

CIFAR10 Image Classification using CNN in Pytorch

Assignment 5 of Artificial Intelligence course

Sadaf Sadeghian

810195419

```
import torch
import time
import numpy as np
import matplotlib.pyplot as plt
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data.sampler import SubsetRandomSampler
from torchvision import transforms

def initWeightsToZero(m):
    if type(m) == nn.Linear:
        m.bias.data.fill_(0)
        m.weight.data.fill_(0)

def evaluateNetwork(Net, normalize=False, batchSize=32, zeroWeight=False, learningR

    trans = [
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]
    transform = transforms.Compose(trans if normalize else [trans[0]])

    cifar = datasets.CIFAR10(root="./data", train=True, download=True, transform=tran
    cifarTestSet = datasets.CIFAR10(root="./data", train=False, download=True, transf

    split = int(0.8 * len(cifar))
    indexes = list(range(len(cifar)))
    trainIndexes, validationIndexes = indexes[:split], indexes[split:]

    # sampler objects
    trainSampler = SubsetRandomSampler(trainIndexes)
    validationSampler = SubsetRandomSampler(validationIndexes)

    # iterator objects
    trainLoader = DataLoader(cifar, batch_size=batchSize, sampler=trainSampler)
    validationLoader = DataLoader(cifar, batch_size=batchSize, sampler=validationSamp
    testLoader = DataLoader(cifarTestSet, batch_size=batchSize)
```

```

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',

# loss function
lossFunction = nn.CrossEntropyLoss()

network = Net()
if(zeroWeight):
    network.apply(initWeightsToZero)

# optimizer
optimizer = optim.SGD(network.parameters(), lr=learningRate, momentum=momentum_)

start_time = time.time()
batchNum = 0
trainBatchLoss, batchNums = [], []
for epoch in range(5):
    trainLoss, validLoss = [], []

    ### Train
    for data, target in trainLoader:
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward Propagation
        output = network(data)

        # compute loss
        loss = lossFunction(output, target)

        # back propagation
        loss.backward()

        # update parameters
        optimizer.step()

    trainLoss.append(loss.item())

    batchNum += 1
    if((batchNum % 20) == 0 and outputType=="batchPlot"):
        trainBatchLoss.append(np.mean(trainLoss))
        batchNums.append(batchNum)
        trainLoss = []

    ### Validation
    for data, target in validationLoader:

        output = network(data)
        loss = lossFunction(output, target)
        validLoss.append(loss.item())

    if(outputType == "ephocText"):
        print("Epoch:", epoch+1, "|Training Loss: ", np.mean(trainLoss), "|Validation
    else:
        print("Epoch:", epoch+1, "|Validation Loss: ", np.mean(validLoss))

if(outputType=="batchPlot"):
    plt.plot(batchNums, trainBatchLoss)
    plt.xlabel("Batch Number")
    plt.ylabel("Train Loss")

```

```

plt.show

elapsedTime = time.time() - start_time
print("Training time:", elapsedTime)

### Test
correct = 0
total = 0
with torch.no_grad():
    for data, target in testLoader:

        output = network(data)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print('Accuracy of the network on the test images: %d %%' % (100 * correct / tota

```

▼ 1) Effect of Weights and Biases Initialization in Neural Network

```

class Net1(nn.Module):

    def __init__(self):
        super(Net1, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.linear1 = nn.Linear(64 * (4 * 4), 512)
        self.linear2 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 1024) ## reshaping

        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

```

▼ Zero Weight Network

```
evaluateNetwork(Net=Net1, zeroWeight=True)
```

```

0it [00:00, ?it/s]Downloading https://www.cs.toronto.edu/~kriz/cifar-10-pyth
100%|██████████| 170434560/170498071 [00:32<00:00, 4670545.55it/s]Files alre
170500096it [00:50, 4670545.55it/s] Epoch: 1 |?
Epoch: 2 |Training Loss: 2.303220442199707 |Validation Loss: 2.302892026048
Epoch: 3 |Training Loss: 2.3034199333190917 |Validation Loss: 2.30300837812
Epoch: 4 |Training Loss: 2.3033737691879272 |Validation Loss: 2.30285633943
Epoch: 5 |Training Loss: 2.3032839889526366 |Validation Loss: 2.30318831254
Training time: 259.6043703556061
Accuracy of the network on the test images: 10 %

```

▼ Network with Random Weights

```
evaluateNetwork(Net=Net1)
```

```

Files already downloaded and verified
Files already downloaded and verified
Epoch: 1 |Training Loss: 1.8007492104530334 |Validation Loss: 1.42652898398
Epoch: 2 |Training Loss: 1.3270365458488464 |Validation Loss: 1.24523507577
Epoch: 3 |Training Loss: 1.1023916879177094 |Validation Loss: 1.08421220128
Epoch: 4 |Training Loss: 0.9363559620857239 |Validation Loss: 1.01846098823
Epoch: 5 |Training Loss: 0.8128979505062103 |Validation Loss: 0.92499303770
Training time: 252.46883487701416
Accuracy of the network on the test images: 67 %

```

As shown above when weights and bias of network is set to zero, the accuracy is really low(10%).

