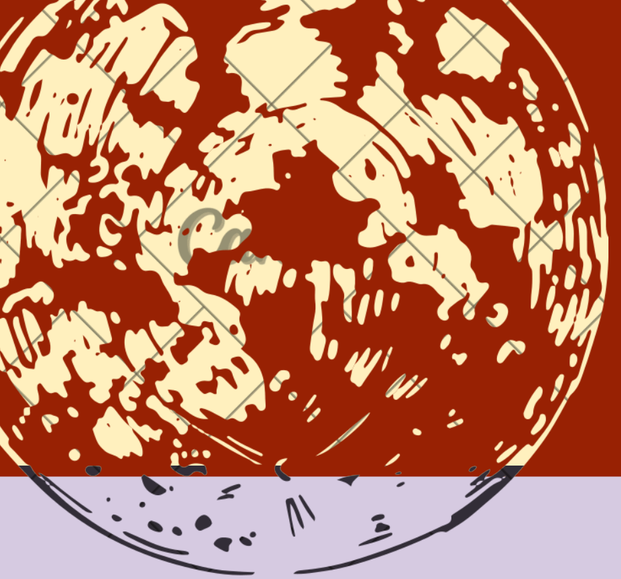
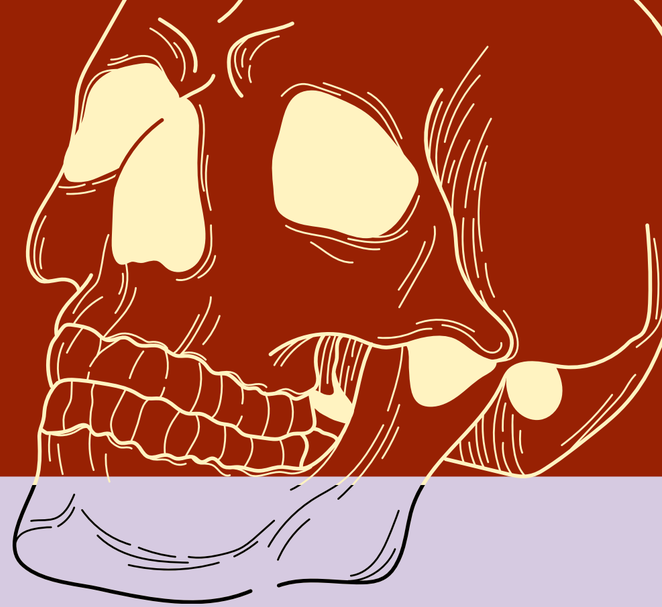
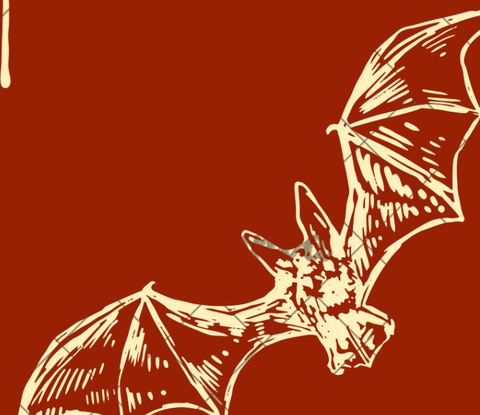




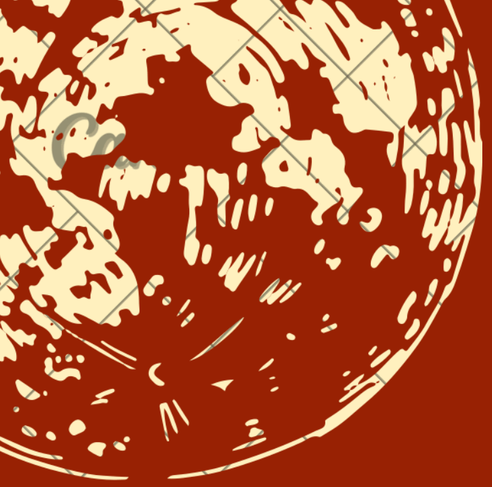


TEXT GENERATION

USING “Crime And Punishment”




Presented By : Zakaria Benlamkadam
Supervized by : Ms. Allaouzi Imane




This project aims to imitate the language and the style of the great author “Fyodor Dostoevsky”





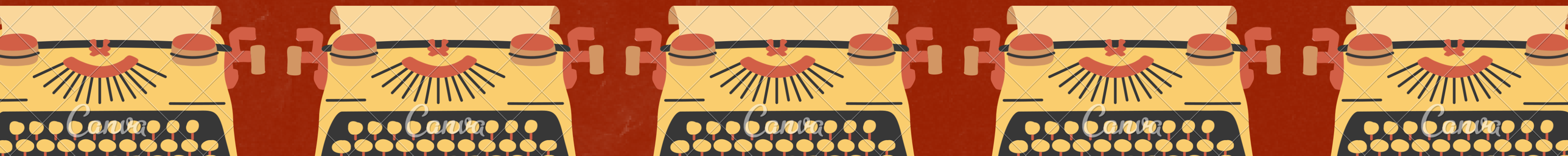
Contents

Fyodor Dostoyevsky

Crime and punishment

THE CHOICE OF "CRIME AND PUNISHMENT" IS NOT RANDOM , I CHOOSE IT BECAUSR OF ITS COMPLEX LANGUAGE WHICH PRESENT THE OPPORTUNITY TO EVALUATE OUR MODEL TO COMPLEX TEXT.

DATASET DESCRIPTION



DATASET DESCRIPTION

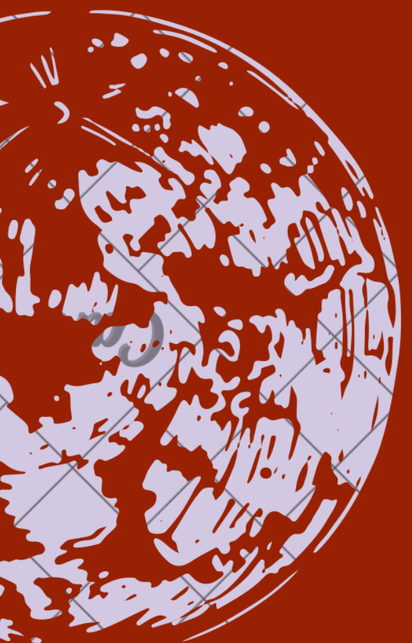
40 CHAPTERS.

