Seq2Seq neural network models for transliteration

DS 303 End-Term Project

Eeshaan Jain Thomas Jacob Vinamra Baghel

May 8, 2021

Indian Institute of Technology Bombay

Outline

Introduction

Transliteration

Transliteration is the conversion of a given name in the source language to a name in the target language, such that the target language name is:

- Phonetically equivalent to the source name
- Conforms to the phonology of the target language
- matches the user intuition of the equivalent of the source language name in the target language

The Machine Aspect of Translation & Transliteration

Machine translation is the automatic conversion of text/speech from one natural language to another. Machine transliteration often adopts approaches from machine translation though there is one subtle difference:

- The objective in translation is to preserve the semantic meaning (while following the syntactic structure)
- The objective in transliteration is to preserve the original pronunciation of the source word (while following the phonological structures)

Need for Machine Transliteration

The biggest significance of transliteration is having a base language of pronunciation for people of different backgrounds and cultures. This is important as it facilitates communication and new language learning.
Machine Transliteration is the process by which a word written in source language is transformed into a target language, accurately and unambiguously, by preserving the phonetic aspects and pronunciation. Generally named entities or proper nouns are transliterated from one orthographic system to another. The role of a machine in transliteration is to connect two languages without human intervention. This is key to having a common platform for people to not only learn new languages but also Automatic Translation.

Description of the Project

Task

We have designed a system capable of taking in images of Hindi words in Devanagari script and giving out a list of all the detected words in their transliterated English form. This was done by pipelining an OCR model and an NMT model (Neural Machine Translation with Attention). The OCR would detect all the Hindi words in the image while the NMT would transliterate the words into their English forms.

In the project, we have implemented and compared 2 separate NMT models. NMT-1 (link) is an Encoder-Decoder model with Attention and NMT-2 (link) is an Encoder-Decoder Translation model with Attention adapted for Transliteration. With this project, we intend to expand our knowledge into the horizons of Optical Character Recognition and Neural Machine Translation.

Motivation

As described in the task, the goal of the project is to create a model which takes the picture of a word in Hindi, and returns the equivalent word in English. The main motivations behind proceeding with this project were as follows:

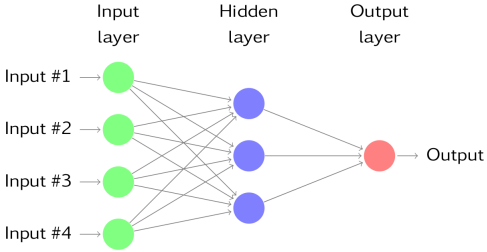
1. There are many a times languages become a barrier to understanding the phonetics of the secondary language (especially to travelers). Such a model will help to reduce that barrier by providing transliteration between the languages
2. This would expose us to two major domains in AI/ML which are optical character recognition, and neural machine translation.

Sequence Models

Sequence Models

Recurrent Neural Networks

Neural Network without Hidden States



Taking number of hidden states as 1 and activation as ϕ, let the minibatch data $\mathbf{X} \in \mathbb{R}^{n \times d}$, and $\mathbf{H} \in \mathbb{R}^{n \times h}$ is given as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h)$$

Neural Network without Hidden States

The output layer is given as

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{\mathbf{hq}} + \mathbf{b}_{\mathbf{q}} \qquad \text{where} \quad \mathbf{O} \in \mathbb{R}^{n \times q}$$

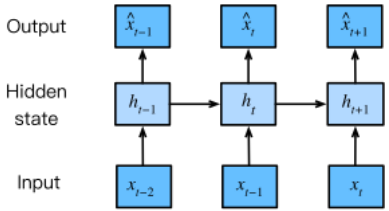
For the task of classification, we follow up this layer with the softmax function

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

This takes an input of q numbers, and normalizes them to a probability distribution, i.e maps the space $\mathbb{R}^q \to [0, 1]^q$

Autoregressive Models

For sequential data, suppose we want to (at time t), predict x_t by the previous data, i.e $x_t \sim \mathbb{P}(x_t|x_{t-1} \cdots x_t)$. All these x_j depend on t directly or indirectly, and we need approximations to calculate this probability efficiently.
Autoregressive models approximate the time sequence to the length $\tau < t$ which makes the sequence length to consider uniform over all cases.



