Language Modeling with Recurrent Neural Networks and Backpropagation Through Time

Kai Ma

***Abstract*—In the past decade, neural network-based approaches have led to explosive growth in the field of machine learning, Language models are notable and popular application of this with the recent success of large language models. In this report, the mathematical foundations of neural network learning algorithms are examined, with a focus on the backpropagation mechanism. This is slightly extended to introduce the concept of recurrent neural networks and "backpropagation through time", allowing us to build a basic (but functional) language model.**

I. Introduction

MACHINE learning is often considered an intimidating subject due to its perceived complexity and lack of explanation at high levels, as many complex models appear to be black boxes. However, the foundations of machine learning are relatively simple with a basic understanding of mathematical concepts such as linear algebra and multivariable calculus. Backpropagation [1] is the standard technique for model training as part of the gradient descent process. The objective of this paper is to provide an overview of neural networks and backpropagation, culminating in the introduction of "backpropagation through time" [2] for recurrent neural networks (RNNs) [3]. The specific application that is implemented is the generation of text in the style of Shakespeare, which demonstrates the function and capability of RNNs.

In this paper, the following is achieved:

- Using multivariable calculus, we establish the mathematical foundations of feedforward neural networks, which is extended to understand multilayer networks and backpropagation.
- We extend the knowledge of multilayer networks and backpropagation to establish the mathematical foundations of recurrent neural networks and backpropagation through time.
- We implement a recurrent neural network from scratch and train it as a language model on the writing of William Shakespeare. Model outputs are observed and evaluated, and compared to a more complex 3-layer RNN.

II. Background

We first introduce the foundational concepts of machine learning by discussing basic feedforward neural networks. This is then extended to recurrent neural networks.

K. Ma is with the Department of Mechanical and Mechatronics Engineering, Faculty of Engineering, University of Waterloo, Waterloo, ON, N2L 3G1, Canada. E-mail: k78ma@uwaterloo.ca

A. Feedforward Neural Networks

1) Problem and Data: The core problem in most machine learning tasks is to make accurate predictions or decisions based on input data. In the context of a neural network, this typically involves learning a function that maps input features to desired output labels or values. To achieve this, a model is trained on a dataset comprising numerous instances, each containing a set of features and usually a label or value.

For all of the work discussed in this paper, we are dealing with a supervised learning scenario. This means that the dataset used for training includes not only the input features but also the corresponding output labels, which are essential for the network to learn the correct mappings. This dataset is defined as:

$$D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$$

where x values indicate inputs features and y values indicate a label of some kind. The specific values and types of x and y dictate the type of problem. For example, sigmoid functions squashes inputs to be between 0 and 1, which can be interpreted as probabilities in binary classification tasks. In a multi-layer networks (covered in Section II-B, this activation and output function may be separated.

2) Model Definition: At a high-level, neural networks ingest a vector of input features $x = \begin{bmatrix} x_1, x_2, \ldots, x_n \end{bmatrix}$, where n is the number of features. The "features" are individual measurable properties or characteristics of a phenomenon being observed; they inputs that we provide to a model, upon which the model makes a prediction or decision.

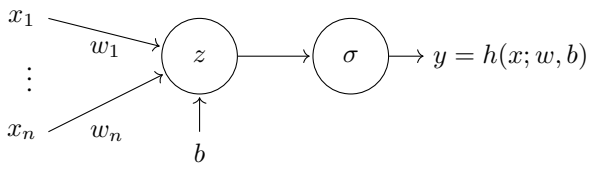

Fig. 1: Single-layer feedforward neural network.

The prediction made by the model is fundamentally defined by a set of **weights** $w = \begin{bmatrix} w_1, w_2, \ldots, w_n \end{bmatrix}$ and a **bias** term b. A weighted sum is produced by taking:

$$z(x; w, b) = w^T x + b = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

Here, w^T is the transpose of w, such that $w^T x$ is essentially a dot product. This weighted sum z is called a **logit**.

At this step, **activation functions** are usually applied to the weighted logit. Activation functions introduce non-linearity

into the network, which allows the network to model more complex relationships between the inputs and outputs. Some examples of activation functions are sigmoid, hyperbolic tangent, and ReLU functions. In a single-layer network, the activation function also serves to interpret the output of the network

We can formalize this entire process as:

$$y = o(x; w, b) = \sigma(z(x; w, b)) \tag{2}$$

where $o(x; w, b)$ is the output of function modeled by the neural network, z is the logit given by the weighted sum operation using w (weights) and b (bias), and $\sigma(\cdot)$ represents the activation function applied to the weighted sum of inputs. This is called the **forward pass**.