22447 LINES.

211948 WORDS.

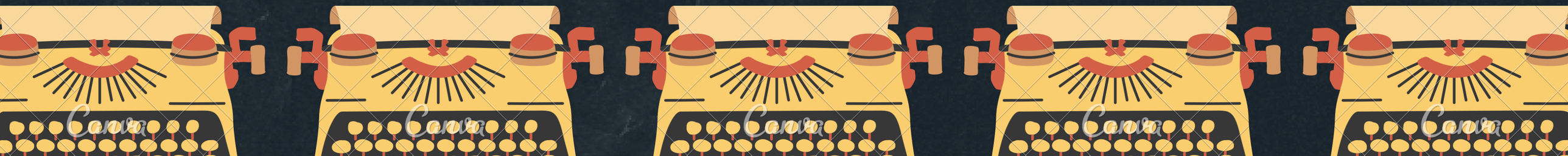
AVERAGE SENTENCE LENGTH: 17.56

technologies that i used



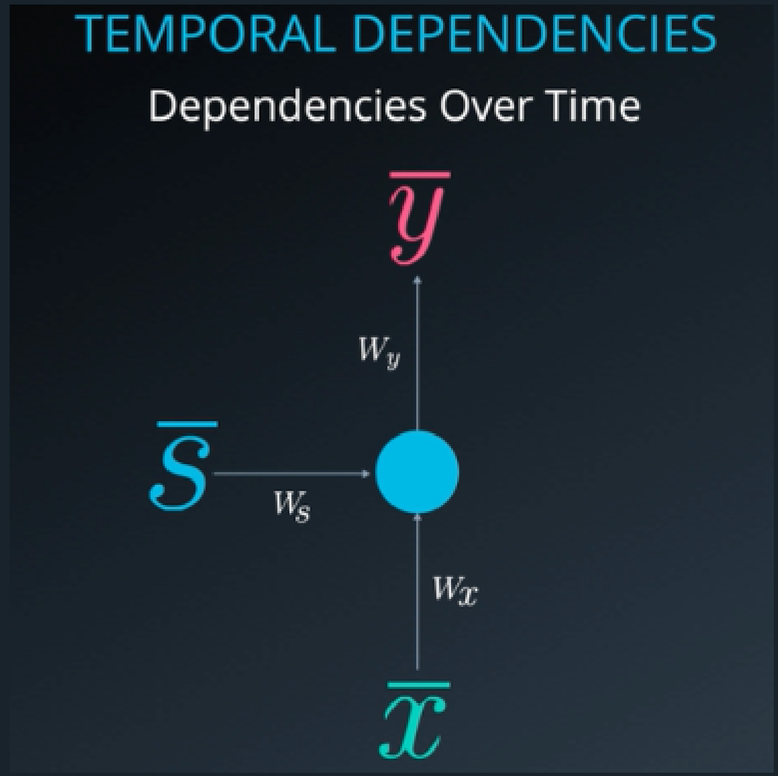
Flask

Recurrent Neural Network



Recurrent Neural Network

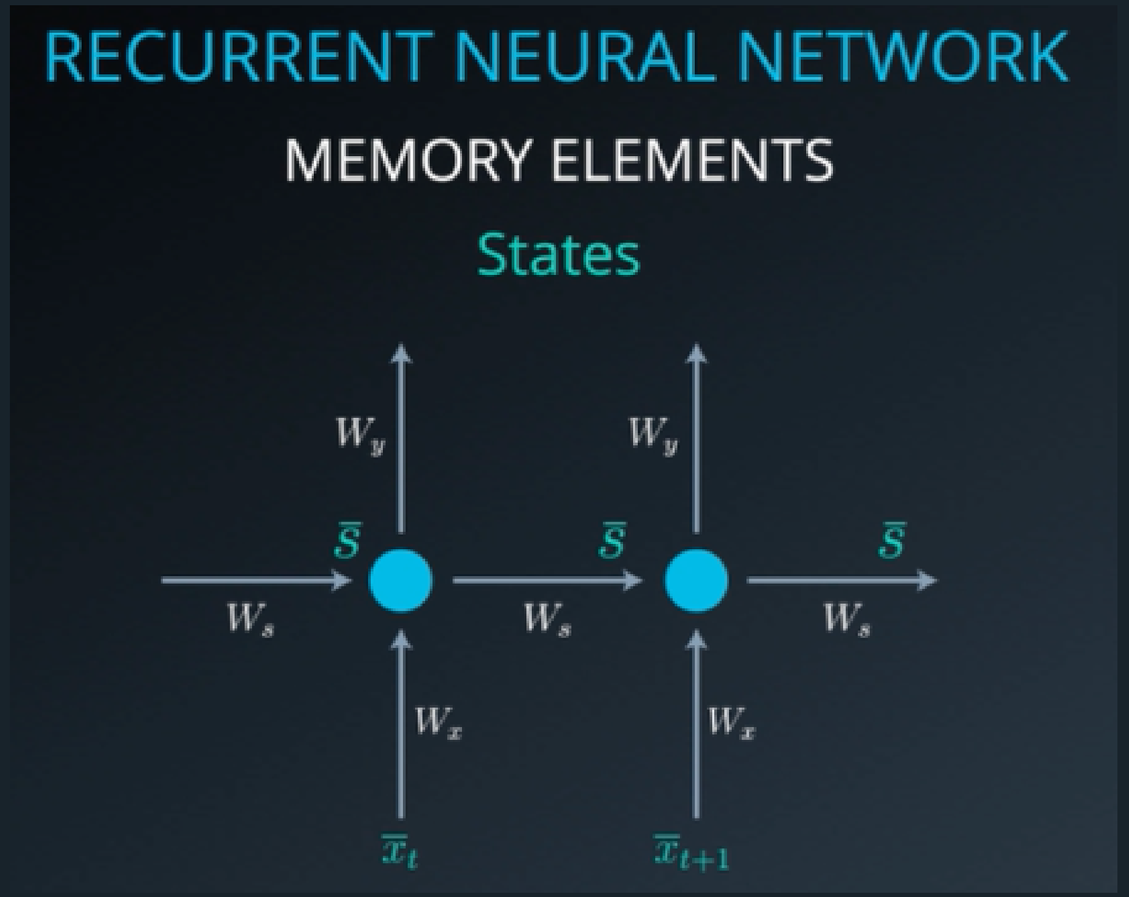
Some applications of deep-learning involve temporal-dependencies i.e. dependencies over time i.e. not just on current input but also on past inputs. RNNs are similar to feed-forward networks but in addition to memory.



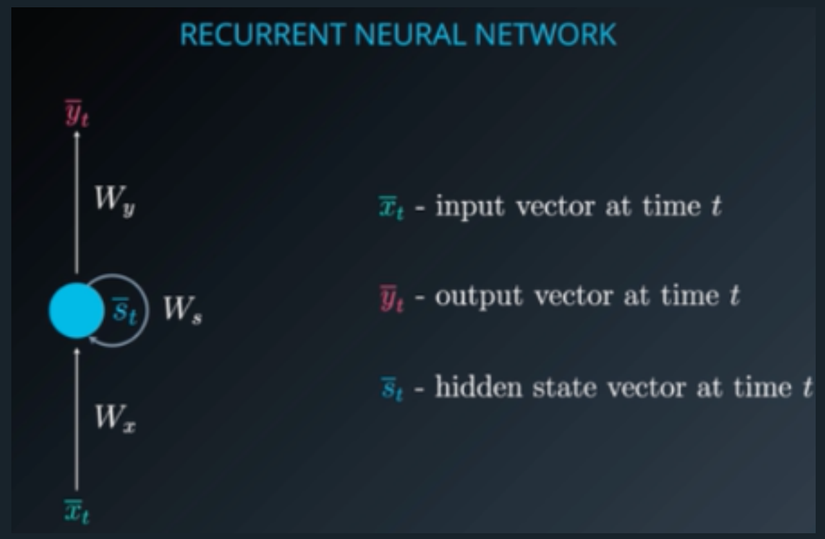
Recurrent Neural Network

In RNNs, the current output y depends not only on current input x, but also on memory element s, that takes into account past inputs.

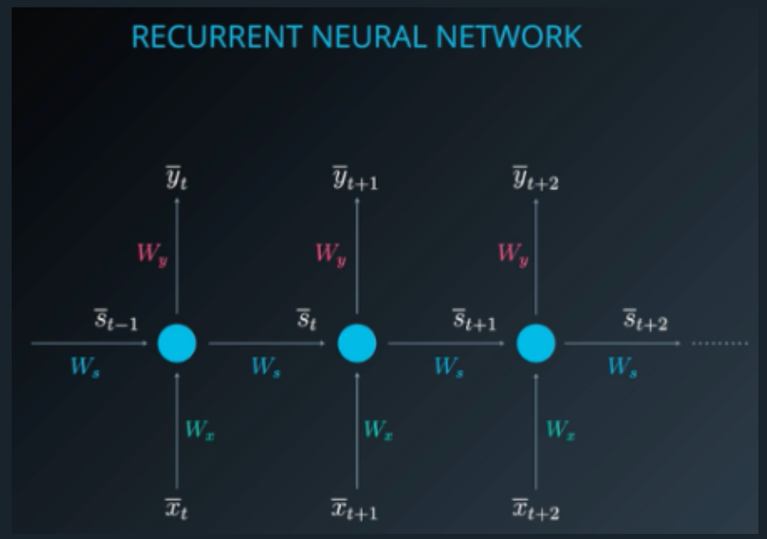
RNNs also attempt to address the need of capturing information in previous inputs by maintaining internal memory elements called States.



Structure of RNNs



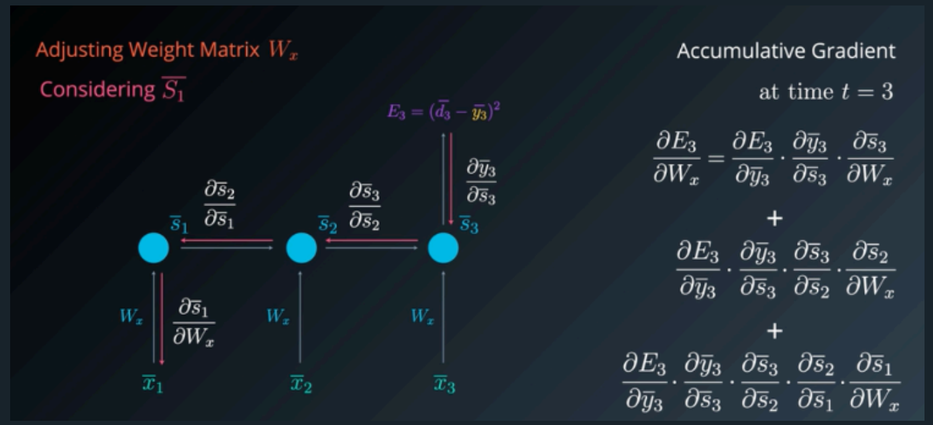
Folded RNN



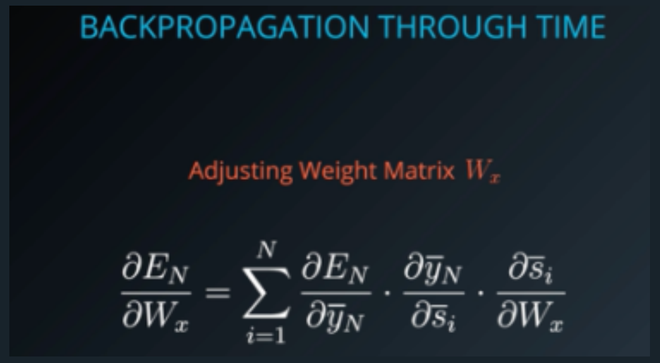
Un-folded RNN

Back Propogation Through Time

Lets look at the timestep t=3, the error associated w.r.t Wx depends on : vector S3 and its predecessor S2 and S1.



