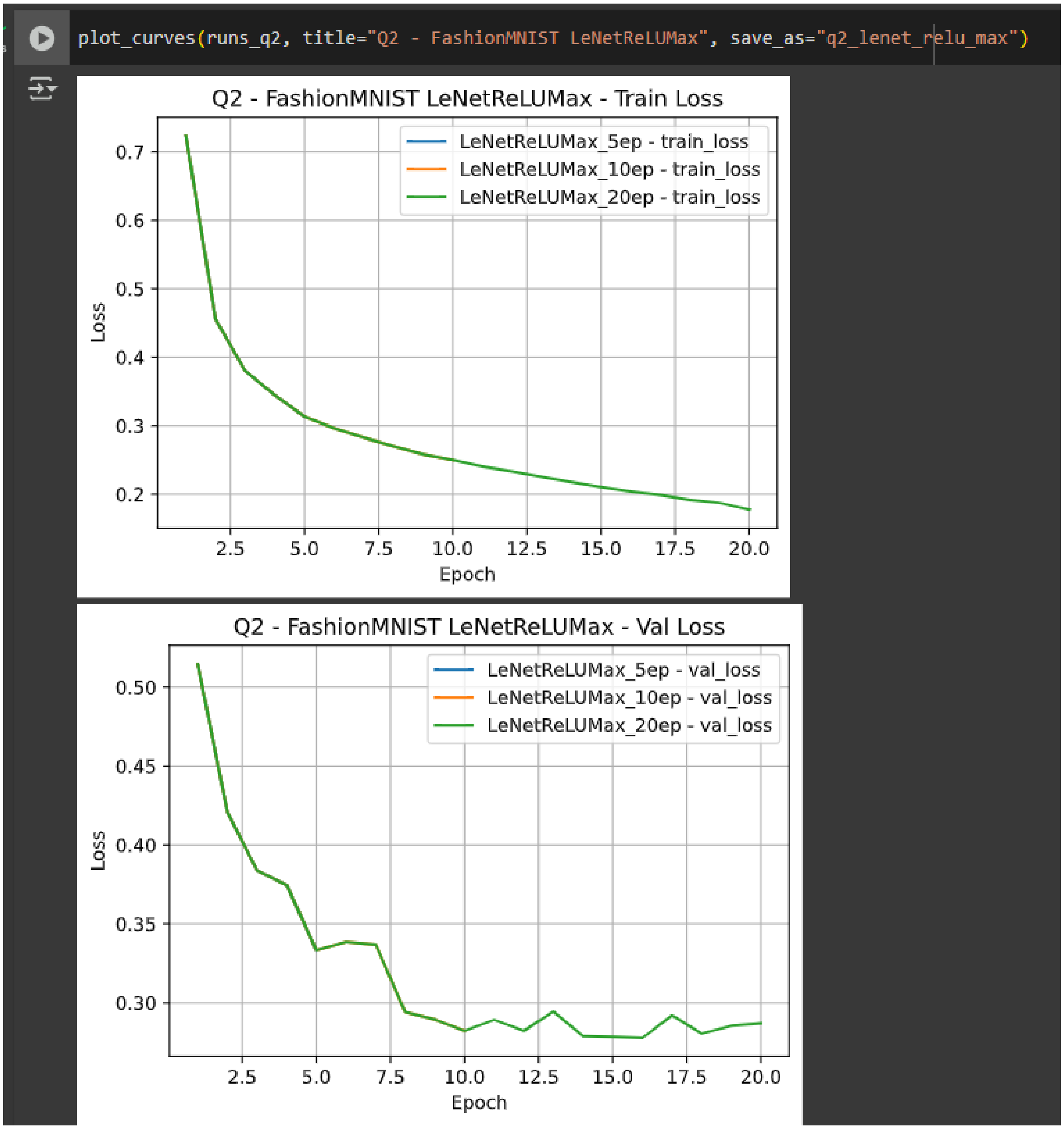
```
plot_curves(runs_q1, title="Q1 - FashionMNIST LeNetOriginal", save_as="q1_lenet_original")
```

