HyperionDev

Recurrent Neural Networks

Task

Visit our website

Introduction

Traditional feedforward networks, while versatile, struggle with sequential data where context and order are crucial. This lesson dives into neural network architectures designed specifically for sequential data such as time series and natural language. By the end, you'll grasp the mechanics and benefits of Recurrent Neural Networks (RNNs) and be ready to deploy them in practical applications.

Let's first see why traditional feedforward networks are not recommended for processing sequenced data.

Limitations of feedforward networks

Even though the feedforward architectures show a wide range of applications, from image recognition to simple classification tasks, there are significant limitations in many cases. Despite their utility, these networks show constraints when it comes to processing sequential data that is used for tasks like language processing, time series analysis, and more.

Sequential data, by its nature, carries information in the order and context of its elements. For instance, the meaning of a sentence in a conversation or the pattern of stock prices over time. Feedforward networks, however, process each input independently, without any memory of past inputs. This lack of memory is like reading a book but forgetting each word as soon as you move on to the next. Without the ability to remember previous inputs, feedforward networks struggle to understand the context or the sequential dependencies that give meaning to this type of data.

In many real-world problems, there are sequences where each piece of information is not just relevant in isolation but also in how it connects to previous and future data. Consider trying to predict what happens next in a video—without remembering previous scenes each frame seems unrelated, making coherent understanding impossible. Similarly, in natural language processing, the meaning of a sentence depends not only on the words used but also on their order and the context established by preceding sentences.

For example, in language processing, the phrase "I have a black cat" relies heavily on the sequence of words to convey meaning. A feedforward network might recognise words individually but fails to understand the order in which they appear, thus missing the narrative thread that connects them. Similarly, in financial forecasting, the significance of a sudden drop in stock prices can only be understood in the context of previous price movements, which feedforward networks are not equipped to handle.

Take note

The need for context and memory in neural networks is not just a technical requirement but a fundamental aspect of mimicking human-like understanding and reasoning. Humans use context and memory extensively to make decisions, learn from past experiences, and understand language.

For neural networks to perform similarly sophisticated tasks, they must go beyond the capabilities of traditional feedforward architectures and employ mechanisms that can hold onto past information, allowing them to make informed predictions and decisions based on a continuous stream of data. This need leads to the exploration and adoption of more advanced neural network designs, such as RNNs, which include memory components to address these limitations.

Recurrent Neural Networks (RNNs)

During a learning phase, recurrent models learn to efficiently represent historical tendencies and patterns directly from data. The hidden layer in an RNN captures all prior history, not just the last `n-1` words, allowing the model to theoretically represent longer context patterns. In addition, recurrent architectures can represent more complex patterns in sequential data. For instance, patterns that depend on words appearing at variable positions in a sentence can be more efficiently encoded by a recurrent architecture. This is because the model can retain specific words within the state of the hidden layer, whereas a feedforward model would need distinct parameters for each potential position of a word in a sentence. This requirement not only increases the total number of parameters in the model but also the amount of training data needed to learn these patterns.

RNNs were foundational to state-of-the-art Natural Language Processing (NLP) applications before **transformers** emerged as the dominant architecture. The concept of "recurrent" was first named by Rumelhart et al. in 1985, even though its foundation was mentioned by Minsky & Papert in 1969. RNNs process sequences of inputs, such as words in NLP, one at a time. With each new input, the network incorporates a hidden representation from the previous step—this is the recurrent connection. This setup allows the network to retain information from all previous inputs, culminating in a final output that reflects the entire input sequence. This capability is important for NLP tasks like classification or translation. RNNs are also used in decoding tasks, where generated tokens are fed back into the model as the next sequence input. An example of this is

PixelRNN by Van den Oord et al. (2016), which uses RNNs to create an autoregressive model for images.

An example of RNNs is presented in Figure 1. In this setup, word embeddings are fed one by one into a chain of identical neural networks. A word embedding is a numerical representation of a word in a high-dimensional space, where words with similar meanings are located close to each other. It allows machines to understand and process text by representing words as vectors of numbers, enabling them to capture semantic and syntactic relationships between words (Turing, 2022). Now, let's get back to the network in Figure 1. Each network produces two outputs: the output embedding, and another output (indicated by orange arrows) that is fed back into the subsequent network along with the next word embedding. Each output embedding contains information about the word and its context within the preceding part of the sentence. Ideally, the final output from the last network in the series contains information about the whole sentence. However, one limitation of RNNs is their tendency to gradually "forget" information about earlier tokens as more inputs are processed (Prince, 2023).

