SkipBlocks on the COCO dataset

Athreyan Mohana Krishnan Sangeetha

February 22, 2025

1 Introduction

The skipblock contains a direct connection of the inut as an output.

2 The imported SkipBlock

[ ]:
```
# Modified class borrowed from https://engineering.purdue.edu/kak/distDLS/
↪DLStudio-2.3.3_CodeOnly.html
class SkipBlock(nn.Module):
            """
            Class Path:   DLStudio  ->  SkipConnections  ->  SkipBlock
            """
            def __init__(self, in_ch, out_ch, downsample=False,␣
↪skip_connections=True):
                super(SkipBlock, self).__init__()
                self.downsample = downsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch
                self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
                self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
                self.bn1 = nn.BatchNorm2d(out_ch)
                self.bn2 = nn.BatchNorm2d(out_ch)
                if downsample:
                    ##  Setting stride to 2 and kernel_size to 1 amounts to␣
↪retaining every
                    ##  other pixel in the image --- which halves the size of␣
↪the image:
                    self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

            def forward(self, x):
                identity = x
                out = self.convo1(x)
                out = self.bn1(out)
                out = nn.functional.relu(out)
                if self.in_ch == self.out_ch:
                    out = self.convo2(out)
```

```
                out = self.bn2(out)
                out = nn.functional.relu(out)
            # if self.downsample:
            #     out = self.downsampler(out)
            #     identity = self.downsampler(identity)
            if self.skip_connections:
                if self.in_ch == self.out_ch:
#                       out += identity
                    out = out + identity
                else:
                    ## To understand the following assignments, recall that␣
↪the data has the
                    ## shape [B,C,H,W]. So it is the second axis that␣
↪corresponds to the channels
                   out[:,:self.in_ch,:,:] += identity
                   out[:,self.in_ch:,:,:] += identity
                    # out = out + torch.cat((identity, identity), dim=1)
            return nn.functional.relu(out)
```

3 Code for the network

[ ]:
```
# The code below is based on code from  https://engineering.purdue.edu/kak/
↪distDLS/DLStudio-2.3.3_CodeOnly.html
class BMEnet2(nn.Module):
    def __init__(self, skip_connections=True, depth=20, channel_size=32):
        super(BMEnet2, self).__init__()
        self.depth = depth
        #First layer to convert 3 rgb channel tensor into a 64 channel tensor
        self.first = SkipBlock(3, channel_size,skip_connections=False,␣
↪downsample=False)

        # First MaxPool layer to bring down the size from 64x64 to 32x32
        self.pool1 = nn.MaxPool2d(2, 2)

        # First list of skipblock layers. Has depth/2 instances of skipblock
        self.multiLayer1 = nn.ModuleList([SkipBlock(channel_size, channel_size,␣
↪skip_connections=skip_connections, downsample=False) for i in range(0,self.
↪depth,2)])

        # First MaxPool layer to bring down the size from 32x32 to 16x16
        self.pool2 = nn.MaxPool2d(2,2)

        # Second list of skipblock layers. Has depth/2 instances of skipblock
        self.multiLayer2 = nn.ModuleList([SkipBlock(channel_size, channel_size,␣
↪skip_connections=skip_connections, downsample=False) for i in range(0,self.
↪depth,2)])
```

```
        #Linear layer to convert to a vector of size 5
        self.fc1 =  nn.Linear(channel_size*16*16, 1000)
        self.fc2 =  nn.Linear(1000, 5)

    def forward(self, x):
        x = self.pool1(nn.functional.relu(self.first(x)))
        #Loop for first list
        for layer in self.multiLayer1:
            x = nn.functional.relu(layer(x))

        x = self.pool2(x)
        #Loop for second list
        for layer in self.multiLayer2:
            x = nn.functional.relu(layer(x))
        x  =  x.view( x.shape[0], - 1 )
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

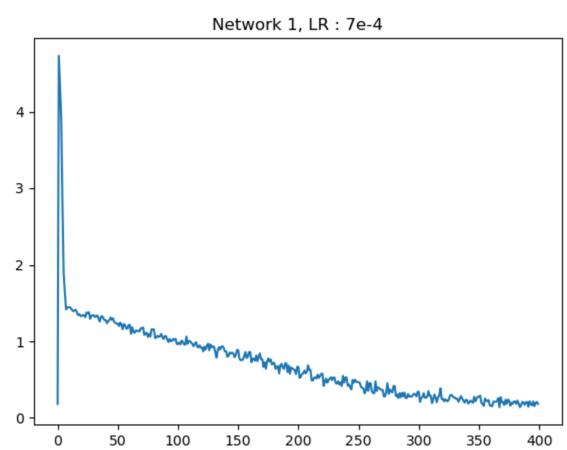
4 Make sure the networks have 40 layers

