Classifying Plankton using Convolutional Neural Networks

Alex Nguyen

Abstract

In this project we apply convolutional neural networks for the task of classifying plankton species from image data hosted by the National Science Bowl. We approach the problem by initially building simple and shallow networks and experiment with varying hyperparameters and deeper architectures. We introduce regularization methods data augmentation to exploit certain image symmetries. We also experiment on other optimizers for our neural networks to test if they improve generalization ability. Our final models were able to achieve accuracies as high as 72.6% and cross entropy losses as low as 0.987 on the validation set.

1. Introduction

Plankton play a critically important role in our ecosystem, since they account for a large portion of photosynthesis that produces oxygen on Earth, and form the foundation of the oceanic food chain. Due to their importance, measuring and monitoring their populations serve as key indicators on the status and health of the ocean and the associated ecosystems.

Traditional manual methods of measuring and classifying plankton are laborious and are extremely infeasible for any sort of large-scale population study. Thus, the goal of the National Science Bowl Competition was to develop scalable, robust, and automated methods of classifying plankton to circumvent manual limitations. Such methods could have major impacts and broad applications related to marine ecosystem health.

2. Dataset

The images used in this project were compiled by the scientists and researchers from the Hatfield Marine Science Center and was hosted by the National Data Science Bowl Competition on Kaggle [1]. The training set contained a total 30,458 grayscale images labeled according to one of 121 different classes of plankton. Sample images of the various plankton are shown in Figure 1.

We split the model in 2 parts, a training set consisting of 25,336 images to train our models, while the rest of the data (5,000 images) is set aside as a validation set to evaluate our models.

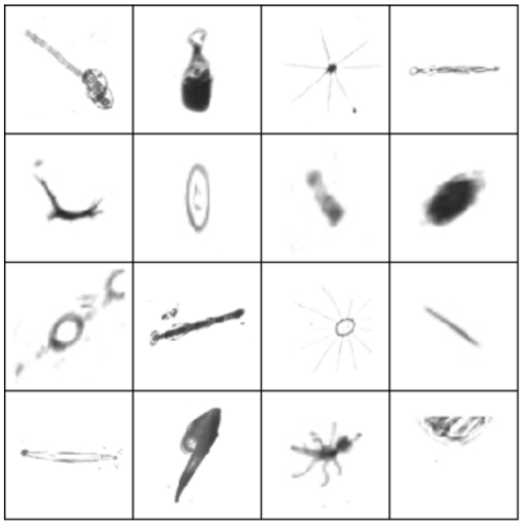
Figure 1: Sample plankton from the dataset

2.1 Preprocessing

Initially the images widely varied in sizes. The width of the images ranged from 40 pixels to as large as 400 pixels wide. In order to feed the images for training, we resized the images all into a uniform size of 80 by 80 pixels. Thus, each image is represented by a 6,400-dimensional vector to use as inputs into the following models.

3. Methods

3.1 Evaluation Metrics

For this project, we used two types of evaluation metrics. The first being the cross-entropy loss function that the neural networks will optimize:

$$L_{Cross\ Entropy} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(g_{ij})$$

Where n refers to the number of images, m is the number of classes, y_{ij} is 1 if a particular image i is in class j, other wise it is 0, and g_{ij} is probability output from the neural network that image i is in class j.

Secondly, we use the more intuitive metric of accuracy defined as:

$$Accuracy = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(y_i = \hat{y}_i)$$

Where y_i refers to the true class of image i, and $\hat{y}_i$ refers to the predicted class of image i. While the two are inherently correlated, accuracy measures the rate of correct classifications, while the cross-entropy loss takes into account the amount uncertainty of each class prediction per output.

3.2 Data Augmentation

Before a batch of images goes in the model for training they are randomly flipped vertically/horizontally, rotated between -7 and 7 degrees, and translated 10% in a random direction. Since the images that undergo these transformations still represent the same image label, training with these augmented images should help our models become invariant to such transformations. This allows us to artificially increase the training data so that at every epoch, the model trains on slightly different images.

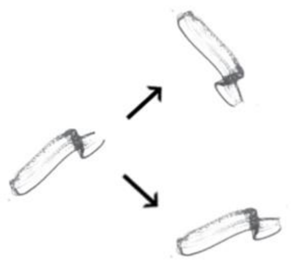

Figure 2: Original Image on the left being transformed. 90° degree clockwise rotation on top transformation, and 10° degree clockwise rotation on the bottom transformation.

3.3 Convolutional Neural Networks

In recent years, convolutional neural networks (CNN's) have become a staple in the field of image classification and computer vision due to their ability to learn spatial hierarchies of patterns. Various CNN architectures like the likes of VGG Net [2] and AlexNet [3] have enjoyed profound successes in classifying a wide variety of objects; winning the ImageNet Large Scale Visual Recognition Challenge, which have served as the impetus for the development of the next generation of large-scale classification systems. Inspired by such successes we employ CNN's for our classification task.