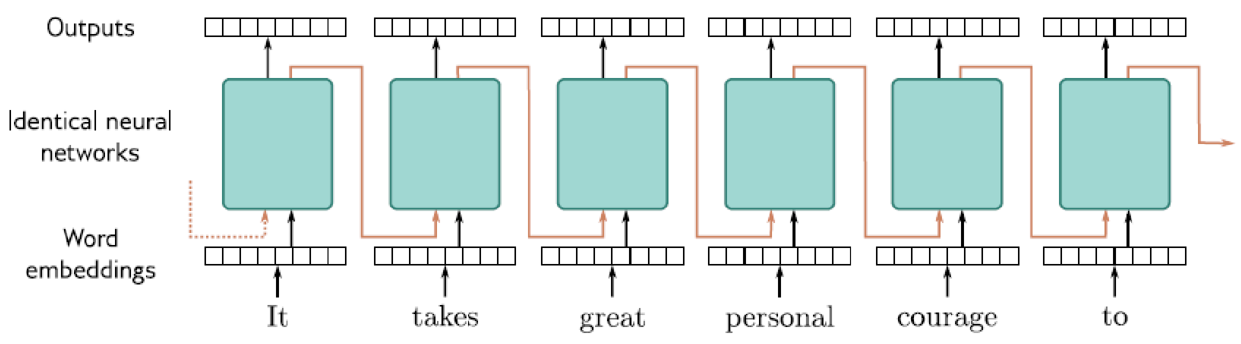

Figure 1: RNN example (Prince, 2023, p. 223)

Architecture and recurrent connections

Structures of RNNs include an input layer, represented by a vector $w(t)$, which encodes the current word ω_t as a **one-hot vector** of the vocabulary size V, meaning the size $w(t)$ equals the vocabulary size. Additionally, there's a vector $s(t-1)$ that carries the output values from the hidden layer at the previous time step. Once the network is trained, the output layer $y(t)$ provides the probability of the next word $P\left(\omega_t, s(t-1)\right)$. The architecture of RNNs is shown in Figure 2.

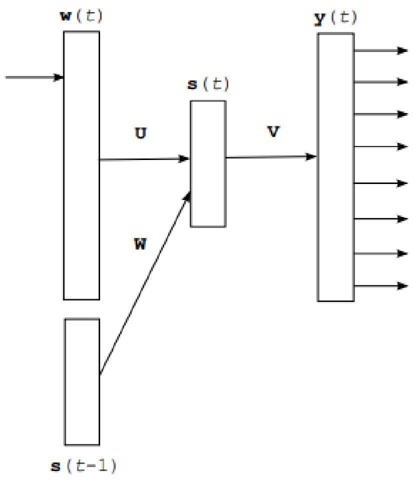

Figure 2: Simple RNN (Mikolov, 2012).

The network is trained using stochastic gradient descent, employing either the standard backpropagation algorithm or backpropagation through time (this concept will be discussed in a later section). The structure of the network includes input, hidden, and output layers along with corresponding weight matrices: U and W that connect the input to the hidden layer, and V connects the hidden layer to the output layer. The outputs for each layer are calculated as follows.

The hidden layer outputs, $s_j(t)$, are computed by applying a sigmoid function f to the weighted sum of the current input and the previous outputs from the hidden layer:

$$s_j(t) = f\left(\sum_i \omega_i(t) u_{ji} + \sum_l s_l(t-1)\omega_{jl}\right).$$

The output layer $y_k(t)$ uses a softmax function g, which helps ensure that the outputs form a valid probability distribution:

$$y_k(t) = g\left(\sum_j s_j(t) \upsilon_{kj}\right)$$

The sigmoid function $f(z)$ and the softmax function $g(z_m)$ are defined as:

$$f(z) = \frac{1}{1+e^{-z}}$$

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

In this neural network model, biases are omitted as their inclusion did not yield significant performance improvements, adhering to the principle of **Occam's razor** to keep the solution as simple as necessary. Additionally, equations for $s(t)$ and $y(t)$ can be represented in a more compact form using matrix-vector multiplication:

$$s(t) = f\left(U_W(t) + W_S(t-1)\right)$$

$$y(t) = g\left(V_S(t)\right).$$

Here, $y(t)$ represents the probability distribution of the next word ω_{t+1} given the historical context. The time complexity for one training or testing step is proportional to:

$$O = H \times H + H \times V = H \times (H + V),$$

where H is the size of the hidden layer and V is the size of the vocabulary.