```
print("The number of layers in the new network BMEnet2 is " + str(len(list(BMEnet2().parameters()))))
The number of layers in the new network BMEnet2 is 172
```

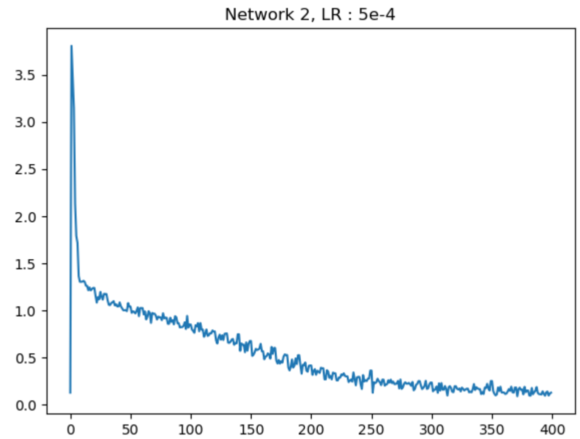
The network has 172 learnable layers

5 The loss graph for the two learning rates

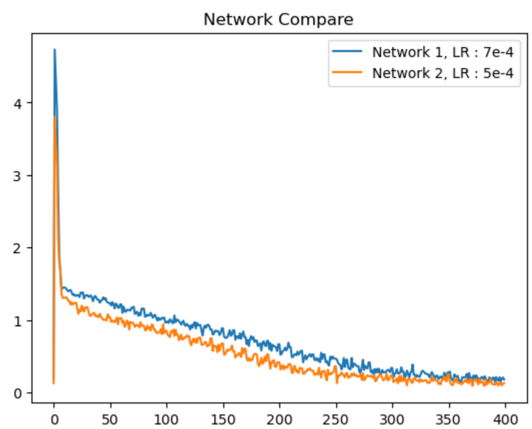
5.1 Network 1 with learning rate 7e-4



5.2 Network 2 with learning rate 5e-4



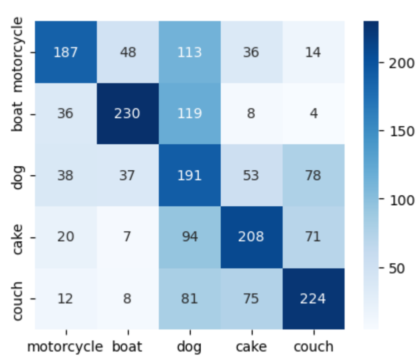
6 Comparison graph between losses



7 Confusion matrices for three matrices

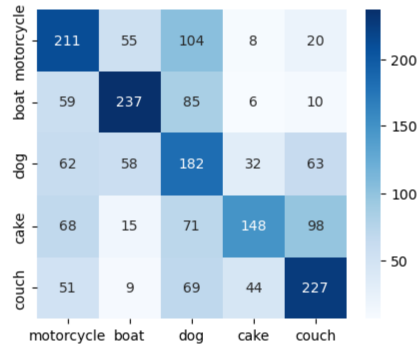
7.1 Confusion matrix for network 1 with learning rate 7e-4

Overall accuracy of the network on the test images: 52 %



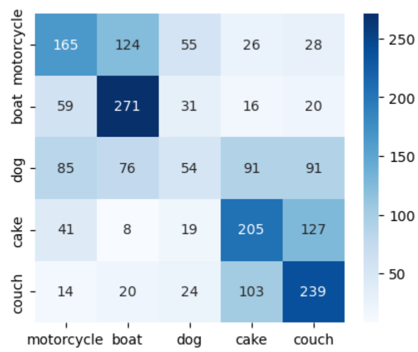
7.2 Confusion matrix for network 2 with learning rate 5e-4

Overall accuracy of the network on the test images: 50 %



7.3 Confusion matrix of network 3 without skipblocks 4

Overall accuracy of the network on the test images: 46 %



8 Comparing results of the networks

Both of the two networks perform better than Network 3 without skipblocks which only achieved an accuracy of 46%. The network constructed with SkipBlocks not only is deeper and hence has potential to generalize more it also trains faster compared with Network 3. This is due to the skipblock mitigating the issue of vanishing gradient.

9 Code

[ ]:
```
import torch
import torchvision.transforms as tvt
from PIL import Image
import numpy as np
from scipy.stats import wasserstein_distance
import os
import random
from torch.utils.data import DataLoader
import time
```

