Hexadecimal Image Captcha classification using CNN

Aman Sah*
M.tech in Aerospace Engineering
Rahul Dhali
M.tech in Aerospace Engineering
Ganesh Valtule
M.Tech (Department of Aerospace Engineering)

Image processing, CNN, Hexadecimal Image Captcha, Logit loss, Adam optimization, parity classification

Abstract

The objective of this report is to develop a solution for classifying the parity (odd or even) of hexadecimal numbers presented in CAPTCHA images. Each CAPTCHA image is 500×100 pixels and contains a four-digit hexadecimal code, where the characters may be rotated and rendered with different colors. The background colors are light in shade, and stray lines of varying thickness and color are added to enhance realism. To accomplish this task, a dataset consisting of 2000 training images with corresponding labels indicating the parity of the hexadecimal numbers is provided. Additionally, reference images of unrotated characters 0-9 and A-F are given for reference purposes.

The approach to solving this problem involves developing a machine-learning model capable of extracting relevant features from the CAPTCHA images and making accurate predictions regarding the parity of the hexadecimal number. Techniques such as image preprocessing, feature extraction, and classification algorithms will be explored and evaluated to identify the most effective solution.

The accuracy and performance of the developed model will be assessed through appropriate evaluation metrics using a validation dataset. The report will also discuss any challenges encountered during the development process and potential strategies for improving the model's performance.

Ultimately, this project aims to provide an automated solution for determining the parity of hexadecimal numbers in CAPTCHA images, which can have practical applications in various domains, such as web security and authentication systems.

1 Introduction

The given data set is shown below in Fig(1), representing 2000 such image captcha. Each CAPTCHA image in this assignment will be 500×100 pixels in size. Each image will contain a code that is a 4-digit hexadecimal number. The font of all these characters would be the same, as would the font size. The Latin character A-F will always be in upper case. However, each character may be rotated (the degree of rotation will always be either $0 \circ$, ± 10 , ± 20 , ± 30, degree and each character may be rendered with a different color. The background color of each image can also change. However, all background colors are light in shade. Each image also has some stray lines in the background which are of varying thickness, varying color, and a shade darker than that of the background. These stray lines are intended to make the CAPTCHAs are more "interesting" and realistic. Hexadecimal numbers are written in base 16 i.e. there are 16 "digits" instead of the usual 10. These digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The last six digits have values going from 10 to 15 i.e. A = 10, B = 11, C = 12 etc. Thus, the hexadecimal number 10 is equal to the decimal number $1 \times 161 + 0 \times$

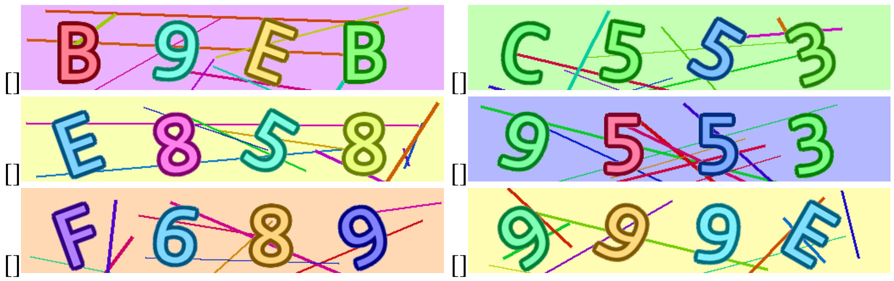


Figure 1: Random Test Set

160 = 16 and the hexadecimal number DECAF is equal to the decimal number D × 164 + E × 163 + C × 162 + A × 161 + F × 160 = 13 × 164 + 14 × 163 + 12 × 162 + 10 × 161 + 15 × 160 = 912559. Your task is to figure out whether the 4 digit hexadecimal number in the image is odd or even.

