BME646 and ECE60146: Homework 4

# Mykhailo Tsysin

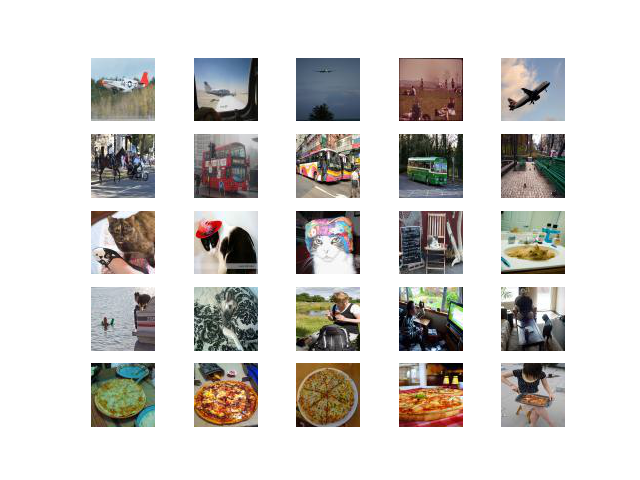
# Feb 20, 2023

In this homework, we will look more closely into how to design a convolutional neural network and how various aspects change the performance. For a full .py solution, please look at the corresponding file.

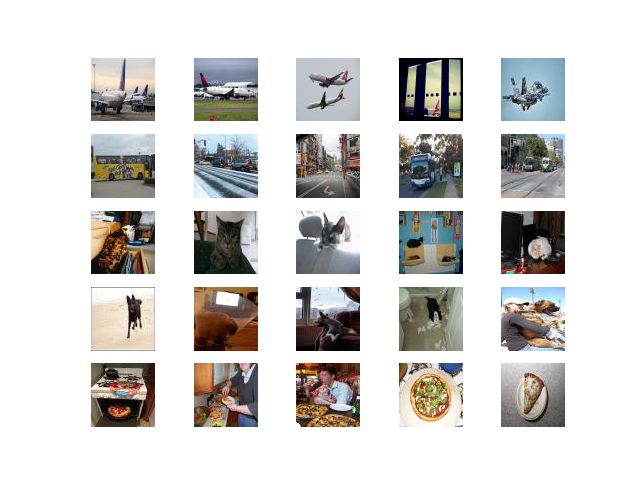
# Task 3.1

We incorporate some dataset downloading logic into the constructor of the dataset. If the “download” flag is turned on, the constructor will erase everything from the data folder and re-download random pictures from the COCO dataset. The “verify” flag serves as a sanity check if the correct number of images is downloaded. The downloaded dataset is then organized into folders where each folder corresponds to a specific class. Also, the “exclusive” flag ensures that we don’t download images with overlapping classes (for example, something like a dog flying an airplane while eating a pizza). That means that each image contains only one of the classes of interest. The output of our dataset loader looks like this:

Training:



Validation:



Since images are resized to 64x64, sometimes it’s hard to see what’s in the picture even for a human. Especially for dogs and cats.

The code for the dataset is:

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

def \_\_init\_\_ (self,

root,

categories\_list,

num\_train = 1500,

num\_val = 500,

train = True,

exclusive = True,

clear = False,

transform = None,

augmentation = None,

download = False,

verify = False,

):

super ().\_\_init\_\_()

# Obtain meta information (e.g. list of file names)

# Initialize data augmentation transforms, etc.

self.transform = transform

self.augmentation = augmentation

self.root = root

self.num = num\_train if train else num\_val

self.categories\_list = categories\_list

if download:

# Download dataset

# Get COCO object

self.dataType='train2014' if train else 'val2014'

annFile='{}/annotations/instances\_{}.json'.format(self.root,self.dataType)

coco=COCO(annFile)

self.catIds = coco.getCatIds(catNms=categories\_list)

# If clear, clear the dataset

if clear:

if os.path.exists(os.path.join(self.root, "data", "train" if train else "val")):

shutil.rmtree(os.path.join(self.root, "data", "train" if train else "val"))

# Create necessary folder structure

for cat in self.categories\_list:

path = os.path.join(self.root, "data", "train" if train else "val", cat)

if not os.path.exists(path):

os.makedirs(path)

# Download images in the dataset

for cat in self.categories\_list:

catIds = coco.getCatIds(catNms=cat)

print("CAT IDs ", catIds)

imgIds = coco.getImgIds(catIds=catIds)

print("ids ", len(imgIds))

random.shuffle(imgIds) # Load in random order

images = coco.loadImgs(imgIds)

count = 0 # count of downloaded images

im\_iter = iter(images)

# print(len(images))

if exclusive:

catIds\_unacceptable = [x for x in self.catIds if x != catIds[0]]

while count < self.num:

im = next(im\_iter, -1)

if im == -1:

raise StopIteration("We've ran out of images")

