Introduction to Deep Learning 2023

Term Project Report

Student ID: 20171257

Name: Seung Hyun Kim

Submission: 2023.12.24

Table of Contents

Ⅰ. Project Objective

The problem of classifying English characters and numbers composed in various fonts will be solved by deep learning algorithms. Additionally, through the project, the following topics will be considered.

- Understanding which deep learning architecture is most efficient for solving this problem.
- Finding the optimal combination of hyperparameters through the Hyperparameter Tuning process.
- Training deep learning models and evaluating their performance.

Ⅱ. Background Knowledge

CNN (Convolutional Neural Network) is a deep learning algorithm that mimics the structure of the human visual system. Here's an explanation of the components and functions of CNN:

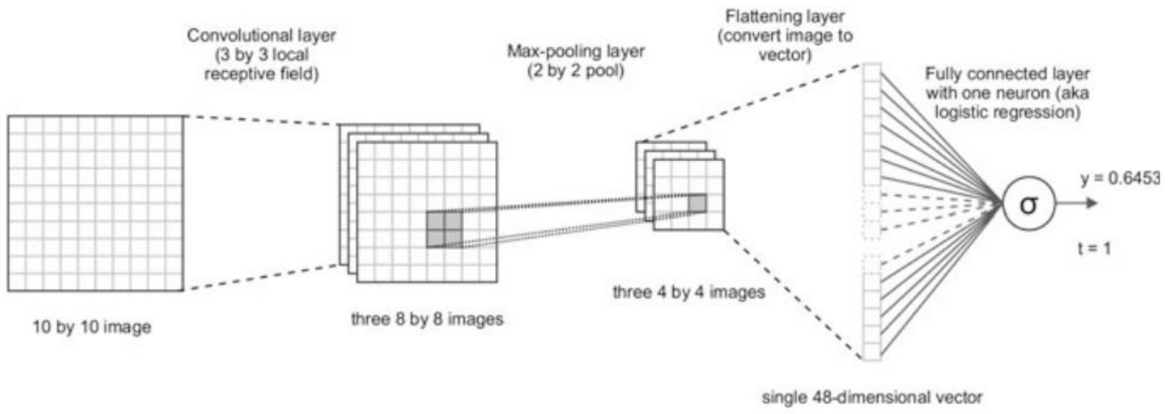
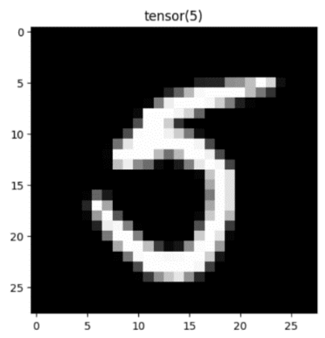

Convolutional layer (3 by 3 local receptive field)
Max-pooling layer (2 by 2 pool)
Flattening layer (convert image to vector)
Fully connected layer with one neuron (aka logistic regression)
σ
y = 0.6453
t = 1
10 by 10 image
three 8 by 8 images
three 4 by 4 images
single 48-dimensional vector

Fig. 6.3 A convolutional neural network with a convolutional layer, a max-pooling layer, a flattening layer and a fully connected layer with one neuron

The structure of CNN can be divided into two parts: one for extracting features from images (Feature Extraction) and the other for classifying them (Classification).



Let's assume that a handwritten digit 5's image is used as the input data for a CNN model. Initially, this image data can be represented as a 3D tensor with dimensions Height x Width x Channel. In this context, the number of Channels is 1 for grayscale images and 3 for RGB color images.

The image data first passes through a Convolutional Layer. In a single Convolutional Layer, there are as many filters as the number of input image Channels. Each channel has its filter, and Convolution operations are applied to create the Output Image. Assuming that the input image size is 17 x 8 x 1, the Kernel size is 3 x 3 x 1, and the Stride is 1, the resulting Output Image size after Convolution would be 15 x 6 x 1, as illustrated in the figure below.

It's worth noting that the figure omitted Bias, but in reality, CNNs are often designed to add the same Bias value to each pixel in the Output Image after the Convolution operation.

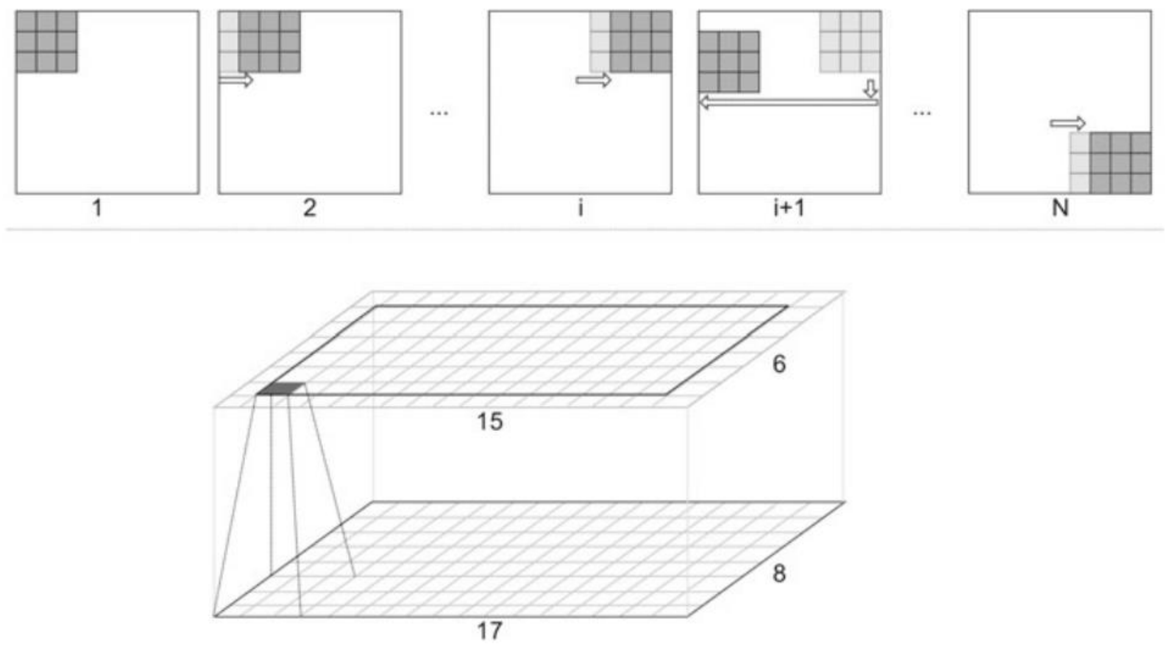
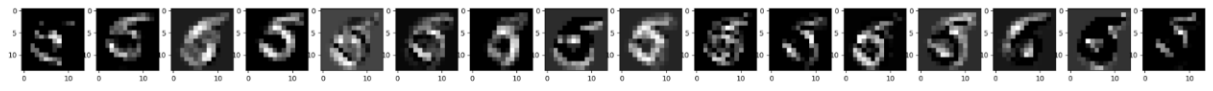