A typical CNN model contains several main parts: the convolution layer, the sub-sampling layer, the fully connected linear layers, and the softmax classifier. When building a CNN model as an image classifier, one or multiple CNN layers are needed and connected in series. The number of CNN layers is highly significant to the classification accuracy: if the number is too small, the data features will not be fully abstracted and utilized in the classification task. But when the number of layers is too large, the run time will increase exponentially and, what is worse, cause some undesirable training problems such as gradient vanishing or exploding.

The initial optimization algorithm we used was the stochastic gradient descent optimizer with the learning rate parameter set to 0.01 and momentum parameter 0.9. For our deepest model, we ran into the problem of exploding gradients, so for that model we lowered the learning rate to 0.004 which yielded gradient stability.

The first model we created was a 2 convolutional layer and 2 fully connect layer neural network. This shallow network was chosen as a baseline model before we moved onto deeper models. The architecture is shown below; each convolutional and fully connected layer used a ReLU activation function (not shown for brevity):

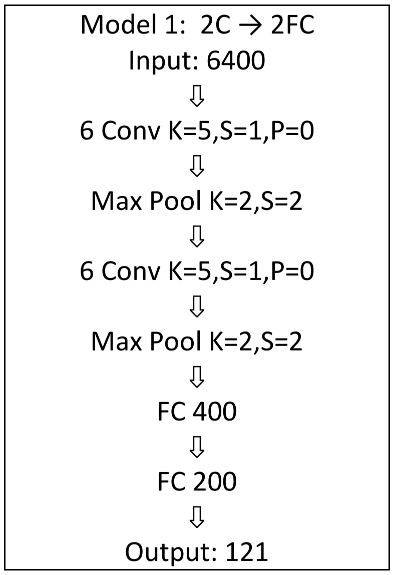

Figure 3: Architecture of the baseline model (ReLU activations are not shown for brevity)

We experimented with batch sizes of 16, 32, 48, and 64 with our baseline model shown. The results on the baseline are shown in Figure 4. The results from this experiment are shown in the appendix. Using the batch size of 16 yielded the lowest validation curve, so we chose it moving forward. In practice smaller batch sizes tend to allow neutral networks to generalize better. [4]

In total we experimented on 5 convolutional neural networks, each with increasing convolutional and fully connected layers. The total ranged from 4 to 9 layers. Seeing the success of the VGG Net [3], we implemented similar architectures onto our own models. In particular, most of the deeper models had increasing channel sizes with respect to the convolutional layers and had convolutional layers stacked on top before each max pooling layer. Like VGG, our deeper networks had a kernel size of 3 by 3, a zero-padding of 1, and a stride of 1. In all our models we used a max pooling layer of kernel size 2 by 2, and a stride of 2. For all the models, each convolutional and fully connected layer used the ReLU activation. The full architectures shown in the appendix.

4. Results

We ran each model for 50 epochs, recording the training and validation metrics for each epoch. The results shown below:

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
Model 1 (3 Conv + 2FC) No data augmentation	0.103	0.9314	2.214	0.571
Model 1 (2C + 2FC)	1.055	0.665	1.301	0.665
Model 2 (3C + 2FC)	0.779	0.740	1.064	0.693
Model 3 (4C + 2FC)	0.625	0.787	0.987	0.712
Model 4 (5C + 3FC)	0.464	0.836	0.972	0.715
Model 5 (6C + 3FC)	0.332	0.841	0.995	0.726

Table 1: Results on the 5 neural networks

From the results, increasing the number of layers did yield generally higher validation metrics, but as the layers increased the validation metrics improved very marginally. This marginal increase is exemplified by how close the validation curves are for the deeper models shown in Figure 5 and Figure 6. Using the baseline model, Model 1 (3 Conv + 2FC) we saw that using data augmentation mitigated the overfitting on the model and improved generalization by a significant amount. The shallow baseline model tended to underfit the data. For the deeper models, there was a greater degree of overfitting after training was done as indicated by the difference between the validation and training metrics.

Models 3, 4 and 5 were very close in terms of validation metrics. Model 5 (6 Conv + 3 FC) performed the best in terms of accuracy, while model 4 (5C + 3 FC) performed the best in terms of the cross-entropy loss.