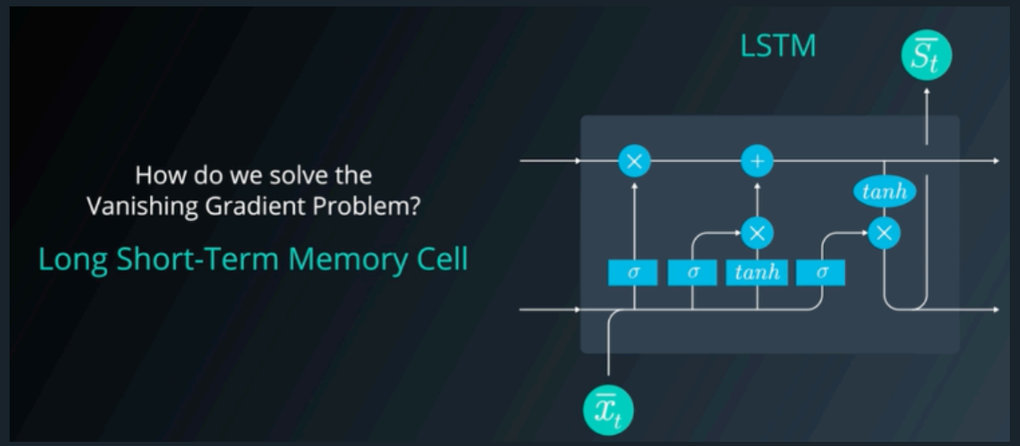
Looking at the previous pattern while calculating the accumulative gradient, we can generalize the formula for Back Propogation Through Time as follows



Drawbacks of RNNs

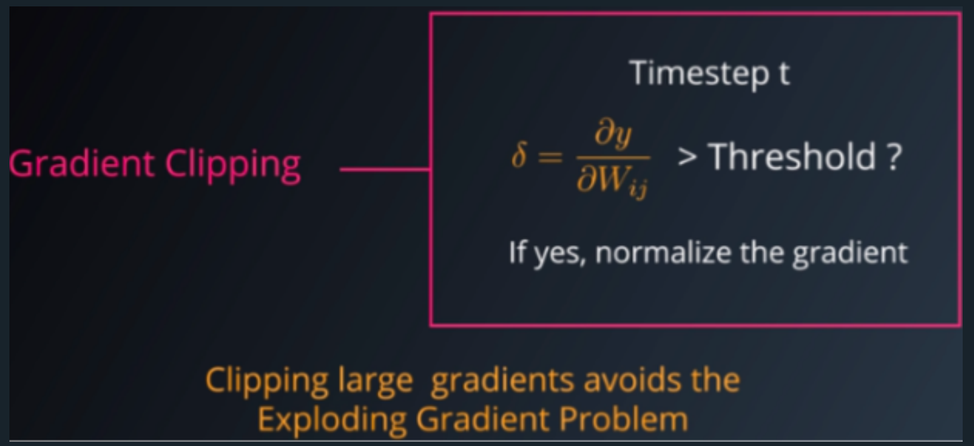
Vanishing Gradient Problem:

In RNNs, if we continue to back-propogate further after 8-9 time steps, the contributions of information (graident) keeps on decreading geometrically over time which is known as the *vanishing gradient problem*. Here is where the **LSTM** comes into picture.



Exploding Gradient Problem

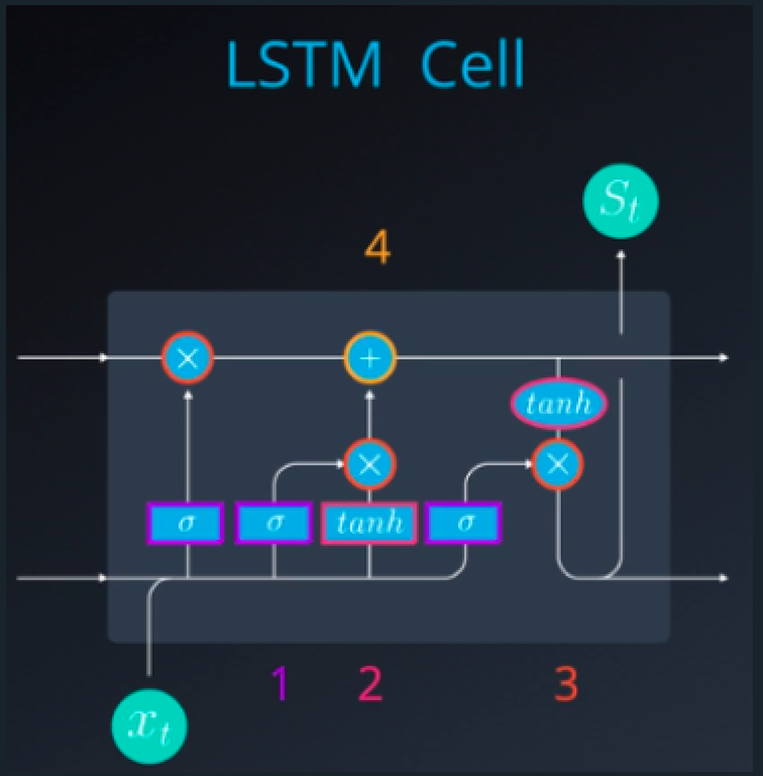
In RNNs we can also have the opposite problem, called the exploding gradient problem, in which the value of the gradient grows uncontrollably. A simple solution for the exploding gradient problem is Gradient Clipping.



Long Short Term Memory Cells

Basics of LSTM

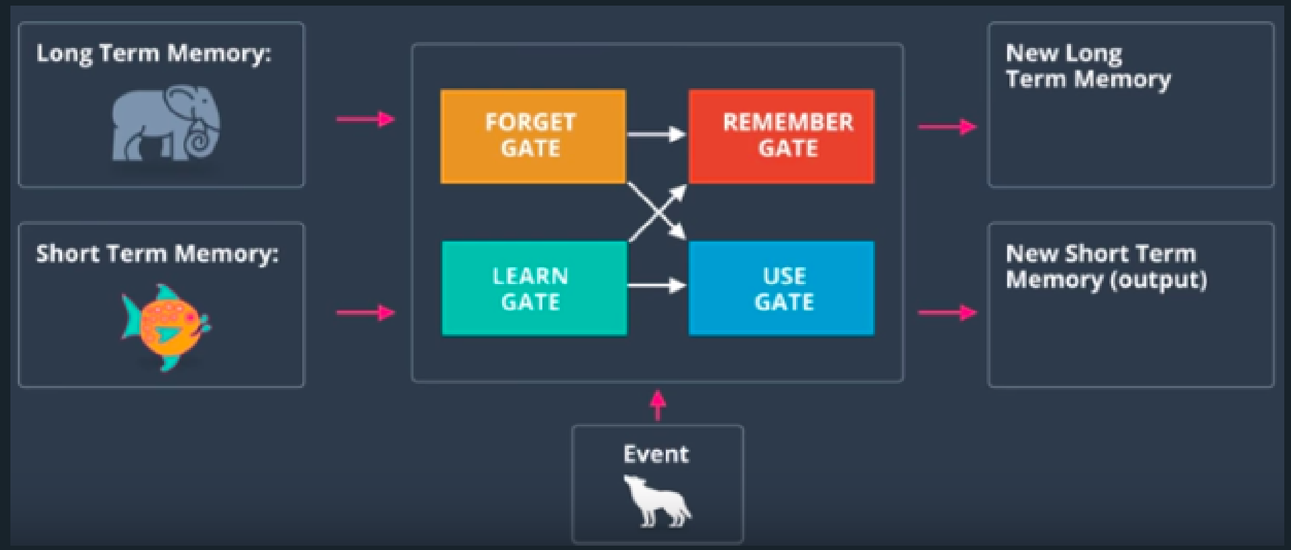
Basic RNN was unable to retain long term memory to make prediction regarding the current picture is that od a wolf or dog. This is where LSTM comes into picture. The LSTM cell allows a recurrent system to learn over many time steps without the fear of losing information due to the vanishing gradient problem. It is fully differentiable, therefore gives us the option of easily using backpropagation when updating the weights. Below is the a sample mathematical model of an LSTM cell



Long Short Term Memory Cells

How LSTMs work?

