1. Residual Neural Networks applied to classification. (20 pt) We will Magain 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.

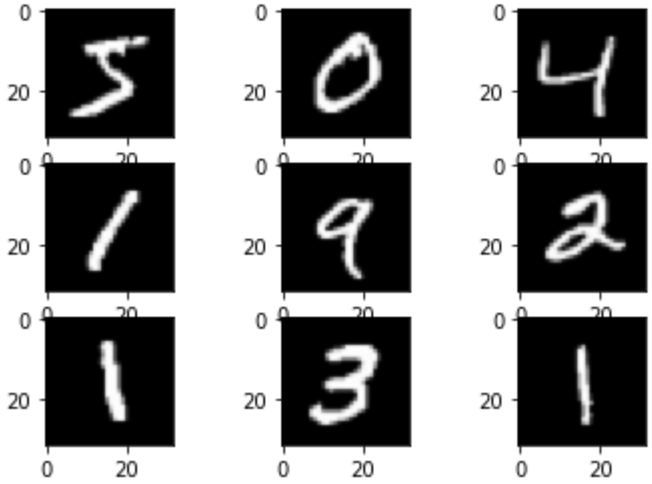
a) (10 pt) Run the model with and without batch normalization. Which give you better test accuracy?

Batch normalization yields slightly better test accuracy average in that it yields 94.9% versus 93.7% without batch normalization.

w/out batch norm

In [1]:
```
import pickle
import numpy as np
(train_X, train_y), (test_X, test_y) = pickle.load(open("mnist.pkl", "rb"))

#shape of dataset
print('X_train: ' + str(train_X.shape))
print('Y_train: ' + str(train_y.shape))
print('X_test:  '  + str(test_X.shape))
print('Y_test:  '  + str(test_y.shape))

#plotting
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure()
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(train_X[i], cmap=plt.get_cmap('gray'))
```

```
X_train: (60000, 32, 32)
Y_train: (60000,)
X_test:  (10000, 32, 32)
Y_test:  (10000,)
```



In [2]:
```
train_X_norm=train_X/255
test_X_norm=test_X/255

import torch
from torch import nn
```

In [3]:
```
torch.cuda.is_available()
```

Out[3]:
```
False
```

In [4]:
```
torch.cuda.device_count()
```

Out[4]:
```
0
```

In [5]:
```
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 [6]:

```
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

def data_gen(X,y, batchsize):
    '''
    Generator for data
    '''

    for i in range(len(X)//batchsize):
        yield X[i*batchsize:(i+1)*batchsize],y[i*batchsize:(i+1)*batchsize]
    i+=1
    yield X[i*batchsize:],y[i*batchsize:]


class Trainer():
    def __init__(self, model, optimizer_type, learning_rate, epoch, batch_si
        """ 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=
        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,draw_curve=Fals
        """ train self.model with specified arguments
        inputs: np.array, The shape of input_transform(input) should be (nda
        outputs: np.array shape (ndata,)
        val_nputs: np.array, The shape of input_transform(val_input) should
        val_outputs: np.array shape (ndata,)
        early_stop: bool
        l2: bool
        silent: bool. Controls whether or not to print the train and val err
        """


        inputs = self.input_transform(torch.tensor(inputs, dtype=torch.float
        outputs = torch.tensor(outputs, dtype=torch.int64)
        val_inputs = self.input_transform(torch.tensor(val_inputs, dtype=tor
        val_outputs = torch.tensor(val_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()
    #shuffle the data in each epoch
    idx =torch.randperm(inputs.size()[0])
    inputs=inputs[idx]
    outputs=outputs[idx]
    train_gen = data_gen(inputs,outputs,self.batch_size)

    epoch_loss = 0
    epoch_acc = 0
    for batch_input,batch_output in train_gen:
        batch_importance = len(batch_output) / len(outputs)
        batch_predictions = self.model(batch_input)
        loss = nn.CrossEntropyLoss()(batch_predictions, batch_output
        if l2:
            l2_lambda = 1e-5
            l2_norm = sum(p.pow(2.0).sum() for p in self.model.param
            loss = loss + l2_lambda * l2_norm
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        epoch_loss += loss.detach().cpu().item() * batch_importance
        acc = torch.sum(torch.argmax(batch_predictions, axis=-1) ==
        epoch_acc += acc.detach().cpu().item() * batch_importance
    val_loss, val_acc = self.evaluate(val_inputs, val_outputs, print
    if n_epoch % 10 ==0 and not silent:
        print("Epoch %d/%d - Loss: %.3f - Acc: %.3f" % (n_epoch + 1,
        print("              Val_loss: %.3f - Val_acc: %.3f" % (val_
    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 draw_curve:
    plt.figure()
    plt.plot(np.arange(self.epoch) + 1,losses,label='Training loss')
    plt.plot(np.arange(self.epoch) + 1,val_losses,label='Validation
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

if early_stop:
    self.model.load_state_dict(weights)