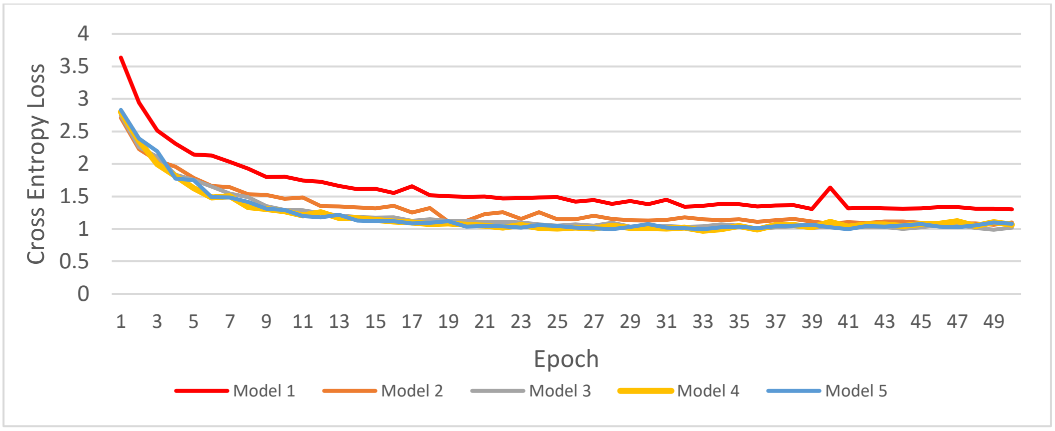

Figure 4: Validation Loss results

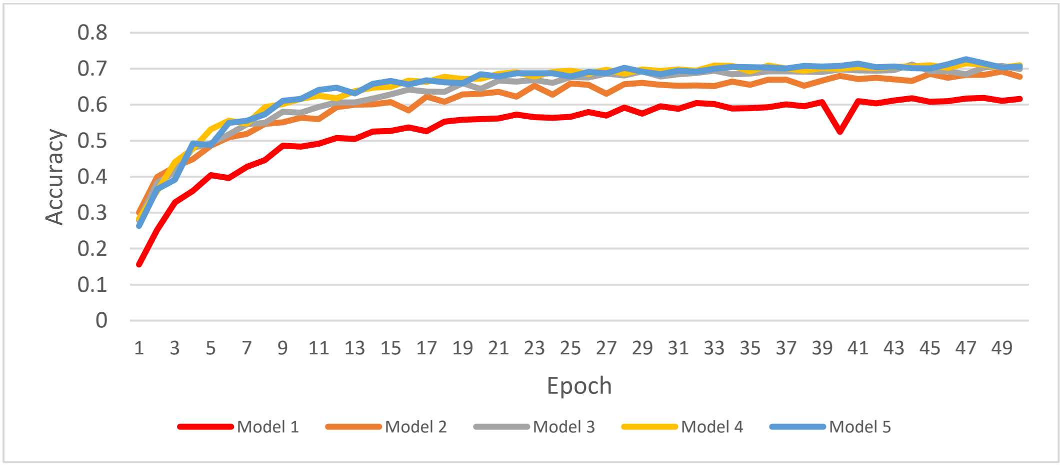

Figure 5: Validation Accuracy results

5. Different Optimization Algorithms

Adam [5] and Adagrad [6] are other popular optimizers that fall into the family of adaptive learning optimizers. These optimizers update each individual parameter to perform larger or smaller updates depending on their importance with a metric constructed from past gradients. We also consider the use of the Nesterov variant [6] for the momentum in the stochastic gradient algorithm that updates gradients based on approximate future parameters. We experiment using these optimizers on Model 5 (6C + 3FC), using the Pytorch default parameters for the optimizers. The results are shown below in Figure 7.

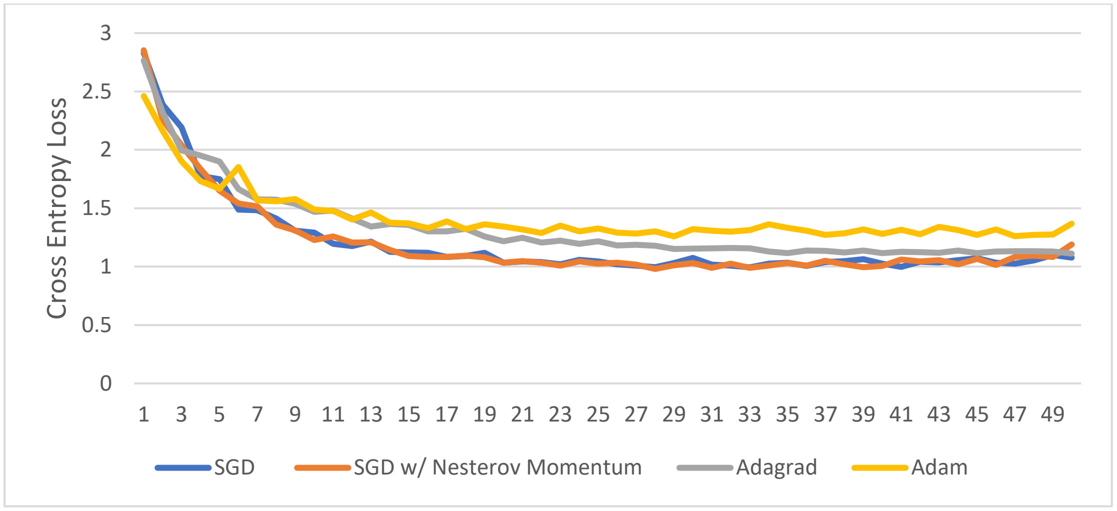

Figure 6: Validation loss from different optimizers on Model 5

The Adam optimizer performed the worse out of the other 3 optimizers tested. From the results, the Adam optimizer shows the best initial loss, but suffers after later epochs. There was little difference between the regular SGD and Nesterov variant. The Adagrad optimizer converged much more slowly compared to the SGD counterparts. Ashia et al. [8] observe the same results where the SGD optimizers performed much better than the adaptive learning optimizers in their experiments on various datasets.