We estimate the entire sequence as

$$\mathbb{P}(x_1, \cdots x_T) = \prod_{t=1}^{T} \mathbb{P}(x_t | x_{t-1}, \cdots x_1)$$

The *Markov Condition* is said to be satisfied if the approximation described above of taking sequence of length τ is accurate. For $\tau = 1$, we have a first-order Markov model given as

$$\mathbb{P}(x_1, \cdots x_T) = \prod_{t=1}^{T} \mathbb{P}(x_t | x_{t-1}) \qquad \text{with} \quad \mathbb{P}(x_1 | x_0) = \mathbb{P}(x_1)$$

Recurrent Neural Networks with Hidden States

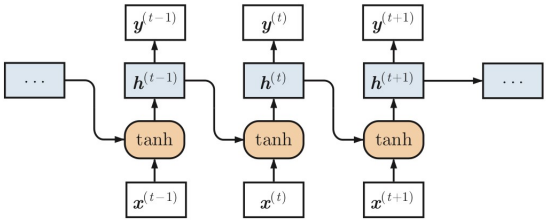
Consider $\mathbf{X_t} \in \mathbb{R}^{n \times d}$ at time t. $\mathbf{H_t}$ denotes the hidden variable at time t, and unlike described in MLP, we take the previous hidden variable into consideration here. More specifically, if $\mathbf{H_t} \in \mathbb{R}^{n \times h}$, we write

$$\mathbf{H_t} = \phi(\mathbf{X_t W_{xh}} + \mathbf{H_{t-1} W_{hh}} + \mathbf{b_h})$$

We effectively retain the sequence's historical information by capturing the $(t-1)^{th}$ time step in the t^{th}.
Such hidden variables are the hidden states of the recurrent neural network. At every stage, the output is calculated as it is in MLP i.e

$$\mathbf{O_t} = \mathbf{H_t W_{hq}} + \mathbf{b_q}$$

Recurrent Neural Networks with Hidden States



Such an RNN is called a “Vanilla RNN”.

In language modeling, the task is to predict the next token based on the current and past tokens. Such a model can be realised using RNNs at a character level. The output is passed through a softmax layer in this case, and the cost is computed using the cross-entropy loss function.

Sequence Models

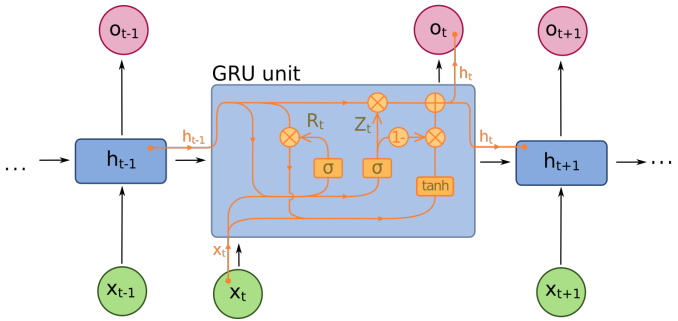
Modern Recurrent Neural Networks

Exploding and Vanishing Gradients

When backpropogating through sequence models with a hidden state, there tend to be repeated matrix products leading to high exponentiated matrices. Such matrices lead to explosion or vanishing of their eigenvalues which in turn leads to the same effect on their gradients. We essentially need three things to solve many of the problems caused by gradient anomalies:

1. **Memory Cell :** It would be beneficial to store any important information we get early on in the sequence in the memory.
2. **Skipping :** When we come across irrelevant information, it would be best to skip it.
3. **Resetting :** We should reset our internal states when we completely change the context such as when bullish market turns to bearish.

Gated Hidden State



GRUs support gating which helps in deciding when to update the gate, and when to reset it. If the initial inference is important, the neural network can choose not to update the hidden state and retain the original information.

Reset and Update Gates

To model these gates, we set them as entries in $(0, 1)$ to perform some convex combinations like how much of previous information is to be remembered and how much is to be passed ahead.
Considering the same dimensions for $\mathbf{X_t}$ and $\mathbf{H_t}$ as described before, we define the reset gate ($\mathbf{R_t}$) and update gate ($\mathbf{Z_t}$) both $\in \mathbb{R}^{n \times h}$ as follows:

$$\mathbf{R_t} = \sigma(\mathbf{X_t W_{xr}} + \mathbf{H_{t-1} W_{hr}} + \mathbf{b_r})$$

$$\mathbf{Z_t} = \sigma(\mathbf{X_t W_{xz}} + \mathbf{H_{t-1} W_{hz}} + \mathbf{b_z})$$

The sigmoid ensures that these stay in the required range.

