AWS Logo

Application of Deep Learning to Text and Image Data

Module 1, Lab 2: Creating a Multilayer Perceptron and Using Dropout Layers

In this notebook, you will implement a simple neural network with multiple layers and analyze the training process. You will then implement dropout layers to prevent overfitting of the neural network.

Multilayer perceptron

The simplest feed-forward neural network architecture is a multilayer perceptron (MLP). An MLP is characterized by several layers of input neurons that are fully connected. Forming an MLP requires at least three layers: input layer, hidden layer, and output layer. An MLP uses backpropagation to train the network.

Dropout layers

To prevent overfitting of neural networks, it's possible to randomly drop a certain percentage of the neurons (or nodes) in the input and hidden layers. This has proven to be an effective technique for regularization and preventing the coadaptation of neurons (for neurons that show correlated behavior). The dropout layer only applies during training of the neural network. Neurons aren't dropped when making predictions (inference).

You will learn the following:

- How to define a single dense-layer neural network model
- How to train the neural network
- Why dropout layers are important
- How to add a dropout layer

You will be presented with activities throughout the notebook:

Activity

No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.

Index

Dataset

The Fashion-MNIST dataset consists of 28x28 (=784) pixel grayscale images from 10 categories. The dataset has 6,000 images in each category for the training dataset and 1,000 in each category for the test dataset.

Your task is to build a classifier that maps the images to their categories. You will use PyTorch predefined layers and the default trainers for a swift and efficient implementation of an MLP.

In [2]:
```
# Install libraries
!pip install -U -q -r requirements.txt
```

In [3]:
```
# Import system library and append path
import sys
sys.path.insert(1, "..")

# Import utility functions that provide answers to challenges
from MLUDTI_EN_M1_Lab2_quiz_questions import *

# Import PyTorch library and plotting library
import torch
import torchvision
from torch import nn
from torchvision import transforms
from torch.utils import data
import matplotlib.pyplot as plt

# Load the image dataset with the torch helper functions

mnist_train = torchvision.datasets.FashionMNIST(
    root="data", train=True, transform=transforms.ToTensor(), download=True
)  # ToTensor converts the image data from PIL type to 32-bit floating point
```

```

mnist_val = torchvision.datasets.FashionMNIST(
    root="data", train=False, transform=transforms.ToTensor(), download=True
)  # ToTensor converts the image data from PIL type to 32-bit floating point

# Pass batches of images to the neural network
batch_size = 256

# To load images in batches, you need the DataLoader helper function
training_loader = data.DataLoader(mnist_train, batch_size, shuffle=True)
validation_loader = data.DataLoader(mnist_val, batch_size, shuffle=False)
```

```
Matplotlib is building the font cache; this may take a moment.
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
n-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
n-images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:02<00:00, 11830597.73it/s]
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/Fashion
MNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
n-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
n-labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 198744.68it/s]
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/Fashion
MNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
-images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:01<00:00, 2259287.93it/s]
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionM
NIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
-labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 13094164.34it/s]
Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/FashionM
NIST/raw
```

Look at some of the images to see what is in the dataset.

In [6]:
```
# To display the images, you need a function that plots them
def show_images(imgs, num_rows, num_cols, titles=None, scale=2):
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.permute(1, 2, 0).numpy(), cmap="gray")
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes

# You can update the num_rows and num_cols variables to change the number of
for data, label in training_loader:
    show_images(data, 4, 4)
    break
```

Define the model

Now that you have imported and reviewed the data, you need to build a linear model with a single dense layer that takes in a vector of length 784 (the number of input features) and returns another vector of length 10 (the number of output classes). Remember that you need to initialize weights and biases to get a first prediction and evaluation of the output that the MLP produces. A good starting point is to use a normal distribution for weights and zeros for biases.

In [7]:
```
# Specify how many classes to predict (this needs to match the labels)
in_features = 784
out_classes = 10

# Single-layer network; flatten is required because you are working with ima
mlp = nn.Sequential(
    nn.Flatten(),
    nn.Linear(
        in_features, out_classes
    ),  # Use CrossEntropyLoss later with SoftMax built in, so no need to ac
)

# Initialize the network
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)
        nn.init.zeros_(m.bias)

mlp.apply(init_weights)
```

Out[7]:
```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=10, bias=True)
)
```

Try it yourself!

Activity

To test your understanding of neural net architectures, run the following cell.

