BREAKING CAPTCHA

RAHUL MATHEW

Content

Data Generation

To generate the training data, I used the Optical Character Recognition(Why not) Wikipedia page. I downloaded the page in pdf and converted it into word for easy processing.

Up until this, it was fairly easy and quick. After this point I made several mistakes.

After loading the document, I split it into separate words. I used some regex to get only words and numbers. I wanted to avoid all sorts of links, special characters like parentheses, hyphens, question marks, exclamation points etc.

This was when I made my first mistake. This document contained the same words multiple times, like optical, character, the, for, etc in different capitalization. I used a set to get rid of all the duplicated words but since the words in the list contained the same words with different capitalization, the set operator did not get rid of them. My final list “deduplicated” list had words which were the same but had different capitalization.

To get rid of duplicated words, I made all the words lower case and then I applied the set operator. That way I was left with only words with none repeated. I picked 100 words from this list.

The next step was to assign a number to each word for classification. My next mistake. I decided to zip the 100 words with a list of numbers from 0 to 99. But every time I ran this script, the word number association would be different. The reason is, the set operator scrambles the order of my words every time. So I decided to go with another approach. Finally I got all my words in proper order with consistent numbering.

Easy Set

To create the easy dataset, in plain background. I downloaded 10 different fonts from different places like google fonts, 1001fonts, etc.

Each font will take a different amount of space. So if I were going to create an image dataset with all these fonts and the words, I need to find out what will

be the size of the largest font with the longest word. Once the size of the largest font is determined, I just use that same size to create the entire easy and hard dataset.

The easy set has 1000 images. One font for every word(10*100).

Once the dataset is created, I save each image. For each image, I save the path to the image, word, number associated with the word in a csv file. This will be used later to load the image.

Hard set

Creating the hard set is time taking.

I had 4 backgrounds, with 5 different noise levels and 5 font colours. The font colour will be the hardest to see. Some of the backgrounds I choose are brown in colour. If you add some Gaussian noise and write text in a colour like orange, the text will be very hard to read.

I randomly changed the capitalization of each letter in each word as well. Before creating the image, each letter has a 50% chance to be uppercase or lowercase.

Using the same size as earlier, I created the Hard set. There are a total of 4*5*5*100 = 100,000 images. That is the reason it was time taking. I could not use all the 100,000 images. I will get to the reason later.

I saved all the paths, words and numbers in csv files just like the Easy set.

Classification Task

This task is rather straightforward. As I had already constructed the dataset that way. I load only the images and the number label just like cifar or mnist dataset. The csv files which were saved earlier with image paths, labels, number labels can be used to load the images.

Loading the images can be a very time consuming task. So I used Multi-processing to speed up this process. Here I made a very big mistake. I will come to this later.

Loading all the 100,000 + 1,000 images was not possible. As after the 60,000 images the process throws an error. Because of that I load only 50,000 + 1,000. Out of 50,000 images, 25k images have one background and the other 25k images have another background.

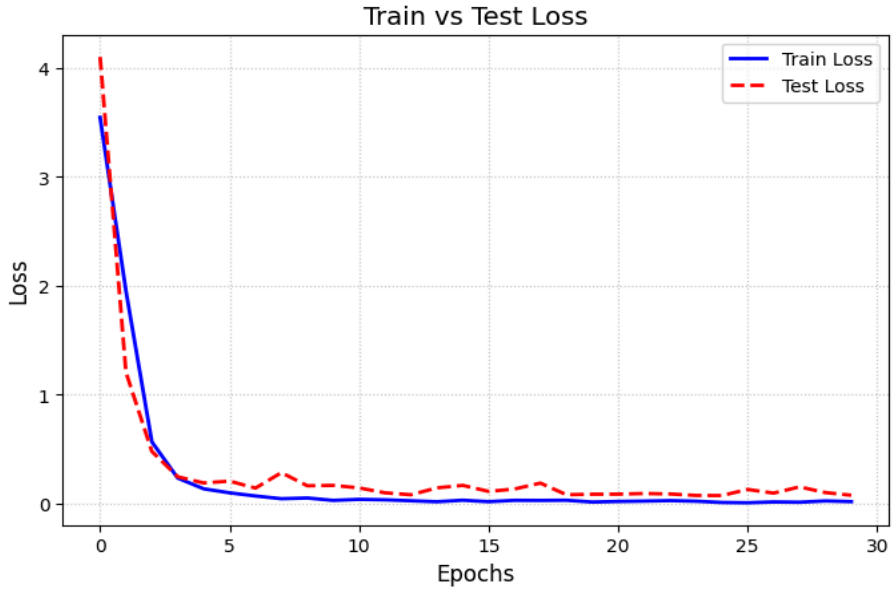
Once the dataset is loaded, split that randomly into train and test set. 90% for the train set and the rest 10% for the test set. Pass the train and test set to data loader classes for batched output.

