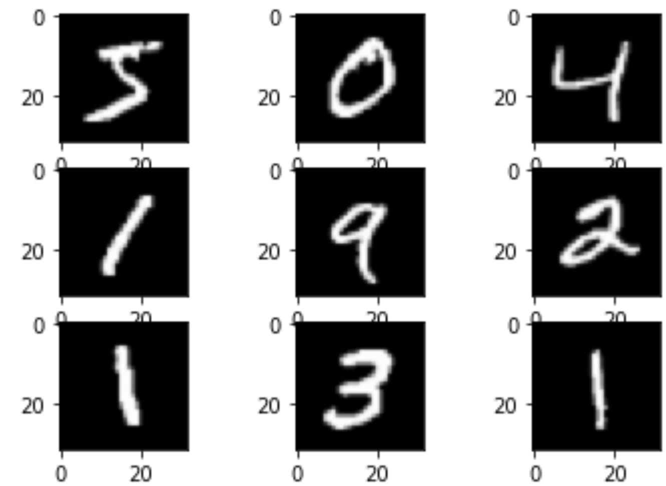
1. Convolutional Neural Networks applied to classification. We will again use the MNIST data set to train, validation, and test but this time using a CNN. As described in lecture, 2D convolutional neural nets are specified by various hyperparameters: a receptive field (filter size, DxWxH), number of filters K, stride S, amount of zero padding P, and type of pooling. We will represent our input data, as well as the hidden layers, as 3D-arrays. Since MNIST images are black-and-white and thus have scalar-valued pixels, the depth of the input image is 1.

(a) (4.5pt) Calculate the dimensionality of the output for the following convolutions sequentially applied

to a black and white MNIST input: (i). Convolution Filter size of 2x2, number of filters 33, stride of 2, padding of 0
(ii). Convolution Filter size of 3x3, number of filters 55, stride of 1, padding of 1 (iii). Convolution Filter size of 3x3, number of filters 77, stride of 1, padding of 1. Followed by a Max Pooling with filter size of 2x2 and stride 2.

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
```

In [3]:

```
from pylab import *
from tqdm import tqdm
from sklearn.model_selection import train_test_split

def train_and_val(model,train_X,train_y,epochs,draw_curve=False,tensorboard_
    """
    Parameters
    --------------
    model: a PyTorch model
    train_X: np.array shape(ndata,nfeatures)
    train_y: np.array shape(ndata)
    epochs: int
    draw_curve: bool
    """
    loss_func = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=5e-4)
    train_X = torch.tensor(train_X, dtype=torch.float)
    train_y = torch.tensor(train_y, dtype=torch.long)
    val_array=[]

    # Split training examples further into training and validation
    train_X,val_X,train_y,val_y=train_test_split(train_X,train_y,test_size=0
    weights = model.state_dict()
    lowest_val_loss = np.inf

    for i in tqdm(range(epochs)):
        pred = model(train_X)
        # in order to work with cross entropy loss, we shift the classes fro
        loss = loss_func(pred, train_y-1)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        #validation
        with torch.no_grad():
            pred = model(val_X)
            val_loss = loss_func(pred, val_y-1)
        val_array.append(val_loss.item())

        if val_loss < lowest_val_loss:
                lowest_val_loss = val_loss
                weights = model.state_dict()

        acc = calculate_accuracy_NN(model,train_X,train_y)
        val_acc = calculate_accuracy_NN(model,val_X,val_y)
        if tensorboard_logger is not None:
                tensorboard_logger.add_scalar("losses", loss, i + 1)
                tensorboard_logger.add_scalar("accuracies", acc, i + 1)
                tensorboard_logger.add_scalar("val_losses", val_loss, i + 1)
                tensorboard_logger.add_scalar("val_accuracies", val_acc, i +

     # The final number of epochs is when the minimum error in validation se
    final_epochs=np.argmin(val_array)+1
    print("Number of epochs with lowest validation:",final_epochs)
    # Recover the model weight, and train with full training data (including
    model.load_state_dict(weights)

    if draw_curve:
```

```
        plt.figure()
        plt.plot(np.arange(len(val_array))+1,val_array,label='Validation los
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

def calculate_accuracy_NN(model,xs,ys):
    with torch.no_grad():
        if not torch.is_tensor(xs):
            xs = torch.tensor(xs,dtype=torch.float)
        pred = model(xs)
        pred= torch.argmax(pred,dim=1)
    pred = pred.detach().numpy()
    if torch.is_tensor(ys):
        ys = ys.detach().numpy()
    return np.sum(ys==pred+1)/len(ys)
```

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{ padding } - \text{dilation} \times (\text{ kernel\_size } - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

In [4]:
```
def h_out(h_in,padding,dilation,kernel_size,stride,num_filter,batch_size):
    h_out_2=(h_in+2*padding-dilation*(kernel_size-1)-1)/stride + 1
    return f'The batch size, number of filters, stride(s) and padding are {[
```

1a)(i)Convolution Filter size of 2x2, number of filters 33, stride of 2, padding of 0

In [5]:
```
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,33,kernel_size=2,stride=2,pac

    def forward(self, x):
        x = nn.Flatten()(self.conv[0](x))
        return x

model = LeNet()
summary(model,(1,32,32))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 33, 16, 16]          165
==========================================================================================
Total params: 165
Trainable params: 165
Non-trainable params: 0
Total mult-adds (M): 0.03
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.00
Estimated Total Size (MB): 0.07
==========================================================================================
```

Out[5]:


```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 33, 16, 16]          165
==========================================================================================
Total params: 165
Trainable params: 165
Non-trainable params: 0
Total mult-adds (M): 0.03
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.00
Estimated Total Size (MB): 0.07
==========================================================================================
```


1a)(ii)Convolution Filter size of 3x3, number of filters 55, stride of 1, padding of 1

In [6]:

```
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,55,kernel_size=3,stride=1,pac

    def forward(self, x):
        x = nn.Flatten()(self.conv[0](x))
        return x

model = LeNet()
summary(model,(1,32,32))
```

```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 55, 32, 32]          550
==========================================================================
===============
Total params: 550
Trainable params: 550
Non-trainable params: 0
Total mult-adds (M): 0.51
==========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.43
Params size (MB): 0.00
Estimated Total Size (MB): 0.44
==========================================================================
===============
```

Out[6]:


```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 55, 32, 32]          550
==========================================================================
===============
Total params: 550
Trainable params: 550
Non-trainable params: 0
Total mult-adds (M): 0.51
==========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.43
Params size (MB): 0.00
Estimated Total Size (MB): 0.44
==========================================================================
===============
```

1a)(iii) Convolution Filter size of 3x3, number of filters 77, stride of 1, padding of 1. Followed by a Max Pooling with filter size of 2x2 and stride 2.

In [7]:

```
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.Conv2d(1,77,kernel_size=3,stride=1,padding=1)
        self.max_pool=nn.MaxPool2d(kernel_size=2,stride=2)


    def forward(self, x):

        x = self.max_pool(self.conv(x))
        x = nn.Flatten()(x)
        return x

model = LeNet()
summary(model,(1,32,32))
```

```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─Conv2d: 1-1                            [-1, 77, 32, 32]          770
├─MaxPool2d: 1-2                         [-1, 77, 16, 16]          --
===========================================================================
===============
Total params: 770
Trainable params: 770
Non-trainable params: 0
Total mult-adds (M): 0.71
===========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.60
Params size (MB): 0.00
Estimated Total Size (MB): 0.61
===========================================================================
===============
```

Out[7]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 77, 32, 32]          770
├─MaxPool2d: 1-2                         [-1, 77, 16, 16]          --
==========================================================================================
Total params: 770
Trainable params: 770
Non-trainable params: 0
Total mult-adds (M): 0.71
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.60
Params size (MB): 0.00
Estimated Total Size (MB): 0.61
==========================================================================================
```

1(b) (4.5pt) The MNIST data set was, in fact, in color (RGB). This means the depth of the input image would

be 3. Calculate the dimensionality of the output for the following convolutions sequentially applied to a RGB MNIST input: (i). Convolution Filter size of 2x2, number of filters 33, stride of 2, padding of 0
(ii). Convolution Filter size of 3x3, number of filters 55, stride of 1, padding of 1. Followed by a max pooling layer of kernel size 3x3, stride of 1, padding of 0 (iii). Convolution Filter size of 3x3, number of filters 77, stride of 1, padding of 1. Followed by a Max Pooling with filter size of 2x2 and stride 2.

1(b)(i). Convolution Filter size of 2x2, number of filters 33, stride of 2, padding of 0

In [8]:
```
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.Conv2d(3,33,kernel_size=2,stride=2,padding=0,dilatior

    def forward(self, x):
        x = nn.Flatten()(self.conv(x))
        return x


model = LeNet()
summary(model,(3,32,32))
```

```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
├─Conv2d: 1-1                            [-1, 33, 16, 16]          429
==========================================================================
===============
Total params: 429
Trainable params: 429
Non-trainable params: 0
Total mult-adds (M): 0.10
==========================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.06
Params size (MB): 0.00
Estimated Total Size (MB): 0.08
==========================================================================
===============
```

Out[8]:

```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
├─Conv2d: 1-1                            [-1, 33, 16, 16]          429
==========================================================================
===============
Total params: 429
Trainable params: 429
Non-trainable params: 0
Total mult-adds (M): 0.10
==========================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.06
Params size (MB): 0.00
Estimated Total Size (MB): 0.08
==========================================================================
===============
```

1b)(ii). Convolution Filter size of 3x3, number of filters 55, stride of 1, padding of 1. Followed by a max pooling layer of kernel size 3x3, stride of 1, padding of 0

In [9]:
```
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.Conv2d(3,55,kernel_size=3,stride=1,padding=1)
        self.max_pool=nn.MaxPool2d(kernel_size=3,stride=1,padding=0)


    def forward(self, x):
        x = self.max_pool(self.conv(x))
        x = nn.Flatten()(x)
        return x


model = LeNet()
summary(model,(3,32,32))
```

```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─Conv2d: 1-1                            [-1, 55, 32, 32]          1,540
├─MaxPool2d: 1-2                         [-1, 55, 30, 30]          --
===========================================================================
===============
Total params: 1,540
Trainable params: 1,540
Non-trainable params: 0
Total mult-adds (M): 1.52
===========================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.43
Params size (MB): 0.01
Estimated Total Size (MB): 0.45
===========================================================================
===============
```