6. Conclusion

Inspired by the successes of convolutional neural networks on the task of image classification, we utilized VGG-like convolutional neural networks onto the task of classifying plankton. To combat the problem of overfitting and poor generalization, we utilized data augmentation techniques which were able to mitigate those problems in our models. Our shallower models tended to underfit the data, but saw marginal improvements when more layers were added. We experiment using alternative adaptive gradient optimizers, but they did not outperform our original SGD optimizers. Moving forward in the future, we would like to implement other methods like creating deeper networks, applying other regularization techniques like dropout, and applying model ensembles to improve performance metrics.

References

[1] Kaggle. National Data Science Bowl. http://www.kaggle.com/c/datasciencebowl, 2014.

[2] Simonyan, K. and Zisserman A. Very deep convolutional networks for large-scale image recognition[j]. arXiv preprint, page arXiv:1409.1556, 2014.

[3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. 2012.

[4] Keskar, N., et al. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv preprint. page arXiv:1609.04836, 2016.

[5] Kingma, D. and Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the International Conference on Learning Representations 2015. 2015.

[6] Duchi, J. et al. Adaptive subgradient methods for online learning and stochastic optimization. The Journal of Machine Learning Research, 2011.

[7] Nesterov, Y. A method of solving a convex programming problem with convergence rate O(1/k 2). In Soviet Mathematics Doklady, volume 27, pages 372–376, 1983.

[8] Wilson, A. et al. The Marginal Value of Adaptive Gradient Methods in Machine Learning. arXiv preprint. arXiv:1705.08292, 2017.