3) Loss Functions: To measure how well the model is performing, a loss function is used. Specifically, loss functions quantify the quality of a prediction $\hat{y} = h(x^{(i)}; \Theta)$ makes for a specific data sample $(x^{(i)}, y^{(i)})$, where Θ represents the parameters of the model (such as w and b). This is usually done by comparing the value predicted by the model, $\hat{y}$, to the actual value y. A low loss value indicates a good prediction, and a high loss indicates a poor prediction.

The exact definition of a loss function depends on the problem at hand. As an example, a very simple loss function is binary $0-1$ loss:

$$L_{01}(h(x; \Theta), y) = \begin{cases} 0 & \text{if } y = h(x; \Theta) \\ 1 & \text{otherwise} \end{cases}$$

This simply outputs a 0 if our model's prediction matches the ground truth and a 1 otherwise.

4) Learning as an Optimization Problem: model and loss function is defined, but how do we find a model that actually performs well? We can introduce a general framework for training models for arbitrarily complicated problems by framing machine learning as an computational minimization problem.

Fundamentally, we define an **objective function** $J(\Theta)$, where Θ are the parameters (w and b) of our model. Since the loss function quantifies the quality of our model, we define the objective function to be:

$$J(\Theta) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{L(h(x^{(i)}; \Theta), y^{i})}_{\text{loss function}} \tag{3}$$

such that we are measuring the loss of the model across the entire dataset.

Generally, the optimization is that we want to find $\Theta^{\star}$ such that:

$$\Theta^{\star} = \text{argmin}_{\Theta}\ J(\Theta) \tag{4}$$

which is to say that we want to find the Θ that minimizes $J(\Theta)$, making this a minimization problem.

5) Training with Gradient Descent: How do we find the minimum of our objective function?

In general, if we're trying to optimize a function *f*, we compute the gradient at our current point x. Since the gradient gives the direction of the maximum rate of increase, we move in the opposite direction, making the function smaller. In terms of a machine learning model, we would have some objective function $J(\Theta)$ defining some surface over Θ, and we want to find the Θ value at the lowest point on the surface. This is called gradient descent [4]

Specifically, we choose an initial value for model parameters Θ, a step-size parameter η (also called the **learning rate**), and a convergence threshold ϵ. Then, as explained above, we find the gradient of the objective function $\nabla_{\Theta} J(\Theta)$ and move along the function in the opposite direction of the gradient to minimize. This is shown in algorithm form in Algorithm 1.

Algorithm 1 Gradient Descent

Input: learning rate η, convergence threshold ϵ
Output: optimized parameters Θ
procedure GRADIENTDESCENT
  Initialize parameters Θ
  repeat
    Compute the gradient $\nabla_{\Theta} J(\Theta)$
    Update the parameters: $\Theta \leftarrow \Theta - \eta \cdot \nabla_{\Theta} J(\Theta)$
    if magnitude of $\nabla_{\Theta} J(\Theta) < \epsilon$ **then**
      break
    end if
  until convergence
  return Θ
end procedure

B. Multi-layer Neural Networks

Multi-layer networks, often referred to as **multi-layer perceptrons** (MLPs), extend the concept of the simple neural network model above into a more complex architecture composed of multiple layers of neurons. These layers include:

1) **Input layer**: This is the layer that receives the input features. It does not apply any computation and simply passes the features to the next layer.
2) **Hidden layers**: One or more layers that compute the intermediate features. These layers are called "hidden" because their outputs are not observed in the training data. The computation in these layers introduces nonlinearity to the system, allowing the network to capture complex patterns.
3) **Output layer**: The final layer that provides the prediction or classification based on the input features and the computations done by the hidden layers.

The forward pass in multi-layer networks involves sequentially passing the input data passes through each layer to produce an output, such that the output of each layer becomes the input to the next layer. Mathematically, for each layer l, this process can be described as:

$$o^{[l]}(x) = \sigma^{[l]}(w^{[l]T} o^{[l-1]}(x) + b^{[l]}) \tag{5}$$

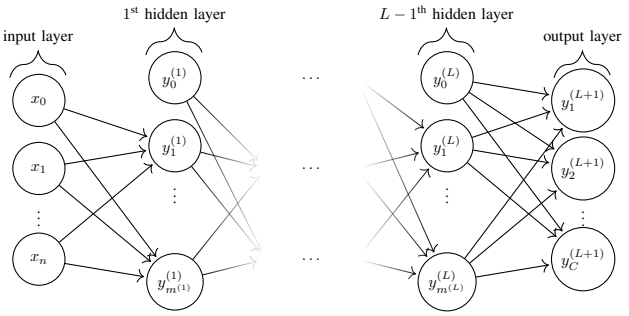

Fig. 2: Multi-layer feedforward network.

where $o^{[l]}(x)$ is the output logit of layer l, $w^{[l]}$ and $b^{[l]}$ are the weights and biases of layer l, $\sigma^{[l]}$ is the activation function for layer l, and $o^{[l-1]}(x)$ is the output of the previous layer. We say that $o^{[0]} = x$ is the input to the network and $o^{[L]}$ is the final output (activation of last layer).

