

## Lecture 2 Recurrent Neural Networks (RNN)

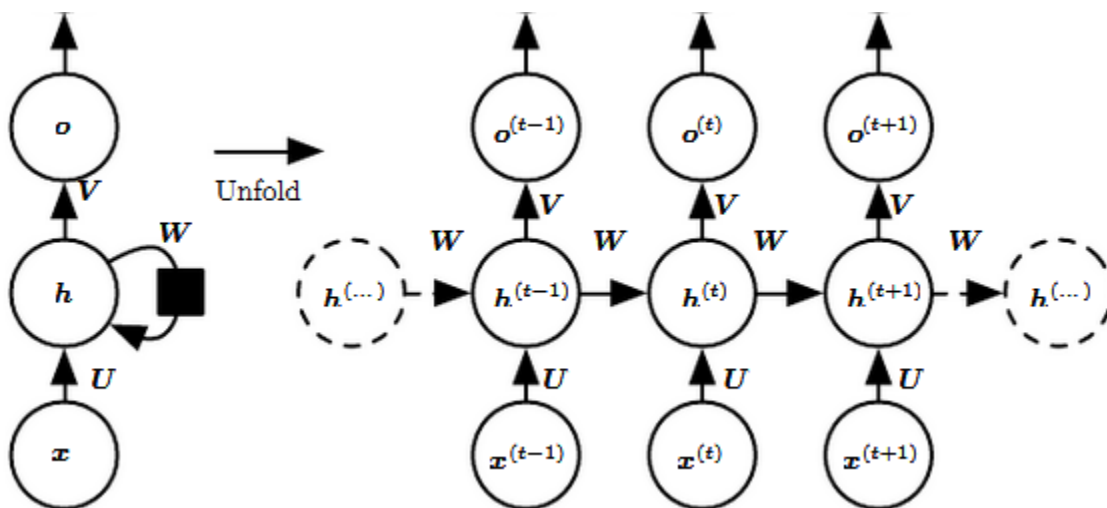
### 1.1 What is RNN?

A recurrent neural network is a neural network that is specialized for processing a sequence of data:

$$x(t) = \{x(1), \dots, x(T)\}$$

with the time step index  $t$  ranging from 1 to  $T$ . For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs. In an NLP problem, if you want to predict the next word in a sentence it is important to know the words before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations. Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far.

Architecture : Let us briefly go through a basic RNN network.



#### Key Components of RNN Layers

- **Input Layer:** Takes the input data at each time step.
- **Hidden Layers (Recurrent Layers):** These layers maintain a hidden state across time steps and are responsible for capturing temporal dependencies in the data. The hidden layer at each time step shares the same weights across all time steps, which is a defining characteristic of RNNs.
- **Output Layer:** Produces the output at each time step or the final output depending on the task (e.g., many-to-many, many-to-one).

**Network (Model Processing):** On the right side, an RNN is being *unrolled* (or unfolded) into a full network. By unrolling we mean that we write out the network for the complete sequence. In a simple RNN (vanilla RNN), there is typically one recurrent hidden layer, which may be unrolled across several time steps. Despite the unrolling process, this counts as one layer because the same set of weights is used at each time step.

For example, if the sequence we care about is a sentence of 3 words, the network would be unrolled into a 3 time-step neural network, one time step for each word. Similarly, we can assume the sequence we care

about is a time series of three trading days, making forecasts on open price or the market movement next day using each day's trading price and volume as inputs. The same set of weights (parameters) are applied to data observations taken into the network at each time step and to be estimated.

More often, a stacked RNN may consist of multiple recurrent layers (hidden layers) stacked on top of each other. For example, a 3-layer RNN would have three recurrent layers, each feeding into the next. The outputs of the first recurrent layer are fed as inputs to the second recurrent layer, and so on. Number of layers in this case = Number of stacked recurrent layers. The input layer (like an embedding layer in NLP tasks) and the output layer are often counted separately, particularly when defining the architecture. They are not typically considered "neural network layers" in the context of the recurrent component but contribute to the total count of layers in the model. In sum, count only the recurrent layers (hidden layers) when determining the number of layers in an RNN.

**Weights:** The left side of the above diagram shows parameters of an RNN. There are three sets of parameters to be estimated here. The RNN has input  $x$  to hidden connections  $h$  parameterized by a weight matrix  $U$ , hidden-to-hidden recurrent connections parameterized by a weight matrix  $W$ , and hidden-to-output connections parameterized by a weight matrix  $V$ . All these weights (parameters)  $\{U, V, W\}$  are shared across time.

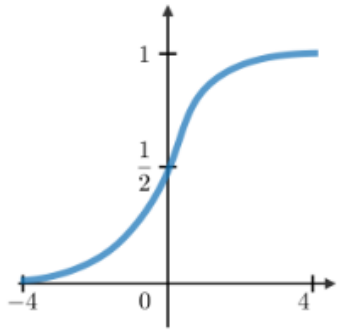
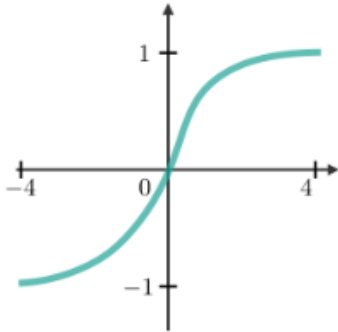
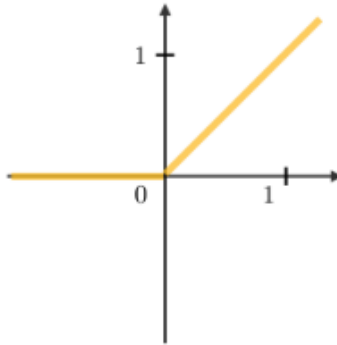
**Input:**  $x(t)$  is taken as the input to the network at time step  $t$ . For example,  $x(1)$ , could be a one-hot vector corresponding to a word of a sentence.  $x(1)$  could also be the market open price of a particular stock.

**Hidden state:**  $h(t)$  represents a hidden state at time  $t$  and acts as "memory" of the network.  $h(t)$  is calculated based on the current input and the previous time step's hidden state:

$$h(t) = f(U \cdot x(t) + W \cdot h(t - 1))$$

The function  $f(\cdot)$  is taken to be a non-linear transformation such as sigmoid, tanh or ReLU.

**Commonly used activation functions:** The most common activation functions used in RNN models are described below:

Sigmoid	Tanh	RELU
$f(z) = \frac{1}{1 + e^{-z}}$	$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$f(z) = \max(0, z)$
		

**Output:**  $o(t)$  illustrates the output of the network. In the figure I just put an arrow after  $o(t)$  which is also often subjected to non-linearity, especially when the network contains further layers downstream. For example,  $o(t)$  could be the open price of a stock the next day.  $o(t)$  could also be the probability that the word sequence is going to take a particular word.

## 1.2 Forward Pass

The figure does not specify the choice of activation function for the hidden units. Before we proceed, we make few assumptions:

- **Assumption 1:** we assume the hyperbolic tangent activation function for hidden layer.
- **Assumption 2:** we assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regard the output :  $o(t)$  as giving the un-normalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector  $\hat{y}$  of normalized probabilities over the output.

The RNN forward pass can thus be represented by the below set of equations.

$$\begin{aligned}a(t) &= b + W h(t - 1) + Ux(t) \\h(t) &= \tanh(a(t)) \\o(t) &= c + Vh(t) \\\hat{y}(t) &= \text{softmax}(o(t))\end{aligned}$$

This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of  $x$  values paired with a sequence of  $y$  values would then be just the sum of the losses over all the time steps. We assume that the outputs :  $o(t)$  are used as the argument to the softmax function to obtain the vector  $\hat{y}$  of probabilities over the output. Let  $\theta$  denote all weights and bias,  $\theta = \{U, V, W, b, c\}$ . We assume that the loss  $L(y|x, \theta)$  is the negative log-likelihood of the true target  $y(t)$  given the input so far.

**Note:**

- Explain what predictive density is and what is likelihood.
- Explain what is softmax process.

## 1.3 Backward Pass

The gradient computation involves performing a forward propagation pass moving left to right through the graph shown above followed by a backward propagation pass moving right to left through the graph. The runtime is  $O(T)$  and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also  $O(T)$ . The back-propagation algorithm applied to the unrolled graph with  $O(T)$  cost is **called back-propagation through time (BPTT)**. Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.

## How Backpropagation Through Time Works:

### Step 1: Forward Pass

- The RNN processes input data one time step at a time.
- At each time step  $t$ , the RNN takes an input  $x_t$  and combines it with the previous hidden state  $h_{t-1}$  to produce a new hidden state  $x_t$ .
- The hidden state  $x_t$  captures information from both the current input and the preceding context, effectively maintaining a form of memory.

### Step 2: Error Calculation

- At the end of the sequence, the output is compared to the target values to calculate the loss (e.g., using Mean Squared Error, Cross-Entropy Loss).
- This loss is then used to measure how far off the predicted output is from the actual output.

### Step 3: Backward Pass (Backpropagation Through Time)

- The backward pass starts by calculating the gradients of the loss concerning the network's weights, but unlike standard neural networks, this process must consider all time steps.
- BPTT unfolds the RNN through all time steps (like a deep feedforward network), treating each step as a layer.

### Step 4: Gradient Calculation

- Gradients are computed by applying the chain rule of calculus, propagating errors backward through time.
- For each time step  $t$ , the gradient of the loss with respect to the weights depends on:
  - The current state's contribution to the loss.
  - The contribution of all future states due to the recursive nature of the RNN.
- The recursive dependencies mean that the gradient at each time step depends not only on that specific time step but also on how past and future time steps affect the hidden state.

### Step 5: Weight Updates

- Using the gradients computed, the model's weights (input weights, recurrent weights, and biases) are updated according to an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam.
- The updates are typically small adjustments aimed at minimizing the loss function over time.

## Key Challenges in BPTT:

**Vanishing and Exploding Gradients:** Gradients can shrink (vanish) or grow (explode) exponentially as they are propagated backward through time, especially in long sequences. This makes learning long-term dependencies difficult because the gradients either become too small to make significant updates or too large to cause instability.

**Computational Complexity:** Since BPTT involves unfolding the RNN through time, it can be computationally expensive, especially with long sequences.

Memory Usage: Storing all the intermediate states and gradients across time steps can consume significant memory, especially for long sequences.

Mitigations:

- Gradient Clipping: To address exploding gradients, gradient clipping limits the gradients to a maximum threshold value, preventing them from growing too large.
- Using Specialized Architectures: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) networks are designed to mitigate the vanishing gradient problem by maintaining information over longer sequences.

In sum, BPTT unrolls the RNN across time steps, treating each step as a layer, and backpropagates errors through all time steps. It adjusts the weights based on how the network's performance deviates from the target, taking into account both past and future contributions of hidden states. Managing BPTT effectively is crucial for training deep and complex RNNs successfully.

### 1.4 Computing Gradients

Given our loss function  $L(y|x, \theta) = L(y|x, U, V, W, b, c)$ , we need to calculate the gradients for our three weight matrices  $U, V, W$  and two bias terms:  $b, c$ . We update them with a learning rate  $\eta$ . Similar to normal back-propagation, the gradient gives us a sense of how the loss is changing with respect to each parameter. First, we update one of the parameters  $W$  to minimize loss with the following equation:

$$W^{k+1} \leftarrow W^k - \eta \frac{\partial L}{\partial W^k} = W^k - \eta \nabla_W L$$

The same is to be done for the other parameters  $U, V, b, c$  as well.

Let us now compute the gradients by BPTT for the RNN equations above. The nodes of our computational graph include the parameters  $U, V, W, b, c$  as well as the sequence of nodes indexed by  $t$  for  $x(t), h(t), o(t)$  and  $L(t)$ . For each node  $t$  we need to compute the gradient  $\nabla L(t)$  recursively, based on the gradient computed at nodes that follow it in the graph.

**Gradient with respect to output  $o(t)$**  is calculated assuming the  $o(t)$  are used as the argument to the softmax function to obtain the vector  $\hat{y}(t)$  of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target  $y(t)$ .