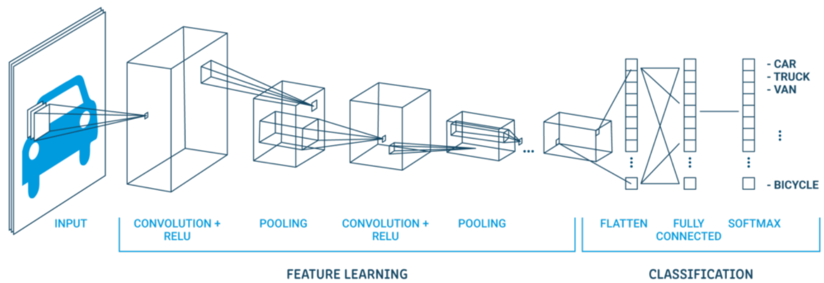
2 Methodology

Convolutional Neural Networks (CNNs) are a type of deep learning model that is widely used in image classification tasks. CNNs are inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. A CNN is typically composed of three types of layers: convolution, pooling, and fully connected layers. The convolution layer is the core building block of a CNN. It applies a set of filters to the input image and produces a set of feature maps. Each filter is a small matrix of weights that is learned during the training process. The filter slides over the input image, performing a dot product at each location, and produces a single output value. The output values are then arranged into a feature map, which highlights the presence of a particular feature in the input image..[ref.] . The pooling layer is used to reduce the spatial size of the feature maps and to make the CNN more robust to small variations in the input image. The most common pooling operation is max pooling, which takes the maximum value in a small region of the feature map and outputs it as the new value for that region. This operation reduces the size of the feature map by a factor of two.[ref.]
The fully connected layer is used to map the high-level features learned by the convolutional layers to the output classes. It takes the flattened output of the last convolutional layer and applies a set of weights to produce a vector of scores for each class. The class with the highest score is then chosen as the predicted output.
In summary, CNNs are a powerful tool for image classification tasks. They are composed of convolutional, pooling, and fully connected layers, which work together to learn spatial hierarchies of features and to map those features to the output classes. The mathematical operations involved in CNNs can be complex, but the intuition behind them is straightforward.

2.1 Image Transformation in CNN

CNN is a powerful algorithm for image processing. These algorithms are currently the best algorithms we have for the automated processing of images. Images contain data of RGB combination. Matplotlib can be used to import an image into memory from a file. The computer doesn't see an image, all it sees is an array of numbers. Color images are stored in 3-dimensional arrays. The first two dimensions correspond to the height and width of the image (the number of pixels). The last dimension corresponds to each pixel's red, green, and blue colors. Convolutional Neural Networks specialized for applications in image video recognition. CNN is mainly used in image analysis tasks like Image recognition, Object detection Segmentation.

There are three types of layers in Convolutional Neural Networks: 1) Convolutional Layer: In a typical neural network each input neuron is connected to the next hidden layer. In CNN, only a small region of the input layer neurons connect to the neuron hidden layer. 2) Pooling Layer: The pooling layer is used to reduce the dimensionality of the feature map. There will be multiple activation pooling layers inside the hidden layer of the CNN. 3) Fully-Connected layer: Fully Connected Layers form the last few layers in the network. The input to the fully connected layer is the output from the final Pooling or Convolutional Layer, which is flattened and then fed into the fully connected layer.



[]

Figure 2: Example, how CNN works [ref.]

One must have come across Resnets while working with CNNs, or at least would have heard of it and we do know that ResNets perform really well on most of Computer vision tasks, but why was there even a need for an architecture like this when we already had other good performing architectures, to answer this, let us understand the drawbacks with other deep neural network architectures that were used before ResNets.

We think that the deeper the neural network, the better the performance, but when researchers experimented with deeper neural nets, it was found that adding more layers to a deep network does not always add up to its performance but rather decreases it, which was due to vanishing gradients in very deep neural networks. As a result, it was proposed that adding more layers to a deep neural network should either increase its performance or let it stay the same, but it should never decrease the performance. In order to achieve this, they came up with the concept of Skip connections/Residual connections, by use of which we can avoid loss of information flow. Let us understand what Skip connections are.