Appendix A: Tables and Figures

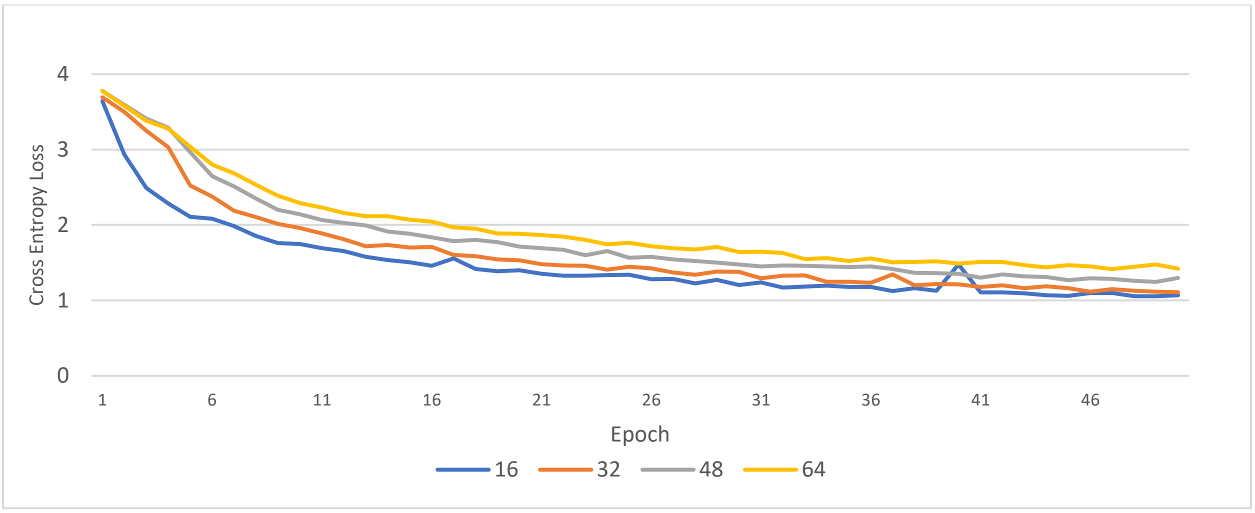

Figure 1: Validation loss by batch size using the baseline model

Model 1 2C → 2FC	**Model 2 3C → 2FC**	**Model 3 5C → 2FC**	**Model 4 5C → 3FC**	**Model 5 5C → 3FC**
Input: 6400	Input: 6400	Input: 6400	Input: 6400	Input: 6400
⇩	⇩	⇩	⇩	⇩
6 Conv K=5,S=1,P=0	20 Conv K=5,S=1,P=0	64 Conv K=3,S=1,P=1	64 Conv K=3,S=1,P=1	64 Conv K=3,S=1,P=1
⇩	⇩	⇩	⇩	⇩
Max Pool K=2,S=2	Max Pool K=2,S=2	Max Pool K=2,S=2	Max Pool K=2,S=2	64 Conv K=3,S=1,P=1
⇩	⇩	⇩	⇩	⇩
6 Conv K=5,S=1,P=0	40 Conv K=5,S=1,P=0	128 Conv K=5,S=1,P=0	128 Conv K=5,S=1,P=0	Max Pool K=2,S=2
⇩	⇩	⇩	⇩	⇩
Max Pool K=2,S=2	Max Pool K=2,S=2	128 Conv K=5,S=1,P=0	128 Conv K=5,S=1,P=0	128 Conv K=5,S=1,P=0
⇩	⇩	⇩	⇩	⇩
FC 400	60 Conv K=3,S=1,P=1	Max Pool K=2,S=2	Max Pool K=2,S=2	128 Conv K=5,S=1,P=0
⇩	⇩	⇩	⇩	⇩
FC 200	Max Pool K=2,S=2	256 Conv K=5,S=1,P=0	256 Conv K=5,S=1,P=0	Max Pool K=2,S=2
⇩	⇩	⇩	⇩	⇩
Output: 121	FC 400	256 Conv K=5,S=1,P=0	256 Conv K=5,S=1,P=0	256 Conv K=5,S=1,P=0
	⇩	⇩	⇩	⇩
	FC 200	Max Pool K=2,S=2	Max Pool K=2,S=2	256 Conv K=5,S=1,P=0
	⇩	⇩	⇩	⇩
	Output: 121	FC 400	FC 2000	Max Pool K=2,S=2
		⇩	⇩	⇩
		FC 200	FC 1000	FC 2000
		⇩	⇩	⇩
		Output: 121	FC 400	FC 1000
			⇩	⇩
			Output: 121	FC 400
				⇩
				Output: 121

Table 1: Architectures of the models tested in the project, ReLU activations after each convolutional/fully connected layer are not shown for brevity

Appendix B: Sample Python Code

```
# Import Libraries

import torch
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import ImageFolder
import torchvision.transforms as transforms
import torchvision


# Set to GPU

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


# ## Model #1: 2C → 2FC


# Baseline model

class Net1(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 17 * 17, 400)
        self.fc2 = nn.Linear(400, 200)
        self.fc3 = nn.Linear(200, 121)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 *17 *17)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


model1 = Net1()
```

```
model1.to(device)


optimizer1 = optim.SGD(model1.parameters(), lr=0.01,momentum=0.9)


# ## Model #2: 3C → 2FC



class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(20, 40, 5)
        self.conv3 = nn.Conv2d(40 , 60, 3,padding=1)
        self.fc1 = nn.Linear(60 * 8 * 8, 400)
        self.fc2 = nn.Linear(400, 200)
        self.fc3 = nn.Linear(200, 121)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 60 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


model2 = Net2()
model2.to(device)

optimizer2 = optim.SGD(model2.parameters(), lr=0.01,momentum=0.9)


# ## Model #3: 5C → 2FC

class Net3(nn.Module):
    def __init__(self):
        super(Net3, self).__init__()
        self.pool = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(1,  64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv3 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv4 = nn.Conv2d(128, 256, 3,padding=1)
        self.conv5 = nn.Conv2d(256, 256, 3,padding=1)
        self.fc1 = nn.Linear(256*10*10, 400)
        self.fc2 = nn.Linear(400, 200)
        self.fc3 = nn.Linear(200, 121)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = F.relu(self.conv2(x))
        x = self.pool(F.relu(self.conv3(x)))
        x = F.relu(self.conv4(x))
        x = self.pool(F.relu(self.conv5(x)))
        x = x.view(-1, 256 * 10 * 10)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


model3 = Net3()
model3.to(device)

optimizer3 = optim.SGD(model3.parameters(), lr=0.01,momentum=0.9)


# ## Model #4: 5C → 3FC



class Net4(nn.Module):
    def __init__(self):
        super(Net4, self).__init__()
        self.pool = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(1,  64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv3 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv4 = nn.Conv2d(128, 256, 3,padding=1)
        self.conv5 = nn.Conv2d(256, 256, 3,padding=1)
        self.fc1 = nn.Linear(256*10*10, 2000)
        self.fc2 = nn.Linear(2000, 1000)
        self.fc3 = nn.Linear(1000, 400)
        self.fc4 = nn.Linear(400, 121)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = F.relu(self.conv2(x))
        x = self.pool(F.relu(self.conv3(x)))
        x = F.relu(self.conv4(x))
        x = self.pool(F.relu(self.conv5(x)))
        x = x.view(-1, 256 * 10 * 10)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

```
model4 = Net4()
model4.to(device)

optimizer4 = optim.SGD(model4.parameters(), lr=0.004,momentum=0.9)


