In [7]:
```
import numpy as np
import pickle
from torch import nn
import torch
from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt

from functools import wraps
from time import time
def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))
        return result
    return wrap
```

In [8]:
```
# load dataset
(train_X_raw, train_y), (test_X_raw, test_y) = pickle.load(open("./mnist.pkl", "rb"))

# normalize features (not labels)
train_X = train_X_raw / train_X_raw.max()
test_X = test_X_raw / test_X_raw.max()
```

In [9]:
```
def create_chunks(complete_list, chunk_size=None, num_chunks=None):
    '''
    Cut a list into multiple chunks, each having chunk_size (the last chunk might be less than chunk_size)
    or having a total of num_chunk chunks
    '''
    chunks = []
    if num_chunks is None:
        num_chunks = math.ceil(len(complete_list) / chunk_size)
```

```
    elif chunk_size is None:
        chunk_size = math.ceil(len(complete_list) / num_chunks)
    for i in range(num_chunks):
        chunks.append(complete_list[i * chunk_size: (i + 1) * chunk_size])
    return chunks

# Shuffle the training data and split into chunks using permutation
# define permutation index to make sure x values (features) are shuffled with their corresponding labels (y
perm_index = np.random.permutation(len(train_X))
# permute to predetermined indices
train_X_perm = train_X[perm_index]
train_y_perm = train_y[perm_index]
# split into three chunks
chunks_X = create_chunks(train_X_perm, num_chunks=3)
chunks_y = create_chunks(train_y_perm, num_chunks=3)
# make test data by combining two chunks
test_X1 = np.concatenate(chunks_X[0:2])
test_y1 = np.concatenate(chunks_y[0:2])
# validation data is WAN chunk
validate_X1 = chunks_X[2]
validate_y1 = chunks_y[2]
```

In [10]:
```
class Trainer():
    def __init__(self, model, optimizer_type, learning_rate, epoch, batch_size, input_transform=lambda x: x
        """ The class for training the model
        model: nn.Module
            A pytorch model
        optimizer_type: 'Adam' or 'sgd'
        learning_rate: float
        epoch: int
        batch_size: int
        input_transform: func
            transforming input. Can do reshape here
        """
        self.model = model
        if optimizer_type == "sgd":
            self.optimizer = SGD(model.parameters(), learning_rate,momentum=0.9)
        elif optimizer_type == "Adam":
            self.optimizer = Adam(self.model.parameters(), lr=learning_rate)

        self.epoch = epoch
        self.batch_size = batch_size
```

```
        self.input_transform = input_transform


    @timing
    def train(self, inputs, outputs, val_inputs, val_outputs,early_stop=False,l2=False,silent=False):
        """ train self.model with specified arguments
        inputs: np.array, The shape of input_transform(input) should be (ndata,nfeatures)
        outputs: np.array shape (ndata,)
        val_nputs: np.array, The shape of input_transform(val_input) should be (ndata,nfeatures)
        val_outputs: np.array shape (ndata,)
        early_stop: bool
        l2: bool
        silent: bool. Controls whether or not to print the train and val error during training

        @return
        a dictionary of arrays with train and val losses and accuracies
        """
        ### convert data to tensor of correct shape and type here ###
        inputs = torch.tensor(inputs, dtype=torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)
        inputs = inputs.reshape(-1, 1, 32, 32)

        losses = []
        accuracies = []
        val_losses = []
        val_accuracies = []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        loss_fn = nn.CrossEntropyLoss()

        for n_epoch in tqdm(range(self.epoch), leave=False):
            self.model.train()
            batch_indices = list(range(inputs.shape[0]))
            random.shuffle(batch_indices)
            batch_indices = create_chunks(batch_indices, chunk_size=self.batch_size)
            epoch_loss = 0
            epoch_acc = 0
            for batch in batch_indices:
                batch_importance = len(batch) / len(outputs)
                batch_input = inputs[batch]
                batch_output = outputs[batch]
                ### make prediction and compute loss with loss function of your choice on this batch ###
```

```
                batch_predictions = self.model(batch_input)
                loss = loss_fn(batch_predictions, batch_output)
                if l2:
                    ### Compute the loss with L2 regularization ###
                    self.optimizer = Adam(self.model.parameters(), weight_decay= 1e-5)
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                ### Compute epoch_loss and epoch_acc
                # num accurately predicted points / num points in batch * importance
                acc = torch.argmax(batch_predictions, dim=1).eq(batch_output).sum().item() / len(batch_out
                epoch_loss += loss.item() * batch_importance
                epoch_acc += acc
            val_loss, val_acc = self.evaluate(val_inputs, val_outputs, print_acc=False)
            if n_epoch % 10 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f - Acc: %.3f" % (n_epoch + 1, self.epoch, epoch_loss, epoch_
                print("              Val_loss: %.3f - Val_acc: %.3f" % (val_loss, val_acc))
            losses.append(epoch_loss)
            accuracies.append(epoch_acc)
            val_losses.append(val_loss)
            val_accuracies.append(val_acc)
            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    # saves current state of model's parameters to dict weights
                    weights = self.model.state_dict()

        if early_stop:
            # loads saved parameters back into model
            self.model.load_state_dict(weights)

        # plot training and validation losses
        plt.figure(figsize=(12,4))
        plt.subplot(1,2,1)
        plt.plot(losses, label='Training Loss')
        plt.plot(val_losses, label='Validation Loss')
        plt.title('Training vs Validation Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

        # plot training and validation accuracy
```

```
        plt.subplot(1,2,2)
        plt.plot(accuracies, label='Training Accuracy')
        plt.plot(val_accuracies, label='Validation Accuracy')
        plt.title('Training vs Validation Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()

        plt.show()

        return {"losses": losses, "accuracies": accuracies, "val_losses": val_losses, "val_accuracies": va

    def evaluate(self, inputs, outputs, print_acc=True):
        """ evaluate model on provided input and output
        inputs: np.array, The shape of input_transform(input) should be (ndata,nfeatures)
        outputs: np.array shape (ndata,)
        print_acc: bool

        @return
        losses: float
        acc: float
        """
        inputs = torch.tensor(inputs, dtype= torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)
        inputs = inputs.reshape(-1, 1, 32, 32)
        loss_fn = nn.CrossEntropyLoss()

        self.model.eval()
        batch_indices = list(range(inputs.shape[0]))
        batch_indices = create_chunks(batch_indices, chunk_size=self.batch_size)
        acc = 0
        loss = 0

        for batch in batch_indices:
            batch_importance = len(batch) / len(outputs)
            batch_input = inputs[batch]
            batch_output = outputs[batch]
            batch_predictions = self.model(batch_input)
            with torch.no_grad():
                # compute predictions and losses
                batch_acc = torch.argmax(batch_predictions, dim=1).eq(batch_output).sum().item() / len(bat
                loss += loss_fn(batch_predictions, batch_output) * batch_importance
```

```
            acc = acc + batch_acc

        if print_acc:
            print("Accuracy: %.3f" % acc)
        return loss, acc
```

Using the CNN developed in HW#8, adapt your architecture to the one shown in the figure below (architecture with two layers each composed of one convolution and one pooling layer.) Use ReLU as your activation function. Use conv/pooling layers that with kernel, stride and padding size that lead to output size of 12x5x5 before flattening. Flatten the resulting feature maps and use two fully connected (FC) layers of output size (300,10). Add an additive skip connection from flattened layer to the second fully connected layer.