3 ResNet-18

ResNet-18 is a popular deep-learning model architecture that belongs to the family of Residual Neural Networks (ResNets). It was introduced by Kaiming He et al. in their paper "Deep Residual Learning for Image Recognition" in 2016. ResNet-18 is specifically designed for image classification tasks and has achieved excellent performance on various benchmark datasets. The key idea behind ResNet-18 is the introduction of residual blocks, which enable the network to learn residual functions instead of trying to directly fit the desired underlying mapping. This helps to alleviate the degradation problem that occurs when deeper networks suffer from increased training errors as they grow deeper.‘ Now, let’s dive deeper into the architecture and working of ResNet-18:

3.1 Basic Building Block

The fundamental building block of ResNet-18 is the residual block. Each residual block consists of two convolutional layers followed by batch normalization and a ReLU activation function. The input to the residual block is passed through the first convolutional layer, batch normalization, and activation function, and then through the second convolutional layer and batch normalization. The output of the second convolutional layer is added to the input (identity shortcut) to form the residual connection. The final output of the residual block is obtained by applying the ReLU activation function to the sum of the input and the residual connection. This formulation enables the network to learn the residual mapping, making it easier to optimize deep networks.ref

3.1.1 Activation function

Artificial neural networks are inspired by the biological neurons within the human body which activate under certain circumstances resulting in a related action performed by the body in response. Artificial neural nets consist of various layers of interconnected artificial neurons powered by activation functions that help in switching them ON/OFF. Like traditional machine learning algorithmsref, here too, there are certain values that neural nets learn in the training phase.

Briefly, each neuron receives a multiplied version of inputs and random weights which is then added with static bias value (unique to each neuron layer), this is then passed to an appropriate activation function which decides the final value to be given out of the neuron. There are various activation functions available as per the nature of input values. Once the output is generated from the final neural net layer, the loss function (input vs output)is calculated and backpropagation is performed where the weights are adjusted to make the loss minimum. Finding optimal values of weights is what the overall operation is focusing around.
As mentioned above, activation functions give out the final value given out from a neuron, but what is an activation function and why do we need it?

So, an activation function is basically just a simple function that transforms its inputs into outputs that have a certain range. There are various types of activation functions that perform this task in a different manner, For example, the sigmoid activation function takes input and maps the resulting values in between 0 to 1.

One of the reasons that this function is added to an artificial neural network in order to help the network learn complex patterns in the data. These functions introduce nonlinear real-world properties to artificial neural networks. Basically, in a simple neural network, x is defined as inputs, w weights, and we pass f (x) which is the value passed to the output of the network. This will then be the final output or the input of another layer.

If the activation function is not applied, the output signal becomes a simple linear function. A neural network without activation function will act as a linear regression with limited learning power. But we also want our neural network to learn non-linear states as we give it complex real-world information such as images, video, text, and sound.

3.2 ReLU Activation Function

ReLU stands for rectified linear activation unit and is considered one of the few milestones in the deep learning revolution. It is simple yet really better than its predecessor activation functions such as sigmoid or tanh.ReLU activation function formula, First, let us define a ReLU function.ReLU function is its derivative both are monotonic. The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. Thus it gives an output that has a range from 0 to infinity. Now let us give some inputs to the ReLU activation function and see how it transforms.

```
f(x) = max(0, x)

def ReLU(x):
    if x > 0:
        return x
    else:
        return 0

    input_series = [x for x in range(-19, 19)]
    calculate outputs for our inputs
    output_series = [ReLU(x) for x in input_series]
```

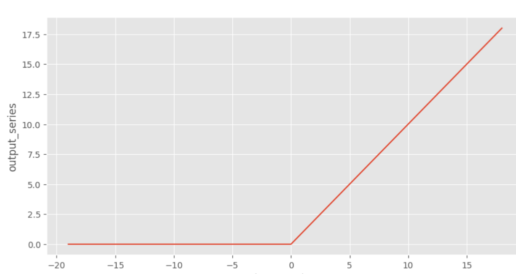


Figure 3: RElu function example graph

ReLU is used as a default activation function and nowadays and it is the most commonly used activation function in neural networks, especially in CNN. Next, we will give input as a list of numbers from -19 to +19. the ReLU function is simple and it consists of no heavy computation as there is no complicated math. The model can, therefore, take less time to train or run. One more important property that we consider the advantage of using the ReLU activation function is sparsity. Usually, a matrix in which most entries are 0 is called a sparse matrix and similarly, we desire a property like this in our neural networks where some of the weights are zero. Sparsity results in concise models that often have better predictive power and less overfitting/noise. In a sparse network, it's more likely that neurons are actually processing meaningful aspects of the problem. For example, in a model detecting human faces in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is not of a face and is a ship or mountain. Since ReLU gives output zero for all negative inputs, it's likely for any given unit to not activate at all which causes the network to be sparse. Now let us see how the ReLu activation function is better than previously famous activation functions such as sigmoid and tanh.

