BOĞAZIÇI UNIVERSITY

DEEP LEARNING
CMPE 597

Assignment 2

Author:
Y. Harun Kıvrıl
26 May 2021

Introduction

This project aims to implement a convolutional neural network (CNN) architecture for CIFAR-10 dataset just by using 3 convolutional operations. For this purpose an architecture with given limitation is prepared by reasoning the hyperparameters of the network (ie. kernel size, padding, out channels, number of full connected layers and their size. Firstly the data separated into train, validation and test and data augmentation techniques are applied to the train set. Later, the network trained on train set and validated on the validation set. After that some improvement techniques such as drop out, batch normalization and creating a dense block is implemented and their effect on the test accuracy is observed. After selecting the best performing model architecture, different training algorithms are used to train this architecture and their convergence properties are evaluated. Finally, for the best performing model the latent space from the beginning, middle and end of the training is obtained and plotted into 2D space using TSN-e and the evolution of the class clusters are examined. Lastly, results are presented with a discussion and conclusion at the end.

1 Data Preparation

CIFAR-10[4] dataset consists of 60k 32x32 colored images of ten classes where each class has 6k observation. Following figure shows the classes and examples of the dataset.

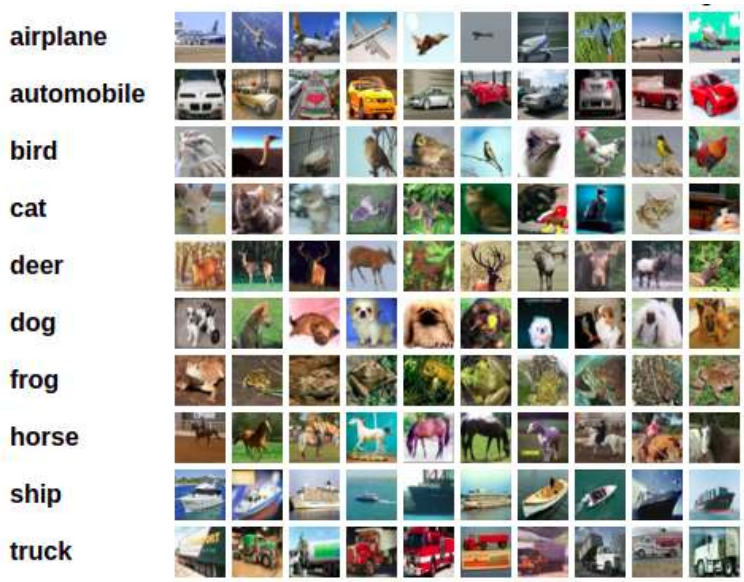

Figure 1: Classes and examples of CIFAR-10 [4] dataset

In the project Pytorch[6] is used as a deep learning framework. Pytorch itself has the functions to load CIFAR-10 dataset. The train and the test set is already separated and 5k image from each

class is in train set and remaining 1k is in test set. However, a validation set does not exists and it is needed for training since the test set will be used to report the performance. Therefore, randomly selected 10% of the train set is used as the validation set and the rest 90% is used in the training. At the end, train set had 45k observation, validation had 5k and test had 10k.

1.1 Data Augmentation

It is desired to train a model such that it is decision is not overfitted to location of the object, the angle it has etc. For a model with good performance, it is better to not to learn this properties and regularize the model by using data augmentation. By regularizing it, the model avoids overfitting and obtains better performance in the test set. Knowing this fact, some data augmentation techniques are introduced only to the train set (since it is the one that will be used in training).

First augmentation used is random cropping, however it is not proper to reduce the size of the images since the crop will not be applied to other sets. So before cropping padding with 4 pixels are made and a cropping of 32x32 is applied without reducing the size or resizing the image. Second one is random horizontal flip. This allows to change the location of the object randomly and ensures that the model does not memorize the location of the objects. The third one is random rotation. Random rotation rotates the images randomly and ensures that the model does not overfit to its alignment. Fourth one is random affine transformation. Actually affine transformation allows to do multiple things at once. In this task shear of 10 degrees and scaling of ± 0.2 is used. This also adds some noise to images to overcome overfitting. Lastly, the images are converted to tensor and each channel is normalized. This last step is also applied to the validation and test sets and since the aim here is not avoiding overfitting but overcome gradient problems.

2 Architecture

The first three step of the architecture is already given in the and they will be convolutional layers with convolution operation and a pooling. For pooling operation maximum pooling is selected since it is better to extract the boundaries of the objects better and it is believed that identifying the boundaries better is preferable in this task. However, there are discussions that says there is not much different between using these two[10]. For the activation function ReLU is used since it is also used in state of the art architectures such as DenseNet. As kernel size 3x3 is used with 1 0-padding and stride 1 which ensures the output image size is the same as input and this property becomes very helpful when making skip connections while creating a dense block from this convolution operations. In addition not reducing the image size is helpful for dealing with the information loss through layers since pooling is also will reduce the image size. For the pooling 2x2

kernel is used with 2 stride. This gets the maximum information of minimum square that has more than one pixel and with 2 stride the same value from a pixel had a chance to be used and cannot be used more than once. It is believed that it is a better way to keep the information.

Another decision to be made is the how many channels to out. This is an important step to control the complexity of the model. Since, the assignment limits the depth of convolutional layers, it makes sense to construct a wide convolutional layers by considering the memory and time limitations. Also powers of two is recommended for the parameters. Therefore, 256 is considered as wide here and (256,128,64) are selected as the number of out channels. There is also another decision here . It is possible to choose (64, 128, 256) or (128, 128, 128) in here by considering the limitation but the choice empirically performed better when the network trained with keeping all other parameters constant.

Another parameters to determine is the number of fully connected layers and their outputs. In here it is possible to have one layer that maps convolutional output to number of classes however this is an opportunity to go deeper. Therefore, the network is trained with 2,3 and 4 full connected layers by keeping other parameters same and it is observed that 3 performs better and 4 does not make much difference. Therefore 3 is selected by considering the training time. Number of outputs of the layers are selected to decrease step by step from the output of convolutional layers to the number of classes by thinking that decreasing the number of features in each layer helps to obtain better representation for 10 classes. As candidates powers of two is considered. Last convolutional layer outputs 1024 features therefore it reduced to 512 first then 128 and finally to 10. The drop out layers also added between fully connected layers however the probabilities are set to 0 initially.

```
Model(
  (conv_layer1): Conv2d(3, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv_layer2): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv_layer3): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1024, out_features=512, bias=True)
  (drop_out1): Dropout(p=0, inplace=False)
  (fc2): Linear(in_features=512, out_features=128, bias=True)
  (drop_out2): Dropout(p=0, inplace=False)
  (fc3): Linear(in_features=128, out_features=10, bias=True)
)
```

Figure 2: Baseline network architecture

3 Training

In order train the model the train dataset provided is shuffled and split into train, validation sets with 0.1 ratio as mentioned in the data preparation section. Later the augmentations are applied to train set and both sets are normalized. As a loss function cross entropy loss is used since the

task is a multi class classification problem. As an optimizer Adam[3] is selected because of its less sensitive to initial learning rate since it contains an adaptive learning rate method. Later a loop over epochs and batches is written. At every step of the batch loop model, loss, zero_grad, backward steps of pytorch is used. At the upper epoch loop after all batches are finished, prediction for validation set is made and for both train and validation for each epoch loss and accuracy values are recorded. During the training the model with best accuracy is tracked and its state dictionary is saved. Also, an early stopping mechanism is introduced and it terminate the training if there is no improvement to best accuracy at last 10 epochs. At the end of the training the parameters and performance metrics are saved as a dictionary and plots for performance metrics are extracted.