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

```
    def evaluate(self, inputs, outputs, print_acc=True):
        if torch.is_tensor(inputs):
            inputs = self.input_transform(inputs)
        else:
            inputs = self.input_transform(torch.tensor(inputs, dtype=torch.f
            outputs = torch.tensor(outputs, dtype=torch.int64)
        self.model.eval()
        gen = data_gen(inputs,outputs,self.batch_size)
        acc = 0
        losses = 0

        for batch_input,batch_output in gen:
            batch_importance = len(batch_output) / len(outputs)
            with torch.no_grad():
                batch_predictions = self.model(batch_input)
                loss = nn.CrossEntropyLoss()(batch_predictions, batch_output
            batch_acc = torch.sum(torch.argmax(batch_predictions, axis=-1) =
            if batch_acc>1:
                raise ValueError(batch_acc)

            losses += loss.detach().cpu().item() * batch_importance
            acc += batch_acc.detach().cpu().item() * batch_importance

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

In [7]:

```
from sklearn.model_selection import train_test_split,KFold
def Kfold(model_func,k,Xs,ys,test_Xs,test_ys,epochs,draw_curve=True,early_st
          ):
    """ Do Kfold cross validation with the specified arguments
    model_func: function.
        Constructor of the model.
    k: int. The number of fold
    Xs: np.array, The shape of Xs.reshape(input_shape) should be (ndata,nfea
    ys: np.array shape (ndata,)
    test_Xs: np.array, The shape of test_Xs.reshape(input_shape) should be (
    test_ys: np.array shape (ndata,)
    epoch: int
    batch_size: int
    early_stop: bool
    lr: float. learning_rate
    l2: bool
    optimizer: 'adam' or 'sgd'
    input_shape: tuple
    """
    # The total number of examples for training the network
    total_num=len(Xs)
    # Built in K-fold function in Sci-Kit Learn
    kf=KFold(n_splits=k,shuffle=True)
    train_acc_all=[]
    test_acc_all=[]
    fold=0
    for train_selector,val_selector in kf.split(range(total_num)):
        fold+=1
        print(f'Fold #{fold}')
        # Decide training examples and validation examples for this fold
        train_Xs, val_Xs, train_ys, val_ys = train_test_split(Xs, ys, test_s
```

```
        model=model_func()
        ### Use the trainer class to train the model ###
        trainer = Trainer(model, optimizer, lr, epochs, batchsize,)
        log=trainer.train(train_Xs, train_ys, val_Xs, val_ys,l2=l2)

        if draw_curve:
            plt.figure()
```

In [8]:

```
from torch import nn
import torch
from torchsummary import summary
            plt.title(f'Fold #{fold} loss')
```

In [9]:

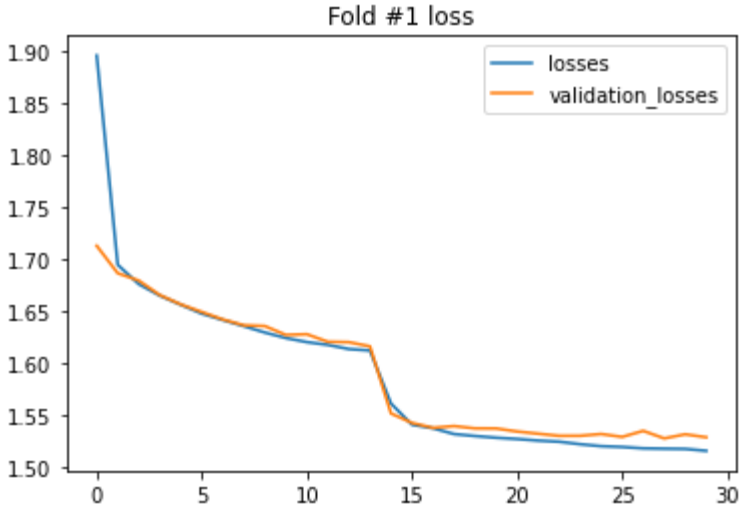
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,7,kernel_size=1),
                                  nn.Conv2d(7,12,kernel_size=2)])
        self.pooling=nn.AvgPool2d(kernel_size=5)
        self.fc = nn.ModuleList([nn.Linear(12*5*5,300),nn.Linear(300,10)])
        self.activation = nn.ReLU()


    def forward(self, x):
        for i in range(1):
            x = self.pooling(self.activation(self.conv[i](x)))
        x = nn.Flatten()(self.activation(self.conv[1](x)))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))

        return x


                                                                           r
                                                                           c
```

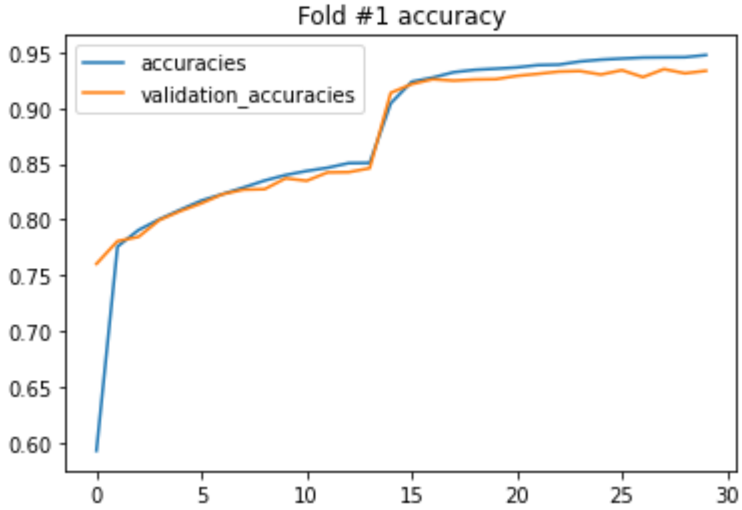
In [10]:

