Part I Q1

Since each layer is linear, if the activation function is linear, then the result is just putting several linear together

Need to show that the result of linear functions is still a linear function.

Let n be the # of linear functions

prove by induction

Base case: $n=1$

$$f = wx + b$$

so f is linear

Inductive step: Assume $n-1$ linear functions can be reduced to 1 single linear function (I.H)

$$f = f_n(f_{n-1}(f_{n-2} \dots (f(x)))\dots)$$

where f_i s are linear functions

By I.H

$$f = f_n(Wx + b) \text{ for some } w \text{ and } b$$

$$= W_n(Wx + b) + b_n$$

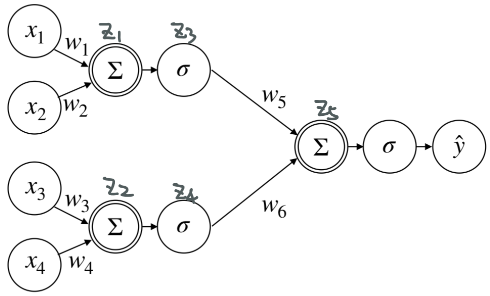
$$= W_n W x + (W_n b + b_n)$$

$$= mx + a \quad \text{where } m = W_n W,\ a = W_n b + b_n$$

so f is linear.

Thus, no matter how many layers we have, we can always reduce it to 1 single layer. So the number of layers has effectively no impact on the network.

Part 1 Q2 :



$z_1 = 1.368$, $z_2 = -1.432$ $z_3 = 0.7971$, $z_4 = 0.1928$, $z_5 = 0.5991$, $\hat{y} = 0.6455$

$$\frac{\partial L}{\partial \hat{y}} = 2\|y - \hat{y}\| = 2(0.6455 - 0.5) = 0.291$$

$$\hat{y} = \sigma(z_5)$$

$$\frac{\partial \hat{y}}{\partial z_5} = \hat{y}(1-\hat{y}) = 0.6455\,(1-0.6455) = 0.2288$$

$$\frac{\partial L}{\partial z_5} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_5} = 2\|y-\hat{y}\| \; \hat{y}(1-\hat{y}) = 0.2288 \cdot 0.291 = 0.0666$$

$$z_5 = w_5 z_3 + w_6 z_4$$

$$\frac{\partial z_5}{\partial z_4} = w_6$$

$$\frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial z_5} \cdot \frac{\partial z_5}{\partial z_4} = 2 w_6 \|y-\hat{y}\| \; \hat{y}(1-\hat{y}) = 0.0666 \cdot (-0.2) = -0.0133$$

$$z_4 = \sigma(z_2)$$

$$\frac{\partial z_4}{\partial z_2} = z_4(1-z_4) = 0.1556$$

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial z_2} = z_4(1-z_4)\, 2 w_6 \|y-\hat{y}\| \; \hat{y}(1-\hat{y}) = 0.1556 \cdot (-0.0133) = -0.0021$$

$$z_2 = w_3 x_3 + w_4 x_4$$

$$\frac{\partial z_2}{\partial w_3} = x_3$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_3} = x_3\, z_4(1-z_4)\, 2 w_6 \|y-\hat{y}\| \; \hat{y}(1-\hat{y}) = -0.3 \cdot (-0.0021)$$

$$= 0.0006$$

Part 1 Q3

Each filter size is $4 \times 4 \times 50$

So apply filter once required $4 \times 4 \times 50 = 800$ multiplications

and $800 - 1 = 799$ addition. In total 1599 FLOP

The output size is $(12 + 2 \cdot 1 - 4)/2 + 1 = 6$

That means we need to apply each filter $6 \times 6 = 36$ times.