In [11]:
```
class RNN(nn.Module):
    """
    This class defines a resnet with batch normalization.
    """
    def __init__(self):
        super(RNN, self).__init__()
        # convolution
        self.conv = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=5, stride=1, padding=2)
        # pooling
        self.pooling = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        # batch norm
        self.bn = nn.BatchNorm2d(num_features=3)
        # flatten results
        self.flatten = nn.Flatten()
        # ReLU as activation function
        self.relu = nn.ReLU()
        # fc layers
        self.fc1 = nn.Linear(16*16*3, 12*5*5)
        self.fc2 = nn.Linear(300, 10)
        # skip connection
        self.skip = nn.Linear(16*16*3, 12*5*5)

    def forward(self, x0):
        # conv -> batch norm -> activation -> pooling
        x1 = self.bn(self.conv(x0)) # 3 * 32 * 32
        x2 = self.relu(x1) # 3 * 32 * 32
        x3 = self.pooling(x2) # 3 * 16 * 16
        # flatten
```

```
        x4 = self.flatten(x3)
        # take output of flattened layer as residual
        res = self.skip(x4)
        # fully connected layer
        x5 = self.fc1(x4) # 12 * 5 * 5
        x6 = self.relu(x5)
        # add residual to output of first fc layer
        x6 = x6 + res
        # second fc layer
        x7 = self.fc2(x6)

        return x7
```

Again, use the ADAM optimizer with learning rate of 1e-3, batchsize of 128, and 30 epochs (you can also train for longer if time permits). Split the MNIST training set into 2/3 for training and 1/3 for validation, you don't need to do KFold this time. Use batch normalization of data, choose some regularization techniques and converge your training to where the loss function is minimal.

1a

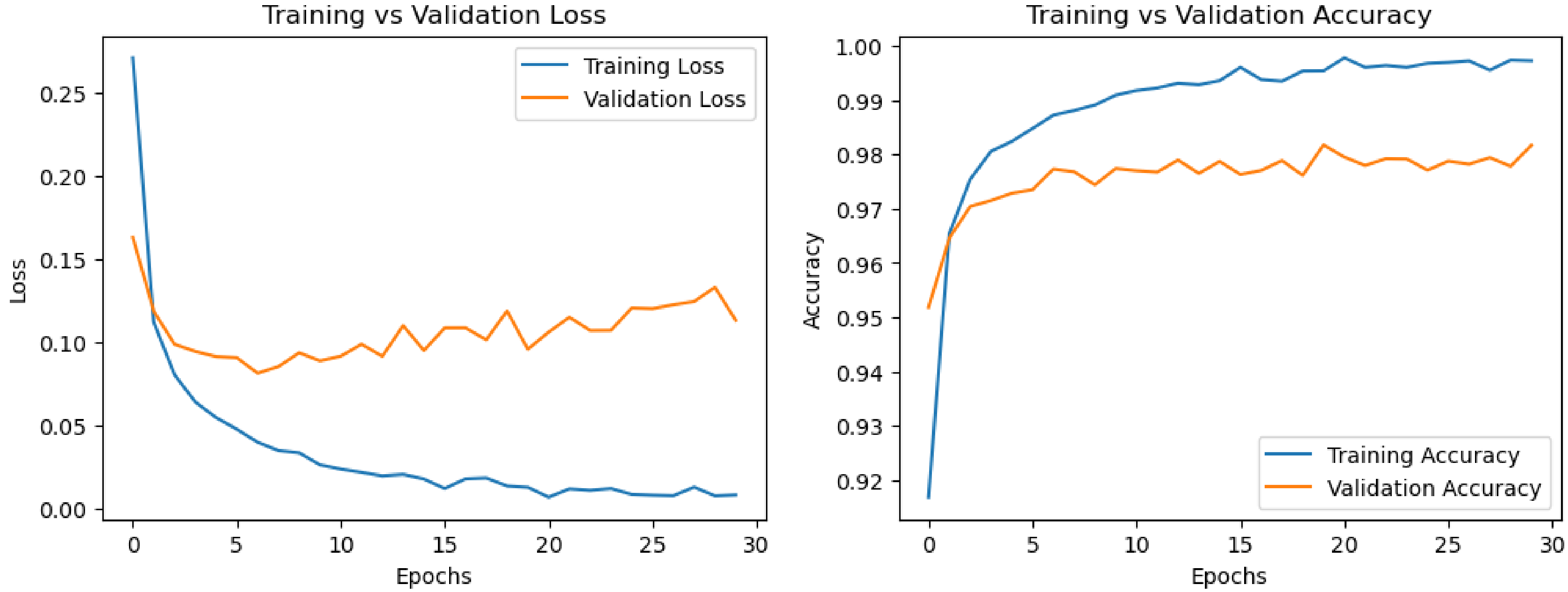
Run the model with and without batch normalization. Which give you better test accuracy?

In [12]:

```
# WITH BATCH NORMALIZATION
kf = KFold(3, shuffle=True, random_state=49)

for idc, (train_index, val_index) in enumerate(kf.split(train_X)):
    X_train_fold, X_val_fold = train_X[train_index], train_X[val_index]
    y_train_fold, y_val_fold = train_y[train_index], train_y[val_index]

    RNN1 = RNN()
    train_RNN1 = Trainer(RNN1, optimizer_type="Adam", learning_rate=1e-3, epoch=30, batch_size=128)
    RNN1_results = train_RNN1.train(X_train_fold, y_train_fold, X_val_fold, y_val_fold, early_stop=True)
```

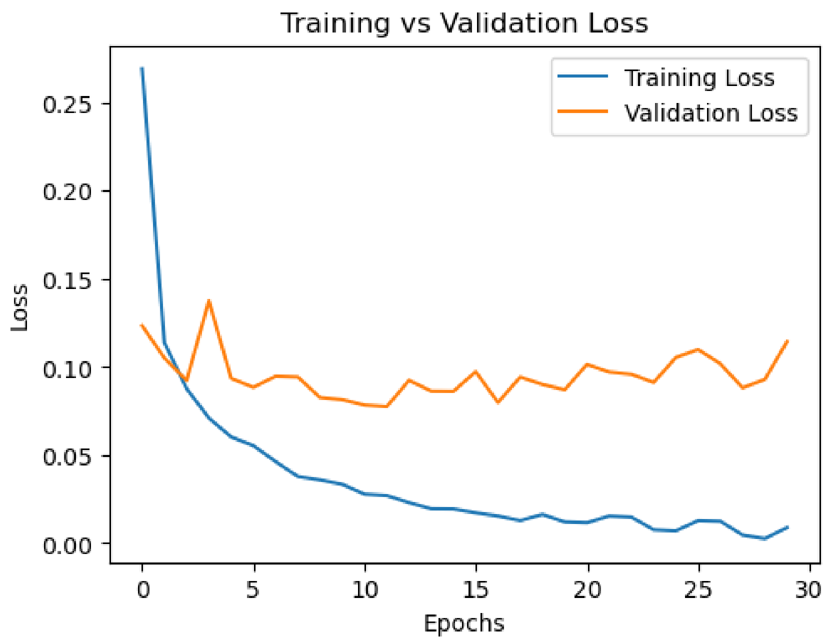
```
  3%|▌         | 1/30 [00:04<01:57,  4.06s/it]
Epoch 1/30 - Loss: 0.271 - Acc: 0.917
              Val_loss: 0.163 - Val_acc: 0.952
 37%|████      | 11/30 [00:48<01:20,  4.22s/it]
Epoch 11/30 - Loss: 0.024 - Acc: 0.992
              Val_loss: 0.092 - Val_acc: 0.977
```

