Senior Project - Final Report

Batbold Gankhuyag

1 Introduction

Image analysis, also known as image recognition, is the process of extracting useful information from a digital image. It has a wide range of applications such as finding shapes, identifying an object in an image, or something even more specific like identifying a person from their face.

Optical Character Recognition (OCR) is a form of image analysis where an image of a text is captured and it extracts the text that appears in the given image to convert it to an editable digital text. By doing so, we do not have to retype the whole text but just use an OCR model to extract the text.

OCR models have been around for a while, implemented in various fields. It can be used to turn a printed document into an editable digital document or convert documents into text that can be read aloud to the users. There exist OCR applications for many languages. However, there is little to no research on OCR for Mongolian Script. Mongolian Script is the traditional alphabet that was used in the Mongolian language a long time ago, which is why there are not many resources to learn it or understand old documents. By creating an OCR for Mongolian Script, we can simply take an image of the document and convert it to the modern alphabet, Cyrillic, to make it easier to understand the language.

2 Problem

How accurate can a machine learning program based on a CRNN model used to recognize images of Mongolian Script be?

3 Literature Review

CRNN models have been implemented for image analysis several times in the past. CRNN works by implementing three layers, the convolutional layers, recurrent layers, and the transcription layers [5]. The convolutional layers are used to extract features of the image, the recurrent layers predict the sequence of the features, and the transcription layers obtain the final result [6].

One research implemented CRNN with a slight improvement to recognize images of texts at a football match [5]. The improvement that they proposed was to add Max-Feature-Maps (MFM) layers into the CRNN model. They integrated the Long Short-Term Memory (LSTM) structure to tackle the problem of gradient disappearing. They created their model using Tensorflow and their input data was in the tfrecord format [5]. Another research also used CRNN to create a text recognition model used to recognize text in the natural scene [6]. The recurrent layers in their model also used bidirectional LSTM, and their model was also built using Tensorflow [6]. CRNN has also been used in the medical field to convert paper documents on patients to digital data [10].

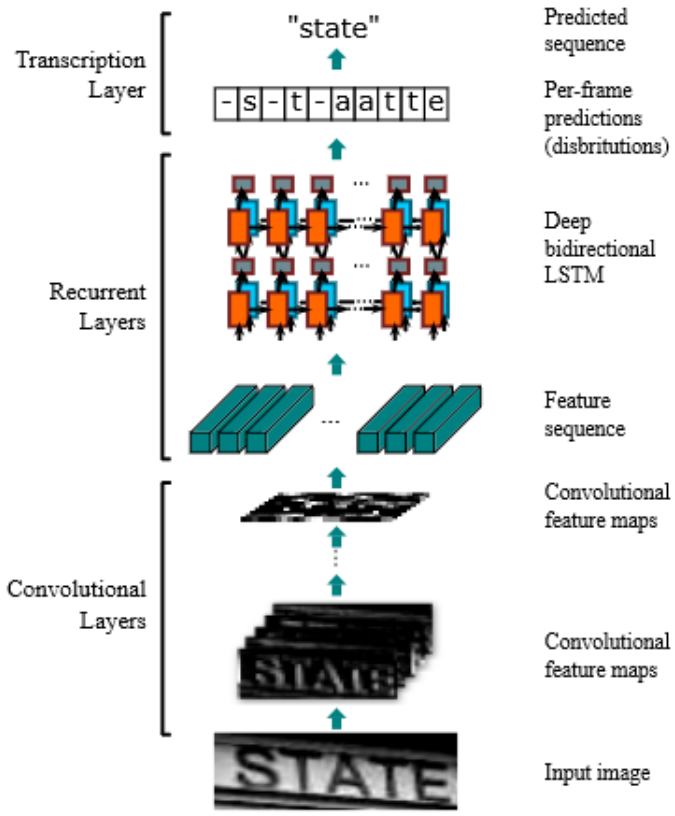
Text recognition is not the only field that CRNN has been used for. They have also been used to recognize musical notes on an image [8]. The model created was similar to previous models where it consists of the convolutional layers and recurrent layers. However, in this research, they used Recurrent Residual Convolutional block to enhance the ability of the model to enrich the context information [8].