LSTM consists of 4 types of gates: 1. Forget Gate 2. Learn Gate 3. Remember Gate 4. Use Gate

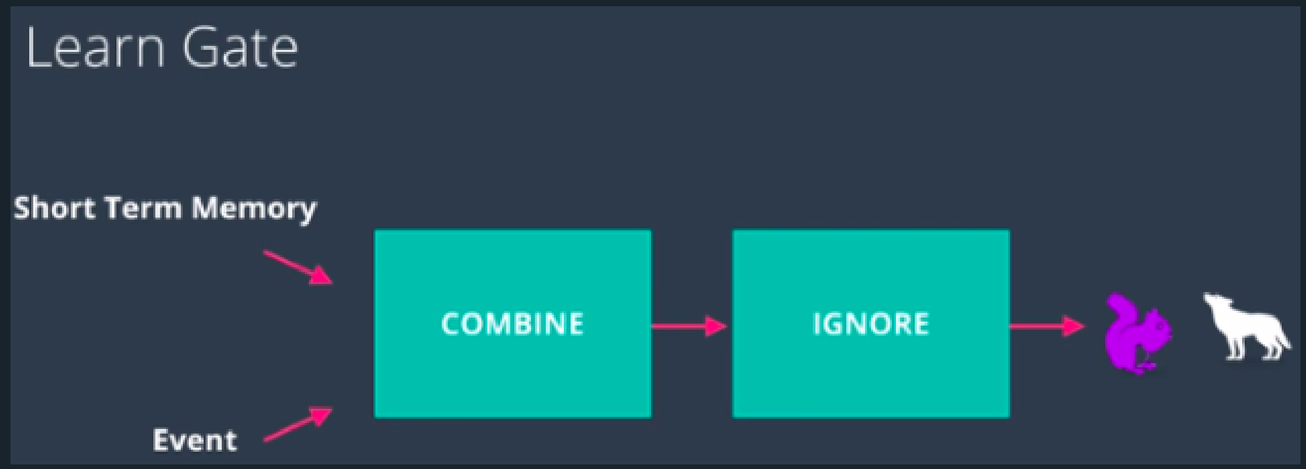


Let’s assume the following LTM = Elephant STM = Fish Event = Wolf/Dog

Long Short Term Memory Cells

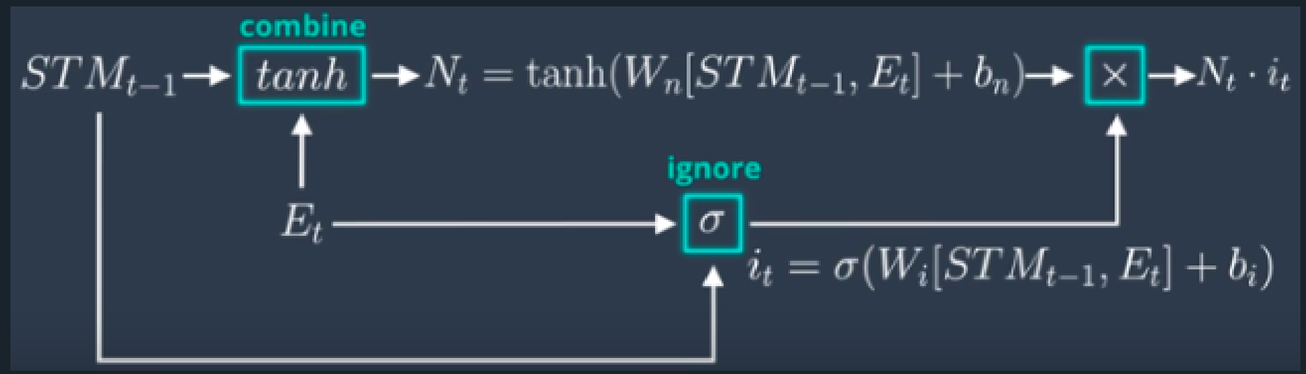
Learn Gate

Learn gate takes into account short-term memory and event and then ignores a part of it and retains only a part of information.



Mathematically Explained

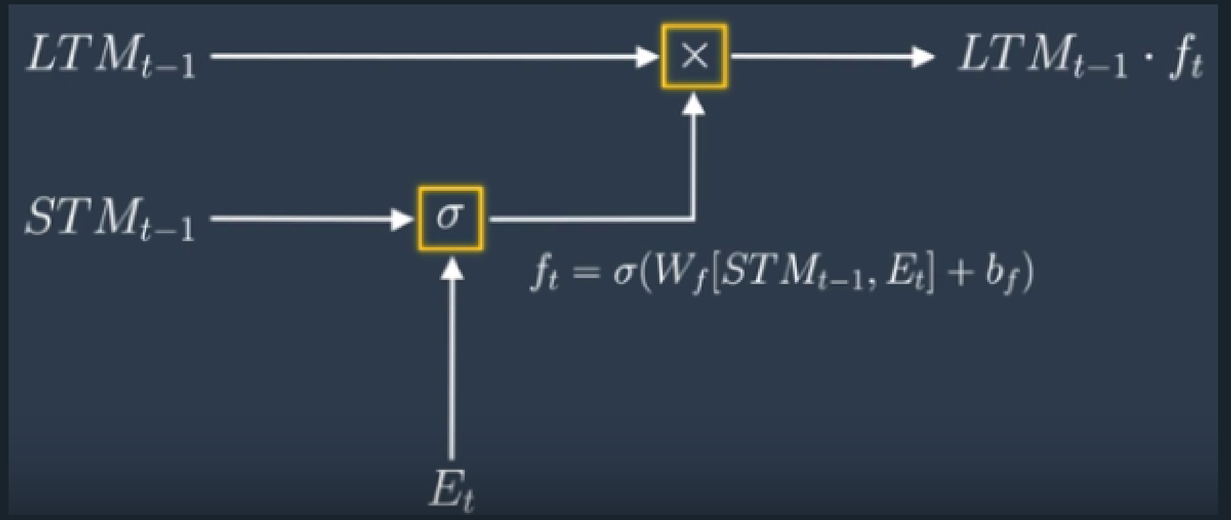
STM and Event are combined together through activation function (tanh), which we further multiply it by a ignore factor as follows



Long Short Term Memory Cells

Forget Gate

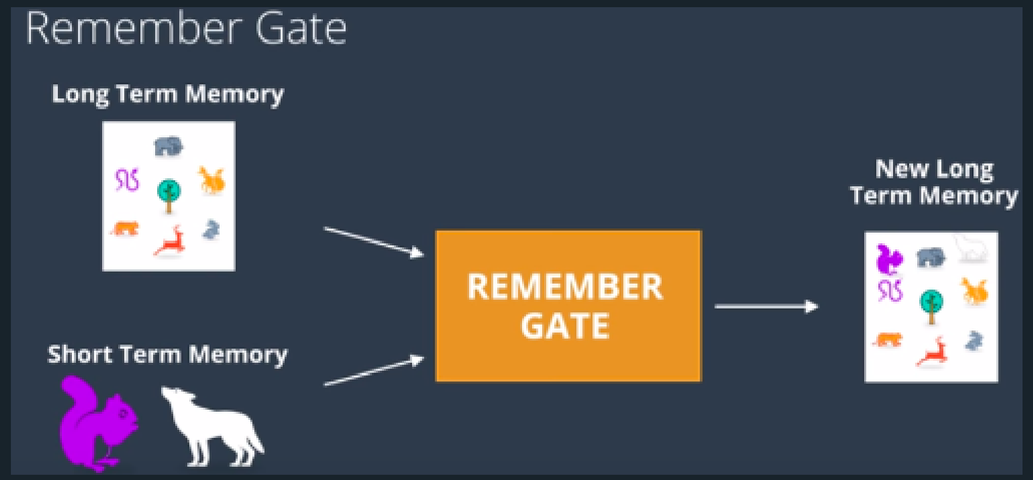
Forget gate takes into account the LTM and decides which part of it to keep and which part of LTM is useless and forgets it. LTM gets multiplied by a forget factor inroder to forget useless parts of LTM.

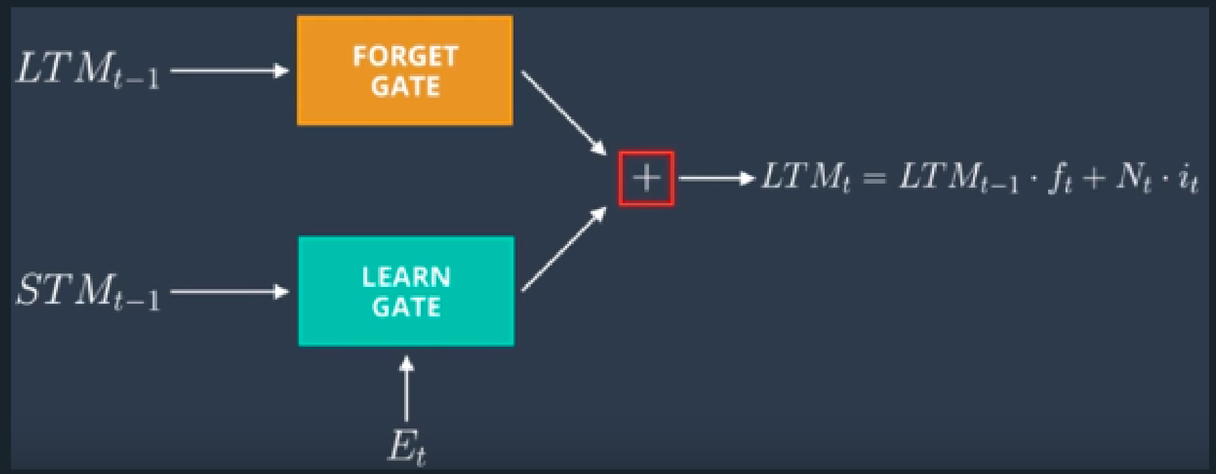


Long Short Term Memory Cells

Remember Gate

Remember gate takes LTM coming from Forget gate and STM coming from Learn gate and combines them together. Mathematically, remember gate adds LTM and STM.

