Captcha Recognition

Amol Vijayachandran

Choosing the task

Choosing a task was a tough decision, each theme offered a unique challenge, all of which I would love to try and tackle later in my free time.

The NLP task would've allowed me to explore word embeddings, a topic I have always been curious about, while the Graphs task's open-endedness was intriguing. The Math&AI task was also appealing, LLM evaluation being something I already had a bit of experience in.

However, I ultimately decided to go with the Computer Vision theme – Captcha Recognition. I have played around with training an image-based neural network before (for the MNIST challenge), which allowed me to quickly grasp the basic ideas I required to start with the task. The task also intrigued me the most, having more of a real usable result when compared to the other tasks.

Task 0 : Dataset Creation

I had a difficult time starting this, mainly because I couldn't decide what exact requirements I had for my dataset. I couldn't decide the image size, fonts to use, and many other variables.

However, I decided to jump right into it, figure it out with experimentation. My initial plan was to use images of size 200x50x3 pixels, have maybe 50 fonts, end up with a total of 250 images per word. With a dictionary of about 600 common words, I would have 150,000 samples to use.

I quickly ran into problems like the word not fitting in the image and not being able to find enough fonts. This resulted in me upping the image size to 250x100x3 and reducing the number of fonts to 22. With some more random parameters, like font size, noise in the image, font color, and random capital letters, I ended up with 133 images per word.

Task 1 : Classification | Architecture

With the basis of the dataset all ready, I proceeded onto task 1 – classifying images from a 100 selected words. My initial plan for the architecture was to use three convolutional layers, with a linear layer. I was going to use Cross-Entropy Loss and the Adam Optimizer with a learning rate of $\alpha = 0.001$. I decided to let the model train for 20 epochs, then see where to change things.

I split my dataset of 13,000 words into train/val partitions, with a ratio of 80:20. When I started training the model, I noticed that I would get very big jumps in the cost of the val dataset (from 0.5 to 112!), but my train cost had a continuous smooth reduction. This led me to believe that the model might be overfitting. To combat this, I added dropout with $p = 0.4$, which didn't really help results.

Task 1 : Classification | Dataset

I looked through the samples in the val dataset and a few things came to mind. One, some fonts were so thin that applying 20% noise basically made the text invisible and two, the size of the text in the images varied widely, despite me using the same font size for all. To combat this, I removed the very thin font, leaving me with 21 fonts. I also added two different font sizes for each existing image, basically doubling my dataset. I had 25,000 samples to now work with.

Learning about practices in ML also led me to split my dataset into train/val/test partitions, with a 70:20:10 ratio. This meant I had 18k samples to train with, 5k to evaluate during training, and 2k to test post training.

Task 1 : Classification | Testing

With an improved dataset, I wanted to experiment with reducing the learning rate as I suspected that was what was causing such huge jumps in my val loss. I made the learning rate $\alpha = 0.0001$, reducing it by a magnitude of 10. To compensate for this, I doubled the number of epochs that I was going to train the model for, making the training process also longer.

Observing the training, I noticed consistent decrease in both train loss and val loss. At the end of the 40 epochs, I had a model that evaluated for 49.23% accuracy on the test dataset, and had similar accuracies for both the train and val datasets too. However, it was clear that the model was capable of achieving more, based on the rate at which it was growing.

Task 1 : Classification | Success!

I decided to let the model run for another 60 epochs – totaling a 100 epochs. This gave me great results, with val accuracy climbing up to 92.75% accuracy. I am confident this could increase with more compute time, but I didn't want to spend any more of my precious Kaggle GPU hours on trying to improve it.

