Character-Level Text Prediction Using Long Short-Term Memory Networks

Decker Mecham MGSC 410

Abstract

This paper explores the application of Long Short-Term Memory (LSTM) networks for character-level text prediction, a task fundamental to Natural Language Processing (NLP). Character-level models predict the next character in a sequence, offering the ability to learn syntax, structure, and style directly from data. Leveraging the sequential processing capabilities of LSTMs, we train a model using a dataset of over 3 million characters. The results demonstrate that LSTM networks can generate coherent, context-aware text in the style of the training corpus. The findings underline the potential of character-level models in applications such as creative writing, automated content generation, and language modeling.

1. Introduction

Natural Language Processing (NLP) is a cornerstone of artificial intelligence, with machine translation, chatbots, summarization applications, and more (Jurafsky & Martin, 2021). While NLP traditionally operates at the word or sentence level, character-level modeling offers unique advantages. Unlike word-based models, character-based models do not require a predefined vocabulary, allowing them to generalize better for unseen data and handle languages with complex morphology.

Character-level text prediction, a specific task within this domain, focuses on predicting the next character in a sequence. While seemingly simple, this task poses significant challenges due to the

dependencies across varying input lengths. Models must grasp local patterns (e.g., syllables) and global context (e.g., sentence structure and narrative flow).

Recurrent Neural Networks (RNNs) are a natural choice for such sequential data. However, traditional RNNs are hindered by the vanishing gradient problem, making them ineffective for long-term dependencies (Hochreiter & Schmidhuber, 1997). Long-short-term memory (LSTM) networks address this limitation by introducing a gating mechanism to control the flow of information. This paper investigates the efficacy of LSTMs for character-level prediction and evaluates their ability to generate coherent and contextually relevant text.

2. Related Work

Character-level models have a long history, beginning with n-gram approaches (Shannon, 1948). These probabilistic methods predict the next character based on the preceding nnn characters but struggle with long-term dependencies due to their fixed context size.

Neural networks introduced significant advancements in sequential tasks. Bengio et al. (2003) proposed a neural probabilistic language model, paving the way for RNN-based approaches. Mikolov et al. (2010) demonstrated the power of RNNs for language modeling but highlighted their limitations with long-range dependencies. Hochreiter and Schmidhuber (1997) introduced LSTMs, which have since become a standard for sequential tasks, including speech recognition (Graves et al., 2013) and machine translation (Sutskever et al., 2014).

Karpathy (2015) popularized character-level LSTM models for text generation, showcasing their ability to mimic styles from various datasets. Our study builds on these works, focusing on training and evaluating an LSTM model on a large text corpus to highlight its practical applications.

3. Methodology

3.1 Dataset Preparation

This study used a corpus of approximately 3 million characters derived from Gutenberg.org. The text is called “War and Peace” by Leo Tolstoy. gutenberg.org/cache/epub/2600/pg2600.txt

The text data is encoded in utf-8

The dataset was preprocessed as follows:

Lowercasing: All text was converted to lowercase to reduce the size of the vocabulary as well as complexity.

Character Mapping: We created a mapping of unique characters to indices and vice versa.

Trimmed Text: Over 8,300 characters were cut from the start of the text before the body of the book started, and the last 18,830 characters were trimmed.

Sequence Generation: The text was divided into overlapping sequences of 150 characters, with the subsequent character as the target output for each sequence.

Unique Characters: The 31 unique characters were one hot encoded into a numeric representation.

```
# Step 2: Character Mapping

# Create a sorted list of unique characters in the text.
# We do this because computers cannot understand language, only numbers.
chars = sorted(list(set(text)))
char_to_int = {char: i for i, char in enumerate(chars)}  # map characters to integers
int_to_char = {i: char for i, char in enumerate(chars)}  # map integers back to characters

# Display the mappings to understand the character encoding
print(f"Unique Characters: {chars}")
print(f"Character to Integer Mapping: {char_to_int}")
```

```
Unique Characters: [' ', '!', ',', '.', '?', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
Character to Integer Mapping: {' ': 0, '!': 1, ',': 2, '.': 3, '?': 4, 'a': 5, 'b': 6, 'c': 7, 'd': 8, 'e': 9, 'f': 10,
```

```
# Step 3: Sequence Creation

# Sequence length, i.e., how many characters the model will look back to predict the next one
sequence_length = 150
sequences = []
next_chars = []



# Loop to create sequences and the corresponding target character
for i in range(0, len(text) - sequence_length):
    sequences.append(text[i:i + sequence_length])
    next_chars.append(text[i + sequence_length])

print(f"Total Sequences Created: {len(sequences)}")
# total sequences calculated as follows ,      (number of total character - sequence length) / stride



# Convert the characters in sequences and next_chars to integers
X = np.zeros((len(sequences), sequence_length), dtype=np.int32)
y = np.zeros((len(sequences)), dtype=np.int32)


for i, seq in enumerate(sequences):
    X[i] = [char_to_int[char] for char in seq]
    y[i] = char_to_int[next_chars[i]]

# Convert y to categorical format for prediction
y = to_categorical(y, num_classes=len(chars))
```

