CHARACTER-LEVEL RNN TRAINED ON ALICE IN WONDERLAND

Ecem Aydemir, 2025

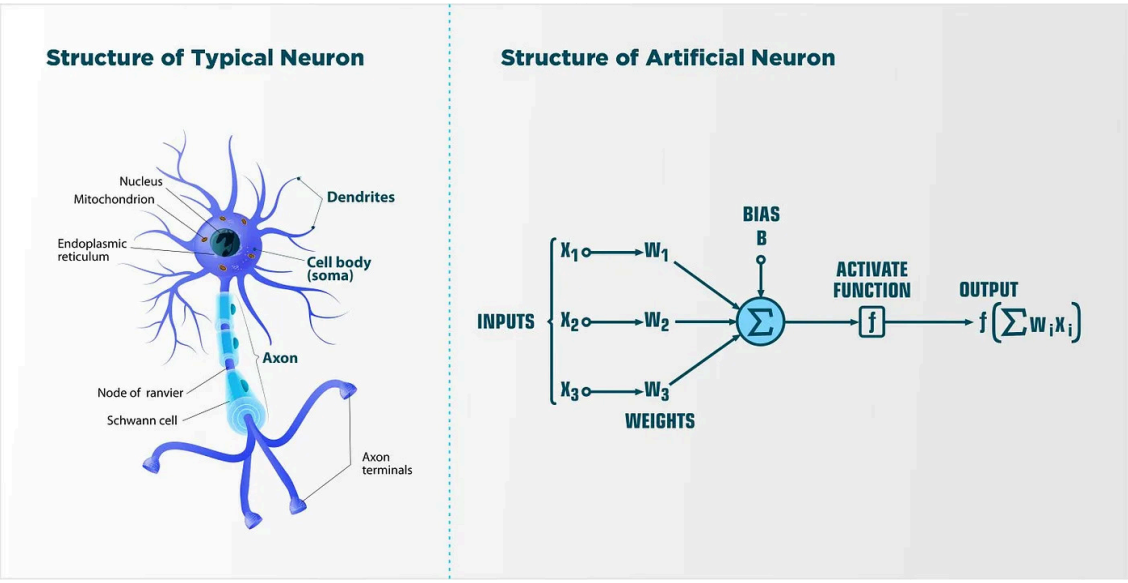
Introduction

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?

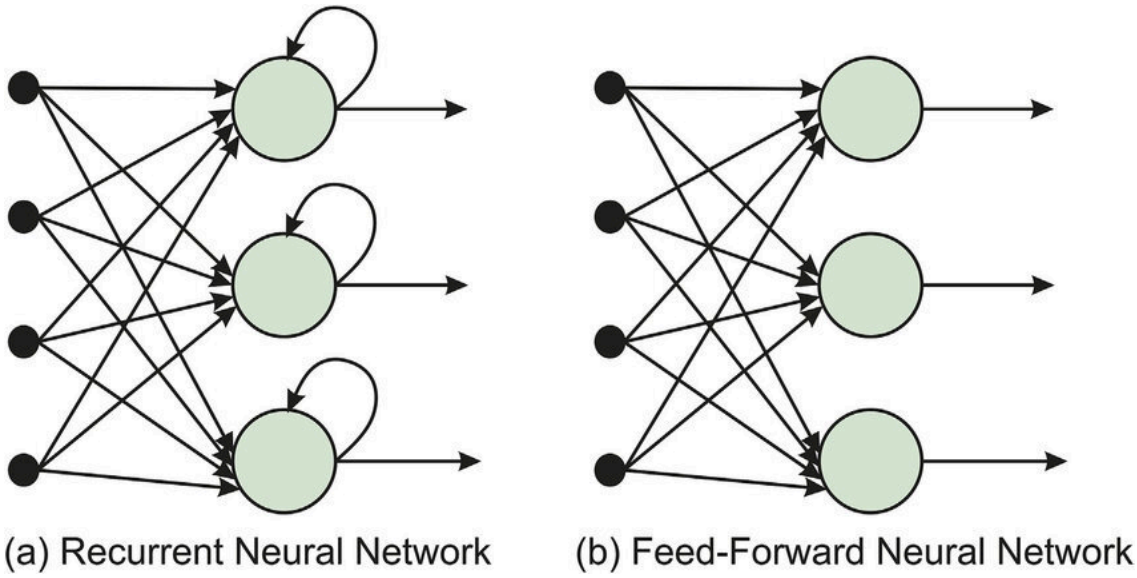
Deep learning refers to a class of machine learning algorithms that employ multi-layer artificial neural networks to automatically learn hierarchical feature representations from raw data. By stacking several processing layers, deep models extract increasingly abstract features, enabling state-of-the-art 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 [ ]:
```
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, Dropout   # RNN la
from tensorflow.keras.utils import to_categorical  # for one-hot encoding

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

2. Load and Preprocess the Text

In [21]:
```
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 [23]:
```
# 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 [25]:
```
seq_length = 100   # we feed 100 characters and try to predict the 101st
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, 100, num_characters)
y_encoded = np.eye(len(chars))[y]   # (num_samples, num_characters)
```

5. Build and Train the RNN Model

In [ ]:
```
# Define the Deep RNN model
model = Sequential()