Initially, before deciding to the final architecture the batch size is decided to be 128, learning rate to be 0.001 and weight decay is set to 0.0003. It is assumed that by keeping this setup constant, relative comparison over improvement techniques can be made. As a base line the model with no improvement is trained and following results are obtained:

	Train	Validation	Test
Loss	0.64	0.5361	**0.5325**
Accuracy	0.775	0.8014	**0.8138**

Figure 3: Results of baseline model

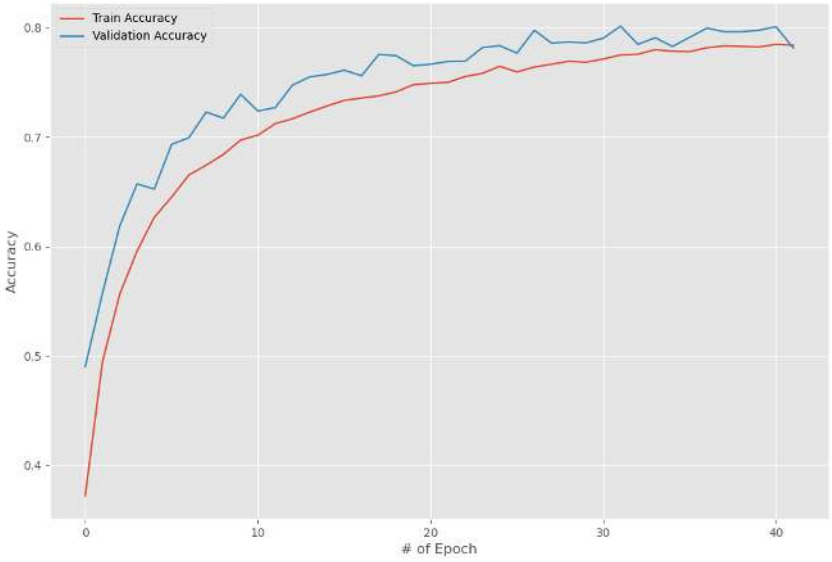

Figure 4: Accuracy plot of baseline model

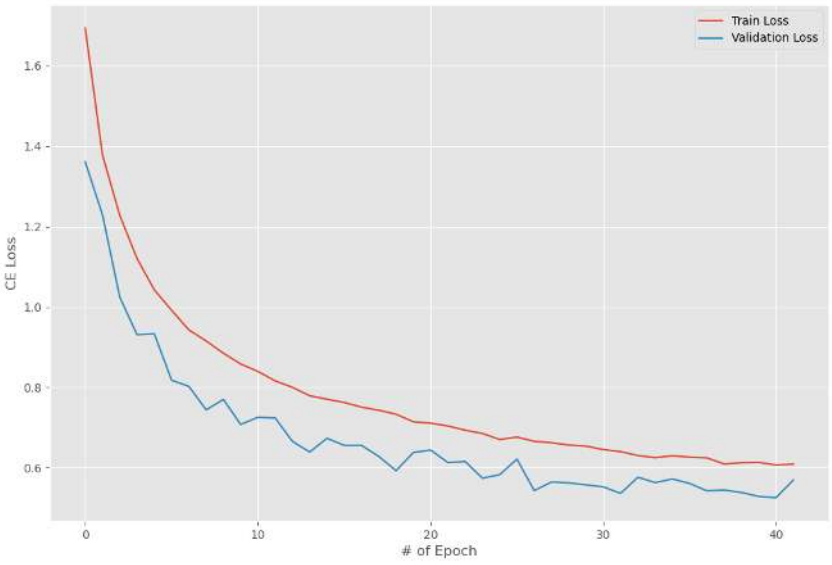

Figure 5: Loss plot of baseline model

Having the validation loss less than the train loss in some plots is considered as a result of using augmentations in train set but not in validation set. it is expected to have a good model if a plot with inverse is observed while using early stopping. Because it means that the model is not just overfitting to augmentations so it can continue to training without getting stopped and it is obtaining good results regardless of the augmentation.

4 Improvement Techniques

In order to improve the model performance 5 different improvement techniques are considered.

- Batch Normalization
- Drop out
- Dense Block
- L2 Regularization
- Spatial Attention

However L2 Regularization is decided to not to be included since there are two other regularization techniques for model performance: data augmentation and drop out. Also, spatial attention is removed from the list since it requires to make use of some convolutional operation and they are limited to three.

4.1 Batch Normalization

Batch normalization has many different advantages. It helps to converge faster, enables use of higher learning rates, it is robust to vanishing/exploding gradients, it has small regularization effect. Therefore, it is expected to obtain some increase in performance by applying batch normalization.

In the architecture 2d batch normalization is placed between convolutional layers and 1d batch normalization is placed between fully connected layers. The network trained with the same hyperparemeters with the baseline model and the following results are obtained:

	Train	Validation	Test
Loss	0.5311	0.4859	**0.4933**
Accuracy	0.8118	0.8142	**0.8263**

Figure 6: Results of model with Batch Normalization

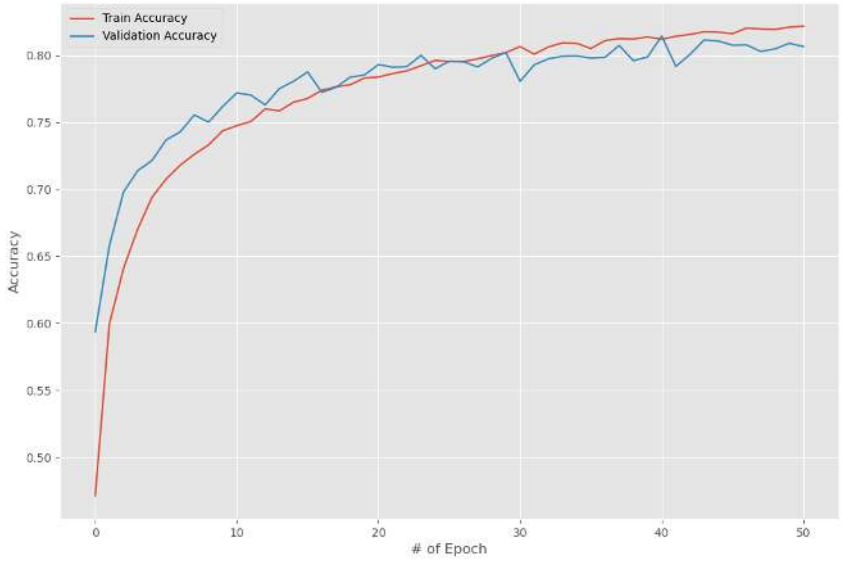

Figure 7: Accuracy plot of model with Batch Normalization

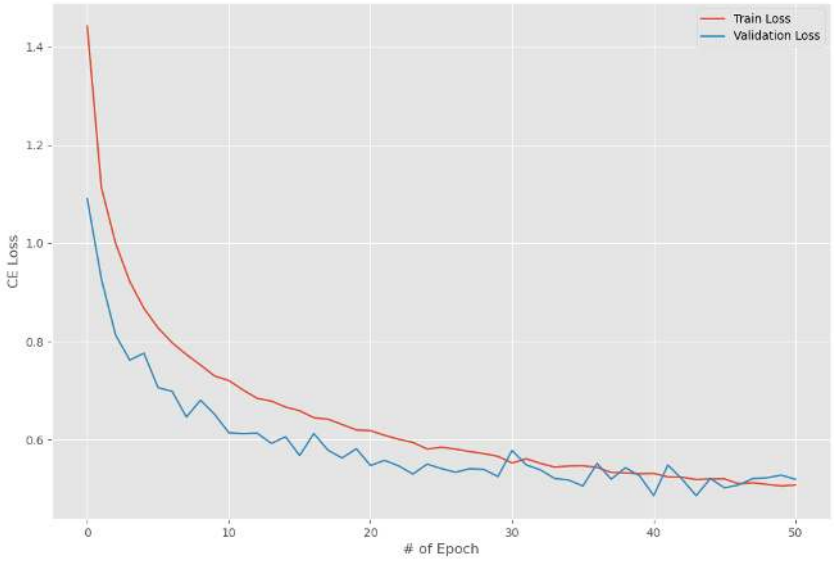

Figure 8: Loss plot of model with Batch Normalization

As the results show the network with batch normalization converged in less number of epochs and obtained better test accuracy.