To be more precise with the latest equation, the term $H \times H$ corresponds to the complex process of computing the hidden state from the previous hidden state, while the term $H + V$ represents the complex process of computing the output from the hidden state. Thus, the total time complexity is the sum of these operations.

Training process

Both feedforward and recurrent architectures in neural network models can be trained using stochastic gradient descent with the well-known backpropagation algorithm. However, to improve performance, especially for recurrent networks, the backpropagation through time algorithm is recommended. This technique allows the propagation of error gradients back through time across the recurrent connections, helping the model to retain valuable information in the hidden layer's state.

Without backpropagation through time, recurrent networks sometimes perform inadequately (Mikolov, 2012). During training with stochastic gradient descent, the network's weight matrices are updated sequentially with each example. The error gradient in the output layer is calculated using a cross-entropy criterion and then backpropagated to the hidden layer, and, in the case of backpropagation through time, through the recurrent connections as well. Validation data are utilised for **early stopping** and to adjust the learning rate, ensuring efficient learning. Typically, training involves multiple iterations over the dataset, requiring about 8-20 epochs to converge. Research shows that randomising the order of sentences in the training data can expedite convergence, thus reducing the number of epochs needed.

In terms of the learning rate management, the coefficient starts at $\alpha = 0.1$. This rate is maintained as long as there is a significant improvement in the validation data, defined as a reduction in entropy of more than 0.3%. Here, "entropy" is cross-entropy loss, that represents the variation between the actual data distribution and the expected probability distribution. In classification problems, cross-entropy loss is frequently used to measure the degree of match between the expected and actual labels. If no further significant improvement is observed, the learning rate is halved at the start of each new epoch until no additional improvements are seen, at which point the training concludes. While the validation dataset is primarily for controlling the learning rate, it is possible to train models without one by manually determining the number of epochs for the full and reduced learning rates, based on prior experiments with training data subsets. Normally, a validation dataset is used to report perplexity results. In addition, since the model does not learn any parameters from the validation data directly, there is no risk of overfitting to this data.

The weight matrices $U, V,$ and W are initially filled with small random numbers. In further experiments, these are often initialised using a normal distribution with a mean of 0 and a variance of 0.1.

The training process for one epoch of an RNN is structured as follows:

1. Set the time counter $t = 0$ and initialise the state of the neurons in the hidden layer $s(t)$ to 1.
2. Increment the time counter t by 1.
3. Present the current word w_t at the input layer $w(t)$.
4. Copy the state of the hidden layer from the previous time step $s(t - 1)$ to the input layer.
5. Perform a forward pass (as previously described) to compute the new state $s(t)$ and the output $y(t)$.
6. Calculate the gradient of the error $e(t)$ at the output layer.
7. Propagate the error backwards through the network and adjust the weights accordingly.
8. Repeat steps 2 to 7 until all training examples have been processed.

The training aims to maximise the likelihood of the training data with the objective function:

$$f(\lambda) = \sum_{t=1}^{T} \log log\, y_{l_t}(t).$$

Here, t ranges from 1 to T, and l_t is the index of the correct predicted word for the t-th sample. The gradient of the error vector in the output layer, $e_0(t)$, is computed using a cross-entropy criterion:

$$e_0(t) = d(t) - y(t),$$

where $d(t)$ is the target vector representing the word $w(t + 1)$, encoded as a 1-of-V vector. This approach is chosen over mean square error to optimise the entropy, perplexity, word error rate, or to maximise the compression ratio.