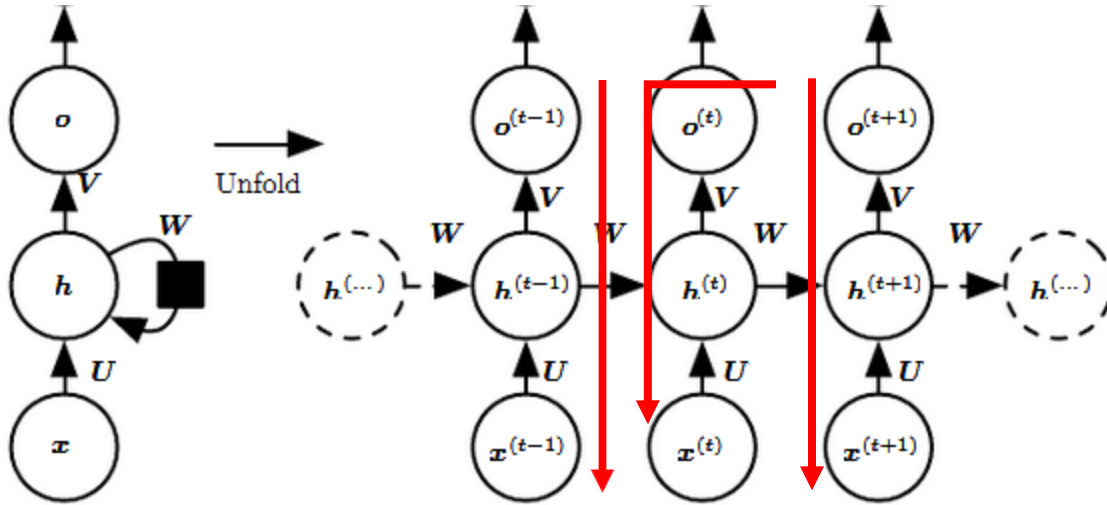
$$(\nabla_{o(t)} L)_i = \frac{\partial L}{\partial o_i(t)} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial o_i(t)} = \hat{y}_i(t) - 1_{i=y(t)}$$

Where  $i$  represents one of the  $N$  possibilities in the softmax process.

Note:

- Explain Gradient Descent Algorithm
- Explain how to derive the above solution.

Let us now understand how the gradient flows through hidden state  $h(t)$ . This we can clearly see from the below diagram that at time  $t$ , hidden state  $h(t)$  has gradient flowing from both current output and the next hidden state  $h(t + 1)$ .



We work our way backward, starting from the end of the sequence. At the final time step  $T$ ,  $h(T)$  only has  $o(T)$  as a descendant, so its gradient is simple:

$$\nabla_{h(T)} L = V' \nabla_{o(T)} L$$

We can then iterate backward in time to back-propagate gradients through time, from  $t = T - 1$  down to  $t = 1$ , noting that  $h(t)$  (for  $t < T$ ) has as descendants both  $o(t)$  and  $h(t + 1)$ . Its gradient is thus given by:

$$\begin{aligned} \nabla_{h(t)} L &= \left( \frac{\partial h(t+1)}{\partial h(t)} \right)' (\nabla_{h(t+1)} L) + \left( \frac{\partial o(t)}{\partial h(t)} \right)' (\nabla_{o(t)} L) \\ &= W' \text{diag}(1 - h(t+1)^2) (\nabla_{h(t+1)} L) + V' (\nabla_{o(t)} L) \end{aligned}$$

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. The gradient calculations using the chain rule for all parameters is:

$$\begin{aligned} \nabla_c L &= \sum_t^T \left( \frac{\partial o(t)}{\partial c} \right)' \nabla_{o(t)} L = \sum_t^T \nabla_{o(t)} L \\ \nabla_b L &= \sum_t^T \left( \frac{\partial h(t)}{\partial b} \right)' \nabla_{h(t)} L = \sum_t^T \text{diag}(1 - h(t)^2) \nabla_{h(t)} L \\ \nabla_V L &= \sum_t^T \sum_i^N \frac{\partial L}{\partial o_i(t)} \nabla_{V_i} o_i(t) = \sum_t^T (\nabla_{o(t)} L) h(t)' \\ \nabla_W L &= \sum_t^T \sum_i^N \frac{\partial L}{\partial h_i(t)} \nabla_{W_i} h_i(t) = \sum_t^T \text{diag}(1 - h(t)^2) (\nabla_{h(t)} L) h(t-1)' \end{aligned}$$

$$\nabla_U L = \sum_t \sum_i^N \frac{\partial L}{\partial h_i(t)} \nabla_{U_i} h_i(t) = \sum_t \text{diag}(1 - h(t)^2) (\nabla_{h(t)} L) x(t)'$$

We are not interested in deriving these equations here, rather implementing these. Below are very good posts that provide detailed derivation of these equations.

<https://jramapuram.github.io/ramblings/rnn-backprop/>

<http://willwolf.io/2016/10/18/recurrent-neural-network-gradients-and-lessons-learned-therein/>

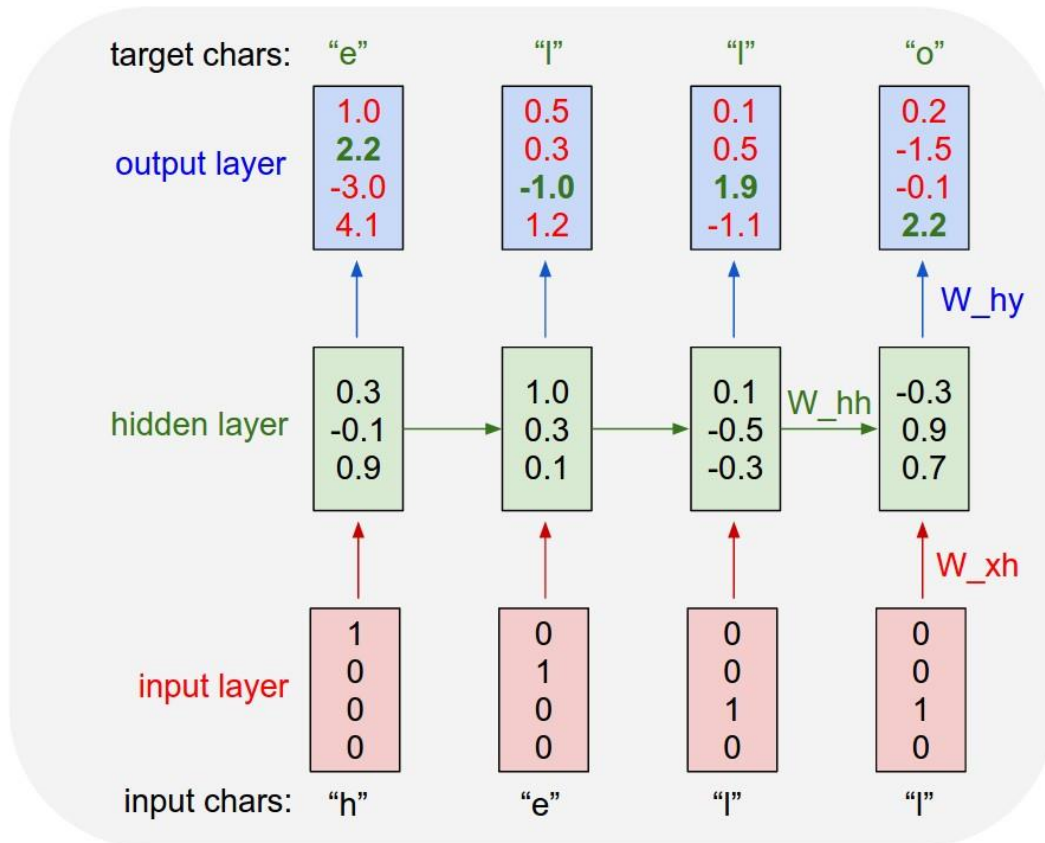
### 1.5 Implementation

After reading about the model processing of Recurrent Neural Networks, I know:

- They work with sequences (eg. stock data, natural language).
- We can train them with backprop, the same way we do with convolutional nets.
- They are flexible and work with a variety of structures.
  - One-to-many: Take an image and return a caption for it.
  - Many-to-one: Take a movie review and return if it's positive or negative.
  - Many-to-Many: Take an English sentence and return a Spanish sentence.

The famous example of Andrej Karpathy's simple Min-Char RNN

(<https://gist.github.com/karpathy/d4dee566867f8291f086>) would have weight matrices with thousands of parameters.



### Example 1: a naive RNN

With simplicity in mind, let's build a network that predicts the next output in the sequence:

{1, 0, 1, 0, 1, 0, 1, 0, ... }

This seems simple. Maybe even too simple, but a sequence like this will allow us to have small weight matrices, the largest of which will be just 3 x 3.

Our vocab\_size x is just 2 x 1 (each input character is 1 or 0) and since we're predicting such a simple pattern, we'll use a hidden\_size h of 3 x 1.

Our network is fed an input of 1,0,1,0 and is tasked with predicting the target output sequence 0,1,0,1, one character at a time. Since our network is untrained, note that it outputs probabilities of roughly 0.50 at each time step.

We define:

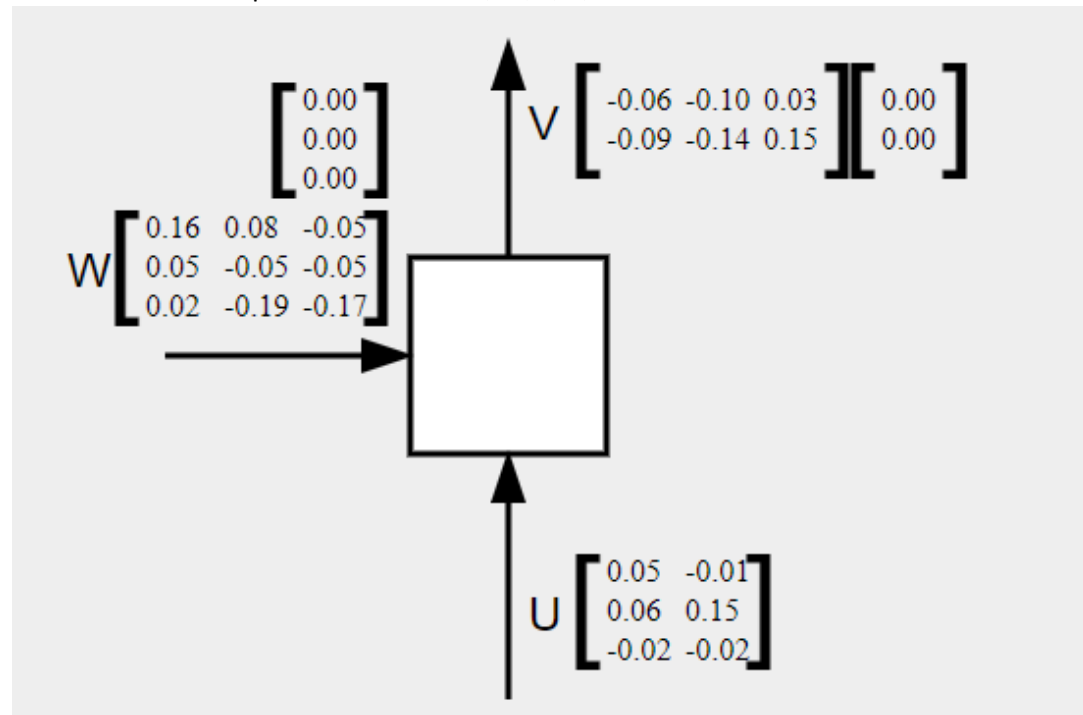
Character "1" is represented by vector,  $character\ 1 = \begin{bmatrix} 0.00 \\ 1.00 \end{bmatrix}$ .

Character "0" is represented by vector,  $character\ 0 = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$ .