4.2 Drop out

The dropout removes random connections and it help to regularize the model. Fully connected layers have many connections with each other and dropping some of this many connection may help to increase the test performance.However, since data augmentation is used, not to have excess regularization effect and not to prevent network from training a small dropout ratio is selected and it is decided to increase it if it obtains a performance increase. By setting the drop out probability to 0.2 in baseline model following results are obtained.

	Train	Validation	Test
Loss	0.6728	0.5688	**0.548**
Accuracy	0.766	0.7935	**0.8125**

Figure 9: Results of model with Drop Out p=0.2

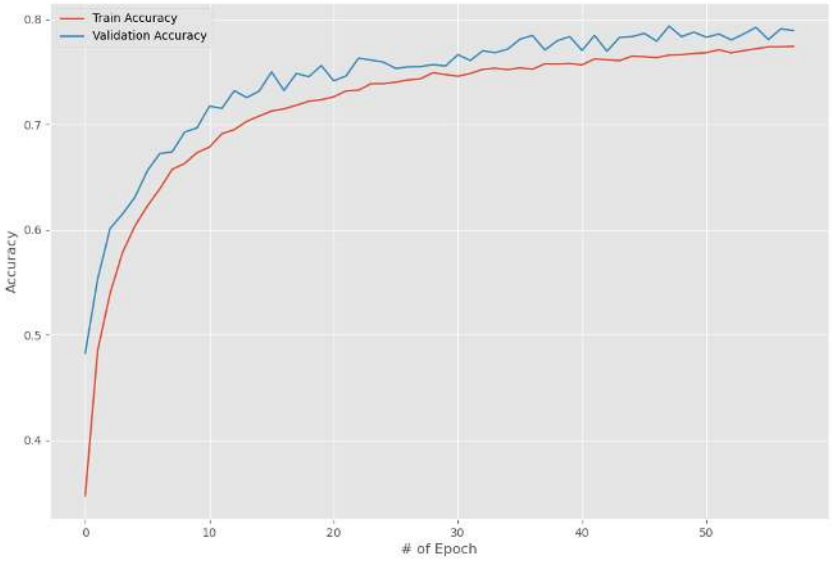

Figure 10: Accuracy plot of model with Drop Out p=0.2

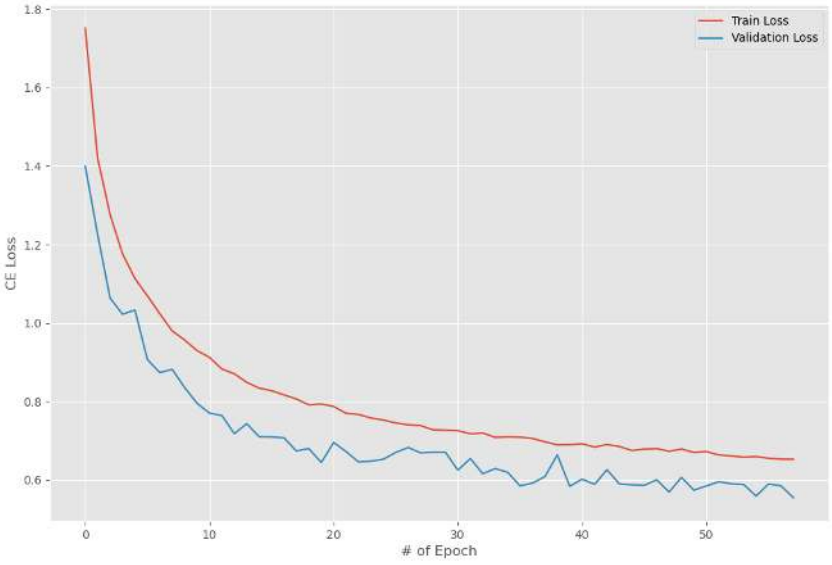

Figure 11: Loss plot of model with Drop Out p=0.2

As the results show there is a decrease in the test accuracy and plots indicate that model is not trained well with the dropout even though the drop out ratio is small. Also the validation and train metrics are not giving confidence. Therefore, it is decided to not to use dropout in this model.

4.3 Dense Block

The aim of using dense blocks are not loosing information as the data propagates deeper through the model. Dense blocks prevents this loss by making connection from previous layers to following layers. It allows to make much deeper networks without loosing performance instead increasing it. However the proposed architecture is not very deep, therefore it is not expected to make the performance better by using a dense block. Also, there could be only one block since the number of convolutional layers are limited.

Implementing dense connections also has a one downside. Adding the results of previous layer's outputs to next one increases the number of parameters dramatically therefore the number of out channels are reduced by factor of two to have reasonable training times. Also the connection to the fully connected layer has much more features when there is dense connections so, the out features of fully connected layers also modified according to that and set to (1024, 128). The dense connections require the output image sizes to be same to concatenate the results later, thus the max pooling layers inside the dense block is removed and only the max pooling after the last convolutional layer is kept. The architecture with this setting is trained and following results are obtained:

	Train	Validation	Test
Loss	0.8047	0.6944	**0.6981**
Accuracy	0.7124	0.7437	**0.7549**

Figure 12: Results of model with dense block

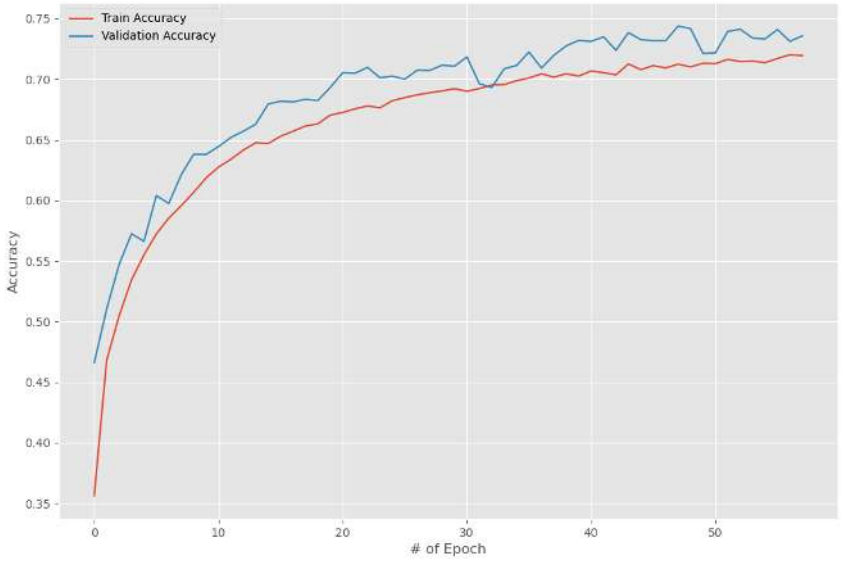

Figure 13: Accuracy plot of model with dense block

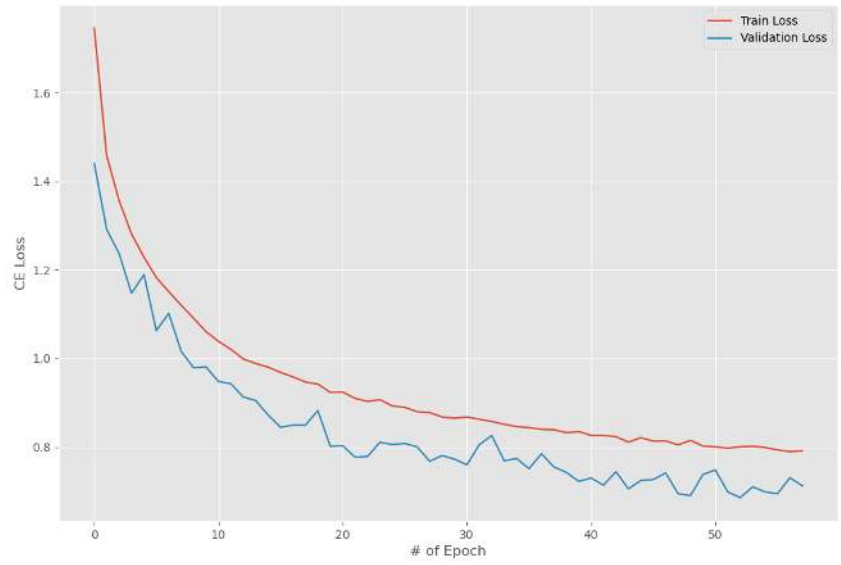

Figure 14: Loss plot of model with dense block

As the results show converting the architecture to a dense block does not helps to obtain better results.

