In [1]:

```python
import torch
import torch.nn as nn
import torchvision
from torchvision import datasets, transforms, models
import numpy as np
import matplotlib.pyplot as plt
import random

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [33]:

```python
#Exercise 1
transofms=transforms.Compose([transforms.ToTensor(),\
                        transforms.Normalize((0.4376821, 0.4437697, 0.47280442),
                                             (0.19803012, 0.20101562, 0.19703614))])
train_set = torchvision.datasets.SVHN(root='./data',download=True, transform=transofms
train_size = 8*len(train_set)//10
validation_size = len(train_set) - train_size
train_set, validation_set=torch.utils.data.random_split(train_set,[train_size, validat
dataset_sizes={'train':train_size,'val':validation_size}
test_set = torchvision.datasets.SVHN(root='./data',split='test',download=True, transfo
datasets={'train':train_set,'val':validation_set,'test':test_set}
dataloaders = {x: torch.utils.data.DataLoader(datasets[x], batch_size=128,
                                              shuffle=True, num_workers=2)
              for x in ['train', 'val','test']}
```

```
Using downloaded and verified file: ./data/train_32x32.mat
Using downloaded and verified file: ./data/test_32x32.mat
```

In [ ]:

```python
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[x for x in classes])
```

In [3]:

```python
#Exercise 2
input_size=32*32*3
output_size=10
def Fcn(l1_size=128,l2_size=64):
  return nn.Sequential(nn.Linear(input_size, l1_size),
                       nn.Linear(l1_size, l2_size),
                       nn.ReLU(),
                       nn.Linear(l2_size, output_size),
                       nn.ReLU())

def Cnn(ker_size=3):
  con_out_size=((32-ker_size+3)//2)
  con_out_size=((con_out_size-ker_size+3)//2)
  con_out_size=con_out_size**2
  return nn.Sequential(nn.Conv2d(in_channels=3,out_channels=10,
                                 kernel_size=ker_size,padding=1,stride=1),
                       nn.ReLU(),
                       nn.MaxPool2d(kernel_size=2,stride=2,padding=0),
                       nn.Conv2d(in_channels=10,out_channels=20,
                                kernel_size=ker_size,padding=1,stride=1),
                       nn.ReLU(),
                       nn.MaxPool2d(kernel_size=2,stride=2,padding=0),
                       nn.Flatten(),
                       nn.Linear(con_out_size*20, 64),
                       nn.ReLU(),
                       nn.Linear(64,output_size))
```

In [10]:

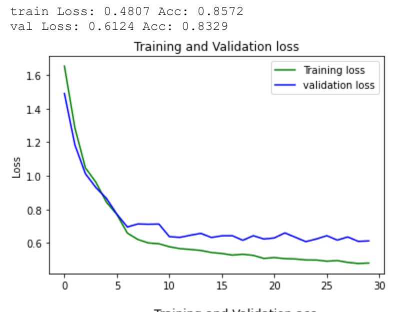
```python
#Exercise 3/4
import copy
def train_model(model, criterion, optimizer,fc=True,num_epochs=30,debug=0):
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    model=model.to(device)
    epoch_loss = {'train':np.zeros(num_epochs),'val':np.zeros(num_epochs)}
    epoch_acc = {'train':np.zeros(num_epochs),'val':np.zeros(num_epochs)}
    for epoch in range(num_epochs):
        if debug==1 or epoch == num_epochs-1:
          print('Epoch {}/{}'.format(epoch, num_epochs - 1))
          print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                if fc:
                  inputs = inputs.view(inputs.shape[0], -1)
                inputs = inputs.to(device)
                labels = labels.to(device)
                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    preds = torch.argmax(outputs,dim=1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss[phase][epoch] = running_loss / dataset_sizes[phase]
            epoch_acc[phase][epoch] = running_corrects.double() / dataset_sizes[phase]


