COURSE NAME : Computer Vision

Image Classification on CIFAR-100
Assignment 4

Pulkit Gera - 20171035

Introduction

Image classification is the problem of classifying various images into different classes. Here we are not concerned with putting them into bounding boxes. Some things to note:

We use Pytorch as a coding framework. All experiments are run for 50 epochs on Google Collab. We use Adam Optimizer with LR (1e-3) with Cross Entropy Loss unless otherwise specified. The Images have been normalized with : `mean=[0.507, 0.487, 0.441], std=[0.267, 0.256, 0.276]`

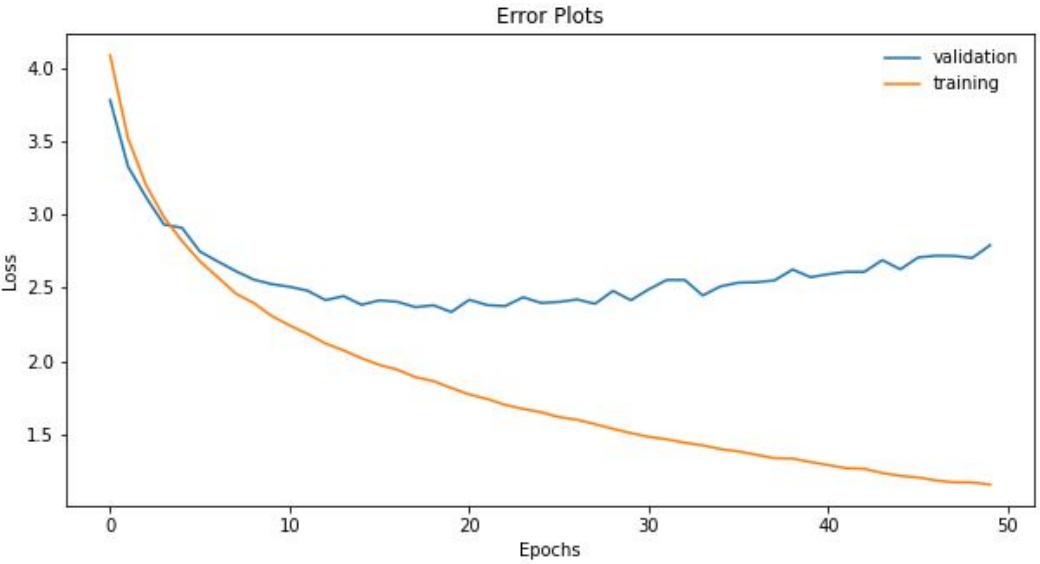
We make 80-20 train val split.

Base Architecture

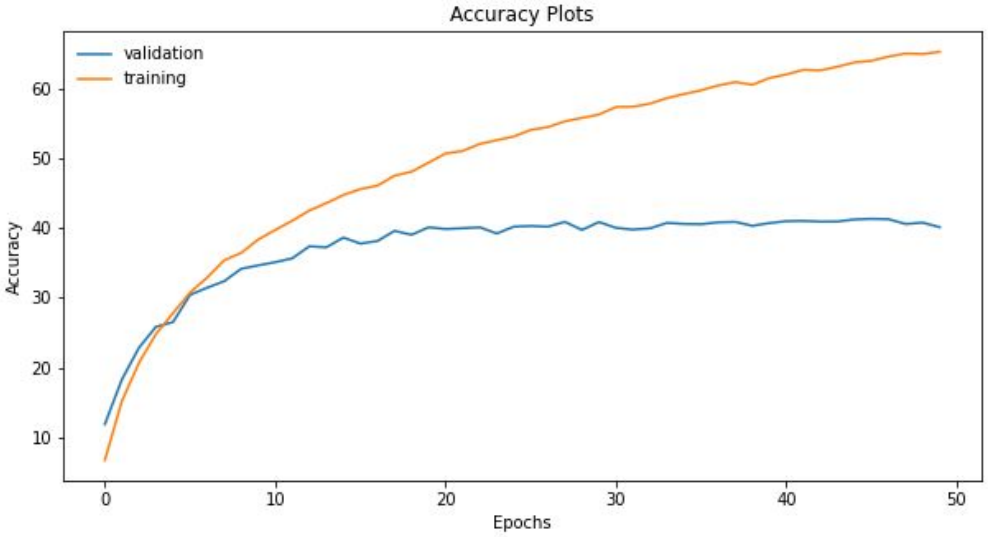
The network consists of 3 convolution layers after which there is ReLU activation and Maxpool. The learnable parameters are only in Convolution Layers and Fully Connected Layers.

