ENGN8536 CLab2 Report

Shreya Chawla (u7195872)

2 Task 1: Implement the Dynamic Convolutional Network

2.1 Fill in the code for the BaseModel() class in model.py

The code for BaseModel is in Appendix A below. The BaseModel is initialized in the *__init()__* function. It has a backbone of pretrained ResNet18 uptil the first basic block of layer 1 (index 0). The non-linear weight generating network, dynamic convolution network, and finally classification network follow the backbone. Then according to the specified architecture (Fig. 1 below), the forward path is in the *forward()* function. Using an assert statement, the returned feature map from the backbone (*v*) is first checked. Based on the *with_dyn* argument, the function either uses the L2 normalized weights generator block as weights for dynamic convolution layer or not. Then the flattened feature map of backbone is flattened and passed to the dynamic convolution layer. Finally it is passed to the classification fully connected layer. The details of initialized model are as follows:

```
BaseModel(
  (backbone): Sequential(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1_index0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (dyn): Sequential(
    (0): Linear(in_features=4096, out_features=576, bias=True)
    (1): Tanh()
  )
  (dc): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (cls): Linear(in_features=64, out_features=10, bias=True)
)
```

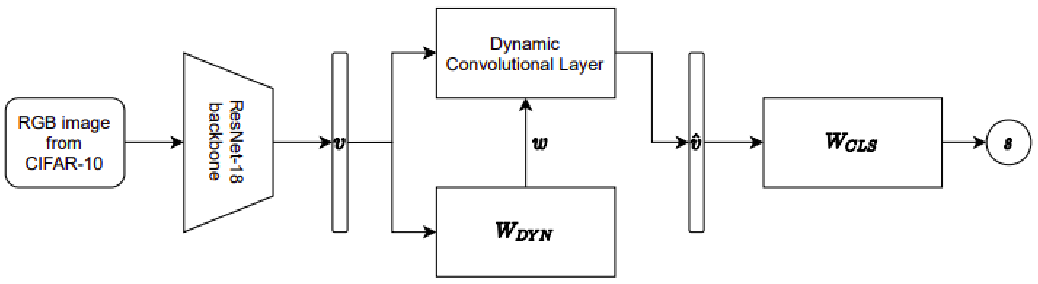


Figure 1: Network semantic for the network used in forward function to connect components

2.2 Fill in the missing code in train.py as indicated by the comments

1. The data is split into two - train and test sets. All the learnable parameters and optimizer are saved at the checkpoint path. The state dictionary of model parameters and optimizers is saved whenever a model with better testing accuracy is observed. This task is performed in the *train* function after comparing the current best test accuracy with the latest one returned by the *evaluate* function. This is useful for drawing inferences on the model in future, resuming training later and for testing the highest performance state of the network. The code for it is in Appendix B.

2. `torch.save` saves all the intermediate variables and outputs along with the learnable parameters in a pickle format. However we only require the learnable parameters (weights and biases) to saved for inference later using `state_dict`. The major disadvantage of saving the entire module is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved, which might lead to the code to break when used in other projects or after refactoring code. This is because pickle does not save the model class itself but rather saves a path to the file containing the class, which is

used during load time [2]. Hence, it is recommended by Pytorch to save learnable parameters and not pickle the entire model. Another reason for the recommendation is that `state_dict` is a Python dictionary object which is easier to save, update, alter, and gives the user much more flexibility and modularity for restoration than saving the entire model [2].

3. The *resume* function loads the best trained model that was saved (in 2.2.1) and optimizer learnable parameters from the checkpoint path to resume training or testing on the network at its best performance state. It uses deepcopy to avoid changing learned parameter values to change in the original variable. The code is in Appendix B.

2.3 Train the network and compare to without dynamic filter (the statistics on which models are being compared are - (1) best testing accuracy, (2) loss, and (3) time for a batch to train)

1. The best accuracy by training network with W_{DYN} initialized weights of DC layer (i.e. setting argument with_dyn as 1) is 55.2% with CrossEntropyLoss of 1.14395, when none of the hyperparameters are changed. However, this was a highly underfitting model. The model was found to have best results when trained for *30* epochs with other default settings like batch size, learning rate etc. Thus, the best testing accuracy for the improved model was 72.5% at the last epoch with a loss of 0.60442. The results are documented in Appendix C for reference.

2. Without tuning hyperparameters, the best testing accuracy achieved for the network without dynamic filter (i.e. setting argument with_dyn as 0) was 70.7% with a CrossEntropyLoss of 0.58848. However, upon inspection it was found that this network was underfitting, thus the need for hyperparameter tuning. The best test accuracy was 74.6% observed at the *13th* epoch out of *15* epochs (if counting from 1) for which it was trained with a loss of 0.37159. The best tuned hyperparameters for this network were when the arguments, except number of epochs (=15), were kept at their default values. This a 4% improvement in accuracy by training for more epochs and also an improvement in loss. Thus, this network thus outperforms the network that does use W_{DYN}. The reason for this is that the input dependent W_{DYN} is translation invariant and very position specific [1]. Here, no data augmentation like flipping, translation etc were performed thus it performs poorly for unseen data.
   In terms of time to train a batch of image data, it varies from 32 to 36 seconds with an average of about 34 seconds for the network without dynamic filter. However, for the network that uses dynamic filter, the batch train time lies in the range of 35 to 39 seconds with an average of 37 seconds. Thus the network takes approximately 3 seconds less on average to train without dynamic filter. This difference is because of the time taken to pass *v* to W_{DYN} fully connected and then passing it through tanh activation to be normalized and then reshaped as weights for the DC network. The data to support these claims (print statements when training/testing - result of running code in Appendix B) is in Appendix D.

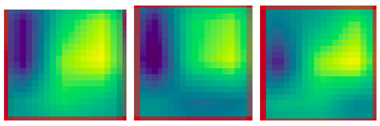
3. To improve the better network (without the dynamic filters) a new class called ImprovedModel is used which adds a fully connected layer before the classification layer and a dropout layer is added in between them. This is done to learn the correlation between final features more closely. This resulted in the improvement of accuracy to 74.7% by using learning rate of 5e-4 learnt over 20 epochs. The details for the network are below with code in Appendix F. The results are in Appendix H.

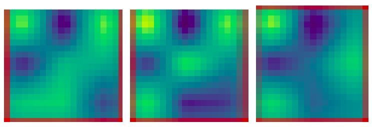
```
ImprovedModel(
  (backbone): Sequential(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1_index0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (dc): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc): Linear(in_features=64, out_features=64, bias=True)
  (bn): Dropout2d(p=0.5, inplace=False)
  (cls): Linear(in_features=64, out_features=10, bias=True)
)
```

3 Task 2: Understand the Dynamic Convolutional Network

3.1 Visualize the dynamic kernels

1. The weights of the 64 filters of the dynamic convolutional layer, which are produced by the weights generator network W_{DYN} for three different plane images are visualized in the question. Three sets of randomly chosen visualizations are shown in Fig. 2 below. W_{DYN} comes after the pre-trained ResNet backbone. Hence, the kernels in this layer encode more complex shapes than its predecessor. These would help in extracting wings, tail, cockpit and other parts of the plane.
As these kernels become more complex in this layer, it gets less likely for them to occur in images of two different classes. All the three sets of images have very similar kernels at the corresponding locations with slightly varying weights. This is because all of them represent the same class “plane”. The yellow regions correspond to higher weights and blue for lower weights. The regions with higher weights have more important features encoded. The three sets of weights in Fig. 2 shows similar trends in weights.This shows that the network is learning weights that correspond to “plane” class to differentiate it from a different class.


