CHARACTER-LEVEL RNN TRAINED ON ALICE IN WONDERLAND

Ecem Aydemir

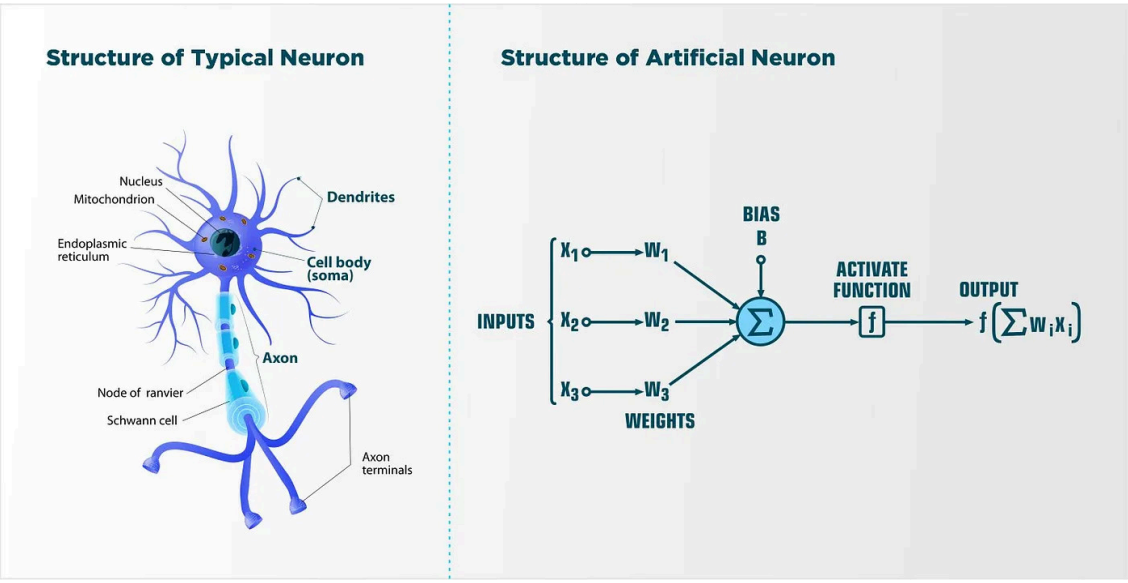
Introduction

Recurrent Neural Networks (RNNs) are a class of neural networks designed for sequence modeling, where the order of inputs carries important information. Unlike feed-forward networks that process each input independently, RNNs maintain a hidden state that can propagate information over time, making them suitable for tasks such as text generation, speech modeling, and time-series prediction.

In this project, we implement a character-level RNN to perform text generation using a corpus based on *Alice in Wonderland*. Our main goal is to explore how the decoding temperature affects the balance between determinism and creativity in generated text.

What Are Artificial Neural Networks?

Artificial Neural Networks (ANNs) are computational models composed of multiple interconnected processing units (neurons). They learn complex patterns through gradient-based optimization and represent the foundation of modern deep learning architectures. In this work, we use a Recurrent Neural Network (RNN) to learn character-level statistical patterns from English text.



What is Deep Learning?