It is because when you have zero weight and bias in all layers, in the first layer activation will be zero or very small, as each neuron output after applying bias and weights on sample inputs will be zero and after applying activation function on zero it will remain zero or close to it. This happens in next layers too and activations of following layers will get nearer to zero. Hence this will lead to no or very few activations and extremely small gradient decents in back propagation. Extremely small gradient decents in back propagation lead to no or negligible update on weights so loss does not decrease and accuracy can not boost up.

On the other hand, initializing weights and biases randomly as pytorch does by default will fix the problem stated before hence the network can decrease loss and increase accuracy significantly during training and achieve a high accuracy (67%).

▼ 2) Two Layer Network

```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5)


        self.linear1 = nn.Linear(16 * (28 * 28), 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))

        x = x.view(-1, 16 * (28 * 28))  ## reshaping

        x = self.linear1(x)
        return x
```

```
evaluateNetwork(Net=Net2, learningRate=0.001)
```

 Files already downloaded and verified
Files already downloaded and verified


Epoch: 1	Training Loss: 1.8041128494262695	Validation Loss: 1.61973992589
Epoch: 2	Training Loss: 1.4799123482227325	Validation Loss: 1.43004662501
Epoch: 3	Training Loss: 1.3764644934654235	Validation Loss: 1.37914164031
Epoch: 4	Training Loss: 1.324451096534729	Validation Loss: 1.364089932871
Epoch: 5	Training Loss: 1.287083025121689	Validation Loss: 1.353053773934

Training time: 99.09717416763306
Accuracy of the network on the test images: 52 %

As number of convolution layers and fully connected layers are less in this 2 layer network(one convolution layer and one output layer which is a fully connected layer), training and processes on data is not as advance and accurate as the first network and as the result we have a network with smaller accuracy (52%) and higher loss.

▼ 3) Network with Normalization

```
evaluateNetwork(Net=Net1, normalize=True)
```

 Files already downloaded and verified
Files already downloaded and verified

Epoch: 1	Training Loss: 1.6318812184333802	Validation Loss: 1.26573838250
Epoch: 2	Training Loss: 1.1342535645961762	Validation Loss: 1.02136254001
Epoch: 3	Training Loss: 0.9116428291082382	Validation Loss: 0.89707992680
Epoch: 4	Training Loss: 0.767465466761589	Validation Loss: 0.925454162846
Epoch: 5	Training Loss: 0.6541620981574059	Validation Loss: 0.86153480250

Training time: 274.42339992523193
Accuracy of the network on the test images: 70 %

Accuracy of network with normalization on input data will increase to 70% as normalized data make the optimization of network much easier and training more efficient. Because when different features do not have similar ranges of values, gradient can oscillate back and forth and take a long time to find its way to the minimum. In this case in the same time(5 epochs) network converges more on normalized data. Training time increased in this case but generally normalization will decrease the training time as the computation on features when they are in same ranges will be easier thus requires less time for training.

▼ 4) Effect of Learning Rate on Network

▼ high learning rate

```
evaluateNetwork(Net=Net1, normalize=True, learningRate=0.07, outputType="batchPlot")
```



Files already downloaded and verified

Files already downloaded and verified

Epoch: 1 | Validation Loss: 1.7008870725814527

Epoch: 2 | Validation Loss: 1.6820313820061974

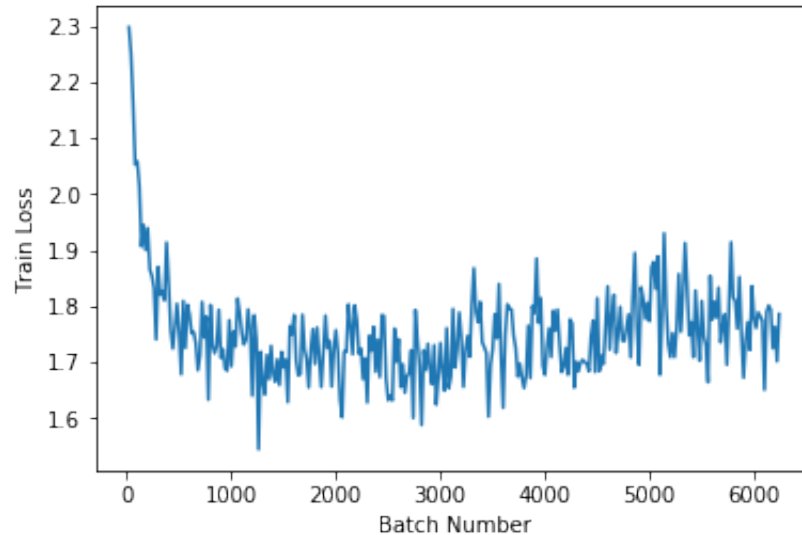
Epoch: 3 | Validation Loss: 1.7564301814514989