a) 4th row 2nd column


b) 4th row 6th column

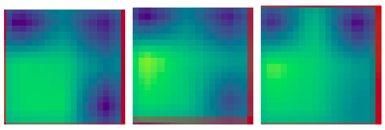
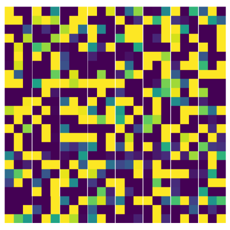
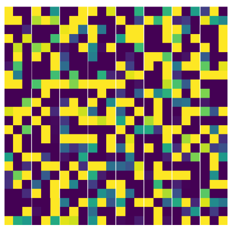
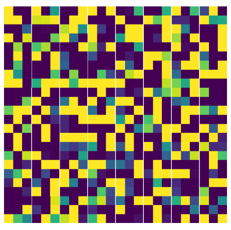

c) 2nd row 1st column

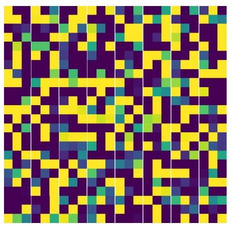
Figure 2: In a), b), and c) different rows and columns for each weight visualization for the class “plane” are compared. In the each of the above triplets, the left one is from plane 1, middle from plane 2 and right weights from plane 3.

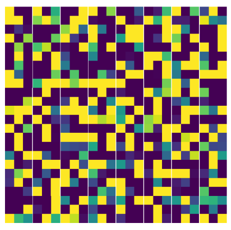
2. The code used to visualize the kernels is in Appendix E and the changed train function is in Appendix G. By visualizing the learned filters (Fig 3), it is observed that the kernels are starting to be tuned to adapt to the image structure by only looking at unlabelled training data. The method proposed in [1] applies a position specific transformation to the image or feature maps. Although the network was trained with default arguments with only 5 epochs there are some differences in how the kernels are activated and the intensity of the weights being represented. These are very subtle as the model is underfitting. Hence, the interclass difference is less visible. However, in Fig. 2, the intraclass similarity due to similar weights for similar class are being triggered. We can conclude that the network is picking up flow information from unlabelled training data. This shows that the network could be used to pretrain networks for various supervised tasks in an unsupervised way [1].

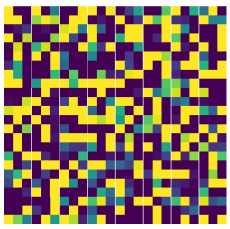

a) plane

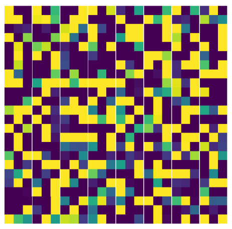

b) car

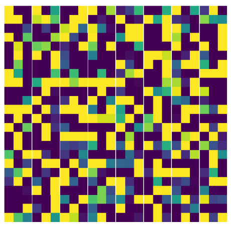

c) bird

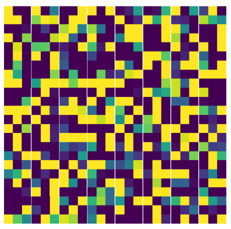

d) cat


e) deer


f) dog


g) frog


h) horse


i) ship

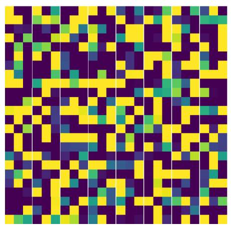

j) truck

Figure 3: From a) to j) represent the 64 kernels in DC layer after passing an image of a specific class through the network in the last training epoch. The model is trained for 5 epochs.

3.2 Discussion

The second part of this question is discussed here. Data augmentation like translation, flip, rotate, crop are analogous to imagination or dreaming in the human world. It helps the model by providing new and different alterations possible to images which aids in a better understanding of image data [3]. In this task, we classify images into 10 different classes. These classes like “dog”, “plane” etc can be present in any portion of the image thus an enlarged data helps by convolution network to look for the object everywhere. The arbitrariness factor improves the robustness and thus should improve its accuracy and lower its error rate on unseen data. This observed trend [4] should thus be beneficial for this task as well. Here, with augmentation of data the W_{DYN} might perform better as it would be able to overcome its position dependence.

References

1. Xu Jia et al. “Dynamic filter networks”. In Advances in neural information processing systems. 2016, pp. 667–675.
2. Matthew Inkawhich. “Saving and Loading Models - PyTorch”. https://pytorch.org/tutorials/beginner/saving_loading_models.html
3. Connor Shorten, and Taghi M. Khoshgoftaar. “A survey on image data augmentation for deep learning”. In Journal of Big Data 6. 2019. https://doi.org/10.1186/s40537-019-0197-0
4. Huang, Gao and Liu, Zhuang and van der Maaten, Laurens and Weinberger, and Kilian Q. “Densely Connected Convolutional Networks”. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). July, 2017.

Appendix A - “model.py”

```
import torch
import torch.nn as nn
import numpy as np
import torchvision
from collections import OrderedDict


class BaseModel(nn.Module):
   def __init__(self, args):
      super(BaseModel, self).__init__()
      self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
      self.args = args
      self.class_num = 10

      ### initialize the BaseModel --------------------------------
      resnet18 = torchvision.models.resnet18(pretrained=True)
      layer1 = [module for module in resnet18.layer1.modules() if not isinstance(module,
nn.Sequential)]
      self.backbone = nn.Sequential(OrderedDict([
         ('conv1', resnet18.conv1),
         ('bn1', resnet18.bn1),
         ('relu', resnet18.relu),
         ('maxpool', resnet18.maxpool),
         ('layer1_index0', layer1[0])]))
      self.dyn = nn.Sequential(nn.Linear(in_features=4096, out_features=576),
                               nn.Tanh())
      self.dc = nn.Conv2d(in_channels=64, out_channels=1, kernel_size=3, stride=1, padding=1)
      self.cls = nn.Linear(in_features=64, out_features=10)

   def forward(self, imgs, with_dyn=True):
      if with_dyn:
         ''' with dynamic convolutional layer '''
         ### complete the forward path --------------------
         v = self.backbone(imgs)
         assert list(v.shape) == [imgs.shape[0], 64, 8, 8]  # Sanity check
         v_flat = torch.flatten(v, 1)
         w = self.dyn(v_flat)
         w = torch.norm(w, dim=0)
         w = w.view(1, 64, 3, 3)
         self.dc.weight.data = w
         v = self.dc(v)
         v = torch.flatten(v, 1)
         cls_scores = self.cls(v)
      else:
         ''' without dynamic convolutional layer '''
         ### complete the forward path --------------------
         v = self.backbone(imgs)
         assert list(v.shape) == [imgs.shape[0], 64, 8, 8]  # Sanity check
         v = self.dc(v)
         v = torch.flatten(v, 1)
         cls_scores = self.cls(v)

      return cls_scores
```