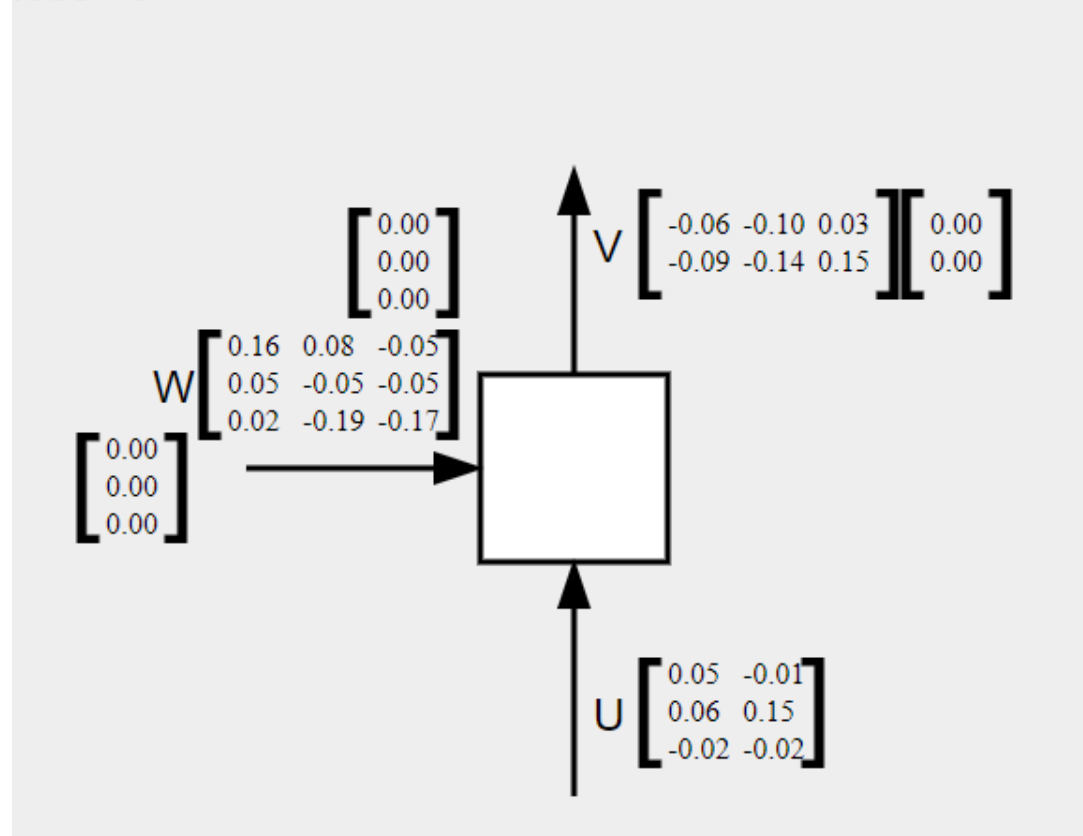
Loop 0:  $t = 0$

Line 9 – 22 initialize parameter values  $U, W, b, V, c$

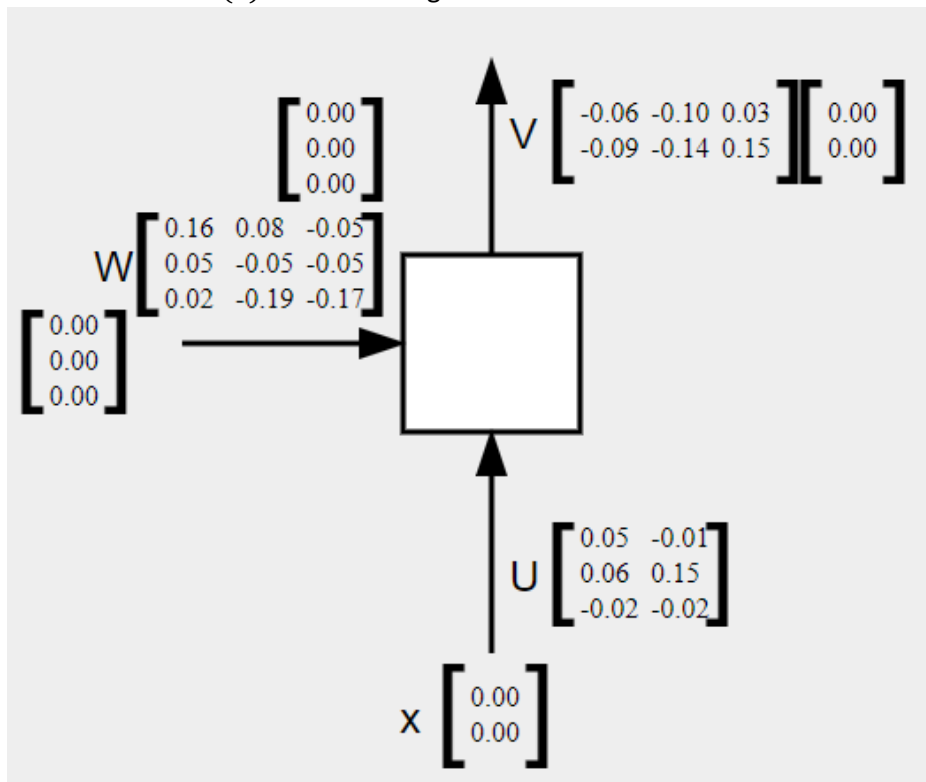


Line 24-28 initialize hidden state  $h(-1)$

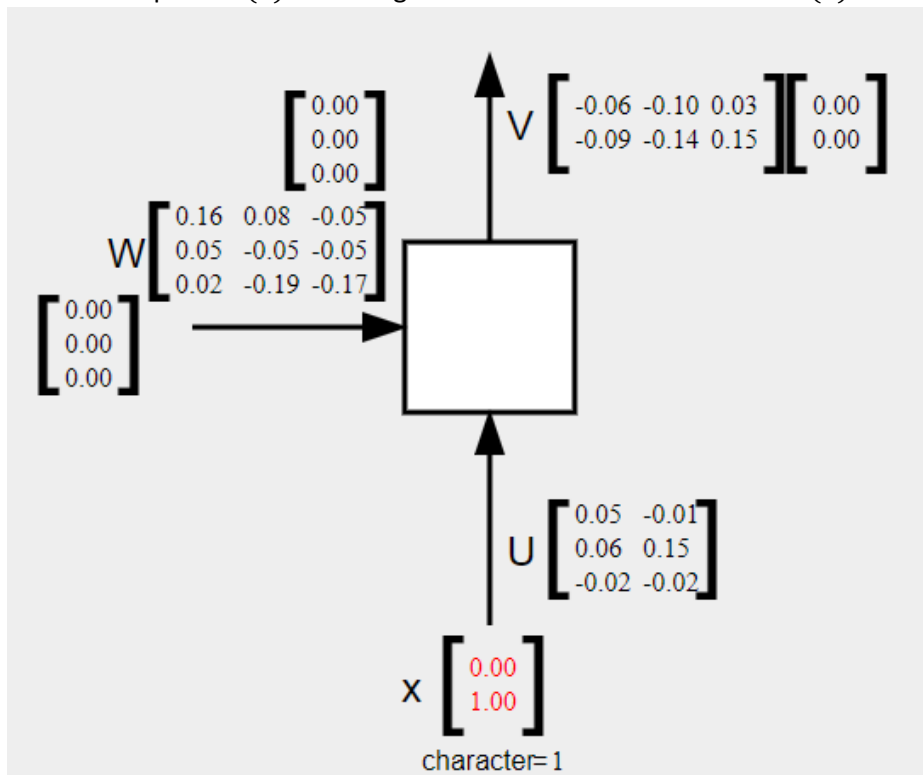
$\text{loss} = 0$



Line 34 initialize  $x(0)$  before feeding into RNN

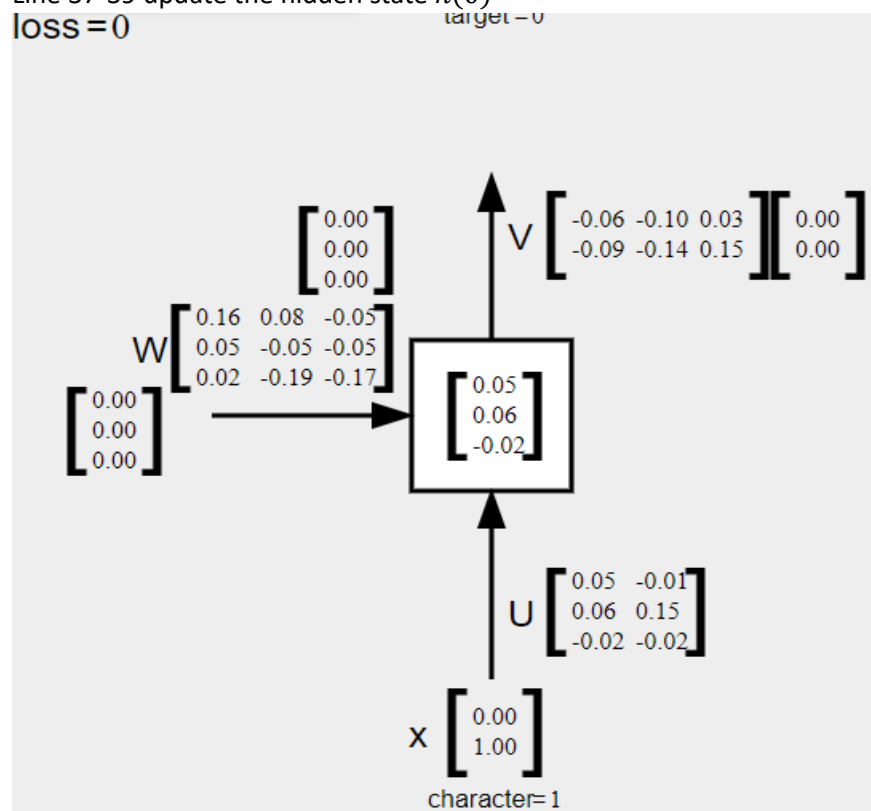


Line 35-36 update  $x(0)$  according to the character value and feed  $x(0)$  into RNN



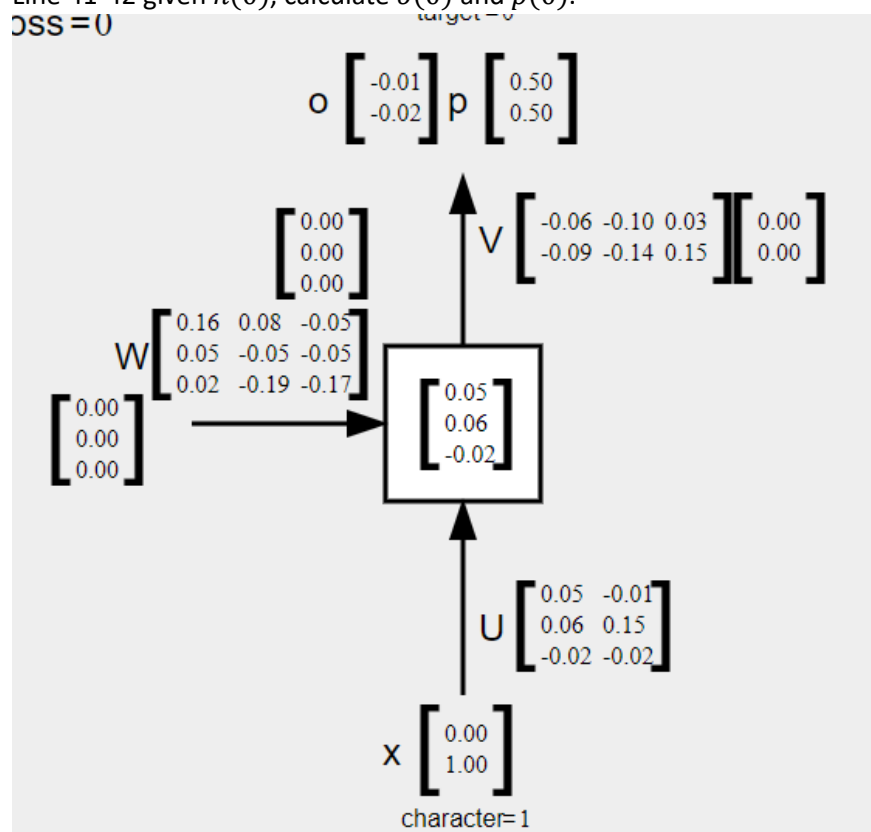
Line 37-39 update the hidden state  $h(0)$

loss = 0



Line 41-42 given  $h(0)$ , calculate  $o(0)$  and  $p(0)$ .

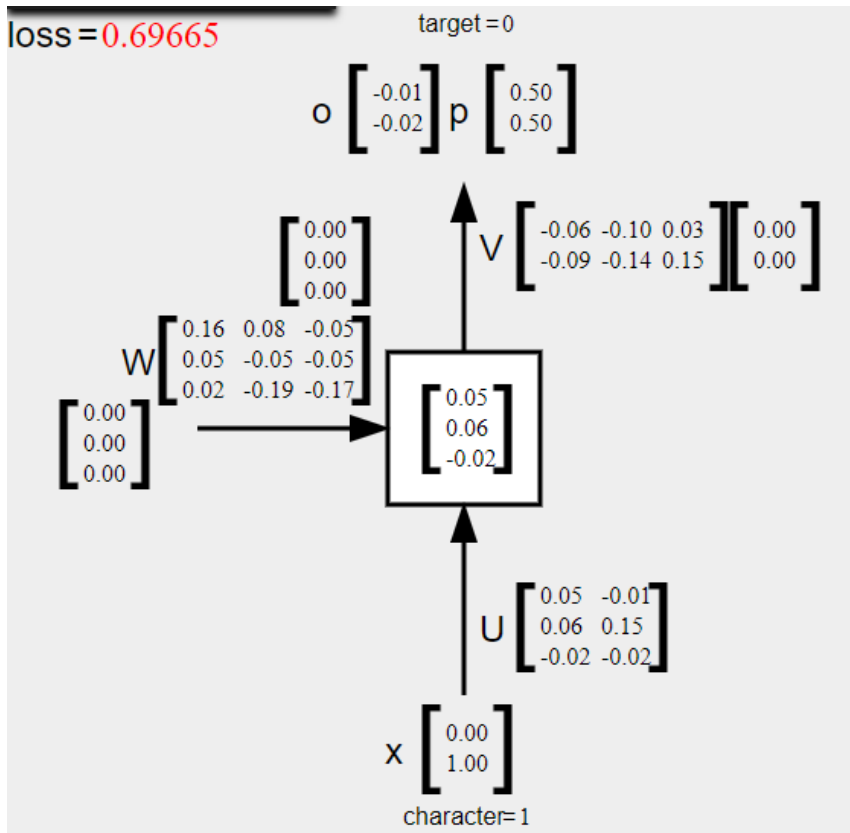
loss = 0



Line 45 calculate the loss function value

loss = 0.69665

target = 0



$$p_1 = \frac{e^{-0.01}}{e^{-0.01} + e^{-0.02}} = 0.49745027539104264$$

$$p_2 = \frac{e^{-0.02}}{e^{-0.01} + e^{-0.02}} = 0.4925005624493796$$

Our network is fed an input of 1,0,1,0 and is tasked with predicting the target output sequence 0,1,0,1.