            if phase == 'train' and (debug==1 or epoch == num_epochs-1):
              print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                  phase, epoch_loss[phase][epoch], epoch_acc[phase][epoch]))
            elif phase == 'val' and (debug==1 or epoch == num_epochs-1):
              print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                  phase, epoch_loss[phase][epoch], epoch_acc[phase][epoch]))

    plt.plot(range(num_epochs), epoch_loss['train'], 'g', label='Training loss')
    plt.plot(range(num_epochs), epoch_loss['val'], 'b', label='validation loss')
    plt.title('Training and Validation loss')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
    plt.clf()
    plt.plot(range(num_epochs), epoch_acc['train'], 'r', label='Training acc')
    plt.plot(range(num_epochs), epoch_acc['val'], 'y', label='validation acc')
    plt.title('Training and Validation acc')
    plt.xlabel('Epochs')
    plt.ylabel('acc')
    plt.legend()
    plt.tight_layout()
    plt.show()
    plt.clf()
    return model
```

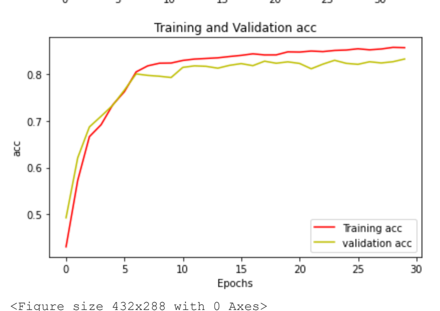
Exercises 3+4

In [11]:

```python
fc128=Fcn()
fc128=train_model(fc128,nn.CrossEntropyLoss(),
                      torch.optim.Adam(fc128.parameters(),lr=0.001),fc=True)
```

```
Epoch 29/29
----------
train Loss: 0.4807 Acc: 0.8572
val Loss: 0.6124 Acc: 0.8329
```






```
<Figure size 432x288 with 0 Axes>
```

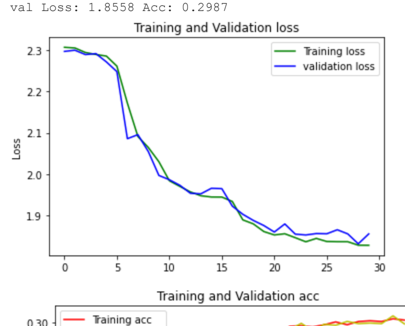
In [12]:

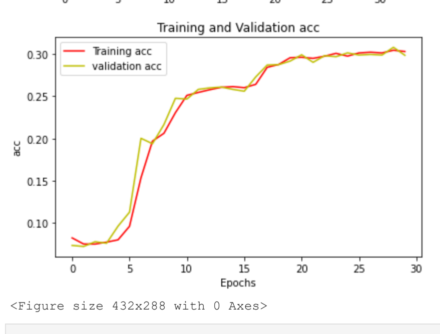
```python
PATH = './SVHN_Fcn_128_64.pth'
torch.save(fc128.state_dict(), PATH)
```

In [13]:

```python
fc512=Fcn(512,256)
fc512=train_model(fc512,nn.CrossEntropyLoss(),
                      torch.optim.Adam(fc512.parameters(),lr=0.001),fc=True)
```

```
Epoch 29/29
----------
train Loss: 1.8282 Acc: 0.3032
val Loss: 1.8558 Acc: 0.2987
```