Weights V between the hidden layer $s(t)$ and the output layer $y(t)$ are updated as follows:

$$v_{jk}(t + 1) = v_{jk}(t) + s_j(t)e_{ok}(t)\alpha,$$

where α is the learning rate, j iterates over the size of the hidden layer and k over the size of the output layer, $s_j(t)$ is output of j-th neuron in the hidden layer and $e_{ok}(t)$ is error gradient of k-th neuron in the output layer. If L2 regularisation is applied, the update formula modifies to:

$$v_{jk}(t + 1) = v_{jk}(t) + s_j(t)e_{ok}(t)\alpha - v_{jk}(t)\beta.$$

Here, β, the regularisation parameter, is typically set to 10^{-6} and the matrix-vector notation is given as:

$$V(t + 1) = V(t) + s(t)e_0(t)^T\alpha - V(t)\beta.$$

Further, errors are propagated back from the output to the hidden layer:

$$e_h(t) = d_h\left(e_0(t)^T V, t\right),$$

where the error vector is obtained using the function d_h that is applied elementwise:

$$d_{hj}(x, t) = x s_j(t)\left(1 - s_j(t)\right).$$

Weights U between the input layer $w(t)$ and the hidden layer $s(t)$ are updated using:

$$u_{ij}(t + 1) = u_{ij}(t) + w_i(t)e_{hj}(t)\alpha - u_{ij}(t)\beta,$$

where matrix-vector notation can be formulated as:

$$U(t + 1) = U(t) + w(t)e_h(t)^T\alpha - U(t)\beta.$$

Finally, the recurrent weights W are updated with:

$$w_{lj}(t + 1) = w_{lj}(t) + s_l(t - 1)e_{hj}(t)\alpha - w_{lj}(t)\beta,$$

and could be presented with matrix-vector notations as:

$$W(t + 1) = W(t) + s(t - 1)e_h(t)^T\alpha - W(t)\beta.$$

These updates strengthen matrix-vector notation to efficiently manage the computation, optimising updates for the active inputs, as inactive neuron weights remain unchanged.

Unfolding through time

The training approach initially described, known as normal backpropagation, trains an RNN similarly to a regular feedforward network with one hidden layer. The primary difference is that the input layer's state is influenced by the state of the hidden layer from the previous time step. However, this method is not optimal: it focuses on predicting the next word based on the previous word and the hidden layer's state but does not actively encourage the hidden layer to store information useful for future predictions. If the network retains long-term context, it is more accidental than intentional.

An improvement to this method is the backpropagation through time algorithm, which better equips the network to learn what information to store in the hidden layer. Conceptually, an RNN operated for N time steps can be visualised as a deep feedforward network with N identical hidden layers (the hidden layers have the same dimensionality and use the same recurrent weight matrices). This is illustrated in Figure 3.

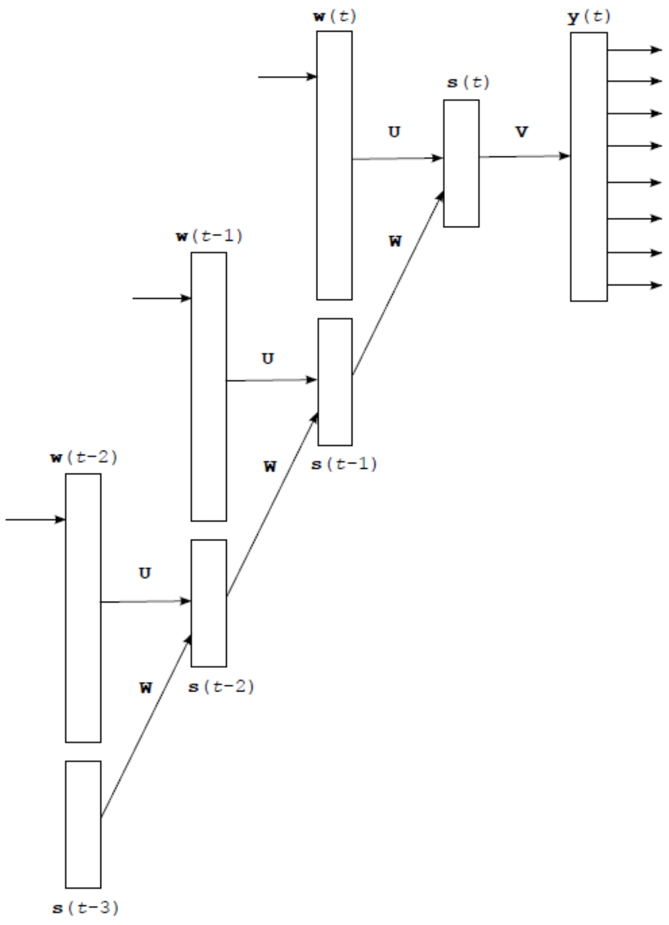

Figure 3: RNN unfolded to illustrate its equivalence to a deep feedforward network, shown here over 3-time steps (Mikolov, 2012).