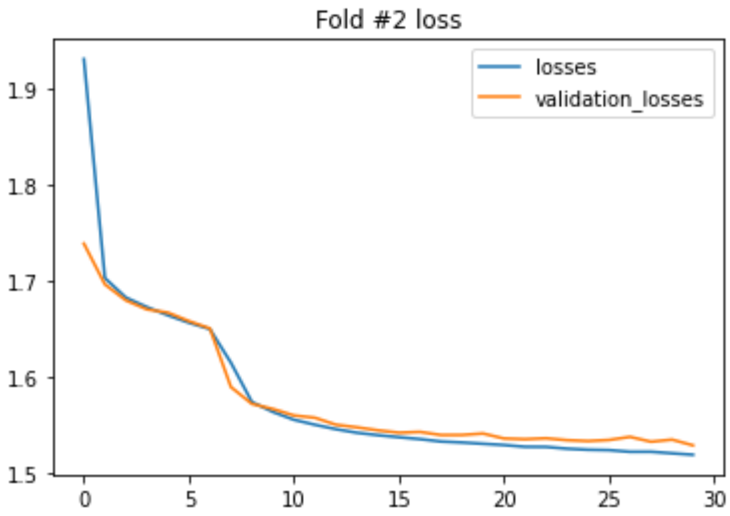
```
train_Xs, val_Xs, train_ys, val_ys = train_test_split(train_X, train_y, test
import numpy as np
model = CNN()
summary(model,(1,32,32))
# trainer = Trainer(model, "adam", 1e-3, 30, 128, input_transform=lambda x :
Kfold(CNN,2,train_X_norm,train_y,test_X_norm,test_y,30,lr=1e-3)
```

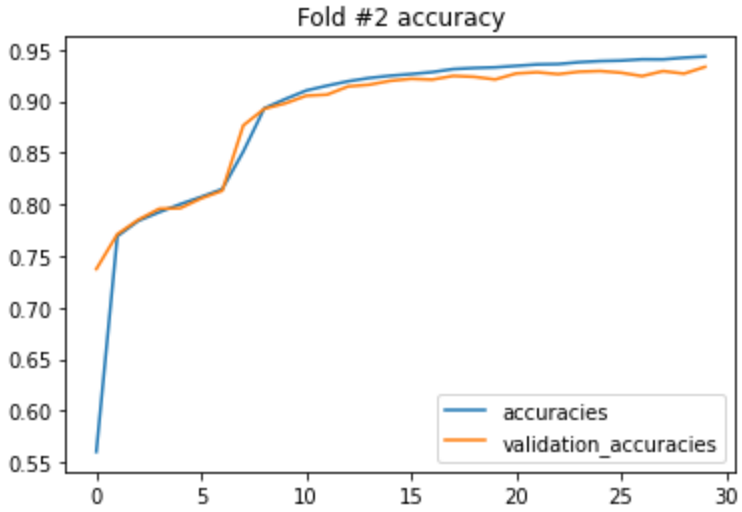
```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 7, 32, 32]           14
├─ReLU: 1-1                              [-1, 7, 32, 32]           --
├─AvgPool2d: 1-2                         [-1, 7, 6, 6]             --
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-2                       [-1, 12, 5, 5]            348
├─ReLU: 1-3                              [-1, 12, 5, 5]            --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-3                       [-1, 300]                 90,300
├─ReLU: 1-4                              [-1, 300]                 --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-4                       [-1, 10]                  3,010
===========================================================================
===============
Total params: 93,672
Trainable params: 93,672
Non-trainable params: 0
Total mult-adds (M): 0.11
===========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.36
Estimated Total Size (MB): 0.42
===========================================================================
===============
Fold #1
  3%|█▍                                          | 1/30 [00:03<01:42,  3.54
s/it]
Epoch 1/30 - Loss: 1.895 - Acc: 0.593
              Val_loss: 1.712 - Val_acc: 0.760
 37%|████████████████                           | 11/30 [00:50<01:29,  4.72
s/it]
Epoch 11/30 - Loss: 1.620 - Acc: 0.844
              Val_loss: 1.627 - Val_acc: 0.835
 70%|██████████████████████████████             | 21/30 [01:47<00:50,  5.66
s/it]
Epoch 21/30 - Loss: 1.526 - Acc: 0.936
              Val_loss: 1.534 - Val_acc: 0.929

func:'train'  took: 157.5428 sec
Train accuracy: 0.9475250000000017
Validation accuracy: 0.9334000000000006
Test accuracy: 0.9363999999999995
Fold #2
  3%|█▍                                          | 1/30 [00:04<02:04,  4.28
s/it]
Epoch 1/30 - Loss: 1.932 - Acc: 0.560
              Val_loss: 1.739 - Val_acc: 0.737
 37%|████████████████                           | 11/30 [00:46<01:25,  4.51
s/it]
```

```
Epoch 11/30 - Loss: 1.555 - Acc: 0.910
              Val_loss: 1.559 - Val_acc: 0.905
 70%|██████████████████████████████             | 21/30 [01:30<00:39,  4.38
s/it]
Epoch 21/30 - Loss: 1.529 - Acc: 0.934
              Val_loss: 1.535 - Val_acc: 0.927

func:'train'  took: 131.9492 sec
Train accuracy: 0.9434250000000013
Validation accuracy: 0.9333500000000009
Test accuracy: 0.9368
Final results:
Training accuracy:0.945475+-0.002050
Testing accuracy:0.936600+-0.000200
```









