ECSE 551 Group 16 Mini-Project 3

Jonathan Arsenault
260585289

Hung-Yang Chang
260899468

Anan Lu
260467054

Abstract

A convolutional neural network is the preferred machine learning model to perform image classification tasks. In this project, image classification is performed on a modified version of the Fashion-MNIST data set. To obtain the best performing model, several network architectures and optimization hyperparameters were considered. The best results were obtained using a VGG-16 network, which has a history of strong performance on image classification tasks. The weights were trained using the Adam optimization algorithm, with a decreasing learning rate to help convergence to a global minimum. The model submitted to the Kaggle competition achieved a preliminary accuracy of 97.9%.

1 Introduction

Image classification is a well-studied problem in machine learning, with applications ranging from autonomous vehicle navigation to facial recognition. It is a supervised learning task in which the contents of an image are used to assign it a label. Convolutional neural networks (CNNs) are particularly well suited for image classification. Using fully-connected networks for image classification is not recommended as the number of parameters to train grows quickly as the amount of pixels in the image grows. To avoid this, CNNs use filters which slide spatially over the image, meaning the parameters in these filters are shared over the entire image. This allows for a drastic reduction in the amount of parameters to train, while simultaneously training a model which is invariant to the relative position of features in the images themselves.

In this project, a CNN is designed to perform classification on a modified version of the Fashion-MNIST data set. This is essentially a 9 class classification problem. The goal is to maximize the performance of the CNN over a reserved validation set of 10000 images, given a training set of 60000 images. Section 2 provides a more detailed description of the data set. Designing this CNN is treated as a two step process. First, the CNN architecture is chosen. Then, the optimization algorithm used to train the CNN is chosen, and its hyperparameters are tuned. Some iteration is performed to ensure the best possible combination is chosen. The various options considered when trying to determine the best combination are described in Section 3. The experimental results are provided in Section 4, along with the final model. The final model uses a VGG-16 architecture [1], and the weights are trained using the Adam optimization algorithm over 40 epochs with dropout implemented in the fully-connected layers and a decreasing learning rate. This model attained 97.9% accuracy when applied to the Kaggle validation set, and was designed to generalize well to the remaining sample images. The report is concluded in Section 5, where the results are discussed along with possible avenues for future work.

2 Data

The modified Fashion-MNIST data set provided for this project consists of 60000 labelled training images and 10000 unlabelled testing images. Each image is 128 by 64 pixels. The intensity of each pixel values range from 0 to 255. Each training image is labelled with a number ranging from 5 to 13 corresponding to the price of the items in the image. Each class is equally distributed. As is standard for image classification tasks, the intensity of each pixel is normalized such that it ranges from -0.5 to 0.5. The class labels are also modified to range from 0 to 8.

Figure 1: Sample images.

3 Proposed Approach

As with most machine learning algorithms, the design space for a CNN is quasi-infinite. However, extensive research in the field has allowed for a more focused design process. This design process is outlined in this section. For clarity purposes, the design process is divided into two sections. First, the choice of CNN architecture is discussed, followed by the process of selecting the method used to train the model.

3.1 CNN Architecture

For the purposes of this report, the network architecture refers to the size and type of each layer in the CNN. A typical CNN is composed of convolutional layers, pooling layers and fully connected layers. Selecting the quantity and sequencing of these layers is a key aspect of CNN design. The size of the filters, the stride and the padding used in these layers must then be determined. Luckily, there are several standard architectures that have been shown to perform well on image classification tasks, such as VGG networks [1], ResNet [4] and GoogLeNet [5].

As suggested during the couse, a VGG network is initially tested. A VGG network uses very small convolution filters in a series of convolutional layers that systematically decrease in size, but increase in depth. Several types of VGG networks exist, ranging from VGG-11 to VGG-19, where the number refers to the number of weighted layers in the network. The deep nature of these networks makes them well suited to image classification tasks. A more detailed description of the architecture is avaiable in [1]. Preliminary testing revealed that a VGG network achieved promising performance. Therefore, the focus of this report was placed on optimizing the performance of a VGG network on this classification task, and testing of different architectures is relegated to future work.

3.2 Model Training

Choosing the best network architecture is of no use if the methodology used to train the model is inadequate. The optimization algorithm itself plays a large role, as do standard hyperparameters such as learning rate and batch size. Each optimization algorithm also has unique hyperparameters that must be tuned to obtain optimal performance. Finally, techniques such as weight decay and dropout must be implemented to ensure that the model generalizes as well as possible.

PyTorch offers many built-in optimizers. The standard optimizer is stochastic gradient descent (SGD). A crucial parameter in SGD is the learning rate. Choosing a learning rate that is too large may lead to divergence, while a learning rate that is too small will lead to very long training times. Optimization algorithms that automatically tune the learning rate during training are now the preferred choice in machine learning. These so-called adaptive optimization algorithms include Adam, RMSProp and AdaGrad, all of which can be implemented in PyTorch.

Despite the attractive nature of the adaptive learning rate, these optimizers may not generalize well when compared to standard SGD [2]. To further diminish the risk of overfitting most of these algorithms can be implemented with some form of regularization, typically in the form of weight decay. Simply put, weight decay adds a penalty term to the cost function to ensure the weights generally remain small, leading to a less complex model [6, Chapter 5]. Weight decay is not the only method used to decrease the risk of overfitting. Dropout can also be used. Dropout is a regularization method in which the output of a given node in the network is set to zero with probability p. This has the effect of spreading the weights more evenly across the nodes in a given layer, greatly reduce the risk of overfitting. Lastly, to prevent overfitting, the model must be trained for the correct number of epochs. Overtraining the model can lead to better performance on the training set, but the performance on the validation set may begin to suffer.

Selecting the optimal learning rate is also key in achieving optimal performance. Early during the training of the model, larger learning rates are preferred to reach the general area of the optimal value faster. As the training progresses, the learning rate should be decreased to take finer steps until the optimal value is reached. It is therefore common to implement some form of learning rate decay, where the learning rate is decreased either gradually or periodically during training. Even when using adaptive optimizers, it is often preferable to implement learning rate scheduling [2]. A parameter that is closely tied to the learning rate is the batch size. It has been found that a small

batch size is preferable when using small learning rates, while large batch sizes perform better when using large learning rates [3].

The challenge in this project is to determine which combination of optimization algorithm, dropout probability, weight decay parameter, learning rate, batch size, and number of epochs will yield the best performance. In Section 4, many experiments are conducted in an attempt to find the best combination.

4 Results

In this section, experiments are performed to elucidate the choice of CNN architecture and model training parameters, as outlined in Section 3. To offer as fair a comparison as possible, preliminary testing was performed to settle on a baseline model. This model uses VGG-16 with dropout set at $p = 0.25$. Adam is used to determine the weights, with a batch size of 32 and a constant learning rate of 1×10^{-4}. The models were trained for 20 epochs. Using weight decay did not seem to improve the model, and further experiments using weight decay are left to future work. This baseline set of parameters was found to yield satisfactory results, but results which could be improved with additional tuning. Unless stated otherwise, the models were trained on 80% of the data and tested on 20% of the data and the accuracy over the testing set is used as a performance metric. This method is not as robust as a K-fold cross-validation, but is reasonable considering the computation times involved in training these models. The implementation of VGG networks was broadly inspired by the implementation found at [7].

4.1 CNN Architecture

In this section, the impact of the depth of the VGG network is investigated. As mentioned in Section 3, the number of weighted layers, N, in a VGG network is typically 11, 13, 16 or 19. Figure 2 shows the effect of N on the accuracy on the validation set and the time to train each model. VGG-11 clearly performs worst than the deeper CNNs, with VGG-16 having maximal accuracy. The time to train each model seems to increase linearly with N. Ultimately, VGG-16 is determined to offer the best trade-off of performance and training time.

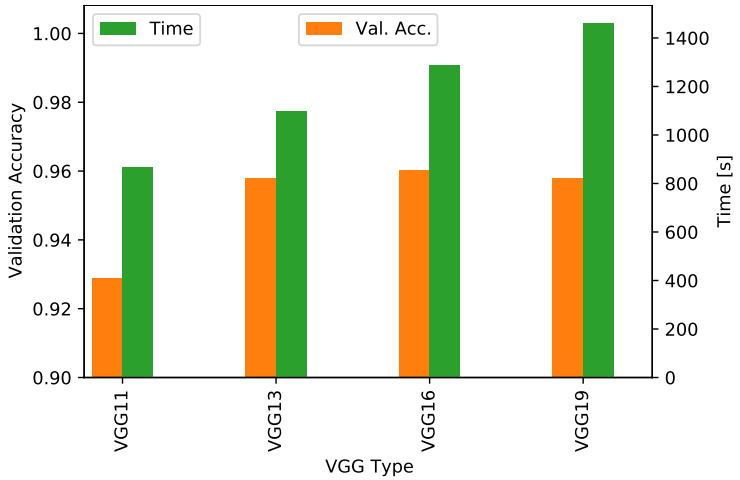

Figure 2: Effect of depth of VGG network on validation accuracy and computation time.

4.2 Selecting the Model Training Parameters

Section 3 outlined the different parameters to be considered for training the weights of the model. Ideally, a grid search would be performed to find the best possible combination of parameters. However, this is not feasible given the time required to train each CNN. In this section, selected results are presented to guide the selection of the required parameters.

4.2.1 Optimization Algorithm

Four different optimization algorithms are tested. They are SGD with momentum (SGD-M), SGD with Nesterov momentum (SGD-NM), Adam, and AdamW, Adam with decoupled weight decay regularization. These were chosen

as they yielded the best results in preliminary testing, and have historically been used as benchmark algorithms. Standard SGD without momentum did not converge in a reasonable amount of time. Momentum was added to accelerate the convergence. The value of the momentum had to be high to yield satisfying results. It was therefore set at 0.99 in both cases. The results are shown in Figure 3. Both Adam variants offered similar performance while significantly reducing the time required to train the model when compared to SGD with momentum.

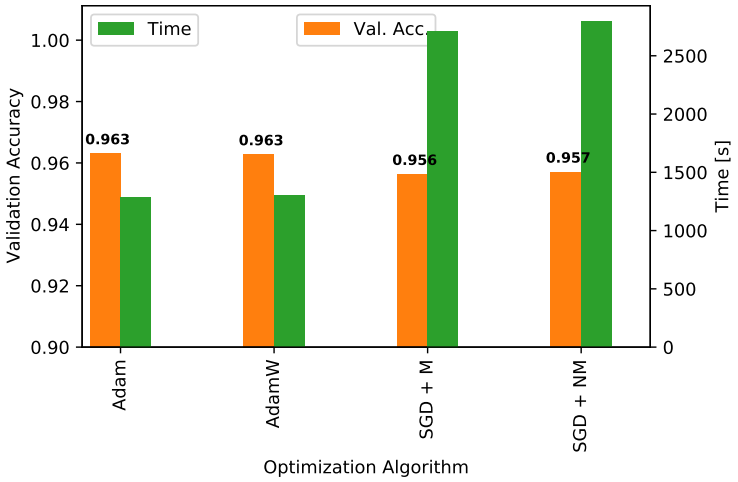

Figure 3: Effect of optimization algorithm on validation accuracy and computation time.