Out[9]:
```
=============================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
=============================================================================
===============
├─Conv2d: 1-1                            [-1, 55, 32, 32]          1,540
├─MaxPool2d: 1-2                         [-1, 55, 30, 30]          --
=============================================================================
===============
Total params: 1,540
Trainable params: 1,540
Non-trainable params: 0
Total mult-adds (M): 1.52
=============================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.43
Params size (MB): 0.01
Estimated Total Size (MB): 0.45
=============================================================================
===============
```

1b)(iii). Convolution Filter size of 3x3, number of filters 77, stride of 1, padding of 1. Followed by a Max Pooling with filter size of 2x2 and stride 2.

In [10]:
```python
from torch import nn
import torch
from torchsummary import summary

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.Conv2d(3,77,kernel_size=3,stride=1,padding=1,dilatior
        self.max_pool=nn.MaxPool2d(kernel_size=2,stride=2)



    def forward(self, x):
        x = self.max_pool(self.conv(x))
        x = nn.Flatten()(x)
        return x


model = LeNet()
summary(model,(3,32,32))
```

```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─Conv2d: 1-1                            [-1, 77, 32, 32]          2,156
├─MaxPool2d: 1-2                         [-1, 77, 16, 16]          --
===========================================================================
===============
Total params: 2,156
Trainable params: 2,156
Non-trainable params: 0
Total mult-adds (M): 2.13
===========================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.60
Params size (MB): 0.01
Estimated Total Size (MB): 0.62
===========================================================================
===============
```

Out[10]:


```
===========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
===========================================================================
===============
├─Conv2d: 1-1                            [-1, 77, 32, 32]          2,156
├─MaxPool2d: 1-2                         [-1, 77, 16, 16]          --
===========================================================================
===============
Total params: 2,156
Trainable params: 2,156
Non-trainable params: 0
Total mult-adds (M): 2.13
===========================================================================
===============
Input size (MB): 0.01
Forward/backward pass size (MB): 0.60
Params size (MB): 0.01
Estimated Total Size (MB): 0.62
===========================================================================
===============
```

1(c) (5pt) Next, implement a CNN to see if we can extract additional features from the MNIST data. For this

start with one convolutional layer with a 5x5 kernel, with stride of 1, zero-padding of size 2, and 3 output channels. Flatten the resulting feature maps and add a second layer of fully connected (FC) layer to the 10- neuron output layer. Use ReLU as your activation function. 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). Use mini-batches of data and converge your training to where the loss function is minimal, and choose some regularization techniques. Using 3-fold cross-validation and report your average test accuracy.

In [11]:
```
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 [12]:
```
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
import torch.nn

def create_chunks(complete_list, chunk_size=None, num_chunks=None):
    '''
    Cut a list into multiple chunks, each having chunk_size (the last chunk
    '''
    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_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,early_stop=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

        @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).reshape(self.input_

        outputs = torch.tensor(outputs, dtype=torch.int64)

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

        loss_func = torch.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.bat
            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]
#                 print(f'batch output shape {batch_output.shape}')
                ### make prediction and compute loss with loss function of y
                batch_predictions = self.model(batch_input)
                loss=loss_func(batch_predictions,batch_output)
#                 print(f'batch prediction shape {batch_predictions.shape}',
                if l2:
                    ### Compute the loss with L2 regularization ###
                    l2_norm = sum([p.pow(2.0).sum().detach().numpy() for p i
                    l2_lambda = 1e-10
                    loss = loss + l2_norm * l2_lambda
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                ### Compute epoch_loss and epoch_acc

                epoch_loss += (loss.item() * batch_importance)
                epoch_acc += (torch.argmax(batch_predictions, axis=1).eq(bat


            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 early_stop:
            self.model.load_state_dict(weights)
```

```
        return {"losses": losses, "accuracies": accuracies, "val_losses": 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 (nda
        outputs: np.array shape (ndata,)
        print_acc: bool

        @return
        losses: float
        acc: float
        """
        inputs = torch.tensor(inputs, dtype=torch.float).reshape(self.input_
        outputs = torch.tensor(outputs, dtype=torch.int64)
        batch_indices = list(range(inputs.shape[0]))
        batch_indices = create_chunks(batch_indices, chunk_size=self.batch_s
        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 = torch.nn.CrossEntropyLoss()(batch_predictions,batch_o

            batch_acc = torch.argmax(batch_predictions, axis=1).eq(batch_out
            losses += loss.detach().item() * batch_importance
            acc += batch_acc * batch_importance

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

In [13]:

```
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
          input_shape=(-1,1,32,32)):
    """ 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
```

In [14]:

```
from torch import nn
import torch
class LeNet(nn.Module):
    def __init__(self):                                                    a
        super(LeNet, self).__init__()
        self.conv = nn.Conv2d(1,3,kernel_size=5,stride=1,padding=2)
        self.fc = nn.Linear(3*32*32,10)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = nn.Flatten()(self.activation(self.conv(x)))
        x = self.activation(self.fc(x))

        return x
```