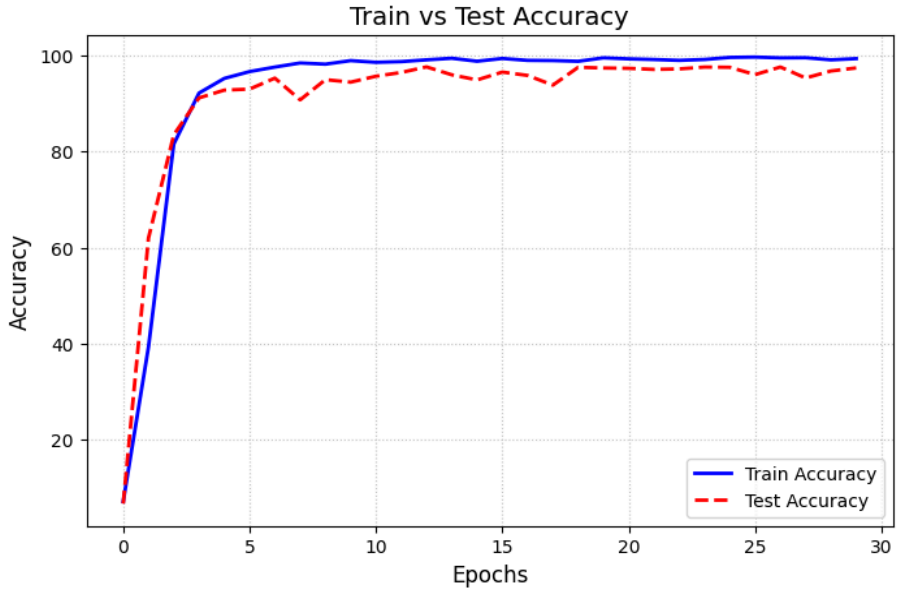
As far as the model goes, I loaded the resnet18 model directly from pytorch. The Resnet18 model which I wrote was taking a lot of time as well as resources to train. This is probably because the resnet18 which is from pytorch is heavily optimized.

Replace the final layer of the loaded resnet18 model with a fully connected layer of 100 classes(loaded resnet18 model has 1000 classes). I added some dropout also for some regularization.

Loaded resnet18 model takes just 6 minutes to finish 30 epochs on an L4 GPU. My resnet18 took 3 minutes just for 1 epoch. In 30 epochs the train accuracy reached 99.36% on 45,900 samples. Test Accuracy reached 97.42% on 5,100 samples.

Here are the Accuracy and Loss plots:





Text Generation

For this task, an Encoder Decoder is best suited. I used a CNN Encoder for the image and a Transformer Decoder for the text. The image is processed by the CNN Encoder and passed to each Decoder block along with the text.

To process the text, it has to be tokenized. I have implemented a character level tokenization with 3 special tokens.

<bos>: beginning of sequence
<eos>: end of sequence
<pad>: pad token

<bos> token is prepended to each word. <eos> is appended to each word. <pad> token is added to each word after the <eos> token to ensure it reaches a certain length.

After each word is tokenized to the same length, input and target sequences should be created. For input sequence, each tokenized word up until the second last element is selected. As for target sequence, shift the sequence by one token.The input and the image will be passed to the model along with a mask for training. During testing, only the image and the input sequence will be passed. The mask is a boolean tensor with the same shape as input tensor with 0 wherever there is pad token and 1 everywhere else.

There is no need for any mask in the Encoder as it is handling images with the same size(here it is a CNN, so no need for any mask). But the Decoder Transformer, which consists of Multi-Head Self Attention and Masked Multi-Head Attention mask has to be used to handle unequal sequence lengths.