Appendix B - “train.py”

```
import os
import time
import torch
import copy

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

criterion = torch.nn.CrossEntropyLoss()


def train(args, model, optimizer, dataloaders):
   trainloader, testloader = dataloaders

   best_testing_accuracy = 0.0

   # training
   for epoch in range(args.epochs):
      model.train()

      batch_time = time.time();
      iter_time = time.time()
      for i, data in enumerate(trainloader):

         imgs, labels = data
         imgs, labels = imgs.to(device), labels.to(device)

         cls_scores = model(imgs, with_dyn=args.with_dyn)
         loss = criterion(cls_scores, labels)

         optimizer.zero_grad()
         loss.backward()
         optimizer.step()

         if i % 100 == 0 and i != 0:
            print('epoch:{}, iter:{}, time:{:.2f}, loss:{:.5f}'.format(epoch, i,
                                                                     time.time() - iter_time,
loss.item()))
            iter_time = time.time()
      batch_time = time.time() - batch_time
      print('[epoch {} | time:{:.2f} | loss:{:.5f}]'.format(epoch, batch_time, loss.item()))
      print('-------------------------------------------------')

      if epoch % 1 == 0:
         testing_accuracy = evaluate(args, model, testloader)
         print('testing accuracy: {:.3f}'.format(testing_accuracy))

         if testing_accuracy > best_testing_accuracy:
            ### compare the previous best testing accuracy and the new testing accuracy
            ### save the model and the optimizer --------------------------------
            best_testing_accuracy = testing_accuracy  # Change the best test id
            checkpoint_path = './{}_checkpoint.pth'.format(args.exp_id)
            # Save learnable parameters and optimizer
            torch.save({
               'model_state_dict': model.state_dict(),
               'optimizer_state_dict': optimizer.state_dict(),
            }, checkpoint_path)
            # Save entire model - will not use as not recommended
            # torch.save(model, checkpoint_path)
            print('new best model saved at epoch: {}'.format(epoch))
   print('--------------------------------------------------')
   print('best testing accuracy achieved: {:.3f}'.format(best_testing_accuracy))


def evaluate(args, model, testloader):
   model.eval()
   total_count = torch.tensor([0.0]);
   correct_count = torch.tensor([0.0])
   for i, data in enumerate(testloader):
      imgs, labels = data
      imgs, labels = imgs.to(device), labels.to(device)

      total_count += labels.size(0)

      with torch.no_grad():
         cls_scores = model(imgs, with_dyn=args.with_dyn)

         predict = torch.argmax(cls_scores, dim=1)
         correct_count += (predict == labels).sum()
   testing_accuracy = correct_count / total_count
   model.train()
   return testing_accuracy.item()


def resume(args, model, optimizer):
   checkpoint_path = './{}_checkpoint.pth'.format(args.exp_id)
   assert os.path.exists(checkpoint_path), ('checkpoint do not exits for %s' %
checkpoint_path)

   ### load the model and the optimizer --------------------------------
   checkpoint = torch.load(checkpoint_path)
   # Loading learnable parameters and optimizer
   model.load_state_dict(copy.deepcopy(checkpoint['model_state_dict']))
   optimizer.load_state_dict(copy.deepcopy(checkpoint['optimizer_state_dict']))
   # Loading entire model - will not use as not recommended
   # model = torch.load(checkpoint_path)
   model.to(device)

   print('Resume completed for the model\n')

   return model, optimizer
```