When I tested the model on the test dataset, I got a satisfying 93.26%. I am confident this architecture is capable of learning more with more compute time, but I did not want to risk overfitting.

```
Epoch [1/100] Train Loss: 4.3706, Train Acc: 0.0389 || Val Loss: 4.0041, Val Acc: 0.0653
Checkpoint saved at epoch 1 with val_loss: 4.0041
Epoch [2/100] Train Loss: 3.8906, Train Acc: 0.0611 || Val Loss: 3.6225, Val Acc: 0.0811
Checkpoint saved at epoch 2 with val_loss: 3.6225
Epoch [3/100] Train Loss: 3.5854, Train Acc: 0.0701 || Val Loss: 3.4222, Val Acc: 0.0791
Checkpoint saved at epoch 3 with val_loss: 3.4222
Epoch [4/100] Train Loss: 3.4282, Train Acc: 0.0721 || Val Loss: 3.3162, Val Acc: 0.0875
Checkpoint saved at epoch 4 with val_loss: 3.3162
Epoch [5/100] Train Loss: 3.3369, Train Acc: 0.0730 || Val Loss: 3.2157, Val Acc: 0.0931
Checkpoint saved at epoch 5 with val_loss: 3.2157
Epoch [6/100] Train Loss: 3.2820, Train Acc: 0.0790 || Val Loss: 3.1742, Val Acc: 0.0913
Checkpoint saved at epoch 6 with val_loss: 3.1742
Epoch [7/100] Train Loss: 3.2324, Train Acc: 0.0843 || Val Loss: 3.1415, Val Acc: 0.0995
Checkpoint saved at epoch 7 with val_loss: 3.1415
Epoch [8/100] Train Loss: 3.1925, Train Acc: 0.0914 || Val Loss: 3.1187, Val Acc: 0.1053
Checkpoint saved at epoch 8 with val_loss: 3.1187
Epoch [9/100] Train Loss: 3.1609, Train Acc: 0.0959 || Val Loss: 3.0847, Val Acc: 0.1097
Checkpoint saved at epoch 9 with val_loss: 3.0847
Epoch [10/100] Train Loss: 3.1261, Train Acc: 0.1019 || Val Loss: 3.0615, Val Acc: 0.1139
Checkpoint saved at epoch 10 with val_loss: 3.0615
Epoch [11/100] Train Loss: 3.0997, Train Acc: 0.1125 || Val Loss: 3.0408, Val Acc: 0.1155
Checkpoint saved at epoch 11 with val_loss: 3.0408
Epoch [12/100] Train Loss: 3.0678, Train Acc: 0.1103 || Val Loss: 2.9899, Val Acc: 0.1359
Checkpoint saved at epoch 12 with val_loss: 2.9899
Epoch [13/100] Train Loss: 3.0306, Train Acc: 0.1220 || Val Loss: 2.9643, Val Acc: 0.1381
...
Epoch [99/100] Train Loss: 0.5653, Train Acc: 0.8411 || Val Loss: 0.3708, Val Acc: 0.9207
Checkpoint saved at epoch 99 with val_loss: 0.3708
Epoch [100/100] Train Loss: 0.5603, Train Acc: 0.8401 || Val Loss: 0.3650, Val Acc: 0.9275
Checkpoint saved at epoch 100 with val_loss: 0.3650
```

Task 1 : Classification

I was also curious about how the model would perform on words that aren't present in the training data.

I tried convincing myself that the model was in fact seeing some patterns, predicting "management" for an image with the word "government", as shown in the top image on the right. The model also predicted "person" for an image showing "strong", where maybe it tried to match the 'o' and the 'n'.

However, this was obviously not entirely true as it predicted the word "several" for an image reading "trouble", with only one character somewhat in the same place. The model at least seemed capable of roughly figuring out how many letters were present in the word, shown by more examples in the next slide.

Predicted: management

Predicted: person

Predicted: several

98%!!!

All of the above samples were guessed with a confidence of 50%+.

Task 2 : Generation | Architecture

Unlike Task 1, I had no idea how to approach the architecture for this task. I had to do a lot of digging around on current OCR architectures and their advantages and disadvantages.

I rounded it down to three main selections:

- CNN-RNN-CTC models
- Attention based Encoder-Decoder models
- Transformer based approaches
- Fully convolutional approaches

Task 2 : Generation | Architecture

Out of these, I immediately disregarded a fully convolutional approach, as there would be a lack of sequential modeling. Captcha recognition is inherently a sequence problem where the order and relationship of characters is critical. A fully convolutional approach would also have difficulty handling variable-length.

After doing some reading on transformer-based approaches, I decided this wouldn't be ideal for the given task. Although transformer models have been revolutionary, the main disadvantages here would be the high compute necessary to train and the large amount of data that would be required.

This left me with two main options, CNN-RNN-CTC models and Attention-based Encoder-Decoder models.

Task 2 : Generation | Architecture

CNN-RNN-CTC Models

- Connectionist Temporal Classification (CTC) loss handles alignment implicitly. It marginalizes over all possible alignments between extracted features & target sequence. Alignment is not explicitly learnt.
- CTC allows for variable-length sequences via a “blank” symbol. However, the assumption of roughly sequential alignment can affect performance when structure of the input is more complex.
- Absence of explicit attention results in long-range dependencies that may not be modeled effectively.

Attention-based Encoder-Decoder Models

- Explicit alignment is learnt through an attention mechanism. The decoder dynamically ”attends” to different parts of the encoded feature map sequence when generating each output token.
- Attention naturally allows for variable-length inputs and outputs. Decoder generates one token at a time, and this allows it to easily adapt to irregularities in the input.
- Attention mechanism allows the decoder to handle long-range dependencies, which is beneficial when dealing with noise.