Character "1" is represented by vector,  $character\ 1 = \begin{bmatrix} 0.00 \\ 1.00 \end{bmatrix}$ .

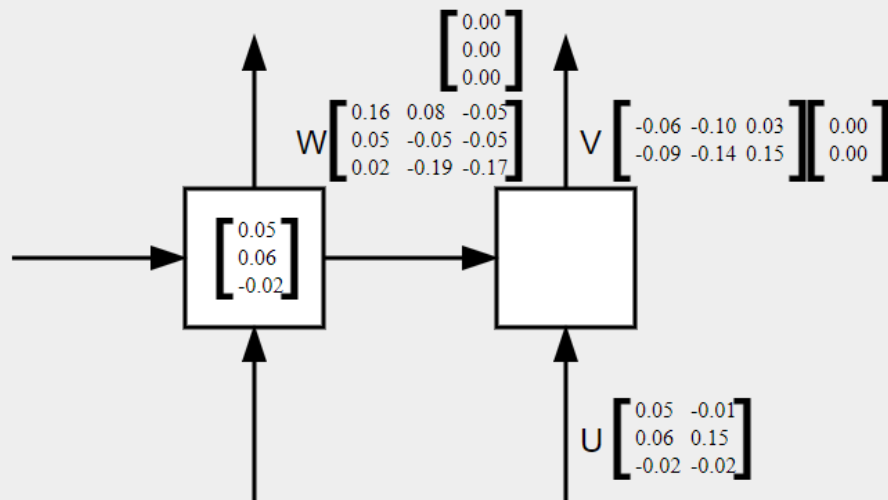
Character "0" is represented by vector,  $character\ 0 = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$ .