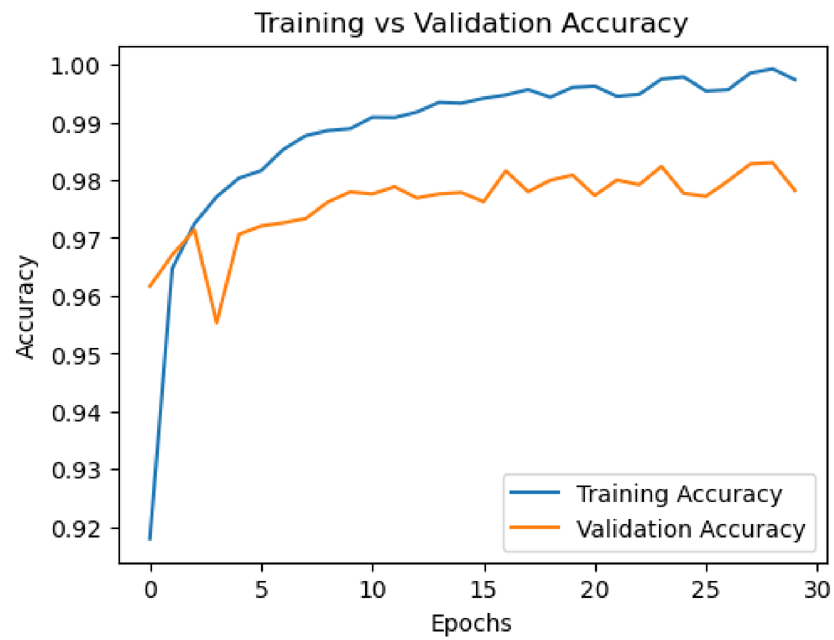
```
 70%|███████   | 21/30 [01:28<00:35,  4.00s/it]
Epoch 21/30 - Loss: 0.007 - Acc: 0.998
              Val_loss: 0.106 - Val_acc: 0.979
```




```
func:'train'  took: 124.6059 sec
  3%|▎         | 1/30 [00:04<02:01,  4.18s/it]
Epoch 1/30 - Loss: 0.257 - Acc: 0.921
              Val_loss: 0.161 - Val_acc: 0.952
 37%|███▋      | 11/30 [00:45<01:16,  4.00s/it]
Epoch 11/30 - Loss: 0.020 - Acc: 0.993
              Val_loss: 0.100 - Val_acc: 0.976
 70%|███████   | 21/30 [01:25<00:36,  4.08s/it]
Epoch 21/30 - Loss: 0.010 - Acc: 0.997
              Val_loss: 0.112 - Val_acc: 0.978
```


```
func:'train'  took: 122.4681 sec
  3%|▍         | 1/30 [00:04<01:57,  4.05s/it]
Epoch 1/30 - Loss: 0.269 - Acc: 0.918
              Val_loss: 0.123 - Val_acc: 0.962
 37%|███▋      | 11/30 [00:44<01:17,  4.06s/it]
Epoch 11/30 - Loss: 0.028 - Acc: 0.991
              Val_loss: 0.078 - Val_acc: 0.978
 70%|███████   | 21/30 [01:25<00:37,  4.13s/it]
Epoch 21/30 - Loss: 0.012 - Acc: 0.996
              Val_loss: 0.101 - Val_acc: 0.977
```






```
func:'train'  took: 123.7244 sec
```

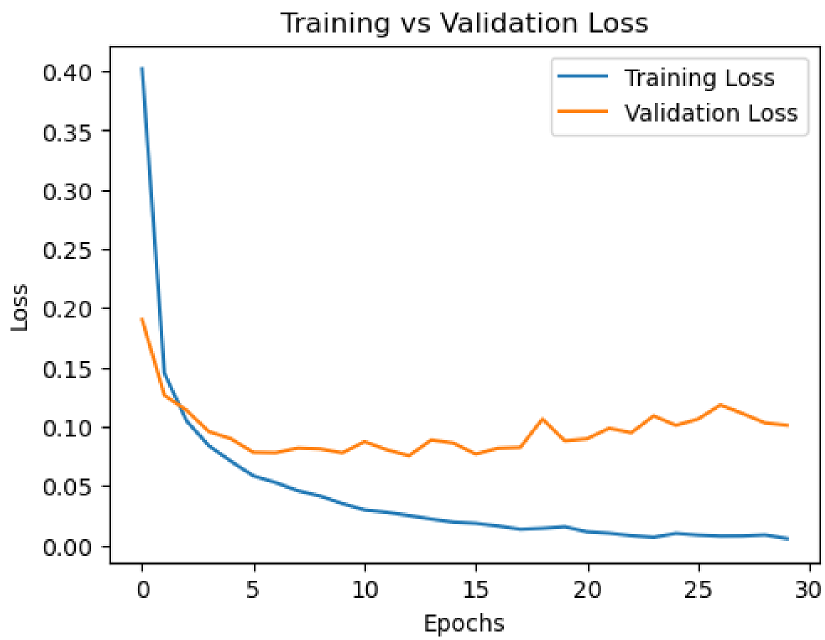
In [16]:
```
class RNN_no_batch_norm(nn.Module):
    """
    This class defines a resnet without batch normalization.
    """
    def __init__(self):
        super(RNN_no_batch_norm, self).__init__()
        # convolution
        self.conv = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=5, stride=1, padding=2)
        # pooling
        self.pooling = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        # flatten results
        self.flatten = nn.Flatten()
        # ReLU as activation function
        self.relu = nn.ReLU()
        # fc layers
        self.fc1 = nn.Linear(16*16*3, 12*5*5)
        self.fc2 = nn.Linear(300, 10)
        # skip connection
        self.skip = nn.Linear(16*16*3, 12*5*5)
```

