Nicholas Marcopoli and Chan Hee Song
Natural Language Processing
Professor David Chiang
December 20, 2019

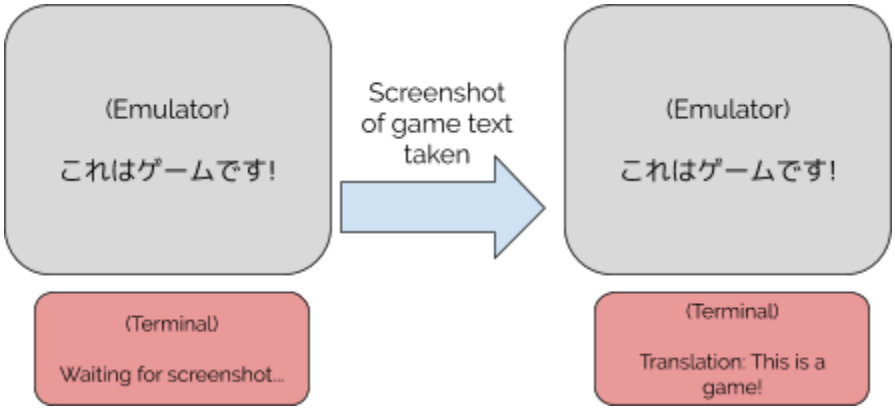
Final Report

Goal

Our project idea is to translate text from Japanese video games into English. The goal of our project has changed quite a bit from our original project idea, especially in terms of how users will interact with it. Our original vision for the project was for a user to play a Japanese game through an emulator on their computer, and he or she could select an option to have the text on screen translated from Japanese into English in real time. The diagram below illustrates our original idea:

Before Translation After Translation

While working through the project, we realized that it's very hard to actually change the text on the screen like in the photo above, even if you have a correctly translated sentence. We also thought that it wasn't quite in the scope of this course since that isn't doing any NLP. We decided instead to have the user keep a terminal open below the emulator. Instead of automatically translating the text on screen and displaying it on screen, we would require the user to take a screenshot of the Japanese text they'd like translated. The translated english text would appear in their open terminal. A new diagram below shows this idea:



We think that this is an acceptable change to our project because it focuses more on the NLP aspects of the problem (OCR and machine translation) while still allowing players to easily understand the Japanese text on their screens. We kept our priority of staying mostly true to the original game systems, meaning that we do the whole NLP pipeline without using cloud OCR or translation services and that we run it on low powered hardware (a Raspberry Pi Model 3 B+). The project will help people who do not speak Japanese enjoy video games that were not released officially in America, like the famous Nintendo GameBoy Advance game *Rhythm Tengoku*, which was released in Japan in the mid 2000's but never released in America.

Methods

We are trying to solve this problem in two parts: extracting the Japanese game text and translating it to English. We decided to use OCR to extract text since this would make our tool usable on any game with readable Japanese text and would not require special hacks to extract text from game binaries. This OCR approach seems to be the most common way of solving the text extraction problem in existing solutions, like in Retroarch's "Retroarch AI Service".[1] These

[1] https://docs.libretro.com/guides/ai-service/

existing solutions use cloud OCR services, like VGTranslate and Google Cloud OCR. Our OCR solution is different than the other video game translation systems we've investigated, as we don't use cloud OCR and instead use the open source Tesseract OCR.[2]

Our baseline method involves simply using the default "jpn" language pack included in Tesseract. To improve on the baseline, we trained Tesseract on a pixel-based font called JackeyFont[3] that resembled some Japanese video games we've played in the past, like Pokemon for the GameBoy Advance. We followed a YouTube guide[4] that detailed the process for training a new English font on Tesseract, which was critical for getting started with the training process. Tesseract provides comprehensive documentation[5] for training new fonts that we used after we watched the video and were more familiar with the steps involved in training a new font. We will compare the performance of the two methods in the "Experiments" section of this report.

