## Assignment #9

**Pieter van Gaalen**

**Convolution Neural Network**

Convolution Neural Networks (CNN) is the most popular type of neural network used for image analysis. CNN can detect patterns which makes it powerful for image analysis. Depending on the number of convolution networks the network can recognize more complex shapes. In the primary layers, the network might be able to recognize edges, corners, or simple shapes. In deeper layers, the network might be able to recognize faces, emotions, animals or whether.

**Table 1 Experiment:**

|  |  |
| --- | --- |
| Experiment | Learning Rate |
| Control | Lr = 1e-2 |
| Modification | Lr = 1e-1 |

**What model are you starting with?**

The control model I will be using is the convolution neural network example provided in the assignment

**What modification are you making?**

For my experiment, I will be modifying the learning rate parameter. The line of code specifically modified was:



**What data will you use?**

The images that were used were from tiny-imagenet-200 and put into a flattened format.

**Training Paraments**

For the CNN the following were used:

* batch limit = 200
* num\_classes = 200
* max\_iter = 50

**Summary of results and your conclusions**

Increasing the learning rate help initially improve the accuracy of the CNN model. This can be seen from the graphs below. The control model was able to reach a validation accuracy of approximately 20% at iteration 70. The modified model was able to reach a validation accuracy of approximately 20% at iteration 50. However, due to hardware restrictions, I was unable to see the long term effects of increasing this parameter. Based on research the following would be expected if the experiment was run longer.

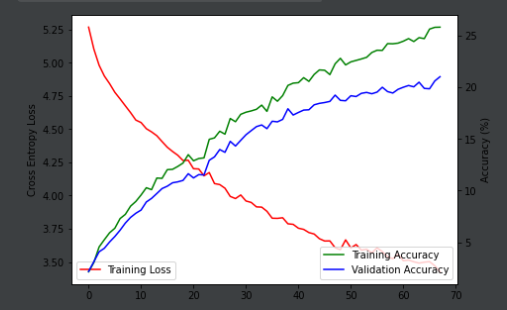
Adjusting the learning rate has trade-offs. Increasing the learning rate too high can cause the model to jump over a minimum and oscillate around the global minimum. Conversely, a learning rate that is too low will take too long to converge or get stuck in a local minimum.

Ideally, the learning rate should be dynamically adjusted during the training of the model. For an untuned model, it is ideal to have a large learning rate to quickly converge to the global minimum. As the model gets closer to the global minimum the learning rate should decrease to prevent overshooting it. Eventually, the learning rate should approach zero to allow for fine-tuning of the model and allow for the most optimized and ideally accurate model.

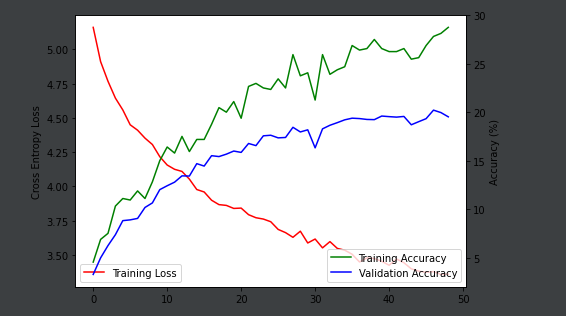
Note: I wanted to get the confusion matrix to work, but I was unable to figure it out. A lesson on how to implement this might be helpful. This assignment was a lot of fun to play around with different parameters.

## Results

**Control Model Results**



**Modified Model Results**

****

## Code

**Control Model Code (from Example)**

class ConvNet(torch.nn.Module):  
   
 def \_\_init\_\_(self, h, w, outputs):  
 super(ConvNet, self).\_\_init\_\_()  
 self.conv1 = torch.nn.Conv2d(3, 16, kernel\_size=5, stride=2)  
 self.bn1 = torch.nn.BatchNorm2d(16)  
 self.conv2 = torch.nn.Conv2d(16, 32, kernel\_size=5, stride=2)  
 self.bn2 = torch.nn.BatchNorm2d(32)  
 self.conv3 = torch.nn.Conv2d(32, 32, kernel\_size=5, stride=2)  
 self.bn3 = torch.nn.BatchNorm2d(32)  
  