When the results of all improvements are considered only the batch norm is added to baseline for final architecture. For the combination of improvements no better results are obtained empirically.

5 Hyperparameter Tuning

Since the decision of the final architecture is made, only the decision of batch size and learning rate is left. In order to tune this parameters one low one middle and one high value is picked and their test performance is compared. For batch size high value is selected as 256 due to the GPU memory limitations middle is decided as 128 and low is set to 64. For the learning rate 0.01, 0.001 and 0.0001 are selected as candidates. By training the model with each combination following table of validation accuracies is obtained:

LR/ Batch Size	64	128	256
0.01	0.6519	0.6871	0.7131
0.001	0.816	0.8142	0.8207
0.0001	**0.8346**	0.8226	0.8173

Figure 15: Validation accuracies for different training hyperparameters

As a result batch size is selected as 64 and learning rate is chosen as 0.0001.

When the model with this hyperparameters is evaluted on the test data:

$$\textit{Test Accuracy} : 0.8339 \quad \textit{Test Loss} : 0.4666$$

And during the training following plots are obtained:

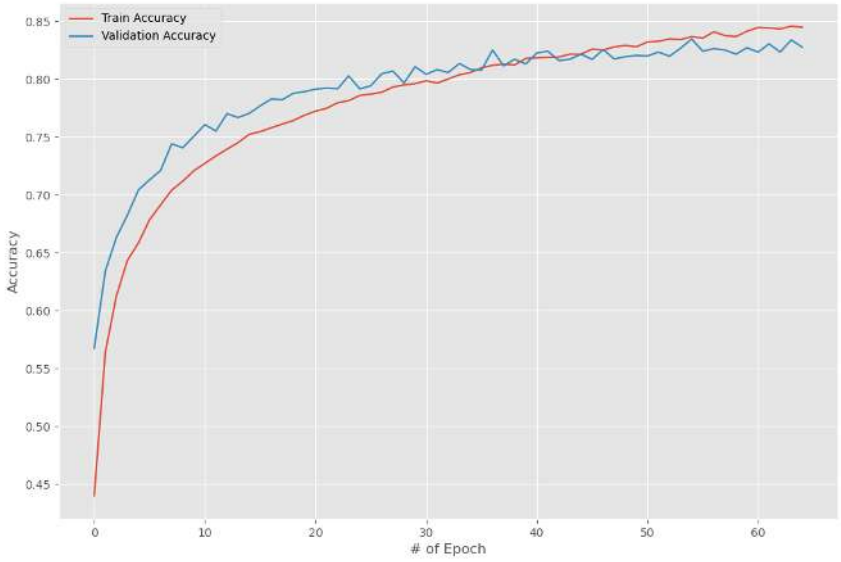

Figure 16: Accuracy plot of the selected model

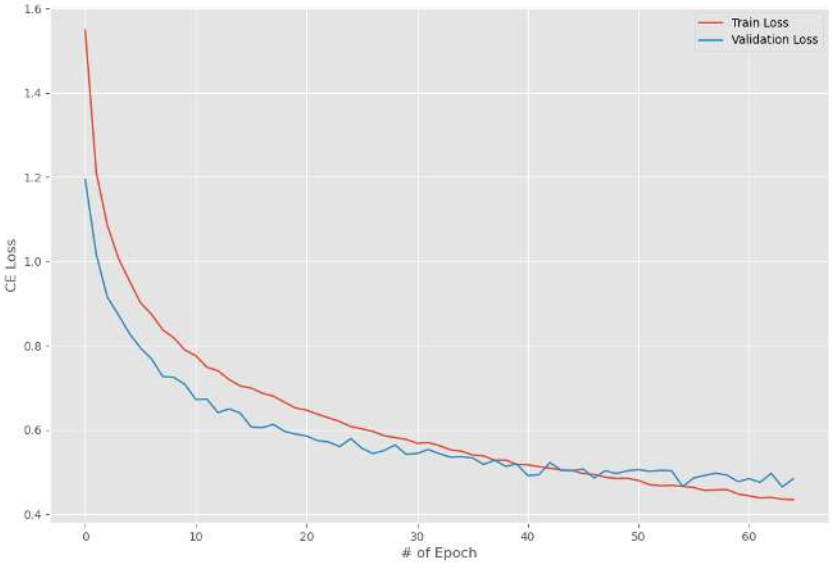

Figure 17: Loss plot of the selected model

6 Training Algorithms

After setting all hyperparameters by training the network with different optimizers, their performance and convergence properties are investigated.

6.1 SGD

Stochastic Gradient Descent algorithm is the most basic algorithm for gradient based training. It basically moves towards to negative of the gradient to minimize the loss. When SDG used in the training of the proposed model following plots are obtained.

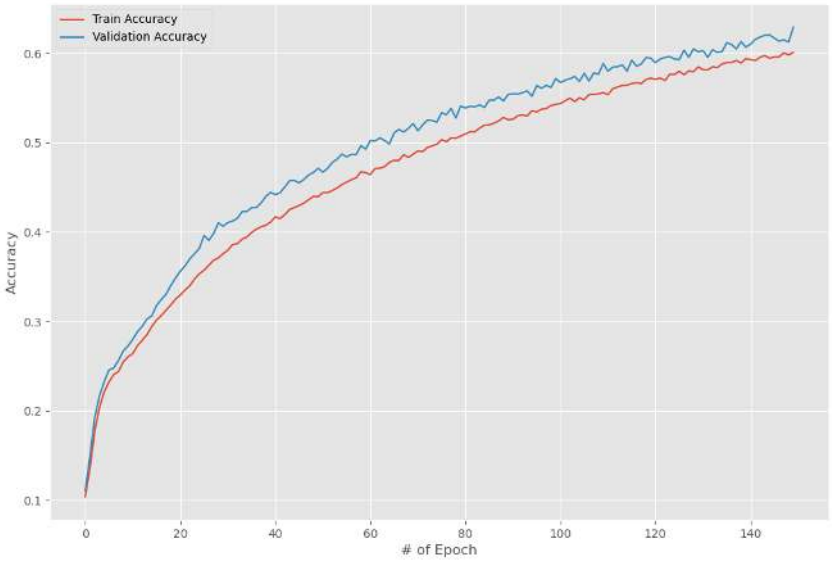

Figure 18: Accuracy plot of the model trained using SGD

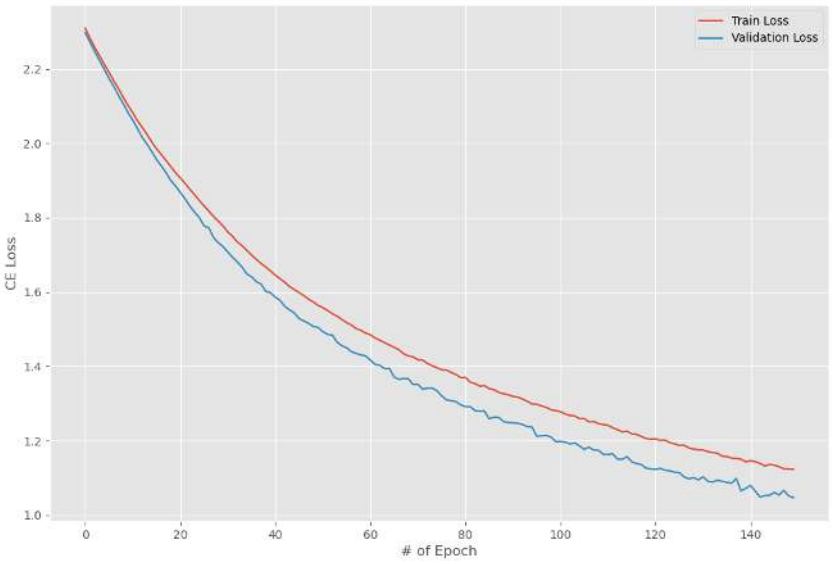

Figure 19: Loss plot of the model trained using SGD

As the plots show SGD converges very slowly it stands around 0.63 accuracy even at the 150th epoch and this training needs to be fastened.