In [8]:
```
# Run this cell to display the question and check your answer
question_1
```

Out[8]:

Which option would you use to create a single-layer neural network with 3 output classes for 16x16 images?

nn.Sequential(nn.Flatten(), nn.Linear(256, 1))

nn.Sequential(nn.Flatten(), nn.Linear(16, 3))

nn.Sequential(nn.Flatten(), nn.Linear(256, 3))

Submit

In [9]:
```
# Display the initial values of the w and b
weight, bias = list(mlp.parameters())

# Print weight and bias tensors
print("Weights:")
print(weight)
print("\nBiases:")
print(bias)
```

```
Weights:
Parameter containing:
tensor([[-0.0101, -0.0164, -0.0061,  ...,  0.0134, -0.0005,  0.0120],
        [ 0.0227,  0.0006,  0.0023,  ..., -0.0009, -0.0151, -0.0158],
        [ 0.0029,  0.0038,  0.0207,  ..., -0.0096,  0.0089,  0.0126],
        ...,
        [ 0.0203,  0.0004,  0.0031,  ...,  0.0099,  0.0154, -0.0139],
        [-0.0192, -0.0003,  0.0032,  ..., -0.0152,  0.0040,  0.0023],
        [ 0.0127,  0.0169, -0.0024,  ...,  0.0078, -0.0073,  0.0103]],
       requires_grad=True)

Biases:
Parameter containing:
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], requires_grad=True)
```

Now that everything is set up, test how well the untrained network performs when making predictions on the test dataset.

In [12]:
```
# Look at 10 predictions in the first batch of data in the training loader
for i, (data, label) in enumerate(training_loader):
    pred = mlp(data)
    print("Predictions:")
    print(torch.softmax(pred, dim=1).argmax(axis=1)[:10])
    print("\nTrue labels:")
    print(label[:10])
    break
```

```
Predictions:
tensor([1, 5, 8, 6, 3, 5, 3, 6, 8, 7])

True labels:
tensor([0, 7, 2, 1, 5, 1, 9, 1, 8, 5])
```

As you can see, the model appears to be randomly guessing. Think about why the predictions are random.

You might recall that you generated a normal distribution for weights and set the biases to zero. Those values have not been updated because you have not performed any training yet. While the code cell above doesn't create good predictions, you can use it to verify that the general architecture of the model works.

Now you are ready to train the neural network.

Train the neural network

The training loop is similar to what you built in the previous lab. The main difference is that you will use `torch.optim` to complete the optimization algorithm. You will learn about different optimizers later in the course. For now, use the well-known stochastic gradient descent (SGD).

In [13]:
```
# Determine if a GPU resource is available; otherwise, use CPU.
device = "cuda" if torch.cuda.is_available() else "cpu"

# This is a multiclass classification, so you want to use nn.CrossEntropyLos
criterion = nn.CrossEntropyLoss()
```

First, you need to write a function to train the neural network. When you imported the data, you broke it into batches, so you need to include a loop for the training batches.

In [14]:
```
# Function to train the network

def train_net(net, train_loader, val_loader, num_epochs=1, learning_rate=0.1
    # Define the optimizer, SGD with learning rate
    optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)

    # Initialize loss and accuracy lists
    train_losses, train_accs, val_accs = [], [], []

    for epoch in range(num_epochs):
        net = net.to(device)
        # Initialize loss and accuracy values
        train_loss, val_loss, train_acc, val_acc = 0.0, 0.0, 0.0, 0.0

        # Training loop: (with autograd and trainer steps)
        # This loop trains the neural network (weights are updated)
        for i, (data, label) in enumerate(train_loader):
            # Zero the parameter gradients
            optimizer.zero_grad()
            data = data.to(device)
            label = label.to(device)
            output = net(data)
            loss = criterion(output, label)  # Compute the total loss in the
            loss.backward()
            train_acc += (output.argmax(axis=1) == label.float()).float().me
            train_loss += loss
            optimizer.step()

        # Validation loop:
        # This loop tests the trained network on the dation dataset. No weig
        for i, (data, label) in enumerate(val_loader):
            data = data.to(device)
            label = label.to(device)
            output = net(data)  # Compute the total loss in the validation b
            val_acc += (output.argmax(axis=1) == label.float()).float().mear
            val_loss += criterion(output, label)

        # Take averages
        train_loss /= len(train_loader)
        train_acc /= len(train_loader)
        val_loss /= len(val_loader)
        val_acc /= len(val_loader)

        train_losses.append(train_loss.item())
        train_accs.append(train_acc.item())
        val_accs.append(val_acc.item())

        print(
            "Epoch %d: train loss %.3f, train acc %.3f, val loss %.3f, val a
            % (
                epoch + 1,
                train_loss.detach().cpu().numpy(),
                train_acc.detach().cpu().numpy(),
                val_loss.detach().cpu().numpy(),
                val_acc.detach().cpu().numpy(),
            )
        )

    return train_losses, train_accs, val_accs
```

Now that you have created a training function, use it to train the model.