 # Number of Linear input connections depends on output of conv2d layers  
 # and therefore the input image size, so compute it.  
 def conv2d\_size\_out(size, kernel\_size = 5, stride = 2):  
 return (size - (kernel\_size - 1) - 1) // stride + 1  
 convw = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(w)))  
 convh = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(h)))  
 linear\_input\_size = convw \* convh \* 32  
 self.head = torch.nn.Linear(linear\_input\_size, outputs)  
  
 # Called with either one element to determine next action, or a batch  
 # during optimization. Returns tensor([[left0exp,right0exp]...]).  
 def forward(self, x):  
 x = torch.nn.functional.relu(self.bn1(self.conv1(x)))  
 x = torch.nn.functional.relu(self.bn2(self.conv2(x)))  
 x = torch.nn.functional.relu(self.bn3(self.conv3(x)))  
 return self.head(x.view(x.size(0), -1))  
  
  
#%%  
  
batch\_limit = 200  
num\_features = train\_data\_np[0].shape # (64, 64, 3)  
num\_classes = 200  
max\_iter = 50  
model = ConvNet(64, 64, num\_classes)  
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)  
criterion = torch.nn.CrossEntropyLoss()  
  
loss\_list = []  
accuracy\_list = []  
valid\_list = []  
  
train\_preds = []  
  
def get\_all\_preds(model, loader):  
 all\_preds = torch.tensor([])  
 for batch in loader:  
 images, labels = batch  
  
 preds = model(images)  
 all\_preds = torch.cat(  
 (all\_preds, preds)  
 ,dim=0  
 )  
 return all\_preds  
  
for i in range(max\_iter):  
 epoch\_loss = 0  
 for index, (data, label) in enumerate(train\_dataset):  
 optimizer.zero\_grad()  
 y\_pred = model(data)  
 loss = criterion(input=y\_pred, target=label)  
 loss.backward()  
 optimizer.step()  
 epoch\_loss += loss.item()  
 print(f"Batch loss ({index+1}/{batch\_limit}): {loss.item()}", end='\r')  
 if index >= batch\_limit -1:  
 break  
 with torch.no\_grad():  
   
 loss\_list.append(epoch\_loss/batch\_limit)  
 y\_pred = model.forward(torch.Tensor(train\_data\_np.reshape(-1, 3, 64, 64)))  
 y\_pred = torch.argmax(y\_pred, dim=1).detach().numpy()  
  
 accuracy\_list.append(accuracy\_score(y\_true=train\_labels, y\_pred=y\_pred)\*100)  
 y\_pred\_val = model.forward(torch.Tensor(valid\_data\_np.reshape(-1, 3, 64, 64)))  
 y\_pred\_val = torch.argmax(y\_pred\_val, dim=1).detach().numpy()  
   
 valid\_list.append(accuracy\_score(y\_true=valid\_labels, y\_pred=y\_pred\_val)\*100)  
 # print(f"Loss at epoch {i}: {loss.item():.4f}\tAccuracy: {accuracy\_list[-1]\*100:.2f}%", end='\r')  
 live\_plot(np.array(loss\_list), np.array(accuracy\_list), valid\_list)

**Modified Model Code**

#%%  
  
from IPython.display import clear\_output  
from matplotlib import pyplot as plt  
import collections  
import numpy as np  
import pandas as pd  
from PIL import Image  
import torch  
import torchvision  
from sklearn.metrics import accuracy\_score  
from sklearn.metrics import plot\_confusion\_matrix  
from sklearn.metrics import confusion\_matrix  
  
import image  
  
#%%  
  
def live\_plot(loss, train\_acc, valid\_acc=None, figsize=(7,5), title=''):  
 clear\_output(wait=True)  
 fig, ax1 = plt.subplots(figsize=figsize)  
 ax1.plot(loss, label='Training Loss', color='red')  
 ax1.legend(loc='lower left')  
 ax1.set\_ylabel('Cross Entropy Loss')  
 ax2 = ax1.twinx()  
 ax2.plot(train\_acc, label='Training Accuracy', color='green')  
 if valid\_acc is not None:  
 ax2.plot(valid\_acc, label='Validation Accuracy', color='blue')  
 ax2.legend(loc='lower right')  
 ax2.set\_ylabel('Accuracy (%)')  
 ax2.set\_xlabel('Epoch')  
 plt.title(title)  
 plt.show()  
  