Task 2 : Generation | Architecture

Finally, I ended up decided that the Attention-Based Encoder-Decoder model would be ideal for this task as features such as explicit alignment, ability to deal with complex input structure, and ability to deal with noise in the image all were beneficial to the given task, where we have data with different fonts of varying complexity and varying noise levels.

The realization that the given task was essentially a sequence-to-sequence problem also heavily influenced my decision to employ an Attention-based Encoder–Decoder model, given their widespread popularity for such applications.

This reddit thread also gave me some clarity on why I should choose which architecture: https://www.reddit.com/r/MachineLearning/comments/7m6lp4/d_cnnrnnctc_vs_attentionencoderdecoder/

Task 2 : Generation | Architecture

With the architecture decided, I wanted to delve into its inner workings. I couldn't find many resources explaining how an Attention-based Encoder-Decoder model works for OCR, but I figured learning about seq2seq in language applications would give me some clarity.

Some resources I found massively helpful in my journey to understand this architecture:

- https://jalammar.github.io/illustrated-transformer/
- https://youtu.be/L8HKweZIOmg?si=mDmE5u6jr750CXI5
- https://youtu.be/PSs6nxngL6k?si=OJDUYHr7-3G35eAX

While trying to understand seq2seq models, I ended up exploring everything — from the basics of RNNs & LSTMs. This proved to be helpful later, when I would implement an RNN for the decoder via LSTM cells.

At this point, I had learnt so much about ML in language applications that I felt confident in even attempting the NLP task ☺.

Task 2 : Generation | Architecture

I had now decided the architecture for the Attention-based Encoder-Decoder model. There were five main components:

- Encoder: This is a CNN, which takes in a batch of images and outputs a feature maps for those images.
- Attention: Fully connected layers that map the encoder outputs and the decoder state into a context vector.
- Decoder: This is an RNN, made with LSTM cells.
- Metrics: Word error rate (WER), Accuracy, and Cross-Entropy Loss

Task 2 : Generation | Encoder

Now, a few details about each module.

The encoder module uses a CNN to extract feature maps from the image.

- Input: (batch, 3, 100, 250)
- Layer 1: (batch, 3, 100, 250) → (batch, 64, 50, 125)
- Layer 2: (batch, 64, 50, 125) → (batch, 128, 25, 62)
- Output: (batch, 128, 25, 62)

The output tensor from the CNN is reshaped where the channels and height dimensions are merged, and the last two dimensions are swapped. This gives us a tensor where data can be sequentially processes, where the width is treated as a time step and the merged channel/height form a feature vector representing that time step.

Task 2 : Generation | Attention

The attention module uses the **Bahdanau Attention** mechanism – also called additive attention – as an alignment mechanism. Rather than compressing the entire input sequence into a single fixed-length vector, it allows for a context vector to be computed which lets the model dynamically focus on different parts of the encoder's outputs at each decoding step.

Key features:

- In each decoding step, **alignment scores** are computed for every encoder hidden state based on the current decoder hidden state.
- The scores are then normalized, which produce **attention weights**. These determine how much influence each part of the input sequence has.
- The weighted sum of the encoder hidden states using the attention weights gives us a **context vector**, which provides the decoder with input relevant to generating the next output token.

Task 2 : Generation | Decoder

The decoder is a RNN that converts the encoded image features into a sequence of output tokens, one time step at a time.

For each time step, there are six main steps:

1) Token embedding – The current input token is converted to an embedding vector
2) Attention – Context vector is retrieved from the attention module
3) Gating – A gating mechanism controls the flow of information
4) LSTM update – The token embedding & gated context vector are passed into an LSTM cell.
5) Output – The updated hidden state is then used to predict the next token
6) Teacher forcing – The module might use the “true” next token as the input for the following time step

Task 2 : Generation | Metrics

There are three main metrics I am using to evaluate the model:

- Word Error Rate (WER): A metric to quantify the difference between the predicted word and the ground truth. S – Substitutions, D – Deletions, I – Insertions, N – Word length

$$\frac{S + D + I}{N}$$

- Accuracy: How many words are guessed completely correct
- Cross-Entropy Loss: Typical loss function, quantifies the difference the ground truth distribution and the predicted distribution.