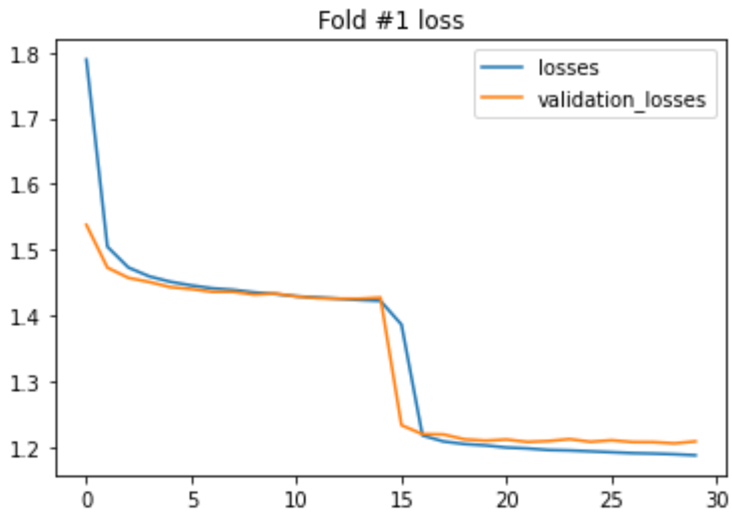
```
            plt.plot(log["val_accuracies"], label="validation_accuracies")
            plt.legend()
            plt.title(f'Fold #{fold} accuracy')

        # Report result for this fold
        if early_stop:
            report_idx= np.argmin(log["val_losses"])
        else:
            report_idx=-1
        test_acc=trainer.evaluate(test_Xs,test_ys,print_acc=False)[1]
        train_acc_all.append(log["accuracies"][report_idx])
        test_acc_all.append(test_acc)
        print("Train accuracy:",log["accuracies"][report_idx])
        print("Validation accuracy:",log["val_accuracies"][report_idx])
```

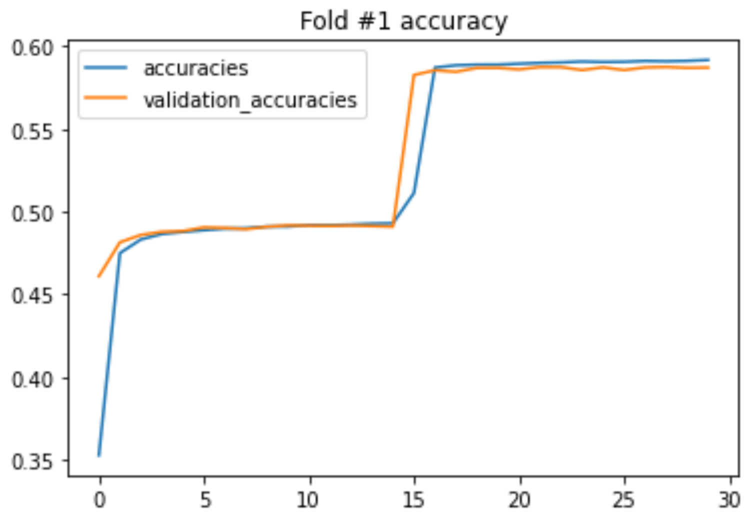
In [15]:

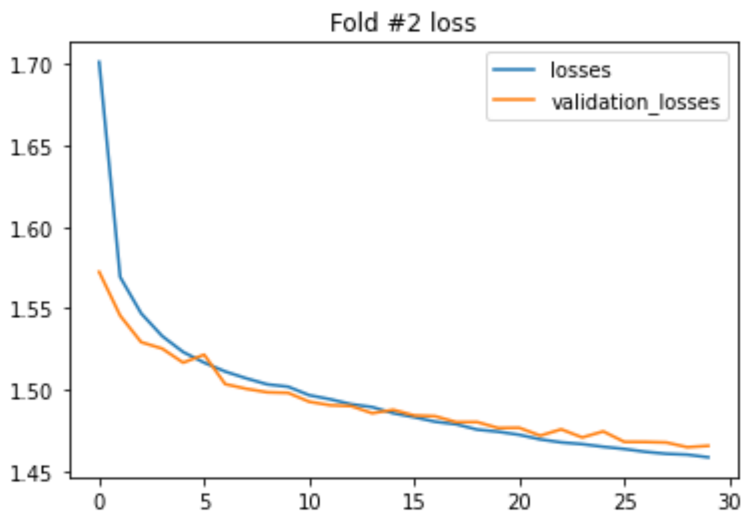
```
model = LeNet()
summary(model,(1,32,32))
Kfold(LeNet,3,train_X_norm,train_y,test_X_norm,test_y,30,lr=1e-3)
    print("Training accuracy:%f+-%f"%(np.average(train_acc_all),np.std(trair
    print("Testing accuracy:%f+-%f"%(np.average(test_acc_all),np.std(test_ac
```