The final output transforms the outputs of the last hidden layer into a form y that is suitable for the specific type of problem the network is intended to solve. Output layers often use similar functions as activation functions. For example, sigmoid functions squashes inputs to be between 0 and 1, which can be interpreted as probabilities in binary classification tasks.

1) Gradient Descent with Backpropagation:: The training process for multi-layer networks follows a similar principle to that of single-layer networks, as discussed in the Training section; we define an objective function and use gradient descent to find model parameters that minimize this objective. The objective function typically includes the loss function L as well as possibly other terms (such as regularization terms). However, the gradient descent process relies on the computation of the gradient of the objective function, $J(\Theta)$. Previously, our Θ consisted of a single weight and bias, but in a multi-layer network, there are multiple layers of weights and biases; we must find the gradient of the objective function with respect to the weights in the neural network. Backpropagation is a method that uses the chain rule to compute these gradients systematically.

In the introduction of gradient descent in Section II-A5, we combined the weights and biases into a single parameter $\Theta = \{w, b\}$. Here, we will consider w and b individually for mathematical clarity.

Let us say we have a multi-layer network with layers $0, \ldots, L$. First, a forward pass is performed. For each layer, we calculate:

$$z^{[l]} = w^{[l]T} h^{[l-1]} + b^{[l]} \tag{6}$$

$$o^{[l]} = \sigma^{[l]}(z^{[l]}) \tag{7}$$

where $o^{[0]} = x$ is the input to the network and $o^{[L]}$ is the final output (activation of last layer).

We then compute the loss using the predicted output $o^{[L]}$ and the true value y. Then, for the chosen objective function $J(o^{[L]}, y)$, we calculate the initial gradient of the loss with respect to the output of the last layer's activations $\frac{\partial J}{\partial o^{[l]}}$.

Then, we arrive at the central mechanism of backpropagation: the **backward pass**. This is performed through the network to compute the gradient of the loss with respect to the activations, weights, and biases of each layer, starting from the output layer L back to the first hidden layer.

For the last layer L, compute the gradients of the loss with respect to the logits:

$$\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} \tag{8}$$

Since h is a function of z, we can use the partial derivative chain rule:

$$\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = \frac{\partial J}{\partial o^{[L]}} \cdot \frac{\partial o^{[L]}}{\partial z^{[L]}} \tag{9}$$

where the last term $\frac{\partial o^L}{\partial z^L}$ is the partial derivative of the activation function at the last layer with respect to its input $z^{[L]}$. We use $\delta^{[l]}$ to represent the "error" for each layer l, quantifying how much a given layer's output should change to minimize the loss for the entire network.

Then, we compute the gradients for the parameters of each layer. For the output layer L, we would have:

$$\nabla_{w^{[L]}} J = \delta^{[L]} \cdot (o^{[L-1]})^T \tag{10}$$

$$\nabla_{b^{[L]}} J = \delta^{[L]} \tag{11}$$

Here, the gradient with respect to the bias b is the delta itself because the derivative of $z^{[L]}$ with respect to $b^{[L]}$ is 1.

Following this method, we can recursive calculate gradients for each preceding layer $l = L-1, L-2, \ldots, 1$:

$$\delta^{[l]} = (w^{[l+1]T} \delta^{[l+1]}) \cdot \frac{\partial \sigma^{[l]}(z^{[l]})}{\partial z^{[l]}} \tag{12}$$

This step propagates the error backward by taking into account the error from the layer above ($\delta^{[l+1]}$) and the gradient of the activation function for the current layer's output.

Then, the gradients for the weights and biases for layer l are computed as follows:

$$\nabla_{w^{[l]}} L = \delta^{[l]} \cdot (o^{[l-1]})^T \tag{13}$$

$$\nabla_{b^{[l]}} L = \delta^{[l]} \tag{14}$$

Finally, we can update the weights and biases using the previous gradient descent method:

$$w^{[l]} = w^{[l]} - \eta \cdot \nabla_{w^{[l]}} L \tag{15}$$

$$b^{[l]} = b^{[l]} - \eta \cdot \nabla_{b^{[l]}} L \tag{16}$$

Backpropagation is efficient because it calculates the gradients for all weights and biases with only two passes through the network: one forward and one backward. By systematically applying the chain rule, backpropagation avoids redundant computations of gradients, thereby facilitating the training of deep neural networks.