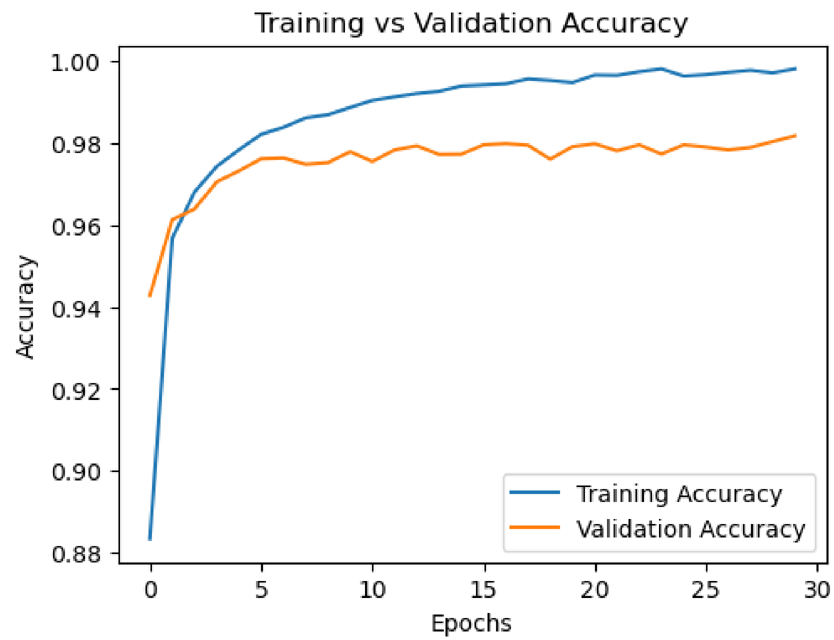
```python
    def forward(self, x0):
        # conv -> activation -> pooling
        x1 = self.conv(x0) # 3 * 32 * 32
        x2 = self.relu(x1) # 3 * 32 * 32
        x3 = self.pooling(x2) # 3 * 16 * 16
        # flatten
        x4 = self.flatten(x3)
        # take output of flattened layer as residual
        res = self.skip(x4)
        # fully connected layer
        x5 = self.fc1(x4) # 12 * 5 * 5
        x6 = self.relu(x5)
        # add residual to output of first fc layer
        x6 = x6 + res
        # second fc layer
        x7 = self.fc2(x6)

        return x7
```

In [14]:

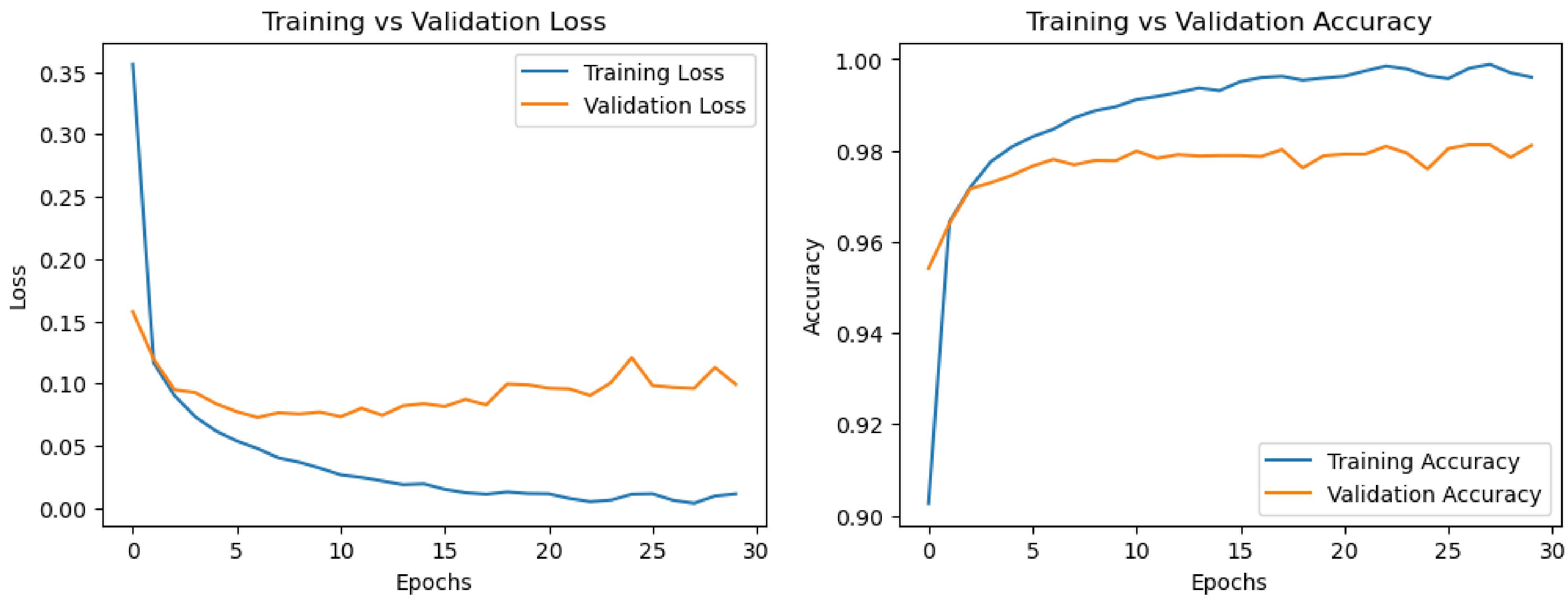
```python
# WITHOUT BATCH NORMALIZATION
kf = KFold(3, shuffle=True, random_state=49)

for idc, (train_index, val_index) in enumerate(kf.split(train_X)):
    X_train_fold, X_val_fold = train_X[train_index], train_X[val_index]
    y_train_fold, y_val_fold = train_y[train_index], train_y[val_index]

    RNN2 = RNN_no_batch_norm()
    train_RNN2 = Trainer(RNN2, optimizer_type="Adam", learning_rate=1e-3, epoch=30, batch_size=128)
    RNN2_results = train_RNN2.train(X_train_fold, y_train_fold, X_val_fold, y_val_fold, early_stop=True)
```

```
  3%|▌         | 1/30 [00:03<01:55,  3.97s/it]
Epoch 1/30 - Loss: 0.402 - Acc: 0.883
              Val_loss: 0.190 - Val_acc: 0.943
 37%|████      | 11/30 [00:41<01:10,  3.72s/it]
Epoch 11/30 - Loss: 0.030 - Acc: 0.990
              Val_loss: 0.087 - Val_acc: 0.975
 70%|███████   | 21/30 [01:19<00:34,  3.82s/it]
Epoch 21/30 - Loss: 0.011 - Acc: 0.996
              Val_loss: 0.090 - Val_acc: 0.980
```