Appendix C - result of running code in Appendix B (“train.py”) with W_{DYN} network

```
epochs=30, rest default argos value
epoch:0, iter:100, time:6.66, loss:4.66336
epoch:0, iter:200, time:7.13, loss:3.89425
epoch:0, iter:300, time:6.80, loss:3.98371
epoch:0, iter:400, time:6.96, loss:3.13178
epoch:0, iter:500, time:4.91, loss:3.95998
epoch:0, iter:600, time:5.40, loss:2.33818
epoch:0, iter:700, time:5.06, loss:2.08297
```

```
[epoch 0 | time:46.16 | loss:1.55001]
-------------------------------------------------
testing accuracy: 0.386
new best model saved at epoch: 0
epoch:1, iter:100, time:5.22, loss:2.37274
epoch:1, iter:200, time:4.96, loss:2.16761
epoch:1, iter:300, time:5.64, loss:2.55517
epoch:1, iter:400, time:5.06, loss:2.09456
epoch:1, iter:500, time:5.04, loss:1.82562
epoch:1, iter:600, time:5.43, loss:2.20768
epoch:1, iter:700, time:5.26, loss:1.92427
[epoch 1 | time:39.85 | loss:2.03804]
-------------------------------------------------
testing accuracy: 0.307
epoch:2, iter:100, time:5.10, loss:1.71076
epoch:2, iter:200, time:5.23, loss:1.95368
epoch:2, iter:300, time:5.07, loss:1.71122
epoch:2, iter:400, time:5.67, loss:1.95801
epoch:2, iter:500, time:5.24, loss:1.72263
epoch:2, iter:600, time:5.22, loss:1.81312
epoch:2, iter:700, time:4.99, loss:1.78522
[epoch 2 | time:39.81 | loss:1.31945]
-------------------------------------------------
testing accuracy: 0.479
new best model saved at epoch: 2
epoch:3, iter:100, time:4.84, loss:1.88116
epoch:3, iter:200, time:4.26, loss:1.46840
epoch:3, iter:300, time:4.38, loss:1.24521
epoch:3, iter:400, time:4.56, loss:1.79357
epoch:3, iter:500, time:4.30, loss:1.43500
epoch:3, iter:600, time:4.52, loss:1.33443
epoch:3, iter:700, time:4.65, loss:1.07833
[epoch 3 | time:34.56 | loss:1.36732]
-------------------------------------------------
testing accuracy: 0.512
new best model saved at epoch: 3
epoch:4, iter:100, time:5.32, loss:1.52675
epoch:4, iter:200, time:4.77, loss:1.26445
epoch:4, iter:300, time:4.48, loss:1.30944
epoch:4, iter:400, time:4.46, loss:1.33959
epoch:4, iter:500, time:4.47, loss:1.46000
epoch:4, iter:600, time:4.62, loss:1.10805
epoch:4, iter:700, time:4.77, loss:1.15165
[epoch 4 | time:36.28 | loss:1.14395]
-------------------------------------------------
testing accuracy: 0.552
new best model saved at epoch: 4
epoch:5, iter:100, time:4.75, loss:1.08094
epoch:5, iter:200, time:4.85, loss:1.21258
epoch:5, iter:300, time:4.72, loss:1.23887
epoch:5, iter:400, time:4.67, loss:1.21088
epoch:5, iter:500, time:5.34, loss:1.29369
epoch:5, iter:600, time:6.10, loss:1.40036
epoch:5, iter:700, time:5.15, loss:1.19267
[epoch 5 | time:38.57 | loss:1.31306]
-------------------------------------------------
testing accuracy: 0.577
new best model saved at epoch: 5
epoch:6, iter:100, time:4.98, loss:0.93933
epoch:6, iter:200, time:4.89, loss:0.94547
epoch:6, iter:300, time:4.79, loss:1.22864
epoch:6, iter:400, time:4.94, loss:1.12099
epoch:6, iter:500, time:5.43, loss:1.30183
epoch:6, iter:600, time:5.33, loss:1.25788
epoch:6, iter:700, time:5.80, loss:1.45897
[epoch 6 | time:39.43 | loss:1.18841]
-------------------------------------------------
testing accuracy: 0.632
new best model saved at epoch: 6
epoch:7, iter:100, time:5.02, loss:1.18946
epoch:7, iter:200, time:5.43, loss:0.95959
```

```
epoch:7, iter:300, time:5.18, loss:1.14391
epoch:7, iter:400, time:5.79, loss:1.11190
epoch:7, iter:500, time:4.94, loss:1.42304
epoch:7, iter:600, time:5.01, loss:1.08129
epoch:7, iter:700, time:4.77, loss:0.98733
[epoch 7 | time:39.86 | loss:1.32852]
-------------------------------------------------
testing accuracy: 0.598
epoch:8, iter:100, time:4.76, loss:1.12049
epoch:8, iter:200, time:4.63, loss:0.89200
epoch:8, iter:300, time:4.59, loss:1.14715
epoch:8, iter:400, time:4.66, loss:1.04237
epoch:8, iter:500, time:5.01, loss:1.25274
epoch:8, iter:600, time:4.94, loss:0.99939
epoch:8, iter:700, time:4.91, loss:0.98082
[epoch 8 | time:36.67 | loss:1.03717]
-------------------------------------------------
testing accuracy: 0.610
epoch:9, iter:100, time:5.07, loss:0.99067
epoch:9, iter:200, time:5.20, loss:1.08329
epoch:9, iter:300, time:5.25, loss:1.13277
epoch:9, iter:400, time:5.00, loss:1.21874
epoch:9, iter:500, time:4.95, loss:1.00913
epoch:9, iter:600, time:5.01, loss:1.06378
epoch:9, iter:700, time:5.10, loss:1.11873
[epoch 9 | time:38.86 | loss:1.15421]
-------------------------------------------------
testing accuracy: 0.657
new best model saved at epoch: 9
epoch:10, iter:100, time:5.08, loss:0.76482
epoch:10, iter:200, time:4.93, loss:1.00561
epoch:10, iter:300, time:4.87, loss:0.72297
epoch:10, iter:400, time:4.89, loss:0.82071
epoch:10, iter:500, time:4.82, loss:0.88779
epoch:10, iter:600, time:5.00, loss:1.14849
epoch:10, iter:700, time:4.95, loss:1.02681
[epoch 10 | time:37.82 | loss:1.00891]
-------------------------------------------------
testing accuracy: 0.633
epoch:11, iter:100, time:5.06, loss:0.78215
epoch:11, iter:200, time:4.96, loss:0.92276
epoch:11, iter:300, time:4.69, loss:0.80764
epoch:11, iter:400, time:4.66, loss:0.90209
epoch:11, iter:500, time:4.65, loss:1.17964
epoch:11, iter:600, time:4.73, loss:0.85514
epoch:11, iter:700, time:4.63, loss:1.01248
[epoch 11 | time:36.41 | loss:0.85032]
-------------------------------------------------
testing accuracy: 0.669
new best model saved at epoch: 11
epoch:12, iter:100, time:5.15, loss:0.89429
epoch:12, iter:200, time:5.16, loss:0.77337
epoch:12, iter:300, time:5.68, loss:0.84603
epoch:12, iter:400, time:5.47, loss:0.72933
epoch:12, iter:500, time:4.99, loss:0.64749
epoch:12, iter:600, time:5.09, loss:0.89044
epoch:12, iter:700, time:5.54, loss:1.00091
[epoch 12 | time:40.89 | loss:0.87278]
-------------------------------------------------
testing accuracy: 0.639
epoch:13, iter:100, time:5.58, loss:0.87989
epoch:13, iter:200, time:5.81, loss:0.99942
epoch:13, iter:300, time:6.05, loss:0.88447
epoch:13, iter:400, time:5.35, loss:0.83884
epoch:13, iter:500, time:4.78, loss:0.97285
epoch:13, iter:600, time:5.08, loss:0.75082
epoch:13, iter:700, time:6.83, loss:0.86508
[epoch 13 | time:42.81 | loss:0.76436]
-------------------------------------------------
testing accuracy: 0.676
new best model saved at epoch: 13
```

```
epoch:14, iter:100, time:5.22, loss:0.88959
epoch:14, iter:200, time:4.91, loss:0.70678
epoch:14, iter:300, time:4.89, loss:0.96874
epoch:14, iter:400, time:4.93, loss:0.69074
epoch:14, iter:500, time:4.86, loss:0.82387
epoch:14, iter:600, time:5.41, loss:0.80664
epoch:14, iter:700, time:5.47, loss:0.65891
[epoch 14 | time:39.00 | loss:0.73816]