6.2 SGD with Momentum

An improvement to SGD is using momentum. It gives faster convergence and helps to avoid local minimums. As a momentum parameter 0.9 is used.

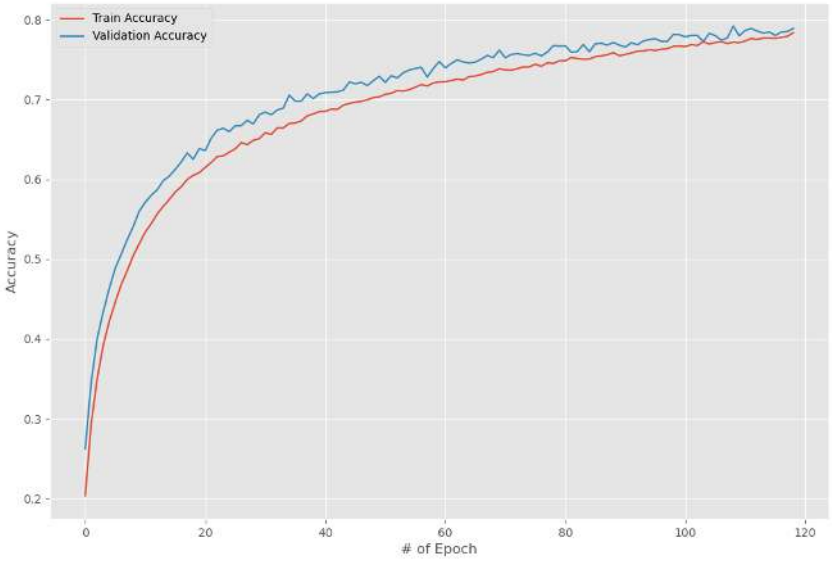

Figure 20: Accuracy plot of the model trained using SGD with 0.9 momentum

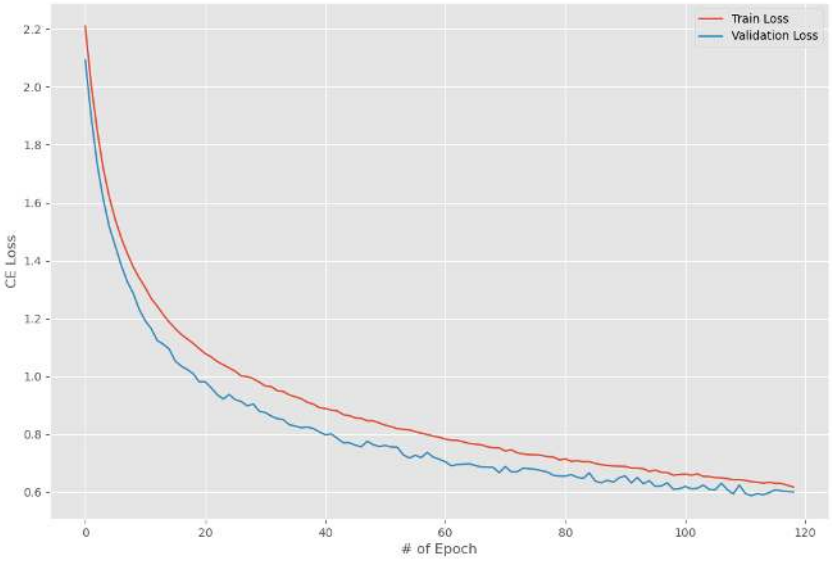

Figure 21: Loss plot of the model trained using SGD with 0.9 momentum

As the plots show when SGD make use of momentum it converges faster however it is still gets to 0.8 accuracy around 120th epoch and early stop stopped it there .

6.3 RMSprop

RMSprop [8] is a popular optimization algorithm. It uses exponentially weighted gradient accumulation to optimize the networks better. By using RMSprop the following plots are obtained.

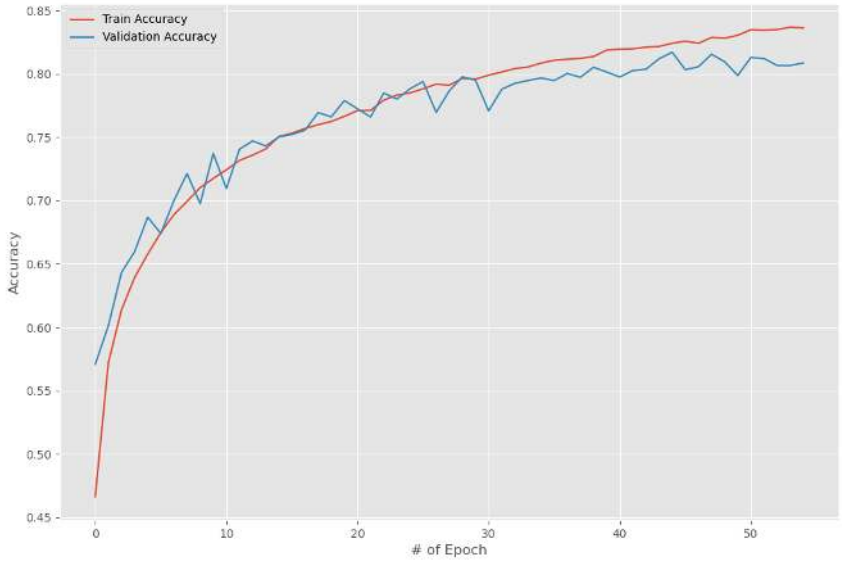

Figure 22: Accuracy plot of the model trained using RMSprop

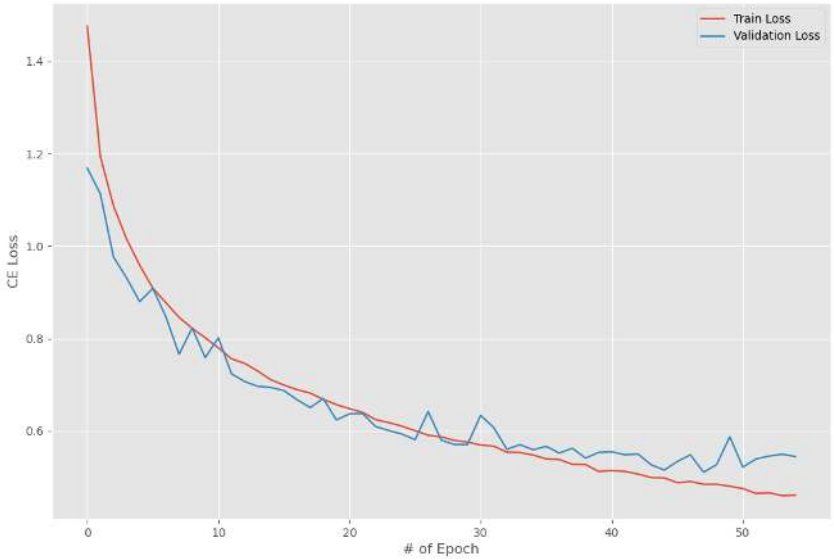

Figure 23: Loss plot of the model trained using RMSprop

As the plots show RMSprop is doing a good job in the training. It converges to high accuracies just like adam but it does not obtain better results than it at the end.