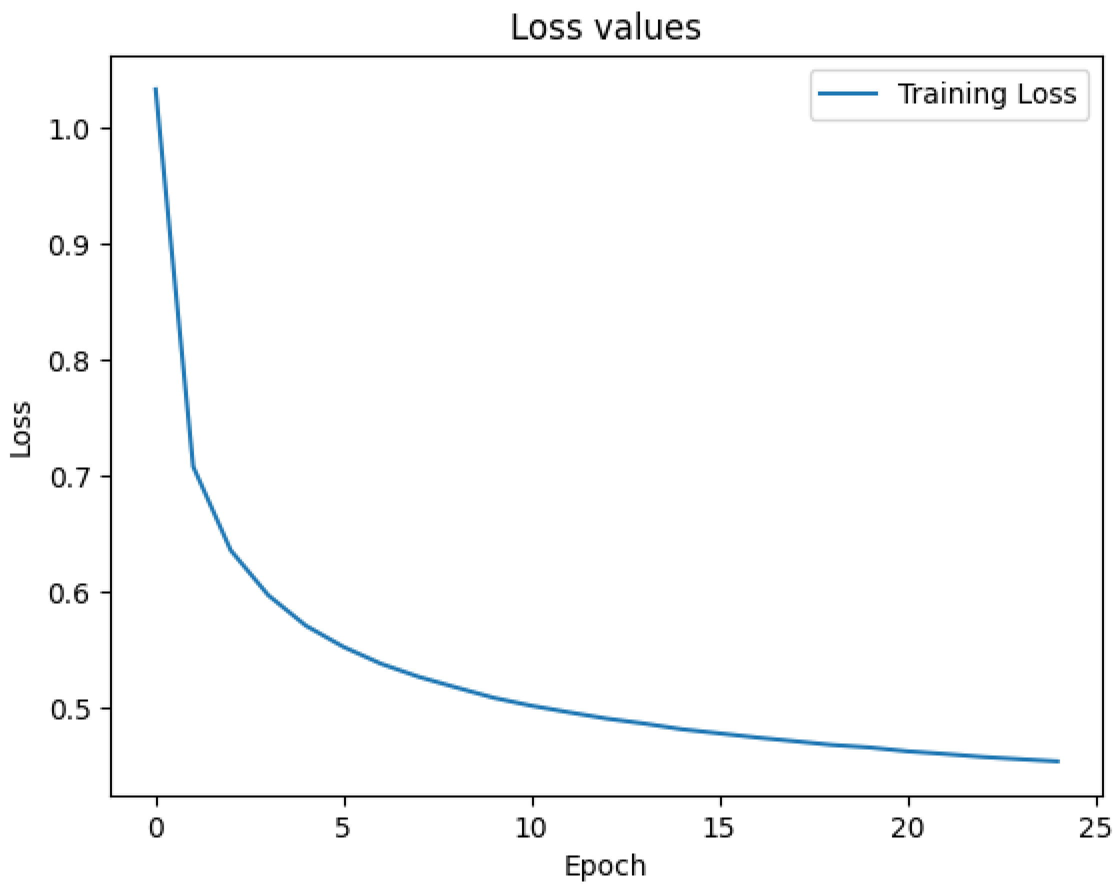
In [15]:
```
# Train the neural network
train_losses, train_accs, val_accs = train_net(
    mlp, training_loader, validation_loader, num_epochs=25, learning_rate=0.
)
```

