Precog Task Report

Tasks done

- **Task 0** - Dataset Generation
  Dataset generation with Pillow and numpy
- **Task 1** - Classifying 100 words
  Used CNN
- **Task 2** - Getting the text in easy and hard images
  Used CRNN
- **Task 3-(Bonus)** - Getting the text in bonus images
  Used CRNN with attention instead of CTC. Implemented attention, **unable to finish implementing decoder.** Will be completed in the future.

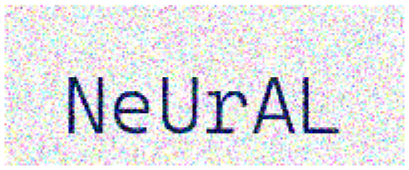
Task 0 - The Dataset

Objective was to create a dataset of (input = image, output = text)

Created a comprehensive dataset generation pipeline capable of producing three types of CAPTCHA images:

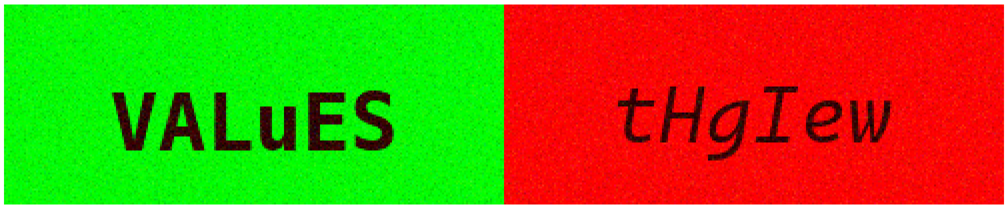
- **Easy Images (`easy_imgs.py`)**:
  - Clean text with plain white background
  - Standard fonts and clear backgrounds. Generated images for multiple fonts for training later.
  - Fixed character positions

- **Hard Images (`hard_imgs.py`)**:
  - Added Gaussian noise
  - Font variations
  - Character Case variations



- **Bonus Images (`bonus_imgs.py`)**:

    - Character order (left to right if green background, right to left if red background)
    - Gaussian noise
    - Font variations



These were generated on a wordlist of 100 Deep Learning related terms. Later for training the OCR model in Task2, we shifted to generating random words and testing on the wordlist generated dataset for more Generalization. Also the code was modified to generated train and test sets in the later tasks.