Task 2 : Generation | Training

I first trained the model for 20 epochs, and got decent results – 87% accuracy and 0.04WER. However, it was clear the model could achieve better results looking at how the loss decreased over each epoch, so I decided to run it for 50 epochs total.

```
100%|██████████| 948/948 [00:38<00:00, 24.47it/s]
Test Loss: 0.0684, Test WER: 0.0150, Test Accuracy: 0.9643
```

Those are very good results! Suspiciously good in fact. I suspected overfitting in the model, but that didn't make sense as the test and train datasets had exclusive samples, with different permutations of fonts, colors, sizes.

Predicted: serious

Predicted: program

Predicted: nature

Predicted: interview

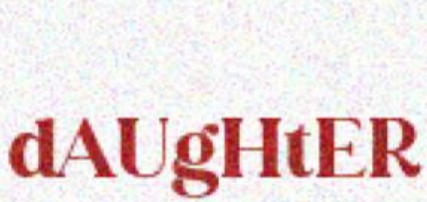
Predicted: thought

Predicted: recognize

Predicted: president

Predicted: state

Predicted: daughter



Task 2 : Generation | Random

I started to experiment around with the model. First off, I wanted to set a baseline for a random simulation (one of the requirements is to perform better than random). I gave the simulation an advantage – it was allowed to to know how many tokens it needs to generate every time.

I got the following results:

```
100%|██████████| 948/948 [00:17<00:00, 52.97it/s]
Random Loss: 3.8405, Random WER: 1.0137, Random Accuracy: 0.0000
```

As expected, these results were horrible. WER being greater than one implies that most predicted words didn't even get single letter correct.

Task 2 : Generation | Experimentation

At this point, I think the task is completed. For the dataset I had created, it achieves performance much much better than a random baseline. However, I was curious on how the model performs on random sequences of letters, wondering whether the model had somehow learnt how to only recognize English words.

```
100%|██████████| 63/63 [00:02<00:00, 24.56it/s]
Exclude Loss: 10.7684, Exclude WER: 0.8708, Exclude Accuracy: 0.0000
```

These results interested me. Better than random, but still really really bad! I was curious at what was happening, and decided to look at things sample by sample.

Task 2 : Generation | Experimentation

From the samples I went through, it seemed like the model had overfit on specific sequences of characters present in the training dataset samples.

I decided to do some more digging around and test it on random words that I could come up with.

Predicted: magazinal

Predicted: qualit

Predicted: harriage

Task 2 : Generation | Experimentation

Clearly my previous assumption doesn't seem to be the case. I had brushed aside overfitting on the dataset, as testing with the same architecture trained for a lower number of epochs gives me similar results. The number of variations in the dataset made me think that the model couldn't overfit, however the variations were only in terms of physical appearance, not the actual words themselves.

I realized this was happening because my dataset only had 600 words. When I had designed the initial dataset, I had prioritized Task 1, where I required many different permutations of samples for each word. However, this task requires a different approach.

Predicted: identify

Predicted: commercial

Predicted: civil

Task 2 : Generation | Re-attempt

At this point, even with gibberish / out of sample words the model was performing better than the random baseline. However, just for experimentation's sake, I decided to create a new dataset and try training the same architecture on it.

Rather than focusing on having multiple permutations for each word, I focused on having a very large dictionary of words. I ended up with about a 100k training samples containing about 50k words. I also realized that the reason for why the val/test metrics were so high before was because the partitions had exclusive permutations, but shared words. This time, I decided to make the words exclusive per partition as well.

Task 2 : Generation | Training

I decided to let the model train for 50 epochs, but stopped it 38 epochs in.

```
Epoch [37/50] | Train Loss: 1.1589, Train WER: 0.4962 | Val Loss: 1.8201, Val WER: 0.6756, Val Accuracy: 0.1050
100%|██████████| 6155/6155 [08:07<00:00, 12.64it/s]
100%|██████████| 1179/1179 [00:45<00:00, 26.16it/s]
Epoch [38/50] | Train Loss: 1.1387, Train WER: 0.4901 | Val Loss: 1.8289, Val WER: 0.6867, Val Accuracy: 0.1009
100%|██████████| 6155/6155 [08:07<00:00, 12.64it/s]
100%|██████████| 1179/1179 [00:44<00:00, 26.20it/s]
```

This was because even 38 epochs in, the val loss was huge – not having decreased in over 20 epochs. I realized that this model wasn't learning anymore. Looking at my architecture, there wasn't much scope for where I could add more capability for the model to capture more complex patterns. I decided it was time to consider an architecture redesign.

Task 2 : Generation | Architecture

Going back to the previous list of architectures, my main other option was CNN-RNN-CTC models. However, I had to rule this out as I was looking for an architecture with capability for more complex representation, which I didn't think a CNN-RNN-CTC model would offer. Despite disregarding transformer-based models due to the high compute necessary, I had to consider it now as I required their capability to model complex relationships through mechanisms like self-attention. The lack of being able to capture sequential relationships concerned me, but this was solvable with a positional encoder.