Q1 - FashionMNIST LeNetOriginal - Train Loss
LeNetOriginal_5ep - train_loss
LeNetOriginal_10ep - train_loss
LeNetOriginal_20ep - train_loss
Loss
1.4
1.2
1.0
0.8
0.6
0.4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch
Q1 - FashionMNIST LeNetOriginal - Val Loss
LeNetOriginal_5ep - val_loss
LeNetOriginal_10ep - val_loss
LeNetOriginal_20ep - val_loss
Loss
0.9
0.8
0.7
0.6
0.5
0.4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch



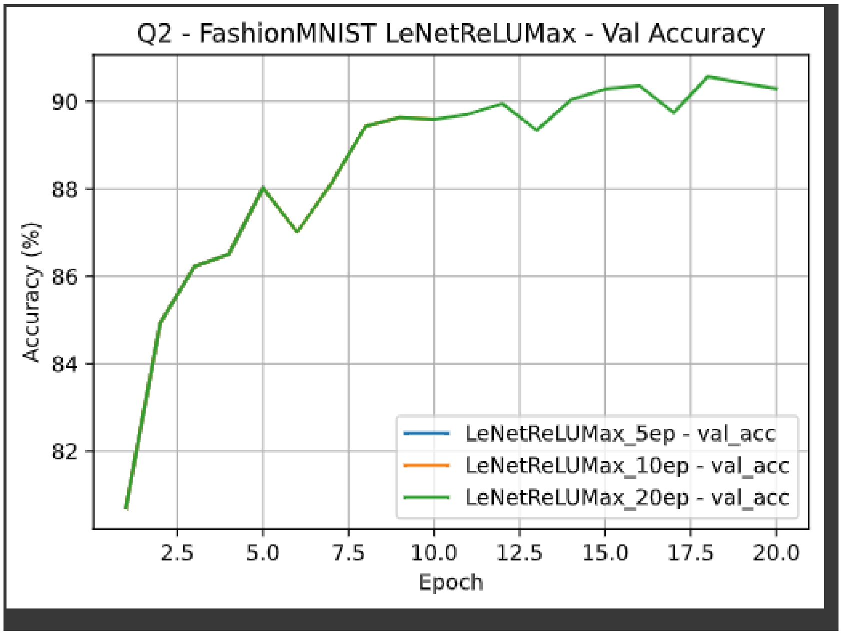
Training LeNet on Fashion-MNIST for 5, 10, and 20 epochs shows improvement with more epochs. Training and validation loss drop often, while validation accuracy goes from the low 70s at 5 epochs to the mid 80s by 20 epochs. Most of the gains occur by 10 epochs, with smaller improvements after that, indicating bad returns and the start of overfitting if training is continued.

Q2: Modified LeNet (ReLU + MaxPool)