Epoch: 4 | Validation Loss: 1.766878911862358

Epoch: 5 | Validation Loss: 1.7785784684050197

Training time: 262.58147072792053

Accuracy of the network on the test images: 35 %

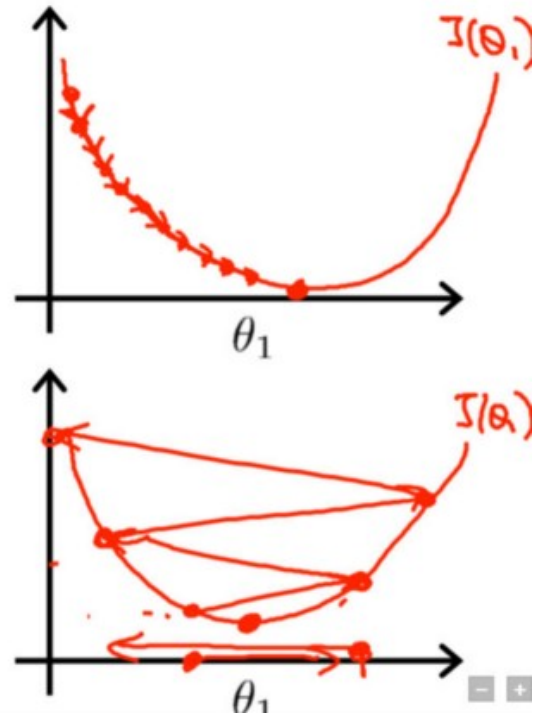


High learning rate means we travel along the downward slope in large steps and because of that we can not get close to the minimum more than a specific value because the step size is larger than our distance from minima and after that amount, we go far from minimum and increases the loss and as the result we can observe that network fails to converge.

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.

If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



▼ good learning rate


```
evaluateNetwork(Net=Net1, normalize=True, learningRate=0.01, outputType="batchPlot")
```



Files already downloaded and verified

Files already downloaded and verified

Epoch: 1 | Validation Loss: 1.3052531023756764

Epoch: 2 | Validation Loss: 1.0275526244800313

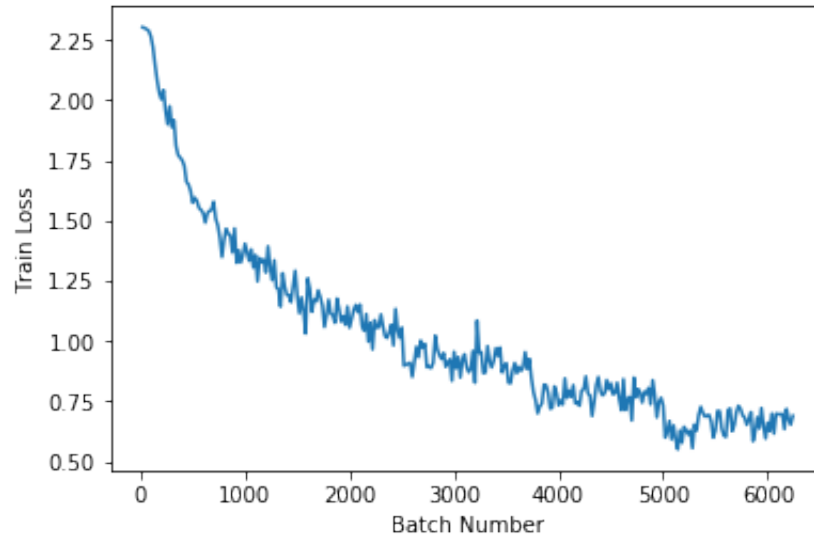
Epoch: 3 | Validation Loss: 0.9210575520992279

Epoch: 4 | Validation Loss: 0.8568399248603052

Epoch: 5 | Validation Loss: 0.8584645741378156

Training time: 271.0380771160126

Accuracy of the network on the test images: 70 %



With a good learning rate, we move towards minima with appropriate step size and eventually networks reaches the minimum and after that the loss remains approximately the same. As the result, it converges to local minima more quickly and needs less time for training and the loss decreases more than two other cases so the best accuracy will be reached with this step size.