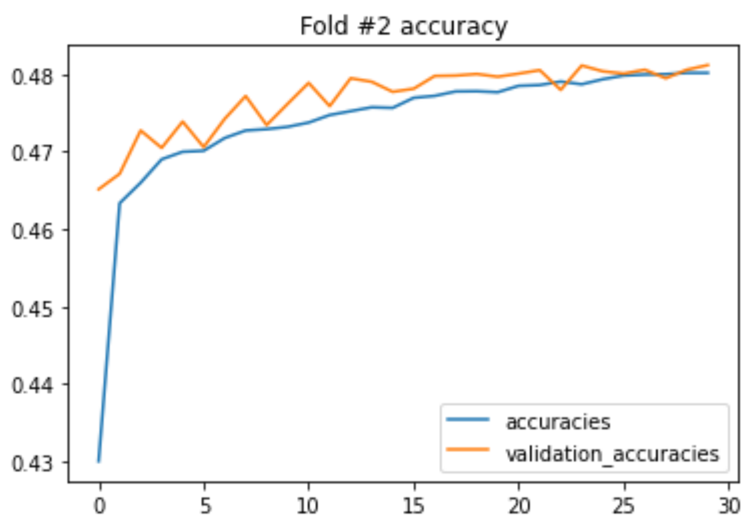
```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
├─Conv2d: 1-1                            [-1, 3, 32, 32]           78
├─ReLU: 1-2                              [-1, 3, 32, 32]           --
├─Linear: 1-3                            [-1, 10]                  30,730
├─ReLU: 1-4                              [-1, 10]                  --
==========================================================================
===============
Total params: 30,808
Trainable params: 30,808
Non-trainable params: 0
Total mult-adds (M): 0.11
==========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.02
Params size (MB): 0.12
Estimated Total Size (MB): 0.14
==========================================================================
===============
Fold #1
  3%|█                                          | 1/30 [00:02<01:19,  2.73
s/it]
Epoch 1/30 - Loss: 1.789 - Acc: 0.353
              Val_loss: 1.538 - Val_acc: 0.461
 37%|████████████████                           | 11/30 [00:28<00:46,  2.47
s/it]
Epoch 11/30 - Loss: 1.429 - Acc: 0.492
              Val_loss: 1.429 - Val_acc: 0.492
 70%|██████████████████████████████             | 21/30 [00:53<00:22,  2.47
s/it]
Epoch 21/30 - Loss: 1.199 - Acc: 0.589
              Val_loss: 1.211 - Val_acc: 0.586

func:'train'  took: 75.4363 sec
Train accuracy: 0.591675
Validation accuracy: 0.5870500000000001
Test accuracy: 0.5901000000000003
Fold #2
  3%|█                                          | 1/30 [00:02<01:14,  2.57
s/it]
Epoch 1/30 - Loss: 1.701 - Acc: 0.430
              Val_loss: 1.572 - Val_acc: 0.465
 37%|████████████████                           | 11/30 [00:29<00:52,  2.75
s/it]
Epoch 11/30 - Loss: 1.497 - Acc: 0.474
              Val_loss: 1.493 - Val_acc: 0.479
 70%|██████████████████████████████             | 21/30 [00:57<00:24,  2.75
s/it]
Epoch 21/30 - Loss: 1.472 - Acc: 0.479
              Val_loss: 1.477 - Val_acc: 0.480
```

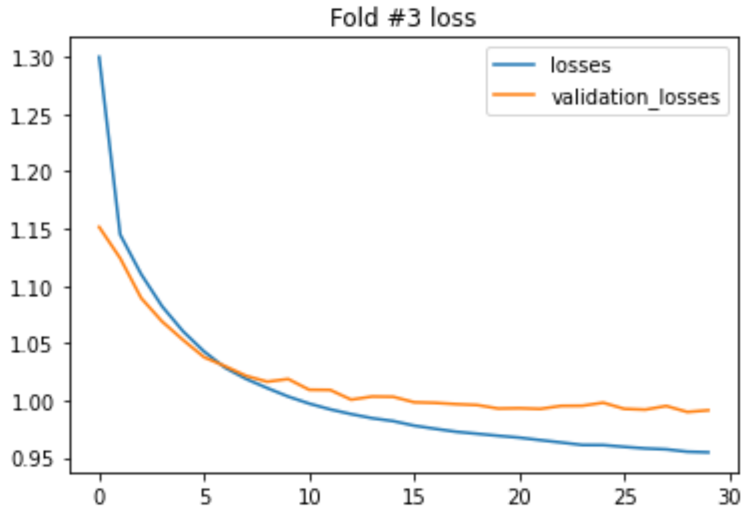
```
func:'train'  took: 82.2531 sec
Train accuracy: 0.4801999999999999
Validation accuracy: 0.4812000000000002
Test accuracy: 0.4777000000000001
Fold #3
  3%|█                                        | 1/30 [00:02<01:23,  2.86
s/it]
Epoch 1/30 - Loss: 1.300 - Acc: 0.549
              Val_loss: 1.151 - Val_acc: 0.567
 37%|████████████████                         | 11/30 [00:31<00:55,  2.92
s/it]
Epoch 11/30 - Loss: 0.997 - Acc: 0.594
              Val_loss: 1.009 - Val_acc: 0.592
 70%|██████████████████████████████           | 21/30 [01:04<00:30,  3.36
s/it]
Epoch 21/30 - Loss: 0.968 - Acc: 0.598
              Val_loss: 0.993 - Val_acc: 0.593

func:'train'  took: 93.8838 sec
Train accuracy: 0.6001500000000003
Validation accuracy: 0.5927500000000002
Test accuracy: 0.5986999999999999
Final results:
Training accuracy:0.557342+-0.054657
Testing accuracy:0.555500+-0.055125
```