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.ReLU(inplace=True),
                               nn.MaxPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.ReLU(inplace=True),
                               nn.MaxPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.ReLU(inplace=True),
                               nn.MaxPool2d(2,2))
    self.fc1 = nn.Sequential(nn.Linear(2048,1024),
                             nn.ReLU(inplace=True))
    self.fc2 = nn.Sequential(nn.Linear(1024,512),
                             nn.ReLU(inplace=True))
    self.fc3 = nn.Linear(512,100)

  def forward(self,x):
    # print(x.shape)
    x = self.conv1(x)
    # print(x.shape)
    x = self.conv2(x)
    # print(x.shape)
    x = self.conv3(x)
    # print(x.shape)
    fc = x.view(x.size(0),-1)
    # print(fc.shape)
    fc = self.fc1(fc)
    fc = self.fc2(fc)
    out = self.fc3(fc)
    return out
```

Results



As we can see, the validation stops after a certain point and the training loss keeps on decreasing. We perform early stopping to ensure we have the model with lowest validation error.



Training loss goes as high as to 65% whereas validation loss remains in the region of 40-42%.

Test Accuracy on the network = **43.68%**

Base Architecture with Batch Norm

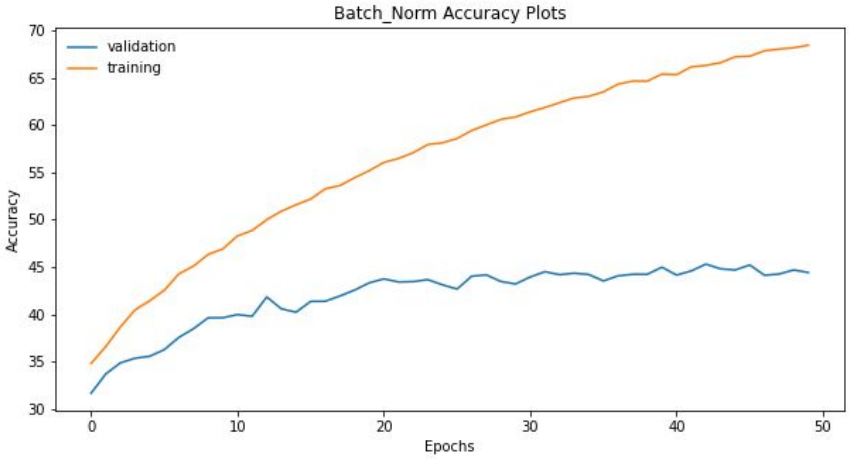
We now add batch norm layers after all the convolution layers. The idea is that if we normalize the inputs in order to improve training why cant we do that while giving input to the hidden layers. It stabilizes training and acts as a sort of a regularizer.

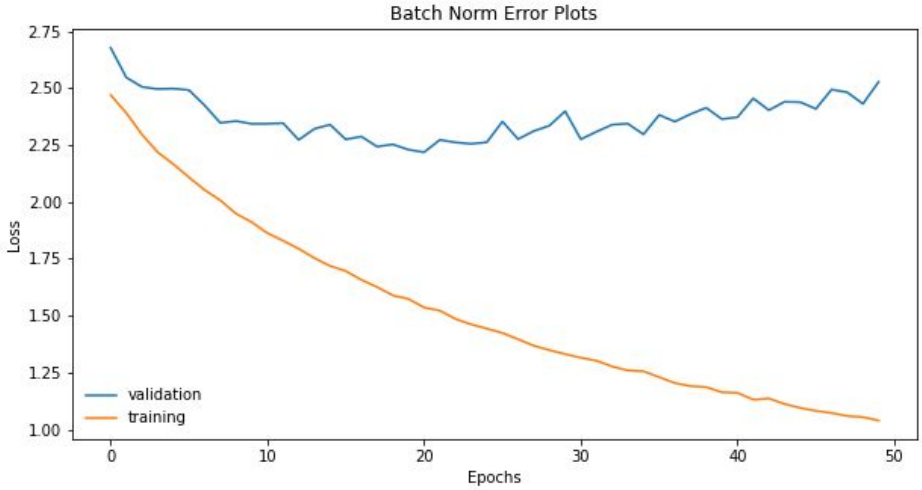
Batch normalization reduces the amount by what the hidden unit values shift around.

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.BatchNorm2d(32),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.BatchNorm2d(64),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.BatchNorm2d(128),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.fc1 = nn.Sequential(nn.Linear(2048,1024),
                               nn.ReLU(inplace=True))
    self.fc2 = nn.Sequential(nn.Linear(1024,512),
                               nn.ReLU(inplace=True))
    self.fc3 = nn.Linear(512,100)

  def forward(self,x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    fc = x.view(x.size(0),-1)
    fc = self.fc1(fc)
    fc = self.fc2(fc)
    out = self.fc3(fc)
    return out
```

