University of Prince Mugrin
College of Computer and Cyber Sciences
Department of Artificial Intelligence

AI417- Introduction to Deep Learning

Course Project – Semester II (Spring 2023)

Students' Emotion Detection Model

Team Members:

Aisha Ahmad | 4010058

Salwa Shama | 4010405

Samah Shama | 4010403

Sana Shama | 4010404

Instructor:

Dr. Ahmed Elhayek

May 22, 2023

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Overview of The Project

To achieve UPM's objective of providing excellent academic programs and a supportive learning environment for students, a deep learning-based model for emotion detection using face recognition can be designed and developed. This model can help monitor student engagement and emotional responses to learning materials during lectures. Additionally, it can be used for statistical research to analyse student reactions to various activities and events organized by different entities at UPM (e.g., clubs) to rank them based on their positive influence.

Overall, this project will help the instructors gain insights into the effectiveness of their teaching methods and adjust them as needed. In addition, it offers objective measures of emotional responses, allowing for better-informed decision-making regarding future actions. In the end, this will lead to in a more productive and interesting learning environment for UPM students.

CHAPTER 2: DATASET

2.1 Data Description

There are many popular data sets on the Web that can be used for facial expression recognition; in our case, we used different kinds of data sets. The first one is FER2013, which contains 28709 examples in the training set and 3589 in the validation set. This dataset divides the emotion into seven feeling expressions: 0 = anger, 1 = disgust, 2 = fear, 3 = happy, 4 = sad, 5 = surprise, and 6 = neutral [1]. The second one is CKPLUS, which contains 981 grayscale training face images and has the same 7 categories of feeling expressions as the FER2013 datasets [2]. Also we used FERPlus the new version of FERP2013, with more quality images and the same number of images, but the difference here is that instead of having 7 categories, we now have 5: 0 = angry, 1 = happy, 2 = neutral, 3 = disgust, and 4 = fear.

CHAPTER 3: DEVELOPMENT MODEL PLAN

3.1 Developing Plan

AI model building planning is key to ensuring that the developing effort, the spent time, and the consumed resources are well organized. It is also a super important piece of the puzzle for delivering high-accuracy models.

By the way, there is no silver bullet for building models because it tightly depends on the project itself and the dataset. In our project "Students' Emotion Detection ", we will be following this plan: As we can see, this plan consists of two parts, one related to model components and another related to model architecture. In the first part, we collected the most styles and

components that have achieved high accuracy whatever for emotion detection or similar applications. The second part is about CNN architectures that have achieved high accuracy whatever by using the same dataset or a different one.

Plan ID	Description				Reason	Accuracy			Effect	Comments	Priority
Plan 00	Original model				XXX		58.10%		X	X	X
Plan 01	Using a more advanced activation function				It helps to enabling the network to model more complex and non-linear relationships between the input and output data, and by addressing some of the limitations of traditional activation functions.				Components		2
	ReLU	LeakyReLU	ELU + 50 epochs								
	Swish	Mish	GELU + 50 epochs								
Plan 02	Using a different optimizer				It helps network to converge faster and avoid local optima, leading to better accuracy and faster training times.						3
	Adam	RMSProp	Adagrad								
Plan 03	Increasing the number of kernels (30)				It helps the network to detect a wider range of features.						1
	64, 128, 128, 256 kernels	64, 128, 256,512 kernels	512, 1024, 2048, 4096 kernels	256, 512, 1024, 2048 kernels							
Plan 04	Increasing the depth of the model				It helps the network to learn more complex representations of the input data.				CNN Architecture		1
	3 COV + 2 FC	5 COV + 3 FC	8 COV + 2 FC								
Plan 05	combine of plan 1 and plan 2				It helps the network to learn more complex representations of the input data & It helps the network to detect a wider range of features (2 in 1)						1
Plan 06	**CNN Architecture:** input (1 X 48 X 48) - COV (32 X 5 X 5) - Batch - Swish - Max (2 X 2) - COV (64 X 5 X 5) - Batch - Swish - Max (2 X 2) - COV (128 X 5 X 5) - Batch - Swish - Max - COV (256 X 5 X 5) - Batch - Swish - Max (2 X 2) - Flatten - FC (1024) - Batch - Swish - Dropout - Softmax - Output				XXX					Starting point	4
Plan 07	**CNN Architecture**: Input (48 X 48 X 1) - COV (32 X 3X 3) - Batch - ReLU - COV (32 X 3 X 3) - Batch - ReLU - Max (2 X 2) - Dropout (0.25) - COV (64 X 3 X 3) - Batch - ReLU - COV (64 X 3 X 3) - Batch - ReLU - Max (2 X 2) - Dropout (0.25) - COV (128 X 3 X 3) - Batch - ReLU - COV (128 X 3 X 3) - Batch - ReLU - Max (2 X 2) - Dropout (0.25) - COV (256 X 3 X 3) - Batch - ReLU - COV (256 X 3 X 3) - Batch - ReLU - Max (2 X 2) - DropOut (0.25) - Flatten - Fully (512) - Batch - ReLU - Dropout (0.5) - Fully (7) - softmax - output				XXX						1
Plan 08	**CCN Architecture :** Input (48 X 48 X 1) - COV (64 X 3 X 3) - Batch - Leaky ReLU - COV (64 X 3 X 3) - Batch - Leaky ReLU - Max (2 X 2) - Dropt (0.25) - COV (128 X 3 X 3) - Batch - Leaky ReLU - COV (128 X 3 X 3) - Batch - Leaky - Max (2 X 2) - Dropout (0.25) - COV (256 X 3 X 3) - Batch - Leaky ReLU - Max (2 X 2) - Dropout (0.25) - COV (512 X 3 X 3) - Batch - Leaky ReLU - COV (512 X 3 X 3) - Batch - Leaky ReLU - Max (2 X 2) - Dropout (0.25) - Flatten - FC (512) - Batch - Leaky ReLU - Dropout (0.5) - FC (7) - Softmax - Output				XXX					Ths Archiecture achieved an accuracy of 99.3% on the test set of CK+ dataset.	1
Plan 09	Skip connection				XXX						4