```
<Figure size 432x288 with 0 Axes>
```

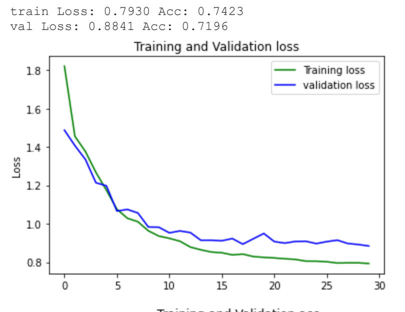
In [14]:

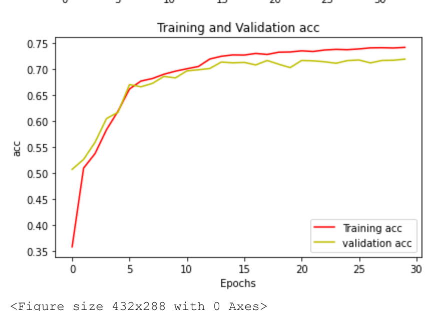
```python
PATH = './SVHN_Fcn_512_256.pth'
torch.save(fc512.state_dict(), PATH)
```

In [27]:

```python
fc64=Fcn(64,32)
fc64=train_model(fc64,nn.CrossEntropyLoss(),
                      torch.optim.Adam(fc64.parameters(),lr=0.001),fc=True,)
```

```
Epoch 29/29
----------
train Loss: 0.7930 Acc: 0.7423
val Loss: 0.8841 Acc: 0.7196
```






```
<Figure size 432x288 with 0 Axes>
```

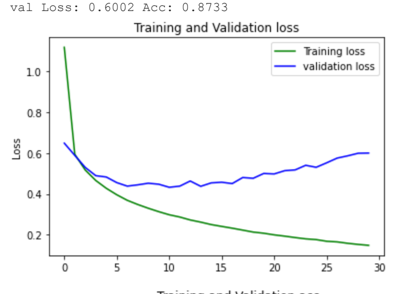
In [28]:

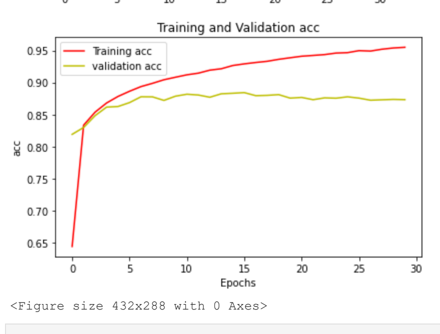
```python
PATH = './SVHN_Fcn_64_32.pth'
torch.save(fc64.state_dict(), PATH)
```

In [17]:

```python
cnn3=Cnn()
cnn3=train_model(cnn3,nn.CrossEntropyLoss(),
                      torch.optim.Adam(cnn3.parameters(),lr=0.001),fc=False)
```

```
Epoch 29/29
----------
train Loss: 0.1475 Acc: 0.9549
val Loss: 0.6002 Acc: 0.8733
```






```
<Figure size 432x288 with 0 Axes>
```

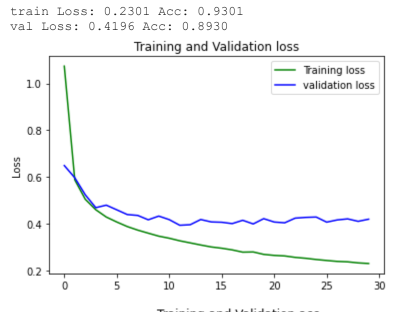
In [18]:

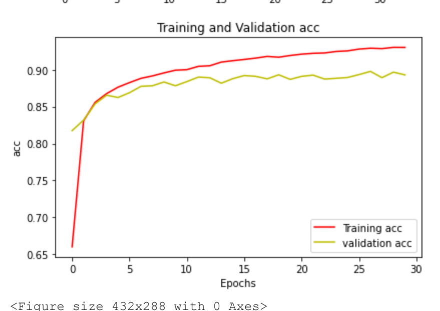
```python
PATH = './SVHN_Cnn_3.pth'
torch.save(cnn3.state_dict(), PATH)
```

In [19]:

```python
cnn10=Cnn(ker_size=10)
cnn10=train_model(cnn10,nn.CrossEntropyLoss(),
                      torch.optim.Adam(cnn10.parameters(),lr=0.001),fc=False)
```

```
Epoch 29/29
----------
train Loss: 0.2301 Acc: 0.9301
val Loss: 0.4196 Acc: 0.8930
```