Results

The test accuracy we get is **46.76%.**





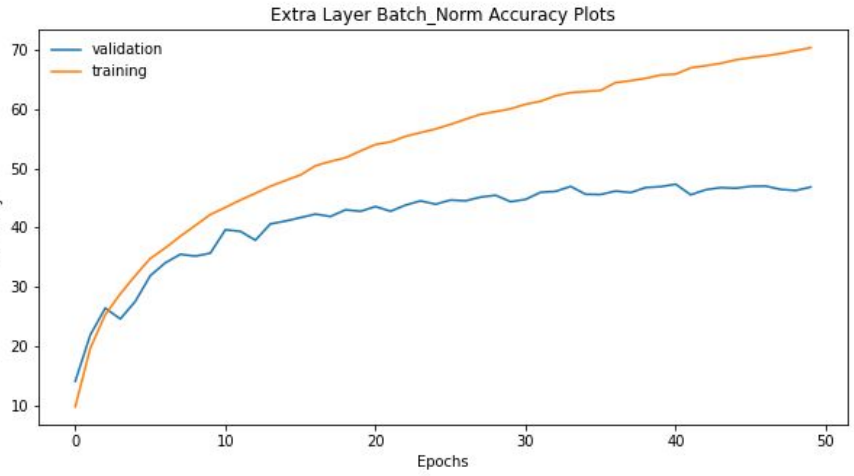
Base + Conv+ BatchNorm (New Base)

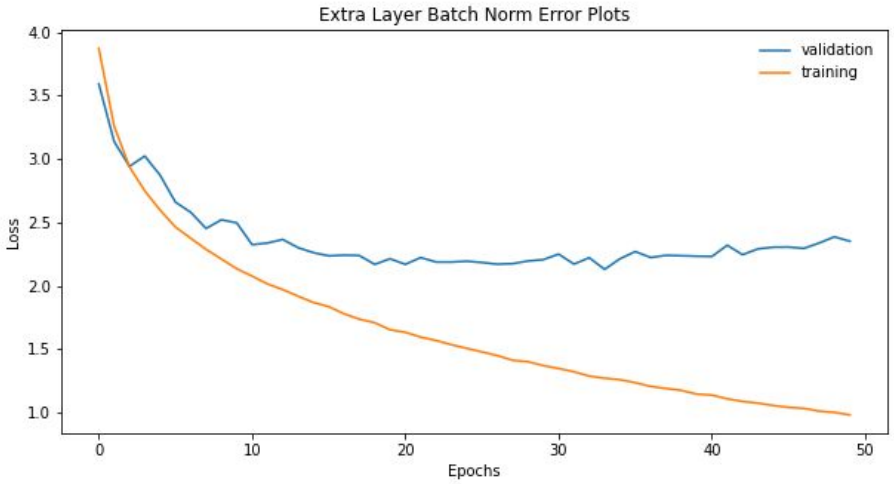
We add another convolution layer to compare our performances. So far it gives the best performance. However deeper networks are not necessarily better which will be shown in the next experiment. We will also use this as our **New Base.**