All things considered, I decided to give transformer based models a try.

Task 2 : Generation | Architecture

The model was going to consist of three main components:

- Encoder: This was the same as the previous architecture, just a CNN to extract feature maps.
- Positional Encoder: Adds information about the position of each token into its embedding. Since transformers process all tokens simultaneously, there is no notion of sequence order. The positional encoding provides a deterministic way to add this order information. Same design as used in the "Attention is All You Need" paper.
- Transformer Decoder: Responsible for generating predictions for each token by attending to both previously generated tokens (using self-attention) and to the encoder's outputs (via cross—attention).

Task 2 : Generation | Architecture

Self-Attention: Each token in a sequence computes its representation by "attending" to every other element in the sequence. This allows the model to capture relationships and dependencies between different parts of the same input, regardless of their positions. When a model "attends" to an input, it evaluates which parts of the input are most informative for a prediction. This is done by comparing a query (current state of decoder) with keys (representations of input), obtaining an "attention score", a representation of the similarity between the query and each key, and focusing on the parts with higher score.

Cross-Attention: Used to related two different sequences of information. The query vectors come from one sequence (i.e. the decoder's current state) while key vectors come from another (encoder's outputs). This allows the model to focus on relevant parts from a different input.

Task 2 : Generation | Resources

I found these resources quite helpful in trying to understand attention. While I don't fully understand the working – I was able to grasp a basic understanding of it:

- https://youtu.be/PSs6nxngL6k?si=TyOAqqtnw25o3Ugj
- https://youtu.be/eMlx5fFNoYc?si=H0e_AWPcLFAMOwn8
- https://youtu.be/zxQyTK8quyY?si=hwM3GsCPSpVPjeRW

Task 2 : Generation | Architecture

My implementation for the Transformer Decoder module was almost entirely using PyTorch's built in built-in transformer decoder layers, which contain built in details for self-attention and cross-attention, along with multi-head processing, etc. The main parameters I had to play around with were the embedding dimension, number of attention heads, and the number of layers.

An additional detail about the decoder: a causal mask is created during the forward pass to prevent a token at a later position from attending to positions later than its own. This helps to maintain the autoregressive property of the model.

Task 2 : Generation | Training

Deciding values for the hyper-parameters was a bit confusing, as I had no prior experience with transformers, and didn't have the time to do a deep-dive and understand their working entirely. I talked with one of my seniors who had done work with transformers, and he suggested I use 8 attention heads, and to play around with the other two parameters.

I decided to start with an embedding dimension of 256, and 3 hidden decoder layers. I chose this as a starting point as it keeps the complexity of the model simple, allowing for faster training whilst having space to increase the complexity if necessary.

Task 2 : Generation | Training

I decided to let the model train for 50 epochs, and saw convergence in the val loss around 15 epochs in.

This was most likely due to the model not being able to represent complex enough relationships, so I decided to first try adding a few hidden layers.

```
Training Epoch 10: 100%|██████████| 6156/6156 [06:23<00:00, 16.06it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.56it/s]
Epoch [10/50] | Train Loss: 1.1877, Train WER: 0.5930 | Val Loss: 1.0813, Val WER: 0.6124, Val Accuracy: 0.1711
Checkpoint saved at epoch 10 with val_loss: 1.0813
Training Epoch 11: 100%|██████████| 6156/6156 [06:24<00:00, 16.00it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.88it/s]
Epoch [11/50] | Train Loss: 1.1480, Train WER: 0.5825 | Val Loss: 1.0692, Val WER: 0.5723, Val Accuracy: 0.1838
Checkpoint saved at epoch 11 with val_loss: 1.0692
Training Epoch 12: 100%|██████████| 6156/6156 [06:21<00:00, 16.13it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.65it/s]
Epoch [12/50] | Train Loss: 1.1134, Train WER: 0.5702 | Val Loss: 1.0835, Val WER: 0.6140, Val Accuracy: 0.1718
Training Epoch 13: 100%|██████████| 6156/6156 [06:21<00:00, 16.13it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.86it/s]
Epoch [13/50] | Train Loss: 1.0797, Train WER: 0.5566 | Val Loss: 1.0659, Val WER: 0.5951, Val Accuracy: 0.1767
Checkpoint saved at epoch 13 with val_loss: 1.0659
Training Epoch 14: 100%|██████████| 6156/6156 [06:22<00:00, 16.07it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.66it/s]
Epoch [14/50] | Train Loss: 1.0490, Train WER: 0.5439 | Val Loss: 1.0750, Val WER: 0.6099, Val Accuracy: 0.1821
Training Epoch 15: 100%|██████████| 6156/6156 [06:23<00:00, 16.07it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.87it/s]
Epoch [15/50] | Train Loss: 1.0196, Train WER: 0.5316 | Val Loss: 1.0477, Val WER: 0.5633, Val Accuracy: 0.1921
Checkpoint saved at epoch 15 with val_loss: 1.0477
Training Epoch 16: 100%|██████████| 6156/6156 [06:20<00:00, 16.19it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.67it/s]
Epoch [16/50] | Train Loss: 0.9909, Train WER: 0.5234 | Val Loss: 1.0425, Val WER: 0.5559, Val Accuracy: 0.1977
Checkpoint saved at epoch 16 with val_loss: 1.0425
Training Epoch 17: 100%|██████████| 6156/6156 [06:19<00:00, 16.23it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.79it/s]
Epoch [17/50] | Train Loss: 0.9636, Train WER: 0.5094 | Val Loss: 1.0430, Val WER: 0.5655, Val Accuracy: 0.2001
Training Epoch 18: 100%|██████████| 6156/6156 [06:19<00:00, 16.23it/s]
Evaluating: 100%|██████████| 1176/1176 [00:39<00:00, 29.69it/s]
Epoch [18/50] | Train Loss: 0.9379, Train WER: 0.4980 | Val Loss: 1.0642, Val WER: 0.6088, Val Accuracy: 0.1903
```