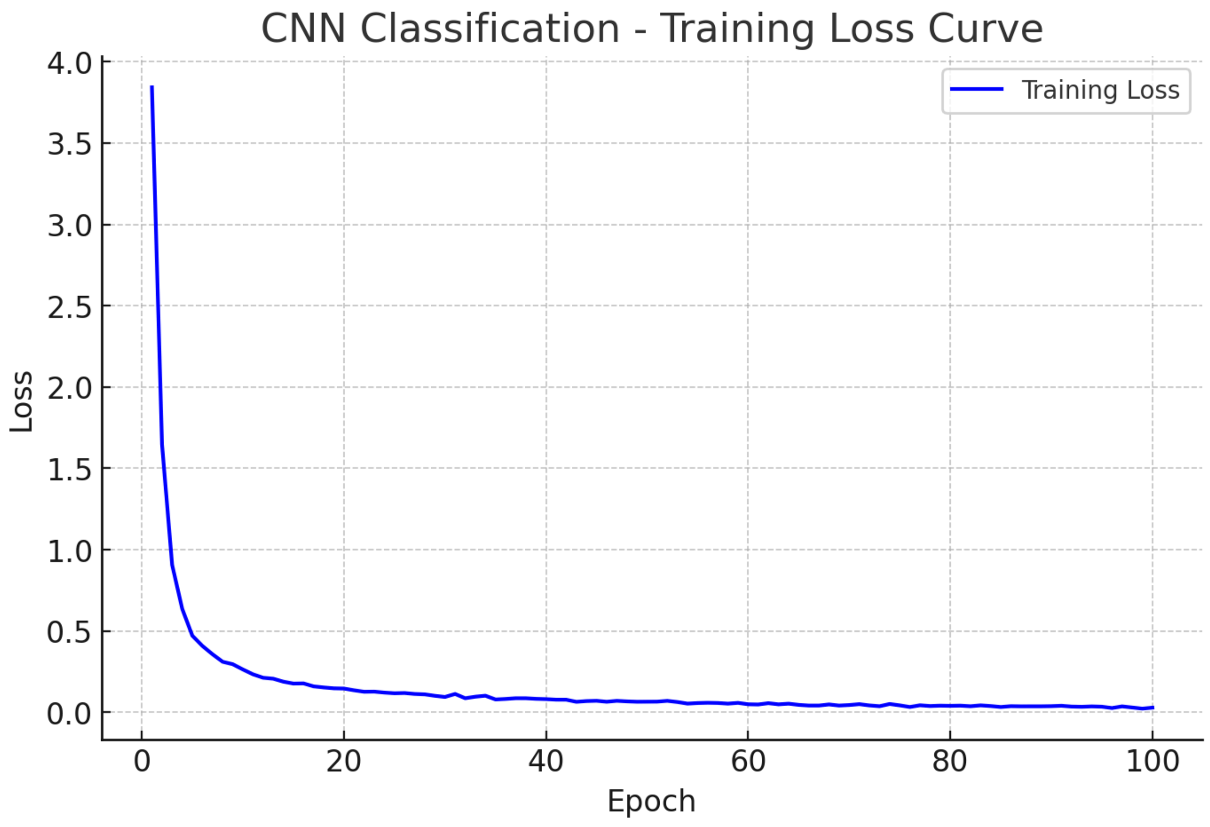
Used Pillow for the dataset generation

Task 1 - Classification

Task was given a captcha, classify which word was in the image, given there are only 100 words. This was fairly easy since only a CNN had to be trained with proper data loaders.

- Observations:
    - Tested first with a model with 6m params - overfitted badly. Made the model memorize the images rather than solve the problem
        - Due to low data. 100 classes, 5 images per class for training, 2 per class for testing
        - Too many params for too less data
    - Tested with increased data on the 6m parameter model -> again overfitting
    - Got a first working version after using 50 images per class for training with 500k params
- Training
    - Trained for 100 epochs
    - Used adam optimizer with a learning rate of 0.001
    - Batch size = 32

- Loss calculated using CEL



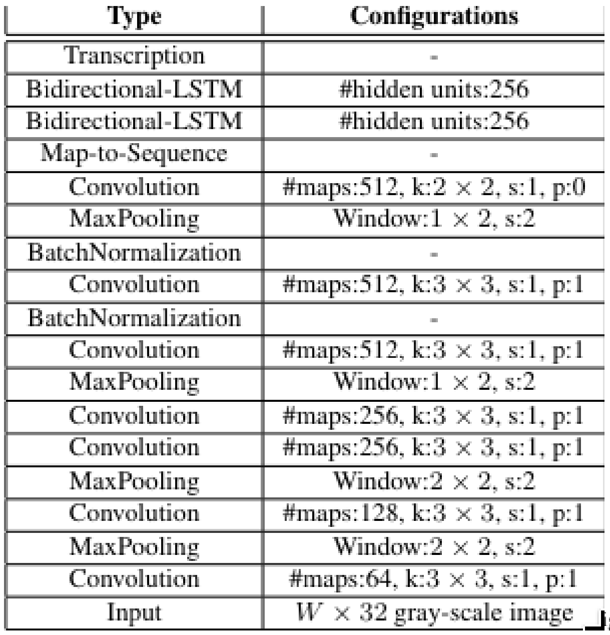
- Accuracy
    - Got 9903 / 10000 with accuracy 99.03% on the train set
    - Got 2838 / 3000 with accuracy 94.60 on the test set

