Visual Translator 

Soumyadeep Chatterjee, *Student Member, IEEE*, Russell Kurihara, *Student Member, IEEE* and Rex Taymany, *Student Member, IEEE*

***Abstract*—**

Language Translators such as Google Translate are currently able to provide real time results with increasing accuracy. However, in many cases they fail to capture the essence and finer traits of human language, and require one to type text in a language one may not be familiar with - giving rise to errors. In this project, we aim to eliminate the step of human error by integrating Optical Character Recognition (OCR) with a 5-layer Neural Network translator. Processing of digital data, optical recognition algorithms, machine learning and natural language processing are used here to create a handheld translator that can be integrated onto both Android and iOS devices.

Index Terms—optical character recognition, time series analysis, machine learning, recurrent neural networks, natural language processing, internet of things

I. Introduction

A. History and Sources

THE foundations of machine learning were set in 1943 by Warren McCulloch and Walter Pitts, who created a model of a neuron based on an electrical circuit, creating the first ever Neural Network. Seven years later, Alan Turing used these principles to create his famous Turing Test; scientists at the time desired that their networks be able to carry out computations, which were cumbersome at the time. Although the early results were promising, research was curtailed by the hardware limitations and design ideologies choices of the time. Save the conceptualization of backpropagation at Stanford University, the prevailing Von Neumann architecture system directed designers to store both instructions and data in the same memory structure. This caused very little progress to be made in the half century since neural networks were first introduced [1].

Deviation from this architecture kicked off rapid developments in data processing and machine learning in the 21st century, owing mainly to massive increases in government funding towards such operations. Researchers were now able to visualize the immense potential of the concept, and the rush to supremacy has accelerated to the present day. Corporations such as Google have invested heavily, beginning with GoogleBrain (2012), which focused on pattern recognition in images and videos, to creating TensorFlow, a backend library for computational machine learning. Modern research has undergone a paradigm shift in many fields as researchers scramble to use decades of data to make groundbreaking observations and progress.

Machine Learning has greatly advanced as a concept, and a variety of Neural Networks have been created to accomplish an expanding variety of tasks, from breathtakingly quick computations to tuning social media interactions and online commerce. However, the true extent of this field has yet to be realized, and with our project we aim to blend these groundbreaking concepts with signal processing techniques to create a product that can be used in one's day-to-day life. This was accomplished by drawing from a swathe of research papers on Recurrent Neural Networks, Natural Language Processing, TensorFlow libraries, Google's widely used Tesseract API (open source OCR), IOT technology, and implemented using the Keras frontend in Python (version 3.6.8).

Global Constraints

The newer versions of Tesseract have been adapted to C++ language. While there do exist API's to run tesseract in a C environment, it still requires the system to have the specific path location for the tesseract program saved on the system. Also, the LCDK has limited memory and the tesseract executable file is 17MB and each language pack is 12MB, so we decided against using an LCDK for the front part of the project. Additionally, from preliminary research, it seems the only way to get tesseract working properly would be through the Linux Software Development Kit for OMAP-L138 Processors TI provides. There is a possible method to work around this by using an external USB with the tesseract and the image to convert on it, but this adds another step in the process to take a picture and upload it to the USB. Utilizing a phone with its own internal storage and camera streamlines the process and is therefore much better for our project.

II. Motivation

After our gentle introduction to machine learning in 113DA, we were all intrigued by differences and possible improvements this concept was making in eminent digital signal processing. All of the team-members had worked with signal processing and communications systems, and were interested in pursuing new developments in machine learning and artificial

Submitted for review on March 23, 2018. This work was required for the class Electrical Engineering 113D, taught by Professor Mike Briggs at the University of California, Los Angeles.

S. Chatterjee, R. Taymany and R. Kurihara are senior Electrical Engineering degree candidates at the University of California, Los Angeles, CA 90025, USA.

intelligence. After learning of the prevalence of the Python programming language in these fields, we all improved our abilities here massively, allowing us to understand and analyze modern systems and algorithms.

First, we were interested in the use of OCR in movie post-production (for effects such as reverse aging an actor), and wanted to work with this new development in image processing. Next, we realized that Google translate was not perfect – this was something we would attempt to fix. Further, we foresaw situations wherein typing another language into a translator would not be feasible; the convergence of these realizations gave rise to our project – challenging ourselves by integrating ML onto an embedded system to create a useful application for translating images of text in foreign languages.

