IV. Solution Approach

Current Solution Approach

Our current solution approach is to build a CRNN model and connect its output to a feedforward neural network using CTC Loss and Adam Optimizer to search for the optimum parameters for our problem.

Below we explained each phase in detail and illustrated our model and its layers. Also you can find examples for our dataset used in CS402.

Generating a Line Dataset for Text Recognition with CRNN

1. Image Generation

In the beginning we generated 256 by 1024 images because we did not apply downsizing on the word images. For a while we did some training using these images and their labels. However this many pixels caused our model to have too many neurons in CNN layers. Thus it made harder for us to optimize the CNN Feature Extractor Model.

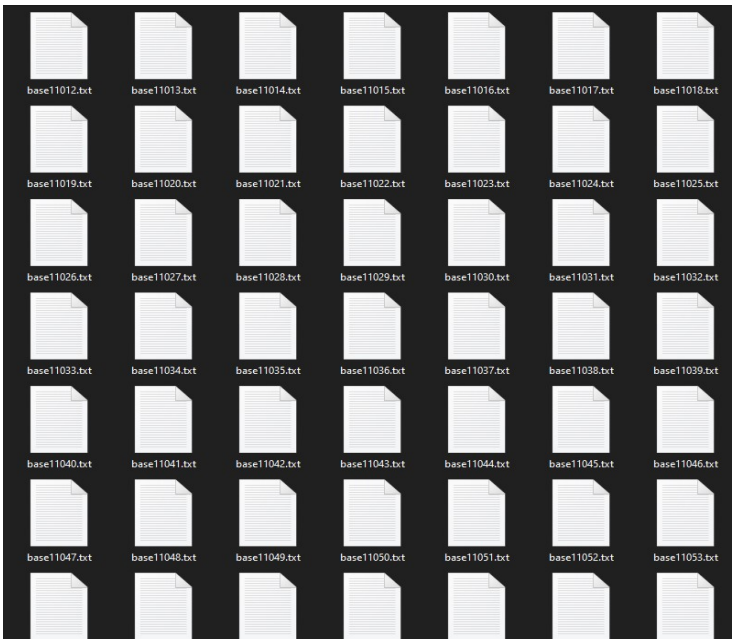
Thus, we decided to apply resizing on word images and made them one fourth of their original size. This enabled us to fit them in a smaller base images.

(Figure 1.1 shows some of our generated base line images)

(Figure 1.2 shows the base image 16558)

2. Target Text Generation

While generating the images, we also generated all the text files for each base image. For example if we have an image named "base1919.png", we also generated a text file named "1919.txt" which has the words in the image in text format.



(Figure 1.3 shows some of our generated base text images)

shopkeepers that

(Figure 1.4 the base16558 text file which was made for base image 16558)

Constructing the CNN Feature Extractor Model

The first part of our CRNN model has a CNN with 7 different layers. Since we have 64 by 256 images as our inputs we used this many layers. Because in each layer except the first one we lowered our height by the factor of 2 since we wanted to have the features in each size from 64 by 256 to 1 by 125.
Our last width is 125 because in the last two convolutional layers we did not want to apply padding on our images since they got really small in terms of their height. Doing any padding at that level may cause us to lose important data at that low resolution.

As our stride parameters we used the stride of 2 in the third layer and in all other layers we applied (2,1) stride. This two typed stride enables us to change the height and width of our input images differently. Since we start from 256 as our width, if we go any deeper than 120 pixels of width or below we may lose characters like 'l', 'i', 'ı', '.', ',', '!'.

From 1 channel we generated up to 512 channels at the end. The number of channels is important because in each channel we store valuable features of our text images. Using these extracted features we will generalize concept of them.

To generalize the features in a great extent, we continued with an LSTM network.

Constructing the LSTM Feature Generalizer Model

Our LSTM model accepts 512 colons and works with 128 hidden states to shrink the knowledge of features into these 128 different neurons. Also it works in a batch first manner, after we receive the output of our CNN model we supply it to LSTM network in format (Batch, Sequence, Colons).

Moreover our LSTM model works in bidirectional manner to read the given colons from both sides. This method enables our model to recognize characters from both right or left according to its features or what lies near to it.

In the beginning we tried to give each colon one by one to our LSTM model however this lead to excessive computation durations and backpropagation errors. This is why we used the batch first method stated above.