In this extended model, errors are propagated from the state $s(t)$ back to $s(t-1)$, and the recurrent weight matrix W is updated accordingly. The error propagation is done recursively and requires storing the states of the hidden layer from previous time steps:

$$e_h(t-\tau-1) = d_h\left(e_h(t-\tau)^T W,\ t-\tau-1\right).$$

Although theoretically possible to unfold for as many time steps as there are training examples, in practice, **error gradients tend to vanish** (or rarely, explode) as they are backpropagated further back in time, making it practical to limit the depth of unfolding—often referred to as truncated backpropagation through time. For word-based language models, unfolding for about five time steps is typically sufficient, yet this setup still allows the network to learn to retain information beyond those five steps.

Here, weights U are updated for backpropagation through time algorithm as follows:

$$u_{ij}(t + 1) = u_{ij}(t) + \sum_{z=0}^{T} \omega_{i}(t - z)e_{hj}(t - z)\alpha - u_{ij}(t)\beta$$

where T is the number of time steps for which the network is unfolded. This can also be expressed in matrix-vector notation:

$$U(t + 1) = U(t) + \sum_{z=0}^{T} W(t - z)e_{h}(t - z)^{T}\alpha - U(t)\beta.$$

It's important to apply the weight updates for U in one comprehensive step rather than incrementally during error backpropagation to avoid training instability. Similarly, the recurrent weights W are updated with:

$$\omega_{lj}(t + 1) = \omega_{lj}(t) + \sum_{z=0}^{T} s_{l}(t - z - 1)e_{hj}(t - z)\alpha - \omega_{lj}(t)\beta,$$

which is simplified to:

$$W(t + 1) = W(t) + \sum_{z=0}^{T} S(t - z - 1)e_{h}(t - z)^{T}\alpha - W(t)\beta.$$

The described method ensures that updates are systematically applied, improving the network's ability to learn and retain relevant patterns across multiple time steps.

Applications

RNNs have found applications in various fields where the processing of sequential data is dominant. Two popular examples of such applications are language modelling and machine translation.

Language modelling

In language modelling, the aim is to predict the probability of a sequence of words occurring in a language. This is important for tasks like text prediction, speech recognition, and even writing assistance. The mathematical representation of a language model attempts to estimate the likelihood of a word sequence: $\omega_1, \omega_2, \ldots, \omega_n$ by modelling the probability of each word given the previous words:

$$P\left(\omega_1, \omega_2, \ldots, \omega_n\right) = \prod_{i=1}^{n} P\left(\omega_1, \omega_2, \ldots, \omega_{i-1}\right).$$

RNNs are suited for this task because they can theoretically maintain information about all previous words in the hidden layer's state, allowing for the context-dependent generation of word probabilities. At each time step t, the RNN updates its hidden state $s(t)$:

$$s(t) = f(U\omega(t) + Ws(t - 1)),$$

where $\omega(t)$ is the current word input, U and W are the weight matrices for the input and recurrent connections, respectively, and f is a nonlinear activation function, often a sigmoid or tanh. The output layer then computes the probability distribution over the vocabulary for the next word ω_{t+1}, using a softmax function:

$$y(t) = softmax\,(Vs(t)),$$

where V is the weight matrix from the hidden state to the output layer.

Extra resource

An experiment where the complete works of Shakespeare were used to train a more complex three-layer RNN with 512 hidden nodes in each layer. After several hours of training, the network generated text that mimics Shakespeare's style, including character dialogues and extended monologues. Despite the network only recognizing character patterns without understanding content, it remarkably produces coherent passages that resemble Shakespeare's dramatic and poetic style, demonstrating the potential of deep learning models to capture and recreate complex literary styles. To learn more about the experiment and others explore **the power of RNNs**.