6.4 Adam

Adam uses adaptive learning rate during the training. It makes use of second moment estimates to obtain this adaptive learning rate. It is considered to be more robust to initial learning rate. The selected model also used adam as optimizer therefore its plots can be found under hyperparameter tuning section. It obtained the best performance among the optimizers. This result was expected since adam uses extra improvements to the others.

7 Latent Space

Lastly, for the best performing model the latent state of the classes are plotted with the weights at the begining of the training, at the 20th epoch and at the end of the training. Since the layer before the output layer has 128 dimension, in order to map 128 dimensional features to two dimension TSN-e [9] dimensionality reduction technique is used.

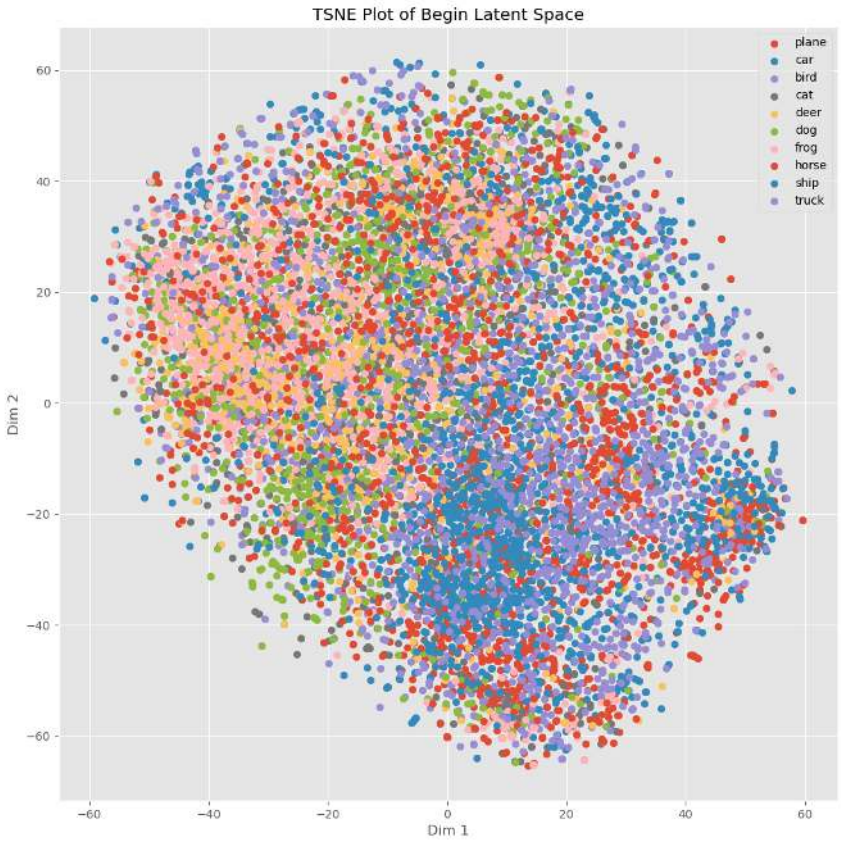

Figure 24: Latent space at the beginning of the training

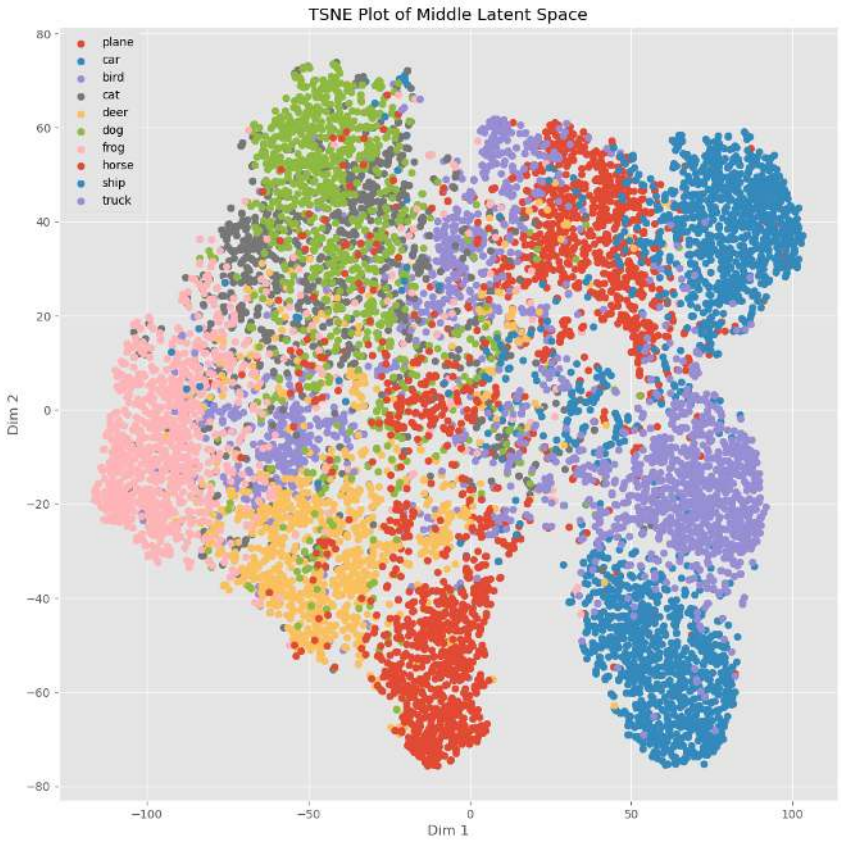

Figure 25: Latent space at the 20th epoch

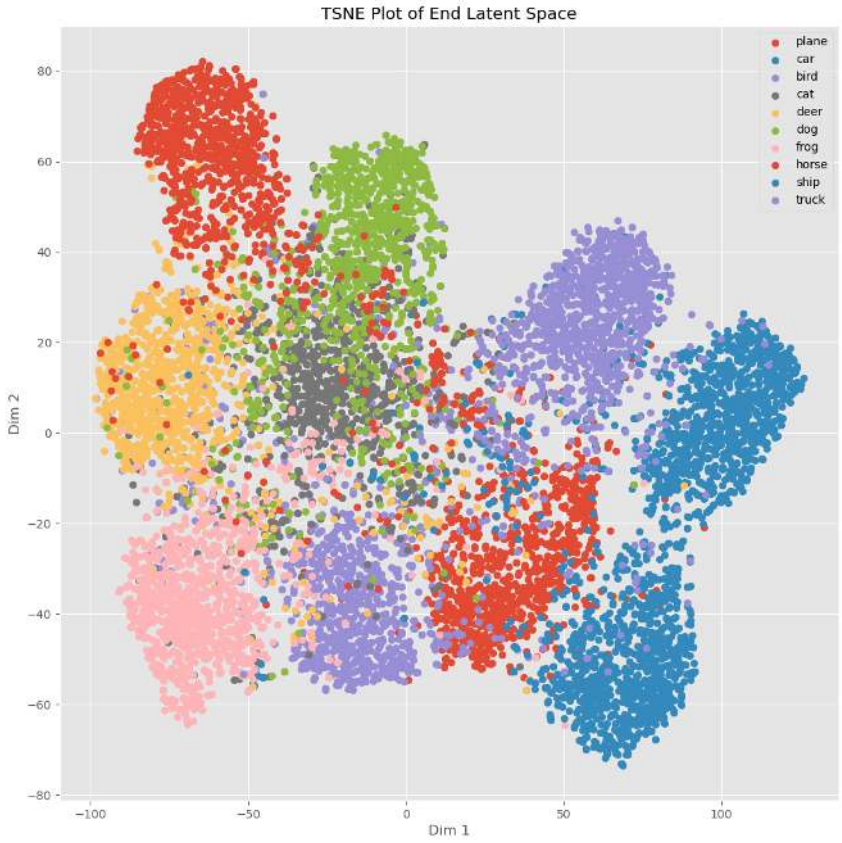

Figure 26: Latent space at the end of the training

The plot of the latent spaces show that the model learns better representations as it is trained and better representations leads to better performance.