There is also another method to create a text recognition model and that by using Wavelet Transformations (WT) [3]. This research used Stationary Wavelet Transformation (SWT) to create an Arabic word recognition model. There is also another form of WT, which is Discrete Wavelet Transformation (DWT) but the reason they used SWT was to compensate for the absence of translation invariance in the DWT. The first step of the process was to normalize the data to create somewhat uniform texts. The next step was to extract the features in the image using SWT, and the final step was to classify the features using various classifiers [3].

4 Technical Material

For this project, I used a Convolutional Recurrent Neural Network (CRNN) model to create an OCR for Mongolian Script. CRNN is composed of the network architectures Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). A CRNN model consists of three parts, convolutional layers, recurrent layers, and transcription layers. The convolutional layers extract a

feature sequence from the images input and the recurrent layers make predictions for the sequence. The transcription layers convert the predictions into their corresponding labels.



When extracting the feature sequence from an image, the CRNN model uses the convolutional layers from a standard CNN model, which is used as the input for the recurrent layers. The recurrent layers are built using RNN. The advantages of using an RNN is that it utilizes contextual information within a sequence, which is more stable and helpful than treating each symbol independently, and an RNN can back-propagate error differentials to its input, and it can operate on sequences of any length.

However, the traditional RNN unit can suffer from the vanishing gradient problem, which limits the range of context it can store. The vanishing gradient problem is when all of the neurons that came before each neuron contributes to the input for that neuron, so, you have to propagate back to these neurons to get context, which slows down the learning process. Long-Short Term Memory (LSTM) network, which consists of a memory cell and three multiplicative gates,

addresses this exact problem. The gates of an LSTM network are the forget gate, input gate and output gate. The forget gate controls what information in the cell to forget upon receiving new information that entered the network. The input gate controls what new information to encode into the cell given new information. The output gate controls what information from the cell is sent to the next network as input. Traditionally, LSTMs are unidirectional, which means that the sequences are read in a left-to-right or right-to-left fashion. However, context from both directions is useful when creating a prediction, so LSTMs can be combined to make it bidirectional, where one reads in a left-to-right fashion and the other reads in a right-to-left fashion. We can also create deep bidirectional LSTM where more than one layer of bidirectional LSTM is used. I implemented implementing deep bidirectional LSTM for this project.

After the predictions are made, the transcription process is left. When transcribing, we find the label sequence with the highest probability. There are two types of transcription, lexicon-free and lexicon-based. Lexicon is the vocabulary of a language or a set of units, and the lexicon-based approach uses the lexicon to label the sequence. I will be using the lexicon-based approach for my project since there is a set of alphabets that the text on an image can be transcribed to.

5 Methodology

To create a CRNN model, I used the Keras functional API in Tensorflow. The functional API is a way for us to create models, allowing us to handle models with shared layers and even multiple inputs or outputs. Deep learning models are usually directed acyclic graphs (DAG) of layers, and the functional API allows us to build this "graph of layers". Models are defined by creating nodes of layers and connecting them directly to each other in pairs to form a graph.

If we go through a simple example, we start off by creating an input node/layer:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(784,))
```

In this case, the input data is a 784-dimensional vector, and we omit the batch size since we only specify the shape of each sample. For example, if we had images of size $(32, 32, 3)$, we would have used:

```
inputs = keras.Input(shape=(32, 32, 3))
```

The `inputs` that is returned contains information about the shape and `dtype` of the input data that we will feed to our model. Then, we create a new node in the graph of layers by calling a layer on this `inputs` object:

```
dense = layers.Dense(64, activation="relu")
x = dense(inputs)
```

The "layer call" action is like drawing an arrow from "inputs" to this layer we just created in the "graph". Essentially, we're "passing" the `inputs` to the dense layer, and we get x as the output. If we add a few more layers to the graph of layers:

```
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10)(x)
```

Similar to our previous step, we're now "passing" x to the new dense layer and using the output of that, x, to "pass" to our last dense layer to get the output.

At this point, we can create a Model by specifying its inputs and outputs in the graph of layer:

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

We can view the summary of the model that we have created along with the output shape of each layer:

```
model.summary()
```

```
_________________________________________________
 Layer (type)                Output Shape
=================================================
 input_2 (InputLayer)        [(None, 784)]

 dense_3 (Dense)             (None, 64)

 dense_4 (Dense)             (None, 64)

 dense_5 (Dense)             (None, 10)
```

Now that we the model, we need to train and evaluate it on a dataset. Suppose we have the training set *x_train* and *y_train*, and the test set *x_test* and *y_test*. Then, we will fit the model on the training set (while monitoring performance on a validation split), then evaluate the model on the test set:

```
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=keras.optimizers.Adam(),
    metrics=["accuracy"]
)

history = model.fit(
    x_train,
    y_train,
    batch_size=64,
    epochs=2,
    validation_split=0.2
)

test_scores = model.evaluate(x_test, y_test)
```

The optimizer function of a model is used to modify the attributes of a neural network, such as weights and learning rate, and it helps reduce the overall loss of a model. For my models, I implemented the Adam algorithm for my optimizer. The Adam optimizer, instead of adapting the learning rates based on the average first moment, the mean, Adam also makes use of the average of the second moment of the gradients, the uncentered variance. A gradient is the derivative of a function and it allows us to predict the effects of a small change [4]. Adam is the combination of two gradient descent methodologies, momentum and Root Mean Square Propagation (RMSP). The Momentum optimization algorithm works by accelerating the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients. RMSP algorithm is an improved version of the AdaGrad algorithm where instead of taking the cumulative sum of squared gradients like in AdaGrad, it takes the 'exponential moving average'. Adam combines the two optimizers and builds upon them to give more optimized gradient descent [1].

The loss function of a model is used to compute the distance between the current output of the algorithm and the expected output of the algorithm. It is used as a measurement of how well a model is performing and we can use this number to adjust our algorithm to improve it. If a model is perfect, the loss will be zero, otherwise, the loss will be a higher number. For my models, I used the Connectionist Temporal Classification (CTC) loss function. CTC loss is used in deep neural networks when we don't know how the input aligns with the output. For example, if the input are the pixels in an image, then CTC can be used to determine how the pixels align to form an output [7]. CTC was also used in some of the literature that I found.

We can then add metrics to our models to be evaluated during training and testing. The most common metric that is added to models is the accuracy metric provided by keras [2]. However, for my models, I created a custom accuracy metric instead of the one provided by keras. Instead of checking for word accuracy, it checks for character accuracy. To create a custom metric, you have to override the *update_state*(), *result*(), and *reset_state*() functions. The *update_state*() function does all the calculations and updates the state variables. The *result*() function returns the value for the metric. The *reset_state*() function resets the state values to a predefined constant at the start of each epoch [9].

The epoch is the total number of times the algorithm runs on the entire dataset. The batch size is the number of samples from the dataset that is utilized in each iteration of the algorithm [2].

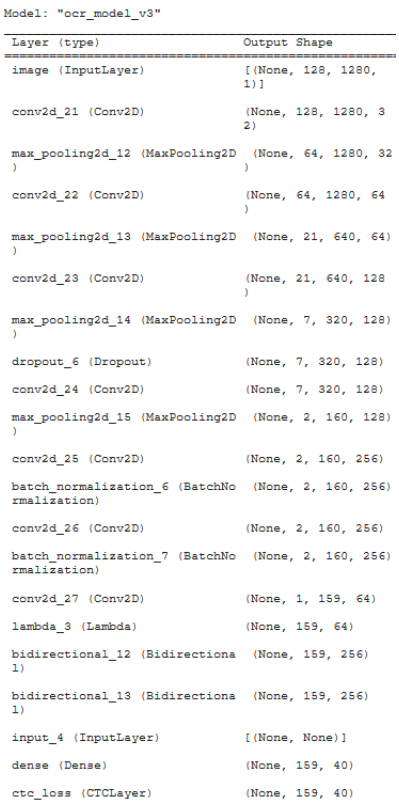
Following the example, I have created the models shown in figure 1 and figure 2. The max-pooling layers in each model is used to downsize the input along its width and height by taking the maximum value over a filter. The filter is shifted across both dimensions by the strides provided for the layer. The output of a max-pooling layer will contain the most prominent features of the previous feature map [2]. The dropout layers randomly set input units to 0, which reduces the number of interconnecting neurons within a neural network and this helps prevent overfitting [2].