4.2.2 Batch Size and Learning Rate

As discussed in [3], batch size and learning rate jointly effect the performance of CNNs. To replicate this, while determining which combination of batch size and learning rate leads to the best results, a grid search was performed. The results are presented in Figure 5. For the purposes of this exercise, the learning rate is kept constant throughout. The best results are seen at a batch size of 32 and learning rate of 10^{-4}. Indeed, for the smallest learning rate, the smallest batch size leads to the best results, while for the largest learning rate, the largest batch size leads to the best results.

4.2.3 Learning Rate Scheduling

To maximize the performance, the learning rate should be varied during training. This helps escape local minima. Many forms of learning rate scheduling are possible. In the report, focus has been put on 2 types of learning rate scheduling. In the first, the learning rate varies between reasonable boundaries. These boundaries can also be adjusted during training. This is known as a cyclical learning rate. The other option is to reduce the learning rate when a given error metric has stopped improving. This is referred to as a plateau-based learning rate.

Three different types of cyclical learning rates were tested, and all yielded results inferior to that of the plateau-based learning rate techniques. Similarly, three of these superior techniques were tested. In the first, the learning rate was multiplied by a factor of 0.1 when the accuracy on the validation set stopped improving or every 10 epochs. The amount of epochs to wait before decreasing the learning rate, regardless of improvement, is the patience of the scheduler. In the second, the multiplication factor was increased to 0.9 and the patience was decreased to 4. The final test used a factor of 0.5 and a patience of 4. Of these three options, the first option had the lowest loss and highest training and validation accuracy. This reveals that the learning rate must be significantly reduced later in the training phase to ensure optimal performance. Decreasing the learning rate too little or too quickly leads to convergence to local minima.

4.2.4 Dropout

The final parameter that is discussed is the dropout. Dropout is following the fully connected layers. The tunable parameter is p, the probability that the output of a given node is zeroed. As dropout is a regularization technique, it should decrease the difference between the training loss and the validation loss. These experiments were run using 5-fold cross-validation, to minimize the effect of randomness. Five different values for p were tested. The effect of p was minimal. The training and validation accuracy only truly began to suffer at $p = 0.5$. Figure 4,

however, demonstrates the there is a range of p that will minimize the variance in the model, as approximated by the difference between the training accuracy and validation accuracy. This range seems to be near $p = 0.3$.

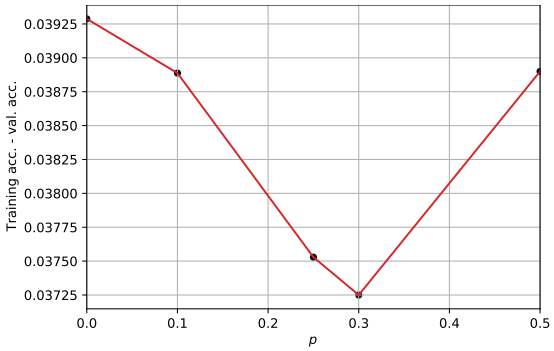

Figure 4: Effect of dropout on model variance.

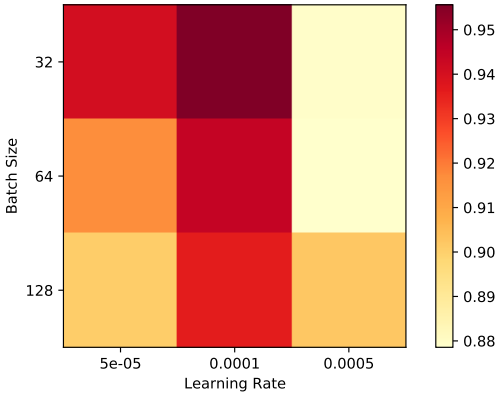

Figure 5: Effect of batch size and learning rate on validation accuracy.

4.3 Best Performing Model

Combining the best performing option from each experiment, while an imperfect method, should yield the best performing model. After trying dozens of additional combinations, the best performing combination ended up being a VGG-16 network trained using the Adam optimization algorithm with a batch size of 32 and dropout with $p = 0.5$. The learning rate was set to 1×10^{-4} for 20 epochs of training and 5×10^{-6} for another 20 epochs. From the experiments, $p = 0.3$ seems like it would have been a better option for the dropout parameter. Additionally, instead of manually decreasing the learning rate, a plateau-based method should have yielded better performance according to the learning rate scheduling tests. In the end, these parameters, while performing well in validation, did not generalize well to the validation set available on Kaggle. As optimizing accuracy over this Kaggle validation set is the ultimate goal, the parameters which maximized this value were chosen.

In the end, several combinations yielded approximately $\sim$98% accuracy on the Kaggle validation set. The tight margins between all the tested models further complicated the decision making. The chosen model described above provided 97.9% accuracy on the Kaggle validation set. This is not the highest accuracy obtained out of all the samples tested, but the high value for the dropout parameter ($p = 0.5$) should provide better generalization than some of the other models tested.

5 Discussion and Conclusion

In this report, a CNN was designed to perform classification on a modified version of the Fashion-MNIST data set. Several experiments were performed to gain insight into the CNN architecture and the method used to train the model. The experiments revealed general trends concerning learning rate, batch size and dropout that led to the best final model. A VGG-16 network with dropout using the Adam optimization algorithm to train the weights yielded the best result after tuning the additional parameters. This CNN provided 97.9% percent accuracy on the Kaggle validation set.

Despite the decidedly strong performance of the final CNN, several other possible avenues exist for future work. Investigating weight decay regularization could be beneficial to improve model generalization. Other widely used CNN architectures, such as ResNet and GoogLeNet, should be tested. Incorporating other error metrics into the analysis, such as loss, precision, recall and F-score should also be considered, as it might yield insights into the weaknesses of the current model.

6 Contribution Statement

1. Jonathan Arsenault: Experimental design, report.
2. Hung-Yang Chang: Experimental design, software implementation.
3. Anan Lu: Experimental design, software implementation.

References

[1] K. Simonyan & A. Zisserman, "Very Deep Convolutional Network for Large-Scale Image Recognition", in *International Conference on Learning Representations*, 2015.

[2] A.C. Wilson, R. Roelofs M. Stern, B. Recht & N. Srebro, "The Marginal Value of Adaptive Gradient Methods in Machine Learning", in *Neural Information Processing Systems (NIPS)*, 2015.

[3] I. Kandel & M. Castelli, "The Effect of Batch Size on the Generalizability of the Convolutional Neural Networks on a Histopathology Dataset", *ICT Express*, vol. 6, no. 4, pp. 312-315, 2020.

[4] K. He, X. Zhang, S. Ren & J. Sun, "Deep Residual Learning for Image Recognition", *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[5] C. Szegedy et al., "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.

[6] I. Goodfellow, Y. Bengion & A. Courville, *Deep Learning*, MIT Press, 2016.

[7] A. Persson, "A from scratch implementation of the VGG architecture", [Online] Available: https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/CNN_architectures/pytorch_vgg_implementation.py, [Accessed: 06-Dec-2020].