III. Approach

In this section, we will highlight our overall plan for completion, the division of labor, and how we overcame difficulties while completing our project.

A. Team Organization

Soumyadeep Chatterjee and Rex Taymany familiarized themselves with the theory behind machine learning, natural language processing and neural networks. Soumyadeep studied the theory and rationale of neural networks to create the encoder-decoder architecture used in language processing and worked on acquiring and preparing the training data, deciding on parallel corpus data from the EU Council. Rex and Soumyadeep combined their experience to progress from here; Soumyadeep worked to design the network architecture, while Rex worked on tuning integral system parameters such as the optimizer algorithm to finish the team's 5-layer RNN to perform the translation. This was implemented using Python code. Russell Kurihara researched a number of OCR applications and implementations, including some open source projects still in development. He also implemented a push-pull token system in Dropbox to exchange files rapidly, creating an easy to use interface for the end user – combining both sides of the project.

B. Plan

Week 3: By this time, we aimed to have a complete understanding of all aspects of our project, such as what we need to code for each section of the system. The optical character recognition system would be in its first iteration, with fine-tuning on accuracy and speed as our next goals in this section. Our Machine translation NN would also be in its first iteration, and we planned to be testing it with simple text and begin fine tuning the parameters. In the weeks that follow, we would develop and improve our network, while working to integrate both parts of our system.

Week 7: We expected to have completed the optical recognition system by this time. Our NN should be able to handle translations fairly accurately; going forward from this point, we would work to further train and tune the system - a process which we would continue with until the end, as this continually improves the system's performance [2]. Further, our system at this stage would be integrated, as the Optical Character Recognition and NN translation systems would work together, allowing us to measure the system parameters such as time taken, system load etc.

Week 10: By this time, we had planned to test the full program on our testing hardware (a Samsung Galaxy S6), work out kinks in the file transfer methodology, and work to translate increasingly complex sentences and phrases.

However, we faced a variety of issues during the last 3 weeks that we spent a considerable amount of time solving. Firstly, our NN architecture was such that we had to find a very specific way to arrange the training data. In Week 8, the team was finally able to create sorting and splitting algorithms – leaving us to focus on tuning training parameters such as batch size, and the distribution of randomly selected data for epoch-by-epoch verification. We realized that projects of this magnitude demanded hardware resources far beyond our reach, and Rex worked on a trial and error system to pick the batch size and training iterations; eventually, we were able to reach a point where the translations made complete sense. In summation, Soumyadeep and Rex were eventually able to reach the team's goal, incorporating three different languages into the system.

Our OCR implementation proved to be less cumbersome. Russell did not follow the weekly milestones set here, instead focusing on researching programs that would work with our system with minimal issues, such as memory allocation. By Week 7, he had set up a file transfer system using Dropbox, but the OCR open source code needed modification for optimal performance. Additionally, he had created the elementary version of the frontend GUI – both this and the OCR were completed toward the end of Week 10, indicating that open source programs may not always be the best choice for projects such as this one.

C. Standard

Various machine learning, image processing and communications techniques were used throughout the project; when we could not find suitable standards, we created our own, as shown below.

All of our code was written in Python, the predominant language in the fields of machine learning and image processing [3]. The Neural Networks were coded in PyCharm, a widely used Python IDE, using the TensorFlow backend – the most prevalent computational engine for machine learning. Keras, a NN library, was used to utilize this backend to call a variety of library functions and data structures integral to any machine learning project. The entire RNN was built using a combination of Keras library functions and standard Python language.

The OCR was an implementation of PyTesseract, which is a Python-based wrapper of Google's widely used Tesseract-OCR Engine. In order to join the OCR with the Neural Network, we found no existing framework for transferring the image taken by the client to the NN for translation, as well as return the translated text. For this, we designed a system using dropbox tokens. The communication is done over IP (internet

protocol). The working of this standard is laid out in the figure below.