Results

The test accuracy we get is **49.50%.**

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.BatchNorm2d(32),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.BatchNorm2d(64),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.BatchNorm2d(128),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv4 = nn.Sequential(nn.Conv2d(128,256,3,padding=1),
                               nn.BatchNorm2d(256),
                                nn.ReLU(inplace=True),
                                 )
    self.fc1 = nn.Sequential(nn.Linear(256*4*4,1024),
                               nn.ReLU(inplace=True))
    self.fc2 = nn.Sequential(nn.Linear(1024,256),
                               nn.ReLU(inplace=True))
    self.fc3 = nn.Linear(256,100)

  def forward(self,x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # print(x.shape)
    fc = x.view(x.size(0),-1)
    # print(fc.shape)
    fc = self.fc1(fc)
    fc = self.fc2(fc)
    out = self.fc3(fc)
    return out
```





New Base+ Linear+Dropout

We add a new Linear Layer in the network (Fully Connected Layer) as well as Dropout.

Dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random.

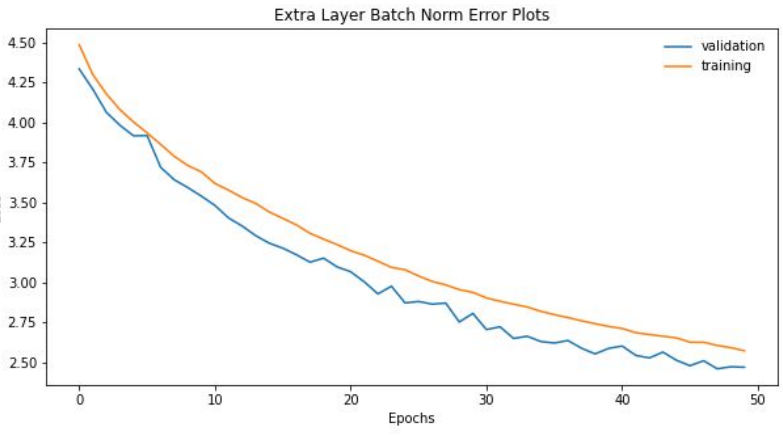
Dropout is useful to prevent overfitting. A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data.

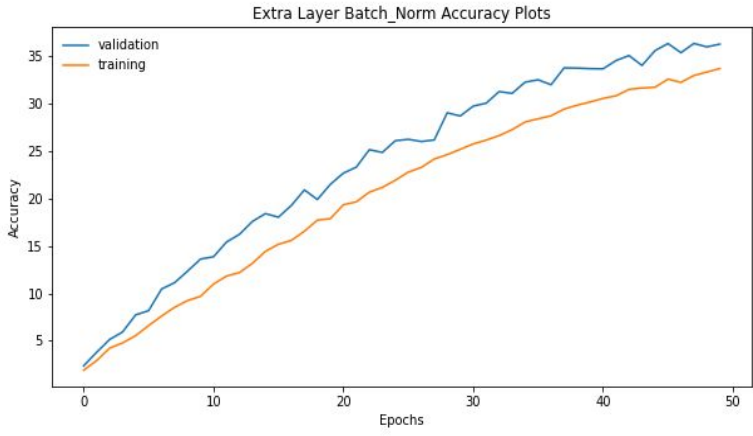
This network doesnt converge in 50 epochs and hence gives a poorer result as compared to others.

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.BatchNorm2d(32),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.BatchNorm2d(64),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.BatchNorm2d(128),
                                nn.ReLU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv4 = nn.Sequential(nn.Conv2d(128,256,3,padding=1),
                               nn.BatchNorm2d(256),
                                nn.ReLU(inplace=True),
                                 )

    self.fc1 = nn.Sequential(nn.Linear(256*4*4,1024),
                               nn.ReLU(inplace=True),
                              nn.Dropout())
    self.fc2 = nn.Sequential(nn.Linear(1024,256),
                               nn.ReLU(inplace=True),
                              nn.Dropout())
    self.fc3 = nn.Sequential(nn.Linear(256,512),
                               nn.ReLU(inplace=True),
                             nn.Dropout())
    self.fc4 = nn.Sequential(nn.Linear(512,128),
                               nn.ReLU(inplace=True),
                             nn.Dropout())
    self.fc5 = nn.Linear(128,100)

  def forward(self,x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # print(x.shape)
```