-------------------------------------------------
testing accuracy: 0.701
new best model saved at epoch: 14
epoch:15, iter:100, time:5.25, loss:0.60171
epoch:15, iter:200, time:5.98, loss:0.69189
epoch:15, iter:300, time:5.66, loss:0.84217
epoch:15, iter:400, time:5.16, loss:0.93076
epoch:15, iter:500, time:6.00, loss:0.81583
epoch:15, iter:600, time:5.31, loss:0.74725
epoch:15, iter:700, time:5.13, loss:0.74044
[epoch 15 | time:41.91 | loss:0.85548]
-------------------------------------------------
testing accuracy: 0.685
epoch:16, iter:100, time:5.76, loss:0.80568
epoch:16, iter:200, time:4.93, loss:0.89403
epoch:16, iter:300, time:5.61, loss:0.79362
epoch:16, iter:400, time:5.14, loss:0.73462
epoch:16, iter:500, time:5.31, loss:0.68932
epoch:16, iter:600, time:5.39, loss:0.94873
epoch:16, iter:700, time:5.26, loss:0.74697
[epoch 16 | time:40.74 | loss:0.86835]
-------------------------------------------------
testing accuracy: 0.665
epoch:17, iter:100, time:5.74, loss:0.43635
epoch:17, iter:200, time:6.32, loss:0.75520
epoch:17, iter:300, time:5.94, loss:0.68209
epoch:17, iter:400, time:5.47, loss:0.93749
epoch:17, iter:500, time:5.56, loss:0.71303
epoch:17, iter:600, time:5.05, loss:0.70107
epoch:17, iter:700, time:5.75, loss:0.63589
[epoch 17 | time:43.44 | loss:0.47313]
-------------------------------------------------
testing accuracy: 0.686
epoch:18, iter:100, time:5.34, loss:0.61547
epoch:18, iter:200, time:5.13, loss:0.63147
epoch:18, iter:300, time:5.04, loss:0.72633
epoch:18, iter:400, time:4.82, loss:0.74217
epoch:18, iter:500, time:5.25, loss:0.89686
epoch:18, iter:600, time:5.79, loss:0.66433
epoch:18, iter:700, time:5.77, loss:0.55955
[epoch 18 | time:40.61 | loss:0.96896]
-------------------------------------------------
testing accuracy: 0.691
epoch:19, iter:100, time:5.57, loss:0.67669
epoch:19, iter:200, time:5.42, loss:0.59327
epoch:19, iter:300, time:5.45, loss:0.64385
epoch:19, iter:400, time:5.47, loss:0.74192
epoch:19, iter:500, time:5.61, loss:0.88586
epoch:19, iter:600, time:5.09, loss:0.60356
epoch:19, iter:700, time:5.00, loss:0.73971
[epoch 19 | time:40.86 | loss:0.57967]
-------------------------------------------------
testing accuracy: 0.693
epoch:20, iter:100, time:5.14, loss:0.63223
epoch:20, iter:200, time:4.87, loss:0.53009
epoch:20, iter:300, time:4.75, loss:0.73093
epoch:20, iter:400, time:5.21, loss:0.52533
epoch:20, iter:500, time:5.79, loss:0.71672
epoch:20, iter:600, time:5.58, loss:0.63863
epoch:20, iter:700, time:5.12, loss:0.71365
[epoch 20 | time:39.76 | loss:0.64635]
-------------------------------------------------
testing accuracy: 0.698
```

```
epoch:21, iter:100, time:5.29, loss:0.63070
epoch:21, iter:200, time:4.89, loss:0.89738
epoch:21, iter:300, time:4.78, loss:0.55434
epoch:21, iter:400, time:4.89, loss:0.51574
epoch:21, iter:500, time:4.80, loss:0.44669
epoch:21, iter:600, time:5.00, loss:0.71594
epoch:21, iter:700, time:5.19, loss:0.73098
[epoch 21 | time:37.94 | loss:0.70176]
-------------------------------------------------
testing accuracy: 0.717
new best model saved at epoch: 21
epoch:22, iter:100, time:5.16, loss:0.65883
epoch:22, iter:200, time:4.98, loss:0.75806
epoch:22, iter:300, time:4.92, loss:0.74204
epoch:22, iter:400, time:4.86, loss:0.59586
epoch:22, iter:500, time:5.44, loss:0.59387
epoch:22, iter:600, time:6.60, loss:0.51480
epoch:22, iter:700, time:5.33, loss:0.55515
[epoch 22 | time:40.78 | loss:0.70496]
-------------------------------------------------
testing accuracy: 0.682
epoch:23, iter:100, time:5.47, loss:0.49601
epoch:23, iter:200, time:5.31, loss:0.58458
epoch:23, iter:300, time:5.65, loss:0.53686
epoch:23, iter:400, time:5.71, loss:0.47012
epoch:23, iter:500, time:5.52, loss:0.52302
epoch:23, iter:600, time:5.39, loss:0.52713
epoch:23, iter:700, time:5.18, loss:0.50962
[epoch 23 | time:41.54 | loss:0.64533]
-------------------------------------------------
testing accuracy: 0.707
epoch:24, iter:100, time:5.26, loss:0.92526
epoch:24, iter:200, time:4.98, loss:0.40280
epoch:24, iter:300, time:5.04, loss:0.49199
epoch:24, iter:400, time:4.72, loss:0.35768
epoch:24, iter:500, time:5.14, loss:0.72948
epoch:24, iter:600, time:5.78, loss:0.59053
epoch:24, iter:700, time:5.25, loss:0.47858
[epoch 24 | time:39.69 | loss:0.77053]
-------------------------------------------------
testing accuracy: 0.717
epoch:25, iter:100, time:5.37, loss:0.46829
epoch:25, iter:200, time:5.10, loss:0.55398
epoch:25, iter:300, time:5.38, loss:0.48520
epoch:25, iter:400, time:5.05, loss:0.77429
epoch:25, iter:500, time:5.40, loss:0.43325
epoch:25, iter:600, time:5.40, loss:0.77841
epoch:25, iter:700, time:5.07, loss:0.45751
[epoch 25 | time:40.01 | loss:0.49445]
-------------------------------------------------
testing accuracy: 0.693
epoch:26, iter:100, time:4.98, loss:0.48612
epoch:26, iter:200, time:4.89, loss:0.50488
epoch:26, iter:300, time:5.12, loss:0.44145
epoch:26, iter:400, time:5.52, loss:0.49871
epoch:26, iter:500, time:4.82, loss:0.42067
epoch:26, iter:600, time:4.85, loss:0.57767
epoch:26, iter:700, time:4.73, loss:0.38308
[epoch 26 | time:38.15 | loss:0.68189]
-------------------------------------------------
testing accuracy: 0.715
epoch:27, iter:100, time:5.10, loss:0.49696
epoch:27, iter:200, time:5.10, loss:0.52592
epoch:27, iter:300, time:5.20, loss:0.33135
epoch:27, iter:400, time:5.35, loss:0.50615
epoch:27, iter:500, time:5.15, loss:0.31709
epoch:27, iter:600, time:5.15, loss:0.44748
epoch:27, iter:700, time:5.52, loss:0.57528
[epoch 27 | time:39.90 | loss:0.61321]
-------------------------------------------------
testing accuracy: 0.716
```

```
epoch:28, iter:100, time:5.31, loss:0.31245
epoch:28, iter:200, time:4.95, loss:0.70106
epoch:28, iter:300, time:4.74, loss:0.60393
epoch:28, iter:400, time:4.74, loss:0.40662
epoch:28, iter:500, time:4.98, loss:0.66151
epoch:28, iter:600, time:4.97, loss:0.60472
epoch:28, iter:700, time:4.86, loss:0.56022
[epoch 28 | time:37.66 | loss:0.43643]
-------------------------------------------------
testing accuracy: 0.721
new best model saved at epoch: 28
epoch:29, iter:100, time:5.13, loss:0.42742
epoch:29, iter:200, time:5.11, loss:0.47122
epoch:29, iter:300, time:5.40, loss:0.50072
epoch:29, iter:400, time:5.08, loss:0.56812
epoch:29, iter:500, time:4.73, loss:0.67486
epoch:29, iter:600, time:4.78, loss:0.62197
epoch:29, iter:700, time:4.77, loss:0.51569
[epoch 29 | time:38.09 | loss:0.60442]
-------------------------------------------------
testing accuracy: 0.725
new best model saved at epoch: 29
-------------------------------------------------
best testing accuracy achieved: 0.725
training finished
```