Machine translation

Another example, machine translation, involves converting a sequence of text from one language to another. RNNs handle this through a process often structured as an encoder-decoder framework. The encoder RNN reads the input sentence, one word at a time, and transforms it into a context vector, typically the final hidden state of the RNN after processing the last word of the input. This context vector aims to retain the content of the entire input sentence. The decoder, another RNN, then takes this context vector and generates the translated sentence one word at a time. The decoder's initial state is set to the encoder's final state, and it begins generating words of the target language from the context:

$$s'(t) = f'\left(Vs^{'}(t - 1) + U'\omega'(t)\right)$$

$$y'(t) = softmax\,(Ws'(t)).$$

Here, $f^{'}, V, U^{'}$, and W are parameter matrices and functions for the decoder, $s'(t)$ represents the decoder's hidden state at time t, and $\omega'(t)$ is the word generated at time t.

Both of these applications illustrate the strength of RNNs in modelling sequences and their dependencies, which is challenging to achieve with other types of neural networks that lack the temporal dynamic behaviour inherent to RNNs. Despite challenges such as vanishing gradients, variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models have helped mitigate these issues, further enhancing the applicability of RNNs in complex tasks like language modelling and machine translation.

RNN example

Now that we have covered the theoretical foundation of RNNs, let’s build one. In this exercise, we will train a simple RNN to predict the next character in sequences from the **Penn Treebank dataset**. This dataset is a collection of annotated text, primarily from the Wall Street Journal. It includes both syntactic annotations, like part-of-speech tags and parse trees, and lexical annotations, such as word segmentation. Here are a few example rows from the Penn Treebank dataset:

```
In a stunning reversal of fortune, the company reported a net loss for the quarter.
Despite the challenges, the project was completed on time and under budget.
Economic growth is expected to slow down in the coming months, analysts say.
```

The model built on the described data set will be evaluated on a test set (a pre-prepared subset of the main dataset) to assess its predictive performance using the loss function and perplexity metric.

1. First, create a virtual environment for all your tasks that require Pytorch. For guidance, refer to the additional reading, **Setting up a Virtual Environment**.

2. Now, install Pytorch in your virtual environment using the **getting started instructions** on the Pytorch website. Take note of dependencies such as the latest compatible version of Python as you may need to use an older version of Python in your virtual environment.

3. Next, let’s import the required libraries: `torch` is essential for using PyTorch's core functionalities; `torch.nn` provides access to RNN layers and loss functions; `torch.optim` is used for optimisation algorithms which adjust model weights during training.

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

4. Ensure that the training dataset is in the specified ptbdataset directory. Then, we can to read the contents of a text file, and store the text in a variable. In addition, we can print the length of the text to the console to see the size of our data.

```
# Read the text from the manually imported file
with open('/ptbdataset/ptb_train.txt', 'r') as file:
    text = file.read()

# Check the length of the text
print("Length of text: ", len(text))
```

Output:

```
Length of text:  5101618
```

5. Now, the task is to prepare the data for training by converting it into a format that a neural network can process and learn from. For that purpose we need to create **character mappings** and appropriate **sequences**. More precisely, character mapping is here to convert characters to numerical form, making them suitable for model input. It also enables decoding of model outputs back to readable text. On the other side, sequence creation transforms the text into manageable chunks of data for the model. These sequences help the model learn patterns in sequences of characters, which is crucial for generating coherent text.

   In terms of character mappings realisation, we start by extracting all unique characters from the text and sorting them. We then create two dictionaries: one that maps each character to a unique integer index, facilitating the conversion of characters into numerical data suitable for the model, and another that performs the reverse mapping to convert numerical indices back to characters, which is crucial for interpreting the model's predictions.

```
# Creating character mappings
chars = sorted(list(set(text)))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

Next, we generate sequences of characters. We define the length of each sequence with ‘`maxlen`’ and determine how much the sequence window moves each time with each step. Using these parameters, we extract overlapping sequences of characters from the text. Each sequence, of length ‘`maxlen`’, is stored along with the character that immediately follows it. This results in two lists: one containing the sequences and the other containing the subsequent characters. Finally, we will print the number of sequences to see the result of our efforts.

```
# Creating sequences
maxlen = 40
step = 3
sentences = []
next_chars = []

for i in range(0, len(text) - maxlen, step):
   sentences.append(text[i: i + maxlen])
   next_chars.append(text[i + maxlen])

print("Number of sequences:", len(sentences))
```

Output:

```
Number of sequences: 1700526
```

6. Next, we vectorize the data for model training in order to prepare the data for the model, allowing it to process and learn from the sequences. **Tensors** are similar to NumPy `ndarrays` but are more computationally efficient because they can run on GPUs. First, we initialize tensors `X` and `y` to store the input and target data, respectively. `X` has dimensions corresponding to the number of sequences, sequence length, and vocabulary size, while `y` corresponds to the number of sequences and vocabulary size. We then iterate over each sequence, converting characters into one-hot encoded vectors. For each sequence in sentences, the characters are mapped to their corresponding indices, and the appropriate positions in `X` are set to 1. The target characters in `y` are also one-hot encoded, representing the character that follows each sequence.