$$l(0) = -\log(p_1)$$

Loop 1: t = 1

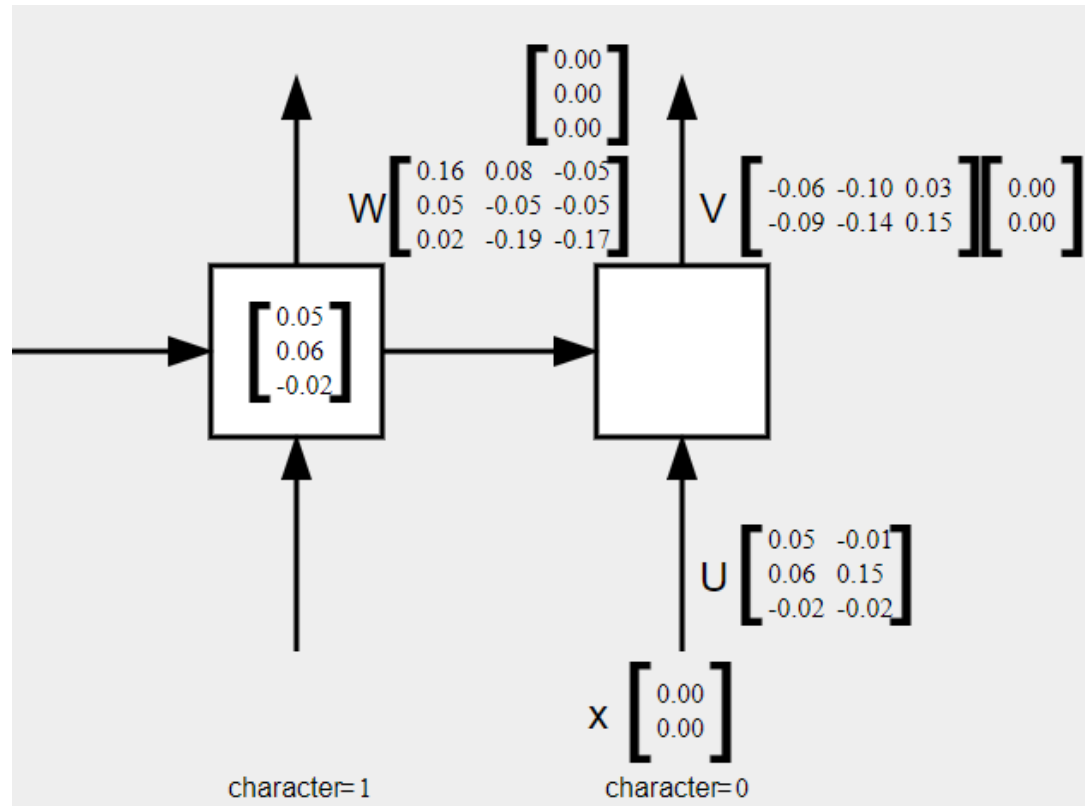
loss = 0.69665

target = 0



character=1

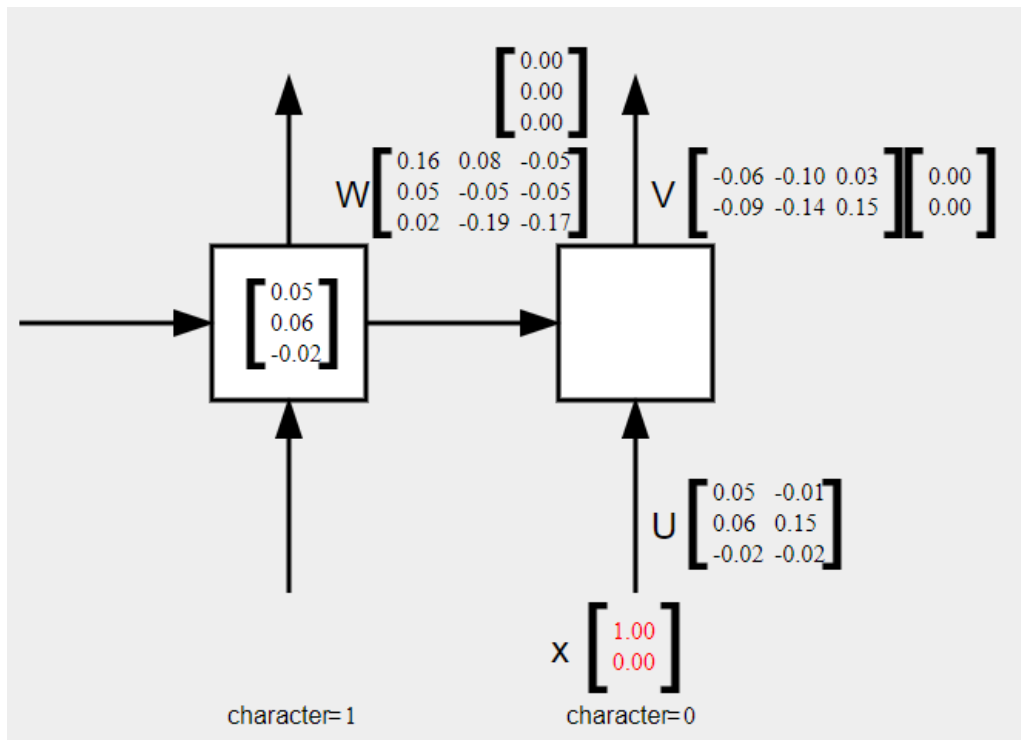
Line 34-35



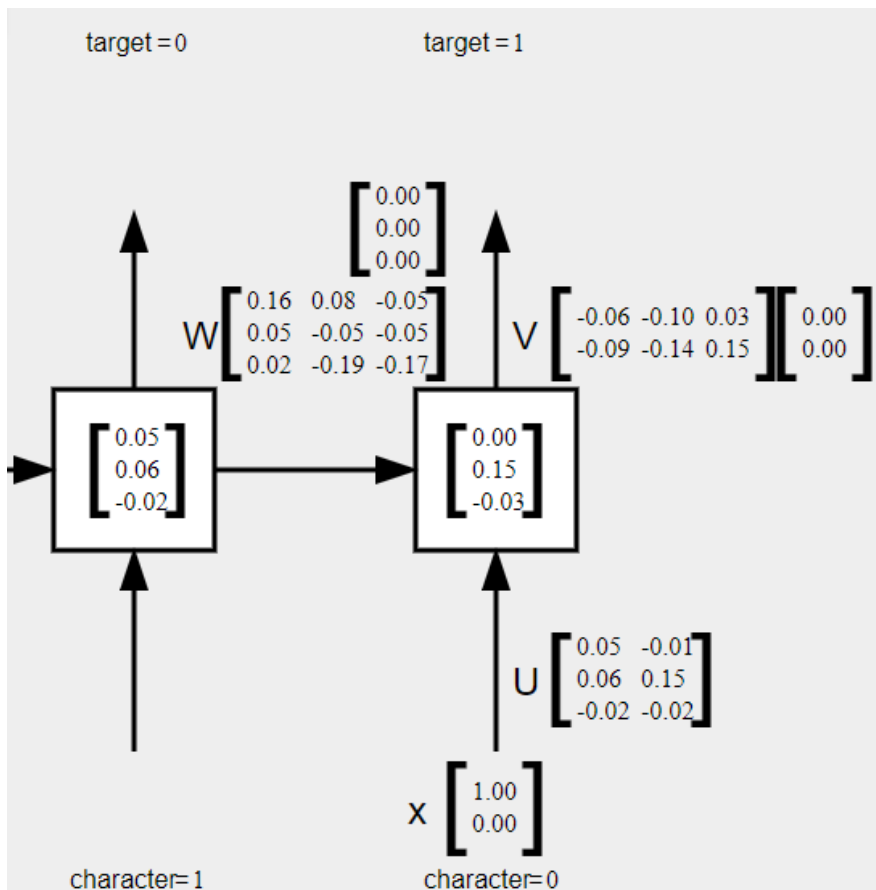
character=1

character=0

Line 36

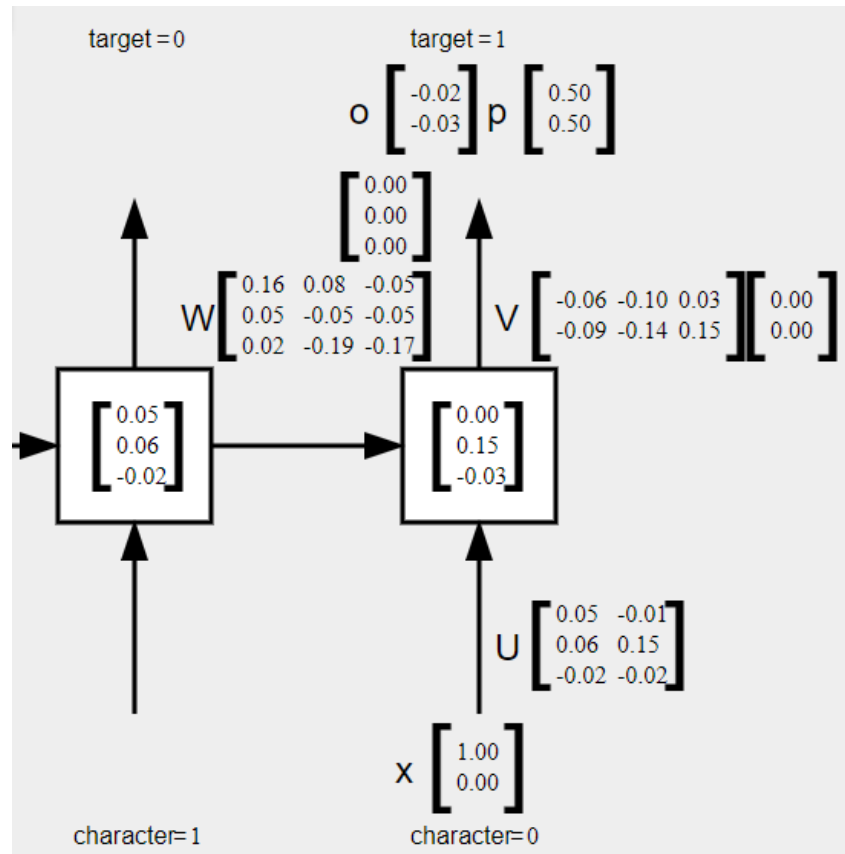


Line 37-39



Line 40-45

Loss = 1.38501



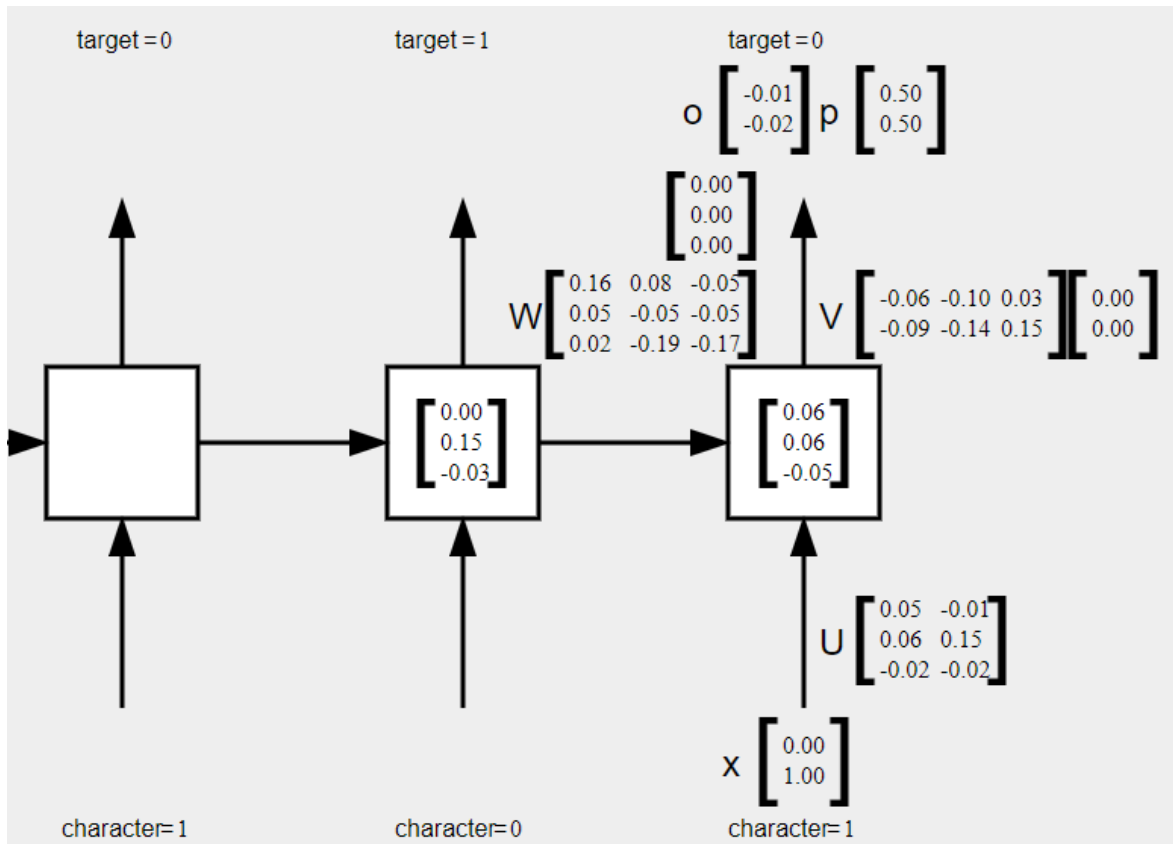
Take a minute to match up the weights in the diagram with the weights in the code. You can hover over them, and it will highlight the relationship for you. If you're really feeling up for it, you could work out the matrix multiplications by hand. The big takeaways here are:

- Computing hidden state is most of the work.
- We use the same weights for every input of our forward pass.
- `hidden_states[t-1]` changes as we go through each step.
- We output probabilities for the target character at each iteration (loop)
- Since our network is untrained the output probabilities are:
  - 50% 0
  - 50% 1

The rest iterations will take calculations as follows:

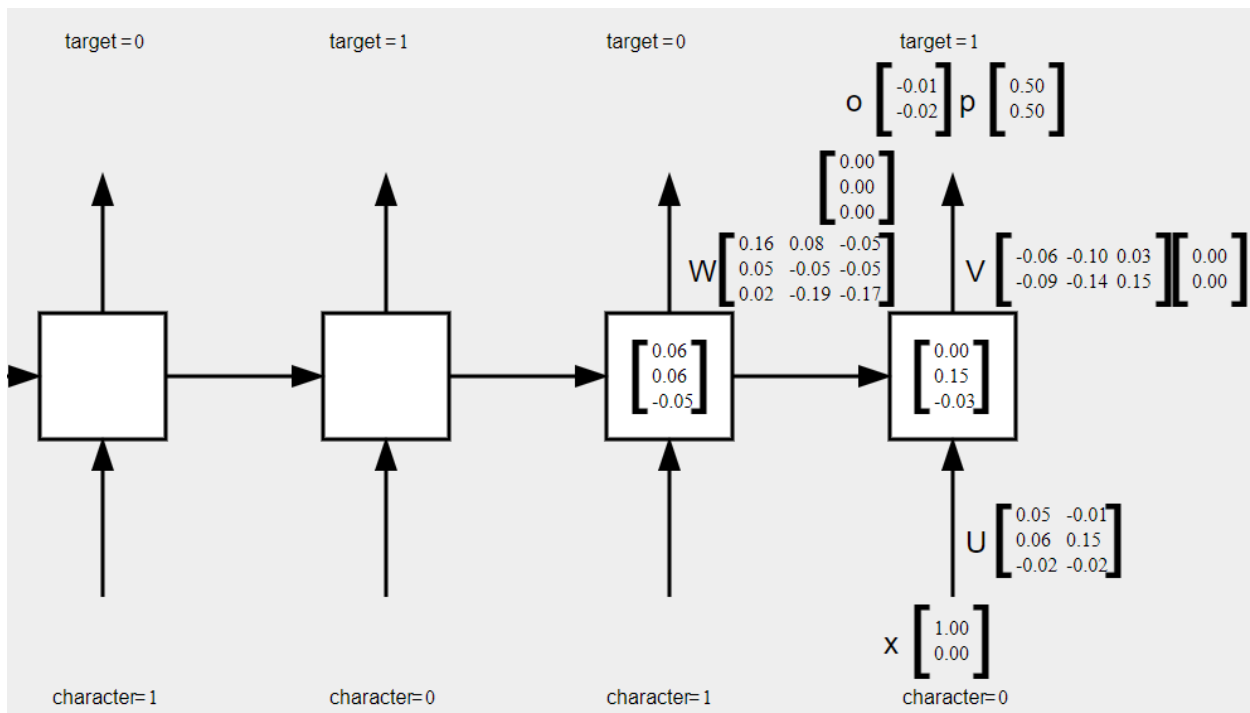
Loop 2: t = 2

Loss = 2.08315



Loop 3: t = 3

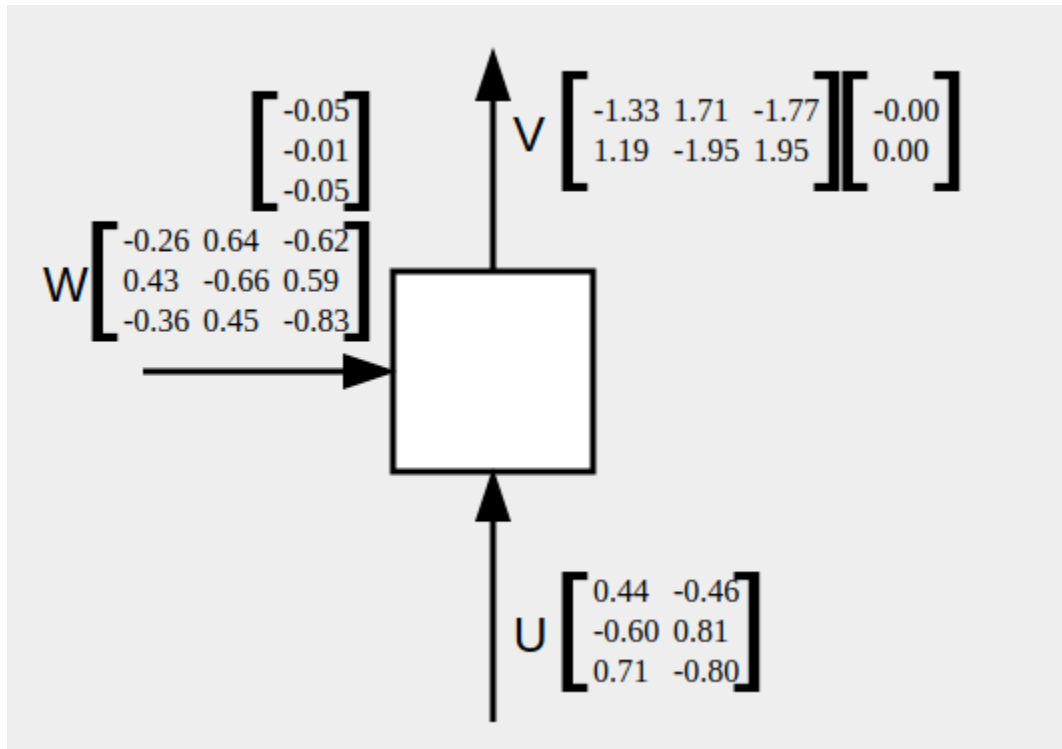
Loss = 2.77402





### Example 2: a trained RNN

So now that we've seen a network that doesn't work, let's look at one that does. After training our network with gradient descent (a visualization I'll save for another time) we're left with weights that look like:



So what happens when we run the network using these weights?

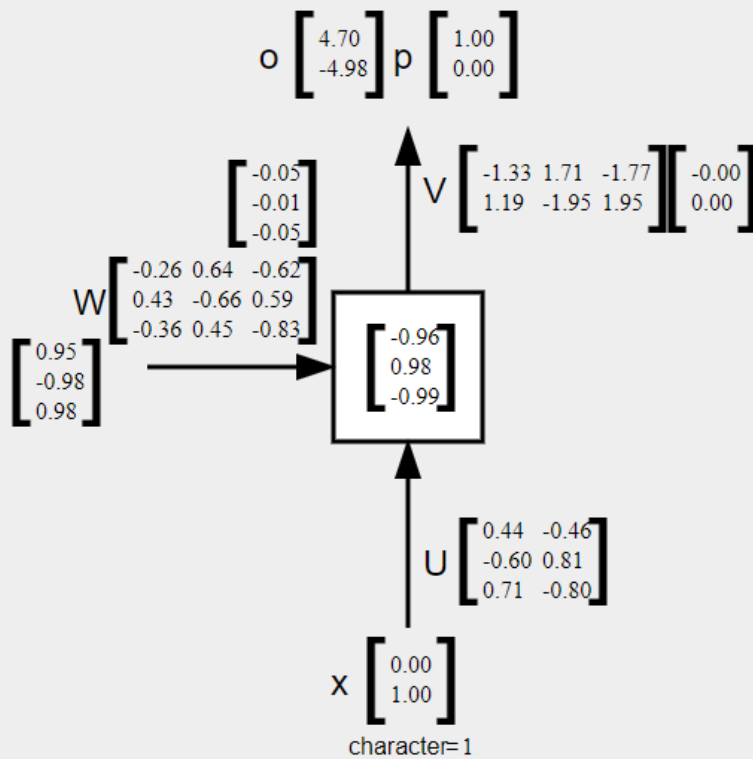
Character 1 is represented by vector,  $character\ 1 = \begin{bmatrix} 0.00 \\ 1.00 \end{bmatrix}$ .

Character 0 is represented by vector,  $character\ 0 = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$ .