Figure 3.1 – Developing Plan

CHAPTER 4: BULID THE MODEL

4.1 Training the Model

Applying our plan to FER 2013, we got the following results: As you can see, unfortunately, the accuracy is not very high. After searching, we found that the highest accuracy that is achieved by this dataset by using CNN is 73.28%, and there is a new version of this dataset called FER2013+ that addresses some issues with FER2013. Even though we had some problems training this data, we played with the CNN architecture, and the highest result we got was 65.1%. For more detail on the training result, visit Chapter 5: Analysis and Discussion [3].

Plan ID	Sub plan				Accuracy				Comments		
Plane 01	Using a more advanced activation function										
	ReLU + 50 epochs	LeakyReLU + 50 epochs	ELU + 50 epochs		Origin	61.40%	58.59%		XXX	f(x) = { x if x > 0, { alpha * x if x <= 0, - Where alpha is a small constant value typically set to 0.01. - It can help to prevent the "dying ReLU" problem, where ReLU neurons can become "dead" and stop outputting any value other than 0. - This can happen when a large number of neurons output negative values, causing the weights to update in a way that makes the neurons always output 0. - LeakyReLU helps to prevent this by allowing a small gradient to flow through the negative values.	f(x) = { x if x > 0, { alpha * (exp(x) - 1) if x <= 0, - Where alpha is a small constant value typically set to 1.0. -It uses an exponential function for negative input values. - It is that it can help to prevent the "dying ReLU" problem by allowing a small gradient to flow through the negative values.
	Swish	Mish	GELU + 10 epochs		Swish is not a built-in activation function in PyTorch	Swish is not a built-in activation function in PyTorch	61.35%		XXX	XXX	f(x) = 0.5 * x * (1 + erf(x / sqrt(2))) -Where erf is the error function. - It uses a different non-linear function for the positive part of the input. - It can help to improve the performance of deep learning models by adding non-linearity to the positive part of the input. - It has a non-zero gradient for all inputs, which can help to prevent the "dying ReLU" problem. - It can be slower than ReLU due to the use of the error function
Plane 02	Using a Different Optimizer										
	Adam	RMSProp	Adagrad		60.73%	57.71%	56.40%		Adam: 1. Works well for large datasets and/or deep architectures. 2. Can converge faster than both AdaGrad and RMSProp. 3. More computationally expensive than AdaGrad and RMSProp.		
Plane 03	Increasing the number of kernels (30)										
	64, 128, 256, 512 kernels	64, 128, 128, 256 kernels	512, 1024, 2048, 4096 kernels	256, 512, 1024, 2048 kernels	61.27%	61.32%	64.41%	65.10%	It helps the network to detect a wider range of features.		
Plane 04	Increasing the depth of the model										
	3 COV + 2 FC	5 COV + 3 FC	8 COV + 2 FC		60.63%	61.05%	64.34%		It helps the network to learn more complex representations of the input data.		
Plane 08	Plan 8 + 50 epochs				65.16%						

