Assignment 2

Student: Fabian Gobet; Student's email: gobetf@usi.ch

November 15, 2023

1 Image classification using CNNs [90/100]

1.1 Data (20 pts)

1. In order to see one image and the frequency distribution per class in both the train and test set, i implemented the following function

```
def histo(dataset, set_name):
    labels = dataset.classes
    counts = [0 for _ in labels]
    dic = {c:None for c in labels}

    for img, lbl in dataset:
        if dic[labels[lbl]] is None:
            dic[labels[lbl]] = img
        counts[lbl] = counts[lbl]+1

    fig = plt.figure(figsize=(10,5))
    fig.suptitle(set_name+" class image sample")
    for i in range(1,11):
        ax = fig.add_subplot(2,5,i)
        ax.imshow(dic[labels[i-1]])
        ax.axis('off')
        ax.set_title(labels[i-1])
    plt.savefig(set_name+"_image_per_class")
    plt.show()

    fig, ax = plt.subplots(figsize=(10,5))
    fig.suptitle(set_name+" frequency per class")
    plt.bar(dataset.classes, counts)
    plt.xticks(fontsize=8)
    plt.ylabel("frequency")
    plt.savefig(set_name+"_histogram")
    plt.show()
```

In the previous function I begin by extracting all classes in the dataset into 'labels'. This list will be used to give a legend to each image. I then iterate through the dataset, saving at least one image per class on a dictionary and also counting the number of images in each class. This then enables me to plot all the images with their class label, as well as to generate an histogram with the frequency of each class. The images are saved into the workspace, therefore for each dataset 2 images are generated: one containing the histogram and the other one image per class.

To load the classes according to pytorch documentation I used the following code

```
dataset_train = torchvision.datasets.CIFAR10(root='./
    data', train=True, download=True)
```

```
dataset_test = torchvision.datasets.CIFAR10(root='./data
    ', train=False, download=True)
```

Notice that setting train parameter to false returns a test set of lower dimension in comparison to the train set.

Having done this, we can use the previous function to generate the information we are looking for, as such I execute

```
histo(dataset_train, "Cifar10 train set")
histo(dataset_test, "Cifar10 test set")
```