There are 20 filters $\Rightarrow$ $20 \times 36 \times 1599$ FLOP

The output size is $6 \times 6 \times 20$

The output size of max pooling is $(6-3)/1 + 1 = 4$

So # of time applying max pooling is $4 \times 4 \times \underset{\substack{\uparrow \\ \text{20 output map}}}{20} = 320$

Each max pooling takes $3 \times 3 - 1 = 8$ FLOP

Total # of FLOP $= 20 \times 36 \times 1599 + 320 \times 8 = 1153840$ FLOP without bias

Since the output dimension is $6 \times 6 \times 20$ and each pixel can have a bias value

So # of bias $= 6 \times 6 \times 20 = 720$

Total # of FLOP $= 1153840 + 720 = 1154560$ FLOP with bias

Part 1 Q4

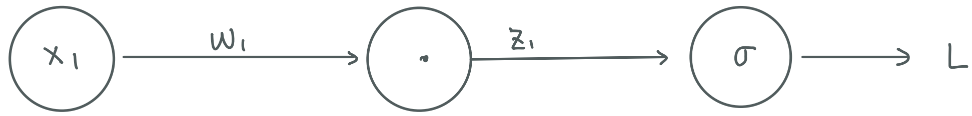
Convolution : padding=0 , stride=1 , filter size = 5x5

subsampling : padding = 0 , stride=2 , filter size = 2x2

Size / operation	Filter	Depth	stride	padding	# of parameter
32x32x1					
convolution	5x5	6	1	0	$(5\times5\times1+1)\cdot6=156$
28x28x6					
subsampling	2x2	6	2	0	
14x14x6					
covolution	5x5	16	1	0	$(5\times5\times6+1)\times16=2416$
10x10x16					
Subsampling	2x2	16	2	0	
5x5x16					
FC					$5\times5\times16\times120=48000$
120					
FC					$120\times84=10080$
84					
GC					84×10

Total # of parameter = 61492

Part 1 Q5 :



I consider only 1D case here because for each neuron

1 input variable will only associated with 1 weight parameter

Consider a input x_1, multiplied by some weight w_1

and going through logistic function

$$\frac{\partial L}{\partial L} = 1$$

$$L = \sigma(z_1)$$

$$\frac{\partial L}{\partial z_1} = L(1-L)$$

$$z_1 = x_1 w_1$$

$$\frac{\partial z_1}{\partial x_1} = w_1$$

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial x_1}$$

$$= \boxed{w_1 L(1-L)}$$

Notice that the equation does not need the value of x_1

Part 1 Q6

(a) The out put range is $(-1, 1)$

where the output range for logistic function is $(0, 1)$

(b) $$\frac{d}{dx}\tanh(x) = \frac{(1+e^{-2x})(2e^{-2x}) - (1-e^{-2x})(-2e^{-2x})}{1+2\cdot e^{-2x}+e^{-4x}}$$

$$= \frac{2e^{-2x}+2\cdot e^{-4x} - 2e^{-4x} + 2e^{-2x}}{1+2\cdot e^{-2x}+e^{-4x}}$$

$$= \frac{4e^{-2x}}{1+2e^{-2x}+e^{-4x}}$$

$$= 1 - \tanh^2(x)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma(2x) = \frac{1}{1+e^{-2x}}$$

$$2\sigma(2x) = \frac{2}{1+e^{-2x}}$$

$$2\sigma(2x) - 1 = \frac{1-e^{-2x}}{1+e^{-2x}} = \tanh(x)$$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) = 1 - [2\sigma(2x)-1]^2$$

$$= 4\sigma(2x) - 4\sigma^2(2x)$$

(c) if you want to map the result to $(0, 1)$ where each output representing the probablity, then sigmoid is a good choice

If the sign is important, we can use $\tanh(x)$ since it maps negative valut to negative value and positive value to positive value.

Also if you want the training time to be shorter, $\tanh(x)$ can be use since i- ha hi her derivatives.