Task 2 : Generation | Training

This didn't see much improvement, and the model ended up converging at around 15 epochs again.

```
Epoch [14/20] | Train Loss: 0.9812, Train WER: 0.5217 | Val Loss: 1.0799, Val WER: 0.6084, Val Accuracy: 0.1767
Training Epoch 15: 100%|██████████| 6156/6156 [07:02<00:00, 14.58it/s]
Evaluating: 100%|██████████| 1030/1030 [00:37<00:00, 27.37it/s]
Epoch [15/20] | Train Loss: 0.9449, Train WER: 0.5039 | Val Loss: 1.0899, Val WER: 0.6650, Val Accuracy: 0.1784
```

This led to me trying to increase the embedding dimension to 512, and see if that helped the model recognize the complex patterns better.

Task 2 : Generation | Training

The model did end up converging at around the same number of epochs, but the val loss was a bit lower this time, and my best WER was also lower.

I decided to call it quits here, I had reached a WER score of around half of the random baseline and I was running out of time to train the model (training 20 epochs took 3 hours!)

```
Epoch [10/20] | Train Loss: 1.0605, Train WER: 0.5594 | Val Loss: 0.9771, Val WER: 0.5076, Val Accuracy: 0.2294
Checkpoint saved at epoch 10 with val_loss: 0.9771
Training Epoch 11: 100%|██████████| 5978/5978 [07:50<00:00, 12.71it/s]
Evaluating: 100%|██████████| 1139/1139 [00:56<00:00, 20.04it/s]
Epoch [11/20] | Train Loss: 1.0169, Train WER: 0.5427 | Val Loss: 0.9993, Val WER: 0.5379, Val Accuracy: 0.2072
Training Epoch 12: 100%|██████████| 5978/5978 [07:51<00:00, 12.67it/s]
Evaluating: 100%|██████████| 1139/1139 [00:56<00:00, 20.04it/s]
Epoch [12/20] | Train Loss: 0.9774, Train WER: 0.5279 | Val Loss: 0.9693, Val WER: 0.5436, Val Accuracy: 0.2299
Checkpoint saved at epoch 12 with val_loss: 0.9693
Training Epoch 13: 100%|██████████| 5978/5978 [07:54<00:00, 12.61it/s]
Evaluating: 100%|██████████| 1139/1139 [00:56<00:00, 20.07it/s]
Epoch [13/20] | Train Loss: 0.9424, Train WER: 0.5137 | Val Loss: 0.9755, Val WER: 0.5129, Val Accuracy: 0.2282
Training Epoch 14: 100%|██████████| 5978/5978 [07:52<00:00, 12.65it/s]
Evaluating: 100%|██████████| 1139/1139 [00:57<00:00, 19.89it/s]
Epoch [14/20] | Train Loss: 0.9076, Train WER: 0.4982 | Val Loss: 0.9949, Val WER: 0.5568, Val Accuracy: 0.2117
Training Epoch 15: 100%|██████████| 5978/5978 [08:01<00:00, 12.40it/s]
Evaluating: 100%|██████████| 1139/1139 [00:56<00:00, 20.19it/s]
Epoch [15/20] | Train Loss: 0.8746, Train WER: 0.4868 | Val Loss: 0.9915, Val WER: 0.5378, Val Accuracy: 0.2245
Training Epoch 16: 100%|██████████| 5978/5978 [07:52<00:00, 12.65it/s]
Evaluating: 100%|██████████| 1139/1139 [01:00<00:00, 18.80it/s]
Epoch [16/20] | Train Loss: 0.8420, Train WER: 0.4732 | Val Loss: 1.0240, Val WER: 0.6119, Val Accuracy: 0.2024
Training Epoch 17: 100%|██████████| 5978/5978 [07:56<00:00, 12.53it/s]
Evaluating: 100%|██████████| 1139/1139 [00:57<00:00, 19.70it/s]
Epoch [17/20] | Train Loss: 0.8127, Train WER: 0.4563 | Val Loss: 1.0207, Val WER: 0.5727, Val Accuracy: 0.2061
Training Epoch 18: 100%|██████████| 5978/5978 [07:59<00:00, 12.46it/s]
Evaluating: 100%|██████████| 1139/1139 [00:57<00:00, 19.98it/s]
Epoch [18/20] | Train Loss: 0.7854, Train WER: 0.4418 | Val Loss: 1.0210, Val WER: 0.5678, Val Accuracy: 0.2135
Training Epoch 19: 100%|██████████| 5978/5978 [07:54<00:00, 12.61it/s]
Evaluating: 100%|██████████| 1139/1139 [00:56<00:00, 20.13it/s]
Epoch [19/20] | Train Loss: 0.7574, Train WER: 0.4278 | Val Loss: 1.0320, Val WER: 0.5432, Val Accuracy: 0.2177
Training Epoch 20: 100%|██████████| 5978/5978 [08:04<00:00, 12.33it/s]
Evaluating: 100%|██████████| 1139/1139 [00:58<00:00, 19.41it/s]
Epoch [20/20] | Train Loss: 0.7317, Train WER: 0.4122 | Val Loss: 1.0682, Val WER: 0.5581, Val Accuracy: 0.2022
```