whereas the images are

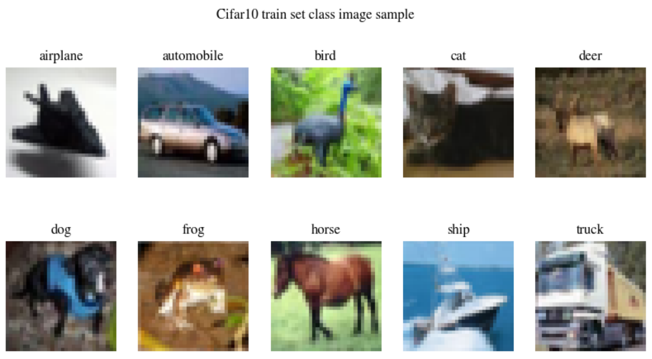


Figure 1: Exercise 1.1.1, train set images

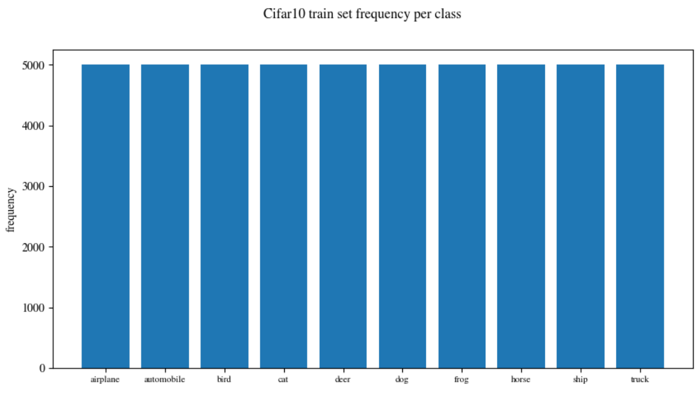


Figure 2: Exercise 1.1.1, train set histogram

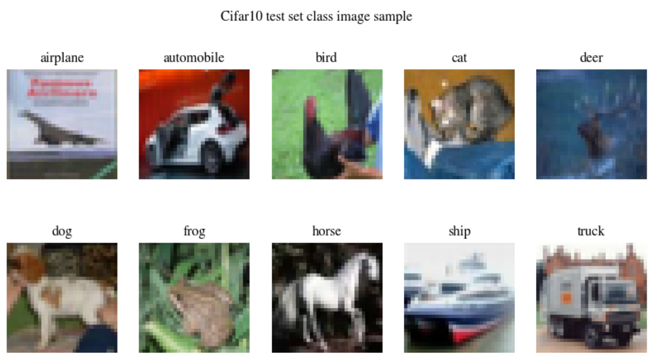


Figure 3: Exercise 1.1.1, test set images

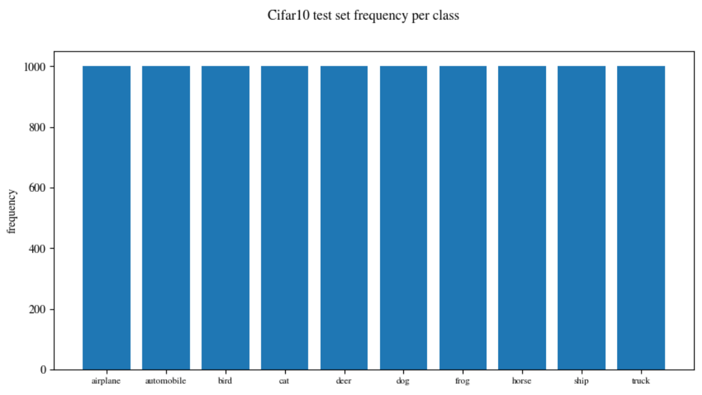


Figure 4: Exercise 1.1.1, test set histogram4

We can see by observation of the histograms that the train has is composed of 40000 images divided by 4000 per class. On the other hand the test set is composed of 1000 images divided by 100 per class.

2. By executing the code

```
sample = dataset_train[0]
print(f"\n1.1.2: dataset_train[0].shape -> type: {type(
    sample)}")
for i in sample:
    print(f"1.1.2: type(dataset_train[0]) components ->
        type: {type(i)}")
```

We can see that each element is a 2-tuple composed of an PIL image and and integer.

Because transforms are only applied to the images once they are acessed, we can define the transformer for the dataset as follows

```
transform1 = transforms.ToTensor()
dataset_train.transform = transform1
dataset_test.transform = transform1
```

Thus running again the the previous code with an addition of

```
print(f"1.1.2: type(transform1(dataset_train[0][0])) ->
    shape: {i.shape}, dtype: {i.dtype}\n")
```

we see that with get a 2-tuple with a torch float32 tensor of dimensions (3,32,32) and an int, arriving to a suitable format. Furthermore, the first dimension with size 3 are the RGB channels, whereas the second and third of size 32 are the width and height in pixels, respectively.

3. Just like before, we may reassign the transform to be applied for the datasets without worrying about overlapping because these are only applied when an element is accessed. Because all the images derive from a PIL file, each pixel value in a channel is in the range of [0,1]. Thus, subtracting 0.5 and dividing by 0.5 will make it so that each element is now in a range of [-1,1], i.e. in a format where the mean is 0 and the standart deviation 1.

   In order to apply both the normalization and tensor conversion, we can compose transformations into a single block as follows

   ```
   transform2 = transforms.Compose([transforms.ToTensor(),
       transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
   dataset_train.transform = transform2
   dataset_test.transform = transform2
   ```

4. To split the dataset we may use the function provided by the *sklearn.model_selection* library, by defining the test_size parameter as 0.2, which means the test size will have 20% of the data that comes form the dataset input argument. We can also enable shuffle so that batches have a variety of classes for images. The code to achieve this is

   ```
   dataset_train, dataset_val = train_test_split(
       dataset_train, test_size=0.2, random_state=42,
       shuffle=True)
   ```

   By executing the previous verification code we can see that the integrity of the elements in the generated datasets is maintained.

1.2 Model (10 pts)

I decided to start with a CNN fairly simple and enough to achieve 70% accuracy on the test set and then work on it from there. My initial CNN is composed of 3 convolutional layers, each of these followed by a MaxPool, followed by 2 fully connected layers. The dimensions and parameters regarding each layer are as follows:

1. Convolution, in=3, out=16, kernel=3, stride=1, padding=1
2. RelU
3. MaxPool, kernel=2, stride=2

4. Convolution, in=16, out=32, kernel=3, stride=1, padding=1
5. RelU
6. MaxPool, kernel=2, stride=2
7. Convolution, in=32, out=64, kernel=3, stride=1, padding=1
8. RelU
9. MaxPool, kernel=2, stride=2
10. Linear, in=1024, out=500
11. RelU
12. Linear, in=500, out=10