Assignment 2

Task 1:

```python
def load_dataset():
    # Define transform
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,)),
                                    ])
    transform_train = transforms.Compose([transforms.RandomHorizontalFlip(),
                                          transforms.ColorJitter(),
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.5,), (0.5,)),
                                          ])
    dataset = datasets.ImageFolder(root='notMNIST_small')
    total_size = len(dataset.targets)
    train_indices, val_indices = train_test_split(list(
        range(total_size)), test_size=total_size - TRAINING_SIZE, stratify=dataset.targets)

    # Use train_test_split twice to split dataset into train, validation and test sets.
    train_dataset = torch.utils.data.Subset(dataset, train_indices)
    val_dataset = torch.utils.data.Subset(dataset, val_indices)

    val_indices, test_indices = train_test_split(val_indices, test_size=total_size - TRAINING_SIZE - VALIDATION_SIZE, stratify=[
                                                 dataset.targets[i] for i in range(len(dataset.targets)) if i in val_indices])

    train_dataset = torch.utils.data.Subset(dataset, train_indices)
    val_dataset = torch.utils.data.Subset(dataset, val_indices)
    test_dataset = torch.utils.data.Subset(dataset, test_indices)
```

```python
    test_dataset = torch.utils.data.Subset(dataset, test_indices)

    train_dataset.dataset = copy(train_dataset.dataset)
    val_dataset.dataset = copy(val_dataset.dataset)
    test_dataset.dataset = copy(test_dataset.dataset)

    train_dataset.dataset.transform = transform_train
    val_dataset.dataset.transform = transform
    test_dataset.dataset.transform = transform

    print(train_dataset.dataset.transform,
          val_dataset.dataset.transform, test_dataset.dataset.transform)
    # Load dataset with batch size = 32
    trainloader = torch.utils.data.DataLoader(
        train_dataset, batch_size=32, shuffle=True)
    validationloader = torch.utils.data.DataLoader(
        val_dataset, batch_size=32, shuffle=True)
    testloader = torch.utils.data.DataLoader(
        test_dataset, batch_size=32, shuffle=True)
    return trainloader, validationloader, testloader
```

The transformation that I choose for validation set and test set are ToTensor() and Normailize(). ToTensor() is required and Normailize() generally speeds up the learning process and leads to faster convergence. For training set, in addition to ToTensor() and Normalize(), I also add RandomHorizontalFlip and ColorJitter because I want neural network to be able recognize letter even it is horizontally flipped or the brightness is different. To split data set into training, validation, test sets, train_test_split is the helper function I use. After removing the corrupted images, the training set will have 15000 images, validation set has 1000 images and test set has 2724 images. A small batch size will result in a longer training time, so I choose 32 in my case which suits best on my pc.

Task 2:

```python
def define_model(hidden_sizes: int):
    # define the neural network model
    # Hyperparameters for our network
    input_size = 3 * 28 * 28
    output_size = 10
    # Build a feed-forward network
    model = nn.Sequential(nn.Linear(input_size, hidden_sizes),
                          nn.ReLU(),
                          nn.Linear(hidden_sizes, output_size),
                          nn.Softmax(dim=1))
    print(model)
    return model
```

The model has only one hidden layer with Relu as activation function and has a softmax function after the output layer.

```python
def train_model(model, trainloader, validationloader, save_file_name, lr, n_epochs=20, max_epochs_stop=2):
    # create model and define the loss

    criterion = nn.CrossEntropyLoss()
    # Optimizers require the parameters to optimize and a learning rate
    optimizer = optim.SGD(model.parameters(), lr=lr)
    overall_start = timer()
    history = []
    valid_loss_min = np.Inf

    for epoch in range(n_epochs):
        train_loss = 0
        validation_loss = 0

        train_acc = 0
```