C. Recurrent Neural Networks

We can extend the background above to formulate recurrent neural networks.

1) Problem and Data: RNNs are designed to process sequential data, which has many useful applications. For example, when a human reads, we understand each word based on your understanding of previous words. Traditional feedforward neural networks cannot use its reasoning about previous events to inform later decisions.

Reflecting this, the input data for RNNs is usually organized into sequences, and each input example (also known as a sample or instance) is structured as a series of time steps. This means that the input feature x for a single training example is no longer just a vector, but rather a matrix, where each row corresponds to a feature vector at a particular time step. The input to the RNN for each sample is then a matrix $X = [x^{(1)}, x^{(2)}, \ldots, x^{(T)}]$, where T is the number of time steps. Similarly, the labels y can also be sequential, representing the desired output at each time step, such that we have $Y = [y^{(1)}, y^{(2)}, \ldots, y^{(T)}]$.

2) Model Definition: RNNs are essentially multi-layer networks (see section) in terms of architecture. However, instead of just the output of each layer getting passed to the next, RNNs have a "hidden state" that serves as a form of memory to captures information about what has been processed so far in a sequence. This architecture of an RNN is shown in Figure 3, which also demonstrates that a RNN can be "unrolled" through time.

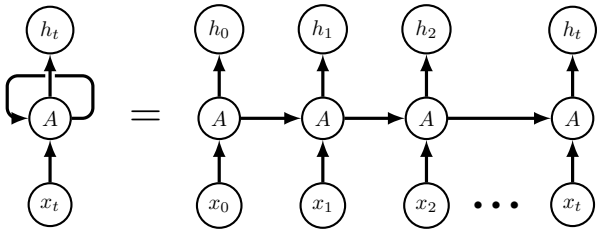

Fig. 3: Recurrent Neural Network.

Let us examine the components of RNNs:

- **Input layer:** Similar to MLPs, this layer receives the input features at each time step.
- **Recurrent layers:** One or more layers where each neuron has a recurrent connection to itself, which allows the network to retain the previous state's information.
- **Output layer:** Produces the prediction for the current time step and can also provide the output for the entire sequence.

The computation of the hidden state at time t can be described as follows:

$$h(t) = \sigma(w_{xh} \cdot x^{(t)} + w_{hh} \cdot h^{(t-1)} + b_h) \tag{17}$$

where:

- $x^{(t)}$ is the input at time t
- $h^{(t)}$ is the hidden state at t
- w_{xh} are weights from the input layer to the hidden layer
- w_{hh} are weights connecting the hidden layer to itself across consecutive time steps. The hidden state from the previous time step $h^{(t-1)}$ is multiplied by these weights.
- b_h is a bias term for hidden state updates.
- σ is an activation function for hidden state updates.

The output at time t is then determined by the hidden state, such that:

$$o(t) = \alpha(w_{hy} \cdot h^{(t)} + b_y) \tag{18}$$

where $o(t)$ is the output at time t, w_{hy} are weights connecting the hidden state to the output, b_y is an output bias term. We also include an output activation function α.

3) Backpropagation Through Time: Due to the time-dependent nature of RNNs, training them requires an algorithm caleld Backpropagation Through Time (BPTT). BPTT is an extension of the standard backpropagation algorithm, which unfolds the RNN across time steps and then calculates gradients for each time step in the sequence. This allows for the network to be trained on the dependencies across time.

Let us consider an objective function $J(\Theta)$ for RNNs, which typically include the loss computed at each time step, summed over all time steps, such that

$$J(\Theta) = \sum_{t=1}^{T} L^{(t)}(o^{(t)}, y^{(t)}; \Theta) \tag{19}$$

We'll denote the objective loss at each timestep t as $J^{(t)}$, which is a function of the true target $y^{(t)}$ and the predicted output $o^{(t)}$.

To update the parameters using gradient descent, we need to calculate the gradient of the objective function with respect to each parameter. For a given parameter θ in Θ, the gradient is:

$$\frac{\partial J}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial \theta} \tag{20}$$

Like regular backpropagation, we compute the compute the gradients by applying the chain rule, starting with weights closest to the output and working backwards. For the output weights w_{hy}, we note that J is a function of $o^{(t)}$, which is a function of w_{hy}. This lets us the gradients are calculated as:

$$\frac{\partial J}{\partial w_{hy}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial w_{hy}} \tag{21}$$

Given $o(t) = \alpha(w_{hy} \cdot h^{(t)} + b_y)$, we have:

$$\frac{\partial o^{(t)}}{\partial w_{hy}} = \alpha'(w_{hy} h^{(t)} + b_y) h^{(t)} \tag{22}$$

where α' is the derivative of the activation function with respect to its input.