Candidate Hidden State

The reset gate and input are integrated to get the candidate hidden state ($\widetilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$) as

$$\widetilde{\mathbf{H}}_t = \tanh\left(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h\right)$$

Here $\odot$ denotes the Hadamard product operator. Due to $\tanh$, the value of the state always remains in $(-1, 1)$. Note that if the values in the reset gate are close to 1, we essentially get back the Vanilla RNN we described above. For entries close to 0, we see that the information of the state gets "reset" to default and we get a MLP with $\mathbf{X}_t$. Till now we see that we haven't incorporated the update gate.

Hidden State

Now to add the effect of the update gate, we simply take the convex combinations of H_{t-1} and $\widetilde{H}_t$ as

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \widetilde{H}_t$$

Like before, we see that if the update gate is close to 1, we retain the previous state and thus information from current input (present in candidate state) is ignored. Instead of the value is close to 0, the hidden state is approximately the candidate hidden state. Such a design helps to solve the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances.

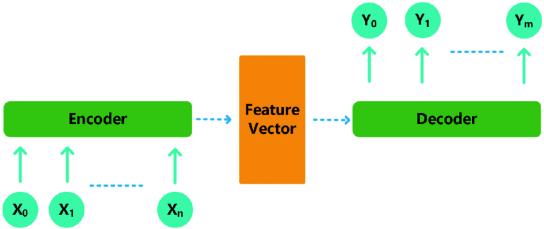
Sequence Models

Encoder Decoder

Need for such a model

In tasks such as Machine Translation and Machine Transliteration, the sequence lengths of input and output need not be same, and hence a more complex architecture as compared to vanilla RNNs is needed. Encoder decoder architecture is one such, in which the encoder takes in an input sequence and transforms it into a state with predefined dimensions, and the decoder takes that state and transforms it into a variable length sequential output. We can utilize vanilla RNNs in such an architecture to implement *sequence to sequence learning*. We use special tokens such as <PAD> (padding) and <EOS> (end of statement) for machine transliteration.

Encoder



Given a sequence of variable length T, the encoder transforms this sequence into a context variable $\mathscr{C}$ and encodes the information in it.

$$\mathsf{H}_t = f(\mathsf{X}_t, \mathsf{H}_t), \qquad \mathscr{C} = g(\mathsf{H}_1, \mathsf{H}_2, \cdots, \mathsf{H}_T)$$

Note that f and g are generalized functions and different implementations can use different of these.

Decoder

During training, we would have the output sequence $\mathsf{Y}_1, \mathsf{Y}_2, \cdots \mathsf{Y}_\gamma$ and for each time step τ, we note that the probability is conditional on the previous outputs and the context variable, i.e $\mathbb{P}(\mathsf{Y}_\tau | \mathsf{Y}_{\tau-1}, \cdots \mathsf{Y}_1, \mathscr{C})$. Such a conditional dependence can be implemented with yet another RNN and model its hidden state as

$$\mathsf{S}_\tau = q(\mathsf{Y}_{\tau-1}, \mathscr{C}, \mathsf{S}_{\tau-1})$$

Here q is also a generalized function, and this state transformation can be forwarded to an output layer and a softmax helps modeling the conditional probability displayed above.

Sequence Models

Attention Mechanisms

Need for attention

Sequence to Sequence models were made to work on language modeling but it often failed with long source inputs, i.e there was a large tendency to forget the initial part, and once it's forgotten, the network made it impossible to regain that information.

Attention mechanisms were developed to memorize long source sentences in Machine Translation tasks. Attention essentially creates shorter paths between the input and the context variable. Each output element gets a specific weight, so there is independent connections per se.