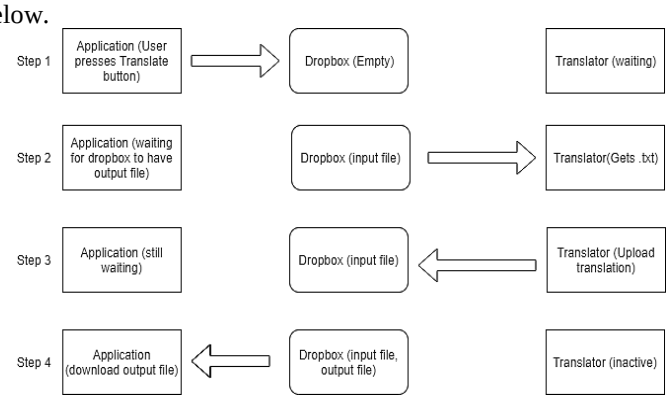


Figure 1: Communication vis-à-vis Dropbox tokens

D. Theory

Our project uses the Tesseract-OCR program to convert images to text. The program works by first taking in an image and storing the outlines of components as blobs which are then converted into text lines and analyzed for proportional text and fixed pitch. Text lines are then broken into words by analyzing both fuzzy spacing and definite spacing. These words are then analyzed for their own letter boundary boxes and unknown letters are chopped into 3-D or 4-D feature vectors which are used to compare twice with the pretrained set of an adaptive classifier. Finally, x and y normalization are used to detect upper-case vs lower-case letters, and the final text is created and saved in a specified format.

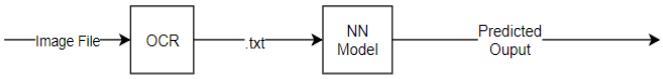


Figure 2: Overall project structure

Our project utilizes deep learning, more specifically the concept of neural networks. Our network, much like the one covered in class, consists of weights and biases that are trained on incoming data and self-updates to converge towards a better weights and biases through training data. When test data is inserted to the network, the activation function in addition to these weights and biases result to an output. The key departure in our project is the utilization of an LSTM (Long Short-Term Memory) network. LSTM is a type of recurrent neural network that attempts to model time or sequence dependent behavior, optimal for tasks such as language [4].

This type of neural network architecture allows one to update the weights and biases like so:

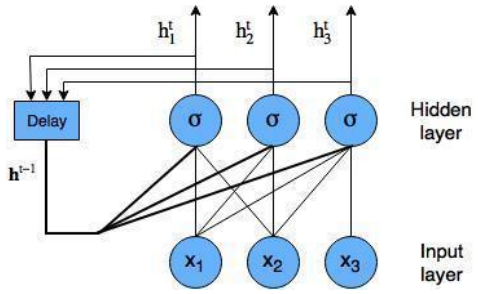


Figure 3: Recurrent neural network diagram with nodes
(Courtesy of AdventuresofMachineLearning.com)
https://adventuresinmachinelearning.com/keras-lstm-tutorial/

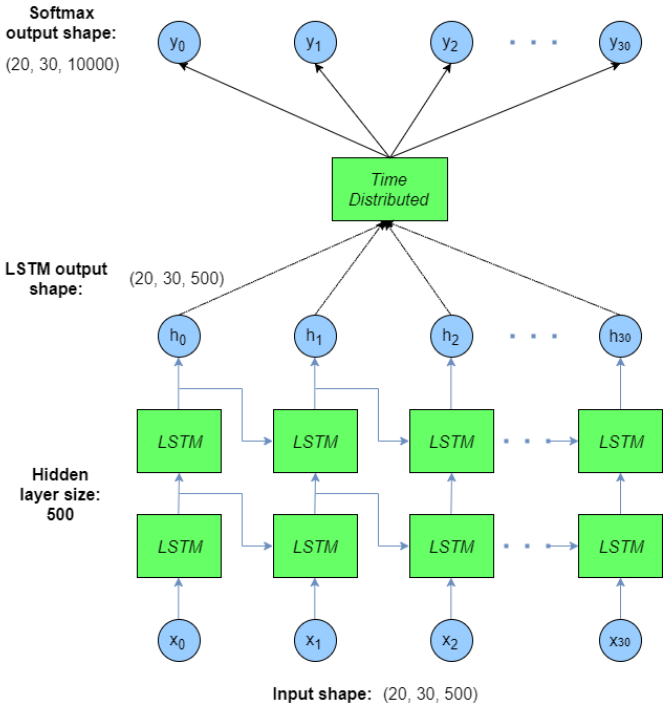
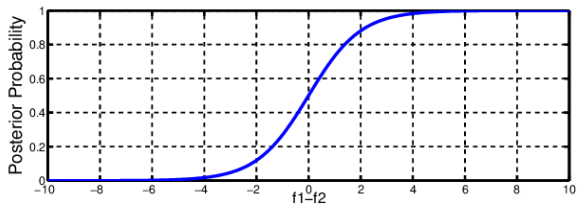