For w_{xh} and w_{hh}, the gradients are more complex due to their influence on the hidden state over time. Using the chain rule, we accumulate gradients from all future steps:

$$\frac{\partial J}{\partial w_{xh}} = \sum_{t=1}^{T} \sum_{k=t}^{T} \frac{\partial L^{(k)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial w_{xh}} \tag{23}$$

$$\frac{\partial J}{\partial w_{hh}} = \sum_{t=1}^{T} \sum_{k=t}^{T} \frac{\partial L^{(k)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial w_{hh}} \tag{24}$$

To compute $\frac{\partial J^{(k)}}{\partial h^{(k)}}$, we apply the chain rule again, considering that $h^{(k)}$ affects all future outputs $o^{(k+1)}, o^{(k+2)}, \ldots, o^{(T)}$ and hidden states $h^{(k+1)}, h^{(k+2)}, \ldots, h^{(T)}$.

Now we can update each parameter θ:

$$\theta = \theta - \eta \frac{\partial J}{\partial \theta} \tag{25}$$

This is written with a general parameter θ for brevity. An example of updating a specific parameter like w_{xh} would just be $w_{xh} = w_{xh} - \eta \frac{\partial J}{\partial w_{xh}}$. This is the most basic parameter update method, and other options more complex options exist, such as AdaGrad [5].

Recurrent neural networks are also easily extended to multi-layer form; each layer simply has its own hidden state. We then backpropagate through all of the layers.

III. Approach

Given the background of basic neural networks, we will now focus on the application of recurrent neural networks to build a simple language model from scratch. The objective of the model is to predict the next character in a sequence given the previous characters. This is done by modeling the probability distribution of the next character. This process is carried out over many sequences from a large body of text during the training phase, during which the model parameters are adjusted.

The specific model is defined as:

$$h_t = \tanh(w_{xh} \cdot x_t + w_{hh} \cdot h_{t-1} + b_h) \tag{26}$$

$$y_t = (w_{hy} \cdot h_t + b_y) \tag{27}$$

$$p_t = \frac{e^{y_t}}{\sum_j e^{y_{t_j}}} \tag{28}$$

$$J = -\sum_{t=1}^{T} \log(p_{t,\text{target}_t}) \tag{29}$$

For the hidden state h_t, the $\tanh$ function is chosen as an activation function to squash inputs to $[-1, 1]$, which makes the mean of activations closer to zero, resulting in faster training. Furthermore, gradients can be positive or negative so that the gradient descent process can travel in more than one direction, which is ideal for learning.

y_t are the unnormalized log probabilities (logits) for next characters at time t. The softmax function is used as an activation function to interpret the logits as a probability distribution. Softmax is a normalized exponential function (shown above as p_t), such that a vector of real numbers is converted into a probability distribution. The outputs of the softmax function are non-negative and sum to 1, making it suitable for interpreting the outputs as probabilities, allowing us to model the probability of the next character being output.

The objective function J takes the summation of cross-entropy loss over the sequence length used for training. Cross-entropy is a popular loss function quantifying how much one probability distribution diverges from another, and is usually defined as:

$$L_{\text{cross-entropy}} = y_i \log(\hat{y}_i) \tag{30}$$

where y_i is the true probability of the i-th class, and $\hat{y}_i$ is the predicted probability of the i-th class, as output by our RNN. Cross-entropy loss measures the difference between the predicted probabilities (the output of the softmax function in a neural network) and the actual distribution (the true labels). Since our data will use a one-hot representation of characters, $y_i = 1$ for the true class, and $y_i = 0$ for all other classes. Thus, for our case, the objective function simplifies to:

$$J(y, \hat{y}) = -\sum y_i \log(\hat{y}_i) = -\log(\hat{y}_{\text{true class}}) \tag{31}$$

Lastly, AdaGrad [5] (short for Adaptive Gradient Algorithm) is used as an optimization method. AdaGrad is a standard optimization method for machine learning. Instead of using a basic parameter update step like $\Theta \leftarrow \Theta - \eta \cdot \nabla_\Theta J(\Theta)$ introduced earlier, it uses a more complex function:

$$\theta_{t+1}(i) = \theta_t(i) - \frac{\eta}{\sqrt{G_t(i) + \epsilon}} \cdot \nabla J(\theta_t(i)) \tag{32}$$

where $\Theta_{t+1}(i)$ is the i-th parameter at time step $t + 1$, η is the initial learning rate and ϵ is a small smoothing term to avoid division by zero. Most notably, G_t is a vector which holds the sum of the squares of past gradients up to time step t, such that:

$$G_t(i) = G_{t-1}(i) + (\nabla J(\theta_t(i))^2 \tag{33}$$

Here, $G_t(i)$ is the i-th component of vector G_t.