```
Model: "ocr_model_v1"
_________________________________________________________________
 Layer (type)                Output Shape
=================================================================
 image (InputLayer)          [(None, 128, 1280,
                              1)]

 Conv1 (Conv2D)              (None, 128, 1280, 6
                              4)

 pool1 (MaxPooling2D)        (None, 64, 640, 64)

 Conv2 (Conv2D)              (None, 64, 640, 64)

 pool2 (MaxPooling2D)        (None, 32, 320, 64)

 reshape (Reshape)           (None, 160, 4096)

 dense1 (Dense)              (None, 160, 64)

 dropout_5 (Dropout)         (None, 160, 64)

 bidirectional_10 (Bidirectiona  (None, 160, 256)
 l)

 bidirectional_11 (Bidirectiona  (None, 160, 128)
 l)

 label (InputLayer)          [(None, None)]

 dense2 (Dense)              (None, 160, 40)

 ctc_loss (CTCLayer)         (None, 160, 40)
```

Figure 1: Model 1

```
Model: "ocr_model_v2"
_________________________________________________________________
 Layer (type)                Output Shape
=================================================================
 image (InputLayer)          [(None, 128, 1280,
                             1)]

 Conv1 (Conv2D)              (None, 128, 1280, 6
                             4)

 pool1 (MaxPooling2D)        (None, 64, 640, 64)

 Conv2 (Conv2D)              (None, 64, 640, 128
                             )

 pool2 (MaxPooling2D)        (None, 32, 320, 128
                             )

 Conv3 (Conv2D)              (None, 32, 320, 256
                             )

 Conv4 (Conv2D)              (None, 32, 320, 256
                             )

 pool3 (MaxPooling2D)        (None, 16, 320, 256
                             )

 Conv5 (Conv2D)              (None, 16, 320, 512
                             )

 norm1 (BatchNormalization)  (None, 16, 320, 512
                             )

 reshape_1 (Reshape)         (None, 320, 8192)

 bidirec1 (Bidirectional)    (None, 320, 256)

 bidirec2 (Bidirectional)    (None, 320, 128)

 label (InputLayer)          [(None, None)]

 dense2 (Dense)              (None, 320, 40)

 ctc_loss (CTCLayer)         (None, 320, 40)
```

Figure 2: Model 2



```
Model: "ocr_model_v3"
_________________________________________________________________
Layer (type)                 Output Shape
=================================================================
image (InputLayer)           [(None, 128, 1280, 1)]

conv2d_21 (Conv2D)           (None, 128, 1280, 32)

max_pooling2d_12 (MaxPooling2D)  (None, 64, 1280, 32)

conv2d_22 (Conv2D)           (None, 64, 1280, 64)

max_pooling2d_13 (MaxPooling2D)  (None, 21, 640, 64)

conv2d_23 (Conv2D)           (None, 21, 640, 128)

max_pooling2d_14 (MaxPooling2D)  (None, 7, 320, 128)

dropout_6 (Dropout)          (None, 7, 320, 128)

conv2d_24 (Conv2D)           (None, 7, 320, 128)

max_pooling2d_15 (MaxPooling2D)  (None, 2, 160, 128)

conv2d_25 (Conv2D)           (None, 2, 160, 256)

batch_normalization_6 (BatchNormalization)  (None, 2, 160, 256)