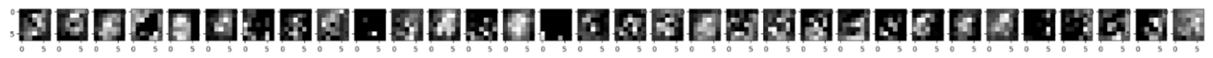
Fig. 6.2 2D Convolutional layer

Additionally, there is a technique called Padding to prevent the reduction in the size of the image as it goes through the Convolution Layer. This technique involves adding pixels with specific values to the edges of the image. In the example above, if Padding is applied with a size of 1, pixels are added in all directions (top, bottom, left, and right), so the Output

Image size would not be 15 x 6 x 1 but instead would remain the same as the input image size, which is 17 x 8 x 1.

The Image that has passed through the Convolutional Layer undergoes the Pooling Layer, which can be described as a layer that reduces the Image's size while emphasizing specific features. There are different Pooling methods, including Max Pooling, Average Pooling, and Min Pooling, but let's explain Max Pooling, which is commonly used in CNNs.
Max Pooling involves using a Kernel to slide over the Image and selecting the maximum value for each region it covers. Typically, the most commonly used Pooling Kernel Size is 2 by 2, with a Stride of 2. In this case, both the Height and Width of the Output Image become half the values of the input Image. This process helps reduce the spatial dimensions of the feature maps while retaining essential information by selecting the most prominent features in each region.





The two images above represent the appearance of the number 5 image after passing through the first and second Convolutional Layer & Pooling Layer, respectively. It can be observed that as the Image progresses through the Convolutional Layer, features become more emphasized, and as it goes through the Pooling Layer, the size decreases.
Up to this point, the Feature Extraction part has been explained. The Image that has passed through this part enters the Classification part. This section consists of multiple Fully Connected Layers, and the data is flattened and no longer exists in the form of an Image. The final output feature count in the Fully Connected Layer should be the same as the number of classes in the Image, and this layer uses an Activation Function such as Softmax to derive probability distributions for each class and selects the class with the highest probability as the final prediction result. (In the example, if the prediction is correct, the predicted result should be 5.)
To improve accuracy, CNNs also employ techniques like Batch Normalization and Dropout. Batch Normalization is a technique that normalizes the inputs of each layer, while Dropout

randomly deactivates some neurons during training. These techniques help prevent overfitting and enhance the model's performance.
CNNs excel at finding patterns in images, making them widely used in applications such as autonomous vehicles, facial recognition, and computer vision.

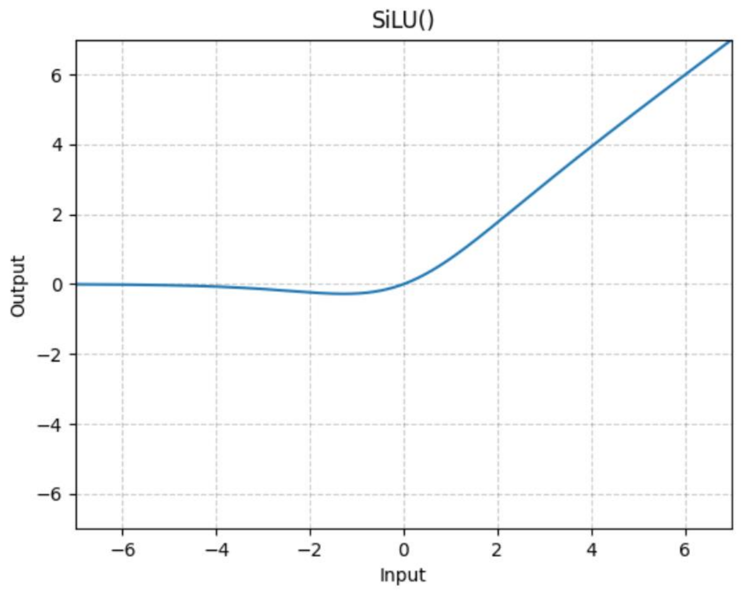
7.3 Adding Feedback Loops and Unfolding a Neural Network

Let us now see how recurrent neural networks work. Remember the vanishing gradient problem? There we have seen that adding layers one after the other would severely cripple the ability to learn weights by gradient descent, since the movements would be really small, sometimes even rounded to zero. Convolutional neural networks solved this problem by using a shared set of weights, so learning even little by little is not a problem since each time the same weights get updated. The only problem is that convolutional neural networks have a very specific architecture making them best suited for images and other limited sequences.

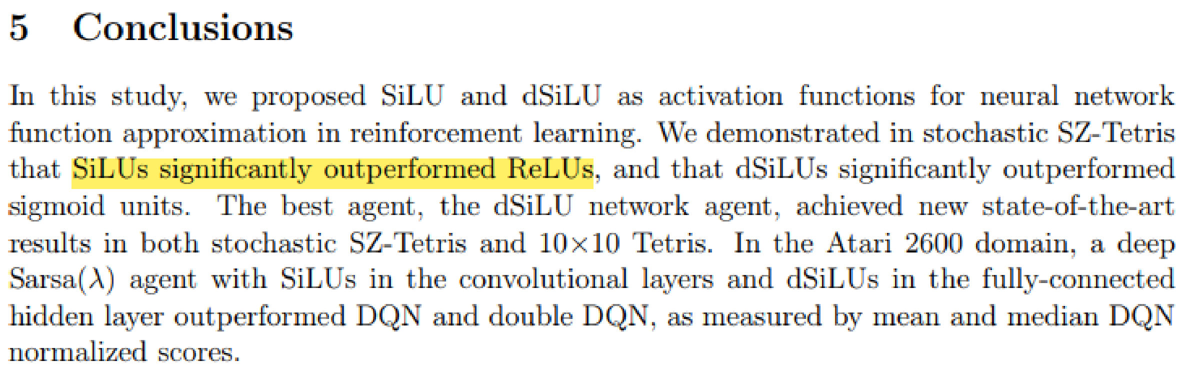
While there are various deep learning models like RNN (including LSTM and GRU), CRNN, and others, it is generally accepted that CNN excels in image processing tasks. Therefore, for this project, CNN has been chosen.

The description of the Activation Function used in the project is as follows.