On the convulutions layers, 'in' and 'out' are with respect to the number of channels, whereas in the Linear (fully connected) it's the number of inputs and outputs. It is worth noting that the third convolution outputs a tensor of dimensions (64,4,4), which is then stretched into a single dimension of size 1024. The code for the implementation is as follows

```
class myNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, 1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, 1, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, 1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2),stride
            =2)
        self.fc1 = nn.Linear(4*4*64, 500)#,bias=False)
        self.fc2 = nn.Linear(500, 10)#,bias=False)

    def forward(self,x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool1(x)
        x = F.relu(self.conv3(x))
        x = self.pool1(x)
        x = x.view(-1,4*4*64)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

1.3 Training (60 pts)

1. For the implementation of the training pipeline I defined a function $train()$ that takes in as arguments a model, an optimizer, a loss function, a train dataloader, a validation dataloader, the number of epochs, the total number of calculated steps and an intteger 'n'. The integer 'n' and the max number of steps are solely used to compute the losses and accuracies in an interval manner, then saving and printing them. The function initializes the lists for the train and validation loss and accuracy, and returns

them and the very end. It also returns a list for at which steps are respective to each element of the prior lists. The running list is used to compute running losses and accuracies for the training dataset. As such, the first part of the function is as follows

```
def train(model, optimizer, loss_fn, train_loader,
    val_loader, num_epochs, max_steps, n):
  train_acc = []
  val_acc = []
  train_loss = []
  val_loss = []
  step_indices = [0]
  step = 0
  running = [0.0,0,0]
```

We want to to initialize these lists at step 0 with an unbiased loss anc accuracy for both the training and validation dataset. Therefore, as a first step i execute the following code

```
with torch.no_grad():
    model.eval()
    for i, (d,t) in enumerate([next(iter(train_loader)),
        next(iter(val_loader))],0):
      d = d.to(DEVICE)
      t = t.to(DEVICE)
      t_yhat = model(d)
      _, pred = torch.max(t_yhat.data,1)
      corr = (pred == t).sum().item()
      t_loss = loss_fn(t_yhat,t)
      if i==0:
          train_loss.append(t_loss.item())
          train_acc.append(corr/len(t))
      else:
          val_loss.append(t_loss.item())
          val_acc.append(corr/len(t))
```

The previous code removes one batch from the training dataloader and computes the loss and accuracy. It then repeats the process for the validation dataloader. Notice that the whole block of code lies withing a torch.no_grad() statement so the gradients are not accumulated once we start the training. The accuracy is computed by extracting the indexes for the class with maximal value withing each training sample. These indices are then compared to that of the targets and summed. This gives as the number of correct predictions in this computation. This value is then divided by the total number of targets in the batch, thus giving the accuracy factor.

For the actual training, I iterate through the train dataloader and for each batch I repeat the above process, while keeping a running loss and running accuracy aswell as the number of batches processed. A count for the step number is kept, and that count is divisible by 'n', I compute the loss and accuracies attending the number of batches and the 'running' list values. I also compute the validation loss and accuracie in a similar fashion to the previous code. Thus, the remnant part of the code looks as follows

```
loss_acc_message(0,num_epochs,step,max_steps,train_loss,
    val_loss,train_acc,val_acc)
  for epoch in range(1,num_epochs+1):
      for batch_num, (t_data,t_targets) in enumerate(
          train_loader,1):
          step = step + 1
          running[2] = running[2]+1
          model.train()
          t_data = t_data.to(DEVICE)
          t_targets = t_targets.to(DEVICE)
          t_yhat = model(t_data)
          t_loss = loss_fn(t_yhat,t_targets)
          running[0] = running[0] + t_loss.item()
          _, t_pred = torch.max(t_yhat.data,1)
          running[1] = running[1] + (t_pred == t_targets
              ).sum().item()
          t_loss.backward()
          if step%n==0 or step==max_steps:
              step_indices.append(step)
              train_loss.append(running[0]/running[2])
              train_acc.append(running[1]/(running[2]*
                  len(t_targets)))
              running = [0.0,0,0]
              with torch.no_grad():
                  model.eval()
                  for val_data, val_labels in val_loader
                      :
                    val_data = val_data.to(DEVICE)
                    val_labels = val_labels.to(DEVICE)
                    v_yhat = model(val_data)
                    _, v_pred = torch.max(v_yhat.data,1)
                    v_correct = (v_pred == val_labels).
                        sum().item()
                    val_acc.append(v_correct/len(
                        val_labels))
                    v_loss = loss_fn(v_yhat,val_labels)
                    val_loss.append(v_loss.item())
              loss_acc_message(epoch,num_epochs,step,
                  max_steps,train_loss,val_loss,
                  train_acc,val_acc)
          optimizer.step()
          optimizer.zero_grad()
  return train_acc,val_acc,train_loss,val_loss,
      step_indices
```

There are some factors to notice in this code. The optimizer step is only taken once the validation loss and accuracy are computed (if step%n=0) because I want to make sure the validation values are not computed on a different model than the training values.

Another key factor to notice is the use of the function *loss_acc_message*(). This function's purpose is to print the current values of loss and accuracy for training and validation at that step. The implementation of this function is as follows

```
def loss_acc_message(epoch,num_epochs,step,max_steps,
    train_loss,val_loss,train_acc,val_acc):
```