Loop 0: t = 0

loss = 0.000062368382

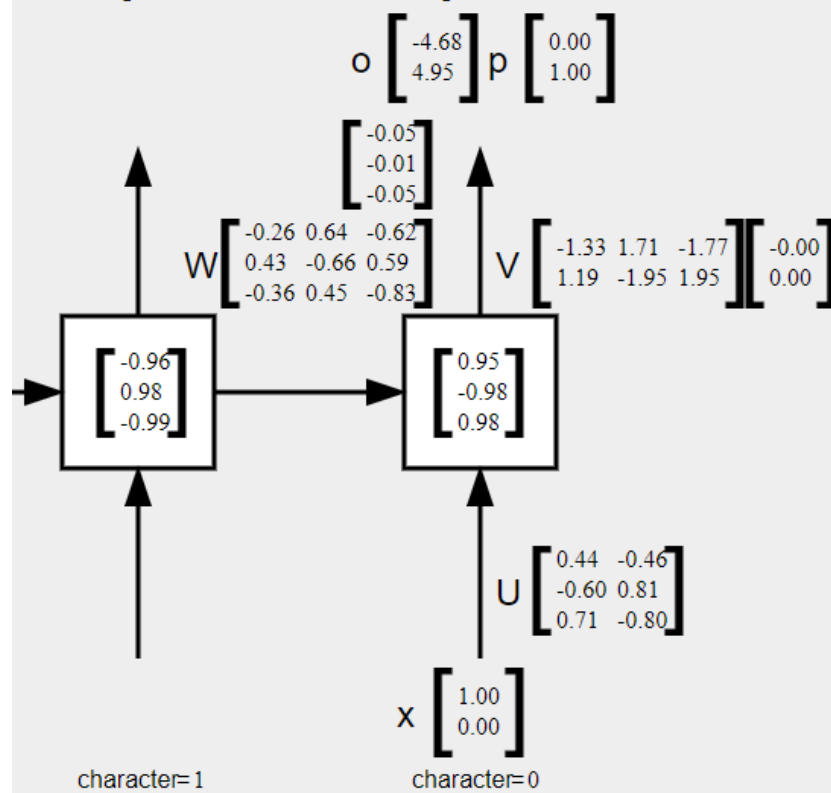
target = 0



Loop 1: t = 1

target = 0

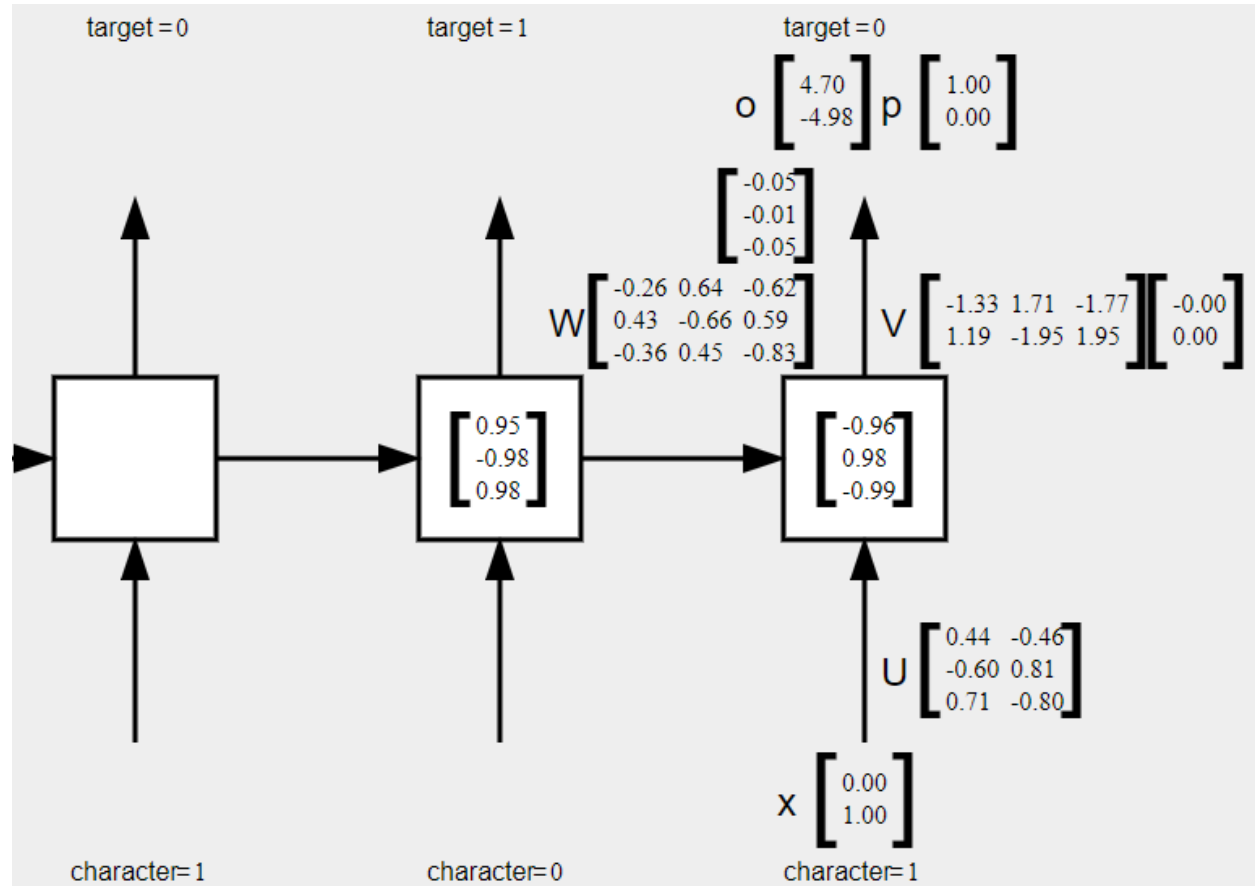
target = 1



(Loss = 0.0001279723)

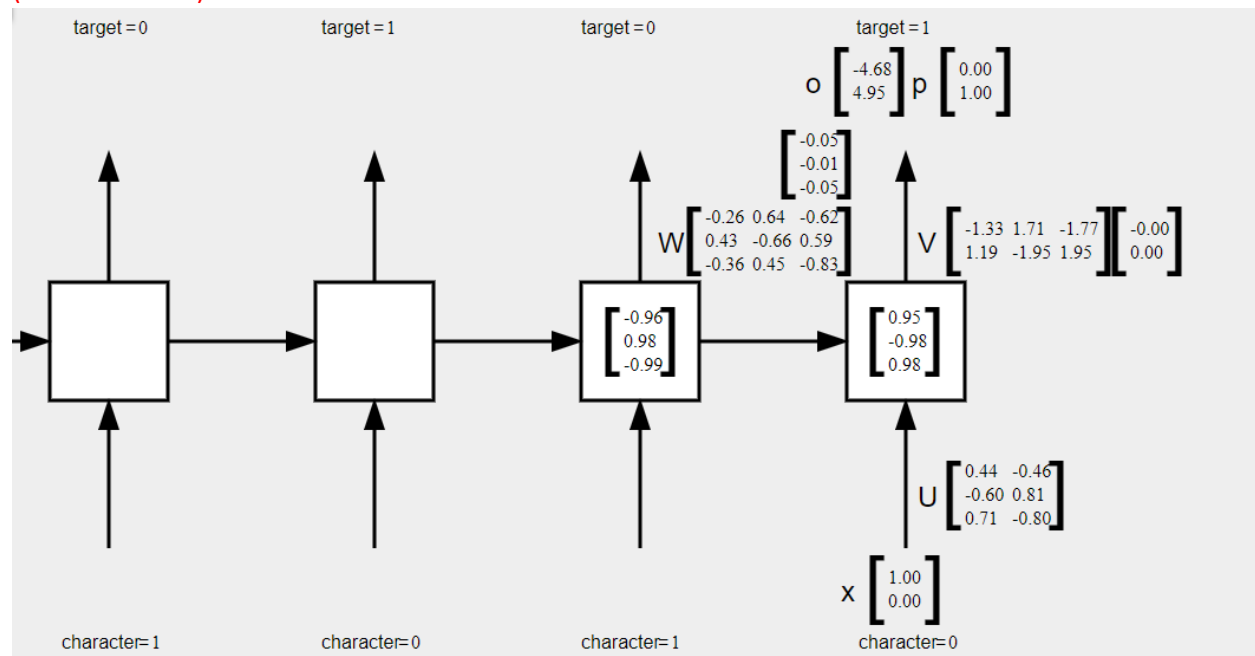
Loop 2: t = 2

(Loss = 0.0001903406)



Loop 3: t = 3

(Loss = 2.77402)



This time our network outputs probabilities for 0 and 1 with almost 100% confidence. If you pay close attention to hidden\_state you'll notice that it also alternates between  $[1,-1,1]$  and  $[-1,1,-1]$ . In fact, the more we start thinking about it, the more it's interesting that hidden\_state learns anything at all. Hidden state is meant to encode useful information about things we've seen in the past, but our problem doesn't depend on past information. Our network should really only care about whatever the current input (1 or 0) to our network is.

### Example 3: Making hidden state useful

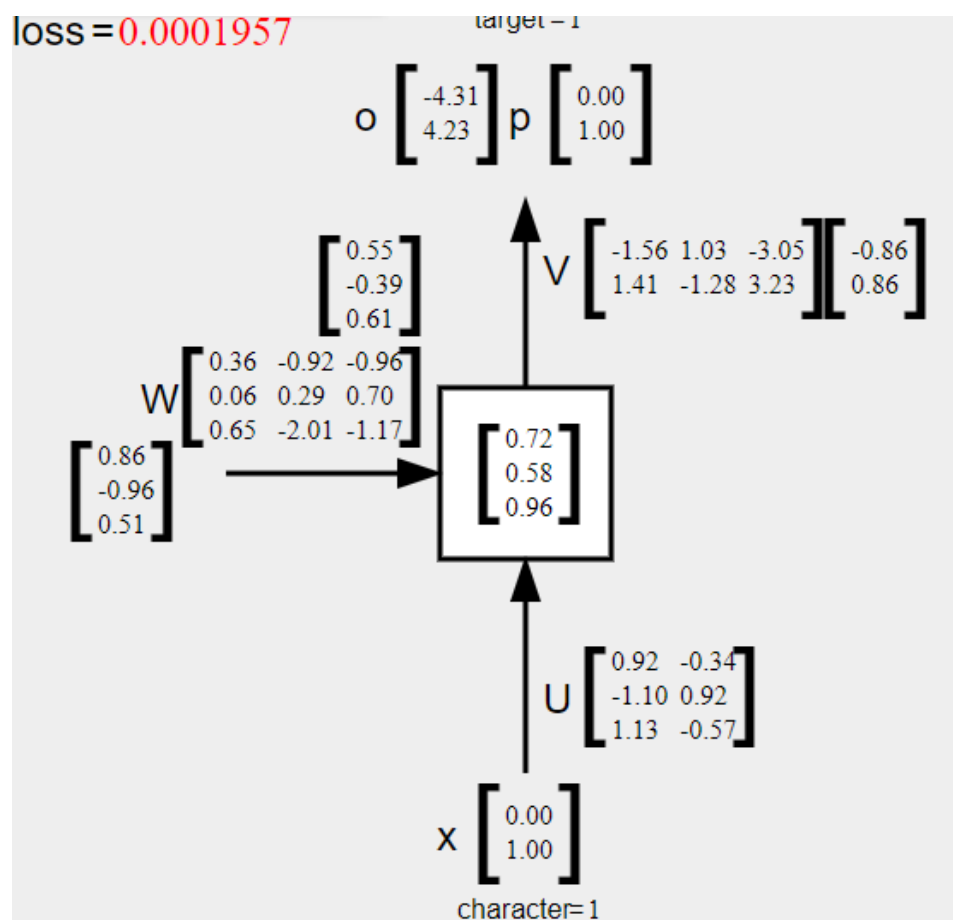
If we want the hidden state to be useful, we'll have to give it a problem where it's actually needed. We'll modify our original sequence slightly and have our network predict the next character in 1,1,0,1,1,0,1,0, ...

Now our network can no longer get away with simply predicting the opposite of the input. It will have to take special care to determine whether we're on the first or second 1 when predicting the output.

Below is an RNN trained to respond to input characters 1,1,0,1 with 1,0,1,1. Keep an eye on hidden\_state as the diagram updates.