1. SiLU(Sigmoid Linear Unit)



SiLU is a function of the form $SiLU(x) = x * \sigma(x)$, which means it combines a linear term with a sigmoid term. The graph of SiLU exhibits characteristics where the function's value is not zero in the negative range, making it a function that possesses both non-linearity and linearity.

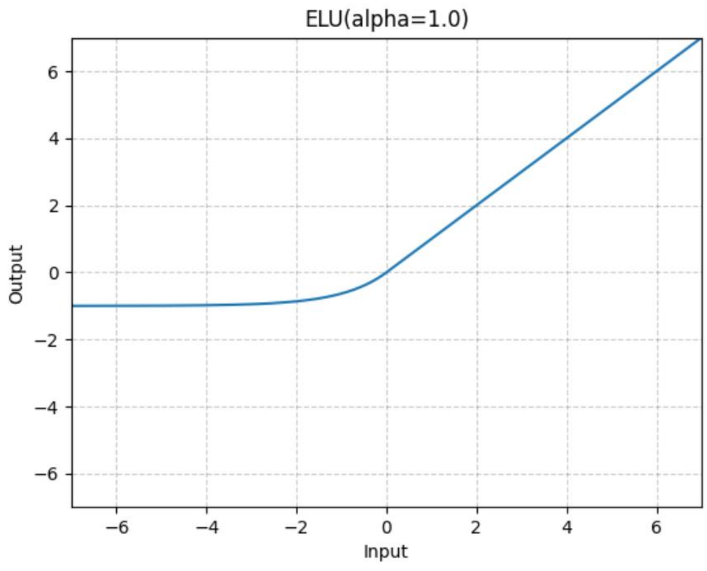


5 Conclusions

In this study, we proposed SiLU and dSiLU as activation functions for neural network function approximation in reinforcement learning. We demonstrated in stochastic SZ-Tetris that SiLUs significantly outperformed ReLUs, and that dSiLUs significantly outperformed sigmoid units. The best agent, the dSiLU network agent, achieved new state-of-the-art results in both stochastic SZ-Tetris and 10×10 Tetris. In the Atari 2600 domain, a deep Sarsa(λ) agent with SiLUs in the convolutional layers and dSiLUs in the fully-connected hidden layer outperformed DQN and double DQN, as measured by mean and median DQN normalized scores.

According to the referenced paper, it is reported that SiLU significantly outperforms ReLU.

2. ELU(Exponential Linear Unit)



$$\mathrm{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\exp(x) - 1), & \text{if } x \leq 0 \end{cases}$$

ELU, or Exponential Linear Unit, is a type of activation function that transforms ReLU in the negative range into an exponential form with a constant α (commonly set to 1.0). In other words, it ensures that the function's value is not zero in the negative range, addressing the dying ReLU problem and serving as an improvement over the ReLU function.

However, in this project, ELU's performance was slightly lower compared to SiLU, and as a result, it was discarded and replaced with the SiLU function.

Ⅲ. How the Project was conducted

Before starting the project, the following code was used to analyze the characteristics of the dataset.

`print(len(train_data))`
`print(len(valid_data))`
 	
`41600`
`15600`
	`# check dataloader`
	`image,label = next(iter(valid_loader))`
	`print(image.shape)`
	`print(label.shape)`
	
	`torch.Size([50, 1, 100, 100])`
	`torch.Size([50])`

The number of data samples in the training set and test set	Images are black and white with 100 by 100 pixels in size

```
# visualize data
# image_show function : num 수 만큼 dataset 내의 data를 보여주는 함수
def image_show(dataset, num):
  fig = plt.figure(figsize=(10,10))

  for i in range(num):
    plt.subplot(1, num, i+1)
    plt.imshow(dataset[i+12200][0].squeeze(), cmap="gray")
    plt.title(dataset[i+12200][1].item())  # .item()을 사용하여 텐서에서 숫자로 변환

image_show(train_data, 8)
```

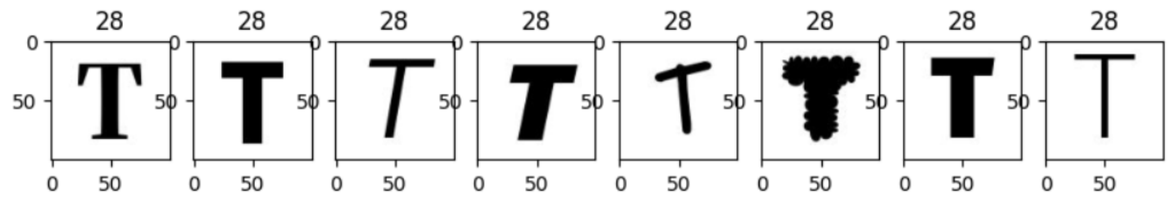
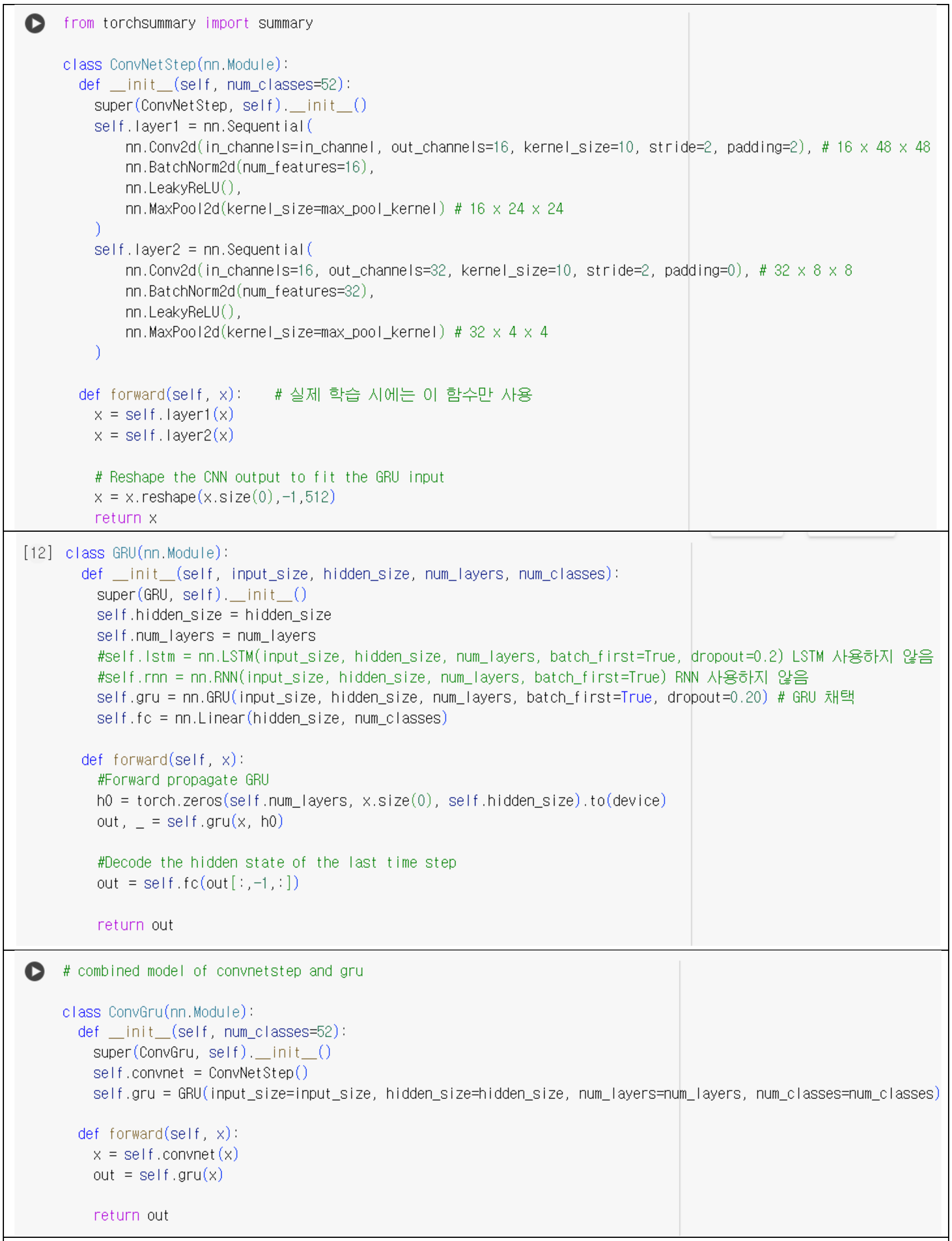