7 Appendix

Code1:miniproject3.py

```
# -*- coding: utf-8 -*-
"""Mini_project3_clean.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Om3obsG3LI_4Iz3Tu2cePcuwLUq_8ce3
"""

import torch

# Check if using GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if torch.cuda.is_available():
  print(f"Nvidia Cuda/GPU is available!")

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
drive.mount('/content/gdrive' )
# %cd '/content/gdrive/MyDrive/Colab/ECSE551Miniproject/Mini-Project3'

! [ ! -z "$COLAB_GPU" ] && pip install torch scikit-learn==0.21.* skorch

import pickle
import matplotlib.pyplot as plt
import numpy as np
import time
import pandas as pd


import torch.nn.functional as F
import torch.optim as optim

from sklearn.model_selection import train_test_split
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
from skorch import NeuralNetClassifier
from skorch.dataset import CVSplit

# tuning learning rate
from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau
import torch.nn as nn

"""# Set parameter

"""

# hyperparameter used later
batch_size = 32
```

```
drop_ratio = 0.5
kernel_size_CNN = (3, 3)
learning_rate = 1e-4
weight_decay = 0


# file name for saving model
save_file_name = 'model_trained_VGG_16_0.25_dropout.tar'
save_file_name_skroch = 'model_trained_VGG_16_0.25_dropout_skorch.tar'

"""# Load Data

you may not be able to run %cd
    '/content/gdrive/MyDrive/Colab/ECSE551Miniproject/Mini-Project3', you need to connect to
    your own path with training and testing dataset
"""

# Read a pickle file and disply its samples
# image data are stored as unit8 so each element is an integer value between 0 and 255
data = pickle.load( open( './Train.pkl', 'rb' ), encoding='float32')
targets = np.genfromtxt('./TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]-5
plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)
print(data.shape, targets.shape)

# normalize images img=(img-mean)/std
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),
])

# normalize images img=(img-mean)/std with data agument
img_transform_augment = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomHorizontalFlip(p=1),
    transforms.Normalize([0.5], [0.5]),
    transforms.RandomRotation(13)
])

# define some classs for reading out data

class MyDataset(Dataset):

    def __init__(self, img_file, label_file, transform=None, idx = None):
      """function load training or testing data"""

      # read out img_file (.pkl)
      self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')

      # read out label_file (.csv)
      # convert label by -5
      self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:]-5

      if idx is not None:
        # idx: binary vector for creating training and validation set.
        # Only return samples where idx is not None

        # set idx for target
        self.targets = self.targets[idx]
        # set idx for data
        self.data = self.data[idx]

      # transform to normalize images
      self.transform = transform

    def __len__(self):
```

```
      """function to get len of target"""

      return len(self.targets)

    def __getitem__(self, index):
      """function to get img and target"""

      # get img and target
      img, target = self.data[index], int(self.targets[index])
      img = Image.fromarray(img.astype('uint8'), mode='L')

      # doing transform to normalize images
      if self.transform is not None:
        img = self.transform(img)

      return img, target

class TestDataset(Dataset):

    def __init__(self, img_file, transform=None, idx = None):
      """function load training or testing data"""

      # read out img_file (.pkl)
      self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')

      if idx is not None:
        # idx: binary vector for creating training and validation set.
        # Only return samples where idx is not None

        # set idx for data
        self.data = self.data[idx]

      # transform to normalize images
      self.transform = transform

    def __len__(self):
      """function to get len of target"""

      return len(self.data)

    def __getitem__(self, index):
      """function to get img and target"""

      # get img and target
      img = self.data[index]
      img = Image.fromarray(img.astype('uint8'), mode='L')

      # doing transform to normalize images
      if self.transform is not None:
        img = self.transform(img)

      return img

# Read image data and their label into a Dataset class
# Random split by training =0.8 and testing = 0.2

idx_train, idx_test = train_test_split(np.arange(60000) , test_size=0.2, random_state=42)

# Read data from pkl and label in csv
data_train = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=idx_train)
data_test = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform, idx=idx_test)

# Read data from pkl and label with data augment in csv
```

```
data_train_augment = MyDataset('./Train.pkl',
    './TrainLabels.csv',transform=img_transform_augment, idx=idx_train)
data_test_augment = MyDataset('./Train.pkl',
    './TrainLabels.csv',transform=img_transform_augment, idx=idx_test)

# Read Test data (10000)
test_final = TestDataset('./Test.pkl',transform=img_transform, idx=np.arange(10000))

train_loader = DataLoader(data_train, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(data_test, batch_size=batch_size, shuffle=False)
test_final_loader = DataLoader(test_final, batch_size=batch_size, shuffle=False)


examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)

# showing data and target shape
print(example_data.shape)
print(example_targets.shape)
print(example_targets)


imgs, labels = (next(iter(train_loader)))
imgs = np.squeeze(imgs)
plt.imshow(imgs[3].cpu().numpy(),cmap='gray', vmin=-1, vmax=1)

train_loader_augment = DataLoader(data_train_augment, batch_size=batch_size, shuffle=True)
test_loader_augment = DataLoader(data_test_augment, batch_size=batch_size, shuffle=False)

examples = enumerate(train_loader_augment)
batch_idx, (example_data, example_targets) = next(examples)

# showing data and target shape with data augment
print(example_data.shape)
print(example_targets.shape)
print(example_targets)

imgs, labels = (next(iter(train_loader_augment)))
imgs = np.squeeze(imgs)
plt.imshow(imgs[3].cpu().numpy(),cmap='gray', vmin=-1, vmax=1)

"""# Define Function
Define train and test

"""

def train(epoch, network, loss_mode, train_loader):
  """function for training neural network"""

  # in training mode
  network.train()

  # parameter to calculate training accuracy
  train_correct = 0
  train_total = 0
  train_loss = 0

  # optimizer = torch.optim.AdamW(network.parameters(), lr=learning_rate, betas=(0.9, 0.999),
      eps=1e-08, weight_decay=0.01, amsgrad=False)
  # scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

  for batch_idx, (data, target) in enumerate(train_loader):


    # transfer data to cuda if available
```

```
    data = data.to(device)
    target = target.to(device)

    # zero grad after tuning one batch
    optimizer.zero_grad()

    # send data to network
    output = network(data)

    # loss model: nll=>negative log liklhood loss, CE=>cross-entropy, nllTA = TA's negative log
        liklhood loss
    if (loss_mode == "nll"):
      error = nn.NLLLoss()
      loss = error(output, target)
    elif (loss_mode == "CE"):
      error = nn.CrossEntropyLoss()
      loss = error(output, target)
    elif (loss_mode == "nllTA"):
      loss = F.nll_loss(output, target)


    # backpropogation
    loss.backward()
    # Gardient Descent
    optimizer.step()

    # adding training loss
    train_loss += loss.item()

    # save correct training label
    pred = output.data.max(1, keepdim=True)[1]
    train_correct += pred.eq(target.data.view_as(pred)).sum()
    train_total += batch_size

    # print out information (epoch, loss, and saving accuracy every 20 batch
    if batch_idx % 150 == 0:
      print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
        epoch, batch_idx * len(data),
        len(train_loader.dataset),
        100. * batch_idx / len(train_loader),
        loss.item()))

      print('Accuracy: {}/{} ({:.0f}%)\n'.format(
      train_correct, train_total,
      100. * train_correct / train_total))

    ## Decay learning rate if needed. put into for loop
    # scheduler.step()

  train_loss /= len(train_loader)

  # print out information (loss, accuarcy)
  print('\nTraining set: Avg. loss: {:.4f}\n'.format(train_loss))
  print("current learning rate: \n", optimizer.param_groups[0]['lr'])

def test(network, loss_mode, test_loader):
  """function for testing neural network"""

  # in testing mode
  network.eval()

  # parameter to calculate testing accuracy
  test_loss = 0
  correct = 0
```

```
  with torch.no_grad():
    for data, target in test_loader:

      # transfer data to cuda if available
      data = data.to(device)
      target = target.to(device)

      # send data to network
      output = network(data)

      # loss model: nll=>negative log liklhood loss, CE=>cross-entropy
      # store loss

      if (loss_mode == "nll"):
        error = nn.NLLLoss()
        test_loss += error(output, target).item()
      elif (loss_mode == "CE"):
        error = nn.CrossEntropyLoss()
        test_loss += error(output, target).item()
      elif (loss_mode == "nllTA"):
        test_loss += F.nll_loss(output, target).item()

      # save correct testing label
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()

  # calculate loss
  test_loss /= len(test_loader.dataset)

  # print out information (loss, accuarcy)
  print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))

"""# Define Network: VGG"""

"""
Code Reference:
    https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/CNN_architectures
"""

# declare number of input and class label
num_in =1
num_class = 9

VGG_types = {
    "VGG11": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "VGG13": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "VGG16": [
        64,
        64,
        "M",
        128,
        128,
        "M",
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        "M",
        512,
        512,
```

```
        512,
        "M",
    ],
    "VGG19": [
        64,
        64,
        "M",
        128,
        128,
        "M",
        256,
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        512,
        "M",
        512,
        512,
        512,
        512,
        "M",
    ],
}

# VGG_net class inherits from nn.Module class
class VGG_net(nn.Module):

    def __init__(self, in_channels=num_in, num_classes=num_class, VGG_type =
        VGG_types["VGG16"]):
      """constructor"""

      # Call the __init__() function of nn.Module class
      super(VGG_net, self).__init__()

      # creating neural network
      # (1) convolutional neural network
      self.in_channels = in_channels
      self.conv_layers = self.create_conv_layers(VGG_type)

      # (2) fully connected neural network

      self.fcs = nn.Sequential(
          nn.Linear(4096, 4096),
          nn.ReLU(),
          nn.Dropout(p = drop_ratio),
          nn.Linear(4096, 4096),
          nn.ReLU(),
          nn.Dropout(p = drop_ratio),
          nn.Linear(4096, num_classes),
      )

    def create_conv_layers(self, architecture):
      """function help to create convolutional neural network"""

      # list for layers
      layers = []

      # define number of input channel
      in_channels = self.in_channels

      # for loop to construct convolutional neural network according to architecture
```

```
        for x in architecture:
            if type(x) == int:
                # create output channel number (when it is not Maxpooling)
                out_channels = x

                # add layer with parameter with convolutional neural network
                layers += [
                    nn.Conv2d(
                        in_channels=in_channels,
                        out_channels=out_channels,
                        kernel_size = kernel_size_CNN,
                        stride=(1, 1),
                        padding=(1, 1),
                    ),

                    # Batchnorm to improve performance
                    nn.BatchNorm2d(x),
                    nn.ReLU(),
                ]

                # input channel for next input
                in_channels = x

            elif x == "M":
              # Maxpooling
              layers += [nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))]

        return nn.Sequential(*layers)

    def forward(self, x):
      """create forward path for VGG"""

      # convolutional neural network
      x = self.conv_layers(x)
      x = x.reshape(x.shape[0], -1)

      # fully connected neural network
      x = self.fcs(x)

      # softmax at the last output layer to get result
      # m = nn.Softmax(dim = 1)
      # return m(x)

      # TA
      return F.log_softmax(x)

"""# Loading/building neural network"""

# flag to check loading or not
Loading = False
# flag to check loading for training or loading for testing
Loading_for_training = True

if (Loading):
  #  initalization network and optimizer before initalization
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)
  # todo adamw
  optimizer = torch.optim.AdamW(network.parameters(), lr=learning_rate, betas=(0.9, 0.999),
      eps=1e-08, weight_decay=weight_decay, amsgrad=False)
  checkpoint = torch.load("model_trained_VGG_16_1127_skorch.tar_0.25_dropout.tar")

  # load network and optimizer (these two is Must load!), epoch and loss (optional)
  network.load_state_dict(checkpoint['model_state_dict'])
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

```
  # epoch = checkpoint['epoch']
  # loss = checkpoint['loss']

  if (Loading_for_training):
    # load model for continuing training
    network.train()
  else:
    # inferenece
    network.eval()

else:
  #  training network from scratch
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)
  optimizer = torch.optim.AdamW(network.parameters(), lr=learning_rate, betas=(0.9, 0.999),
      eps=1e-08, weight_decay=weight_decay, amsgrad=False)

# check network
print(network)

train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in range(3)]

"""# Model Trainig Method 1
Use Our own train/test function

"""

learning_rate = 1e-4
optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate)
# optimizer = torch.optim.AdamW(network.parameters(), lr=learning_rate, betas=(0.9, 0.999),
    eps=1e-08, weight_decay=0, amsgrad=False)

# record start time
startall = time.time()

print("inital condition")
print("lr: ", learning_rate)

for epoch in range(1, 20):

  # record each epoch start time
  start_epoch = time.time()

  # doing cross-validation each epoch
  idx_train, idx_test = train_test_split(np.arange(58000) , test_size=0.2)

  # Read data from pkl and label in csv
  data_train = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
      idx=idx_train)
  data_test = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
      idx=idx_test)

  train_loader = DataLoader(data_train, batch_size=batch_size, shuffle=True)
  test_loader = DataLoader(data_test, batch_size=batch_size, shuffle=False)

  # # Read data from pkl and label in csv with augment
  # data_train_augment = MyDataset('./Train.pkl',
      './TrainLabels.csv',transform=img_transform_augment, idx=idx_train)
  # data_test_augment = MyDataset('./Train.pkl',
      './TrainLabels.csv',transform=img_transform_augment, idx=idx_test)

  # train_loader_augment = DataLoader(data_train_augment, batch_size=batch_size, shuffle=True)
```

```
  # test_loader_augment = DataLoader(data_test_augment, batch_size=batch_size, shuffle=False)

  ## training
  # print ("================================================ augment
      ==========================================")
  # train(epoch, network, loss_mode="nllTA", train_loader = train_loader_augment)
  print ("================================================ without augment
      ==========================================")
  train(epoch, network, loss_mode="nllTA", train_loader = train_loader)

  # testing
  test(network, loss_mode="nllTA", test_loader = test_loader)

  print(f"Execution Time For each epoch: {time.time()-start_epoch:.2f} s\n")

print(f"Execution Time: {time.time()-startall:.2f} s")

data_test = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(58001,60000))
test_loader = DataLoader(data_test, batch_size=batch_size, shuffle=False)

test(network, loss_mode="nllTA", test_loader = test_loader)

save_file_name_skroch = 'batch16.tar'

torch.save({
      'model_state_dict': network.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
      }, save_file_name_skroch, _use_new_zipfile_serialization=False)

print ("save network! file name: ", save_file_name_skroch)

# flag to check loading or not
Loading = True
# flag to check loading for training or loading for testing
Loading_for_training = True

print (save_file_name_skroch)

if (Loading):
  #  initalization network and optimizer before initalization
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)
  # todo adamw
  optimizer = torch.optim.AdamW(network.parameters(), lr=learning_rate, betas=(0.9, 0.999),
      eps=1e-08, weight_decay=weight_decay, amsgrad=False)
  checkpoint = torch.load(save_file_name_skroch)

  # load network and optimizer (these two is Must load!), epoch and loss (optional)
  network.load_state_dict(checkpoint['model_state_dict'])
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
  # epoch = checkpoint['epoch']
  # loss = checkpoint['loss']

  if (Loading_for_training):
    # load model for continuing training
    network.train()
  else:
    # inferenece
    network.eval()

print (network)

"""# Model Trainig Method 2
Use skorch
```

```
"""

sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))
y_data = np.array([y for x, y in iter(dataset)])

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau
from skorch.callbacks import Callback

network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
    VGG_types["VGG16"]).to(device)

torch.manual_seed(0)
learning_rate = 1e-111
weight_decay = 0


# ref: https://github.com/skorch-dev/skorch/issues/505
# Monitor to check learning rate
class Monitor(Callback):
    def on_epoch_end(self, network, **kwargs):
      print("current learning rate: ", network.optimizer_.param_groups[0]['lr'])