Results

The test accuracy we get is **38.40%.**





New Base with ELU

Note(we **include dropout** in New Base).

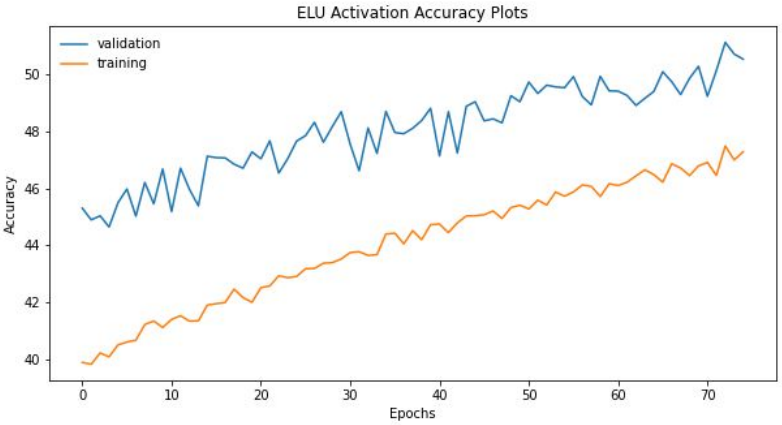
Exponential Linear Unit or its widely known name ELU is a function that tends to converge cost to zero faster and produce more accurate results. ELU is very similar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output is equal to -α whereas RELU sharply smoothes.

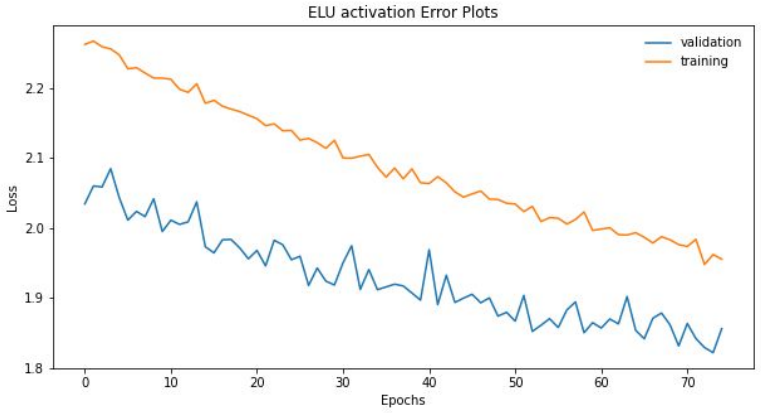
We run this for 75 epochs due to the weird behaviour that we observe,valid_loss<train_loss

Results

The test accuracy we get is **56.87%**

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.BatchNorm2d(32),
                                nn.ELU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.BatchNorm2d(64),
                                nn.ELU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.BatchNorm2d(128),
                                nn.ELU(inplace=True),
                                 nn.MaxPool2d(2,2))
    self.conv4 = nn.Sequential(nn.Conv2d(128,256,3,padding=1),
                               nn.BatchNorm2d(256),
                                nn.ELU(inplace=True),
                                 )

    self.fc1 = nn.Sequential(nn.Linear(256*4*4,1024),
                                nn.ELU(inplace=True),
                               nn.Dropout())
    self.fc2 = nn.Sequential(nn.Linear(1024,256),
                                nn.ELU(inplace=True),
                               nn.Dropout())
    self.fc3 = nn.Linear(256,100)