The loss function is defined as CrossEntropyLoss. The optimizer is SGD.

```python
        for ii, (images, labels) in enumerate(trainloader):
            # Flatten MNIST images into a 2352 long vector
            images = images.view(images.shape[0], -1)

            # Training pass
            optimizer.zero_grad()

            output = model(images)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * images.size(0)

            # Calculate accuracy by finding max log probability
            _, pred = torch.max(output, dim=1)
            correct_tensor = pred.eq(labels.data.view_as(pred))
            # Need to convert correct tensor from int to float to average
            accuracy = torch.mean(correct_tensor.type(torch.FloatTensor))
            # Multiply average accuracy times the number of examples in batch
            train_acc += accuracy.item() * images.size(0)
            print(
                f'Epoch: {epoch}\t{100 * (ii + 1) / len(trainloader):.2f}% complete. {timer()
                end='\r')
```

For each epoch, training loss and training accuracy has been calculated.

```python
            with torch.no_grad():
                model.eval()

                for images, labels in validationloader:
                    images = images.view(images.shape[0], -1)
                    # forward pass
                    output = model(images)

                    # validation loss
                    loss = criterion(output, labels)
                    validation_loss += loss.item() * images.size(0)

                    # Calculate validation accuracy
                    _, pred = torch.max(output, dim=1)
                    correct_tensor = pred.eq(labels.data.view_as(pred))
                    accuracy = torch.mean(
                        correct_tensor.type(torch.FloatTensor))
                    # Multiply average accuracy times the number of examples
                    validation_acc += accuracy.item() * images.size(0)

                # calculate average loss and average accuracy for training and validation
                train_loss = train_loss / len(trainloader.dataset)
                validation_loss = validation_loss / \
                    len(validationloader.dataset)

                train_acc = train_acc / len(trainloader.dataset)
                validation_acc = validation_acc / len(validationloader.dataset)
```

The validation accuracy and validation loss are calculated as well.

```python
            )
        if validation_loss < valid_loss_min:
            # Save model
            torch.save(model.state_dict(), save_file_name)
            # Track improvement
            epochs_no_improve = 0
            valid_loss_min = validation_loss
            best_epoch = epoch

        else:
            epochs_no_improve += 1
            # Trigger early stopping
            if epochs_no_improve >= max_epochs_stop:
                print(
                    f'\nEarly Stopping! Total epochs: {epoch}. Best epoch: {best_epoch} with loss: {valid_loss_
                )
                total_time = timer() - overall_start
                print(
                    f'{total_time:.2f} total seconds elapsed. {total_time / (epoch+1):.2f} seconds per epoch.'
                )

                # Load the best state dict
                model.load_state_dict(torch.load(save_file_name))
                # Attach the optimizer
                model.optimizer = optimizer

                # Format history
```

The early stop is implemented as well. By default, 2 consecutive increasing in validation loss will stop the training process.