```python
import matplotlib.pyplot as plt
import json
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt

# from google.colab import drive
# import google.colab.auth
# google.colab.auth.authenticate_user()
# drive.mount('/content/drive')

class COCO_mod ( torch . utils . data . Dataset ):

    def __init__ ( self , root, class_labels ):
        super () . __init__ ()
        self.root = root
        self.datapoints = os.listdir(root)
        self.size = len([name for name in self.datapoints])
        self.class_labels = class_labels
        print("Dataset initalized")

    def __len__ (self):
        return self.size

    def __getitem__ ( self , index ):
        item = Image.open(self.root + self.datapoints[index%self.size]).
↪convert("RGB")
        name = self.datapoints[index%self.size].split('.')[0]
        name = name.split(' ')[0]
        if self.class_labels[name] == 4:
            return_label = torch.FloatTensor([1,0,0,0,0])
        elif self.class_labels[name] == 9:
            return_label = torch.FloatTensor([0,1,0,0,0])
        elif self.class_labels[name] == 18:
            return_label = torch.FloatTensor([0,0,1,0,0])
        elif self.class_labels[name] == 61:
            return_label = torch.FloatTensor([0,0,0,1,0])
        elif self.class_labels[name] == 63:
            return_label = torch.FloatTensor([0,0,0,0,1])
        # [4, 9, 18, 61, 63]
        return self.tensorify(item), return_label

    def tensorify(self,image):
        augmenter = tvt.Compose([
```

```
            tvt.ToTensor(),
            tvt.CenterCrop([64,64])
        ])
        return augmenter(image)

val_root = "compressed/val/"
train_root = "compressed/train/"
f = open('image_lists.json')
class_labels = json.load(f)

my_val_dataset = COCO_mod(val_root,class_labels["val"])
length_val = len(my_val_dataset)
# randomList = [random.randint(0,length_val) for _ in range(1000)]
# val_dataset = [my_val_dataset.__getitem__(i) for i in range(length_val)]

my_train_dataset = COCO_mod(train_root,class_labels["train"])
length_train = len(my_train_dataset)
# randomList = [random.randint(0,length_train) for _ in range(1000)]
# train_dataset = [my_train_dataset.__getitem__(i) for i in range(length_train)]
print("Dataset done")

val_dataloader = DataLoader ( my_val_dataset , batch_size = 8,shuffle = True )

train_dataloader = DataLoader ( my_train_dataset , batch_size = 8,shuffle = True␣
 ↪)
print("Dataloader done")

# Modified class borrowed from https://engineering.purdue.edu/kak/distDLS/
 ↪DLStudio-2.3.3_CodeOnly.html
class SkipBlock(nn.Module):
            """
            Class Path:   DLStudio  ->  SkipConnections  ->  SkipBlock
            """
            def __init__(self, in_ch, out_ch, downsample=False,␣
 ↪skip_connections=True):
                super(SkipBlock, self).__init__()
                self.downsample = downsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch
                self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
                self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
                self.bn1 = nn.BatchNorm2d(out_ch)
                self.bn2 = nn.BatchNorm2d(out_ch)
                if downsample:
                    ##  Setting stride to 2 and kernel_size to 1 amounts to␣
 ↪retaining every
```

```
                    ##  other pixel in the image --- which halves the size of␣
↪the image:
                    self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

            def forward(self, x):
                identity = x
                out = self.convo1(x)
                out = self.bn1(out)
                out = nn.functional.relu(out)
                if self.in_ch == self.out_ch:
                    out = self.convo2(out)
                    out = self.bn2(out)
                    out = nn.functional.relu(out)
                # if self.downsample:
                #     out = self.downsampler(out)
                #     identity = self.downsampler(identity)
                if self.skip_connections:
                    if self.in_ch == self.out_ch:
#                       out += identity
                        out = out + identity
                    else:
                        ## To understand the following assignments, recall that␣
↪the data has the
                        ## shape [B,C,H,W]. So it is the second axis that␣
↪corresponds to the channels
                        out[:,:self.in_ch,:,:] += identity
                        out[:,self.in_ch:,:,:] += identity
                        # out = out + torch.cat((identity, identity), dim=1)
                return nn.functional.relu(out)