Image: The images of the alphabet "T" composed in various fonts.

The initial model was implemented in a CRNN format, which is a combination of CNN and RNN (GRU). However, there were not any significant improvements over CNN in terms of error and accuracy, as will be mentioned later. Therefore, the CRNN model was discarded. (Additionally, it had the drawback of longer training times and larger model file sizes.)



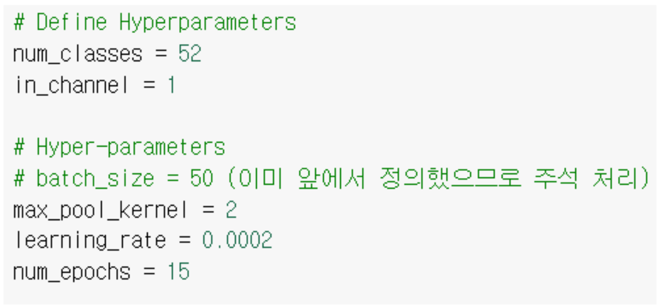
```
from torchsummary import summary

class ConvNetStep(nn.Module):
  def __init__(self, num_classes=52):
    super(ConvNetStep, self).__init__()
    self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=in_channel, out_channels=16, kernel_size=10, stride=2, padding=2), # 16 x 48 x 48
        nn.BatchNorm2d(num_features=16),
        nn.LeakyReLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) # 16 x 24 x 24
    )
    self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=16, out_channels=32, kernel_size=10, stride=2, padding=0), # 32 x 8 x 8
        nn.BatchNorm2d(num_features=32),
        nn.LeakyReLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) # 32 x 4 x 4
    )

  def forward(self, x):    # 실제 학습 시에는 이 함수만 사용
    x = self.layer1(x)
    x = self.layer2(x)

    # Reshape the CNN output to fit the GRU input
    x = x.reshape(x.size(0),-1,512)
    return x
```

```
[12] class GRU(nn.Module):
       def __init__(self, input_size, hidden_size, num_layers, num_classes):
         super(GRU, self).__init__()
         self.hidden_size = hidden_size
         self.num_layers = num_layers
         #self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, dropout=0.2) LSTM 사용하지 않음
         #self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True) RNN 사용하지 않음
         self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True, dropout=0.20) # GRU 채택
         self.fc = nn.Linear(hidden_size, num_classes)

       def forward(self, x):
         #Forward propagate GRU
         h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
         out, _ = self.gru(x, h0)

         #Decode the hidden state of the last time step
         out = self.fc(out[:,-1,:])

         return out
```

```
# combined model of convnetstep and gru

class ConvGru(nn.Module):
  def __init__(self, num_classes=52):
    super(ConvGru, self).__init__()
    self.convnet = ConvNetStep()
    self.gru = GRU(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, num_classes=num_classes)

  def forward(self, x):
    x = self.convnet(x)
    out = self.gru(x)

    return out
```

Initial Model: CRNN

The final results after replacing the model with CNN and conducting Hyperparameter Tuning are listed below in order.



```
# Define Hyperparameters
num_classes = 52
in_channel = 1

# Hyper-parameters
# batch_size = 50 (이미 앞에서 정의했으므로 주석 처리)
max_pool_kernel = 2
learning_rate = 0.0002
num_epochs = 15
```

Image: Final Selected Hyperparameter Values

The value of Epoch was set to a maximum limit of 15 to comply with the time constraint. Since the code was designed to save the model only when the Validation Error decreased, it was best to set it to the largest feasible value.
The learning rate was initially set to 0.001 but was adjusted to 0.0002 as shown in the image below due to the occurrence of overshooting.