Figure 4: Keras LSTM architecture
(Courtesy of AdventuresofMachineLearning.com)
https://adventuresinmachinelearning.com/keras-lstm-tutorial/

What this network seeks to fix is the issue of the neural network vanishing gradient problem, one that is inherently amplified in our project. Since we are trying to model dependencies between words/sequences for our translator, there will be times where other words/sequences are placed in between where we want to find these dependencies. This presents a problem in traditional neural networks as layers pick up these words/sequence batches and processes them. By the time it comes to update our weights and biases, the parameters will correspond to a zero gradient, essentially saying that the parameters do not need to be updated (when they clearly should be) and thus the network fails to learn long-term dependencies. By updating these weights sequentially rather than in a batch, we minimize this effect and update the weights to correctly reflect the language model. After deciding to use the LSTM network, we utilized the Keras and Tensorflow libraries to implement our network with key choice parameters.

For our activation function, we utilized the softmax function (essentially a sigmoid activation function but extended to use in multiclass classification) [5].



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}}$$

Figure 5: Softmax Activation Function (Courtesy of Binghui Chen) https://www.researchgate.net/figure/Decomposition-of-typical-softmax-layer-in-DCNN-It-can-be-rewritten-into-three-parts_fig1_319121953

The softmax function exists only between 0 and 1; it is the ideal choice of predicting the probability as the output as it maps the non-normalized output of our network to a probability distribution. This allows us to model language much more accurately since we are interested in predicting the next sequence of words from previous sequences and states in the form of probabilities. The figure above illustrates how all values are essentially "squished" into a probability between 0 and 1 by using its equation.

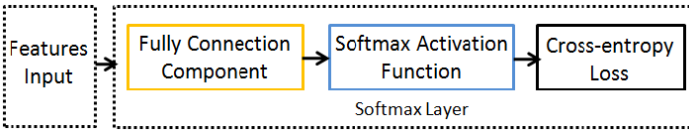
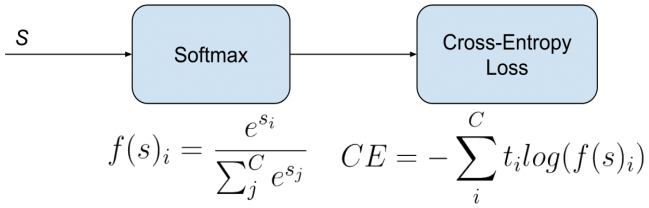


Figure 6: Decomposition of typical softmax layer in DCNN (Courtesy of Binghui Chen) https://www.researchgate.net/figure/Decomposition-of-typical-softmax-layer-in-DCNN-It-can-be-rewritten-into-three-parts_fig1_319121953

In order to have fast runtime and accurate results, we opted to use the Adam (Adaptive moment estimation) for our optimizer and categorical cross entropy for our loss function. These two work in tandem as the optimizer solves the gradient descent problem for the cost function which is made from the loss functions (e.g a cost function made from the average of all loss functions in the training set). First we will discuss our choice of optimizer and its importance in any neural network.

`In the words of Jason Brownlee from machinelearningmastery.com, "the choice of optimization algorithm for your deep learning model can mean the difference between good results in minutes, hours, and days". The Adam optimizer allows us to achieve this goal by utilizing stochastic gradient descent while combining the usefulness of AdaGrad (Adaptive Gradient Algorithm) to help with sparse gradients such as natural language and RMSProp (Root Mean Square Propagation) to help with noisy data. Both maintain a per-parameter learning rate that is adjusted on previous magnitude changes in the gradient descent, allowing for fast and accurate convergence. Secondly, categorical cross-entropy allowed us to compute loss functions in the case of multi-class classification as it is commonly used in conjunction with softmax activation functions as depicted in the figure above. The process of calculating the cross-entropy loss from a softmax calculation is shown below [6]:



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \quad CE = -\sum_i^C t_i log(f(s)_i)$$