#  callbacks function to check and change lr
callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                   LRScheduler(policy=ReduceLROnPlateau,
                   monitor = "train_loss",
                  #  factor = 0.5,
                  #  patience = 10,
                  #  min_lr = learning_rate*0.01
                   ))
    ]

cnn = NeuralNetClassifier(
    network,
    max_epochs = 10,
    lr= learning_rate,

    criterion = torch.nn.NLLLoss,
    # criterion= torch.nn.CrossEntropyLoss,

    # AdamW
    optimizer = torch.optim.AdamW,
    optimizer__weight_decay=0.01,

    # SGD
    # optimizer = torch.optim.SGD,
    # optimizer__weight_decay = weight_decay,
    # optimizer__momentum = 0.9,
    # optimizer__nesterov = True,

    batch_size = 32,
    device=device,
    iterator_train__num_workers = 4,
    iterator_valid__num_workers = 4,
    callbacks=callbacks,
)

print("inital condition")
print("lr: ", learning_rate, "weight_decay: ", weight_decay)
```

```
# record start time
startall = time.time()

# training
cnn.fit(dataset, y=y_data)

print(f"Execution Time: {time.time()-startall:.2f} s")

learning_rate = 5e-4
weight_decay = 2e-4

#  callbacks function to check and change lr
callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                  LRScheduler(policy=ReduceLROnPlateau,
                  monitor = "train_loss",
                  factor = 0.8,
                  patience = 10,
                  min_lr = learning_rate*0.01
                  ))
   ]


cnn = NeuralNetClassifier(
    network,
    max_epochs = 100,
    lr= learning_rate,

    # # AdamW
    # optimizer = torch.optim.AdamW,
    # optimizer__weight_decay=weight_decay,

    # SGD
    optimizer = torch.optim.SGD,
    optimizer__weight_decay = weight_decay,
    optimizer__momentum = 0.99,
    optimizer__nesterov = True,

    batch_size = 32,
    device=device,
    iterator_train__num_workers = 4,
    iterator_valid__num_workers = 4,
    callbacks=callbacks,
)

print("inital condition")
print("lr: ", learning_rate, "weight_decay: ", weight_decay)

# record start time
startall = time.time()

# training
cnn.fit(dataset, y=y_data)

print(f"Execution Time: {time.time()-startall:.2f} s")

save_file_name_skroch = 'high2.tar'

print ("save network \n")
torch.save({
     'model_state_dict': network.state_dict(),
     'optimizer_state_dict': optimizer.state_dict(),
     }, save_file_name_skroch, _use_new_zipfile_serialization=False)
```

```
learning_rate = 0.0001
weight_decay = 2e-4

#  callbacks function to check and change lr
callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                   LRScheduler(policy=ReduceLROnPlateau,
                   monitor = "train_loss",
                   factor = 0.8,
                   patience = 10,
                   min_lr = learning_rate*0.01
                   ))
   ]


cnn = NeuralNetClassifier(
   network,
   max_epochs = 100,
   lr= learning_rate,

   # # AdamW
   # optimizer = torch.optim.AdamW,
   # optimizer__weight_decay=weight_decay,

   # SGD
   optimizer = torch.optim.SGD,
   optimizer__weight_decay = weight_decay,
   optimizer__momentum = 0.99,
   optimizer__nesterov = True,

   batch_size = 32,
   device=device,
   iterator_train__num_workers = 4,
   iterator_valid__num_workers = 4,
   callbacks=callbacks,
)

print("inital condition")
print("lr: ", learning_rate, "weight_decay: ", weight_decay)

# record start time
startall = time.time()

# training
cnn.fit(dataset, y=y_data)

print(f"Execution Time: {time.time()-startall:.2f} s")

print (save_file_name_skroch)
# flag to check loading or not
Loading = True
# flag to check loading for training or loading for testing
Loading_for_training = True

if (Loading):
  #  initalization network and optimizer before initalization
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)
  optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate)
  checkpoint = torch.load(save_file_name_skroch)

  # load network and optimizer (these two is Must load!), epoch and loss (optional)
  network.load_state_dict(checkpoint['model_state_dict'])
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

```
  if (Loading_for_training):
    # load model for continuing training
    network.train()

"""# Test and Save data

"""

test_final_loader = DataLoader(test_final, batch_size=batch_size, shuffle=False)
y_pred = []

def pred(testdata = test_final_loader):
    # evaluation mode
    network.eval()

    test_loss = 0
    correct = 0

    with torch.no_grad():
        for data in testdata:
          # transfer data to cuda if available
          data = data.to(device)

          output = network(data)
          pred = output.data.max(1, keepdim=True)[1] + 5
          # print (pred.shape)
          y_pred.append(pred.tolist())

    y_pred_final = [item for sublist in y_pred for item in sublist]
    y_pred_final2 = [item for sublist in y_pred_final for item in sublist]
    return y_pred_final2


def store_csv(data, outfile_name):
  rawdata= {'id': np.arange(10000),'class':data}
  a = pd.DataFrame(rawdata, columns = ['id','class'])
  a.to_csv(outfile_name,index=False, header=True)
  print ("store finsih!")

# record start time
startall = time.time()
y_pred = pred(testdata = test_final_loader)

print(f"Execution Time: {time.time()-startall:.2f} s")

store_csv(y_pred,"submit.csv")
```