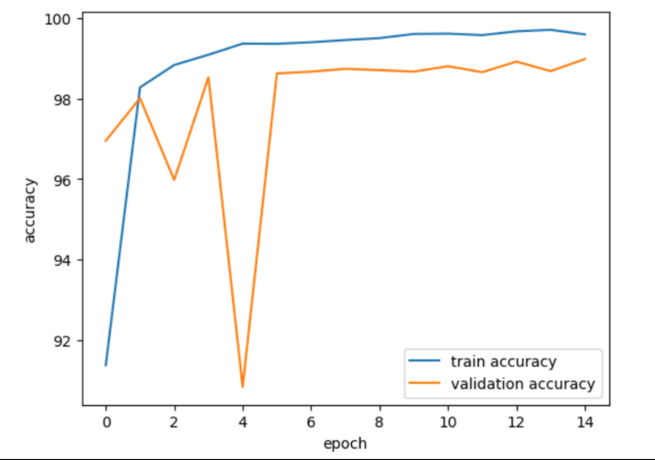
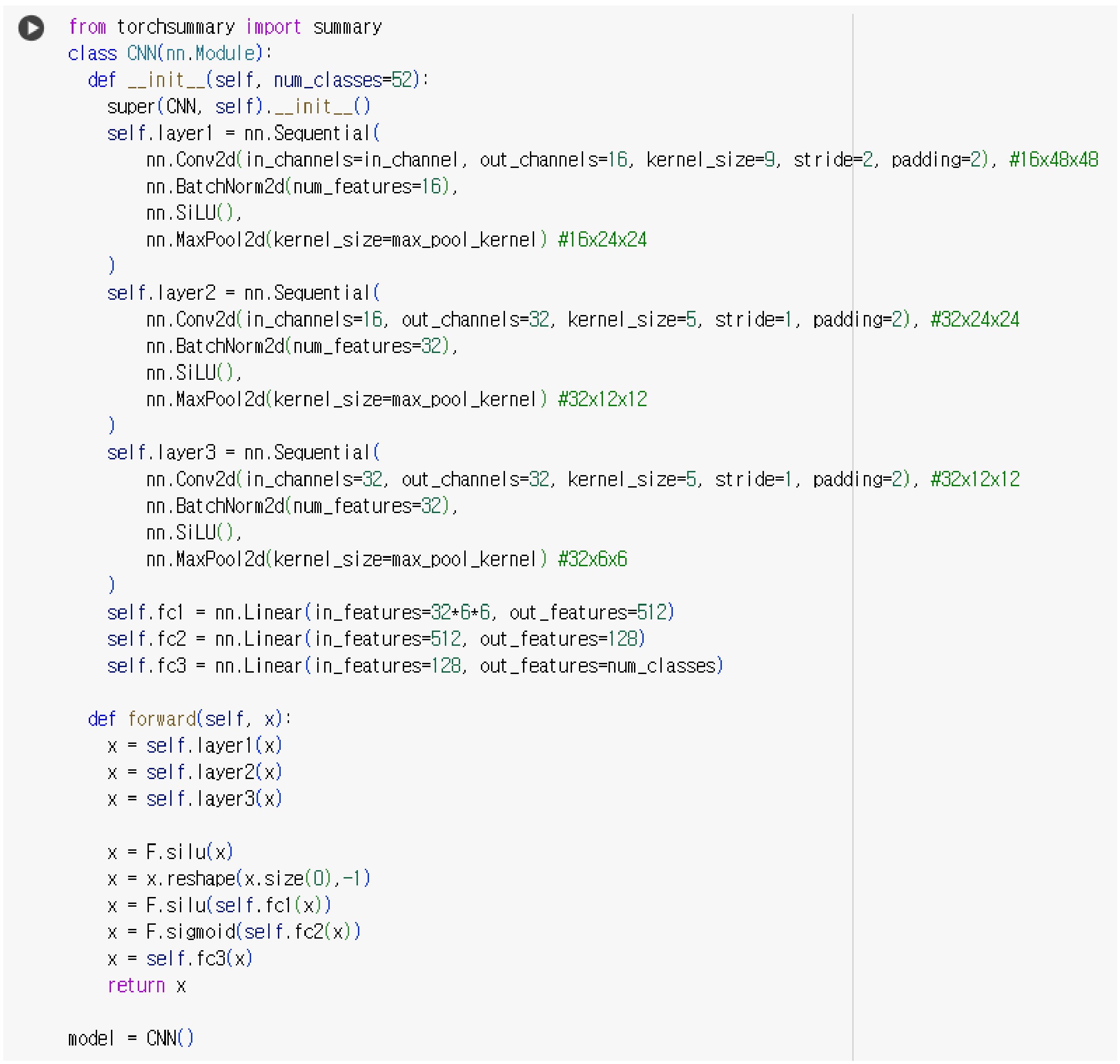


Image: Accuracy graph when the learning rate was set to 0.001. The output is very unstable.



```python
from torchsummary import summary
class CNN(nn.Module):
  def __init__(self, num_classes=52):
    super(CNN, self).__init__()
    self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=in_channel, out_channels=16, kernel_size=9, stride=2, padding=2), #16x48x48
        nn.BatchNorm2d(num_features=16),
        nn.SiLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) #16x24x24
    )
    self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=2), #32x24x24
        nn.BatchNorm2d(num_features=32),
        nn.SiLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) #32x12x12
    )
    self.layer3 = nn.Sequential(
        nn.Conv2d(in_channels=32, out_channels=32, kernel_size=5, stride=1, padding=2), #32x12x12
        nn.BatchNorm2d(num_features=32),
        nn.SiLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) #32x6x6
    )
    self.fc1 = nn.Linear(in_features=32*6*6, out_features=512)
    self.fc2 = nn.Linear(in_features=512, out_features=128)
    self.fc3 = nn.Linear(in_features=128, out_features=num_classes)

  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)

    x = F.silu(x)
    x = x.reshape(x.size(0),-1)
    x = F.silu(self.fc1(x))
    x = F.sigmoid(self.fc2(x))
    x = self.fc3(x)
    return x

model = CNN()
```

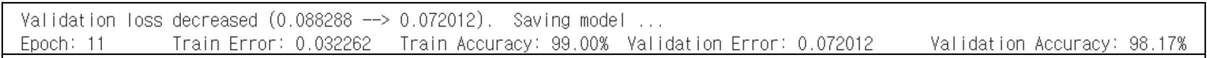
Image: The final CNN model. All ReLU functions were replaced with SiLU functions from the original skeleton code, and sigmoid was chosen as the activation function for the second FC Layer. The output shape for each layer can be calculated as follows.

Layer	Computation	Channel x Width x Height
Input Layer	-	1 x 100 x 100
1st Convolutional Layer	((100+2 x 2)-(10-2))/2=48	16 x 48 x 48
1st Pooling Layer	48/2=24	16 x 24 x 24
2nd Convolutional Layer	(24+2 x 2)-(5-1)=24	32 x 24 x 24
2nd Pooling Layer	24/2=12	32 x 12 x 12
3rd Convolutional Layer	(12+2 x 2)-(5-1)=12	32 x 12 x 12
3rd Pooling Layer	12/2=6	32 x 6 x 6

Layer	Size
1st Flattening Layer	32 x 6 x 6 -〉 1,152
1st FC Layer	1,152-〉512
2nd FC Layer	512-〉128
3rd FC Layer	128-〉52

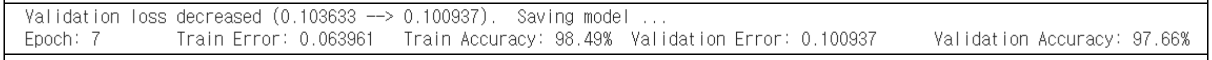
As will be discussed in the later results and discussion section, the number of parameters is 590,336 in the first FC Layer. This can be calculated as (32 x 6 x 6 + 1) x 512 = 590,336. As the reader can see from this equation, if either the Output Size of the Feature Extraction part (32 x 6 x 6) or the Output Size of the first FC Layer (512) doubles, the number of parameters in that layer will also increase proportionally, resulting in approximately 1.2 million parameters for that layer. Consequently, the total number of parameters would exceed the upper limit of 1,200,000. Therefore, 32 x 6 x 6 and 512 are the Output Size values set to satisfy the upper limit, and they cannot have any larger values. (To be more precise, the limit is up to (32 x 7 x 7, 512); beyond that is not allowed. Typically, the channel count and FC output size are adjusted in increments of two.)