# 1st RNN Layer (Input Layer)
# We use 256 neurons to capture more details.
# return_sequences=True is CRITICAL here. It ensures this layer outputs the
# full sequence of hidden states (one for each time step) to the next RNN layer.
model.add(SimpleRNN(256, return_sequences=True, input_shape=(seq_length, len(chars))

# Dropout Layer
# Randomly drops 20% of the connections to prevent overfitting.
model.add(Dropout(0.2))

# 2nd RNN Layer (Hidden Layer)
# This layer receives the sequence from the first layer.
# We do NOT use return_sequences=True here (default is False),
# because we only need the final hidden state to predict the next character.
model.add(SimpleRNN(128))

# Dropout Layer
model.add(Dropout(0.2))
```

```
# Output Layer
# Maps the final hidden state to the probability of each character.
model.add(Dense(len(chars), activation='softmax'))

# Compile
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.summary()

# Train
# You might want to use a callback (like ModelCheckpoint) to save the best weights.
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 [40]:
```
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=100, 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 [45]:
```python
import random

# 1. Select a random starting point from the text
# We subtract seq_length to avoid going out of bounds
start_index = random.randint(0, len(text) - seq_length - 1)

# 2. Extract a full sequence of 100 characters (Warm Start)
# Instead of using a short phrase and padding with zeros (Cold Start),
# we feed the model a complete context. This helps the RNN state 'warm up'
# and understand the syntax before predicting the first new character.
long_seed_text = text[start_index : start_index + seq_length]

print(f"--- Input Sequence (Actual Text from Book) ---\n'{long_seed_text}'\n")

# 3. Generate text using the trained model
# We set temperature=0.6 for a balance between creativity and consistency
generated_text = generate_sentence(model, long_seed_text, char_to_idx, idx_to_char,
                                   seq_length=seq_length, temperature=0.6)

# 4. Display the result
print("--- Generated Continuation ---")
# Print the full text, showing where the seed ends and generation begins
print(generated_text)
```

```
--- Input Sequence (Actual Text from Book) ---
'such a subject! our family always hated cats nasty, low, vulgar things!dont let me
hear the name aga'

--- Generated Continuation ---
such a subject! our family always hated cats nasty, low, vulgar things!dont let me
hear the name again i lowhs gaingad wat to nat but dornis dowh, dod
```

In [47]:
```python
# We use the 'long_seed_text' variable we created in the previous cell.
# This ensures the model gets a full 100-character context for the temperature test

print(f"--- Seed Text (First 50 chars) ---\n'{long_seed_text[:50]}...'\n")

# Try different temperature values to observe diversity in generation
for temp in [0.2, 0.5, 0.8, 1.0]:
    print(f"\n--- Temp {temp} ---")

    # We pass the full 100-char seed to get the best results
    gen_text = generate_sentence(model, long_seed_text, char_to_idx, idx_to_char,
                                 seq_length=100, temperature=temp)

    # We only print the NEWLY generated part to save screen space
    # (The part after the seed)
    print(gen_text[len(long_seed_text):])
```

```
--- Seed Text (First 50 chars) ---
'such a subject! our family always hated cats nasty...'