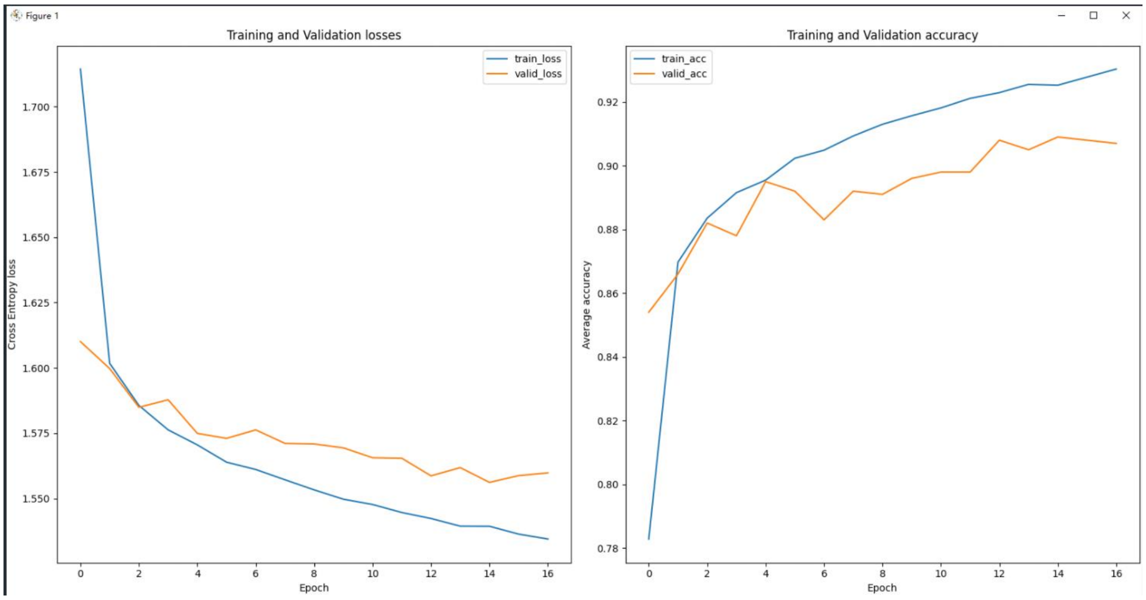
```python
# ============= Task 2 =============
LR_RATE = [0.1, 0.05, 0.03, 0.01, 0.005]
valid_loss_min = np.Inf
for lr in LR_RATE:
    model = define_model(HIDDEN_SIZE)
    print(f'Trianing on leanring rate: {lr}')
    model, history, valid_loss = train_model(
        model, trainloader, validationloader, FILE_NAME, lr, n_epochs=20)
    if valid_loss < valid_loss_min:
        valid_loss_min = valid_loss
        best_model = model
        best_history = history
        best_lr = lr
print('The best learning rate is ', best_lr)
fig = plt.figure(figsize=(16, 8), constrained_layout=True)
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
for c in ['train_loss', 'valid_loss']:
    ax1.plot(best_history[c], label=c)
ax1.legend()
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Cross Entropy loss')
ax1.set_title('Training and Validation losses')

for c in ['train_acc', 'valid_acc']:
    ax2.plot(best_history[c], label=c)
ax2.legend()
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Average accuracy')
ax2.set_title('Training and Validation accuracy')

test_loss, test_acc = test_error_acc(best_model, testloader)
```

The five learning rates I choose are 0.1, 0.05, 0.03, 0.01, 0.005. The training process takes a bit longer time, so I've saved the print messages in Task 2 print result.txt file. It turns out that 0.1 works the best. The plot is shown below.

The plot has been saved as plot_result_task2.png. And for this model, the test loss is 1.5706202945345125 and the test accuracy is 89.46%. (Result can be different if training multiple times)



The best training loss and validation loss are 1.5394 and 1.5562

Task 3:

```python
# =========== task 3 ================
# Based on task 2, the best learning rate is 0.1
try:
    lr = best_lr
except:
    lr = 0.1
valid_loss_min = np.Inf
layer_size = [100, 500, 1000]
valid_loss_list = []
best_size = 0
for size in layer_size:
    model = define_model(size)
    print(f"Training for size: {size}")
    model, history, valid_loss = train_model(
        model, trainloader, validationloader, FILE_NAME, lr, n_epochs=20)
    valid_loss_list.append(valid_loss)
    if valid_loss < valid_loss_min:
        best_size = size
        valid_loss_min = valid_loss
        best_model = model
print(
    f'The validation loss are {valid_loss_list[0]:.4f}, {valid_loss_list[1]:.4f}, {valid_loss_list[2]:.4f}')
test_loss, test_acc = test_error_acc(best_model, testloader)
print(
    f'The best model has {best_size} units in hidden layer')
print(
    f'The test loss is {test_loss}. The test accuracy is {100 * test_acc:.2f}%')
```

Training model with layer size = 100, 500, and 1000 separately. I use 0.1 as learning rate. The validation losses are 1.5982, 1.5863, 1.5828

The best model has 1000 units in hidden layer

The test loss is 1.5748151756068158. The test accuracy is 89.24%

The validation losses are extremely close to each other with hidden unit = 1000 works slightly better than 500 which works slightly better than 100. Thus, the more the number of units are, the lower the loss is. The printed messages are stored in Task 3 print result.txt file.