Definition

Consider an input of length T and output of length γ. For all time steps t, there is a encoder hidden state $\mathbf{H_t}$ and a decoder hidden state $\mathbf{S}_\tau = q(\mathbf{Y}_{\tau-1}, \mathcal{C}_\tau, \mathbf{S}_{\tau-1})$ for output as position τ. The context vector is given as $\mathcal{C}_\tau = \sum_t \alpha_{\tau,t}\mathbf{H_t}$ where the weights $\alpha_{\tau,t}$ are given by

$$\alpha_{\tau,t} = \text{align}(\mathbf{Y}_\tau, \mathbf{X_t}) = \text{softmax}(\text{score}(\mathbf{S}_{\tau-1}, \mathbf{H_t}))$$

The weights assigned by the model define how much of each source hidden state should be considered for each output. Originally the score was modeled by a single hidden layer network as

$$\text{score}(\mathbf{S}_{\tau-1}, \mathbf{H_t}) = \mathbf{V}^\mathsf{T} \tanh\left(\mathbf{W}[\mathbf{S}_\tau, \mathbf{h_t}]\right)$$

Breakdown of the Project

Parts of the Project

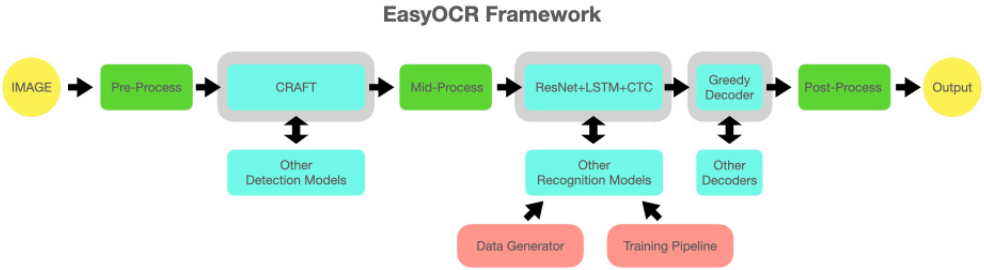
The project has been divided into three major sections:

1. Studying about OCR and NMT
2. Working on the OCR model
3. Working on two independent NMT models

Breakdown of the Project

OCR Model

EasyOCR - The OCR Model



OCR stands for Optical Character Recognition. The model we have used in the project is EasyOCR by *JaidedAI*.

EasyOCR - Architecture

The algorithm is divided into 2 main parts: Detection and Recognition.

1. Detection is done using the CRAFT algorithm from this paper.
2. Recognition is based on CRNN. It is composed of feature extraction using Resnet, sequence labeling (LSTM) and decoding (CTC).

The training pipeline of this architecture is a modified version of deep-text-recognition-benchmark. Data synthesis is based on TextRecognitionDataGenerator.

EasyOCR - Use

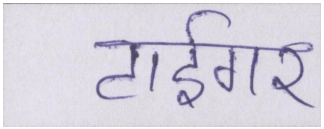
EasyOCR has been written in Pytorch. Its use in our project is simple - we directly use it as a package. We installed it using pip and declared the object *reader*.

```
import easyocr
reader = easyocr.Reader(['hi']) # hi for Hindi
```

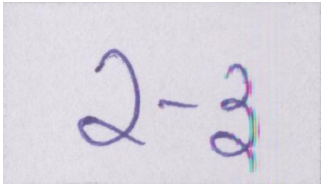
The function OCR takes in the image path as input (img_file) and returns a list of detected Hindi words in the image.

```
return reader.readtext(img_file, detail = 0)
```

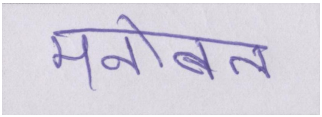
EasyOCR - Example



Input: Path/to/tiger.jpg
Output: ['□□□□□']
Tiger in Hindi



Input: Path/to/2-3.jpg
Output: ['□-□']
2-3 in Hindi



Input: Path/to/manobat.jpg
Output: ['□□□□□']
Manobat in Hindi

EasyOCR Examples

Note that there was an issue with typesetting of Devanagri font in pdflatex, and it wasn't rendered. Hence the explanation of output is written.

Breakdown of the Project

NMT Model-1
Encoder Decoder model with Attention

Preprocessing

The following main steps were involved in the preprocessing of the dataset:

1. Creation of lookup tables, comprising making dictionaries (i.e. mapping tables) of character to corresponding character index and vice versa for both the source (Hindi) and target (English) characters. The <PAD> token was added to both vocabularies as the first token.
2. Extraction of the set of characters making up each word: English were straightforward and for Hindi, the relevant Unicode characters are from 2304 to 2432.
3. Conversion of each extracted character into an index with the help of the previously created lookup tables (Creating v2i and i2v dictionaries where i is the index and v is the vocabulary)

Model Architecture

This was the first model (naive) we built after reading of the relevant papers on sequence to sequence architecture and attention. A simple encoder-decoder model is used consisting of a single GRU in each encoder and decoder, and an embedding dimension of 256. The attention used is Bahdanau attention given as

$$\text{score}(S_\tau, H_t) = V^T \tanh\left(W[S_\tau; H_t]\right)$$

The model in entirety is defined on the next slide, with all the corresponding dimensions. (Note that our Hindi vocabulary had a size of 129, and English vocabulary had of 27 in this case).