w/batch norm

In [11]:

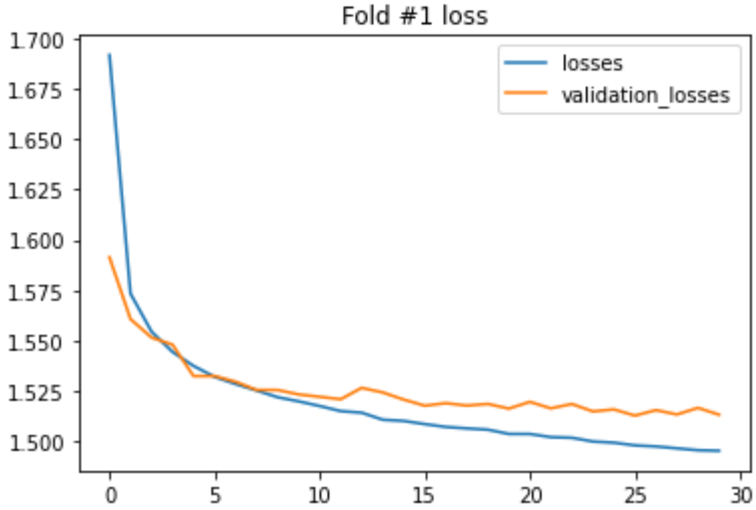
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,7,kernel_size=1),
                                   nn.Conv2d(7,12,kernel_size=2)])
        self.pooling=nn.AvgPool2d(kernel_size=5)
        self.fc = nn.ModuleList([nn.Linear(12*5*5,300),nn.Linear(300,10)])
        self.activation = nn.ReLU()
        self.bn = [nn.BatchNorm2d(7),nn.BatchNorm2d(12)]