〈Error & Accuracy Timeline 〉



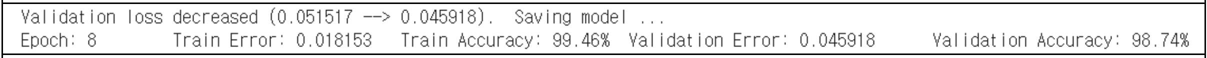
```
Validation loss decreased (0.088288 --> 0.072012).  Saving model ...
Epoch: 11       Train Error: 0.032262   Train Accuracy: 99.00%  Validation Error: 0.072012      Validation Accuracy: 98.17%
```

Image: The initial model was CRNN.



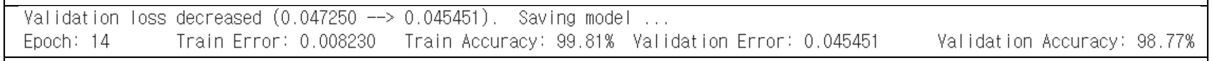
```
Validation loss decreased (0.103633 --> 0.100937).  Saving model ...
Epoch: 7        Train Error: 0.063961   Train Accuracy: 98.49%  Validation Error: 0.100937      Validation Accuracy: 97.66%
```

Image: The model was changed to CNN.



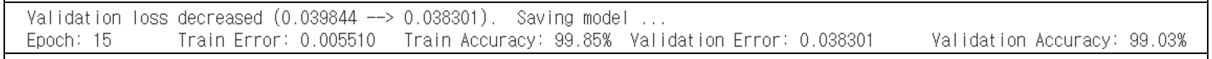
```
Validation loss decreased (0.051517 --> 0.045918).  Saving model ...
Epoch: 8        Train Error: 0.018153   Train Accuracy: 99.46%  Validation Error: 0.045918      Validation Accuracy: 98.74%
```

Image: ReLU was replaced with ELU and SiLU. The activation function of the 2nd FC Layer was changed to Sigmoid.



```
Validation loss decreased (0.047250 --> 0.045451).  Saving model ...
Epoch: 14       Train Error: 0.008230   Train Accuracy: 99.81%  Validation Error: 0.045451      Validation Accuracy: 98.77%
```

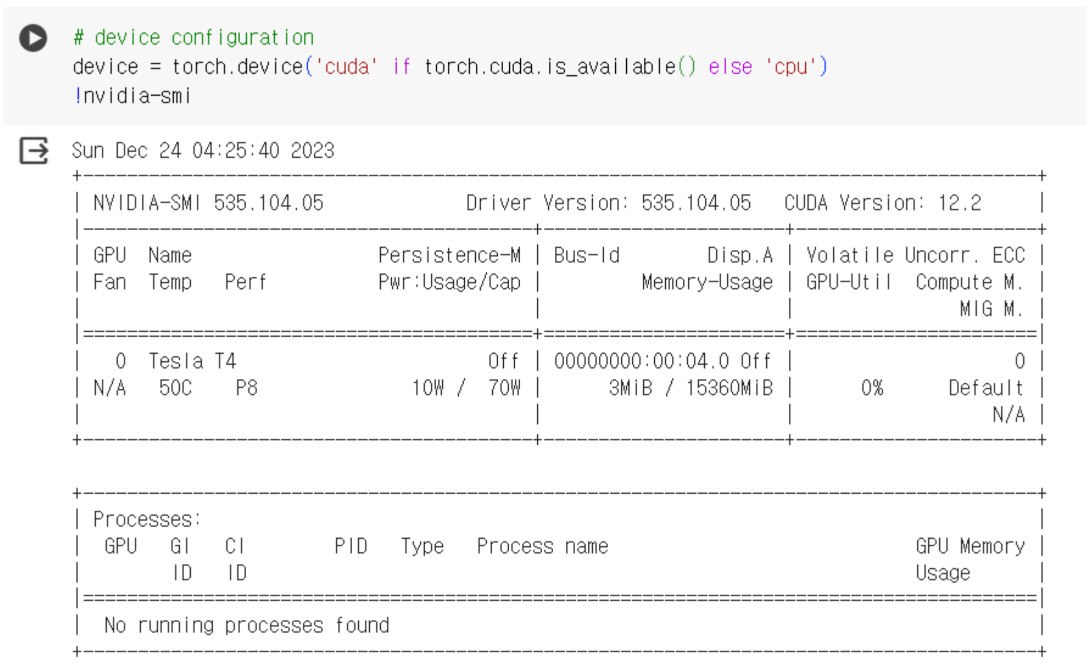
Image: The learning rate was adjusted to 0.0002.



```
Validation loss decreased (0.039844 --> 0.038301).  Saving model ...
Epoch: 15       Train Error: 0.005510   Train Accuracy: 99.85%  Validation Error: 0.038301      Validation Accuracy: 99.03%
```

Image: After replacing ELU with SiLU, the accuracy is the highest (99.03%)