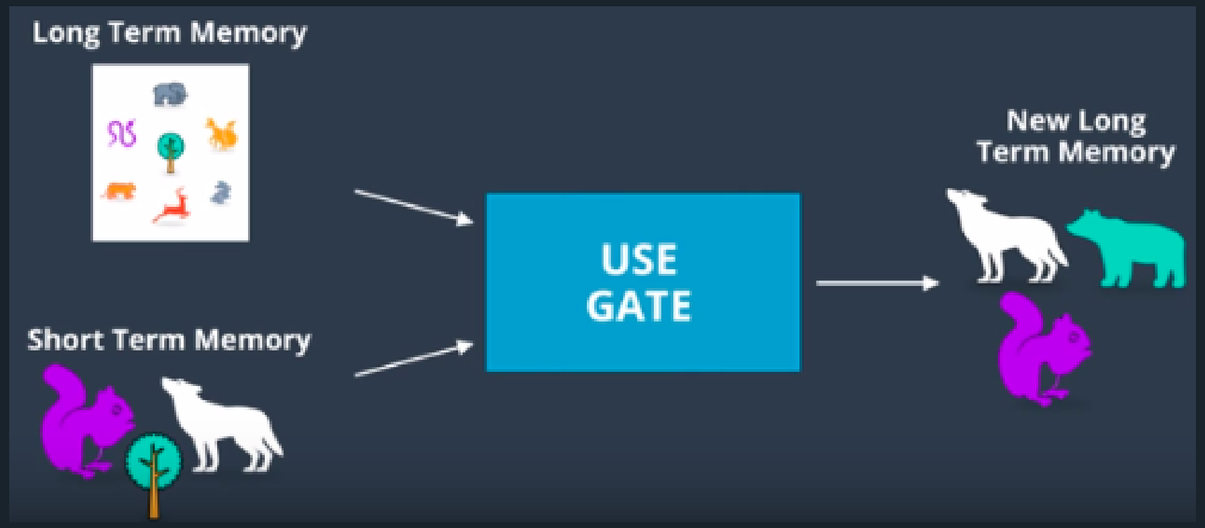


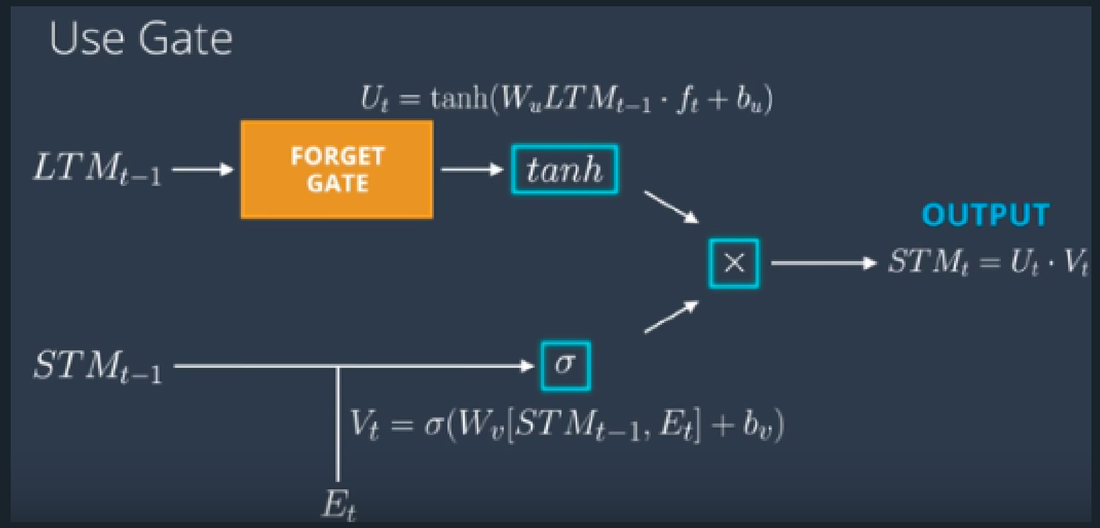


Long Short Term Memory Cells

Use Gate

Use gate takes what is useful from LTM and what's useful from STM and generates a new LTM.

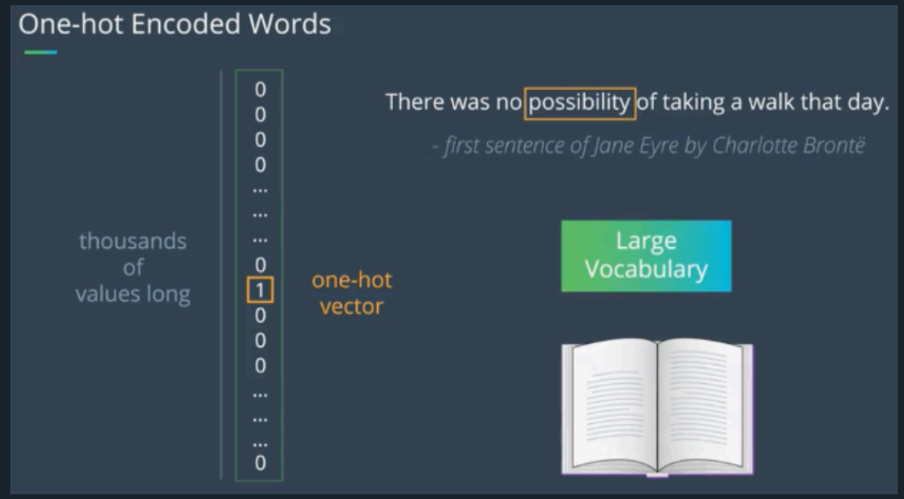




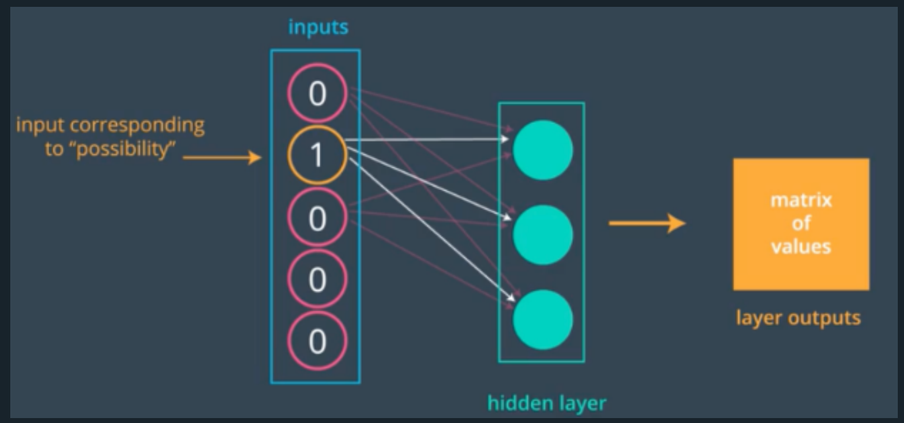
RNNs and LSTM for Text Generation

Drawbacks of one-hot encoding

Considering an example of an excert from a book containing large collection of dataset and when you use these words as an input to RNN, we can one-hot encode them, but this would mean that we will end up having giant vector with mostly zeros except that one entry as shown below:



Then we pass this one-hot encoded vector into hidden-layer of RNN and the result is a huge matrix of values most of which are zeros because of the initial one-hot encoding and this is really computaionally inefficient.



This is where ***Embeddings*** come into picture.

RNNs and LSTM for Text Generation

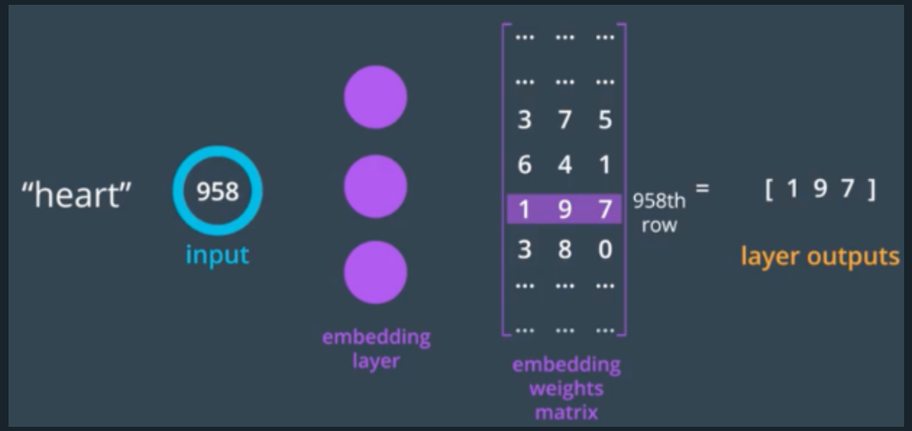
Word Embeddings

Word embeddings is a general technique of reducing the dimensionality of text data, but the embedding models can also learn some interesting traits about words in a vocabulary.