conv2d_26 (Conv2D)           (None, 2, 160, 256)

batch_normalization_7 (BatchNormalization)  (None, 2, 160, 256)

conv2d_27 (Conv2D)           (None, 1, 159, 64)

lambda_3 (Lambda)            (None, 159, 64)

bidirectional_12 (Bidirectional)  (None, 159, 256)

bidirectional_13 (Bidirectional)  (None, 159, 256)

input_4 (InputLayer)         [(None, None)]

dense (Dense)                (None, 159, 40)

ctc_loss (CTCLayer)          (None, 159, 40)
```

Figure 3: Model 3

6 Results

The training of my models was done on a dataset of around 50,000 images of dimension 1280x128 with a batch size of 128 and epochs of 3. The table below shows the result of my models:

Model	**Character Accuracy**
1	3%
2	2%
3	3%

As we can see from the table, all of the models performed very poorly on the

dataset that was provided. A possible explanation for this poor performance could be the low epoch. Perhaps if the models had been trained on the dataset a few more times, it might have produced better results. A more likely explanation is that the model was overfitting on the padding that was added to the labels. Since I am using the CTC loss function on a batch of data, all the labels had to be the same length so that the loss could be calculated for each batch element. The labels were also passed into the model to be used by the CTC loss function and the models only accept inputs of fixed length. Hence, padding was added the suffix of each label to make them all equal length. However, I believe that this may have negatively impacted the model since it was learning on the padding as well as the relevant information.

This may have not been a problem if the labels were very close in length but the dataset that I used for models have a huge variance in the length of the labels. Consequently, some of the labels were mostly padding which affected the training process. Given more time, I could have looked into using the CTC loss function on a batch of varying lengths so that the model will not take them into account.

7 Future Directions

The most obvious next step is to get the models actually working by figuring out how to use labels of varying length for the CTC loss function. This way, we can properly determine how well a CRNN model can recognize text on an image.

For my project, the images that I used for my dataset were not pictures taken with a camera but they were screen shots of typed letters. So a possible future direction is to take images of words in Mongolian Script with a camera and use them as the dataset.

Another possible future direction is to use handwritten Mongolian Script and not only typed letters. I believe that this will be more practical than my project because if it is successful, it can be used to transcribe old documents in Mongolian Script.

References

[1] Adam, Dec 2021.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving,

Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Atallah Mahmoud Al-Shatnawi, Faisal Al-Saqqar, and Alireza Souri. Arabic handwritten word recognition based on stationary wavelet transform technique using machine learning. *ACM Trans. Asian Low-Resour. Lang. Inf. Process.*, 21(3), dec 2021.

[4] Jason Brownlee. What is a gradient in machine learning?, Oct 2021.

[5] Lei Chen and Shaobin Li. Improvement research and application of text recognition algorithm based on crnn. In *Proceedings of the 2018 International Conference on Signal Processing and Machine Learning*, SPML '18, page 166–170, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Yilin Chen and Juan Yang. Research on scene text recognition algorithm basedon improved crnn. In *Proceedings of the 2020 4th International Conference on Digital Signal Processing*, ICDSP 2020, page 107–111, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Awni Hannun. Sequence modeling with ctc, Jan 2020.

[8] Aozhi Liu, Lipei Zhang, Yaqi Mei, Baoqiang Han, Zifeng Cai, Zhaohua Zhu, and Jing Xiao. Residual recurrent crnn for end-to-end optical music recognition on monophonic scores. In *Proceedings of the 2021 Workshop on Multi-Modal Pre-Training for Multimedia Understanding*, MMPT '21, page 23–27, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Derrick Mwiti. Keras metrics: Everything you need to know, Nov 2021.

[10] Luyang Zhao and Kebin Jia. Application of crnn based ocr in health records system. In *Proceedings of the 3rd International Conference on Multimedia Systems and Signal Processing*, ICMSSP '18, page 46–50, New York, NY, USA, 2018. Association for Computing Machinery.