Figure 7: Categorical Cross-Entropy loss (Courtesy of Binghui Chen) https://www.researchgate.net/figure/Decomposition-of-typical-softmax-layer-in-DCNN-It-can-be-rewritten-into-three-parts_fig1_319121953

E. Software / Hardware

For the front-end of the application we used Python as main language mainly to be consistent with the machine learning aspect of the project. Additional benefits to using Python came in the form of easy to use packages for aspects of the projects. For example, Tesseract in any other language requires using their APIs and delving through their documentation to make things work, where the python wrapper called Pytesseract lets us use a single line of code so long as tesseract is in the PATH, or otherwise needs a location specified for the tesseract executable. Python also had the benefits of the Kivy package, which allows for easy construction of GUIs through intuitive widget trees defined in a string and interpreted through Kivy's kv language builder. There is also documentation and instructions for using the buildozer library along with python-for-android to create .apk files to run on android systems, which is the final goal of our project.

For our neural network we utilized the Keras and Tensorflow libraries to train our LSTM network. These libraries provided a full range of functions that allowed us to quickly prototype test models allowing us sufficient time for parameter tuning. Tensorflow also supported GPU accelerated training which helped reduce waiting times for multiple tests. We utilized Python as the main programming language as we had access to several libraries via pip that were crucial in cleaning our data and preprocessing it for the neural network (which was also written in Python). Multidimensional arrays played a huge role in most of the programming as we had to constantly match array sizes and transfer them into proper sizes before any processing could occur. Many of these arrays are pickled using the Python pickle library to save space by serializing and deserializing them.

On the hardware side, we decided to use a desktop computer specced with an Intel i7-7700K CPU @ 4.20GHz in conjunction with a NVIDIA GeForce GTX 1080 GPU to exclusively train the neural network model. We opted not to use the LCDK simply due to the fact that the training set, pkl files, and the model itself presented too much strain on memory. A possible workaround would be to downscale the project drastically to fit on the LCDK leaving the task of porting the necessary libraries into C language. The

capabilities would be very limited on the LCDK if this downscale was implemented but it is completely possible.

F. Procedure

Our system in built upon two main parts: the OCR and the

Figure 8: Some example use cases

neural translation model. Input is fed into the OCR in the form of a picture or screenshot of a choice text. The input is then processed into a txt file that is transferred wirelessly to an off-site processing device (e.g laptop, desktop, server, etc.). The desired input is ran through the off-site translator and sends back a translated file to the recipient's device.

IV. Results

Introductory sentence. (g5 and g6 apply to entire section).