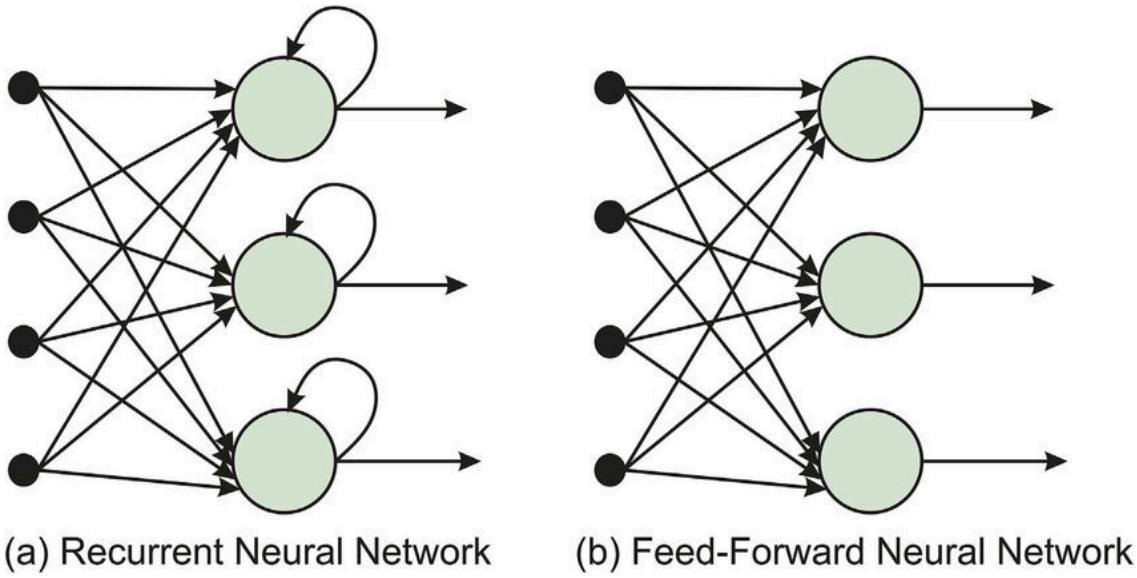
Deep learning refers to a class of machine learning algorithms that employ multi-layer artificial neural networks to automatically learn hierarchical feature representations from raw data. By stacking several processing layers, deep models extract increasingly abstract features, enabling state-of-the-art performance in domains including computer vision, speech processing, and natural language understanding.

What Are Recurrent Neural Networks (RNNs)?

Recurrent Neural Networks (RNNs) are a class of artificial neural network architectures specifically developed to model sequential data. Unlike traditional feed-forward networks that treat each input independently, RNNs incorporate recurrent connections that enable the network to preserve and reuse information from previous time steps.

Historically, the foundation of recurrent computation was introduced in the 1980s through Simple Recurrent Networks, and further formalized in the early 1990s by Jeff Elman. The key idea behind RNNs is that each output should be influenced not only by the current input, but also by prior context. This temporal dependency is essential for sequence-based tasks, especially in natural language where the interpretation of each token depends on the preceding elements.

Conventional neural networks lack this notion of memory, making them insufficient for tasks such as text modeling, speech recognition, and time-series prediction. RNNs overcome this by maintaining a hidden state that evolves over time, enabling the learning of contextual, long-range dependencies. Therefore, RNNs constitute the fundamental basis of many modern sequence modeling approaches used in natural language processing and related fields.



How Do Recurrent Neural Networks Work?

The fundamental idea behind RNNs is that at each time step the network processes two pieces of information simultaneously:

1. the current input(x_t)
2. the previous hidden state (h_{t-1})

These two pieces of information are combined using the same set of weights to obtain a new hidden state h_t. The hidden state acts like the “memory” of the network and carries past information forward. If the network is tasked with producing an output y_t, this output can also be computed from the new hidden state.

Mathematical Formulation

At each time step, an RNN produces a new hidden state by using the previous hidden state and the current input:

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$$

$$y_t = W_{hy} \cdot h_t + c$$

where:

- x_t : input at time step t
- h_{t-1} : previous hidden state
- h_t : current hidden state
- y_t : output at time step t
- $W_{xh},\ W_{hh},\ W_{hy}$: weight matrices
- $b,\ c$: bias terms
- $\tanh$: hyperbolic tangent activation function

This simple recurrent formulation follows the classical Elman network structure originally introduced in the early 1990s (Elman, 1990).

Unrolling Through Time

Since RNNs operate on data that evolves over time, the same neural network cell is reused at every time step. To better visualize this structure, an RNN is typically represented in an “unrolled” form over time.

The following illustration shows a simplified view of an RNN operating over three time steps:

$$\begin{array}{ccccc} x_1 & & x_2 & & x_3 \\ \downarrow & & \downarrow & & \downarrow \\ h_1 & \longrightarrow & h_2 & \longrightarrow & h_3 \\ \downarrow & & \downarrow & & \downarrow \\ y_1 & & y_2 & & y_3 \end{array}$$

For example, here h_2 depends not only on x_2 but also on the previous hidden state h_1. In other words, the RNN “remembers” what happened earlier and evaluates the new information together with past information. Thanks to this recurrent structure, the model can learn the context of the sequence.

In an RNN, each hidden state is computed using both the current input and the previous hidden state. Specifically, h_2 is influenced not only by the input of that step x_2 but also by the hidden state computed in the previous step h_1.