3.3 ResNet-18 Architecture

ResNet-18 consists of several stacked residual blocks, forming the overall architecture of the model. The initial layer is a standard convolutional layer with a large kernel size (7x7) and a stride of 2, followed by batch normalization and a ReLU activation function. This is followed by a max-pooling layer. Then, four stages of residual blocks are stacked together, each with a different number of residual blocks. The number of filters in the convolutional layers gradually increases from stage to stage, while the spatial dimensions are reduced through stridden convolutions. Finally, an adaptive average pooling layer is applied to reduce the spatial dimensions to a fixed size, and a fully connected layer is used for the final classification.

3.4 Shortcut Connections

The identity shortcut connections in ResNet-18 are crucial for addressing the degradation problem. By adding the input to the output of the residual block, the network can learn to make small incremental changes to the identity mapping, rather than trying to learn the entire mapping from scratch. These shortcuts also help propagate gradients more effectively during backpropagation, allowing for better optimization of deeper networks

3.5 Training and Optimization

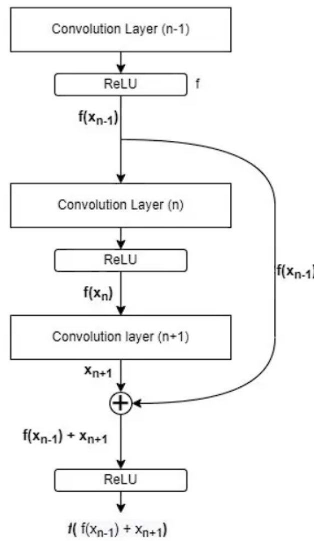
ResNet-18 is typically trained using a variant of stochastic gradient descent (SGD) called mini-batch SGD. The loss function used depends on the specific task, but commonly used loss functions include cross-entropy loss for classification tasks. During training, the model's parameters are updated to minimize the loss between the predicted outputs and the ground truth labels using backpropagation and gradient descent optimization algorithms

3.6 Transfer Learning: ResNet-18

like other deep learning models, can benefit from transfer learning. Transfer learning involves leveraging the pre-trained weights of a model that was trained on a large dataset (such as ImageNet) and fine-tuning it on a smaller, task-specific dataset. By using pre-trained weights as initializations, the model can start with better representations and learn task-specific features more efficiently. In summary, ResNet-18 is a deep convolutional neural network architecture that addresses the degradation problem associated with training deep networks. By introducing residual connections, the model can learn residual mappings and optimize deeper networks more effectively. This architecture has achieved state-of-the-art results in image classification tasks and serves as a foundation for more advanced ResNet variants.

4 skip connections

A Skip/Residual connection takes the activations from an (n-1) convolution layer and adds it to the convolution output of (n+1) layer and then applies ReLU on this sum, thus Skipping the n layer.Fig.3



[]

Figure 4: Skip Connections[ref.]

explains how a skip connection works. (Here I am using f(x) to denote Relu applied on x where x is the output after applying the Convolution operation. But how does this even help, simply put, if the n layer is not learning anything even then we won't lose any information, because at (n+1) we are using the output of the (n-1) layer as well when we move forward and then applying activation on this sum. Thus we are enabling the network to skip one ReLU activation in between if it does not provide any useful information or provides no information at all i.e., 0, and the network will be using the previous information, thus maintaining consistent performance. If anyways both layers are providing significant information, thus having previous information will anyways boost the performance.