Code2:miniproject_3supplemental.py

```
# -*- coding: utf-8 -*-
"""miniproj3_supplementary_results.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ixee_8wHVEwH-_5_cKQwbu4FK1xEU3Gp
"""

! [ ! -z "$COLAB_GPU" ] && pip install torch scikit-learn==0.21.* skorch

import pickle
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import torch
import time
import pandas as pd

from itertools import islice
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.metrics import log_loss, make_scorer
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from skorch import NeuralNetClassifier
from skorch.dataset import CVSplit

from skorch.helper import SliceDataset

# Check CPU/GPU
USE_CUDA = 0
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if torch.cuda.is_available():
  USE_CUDA = 1
  print(f"Nvidia Cuda/GPU is available!")

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)

"""# Load Data"""

from google.colab import drive
drive.mount('/content/gdrive' )

# Commented out IPython magic to ensure Python compatibility.
# can't upload input data files to Github because it's too big?
# so each of us will need to change this to where we store data on google drive
```

```
# %cd '/content/gdrive/MyDrive/Colab/ECSE551Miniproject/Mini-Project3'

# Read a pickle file and disply its samples
# Note that image data are stored as unit8 so each element is an integer value between 0 and 255
data = pickle.load( open( './Train.pkl', 'rb' ), encoding='float32')
targets = np.genfromtxt('./TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]-5
plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)
print(data.shape, targets.shape)

# Transforms are common image transformations. They can be chained together using Compose.
# Here we normalize images img=(img-0.5)/0.5
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

class MyDataset(Dataset):

    def __init__(self, img_file, label_file, transform=None, idx = None):
      """function load training or testing data"""

      # read out img_file (.pkl)
      self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')

      # read out label_file (.csv)
      self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:]-5

      if idx is not None:
        # idx: binary vector for creating training and validation set.
        # Only return samples where idx is not None

        # set idx for target
        self.targets = self.targets[idx]
        # set idx for data
        self.data = self.data[idx]

      # transform to normalize images
      self.transform = transform

    def __len__(self):
      """function to get len of target"""

      return len(self.targets)

    def __getitem__(self, index):
      """function to get img and target"""

      # get img and target
      img, target = self.data[index], int(self.targets[index])
      img = Image.fromarray(img.astype('uint8'), mode='L')

      # doing transform to normalize images
      if self.transform is not None:
        img = self.transform(img)

      return img, target

class TestDataset(Dataset):

    def __init__(self, img_file, transform=None, idx = None):
      """function load training or testing data"""

      # read out img_file (.pkl)
      self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
```

```
      if idx is not None:
        # idx: binary vector for creating training and validation set.
        # Only return samples where idx is not None

        # set idx for data
        self.data = self.data[idx]

      # transform to normalize images
      self.transform = transform

    def __len__(self):
      """function to get len of target"""

      return len(self.data)

    def __getitem__(self, index):
      """function to get img and target"""

      # get img and target
      img = self.data[index]
      img = Image.fromarray(img.astype('uint8'), mode='L')

      # doing transform to normalize images
      if self.transform is not None:
        img = self.transform(img)

      return img

# Read image data and their label into a Dataset class
# Test a portion!! by changing idx!!
sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))

# Read Test data (10000)
test_final = TestDataset('./Test.pkl',transform=img_transform, idx=np.arange(10000))

# prepare y_data for sklearn to work
y_data = np.array([y for x, y in iter(dataset)])

"""# Fit a CNN"""

# Imports
"""
Code Reference:
    https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/CNN_architectures
"""

# hyperparameter
drop_ratio = 0.25
kernel_size_CNN = (3,3)

# declare number of input and class label
num_in =1
num_class = 9

VGG_types = {
    "VGG11": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "VGG13": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "VGG16": [
        64,
        64,
        "M",
        128,
        128,
```

```
        "M",
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        "M",
        512,
        512,
        512,
        "M",
    ],
    "VGG19": [
        64,
        64,
        "M",
        128,
        128,
        "M",
        256,
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        512,
        "M",
        512,
        512,
        512,
        512,
        "M",
    ],
}

# VGG_net class inherits from nn.Module class
class VGG_net(nn.Module):

    def __init__(self, in_channels=num_in, num_classes=num_class, VGG_type =
        VGG_types["VGG16"]):
      """constructor"""

      # Call the __init__() function of nn.Module class
      super(VGG_net, self).__init__()

      # creating neural network
      # (1) convolutional neural network
      self.in_channels = in_channels
      self.conv_layers = self.create_conv_layers(VGG_type)

      # (2) fully connected neural network
      self.fcs = nn.Sequential(
          nn.Linear(4096, 4096),
          nn.ReLU(),
          nn.Dropout(p = drop_ratio),
          nn.Linear(4096, 4096),
          nn.ReLU(),
          nn.Dropout(p = drop_ratio),
          nn.Linear(4096, num_classes),
      )
```

```
    def create_conv_layers(self, architecture):
      """function help to create convolutional neural network"""

      # list for layers
      layers = []

      # define number of input channel
      in_channels = self.in_channels

      # for loop to construct convolutional neural network according to architecture
      for x in architecture:
          if type(x) == int:
              # create output channel number (when it is not Maxpooling)
              out_channels = x

              # add layer with parameter with convolutional neural network
              layers += [
                  nn.Conv2d(
                      in_channels=in_channels,
                      out_channels=out_channels,
                      kernel_size = (3,3),
                      stride=(1, 1),
                      padding=(1, 1),
                  ),

                  # Batchnorm to improve performance
                  nn.BatchNorm2d(x),
                  nn.ReLU(),
              ]

              # input channel for next input
              in_channels = x

          elif x == "M":
            # Maxpooling
            layers += [nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))]

      return nn.Sequential(*layers)

    def forward(self, x):
      """create forward path for VGG"""

      # convolutional neural network
      x = self.conv_layers(x)
      x = x.reshape(x.shape[0], -1)

      # fully connected neural network
      x = self.fcs(x)

      # softmax at the last output layer to get result
      m = nn.Softmax(dim = 1)
      return m(x)

#load model from previous tests or train a new model

Loading_flag = False

if (Loading_flag):
  #  initalization
  network = VGG_net().to(device)
  optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate)
  checkpoint = torch.load("./model_trained_1126.tar")

  # load network
  network.load_state_dict(checkpoint['model_state_dict'])
```

```
  # load optimizer
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

  epoch = checkpoint['epoch']
  loss = checkpoint['loss']
  network.train()
else:
  network = VGG_net().to(device)
  #optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate)


print (network)

#setting up skorch NeuralNetClassifier with Callbacks

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau

from skorch.callbacks import Callback

class Monitor(Callback):
    def on_epoch_end(self, net, **kwargs):
        print('learning rate =',net.optimizer_.param_groups[0]['lr'])

callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                   LRScheduler(policy=ReduceLROnPlateau,
                   monitor = "train_loss",
                   ))
    ]

drop_ratio=0
kernel_size_CNN=(4,4)

torch.manual_seed(0)
network = VGG_net().to(device)

print (device)
cnn = NeuralNetClassifier(
    network,
    max_epochs=20,
    lr=1e-4,
    optimizer=torch.optim.Adam,
    batch_size=32,
    device=device,
    iterator_train__num_workers=4,
    iterator_valid__num_workers=4,
    callbacks=callbacks,
    train_split=CVSplit(3)
)

#fit a VGG net with skorch

import time

#fit
startall = time.time()
cnn.fit(dataset, y=y_data)
endall = time.time()

print(f"Execution Time: {endall-startall:.2f} s")

# save trained model
```

```
print ("save network \n")
torch.save({
      'model_state_dict': network.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
      }, 'model_trained_VGG_16_lr1e-4_80epochs_lr-5_20epochs_lr1e-6_15epochs_1129.tar',
          _use_new_zipfile_serialization=False)

"""# Grid Search

Grid search template with skorch. We did not use these for tests in the end, due to very long
    training time.
"""

from sklearn.model_selection import GridSearchCV
from skorch.helper import SliceDataset

cnn.set_params(max_epochs=5, verbose=False, train_split=False, callbacks=[])

#parameters to grid search
uni_kernel_test = [3, 5, 7]

params = {
    'module_uni_kernel': uni_kernel_test,
}

cnn.initialize();

gs = GridSearchCV(cnn, param_grid=params, scoring='accuracy', verbose=1, cv=3)

data_train_sliceable = SliceDataset(dataset)

gs.fit(data_train_sliceable, y_data)

# Plot results
parameter_plot = uni_kernel_test

save_fig = False
fig, ax = plt.subplots()
ax.plot(parameter_plot, gs.cv_results_["mean_test_score"])
#ax.set_xscale("log")
ax.set_xlabel(r"$\kernel size$")
ax.set_ylabel("Mean Test Score")
ax.grid()

ax2 = ax.twinx()
ax2.plot(parameter_plot, gs.cv_results_["mean_fit_time"], color="r", linestyle="--")
ax2.set_ylabel("Mean Fit Time [s]")

if save_fig:
  plt.tight_layout()
  plt.savefig("alpha_grid_search.pdf", bbox_inches="tight", pad_inches=0)

cnn.optimizer_.param_groups[0]['lr']

"""# Test batch size and learning rate

Fixed parameters:
- VGG-16
- drop ratio = 0.25
- Optimizer = Adam, no weight decay
- epochs = 20

Testing parameters:
- batch size = 32, 64, 128
- learning rate = 5e-5, 1e-4, 5e-4 (fixed)
```

```
- find corrolation of these two parameters
"""

batch_size_pool = [32, 64, 128]
learning_rate_pool = [5e-5, 1e-4, 5e-4]
batch_size = []
learning_rate = []
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []
callbacks = []

for b_size in batch_size_pool:

  for l_r in learning_rate_pool:

    network = VGG_net().to(device)
    torch.manual_seed(0)
    cnn = NeuralNetClassifier(
        network,
        max_epochs=20,
        lr=l_r,
        optimizer=torch.optim.Adam,
        batch_size=b_size,
        device=device,
        iterator_train__num_workers=4,
        iterator_valid__num_workers=4,
        callbacks=callbacks,
    )
    print(f"fitting with batch size of {cnn.get_params()['batch_size']} and learning rate of
        {cnn.get_params()['lr']}")

    startall = time.time()
    cnn.fit(dataset, y=y_data)
    endall = time.time()
    timeall = endall-startall

    train_loss.append(np.min(cnn.history[:, 'train_loss']))
    valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
    valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
    dur.append(np.average(cnn.history[:, 'dur']))
    batch_size.append(b_size)
    learning_rate.append(l_r)
    total_time.append(timeall)

    print(f"batch size: {b_size}")
    print(f"learning rate: {l_r}")
    print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
    print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
    print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
    print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
    print(f"Total Time: {timeall:.2f} s")

"""Observation on the side:
- Batch size 32, learning rate 5e-5, max accuracy starts to oscillate as early as epoch #6.
- Batch size 64, learning rate 5e-5, did some crazy jumping around
- max valid accuracy sometimes doesn't occur at min valid loss
"""

epochs =[20, 17, 19, 18, 20, 15, 15, 20, 20]

rawdata= {'batch_size': batch_size,'learning_rate': learning_rate, 'train_loss': train_loss,
         'valid_loss': valid_loss, 'valid_acc': valid_acc, 'time/epoch': dur,
```

```
        'total_time': total_time, 'best_result_epoch': epochs}
bsize_lr = pd.DataFrame(rawdata, columns = ['batch_size','learning_rate','train_loss',
                                            'valid_loss','valid_acc','time/epoch','total_time',
                                                'best_result_epoch'])

bsize_lr

bsize_lr.to_csv("bsize_lr.csv",index=False, header=True)

np.max(cnn.history[:, 'valid_acc'])

drop_ratio

"""# Test dropout ratio

Fixed parameters:
- VGG-16
- batch size = 32
- Optimizer = Adam, no weight decay
- learning rate = 1e-4

Testing parameters:
- drop ratio = 0, 0.1, 0.25, 0.3, 0.5
- changing epochs and learning rate to better understand the results
- added 5-fold cross-validate to obtain reliable results

### Tests with dropout ratio for different epochs

#### 20 epochs, fixed lr=1e-4
"""

drop_ratio_pool = [0, 0.1, 0.25, 0.3, 0.5]
dropout_ratio = []
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []
callbacks = []

for dr in drop_ratio_pool:

  drop_ratio=dr

  network = VGG_net().to(device)
  torch.manual_seed(0)
  cnn = NeuralNetClassifier(

      network,
      max_epochs=20,
      lr=1e-4,
      optimizer=torch.optim.Adam,
      batch_size=32,
      device=device,
      iterator_train__num_workers=4,
      iterator_valid__num_workers=4,
      callbacks=callbacks,
  )
  print(f"fitting with drop ratio of {dr}")

  startall = time.time()
  cnn.fit(dataset, y=y_data)
  endall = time.time()
  timeall = endall-startall
```

```
  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  dropout_ratio.append(dr)
  total_time.append(timeall)

  print(f"drop ratio: {drop_ratio}")
  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

epochs =[20, 17, 20, 20, 20]

rawdata= {'drop_ratio': dropout_ratio, 'train_loss': train_loss,
         'valid_loss': valid_loss, 'valid_acc': valid_acc, 'time/epoch': dur,
         'total_time': total_time, 'best_result_epoch': epochs}
drop_fixed_lr = pd.DataFrame(rawdata, columns = ['drop_ratio','train_loss',
                                        'valid_loss','valid_acc','time/epoch','total_time',
                                            'best_result_epoch'])

drop_fixed_lr

drop_fixed_lr.to_csv("drop_fixed_lr.csv",index=False, header=True)

"""#### 20 epochs, with LRScheduler=ReduceLROnPlateau"""

callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                   LRScheduler(policy=ReduceLROnPlateau,
                   monitor = "train_loss"
                   ))
   ]

drop_ratio_pool = [0, 0.1, 0.25, 0.3, 0.5]
dropout_ratio = []
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []

for dr in drop_ratio_pool:

  drop_ratio=dr

  network = VGG_net().to(device)
  torch.manual_seed(0)
  cnn = NeuralNetClassifier(

      network,
      max_epochs=20,
      lr=1e-4,
      optimizer=torch.optim.Adam,
      batch_size=32,
      device=device,
      iterator_train__num_workers=4,
      iterator_valid__num_workers=4,
      callbacks=callbacks,
  )
  print(f"fitting with drop ratio of {dr}")

  startall = time.time()
```

```
  cnn.fit(dataset, y=y_data)
  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  dropout_ratio.append(dr)
  total_time.append(timeall)

  print(f"drop ratio: {drop_ratio}")
  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

epochs =[19, 20, 20, 20, 18]

rawdata= {'drop_ratio': dropout_ratio, 'train_loss': train_loss,
         'valid_loss': valid_loss, 'valid_acc': valid_acc, 'time/epoch': dur,
         'total_time': total_time, 'best_result_epoch': epochs}
drop_lrschedule = pd.DataFrame(rawdata, columns = ['drop_ratio','train_loss',
                                        'valid_loss','valid_acc','time/epoch','total_time',
                                            'best_result_epoch'])

drop_lrschedule

drop_lrschedule.to_csv("drop_lrschedule.csv",index=False, header=True)

"""#### 30 epochs, with LRScheduler=ReduceLROnPlateau"""

drop_ratio_pool = [0, 0.1, 0.25, 0.3, 0.5]
dropout_ratio = []
train_loss = []
valid_loss = []
valid_loss_last_epoch = []
valid_acc = []
dur = []
total_time = []
callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                   LRScheduler(policy=ReduceLROnPlateau,
                   monitor = "train_loss"
                   ))
   ]

for dr in drop_ratio_pool:

  drop_ratio=dr

  network = VGG_net().to(device)
  torch.manual_seed(0)
  cnn = NeuralNetClassifier(

      network,
      max_epochs=30,
      lr=1e-4,
      optimizer=torch.optim.Adam,
      batch_size=32,
      device=device,
      iterator_train__num_workers=4,
      iterator_valid__num_workers=4,
      callbacks=callbacks,
```

```
  )
  print(f"fitting with drop ratio of {dr}")

  startall = time.time()
  cnn.fit(dataset, y=y_data)
  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_loss_last_epoch.append(cnn.history[-1, 'valid_loss'])
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  dropout_ratio.append(dr)