Task 2 : Generation | Evaluation

I decided to implement another metric to the evaluation, character-level accuracy. This measures how many of the characters are the same between the predicted sequence and the ground truth and assigns a value of $0 - 1$ per predicted sequence.

Evaluating the model on the test dataset, I got the following results:

```
Evaluating: 100%|██████████| 567/567 [00:31<00:00, 17.88it/s]
Test Loss: 0.9192, Test WER: 0.5200, Test Accuracy: 0.2420, Test Char Accuracy: 0.3938
```

Comparing this to the random baseline:

```
100%|██████████| 567/567 [00:14<00:00, 39.33it/s]
Random Loss: 3.8359, Random WER: 1.1398, Random Accuracy: 0.0045, Random Char Accuracy: 0.0351
```

Clearly, our model outperforms the random baseline by quite a bit.

Task 2 : Generation | Evaluation

I was curious what was happening sample by sample, so I took a look.

Surprisingly, it seemed like the model was just spitting out garbage – focusing on a few specific characters present in the image and ignoring the rest. It also seemed to mess up the sequence of letters in a few samples, maybe a problem with the positional encoder? Even the number of tokens didn't seem to be correct, which explains why the WER wasn't going down (due to the high deletion cost). Reducing the max token count to 20 brings down the WER on the test set to 0.46.

Looking around on the internet about this problem, this seems to be a common problem with transformers in similar tasks, like captioning.

Predicted: aatt

Predicted: waw

Predicted: iut

Task 2 : Generation | Experimentation

Going back to the previous random sequence of letters experiment, I wanted to compare the new architecture with the old one:

```
Evaluating: 100%|██████████| 63/63 [00:03<00:00, 20.28it/s]
Exclude Loss: 4.0464, Exclude WER: 0.6855, Exclude Accuracy: 0.0020
```

New architecture

```
100%|██████████| 63/63 [00:02<00:00, 24.56it/s]
Exclude Loss: 10.7684, Exclude WER: 0.8708, Exclude Accuracy: 0.0000
```

Previous architecture

Clearly, the new architecture performs better (despite being trained for less than half the epochs).

Task 2 : Generation | Conclusion

With these results, I am satisfied with the performance of the model.

I would definitely like to explore other architectures, spend some more time investigating the reason the transformer stops learning so early. My current guess is that I didn't have enough data, but I didn't have enough time to try and train a model with even more data. I would also like to go back to simpler architectures, try out CRNNs, maybe some simple segmentation models.