Task 2 - OCR

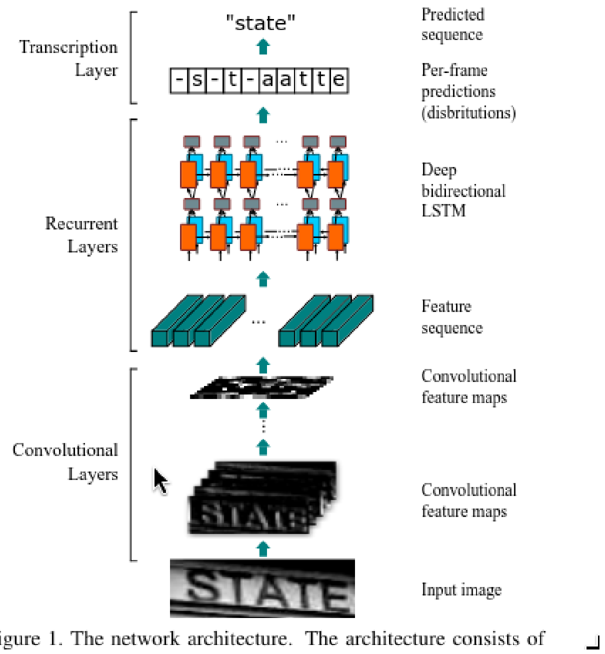
Captcha's belong not only to this 100 words, in this task I had to extract the text itself.

- A CNN can only classify images. If we were to make each word a dictionary into a class, it will be 9M classes. But still is not enough since there can be random meaningless words of any length.

- So I read this paper on CRNN. https://arxiv.org/abs/1507.05717



Type	Configurations
Transcription	-
Bidirectional-LSTM	#hidden units:256
Bidirectional-LSTM	#hidden units:256
Map-to-Sequence	-
Convolution	#maps:512, k:2 $\times$ 2, s:1, p:0
MaxPooling	Window:1 $\times$ 2, s:2
BatchNormalization	-
Convolution	#maps:512, k:3 $\times$ 3, s:1, p:1
BatchNormalization	-
Convolution	#maps:512, k:3 $\times$ 3, s:1, p:1
MaxPooling	Window:1 $\times$ 2, s:2
Convolution	#maps:256, k:3 $\times$ 3, s:1, p:1
Convolution	#maps:256, k:3 $\times$ 3, s:1, p:1
MaxPooling	Window:2 $\times$ 2, s:2
Convolution	#maps:128, k:3 $\times$ 3, s:1, p:1
MaxPooling	Window:2 $\times$ 2, s:2
Convolution	#maps:64, k:3 $\times$ 3, s:1, p:1
Input	$W \times 32$ gray-scale image



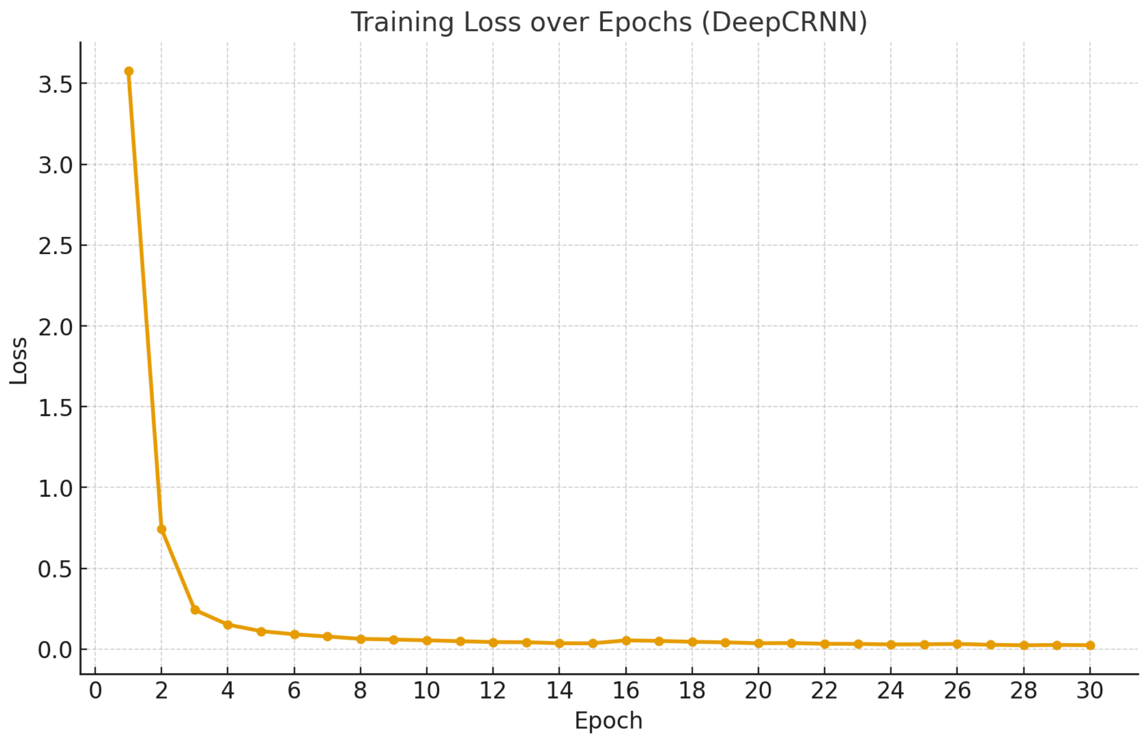
igure 1. The network architecture. The architecture consists of