```
print(f"Epoch {epoch}/{num_epochs}, Step {step}/{
    max_steps}")
print(f"Train -> accuracy: {train_acc[-1]*100:.2f},
    loss: {train_loss[-1]:.4f}")
print(f"Validation -> accuracy: {val_acc[-1]*100:.2f},
     loss: {val_loss[-1]:.4f}\n")
```

2. Before training the model, it is necessary to define the batch size, the loss function, initial learning rate and the maximum number of steps. It is also necessary to move all the sets into a dataloader, where the training dataset will be batched and the others wont (maximal batch size).

   To achieve this I executed

```
batch_size = 32
num_epochs = 10
learning_rate = 1e-3
weight_decay = 0
max_steps = -((-len(dataset_train)*num_epochs)//
    batch_size)
n = max_steps//25
```

   Notice that setting 'n' as the division of *max_steps* by 25 means that we will record 25 times the losses and accuracies.

   Moving onto the definition of the dataloaders we have

```
train_loader = DataLoader(dataset=dataset_train,
    batch_size=batch_size, num_workers=2, shuffle=True)
val_loader = DataLoader(dataset=dataset_val, batch_size=
    len(dataset_val), num_workers=2)
test_loader = DataLoader(dataset=dataset_test,
    batch_size=len(dataset_test), num_workers=2)
```

   Because *val_loader* and *test_loader* will be maximal in batch size, we define the batch as the length of the respective dataset. Furthermore, the train dataset is in a dataloader with shuffle to achieve sampling diversity.

   For the optimizer I chose Adam and for the loss function the Cross Entropy Loss because we are solving a classification problem.

   This project was developed in Google Colab, therefore the device is set to 'cuda:0' with an if condition in case it is not available. The code with respect to this segment is as follows

```
DEVICE = torch.device("cuda:0" if torch.cuda.
    is_available() else "cpu")
model = myNet().to(DEVICE)
optimizer = optim.Adam(model.parameters(), lr=
    learning_rate, weight_decay=weight_decay)
loss_fn = nn.CrossEntropyLoss()
```

   In order to train the model, I just have to run the function that was defined as *train*() as follows

```
train_acc, val_acc, train_loss, val_loss, step_indices =
    train(model, optimizer, loss_fn, train_loader,
    val_loader, num_epochs, max_steps, n)
```

The training ends with a training accuracy of 92.57% and a validation accuracy of 78.18%. In order to compute the test accuracy I defined a function called *test_model*() which takes in as arguments the model and the test dataloader as follows

```
def test_model(model,test_loader):
  with torch.no_grad():
    model.eval()
    for test_data, test_labels in test_loader:
      test_data = test_data.to(DEVICE)
      test_labels = test_labels.to(DEVICE)
      test_yhat = model(test_data)
      _, test_pred = torch.max(test_yhat,1)
      test_corr = (test_pred == test_labels).sum().item
          ()
      acc = (test_corr*100)/len(test_labels)
      print(f"Test -> accuracy: {acc:.2f}\n")
```

This code is similar to the previous explained code, so no comments will be added to avoid redundancy.

After executing

```
test_model(model,test_loader)
```

The result comes in as 71.77% accuracy for the test set.

3. To save the parameters of the model I created a function called *checkpoint*() which takes in as arguments the model, the optimizer and a string for the file name (optional argument)

```
def checkpoint(model,optimizer,save_name=None):
  dic = {
      'model_state' : model.state_dict(),
      'optimizer' : optimizer.state_dict()
  }
  if save_name is not None:
    torch.save(dic, save_name+".pt")
  return dic
```

Having implemented this function i executed the following code, effectively creating a checkpoint and saving the model into a .pt file

```
chkp = checkpoint(model,optimizer,"Fabian_Gobet_1")
```

4. In order to plot the losses and accuracies I implemented a function *plot_results*() that takes in as main arguments the list of values for x-axis, both the train and validation y-values, a label for each of these and a title.

```
def plot_results(x,y1,y2,ylabel,label1,label2,title,
    save_name=None,show=False,color1='blue',color2='
    green'):
  fig, _ = plt.subplots(figsize=(7,5))
  fig.suptitle(title)
  plt.plot(x, y1, color=color1, label=label1)
  plt.plot(x, y2, color=color2, label=label2)
  plt.xlabel("steps")
  plt.ylabel(ylabel)
```

```
    plt.legend()
    if(save_name is not None):
        plt.savefig("/content/mydata/"+save_name)
    if(show):
        plt.show()
```

Having this implementation, i executed the following code