  total_time.append(timeall)

  print(f"drop ratio: {drop_ratio}")
  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"valid loss at last epoch: {cnn.history[-1, 'valid_loss']}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

rawdata= {'drop_ratio': dropout_ratio, 'train_loss': train_loss,
         'valid_loss_min': valid_loss, 'last_epoch_val_loss': valid_loss_last_epoch,
             'valid_acc': valid_acc, 'time/epoch': dur,
         'total_time': total_time}
drop_lrschedule_30e = pd.DataFrame(rawdata, columns = ['drop_ratio','train_loss',
                                        'valid_loss_min',
                                            'last_epoch_val_loss','valid_acc','time/epoch',
                                        'total_time'])

drop_lrschedule_30e

drop_lrschedule_30e.to_csv("drop_lrschedule_30e.csv",index=False, header=True)

"""### cross-validate"""

# Define callbacks to print results at end of each fold

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau

from skorch.callbacks import Callback

class Monitor(Callback):


    def on_train_begin(self, net, **kwargs):
      print('************cross validate start****')

    def on_train_end(self, net, **kwargs):

      print(f"drop ratio: {drop_ratio}")
      print(f"lowest train loss: {np.min(net.history[:, 'train_loss'])}")
      print(f"lowest valid loss: {np.min(net.history[:, 'valid_loss'])}")
      print(f"valid loss at last epoch: {net.history[-1, 'valid_loss']}")
      print(f"max valid accuracy: {np.max(net.history[:, 'valid_acc'])}")
      print(f"time per epoch: {np.average(net.history[:, 'dur'])}")

callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                  LRScheduler(policy=ReduceLROnPlateau,
                  monitor = "train_loss",
```

```
                    ))
]

"""#### drop ratio = 0"""

from sklearn import datasets, linear_model
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

from skorch.helper import SliceDataset

from sklearn.metrics import log_loss, make_scorer
from sklearn.metrics import accuracy_score

drop_ratio=0
network = VGG_net().to(device)
torch.manual_seed(0)
cnn = NeuralNetClassifier(

      network,
      max_epochs=20,
      lr=1e-4,
      optimizer=torch.optim.Adam,
      batch_size=32,
      device=device,
      iterator_train__num_workers=4,
      iterator_valid__num_workers=4,
      callbacks=callbacks,
)

data_train_sliceable = SliceDataset(dataset)

startall = time.time()
scores = cross_validate(cnn, X=data_train_sliceable, y=y_data, cv=5, return_train_score=True)
endall = time.time()
timeall = endall-startall

print(f"Total Time: {timeall:.2f} s")

dropout = [0, 0.1, 0.25, 0.3, 0.5]
train_loss = []
train_acc = []
valid_loss = []
valid_loss_last_epoch = []
valid_acc = []
dur = []

scores

dr_0_train_loss = [0.006198, 0.004793, 0.004068, 0.008388, 0.004944]
dr_0_train_acc = scores['train_score']
dr_0_valid_loss = [0.2409, 0.2385, 0.2422, 0.2537, 0.2479]
dr_0_valid_loss_last_epoch = [0.3374, 0.3482, 0.3728, 0.3457, 0.3526]
dr_0_valid_acc = scores['test_score']
dr_0_dur = [81.63, 81.71, 81.43, 81.28, 81.31]

train_loss.append(np.average(dr_0_train_loss))
train_acc.append(np.average(dr_0_train_acc))
valid_loss.append(np.average(dr_0_valid_loss))
valid_loss_last_epoch.append(np.average(dr_0_valid_loss_last_epoch))
valid_acc.append(np.average(dr_0_valid_acc))

rawdata= {'train_loss': dr_0_train_loss, 'train_acc': dr_0_train_acc,
```

```
         'valid_loss_min': dr_0_valid_loss, 'last_epoch_val_loss': dr_0_valid_loss_last_epoch,
         'valid_acc': dr_0_valid_acc, 'time/epoch': dr_0_dur,
         }
drop_000 = pd.DataFrame(rawdata, columns = ['train_loss', 'train_acc',
                                        'valid_loss_min',
                                            'last_epoch_val_loss','valid_acc','time/epoch',
                                        ])
drop_000.to_csv("drop_000.csv",index=False, header=True)

drop_000

"""#### drop ratio = 0.1"""

from sklearn import datasets, linear_model
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

from skorch.helper import SliceDataset

from sklearn.metrics import log_loss, make_scorer
from sklearn.metrics import accuracy_score

drop_ratio=0.1
network = VGG_net().to(device)
torch.manual_seed(0)
cnn = NeuralNetClassifier(

     network,
     max_epochs=20,
     lr=1e-4,
     optimizer=torch.optim.Adam,
     batch_size=32,
     device=device,
     iterator_train__num_workers=4,
     iterator_valid__num_workers=4,
     callbacks=callbacks,
)

data_train_sliceable = SliceDataset(dataset)

startall = time.time()
scores = cross_validate(cnn, X=data_train_sliceable, y=y_data, cv=5, return_train_score=True)
endall = time.time()
timeall = endall-startall

print(f"Total Time: {timeall:.2f} s")

dr_01_train_loss = [0.008861, 0.01379, 0.008249, 0.01028, 0.007738]
dr_01_train_acc = [0.987582, 0.985437, 0.987647, 0.987584, 0.985188]
dr_01_valid_loss = [0.2688, 0.2799, 0.2498, 0.2548, 0.2795]
dr_01_valid_loss_last_epoch = [0.3486, 0.3421, 0.3384, 0.3253, 0.3977]
dr_01_valid_acc = [0.946443, 0.947680, 0.950904, 0.948404, 0.945570]
dr_01_dur = [81.63, 81.63, 81.55, 81.49, 81.37]


rawdata= {'train_loss': dr_01_train_loss, 'train_acc': dr_01_train_acc,
         'valid_loss_min': dr_01_valid_loss, 'last_epoch_val_loss': dr_01_valid_loss_last_epoch,
         'valid_acc': dr_01_valid_acc, 'time/epoch': dr_01_dur,
         }
drop_010 = pd.DataFrame(rawdata, columns = ['train_loss', 'train_acc',
                                        'valid_loss_min',
                                            'last_epoch_val_loss','valid_acc','time/epoch',
                                        ])
```

```
drop_010.to_csv("drop_010.csv",index=False, header=True)

drop_010

"""#### drop ratio = 0.25"""

from sklearn import datasets, linear_model
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

from skorch.helper import SliceDataset

from sklearn.metrics import log_loss, make_scorer
from sklearn.metrics import accuracy_score

drop_ratio=0.25
network = VGG_net().to(device)
torch.manual_seed(0)
cnn = NeuralNetClassifier(

     network,
     max_epochs=20,
     lr=1e-4,
     optimizer=torch.optim.Adam,
     batch_size=32,
     device=device,
     iterator_train__num_workers=4,
     iterator_valid__num_workers=4,
     callbacks=callbacks,
)

data_train_sliceable = SliceDataset(dataset)

startall = time.time()
scores = cross_validate(cnn, X=data_train_sliceable, y=y_data, cv=5, return_train_score=True)
endall = time.time()
timeall = endall-startall

print(f"Total Time: {timeall:.2f} s")

dr_025_train_loss = [0.008787, 0.01373, 0.01413, 0.01181, 0.01521]
dr_025_train_acc = scores['train_score']
dr_025_valid_loss = [0.2537, 0.2486, 0.2571, 0.2849, 0.2846]
dr_025_valid_loss_last_epoch = [0.3452, 0.3245, 0.3129, 0.3575, 0.3739]
dr_025_valid_acc = scores['test_score']
dr_025_dur = [47.15, 47.21, 47.18, 47.12, 47.12]

rawdata= {'train_loss': dr_025_train_loss, 'train_acc': dr_025_train_acc,
         'valid_loss_min': dr_025_valid_loss, 'last_epoch_val_loss':
             dr_025_valid_loss_last_epoch,
         'valid_acc': dr_025_valid_acc, 'time/epoch': dr_025_dur,
         }
drop_025 = pd.DataFrame(rawdata, columns = ['train_loss', 'train_acc',
                                     'valid_loss_min',
                                         'last_epoch_val_loss','valid_acc','time/epoch',
                                     ])
drop_025.to_csv("drop_025.csv",index=False, header=True)

drop_025.to_csv("drop_025.csv",index=False, header=True)

"""#### drop ratio = 0.3"""

from sklearn import datasets, linear_model
```

```
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

from skorch.helper import SliceDataset

from sklearn.metrics import log_loss, make_scorer
from sklearn.metrics import accuracy_score

drop_ratio=0.3
network = VGG_net().to(device)
torch.manual_seed(0)
cnn = NeuralNetClassifier(

     network,
     max_epochs=20,
     lr=1e-4,
     optimizer=torch.optim.Adam,
     batch_size=32,
     device=device,
     iterator_train__num_workers=4,
     iterator_valid__num_workers=4,
     callbacks=callbacks,
)

data_train_sliceable = SliceDataset(dataset)

startall = time.time()
scores = cross_validate(cnn, X=data_train_sliceable, y=y_data, cv=5, return_train_score=True)
endall = time.time()
timeall = endall-startall

print(f"Total Time: {timeall:.2f} s")

dr_03_train_loss = [0.01482, 0.02010, 0.01662, 0.01409, 0.01193]
dr_03_train_acc = scores['train_score']
dr_03_valid_loss = [0.2816, 0.2688, 0.2778, 0.2720, 0.2719]
dr_03_valid_loss_last_epoch = [0.3363, 0.3195, 0.3281, 0.3280, 0.3723]
dr_03_valid_acc = scores['test_score']
dr_03_dur = [46.81, 46.64, 46.65, 46.69, 46.31]

rawdata= {'train_loss': dr_03_train_loss, 'train_acc': dr_03_train_acc,
         'valid_loss_min': dr_03_valid_loss, 'last_epoch_val_loss': dr_03_valid_loss_last_epoch,
         'valid_acc': dr_03_valid_acc, 'time/epoch': dr_03_dur,
         }
drop_03 = pd.DataFrame(rawdata, columns = ['train_loss', 'train_acc',
                                        'valid_loss_min',
                                            'last_epoch_val_loss','valid_acc','time/epoch',
                                        ])
drop_03.to_csv("drop_030.csv",index=False, header=True)

drop_03

"""#### drop ratio = 0.5"""

from sklearn import datasets, linear_model
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC

from skorch.helper import SliceDataset

from sklearn.metrics import log_loss, make_scorer
```

```
from sklearn.metrics import accuracy_score

drop_ratio=0.5
network = VGG_net().to(device)
torch.manual_seed(0)
cnn = NeuralNetClassifier(

     network,
     max_epochs=20,
     lr=1e-4,
     optimizer=torch.optim.Adam,
     batch_size=32,
     device=device,
     iterator_train__num_workers=4,
     iterator_valid__num_workers=4,
     callbacks=callbacks,
)

data_train_sliceable = SliceDataset(dataset)

startall = time.time()
scores = cross_validate(cnn, X=data_train_sliceable, y=y_data, cv=5, return_train_score=True)
endall = time.time()
timeall = endall-startall

print(f"Total Time: {timeall:.2f} s")

dr_05_train_loss = [0.02364, 0.02600, 0.02608, 0.02290, 0.02437]
dr_05_train_acc = scores['train_score']
dr_05_valid_loss = [0.2986, 0.2919, 0.2982, 0.2922, 0.3029]
dr_05_valid_loss_last_epoch = [0.3481, 0.3465, 0.3684, 0.3652, 0.3895]
dr_05_valid_acc = scores['test_score']
dr_05_dur = [46.44, 46.45, 46.42, 46.68, 46.54]

rawdata= {'train_loss': dr_05_train_loss, 'train_acc': dr_05_train_acc,
         'valid_loss_min': dr_05_valid_loss, 'last_epoch_val_loss': dr_05_valid_loss_last_epoch,
         'valid_acc': dr_05_valid_acc, 'time/epoch': dr_05_dur,
         }
drop_05 = pd.DataFrame(rawdata, columns = ['train_loss', 'train_acc',
                                        'valid_loss_min',
                                            'last_epoch_val_loss','valid_acc','time/epoch',
                                        ])
drop_05.to_csv("drop_050.csv",index=False, header=True)

"""#### final data"""

drop_000 = pd.read_csv("drop_000.csv")
drop_010 = pd.read_csv("drop_010.csv")
drop_025 = pd.read_csv("drop_025.csv")
drop_030 = pd.read_csv("drop_030.csv")
drop_050 = pd.read_csv("drop_050.csv")

drop_000

drop = [0, 0.1, 0.25, 0.3, 0.5]
train_loss = [np.average(drop_000['train_loss']),
              np.average(drop_010['train_loss']),
              np.average(drop_025['train_loss']),
              np.average(drop_030['train_loss']),
              np.average(drop_050['train_loss'])]
train_acc = [np.average(drop_000['train_acc']),
             np.average(drop_010['train_acc']),
             np.average(drop_025['train_acc']),
             np.average(drop_030['train_acc']),
             np.average(drop_050['train_acc'])]
```

```
valid_loss = [np.average(drop_000['valid_loss_min']),
              np.average(drop_010['valid_loss_min']),
              np.average(drop_025['valid_loss_min']),
              np.average(drop_030['valid_loss_min']),
              np.average(drop_050['valid_loss_min'])]
valid_loss_last_epoch = [np.average(drop_000['last_epoch_val_loss']),
                         np.average(drop_010['last_epoch_val_loss']),
                         np.average(drop_025['last_epoch_val_loss']),
                         np.average(drop_030['last_epoch_val_loss']),
                         np.average(drop_050['last_epoch_val_loss'])]
valid_acc = [np.average(drop_000['valid_acc']),
             np.average(drop_010['valid_acc']),
             np.average(drop_025['valid_acc']),
             np.average(drop_030['valid_acc']),
             np.average(drop_050['valid_acc'])]

rawdata= {'drop_ratio': drop, 'train_loss': train_loss, 'train_acc': train_acc,
          'valid_loss_min': valid_loss, 'last_epoch_val_loss': valid_loss_last_epoch,
              'valid_acc': valid_acc,
          }
drop_cv = pd.DataFrame(rawdata, columns = ['drop_ratio','train_loss', 'train_acc',
                                           'valid_loss_min', 'last_epoch_val_loss','valid_acc'
                                           ])

drop_cv

drop_cv.to_csv("drop_cv.csv",index=False, header=True)

fig, ax = plt.subplots()

ax2 = ax.twinx()

drop_cv.plot(x = "drop_ratio", y = "valid_acc", kind='line', color='C1', ax=ax, label="Val.
    Acc.")
drop_cv.plot(x = "drop_ratio", y = "train_acc", kind='line', color='C2', ax=ax2, label="train.
    Acc.")
ax.set_xlabel("Drop ratio")
ax.set_ylabel('Validation Accuracy')
ax2.set_ylabel('Train Accuracy')

fig, ax = plt.subplots()

ax2 = ax.twinx()

width = 0.2

drop_cv.plot(x = "drop_ratio", y = "valid_loss_min", kind='line', color='C1', ax=ax,
    label="Val. Loss.")
drop_cv.plot(x = "drop_ratio", y = "train_loss", kind='line', color='C2', ax=ax2, label="train.
    Loss.")
ax.set_xlabel("Drop ratio")
ax.set_ylabel('Validation Loss')
ax2.set_ylabel('Train Loss')

fig, ax = plt.subplots()

ax2 = ax.twinx()

width = 0.2

drop_cv.plot(x = "drop_ratio", y = "valid_acc", kind='line', color='C1', ax=ax, width=width,
    position=1, label="Val. Acc.")
drop_cv.plot(x = "Network_type", y = "total_time", kind='bar', color='C2', ax=ax2, width=width,
    position=0,label="Time")
ax.set_xlabel("VGG Type")
```

```
ax.set_ylabel('Validation Accuracy')
ax2.set_ylabel('Time [s]')
ax.set_ylim(0.9)
ax.legend(loc=9);
plt.tight_layout()
plt.savefig("plots/vgg_type.pdf", bbox_inches="tight", pad_inches=0)

# final data

rawdata= {'drop_ratio': dropout, 'train_loss': train_loss, 'train_acc': train_acc,
         'valid_loss_min': valid_loss, 'last_epoch_val_loss': valid_loss_last_epoch,
             'valid_acc': valid_acc, 'time/epoch': dur,
         }
drop_cv = pd.DataFrame(rawdata, columns = ['drop_ratio','train_loss', 'train_acc',
                                           'valid_loss_min',
                                               'last_epoch_val_loss','valid_acc','time/epoch',
                                           ])

"""# Test weight decay + drop ratio

Fixed parameters:
- VGG-16
- batch size = 32
- learning rate: LRScheduler=ReduceLROnPlateau, initial lr=1e-4
- Optimizer = AdamW
- epochs = 20

Testing parameters:
- drop ratio = 0.25,0.3,0.5
- weight decay = 0.005, 0.01 0.02
"""

import skorch

weight_decay_pool = [0.005, 0.01, 0.02]
drop_out_pool = [0.25, 0.3, 0.5]
drop = []
weight_decay = []
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []
callbacks = callbacks=[
       ('print', Monitor()), ('lr_scheduler',
                  LRScheduler(policy=ReduceLROnPlateau,
                  monitor = "train_loss",
                  ))
   ]

for wd in weight_decay_pool:

  for dr in drop_out_pool:

    drop_ratio = dr
    network = VGG_net().to(device)
    torch.manual_seed(0)
    cnn = NeuralNetClassifier(
        network,
        max_epochs=20,
        lr=1e-4,
        optimizer=torch.optim.AdamW,
        optimizer__weight_decay=wd,
        batch_size=32,
        device=device,
```

```
        iterator_train__num_workers=4,
        iterator_valid__num_workers=4,
        callbacks=callbacks,
        train_split = skorch.dataset.CVSplit(10)
    )
    print(f"fitting with weight decay of {wd} and drop ratio of {dr}")

    startall = time.time()
    cnn.fit(dataset, y=y_data)
    endall = time.time()
    timeall = endall-startall

    train_loss.append(np.min(cnn.history[:, 'train_loss']))
    valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
    valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
    dur.append(np.average(cnn.history[:, 'dur']))
    drop.append(dr)
    weight_decay.append(wd)
    total_time.append(timeall)

    print(f"drop ratio: {dr}")
    print(f"weight decay: {wd}")
    print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
    print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
    print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
    print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
    print(f"Total Time: {timeall:.2f} s")

rawdata= {'drop_ratio': drop,'weight_decay': weight_decay, 'train_loss': train_loss,
         'valid_loss': valid_loss, 'valid_acc': valid_acc, 'time/epoch': dur,
         'total_time': total_time}
weight_decay_dropout = pd.DataFrame(rawdata, columns =
    ['drop_ratio','weight_decay','train_loss',
                                      'valid_loss','valid_acc','time/epoch','total_time'])

weight_decay_dropout

weight_decay_dropout.to_csv("weight_decay_dropout.csv",index=False, header=True)

"""# Test VGG 11,13,16,19
hypeparameter:

1. learning rate: 1e-4
2. batch size: 32
3. drop_ratio = 0.25
4. kernel_size_CNN = (3, 3)
5. optimizer: Adam
6. LR_scheduler: ReduceLROnPlateau(default)


run for 20 epoch


"""

learning_rate = 1e-4
batch_size = 32
max_epochs = 20

sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))
y_data = np.array([y for x, y in iter(dataset)])

from skorch.callbacks import LRScheduler
```