DeepCRNN architecture I used

  - This paper suited our use case.
  - Like the model used in problem 1, it has a convolution layer to get the spacial features. The convolution layers here also resemble the VGG-VeryDeep architecture.
  - Then from going from left to right on the feature maps generated, a vector of feature sequences is generated. This means the i^{th} feature vector is the concatenation of the i^{th} column of all maps.
  - This feature sequence is then the input to the Bi-directional LSTM. We need to use an RNN architecture here to get the sequence since the length of the captcha text isnt constant.
  - Then Transcription, it is the process of converting per frame predictions made by the RNN into a label sequence.
  - Finally loss is calculated using CTC (Conectionist Temporal Classifications), this allows us to calculate loss for variable length outputs.
    - CTC allows the model to learn alignments between variable-length input sequences and variable-length output sequences without requiring explicit alignment during training
    - It introduces a "blank" token that represents no character, allowing the model to handle repeated characters and variable spacing
    - The CTC loss computes the probability of all possible alignments that could produce the target sequence and sums them up
    - During inference, CTC decoding (like beam search) is used to find the most likely character sequence by removing blanks and consecutive duplicates
- This approach could be made better using architectures like ViT (Vision Transformer) or Swin Transformer. But it is also computationally very expensive and requires too much data.

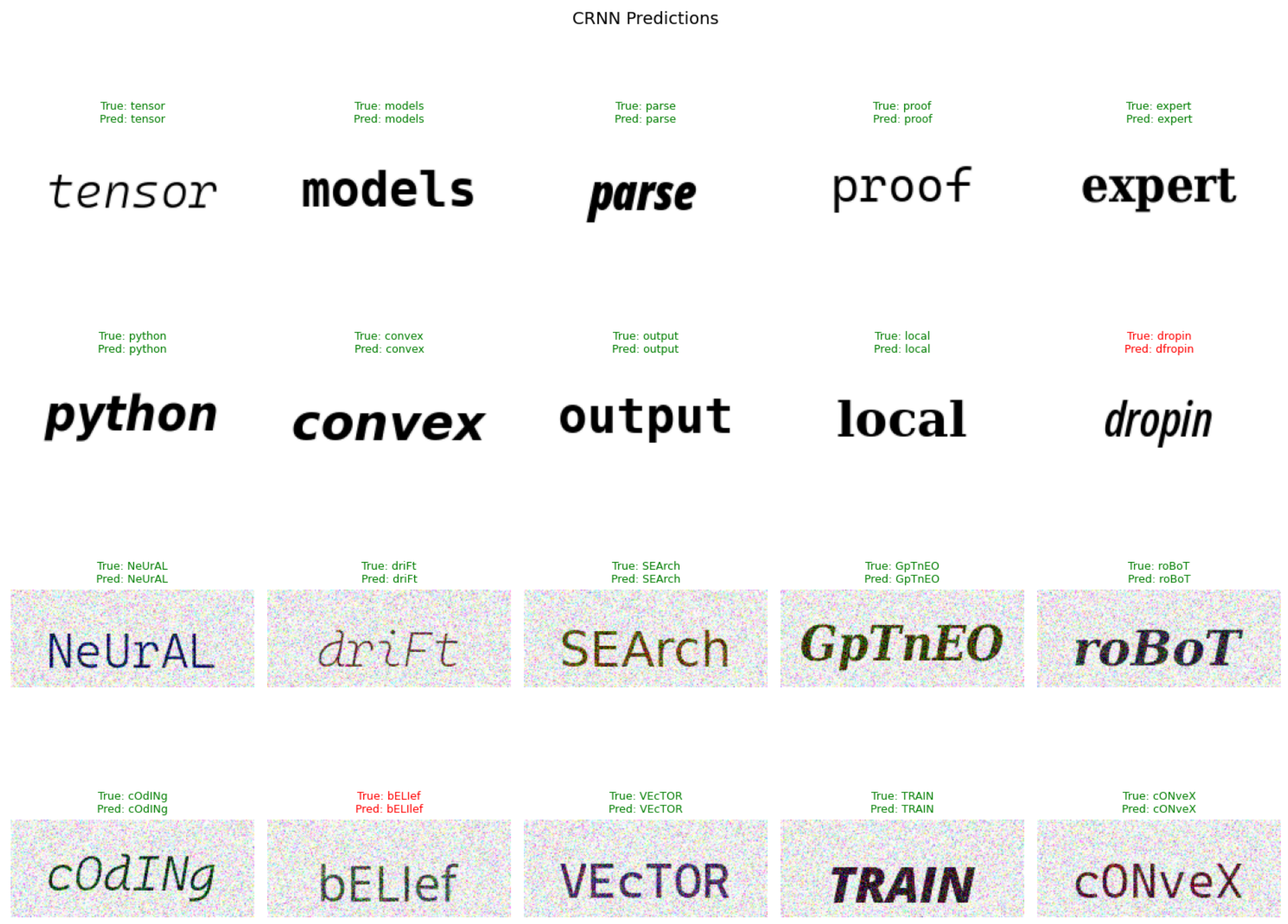
    - Note having an attention layer in general could make this much better like additive attention for CNNs. https://arxiv.org/abs/1409.0473 .This was also the paper where the concept of attention was first introduced too :)