5 Loss Function And Optimizations

5.1 Loss Function

Our code uses the class as the loss function. This loss function is suitable for binary classification problems where the model predicts a single scalar value for each input sample. It combines a sigmoid activation function and binary cross-entropy loss. **nn.BCEWithLogitsLoss** is an implementation of the binary cross-entropy loss, which is commonly used when dealing with binary classification tasks. It operates on logits, which are the output of the final layer of the model before applying a sigmoid activation function.

5.2 BCEWithLogitsLoss

This loss combines a Sigmoid layer and the BCELoss in one single class. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.The unreduced (i.e. with reduction set to 'none') loss can be described as where N is the batch size. If a reduction is 'none' (default 'mean'), then

$$\ell(x, y) = L = \begin{bmatrix} l_1 \\ \vdots \\ l_N \end{bmatrix}^\top, \quad l_n = -w_n[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))]$$

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets t[i] should be numbers between 0 and 1. It's possible to trade off recall and precision by adding weights to positive examples. In the case of multi-label classification, the loss can be described as:

$$\ell_c(x, y) = L_c = \begin{bmatrix} l_{1,c} \\ \vdots \\ l_{N,c} \end{bmatrix}^\top, \quad l_{n,c} = -w_{n,c}[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))]$$

The loss function for binary classification with the class number c is defined as:

where C is the class number, N is the number of samples in the batch, and P_c is the weight of the positive answer for class C. $P_c > 1$ increases the recall, while $P_c < 1$ increases the precision.For example, if a dataset contains 100 positive and 300 negative examples of a single class, then Pos_weight for the class should be equal to $\frac{300}{100} = 3$. The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

5.3 Binary Cross Entropy (BCE)

Binary Cross Entropy (BCE) is a common loss function used in binary classification tasks. It measures the dissimilarity between the predicted probability distribution and the true binary labels. In binary classification, we have two classes: positive and negative. The BCE loss is designed to quantify the error or discrepancy between the predicted probabilities and the true binary labels.

In many cases, the output of a model's final layer is transformed using the sigmoid activation function to obtain probabilities that range between 0 and 1. The sigmoid function maps the raw outputs, also known as logits, to the probability of the positive class. However, there are situations where the sigmoid activation is not applied, and the model's output consists of unnormalized scores or logits.

The BCElogit loss is specifically designed to handle this scenario. It applies the BCE loss directly to the logits, without the need for a sigmoid transformation. The purpose of the BCElogit loss is to penalize incorrect predictions and encourage the model to output higher logits for positive examples and lower logits for negative examples. Now, let's break down the formula for the BCElogit loss step by step:

max(logit, 0): This term ensures that the loss is non-negative. It is used to handle cases where the predicted logit is negative. If the predicted logit is greater than or equal to zero, the max(logit, 0) term evaluates to logit itself. However, if the predicted logit is negative, the term becomes zero. This component ensures that incorrect predictions are penalized.

logit * label: This term is subtracted from the previous component. It represents the element-wise multiplication between the predicted logit and the true binary label. When the true label is 1, this term encourages higher logits for positive examples. Conversely, when the true label is 0, the term has no effect on the loss since multiplying by 0 results in 0.

log(1 + exp(-abs(logit))): This term is added to the previous components. It helps to smooth the loss function and stabilize the optimization process. The term involves taking the absolute value of the predicted logit, negating it, and passing it through the exponential function. The logarithm is applied to ensure numerical stability. This component contributes to the loss by increasing its value when the predicted logit deviates significantly from zero. By combining these components, the BCElogit loss encourages the model to output higher logits for positive examples and lower logits for negative examples. It penalizes incorrect predictions and helps train the model to make better binary classification decisions.

It's worth mentioning that different frameworks and libraries might implement the BCElogit loss with slight variations for numerical stability and efficiency. The formula provided here represents the core idea of the BCElogit loss, but specific implementations may differ.

$$\text{BCElogit loss} = \max(\text{logit}, 0) - \text{logit} \cdot \text{label} + \log(1 + \exp(-|\text{logit}|)) \tag{1}$$

5.4 Adam optimizer

5.5 Adam optimizer with BCElogit loss

The Adam optimizer is a popular optimization algorithm used for training neural networks. It combines the benefits of AdaGrad and RMSprop. The BCElogit loss is the binary cross-entropy loss applied directly to the logits, without a sigmoid transformation.