Appendix D - result of running code in Appendix B (“train.py”) without W_{DYN} network

```
epochs=15, rest default argos value
epoch:0, iter:100, time:4.79, loss:1.68822
epoch:0, iter:200, time:3.86, loss:1.34278
epoch:0, iter:300, time:4.06, loss:1.18349
epoch:0, iter:400, time:4.09, loss:1.36585
epoch:0, iter:500, time:4.11, loss:1.21344
epoch:0, iter:600, time:4.11, loss:1.42703
epoch:0, iter:700, time:4.50, loss:1.00182
[epoch 0 | time:32.39 | loss:0.96481]
-------------------------------------------------
testing accuracy: 0.604
new best model saved at epoch: 0
epoch:1, iter:100, time:4.78, loss:1.14486
epoch:1, iter:200, time:4.57, loss:1.46873
epoch:1, iter:300, time:4.98, loss:1.01009
epoch:1, iter:400, time:4.98, loss:0.88131
epoch:1, iter:500, time:5.28, loss:0.89120
epoch:1, iter:600, time:4.76, loss:1.04200
epoch:1, iter:700, time:5.08, loss:0.99402
[epoch 1 | time:37.16 | loss:1.07104]
-------------------------------------------------
testing accuracy: 0.638
new best model saved at epoch: 1
epoch:2, iter:100, time:4.93, loss:0.88776
epoch:2, iter:200, time:4.84, loss:0.86487
epoch:2, iter:300, time:4.22, loss:0.78703
epoch:2, iter:400, time:4.50, loss:0.98544
epoch:2, iter:500, time:4.47, loss:0.81874
epoch:2, iter:600, time:4.31, loss:0.88671
epoch:2, iter:700, time:4.29, loss:1.02854
[epoch 2 | time:34.18 | loss:0.70711]
-------------------------------------------------
testing accuracy: 0.694
new best model saved at epoch: 2
epoch:3, iter:100, time:4.48, loss:1.02819
epoch:3, iter:200, time:4.73, loss:0.74736
epoch:3, iter:300, time:4.60, loss:0.71771
epoch:3, iter:400, time:4.70, loss:0.86878
epoch:3, iter:500, time:4.71, loss:0.89844
epoch:3, iter:600, time:4.73, loss:0.73069
epoch:3, iter:700, time:4.97, loss:0.83606
[epoch 3 | time:36.20 | loss:0.73831]
```

```
-------------------------------------------------
testing accuracy: 0.707
new best model saved at epoch: 3
epoch:4, iter:100, time:4.70, loss:0.87713
epoch:4, iter:200, time:4.92, loss:0.77343
epoch:4, iter:300, time:4.35, loss:0.87765
epoch:4, iter:400, time:4.21, loss:0.83624
epoch:4, iter:500, time:4.38, loss:0.84177
epoch:4, iter:600, time:4.86, loss:0.60446
epoch:4, iter:700, time:4.19, loss:0.76737
[epoch 4 | time:34.56 | loss:0.58848]
-------------------------------------------------
testing accuracy: 0.699
epoch:5, iter:100, time:4.83, loss:0.66113
epoch:5, iter:200, time:4.75, loss:0.70278
epoch:5, iter:300, time:4.52, loss:0.72900
epoch:5, iter:400, time:4.44, loss:0.56296
epoch:5, iter:500, time:4.56, loss:0.92771
epoch:5, iter:600, time:4.46, loss:1.11801
epoch:5, iter:700, time:4.24, loss:0.83432
[epoch 5 | time:34.60 | loss:0.79838]
-------------------------------------------------
testing accuracy: 0.702
epoch:6, iter:100, time:4.41, loss:0.77595
epoch:6, iter:200, time:4.38, loss:0.72965
epoch:6, iter:300, time:4.40, loss:0.85096
epoch:6, iter:400, time:5.14, loss:0.61592
epoch:6, iter:500, time:4.98, loss:0.77759
epoch:6, iter:600, time:4.84, loss:0.67435
epoch:6, iter:700, time:5.23, loss:0.76717
[epoch 6 | time:36.75 | loss:0.74421]
-------------------------------------------------
testing accuracy: 0.722
new best model saved at epoch: 6
epoch:7, iter:100, time:4.66, loss:0.42273
epoch:7, iter:200, time:4.46, loss:0.46398
epoch:7, iter:300, time:4.21, loss:0.82468
epoch:7, iter:400, time:4.20, loss:0.56721
epoch:7, iter:500, time:4.34, loss:0.91973
epoch:7, iter:600, time:4.51, loss:0.77650
epoch:7, iter:700, time:6.14, loss:0.72766
[epoch 7 | time:35.50 | loss:0.71977]
-------------------------------------------------
testing accuracy: 0.709
epoch:8, iter:100, time:4.68, loss:0.67004
epoch:8, iter:200, time:4.53, loss:0.50005
epoch:8, iter:300, time:4.39, loss:0.70306
epoch:8, iter:400, time:4.54, loss:0.75131
epoch:8, iter:500, time:4.48, loss:0.76352
epoch:8, iter:600, time:4.10, loss:0.70632
epoch:8, iter:700, time:4.02, loss:0.58338
[epoch 8 | time:33.52 | loss:0.63785]
-------------------------------------------------
testing accuracy: 0.709
epoch:9, iter:100, time:4.10, loss:0.58689
epoch:9, iter:200, time:4.48, loss:0.95607
epoch:9, iter:300, time:5.13, loss:0.44076
epoch:9, iter:400, time:4.70, loss:0.72016
epoch:9, iter:500, time:4.22, loss:0.62683
epoch:9, iter:600, time:4.85, loss:0.44897
epoch:9, iter:700, time:5.32, loss:0.72462
[epoch 9 | time:36.45 | loss:0.80229]
-------------------------------------------------
testing accuracy: 0.732
new best model saved at epoch: 9
epoch:10, iter:100, time:5.90, loss:0.30448
epoch:10, iter:200, time:5.72, loss:0.71878
epoch:10, iter:300, time:4.67, loss:0.50892
epoch:10, iter:400, time:4.96, loss:0.40206
epoch:10, iter:500, time:4.97, loss:0.54653
epoch:10, iter:600, time:5.56, loss:0.66159
```

```
epoch:10, iter:700, time:4.70, loss:0.58404
[epoch 10 | time:39.68 | loss:0.79825]
--------------------------------------------------
testing accuracy: 0.740
new best model saved at epoch: 10
epoch:11, iter:100, time:5.19, loss:0.55532
epoch:11, iter:200, time:4.83, loss:0.51779
epoch:11, iter:300, time:5.10, loss:0.39268
epoch:11, iter:400, time:5.14, loss:0.50070
epoch:11, iter:500, time:5.40, loss:0.83713
epoch:11, iter:600, time:4.84, loss:0.66599
epoch:11, iter:700, time:5.40, loss:0.68877
[epoch 11 | time:39.19 | loss:0.59439]
--------------------------------------------------
testing accuracy: 0.726
epoch:12, iter:100, time:4.48, loss:0.47523
epoch:12, iter:200, time:4.62, loss:0.44085
epoch:12, iter:300, time:4.77, loss:0.70992
epoch:12, iter:400, time:4.94, loss:0.47629
epoch:12, iter:500, time:4.53, loss:0.44232
epoch:12, iter:600, time:5.72, loss:0.61151
epoch:12, iter:700, time:4.97, loss:0.55631
[epoch 12 | time:37.29 | loss:0.58048]
--------------------------------------------------
testing accuracy: 0.746
new best model saved at epoch: 12
epoch:13, iter:100, time:4.90, loss:0.43390
epoch:13, iter:200, time:4.74, loss:0.64225
epoch:13, iter:300, time:4.67, loss:0.39349
epoch:13, iter:400, time:4.93, loss:0.35279
epoch:13, iter:500, time:4.70, loss:0.66584
epoch:13, iter:600, time:4.75, loss:0.31953
epoch:13, iter:700, time:4.75, loss:0.68339
[epoch 13 | time:36.54 | loss:0.53131]
--------------------------------------------------
testing accuracy: 0.733
epoch:14, iter:100, time:4.86, loss:0.35963
epoch:14, iter:200, time:4.71, loss:0.55037
epoch:14, iter:300, time:4.81, loss:0.52521
epoch:14, iter:400, time:5.08, loss:0.37555
epoch:14, iter:500, time:4.72, loss:0.52412
epoch:14, iter:600, time:4.82, loss:0.63301
epoch:14, iter:700, time:4.66, loss:0.30775
[epoch 14 | time:36.89 | loss:0.37159]
--------------------------------------------------
testing accuracy: 0.730
--------------------------------------------------
best testing accuracy achieved: 0.746
training finished
```