```
plot_results(step_indices,train_loss,val_loss,'loss','
    train loss','validation loss','Train/Validation
    losses',show=False, save_name="losses_1")
plot_results(step_indices,torch.tensor(train_acc)*100,
    torch.tensor(val_acc)*100,'% accuracy','train
    accuracy','validation accuracy','Train/Validation
    accuracies',show=False,save_name="accs_1")
```

thus generating the following images

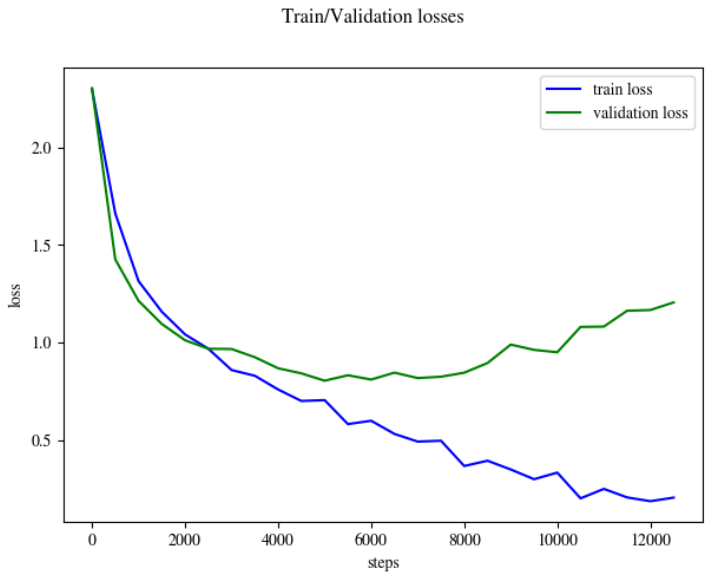


Figure 5: Exercise 1.3.4, Losses during training

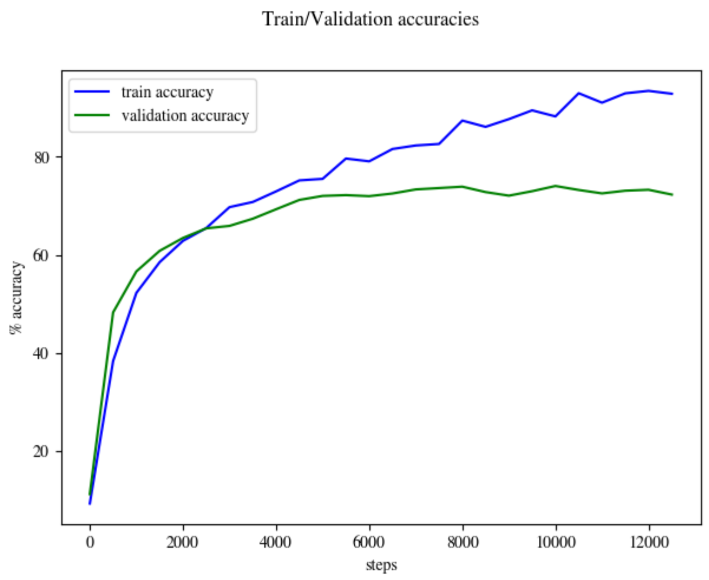


Figure 6: Exercise 1.3.4, Accuracies during training

By looking at the losses plot there is a clear classic sign of over-fitting, where the model keeps improving on the training set but starts to get worse on the validation set. This happens because the model is learning the training set too well and fails to generalize.

5. (a) Because the over-fitting of the model seems to be a the major factor to which one should attend, in order to reduce this I tried introducing L2 regularization via the Adam optimizer weight decay parameter. After trying a few different values, 1e-5 seemed to provide the best results, raising accuracy in 10 epochs to a value of 72.89%. Any bigger values of weight decay would lower the ceiling value for accuracy that the model could reach.

   (b) At this point over-fitting was still and issue so I Introduced dropout after the last convulutional layer and after the first dense layer. This reduced over-fitting immensely, but also reduced rate of convergence by a lot. Therefore I increased the number of epochs to 25 and achieved an accuracy on the test set of 74.45

```
...
self.drop1 = nn.Dropout(0.5)
def forward(self,x):
    x = F.relu(self.conv1(x))
    x = self.pool1(x)
    x = F.relu(self.conv2(x))
    x = self.pool1(x)
    x = F.relu(self.conv3(x))
    x = F.Relu(self.pool1(x))
    x = self.drop1(x)
    x = x.view(-1,4*4*64)
    x = F.relu(self.fc1(x))
```

```
        x = self.drop1(x)
        x = self.fc2(x)
        return x
```

(c) Overfitting was still present at this point, so I changed the transform applied to the train and validation dataset, including random horizontal flipping (0.5 probability) and color jitter, raising the accuracy to 75.50%.