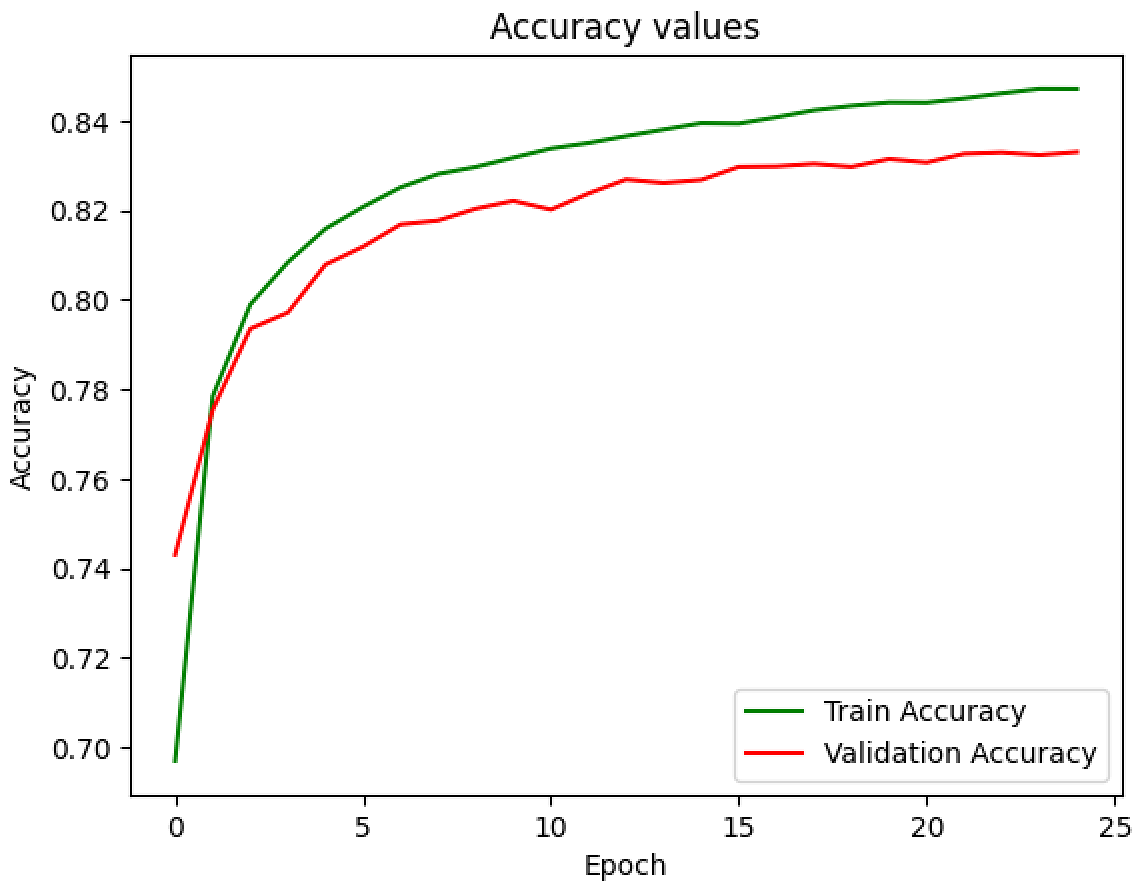
```
Epoch 1: train loss 1.033, train acc 0.697, val loss 0.783, val acc 0.743
Epoch 2: train loss 0.708, train acc 0.778, val loss 0.681, val acc 0.775
Epoch 3: train loss 0.636, train acc 0.799, val loss 0.629, val acc 0.794
Epoch 4: train loss 0.597, train acc 0.808, val loss 0.605, val acc 0.797
Epoch 5: train loss 0.571, train acc 0.816, val loss 0.579, val acc 0.808
Epoch 6: train loss 0.553, train acc 0.821, val loss 0.567, val acc 0.812
Epoch 7: train loss 0.538, train acc 0.825, val loss 0.552, val acc 0.817
Epoch 8: train loss 0.527, train acc 0.828, val loss 0.542, val acc 0.818
Epoch 9: train loss 0.518, train acc 0.830, val loss 0.534, val acc 0.820
Epoch 10: train loss 0.509, train acc 0.832, val loss 0.528, val acc 0.822
Epoch 11: train loss 0.502, train acc 0.834, val loss 0.526, val acc 0.820
Epoch 12: train loss 0.496, train acc 0.835, val loss 0.518, val acc 0.824
Epoch 13: train loss 0.491, train acc 0.837, val loss 0.513, val acc 0.827
Epoch 14: train loss 0.487, train acc 0.838, val loss 0.510, val acc 0.826
Epoch 15: train loss 0.482, train acc 0.840, val loss 0.507, val acc 0.827
Epoch 16: train loss 0.478, train acc 0.839, val loss 0.502, val acc 0.830
Epoch 17: train loss 0.475, train acc 0.841, val loss 0.500, val acc 0.830
Epoch 18: train loss 0.471, train acc 0.842, val loss 0.496, val acc 0.830
Epoch 19: train loss 0.468, train acc 0.843, val loss 0.495, val acc 0.830
Epoch 20: train loss 0.466, train acc 0.844, val loss 0.493, val acc 0.832
Epoch 21: train loss 0.463, train acc 0.844, val loss 0.491, val acc 0.831
Epoch 22: train loss 0.460, train acc 0.845, val loss 0.487, val acc 0.833
Epoch 23: train loss 0.458, train acc 0.846, val loss 0.485, val acc 0.833
Epoch 24: train loss 0.456, train acc 0.847, val loss 0.484, val acc 0.832
Epoch 25: train loss 0.454, train acc 0.847, val loss 0.484, val acc 0.833
```