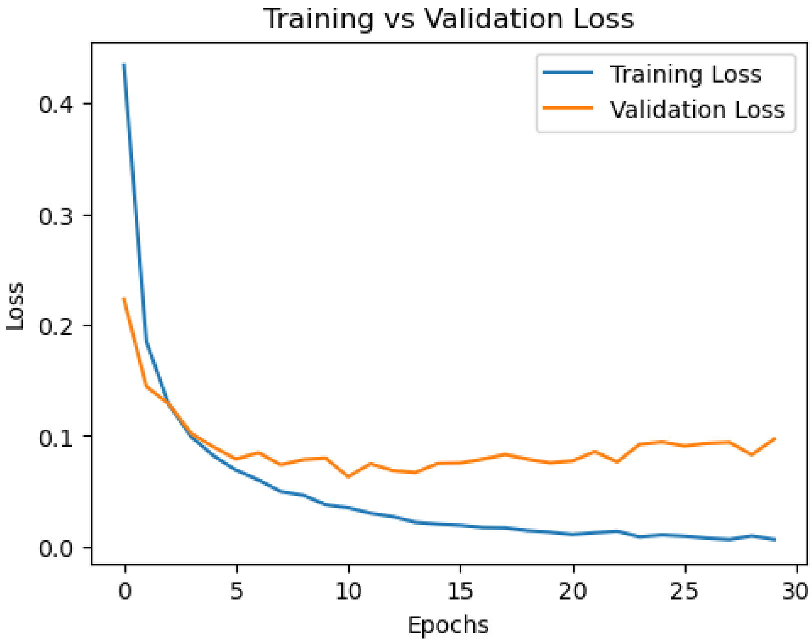


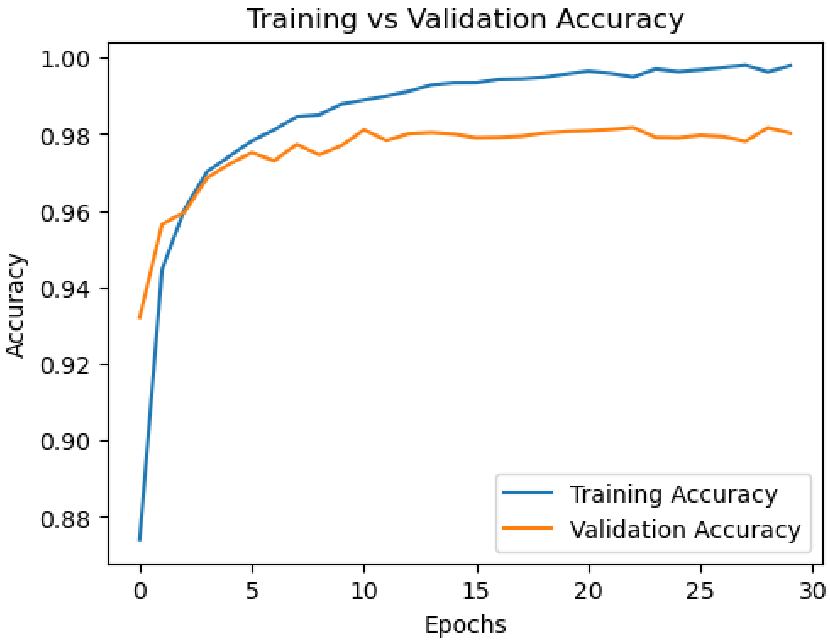



```
func:'train'  took: 114.5180 sec
  3%|▌         | 1/30 [00:03<01:49,  3.76s/it]
Epoch 1/30 - Loss: 0.356 - Acc: 0.903
              Val_loss: 0.158 - Val_acc: 0.954
 37%|████      | 11/30 [00:40<01:10,  3.69s/it]
Epoch 11/30 - Loss: 0.027 - Acc: 0.991
              Val_loss: 0.073 - Val_acc: 0.980
 70%|███████   | 21/30 [01:18<00:33,  3.69s/it]
Epoch 21/30 - Loss: 0.011 - Acc: 0.996
              Val_loss: 0.096 - Val_acc: 0.979
```




```
func:'train'  took: 110.8422 sec
  3%|▍         | 1/30 [00:03<01:52,  3.88s/it]
Epoch 1/30 - Loss: 0.434 - Acc: 0.874
              Val_loss: 0.223 - Val_acc: 0.932
 37%|████      | 11/30 [00:41<01:11,  3.76s/it]
Epoch 11/30 - Loss: 0.035 - Acc: 0.989
              Val_loss: 0.063 - Val_acc: 0.981
 70%|███████   | 21/30 [01:19<00:35,  3.94s/it]
Epoch 21/30 - Loss: 0.011 - Acc: 0.997
              Val_loss: 0.077 - Val_acc: 0.981
```






```
func:'train'  took: 113.4218 sec
```

The accuracy with and without batch normalization is about the same. The model without batch normalization ran ~40 seconds faster though. This is not a big deal here but I imagine if I were running a larger dataset for more epochs the time difference could become significant.

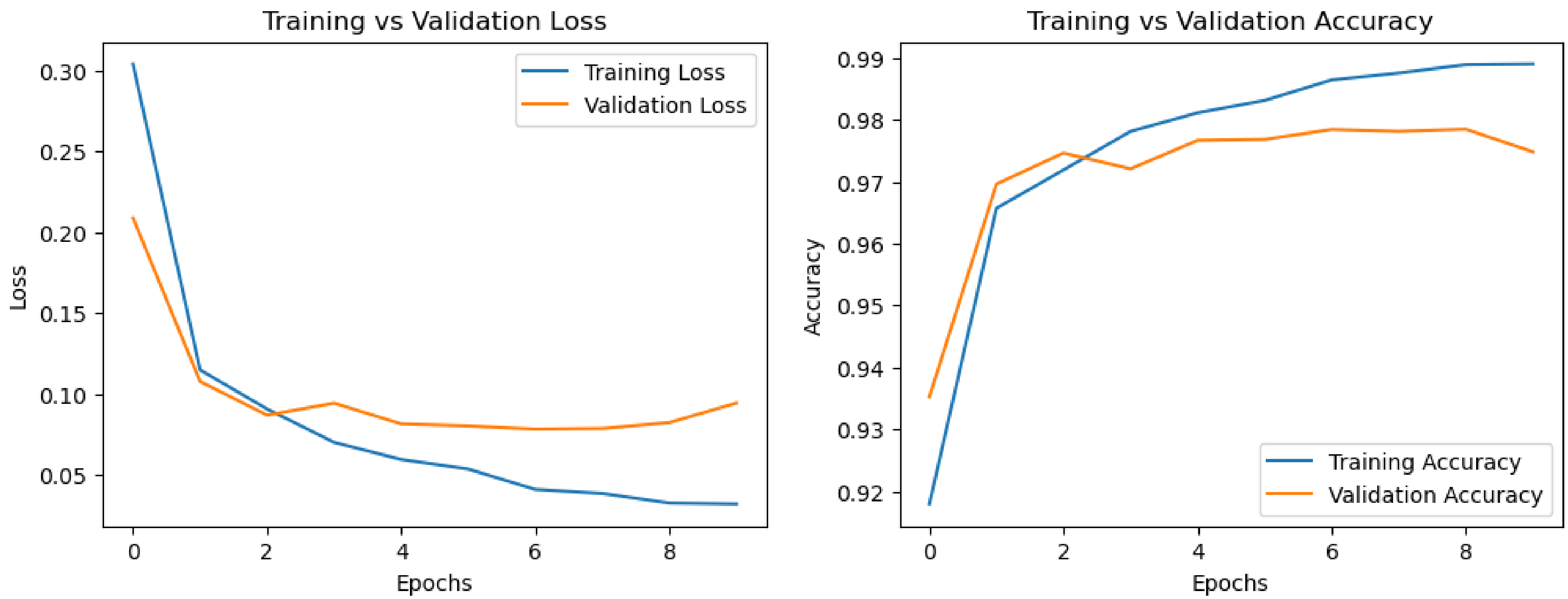
1b

Run the model with and without the skip connection at learning rate of 5e-3 for 10 epochs. Do you see faster training and/or better test accuracy with the skip connection?