Appendix E - code in “utils.py”

```
import matplotlib.pyplot as plt
import numpy as np


def plot_weights(weight_tensor):
   t = weight_tensor
   nplots = t.shape[0] * t.shape[1]
   ncols = 8

   nrows = 1 + nplots // ncols
   # convert tensor to numpy image
   npimg = np.array(t.numpy(), np.float32)

   count = 0
   fig = plt.figure(figsize=(ncols, nrows))

   # looping through all the kernels in each channel
   for i in range(t.shape[0]):
      for j in range(t.shape[1]):
         count += 1
         ax1 = fig.add_subplot(nrows, ncols, count)
         npimg = np.array(t[i, j].numpy(), np.float32)
         npimg = (npimg - np.mean(npimg)) / np.std(npimg)
         npimg = np.minimum(1, np.maximum(0, (npimg + 0.5)))
         ax1.imshow(npimg)
         ax1.axis('off')
         ax1.set_xticklabels([])
         ax1.set_yticklabels([])
   plt.show()
```

Appendix F

```
class ImprovedModel(nn.Module):
    def __init__(self, args):
        super(ImprovedModel, self).__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.args = args
        self.class_num = 10

        ### initialize the ImprovedModel ---------------------------------
        resnet18 = torchvision.models.resnet18(pretrained=True)
        layer1 = [module for module in resnet18.layer1.modules() if not isinstance(module, 
nn.Sequential)]
        self.backbone = nn.Sequential(OrderedDict([
            ('conv1', resnet18.conv1),
            ('bn1', resnet18.bn1),
            ('relu', resnet18.relu),
            ('maxpool', resnet18.maxpool),
            ('layer1_index0', layer1[0])]))
        self.dc = nn.Conv2d(in_channels=64, out_channels=1, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(in_features=64, out_features=64)
        self.bn = nn.Dropout2d()
        self.cls = nn.Linear(in_features=64, out_features=10)

    def forward(self, imgs, with_dyn=True):
        ''' without dynamic convolutional layer '''
        v = self.backbone(imgs)
        assert list(v.shape) == [imgs.shape[0], 64, 8, 8]  # Sanity check
        v = self.dc(v)
        v = torch.flatten(v, 1)
        out = self.fc(v)
        out = self.bn(out)
        cls_scores = self.cls(out)

        return cls_scores
```

Appendix G - modified train function from “train.py” (Appendix B)

```
def train(args, model, optimizer, dataloaders):
    trainloader, testloader = dataloaders
    labs=[]
    best_testing_accuracy = 0.0

    # training
    for epoch in range(args.epochs):
        model.train()

        batch_time = time.time();
        iter_time = time.time()
        for i, data in enumerate(trainloader):

            imgs, labels = data
            imgs, labels = imgs.to(device), labels.to(device)

            cls_scores = model(imgs, with_dyn=args.with_dyn)
            loss = criterion(cls_scores, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if args.with_dyn == 1 and epoch == args.epochs-1 and i < 64:
               lab = labels[i].to(device)
               if 0 <= lab < 10 and (lab not in labs):
                  labs.append(lab)
                  print(labs)
                  kernels = model.dc.weight.data.detach().clone()
                  plot_weights(kernels)

            if i % 100 == 0 and i != 0:
               print('epoch:{}, iter:{}, time:{:.2f}, loss:{:.5f}'.format(epoch, i,
                                                                       time.time() - iter_time,
loss.item()))
               iter_time = time.time()
         batch_time = time.time() - batch_time
         print('[epoch {} | time:{:.2f} | loss:{:.5f}]'.format(epoch, batch_time, loss.item()))
         print('-------------------------------------------------')

         if epoch % 1 == 0:
            testing_accuracy = evaluate(args, model, testloader)
            print('testing accuracy: {:.3f}'.format(testing_accuracy))

            if testing_accuracy > best_testing_accuracy:
               ### compare the previous best testing accuracy and the new testing accuracy
               ### save the model and the optimizer --------------------------------
               best_testing_accuracy = testing_accuracy  # Change the best test id
               checkpoint_path = './{}_checkpoint.pth'.format(args.exp_id)
               # Save learnable parameters and optimizer
               torch.save({
                  'model_state_dict': model.state_dict(),
                  'optimizer_state_dict': optimizer.state_dict(),
               }, checkpoint_path)
               # Save entire model - will not use as not recommended
               # torch.save(model, checkpoint_path)
               print('new best model saved at epoch: {}'.format(epoch))
      print('-------------------------------------------------')
      print('best testing accuracy achieved: {:.3f}'.format(best_testing_accuracy))
```