- Another intersting approach is to embed word images and text strings in a vector space. So now the OCR problem is converted into a retrieval problem. But then again this isn't feasible since there are infinite number of possibilities of words (words can be meaningless or random).
- How the model can be made better
    - More data
    - More deeper models, that is more parameters.
    - Note: both should be together, one by itself wont work
        - Too much data on a low parameter model will not train well
        - Too less data on a large parameter model will cause it to overfit badly, as experieced before in Task 1.
    - **Input size can be reshaped like the paper before passing it onto the model. Because of this my RNN input size is `512*11` dimensions, whereas in the paper is just 512. And that's also why in the paper the input to the lstm is `512*1*width`. This doesn't have too much issues just that the training wastes too many resources.**
- Articles Reffered too
    - https://medium.com/@piyushkashyap045/understanding-pytorch-autograd-a-complete-guide-for-deep-learning-practitioners-f5dd1f43b417
    - https://pub.towardsai.net/optical-character-recognition-ocr-with-cnn-lstm-attention-seq2seq-538a57404de3
    - https://medium.com/@anishnama20/understanding-bidirectional-lstm-for-sequential-data-processing-b83d6283befc
- Observations
    - batch size 32. Tried with 8, not much improvement. Decided to stick to 32 since others took too much time to train
    - after training on random generated words (50k images), now validating with the word list dataset
        - smolCRNN trained on the Wordlist_dataset itself
            - 100% accuracy on train
            - 96% accuracy on test
            - most likely overfitted. So generated random datasets and tested again.
        - smolCRNN gets
            - Accuracy: 93.19% (46595/50000)
            - Accuracy: 84.04% (12606/15000)