8 Discussion & Conclusion

In this assignment a Convolutional Neural Network architecture with only three convolution operation is constructed. While doing that the parameters such as kernel size, number of out channels, number of fully connected layers are selected by reasoning the choice. Later a baseline model is trained with the initial architecture and 0.8148 test accuracy obtained for the base model. After that some improvements over the model is considered. Firstly batch normalization is introduced and more than %1 increase is observed with this improvement. Secondly dropout is used as an addition to baseline model. The drop out couldn't achieve better results and its validation and train metrics also was not giving any trust to this extension. Lastly, convolutional layers are converted to a dense

block, however dense blocks are for going deeper and surely the model is not deep. Having a dense block made training harder and worsen the results. Also no performance increase observed with the combinations of the improvements As a result model with batch normalization selected. After selecting the model the training hyperparameters are tuned and the tuned model obtained 0.8339 accuracy in the test set.

At the second part the hyperparameters for the model are fixed and it has trained with different optimizers. In this part it is observed that adam optimizes this model best with the given conditions. This was also expected since adam has extra improvements to the other algorithms. Rmsprop obtained similar converegence pattern but it obtained a little bit less accuracy. SGD with momentum was not as good as the ones mentioned however it definetly make SGD faster and helps to reach not bad results in 117 epochs but then stopped by early stopping condition. SGD showed very slow performance and proved the necessity of the use of extentions.

At the last part latent space for the best model is plotted at the beginnig, middle and end of the training. The plots showed that as models learns this latent space seperates the classes from each other better and leads to better results.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[5] Michael A. Nielsen. Neural networks and deep learning, 2018.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[9] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[10] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

9 Appendix

9.1 Appendix A (Codes)

model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Model(nn.Module):

    def __init__(self, image_size = 32, num_classes = 10, in_channel = 3, out_channels = (16,32,64),
                 fc_out = (4096, 1024), kernel = 3, pad = 1, stride = 1, pool_kernel = 2, pool_stride = 2,
                 batch_norm=False, drop_out=0, dense=False):
        super().__init__()
        self.image_size = image_size
        self.batch_norm = batch_norm
        self.drop_out = drop_out
        self.kernel = kernel
        self.pad = pad
        self.stride = stride
        self.pool_kernel = pool_kernel
        self.pool_stride = pool_stride
        self.dense = dense

        c1_out, c2_out, c3_out = out_channels

        self.conv_layer1 = nn.Conv2d(in_channel,c1_out,kernel, stride, pad)
        c2_in = c1_out + in_channel if dense else c1_out

        self.conv_layer2 = nn.Conv2d(c2_in, c2_out,kernel, stride, pad)
        c3_in = c2_out + c2_in if dense else c2_out

        self.conv_layer3 = nn.Conv2d(c3_in, c3_out,kernel, stride, pad)
        fc_in_ch = c3_out

        if self.batch_norm:
            self.batch_norm1 = nn.BatchNorm2d(c1_out)
            self.batch_norm2 = nn.BatchNorm2d(c2_out)
            self.batch_norm3 = nn.BatchNorm2d(c3_out)
            self.batch_norm4 = nn.BatchNorm1d(fc_out[0])
            self.batch_norm5 = nn.BatchNorm1d(fc_out[1])

        output_image_size = self._calculate_image_size()
        self.fc1_input_size = fc_in_ch * output_image_size**2
        self.fc1 = nn.Linear(self.fc1_input_size, out_features=fc_out[0])
        self.drop_out1 = nn.Dropout(self.drop_out)
        self.fc2 = nn.Linear(fc_out[0], out_features=fc_out[1])
        self.drop_out2 = nn.Dropout(self.drop_out)
```

```
        self.fc3 = nn.Linear(fc_out[1], out_features=num_classes)


    def forward(self, x):

        c1 = self.conv_layer1(x)
        if self.batch_norm:
            c1 = self.batch_norm1(c1)
        a1 = F.relu(c1)
        if not self.dense:
            mp1 = F.max_pool2d(a1, kernel_size = self.pool_kernel, stride=self.
                pool_stride)
        else:
            mp1 = a1

        in2 = torch.cat([x, mp1],1) if self.dense else mp1
        c2 = self.conv_layer2(in2)
        if self.batch_norm:
            c2 = self.batch_norm2(c2)
        a2 = F.relu(c2)
        if not self.dense:
            mp2 = F.max_pool2d(a2, kernel_size = self.pool_kernel, stride=self.
                pool_stride)
        else:
            mp2 = a2

        in3 = torch.cat([x, mp1, mp2],1) if self.dense else mp2
        c3 = self.conv_layer3(in3)
        if self.batch_norm:
            c3 = self.batch_norm3(c3)
        a3 = F.relu(c3)
        mp3 = F.max_pool2d(a3, kernel_size = self.pool_kernel, stride=self.pool_stride)

        in_fc = mp3

        in_fc = in_fc.view(-1, self.fc1_input_size)
        fc1 = self.fc1(in_fc)
        if self.batch_norm:
            fc1 = self.batch_norm4(fc1)
        fca1 = F.relu(fc1)
        fca1 = self.drop_out1(fca1)
        fc2 = self.fc2(fca1)
        if self.batch_norm:
            fc = self.batch_norm5(fc2)
        fca2 = F.relu(fc2)
        fca2 = self.drop_out2(fca2)
        fc3 = self.fc3(fca2)

        return fc3

    def _calculate_image_size(self):
         #For convolution 1
        output_image_size = (self.image_size + 2*self.pad - self.kernel)//self.stride +
            1
```

```
        # For pooling 1
        if not self.dense:
            output_image_size = (output_image_size - self.pool_kernel)//self.pool_stride
                + 1

        #For convolution 2
        output_image_size = (output_image_size + 2*self.pad - self.kernel)//self.stride
            + 1
        # For pooling 2
        if not self.dense:
            output_image_size = (output_image_size - self.pool_kernel)//self.pool_stride
                + 1

        #For convolution 3
        output_image_size = (output_image_size + 2*self.pad - self.kernel)//self.stride
            + 1
        # For pooling 3
        output_image_size = (output_image_size - self.pool_kernel)//self.pool_stride + 1

        return output_image_size
```

train.py

```
import os

import torch
import torch.nn as nn
from torch.utils.data import sampler
import torchvision
import torchvision.transforms as transforms
from model import Model
from tqdm import tqdm
import pickle
import matplotlib.pyplot as plt
import numpy as np

MAX_EPOCH = 150
early_stop_steps = 10
opt_name = "adam"
batch_size=64
lr = 1e-4

batch_norm=1
dense=0
dropout=0

momentum = 0.9 #For sgd
wd = 3e-4 # For adam

split_ratio = 0.1
seed =3136
torch.manual_seed(seed)

save_folder = f"../models/bs_{batch_size}_lr_{lr}_opt_{opt_name}_bn_{batch_norm}_do_{
    dropout}_dense_{dense}_momentum_{momentum}_3/"
```

```
os.makedirs(save_folder, exist_ok=True)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

transform_train=transforms.Compose([
    transforms.Pad(4),
    transforms.RandomCrop(size=(32,32)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])

transform_test=transforms.Compose([
    transforms.Resize((32,32)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_train)

valset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_test)

n_obs = len(trainset)
indices = list(range(n_obs))
split_idx = int(np.floor(split_ratio * n_obs))
np.random.seed(seed)
np.random.shuffle(indices)

train_idx, valid_idx = indices[split_idx:], indices[:split_idx]
train_sampler = sampler.SubsetRandomSampler(train_idx)
valid_sampler = sampler.SubsetRandomSampler(valid_idx)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=False, sampler=train_sampler, num_workers
                                              =10)
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
                                        shuffle=False, sampler=valid_sampler, num_workers
                                            =10)


if dense:
    fc_out = (1024, 128)
    out_channels = (128,64,32)
else:
    fc_out = (512, 128)
    out_channels = (256,128,64)

model = Model(batch_norm=batch_norm, out_channels=out_channels, drop_out=dropout, dense
    =dense, fc_out=fc_out).to(device)
print(model)
```