```
<Figure size 432x288 with 0 Axes>
```

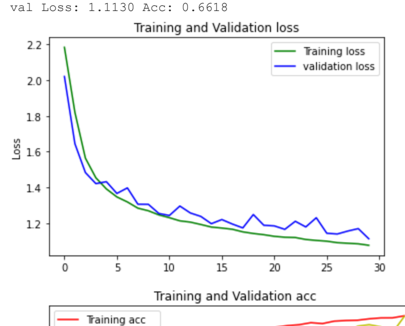
In [20]:

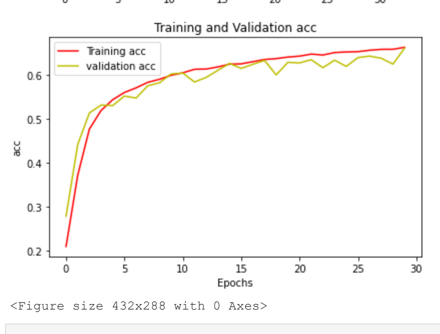
```python
PATH = './SVHN_Cnn_10.pth'
torch.save(cnn10.state_dict(), PATH)
```

In [21]:

```python
cnn1=Cnn(ker_size=1)
cnn1=train_model(cnn1,nn.CrossEntropyLoss(),
                      torch.optim.Adam(cnn1.parameters(),lr=0.001),fc=False)
```

```
Epoch 29/29
----------
train Loss: 1.0765 Acc: 0.6635
val Loss: 1.1130 Acc: 0.6618
```






```
<Figure size 432x288 with 0 Axes>
```

In [22]:

```python
PATH = './SVHN_Cnn_1.pth'
torch.save(cnn1.state_dict(), PATH)
```

problem 1:

the cnn with kernel size of 10 preforms the best with acc of 0.8930 and loss of 0.4196.

from compering the losses on the training and validation sets we can the network overfits as the is a differnce of almost 0.15 and

In [23]:

```python
PATH = './SVHN_Cnn_10.pth'
cnn10.load_state_dict(torch.load(PATH))
```

Out[23]:

```
<All keys matched successfully>
```

In [24]:

```python
#Exercise 5
cnn10.eval()
correct_count, all_count = 0, 0
with torch.no_grad():
  for inputs,labels in dataloaders['test']:
      inputs = inputs.to(device)
      labels = labels.to(device)
      outputs = cnn10(inputs)
      _, preds = torch.max(outputs, 1)
      for i in range(len(labels)):
        pred_label = preds.cpu().data[i]
        true_label = labels.cpu().data[i]
        if(true_label == pred_label):
          correct_count += 1
      all_count += (len(labels))

print("Number Of Images Tested =", all_count)
print("\nModel Accuracy =", (correct_count/all_count))
```

```
Number Of Images Tested = 26032

Model Accuracy = 0.8894821757836509
```

problem 2:

it is not guaranteed that the best model on the validation set will give the best results on the test set.

the assumption we make using the validation to choose the hyper parameters is that the data in both the validation and the test sets has the same distrubtion.

In [25]:

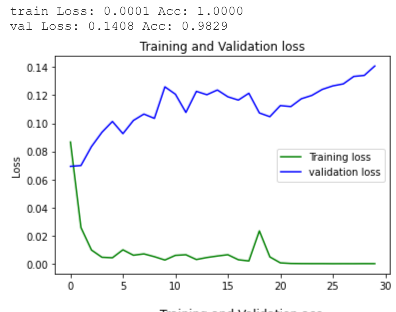
```python
#Exercise 6
model_conv = torchvision.models.resnet18(pretrained=True)