▼ low learning rate

```
evaluateNetwork(Net=Net1, normalize=True, learningRate=0.0001, outputType="batchPlo
```



Files already downloaded and verified

Files already downloaded and verified

Epoch: 1 | Validation Loss: 2.2988300582471366

Epoch: 2 | Validation Loss: 2.293507007745127

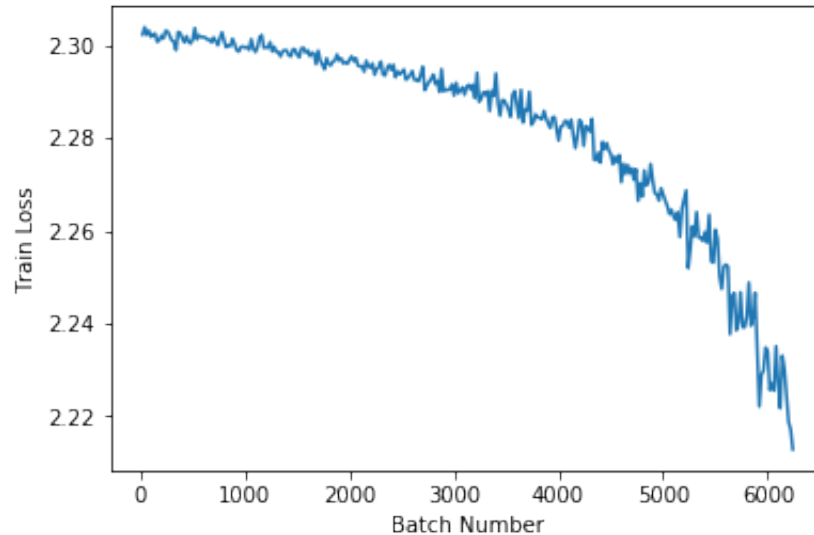
Epoch: 3 | Validation Loss: 2.2846795370022708

Epoch: 4 | Validation Loss: 2.2658587438991655

Epoch: 5 | Validation Loss: 2.2146633547335006

Training time: 272.2673370838165

Accuracy of the network on the test images: 22 %




Low learning rate means that we are moving along downward slope with very small step size and because of that it takes a lot of time to reduce loss and we spend more time far from minima and searching for it in comparison with a good learning rate. As you can see above, small step size will not let the loss to decrease enough in 5 epoches and after our training the loss lessened for a small amount and it is still a big amount for loss(2.21).

▼ 5) Effect of Batch Size on Network

▼ batch size = 64

with learning rate = 0.01(default value and as it was before)

```
evaluateNetwork(Net=Net1, normalize=True, batchSize=64)
```


 Files already downloaded and verified
Files already downloaded and verified

Epoch: 1	Training Loss: 1.7889031579971313	Validation Loss: 1.42242395498
Epoch: 2	Training Loss: 1.2913023562431336	Validation Loss: 1.19050387022
Epoch: 3	Training Loss: 1.0531460809707642	Validation Loss: 1.03415834751
Epoch: 4	Training Loss: 0.9085641683578491	Validation Loss: 0.93576855720
Epoch: 5	Training Loss: 0.7908004652500152	Validation Loss: 0.84933661081

Training time: 250.8223271369934
Accuracy of the network on the test images: 70 %

with learning rate = 0.05

```
evaluateNetwork(Net=Net1, normalize=True, batchSize=64, learningRate=0.05)
```

 Files already downloaded and verified
Files already downloaded and verified


Epoch: 1	Training Loss: 1.5896427129745483	Validation Loss: 1.25761269915
Epoch: 2	Training Loss: 1.163565060043335	Validation Loss: 1.129494930908
Epoch: 3	Training Loss: 1.0276961307525634	Validation Loss: 1.05576895528
Epoch: 4	Training Loss: 0.9302632021427154	Validation Loss: 1.06889185821
Epoch: 5	Training Loss: 0.8714286916255951	Validation Loss: 1.17135633252

Training time: 245.6736147403717
Accuracy of the network on the test images: 61 %

▼ batch size = 256

with learning rate = 0.01 (default value and as it was before)

```
evaluateNetwork(Net=Net1, normalize=True, batchSize=256.)
```


 Files already downloaded and verified
Files already downloaded and verified

Epoch	Training Loss	Validation Loss
Epoch: 1	2.191500560493226	1.96332482695
Epoch: 2	1.7911584028013192	1.64617900550
Epoch: 3	1.5512089304103973	1.49173874557
Epoch: 4	1.3945331786088884	1.41390042304
Epoch: 5	1.275004750604083	1.22248910665

