MSSE 277B: Machine Learning Algorithms

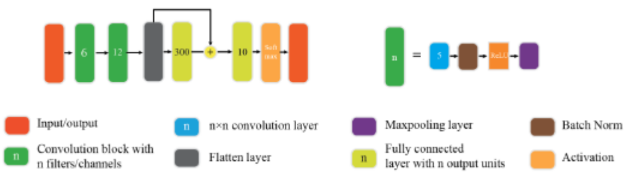
Homework assignment #9: Residual Neural Network

Assigned Apr. 6 and Due Apr. 18

Student Name: Charis Liao

1. Residual Neural Networks applied to classification. (20 pt) We will again use the MNIST data set to train, validation, and test but this time using a ResNN. As described in lecture, we are going to formulate a skip connection in order to improve gradient flow.

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. 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.



In [4]:
```python
# Import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from functools import wraps
from time import time
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, KFold
%matplotlib inline
```

(a) (10 pt) Run the model with and without batch normalization. Which give you better test accuracy?

In [5]:
```python
# import dataset
df = pd.read_pickle('mnist.pkl')
```

In [6]:
```python
# Normalize the entire data set
x_train = df[0][0] / 255.0
y_train = df[0][1]
x_test = df[1][0] / 255.0
y_test = df[1][1]
```

In [56]:
```python
class RNN(nn.Module):
    def __init__(self, batch_norm=False, skip=False):
        super(RNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=1)  # (B, 6, 32, 32)
        self.pool1 = nn.MaxPool2d(kernel_size=2)  # (B, 6, 16, 16)
        self.conv2 = nn.Conv2d(6, 12, kernel_size=3, stride=1, padding=1)  # (B, 12, 15, 15)
        self.pool2 = nn.MaxPool2d(kernel_size=3)  # (B, 12, 5, 5)
        self.fc = nn.ModuleList([nn.Linear(300, 300),nn.Linear(300,10)])
        self.activation = nn.ReLU()
        self.bn = [nn.BatchNorm2d(6), nn.BatchNorm2d(12)]
        self.batch_norm = batch_norm
        self.skip = skip

    def forward(self, x):
        x = x.view(-1, 1, 32, 32)
        x = self.conv1(x)
        if self.batch_norm:
            x = self.bn[0](x)
        x = self.pool1(self.activation(x))
        x = self.conv2(x)
        if self.batch_norm:
            x = self.bn[1](x)
        x = self.pool2(self.activation(x))
        x = x.flatten(start_dim=1)
        if self.skip:
            residual = x.clone()
        else:
            residual = 0

        x = self.fc[0](x)
        x += residual
        x = self.activation(x)
```

```python
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x
```

In [21]:
```python
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 [42]:
```python
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 chun
    '''
    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

class Trainer():
    def __init__(self, model, optimizer_type, learning_rate, epoch, batch_size, input_transform = lambda x:x.reshape(x.shape[0],-1)):
        """ 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(model.parameters(), 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 = self.input_transform(inputs)
        val_inputs = self.input_transform(val_inputs)
        inputs = torch.tensor(inputs, dtype=torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)

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

        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 = nn.CrossEntropyLoss()(batch_predictions, batch_output)
                if l2:
                    ### Compute the loss with L2 regularization ###
                    l2_lambda = 1e-5
                    l2_norm = sum([p.pow(2.0).sum() for p in self.model.parameters()])
                    loss = loss + l2_lambda * l2_norm
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                ### Compute epoch_loss and epoch_acc
```

```
                epoch_loss += loss.detach().item() * batch_importance
                pred = torch.argmax(batch_predictions, axis=-1)
                acc = torch.sum(pred == batch_output) / len(batch_predictions)
                epoch_acc += acc.detach().item() * batch_importance
            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_acc))
                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
                    weights = self.model.state_dict()

        if early_stop:
            self.model.load_state_dict(weights)