This relationship can be mathematically expressed as:

$$h_2 = \tanh(W_{xh} \cdot x_2 + W_{hh} \cdot h_1 + b)$$

Here, W_{xh} and W_{hh} are weight matrices, and b is the bias term. As can be seen, h_2 is directly built upon h_1, which allows the RNN to “remember” past information. Moreover, since h_1 already depends on x_1, h_2 implicitly carries information about both x_1 and x_2.

Advantages and Disadvantages of RNNs

Advantages

- **Naturally suited for sequential data:**
  RNNs are sensitive to the order of inputs, which makes them highly suitable for sequence-dependent data such as text, speech, and time series.
- **Efficient learning due to parameter sharing:**
  The same weights are used at every time step, reducing the number of parameters and improving generalization.
- **Contextual learning capability:**
  By using information from previous time steps, RNNs can interpret the current input within its context, which is particularly useful in language processing tasks.
- **Chained structure:**
  Because each step depends on the previous one, temporal relationships can be modeled across the sequence.

Disadvantages

- **Vanishing gradient problem:** In long sequences, gradients can become extremely small during backpropagation, causing the network to stop learning and gradually forget earlier information.
- **Exploding gradient problem:**
  When gradients grow too large, the model becomes unstable, which makes training difficult.
- **Weak at modeling long-term dependencies:**
  Traditional RNNs struggle to retain information from far earlier time steps, which is why architectures such as LSTM and GRU are needed.
- **Slow training process:**
  Since computations must follow the temporal order step by step, RNNs cannot be efficiently parallelized, resulting in slower training.

Python Implementation – Letter Prediction with an RNN

We will use a character-level RNN model to generate meaningful text. For example, when we provide a starting sequence such as "alice was sitti", we expect the model to produce a reasonable continuation.

1. Import Required Libraries

In [94]:
```
import numpy as np                                # numerical operations and array h
from tensorflow.keras.models import Sequential    # sequential model architecture
from tensorflow.keras.layers import SimpleRNN, Dense, Embedding   # RNN layers and
from tensorflow.keras.utils import to_categorical  # for one-hot encoding

import warnings
warnings.filterwarnings("ignore")
```

2. Load and Preprocess the Text

In [97]:
```
import os
import re

desktop_path = os.path.join(os.path.expanduser("~"), "Desktop")
```

```
file_path = os.path.join(desktop_path, "alice.txt")

# Read the text file
with open(file_path, "r", encoding="utf-8") as f:
    text = f.read().lower()

# Remove unwanted characters
text = re.sub(r'[^a-z .,!?]', '', text)

# Use only the first 30,000 characters
text = text[:30000]
```

3. Convert Characters to Numeric Representation

In [102…

```
# map each character to a unique numeric index

chars = sorted(list(set(text)))
char_to_idx = {c: i for i, c in enumerate(chars)}
idx_to_char = {i: c for i, c in enumerate(chars)}
```

4. Create the Training Data

In [105…

```
seq_length = 40   # we feed 40 characters and try to predict the 41st
step = 3          # take a new sample every 3 characters
X = []
y = []

for i in range(0, len(text) - seq_length, step):
    input_seq = text[i: i + seq_length]
    target_char = text[i + seq_length]
    X.append([char_to_idx[char] for char in input_seq])
    y.append(char_to_idx[target_char])

X = np.array(X)
y = np.array(y)

# np.eye() creates an identity matrix of size n x n
X_encoded = np.eye(len(chars))[X]   # (num_samples, 40, num_characters)
y_encoded = np.eye(len(chars))[y]   # (num_samples, num_characters)
```

5. Build and Train the RNN Model

In [ ]:

```
# define a simple character-level RNN model

model = Sequential()
model.add(SimpleRNN(128, input_shape=(seq_length, len(chars))))   # recurrent layer
model.add(Dense(len(chars), activation='softmax'))                # output layer ov

# compile with categorical cross-entropy since we are predicting a single character
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.summary()

# train the model
model.fit(X_encoded, y_encoded, batch_size=128, epochs=100)
```

The resulting model has approximately 24k trainable parameters, which keeps the architecture lightweight while still being expressive enough for character-level sequence modeling.