# Check for exclusivity:

if exclusive:

annIds = coco.getAnnIds(imgIds = [im['id']])

anns = coco.loadAnns(annIds)

not\_exclusive = False

for ann in anns:

if ann["category\_id"] in catIds\_unacceptable:

not\_exclusive = True

break

if not\_exclusive: # image has 2 or more categories from the list

continue

im\_pil = self.get\_img\_from\_url(im['coco\_url'])

save\_path = os.path.join(self.root, "data", "train" if train else "val", cat, im['file\_name'])

im\_pil.save(save\_path)

count += 1

if verify:

"""Verify if the dataset has required number of pictures and that the number is"""

path = os.path.join(self.root, "data", "train" if train else "val")

categories\_in\_folder = [cat for cat in os.listdir(path) if not cat.startswith('.')]

for cat in tqdm.tqdm(categories\_in\_folder):

if cat not in self.categories\_list:

raise ValueError(f"Unknown category {cat}")

if len(os.listdir(os.path.join(path, cat))) != self.num:

raise ValueError(f"Wrong number of pictures: {len(os.listdir(os.path.join(path, cat)))}")

if os.path.isfile(os.path.join(path, cat)):

raise ValueError("Uncategorized files")

for cat in tqdm.tqdm(self.categories\_list):

if cat not in os.listdir(path):

raise ValueError("Some categories are not downloaded!")

print("Verification Successful")

# Now, assuming we have everything downloaded and allocated in folders

self.path = os.path.join(self.root, "data", "train" if train else "val")

self.img\_dict = {cat: os.listdir(os.path.join(self.path, cat)) for cat in self.categories\_list}

self.cat\_idx\_encoding = {i: cat for cat, i in zip(self.categories\_list, list(range(len(self.categories\_list))))}

# filter out system files

# self.imgs = [file for file in files if not file.startswith('.') and file.endswith('.jpg')]

def \_\_len\_\_ (self):

# Return the total number of images

return len(self.categories\_list) \* self.num

def \_\_getitem\_\_ (self, index):

# Read an image at index

# Return the tuple : ( augmented tensor , integer label )

# Get category:

cat\_idx = index // self.num

img\_index = index % self.num

category = self.cat\_idx\_encoding[cat\_idx]

im = Image.open(os.path.join(self.path, category, self.img\_dict[category][img\_index]))

if self.transform:

im = self.transform(im)

im = im.to(dtype=torch.float64)

if self.augmentation:

im = self.augmentation(im)

return im, cat\_idx

@staticmethod

def get\_img\_from\_url(url):

pass

# Download the image from the URL

with urllib.request.urlopen(url) as url\_response:

img\_data = url\_response.read()

# Resize image:

im = Image.open(BytesIO(img\_data))

if im.mode != "RGB":

im = im.convert(mode = "RGB")

im = im.resize((64,64), Image.BOX)

return im

# Task 3.2

The 3 neural network classes are presented here:

class HW4Net(nn.Module):

"""Base neural network form the HW description"""

def \_\_init\_\_(self, out\_classes):

super(HW4Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, 3) # Out size 16 \* 14 \* 14

self.pool = nn.MaxPool2d(2, 2) # Out size 16 \* 7 \* 7

self.conv2 = nn.Conv2d(16, 32, 3) # Out size 32 \* 5 \* 5

self.fc1 = nn.Linear(32 \* 14 \* 14, 64) # Add actual value

self.fc2 = nn.Linear(64, out\_classes)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x))) # Out size 16 \* 62 \* 62 -> # Out size 16 \* 31 \* 31

x = self.pool(F.relu(self.conv2(x))) # Out size 32 \* 29 \* 29 -> # Out size 32 \* 14 \* 14

x = x.view(x.shape[0], -1)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

class HW4Net\_v2(nn.Module):

"""Modification: added padding to the convolutional layers"""

def \_\_init\_\_(self, out\_classes):

super(HW4Net\_v2, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)

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

self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)

self.fc1 = nn.Linear(32 \* 16 \* 16, 64) # Add actual value

self.fc2 = nn.Linear(64, out\_classes)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x))) # Out size 16 \* 62 \* 62 -> # Out size 16 \* 31 \* 31

x = self.pool(F.relu(self.conv2(x))) # Out size 32 \* 29 \* 29 -> # Out size 32 \* 14 \* 14

x = x.view(x.shape[0], -1)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

class HW4Net\_v3(nn.Module):

"""Adding more convo layers + max pool"""

def \_\_init\_\_(self, out\_classes, num\_extra\_layers):

self.num\_extra\_layers = num\_extra\_layers

super(HW4Net\_v3, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)