# ## Model #5: 6C → 3FC



class Net5(nn.Module):
    def __init__(self):
        super(Net5, self).__init__()
        self.pool = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(1,  64, 3, padding=1)
        self.conv2 = nn.Conv2d(64,  64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv5 = nn.Conv2d(128, 256, 3,padding=1)
        self.conv6 = nn.Conv2d(256, 256, 3,padding=1)
        self.fc1 = nn.Linear(256*10*10, 2000)
        self.fc2 = nn.Linear(2000, 1000)
        self.fc3 = nn.Linear(1000, 400)
        self.fc4 = nn.Linear(400, 121)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.pool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = self.pool(F.relu(self.conv6(x)))
        x = x.view(-1, 256 * 10 * 10)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x


model5 = Net5()
model5.to(device)

optimizer5 = optim.SGD(model5.parameters(), lr=0.004,momentum=0.9)


# ## Train & Evaluate functions
#
# Code is initially from : https://www.kaggle.com/juiyangchang/cnn-
     with-pytorch-0-995-accuracy
```

```
def train_model(epoch,model,optimizer,train_loader):
    model.train()
    #exp_lr_scheduler.step()

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

        if torch.cuda.is_available():
            data = data.cuda()
            target = target.cuda()

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

        loss.backward()
        optimizer.step()

        if (batch_idx + 1)% 300 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss:
     {:.6f}'.format(
                epoch, (batch_idx + 1) * len(data),
     len(train_loader.dataset),
                100. * (batch_idx + 1) / len(train_loader),
     loss.item()))




def evaluate(data_loader,model):
    model.eval()
    loss = 0
    correct = 0

    for data, target in data_loader:
        #data, target = Variable(data, volatile=True),
     Variable(target)
        if torch.cuda.is_available():
            data = data.cuda()
            target = target.cuda()

        output = model(data)

        loss += F.cross_entropy(output, target,
     reduction="sum").item()

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

    loss /= len(data_loader.dataset)
    print('\nAverage Train loss: {:.4f}, Accuracy: {}/{}
      ({:.3f}%)\n'.format(
        loss, correct, len(data_loader.dataset),
        100. * correct / len(data_loader.dataset)))

    t_loss.append(loss)
    t_acc.append(correct.item()/len(data_loader.dataset))



def evaluate_val(data_loader,model):
    model.eval()
    loss = 0
    correct = 0

    for data, target in data_loader:
        #data, target = Variable(data, volatile=True),
      Variable(target)
        if torch.cuda.is_available():
            data = data.cuda()
            target = target.cuda()

        output = model(data)

        loss += F.cross_entropy(output, target,
      reduction="sum").item()

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

    loss /= len(data_loader.dataset)

    print('\nAverage Validation loss: {:.4f}, Accuracy: {}/{}
      ({:.2f}%)\n'.format(
        loss, correct, len(data_loader.dataset),
        100. * correct / len(data_loader.dataset)))

    v_loss.append(loss)
    v_acc.append(correct.item()/len(data_loader.dataset))




# Set cross entropy loss

criterion = nn.CrossEntropyLoss()