During training, the image is passed to the Encoder. After passing through all the cnn blocks positional embedding is applied after the output is flattened. This image embedding is passed to the multi-head attention. The image embedding will become the key and query tensors. The Decoder gets the text embedding after positional embedding is applied. This gets passed to the masked-multi head attention. Here two masks will be applied, one mask is causal and the other is a padding mask. The causal mask is applied to ensure the earlier tokens do not see future tokens. The second mask is a padding

mask to ensure that pad tokens are not attended. Masked multi-head attention is followed by a layer normalization and residual connection.

The output of the masked multi-head attention becomes the query for multi-head attention. The image embedding will become the key and query tensors. Once the attention operation is done, a layer normalization and residual connection is applied. The entire output gets passed to a fully connected layer.

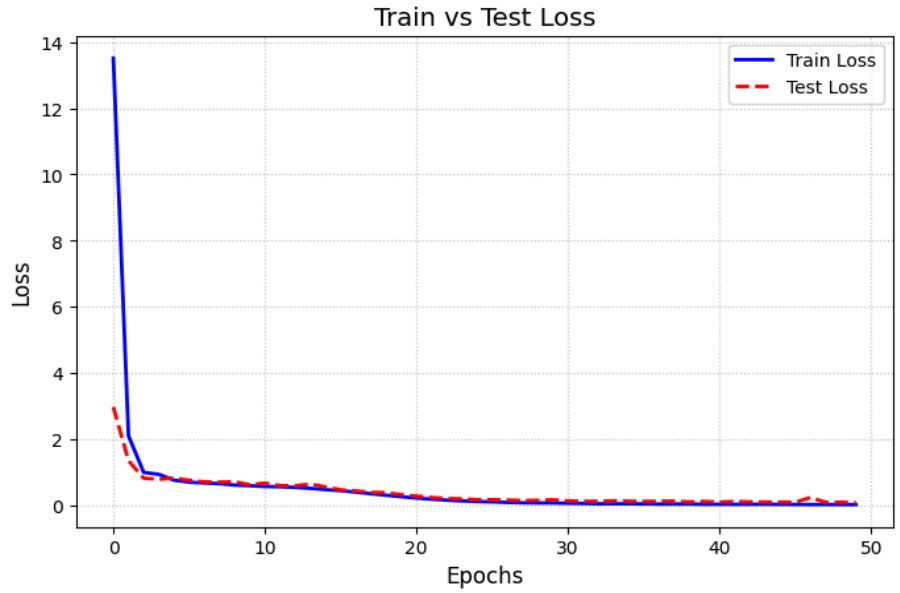
Causal mask is added while padding mask is multiplied. Causal mask is a simple upper trigonal matrix consisting of negative infinities to ensure that after softmax operation, the upper trigonal elements become zero. Ideally the padding mask also consists of negative infinities in the padding tokens to ensure it becomes zero in the padding tokens.

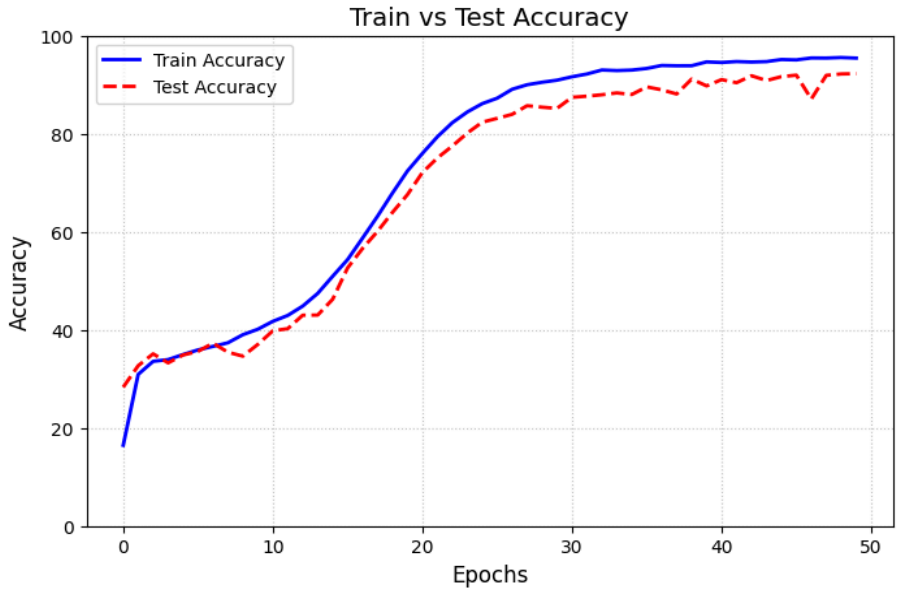
If the padding mask consists of negative infinities and is added to query tokens, entire rows will consist of negative infinities. If softmax is applied on top of that row, it will not become zero. As softmax internally subtracts the largest value to ensure no overflow, some values in that row will not become zero.