The purpose of this is to the learning rate η to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent ones. This is particularly useful for dealing with sparse data (like text), where some features (like some words) may appear very infrequently.

To understand and quantify model performance, we keep track of loss values throughout the training process. Model output is also sampled throughout the training process to gain a qualitative understanding of model behaviour.

IV. Implementation

Each part of the model and learning algorithm introduced in Section III is implemented in Python code. The only library used is NumPy [6].

A. Problem and Data

To define our language modeling problem, we first handle and process data; characters are given a numerical representation to allow for easier computation. Text data from a simple text file (`input.txt`) is read and processed. Each unique character is identified, and two mappings are created: one for converting characters to unique integers `char_to_ix`, and another for the reverse `ix_to_char`.

```
data = open('input.txt', 'r').read()
chars = list(set(data))
data_size,vocab_size = len(data),len(chars)
char_to_ix = {ch: i for i,
            ch in enumerate(chars)}
ix_to_char = {i: ch for i,
            ch in enumerate(chars)}
```

B. Model Definition

We then begin by defining the RNN model, initializing model weights with random values and biases to zero. Hyperparameters such as learning rate and hidden layer size are also defined.

The `seq_length` variable specifies the length of the sequence of characters that the RNN will consider at one time (also known as the number of time steps the RNN will unroll), or the "window" of characters it looks at to make a single prediction. In each training iteration, the model processes `seq_length` characters as a batch. It doesn't process the whole text file at once but rather in these smaller sequences. Then, the RNN is unrolled for `seq_length` steps in time during the backpropagation through time. This means that when calculating gradients, the RNN considers the influence of `seq_length` previous characters at each step. A larger `seq_length` allows the network to learn more extended patterns (longer dependencies), but it also requires more memory and computation. Conversely, a shorter `seq_length` may limit the context the model can learn but is computationally less expensive.

```
# MODEL PARAMETERS
w_xh = np.random.randn(hidden_size,
                       vocab_size)*0.01
w_hh = np.random.randn(hidden_size,
                       hidden_size)*0.01
w_hy = np.random.randn(vocab_size,
                       hidden_size)*0.01
bh = np.zeros((hidden_size, 1))
by = np.zeros((vocab_size, 1))

# HYPERPARAMETERS
hidden_size = 100
seq_length = 25
learning_rate = 1e-2
```

C. Loss Function

Next, we define a loss function which computes the forward pass, loss, and backward pass of the network. It takes a list of input characters and target characters (both encoded as integers), and the previous hidden state. It outputs the loss, gradients for the model parameters, and the last hidden state.

Here, we encode the integer representation of data points into a one-hot vector, such that every element in the vector is zero, except the element at the index of the character (from our previous `char_to_ix` operation) is 1. Then: - The hidden state `hs[t]` is updated using the previous hidden state and the current input. - The raw outputs `ys[t]` are calculated by applying the weights from the hidden state to the output layer. - The probabilities `ps[t]` for the next character are calculated using the softmax function. - The cross-entropy loss for the current prediction is added to the total loss.

```
# LOSS/OBJECTIVE FUNCITON
def lossFun(inputs, targets, hprev):

xs, hs, ys, ps = {}, {}, {}, {}
hs[-1] = np.copy(hprev)
loss = 0

# FORWARD PASS
for t in range(len(inputs)):

    # 1-OF-K ENCODING
    xs[t] = np.zeros((vocab_size,1))
    xs[t][inputs[t]] = 1

    # HIDDEN STATE
    hs[t] = np.tanh(np.dot(Wxh, xs[t])
            + np.dot(Whh, hs[t-1]) + bh)

    # LOGITS
    ys[t] = np.dot(Why, hs[t]) + by

    # CHARACTER PROBABILITIES
    sum_logprobs = np.sum(np.exp(ys[t])
    ps[t] = np.exp(ys[t])/sum_logprobs

    # SOFTMAX AND CROSS-ENTROPY
    loss += -np.log(ps[t][targets[t],0])
```