Model Architecture

```
Encoder_Decoder(
(encoder_rnn): GRU(129, 256)
(decoder_rnn): GRU(512, 256)
(hidden_to_output): Linear(in_features=256, out_features=27,
bias=True)
(softmax): LogSoftmax(dim=2)
(U): Linear(in_features=256, out_features=256, bias=True)
(W): Linear(in_features=256, out_features=256, bias=True)
(attention): Linear(in_features=256, ou
```

Relevant PyTorch General functions

1. **torch.utils.data.Dataset :** This is an abstract class which represents our dataset. A custom dataloader has been written which inherits from this.
2. **torch.device :** This is an object which tells us where the tensors are saved
3. **torch.cuda.is_available :** Checks if cuda (GPU) is available on the system.
4. **torch.zeros :** Returns a tensor with 0s with the specified shape.
5. **torch.bmm :** Does batch matrix multiplication.
6. **torch.cat :** Concatenates sequence of tensors in given dimension.
7. **torch.argmax :** Returns the index of the maximum value in the tensor.

Relevant PyTorch Layer Functions

1. **torch.nn.GRU :** Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.
2. **torch.nn.Linear :** Applies a linear transformation to the incoming data - $y = xA^T + b$.
3. **torch.nn.LogSoftmax :** Applies n dimensional softmax followed by a logarithm on the input.

Loss Function & Optimizer

The loss function used for this model is the negative log likelihood loss, torch.nn.NLLLoss, which is applied only on models with the softmax function as an output activation layer.
The optimizer used was Adam, torch.optim.Adam and its formulation is given as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon}\hat{m}_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad , \quad m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{v}_t = \frac{m_t}{1 - \beta_2^t} \quad , \quad m_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Accuracy Metrics

For this vanilla encoder-decoder model, we used a simple character level accuracy which truncates the output to the characters given in the ground truth and binary compares the characters. The following table tells the result (note that the model was trained over 5 sets of hyperparameters, each for 1000 iterations - the top (0.0*xy*) represents the learning rate and the bottom (32/64/128) represents the batch size.

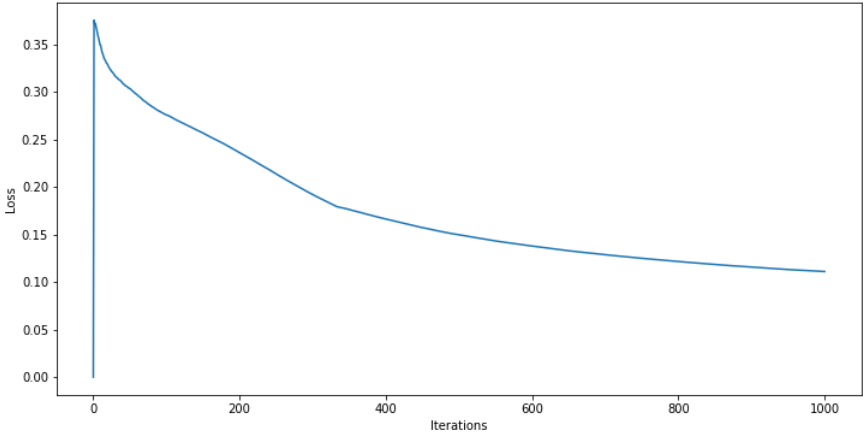
Model	0.001 64	0.01 64	0.001 32	0.01 32	0.001 128
Accuracy	78.13	76.40	78.04	54.87	11.03

Results

We see that batch size of 64 along with learning rate of 10^{-3} gave the best result, however if trained for more iterations, the results will improve. Also notice that increasing the batch size from 64 to 128 adversely affected the model dropping its accuracy by more than 65%!