# The code below is based on code from  https://engineering.purdue.edu/kak/
↪distDLS/DLStudio-2.3.3_CodeOnly.html
class BMEnet2(nn.Module):
    def __init__(self, skip_connections=True, depth=20, channel_size=32):
        super(BMEnet2, self).__init__()
        self.depth = depth
        #First layer to convert 3 rgb channel tensor into a 64 channel tensor
        self.first = SkipBlock(3, channel_size,skip_connections=False,␣
↪downsample=False)

        # First MaxPool layer to bring down the size from 64x64 to 32x32
        self.pool1 = nn.MaxPool2d(2, 2)

        # First list of skipblock layers. Has depth/2 instances of skipblock
        self.multiLayer1 = nn.ModuleList([SkipBlock(channel_size, channel_size,␣
↪skip_connections=skip_connections, downsample=False) for i in range(0,self.
↪depth,2)])
```

```
        # First MaxPool layer to bring down the size from 32x32 to 16x16
        self.pool2 = nn.MaxPool2d(2,2)

        # Second list of skipblock layers. Has depth/2 instances of skipblock
        self.multiLayer2 = nn.ModuleList([SkipBlock(channel_size, channel_size, 
 ↪skip_connections=skip_connections, downsample=False) for i in range(0,self.
 ↪depth,2)])

        #Linear layer to convert to a vector of size 5
        self.fc1 =  nn.Linear(channel_size*16*16, 1000)
        self.fc2 =  nn.Linear(1000, 5)

    def forward(self, x):
        x = self.pool1(nn.functional.relu(self.first(x)))
        #Loop for first list
        for layer in self.multiLayer1:
            x = nn.functional.relu(layer(x))

        x = self.pool2(x)
        #Loop for second list
        for layer in self.multiLayer2:
            x = nn.functional.relu(layer(x))
        x  =  x.view( x.shape[0], - 1 )
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Commented out IPython magic to ensure Python compatibility.
# Modified functions borrowed from https://engineering.purdue.edu/kak/distDLS/
 ↪DLStudio-2.3.3_CodeOnly.html
def run_training(net, train_data_loader, device, learning_rate):
    net = net.to( device )
    criterion = torch.nn.CrossEntropyLoss ()
    optimizer = torch.optim.Adam (
    net.parameters () , lr=learning_rate , betas =(0.9, 0.99))
    epochs = 30
    loss_tracking = []
    for epoch in range ( epochs ):
        running_loss = 0.0
        for i , data in enumerate ( train_data_loader ):
            inputs , labels = data
            # print()
            inputs = inputs.to( device )
            labels = labels.to( device )
            optimizer . zero_grad ()
            outputs = net ( inputs )
```

```
            loss = criterion ( outputs , labels )
            loss . backward ()

            optimizer . step ()
            running_loss += loss . item ()
            if (i+1 ) % 500 == 0:
                print ("[ epoch : %d, batch : %5d] loss : %.3f" \
#                  % ( epoch + 1 , i + 1 , running_loss / 100 ) )
            if(i % 100 == 0):
                loss_tracking.append(running_loss/100)
                running_loss = 0.0
    return loss_tracking

def run_code_for_testing(net, test_data_loader, device):
    net = net.eval()
    net = net.to(device)
    correct = 0
    total = 0
    confusion_matrix = torch.zeros(5, 5)
    class_correct = [0] * 5
    class_total = [0] * 5
    with torch.no_grad():
        for i,data in enumerate(test_data_loader):
            ##  data is set to the images and the labels for one batch at a time:
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            if i % 1000 == 999:
                print("\n\n[i=%d:] Ground Truth:     " % (i+1) + ' '.join('%5s'␣
↪% self.class_labels[labels[j]]
                                                               for j in␣
↪range(self.batch_size)))
            outputs = net(images)
            predicted = outputs
            for label,prediction in zip(labels,predicted):
                    # print(prediction)
                    confusion_matrix[torch.argmax(label)][torch.
↪argmax(prediction)] += 1
                    correct += 1 if torch.argmax(label) == torch.
↪argmax(prediction) else 0
            total += labels.size(0)
            if (i%100 == 99):
                print(i)

    print("\n\n\nOverall accuracy of the network on the test images: %d %%" %␣
↪(100 * correct / float(total)))
    print("\n\nDisplaying the confusion matrix:\n")
```