```python
# Vectorizing the data
X = torch.zeros((len(sentences), maxlen, len(chars)),
dtype=torch.float32)
y = torch.zeros((len(sentences), len(chars)), dtype=torch.float32)

for i, sentence in enumerate(sentences):
   for t, char in enumerate(sentence):
       X[i, t, char_indices[char]] = 1
   y[i, char_indices[next_chars[i]]] = 1
```

7. It is time to define the `RNNModel` class which specifies a simple network with an input layer, a hidden layer, and a fully connected output layer. The forward method processes input sequences through the RNN, then applies a fully connected layer to the RNN's output and a softmax activation to produce probability distributions over the possible next characters.

```python
# Building the model
class RNNModel(nn.Module):
   def __init__(self, input_size, hidden_size, output_size):
       super(RNNModel, self).__init__()
       self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
       self.fc = nn.Linear(hidden_size, output_size)
       self.softmax = nn.Softmax(dim=1)

   def forward(self, x):
       out, _ = self.rnn(x)
       out = self.fc(out[:, -1, :])
       out = self.softmax(out)
       return out
```

8. Finally, it is time to initiate our RNN model by defining the size of the input feature space, the number of units in the hidden layer and the number of possible output classes.

```python
input_size = len(chars)
hidden_size = 128
output_size = len(chars)

model = RNNModel(input_size, hidden_size, output_size)
```

9. Before the learning phase, we should choose the loss function to evaluate how well the model's predictions match the actual data, providing a measure of performance. Also, the Adam optimizer is selected to adjust the model's parameters during training, aiming to minimise the loss and improve the model's accuracy. This optimizer is an advanced optimization algorithm that combines the benefits of the Adaptive Gradient Algorithm and Root Mean Square Propagation. It adapts the learning rate for each parameter by maintaining and updating running averages of both the gradients and their squares.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

10. The learning phase starts now by training the model over two epochs using mini-batches of 128 samples each. In each batch, the model performs a forward pass to predict the outputs and compute the loss. Then, it performs a backward pass to calculate gradients, and finally updates the model parameters using the optimizer.

```
# Training the model
num_epochs = 2
batch_size = 128
for epoch in range(num_epochs):
   for i in range(0, len(X), batch_size):
       X_batch = X[i:i+batch_size]
       y_batch = y[i:i+batch_size]

       # Forward pass
       outputs = model(X_batch)
       loss = criterion(outputs, y_batch)

       # Backward pass and optimisation
       optimizer.zero_grad()
       loss.backward()
       optimizer.step()

   print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')
```

In the last two rows of the previous code snippet, we wanted to print the loss values. The printed results shown below indicate that the loss value remained constant across the two epochs, suggesting that the model did not improve its performance during training. This could imply issues such as insufficient learning or a need for improved hyperparameters. In a previous lesson covering Neural Networks, we discussed hyperparameters which can be selected or tuned to achieve better performances. We encourage you now to use this knowledge to update the model to show better loss results.

```
Epoch [1/2], Loss: 3.7936
Epoch [2/2], Loss: 3.7936
```

11. The training phase is finished. Now let's evaluate our model on the test data which is, in a similar manner as training data, imported to the specified directory and stored in an appropriate test variable.

```
with open('/ptbdataset/ptb_test.txt', 'r') as file:
   test_text = file.read()
```

12. The task is now identical to the training data, let's create appropriate sequences for the test set and vectorize them.