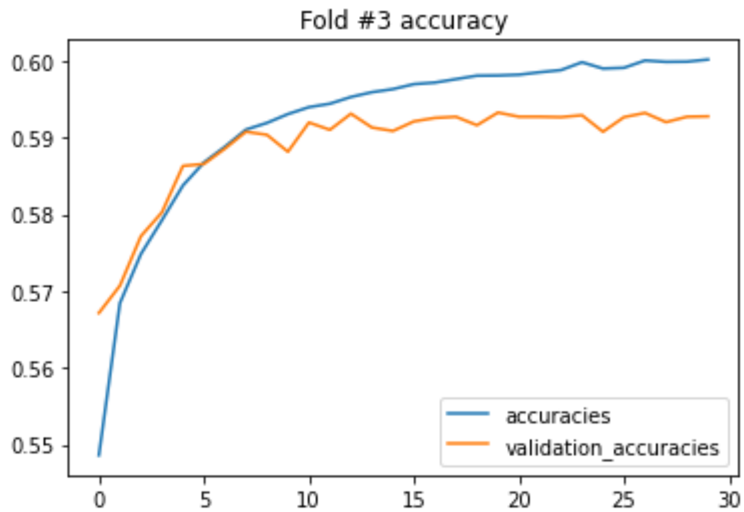



Fold #1 accuracy
accuracies
validation_accuracies
0.60
0.55
0.50
0.45
0.40
0.35
0
5
10
15
20
25
30


Fold #2 loss
losses
validation_losses
1.70
1.65
1.60
1.55
1.50
1.45
0
5
10
15
20
25
30


Fold #2 accuracy
accuracies
validation_accuracies
0.48
0.47
0.46
0.45
0.44
0.43
0
5
10
15
20
25
30


Fold #3 loss
losses
validation_losses
1.30
1.25
1.20
1.15
1.10
1.05
1.00
0.95
0
5
10
15
20
25
30


Fold #3 accuracy
accuracies
validation_accuracies
0.60
0.59
0.58
0.57
0.56
0.55
0
5
10
15
20
25
30

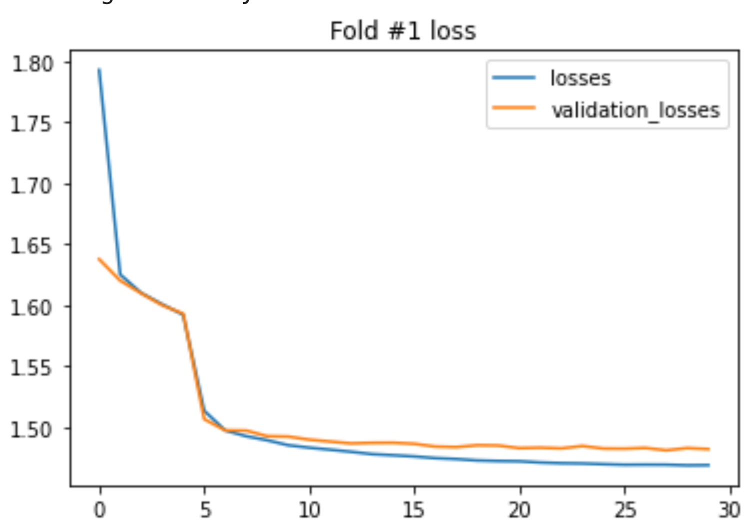
1(d) (6pt) Now build a deeper (more layers) architecture with two layers each composed of one convolution and one pooling layer. Flatten the resulting feature maps and use two fully connected (FC) layers. Use conv/pooling layers that with kernel, stride and padding size of your choice. Use ReLU as your activation function. 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). Use mini-batches of data and converge your training to where the loss function is minimal, and choose some regularization techniques. Using 3-fold cross-validation report and your average test accuracy. You should aim for getting test accuracy above 98.5%.