Training time: 236.6599543094635
Accuracy of the network on the test images: 56 %

with learning rate = 0.1

```
evaluateNetwork(Net=Net1, normalize=True, batchSize=256, learningRate=0.1)
```

 Files already downloaded and verified
Files already downloaded and verified

Epoch	Training Loss	Validation Loss
Epoch: 1	1.802912678688195	1.34268413782
Epoch: 2	1.2493421849171826	1.14545345008
Epoch: 3	1.0586192262400487	1.07317802906
Epoch: 4	0.9040429352954694	1.00757960087
Epoch: 5	0.7724812983707258	0.92661058451

Training time: 229.06286644935608
Accuracy of the network on the test images: 68 %

We can observe that higher batch sizes lead to lower test accuracy. Network has 70% accuracy for batch size of 32 and 64, but for batch size of 256 accuracy reduces to 56%.

Higher batch sizes cons:

- Large batch size leads to slower convergence to that optima in comparison with smaller batch sizes so network will have a slower training with large batch sizes
- Too large batch size will lead to poor generalization and perhaps overfitting. Because it seems that higher variance gradients of minibatch gradient descent actually being tied to generalization.

Higher batch sizes pros:

- With small batch sizes, the model is not guaranteed to converge to the global optima because it will bounce around the global optima but Larger minibatch size means more "accurate" gradients and being closer to the "true" gradient
- Small batch results in rapid learning but a volatile learning process with higher variance in the accuracy. Larger batch sizes slow down the learning process but the final stages result in a convergence to a more stable model

With increasing the learning rate, we can recover the low accuracy caused by using a larger batch size. With 256 for batch size we achieve 56% accuracy when learning rate equals to 0.01 but this value increases to 68% for learning rate of 0.1 which is a huge improvement.

I think it is because with larger batch sizes gradients will be accurate, thus you can take larger steps without detriment and loss.

▼ 6) Different Activation Functions

▼ tanh

```

class Net_tanh(nn.Module):

    def __init__(self):
        super(Net_leakyRelu, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.linear1 = nn.Linear(64 * (4 * 4), 512)
        self.linear2 = nn.Linear(512, 10)


    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x)))
        x = self.pool(F.leaky_relu(self.conv2(x)))
        x = self.pool(F.leaky_relu(self.conv3(x)))

        x = x.view(-1, 1024)  ## reshaping

        x = F.leaky_relu(self.linear1(x))
        x = self.linear2(x)
        return x

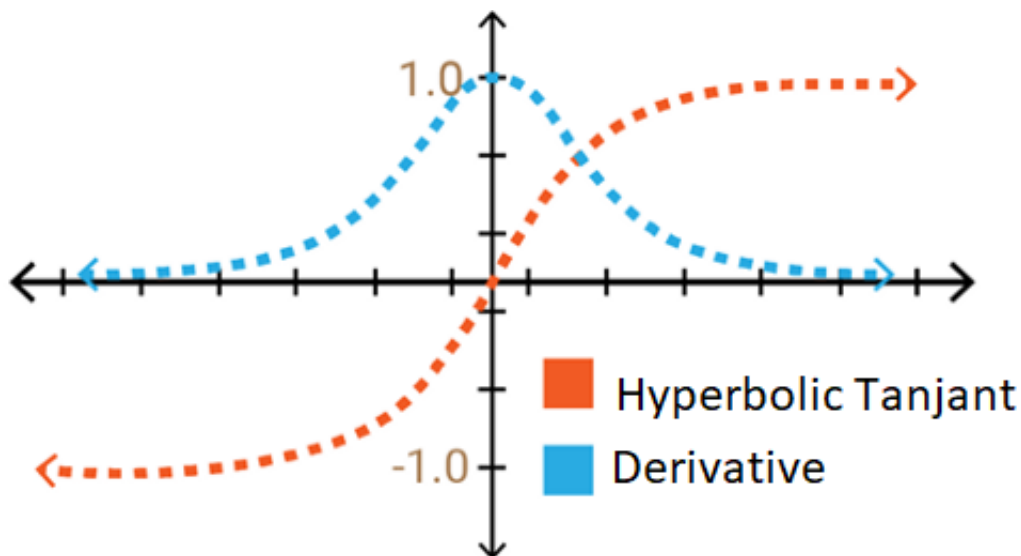
```

```
evaluateNetwork(Net=Net_ackFunc, normalize=True)
```

 Files already downloaded and verified
Files already downloaded and verified

Epoch: 1	Training Loss: 1.5954821367263794	Validation Loss: 1.30959294360
Epoch: 2	Training Loss: 1.0940746612548828	Validation Loss: 1.01590240991
Epoch: 3	Training Loss: 0.8947905774593353	Validation Loss: 0.91901419385
Epoch: 4	Training Loss: 0.7533263073444366	Validation Loss: 0.85144029276
Epoch: 5	Training Loss: 0.6467311169266701	Validation Loss: 0.90406991127

Training time: 268.95976734161377
Accuracy of the network on the test images: 69 %



Hyper bolic Tangent function gives us less accuracy. This function as an activation function suffer from the 'vanishing gradient problem' because as you can see the value of function changes slightly towards the ends of function which will cause derivatives to be very small in this regions and converge to zero. As the result even if the input of neuron differs hugely in this region, its output will not change or changes for an extremely small value. Vanishing problem can lead to minimal or zero leaning as there will be no update in weights.