  
#%%  
  
def load\_train\_dataset():  
 data\_path = './tiny-imagenet-200/train/'  
 train\_dataset = torchvision.datasets.ImageFolder(  
 root=data\_path,  
 transform=torchvision.transforms.ToTensor()  
 )  
 train\_loader = torch.utils.data.DataLoader(  
 train\_dataset,  
 batch\_size=32,  
 num\_workers=0,  
 shuffle=True  
 )  
 return train\_loader, train\_dataset.class\_to\_idx  
  
#%%  
  
def load\_valid\_dataset(class\_to\_idx):  
 data\_path = './tiny-imagenet-200/val/images'  
 label\_file = open('./tiny-imagenet-200/val/val\_annotations.txt', 'r')  
 label\_df = pd.read\_csv(label\_file, delim\_whitespace=True, header=None)  
 label\_df[1] = label\_df[1].apply(lambda x : class\_to\_idx[x])  
 valid\_data = []  
 for row in label\_df.iterrows():  
 image = Image.open(f'{data\_path}/{row[1][0]}')  
 image = torchvision.transforms.functional.to\_tensor(image)  
 label = row[1][1]  
 valid\_data.append((image, label))  
 return valid\_data  
  
#%%  
  
def load\_data\_np(batches=100):  
 # Load a subsample of training data and all of the validation data into a flattened NumPy format.  
 train\_data\_np = []  
 train\_labels = []  
 batch\_limit = batches  
 for index, (data, label) in enumerate(train\_dataset):  
 for i in range(data.shape[0]):  
 train\_data\_np.append(data[i].detach().numpy().flatten())  
 train\_labels.append(int(label[i].detach().numpy()))  
 if index >= batch\_limit - 1:  
 break  
 train\_data\_np = np.vstack(train\_data\_np)  
 train\_labels = np.array(train\_labels)  
  
 valid\_data\_np = []  
 valid\_labels = []  
 for data, label in valid\_dataset:  
 if data.shape[0] < 3:  
 data = torch.cat([data[0], data[0], data[0]], dim=0) # Convert grayscale images into RGB format  
 valid\_data\_np.append(data.detach().numpy().flatten())  
 valid\_labels.append(int(label))  
 valid\_data\_np = np.vstack(valid\_data\_np)  
 valid\_labels = np.array(valid\_labels)  
 return train\_data\_np, train\_labels, valid\_data\_np, valid\_labels  
  
#%%  
  
train\_dataset, class\_to\_idx = load\_train\_dataset()  
valid\_dataset = load\_valid\_dataset(class\_to\_idx)  
  
#%% md  
  
*## Visualize Images and Labels*#%%  
  
label\_map = pd.read\_csv("./tiny-imagenet-200/words.txt", delimiter='\t', header=None)  
label\_map = dict(zip(label\_map[0], label\_map[1]))  
idx\_to\_class = {v:k for (k,v) in class\_to\_idx.items()}  
for index, (data, labels) in enumerate(train\_dataset):  
 break  
for image\_index in range(len(data)):  
 image = data[image\_index].detach().numpy().T  
 label = labels.detach().numpy()[image\_index]  
 plt.imshow(image)  
 plt.title(f"{label\_map[idx\_to\_class[label]]}")  
 plt.figure()  
plt.show()  
  
#%% md  
  
*## Linear Model in PyTorch*#%%  
  
# Load data into 'flattened' NumPy arrays  
train\_data\_np, train\_labels, valid\_data\_np, valid\_labels = load\_data\_np(batches=20)  
print(f"Training Data Shape: {train\_data\_np.shape}")  
print(f"Training Labels Shape: {train\_labels.shape}")  
print(f"Validation Data Shape: {valid\_data\_np.shape}")  
print(f"Validation Labels Shape: {valid\_labels.shape}")  
  