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

    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 = self.input_transform(inputs)
        inputs = torch.tensor(inputs, dtype=torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)
        self.model.eval() # change to evaluation mode so that it won't calculate gradient
        batch_indices = list(range(inputs.shape[0]))
        batch_indices = create_chunks(batch_indices, chunk_size=self.batch_size)
        acc = 0
        losses = 0
        for batch in batch_indices:
            batch_importance = len(batch) / len(outputs)
            batch_input = inputs[batch]
            batch_output = outputs[batch]
            with torch.no_grad():
                ### Compute prediction and loss###
                batch_predictions = self.model(batch_input)
                loss = nn.CrossEntropyLoss()(batch_predictions, batch_output)
            pred = torch.argmax(batch_predictions, axis=-1)
            batch_acc = torch.sum(pred == batch_output) / len(batch_predictions)
            losses += loss.detach().item() * batch_importance
            acc += batch_acc.detach().item() * batch_importance
        if print_acc:
            print("Accuracy: %.3f" % acc)
        return losses, acc
```

In [50]:
```
def training_model(batch_normalization, skip_connection, x_train, y_train, epochs=30, batch_size=128, learning_rate=1e-3, draw_curve=True, l2=
    """
    Parameters
    ----------
    splits : int
        The number of folds for cross-validation.
    model : nn.Module
        The model to be trained.
    x_train : tensor
        The training data.
    y_train : tensor
        The training labels.
    epochs : int
        The number of epochs.
    batch_size : int
        The batch size for training.
    learning_rate : float
        The learning rate for training.
    draw_curve : bool, optional
        Whether to draw the loss and accuracy curves. The default is True.
    """

    X_train, X_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.33, random_state=42)

    model = RNN(batch_norm = batch_normalization, skip = skip_connection)
    trainer = Trainer(model, "adam", learning_rate, epochs, batch_size, input_transform = lambda x:x.reshape(x.shape[0],-1))
    results = trainer.train(X_train, y_train, X_val, y_val, early_stop=True, l2=l2, silent=False)
    losses = results["losses"]
    accuracies = results["accuracies"]

    val_losses = results["val_losses"]
    val_accuracies = results["val_accuracies"]

    print(f"accuracy: {accuracies[-1]}")
    print(f"validation accuracy: {val_accuracies[-1]}")


    if draw_curve:
        plt.plot(losses, label="train")
        plt.plot(val_losses, label="val")
        plt.title(f"Training v.s. Validation Loss")
        plt.legend()
        plt.show()
```

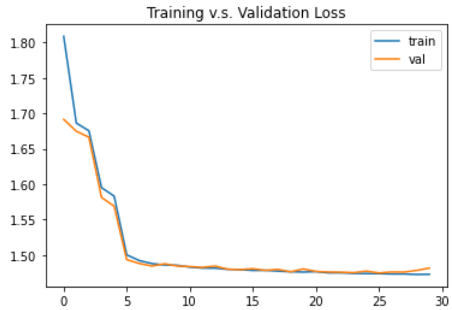
```python
        plt.plot(accuracies, label="train")
        plt.plot(val_accuracies, label="val")
        plt.title(f"Training v.s. Validation Accuracy")
        plt.legend()
        plt.show()
```

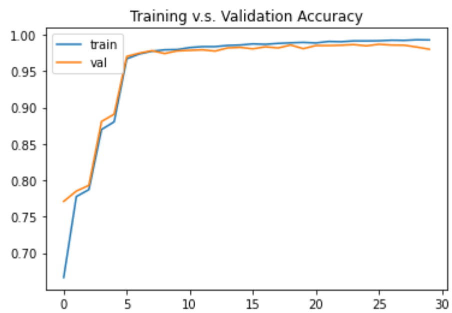
In [51]:

```python
# model without batch normalization & skip connection
training_model(False, False, x_train, y_train, epochs=30, batch_size=128, learning_rate=1e-3, draw_curve=True, l2=True)
```

```
  3%|▎         | 1/30 [00:06<03:02,  6.28s/it]
Epoch 1/30 - Loss: 1.809 - Acc: 0.666
              Val_loss: 1.692 - Val_acc: 0.771
 37%|███▋      | 11/30 [01:09<02:00,  6.32s/it]
Epoch 11/30 - Loss: 1.483 - Acc: 0.982
              Val_loss: 1.484 - Val_acc: 0.978
 70%|███████   | 21/30 [02:11<00:55,  6.16s/it]
Epoch 21/30 - Loss: 1.477 - Acc: 0.989
              Val_loss: 1.477 - Val_acc: 0.985

func:'train'  took: 188.2645 sec
accuracy: 0.9927611940298522
validation accuracy: 0.9799494949254117
```



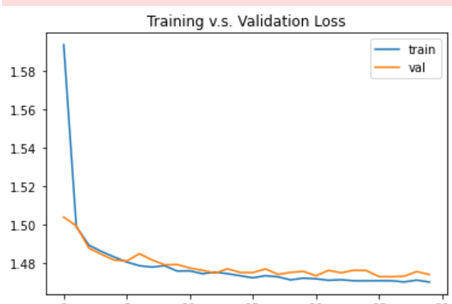


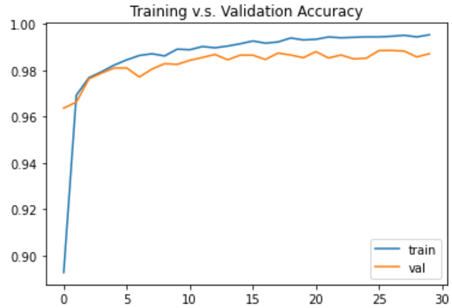
In [52]:

```python
# model with batch normalization but no skip connection
training_model(True, False, x_train, y_train, epochs=30, batch_size=128, learning_rate=1e-3, draw_curve=True, l2=True)
```

```
  3%|▎         | 1/30 [00:07<03:34,  7.41s/it]
Epoch 1/30 - Loss: 1.594 - Acc: 0.893
              Val_loss: 1.504 - Val_acc: 0.964
 37%|███▋      | 11/30 [01:20<02:19,  7.34s/it]
Epoch 11/30 - Loss: 1.476 - Acc: 0.989
              Val_loss: 1.478 - Val_acc: 0.984
 70%|███████   | 21/30 [02:32<01:04,  7.20s/it]
Epoch 21/30 - Loss: 1.472 - Acc: 0.993
              Val_loss: 1.474 - Val_acc: 0.988

func:'train'  took: 217.5281 sec
accuracy: 0.9953731343283608
validation accuracy: 0.987171717268048
```





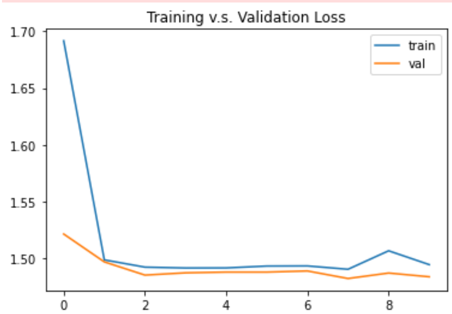
The one with batch normalization gives a higher test accuracy.

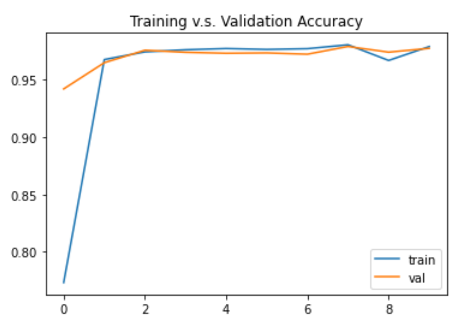
b) (10 pt) Run the model with and without the skip connection at learning rate of 5e-3 for 10 epochs. Do you see faster training and/or better test accuracy with the skip connection?

In [54]:
```
# model without skip connection
training_model(True, False, x_train, y_train, epochs=10, batch_size=128, learning_rate=5e-3, draw_curve=True, l2=True)
```

```
 10%|█         | 1/10 [00:07<01:06,  7.41s/it]
Epoch 1/10 - Loss: 1.692 - Acc: 0.773
              Val_loss: 1.521 - Val_acc: 0.942

func:'train'  took: 72.5502 sec
accuracy: 0.9790298507462685
validation accuracy: 0.9774747475710782
```

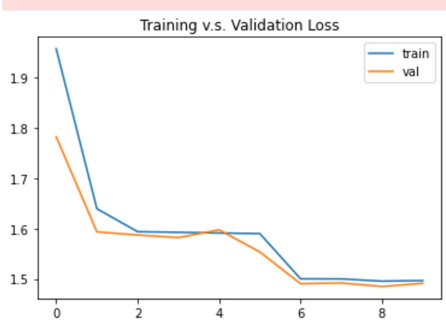


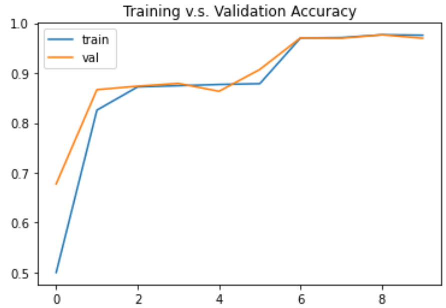


In [57]:
```
# model without skip connection
training_model(True, True, x_train, y_train, epochs=10, batch_size=128, learning_rate=5e-3, draw_curve=True, l2=True)
```

```
 10%|█         | 1/10 [00:07<01:07,  7.48s/it]
Epoch 1/10 - Loss: 1.958 - Acc: 0.500
              Val_loss: 1.783 - Val_acc: 0.678

func:'train'  took: 72.9329 sec
accuracy: 0.9757213930348255
validation accuracy: 0.969696969672887
```





I see a faster training but not better test accuracy with skip connection.

You've reached the end of homework 9


```
In [ ]:
```