After training finishes, you can create plots of the training loss, training accuracy, and validation accuracy. This will help you determine how well your model is performing.

In [16]:
```python
# Define a function to plot the training losses
def plot_losses(train_losses, train_acc, val_acc):

    plt.plot(train_losses, label="Training Loss")
    plt.title("Loss values")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    plt.plot(train_acc, "g", label="Train Accuracy")
    plt.plot(val_acc, "red", label="Validation Accuracy")
    plt.title("Accuracy values")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()
```

In [17]:
```python
# Plot the training loss function and accuracy
plot_losses(train_losses, train_accs, val_accs)
```





As you look at the graphs, think about the following questions.

1. What do you notice about the training loss?
2. Was 25 epochs enough?
3. Why is the validation accuracy lower than the training accuracy?
4. Is the accuracy high enough to consider this a good model?

What other questions do you have after reviewing the graphs?

Try it yourself!

Activity

To test your understanding of epochs and learning rate, run the following cell.

In [18]:
```
# Run this cell to display the question and check your answer
question_2
```

Out[18]:

Which option would you use for the train_net() function to use 20 epochs and a learning rate of 0.5?

train_net(mlp, training_loader, validation_loader, num_epochs=20, lr=0.5)

train_net(mlp, training_loader, validation_loader, num_epochs=0.5, learning_rate=20)

train_net(mlp, training_loader, validation_loader, num_epochs=20, learning_rate=0.5)

Submit

Add a dropout layer

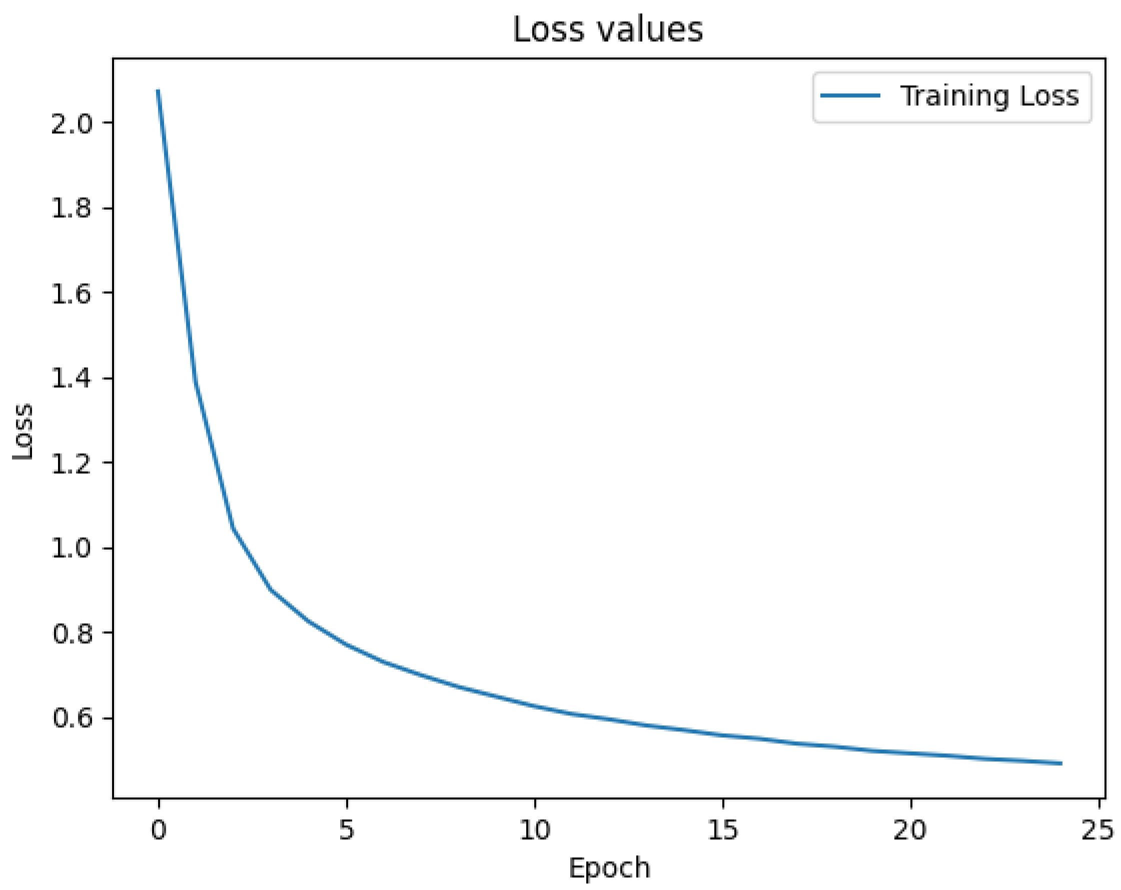
In this final step, you will add a dropout layer to prevent overfitting. A dropout layer randomly drops a certain percentage or number of neurons in a given layer. You can specify how much to drop with `nn.Dropout`.