I am attaching the graph of loss function for the best model, however the graphs for all are present in the folder by the name “lr={}-nb=1000-bs={}-e=256.png”. The corresponding PyTorch models are also presented with the same name and extension as .pt

Loss Plot


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Loss
0
200
400
600
800
1000
Iterations

Breakdown of the Project

NMT Model-2
Encoder Decoder Translation model with Attention Adapted for Transliteration

Preprocessing

The following main steps were involved in the preprocessing of the dataset:

1. Creation of lookup tables, comprising making dictionaries (i.e. mapping tables) of character to corresponding character id and vice versa for both the source and target characters (vocabulary)
2. Extraction of the set of characters making up each word
3. Conversion of each extracted character into an index with the help of the previously created lookup tables
4. Finally, all the data was saved onto an external file for future reference

Model Architecture

The model broadly consists of an encoder to build a “thought” vector, which is basically a sequence of numbers that represents the sentence meaning, and a decoder which then processes the sentence vector to emit a translation. The particular encoder and decoder architectures used for this implementation of translation are as follows:

1. **Encoder:** First step is the setting up of word embeddings. Given the categorical nature of words, the model must first look up the source and target embeddings to retrieve the corresponding word representations. The embedding weights, one set per language, are usually learned during training. Following this, the output of the embedding layer is fed to a GRU layer, and the output of the GRU layer forms the input to the decoder.

2. **Decoder:** In the decoder, we first compute the attention weights. We use attention because through this, the context vector enables the decoder to focus on certain parts of the input when predicting its output. These are computed by first passing the encoder output and hidden inputs through 2 separate tf.keras.layers.Dense functions, combining the outputs, applying the tanh activation function, followed by another tf.keras.layers.Dense function, and finally a softmax function. The attention weights multiplied by the encoder outputs form the context vector. This context vector is then passed through a tf.keras.layers.Embedding function, followed by a GRU layer and finally a tf.keras.layers.Dense function whose output is the prediction.

Relevant TensorFlow General Functions

1. **tf.keras.preprocessing.sequence.pad_sequences:** This function transforms a list of sequences into a 2D Numpy array of shape (num_samples, num_timesteps). num_timesteps is generally the maxlen argument if provided.
2. **tf.data.Dataset.from_tensor_slices:** This provides a simple way to create a dataset from a python list
3. **tf.nn.sparse_softmax_cross_entropy_with_logits:** This measures the probability error in discrete classification tasks in which the classes are mutually exclusive.
4. **tf.train.Checkpoint:** A Checkpoint object can be constructed to save either a single or group of trackable objects to a checkpoint file. It maintains a save_counter for numbering checkpoints.

Relevant TensorFlow Layer Functions

1. **tf.keras.layers.Embedding:** This function turns positive integers (indexes) into dense vectors of fixed size.
2. **tf.keras.layers.GRU:** This provides an implementation of the Gated Recurrent Unit layer
3. **tf.keras.layers.Dense:** This is an implementation of a regular densely-connected NN layer.

Loss Function & Optimizer

The loss function used for this model is a tweaked version of the normal cross entropy loss, tf.nn.sparse_softmax_cross_entropy_with_logits, which basically measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). We use a mask before feeding in the values to the loss function so that only the unequal pair of values are penalized.
The cross-entropy function is given by:

$$\mathrm{H}(\mathbf{p}, \mathbf{q}) = -\sum_{x} p(x) \log p(x) q(x)$$

The optimizer used for this task is the common Adam optimizer. We keep the learning rate at the default value of 0.001 and proceed.

Accuracy Metrics

Since this is a unique problem where the usual kind of accuracy metrics can't be employed due to the often differing input and output sizes, we have used different metrics which compute measures such as "distance" and "similarity" between two strings and used these to quantify our model performance. The specific accuracy metrics we have employed from the Levenshtein library are as follows:

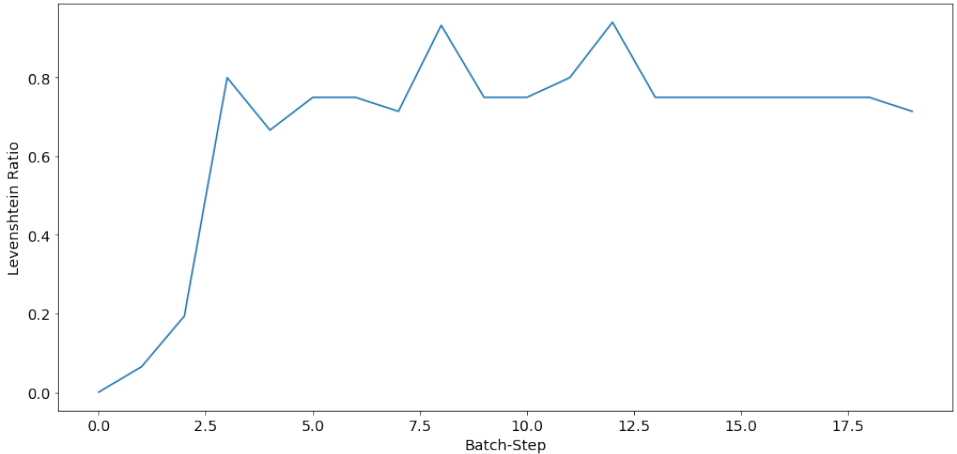
1. **ratio:** To compute similarity of two strings.
2. **jaro_winkler:** To compute Jaro string similarity metric of two strings. The Jaro-Winkler string similarity metric is a modification of Jaro metric giving more weight to common prefix, as spelling mistakes are more likely to occur near ends of words.
3. **distance:** To compute absolute Levenshtein distance of two strings.

Results

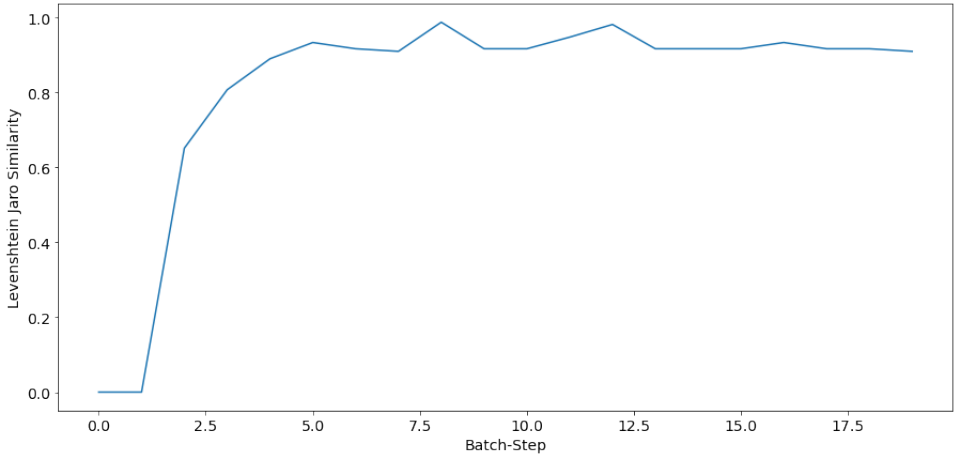
Metric Name	Value
Average Loss	0.0903
Average Ratio	0.921
Average Jaro	0.959
Average Distance	0.809

The model was trained for 2 epochs, with each epoch going through 29821 training samples. Training was performed in mini-batches with the batch size being 32, giving a total of 931 batches. The final metric values obtained are in the table. In the following slides are the plots of the variation of the different accuracy metrics.

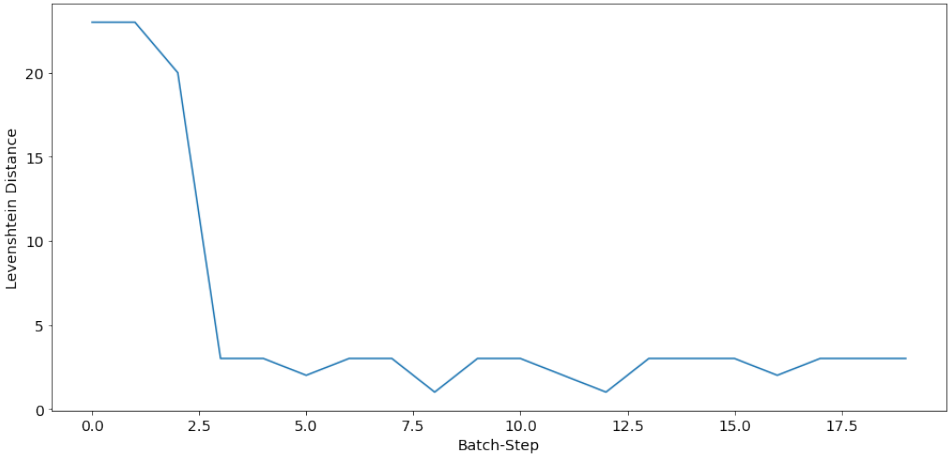
Results - Average Levenshtein Ratio Variation