    def forward(self, x):
        for i in range(1):
            x = self.pooling(self.activation(self.bn[i](self.conv[i](x))))
        x = nn.Flatten()(self.activation(self.bn[1](self.conv[1](x))))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x
```

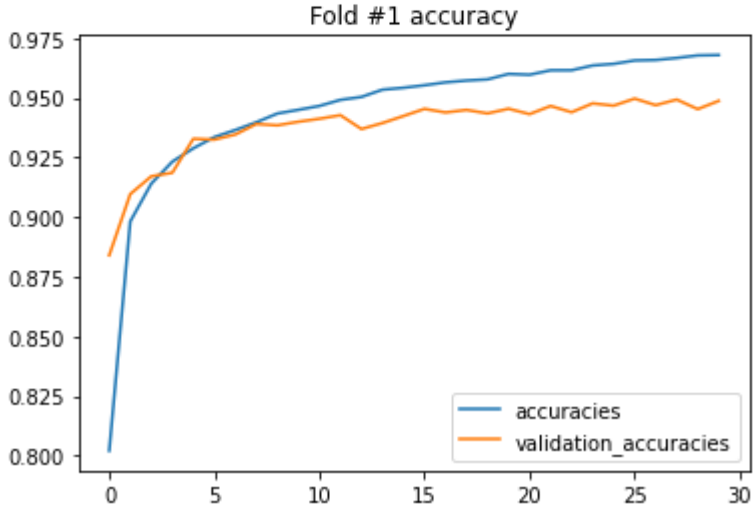
In [12]:

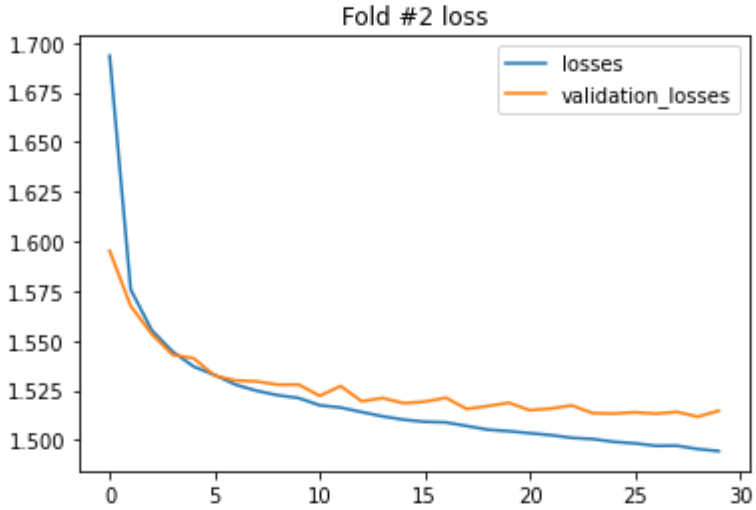
```
model = CNN()
summary(model,(1,32,32))
Kfold(CNN,2,train_X_norm,train_y,test_X_norm,test_y,30,lr=1e-3)
```

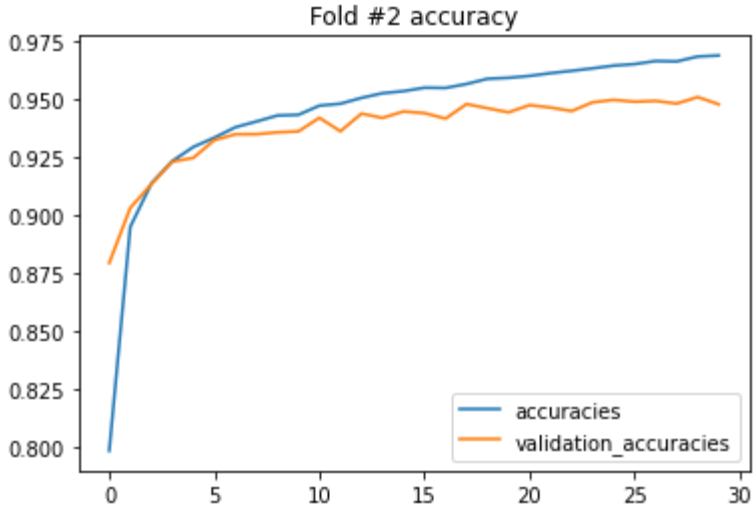
```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 7, 32, 32]           14
├─ReLU: 1-1                              [-1, 7, 32, 32]           --
├─AvgPool2d: 1-2                         [-1, 7, 6, 6]             --
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-2                       [-1, 12, 5, 5]            348
├─ReLU: 1-3                              [-1, 12, 5, 5]            --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-3                       [-1, 300]                 90,300
├─ReLU: 1-4                              [-1, 300]                 --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-4                       [-1, 10]                  3,010
===========================================================================
===============
Total params: 93,672
Trainable params: 93,672
Non-trainable params: 0
Total mult-adds (M): 0.11
===========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.36
Estimated Total Size (MB): 0.42
===========================================================================
===============
Fold #1
  3%|█▎                                          | 1/30 [00:15<07:41, 15.92
s/it]
Epoch 1/30 - Loss: 1.692 - Acc: 0.802
              Val_loss: 1.591 - Val_acc: 0.884
 37%|███████████████▋                           | 11/30 [02:51<04:55, 15.56
s/it]
Epoch 11/30 - Loss: 1.518 - Acc: 0.947
              Val_loss: 1.522 - Val_acc: 0.941
 70%|██████████████████████████████             | 21/30 [05:44<02:31, 16.88
s/it]
Epoch 21/30 - Loss: 1.504 - Acc: 0.960
              Val_loss: 1.520 - Val_acc: 0.943

func:'train'  took: 490.9896 sec
Train accuracy: 0.9679750000000017
Validation accuracy: 0.9487499999999996
Test accuracy: 0.9486999999999997
Fold #2
  3%|█▎                                          | 1/30 [00:17<08:41, 17.97
s/it]
```

```
Epoch 1/30 - Loss: 1.694 - Acc: 0.799
              Val_loss: 1.595 - Val_acc: 0.880
 37%|████████████████                           | 11/30 [03:27<05:54, 18.67
s/it]
Epoch 11/30 - Loss: 1.518 - Acc: 0.947
              Val_loss: 1.522 - Val_acc: 0.942
 70%|██████████████████████████████             | 21/30 [06:26<02:40, 17.79
s/it]
Epoch 21/30 - Loss: 1.503 - Acc: 0.960
              Val_loss: 1.515 - Val_acc: 0.947