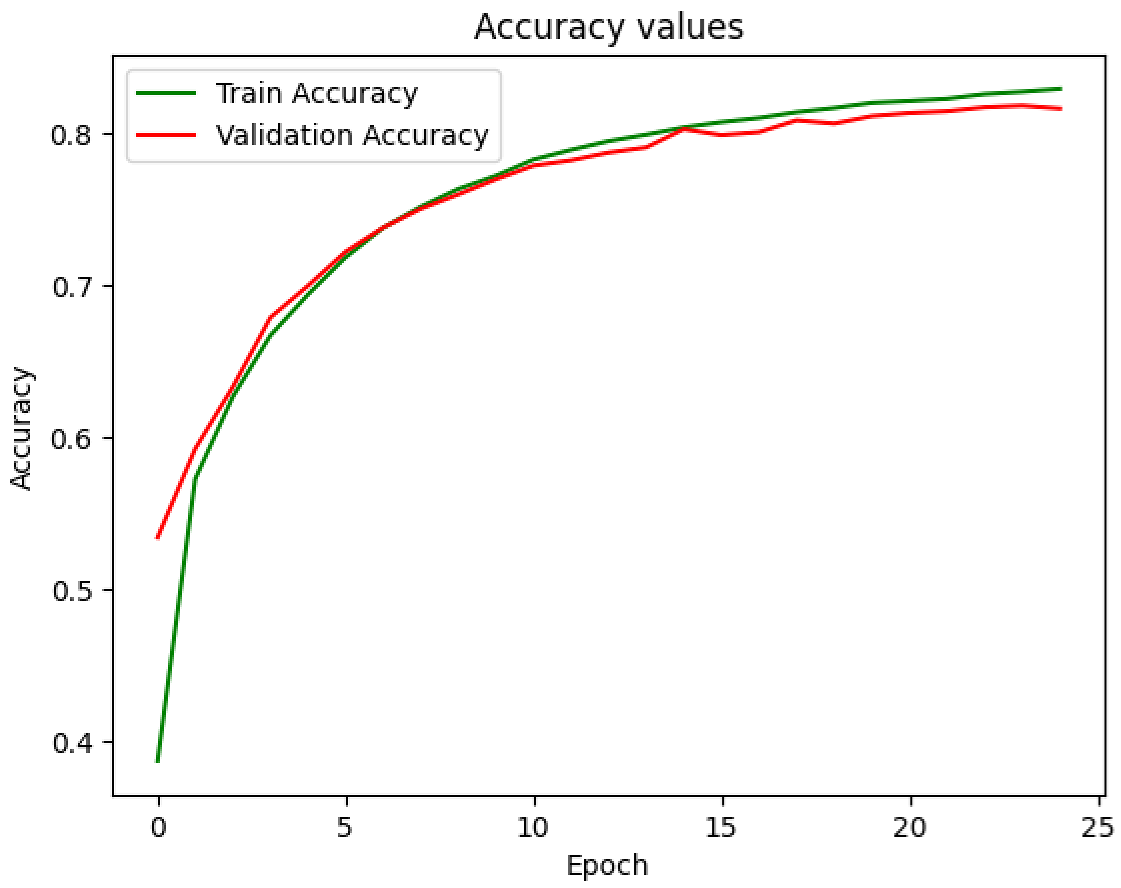
Add another layer and a dropout layer after it to see how that affects the loss and accuracy values.