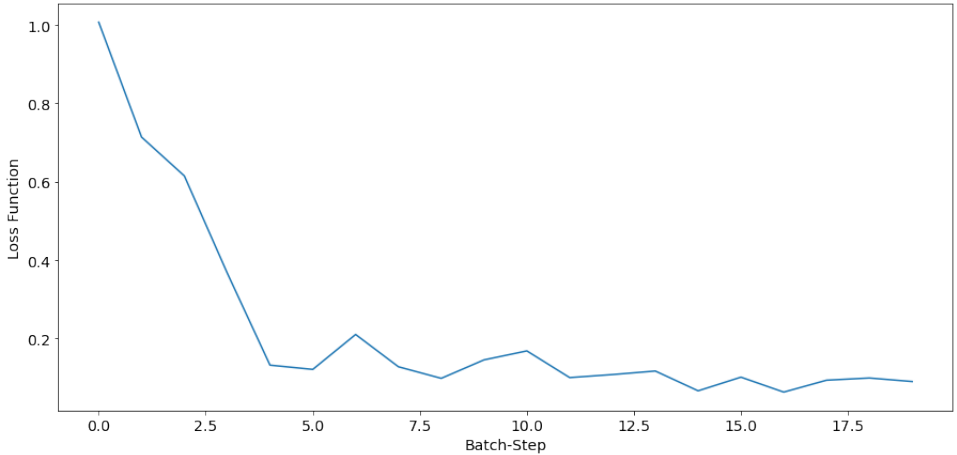
Results - Average Levenshtein Jaro Similarity Variation



Results - Average Levenshtein Distance Variation



Results - Loss Function Variation



File Tree

Directory - Relevant files

19D070022_190070068_190010070/ |– Adapted Transliteration Model/
| |– DS303_Project_v3_0.ipynb
| |– Data_preprocessing.py
| |– transliteration.txt
| ‘– *.png
|– Basic Encoder Decoder/
| |– Basic_Encoder_Decoder_Attention_Model.ipynb
| |– lr=0.0xy-nb=1000-bs=xyz-e=256.pt
| ‘– lr=0.0xy-nb=1000-bs=xyz-e=256.png
|– Trial (DeepTrans)/
| ‘– DeepTrans.ipynb
|– README.md
|– preprocessing.py
‘– DS303-Project.pdf

Contributions

Everyone: Read the relevant papers and blogs and understood the concepts in depth.

Eeshaan Jain: Implemented the model NMT-1 and worked on the presentation

Thomas Jacob: Implemented the model NMT-2 and worked on the presentation

Vinamra Baghel: Implemented EasyOCR part of the project, studied and efforted(but couldn't complete due to version mismatch issues) to implement a more complex seq2seq model described here: DeepTrans(link), and worked on the presentation

Thank You

- Eeshaan Jain
- Thomas Jacob
- Vinamra Baghel

References

[1] Mihaela Rosca, Thomas Breuel (2016). *Sequence-to-sequence neural network models for transliteration.*

[2] Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C. (2003). *A neural probabilistic language model.*

[3] Werbos, P. J. (1990). *Backpropagation through time: what it does and how to do it.*

[4] Hochreiter, S., Schmidhuber, J. (1997).*Long short-term memory.*

[5] Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y. (2014).*On the properties of neural machine translation: encoder-decoder approaches.*

[6] Sutskever, I., Vinyals, O., Le, Q. V. (2014). *Sequence to sequence learning with neural networks.*

References

[7] Graves, A. (2013). *Generating sequences with recurrent neural networks*

[8] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). *Attention is all you need.*

[9] Mohamed Seghir Hadj Ameura, Farid Mezianeb, Ahmed Guessouma *Arabic Machine Transliteration using an Attention-based Encoder-decoder Model*

[10] *Character Region Awareness for Text Detection* Youngmin Baek, Bado Lee, Dongyoon Han, Sangdoo Yun, Hwalsuk Lee

[11] *An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition* Baoguang Shi, Xiang Bai, Cong Yao

[12] *Deep Residual Learning for Image Recognition* Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

References

[13] Connectionist Temporal Classification (CTC)

[14] Attention in RNNs

[15] Intuitive Understanding of Attention Mechanism in Deep Learning

[16] Neural Machine Translation with Attention - tutorial (Tensorflow)

[17] **Dataset:** FIRE ’13 Transliteration dataset

[18] Translation of text from English to Hindi