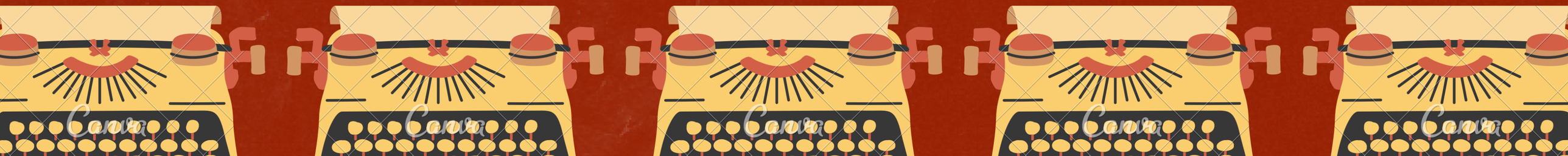
Embeddings can improve the ability of neural networks to learn from text data by representing them as lower dimensional vectors.

The idea here is when we multiply one-hot encoded vector with weight-matrix, returns only the row of the matrix that corresponds to the 1 or the on input unit.

Hence, instead of doing matrix multiplication, we use weight-matrix as a look-up table and instead of representing words as one-hot vectors, we encode each word with a unique integer.



PREPROCESSING THE DATA



Preprocessing the data

Tokenize punctuation: Replace punctuation marks with unique tokens to standardize their representation.

Split into words: Tokenize the text into individual words.

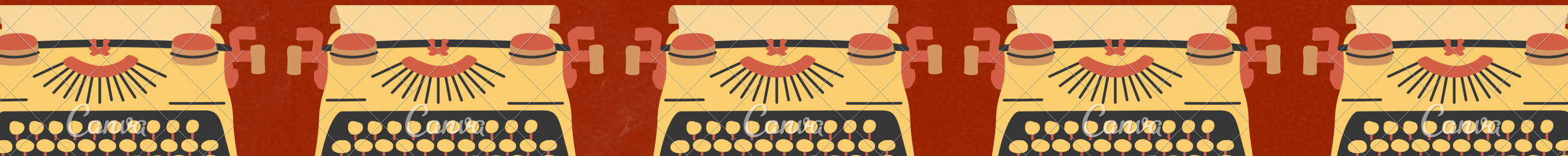
Convert text to integers: convert the entire text into a sequence of integers using the lookup table.

Convert to lowercase: Ensure consistency by converting all text to lowercase.

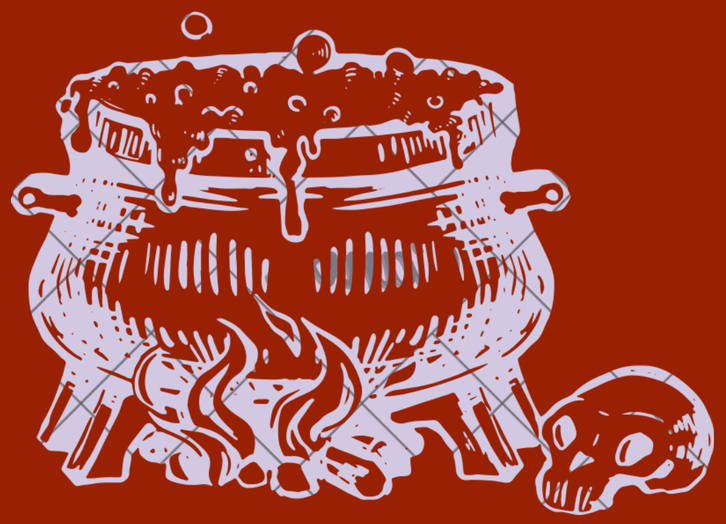
Create lookup tables: Map each unique word to a unique integer and vice versa.

Save the processed data: Store the processed data and lookup tables in a file for later use.

MODEL ARCHITECTURE



RNN



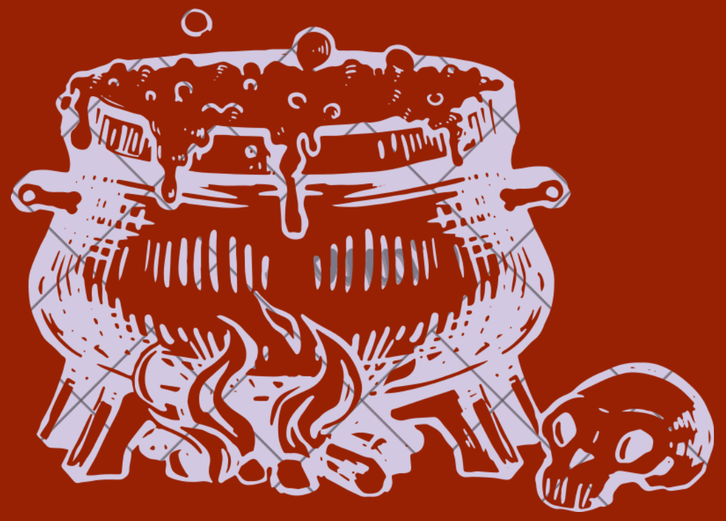
Embedding Layer	Converts input word indices into dense vectors of fixed size (embedding_dim)
LSTM Layer	Performs the core computation of the RNN by processing sequential input data and capturing long-range dependencies
Dropout Layer	Applies dropout regularization to the LSTM output to prevent overfitting
Fully Connected Layer	Transforms the LSTM output into the desired output size , which could represent the probability distribution over the vocabulary for language modeling tasks.

RNN

```python
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0.5):
        """
        Initialize the PyTorch RNN Module
        """
        super(RNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout, batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim * 2, output_size) #transform the hidden state to the desired output size.

        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.output_size = output_size

    def forward(self, x, hidden):
        """
        Forward propagation of the neural network: how the input data moves through the layers of the network.
        """
        x = self.embedding(x)
        lstm_out, hidden = self.lstm(x, hidden)
        lstm_out = self.dropout(lstm_out)
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim * 2) #ensures that the tensor is stored in a contiguous block of memory.
        out = self.fc(lstm_out)
        out = out.view(x.size(0), -1, self.output_size)
        out = out[:, -1] #the prediction for the last time step of each input sequence.
        return out, hidden

    def init_hidden(self, batch_size):
        """
        Initialize the hidden state
        """
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers * 2, batch_size, self.hidden_dim).zero_().to(device),
                  weight.new(self.n_layers * 2, batch_size, self.hidden_dim).zero_().to(device))
        return hidden
```

RNN

Version 2



2 x

Embedding Layer

Converts input word indices into dense vectors of fixed size (embedding_dim)

LSTM Layer

GRU Layer

GRUs are computationally more efficient than LSTMs because they have fewer parameters, leading to faster training times and lower memory usage.

Dropout Layer

Applies dropout regularization to the LSTM output to prevent overfitting

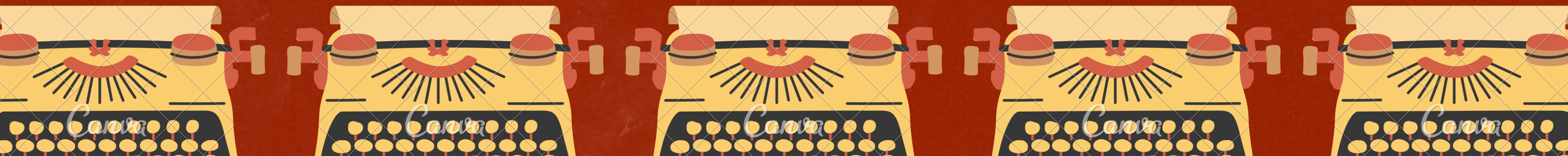
Fully Connected Layer

RNN

Version 2

```python
class RNN(nn.Module):
    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0.5):
        """
        Initialize the PyTorch RNN Module
        """
        super(RNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        self.lstm1 = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout, batch_first=True, bidirectional=True)
        self.lstm2 = nn.LSTM(hidden_dim * 2, hidden_dim, n_layers, dropout=dropout, batch_first=True, bidirectional=True)

        self.gru = nn.GRU(hidden_dim * 2, hidden_dim, n_layers, dropout=dropout, batch_first=True, bidirectional=True)

        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim * 2, output_size)

        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.output_size = output_size

    def forward(self, x, hidden):
        """
        Forward propagation of the neural network
        """
        h_lstm, c_lstm = hidden

        x = self.embedding(x)
        lstm_out, (h_lstm, c_lstm) = self.lstm1(x, (h_lstm, c_lstm))
        lstm_out, (h_lstm, c_lstm) = self.lstm2(lstm_out, (h_lstm, c_lstm))

        # Only pass the hidden state to the GRU layer
        gru_out, h_gru = self.gru(lstm_out, h_lstm)

        gru_out = self.dropout(gru_out)
        gru_out = gru_out.contiguous().view(-1, self.hidden_dim * 2)
        out = self.fc(gru_out)
        out = out.view(x.size(0), -1, self.output_size)
        out = out[:, -1]
        return out, (h_gru, c_lstm)

    def init_hidden(self, batch_size):

        Initialize the hidden state
        """
        weight = next(self.parameters()).data
        num_directions = 2  # Since the LSTM is bidirectional
        h0 = weight.new(self.n_layers * num_directions, batch_size, self.hidden_dim).zero_().to(device)
        c0 = weight.new(self.n_layers * num_directions, batch_size, self.hidden_dim).zero_().to(device)
        return (h0, c0)
```