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
from skorch.callbacks import Callback



#  callbacks function to check and change lr
callbacks=[
          ('lr_scheduler',
           LRScheduler(
               policy=ReduceLROnPlateau,
               monitor = "train_loss",
               )
           )
         ]

Network_type = ["VGG11", "VGG13", "VGG16", "VGG19"]
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []

for i in Network_type:

  print ("testing: ", i, " now")
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types[i]).to(device)

  cnn = NeuralNetClassifier(
      network,
      max_epochs = max_epochs,
      lr= learning_rate,

      # AdamW
      optimizer = torch.optim.Adam,
      # optimizer__weight_decay=weight_decay,

      batch_size = 32,
      device=device,
      iterator_train__num_workers = 4,
      iterator_valid__num_workers = 4,
      callbacks=callbacks,
  )

  # record start time
  startall = time.time()

  # training
  cnn.fit(dataset, y=y_data)

  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  total_time.append(timeall)

  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")
```

```
import pandas as pd
rawdata= {'Network_type': Network_type, 'train_loss': train_loss, 'valid_loss': valid_loss,
    'valid_acc': valid_acc, 'time/epoch': dur, 'total_time': total_time}
VGG_compare = pd.DataFrame(rawdata, columns =
    ['Network_type','train_loss','valid_loss','valid_acc','time/epoch','total_time'])
print (VGG_compare)

VGG_compare.to_csv("VGG_compare.csv",index=False, header=True)

"""# Test optimizer: Adam, AdamW, SGD_momentum, SGD_momentum_nesterov
hypeparameter:

1. learning rate: 1e-4
2. batch size: 32
3. drop_ratio = 0.25
4. kernel_size_CNN = (3, 3)
5. structure : VGG-16
6. LR_scheduler: ReduceLROnPlateau(default)

run for 20 epoch

# optimizer hyperparmeter

1. Adam
2. Adam: weight decay = 1e-3
3. SGD_momentum: momentum =0.99, weight decay = 1e-3
4. SGD_momentum_nesterov : momentum =0.99, weight decay = 1e-3, nesterov = true


"""

learning_rate = 1e-4
batch_size = 32
max_epochs = 20

sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))
y_data = np.array([y for x, y in iter(dataset)])

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau
from skorch.callbacks import Callback

torch.manual_seed(0)

#  callbacks function to check and change lr
callbacks=[
          ('lr_scheduler',
           LRScheduler(
               policy=ReduceLROnPlateau,
               monitor = "train_loss",
               )
           )
         ]

optimizer_type = ["Adam", "AdamW", "SGD_momentum", "SGD_momentum_nesterov"]
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []

for i in optimizer_type:
```

```
print ("testing: ", i, " now")
network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
    VGG_types["VGG16"]).to(device)

if (i=="Adam"):
  cnn = NeuralNetClassifier(
      network,
      max_epochs = max_epochs,
      lr= learning_rate,
      optimizer = torch.optim.Adam,
      batch_size = 32,
      device=device,
      iterator_train__num_workers = 4,
      iterator_valid__num_workers = 4,
      callbacks=callbacks,
  )
elif (i == "AdamW"):
   cnn = NeuralNetClassifier(
      network,
      max_epochs = max_epochs,
      lr= learning_rate,
      optimizer = torch.optim.AdamW,
      optimizer__weight_decay=weight_decay,
      batch_size = 32,
      device=device,
      iterator_train__num_workers = 4,
      iterator_valid__num_workers = 4,
      callbacks=callbacks,
  )
elif (i == "SGD_momentum"):
   cnn = NeuralNetClassifier(
      network,
      max_epochs = max_epochs,
      lr= learning_rate,
      optimizer = torch.optim.SGD,
      optimizer__weight_decay = weight_decay,
      optimizer__momentum = 0.9,
      optimizer__nesterov = False,
      batch_size = 32,
      device=device,
      iterator_train__num_workers = 4,
      iterator_valid__num_workers = 4,
      callbacks=callbacks,
  )
elif (i == "SGD_momentum_nesterov"):
  cnn = NeuralNetClassifier(
      network,
      max_epochs = max_epochs,
      lr= learning_rate,
      optimizer = torch.optim.SGD,
      optimizer__weight_decay = weight_decay,
      optimizer__momentum = 0.9,
      optimizer__nesterov = True,
      batch_size = 32,
      device=device,
      iterator_train__num_workers = 4,
      iterator_valid__num_workers = 4,
      callbacks=callbacks,
  )