  def forward(self,x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # print(x.shape)
    fc = x.view(x.size(0),-1)
    # print(fc.shape)
    fc = self.fc1(fc)
    fc = self.fc2(fc)
    out = self.fc3(fc)
    return out
```





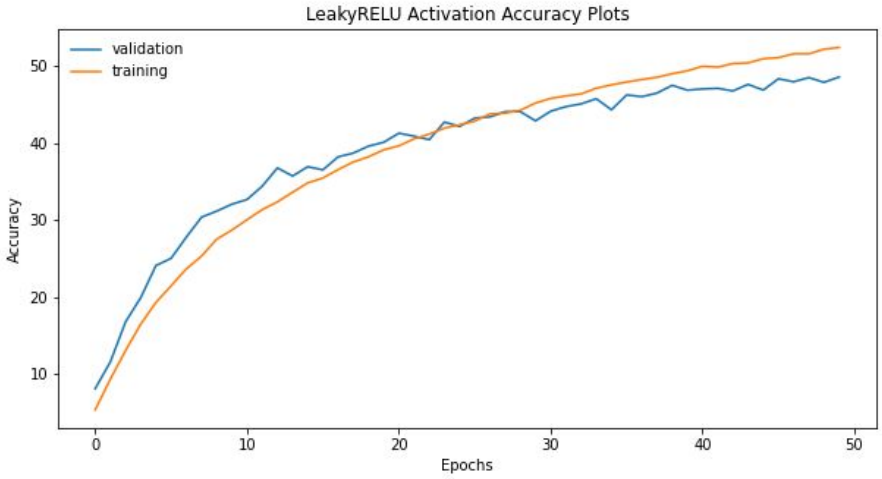
New Base with LeakyReLU

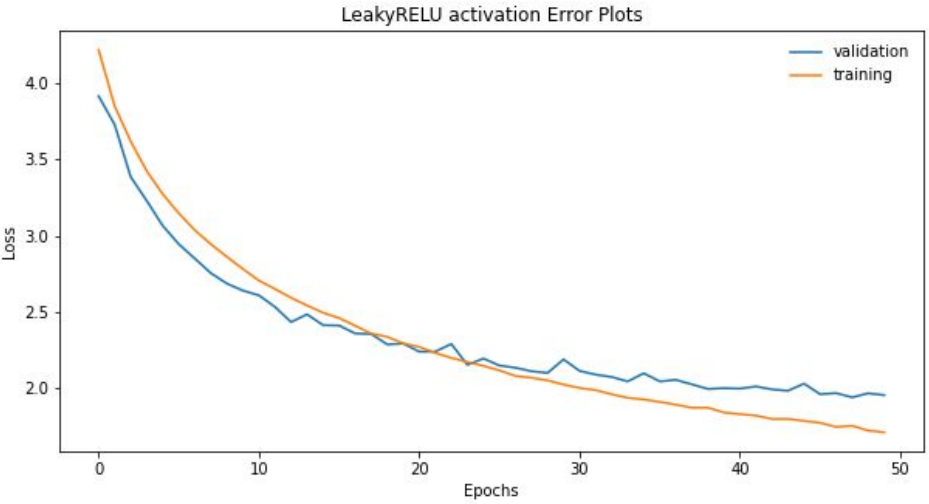
LeakyRelu is a variant of ReLU. Instead of being 0 when $z<0$, a leaky ReLU allows a small, non-zero, constant gradient α . Leaky ReLUs are one attempt to fix the “dying ReLU” problem by having a small negative slope (of 0.01, or so).

Results

The test accuracy we get is **51.23%**

```
def __init__(self):
  super(Model,self).__init__()
  self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                             nn.BatchNorm2d(32),
                              nn.LeakyReLU(inplace=True),
                               nn.MaxPool2d(2,2))
  self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                             nn.BatchNorm2d(64),
                              nn.LeakyReLU(inplace=True),
                               nn.MaxPool2d(2,2))
  self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                             nn.BatchNorm2d(128),
                              nn.LeakyReLU(inplace=True),
                               nn.MaxPool2d(2,2))
  self.conv4 = nn.Sequential(nn.Conv2d(128,256,3,padding=1),
                             nn.BatchNorm2d(256),
                              nn.LeakyReLU(inplace=True),
                               )