5.6 Mathematical Formulation

1. Initialize the Adam optimizer variables:

```
    m = 0  (initialization of first-moment estimate)
    v = 0  (initialization of second-moment estimate)
beta1 = 0.9  (exponential decay rate for the first-moment estimate)
beta2 = 0.999  (exponential decay rate for the second moment estimate)
epsilon = 1e-8  (small constant to prevent division by zero)
```

2. Compute the gradient of the BCElogit loss with respect to the model's parameters:

```
gradient = compute_gradient(loss, parameters)
```

3. Update the first-moment estimate:

```
m = beta1 * m + (1 - beta1) * gradient
```

4. Update the second-moment estimate:

```
v = beta2 * v + (1 - beta2) * (gradient ** 2)
```

5. Compute the bias-corrected first-moment estimate:

```
m_hat = m / (1 - beta1 ** t)  (t is the current iteration or time step)
```

6. Compute the bias-corrected second-moment estimate:

```
v_hat = v / (1 - beta2 ** t)
```

7. Update the model's parameters using the Adam update rule:

```
parameters = parameters - (learning_rate * m_hat) / (sqrt(v_hat) + epsilon)
```

5.7 Background Code Implementation

Here's an example of how you can implement the Adam optimizer with the BCElogit loss in Python using PyTorch:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define your model
model = YourModel()

# Define the BCElogit loss
loss_function = nn.BCEWithLogitsLoss()

# Define the Adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-8)
```

```
# Training loop
for epoch in range(num_epochs):
    # Clear gradients
    optimizer.zero_grad()

    # Forward pass
    logits = model(inputs)
    loss = loss_function(logits, labels)

    # Backward pass
    loss.backward()

    # Update model parameters
    optimizer.step()
```

In this code snippet, replace `YourModel` with the actual model class or instance you are using. Make sure to import the necessary modules, including `torch`, `torch.nn`, and `torch.optim`.

Next, define the BCElogit loss function using `nn.BCEWithLogitsLoss()`. This loss function is suitable for binary classification tasks where the model's final output is not passed through a sigmoid activation function.

Then, define the Adam optimizer using `optim.Adam()` and provide the model's parameters along with other required arguments like the learning rate (`lr`), beta values (`betas`), and epsilon (`eps`). Adjust these hyperparameters based on your specific requirements.

Inside the training loop, start by clearing the gradients of the optimizer using `optimizer.zero_grad()` to prevent them from accumulating between iterations.

Proceed with the forward pass through the model to obtain the logits using `model(inputs)`. Pass the logits and the corresponding labels to the BCElogit loss function using `loss_function(logits, labels)` to compute the loss.

Perform the backward pass by calling `loss.backward()` to compute the gradients of the loss with respect to the model's parameters.

Finally, update the model's parameters using the `optimizer.step()` method, which applies the Adam optimization algorithm to update the model's parameters based on the computed gradients and the selected learning rate.

You can customize this code snippet by replacing `inputs` and `labels` with your own input data and labels, respectively. Additionally, modify the number of epochs (`num_epochs`) to control the duration of training.

Remember to adapt the code to your specific use case and framework, ensuring that the necessary packages are imported and the model, loss function, and optimizer are appropriately defined.

5.8 Our Model's Code

```
from torch import nn


class Block(nn.Module):

    def __init__(self, in_channels, out_channels, identity_downsample=None, stride=1):
        super(Block, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
```

```
        self.identity_downsample = identity_downsample

    def forward(self, x):
        identity = x
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)
        x += identity
        x = self.relu(x)
        return x


class ResNet_18(nn.Module):

    def __init__(self, image_channels, num_classes):

        super(ResNet_18, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(image_channels,64,kernel_size=7,
        stride=2,padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3,
        stride=2,padding=1)

        #resnet layers
        self.layer1 = self.__make_layer(64, 64, stride=1)
        self.layer2 = self.__make_layer(64, 128, stride=2)
        self.layer3 = self.__make_layer(128, 256, stride=2)
        self.layer4 = self.__make_layer(256, 512, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def __make_layer(self, in_channels, out_channels, stride):

        identity_downsample = None
        if stride != 1:
            identity_downsample = self.identity_downsample(in_channels, out_channels)

        return nn.Sequential(
            Block(in_channels,out_channels,
            identity_downsample=identity_downsample, stride=stride),
            Block(out_channels,out_channels)
        )

    def forward(self, x):