        - Wordlist_captcha dataset (testing only, trained on the general random dataset)
            - Accuracy: 74.17% (6379/8600) - wordlist train
            - Accuracy: 76.12% (1964/2580) - wordlist train
            - Increasing epochs from 15 to 30 did nothing.
            - The low accuracy also means the smolCRNN model previously overfitted.
    - DeepCRNN gets (19M params)



        - trained for 15 epochs
            - 90% accuracy on train set
            - 89% accuracy on test set
            - Testing on Wordlist_captcha dataset (testing only, trained on the general random dataset)
                - Accuracy: 91.60% (7880/8600) - wordlist train part
                - Accuracy: 92.44% (2383/2580) - wordlist test part
        - trained for 30 epochs
            - Accuracy: 96.12% (48060/50000) - train
            - Accuracy: 94.39% (14158/15000) - test
            - Testing on Wordlist_captcha dataset (testing only, trained on the general random dataset)
                - Accuracy: 95.20% (8187/8600) - wordlist train part
                - Accuracy: 95.54% (2465/2580) - wordlist test part

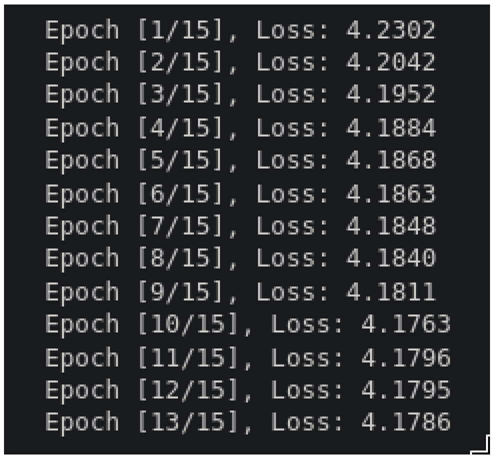
    - Initially I noticed the wordlist accuracy was low. This was because **I accidently didn't generate random worded captcha's with numbers** and the wordlist dataset had words with numbers like "AIword1" to "AIword10". So I saw only 80% accuracy. Removing the numbers increased accuracy to 95%. Didn't retrain as it would take long, but it will surely work on numbers if needed.
- Areas of Improvement:
    - The model can be made more parameter efficient. The DeepCRNN model was implemented as it is from the paper to test accuracy. It has too large parameters.
    - Accuracy could be made higher by reducing batch_size and training for more epochs.
    - Forgot to add numbers to the train dataset, maybe could try that later. Training takes hours.
    - Solution:
        - If time permits, Implement a model between DeepCRNN and smolCRNN with batch_size 8 or 4 for 100 epochs and then test the accuracy.
- Inference



- Training
    - Trained on Kaggle for 30 epochs with a batch size of 32 on P100 16gb GPU.

Task 3 - Bonus

So we cant use the DeepCRNN model (CRNN + CTC) from Task2 here since ctc assumes left to right order.
I tried training it, but then the loss was flatlining.



```
Epoch [1/15], Loss: 4.2302
Epoch [2/15], Loss: 4.2042
Epoch [3/15], Loss: 4.1952
Epoch [4/15], Loss: 4.1884
Epoch [5/15], Loss: 4.1868
Epoch [6/15], Loss: 4.1863
Epoch [7/15], Loss: 4.1848
Epoch [8/15], Loss: 4.1840
Epoch [9/15], Loss: 4.1811
Epoch [10/15], Loss: 4.1763
Epoch [11/15], Loss: 4.1796
Epoch [12/15], Loss: 4.1795
Epoch [13/15], Loss: 4.1786
```

CTC will try to fit, squish the feature sequence but wont be able to reverse it. Hence may work partially but not acceptable accuracy.