3.2 Model Architecture

To enhance the model's performance and ensure efficient learning of complex character relationships, we implemented a multi-layered LSTM architecture with additional optimizations. This architecture leverages bidirectional processing, dropout regularization, batch normalization, and a learning rate scheduler for better convergence. Below are the key components of the model:

Model Components

Embedding Layer:

The first layer maps input characters into dense 128-dimensional vectors. This allows the model to learn compact representations of the characters and their semantic relationships.

```
model.add(Embedding(input_dim=len(chars), output_dim=128, input_length=sequence_length))
```

First Bidirectional LSTM Layer:

This layer contains 256 units and processes the sequence in both forward and backward directions, enabling the model to capture context from both ends of the input sequence. Dropout (30%) is applied to reduce overfitting.

```
# Bidirectional LSTM helps capture context in both forward and backward directions
model.add(Bidirectional(LSTM(256, return_sequences=True)))
model.add(Dropout(0.3))  # Dropout to reduce overfitting
```

Batch Normalization:

A batch normalization layer ensures stable training by normalizing the previous layer's output, accelerating learning and improving convergence.

```
model.add(BatchNormalization())
```

Second Bidirectional LSTM Layer:

A second bidirectional LSTM layer with 512 units processes the sequence further to enable deeper learning. Similar to the first LSTM layer, dropout and batch normalization are applied.

```
model.add(Bidirectional(LSTM(512, return_sequences=True)))
model.add(Dropout(0.3))
model.add(BatchNormalization())
```

Third LSTM Layer:

This unidirectional LSTM layer with 256 units condenses the information learned from the previous layers into a single vector. Dropout is again applied to prevent overfitting.

```
model.add(LSTM(256, return_sequences=False))
model.add(Dropout(0.3))
```

Dense Layer:

An intermediate dense layer with 256 units and ReLU activation captures non-linear relationships in the data. Dropout is included for regularization.

```
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.3))
```

Output Layer:

The final dense layer uses SoftMax activation to output a probability distribution over all possible characters, enabling the prediction of the next character in the sequence.

```
model.add(Dense(len(chars), activation='softmax'))
```

Optimization and Compilation

We used the Adam optimizer with gradient clipping (to a norm of 1.0) to optimize training to prevent exploding gradients, which are common in deep recurrent models. The learning rate was set to 0.001. Additionally, a learning rate scheduler (ReduceLROnPlateau) was implemented to dynamically reduce the learning rate if no improvement in training loss was observed for three consecutive epochs.

```
model.compile(optimizer=Adam(learning_rate=0.001, clipnorm=1.0), loss='categorical_crossentropy', metrics=['accuracy'])

# Learning rate scheduler to reduce learning rate if there's no improvement
reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.5, patience=3, min_lr=1e-5)
```

Summary of Model Architecture

The complete model architecture is summarized as follows:

Model: "sequential_5"

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 150, 128)	3,968
bidirectional_10 (Bidirectional)	(None, 150, 512)	788,480
dropout_20 (Dropout)	(None, 150, 512)	0
batch_normalization_10 (BatchNormalization)	(None, 150, 512)	2,048
bidirectional_11 (Bidirectional)	(None, 150, 1024)	4,198,400
dropout_21 (Dropout)	(None, 150, 1024)	0
batch_normalization_11 (BatchNormalization)	(None, 150, 1024)	4,096
lstm_17 (LSTM)	(None, 256)	1,311,744
dropout_22 (Dropout)	(None, 256)	0
dense_10 (Dense)	(None, 256)	65,792
dropout_23 (Dropout)	(None, 256)	0
dense_11 (Dense)	(None, 31)	7,967

Total params: 19,141,343 (73.02 MB)
Trainable params: 6,379,423 (24.34 MB)
Non-trainable params: 3,072 (12.00 KB)
Optimizer params: 12,758,848 (48.67 MB)

Rationale Behind Design Choices

Bidirectional LSTMs: These layers improve the model's ability to capture both forward and backward dependencies in the text, which is critical for language tasks.