--- Temp 0.2 ---
in the mouse come paot then whe wank ow yor ang or

--- Temp 0.5 ---
in the mouse she wat she cause on te the bound ghe

--- Temp 0.8 ---
in the rould the shink.

--- Temp 1.0 ---
in the moure, wom, she soute as the want for ur th
```

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.

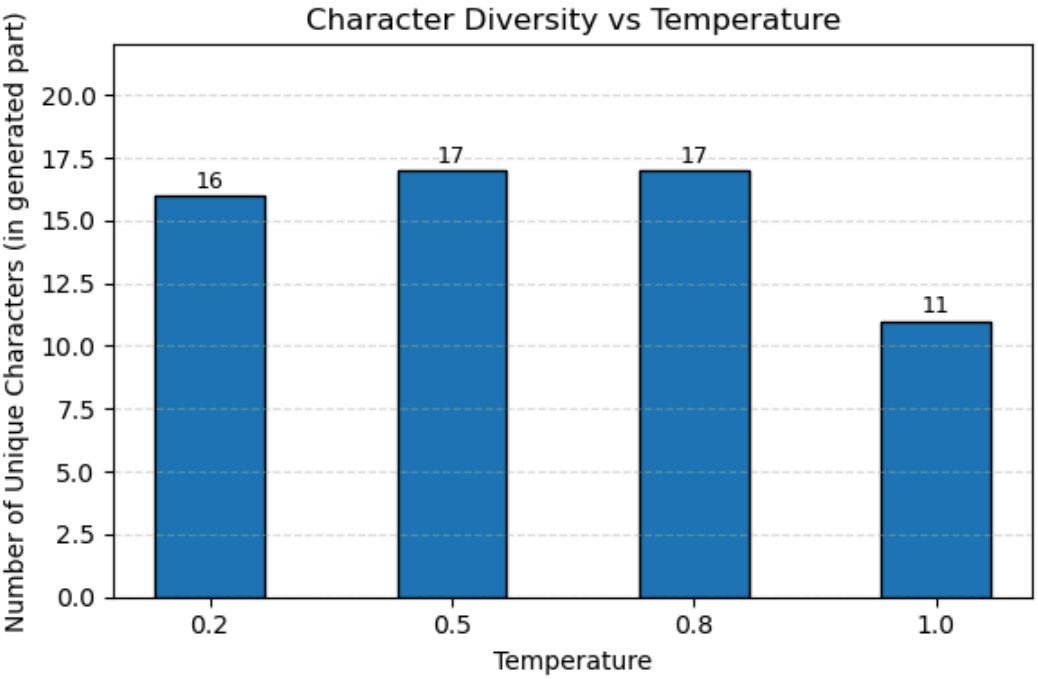
Character Diversity Across Temperature Values

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 [74]:
```
import matplotlib.pyplot as plt

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

# Use the 'long_seed_text' variable (100 chars) from the book to ensure a valid con
start_seed = long_seed_text

# Generate one sample for each temperature and measure diversity
for t in temps:
    # We must pass seq_length=100 to match the training configuration
    gen = generate_sentence(model, start_seed, char_to_idx, idx_to_char, seq_length

    # We slice the string to analyze ONLY the newly generated characters, ignoring
    # This provides a more accurate measure of the model's creative diversity.
    generated_part = gen[len(start_seed):]

    unique_count = len(set(generated_part))
    diversity.append(unique_count)

    print(f"Temp {t} -> '{generated_part}'  |  unique chars: {unique_count}")