Figure 4.1 – FER2013 Training Results

Plan ID	Sub plan				Accuracy				Comments
Plane 03	Increasing the number of kernels (30)								
	64, 128, 256, 512 kernels	64, 128, 128, 256 kernels	512, 1024, 2048, 4096 kernels	256, 512, 1024, 2048 kernels	67.80%	67.46%	67.26%	68.89%	It helps the network to detect a wider range of features.
Plane 04	Increasing the depth of the model								
	3 COV + 2 FC	8 COV + 3 FC	8 COV + 2 FC		61.93%	70.37%	XXX		It helps the network to learn more complex representations of the input data.
Plane 08	Plan 8 + 50 epochs								

Figure 4.2 – FERPlus Training Results

As we can see in Figure 4.1, yes, not all plans are applied because, at the beginning, we felt there was something wrong and the model accuracy didn't improve a lot, and we figured out the reason that we mentioned earlier. As they say, "You don't need to drink the entire ocean to confirm that it is salty". These few plans were enough for us to discover we had a problem. In terms of FERPlus dataset, the highest result we achieved was 70.37%.

The same plan was applied to the CKPLUS dataset, and as we can see, there are better results in the training, with the highest accuracy being XXX, but honestly, in the testing data, the results are not very good, and that was expected for the small amount of data we trained the model on. For more details on training results, visit Chapter 5: Analysis and Discussion.

Plan ID	Sub plan			Accuracy			Comments		
Plane 01	Using a more advanced activation function								
	ReLU + 50 epochs	LeakyReLU + 50 epochs	ELU + 50 epochs	97.97%	97.97%	98.98%	XXX	f(x) = { x if x > 0, { alpha * x if x <= 0, - Where alpha is a small constant value typically set to 0.01. - It can help to prevent the "dying ReLU" problem, where ReLU neurons can become "dead" and stop outputting any value other than 0. - This can happen when a large number of neurons output negative values, causing the weights to update in a way that makes the neurons always output 0. - LeakyReLU helps to prevent this by allowing a small gradient to flow through the negative values.	f(x) = { x if x > 0, { alpha * (exp(x) - 1) if x <= 0, - Where alpha is a small constant value typically set to 1.0. -It uses an exponential function for negative input values. - It is that it can help to prevent the "dying ReLU" problem by allowing a small gradient to flow through the negative values.
	Swish	Mish	GELU + 10 epochs	Swish is not a built-in activation function in PyTorch	Swish is not a built-in activation function in PyTorch	97.46%	XXX	XXX	f(x) = 0.5 * x * (1 + erf(x / sqrt(2))) -Where erf is the error function. - It uses a different non-linear function for the positive part of the input. - It can help to improve the performance of deep learning models by adding non-linearity to the positive part of the input. - It has a non-zero gradient for all inputs, which can help to prevent the "dying ReLU" problem. - It can be slower than ReLU due to the use of the error function
Plane 02	Using a Diffrent Optimizer								
	Adam	RMSProp	Adagrad	95.94%	84.77%	92.39%	Adam: 1. Works well for large datasets and/or deep architecures. 2. Can converge faster than both AdaGrad and RMSProp. 3. More computationally expensive than AdaGrad and RMSProp.		
Plane 03									
Plane 04									
Plane 05									
Plane 08	Plan 8 + 50 epochs			98.98%					

Figure 4.3 – CKPLUS Training Results

Before wrapping up this chapter, it must be mentioned that we tried many architectural model styles with different combinations, and those were just samples.

4.2 Real- Time Testing

In order to ensure the accuracy and relevance of our "Students' Emotion Detection" project, it was crucial to implement a testing model that could operate in real time using data from a camera feed. To achieve this, we leveraged the capabilities of the OpenCV (Open Source Computer Vision) library.

OpenCV is a powerful software library that specializes in computer vision and machine learning. It offers a comprehensive suite of tools, functions, and algorithms designed to handle a wide range of computer vision tasks. These include image and video processing, object detection and tracking, feature extraction, camera calibration, and more.