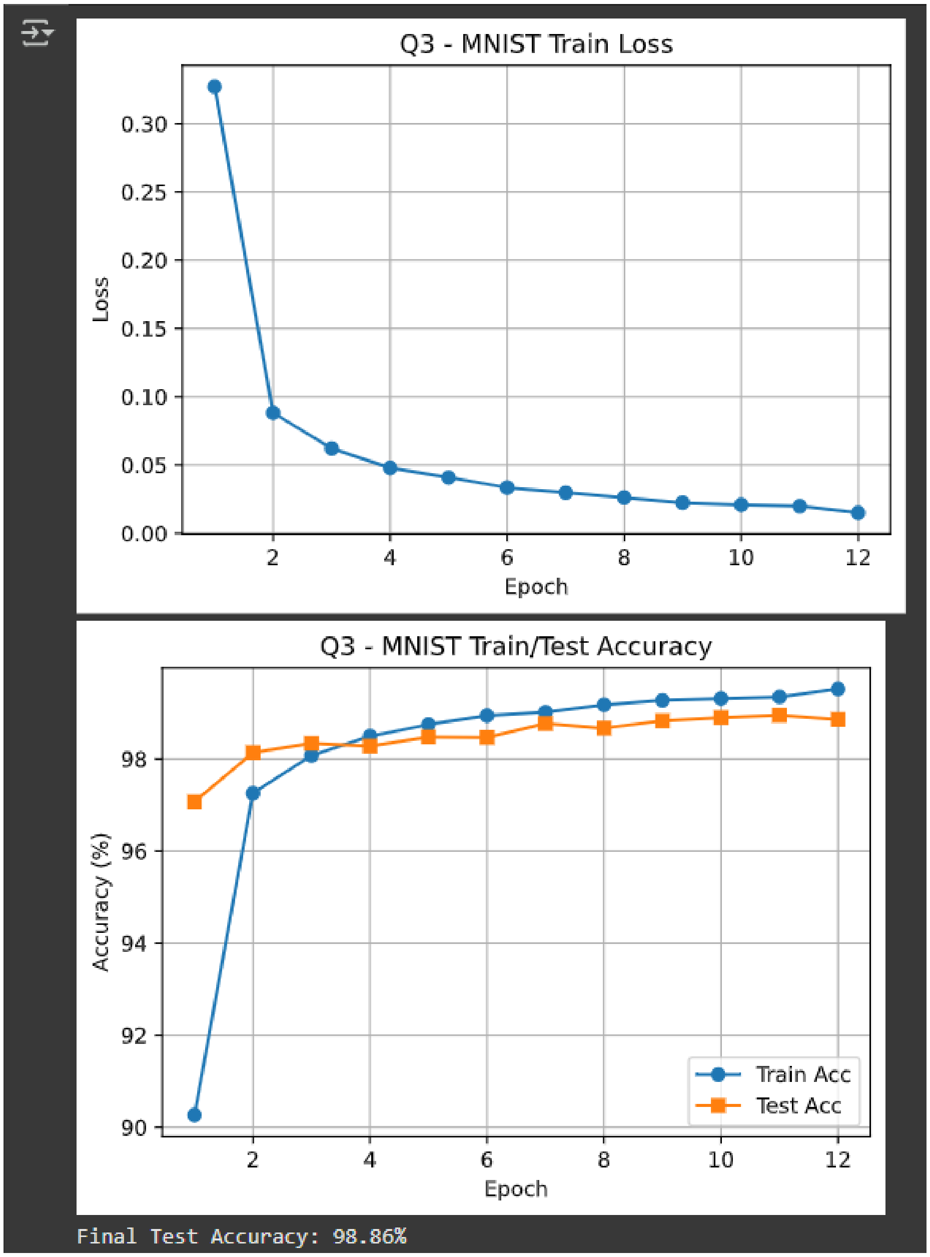
```
plot_curves(runs_q2, title="Q2 - FashionMNIST LeNetReLUMax", save_as="q2_lenet_relu_max")
```

Q2 - FashionMNIST LeNetReLUMax - Train Loss
LeNetReLUMax_5ep - train_loss
LeNetReLUMax_10ep - train_loss
LeNetReLUMax_20ep - train_loss
Loss
0.7
0.6
0.5
0.4
0.3
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch
Q2 - FashionMNIST LeNetReLUMax - Val Loss
LeNetReLUMax_5ep - val_loss
LeNetReLUMax_10ep - val_loss
LeNetReLUMax_20ep - val_loss
Loss
0.50
0.45
0.40
0.35
0.30
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch



The modified LeNet with ReLU and max pooling works better than the original. Both training and validation loss drop faster and end up lower, and validation accuracy rises more quickly, reaching about 90% by 20 epochs compared to the mid 80s with sigmoid and average pooling. Most of the improvement shows up by 10 epochs, but the ReLU+Max model still gets a bit better by 20. This happens because ReLU avoids vanishing gradients and max pooling keeps the strongest parts, so the network trains faster and generalizes better than the original.

Q3: Training on MNIST



```
Final Test Accuracy: 98.86%
```

Training LeNet with ReLU and max pooling on the MNIST dataset produces very strong results. The training loss drops rapidly within the first few epochs and continues to decrease steadily, while both training and test accuracy climb quickly above 97% by epoch 2 and flatten out near 99% by the end of training. The final test accuracy of about 98.9% highlights how well the model generalizes on this simpler dataset compared to FashionMNIST, since handwritten digits are less complex and more consistent in structure than clothing images. Overall, MNIST confirms the effectiveness of the modified LeNet architecture, achieving high accuracy with fast convergence and minimal overfitting.