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

self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)

self.conv3 = nn.Conv2d(32, 32, 3, padding = 1)

self.fc1 = nn.Linear(32 \* 16 \* 16, 64) # Add actual value

self.fc2 = nn.Linear(64, out\_classes)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x))) # Out size 16 \* 64 \* 64 -> # Out size 16 \* 32 \* 32

x = self.pool(F.relu(self.conv2(x))) # Out size 32 \* 32 \* 32 -> # Out size 32 \* 16 \* 16

for \_ in range(self.num\_extra\_layers):

x = F.relu(self.conv3(x)) # Out size 32 \* 16 \* 16

x = x.view(x.shape[0], -1)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

Also, the training and valiadtion loops are as follows:

def train(net, save = False):

# Choose device

if torch.cuda.is\_available()== True:

device = torch.device("cuda:0")

else:

device = torch.device("cpu")

net.train()

# Create transform

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch = 10

train\_dataset = COCODataset(

root=ROOT,

categories\_list=['airplane', 'bus', 'cat', 'dog', 'pizza'],

num\_train=1500,

num\_val=500,

download = False,

verify = True,

train = True,

transform=transform

)

train\_data\_loader = torch.utils.data.DataLoader(dataset = train\_dataset,

batch\_size = batch,

shuffle = True,

num\_workers = 0)

net = net.to(device)

criterion = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(

net.parameters(),

lr=1e-3,

betas=(0.9, 0.99)

)

losses = []

epochs = 10

file\_log = tqdm.tqdm(total=0, position=1, bar\_format='{desc}')

outer = tqdm.tqdm(total=epochs, desc='Epochs', position=0)

for epoch in range(epochs):

running\_loss = 0.0

inner = tqdm.tqdm(total=len(train\_data\_loader), desc='Batches', position=0)

for i, data in enumerate(train\_data\_loader):

inputs, labels = data

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) % 100 == 0:

file\_log.set\_description\_str(

"[epoch: %d, batch: %5d] loss: %.3f" % (epoch + 1, i + 1, running\_loss / 100)

)

losses.append(running\_loss / 100)

running\_loss = 0.0

inner.update(1)

outer.update(1)

if save:

torch.save(net.state\_dict(), ROOT+'/model')

return losses

def val(net):

# Choose device

if torch.cuda.is\_available()== True:

device = torch.device("cuda:0")

else:

device = torch.device("cpu")

net.eval()

# Create transform

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch = 10

val\_dataset = COCODataset(

root=ROOT,

categories\_list=['airplane', 'bus', 'cat', 'dog', 'pizza'],

num\_train=1500,

num\_val=500,

download = False,

verify = True,

train = False,

transform=transform

)

val\_data\_loader = torch.utils.data.DataLoader(dataset = val\_dataset,

batch\_size = batch,

shuffle = True,

num\_workers = 0)

test\_loss, correct = 0, 0

criterion = torch.nn.CrossEntropyLoss()

size = len(val\_data\_loader.dataset)

true\_labels = []

pred\_labels = []

with torch.no\_grad():

for i, data in tqdm.tqdm(enumerate(val\_data\_loader)):

inputs, labels = data

inputs = inputs.to(device)

labels = labels.to(device)

outputs = net(inputs)

test\_loss += criterion(outputs, labels).item()

correct += (outputs.argmax(1) == labels).type(torch.float).sum().item()

pred\_labels.extend(outputs.argmax(1).view(-1).numpy())

true\_labels.extend(labels.view(-1).numpy())

test\_loss /= size #batch

correct /= size

print(f"Test Error: \n Accuracy: {(100\*correct):>0.1f}%, Avg loss: {test\_loss:>8f} \n")

labels = ['airplane', 'bus', 'cat', 'dog', 'pizza']

return confusion\_matrix(true\_labels, pred\_labels)

The first network is a network from the homework description, the second has padding added to the layers, and the 3-rd has 10 more convolutional layers stacked on top of it. The performance is generalized in the following graph (blue – v1, orange – v2, green – v3):Chart, line chart, histogram

Description automatically generated

We see that adding padding to convolutional layers slightly improves convergence. For a very deep network, the training process is a lot slower and seems so saturate at a higher loss. This might be a result of the vanishing gradients problem.

The 3 confusion matrices are presented as well (in order, v1, v2, v3):Table

Description automatically generated with medium confidencev1 – accuracy 57.8%Table

Description automatically generated with medium confidence v2 – accuracy 56.9%Table, calendar

Description automatically generated v2 – accuracy 61.8%

Interestingly, although the training loss on the “deep” network was larger, it generalizes better, giving higher accuracy on the validation dataset. But when the number of epochs increases (20-30), the performance of the third network deteriorates.

All models struggle to differentiate between cats and dogs. As mentioned before, even for me it’s hard to see it on a 64x64 image. We need finer features for improved performance.

To make the classification performance better, we can:

1. Increase image resolution
2. Improve the networks, for example, add batch normalization layers.