Constructing the Fully Connected Character Matcher Model

At the end of our CRNN model we placed a fully connected layer to map the received hidden states to our characters. In our dataset we have 80 different characters including the blank character for the CTC Loss.
During training this fully connected network learns how to match those 128 neurons received from LSTM to integer labeled characters.

Using the CTC Loss Function

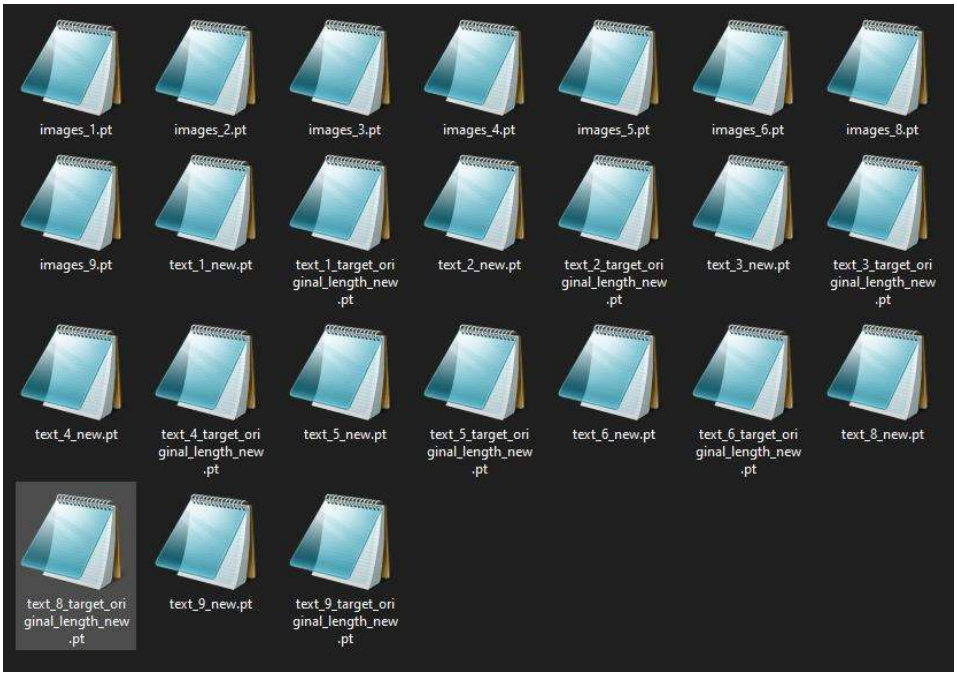
Before using the CTC Loss we searched for some quality documentation and expert help because in Pytorch this loss function can be processed on either CPU or GPU. In our approach we wanted to use the GPU CTC Loss because all of our model was designed to be trained on a GPU. If we were to use the CPU CTC Loss we would end up losing too much time for the type conversion of the tensors from GPU Tensor to CPU Tensor.

1. Preparing Appropriate Tensor Files

Since we decided to use GPU version we had to design our tensors in a way that we can use them without doing too much operation on them in runtime to reduce the computation time. However we had to make every target label array in the same size because if they have different sizes we can not stack them to make a "target label tensor". In our code we do batch shuffling and to do batch shuffling we have to make a bigger tensor object which has all the tensors (image tensor, target label tensor, target length tensor) in it. Later we shuffle this bigger tensor in small amount of time.

Thus we generated each of our target label tensors with the same size and in our code before giving them to CTC Loss, we prepared them in a function called "get_flat_target".

Below you can find some of our tensor files. As you can see we have three types of tensor files named as "images_index.pt", "text_index_new.pt" and "text_index_target_original_length_new.pt"



(Figure 1.5 shows our generated tensor files using images and texts shown above)

2. Converting Tensor Objects

CTC Loss requires four input. In GPU version three of them has to be a GPU tensor file and one of them has to be a CPU tensor file. Normally we did not expect such a situation since the loss calculation happens on GPU however it requires the target label tensor (1 dimensional, flattened target labels) to be a CPU tensor. We did not have to look for the CUDNN implementation but we guess it is because the GPU gets each value in that array in a sequential manner from RAM.