```
# Creating sequences
test_sentences = []
test_next_chars = []
for i in range(0, len(test_text) - maxlen, step):
   test_sentences.append(test_text[i: i + maxlen])
   test_next_chars.append(test_text[i + maxlen])

# Vectorizing process
X_test = torch.zeros((len(test_sentences), maxlen, len(chars)),
dtype=torch.float32)
y_test = torch.zeros((len(test_sentences), len(chars)),
dtype=torch.float32)
for i, sentence in enumerate(test_sentences):
   for t, char in enumerate(sentence):
       X_test[i, t, char_indices[char]] = 1
   y_test[i, char_indices[test_next_chars[i]]] = 1
```

13. Now we build an evaluation function that performs two tasks. The first tas is to calculate the model's loss by comparing the predicted values to the true labels using a loss function. And the second task is to compute the perplexity, which is a measure of how well the model predicts the test data. Perplexity is derived from the loss and provides an indication of the model’s performance in terms of prediction accuracy.

```
def evaluate_model(model, X_test, y_test):
   with torch.no_grad():

       # Forward pass
       y_pred = model(X_test)

       # Compute loss
       criterion = nn.CrossEntropyLoss()
       loss = criterion(y_pred, y_test)

       # Calculate perplexity
       perplexity = torch.exp(loss)

   return loss.item(), perplexity.item()
```

14. This final part of the practical exercise evaluates the model's performance. The ‘`evaluate_model`’ function is called to obtain these metrics.

```
# Evaluate the model on the test set
test_loss, test_perplexity = evaluate_model(model, X_test, y_test)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Perplexity: {test_perplexity:.4f}')
```

Output:

```
Test Loss: 3.7626
Test Perplexity: 43.0603
```

The printed results indicate that the model's performance on the test set has a loss of 3.7626, reflecting the average error between the predicted and actual values. The perplexity of 43.0603 suggests how well the model predicts the next character in a sequence, with lower perplexity values generally indicating better predictive accuracy.

Instructions

Deep learning models are computationally expensive. If you run into problems on your local machine you can try building your model in **Google Colab**. You can refer to the section on Google Colab in the **Installation, Sharing, and Collaboration** additional reading for more information. You may need to modify the paths for importing the training and testing datasets depending on where you access these files from in Google Colab.

Practical task

Your task is to implement a simple RNN using Python and Pytorch, a deep-learning framework, to generate text based on the Shakespeare dataset.

The dataset represents a collection of text extracted from the works of William Shakespeare. It contains a variety of plays, sonnets, and his other literary works. The text is in English and includes a wide range of vocabulary and linguistic styles typical of Shakespearean literature.

When you implement your model, train it for 10 epochs and generate a sample text of 100 words.

1. To start, make a copy of the **shakespeare_template.ipynb** and name it **shakespeare_task.ipynb**
2. Then, import the tinyshakespeare dataset, as well as all required libraries.
3. Next, prepare text data for training a character-based language model capable of predicting the next character in a sequence based on the preceding characters. Unlike word-based models, which predict the next word, character-based models work at a finer granularity, handling individual characters. To make proper preparations, you should start by creating character mappings and sequences. To complete this task, please refer to the Penn tree example.
4. Now, vectorize input sequences into binary matrices representing the presence of characters in the vocabulary. In addition, encode target outputs as one-hot vectors indicating the next character in the sequence.
5. Next, import the required libraries and build your simple RNN model.
6. Train the model for 10 epochs.

7. Create a function which should generate text with the specified word count (100 words) using a trained language model.
    a. One approach to do that is to start from a given seed text, iterate to generate additional words until the desired word count is reached. Also, you can utilise the model's predictions to select the next character probabilistically, ensuring coherent text generation. Finally, at the end of the created function, you should return the generated text with the specified word count.
8. Use your function to generate sample text with 100 words and print it.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Marcus, M. P., Marcinkiewicz, M. A., & Santorini, S. (1993). Building a large annotated corpus of English: the. Computational Linguistics. **https://doi.org/10.5555/972470.972475**

Mikolov, T. (2012). Statistical language models based on neural networks. Brno University of Technology, Faculty of information technology, Department of computer graphics and multimedia. **https://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf**

Minsky, M., & Papert, S. A. (1969). Perceptron's: An introduction to computational geometry. MIT Press.

Prince, S. J. (2024). Understanding Deep Learning. MIT press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). Learning internal representations by error propagation. Technical Report, La Jolla Institute for Cognitive Science, UCSD.

Turing. (2022, February 10). A Guide on Word embeddings in NLP. **https://www.turing.com/kb/guide-on-word-embeddings-in-nlp**

Van den Oord, A., Kalchbrenner, N., & Kavukcuoglu, K. (2016). Pixel recurrent neural networks. International Conference on Machine Learning, 1747–1756.