  self.fc1 = nn.Sequential(nn.Linear(256*4*4,1024),
                             nn.LeakyReLU(inplace=True),
                            nn.Dropout())
  self.fc2 = nn.Sequential(nn.Linear(1024,256),
                             nn.LeakyReLU(inplace=True),
                            nn.Dropout())
  self.fc3 = nn.Linear(256,100)

def forward(self,x):
  x = self.conv1(x)
  x = self.conv2(x)
  x = self.conv3(x)
  x = self.conv4(x)
  # print(x.shape)
  fc = x.view(x.size(0),-1)
  # print(fc.shape)
  fc = self.fc1(fc)
  fc = self.fc2(fc)
  out = self.fc3(fc)
  return out
```





New Base with Average Pool

Max pooling is a downsampling strategy, thereby reducing the dimensionality of input representation and thus allowing assumptions to be made about features contained in the subregions binned.

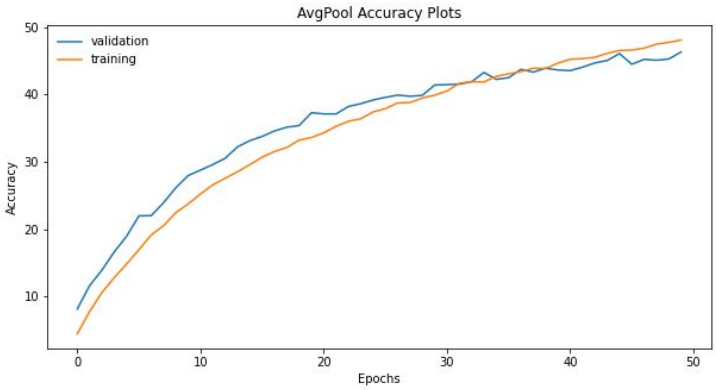
Instead of taking Max while downsampling we can take the Average of the samples which gives us AveragePool.

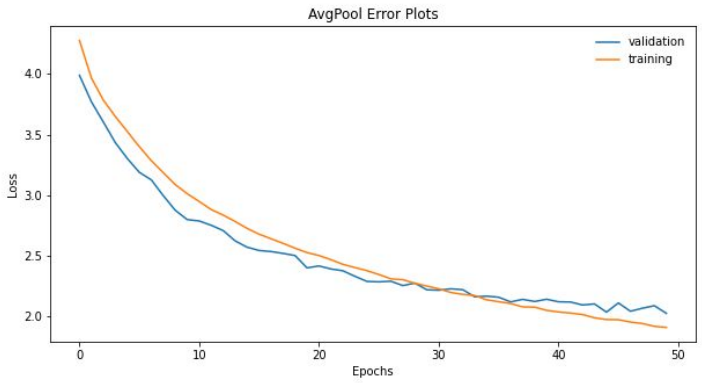
Why do we perform pooling? Answer: To reduce variance, reduce computation complexity (as 2*2 max pooling/average pooling reduces 75% data) and extract low level features from neighbourhood.

Results

The test accuracy we get is **48.44%**

```
class Model(nn.Module):
  def __init__(self):
    super(Model,self).__init__()
    self.conv1 = nn.Sequential(nn.Conv2d(3,32,3,padding=1),
                               nn.BatchNorm2d(32),
                                nn.ReLU(inplace=True),
                                 nn.AvgPool2d(2,2))
    self.conv2 = nn.Sequential(nn.Conv2d(32,64,3,padding=1),
                               nn.BatchNorm2d(64),
                                nn.ReLU(inplace=True),
                                 nn.AvgPool2d(2,2))
    self.conv3 = nn.Sequential(nn.Conv2d(64,128,3,padding=1),
                               nn.BatchNorm2d(128),
                                nn.ReLU(inplace=True),
                                 nn.AvgPool2d(2,2))
    self.conv4 = nn.Sequential(nn.Conv2d(128,256,3,padding=1),
                               nn.BatchNorm2d(256),
                                nn.ReLU(inplace=True),
                                 )