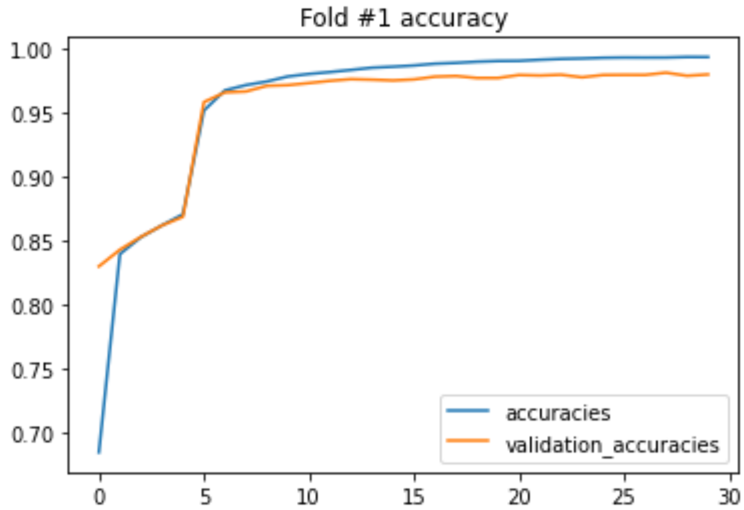
In [16]:
```
from torch import nn
import torch
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.ModuleList([nn.Conv2d(1,6,kernel_size=2),
                                   nn.Conv2d(6,25,kernel_size=2)])
        self.pooling=nn.AvgPool2d(kernel_size=2)
        self.fc = nn.ModuleList([nn.Linear(25*14*14,84),nn.Linear(84,10)])
        self.activation = nn.ReLU()
        self.dropout= nn.Dropout(p=0.15)


    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
```

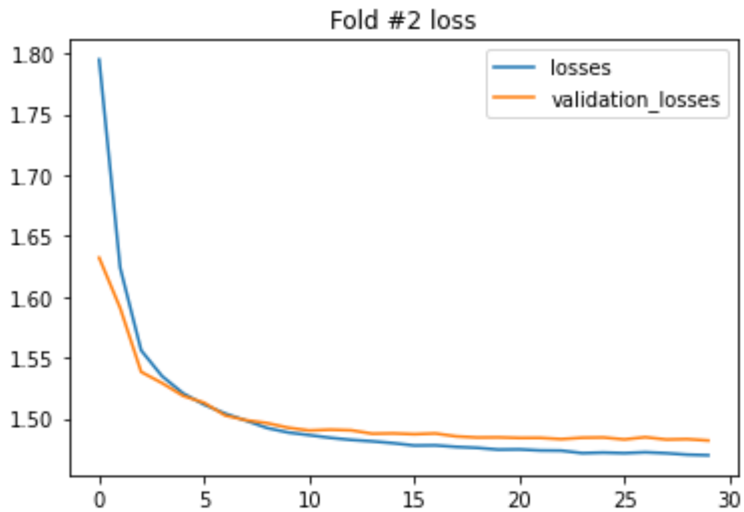
In [17]:
```
model = LeNet()
summary(model,(1,32,32))
Kfold(LeNet,3,train_X_norm,train_y,test_X_norm,test_y,30,lr=1e-3,l2=True)
```

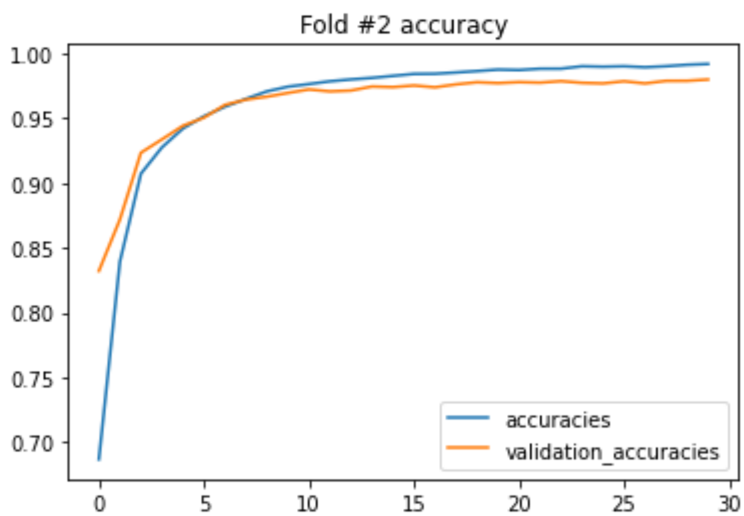
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-1                       [-1, 6, 31, 31]           30
├─ReLU: 1-1                              [-1, 6, 31, 31]           --
├─AvgPool2d: 1-2                         [-1, 6, 15, 15]           --
├─ModuleList: 1                          []                        --
|    └─Conv2d: 2-2                       [-1, 25, 14, 14]          625
├─ReLU: 1-3                              [-1, 25, 14, 14]          --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-3                       [-1, 84]                  411,684
├─ReLU: 1-4                              [-1, 84]                  --
├─ModuleList: 1                          []                        --
|    └─Linear: 2-4                       [-1, 10]                  850
==========================================================================================
Total params: 413,189
Trainable params: 413,189
Non-trainable params: 0
Total mult-adds (M): 0.55
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.08
Params size (MB): 1.58
Estimated Total Size (MB): 1.66
==========================================================================================
Fold #1
  3%|█                                     | 1/30 [00:05<02:27,  5.10 s/it]
Epoch 1/30 - Loss: 1.793 - Acc: 0.685
              Val_loss: 1.638 - Val_acc: 0.830
 37%|████████████████                      | 11/30 [01:06<01:59,  6.30 s/it]
Epoch 11/30 - Loss: 1.483 - Acc: 0.980
              Val_loss: 1.490 - Val_acc: 0.973
 70%|██████████████████████████████        | 21/30 [02:09<00:54,  6.09 s/it]
Epoch 21/30 - Loss: 1.472 - Acc: 0.990
              Val_loss: 1.483 - Val_acc: 0.979