By harnessing the capabilities of OpenCV, we were able to develop a real-time testing model that seamlessly integrates with a camera. This model can capture live images from the camera and apply our trained emotion detection model on the spot. This allows us to instantly analyze the emotions of students as they are being captured by the camera, providing valuable insights and real-time feedback.

By utilizing the OpenCV library, we have enhanced our project with the ability to perform emotion detection in real time, making it more applicable and practical in various educational settings.

CHAPTER 5: ANALYSIS AND DISCUSSION

This chapter is the thunder of this report. We will discuss what we have learned from this project by going through the following topics: Successful Design Features and Reasons, Unsuccessful Design Features and Reasons, Additional Design Features and Improvements, Reasons for Applying Some Changes and Change's Reason that Resolved the Issue.

5.1 Successful Design Features and Reasons

The best performing model achieved 70% accuracy on a clean version of the FER2013 dataset. It is composed of 8 convolutional layers and 3 fully connected layers. Let's examine the individual components of the model in more detail.

1. Convolutional Layers: The model has four convolutional layers (conv1, conv2, conv3, conv4). Each convolutional layer uses a 3x3 kernel size with a stride of 1 and padding of 1. The first convolutional layer takes in a single channel (grayscale) image, while the remaining layers take in 64, 128, 256, and 512 channels respectively. Each convolutional layer is followed by a batch normalization layer which normalizes the activations of the previous layer. This is followed by a ReLU activation function and each convolutional layer is also followed by a max pooling layer which reduces the spatial dimensions of the output by a factor of 2. The last ReLU activation function in the fully connected layers is followed by a dropout layer with a dropout rate of 0.5 to prevent overfitting.
2. Fully Connected Layers: The model has three fully connected layers. The first fully connected layer takes in the flattened output of the last convolutional layer, which has a shape of 512x3x3. The first fully connected layer has 120 output units, followed by

a ReLU activation function. The second fully connected layer has 84 output units, followed by another ReLU activation function. The third fully connected layer has 5 output units, which corresponds to the number of classes that the model is trained to classify.

To improve the model's performance, we increased the number of kernels in the convolutional layers, allowing it to detect a wider range of features. We also intensified the model's depth, enabling it to learn more complex representations of the input data. These modifications were the primary drivers behind the observed enhancement in performance.

5.2 Unsuccessful Design Features and Reasons

FER2013 Dataset:

In plan 1, we tried to modify the components of the model by changing the activation function. First, we tried changing the activation function to LeakyReLU and at the same time changed the number of epochs to 50, but we got an accuracy of 61.40%, compared to the original accuracy of 70% for these datasets. Then, we tried a second attempt by changing the activation function to GELU with 10 epochs, but the accuracy decreased to become 61.35% and the same thing happened when we tried using ELU with 50 epochs and get an accuracy of 58.58%. There are another activation functions we planned to apply but unfortunately, Pytorch doesn't support them.

In plan 2, the goal of this plan is to use different types of optimizers we start with the Adam function and got 60.73%, but we when change it to RMSProp and Adagrad, the accuracy of the function decreased to become 57.71% and 56.40% respectively. So, we continue to use the Adam optimizer.

In plan 3 we tried to increase the number of kernels. In the first two sub plans we used 513 and 256 kernels, but the accuracy of the model doesn't change that much, it increased from 61.27% to 61.32%. However, after increasing the number of kernels, we noticed that the accuracy increased to 65.10% with 2048 kernels.

All previous plans we were to change some features in the model, but now in plans 4 and 8 we try to change the architecture of the model. Here in plan 4, we try to increase the depth of the model. We tried increasing the number of conventional layers to 3, followed by 2 fully connected layers, we get 60.63% so, we thought it is good so, so we increased the number of conventional layers to become 5 conventional layers with 3 fully connected layers, but the accuracy not increased that it much, it just 1 % to become 61.05% and last try here we got 64.35% accuracy when we increased the conventional layer to become 8 and followed by 2 fully connected layer.

Finally, we have a plan 8 here, we got the best accuracy by using this dataset where the accuracy was 65.16%.

CKPLUS Dataset:

Here, we used the same planed that we used with the FER2013 dataset. We change the dataset because we didn't get a high accuracy when we used it, so we decided to change it to the CK+ dataset. We applied the same previous plan to this new dataset, and we got high accuracy reach to 98.98% in the first plan and 95. 94% of the second plan, but we did not go further than that, why? because we discovered later that Dataset had a few images (training dataset) compared to the previous Dataset, so it gave good results in training, but when we test the model, unfortunately, it gave very low accuracy, reaching 22%.