# record start time
startall = time.time()

# training
```

```
  cnn.fit(dataset, y=y_data)

  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  total_time.append(timeall)

  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

import pandas as pd
rawdata= {'optimizer_type': optimizer_type, 'train_loss': train_loss, 'valid_loss': valid_loss,
    'valid_acc': valid_acc, 'time/epoch': dur, 'total_time': total_time}
VGG_compare = pd.DataFrame(rawdata, columns =
    ['optimizer_type','train_loss','valid_loss','valid_acc','time/epoch','total_time'])
print (VGG_compare)

VGG_compare.to_csv("optimizer_compare.csv",index=False, header=True)

"""Wait until SGD converges..."""

optimizer_type = ["SGD_momentum", "SGD_momentum_nesterov"]

train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []

max_epochs = 50

callbacks=[
       ('lr_scheduler',
       LRScheduler(
           policy=ReduceLROnPlateau,
           factor = 0.5,
           monitor = "train_loss",
           )
       )
     ]


for i in optimizer_type:

  print ("testing: ", i, " now")
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)

  if (i=="Adam"):
    cnn = NeuralNetClassifier(
        network,
        max_epochs = max_epochs,
        lr= learning_rate,
        optimizer = torch.optim.Adam,
        batch_size = 20,
        device=device,
        iterator_train__num_workers = 4,
        iterator_valid__num_workers = 4,
```

```
        callbacks=callbacks,
    )
elif (i == "AdamW"):
    cnn = NeuralNetClassifier(
        network,
        max_epochs = max_epochs,
        lr= learning_rate,
        optimizer = torch.optim.AdamW,
        optimizer__weight_decay=weight_decay,
        batch_size = 32,
        device=device,
        iterator_train__num_workers = 4,
        iterator_valid__num_workers = 4,
        callbacks=callbacks,
    )
elif (i == "SGD_momentum"):
    cnn = NeuralNetClassifier(
        network,
        max_epochs = max_epochs,
        lr= learning_rate,
        optimizer = torch.optim.SGD,
        optimizer__weight_decay = 0,
        optimizer__momentum = 0,
        optimizer__nesterov = False,
        batch_size = 32,
        device=device,
        iterator_train__num_workers = 4,
        iterator_valid__num_workers = 4,
        callbacks=callbacks,
    )
elif (i == "SGD_momentum_nesterov"):
    cnn = NeuralNetClassifier(
        network,
        max_epochs = max_epochs,
        lr= learning_rate,
        optimizer = torch.optim.SGD,
        optimizer__weight_decay = 0,
        optimizer__momentum = 0,
        optimizer__nesterov = True,
        batch_size = 32,
        device=device,
        iterator_train__num_workers = 4,
        iterator_valid__num_workers = 4,
        callbacks=callbacks,
    )

# record start time
startall = time.time()

# training
cnn.fit(dataset, y=y_data)

endall = time.time()
timeall = endall-startall

train_loss.append(np.min(cnn.history[:, 'train_loss']))
valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
dur.append(np.average(cnn.history[:, 'dur']))
total_time.append(timeall)

print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
```

```
  print(f"Total Time: {timeall:.2f} s")

import pandas as pd
rawdata= {'train_loss': train_loss, 'valid_loss': valid_loss, 'valid_acc': valid_acc,
    'time/epoch': dur, 'total_time': total_time}
VGG_compare = pd.DataFrame(rawdata, columns =
    ['train_loss','valid_loss','valid_acc','time/epoch','total_time'])
print (VGG_compare)

"""# Test LR_scheduler: CyclicLR_triangular
hypeparameter:

1. learning rate: 1e-4
2. batch size: 32
3. drop_ratio = 0.25
4. kernel_size_CNN = (3, 3)
5. structure : VGG-16
6. optimizer : Adam


run for 20 epoch

# LR_scheduler hyperparmeter
referecne:
    https://medium.com/swlh/cyclical-learning-rates-the-ultimate-guide-for-setting-learning-rates-for-neural-netw

fixed parameter without momentum

base_lr = 1e-5,
max_lr = 1e-4,
step_size_up = 5,


1. CyclicLR_triangular
2. CyclicLR_triangular2
3. CyclicLR_exp_range



"""

learning_rate = 1e-4
batch_size = 32
max_epochs = 20

sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))
y_data = np.array([y for x, y in iter(dataset)])

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau
from skorch.callbacks import Callback

class Monitor(Callback):
    def on_epoch_end(self, network, **kwargs):
      print("current learning rate: ", network.optimizer_.param_groups[0]['lr'])

torch.manual_seed(0)
CyclicLR_type = ["CyclicLR_triangular", "CyclicLR_triangular2", "CyclicLR_exp_range"]
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []
```

```
for i in CyclicLR_type:

  print ("testing: ", i, " now")
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)

  if (i=="CyclicLR_triangular"):
    callbacks=[
        ('print', Monitor()),
        ('lr_scheduler',
        LRScheduler(
            policy=torch.optim.lr_scheduler.CyclicLR,
            base_lr = 1e-5,
            max_lr = 1e-4,
            step_size_up = 5,
            cycle_momentum=False,
            mode='triangular',
            monitor = "train_loss",
            )
        )
      ]
  elif (i == "CyclicLR_triangular2"):
    callbacks=[
        ('print', Monitor()),
        ('lr_scheduler',
        LRScheduler(
            policy=torch.optim.lr_scheduler.CyclicLR,
            base_lr = 1e-5,
            max_lr = 1e-4,
            step_size_up = 5,
            cycle_momentum=False,
            mode='triangular2',
            monitor = "train_loss",
            )
        )
      ]
  elif (i == "CyclicLR_exp_range"):
    callbacks=[
        ('print', Monitor()),
        ('lr_scheduler',
        LRScheduler(
            policy=torch.optim.lr_scheduler.CyclicLR,
            base_lr = 1e-5,
            max_lr = 1e-4,
            step_size_up = 5,
            cycle_momentum=False,
            mode='exp_range',
            monitor = "train_loss",
            )
        )
      ]

  cnn = NeuralNetClassifier(
        network,
        max_epochs = max_epochs,
        lr= learning_rate,
        optimizer = torch.optim.Adam,
        batch_size = 32,
        device=device,
        iterator_train__num_workers = 4,
        iterator_valid__num_workers = 4,
        callbacks=callbacks,
    )
  # record start time
```

```
  startall = time.time()

  # training
  cnn.fit(dataset, y=y_data)

  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  total_time.append(timeall)

  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

import pandas as pd
rawdata= {'CyclicLR_type': CyclicLR_type, 'train_loss': train_loss, 'valid_loss': valid_loss,
    'valid_acc': valid_acc, 'time/epoch': dur, 'total_time': total_time}
VGG_compare = pd.DataFrame(rawdata, columns =
    ['CyclicLR_type','train_loss','valid_loss','valid_acc','time/epoch','total_time'])
print (VGG_compare)

VGG_compare.to_csv("CyclicLR_type_compare.csv",index=False, header=True)

"""# Test LR_scheduler: ReduceLROnPlateau
hypeparameter:

1. learning rate: 1e-4
2. batch size: 32
3. drop_ratio = 0.25
4. kernel_size_CNN = (3, 3)
5. structure : VGG-16
6. optimizer : Adam


run for 20 epoch

# ReduceLROnPlateau hyperparmeter

1. ReduceLROnPlateau_default
2. ReduceLROnPlateau_fatcor_0.9_patience_4
3. ReduceLROnPlateau_fatcor_0.5_patience_4




"""

learning_rate = 1e-4
batch_size = 32
max_epochs = 20

sample_number = 60000
dataset = MyDataset('./Train.pkl', './TrainLabels.csv',transform=img_transform,
    idx=np.arange(sample_number))
y_data = np.array([y for x, y in iter(dataset)])

from skorch.callbacks import LRScheduler
from torch.optim.lr_scheduler import ReduceLROnPlateau
from skorch.callbacks import Callback
```

```
torch.manual_seed(0)

ReduceLROnPlateau_type = ["ReduceLROnPlateau_default",
    "ReduceLROnPlateau_fatcor_0.9_patience_4", "ReduceLROnPlateau_fatcor_0.5_patience_4"]
train_loss = []
valid_loss = []
valid_acc = []
dur = []
total_time = []

for i in ReduceLROnPlateau_type:

  print ("testing: ", i, " now")
  network = VGG_net(in_channels=num_in, num_classes=num_class, VGG_type =
      VGG_types["VGG16"]).to(device)

  if (i=="ReduceLROnPlateau_default"):
    callbacks=[
          ('lr_scheduler',
           LRScheduler(
               policy=ReduceLROnPlateau,
               monitor = "train_loss",
               )
           )
         ]
  elif (i == "ReduceLROnPlateau_fatcor_0.9_patience_4"):
    callbacks=[
          ('lr_scheduler',
           LRScheduler(
               policy=ReduceLROnPlateau,
               factor = 0.9,
               patience = 4,
               monitor = "train_loss",
               )
           )
         ]
  elif (i == "ReduceLROnPlateau_fatcor_0.5_patience_4"):
    callbacks=[
          ('lr_scheduler',
           LRScheduler(
               policy=ReduceLROnPlateau,
               factor = 0.5,
               patience = 4,
               monitor = "train_loss",
               )
           )
         ]

  cnn = NeuralNetClassifier(
       network,
       max_epochs = max_epochs,
       lr= learning_rate,
       optimizer = torch.optim.Adam,
       batch_size = 32,
       device=device,
       iterator_train__num_workers = 4,
       iterator_valid__num_workers = 4,
       callbacks=callbacks,
  )
  # record start time
  startall = time.time()

  # training
  cnn.fit(dataset, y=y_data)
```

```
  endall = time.time()
  timeall = endall-startall

  train_loss.append(np.min(cnn.history[:, 'train_loss']))
  valid_loss.append(np.min(cnn.history[:, 'valid_loss']))
  valid_acc.append(np.max(cnn.history[:, 'valid_acc']))
  dur.append(np.average(cnn.history[:, 'dur']))
  total_time.append(timeall)

  print(f"lowest train loss: {np.min(cnn.history[:, 'train_loss'])}")
  print(f"lowest valid loss: {np.min(cnn.history[:, 'valid_loss'])}")
  print(f"max valid accuracy: {np.max(cnn.history[:, 'valid_acc'])}")
  print(f"time per epoch: {np.average(cnn.history[:, 'dur'])}")
  print(f"Total Time: {timeall:.2f} s")

import pandas as pd
rawdata= {'ReduceLROnPlateau_type': ReduceLROnPlateau_type, 'train_loss': train_loss,
    'valid_loss': valid_loss, 'valid_acc': valid_acc, 'time/epoch': dur, 'total_time':
    total_time}
VGG_compare = pd.DataFrame(rawdata, columns =
    ['ReduceLROnPlateau_type','train_loss','valid_loss','valid_acc','time/epoch','total_time'])
print (VGG_compare)

VGG_compare.to_csv("ReduceLROnPlateau_type_compare.csv",index=False, header=True)
```