func:'train'  took: 542.1154 sec
Train accuracy: 0.9686750000000014
Validation accuracy: 0.9476999999999999
Test accuracy: 0.9493999999999999
Final results:
Training accuracy:0.968325+-0.000350
Testing accuracy:0.949050+-0.000350
```









(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 (better test accuracy) with the skip connection?

The model with skip connection had a slightly better test accuracy of 92.6% as opposed to 92.5% without skip connection. However, the average timing for skip connection was longer to about 176 seconds versus 161 seconds without skip connection. Thus, it seems that there was some faster training results without skip connection then with. The difference seemed to about 15 seconds.

w/skip

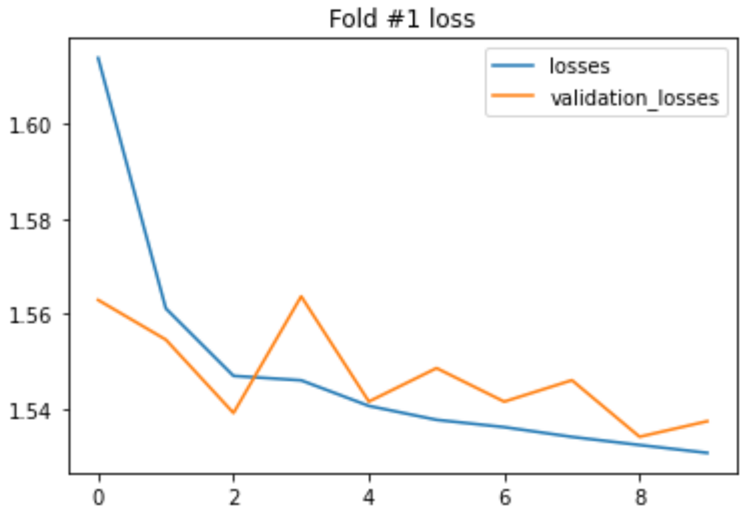
In [13]:

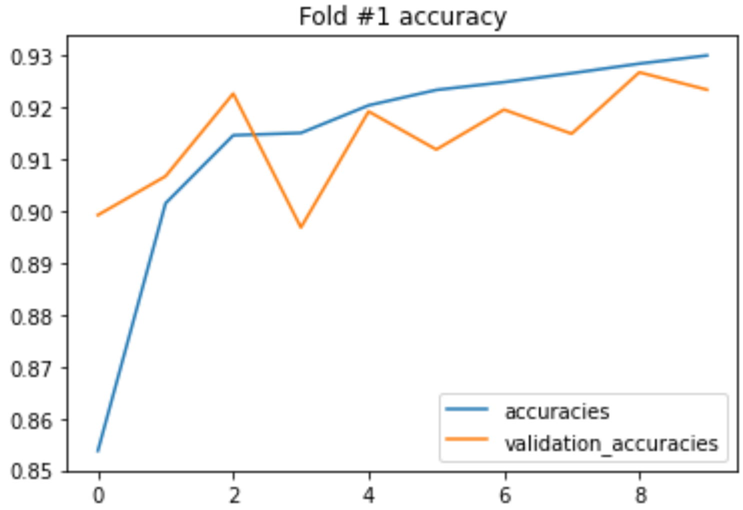
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,7,kernel_size=1),
                                   nn.Conv2d(7,12,kernel_size=2)])
        self.pooling=nn.AvgPool2d(kernel_size=5)
        self.fc = nn.ModuleList([nn.Linear(12*5*5,300),nn.Linear(300,10)])
        self.activation = nn.ReLU()
        self.bn = [nn.BatchNorm2d(7),nn.BatchNorm2d(12)]

    def forward(self, inp):
        residual = inp
        x = self.bn[0](self.conv[0](inp))
        x = x+residual
        x = self.pooling(self.activation(x))
        x = nn.Flatten()(self.activation(self.bn[1](self.conv[1](x))))
        res2 = x
        y = self.fc[0](x)
        y = y+res2
        y = self.activation(y)
        y = nn.Softmax(dim=-1)(self.fc[1](y))
        return y
```

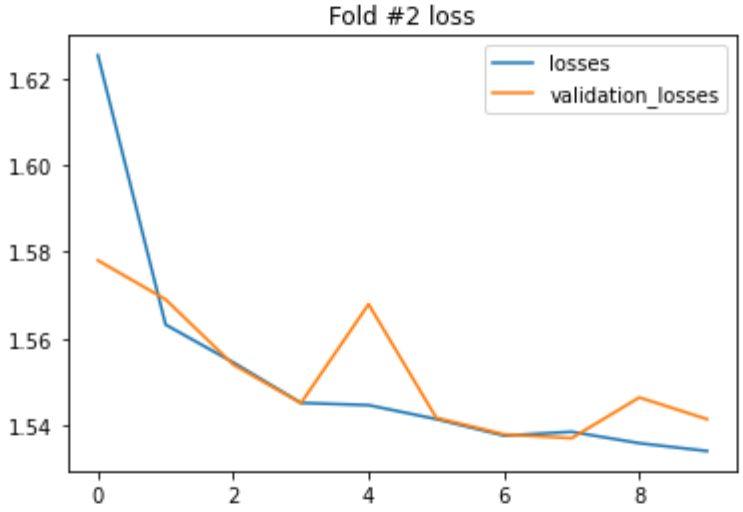
In [14]:

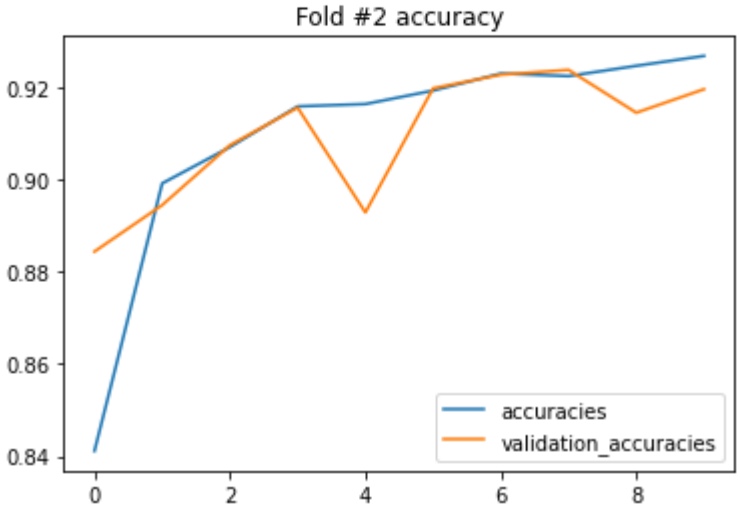
```
model = CNN()
summary(model,(1,32,32))
Kfold(CNN,2,train_X_norm,train_y,test_X_norm,test_y,10,lr=5e-3)
```

```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 7, 32, 32]           14
├─ReLU: 1-1                              [-1, 7, 32, 32]           --
├─AvgPool2d: 1-2                         [-1, 7, 6, 6]             --
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-2                       [-1, 12, 5, 5]            348
├─ReLU: 1-3                              [-1, 12, 5, 5]            --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-3                       [-1, 300]                 90,300
├─ReLU: 1-4                              [-1, 300]                 --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-4                       [-1, 10]                  3,010
===========================================================================
===============
Total params: 93,672
Trainable params: 93,672
Non-trainable params: 0
Total mult-adds (M): 0.11
===========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.36
Estimated Total Size (MB): 0.42
===========================================================================
===============
Fold #1
 10%|████                                       | 1/10 [00:16<02:32, 16.97
s/it]
Epoch 1/10 - Loss: 1.614 - Acc: 0.854
              Val_loss: 1.563 - Val_acc: 0.899

func:'train'  took: 169.1837 sec
Train accuracy: 0.9300499999999999
Validation accuracy: 0.923450000000001
Test accuracy: 0.9296999999999999
Fold #2
 10%|████                                       | 1/10 [00:17<02:40, 17.88
s/it]
Epoch 1/10 - Loss: 1.625 - Acc: 0.841
              Val_loss: 1.578 - Val_acc: 0.884

func:'train'  took: 182.4464 sec
Train accuracy: 0.9268750000000002
Validation accuracy: 0.9196500000000014
Test accuracy: 0.9232000000000001
Final results:
Training accuracy:0.928463+-0.001587
Testing accuracy:0.926450+-0.003250
```