Dropout and Batch Normalization: These layers prevent overfitting and stabilize training, particularly in deeper architecture.

Gradient Clipping: Essential for mitigating exploding gradients, especially in recurrent models with large LSTM layers.

The architecture's depth, regularization, and bidirectionality makes it well-suited for character-level prediction tasks on large text datasets.

3.3 Training Configuration

The model was trained for 10 epochs with a batch size of 64. Early stopping was employed to prevent overfitting. The Adam optimizer (Kingma & Ba, 2014) was chosen for its adaptive learning rate capabilities, and categorical cross-entropy was used as the loss function.

3.4 Text Generation

A sliding window approach was used to generate text:

The parameters are as follows:

model: The trained neural network model.

seed_text: The initial string to begin text generation. This must match the input format the model expects (e.g., sequence length).

num_chars: The number of characters to generate.

temperature: A parameter to control the randomness of predictions. Lower values make predictions more deterministic, while higher values add randomness.

```python
import numpy as np

def generate_text(model, seed_text, num_chars=500, temperature=1.0):
    generated_text = seed_text
    for _ in range(num_chars):
        input_seq = np.array([char_to_int[char] for char in seed_text]).reshape(1, -1)
        predictions = model.predict(input_seq, verbose=0)[0]

        # Adjust predictions by temperature
        predictions = np.log(predictions + 1e-10) / temperature
        predictions = np.exp(predictions) / np.sum(np.exp(predictions))

        # Sample the next character based on adjusted probabilities
        next_index = np.random.choice(len(predictions), p=predictions)
        next_char = int_to_char[next_index]

        generated_text += next_char
        seed_text = seed_text[1:] + next_char

    return generated_text



# Testing the generation function with a seed text
seed_text = text[:sequence_length]  # Starting text to kick off the prediction
generated_text = generate_text(model, seed_text)
print("Generated Text:")
```

4. Results

4.1 Model Performance

```
Training data shape: (2473596, 150) (2473596, 31)
Testing data shape: (618400, 150) (618400, 31)
Epoch 1/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.4100 - loss: 2.0063
Epoch 1: val_loss improved from inf to 1.52762, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1801s 47ms/step - accuracy: 0.4100 - loss: 2.0063 - val_accuracy: 0.5345 - val_loss: 1.5276 - learning_rate: 0.0010
Epoch 2/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5135 - loss: 1.6277
Epoch 2: val_loss improved from 1.52762 to 1.45139, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1802s 47ms/step - accuracy: 0.5135 - loss: 1.6277 - val_accuracy: 0.5570 - val_loss: 1.4514 - learning_rate: 0.0010
Epoch 3/10
38649/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5309 - loss: 1.5643
Epoch 3: val_loss improved from 1.45139 to 1.43819, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1803s 47ms/step - accuracy: 0.5309 - loss: 1.5643 - val_accuracy: 0.5611 - val_loss: 1.4382 - learning_rate: 0.0010
Epoch 4/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5261 - loss: 1.5814
Epoch 4: val_loss improved from 1.43819 to 1.43261, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5261 - loss: 1.5814 - val_accuracy: 0.5630 - val_loss: 1.4326 - learning_rate: 0.0010
Epoch 5/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5326 - loss: 1.5588
Epoch 5: val_loss improved from 1.43261 to 1.42602, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5326 - loss: 1.5588 - val_accuracy: 0.5646 - val_loss: 1.4260 - learning_rate: 0.0010
Epoch 6/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5312 - loss: 1.5650
Epoch 6: val_loss did not improve from 1.42602
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1803s 47ms/step - accuracy: 0.5312 - loss: 1.5650 - val_accuracy: 0.5611 - val_loss: 1.4368 - learning_rate: 0.0010
Epoch 7/10
38649/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5249 - loss: 1.5864
Epoch 7: val_loss improved from 1.42602 to 1.39618, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5249 - loss: 1.5864 - val_accuracy: 0.5730 - val_loss: 1.3962 - learning_rate: 0.0010
Epoch 8/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5410 - loss: 1.5313
Epoch 8: val_loss improved from 1.39618 to 1.38359, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5410 - loss: 1.5313 - val_accuracy: 0.5762 - val_loss: 1.3836 - learning_rate: 0.0010
Epoch 9/10
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5486 - loss: 1.5042
Epoch 9: val_loss improved from 1.38359 to 1.36875, saving model to best_model.keras
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5486 - loss: 1.5042 - val_accuracy: 0.5813 - val_loss: 1.3687 - learning_rate: 0.0010
Epoch 10/10
38649/38650 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.5511 - loss: 1.4931
Epoch 10: val_loss did not improve from 1.36875
38650/38650 ━━━━━━━━━━━━━━━━━━━━ 1804s 47ms/step - accuracy: 0.5511 - loss: 1.4931 - val_accuracy: 0.5729 - val_loss: 1.3958 - learning_rate: 0.0010
Restoring model weights from the end of the best epoch: 9.
<keras.src.callbacks.history.History at 0x7f00d414d9f0>
```