A. Description of Results

Tesseract and App Packaging

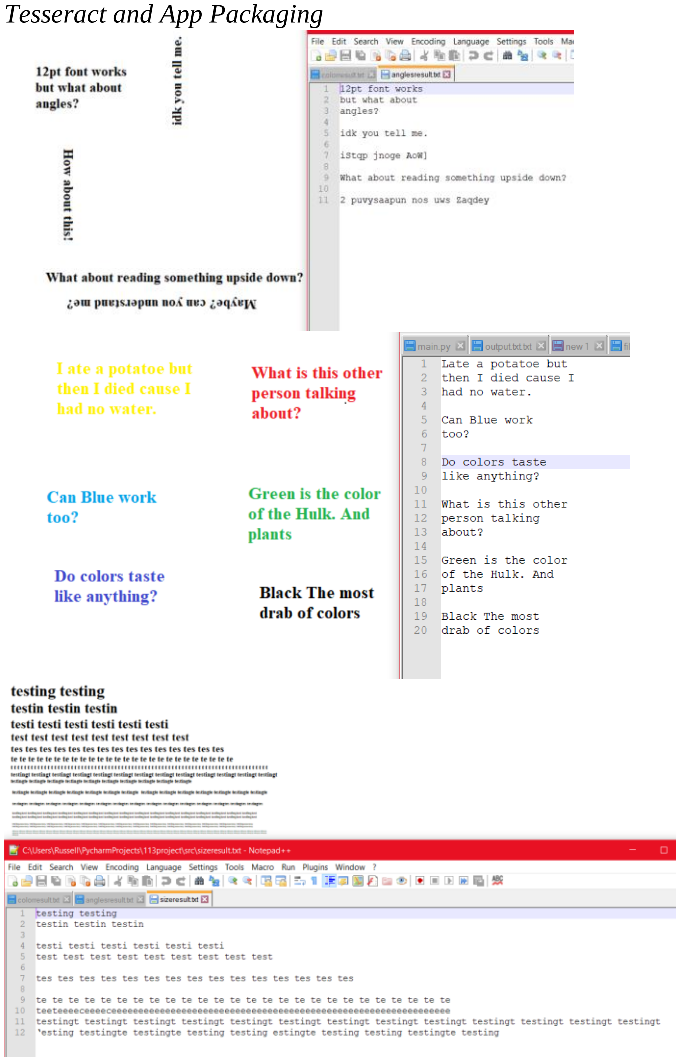


Figure 8: OCR workings

The application was created successfully and can perform the conversion from image to text and upload/download from dropbox that was planned. We also added a file selector for ease-of-use and a small txt file viewer for convenience. We did not manage to package the program into a .apk file and run it on an Android phone, mostly due to issues with the python-for-android and buildozer libraries and documentation not updated.

Translator

The neural network translator works fairly well and is able to

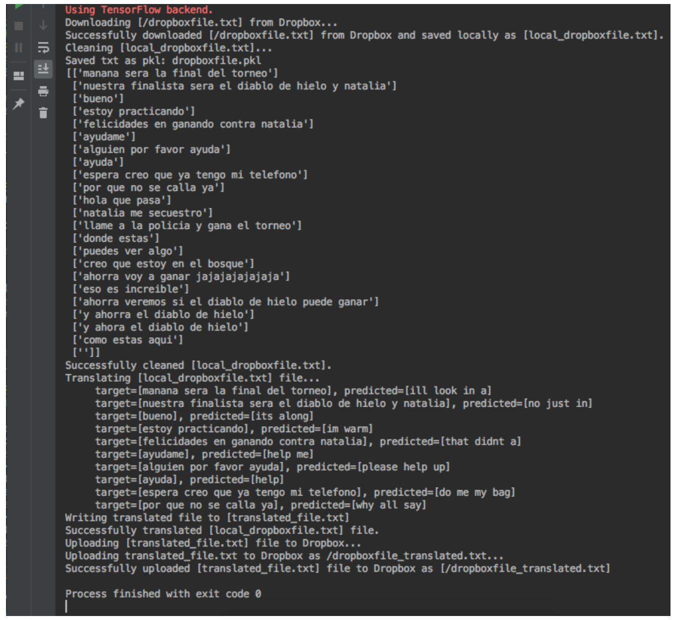


Figure 9: Example output

accurately translate most common phrases. We noticed that sentence length plays a very big role in how the system operates. A network trained mostly on short sentences or long sentences does very well on translating its respective lengths, however, when the training data is mixed with an equal proportion of long and short sentences, the result slightly underperforms in both short and long sentences. Shown here are some results of our fully trained network translating Spanish (target) to English (predicted).

B. Discussion of results

Testing and App Packaging

Testing done for the capabilities of tesseract revealed that it capable of converting text above 10 pt font accurately, however it struggles with repetition of a few letters for text smaller than this. Tesseract can handle variation in skew of to 90 degrees, but will try to read lines with more than 90 degrees of skew as if they were skewed less than 90 degrees and interpret the line upside-down. Finally, multiple colors of text can be used without a problem.

We attempted to convert the program into a .apk with the python-for-android program, however it seems that the python package is currently behind android sdk/ndk updates so something has been updated in the dependencies so that attempting to compile a .apk using either python-for-android or buildozer, the wrapper for python-for-android meant for kivy applications, resulted in small errors in the library files that prevented it from completing, which we did not have the knowledge to fix.

The actual application produces a graphical interface which does work correctly, though installation of tesseract is a requirement as we have not gotten anywhere close to an automated setup process. Uploading to a dropbox server works in both directions, though the current process keeps retrying for downloads and there is no direct communication between the two portions of the project.

10 pt font is the equivalent of 13 pixels, suggesting that the image must have a quality of at least 13 pixels per letter in order for tesseract to be mostly accurate. Considering that most cameras can take pictures of at least 480p this means that tesseract will be accurate for most photos with reasonable zoom on the text.