5.3 Additional Design Features and Improvements

The story of training the model doesn't stop at this plan that we wrote, there are a lot of improvements and techniques that can be added. In terms of model components, we can play with the type of pooling and size of strides; we can also increase the number of epochs; we can modify the regularization techniques, in this case dropout; in terms of the architect itself, we can apply skip connections to the model; and we can also try to use the transform model.

5.4 Reasons for Applying Some Changes

1- Changing the dataset: The dataset was old and not clean, containing outdated or irrelevant data. This posed a problem as using outdated or irrelevant data could lead to inaccurate or biased predictions.

2- Increasing dataset size: The dataset was too small for training, leading to overfitting. Overfitting occurs when a model becomes too specialized in the training data and fails to generalize well to unseen data.

3- Change the model architecture: The low accuracy observed during both training and testing indicated that the current model architecture was not effective in capturing the underlying patterns in the data.

5.5 Change's Reason that Resolved the Issue

1- Changing the dataset:

By changing the dataset, we could ensure the availability of up-to-date and relevant information. This change helped in improving the quality of the data used for training and ensured that the model learned from the most recent and relevant information.

To address this issue, we obtained a cleaned dataset by removing outdated or irrelevant data points and ensuring that the data was up-to-date. By using a cleaned dataset, we ensured that the model learned from the most relevant and recent information, improving its ability to make accurate predictions.

2- Increasing dataset size:

Increasing the dataset size allowed for better generalization and reduced overfitting. With a larger dataset, the model had access to more diverse examples, enabling it to learn a wider range of patterns and make more accurate predictions on new, unseen data.

3- Change the model architecture:

By changing the model architecture, we introduced a new network design that was better equipped to learn and capture the complex patterns and relationships within the data. This change involved adding more layers, adjusting the number of neurons, and experimenting with different activation functions.

The new model architecture had a higher capacity to learn and generalize from the data, resulting in improved accuracy during the training and testing phases. It was able to better extract relevant features and representations from the data, leading to more accurate predictions.

CHAPTER 6: CONCLUSION

6.1 Conclusion

In conclusion, we applied our knowledge of deep learning to our project. We began by writing a comprehensive training plan to re-architect the model of CNN by going through fully connected layers, depth neural networks, and convolutional neural network concepts. Throughout the project, we utilized the techniques and concepts we learned in our lectures and improved our understanding through hands-on application. Also, we learned something very important: even if you have a good architecture model, that doesn't mean you will get high accuracy because both models and the cleans of data are highly coupled together. As Abraham Lincom said, "Give me six hours to chop down a tree, and I will spend the first four sharpening the axe", The same story can be used in preparing the data and then training the model. Overall, this project allowed us to gain practical experience and reinforce our understanding of the course material.

6.2 References

[1] M. Sambare, “Fer-2013,” Kaggle, https://www.kaggle.com/datasets/msambare/fer2013 (accessed May 21, 2023).

[2] A. Shawon, “CKPLUS,” Kaggle, https://www.kaggle.com/datasets/shawon10/ckplus?datasetId=65125&sortBy=voteCount (accessed May 21, 2023).

[3] Y. Khaireddin and Z. Chen, “Facial emotion recognition: State of the art performance on FER2013,” arXiv.org, https://arxiv.org/abs/2105.03588 (accessed May 21, 2023).

APPENDICES

1. Students' Emotion Detection Code

Below is the source code for our students' emotion detection program, which is implemented in Python.

```
!git clone https://github.com/sana-shamma/AI-417-Project.git
```

```
[ ]  # import
     import torch
     import torch.nn as nn                  #for sequence api in torch
     from torch.utils.data import DataLoader #for loading images
     import numpy as np                     #just in case if you need numpy arrays
     import torchvision.transforms as T     #Used for data preprocessing and converting images to tensors
     import torchvision.datasets as dset
     import torch.optim as optim            #For using the desired parameter update
     import torch.nn.functional as F
```

```
[ ]  USE_GPU = True

     if USE_GPU and torch.cuda.is_available():
         device = torch.device('cuda')
     else:
         device = torch.device('cpu')

     dtype = torch.float32

     print("Using device: ",device)

     #----------------------------------------Loading Dataset-------------------------------------------------#
     transform = T.Compose([T.RandomHorizontalFlip(), T.Grayscale(num_output_channels=1), T.ToTensor()])

     #Training
     train_data = dset.ImageFolder("/content/AI-417-Project/train",transform=transform)
     loaded_train = DataLoader(train_data,batch_size=64,shuffle=True)

     #Validation
     validation_data = dset.ImageFolder("/content/AI-417-Project/validation",transform=transform)
     loaded_validation = DataLoader(validation_data,batch_size=64,shuffle=False)

     loss_history = []
     validation_acc = []
     training_acc = []
     #--------------------------------------------------------------------------------------------------------#
```