Generally, the calculation with this function is more than a simple function and leads to more time for training.

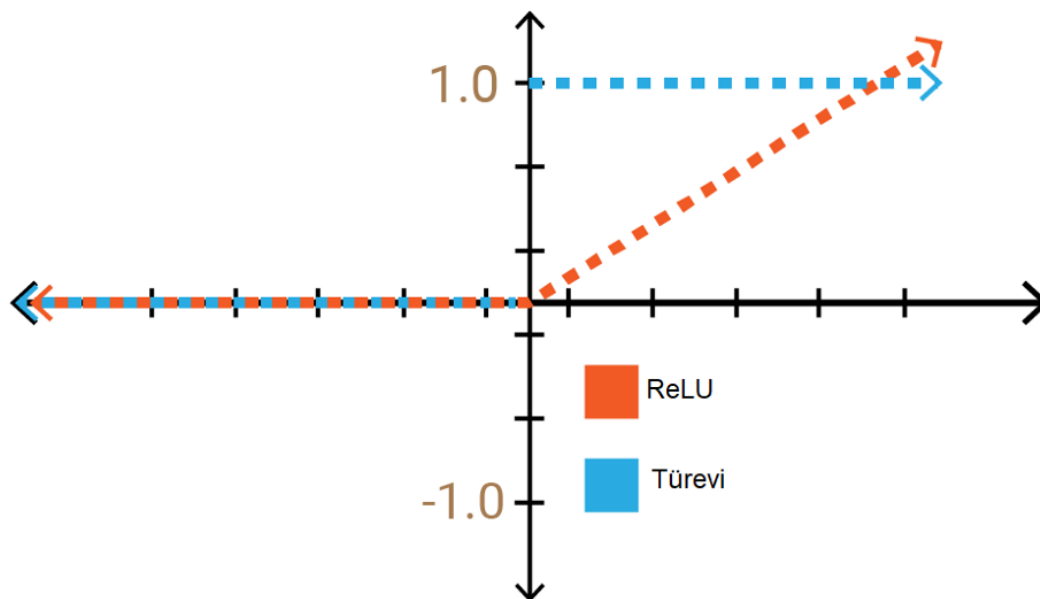
▼ ReLU

```
evaluateNetwork(Net=Net1, normalize=True)
```

```

Files already downloaded and verified
Files already downloaded and verified
Epoch: 1 | Training Loss: 1.6489029582500458 | Validation Loss: 1.28851316759
Epoch: 2 | Training Loss: 1.141836962556839 | Validation Loss: 1.023361399912
Epoch: 3 | Training Loss: 0.9366183966636658 | Validation Loss: 0.94283542046
Epoch: 4 | Training Loss: 0.7951607717514038 | Validation Loss: 0.86703613609
Epoch: 5 | Training Loss: 0.6696334651947021 | Validation Loss: 0.82476578876
Training time: 269.4092044830322
Accuracy of the network on the test images: 70 %

```



ReLU function is a good estimator because it is possible to converge with any function by combinations of ReLU.

As you can see we have the value of 0 on the negative axis which can have a positive and a negative effect on network :

(+)Sparsity Having the value of 0 on the negative axis means that some neurons will not activate.

Sparsity results better predictive power and less overfitting. In a sparse network, it's more probable that neurons are processing meaningful aspects of the problem so the model will be more concise. Also as the result of less activations, the calculation load is less thus the network will run faster.

(-)Dying ReLU Slope of ReLU in the negative range is also 0, So once a neuron gets negative, it's unlikely for it to recover. A ReLU neuron is called 'dead' if it always outputs 0 because of getting stuck in negative side. Learning is not happening in this area and dead neurons will not play any role in processing the input. The dying problem is likely to occur when there is a large negative bias.