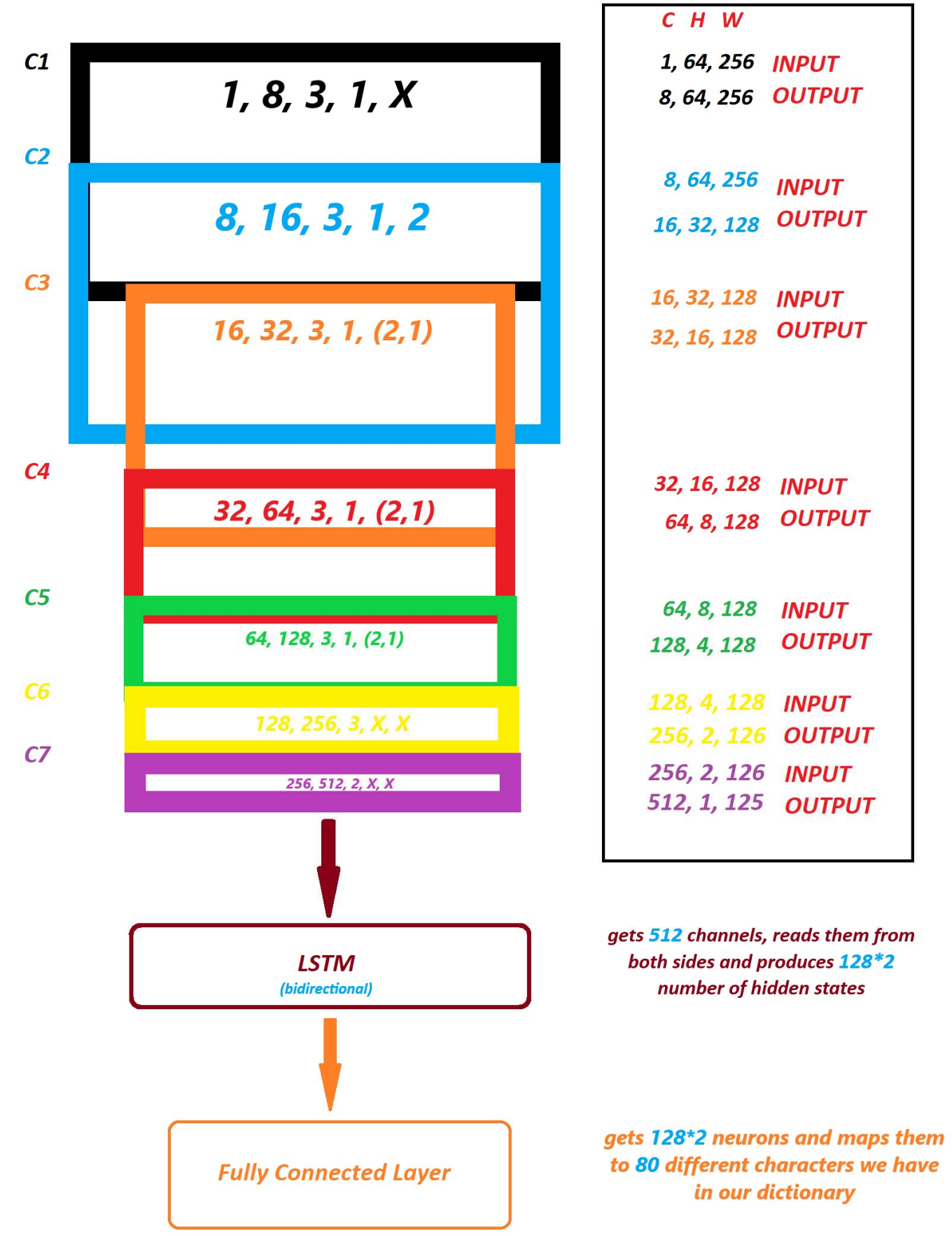
3. Getting the Loss and Applying Backward Pass

After we calculate the loss for that iteration, we apply a backpropagation for that batch on our model. Also we print the loss value to see and supervise our model to make the training faster.

Using the Adam Optimizer

After applying the backpropagation we optimize the values of our model. Optimizaiton means finding some set of parameters to reduce the loss calculated in machine learning.
Moreover we use the stochastic gradient descent to increase our chance of finding these set of parameters in earlier stages of our training.