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
```

```
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        return x

    def identity_downsample(self, in_channels, out_channels):

        return nn.Sequential(
            nn.Conv2d(in_channels,out_channels,
            kernel_size=3,stride=2,padding=1),
            nn.BatchNorm2d(out_channels)
        )
```

The below table explains the model summary.

5.9 Model Summary

Table 1: Summary of the Model

Layer (type)	Output Shape	Param #
Conv2d-1	[32, 64, 50, 250]	12,608
BatchNorm2d-2	[32, 64, 50, 250]	128
ReLU-3	[32, 64, 50, 250]	0
MaxPool2d-4	[32, 64, 25, 125]	0
Conv2d-5	[32, 64, 25, 125]	36,928
BatchNorm2d-6	[32, 64, 25, 125]	128
ReLU-7	[32, 64, 25, 125]	0
...	...	...
AdaptiveAvgPool2d-67	[32, 512, 1, 1]	0
Linear-68	[32, 1]	513

Total params: 12,561,217
Trainable params: 12,561,217
Non-trainable params: 0

Input size (MB): 24.41
Forward/backward pass size (MB): 2079.98
Params size (MB): 47.92
Estimated Total Size (MB): 2152.32

6 Results and Conclusion

The training process is initiated by printing "Training" and calling the `train_step()` function. This function likely performs the training step for the model, including forward and backward passes, updating weights using the optimizer, calculating the loss, and computing the accuracy. The testing process is initiated by printing "Testing" and calling the `test_step()` function. This function likely performs the evaluation step for the model, calculating the loss and accuracy on the test dataset. The loop continues for each epoch, printing the progress and updating the model. After the loop ends, the timer is stopped using `time.perf_counter()` and the elapsed time is stored in the `model_4_time` variable. Finally, the execution time of the model is printed.

6.1 Training and Testing Results

The code includes a training loop that runs for a specified number of epochs, performing training and testing at each epoch. The training results, such as loss and accuracy, are printed during each training iteration, while the testing results are printed after each epoch. Upon completion of the first few epochs, the accuracy of the model is found to be around 0.5, but after 7 epochs, it is found to be nearly 1.0.

6.2 Loss and Accuracy

The graph of training and testing loss with respect to the number of epochs can be seen below:

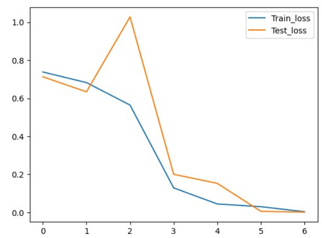


Figure 5: Training and Testing Loss with epochs

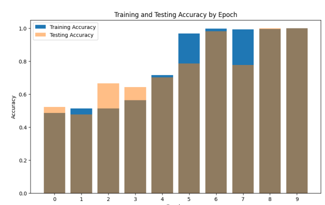


Figure 6: Training and Testing Accuracy with epochs

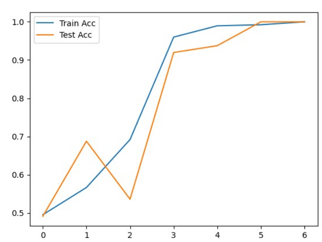


Figure 7: Training and Testing Accuracy with epochs

The graph of training and testing accuracy with respect to the number of epochs can be seen below: The accuracy graph is shown below, from here we see that the model used and the hyperparameter tuning we did are acceptable and we reached o 100 percent accuracy on the secret test data set.

7 References

1.Gentle Dive into Math Behind Convolutional Neural Networks[here]
2. Dive into CNN[here]
3. Residual Networks (ResNet) and ResNeXt[here]
4. Machine Learning Tutorial For Beginners[here]
5. ResNet — Understand and Implement from scratch.here
6.BCEWITHLOGITSLOSS .here. source code
8. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.here
7. Mathematics for Machine Learning .here