7. Define the Prediction Function

In [109…

```python
def sample(preds, temperature=1.0):
    # sampling function that adjusts the probability distribution
    # using the temperature parameter for more or less randomness
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds + 1e-8) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


def generate_sentence(model, seed_text, char_to_idx, idx_to_char,
                      seq_length=40, max_len=50, temperature=0.7):
    """
    Generate text given a seed string using the trained RNN model.
    Stops early if an end-of-sentence punctuation mark is generated.
    """
    sentence = seed_text

    for _ in range(max_len):
        # encode the current sequence
        input_seq = [char_to_idx.get(c, 0) for c in sentence[-seq_length:]]

        # pad if shorter than required length
        if len(input_seq) < seq_length:
            input_seq = [0] * (seq_length - len(input_seq)) + input_seq

        input_encoded = np.eye(len(char_to_idx))[input_seq]
        input_encoded = input_encoded.reshape(1, seq_length, len(char_to_idx))

        # predict the next character
        prediction = model.predict(input_encoded, verbose=0)
        next_index = sample(prediction[0], temperature=temperature)
        next_char = idx_to_char[next_index]

        sentence += next_char

        # optional early stopping
        if next_char in ['.', '!', '?']:
            break

    return sentence
```

7. Test Text Generation

In [112…

```python
start = "alice was beginning to get very tired "
print("Generated sentence:\n")

# lower temperature = more deterministic predictions
print(generate_sentence(model, start, char_to_idx, idx_to_char, temperature=0.2))
```

```
Generated sentence:

alice was beginning to get very tired ing wint of it cemas ice ass som nomy uthe to
the
```

In [114…

```python
start = "alice was sitting"

# try different temperature values to observe diversity in generation
for temp in [0.2, 0.5, 0.8, 1.0]:
    print(f"\nTemp {temp}:")
    print(generate_sentence(model, start, char_to_idx, idx_to_char, temperature=tem
```

```
Temp 0.2:
alice was sitting woot ais.

Temp 0.5:
alice was sitting woot ais.

Temp 0.8:
alice was sitting wout at irsceriaco verter , and sored.

Temp 1.0:
alice was sitting wor saimm!
```

These samples are not intended to be grammatically correct English sentences. Instead, they illustrate how the model captures local character statistics and how the temperature parameter controls the trade-off between repetition and randomness.

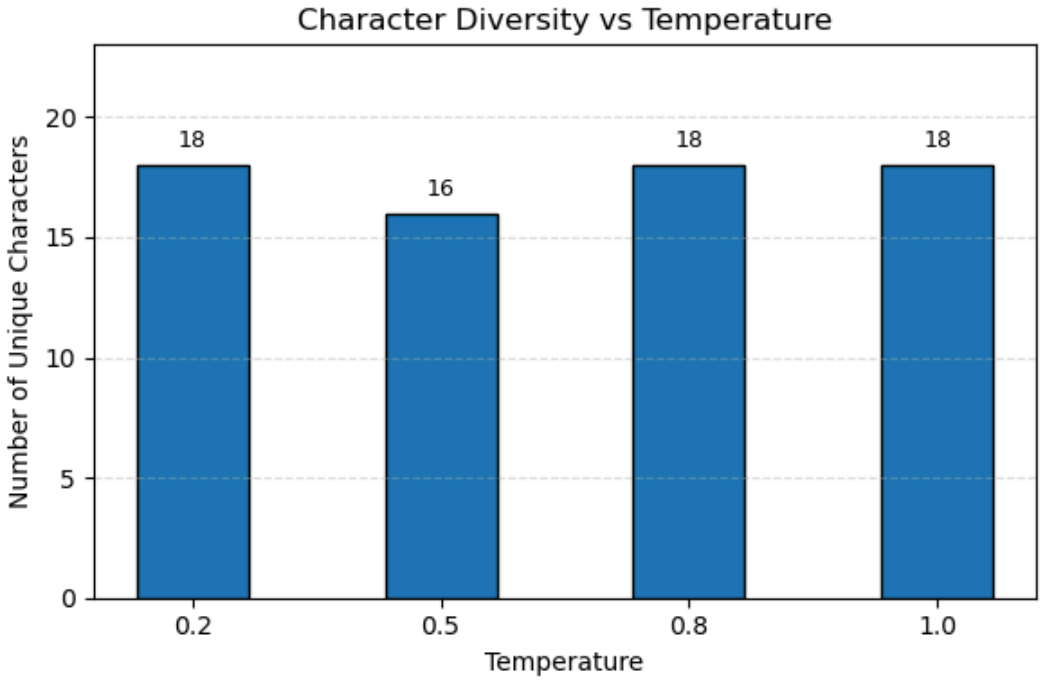
Character Diversity Across Temperature Values

We evaluate the effect of temperature on character-level diversity by counting the number of unique characters produced in the generated samples. Higher temperatures yield more diverse but less coherent text, while lower temperatures tend to repeat a smaller set of characters.