Fold #1 loss


Fold #1 accuracy


Fold #2 loss

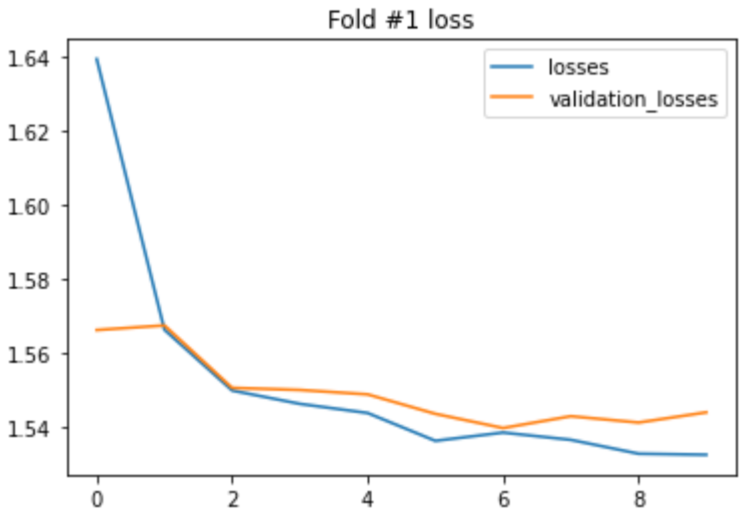


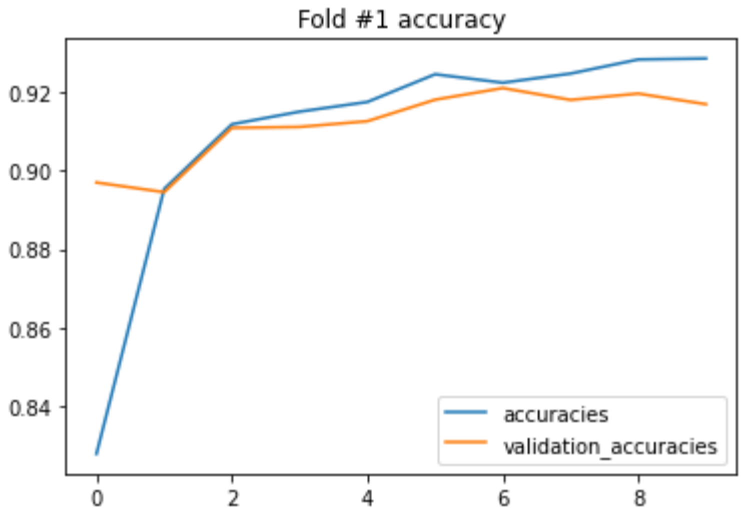
w/out skip

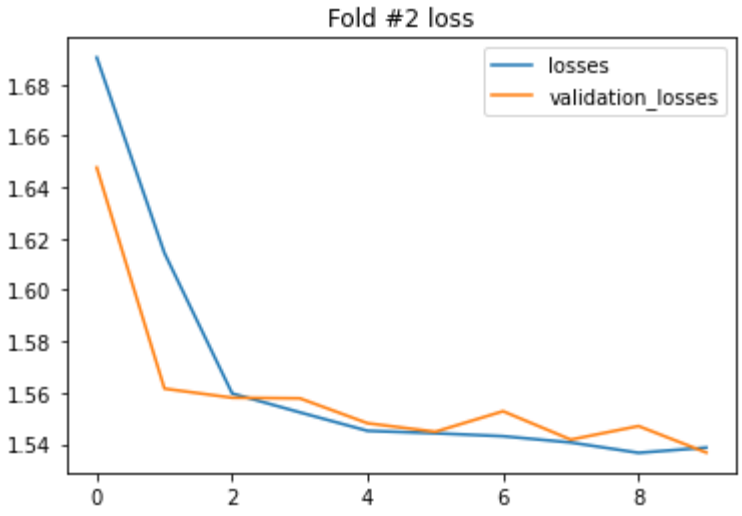
In [15]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,7,kernel_size=1),
                                   nn.Conv2d(7,12,kernel_size=2)])
        self.pooling=nn.AvgPool2d(kernel_size=5)
        self.fc = nn.ModuleList([nn.Linear(12*5*5,300),nn.Linear(300,10)])
        self.activation = nn.ReLU()
        self.bn = [nn.BatchNorm2d(7),nn.BatchNorm2d(12)]

    def forward(self, x):
        for i in range(1):
            x = self.pooling(self.activation(self.bn[i](self.conv[i](x))))
        x = nn.Flatten()(self.activation(self.bn[1](self.conv[1](x))))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x
```

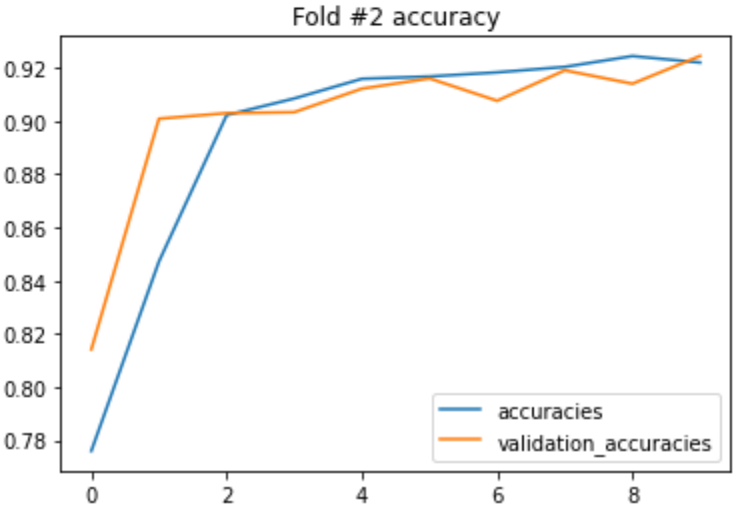
In [16]:
```
model = CNN()
summary(model,(1,32,32))
Kfold(CNN,2,train_X_norm,train_y,test_X_norm,test_y,10,lr=5e-3)
```

```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 7, 32, 32]           14
├─ReLU: 1-1                              [-1, 7, 32, 32]           --
├─AvgPool2d: 1-2                         [-1, 7, 6, 6]             --
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-2                       [-1, 12, 5, 5]            348
├─ReLU: 1-3                              [-1, 12, 5, 5]            --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-3                       [-1, 300]                 90,300
├─ReLU: 1-4                              [-1, 300]                 --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-4                       [-1, 10]                  3,010
===========================================================================
===============
Total params: 93,672
Trainable params: 93,672
Non-trainable params: 0
Total mult-adds (M): 0.11
===========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.36
Estimated Total Size (MB): 0.42
===========================================================================
===============
Fold #1
 10%|████                                        | 1/10 [00:16<02:24, 16.03
s/it]
Epoch 1/10 - Loss: 1.640 - Acc: 0.828
              Val_loss: 1.566 - Val_acc: 0.897

func:'train'  took: 160.6669 sec
Train accuracy: 0.92835
Validation accuracy: 0.9167500000000017
Test accuracy: 0.9231000000000003
Fold #2
 10%|████                                        | 1/10 [00:15<02:23, 15.89
s/it]
Epoch 1/10 - Loss: 1.691 - Acc: 0.776
              Val_loss: 1.648 - Val_acc: 0.814

func:'train'  took: 160.6686 sec
Train accuracy: 0.9220250000000004
Validation accuracy: 0.9245000000000011
Test accuracy: 0.9271999999999998
Final results:
Training accuracy:0.925188+-0.003162
Testing accuracy:0.925150+-0.002050
```


Fold #1 loss
losses
validation_losses
1.64
1.62
1.60
1.58
1.56
1.54
0
2
4
6
8


Fold #1 accuracy
accuracies
validation_accuracies
0.92
0.90
0.88
0.86
0.84
0
2
4
6
8


Fold #2 loss
losses
validation_losses
1.68
1.66
1.64
1.62
1.60
1.58
1.56
1.54
0
2
4
6
8


Fold #2 accuracy
0.92
0.90
0.88
0.86
0.84
0.82
0.80
0.78
0
2
4
6
8
accuracies
validation_accuracies