▼ leakyrelu


```

class Net_leakyRelu(nn.Module):
    def __init__(self):
        super(Net_leakyRelu, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.linear1 = nn.Linear(64 * (4 * 4), 512)
        self.linear2 = nn.Linear(512, 10)


    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x)))
        x = self.pool(F.leaky_relu(self.conv2(x)))
        x = self.pool(F.leaky_relu(self.conv3(x)))

        x = x.view(-1, 1024)  ## reshaping

        x = F.leaky_relu(self.linear1(x))
        x = self.linear2(x)
        return x

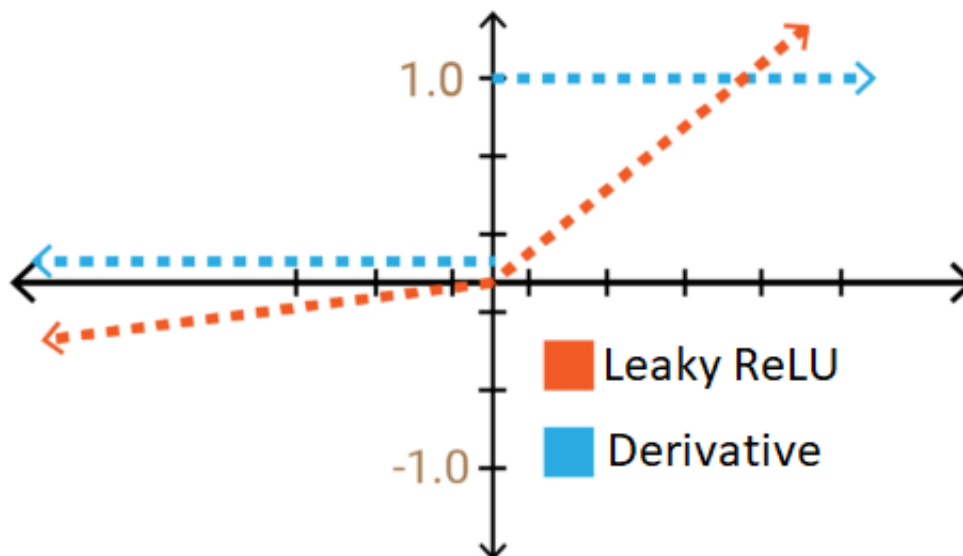
evaluateNetwork(Net=Net_leakyRelu, normalize=True)

```

 Files already downloaded and verified
Files already downloaded and verified

Epoch: 1	Training Loss: 1.6585179952144622	Validation Loss: 1.34729205571
Epoch: 2	Training Loss: 1.1499838267803193	Validation Loss: 1.05458026191
Epoch: 3	Training Loss: 0.9314553303241729	Validation Loss: 0.87238463464
Epoch: 4	Training Loss: 0.7814177124500274	Validation Loss: 0.90075320881
Epoch: 5	Training Loss: 0.6613423487782478	Validation Loss: 0.86237981915

Training time: 315.2214798927307
Accuracy of the network on the test images: 70 %



Leaky ReLU has a small slope for negative values, instead of altogether zero. As the result of that, we will no longer suffer from 'dying ReLU' problem. Also leaky ReLU is more balanced, and may therefore learn faster and gain a larger accuracy.

Training time of leaky ReLU is more than ReLU because we have more computation using this function.

▼ softplus

```
class Net_softplus(nn.Module):

    def __init__(self):
        super(Net_softplus, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.linear1 = nn.Linear(64 * (4 * 4), 512)
        self.linear2 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(F.softplus(self.conv1(x)))
        x = self.pool(F.softplus(self.conv2(x)))
        x = self.pool(F.softplus(self.conv3(x)))

        x = x.view(-1, 1024)  ## reshaping

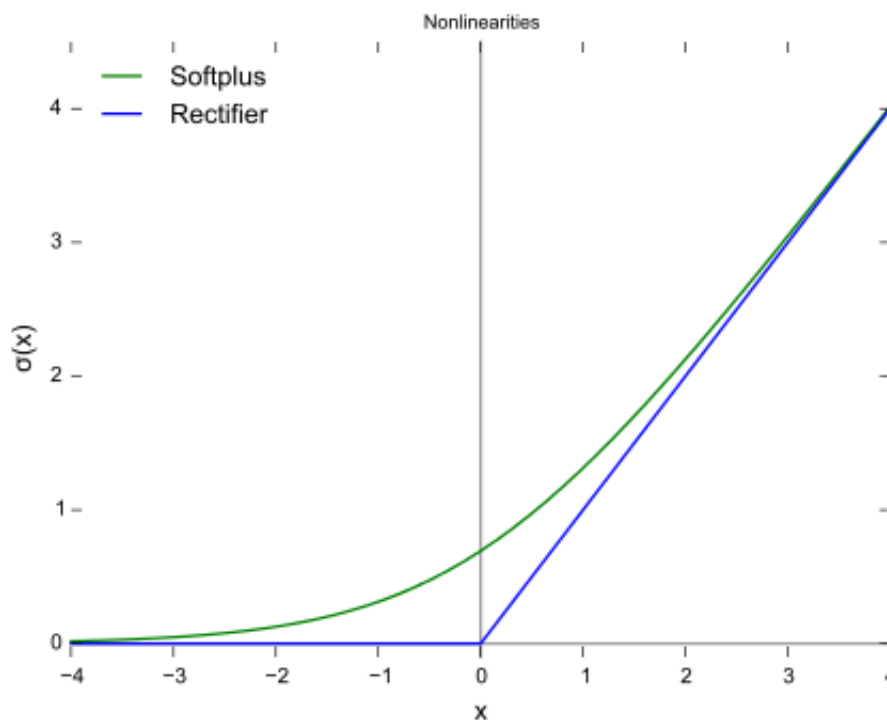
        x = F.softplus(self.linear1(x))
        x = self.linear2(x)
        return x
```

```
evaluateNetwork(Net=Net_softplus, normalize=True)
```