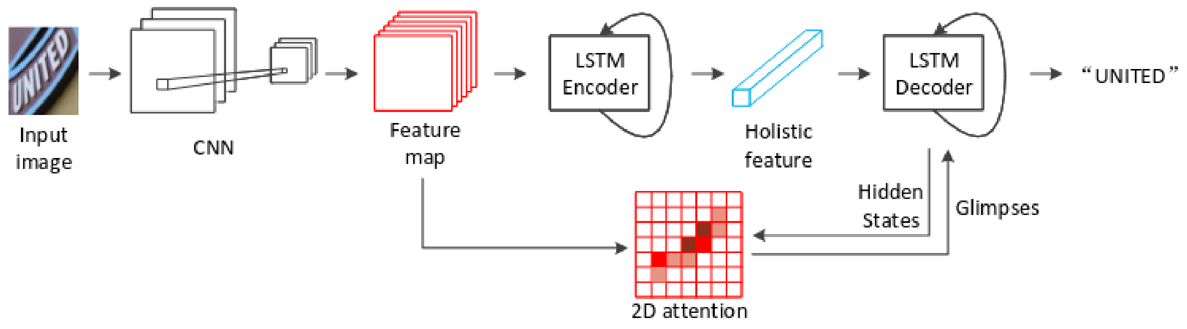
So we need a seq2seq model. or anything that makes use of attention
Attention does not assume monotonicity → it can learn to read left→right (green) or right→left (red).
So I'm thinking about replacing it with a decoder maybe and definitely resizing the input. Basically CRNN will be the encoder.

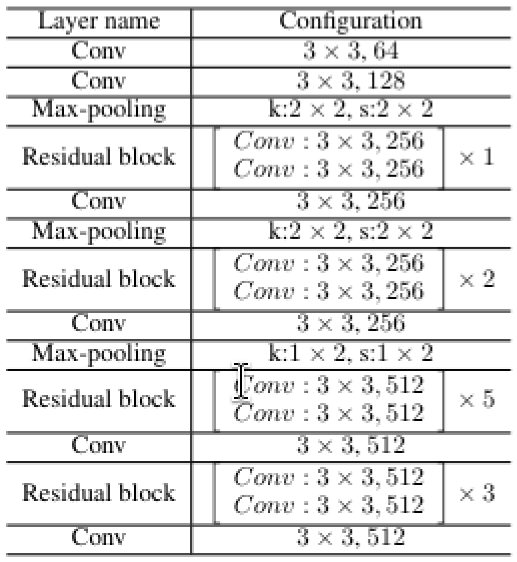
This is where I realize I have re-invented a seq2seq model

We can follow the

- Show, Attend and Tell paper : https://arxiv.org/abs/1502.03044
- Show, Attend and Read paper : https://arxiv.org/abs/1811.00751

Architecture





Layer name	Configuration	
Conv	$3 \times 3, 64$	
Conv	$3 \times 3, 128$	
Max-pooling	k:2×2, s:2×2	
Residual block	$Conv: 3 \times 3, 256$ $Conv: 3 \times 3, 256$	$\times 1$
Conv	$3 \times 3, 256$	
Max-pooling	k:2×2, s:2×2	
Residual block	$Conv: 3 \times 3, 256$ $Conv: 3 \times 3, 256$	$\times 2$
Conv	$3 \times 3, 256$	
Max-pooling	k:1×2, s:1×2	
Residual block	$Conv: 3 \times 3, 512$ $Conv: 3 \times 3, 512$	$\times 5$
Conv	$3 \times 3, 512$	
Residual block	$Conv: 3 \times 3, 512$ $Conv: 3 \times 3, 512$	$\times 3$
Conv	$3 \times 3, 512$	

Encoder

- The encoder will be A CNN which gets the feature maps. This helps get the spacial features of the image.
    - Unlike the mistake in Task2, we scale the image to `32x64` before passing it onto the CNN.
    - The CNN is build following the VGG architecture.
    - The output is then passed through a Bi-directional LSTM similar to CRNN to get the sequential text information.
    - Produces $a = \{a_1, a_2, \dots, a_L\}, \quad a_i \in \mathbb{R}^D$

Additive attention

- We use additive attention which was introduced by `Dzmitry Bahdanau`.
- How it works is, using the feature map, we flatten the feature map. $(b, c, h, w) -> (w, b, c * h)$. The width is considered the time index while passing this sequence to an RNN.