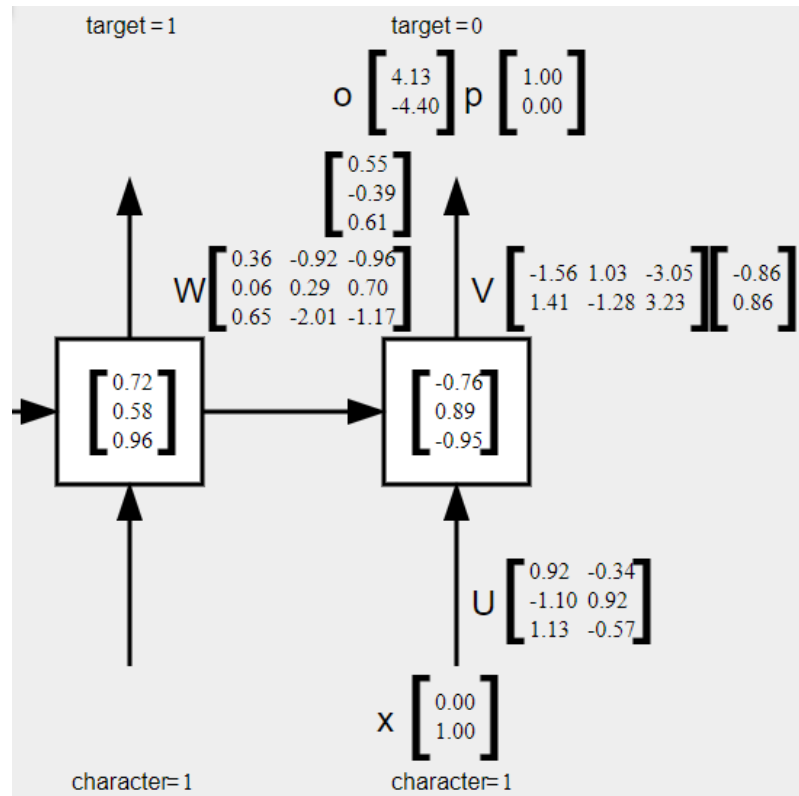
Loop 0:  $t = 0$

loss = 0.0001957



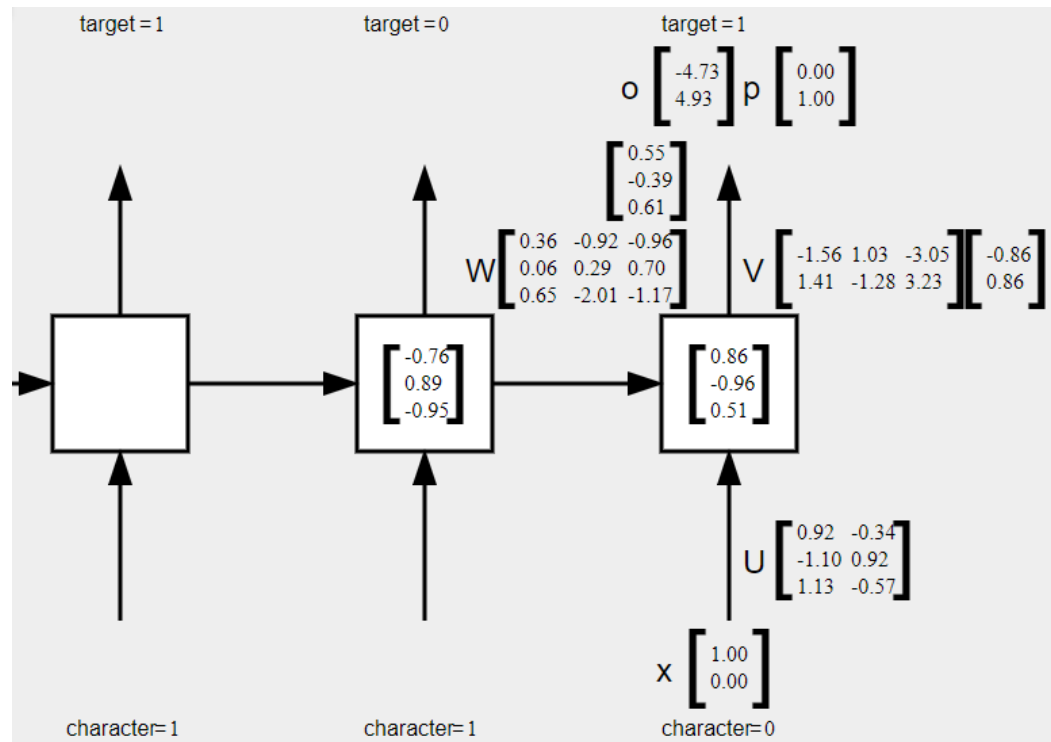
Loop 1: t = 1

Loss = 0.0003945



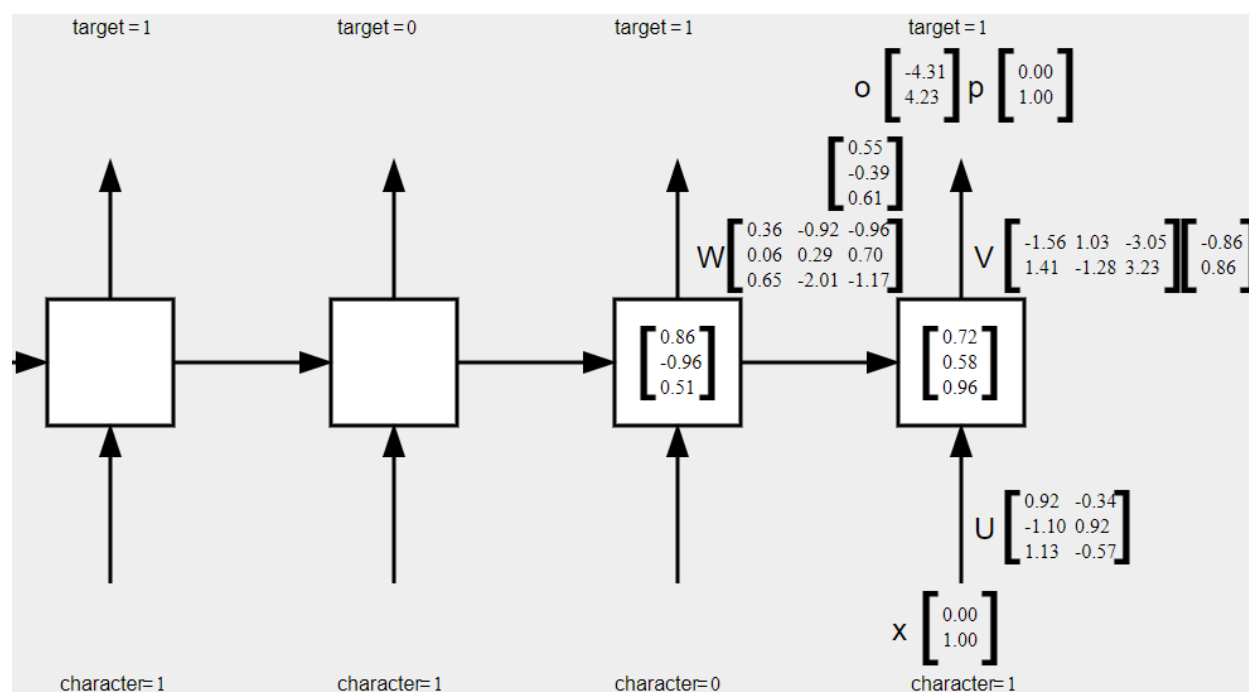
Loop 2: t = 2

Loss = 0.0004583



Loop 3: t = 3

Loss = 2.77402

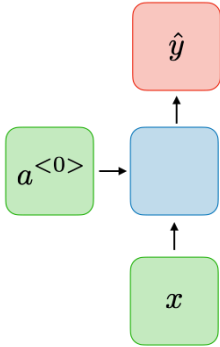
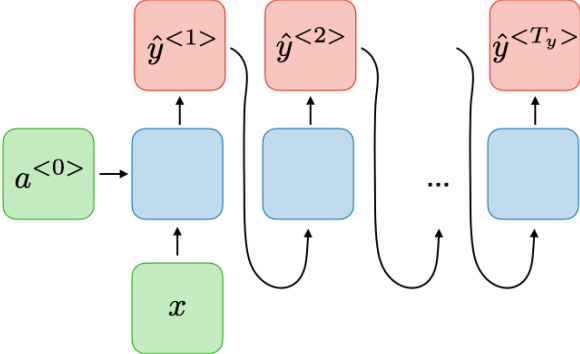
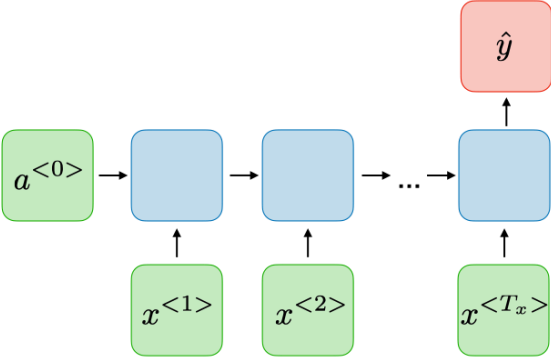


This time hidden state is actually encoding useful information we can observe directly. Whenever the network sees the first 1 in the sequence, hidden state is set to  $[0.72, 0.58, 0.96]$ . You can see this at the first and final steps of the animation. In contrast when the network sees the second 1, hidden state is set to  $[-0.76, 0.89, 0.95]$ . Varying hidden state like this allows our network to output the proper probabilities at each step despite our inputs (1) and parameters ( $U$ ,  $W$  and  $V$ ) being the same.

Indeed, much of the power of RNNs stems from hidden state and interesting ways of convincing our network to either remember or forget different things about the input sequences. Andrej's original blog post covers this in detail for a number of different domains.

## 1.6 Applications of RNNs

RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification

<p>Many-to-many</p> <p><math>T_x = T_y</math></p>		<p>Name entity recognition</p>
<p>Many-to-many</p> <p><math>T_x \neq T_y</math></p>		<p>Machine translation</p>

### 1.7 Variants of RNN

The table below sums up the other commonly used RNN architectures:

Bidirectional (BRNN)	Deep (DRNN)



## Appendix

### A.1 Loss Functions Used in GenAI

#### Mean Squared Error (MSE) Loss

**Use Case:** Primarily used for regression tasks where the output is continuous.

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

$N$ : Number of data points.

$y_i$ : Actual value.

$\hat{y}_i$ : Predicted value.

MSE calculates the average squared difference between actual and predicted values. It heavily penalizes larger errors.

#### Binary Cross-Entropy Loss (Log Loss)

**Use Case:** Used in binary classification tasks.

$$\text{Binary Cross Entropy} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(f(\hat{y}_i)) + (1 - y_i) \log(1 - f(\hat{y}_i))]$$

$y_i$ : Actual label.

$f(\hat{y}_i)$ : Predicted probability for the label.

Measures the dissimilarity between the predicted probabilities and the actual binary class labels. The loss is high when the predicted probability diverges from the actual label.

#### Categorical Cross-Entropy Loss

**Use Case:** Used for multi-class classification tasks where each sample belongs to one of several classes.

$$\text{Categorical Cross Entropy} = -\sum_{i=1}^N \sum_{j=1}^C y_{i,j} \log(f(\hat{y}_{i,j}))$$

$C$ : Number of Classes

$y_i$ : Actual class label (one-hot encoded)

$f(\hat{y}_{i,j})$ : Predicted probability for the class  $j$ .

**Explanation:** Penalizes incorrect predictions by measuring the distance between the true class distribution and the predicted probability distribution.

#### Likelihood

When predicting the target variable  $y(t)$  at time  $t$ , the predictive density obtained using the RNN is  $p[y(t)|x(t), \theta]$ , which is, the probability that the RNN assigns to the actual  $y(t)$ . The RNN models uses

data input  $x(t)$  and parameter value  $\theta = \{U, V, W, b, c\}$  to get the probability. The prediction density thus can be expressed using conditional probability.

The likelihood  $l(y|x, \theta)$  is defined as the product of these predictive probabilities over time. In particular,

$$l(y|x, \theta) = \prod_{t=1}^T p[y(t)|x(t), \theta] = \prod_{t=1}^T p[y(t)|x(t), U, V, W, b, c]$$

the loss  $L(y|x, \theta)$  is the negative log-likelihood of the true target  $y(t)$  given the input so far.

$$\begin{aligned} L(y|x, \theta) &= -\log\{l(y|x, \theta)\} \\ &= -\sum_{t=1}^T \log\{p[y(t)|x(t), \theta]\} \\ &= -\sum_{t=1}^T \log\{p[y(t)|x(t), U, V, W, b, c]\} \end{aligned}$$

RNN outputs  $o(t)$  are used as the argument into the softmax function to obtain the vector of probabilities over the output.  $p(y(t)|x(t), \theta)$  is the probability in the vector that corresponds to the actual  $y(t)$ . The loss  $L(y|x, \theta)$  is the negative log-likelihood of the true target  $y(t)$  for  $t = 1 \dots, T$  given the input  $x$  and parameter estimates of  $U, V, W, b, c$ .

## A.2 Softmax

The softmax function takes an  $N$ -dimensional vector of arbitrary real values and produces another  $N$ -dimensional vector with real values in the range  $(0, 1)$  that adds up to 1.0. It maps  $S(o): R^N \rightarrow R^N$ .

$$S(o): \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_N \end{bmatrix} \rightarrow \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix}$$

Here each  $o(t)$  over time is composed of the  $N$ -dimensional vector. The time stamp is omitted to avoid clutter in notation. The actual per-element formula is:

$$p_j = \frac{e^{o_j}}{\sum_{k=1}^N e^{o_k}}, \forall j \in 1, \dots, N$$

It's easy to see that is always positive (because of the exponents); moreover, since the numerator appears in the denominator summed up with some other positive numbers. Therefore, it's in the range  $(0, 1)$ .

For example, the 3-element vector  $[1.0, 2.0, 3.0]$  gets transformed into  $[0.09, 0.24, 0.67]$ . The order of elements by relative size is preserved, and they add up to 1.0. Let's tweak this vector slightly into:  $[1.0, 2.0, 5.0]$ . We get the output  $[0.02, 0.05, 0.93]$ , which still preserves these properties. Note that as the last element is farther away from the first two, its softmax value is dominating the overall slice of size 1.0 in the output. Intuitively, the softmax function is a "soft" version of the maximum function. Instead of just selecting one maximal element, softmax breaks the vector up into parts of a whole (1.0) with the

maximal input element getting a proportionally larger chunk, but the other elements getting some of it as well.

The properties of softmax (all output values in the range (0, 1) and sum up to 1.0) make it suitable for a probabilistic interpretation that's very useful in machine learning. In particular, in multiclass classification tasks, we often want to assign probabilities that our input belongs to one of a set of output classes.

If we have  $N$  output classes, we're looking for an  $N$ -vector of probabilities that sum up to 1; sounds familiar?

We can interpret softmax as follows:

$$S_j = P(y = j | o)$$

Where  $y$  is the output class numbered  $1, \dots, N$ .  $o$  is any  $N$ -vector. The most basic example is multiclass logistic regression, where an input vector  $x$  is multiplied by a weight matrix  $\theta$ , and the result of this dot product is fed into a softmax function to produce probabilities.