A complex model with 10 epochs, 64 batch size, and 150-character sequence length.

```
Epoch 1/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 112s 12ms/step - accuracy: 0.3460 - loss: 2.2538 - val_accuracy: 0.5414 - val_loss: 1.5352
Epoch 2/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5360 - loss: 1.5518 - val_accuracy: 0.5731 - val_loss: 1.4255
Epoch 3/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 109s 12ms/step - accuracy: 0.5624 - loss: 1.4576 - val_accuracy: 0.5848 - val_loss: 1.3781
Epoch 4/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5742 - loss: 1.4105 - val_accuracy: 0.5918 - val_loss: 1.3538
Epoch 5/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5807 - loss: 1.3866 - val_accuracy: 0.5979 - val_loss: 1.3352
Epoch 6/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 109s 12ms/step - accuracy: 0.5876 - loss: 1.3632 - val_accuracy: 0.5998 - val_loss: 1.3262
Epoch 7/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5906 - loss: 1.3511 - val_accuracy: 0.6035 - val_loss: 1.3143
Epoch 8/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5915 - loss: 1.3438 - val_accuracy: 0.6050 - val_loss: 1.3073
Epoch 9/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 109s 12ms/step - accuracy: 0.5939 - loss: 1.3338 - val_accuracy: 0.6072 - val_loss: 1.3017
Epoch 10/10
9375/9375 ━━━━━━━━━━━━━━━━━━━━ 108s 12ms/step - accuracy: 0.5971 - loss: 1.3268 - val_accuracy: 0.6086 - val_loss: 1.2966
4688/4688 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.6080 - loss: 1.3001
Test Loss: 1.2965930700302124, Test Accuracy: 0.608613908290863
```

A model with much simpler architecture trained on 1/4th of the data (750,000 characters) and using 50-character sequences. At around 100 seconds per epoch, this simple model is 18x faster at training than the 1800 seconds per epoch of the more complex model.

4.2 Generated Text

The text generated will be in the form of War and Peace. Generated is in italics. I used both models above to generate text.

well, prince, so genoa and lucca are now just family estates of thebuonapartes. but i warn you, if you dont tell me that this means war, *if you still they see him. he all had taken the gay with the form for him*

well, prince, so genoa and lucca are now just family estates of thebuonaparte *as if against the russian officers were seeing the emperor and the princess and apparent your with*

well, prince, so genoa and lucca are now just family estates of thebuonaparte. *the russian distance and lettered to the count came the bottle of conversation, and even for him*

well, prince, so genoa and lucca are now just family estates of thebuonaparte *went in the countess. but it was could be an austrian regiment was noticed the staff great respectful to the princess startled in the soldiers. She*

The following text is generated with the Generative Pre-trained Transformer GPT2-xl by Open AI, which has 1.5 billion parameters and is made available through hugging face.

Well, Prince, so Genoa and Lucca are now just family estates of the Buonapartes. *Genoa's estate is now the only one left in the world.*

"——But, what about the prince, you know. He's still alive, right?"

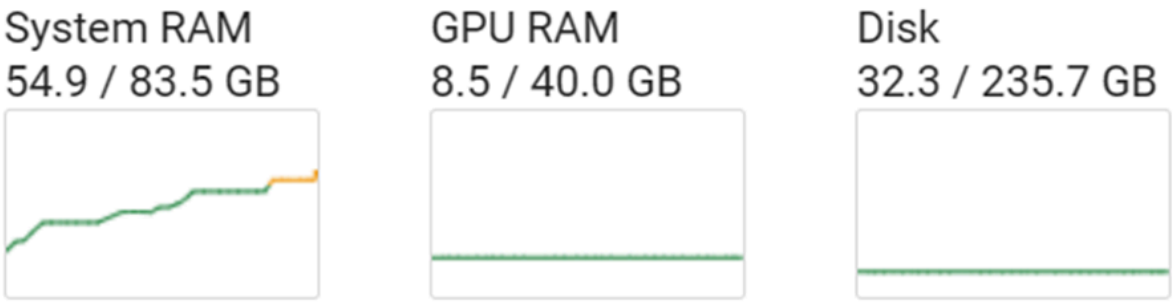
4.3 Limitations

While the model performed well, it occasionally produced repetitive or nonsensical sequences, particularly with limited training data or short input sequences.