Files already downloaded and verified
 Files already downloaded and verified

Epoch: 1	Training Loss: 2.2085207825660707	Validation Loss: 1.97475995421
Epoch: 2	Training Loss: 1.7062097623825074	Validation Loss: 1.54925002000
Epoch: 3	Training Loss: 1.4292797028064728	Validation Loss: 1.38952788282
Epoch: 4	Training Loss: 1.2724703412532807	Validation Loss: 1.26096659403
Epoch: 5	Training Loss: 1.151931735610962	Validation Loss: 1.19774297403

Training time: 730.3430860042572
 Accuracy of the network on the test images: 58 %



ReLU and Softplus are largely similar, except near 0 where the softplus is smooth and differentiable. It's much easier and efficient to compute ReLU and its derivative than for the softplus function which has log and exp in its formulation. As computing the activation function and its derivative is as frequent in deep learning thus with using ReLU the forward and backward passes are much faster. So as you can see the training time using softplus is more than ReLU and the accuracy decrease to 58%.

▼ 7) Effect of Momentum on SGD

▼ without momentum

```
evaluateNetwork(Net=Net1, normalize=True, momentum_=0.)
```

Files already downloaded and verified
Files already downloaded and verified

Epoch	Training Loss	Validation Loss
Epoch: 1	2.206315065193176	1.957371818752
Epoch: 2	1.7846874520301819	1.63725133978
Epoch: 3	1.5707276392936707	1.50433045835
Epoch: 4	1.434236606168747	1.378247215724
Epoch: 5	1.327245115852356	1.281072553354

Training time: 267.7429630756378
Accuracy of the network on the test images: 53 %



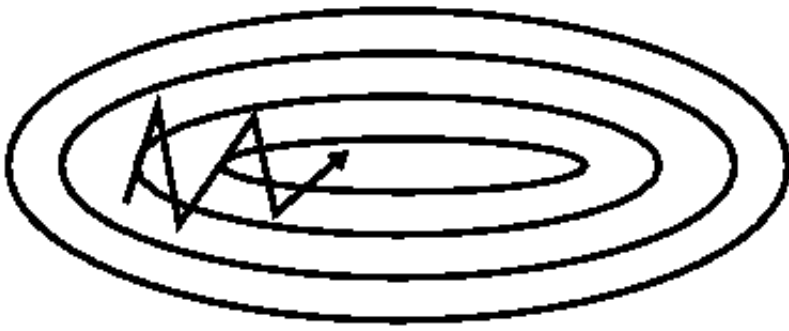
▼ with momentum

```
evaluateNetwork(Net=Net1, normalize=True)
```

Files already downloaded and verified
Files already downloaded and verified

Epoch	Training Loss	Validation Loss
Epoch: 1	1.6441562950611115	1.37870875734
Epoch: 2	1.1420481600284575	1.03390513060
Epoch: 3	0.915047332572937	0.943318764432
Epoch: 4	0.7774663432598115	0.86387454635
Epoch: 5	0.6579337474942207	0.82054709997

Training time: 268.9390833377838
Accuracy of the network on the test images: 71 %



As the results are shown above accuracy of network with momentum is higher and training times are approximately same.

In SGD the exact derivate of our loss function is not computed, instead an estimation of it on a small batch is computed. As the result because this Derivatives are noisy, this will cause us not to always move in the optimal direction.

Momentum which is an exponentially weighed average of our gradients can help us by denoising the data and bring it closer to the right direction and fix the problem we mentioned before for SGD.

As the result SGD with momentum can provide us a better estimate of actual derivate thus it works better and faster than classic SGD. Momentum helps to accelerate gradient vectors in the right directions, leading to faster converging and less time for training also more accuracy will be achieved.