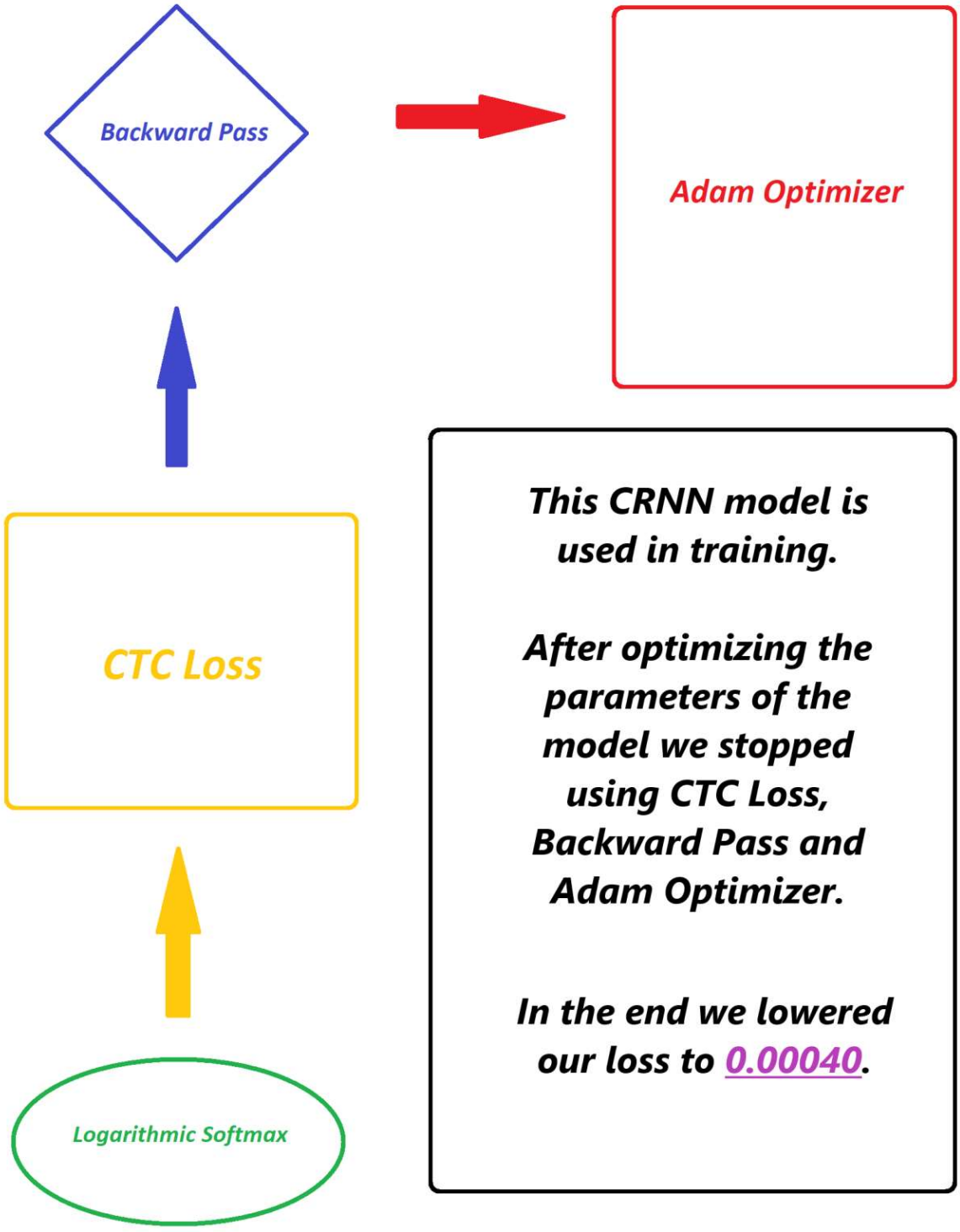
Illustration of the Model Part 1 (Model)



(Figure 1.6, shows our CRNN model)

Illustration of the Model Part 2 (Training & Optimizing Functions)

The part 1 illustrated above sends the inputs of Logarithmic Softmax to the part 2 of our model. As noted this part (except the Logarithmic Softmax) is used in the training phase.



(Figure 1.7, shows the rest of our model used in the training part)