```
transform3 = transforms.Compose([transforms.
    RandomHorizontalFlip(),transforms.ColorJitter(
    brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),transforms.Normalize
    ((0.5,0.5,0.5),(0.5,0.5,0.5))])
dataset_train.transform = transform3
```

(d) In order to further decrease over-fitting, I hypothesised that the bias on the model could be preventing generalization, this because bias values accommodate to the training in a less restrictive way than other learnable parameters. Thus, I experimented in with excluding several bias and found that having no bias at all provided the best effect, raising test accuracy to 76.63

```
...
self.conv1 = nn.Conv2d(3, 16, 3, 1, padding=1,bias=
    False)
self.conv2 = nn.Conv2d(16, 32, 3, 1, padding=1,bias=
    False)
self.conv3 = nn.Conv2d(32, 64, 3, 1, padding=1,bias=
    False)
self.pool1 = nn.MaxPool2d(kernel_size=(2,2),stride
    =2)
self.fc1 = nn.Linear(4*4*64, 500, bias=False)
self.fc2 = nn.Linear(500, 10, bias=False)
...
```

(e) At this point I was experiencing a a validation loss that would jump back and forth, indicating that the learning rate might be high. With some experimentation I gained some stability but lost rate of convergence. Hence, for my final decision, at cost of computation power, I introduced batch normalization in the convolutional layers and found a sweet spot for the learning rate at 7e-4. Batch normalization reduces over-fitting and accelerates rate of convergence at expense of computational power. With this change, I reached a final test accuracy of 77.70%.

```
...
self.conv1 = nn.Conv2d(3, 16, 3, 1, padding=1,bias=
    False) # 32 16
self.bn1 = nn.BatchNorm2d(16)
self.conv2 = nn.Conv2d(16, 32, 3, 1, padding=1,bias=
    False) # 16 8
self.bn2 = nn.BatchNorm2d(32)
self.conv3 = nn.Conv2d(32, 64, 3, 1, padding=1,bias=
    False) # 8 4
self.bn3 = nn.BatchNorm2d(64)
...
x = F.relu(self.conv1(x))
x = self.bn1(x)
```

```
x = self.pool1(x)
x = F.relu(self.conv2(x))
x = self.bn2(x)
x = self.pool1(x)
x = F.relu(self.conv3(x))
x = self.bn3(x)
x = F.Relu(self.pool1(x))
x = self.drop1(x)
...
```

The final result regarding losses and accuracy can be seen in the following images

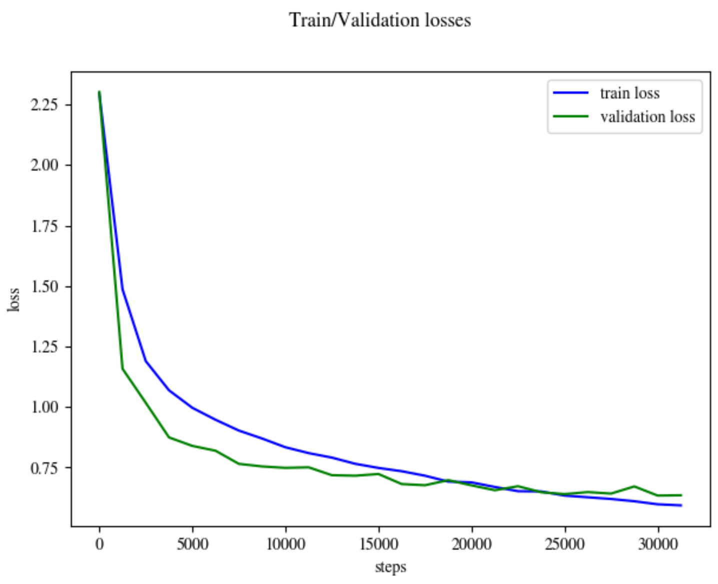


Figure 7: Exercise 1.3.5, Losses during optimized training

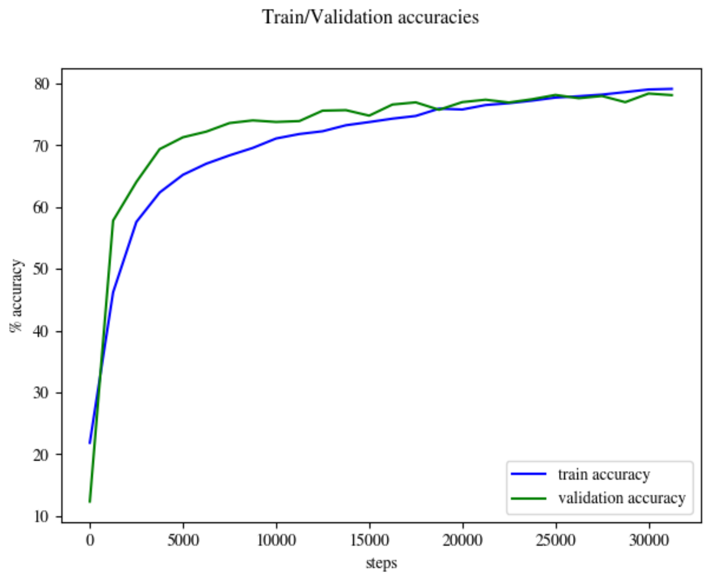


Figure 8: Exercise 1.3.5, Accuracies during optimized training

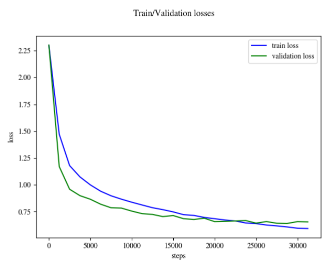
6. For this part i simply repeated the previous code to save a '.pt' file, changing only the name ending to '2'.

1.4 Bonus question* (5 pts)

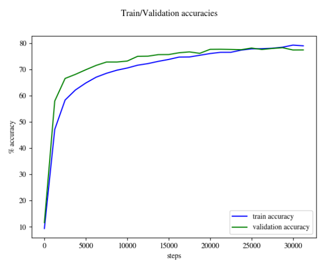
Because the training was defined via a function, for this exercise all it was necessary to do inside the for loop is to redefine the model, set the optimizer, train the model and plot the results. Hence the following code is implemented