    self.fc1 = nn.Sequential(nn.Linear(256*4*4,1024),
                               nn.ReLU(inplace=True),
                              nn.Dropout())
    self.fc2 = nn.Sequential(nn.Linear(1024,256),
                               nn.ReLU(inplace=True),
                              nn.Dropout())
    self.fc3 = nn.Linear(256,100)

  def forward(self,x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # print(x.shape)
    fc = x.view(x.size(0),-1)
    # print(fc.shape)
    fc = self.fc1(fc)
    fc = self.fc2(fc)
    out = self.fc3(fc)
    return out
```





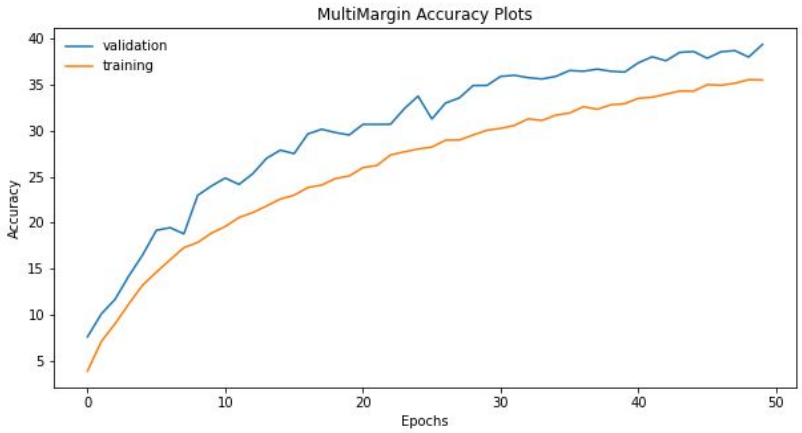
New Base with MultiMargin Loss Function

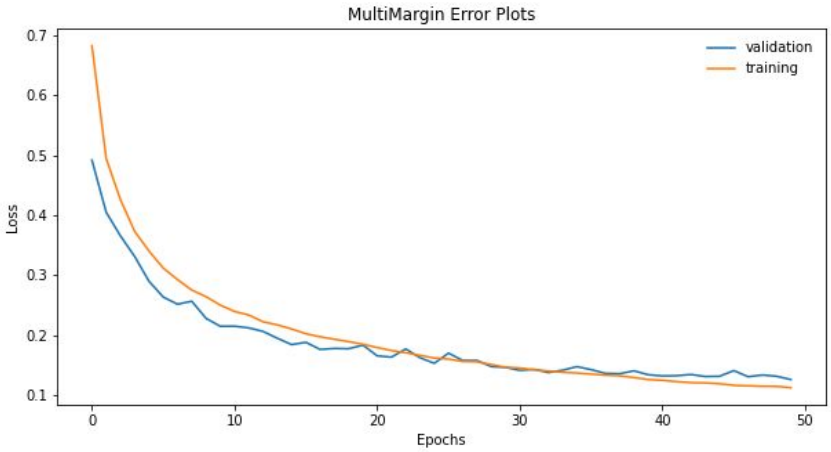
Margin Loss: This name comes from the fact that these losses use a margin to compare samples representations distances. It measures the loss given inputs *x1, x2*, and a label tensor *y* with values (1 or -1). If y == 1 then it assumed the first input should be ranked higher than the second input, and vice-versa for y == -1. Here we use it for multi class. However its not made for classification but rather for ranking

```
error = nn.MultiMarginLoss()
```

Results

The test accuracy we get is **40.44%**





New Base with SGD Optimizer

It takes more steps to converge as compared to Adam Optimizer. Therefore it doesnt converge in 50 steps.

```
optimizer = optim.SGD(model.parameters(),lr=1e-3)
```

Results

The test accuracy we get is **41.44%**

New Base with RMSPROP Optimizer

Its faster per epoch as compared to Adam but doesnt converge in as many steps as Adam.

```
optimizer = optim.RMSprop(model.parameters(),lr=1e-3,weight_decay=5*1e-4)
```

Results

The test accuracy we get is **46.16%**