The translation model is used to translate the Japanese output from the OCR into English. We choose OpenNMT-tf[6] toolkit to run our NMT system, because of the modularized design and ease-to use. We trained SentencePiece[7] model with 16,000 token size to preprocess our data and used fastText[8] pretrained embeddings. For our baseline model, we used LSTM seq2seq model (RNN-Small shown in the experiment section) to translate the text. Since we are constraint resource-wise while needing better performance, we tried to improve from our baseline in those factors. First, we tested different model architecture in terms of speed and translation quality. We tested 3 variants of LSTM, all different in hidden size and 4 variants of the Transformer,

[2] https://github.com/tesseract-ocr/tesseract
[3] https://archive.org/details/jackeyfont
[4] https://youtu.be/TpD76k2HYms
[5] https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract-4.00
[6] https://github.com/OpenNMT/OpenNMT-tf
[7] https://github.com/google/sentencepiece
[8] https://fasttext.cc

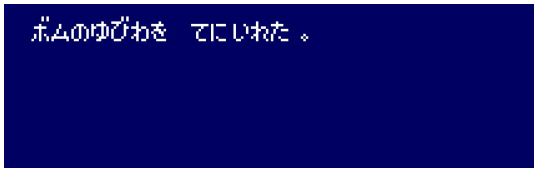
different in number of attention heads, hidden size, pointwise-ffn inner dimension, and using either relative positional encoding or absolute positional encoding. Then, we chose Transformer with relative positional encoding, which showed the best tradeoff between translation quality and speed. We thought this was due to the Transformer of being parallelizable during the encoding of the text. Our choice of Transformer differ from the vanilla Transformer in that it uses a relative positional encoding, a variant of positional encoding that only encodes the relative position of the text given a fixed sized window. Furthermore, we tied the input and the output embeddings for the decoder, a trick[9] used to decrease model parameters and improve training/translation time. By using Transformers, we could improve the BLEU score on the JESC dataset by 4 BLEU. Furthermore, translation speed increased upto 60% compared to the worst-performing RNN Big. We show the experimental results in the next section.

Experiments

OCR

To test our OCR solution, needed to capture some video game screenshots and label them with their correct text. This was initially a difficult task, since manually transcribing each screenshot is not only time consuming but also complicated, since we do not know how to type Japanese characters using our English keyboards. However, after some searching, we found an excellent Super Nintendo emulator, Wanderbar[10], that allows us to view Japanese text as it's displayed on the screen. With this, we were able to more easily copy and paste the Japanese text into our image labels. An example of a labelled screenshot of Final Fantasy IV is below:

[9] https://arxiv.org/pdf/1608.05859.pdf
[10] https://legendsoflocalization.com/wanderbar/



Label: ボムのゆびわを てにいれた。

Our complete OCR test dataset is composed of 16 images from Final Fantasy IV, 17 images from Final Fantasy VI, and 7 images from Breath of Fire II. We measure success based on the CER, computed with the CER Python module[11] used in Homework 2, between the model and the ground truth. Our baseline method was using the default "jpn" configuration included in the Tesseract installation, and our improved version is trained on the pixel font mentioned earlier. We ran the OCR (both the default "jpn" configuration and our custom-trained one) on the screenshots from the individual games, computed the CER on a per-game basis, and finally computed the total CER for all games. We also removed all whitespace from our results and our labels, since we wanted to focus on the actual characters that it was able to recognize and not the word separation, which it often missed. Our results are reported in *Table 1* below. In general, we found that text with a non-opaque background gave very poor results and for the most part kept our dataset to games that have solid-colored text boxes. The model seemed to perform best on games where the text was wider (e.g. Final Fantasy) and seemed to perform worse on skinny, smaller texts (e.g. Breath of Fire).