  
#%% md  
  
*## Convolutional Neural Network in PyTorch*#%%  
  
class ConvNet(torch.nn.Module):  
   
 def \_\_init\_\_(self, h, w, outputs):  
 super(ConvNet, self).\_\_init\_\_()  
 self.conv1 = torch.nn.Conv2d(3, 16, kernel\_size=5, stride=2)  
 self.bn1 = torch.nn.BatchNorm2d(16)  
 self.conv2 = torch.nn.Conv2d(16, 32, kernel\_size=5, stride=2)  
 self.bn2 = torch.nn.BatchNorm2d(32)  
 self.conv3 = torch.nn.Conv2d(32, 32, kernel\_size=5, stride=2)  
 self.bn3 = torch.nn.BatchNorm2d(32)  
  
 # Number of Linear input connections depends on output of conv2d layers  
 # and therefore the input image size, so compute it.  
 def conv2d\_size\_out(size, kernel\_size = 5, stride = 2):  
 return (size - (kernel\_size - 1) - 1) // stride + 1  
 convw = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(w)))  
 convh = conv2d\_size\_out(conv2d\_size\_out(conv2d\_size\_out(h)))  
 linear\_input\_size = convw \* convh \* 32  
 self.head = torch.nn.Linear(linear\_input\_size, outputs)  
  
 # Called with either one element to determine next action, or a batch  
 # during optimization. Returns tensor([[left0exp,right0exp]...]).  
 def forward(self, x):  
 x = torch.nn.functional.relu(self.bn1(self.conv1(x)))  
 x = torch.nn.functional.relu(self.bn2(self.conv2(x)))  
 x = torch.nn.functional.relu(self.bn3(self.conv3(x)))  
 return self.head(x.view(x.size(0), -1))  
  
  
#%%  
  
batch\_limit = 200  
num\_features = train\_data\_np[0].shape # (64, 64, 3)  
num\_classes = 200  
max\_iter = 50  
model = ConvNet(64, 64, num\_classes)  
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)  
criterion = torch.nn.CrossEntropyLoss()  
  
loss\_list = []  
accuracy\_list = []  
valid\_list = []  
  
train\_preds = []  
  
def get\_all\_preds(model, loader):  
 all\_preds = torch.tensor([])  
 for batch in loader:  
 images, labels = batch  
  
 preds = model(images)  
 all\_preds = torch.cat(  
 (all\_preds, preds)  
 ,dim=0  
 )  
 return all\_preds  
  
for i in range(max\_iter):  
 epoch\_loss = 0  
 for index, (data, label) in enumerate(train\_dataset):  
 optimizer.zero\_grad()  
 y\_pred = model(data)  
 loss = criterion(input=y\_pred, target=label)  
 loss.backward()  
 optimizer.step()  
 epoch\_loss += loss.item()  
 print(f"Batch loss ({index+1}/{batch\_limit}): {loss.item()}", end='\r')  
 if index >= batch\_limit -1:  
 break  
 with torch.no\_grad():  
   
 loss\_list.append(epoch\_loss/batch\_limit)  
 y\_pred = model.forward(torch.Tensor(train\_data\_np.reshape(-1, 3, 64, 64)))  
 y\_pred = torch.argmax(y\_pred, dim=1).detach().numpy()  
  
 accuracy\_list.append(accuracy\_score(y\_true=train\_labels, y\_pred=y\_pred)\*100)  
 y\_pred\_val = model.forward(torch.Tensor(valid\_data\_np.reshape(-1, 3, 64, 64)))  
 y\_pred\_val = torch.argmax(y\_pred\_val, dim=1).detach().numpy()  
   
 valid\_list.append(accuracy\_score(y\_true=valid\_labels, y\_pred=y\_pred\_val)\*100)  
 # print(f"Loss at epoch {i}: {loss.item():.4f}\tAccuracy: {accuracy\_list[-1]\*100:.2f}%", end='\r')  
 live\_plot(np.array(loss\_list), np.array(accuracy\_list), valid\_list)