To ensure the entire pad query tokens becomes zero(as they should not be attended), padding mask is multiplied to the output of softmax operation. Causal mask is added before the softmax operation.

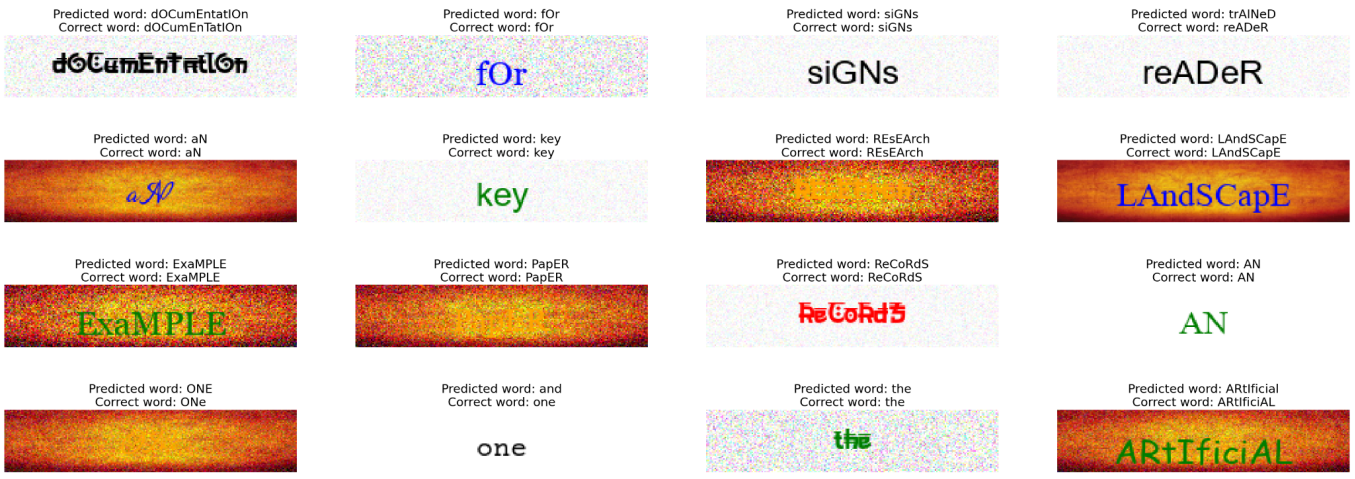
The loss function to this Transformer model will calculate values for all the tokens in the input. But the pad tokens should not contribute to the loss value. For that multiply the final loss vector with the padding mask before the reduction.

Here are the plots:





The trained model tested on some test samples



Additional findings

In the classification task I had mentioned that I made a big mistake with the loading of images using multi-processing. I used ProcessPoolExecutor for loading images in parallel. Initially I used executor.submit to load the images. submit method when used to load images scrambles the order while loading the images. So when my images were finally loaded, the labels and images were mismatched.

I trained a resnet18 model just like in the classification task on this scrambled dataset. But my training loss is still reduced. My training accuracy still reached 99%. In some runs it even reached 100% accuracy. But as expected my test accuracy was poor. Barely better than random.

This kind of behaviour indicates only one thing. The model has successfully memorized the entire dataset. There is no deeper pattern in this dataset as the inputs and outputs are scrambled.

This is consistent with findings of this 2017 paper, Understanding deep learning requires rethinking generalization. In this paper, the authors scrambled the cifar10 and imagenet dataset images and labels. Their model again was able to completely memorize the entire dataset. The training accuracy reached 100% but the test accuracy as expected was barely better than random. This randomized dataset is all noise and no signal. It has successfully memorized pure noise.. But when the original dataset was given, it performed well and was able to score high on both the train and test datasets. The model has successfully generalized outside of the train set.

The findings of this paper are very interesting. It calls into question the capabilities of over parametrized models. Today's Large Language models have billions of parameters. They are most likely over parametrized compared to the number of training tokens. But they are still able to generalize beyond their dataset.