Then, we do the backward pass. We loop backward through the sequence and compute: - The gradients for the output weights (`dWhy`) and biases (`dby`) - The gradient for the hidden state (`dh`) is calculated and then adjusted for the effect of the tanh activation function (`dhaw`). - The gradients for the input-to-hidden weights (`dWxh`, hidden-to-hidden weights `dWhh`, and hidden biases `dbh` are updated. - The `dhnext` gradient is prepared for the next iteration of the loop.

```
# BACKWARD PASS
dWxh, dWhh, dWhy = np.zeros_like(Wxh) ...
dbh, dby = np.zeros_like(bh),...
dhnext = np.zeros_like(hs[0])

for t in reversed(range(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1
    dWhy += np.dot(dy, hs[t].T)
    dby += dy

    # BACKPROP THROUGH h
    dh = np.dot(Why.T, dy) + dhnext

    # BACKPROP THROUGH tanh
    dhraw = (1 - hs[t] * hs[t]) * dh

    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
```

We add a gradient clipping step 1. to prevent exploding gradients, the gradients are clipped to be between -5 and 5. The exploding gradient problem is discussed more in the Limitations section.

```
# GRADIENT CLIPPING
for dparam in [dWxh,dWhh,dWhy,dbh,dby]:
    np.clip(dparam, -5, 5, out=dparam)
```

V. Results

For data, all of the works of William Shakespeare are concatenated and put into the 'input.txt'. Thus, the objective of the model is to generate writing in the style of Shakespeare.

Since our training loop samples some model output every 100 iterations (sequences trained), we can examine the output and loss of the model over time. We start with a single-layer RNN with a hidden layer size of 100.

At iteration zero, the model is not yet trained so our output is very random:

```
----
iter 0, loss: 104.35967499060304
----
CqueXY:RZWqPDTWPalrv'WFLv.Hms,$cjDKHzq
$KowgybRzbobjmaTHR'bdgCdBNSxmgajiJXAt
:YKewE&rgDznhSPwqp&KMSZSR$KlwjHkfXHV'
aZFUIP-;MKxuPVDUmdB3CbvkrbHlsDnAXLZcL
cvUjcTS,hvmYqUbhxj?u?kwCnFvztOCI&MxIYcT
```

As we keep training, the model output gradually improves, outputting English words (and sequences that somewhat resemble English). We can also begin to observe a somewhat Shakespearean style!

```
---
iter 22600, loss: 54.97304638057519
---
Ate you chis lose! Meat and the our that
vistrers? Let not for where, dame my forsta
```

Eventually, after a lot of iterations (almost a million), the loss is much lower and the output looks quite good!

```
----
iter 997100, loss: 44.43487950312566
----
and shall of my heakse herrage, How shaid
I thoughty sween dofl ampaut
To kasw, of the fuld on alinion;
Te to-near hare the comery, not mach
save you lagd thtures beace re
```

How can we get even better output? The obvious solution is to employ the classic deep learning strategy: more layers and more parameters. Thus, we change the model implementation slightly to run a 3-layer RNN. The hidden layer size was also increased to 512. Much of the code is the same as the single-layer RNN, above, just extended to 3 layers. For example, the backpropagation step becomes:

```
for t in reversed(range(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1
    dWhy += np.dot(dy, hs3[t].T)
    dby += dy
    dh3 = np.dot(Why.T, dy) + dhnext3
    dh3raw = (1 - hs3[t] * hs3[t]) * dh3
    dbh3 += dh3raw
    dWxh3 += np.dot(dh3raw, hs2[t].T)
    dWhh3 += np.dot(dh3raw, hs3[t - 1].T)
    dhnext3 = np.dot(Whh3.T, dh3raw)

    dh2 = np.dot(Wxh3.T, dh3raw) + dhnext2
    dh2raw = (1 - hs2[t] * hs2[t]) * dh2
    dbh2 += dh2raw
    dWxh2 += np.dot(dh2raw, hs[t].T)
    dWhh2 += np.dot(dh2raw, hs2[t - 1].T)
    dhnext2 = np.dot(Whh2.T, dh2raw)

    dh = np.dot(Wxh2.T, dh2raw) + dhnext
    dhraw = (1 - hs[t] * hs[t]) * dh
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t - 1].T)
    dhnext = np.dot(Whh.T, dhraw)
```

Here are some sample outputs:

```
----
iter 468200, loss: 42.90417002098809
----
GLAMUSE:
As thou peasant kister,
he revere his this rrous,
To gentle,
And vor gud!

DUDN VINCE:
All that now conly wath are bedy, our
greacd afparfer po,
And vorn of d

----
iter 421000, loss: 44.14635583930259
----
House with a doy, new ade
bid but condesiegh as her againss clavk;
Arature make heart keel that all, where;
So had it prait, are is your make with not
```