Appendix H - result of running code in Appendix F (“model.py”) without W_{DYN} network

```
epochs=20, lr=5e-4, model=ImprovedModel, rest same default values
epoch:0, iter:100, time:5.89, loss:1.79380
epoch:0, iter:200, time:5.43, loss:1.62879
epoch:0, iter:300, time:5.83, loss:1.34657
epoch:0, iter:400, time:5.43, loss:1.63875
epoch:0, iter:500, time:5.55, loss:1.09792
epoch:0, iter:600, time:4.99, loss:1.23404
epoch:0, iter:700, time:4.91, loss:1.22944
[epoch 0 | time:41.75 | loss:1.54180]
-------------------------------------------------
testing accuracy: 0.577
new best model saved at epoch: 0
epoch:1, iter:100, time:5.15, loss:1.35563
epoch:1, iter:200, time:4.71, loss:1.33039
epoch:1, iter:300, time:5.62, loss:1.33073
epoch:1, iter:400, time:5.06, loss:1.24299
epoch:1, iter:500, time:4.58, loss:1.19945
epoch:1, iter:600, time:4.65, loss:1.25895
epoch:1, iter:700, time:4.36, loss:0.97900
[epoch 1 | time:37.09 | loss:0.92469]
-------------------------------------------------
testing accuracy: 0.582
new best model saved at epoch: 1
epoch:2, iter:100, time:5.61, loss:1.03361
epoch:2, iter:200, time:5.31, loss:1.12525
epoch:2, iter:300, time:5.27, loss:0.93554
```

```
epoch:2, iter:400, time:6.51, loss:0.96227
epoch:2, iter:500, time:5.45, loss:1.00872
epoch:2, iter:600, time:5.53, loss:0.92515
epoch:2, iter:700, time:4.56, loss:1.09278
[epoch 2 | time:41.18 | loss:0.91481]
-------------------------------------------------
testing accuracy: 0.660
new best model saved at epoch: 2
epoch:3, iter:100, time:4.39, loss:1.04207
epoch:3, iter:200, time:5.89, loss:0.91494
epoch:3, iter:300, time:4.65, loss:1.01634
epoch:3, iter:400, time:4.40, loss:1.09209
epoch:3, iter:500, time:4.45, loss:0.93683
epoch:3, iter:600, time:4.42, loss:1.06518
epoch:3, iter:700, time:4.25, loss:0.81213
[epoch 3 | time:35.49 | loss:0.77136]
-------------------------------------------------
testing accuracy: 0.667
new best model saved at epoch: 3
epoch:4, iter:100, time:7.68, loss:0.84757
epoch:4, iter:200, time:5.99, loss:0.92840
epoch:4, iter:300, time:5.16, loss:0.70863
epoch:4, iter:400, time:5.23, loss:0.90107
epoch:4, iter:500, time:4.73, loss:0.97415
epoch:4, iter:600, time:5.16, loss:0.84136
epoch:4, iter:700, time:5.02, loss:0.89633
[epoch 4 | time:42.02 | loss:0.75929]
-------------------------------------------------
testing accuracy: 0.685
new best model saved at epoch: 4
epoch:5, iter:100, time:5.25, loss:0.87808
epoch:5, iter:200, time:4.91, loss:0.95037
epoch:5, iter:300, time:5.50, loss:0.79480
epoch:5, iter:400, time:5.43, loss:0.74630
epoch:5, iter:500, time:5.83, loss:0.78715
epoch:5, iter:600, time:5.16, loss:0.59433
epoch:5, iter:700, time:5.93, loss:0.76733
[epoch 5 | time:41.37 | loss:0.60427]
-------------------------------------------------
testing accuracy: 0.697
new best model saved at epoch: 5
epoch:6, iter:100, time:5.73, loss:0.64476
epoch:6, iter:200, time:5.49, loss:1.03309
epoch:6, iter:300, time:5.25, loss:0.70814
epoch:6, iter:400, time:5.72, loss:0.85160
epoch:6, iter:500, time:5.24, loss:0.91971
epoch:6, iter:600, time:5.89, loss:0.97446
epoch:6, iter:700, time:4.95, loss:0.78539
[epoch 6 | time:41.54 | loss:0.91662]
-------------------------------------------------
testing accuracy: 0.705
new best model saved at epoch: 6
epoch:7, iter:100, time:6.44, loss:0.86332
epoch:7, iter:200, time:4.84, loss:0.98862
epoch:7, iter:300, time:5.25, loss:0.68715
epoch:7, iter:400, time:5.30, loss:1.00643
epoch:7, iter:500, time:5.46, loss:0.78018
epoch:7, iter:600, time:5.27, loss:0.85774
epoch:7, iter:700, time:5.36, loss:0.75025
[epoch 7 | time:41.21 | loss:0.82625]
-------------------------------------------------
testing accuracy: 0.713
new best model saved at epoch: 7
epoch:8, iter:100, time:5.64, loss:1.01164
epoch:8, iter:200, time:5.43, loss:0.76935
epoch:8, iter:300, time:5.22, loss:0.87667
epoch:8, iter:400, time:5.21, loss:0.76223
epoch:8, iter:500, time:6.19, loss:0.85212
epoch:8, iter:600, time:6.44, loss:0.59262
epoch:8, iter:700, time:5.23, loss:0.69754
[epoch 8 | time:42.97 | loss:0.73187]
```

```
--------------------------------------------------
testing accuracy: 0.716
new best model saved at epoch: 8
epoch:9, iter:100, time:6.16, loss:0.51978
epoch:9, iter:200, time:6.63, loss:0.86902
epoch:9, iter:300, time:5.37, loss:0.76079
epoch:9, iter:400, time:5.25, loss:0.69377
epoch:9, iter:500, time:4.93, loss:0.78683
epoch:9, iter:600, time:5.04, loss:0.89675
epoch:9, iter:700, time:4.82, loss:0.70047
[epoch 9 | time:41.37 | loss:0.55175]
--------------------------------------------------
testing accuracy: 0.708
epoch:10, iter:100, time:4.80, loss:0.63889
epoch:10, iter:200, time:4.41, loss:0.96686
epoch:10, iter:300, time:4.45, loss:0.81672
epoch:10, iter:400, time:4.53, loss:0.63458
epoch:10, iter:500, time:4.69, loss:0.81055
epoch:10, iter:600, time:4.57, loss:0.74500
epoch:10, iter:700, time:4.66, loss:0.62073
[epoch 10 | time:35.35 | loss:0.99880]
--------------------------------------------------
testing accuracy: 0.716
epoch:11, iter:100, time:5.13, loss:0.54886
epoch:11, iter:200, time:5.00, loss:0.74023
epoch:11, iter:300, time:4.70, loss:0.79132
epoch:11, iter:400, time:4.53, loss:0.62955
epoch:11, iter:500, time:4.70, loss:0.54841
epoch:11, iter:600, time:4.65, loss:0.49955
epoch:11, iter:700, time:4.40, loss:0.94188
[epoch 11 | time:36.03 | loss:1.04328]
--------------------------------------------------
testing accuracy: 0.722
new best model saved at epoch: 11
epoch:12, iter:100, time:5.80, loss:0.59704
epoch:12, iter:200, time:5.18, loss:0.68880
epoch:12, iter:300, time:4.87, loss:0.77397
epoch:12, iter:400, time:5.55, loss:1.03296
epoch:12, iter:500, time:5.67, loss:0.60443
epoch:12, iter:600, time:5.28, loss:0.71553
epoch:12, iter:700, time:5.69, loss:0.95850
[epoch 12 | time:41.76 | loss:0.65590]
--------------------------------------------------
testing accuracy: 0.719
epoch:13, iter:100, time:5.08, loss:0.71876
epoch:13, iter:200, time:4.94, loss:0.56410
epoch:13, iter:300, time:4.81, loss:0.51248
epoch:13, iter:400, time:5.88, loss:0.75953
epoch:13, iter:500, time:5.41, loss:0.66307
epoch:13, iter:600, time:5.73, loss:0.88446
epoch:13, iter:700, time:5.25, loss:0.63018
[epoch 13 | time:40.77 | loss:0.52732]
--------------------------------------------------
testing accuracy: 0.721
epoch:14, iter:100, time:5.69, loss:0.61891
epoch:14, iter:200, time:4.32, loss:0.77129
epoch:14, iter:300, time:4.39, loss:0.58738
epoch:14, iter:400, time:5.27, loss:0.55170
epoch:14, iter:500, time:5.15, loss:0.52152
epoch:14, iter:600, time:4.35, loss:0.70710
epoch:14, iter:700, time:5.01, loss:0.63229
[epoch 14 | time:38.65 | loss:0.77328]
--------------------------------------------------
testing accuracy: 0.742
new best model saved at epoch: 14
epoch:15, iter:100, time:6.24, loss:0.73240
epoch:15, iter:200, time:5.09, loss:0.41115
epoch:15, iter:300, time:5.09, loss:0.42440
epoch:15, iter:400, time:4.43, loss:0.75409
epoch:15, iter:500, time:4.41, loss:0.53731
epoch:15, iter:600, time:4.81, loss:0.74041
```

```
epoch:15, iter:700, time:4.89, loss:0.47680
[epoch 15 | time:37.75 | loss:0.73671]
-------------------------------------------------
testing accuracy: 0.746
new best model saved at epoch: 15
epoch:16, iter:100, time:4.59, loss:0.66955
epoch:16, iter:200, time:4.61, loss:0.64611
epoch:16, iter:300, time:5.11, loss:0.51189
epoch:16, iter:400, time:6.20, loss:0.36393
epoch:16, iter:500, time:5.03, loss:0.64394
epoch:16, iter:600, time:6.01, loss:0.49358
epoch:16, iter:700, time:5.40, loss:0.46126
[epoch 16 | time:40.45 | loss:0.34456]
-------------------------------------------------
testing accuracy: 0.738
epoch:17, iter:100, time:5.50, loss:0.76248
epoch:17, iter:200, time:5.28, loss:0.35291
epoch:17, iter:300, time:5.34, loss:0.62032
epoch:17, iter:400, time:5.69, loss:0.51829
epoch:17, iter:500, time:5.43, loss:0.48478
epoch:17, iter:600, time:5.58, loss:0.68726
epoch:17, iter:700, time:5.99, loss:0.64458
[epoch 17 | time:42.17 | loss:0.81196]
-------------------------------------------------
testing accuracy: 0.732
epoch:18, iter:100, time:5.76, loss:0.45850
epoch:18, iter:200, time:5.79, loss:0.75316
epoch:18, iter:300, time:5.66, loss:0.79613
epoch:18, iter:400, time:5.79, loss:0.53628
epoch:18, iter:500, time:5.45, loss:0.63198
epoch:18, iter:600, time:6.14, loss:0.72104
epoch:18, iter:700, time:5.89, loss:0.70041
[epoch 18 | time:44.22 | loss:0.44870]
-------------------------------------------------
testing accuracy: 0.729
epoch:19, iter:100, time:5.94, loss:0.69665
epoch:19, iter:200, time:5.91, loss:0.51378
epoch:19, iter:300, time:5.90, loss:0.44083
epoch:19, iter:400, time:7.14, loss:0.66899
epoch:19, iter:500, time:7.04, loss:0.56866
epoch:19, iter:600, time:6.58, loss:0.56538
epoch:19, iter:700, time:6.03, loss:0.67399
[epoch 19 | time:48.90 | loss:0.68658]
-------------------------------------------------
testing accuracy: 0.747
new best model saved at epoch: 19
-------------------------------------------------
best testing accuracy achieved: 0.747
training finished
```