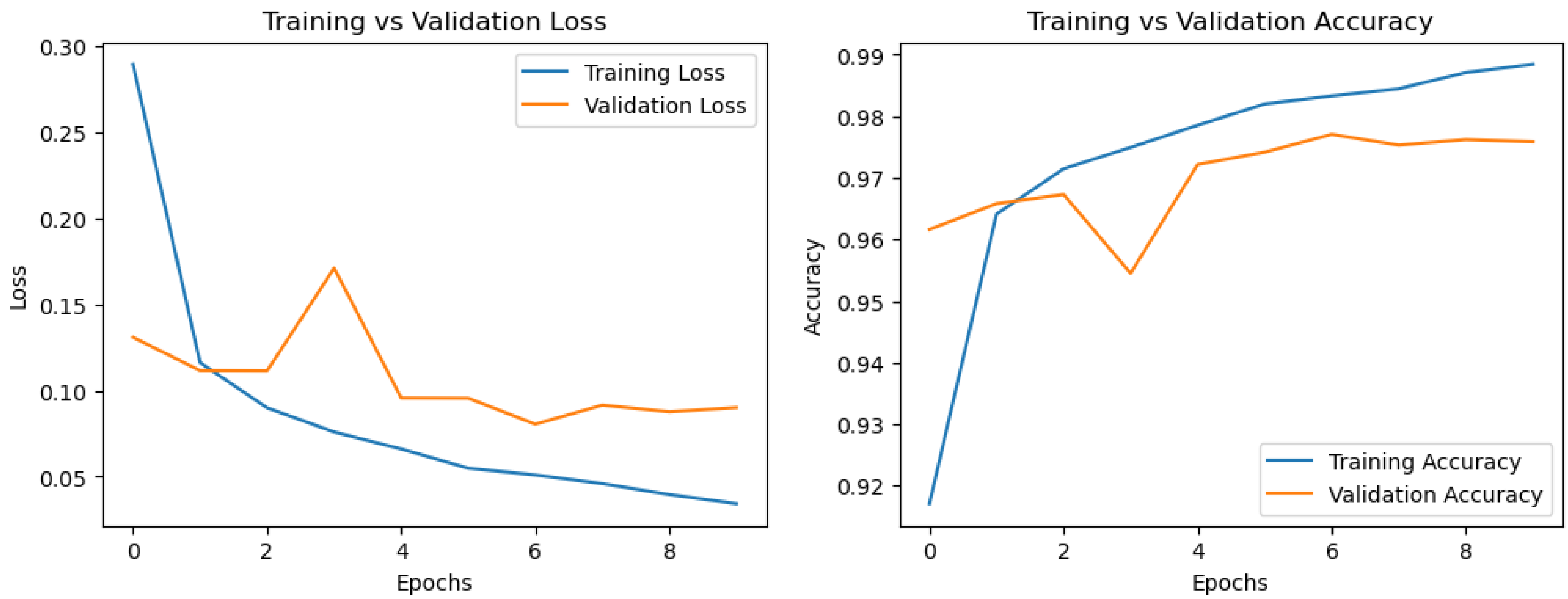
In [15]:
```
# WITH SKIP CONNECTION
kf = KFold(3, shuffle=True, random_state=49)

for idc, (train_index, val_index) in enumerate(kf.split(train_X)):
    X_train_fold, X_val_fold = train_X[train_index], train_X[val_index]
    y_train_fold, y_val_fold = train_y[train_index], train_y[val_index]

    RNN3 = RNN()
```

```
train_RNN3 = Trainer(RNN3, optimizer_type="Adam", learning_rate=5e-3, epoch=10, batch_size=128)
RNN3_results = train_RNN3.train(X_train_fold, y_train_fold, X_val_fold, y_val_fold, early_stop=True)
```

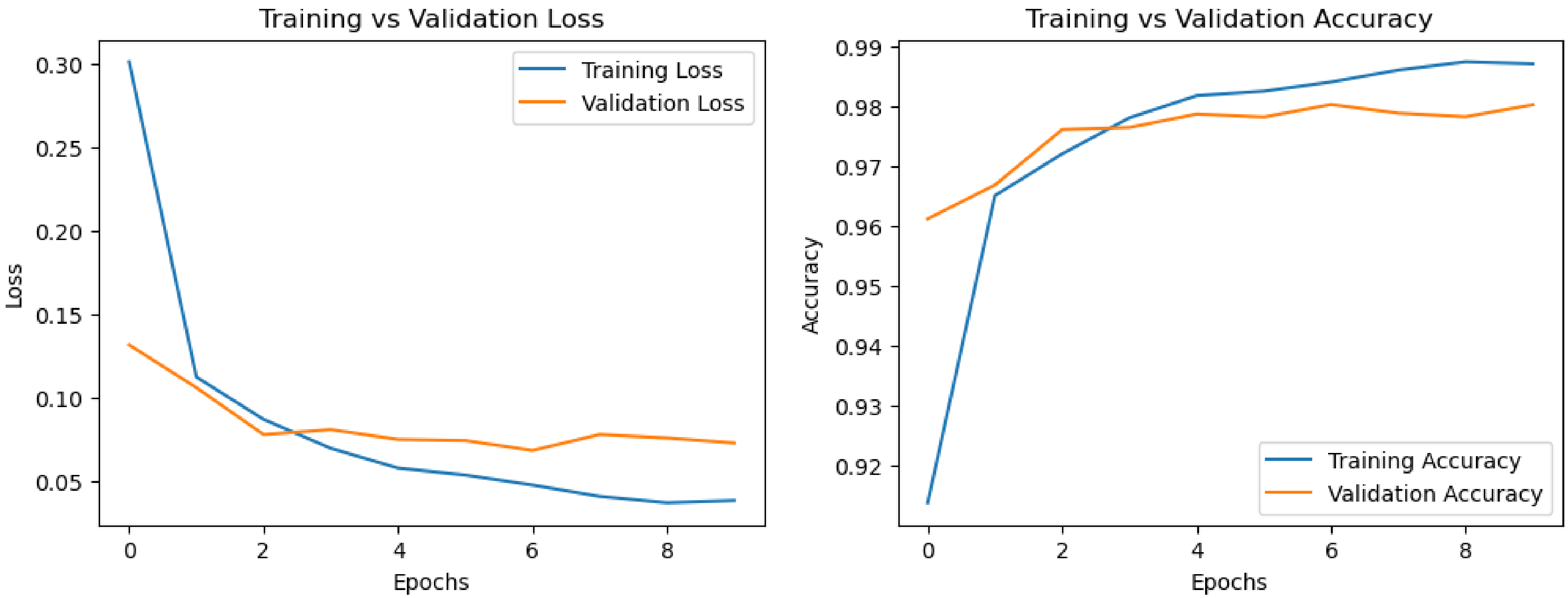
```
 10%|█          | 1/10 [00:04<00:40,  4.49s/it]
Epoch 1/10 - Loss: 0.304 - Acc: 0.918
              Val_loss: 0.209 - Val_acc: 0.935
```




```
func:'train'  took: 41.5450 sec
 10%|█          | 1/10 [00:04<00:38,  4.27s/it]
Epoch 1/10 - Loss: 0.289 - Acc: 0.917
              Val_loss: 0.131 - Val_acc: 0.962
```




```
func:'train'  took: 41.7714 sec
 10%|█         | 1/10 [00:04<00:38,  4.25s/it]
Epoch 1/10 - Loss: 0.301 - Acc: 0.914
              Val_loss: 0.132 - Val_acc: 0.961
```




```
func:'train'  took: 42.5116 sec
```