This generates higher quality output with less iterations (although each iteration takes longer). The model also begins to pick up on the structure of plays and generates character names! A particularly frequent character name is "King", which makes sense as many Shakespeare plays have kings in them.

```
----
iter 458900, loss: 42.673977586462485
----
KING.
O'l lid, your comes tell, jetite;
Uraty sut, to be not me trance thesinem
The jobreined yearsick.
```

To achieve a more concrete understanding of model perfor-

mance over time, the loss is plotted over a million training iterations for the initial single-layer RNN. After consistent decrease in loss at the beginning, the seemed to plateau in terms of loss around a value of 40. The loss would still gradually decrease, but at a very slow rate and very inconsistently (moving up and down). This may indicate the weakness of our basic RNN architecture; our model may simply not be complex enough to fully capture the patterns in the data, or be susceptible to problems during training like the vanishing and exploding gradient problems (discussed in Section VI-A).

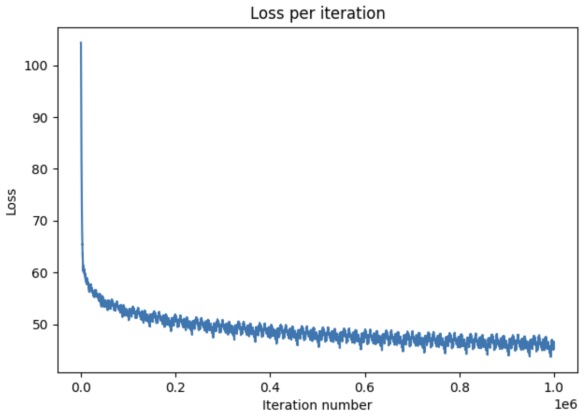

Fig. 4: Model loss value over 1 million training iterations.

VI. Discussion

Given the simplicity of our implementation and the relatively small size of data and lack of computational resources, the RNN performs very well. Not only does it spell English words (or at least produce English-like output), but it is able to capture Shakespearean style and play structure with character names and dialogue. However, there are some significant limitations that lead to room for improvement.

A. Limitations

Notably, models seemed to plateau in terms of loss around a value of 40. The loss would still gradually decrease, but at a very slow rate and very inconsistently (moving up and down). This may indicate the weakness of our basic RNN architecture; our model may simply not be complex enough to fully capture the patterns in the data, or be susceptible to problems during training like the vanishing and exploding gradient problems [7].

RNNs are plagued by vanishing and exploding gradients slope of the loss function along the error curve.

When the gradient is too small (vanishing), updates to the weight parameters until they become insignificant, which means that the model learns very slowly or not at all. Exploding gradients occur when the gradient is too large, creating an unstable model (NaN results).

These problems especially bad for the recursive structure of RNNs, as the same weights are applied recursively at each time step of the input sequence. This means that during backpropagation, the gradients of these weights are also applied recursively through time. Because the same weights are used at each time step, any small gradients become exponentially smaller as they are propagated through each time step (vanishing), and conversely, any large gradients become exponentially larger (exploding).

RNN-type networks with specialized architectures, such as Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRU) mitigate the vanishing and exploding gradient problems. This is done by adding mechanisms to allow the information to bypass the traditional path through the RNN. This means that the gradient has a shortcut where it can flow without being multiplied by the weight matrix at each time step, thus reducing the risk of vanishing or exploding. These architectures are also better at capturing long-term dependencies by using a "memory cell" that can maintain information in memory for long periods of time.

VII. Conclusion

This project demonstrated the power of basic multivariable calculus concepts in terms of complex and relevant applications, namely language modeling. A deep understanding of neural networks is established from first principles, then expanded to multilayer networks and eventually recurrent networks. Based on this understanding, implementation and experimentation were conducted for the interesting problem of Shakespeare generation. Overall, this paper provided an opportunity to gain a deeper understanding of machine learning and to build a small but powerful application.

Acknowledgment

The tikz diagrams for neural network architecture (Figures 1, 2, 3) are based on source code from David Stutz's repository of LaTeX resources [8].

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[2] M. Mozer, "A focused backpropagation algorithm for temporal pattern recognition," *Complex Systems*, vol. 3, 01 1995.

[3] S.-I. Amari, "Learning patterns and pattern sequences by self-organizing nets of threshold elements," *IEEE Transactions on Computers*, vol. C-21, no. 11, pp. 1197–1206, 1972.

[4] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[5] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[7] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR*, vol. abs/1211.5063, 2012. [Online]. Available: http://arxiv.org/abs/1211.5063

[8] D. Stutz, "Collection of latex resources and examples," https://github.com/davidstutz/latex-resources.