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 10)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv =  torch.optim.Adam(model_conv.parameters(),lr=0.001)
```

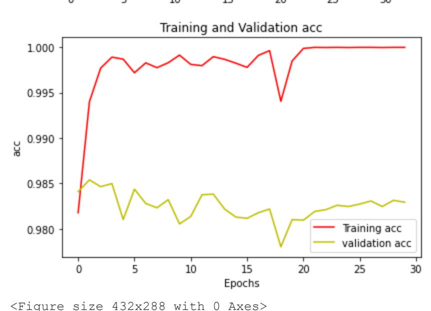
```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.cac
he/torch/hub/checkpoints/resnet18-5c106cde.pth
```

In [34]:

```python
model_conv = train_model(model_conv, criterion, optimizer_conv,fc=False)
```

```
Epoch 29/29
----------
train Loss: 0.0001 Acc: 1.0000
val Loss: 0.1408 Acc: 0.9829
```






```
<Figure size 432x288 with 0 Axes>
```

problem 3

i chose Finetuning CNN as my transefer learning method because i think that the majority of the learning in the problem lays in recodnising lines and coners hence the first few layers of the res-net should do the majority of the work. Also there is a major differnce between the images in imgnet and our dataset because of that i belive that CNN as a fixed feature extractor wouldnt preform as good.

problem 4

the results of the new network are better then the results of all the CNN an FC networks i've implemented. those are the results on the validation set

1. best CNN - Loss: 0.4196 Acc: 0.8930
2. best FC - Loss: 0.6124 Acc: 0.8329
3. new net - Loss: 0.1408 Acc: 0.9829

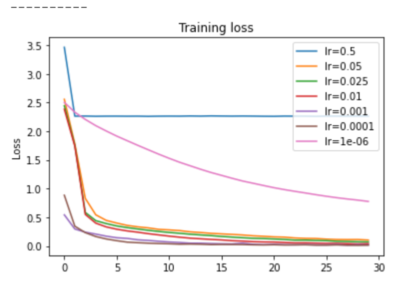
i belive the depth of the network along side with residual block allow the res-net to learn better without vanishing gradient problem therefor it can solve the problem better

problem 5

In [ ]:

```python
#Exercise 6
for lr in [0.5,0.05,0.025,0.01,0.001,0.0001,0.000001] :
  print('lr= {}'.format(lr))
  print('-' * 10)
  model_conv = torchvision.models.resnet18(pretrained=True)

  # Parameters of newly constructed modules have requires_grad=True by default
  num_ftrs = model_conv.fc.in_features
  model_conv.fc = nn.Linear(num_ftrs, 10)

  model_conv = model_conv.to(device)

  criterion = nn.CrossEntropyLoss()

  optimizer_conv =  torch.optim.Adam(model_conv.parameters(),lr=lr)
  epoch_loss = np.zeros(30)
  for epoch in range(30):
        model_conv.train()  # Set model to training mode
        running_loss = 0.0
        running_corrects = 0


        # Iterate over data.
        for inputs, labels in dataloaders['train']:
            inputs = inputs.to(device)
            labels = labels.to(device)
            # zero the parameter gradients
            optimizer_conv.zero_grad()

            # forward
            # track history if only in train
            with torch.set_grad_enabled(True):
                outputs = model_conv(inputs)
                preds = torch.argmax(outputs,dim=1)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer_conv.step()

            # statistics
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss[epoch] = running_loss / dataset_sizes['train']


  plt.plot(range(30), epoch_loss, label=('lr='+str(lr)))


plt.title('Training loss')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
lr= 0.5
----------
lr= 0.05
----------
lr= 0.025
----------
lr= 0.01
----------
lr= 0.001
----------
lr= 0.0001
----------
lr= 1e-06
----------
```



As we can see in the graph, having a very high learning rate leads to the network not learning at all as the steps in the gradient diraction are too big, so the networkj doesnt converage into any kind of minimum. when the leaning rate is to low the steps in the diraction of the gradient are too small so there is a mall improvent but the chances of converging into local minmum increase. for the rates in between the learning rate changes the speed of converages but the differnces are that big meaning the steps are of a good size