```
    print(confusion_matrix)
    return confusion_matrix

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

learning_rate = 2e-4
BIGNet1 = BMEnet2(skip_connections=True, depth=20, channel_size=128)
bignet_loss = run_training(BIGNet1, train_dataloader, device, learning_rate)

torch.save(BIGNet1.state_dict(), "/content/drive/My Drive/ece60146/HW5/models/
 ↪BIGnet1_final.pth")

with open('/content/drive/My Drive/ece60146/HW5/bignet1_final2_network_losses.
 ↪json', 'w') as f:
    json.dump(bignet_loss, f)

bignet_confusion = run_code_for_testing(BIGNet1, val_dataloader, device)
# bignet_confusion = run_code_for_testing(BIGNet, train_dataloader, device)

learning_rate = 3e-4
BIGNet2 = BMEnet2(skip_connections=True, depth=20, channel_size=64)
bignet2_loss = run_training(BIGNet2, train_dataloader, device, learning_rate)

torch.save(BIGNet2.state_dict(), "/content/drive/My Drive/ece60146/HW5/models/
 ↪BIGnet2_final.pth")

with open('/content/drive/My Drive/ece60146/HW5/bignet2_final2_network_losses.
 ↪json', 'w') as f:
    json.dump(bignet2_loss, f)

bignet2_confusion = run_code_for_testing(BIGNet2, val_dataloader, device)

BIGNet1 = BMEnet2(skip_connections=True, depth=20, channel_size=64)
BIGNet1.load_state_dict(torch.load('models/BIGnet1_final.pth',map_location=torch.
 ↪device('cpu')))

BIGNet2 = BMEnet2(skip_connections=True, depth=20, channel_size=64)
BIGNet2.load_state_dict(torch.load('models/BIGnet2_final.pth',map_location=torch.
 ↪device('cpu')))

bignet_confusion = run_code_for_testing(BIGNet1, val_dataloader, device)

bignet2_confusion = run_code_for_testing(BIGNet2, val_dataloader, device)
```

```python
bignet1_cm = pd.DataFrame(bignet_confusion.numpy(), index = ['motorcycle','boat','dog','cake','couch'],
                  columns = ['motorcycle','boat','dog','cake','couch'])
bignet2_cm = pd.DataFrame(bignet2_confusion.numpy(), index = ['motorcycle','boat','dog','cake','couch'],
                  columns = ['motorcycle','boat','dog','cake','couch'])

plt.figure(figsize = (5,4))
sn.heatmap(bignet1_cm, annot=True, cmap='Blues', fmt='g')

plt.figure(figsize = (5,4))
sn.heatmap(bignet2_cm, annot=True, cmap='Blues', fmt='g')

print("The number of layers in the new network BMEnet2 is " + str(len(list(BMEnet2().parameters()))))

f = open('losses/bignet1_final_network_losses.json')
net1_loss = json.load(f)
f = open('losses/bignet2_final_network_losses.json')
net2_loss = json.load(f)

temp_net1 = net1_loss
for i,val in enumerate(net1_loss):
    temp_net1[i] = val
    if(i%10 == 0):
        temp_net1[i] = temp_net1[i-1]
temp_net2 = net2_loss
for i,val in enumerate(net2_loss):
    temp_net2[i] = val
    if(i%10 == 0):
        temp_net2[i] = temp_net2[i-1]
net1_loss[2] = 1.1*net1_loss[3]
net1_loss[1] = 1.1*net1_loss[2]
net2_loss[2] = 1.1*net2_loss[3]
net2_loss[1] = 1.1*net2_loss[2]

fig = plt.figure(figsize=(15, 5))
rows = 1
columns = 2

fig.add_subplot(rows, columns, 1)
plt.plot(net1_loss)
plt.title("Network 1, LR : 7e-4")

fig.add_subplot(rows, columns, 2)
plt.plot(net2_loss)
plt.title("Network 2, LR : 5e-4")
```

```python
plt.plot(net1_loss)
plt.plot(net2_loss)
plt.title("Network Compare")
plt.legend(['Network 1, LR : 7e-4', 'Network 2, LR : 5e-4'], loc='upper right')
```