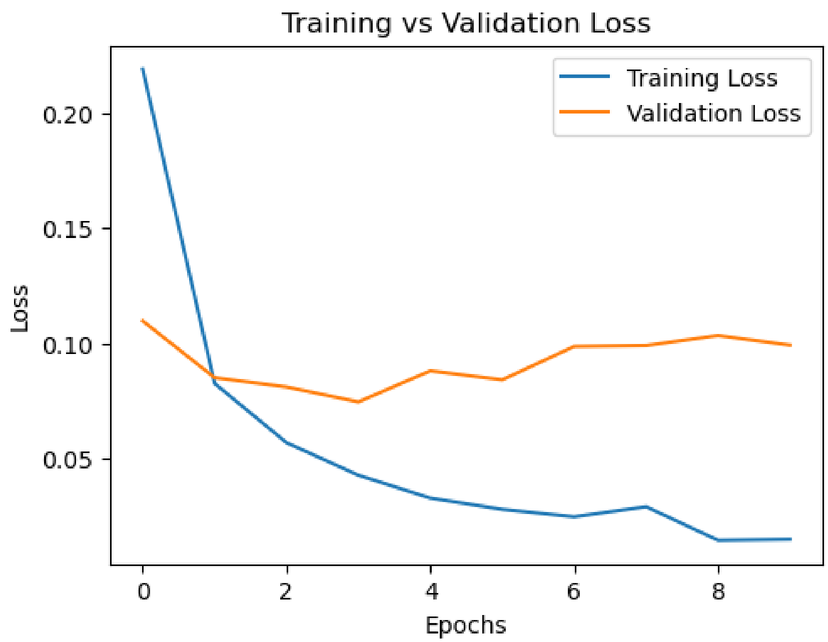
In [21]:
```
class SimpleCNN(nn.Module):
    """
    This class defines a simple convolutional neural network.
    """
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # convolution
        self.conv = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=5, stride=1, padding=2)
        # pooling
        self.pooling = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        # batch norm
        self.bn = nn.BatchNorm2d(num_features=3)
        # flatten results
        self.flatten = nn.Flatten()
        # ReLU as activation function
        self.relu = nn.ReLU()
        # fully connected layers
        self.fc1 = nn.Linear(16*16*3, 12*5*5)
        self.fc2 = nn.Linear(300, 10)
```

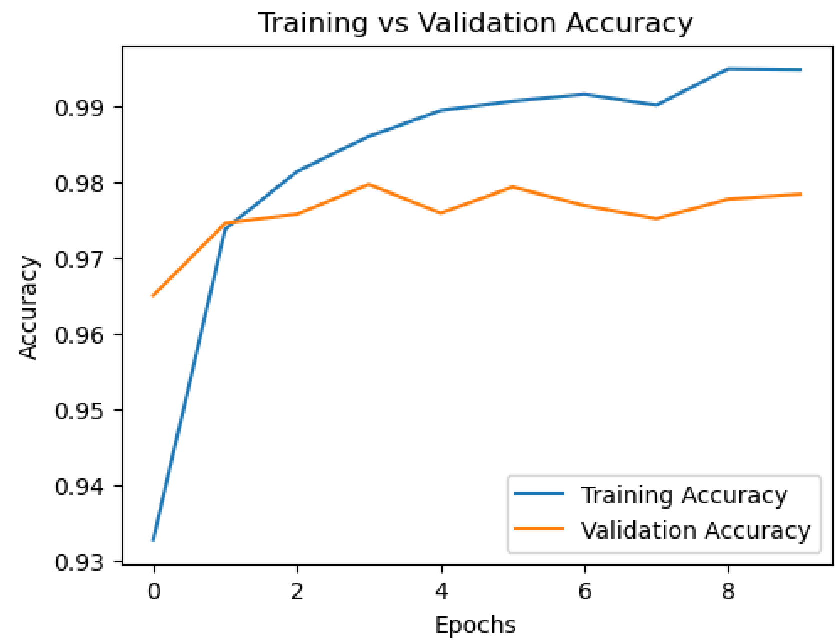
```
    def forward(self, x0):
        # conv -> batch norm -> activation -> pooling
        x1 = self.bn(self.conv(x0)) # 3 * 32 * 32
        x2 = self.relu(x1) # 3 * 32 * 32
        x3 = self.pooling(x2) # 3 * 16 * 16
        # flatten
        x4 = self.flatten(x3)
        # fully connected layer
        x5 = self.fc1(x4) # 12 * 5 * 5
        x6 = self.relu(x5)
        # second fc layer
        x7 = self.fc2(x6)

        return x7
```

In [22]:

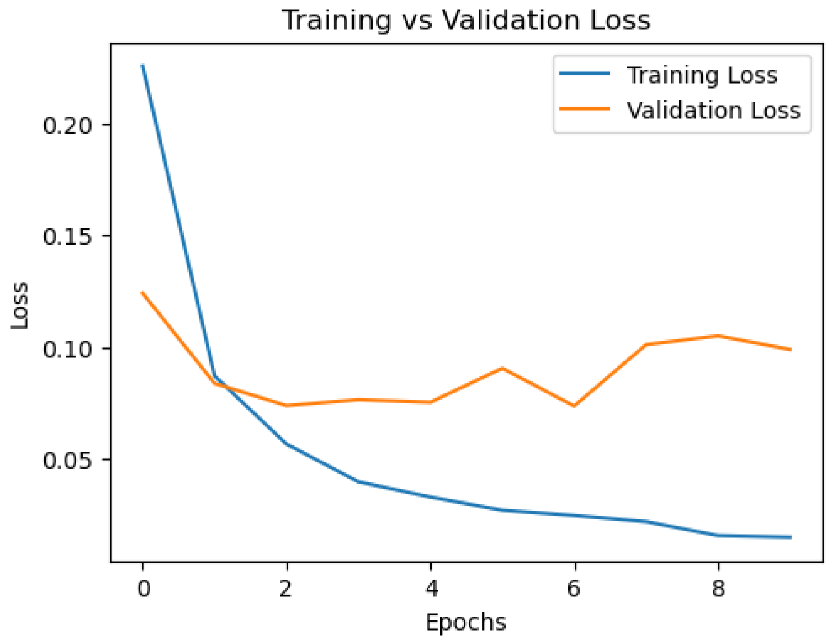
```
# WITHOUT SKIP CONNECTION
kf = KFold(3, shuffle=True, random_state=49)

for idc, (train_index, val_index) in enumerate(kf.split(train_X)):
    X_train_fold, X_val_fold = train_X[train_index], train_X[val_index]
    y_train_fold, y_val_fold = train_y[train_index], train_y[val_index]

    model = SimpleCNN()
    train_model = Trainer(model, optimizer_type="Adam", learning_rate=5e-3, epoch=10, batch_size=128)
    model_results = train_model.train(X_train_fold, y_train_fold, X_val_fold, y_val_fold, early_stop=True)
```