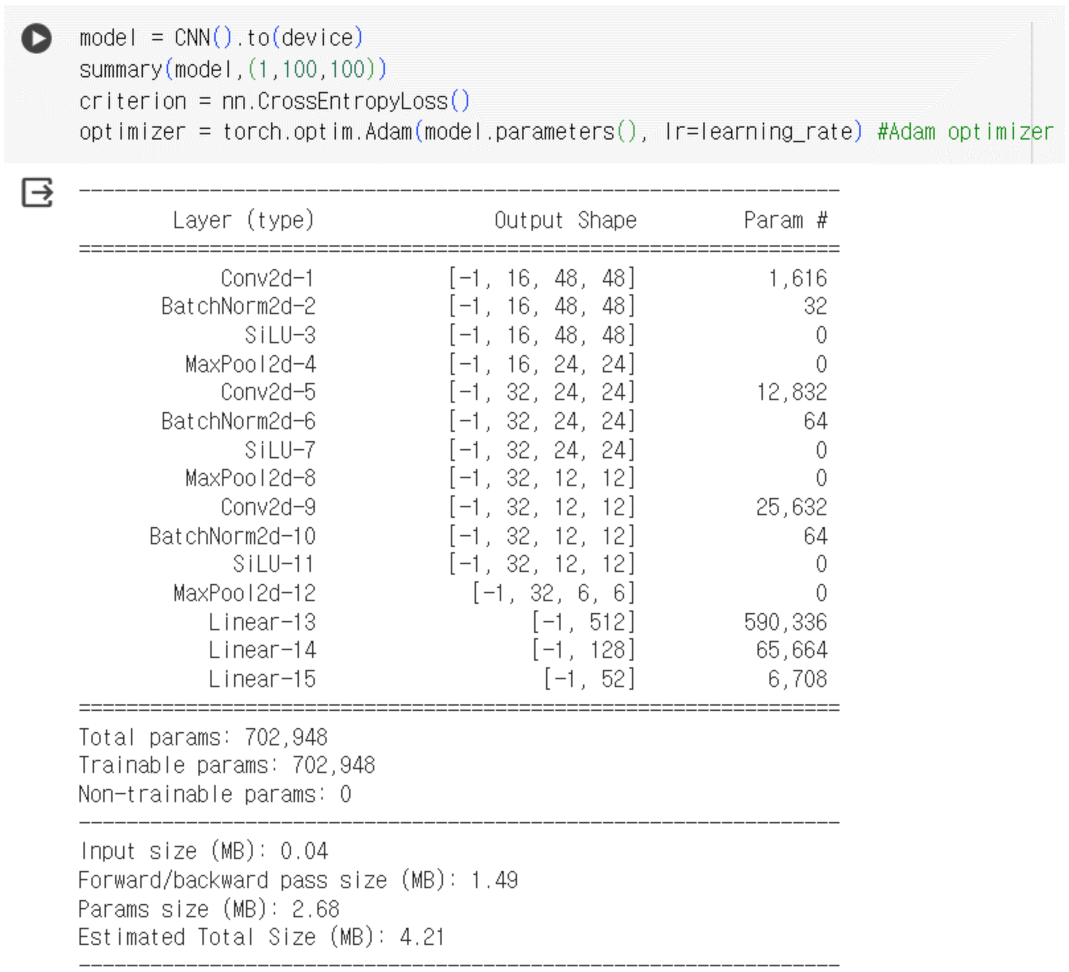
Ⅳ. Conclusion and Discussion



```
# device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
!nvidia-smi
```

```
Sun Dec 24 04:25:40 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   50C    P8              10W /  70W |      3MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Image: !nvidia-smi cell and output result. A free version of the T4 GPU was used.



```
model = CNN().to(device)
summary(model,(1,100,100))
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) #Adam optimizer
```

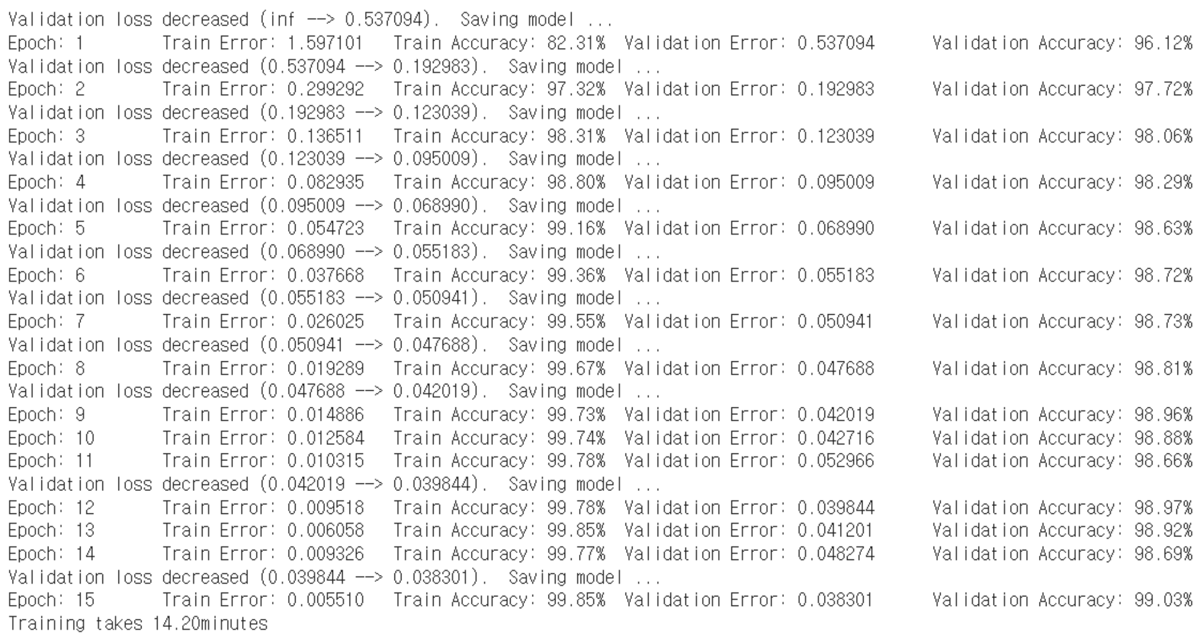
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 48, 48]           1,616
       BatchNorm2d-2           [-1, 16, 48, 48]              32
              SiLU-3           [-1, 16, 48, 48]               0
         MaxPool2d-4           [-1, 16, 24, 24]               0
            Conv2d-5           [-1, 32, 24, 24]          12,832
       BatchNorm2d-6           [-1, 32, 24, 24]              64
              SiLU-7           [-1, 32, 24, 24]               0
         MaxPool2d-8           [-1, 32, 12, 12]               0
            Conv2d-9           [-1, 32, 12, 12]          25,632
      BatchNorm2d-10           [-1, 32, 12, 12]              64
             SiLU-11           [-1, 32, 12, 12]               0
        MaxPool2d-12             [-1, 32, 6, 6]               0
           Linear-13                  [-1, 512]         590,336
           Linear-14                  [-1, 128]          65,664
           Linear-15                   [-1, 52]           6,708
================================================================
Total params: 702,948
Trainable params: 702,948
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 1.49
Params size (MB): 2.68
Estimated Total Size (MB): 4.21
----------------------------------------------------------------
```

Image: Final Model Summary. The total number of model parameters is 702,948, which complies with the upper limit of 1,200,000. It can be observed that the number of parameters in the first FC Layer dominates the total model parameters.

Note that the number of parameters in the Convolutional Layer can be calculated as follows.