# ## Get data and hold transformations
```

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, subset, transform=None):
        self.subset = subset
        self.transform = transform

    def __getitem__(self, index):
        x, y = self.subset[index]
        if self.transform:
            x = self.transform(x)
        return x, y

    def __len__(self):
        return len(self.subset)




# Hold transformations

transform_notrans = transforms.Compose(
    [transforms.Grayscale(num_output_channels=1),
     transforms.Resize((80,80)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.95,), std=(0.2,))])

transform_flips = transforms.Compose(
    [transforms.Grayscale(num_output_channels=1),
     transforms.RandomHorizontalFlip(),
     transforms.RandomVerticalFlip(),
     transforms.Resize((80,80)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.95,), std=(0.2,))])


transform = transforms.Compose(
    [transforms.Grayscale(num_output_channels=1),
     transforms.RandomHorizontalFlip(),
     transforms.RandomVerticalFlip(),

      transforms.RandomAffine(degrees=7,translate=(0.1,0.1),fillcolor=2
      55),
     transforms.Resize((80,80)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.95,), std=(0.2,))])


# Load data and get train/validation splits

data = ImageFolder(root='data/train')
```

```
train,validation = torch.utils.data.random_split(data, (len(data)-
     validation_size,validation_size))

validation = MyDataset(validation,transform = transform_notrans)
validation_loader = torch.utils.data.DataLoader(validation,
     batch_size=16, shuffle = True, num_workers=0)



# Sample a random image

x,y = data[np.random.randint(30336)]


plt.imshow(x.numpy()[0],cmap="gray")
plt.show()

print(data.classes[y])



train_notrans = MyDataset(train,transform = transform_notrans)
train_flips = MyDataset(train,transform = transform_flips)
train = MyDataset(train,transform=transform)



train_loader_notrans = torch.utils.data.DataLoader(train_notrans,
     batch_size=32, shuffle = True, num_workers=0)
train_loader_flips = torch.utils.data.DataLoader(train_flips,
     batch_size=32, shuffle = True, num_workers=0)
train_loader = torch.utils.data.DataLoader(train, batch_size=32,
     shuffle = True, num_workers=0)



trans_test = pd.DataFrame(index=[x for x in range(30)])



v_loss = []
v_acc = []



for epoch in range(30):
    train_model(epoch,model1,optimizer1,train_loader_notrans)
    evaluate(train_loader_notrans,model1)
    evaluate_val(validation_loader,model1)

trans_test = trans_test.assign(no1 = pd.Series(v_loss))
trans_test = trans_test.assign(no2 = pd.Series(v_acc))
```

```
# This is to reset the model

for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()


# to hold validation values

v_loss = []
v_acc = []


for epoch in range(30):
    train_model(epoch,model1,optimizer1,train_loader_flips)
    evaluate(train_loader_flips,model1)
    evaluate_val(validation_loader,model1)

trans_test = trans_test.assign(flip1 = pd.Series(v_loss))
trans_test = trans_test.assign(flip2 = pd.Series(v_acc))



for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()


v_loss = []
v_acc = []


for epoch in range(30):
    train_model(epoch,model1,optimizer1,train_loader)
    evaluate(train_loader,model1)
    evaluate_val(validation_loader,model1)

trans_test = trans_test.assign(all1 = pd.Series(v_loss))
trans_test = trans_test.assign(all2 = pd.Series(v_acc))



#save to csv file so we can create chart in Excel
trans_test.to_csv("trans_test.csv")


# # Determining the Batch size
# Create different train loaders with different batch sizes

train_loader16 = torch.utils.data.DataLoader(train, batch_size=16,
      shuffle = True, num_workers=4)
train_loader32 = torch.utils.data.DataLoader(train, batch_size=32,
      shuffle = True, num_workers=4)
train_loader48 = torch.utils.data.DataLoader(train, batch_size=48,
      shuffle = True, num_workers=4)
train_loader64 = torch.utils.data.DataLoader(train, batch_size=64,
      shuffle = True, num_workers=4)



# empty dataframe to hold results

train_batch_test = pd.DataFrame(index=[x for x in range(50)])
val_batch_test = pd.DataFrame(index=[x for x in range(50)])


# ## Using model 1 only

# ## Batch size = 16



for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()

n_epochs = 50

t = []
v = []


for epoch in range(n_epochs):
    train_model(epoch,model1,optimizer1,train_loader16)
    evaluate(train_loader16,model1)
    evaluate_val(validation_loader,model1)

train_batch_test = train_batch_test.assign(model1_bs16 = pd.Series(t))
val_batch_test = train_batch_test.assign(model1_bs16 = pd.Series(v))


# ## Batch size = 32


for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()
```

```
n_epochs = 50

t = []
v = []


for epoch in range(n_epochs):
    train_model(epoch,model1,optimizer1,train_loader32)
    evaluate(train_loader32,model1)
    evaluate_val(validation_loader,model1)

train_batch_test = train_batch_test.assign(model1_bs32 = pd.Series(t))
val_batch_test = train_batch_test.assign(model1_bs32 = pd.Series(v))


# ## Batch size = 48


for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()

n_epochs = 50

t = []
v = []


for epoch in range(n_epochs):
    train_model(epoch,model1,optimizer1,train_loader48)
    evaluate(train_loader48,model1)
    evaluate_val(validation_loader,model1)

train_batch_test = train_batch_test.assign(model1_bs48 = pd.Series(t))
val_batch_test = train_batch_test.assign(model1_bs48 = pd.Series(v))


# ## Batch size = 64


for layer in model1.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()

n_epochs = 50

t = []
v = []


for epoch in range(n_epochs):