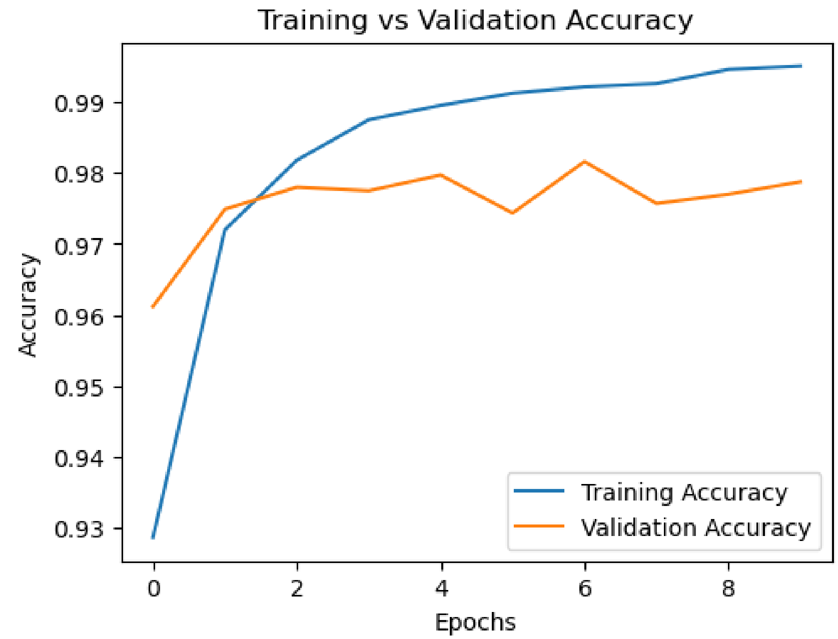
```
 10%|█         | 1/10 [00:04<00:36,  4.05s/it]
Epoch 1/10 - Loss: 0.219 - Acc: 0.933
              Val_loss: 0.110 - Val_acc: 0.965
```



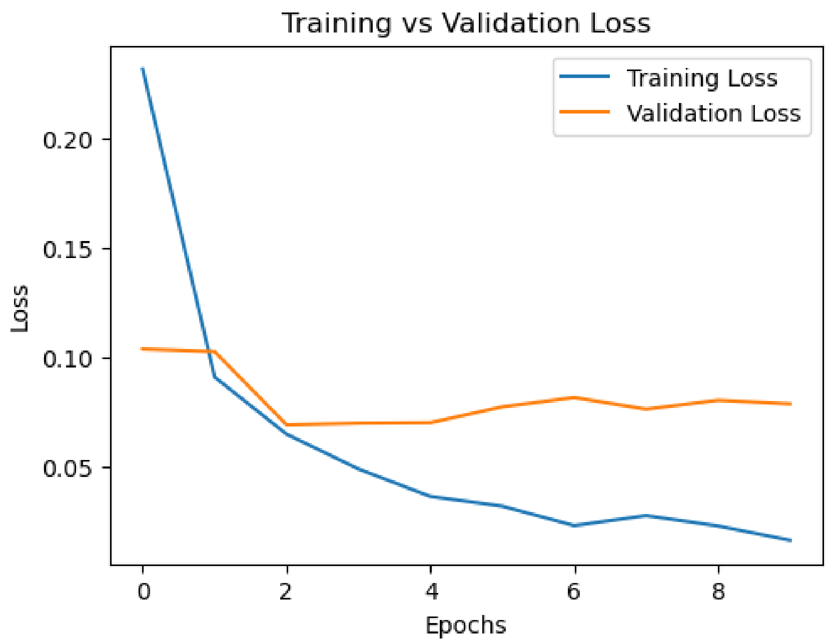


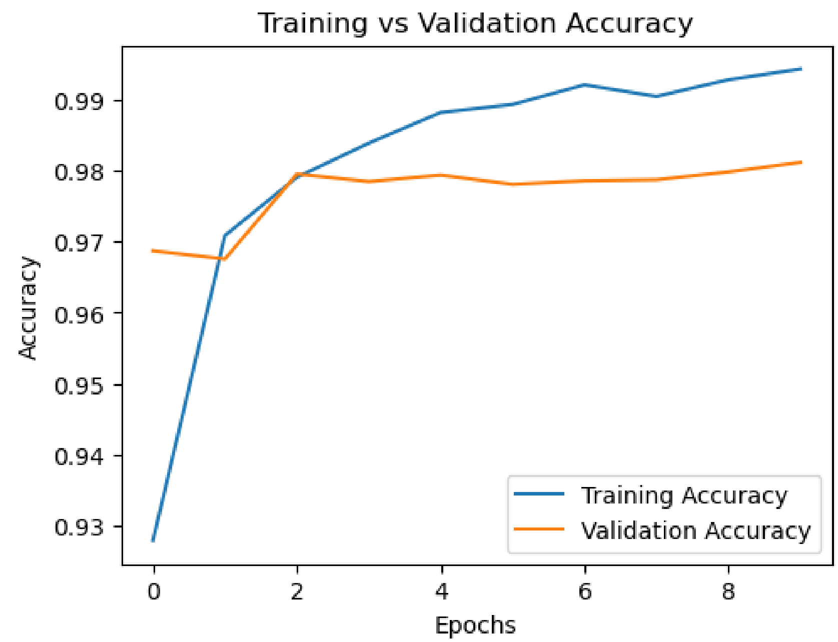

```
func:'train'  took: 41.5236 sec
 10%|█         | 1/10 [00:04<00:37,  4.19s/it]
Epoch 1/10 - Loss: 0.225 - Acc: 0.929
              Val_loss: 0.124 - Val_acc: 0.961
```






```
func:'train'  took: 40.7463 sec
 10%|█         | 1/10 [00:04<00:37,  4.17s/it]
Epoch 1/10 - Loss: 0.232 - Acc: 0.928
              Val_loss: 0.104 - Val_acc: 0.969
```






```
func:'train'  took: 41.9085 sec
```

There is no significant difference between accuracy and training time between the RNN and CNN. This is surprising, as I thought the RNN would be significantly better. Maybe if the skip connection was from earlier in the system (maybe after the first activation layer) it would be better. I'm not sure, just an idea.

```
In [ ]:
```