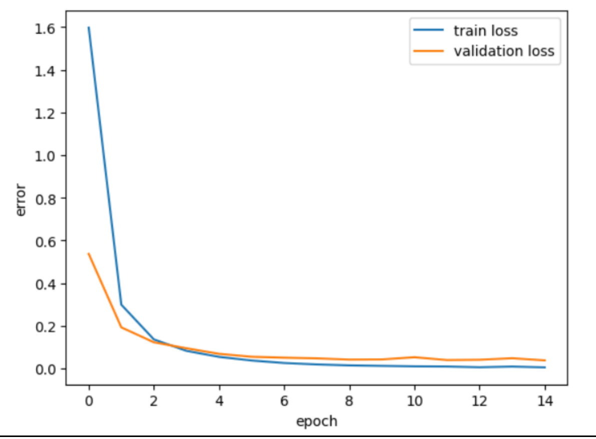
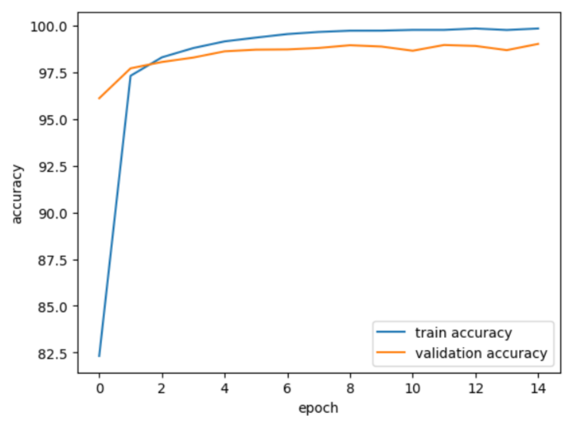
Number of Parameters=(Filter Height×Filter Width×Input Channels+1)×Output Channels

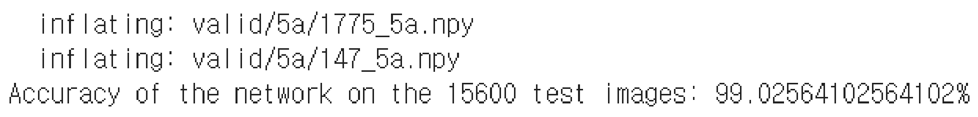
Ex) In the second Convolutional Layer, # of params=(5 x 5 x 16 + 1) x 32 = 12,832



```
Validation loss decreased (inf --> 0.537094).  Saving model ...
Epoch: 1        Train Error: 1.597101   Train Accuracy: 82.31%  Validation Error: 0.537094      Validation Accuracy: 96.12%
Validation loss decreased (0.537094 --> 0.192983).  Saving model ...
Epoch: 2        Train Error: 0.299292   Train Accuracy: 97.32%  Validation Error: 0.192983      Validation Accuracy: 97.72%
Validation loss decreased (0.192983 --> 0.123039).  Saving model ...
Epoch: 3        Train Error: 0.136511   Train Accuracy: 98.31%  Validation Error: 0.123039      Validation Accuracy: 98.06%
Validation loss decreased (0.123039 --> 0.095009).  Saving model ...
Epoch: 4        Train Error: 0.082935   Train Accuracy: 98.80%  Validation Error: 0.095009      Validation Accuracy: 98.29%
Validation loss decreased (0.095009 --> 0.068990).  Saving model ...
Epoch: 5        Train Error: 0.054723   Train Accuracy: 99.16%  Validation Error: 0.068990      Validation Accuracy: 98.63%
Validation loss decreased (0.068990 --> 0.055183).  Saving model ...
Epoch: 6        Train Error: 0.037668   Train Accuracy: 99.36%  Validation Error: 0.055183      Validation Accuracy: 98.72%
Validation loss decreased (0.055183 --> 0.050941).  Saving model ...
Epoch: 7        Train Error: 0.026025   Train Accuracy: 99.55%  Validation Error: 0.050941      Validation Accuracy: 98.73%
Validation loss decreased (0.050941 --> 0.047688).  Saving model ...
Epoch: 8        Train Error: 0.019289   Train Accuracy: 99.67%  Validation Error: 0.047688      Validation Accuracy: 98.81%
Validation loss decreased (0.047688 --> 0.042019).  Saving model ...
Epoch: 9        Train Error: 0.014886   Train Accuracy: 99.73%  Validation Error: 0.042019      Validation Accuracy: 98.96%
Epoch: 10       Train Error: 0.012584   Train Accuracy: 99.74%  Validation Error: 0.042716      Validation Accuracy: 98.88%
Epoch: 11       Train Error: 0.010315   Train Accuracy: 99.78%  Validation Error: 0.052966      Validation Accuracy: 98.66%
Validation loss decreased (0.042019 --> 0.039844).  Saving model ...
Epoch: 12       Train Error: 0.009518   Train Accuracy: 99.78%  Validation Error: 0.039844      Validation Accuracy: 98.97%
Epoch: 13       Train Error: 0.006058   Train Accuracy: 99.85%  Validation Error: 0.041201      Validation Accuracy: 98.92%
Epoch: 14       Train Error: 0.009326   Train Accuracy: 99.77%  Validation Error: 0.048274      Validation Accuracy: 98.69%
Validation loss decreased (0.039844 --> 0.038301).  Saving model ...
Epoch: 15       Train Error: 0.005510   Train Accuracy: 99.85%  Validation Error: 0.038301      Validation Accuracy: 99.03%
Training takes 14.20minutes
```

Image: Model training results. The training time was 14.20 minutes, complying with the 15-minute limit, and the final error and accuracy on the validation set are 0.038301 and 99.03%, respectively.

	
Error vs Epoch	Accuracy vs Epoch



```
   inflating: valid/5a/1775_5a.npy
   inflating: valid/5a/147_5a.npy
 Accuracy of the network on the 15600 test images: 99.02564102564102%
```

Since the test dataset was not publicly available, the validation set was used instead when creating the test.py file.

There is a general misconception that the CRNN model, which combines CNN and RNN, is superior to the CNN model. However, as the project progressed, it became clear that the simpler CNN model was more suitable for image classification tasks. While the most appropriate model for this project was CNN, each model has its features and pros and cons, so the optimal model may vary depending on the dataset and problem type. Additionally, the tuning process for hyperparameters and activation functions required a lot of trial and error, but the process cannot be overlooked since it is an essential task for drawing the best performance out of the model.

Ⅴ. References

Sandro Skansi, Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence, Springer, 2018.02.06., pp. 128-134, 145-146.

Introduction to Deep Learning Lecture notes, Sogang University Dept of Electronic Engineering, https://github.com/ye0njinkim/EEE4178_2023_Hands_On/blob/main/Day3_2023.ipynb.

SILU, Pytorch.org, SiLU — PyTorch 2.1 documentation.

Stefan Elfwing, Eiji Uchibe, Kenji Doya, Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning, 2 Nov 2017.

ELU, Pytorch.org, ELU — PyTorch 2.1 documentation.