```
for seed in range(10):
    torch.manual_seed(seed)
    model = myNet().to(DEVICE)
    optimizer = optim.Adam(model.parameters(), lr=
        learning_rate, weight_decay=weight_decay)
    train_acc,val_acc,train_loss,val_loss,step_indices =
        train(model, optimizer, loss_fn, train_loader,
        val_loader, num_epochs, max_steps, n)
    plot_results(step_indices,train_loss,val_loss,'loss','
        train loss','validation loss','Train/Validation
        losses',show=False, save_name="losses_seed_"+str(seed
        ))
    plot_results(step_indices,torch.tensor(train_acc)*100,
        torch.tensor(val_acc)*100,'% accuracy','train
        accuracy','validation accuracy','Train/Validation
        accuracies',show=False,save_name="accs_seed_"+str(
        seed))
    test_model(model,test_loader)
```

As a result we can observe the following plots

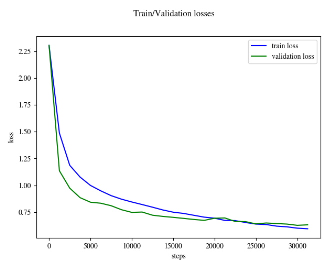


(a) seed 0 losses

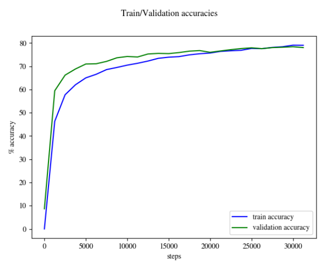


(b) seed 0 accuracies

Figure 9: Seed 0 plots

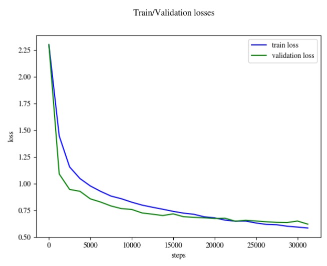


(a) seed 1 losses

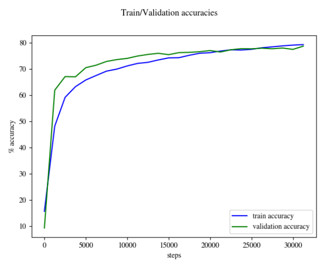


(b) seed 1 accuracies

Figure 10: Seed 1 plots

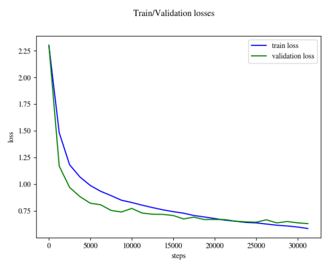


(a) seed 2 losses

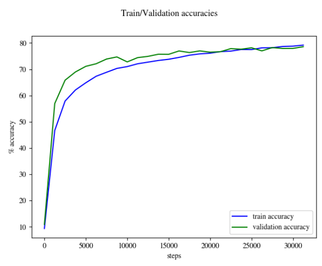


(b) seed 2 accuracies

Figure 11: Seed 2 plots

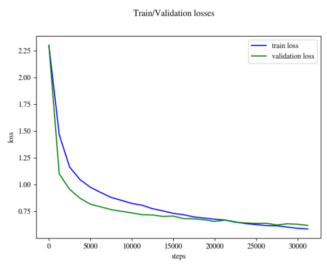


(a) seed 3 losses

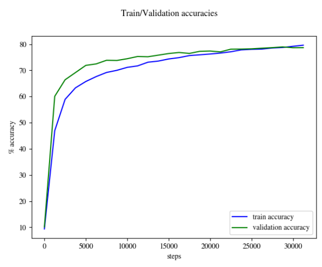


(b) seed 3 accuracies

Figure 12: Seed 3 plots

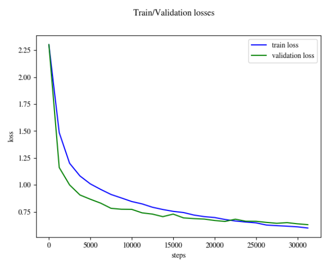


(a) seed 4 losses

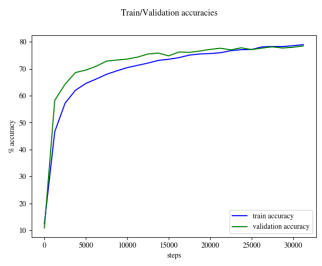


(b) seed 4 accuracies

Figure 13: Seed 4 plots

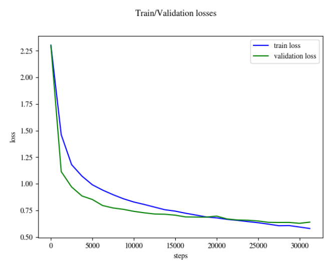


(a) seed 5 losses

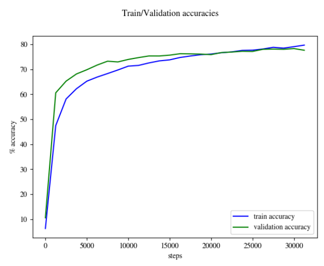


(b) seed 5 accuracies