# Plotting the results
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 (in generated part)")
plt.title("Character Diversity vs Temperature")

# Adjust y-axis limit for better visibility
if len(diversity) > 0:
    plt.ylim(0, max(diversity) + 5)

plt.grid(axis="y", linestyle="--", alpha=0.4)

# Annotate the numeric value on top of each bar
for x, val in zip(positions, diversity):
    plt.text(x, val + 0.1, str(val), ha="center", va="bottom", fontsize=9)

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

```
Temp 0.2 -> 'in the mouse come paot then bory hat her mane the '  |  unique chars:
16
Temp 0.5 -> 'in the rould there wathed amesand a sover ange sam'  |  unique chars:
17
Temp 0.8 -> 'in i low the poonver, and she she was niw.'  |  unique chars: 17
Temp 1.0 -> 'in the foule!'  |  unique chars: 11
```



Discussion & Conclusion

In this experiment, we upgraded the baseline SimpleRNN model to a **Deep RNN architecture** by stacking two recurrent layers (256 and 128 units) and introducing **Dropout regularization** to prevent overfitting. We also increased the **sequence length to 100**, allowing the model to capture longer-term dependencies and context compared to the initial 40-character window.

Key Observations:

1. **Context Awareness:** By using a 100-character warm-start (instead of zero-padding), the model successfully demonstrated semantic association (e.g., associating "cats" in the seed text with "mouse" in the generated output), proving it learned local word associations beyond just character probabilities.
2. **Temperature & Structure:** The quantitative analysis (Figure above) revealed a critical trade-off. While moderate temperatures (0.5 - 0.8) maintained a balance between diversity and coherence, the highest temperature (1.0) led to structural instability. As seen in the generated samples, high entropy caused the model to predict premature punctuation (e.g., `'in the foule!'`), triggering the early stopping mechanism and paradoxically reducing the unique character count.

Conclusion: This project highlights that while character-level RNNs can learn local syntax and spelling patterns, balancing creativity (high temperature) with structural integrity requires careful tuning. The transition to a Deep RNN architecture significantly improved the coherence of the generated text compared to shallow networks.

Future Work

Building upon the sequence modeling fundamentals established in this character-level RNN project, the next phase of research focuses on **unsupervised anomaly detection** in industrial time-series data using the **NASA Bearing Dataset**.

The planned methodology includes:

- **Architecture Upgrade:** Transitioning from SimpleRNN to **LSTM Autoencoders** to effectively capture long-term temporal dependencies in vibration data without suffering from the vanishing gradient problem.
- **Unsupervised Learning:** Training the model exclusively on "normal" operational data to learn the distribution of healthy bearing signals.
- **Anomaly Detection:** Implementing a threshold-based detection mechanism using **Reconstruction Error (MSE)**. The model will identify anomalies when the reconstruction error exceeds a statistically defined threshold, indicating a potential mechanical failure.

References

1. **Elman, J. L. (1990).** Finding structure in time. *Cognitive Science*, 14(2), 179-211.
2. **Chollet, F. (2021).** *Deep Learning with Python* (2nd Edition). Manning Publications.
3. **Goodfellow, I., Bengio, Y., & Courville, A. (2016).** *Deep Learning*. MIT Press.
4. **Carroll, L. (2008).** *Alice's Adventures in Wonderland* [EBook #11]. Project Gutenberg. Available at: gutenberg.org
5. **Olah, C. (2015).** Understanding LSTM Networks. *colah.github.io*. Available at: colah.github.io
6. **Brownlee, J. (2022).** A Gentle Introduction to Recurrent Neural Networks. *Machine Learning Mastery*.
7. **Van Rossum, G. (2023).** Python Programming Language. Python Software Foundation. Available at: docs.python.org
8. **TensorFlow Developers.** Recurrent Neural Networks (RNN) with Keras. *TensorFlow Core Documentation*. Available at: tensorflow.org