For an NLP text generation model to perform well, it needs far more parameters than the few millions we had. Something like gpt2-xl uses 1.5 billion parameters. It took me over 5 hours to train my model using 19.1 million parameters. Our model was trained using Nvidia's a100 gpu, which cost $7,500. This resulted in a decent accuracy of 60%, which is far better than random character prediction, which would yield 1/31 or around 3% prediction accuracy. Our accuracy, being 20x better than random, is a good start for a character prediction model. The amount of text used was also small; 3 million characters may seem like a lot, but it's a small amount.

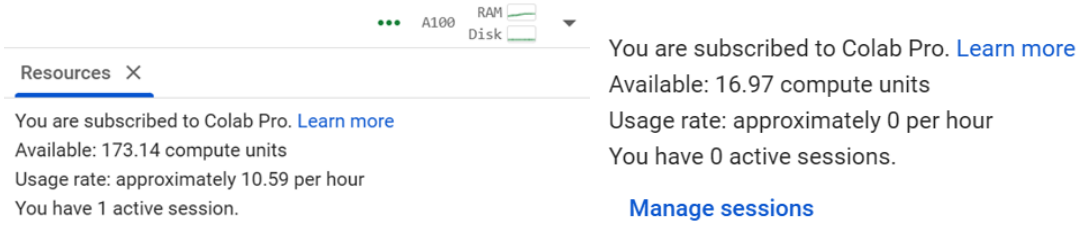
In the future, I would most likely use 10 million—100 million characters to train my model. The text we used was also complex in style, as it was from Tolstoy. This text is hard to replicate.

The model I showed here in the paper was a little worse than a few of my best models. This is because I changed the sequence length from 50 to 150. With a longer sequence length, it's more intended for word prediction, not character-by-character. Character prediction is a very local method; you only need to look at a few characters before predicting the next character.



Metrics of Nvidia's A100 gpu with 40 GB of vRAM, ≈ $7500

Some of my models used almost 80 GB of system memory, with a few crashes due to running out of memory. Batch size significantly increases memory usage.



I used over 150 computing units, equivalent to over 14 hours of A100 Training.

I would run small models on a subsection of my data to determine which models performed the best. I kept running into a barrier of about 60% accuracy. I could easily break into 70% territory with more text and computing power. My model would need at least 10 million characters in my text data to be at 70%.

5. Discussion

5.1 Strengths of LSTMs

LSTMs excel at capturing long-term dependencies, enabling them to learn patterns in text spanning multiple sentences. This is evident in the model's ability to mimic the narrative style of the training corpus.

5.2 Limitations and Challenges

- **Data Dependency**: The model's performance heavily depends on the training dataset's size and quality.
- **Computational Cost**: Training LSTMs on large datasets requires significant computational resources.
- **Lack of Creativity**: The model cannot generate novel content, only recombining learned patterns.

5.3 Future Directions

Future research could explore:

- **Transformer Models**: Models like GPT (Brown et al., 2020) could outperform LSTMs in coherence and scalability.
- **Beam Search**: Incorporating beam search into text generation could enhance the quality of predictions by exploring multiple paths.

References

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. Journal of Machine Learning Research, 3, 1137-1155.
http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

Graves, A., Mohamed, A., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. IEEE ICASSP, 6645-6649.
https://doi.org/10.1109/ICASSP.2013.6638947

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780.
https://doi.org/10.1162/neco.1997.9.8.1735

Jurafsky, D., & Martin, J. H. (2021). Speech and Language Processing (3rd ed.). Pearson.
https://web.stanford.edu/~jurafsky/slp3/

Karpathy, A. (2015). The unreasonable effectiveness of recurrent neural networks. Andrej Karpathy Blog.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
https://arxiv.org/abs/1412.6980

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). Recurrent neural network-based language model. Interspeech 2010, 1045-1048.
https://www.isca-speech.org/archive/pdfs/interspeech_2010/mikolov10_interspeech.pdf

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. Advances in Neural Information Processing Systems (NeurIPS), 27, 3104-3112. https://arxiv.org/abs/1409.3215

Brown, T., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

Available at: https://arxiv.org/abs/2005.14165