Table 1: OCR Test Results

Game	Number of Images	CER (Default)	CER (Trained)
Final Fantasy IV	16	0.5639344262295082	0.3245901639344262
Final Fantasy VI	17	0.7551963048498845	0.5750577367205543
Breath of Fire	7	0.7348066298342542	0.6519337016574586

[11] https://github.com/ND-CSE-40657/hw2/blob/master/cer.py

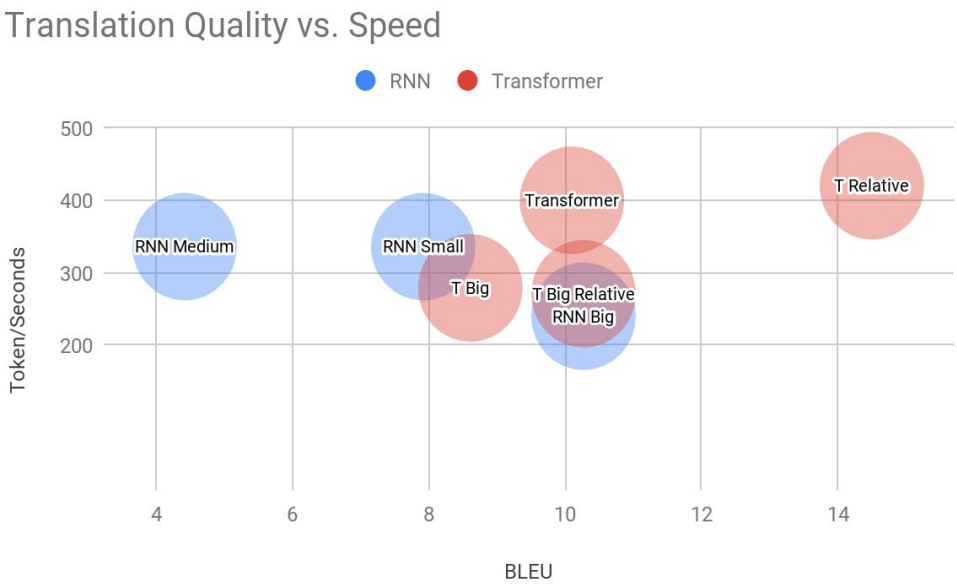
Total CER (Default): 0.6877040261153428	Total CER (Trained): 0.5070729053318824

While our results aren't mind blowing by any means, our trained solution performs significantly better than our untrained solution, which we believe is because our trained solution includes fonts that resemble the pixel fonts used in the games. We reduce the total CER from 68.7% to 50.7%, which is quite a big change, and from 56.3% to 32.4% on Final Fantasy IV, whose fonts matched closely with the trained pixel font. This means that for Final Fantasy IV, we are actually getting most of the characters correct! Our solution also performs well, empirically, on the Pokemon GameBoy Advance games that inspired us to do this project, but we were unable to easily extract the Japanese text from these games and thus did not include any real metrics for those games.

Translation

We use 2 metrics to test our translation system, translation quality and speed. To test our translation system, we tested our system using the test set of the JESC. We chose JESC dataset because it most closely matched with the video game domain, where most of the text is a conversation. We tested 7 different models, RNN Small (2 layers with 512 hidden size), RNN Medium (4 layers with 512 hidden size), RNN Big (4 layers with 1024 hidden size), Transformer (8 attention heads, 512 hidden units, 2048 pointwise-FFN dimension size), Transformer-relative (same as Transformer, using relative positional encoding), Transformer-Big (16 attention heads, 1024 hidden units, 4096 pointwise-FFN dimension size), Transformer-Big-relative (same as Transformer-Big, using relative positional encoding). The results are as follows, all tested on the JESC test set.

As can be seen in the chart below, Transformer with relative positional encoding performed the best in terms of speed and quality. The Transformer model had less parameter too, with 114 million parameters compared to 240 million parameters in RNN small (which has smallest parameters). We improved from our baseline RNN-small by 6 BLEU scores and 60% speed up during translation. The speedup came from the Transformer-relative model having fewer parameters and the increase translation quality came from Transformer's better architecture.



To test the system in the video game domain, we extracted a small parallel paragraph from the Final Fantasy Game and compared our results with Google Translate. Below are the short excerpt from the ground truth, our NMT system, from Google Translate.