```
loss_fn = nn.CrossEntropyLoss()
torch.save(model, save_folder +"model_obj.pkl")

if opt_name == "adam":
    optimizer = torch.optim.Adam(model.parameters(), lr = lr, weight_decay=wd)
elif opt_name =="sgd":
    optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum)
elif opt_name == "rmsprop":
    optimizer = torch.optim.RMSprop(model.parameters(), lr=lr)
else:
    raise ValueError("Optimizer not supported.")


train_losses=[]
train_accuracies=[]
val_losses=[]
val_accuracies=[]
best_val_acc = 0
early_stop_count = 0
for epoch in range(MAX_EPOCH):
    train_loss = 0
    train_corrects = 0
    for images, labels in tqdm(trainloader):
        images = images.to(device)
        labels = labels.to(device)
        out = model(images)
        loss = loss_fn(out, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        _, preds = torch.max(out, 1)

        train_loss += loss.item()
        train_corrects += torch.sum(preds == labels.data)

    val_loss = 0
    val_corrects = 0
    with torch.no_grad():
        for val_images, val_labels in valloader:
            val_images = val_images.to(device)
            val_labels = val_labels.to(device)
            val_out = model(val_images)
            loss = loss_fn(val_out, val_labels)

            _, preds = torch.max(val_out, 1)

            val_loss += loss.item()
            val_corrects += torch.sum(preds == val_labels.data)

    train_loss = train_loss/(len(trainloader))
    train_acc = train_corrects/(len(trainloader)*batch_size)
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
```

```
137
138     val_loss = val_loss/(len(valloader))
139     val_acc = val_corrects/(len(valloader)*batch_size)
140     val_losses.append(val_loss)
141     val_accuracies.append(val_acc)
142
143     if val_acc > best_val_acc:
144         early_stop_count=0
145         best_val_acc = val_acc
146         torch.save(model.state_dict(), save_folder + "best_model.pkl")
147     else:
148         early_stop_count+=1
149
150     if early_stop_count >= early_stop_steps:
151         break
152
153     print(f"""
154     Epoch {epoch}: Train Loss: {train_loss}, Val Loss: {val_loss}
155                   Train Acc: {train_acc}, Val Acc: {val_acc}""")
156
157     if epoch==20:
158         torch.save(model.state_dict(), save_folder + "middle_model.pkl")
159
160 metrics = {}
161 metrics["train_accuracy"] = train_accuracies
162 metrics["validation_accuracy"] = val_accuracies
163 metrics["train_loss"] = train_losses
164 metrics["validation_loss"] = val_losses
165 metrics["batch_size"] = batch_size
166 metrics["learning_rate"] = lr
167 metrics["batch_norm"] = batch_norm
168 metrics["drop_out"] = dropout
169 metrics["momentum"] = momentum
170 metrics["weight_decay"] = wd
171
172 with open(save_folder + "summary.pkl", "wb") as file:
173     pickle.dump(metrics, file)
174
175 plt.style.use("ggplot")
176 plt.figure(figsize=(12,8))
177 plt.plot(list(range(len(train_accuracies))), train_accuracies, label='Train Accuracy')
178 plt.plot(list(range(len(val_accuracies))), val_accuracies, label='Validation Accuracy')
179 plt.xlabel("# of Epoch")
180 plt.ylabel("Accuracy")
181 plt.legend()
182 plt.savefig(save_folder + "accuracy_plot.png")
183
184 plt.figure(figsize=(12,8))
185 plt.plot(list(range(len(train_losses))), train_losses, label='Train Loss')
186 plt.plot(list(range(len(val_losses))), val_losses, label='Validation Loss')
187 plt.xlabel("# of Epoch")
188 plt.ylabel("CE Loss")
189 plt.legend()
190 plt.savefig(save_folder + "loss_plot.png")
```

```

print("Best Acc: ", best_val_acc)
```

eval.py

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from model import Model
import numpy as np
from sklearn.manifold import TSNE
import pickle
plt.style.use("ggplot")

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

plot_latent_space =False
model_folder = "./bs_64_lr_0.0001_opt_adam_bn_1_do_0_dense_0_momentum_0/"

model_path = model_folder + "model_obj.pkl"
state_dict_path = model_folder + "best_model.pkl"
batch_size = 64

transform_test=transforms.Compose([
    transforms.Resize((32,32)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_test)

testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=10)


model = torch.load(model_path).to(device=device)
print(model)
model.load_state_dict(torch.load(state_dict_path))
model.eval()
loss_fn = torch.nn.CrossEntropyLoss()
latent_space_end = []

test_loss = 0
test_corrects = 0
with torch.no_grad():
    for test_images, test_labels in testloader:
        test_images = test_images.to(device)
        test_labels = test_labels.to(device)
        test_out = model(test_images)
        loss = loss_fn(test_out, test_labels)
        _, preds = torch.max(test_out, 1)
        test_loss += loss.item()
        test_corrects += torch.sum(preds == test_labels.data)

```

```
test_loss = (test_loss/(len(testloader)))
test_acc = test_corrects/(len(testloader)*batch_size)

print(f"Test acc: {test_acc}, test loss: {test_loss}")

with open( model_folder + "summary.pkl","rb") as file:
    summary = pickle.load(file)

model_index = np.argmax(summary["validation_accuracy"])
print(
    f"""
Train Loss: {summary["train_loss"][model_index]}
Validation Loss: {summary["validation_loss"][model_index]}
Train Accuracy: {summary["train_accuracy"][model_index]}
Validation Accuracy" {summary["validation_accuracy"][model_index]}
    """
)

if plot_latent_space:
    with torch.no_grad():

        layer_out = {}
        def hook(module_, input_, output_):
            layer_out["value"] = output_.detach().cpu().numpy()

        begin_latent = []
        middle_latent = []
        end_latent = []
        labels = []
        begin_model = torch.load(model_path).to(device=device)
        middle_model = torch.load(model_path).to(device=device)
        middle_model.load_state_dict(torch.load(model_folder + "middle_model.pkl"))

        for test_images, test_labels in testloader:
            test_images = test_images.to(device)
            labels.append(test_labels.cpu().numpy())

            begin_model.fc2.register_forward_hook(hook)
            _ = begin_model(test_images)
            begin_latent.append(layer_out["value"])

            middle_model.fc2.register_forward_hook(hook)
            _ = middle_model(test_images)
            middle_latent.append(layer_out["value"])

            model.fc2.register_forward_hook(hook)
            _ = model(test_images)
            end_latent.append(layer_out["value"])

    begin_latent = np.concatenate(begin_latent, axis=0)
    middle_latent = np.concatenate(middle_latent, axis=0)
    end_latent = np.concatenate(end_latent, axis=0)
    labels = np.concatenate(labels, axis=0)

```

```
    def plot_latent_space(values, labels, name=""):
        classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
        tsne_model = TSNE(perplexity=40, n_components=2, init='pca', n_iter=2500,
            random_state=3136)
        values_tsne = tsne_model.fit_transform(values)
        x = values_tsne[:,0]
        y = values_tsne[:,1]

        label_numbers = np.unique(labels)
        col_map = {label:f"C{i}" for i, label in enumerate(label_numbers)}
        plt.figure(figsize=(12,12))
        for label_number in label_numbers:
            label_name = classes[label_number]
            plt.scatter(x[labels==label_number], y[labels==label_number], label=
                label_name)
        plt.legend()
        plt.xlabel("Dim 1")
        plt.ylabel("Dim 2")
        plt.title(f"TSNE Plot of {name} Latent Space")
        plt.savefig(f"{model_folder}{name}.png")

    plot_latent_space(begin_latent, labels, name="Begin")
    plot_latent_space(middle_latent, labels, name="Middle")
    plot_latent_space(end_latent, labels, name="End")
```