Figure – Character Diversity vs Temperature.
The bar plot shows that increasing the temperature generally increases the number of distinct characters used by the model. At low temperatures the model behaves more deterministically, whereas higher temperatures encourage exploration of less probable characters.

In [52]:
```python
import matplotlib.pyplot as plt

temps = [0.2, 0.5, 0.8, 1.0]
diversity = []

start = "alice was sitting"  # or any other seed text you prefer

# generate one sample for each temperature and measure diversity
for t in temps:
    gen = generate_sentence(model, start, char_to_idx, idx_to_char, temperature=t)
    diversity.append(len(set(gen)))
    print(f"Temp {t} -> '{gen}'  |  unique chars: {len(set(gen))}")

positions = range(len(temps))  # x-axis positions: 0,1,2,3

plt.figure(figsize=(6, 4))
plt.bar(positions, diversity, width=0.45, edgecolor="black")
plt.xticks(positions, temps)
plt.xlabel("Temperature")
plt.ylabel("Number of Unique Characters")
plt.title("Character Diversity vs Temperature")
plt.ylim(0, max(diversity) + 5)
plt.grid(axis="y", linestyle="--", alpha=0.4)

# write the numeric value on top of each bar
for x, val in zip(positions, diversity):
    plt.text(x, val + 0.5, str(val), ha="center", va="bottom", fontsize=9)

plt.tight_layout()
plt.show()
```

```
Temp 0.2 -> 'alice was sitting and proikgownthe hed neai t cer serieg imaller sh'
|  unique chars: 18
Temp 0.5 -> 'alice was sitting and in it out rer ingoun alice wadses of rice anl'
|  unique chars: 16
Temp 0.8 -> 'alice was sitting and in ieg and ho kenged an the whand andedruptat'
|  unique chars: 18
Temp 1.0 -> 'alice was sittingsand her tloter an, doon thi gained of the heut f '
|  unique chars: 18
```



Discussion

The experiments clearly show that the temperature parameter has a direct impact on the behavior of the character-level RNN. At low temperatures, the output distribution becomes sharply peaked around the most likely characters, leading to highly repetitive and predictable sequences. This improves local consistency but limits the model's ability to explore alternative continuations.

As the temperature increases, the distribution flattens and the model starts sampling less probable characters. This results in higher diversity and more surprising outputs, but also introduces spelling errors and grammatical inconsistencies. In practice, this highlights a fundamental trade-off between coherence and creativity in neural text generation, even when using a relatively simple recurrent architecture.

Conclusion

This project demonstrates how a character-level RNN can be used for sequence modeling and text generation, and how the decoding temperature shapes the behavior of the model. While the network is able to learn local character patterns from the corpus, it struggles to maintain long-range coherence, especially at higher temperature values. The results emphasize that generative performance depends not only on the neural architecture itself but also on decoding hyperparameters such as temperature, which must be chosen according to the desired balance between stability and creativity.

Future Work

As a next step, I plan to extend this project from text-based character sequences to physical sensor time-series using LSTM Autoencoders for unsupervised anomaly detection. The approach will include converting raw detector signals into sequential windows, training reconstruction models on normal operation data, and detecting anomalies based on reconstruction error thresholding. The final goal is to evaluate this methodology on open datasets such as CERN Open Data or seismic monitoring records.

References

1. Python Software Foundation. "Python Official Documentation." https://docs.python.org/
2. Chollet, F. (2021). Deep Learning with Python (2nd Edition). Manning Publications.
3. Van Rossum, G. (2023). Python Programming Language. Python Software Foundation.
4. Nielsen, M. (2015). Neural Networks and Deep Learning. Determination Press.
5. Olah, C. (2015). Understanding LSTM Networks. colah.github.io
6. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
7. Brownlee, J. (2022). A Gentle Introduction to Recurrent Neural Networks. Machine Learning Mastery.
8. Wikipedia Contributors. "Recurrent Neural Network." https://en.wikipedia.org/wiki/Recurrent_neural_network