Bonus Task | Training

For this bonus task, I decided to go with my transformer-based architecture, as I felt that would have more capability at capturing the difference between the green and red backgrounds. I decided to just train the model on the bonus dataset I created and see what kind of results I get. I decided to train the model for 20 epochs, but terminated it early as the model had converged for the val dataset.

```
Training Epoch 15: 100%|██████████| 11923/11923 [15:45<00:00, 12.61it/s]
Evaluating: 100%|██████████| 2287/2287 [01:59<00:00, 19.07it/s]
Epoch [15/20] | Train Loss: 0.8209, Train WER: 0.4489 | Val Loss: 1.2327, Val WER: 0.5888, Val Accuracy: 0.1605, Val Char Accuracy: 0.3852
Training Epoch 16: 100%|██████████| 11923/11923 [16:41<00:00, 11.91it/s]
Evaluating: 100%|██████████| 2287/2287 [02:01<00:00, 18.77it/s]
Epoch [16/20] | Train Loss: 0.7900, Train WER: 0.4360 | Val Loss: 1.2202, Val WER: 0.5528, Val Accuracy: 0.1732, Val Char Accuracy: 0.3894
Training Epoch 17: 100%|██████████| 11923/11923 [16:52<00:00, 11.78it/s]
Evaluating: 100%|██████████| 2287/2287 [02:02<00:00, 18.63it/s]
Epoch [17/20] | Train Loss: 0.7611, Train WER: 0.4214 | Val Loss: 1.2771, Val WER: 0.5929, Val Accuracy: 0.1647, Val Char Accuracy: 0.3860
Training Epoch 18: 100%|██████████| 11923/11923 [16:53<00:00, 11.77it/s]
Evaluating: 100%|██████████| 2287/2287 [02:00<00:00, 19.03it/s]
Epoch [18/20] | Train Loss: 0.7353, Train WER: 0.4116 | Val Loss: 1.2676, Val WER: 0.5681, Val Accuracy: 0.1686, Val Char Accuracy: 0.3880
Training Epoch 19: 100%|██████████| 11923/11923 [16:54<00:00, 11.75it/s]
Evaluating: 100%|██████████| 2287/2287 [02:00<00:00, 18.93it/s]
```

Bonus Task | Evaluation

I felt the best way to quantify these results was to compare them to how the Task 2 transformer model performed on the test dataset and obviously how a random baseline would perform.

```
100%|██████████| 1154/1154 [00:22<00:00, 50.19it/s]
Random Loss: 3.8413, Random WER: 1.1334, Random Accuracy: 0.0042, Random Char Accuracy: 0.0344
```

Random baseline

```
Evaluating: 100%|██████████| 1154/1154 [00:46<00:00, 24.69it/s]
Test Loss: 3.5688, Test WER: 2.4777, Test Accuracy: 0.0000, Test Char Accuracy: 0.0683
```

Task 2 Transformer

```
Evaluating: 100%|██████████| 1154/1154 [00:38<00:00, 30.09it/s]
Test Loss: 1.0903, Test WER: 0.5290, Test Accuracy: 0.1785, Test Char Accuracy: 0.3929
```

Bonus Transformer

Bonus Task | Conclusion

Clearly, the architecture is capable of learning that green images must be read in forward, and that red images must be read in reverse. This is inferred off the fact that the newly trained model does much much better than the transformer trained on the task 2 dataset, which surprisingly got a horrible score.

I decided that this task was completed, because any attempts to improve the results would require much more data and compute (something I didn't have) or required time to research and implement a new architecture (also something I didn't have).

Conclusion

In this project, I explored Captcha Recognition using Computer Vision, navigating through dataset creation, classification, and sequence generation. The journey involved overcoming challenges with dataset design, refining model architectures, and experimenting with different approaches to achieve optimal performance.

For classification, I achieved a strong 93.26% accuracy using a CNN-based model. For generation, an Attention-based Encoder-Decoder model initially showed promising results but struggled with generalization. After experimenting with dataset variations and architectural choices, I switched to a Transformer-based approach, which demonstrated improvements but introduced new challenges, such as lack of sequential processing.

Conclusion

Overall, this task helped me learn a lot of new concepts ranging from the basics such as RNNs all the way up to Transformers. I really enjoyed the experimentation process, playing around with hyperparameters for the current architecture and trying to learn about other architectures while the model trained. I'm looking forward to working on this task later in my free time, hopefully I'll be able to get better results ☺.

Going ahead, my main steps are going to be:

- Refining the transformer architecture, look at attention maps, try to understand why the model is converging so fast.
- Experimenting with more basic architectures, like CRNNs.