In [19]:
```
# Add a hidden layer and dropout layer in between
mlp_dropout = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 784),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, out_classes),
)

num_epochs = 25

# Train the model by using the newly defined neural network
train_losses, train_accs, val_accs = train_net(
    mlp_dropout,
    training_loader,
    validation_loader,
    num_epochs=num_epochs,
    learning_rate=0.01,
)
```

```
Epoch 1: train loss 2.072, train acc 0.387, val loss 1.728, val acc 0.534
Epoch 2: train loss 1.388, train acc 0.572, val loss 1.161, val acc 0.592
Epoch 3: train loss 1.043, train acc 0.627, val loss 0.964, val acc 0.633
Epoch 4: train loss 0.898, train acc 0.667, val loss 0.865, val acc 0.679
Epoch 5: train loss 0.824, train acc 0.694, val loss 0.803, val acc 0.700
Epoch 6: train loss 0.770, train acc 0.718, val loss 0.762, val acc 0.722
Epoch 7: train loss 0.729, train acc 0.738, val loss 0.721, val acc 0.738
Epoch 8: train loss 0.698, train acc 0.752, val loss 0.694, val acc 0.750
Epoch 9: train loss 0.670, train acc 0.763, val loss 0.668, val acc 0.760
Epoch 10: train loss 0.647, train acc 0.772, val loss 0.646, val acc 0.770
Epoch 11: train loss 0.625, train acc 0.783, val loss 0.632, val acc 0.779
Epoch 12: train loss 0.606, train acc 0.789, val loss 0.618, val acc 0.782
Epoch 13: train loss 0.594, train acc 0.795, val loss 0.604, val acc 0.787
Epoch 14: train loss 0.579, train acc 0.799, val loss 0.592, val acc 0.791
Epoch 15: train loss 0.568, train acc 0.804, val loss 0.575, val acc 0.803
Epoch 16: train loss 0.556, train acc 0.807, val loss 0.569, val acc 0.799
Epoch 17: train loss 0.548, train acc 0.810, val loss 0.566, val acc 0.801
Epoch 18: train loss 0.536, train acc 0.814, val loss 0.554, val acc 0.809
Epoch 19: train loss 0.529, train acc 0.817, val loss 0.545, val acc 0.807
Epoch 20: train loss 0.520, train acc 0.820, val loss 0.540, val acc 0.811
Epoch 21: train loss 0.514, train acc 0.821, val loss 0.529, val acc 0.813
Epoch 22: train loss 0.508, train acc 0.823, val loss 0.526, val acc 0.815
Epoch 23: train loss 0.501, train acc 0.826, val loss 0.519, val acc 0.817
Epoch 24: train loss 0.496, train acc 0.827, val loss 0.512, val acc 0.818
Epoch 25: train loss 0.490, train acc 0.829, val loss 0.507, val acc 0.816
```

In [20]:
```
# Plot the loss function and accuracy graphs
plot_losses(train_losses, train_accs, val_accs)
```





As you look at the graphs, think about the following questions.

1. How do they compare to your original model without the dropout layer?
2. Is the accuracy of the new model better?
3. How does this impact the number of epochs that you need?
4. Does changing any of the settings (such as the dropout, learning rate, or epochs) improve the accuracy?

Try it yourself!

Activity

To test your understanding of dropout layers, run the following cell.

In [21]:
```
# Run this cell to display the question and check your answer
question_3
```

Out[21]:

Which option would you use to specify a dropout layer that drops 70 percent of the nodes?

nn.Dropout(70%)

nn.Dropout(0.7)

nn.Dropout(1-0.7)

Submit

Conclusion

In this notebook, you learned how to build a more advanced neural network. Topics such as dense networks and dropout layers should start to make more sense as you build more understanding about building models.

Next lab

In the next lab, you will learn how to build an end-to-end neural network.