Task 4:

```
def define_model_two_layers(hidden_sizes: list):
    # define the neural network model
    # Hyperparameters for our network
    input_size = 3 * 28 * 28
    output_size = 10
    # Build a feed-forward network
    model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                          nn.ReLU(),
                          nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                          nn.ReLU(),
                          nn.Linear(hidden_sizes[1], output_size),
                          nn.Softmax(dim=1))
    print(model)
    return model

```

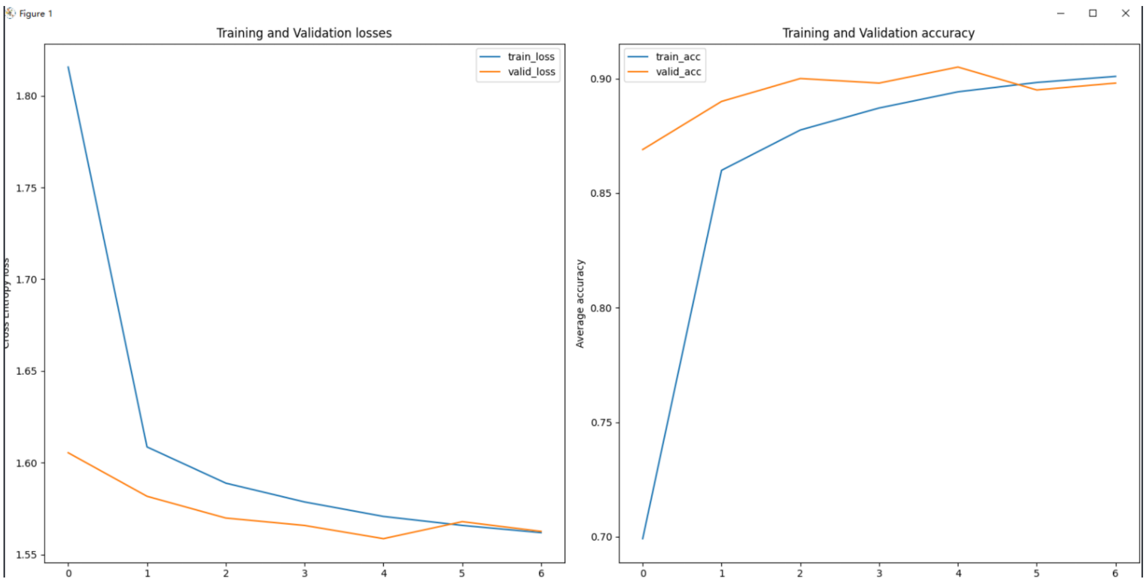
The model is defined above

```python
# =========== task 4 ===============
try:
    lr = best_lr
except:
    lr = 0.1
model = define_model_two_layers([500, 500])
model, history, valid_loss = train_model(
    model, trainloader, validationloader, FILE_NAME, lr, n_epochs=20)
try:
    fig.clear()
except:
    fig = plt.figure(figsize=(16, 8), constrained_layout=True)
ax3 = fig.add_subplot(121)
ax4 = fig.add_subplot(122)
for c in ['train_loss', 'valid_loss']:
    ax3.plot(history[c], label=c)
ax3.legend()
ax3.set_xlabel('Epoch')
ax3.set_ylabel('Cross Entropy loss')
ax3.set_title('Training and Validation losses')

for c in ['train_acc', 'valid_acc']:
    ax4.plot(history[c], label=c)
ax4.legend()
ax4.set_xlabel('Epoch')
ax4.set_ylabel('Average accuracy')
ax4.set_title('Training and Validation accuracy')

test_loss, test_acc = test_error_acc(model, testloader)
print(
    f'The test loss is {test_loss}. The test accuracy is {100 * test_acc:.2f}%')
plt.show()
```

The learning rate I use is 0.1 since it works the best in task 2. The validation loss and training loss are shown below. The plot has been saved as plot_result_task4.png.