Figure 14: Seed 5 plots

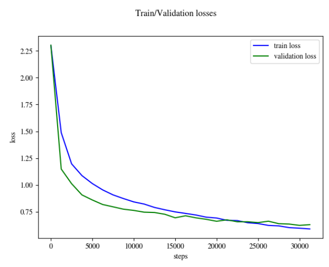


(a) seed 6 losses

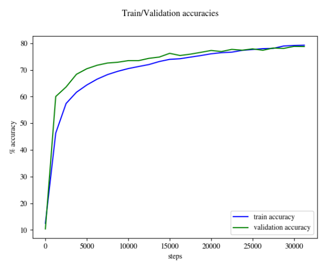


(b) seed 6 accuracies

Figure 15: Seed 6 plots

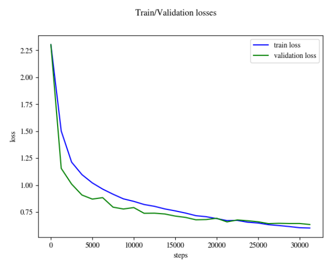


(a) seed 7 losses

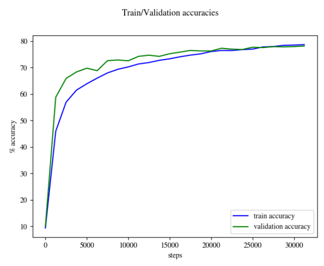


(b) seed 7 accuracies

Figure 16: Seed 7 plots

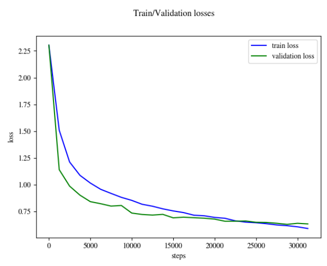


(a) seed 8 losses

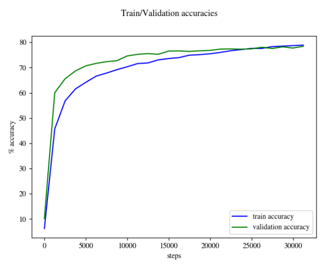


(b) seed 8 accuracies

Figure 17: Seed 8 plots



(a) seed 9 losses



(b) seed 9 accuracies

Figure 18: Seed 9 plots

When comparing two models and deciding which one is better we must not only consider the accuracy (in classification) but how well they generalize given different random states. That is to say, there is some weight pondering that must be done regarding the accuracies and the robustness of the model.

By looking at this experiment, the model stays consistent throughout different random states, which favours the argument that it is robust.

Questions [5 points]

ResNets (Residual Networks) are a type of deep neural network architecture designed to address the vanishing gradient problem in very deep networks. They use residual blocks, where the input of a layer is combined with the output through a shortcut connection, allowing the network to learn residual functions. These shortcut connections, also known as skip connections or identity mappings, enable the gradient to flow directly through the network without encountering diminishing gradients, facilitating the training of very deep networks. ResNets have demonstrated superior performance over traditional Convolutional Neural Networks (ConvNets) in image classification tasks, particularly as the depth of the network increases.