TRAINING PROCESS



Training Process

During the training process we fine-tuned various hyperparameters to optimize the model‘s performance.
And we go through the training loop , including forward and backward propagation , loss calculation and optmization techniques.

```
# Hyperparameters
sequence_length = 10
batch_size = 128
num_epochs = 50
learning_rate = 0.001

vocab_size = len(vocab_to_int)
output_size = vocab_size
embedding_dim = 200
hidden_dim = 250
n_layers = 2
dropout = 0.5

train_on_gpu = torch.cuda.is_available()

rnn = RNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout)
if train_on_gpu:
    rnn.cuda()

optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()
```

RNN VERSION 1

```
Epoch: 1/50, Batch: 100/2129, Loss: 6.152224063873291
Epoch: 1/50, Batch: 200/2129, Loss: 6.339150428771973
Epoch: 1/50, Batch: 300/2129, Loss: 5.908645629882812
Epoch: 1/50, Batch: 400/2129, Loss: 5.277369499206543
```

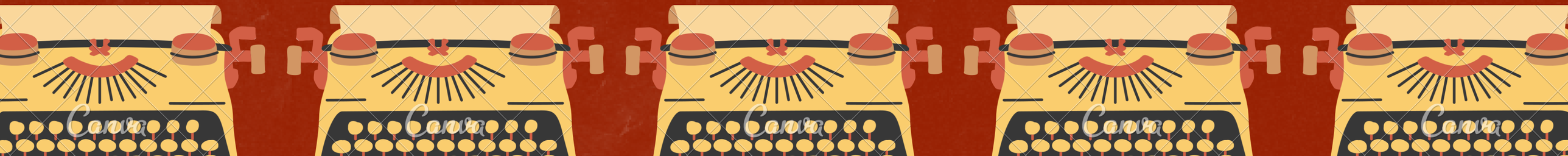
```
Epoch: 50/50, Batch: 1500/2129, Loss: 3.1983458995819092
Epoch: 50/50, Batch: 1600/2129, Loss: 3.0994338989257812
Epoch: 50/50, Batch: 1700/2129, Loss: 3.126350164413452
Epoch: 50/50, Batch: 1800/2129, Loss: 2.926783800125122
Epoch: 50/50, Batch: 1900/2129, Loss: 3.1648609638214111
Epoch: 50/50, Batch: 2000/2129, Loss: 2.8560307025909424
Epoch: 50/50, Batch: 2100/2129, Loss: 2.7161591053009033
```

RNN VERSION 2

```
Epoch: 1/50, Batch: 100, Loss: 6.487211604118347
Epoch: 1/50, Batch: 200, Loss: 6.01999490737915
Epoch: 1/50, Batch: 300, Loss: 5.801189517974853
Epoch: 1/50, Batch: 400, Loss: 5.695791940689087
Epoch: 1/50, Batch: 500, Loss: 5.5745749139785765
Epoch: 1/50, Batch: 600, Loss: 5.479124393463135
```

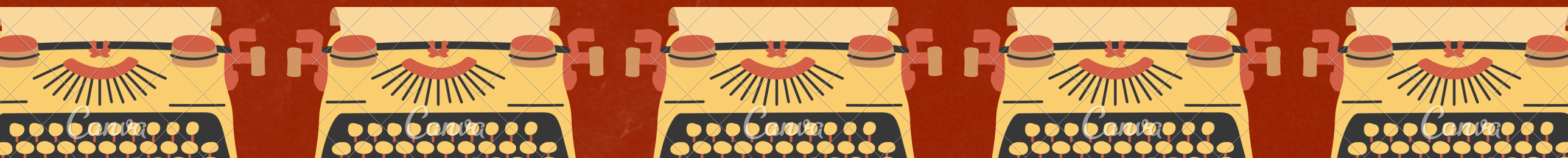
```
Epoch: 50/50, Batch: 1100, Loss: 0.9630285692214966
Epoch: 50/50, Batch: 1200, Loss: 0.9380970239639282
Epoch: 50/50, Batch: 1300, Loss: 0.9909526312351227
Epoch: 50/50, Batch: 1400, Loss: 0.9880974060297012
Epoch: 50/50, Batch: 1500, Loss: 0.9835271155834198
Epoch: 50/50, Batch: 1600, Loss: 1.0253243923187256
Epoch: 50/50, Batch: 1700, Loss: 1.043403000235577
Epoch: 50/50, Batch: 1800, Loss: 1.0363792097568512
Epoch: 50/50, Batch: 1900, Loss: 1.0732486563920975
```

TEXT GENERATION



generate text

```python
def generate(rnn, prime_ids, int_to_vocab, token_dict, pad_value, predict_len=100):
    rnn.eval()
    current_seq = np.full((1, sequence_length), pad_value)
    prime_len = len(prime_ids)
    current_seq[0, -prime_len:] = prime_ids
    predicted = [int_to_vocab[idx] for idx in prime_ids]
    hidden = rnn.init_hidden(1)

    for _ in range(predict_len):
        current_seq = torch.LongTensor(current_seq).to(device)
        output, hidden = rnn(current_seq, hidden)
        p = F.softmax(output, dim=1).data
        if train_on_gpu:
            p = p.cpu()
        top_k = 5
        p, top_i = p.topk(top_k)
        top_i = top_i.numpy().squeeze()
        p = p.numpy().squeeze()
        word_i = np.random.choice(top_i, p=p/p.sum())
        word = int_to_vocab[word_i]
        predicted.append(word)
        current_seq = np.roll(current_seq.cpu(), -1, 1)
        current_seq[0, -1] = word_i

    gen_sentences = ' '.join(predicted)
    for key, token in token_dict.items():
        ending = ' ' if key in ['\n', '(', '"'] else ''
        gen_sentences = gen_sentences.replace(' ' + token.lower(), key)
    gen_sentences = gen_sentences.replace('\n ', '\n')
    gen_sentences = gen_sentences.replace('( ', '(')
    return gen_sentences
```



Text Generator

Text

i am not alone

Length

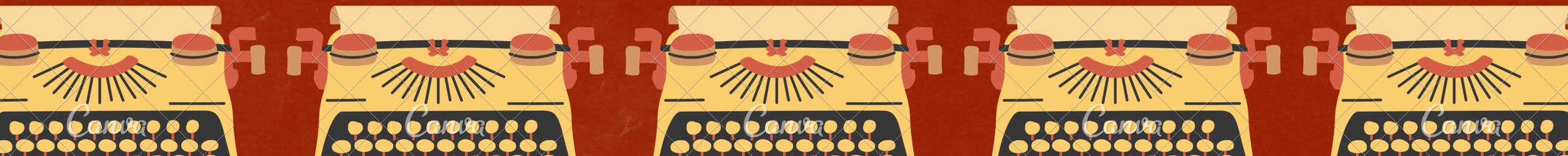
80

Generate

Generated Text:

i am not alone?’ and i shall be in prison and has not a effect on you.i know what you have done,” he said suddenly in despair.“how do you suppose that i have been expecting you in my presence and not a question to do anything for it,” he said,addressing the gentleman,crossing herself,and that he had no reason to go to his senses

EVALUATION

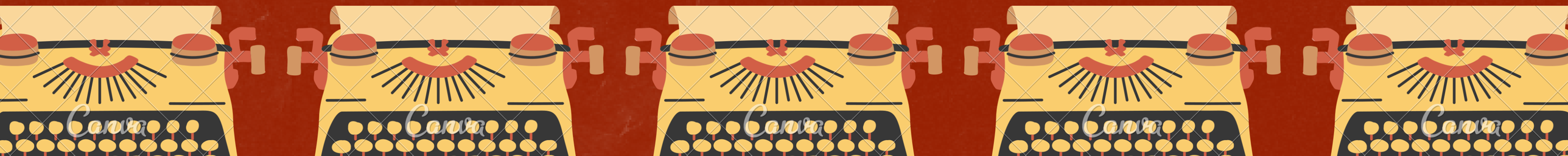


BLEU METRIC

BLEU stands for *Bilingual Evaluation Understudy*.