Translator

Examining various hyper and tuning parameters, we noticed that altering batch sizes plays a key role in training time and memory usage during runtime. The batch size parameter dictates how much of the data is trained on the network at one time. This is necessary since storing the entire dataset into the array would require an immense amount of memory and is overall not as efficient as computing them in batches. From the graphs above, we see that there is tradeoff between batch size and val_loss. In general, we want val_loss as low as possible but as batch size increases, our val_loss also increases. The tradeoff we see from through graphs is that larger batch sizes tend to take less time but converge to a higher val_loss while smaller batch sizes tend to take longer but converge to a lower val_loss.

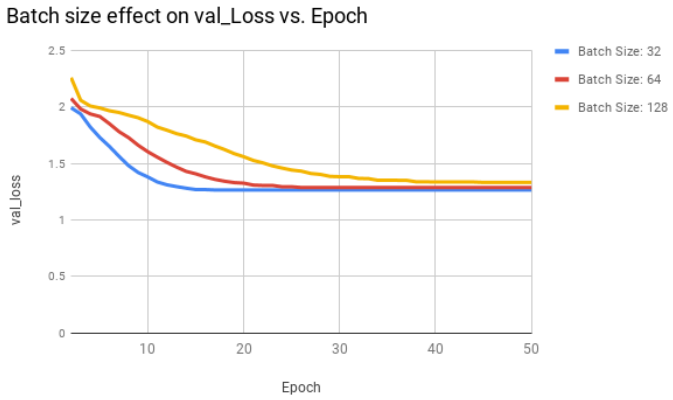


Figure 10: Batch size effects on val_loss

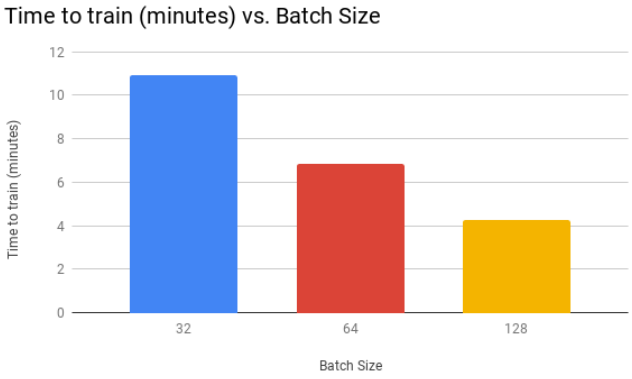


Figure 11: Batch size effect on training time

In relation to the LCDK, the role that batch sizes play is a valuable relationship since smaller batch sizes naturally correspond to smaller memory allocation during runtime thus making it more feasible to fit the program on it. Since CCS uses the C language, we can further optimize the training time by allocating byte size batch sizes to further increase memory allocation performance during runtime. Furthermore, we can exploit the fact that smaller batch sizes tend to converge much quicker than larger batch sizes. From our runs we saw that it takes ~13 seconds/epoch for a batch size of 32, ~8 seconds/epoch for a batch size of 64, and ~5 seconds/epoch for a batch size of 128. Examining the line graph above, we can see that the batch size of 32 converges at around 20 epochs. Therefore, we could essentially cut the training time from 655 seconds (~11 minutes) down to about 260 seconds (~4.3 minutes) by cutting our epochs back from 50 to 20. In essence we have just optimized a system that uses low memory allocation at runtime with low val_loss and short training time which is a step towards fitting such a massive program onto the LCDK.

V. YouTube Link

https://youtu.be/I0lPIoIyj3g,

please navigate to this in case of issues:

https://www.youtube.com/watch?v=I0lPIoIyj3g&fbclid=IwAR3YfViLn6szfKotahoHwNfm-7lKPblBdgMFdSUo1DkvvSnH27oAFzvrRR4

References

[1] Mayo, h., punchihewa, h., emile, j., & morrison, j. (2018). History of machine learning. Retrieved from https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html

[2] Introduction to Artificial Neural Networks. (n.d.). *University of Nevada, Reno*, 2-9.

[3] Top 5 best Programming Languages for Artificial Intelligence field. (2018, October 15). Retrieved from https://www.geeksforgeeks.org/top-5-best-programming-languages-for-artificial-intelligence-field/

[4] Keras LSTM tutorial - How to easily build a powerful deep learning language model. (2018, February 03). Retrieved from

[5] 1395550283894582. (2017, September 06). Activation Functions in Neural Networks. Retrieved from

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[6] R. (2018, May 23). Retrieved January 12, 2019, from https://gombru.github.io/2018/05/23/cross_entropy_loss/