It early stops at epoch 6.

The best training Loss is 1.5708 and the best validation Loss: 1.5587 which happens at epoch4.

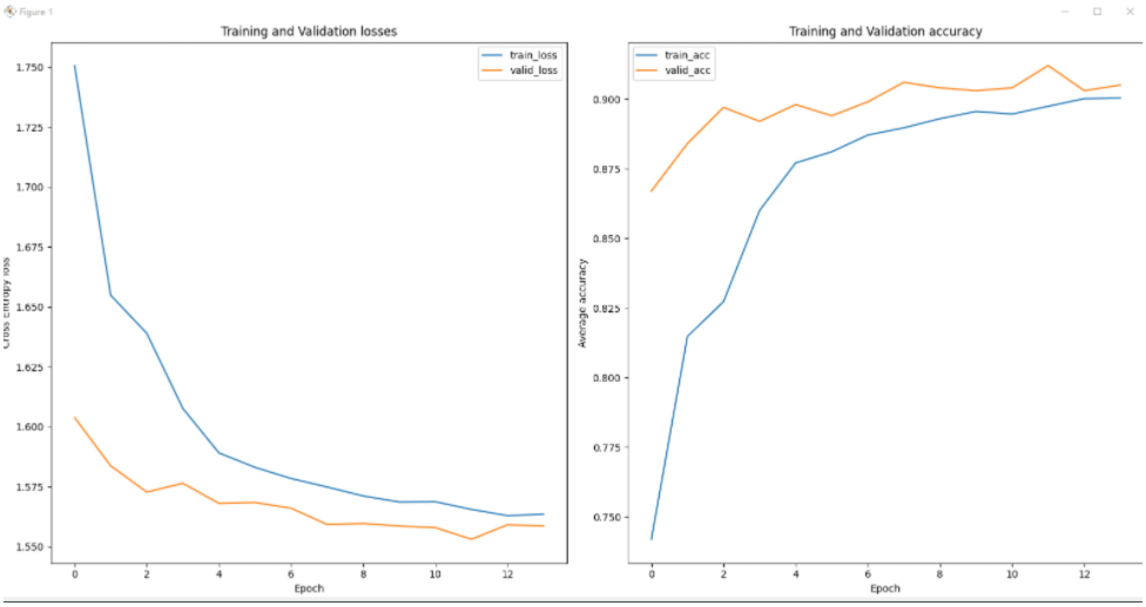
The test loss is 1.5820134526307363. The test accuracy is 88.40%

Compare with model that has only 1 layer with 1000 hidden units, the validation loss of two layers neural network is slightly lower. However, the test results are quite closed. The print messages are stored in Task 4 print result.txt file.

Task 5:

```python
def define_model_with_dropout(hidden_sizes: int):
    # define the neural network model
    # Hyperparameters for our network
    input_size = 3 * 28 * 28
    output_size = 10
    # Build a feed-forward network
    model = nn.Sequential(nn.Linear(input_size, hidden_sizes),
                          nn.ReLU(),
                          nn.Dropout(0.5),
                          nn.Linear(hidden_sizes, output_size),
                          nn.Softmax(dim=1))
    print(model)
    return model
```

The model is defined above. The final training loss and validation loss for this model is 1.5654 and 1.5530. The training loss and validation loss in task 2 are 1.5394 and 1.5562. As we can see the training loss is higher when applying dropout. That's because the dropout prevent model from overfitting, so the training loss is expected to be higher. The print messages are stored in Task 5 print result.txt file.



The test loss is 1.5818051213559995. The test accuracy is 87.89%

The plot has been saved as plot_result_task5.png.