Table 2: Comparison of Game Translations

Ground Truth	Long ago, the War of the Magi reduced the world to a scorched wasteland, and Magic simply ceased to exist.
Google Translate	When all the Great Wars were burned, when the battle was over, the power of "magic" disappeared from the world

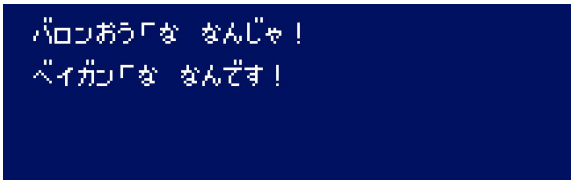
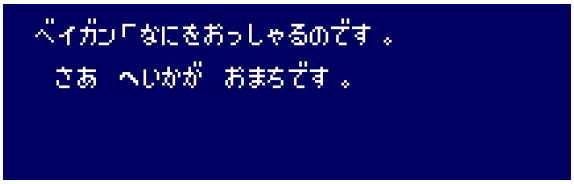
Our NMT	when that battle was over, the power of magic has disappeared.
Ground Truth	1000 years have passed... Iron, gunpowder and steam engines have been rediscovered, and high technology reigns.
Google Translate	And 1000… Iron, gunpowder, steam engine People used the power of machinery to revive the world
Our NMT	and for 1000 years, iron, gunpowder, steam, and steam workers have built the world.
Ground Truth	But there are those who would enslave the world by reviving the dread destructive power known as "Magic".
Google Translate	Here again there is a person who tries to revive the power of the legendary “magic” and dominate the world with its powerful armed forces…
Our NMT	there are also people who seek to control the world by the legendary magic.
Ground Truth	Can it be that those in power are on the verge of repeating a senseless and deadly mistake?
Google Translate	Is the person trying to repeat that mistake again?
Our NMT	are people trying to turn that frown back?

When we run the BLEU metric, both Google Translate and our NMT gets 0 BLEU score mostly due to 0 4-gram matches. However, our unigram match and bi-gram matches are around twice compared to Google Translate, showing that our local NMT is competitive to Google Translate’s services. Empirically a human user can understand what is going on in the game, serving the purpose of the translation.

Putting it all together

Finally, we ran our completed system, with the output of our OCR fed into the input of our translation system. Since we knew that our BLEU score was 0.0, we decided not to report that metric for this part, and instead simply show a couple of random examples of the translations we received from the system:

Table 3: Results of Full System

Image	Ground Truth	Translation
	King W... What. Baigan W... What is it.	what is it?
	Baigan What are you trying to say. This way, Cecil.	what are you talking about?
	An eerie glow surrounds you.	the shining light is a sword.
	Guard We won_t hand over the Esper!!	i'm sorry.

Despite the poor BLEU score, we were impressed by some of these translations. We don't know of any way to properly quantify these results, but we think that they are generally helpful to someone who knows a little bit of background about the game they're playing and are

trying to navigate the story or menus in a language they're unfamiliar with. For fun, here's a screenshot of the Pokemon game we were inspired by, and its translation:

Translation: if you don't know anything, it'll help you with a l button or a r button.

This is actually a great translation! The player immediately knows that he can get assistance by pressing the "L" or "R" button on his or her device. Overall, we considered the full system experiment a success.

Conclusions

One of the biggest takeaways we've learned from this project is how error can stack in a multi-part system. Seemingly acceptable error rates in our OCR systems could turn into complete mistranslations by the language model. However, even with this challenge, our solution seemed to perform pretty well on our test data set. At the very least, we believe that someone who really wants to play a Japanese game could use this system and have some sense of what's going on. It was also interesting to see our language model perform better than Google Translate in terms of BLEU score on the test data.

A disappointment we came across was that we weren't able to actually test this full system on the Raspberry Pi like we had planned due to some logistical issues, such as us having a hard time finding an emulator that supported the Raspberry Pi's architecture. However, given the measured performance of our smaller language model and the easily extendable Tesseract language pack, we believe that our system would work on such low powered hardware.

Overall, we are very proud of how this project turned out. We hope that we can eventually build on our current system and release a completed product to the open source community, or even integrate our solution into an emulator like Retroarch.

Replicability

Our GitHub repository and all documentation, including links and datasets, is located at the following link:

https://github.com/nmarcopo/videoGameTranslator