The metric used in comparing a candidate to one or more reference . And the output lies in the range of 0-1, where a score closer to 1 indicates good quality generations.

$$\text{BLEU} = \min\left(1, \frac{\textit{output-length}}{\textit{reference-length}}\right) \left(\prod_{i=1}^{4} \textit{precision}_i\right)^{\frac{1}{4}}$$



ROUGE METRIC

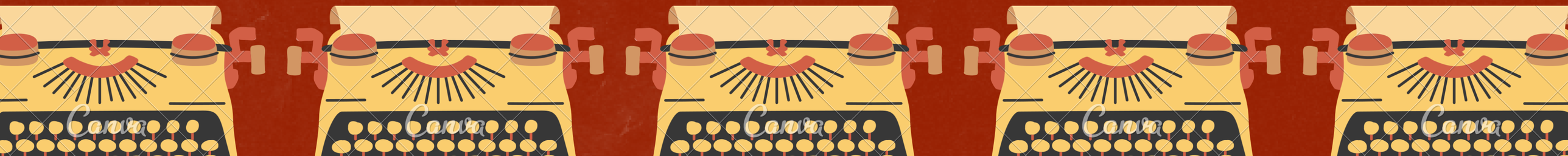
ROUGE stands for Recall-Oriented Understudy for Gisting

Rouge-1 calculates the overlap of unigram(individual word) between the candidate and reference text pieces.

Rouge-L calculates the overlap of the longest co-occurring in sequence n-grams between candidate and reference text pieces

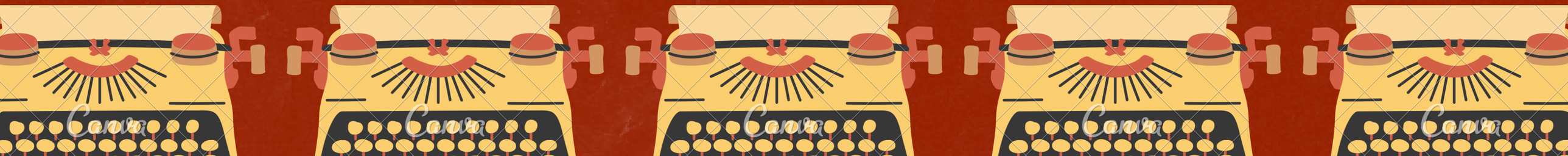
R1-recall $\longrightarrow$ $\frac{number\_of\_overlapping\_words}{total\_words\_in\_reference\_summary}$

R1-precision $\longrightarrow$ $\frac{number\_of\_overlapping\_words}{total\_words\_in\_system\_summary}$



Evaluation

```
reference_texts = sentences[100:200]

# Reverse tokenization for reference texts
reference_texts = [reverse_tokenize(ref, token_dict) for ref in reference_texts]

generated_script = generate(rnn_loaded, prime_ids, int_to_vocab, token_dict, vocab_to_int[SPECIAL_WORDS['PADDING']], gen_length)

# Reverse tokenization for generated text
generated_script = reverse_tokenize(generated_script, token_dict)

# Tokenize the generated text
generated_tokens = generated_script.split()

# Tokenize reference texts
reference_tokens = [ref.split() for ref in reference_texts]

# BLEU score
bleu_score = sentence_bleu(reference_tokens, generated_tokens)
print("BLEU Score:", bleu_score)

#ROUGE score
scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
rouge_scores = scorer.score(' '.join(reference_texts), generated_script)
print("ROUGE-1 Score:", rouge_scores['rouge1'])
print("ROUGE-L Score:", rouge_scores['rougeL'])
```



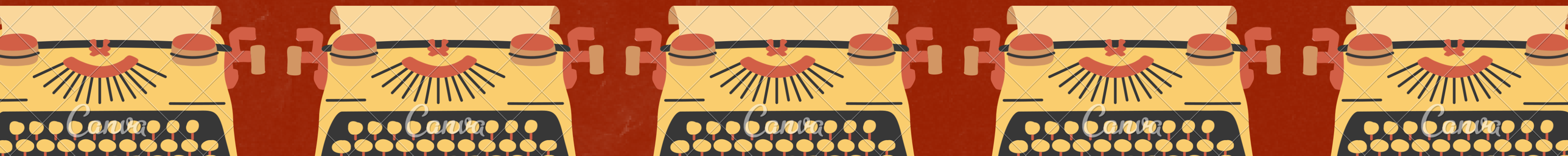
RNN VERSION 1

```
BLEU Score: 0.6504011927452344
ROUGE-1 Score: Score(precision=1.0, recall=0.012591119946984758, fmeasure=0.024869109947643978)
ROUGE-L Score: Score(precision=0.7368421052631579, recall=0.009277667329357l9, fmeasure=0.018324607329842934)
```

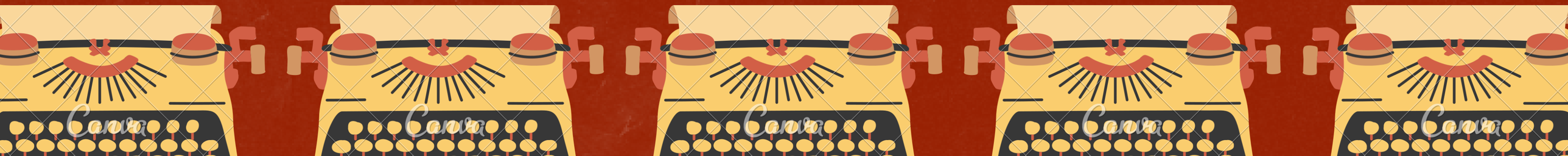
RNN VERSION 2

```
BLEU Score: 0.10944396170878806
ROUGE-1 Score: Score(precision=0.04405286343612335, recall=0.0066269052352551355, fmeasure=0.01152073732718894)
ROUGE-L Score: Score(precision=0.030837004405286344, recall=0.004638833664678595, fmeasure=0.0080645161290322 6)
```

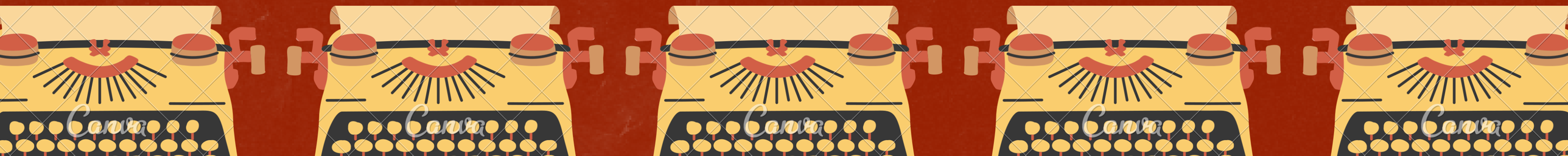
Future work

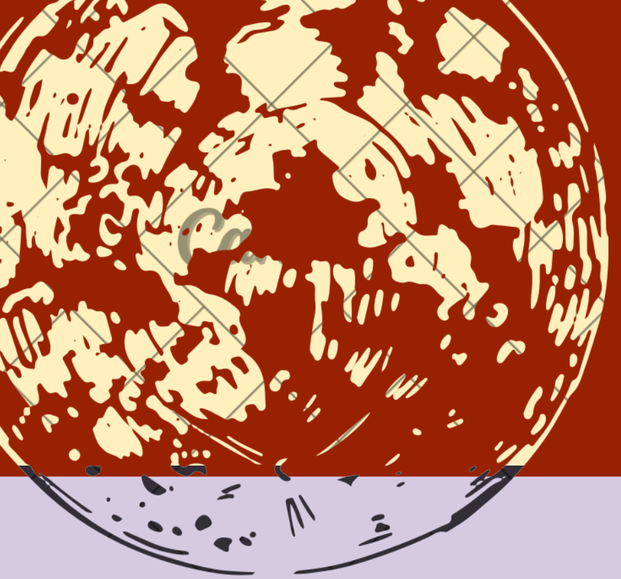


DEMO

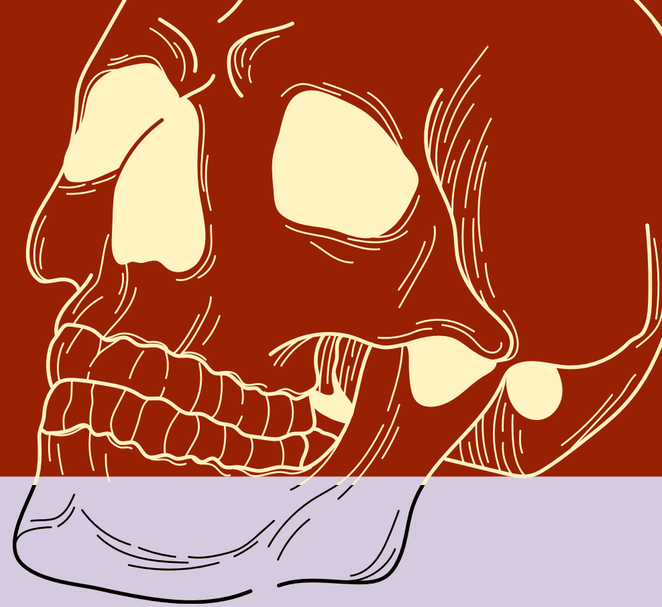


Q&A









The End

Thanks for listening!