    train_model(epoch,model1,optimizer1,train_loader64)
    evaluate(train_loader64,model1)
    evaluate_val(validation_loader,model1)

train_batch_test = train_batch_test.assign(model1_bs64 = pd.Series(t))
val_batch_test = train_batch_test.assign(model1_bs64 = pd.Series(v))



# output as csv

val_batch_test.to_csv("Batch_Size.csv")


# # Train models!


train_loss.to_csv("trainloss.csv")
train_acc.to_csv("trainacc.csv")
val_loss.to_csv("valloss.csv")
val_acc.to_csv("valacc.csv")





# dataframes to hold results

train_loss = pd.DataFrame(index=[x for x in range(50)])
train_acc = pd.DataFrame(index=[x for x in range(50)])
val_loss = pd.DataFrame(index=[x for x in range(50)])
val_acc  =  pd.DataFrame(index=[x for x in range(50)])

# 50 epochs
n_epochs = 50


# ## Model 1


t_loss = []
t_acc = []

v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model1,optimizer1,train_loader16)
    evaluate(train_loader16,model1)
    evaluate_val(validation_loader,model1)
```

```
train_loss = train_loss.assign(Model_1 = pd.Series(t_loss))
val_loss = val_loss.assign(Model_1 = pd.Series(v_loss))
train_acc = train_acc.assign(Mode1_1 = pd.Series(t_acc))
val_acc = val_acc.assign(Model_1=pd.Series(v_acc))


# ## Model 2



t_loss = []
t_acc = []

v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model2,optimizer2,train_loader16)
    evaluate(train_loader16,model2)
    evaluate_val(validation_loader,model2)




train_loss = train_loss.assign(Model_2 = pd.Series(t_loss))
val_loss = val_loss.assign(Model_2 = pd.Series(v_loss))
train_acc = train_acc.assign(Mode1_2 = pd.Series(t_acc))
val_acc = val_acc.assign(Model_2=pd.Series(v_acc))


# ## Model 3



t_loss = []
t_acc = []

v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model3,optimizer3,train_loader16)
    evaluate(train_loader16,model3)
    evaluate_val(validation_loader,model3)
```

```
train_loss = train_loss.assign(Model_3 = pd.Series(t_loss))
val_loss = val_loss.assign(Model_3 = pd.Series(v_loss))
train_acc = train_acc.assign(Model_3 = pd.Series(t_acc))
val_acc = val_acc.assign(Model_3=pd.Series(v_acc))


# ## Model 4


t_loss = []
t_acc = []

v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model4,optimizer4,train_loader16)
    evaluate(train_loader16,model4)
    evaluate_val(validation_loader,model4)




train_loss = train_loss.assign(Model_4 = pd.Series(t_loss))
val_loss = val_loss.assign(Model_4 = pd.Series(v_loss))
train_acc = train_acc.assign(Model_4 = pd.Series(t_acc))
val_acc = val_acc.assign(Model_4 =pd.Series(v_acc))


# ## Model 5


t_loss = []
t_acc = []

v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model5,optimizer5,train_loader16)
    evaluate(train_loader16,model5)
    evaluate_val(validation_loader,model5)
```

```
train_loss = train_loss.assign(Model_5 = pd.Series(t_loss))
val_loss = val_loss.assign(Model_5 = pd.Series(v_loss))
train_acc = train_acc.assign(Model_5 = pd.Series(t_acc))
val_acc = val_acc.assign(Model_5 =pd.Series(v_acc))


# Save to csv file for graphing results on excel

train_loss.to_csv("trainloss2.csv")
train_acc.to_csv("trainacc2.csv")
val_loss.to_csv("valloss2.csv")
val_acc.to_csv("valacc2.csv")


# ## Different Optimizations!



# empty dataframe to hold values

Opt_val_loss  =  pd.DataFrame(index=[x for x in range(50)])
Opt_val_acc  =  pd.DataFrame(index=[x for x in range(50)])




# we use Model 5

class Net5(nn.Module):
    def __init__(self):
        super(Net5, self).__init__()
        self.pool = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(1,  64, 3, padding=1)
        self.conv2 = nn.Conv2d(64,  64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv5 = nn.Conv2d(128, 256, 3,padding=1)
        self.conv6 = nn.Conv2d(256, 256, 3,padding=1)
        self.fc1 = nn.Linear(256*10*10, 2000)
        self.fc2 = nn.Linear(2000, 1000)
        self.fc3 = nn.Linear(1000, 400)
        self.fc4 = nn.Linear(400, 121)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.pool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = self.pool(F.relu(self.conv6(x)))
        x = x.view(-1, 256 * 10 * 10)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x


model5 = Net5()
model5.to(device)


# Initialize diffferent optimizers
optimizer5_nesterov = optim.SGD(model5.parameters(),
     lr=0.004,momentum=0.9,nesterov=True)
optimizer5_adagrad = optim.Adagrad(model5.parameters())
optimizer5_adam = optim.Adam(model5.parameters())


# ## Train with Nesterov's Algorithm


v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model5,optimizer5_nesterov,train_loader16)
    evaluate(train_loader16,model5)
    evaluate_val(validation_loader,model5)


Opt_val_loss = Opt_val_loss.assign(Nesterov = pd.Series(v_loss))
Opt_val_acc = Opt_val_acc.assign(Nesterov = pd.Series(v_acc))


# ## Adagrad


for layer in model5.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()


v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model5,optimizer5_adagrad,train_loader16)
    evaluate(train_loader16,model5)
    evaluate_val(validation_loader,model5)


Opt_val_loss = Opt_val_loss.assign(Adagrad = pd.Series(v_loss))
Opt_val_acc = Opt_val_acc.assign(Adagrad = pd.Series(v_acc))


# ## Adam


for layer in model5.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()


v_loss = []
v_acc = []



for epoch in range(n_epochs):
    train_model(epoch,model5,optimizer5_adam,train_loader16)
    evaluate(train_loader16,model5)
    evaluate_val(validation_loader,model5)


Opt_val_loss = Opt_val_loss.assign(Adam = pd.Series(v_loss))
Opt_val_acc = Opt_val_acc.assign(Adam = pd.Series(v_acc))


# save results!

Opt_val_loss.to_csv("opt_val.csv")
Opt_val_acc.to_csv("opt_val_acc.csv")
```