func:'train'  took: 184.1152 sec
Train accuracy: 0.9929749999999995
Validation accuracy: 0.9793999999999979
Test accuracy: 0.9797000000000009
Fold #2
  3%|█                                     | 1/30 [00:06<03:00,  6.21 s/it]
```

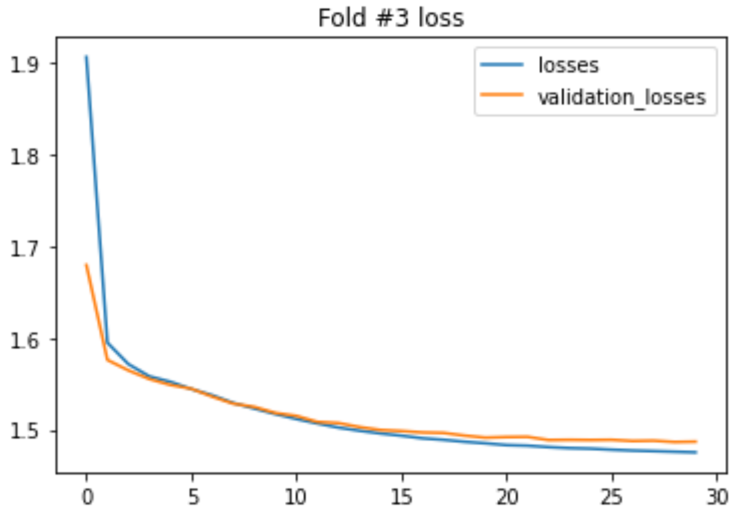
```
Epoch 1/30 - Loss: 1.795 - Acc: 0.687
              Val_loss: 1.632 - Val_acc: 0.832
 37%|████████████████                              | 11/30 [01:03<01:44,  5.49
s/it]
Epoch 11/30 - Loss: 1.486 - Acc: 0.976
              Val_loss: 1.490 - Val_acc: 0.972
 70%|██████████████████████████████                | 21/30 [01:57<00:49,  5.45
s/it]
Epoch 21/30 - Loss: 1.475 - Acc: 0.987
              Val_loss: 1.484 - Val_acc: 0.978

func:'train'  took: 170.5429 sec
Train accuracy: 0.9916249999999996
Validation accuracy: 0.9796499999999998
Test accuracy: 0.9812000000000007
Fold #3
  3%|█                                              | 1/30 [00:05<02:37,  5.43
s/it]
Epoch 1/30 - Loss: 1.907 - Acc: 0.575
              Val_loss: 1.680 - Val_acc: 0.804
 37%|████████████████                              | 11/30 [01:03<01:49,  5.74
s/it]
Epoch 11/30 - Loss: 1.512 - Acc: 0.953
              Val_loss: 1.515 - Val_acc: 0.949
 70%|██████████████████████████████                | 21/30 [02:00<00:51,  5.71
s/it]
Epoch 21/30 - Loss: 1.483 - Acc: 0.980
              Val_loss: 1.492 - Val_acc: 0.971

func:'train'  took: 172.1714 sec
Train accuracy: 0.9870000000000009
Validation accuracy: 0.9747999999999984
Test accuracy: 0.9754000000000006
Final results:
Training accuracy:0.990533+-0.002559
Testing accuracy:0.978767+-0.002458
```

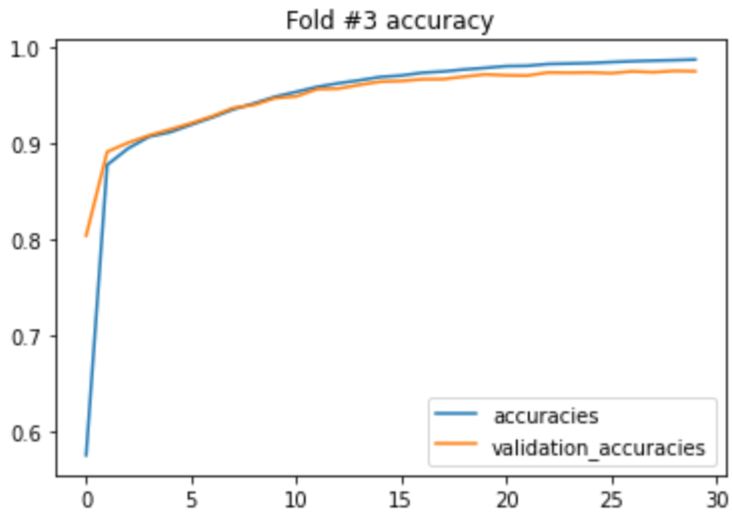



Fold #1 accuracy
accuracies
validation_accuracies


Fold #2 loss
losses
validation_losses


Fold #2 accuracy
accuracies
validation_accuracies





Tried adding more layers and channels but found this to yield the highest test accuracy though it is not 98.5%. I tried adjusting the dropout and input/output channels plus the kernel sizes to no avail.

```
In [ ]:
```