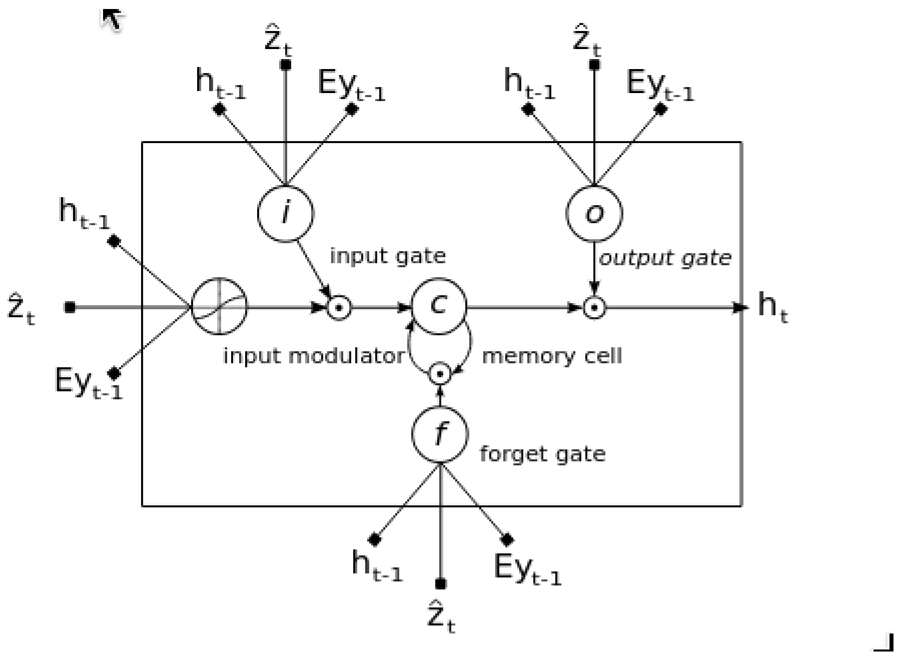
- For each location i, the mechanism generates a positive weight α_i, which can be interpreted as the probability that location i is the correct place to focus on for producing the next word (**stochastic attention mechanism**).

The weight α_i of each annotation vector is computed by an attention model f_{att} for which we use a multilayer perceptron conditioned on the previous hidden state h_{t-1}.
$e_{t,i} = v^\top \tanh\left(W_a a_i + W_h h_{t-1} + b\right)$
$e_{t,i} = f_{\text{att}}(a_{t-1}, h_i)$

- There is a MLP layer which makes the attention value for each width index.
- This is then normalized using softmax to get a probability distribution
  $\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{L} \exp(e_{t,k})}$
- Then a context vector is created by multiplying the attention score at each width to get a `width x 1` dimension vector.
  $z_t = \sum_{i=1}^{L} \alpha_{t,i} h_i$

Decoder

- This context vector either concatenated or added to the input feature vector from the CNN and then passed into a LSTM for decoding. The input to the LSTM is
  $x_t = [Embed(y_t - 1); z_t]$
  i.e., embedding of the previous character concatenated with the context vector.



The initial memory state and hidden state of the LSTM are predicted by an average of the annotation vectors fed through two separate MLPs (init,c and init,h):
$c_0 = f_{\text{init},c}\left(\frac{1}{L}\sum_{i=1}^{L} a_i\right)$
$h_0 = f_{\text{init},h}\left(\frac{1}{L}\sum_{i=1}^{L} a_i\right)$

Training

- Training is with **cross-entropy loss** over predicted vs ground truth characters.
- Teacher forcing is typically used (feeding ground-truth previous char during training).
- Evaluation with CER/WER (character/word error rate).