Figure 1 – Loading Dataset

```
#--------------------Creating a method for predicting validation accuracy----------------------#
#
# Computes the accuracy of the given model on the given data loader.
# Args:
#     loader: A PyTorch DataLoader object that provides a stream of input data.
#     model: A PyTorch model object that takes input data and produces output scores.
# Returns:
#     None. Prints the accuracy of the model on the given data loader.
#
def check_accuracy_part(loader, model):
    print('Checking accuracy on validation set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=torch.float)
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc))
#-----------------------------------------------------------------------------------------------#
```

Figure 2 – Validation Function

```python
#---------------------------------------Visualizing the image-----------------------------------#
import torch
import torchvision
import random

# Assume that 'loaded_train' is a PyTorch DataLoader object containing the dataset
dataiter = iter(loaded_train)
images, labels = next(dataiter)

# Create a dictionary for mapping labels to expressions
expression = {0: "angry", 1: "disgust", 2: "fear", 3: "happy", 4: "neutral", 5: "sad", 6: "surprise"}

# Select a random image to display
random_idx = random.randint(0, 63)
print("Target label: ", expression[int(labels[random_idx].item())])

# Convert the image tensor to a numpy array and transpose the dimensions to match matplotlib's format
image_np = images[random_idx].permute(1, 2, 0).numpy()

# Convert the data type to uint8
image_np = (image_np * 255).astype(np.uint8)

# Display the image using PyTorch's built-in image display function
torchvision.transforms.functional.to_pil_image(image_np).show()
#-----------------------------------------------------------------------------------------------#
```

Figure 3 – Visualizing Images

```python
#-------------------------------------------------Trianing Model--------------------------------------#
#
# Trains the given model on the given optimizer using the given number of epochs.
# Args:
#     model: A PyTorch model object to be trained.
#     optimizer: A PyTorch optimizer object to use for gradient descent.
#     epochs: An integer specifying the number of epochs to train for (default 1).
# Returns:
#     None. Trains the model in-place and prints the loss and accuracy during training.
#
def train_part(model, optimizer, epochs=1):

    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        print("epoch: ",e+1)
        for t, (x, y) in enumerate(loaded_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % 100 == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part(loaded_validation, model)
                print()
#-----------------------------------------------------------------------------------------------#
```

Figure 4 – Training Model

```
#-------------------------------------Arh Model-----------------------------------------------------#

model = None
optimizer = None

#First architecture #1,32,32
conv1 = nn.Sequential(
    nn.Conv2d(1,512,kernel_size=(3,3),bias=True,padding=1), #512,48,48
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2,2))  #Sampling image to half  512,24,24
)
conv2 = nn.Sequential(
    nn.Conv2d(512,128,kernel_size=(3,3),padding=1,bias=True), #128,24,24
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2,2))    #128,12,12
)
conv3 = nn.Sequential(
    nn.Conv2d(128,64,kernel_size=(3,3),bias=True,padding=1), #64,12,12
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2,2))   #64,6,6
)
conv4 = nn.Sequential(
    nn.Conv2d(64,256,kernel_size=(3,3),bias=True,padding=1), #64,6,6
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2,2))   #256,3,3
)
fc = nn.Sequential(
    nn.Flatten(),
    nn.Linear(256*3*3,7),
)
model = nn.Sequential(
    conv1,
    conv2,
    conv3,
    conv4,
    fc
)
learning_rate=0.001
optimizer = optim.Adam(model.parameters(),lr=learning_rate)
train_part(model, optimizer, epochs=10)
#-------------------------------------------------------------------------------------------------#
```

Figure 5 – Model Arch

```
#--------------------------------Best Pefrmance--------------------------------#
#Best model
best_model = model
check_accuracy_part(loaded_validation,best_model)
#------------------------------------------------------------------------------#
```

Figure 6 – Model Accuracy