### Note: Review Logistic Regression

In particular, let  $\theta = \{\beta_0, \beta_1\}$  we write the equation for logistic regression as follows:

$$y = \frac{\exp(\beta_0 + \beta_1 x)}{1 + \exp(\beta_0 + \beta_1 x)}$$

In the above equation,  $\beta_0$  and  $\beta_1$  are the two coefficients of the input  $x$ . We estimate these two coefficients using Maximum Likelihood Estimation.  $y$  takes the form of sigmoid function, with outputs ranging from 0 to 1, and often interpreted as probabilities.

### A.3 Derivatives of Softmax

Before diving into computing the derivative of softmax, let's start with some preliminaries from vector calculus.

Softmax is fundamentally a vector function. It takes a vector as input and produces a vector as output; in other words, it has multiple inputs and multiple outputs. Therefore, we cannot just ask for "the derivative of softmax"; We should instead specify:

1. Which component (output element) of softmax we're seeking to find the derivative of.
2. Since softmax has multiple inputs, with respect to which input element the partial derivative is computed.

If this sounds complicated, don't worry. This is exactly why the notation of vector calculus was developed. What we're looking for is the partial derivatives:

$$\frac{\partial S_i}{\partial o_j}$$

This is the partial derivative of the  $i$ -th output w.r.t. the  $j$ -th input. A shorter way to write it that we'll be using going forward is:  $\nabla_j S_i$ .

Since softmax is a  $R^N \rightarrow R^N$  function, the most general derivative we compute for it is the Jacobian matrix:

$$\nabla S = \begin{bmatrix} \nabla_1 S_1 & \cdots & : \nabla_N S_1 \\ \vdots & \ddots & \vdots \\ \nabla_1 S_N & \cdots & : \nabla_N S_N \end{bmatrix}$$

In machine learning literature, the term "gradient" is commonly used to stand in for the derivative. Strictly speaking, gradients are only defined for scalar functions (such as loss functions); for vector functions like softmax it's imprecise to talk about a "gradient"; the Jacobian is the fully general derivative of a vector function, but in most places, I'll just be saying "derivative".

## A.4 Gradient Descent

### A.4.1 Gradient Descent Introduction

Gradient Descent is an optimization algorithm used to train machine learning algorithms; most notably artificial neural networks used in deep learning. Two Important variants of Gradient Descent are Batch Gradient Descent and Stochastic Gradient Descent(SGD).

The job of the algorithm is to find a set of internal model parameters that perform well against some performance measure such as logarithmic loss or mean squared error.

Optimization is a type of searching process and you can think of this search as learning. The optimization algorithm is called "gradient descent", where "gradient" refers to the calculation of an error gradient or slope of error and "descent" refers to the moving down along that slope towards some minimum level of error.

The algorithm is iterative. This means that the search process occurs over multiple discrete steps, each step hopefully slightly improving the model parameters.

Each step involves using the model with the current set of internal parameters to make predictions on some samples, comparing the predictions to the real expected outcomes, calculating the error, and using the error to update the internal model parameters.

This update procedure is different for different algorithms, but in the case of artificial neural networks, the backpropagation update algorithm is used.

Gradient is the slope of a curve at a given point in a specified direction. Each direction is represented by a parameter, In the case of a univariate function, say,  $f(\theta)$ , Gradient is simply the first derivative at a selected point. In the case of a multivariate function, say,  $f(\theta_1, \theta_2)$ , Gradient is a vector of derivatives in each direction.

We have to solve the following optimization problem:

$$\theta^* \arg \min L(\theta)$$

$L$ : loss function (if max, called objective function)

$\theta$ : parameters. In RNN's case,  $\theta = \{U, V, W, b, c\}$ .

Suppose that  $\theta$  has two variables  $\{\theta_1, \theta_2\}$ , learning rate is  $\eta$ .

Randomly start at  $\theta^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^0)/\partial \theta_1 \\ \partial L(\theta_2^0)/\partial \theta_2 \end{bmatrix}$$

$$\begin{bmatrix} \theta_1^2 \\ \theta_2^2 \end{bmatrix} = \begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^1)/\partial \theta_1 \\ \partial L(\theta_2^1)/\partial \theta_2 \end{bmatrix}$$

i.e. Gradient (at each step) is defined as, taking partial derivative of the loss function with respect to  $\{\theta_1, \theta_2\}$  and stacking them into a vector:

$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta_1)/\partial \theta_1 \\ \partial L(\theta_2)/\partial \theta_2 \end{bmatrix}$$

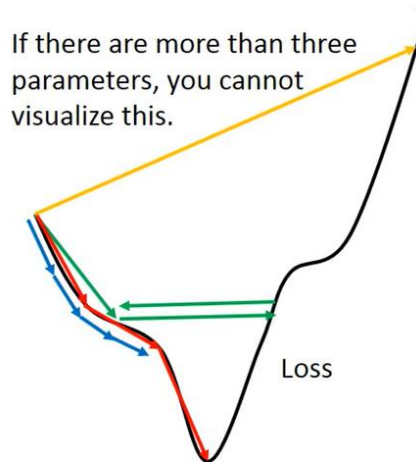
In vector representation:

$$\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$$

$$\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

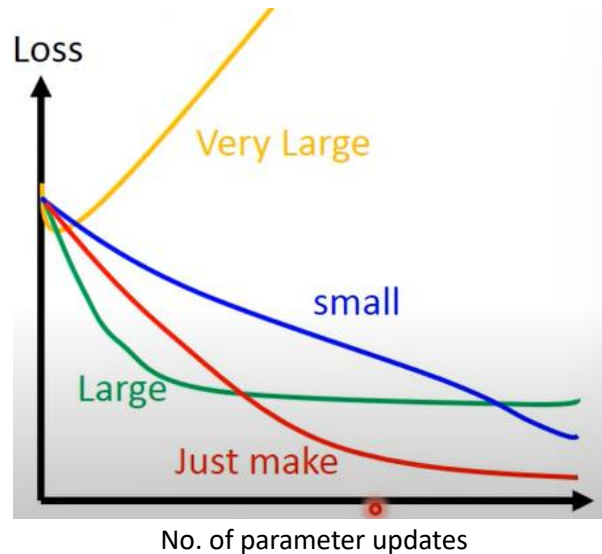
#### A.4.2 Gradient Descent Tips

1. Tuning the learning rate,  $\eta$  – select the learning rate (step length) carefully.
  - If too small (blue), the step length is too small, the moving is too slow, which leads to iteration number exceeds max limit.
  - If too large, the step length is too large to move down, but may jump to the other side of the loss function.
  - If very large, the step length might be too large and moves up instead.



2. Visualize Loss function changes versus every time the parameter updates. Make sure the learning rate is “just make” to the descendance of loss function.
  - If too small, the descent rate is too slow.
  - If too large, the loss function decreases to a level and then remains flat.
  - If Very large, the loss function flies out.

**Tips:** draw the first few steps when the gradient descent starts – check if the loss function descent with respect to parameter changes in a “just make” rate.



3. **Vanishing/exploding gradient:** The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

#### A.4.3 Adaptive Learning Rate

- General rule: learning rate reduces as parameter updates. At the beginning, we are far from the destination, so we use larger learning rate (step length). After several epochs, we are close to the destination, so we reduce the learning rate.  
e.g.,  $1/k$  decay,  $\eta^k = \eta / \sqrt{(k + 1)}$ , the more you update parameters, the smaller the  $\eta$ , where  $k$  is the number of times you update the parameters.
- Learning rate cannot be “one size fits all.”  
Different parameters may have to use different learning rates.

#### A.4.4 Ada-Grad

Consider each parameter  $\theta$  separately,

- Divide the learning rate of each parameter by the root mean square of its previous derivatives.  
Vanilla Gradient descent

$$\theta^{k+1} \leftarrow \theta^k - \eta^k \nabla L(\theta^k)$$

Ada-Grad

$$\theta^{k+1} \leftarrow \theta^k - \frac{\eta^k}{\sigma^k} \nabla L(\theta^k)$$

$\nabla L(\theta^k)$  is the value of the partial derivatives.  $\nabla L(\theta^k) = \frac{\partial L(\theta^k)}{\partial \theta}$ .  $\sigma^k$  is the root mean square of all the previous derivatives of a parameter  $\theta$ . Calculate both for each parameter. For example,

$$\theta^1 \leftarrow \theta^0 - \frac{\eta^0}{\sigma^0} \nabla L(\theta^0) \text{ where } \sigma^0 = \sqrt{\nabla L(\theta^0)^2}.$$

$$\theta^2 \leftarrow \theta^1 - \frac{\eta^1}{\sigma^1} \nabla L(\theta^1) \text{ where } \sigma^1 = \sqrt{\frac{1}{2} [\nabla L(\theta^0)^2 + \nabla L(\theta^1)^2]}$$

### A.5 Newton Method

Let's first consider a function  $f$  of one variable. Consulting Taylor, we know that the second order approximation of  $L(\theta)$  about a point  $\theta + \epsilon$  is given by.

$$L(\theta + \epsilon) = L(\theta) + L'(\theta)\epsilon + \frac{1}{2}L''(\theta)\epsilon^2$$

Our Taylor approximation is minimized when:

$$\frac{\partial}{\partial \epsilon} \left[ L(\theta) + L'(\theta)\epsilon + \frac{1}{2}L''(\theta)\epsilon^2 \right] = L'(\theta) + L''(\theta)\epsilon = 0$$

which corresponds to a step size of

$$\epsilon = -\frac{L'(\theta)}{L''(\theta)}.$$

Given the fact that  $\theta^{k+1} = \theta^k + \epsilon$ , We can thus adopt a new iteration scheme:

$$\theta^{k+1} = \theta^k - \frac{L'(\theta^k)}{L''(\theta^k)} = \theta^k - \nabla L(\theta^k)/L''(\theta^k)$$

**Key Difference from Gradient Descent:** Newton Method takes into account of the second-order behavior of the objective function.

### A.6 Stochastic Gradient Descent (SGD) Algorithm

There are a few downsides of the gradient descent algorithm. We need to take a closer look at the amount of computation we make for each iteration of the algorithm.

Say we collect 10,000 data points and our RNN has  $K$  parameters. The sum of squared residuals consists of as many terms as there are data points, so 10,000 terms in our case. We need to compute the derivative of this function with respect to each parameter, so in effect we will be doing

$$10,000 \times K$$

computations per iteration. It is common to take 1000 iterations. If  $K = 100$ , in effect we have

$$1,000,000 \text{ (one million)} \times 1000 = 1,000,000,000 \text{ (one billion)}$$

computations to complete the algorithm. That is pretty much an overhead and hence gradient descent is slow on huge data. Stochastic gradient descent thus comes to our rescue.

Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models. It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.

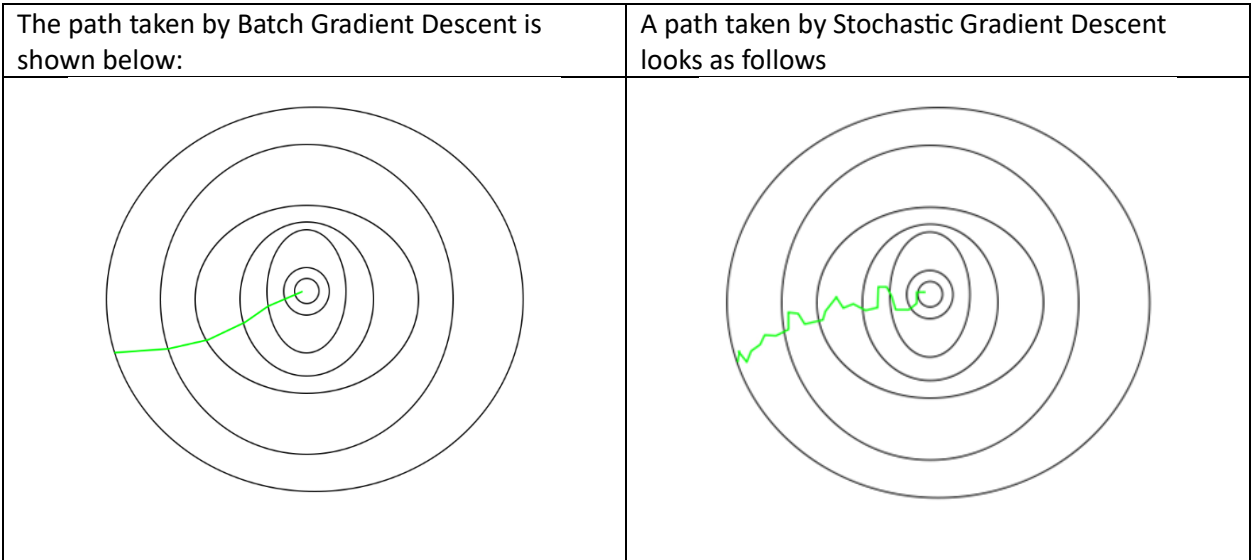
In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent.

The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

**Stochastic Gradient Descent Algorithm**

- Initialization: Randomly initialize the parameters of the model.
- Set Parameters: Determine the number of iterations and the learning rate ( $\alpha$ ) for updating the parameters.
- Stochastic Gradient Descent Loop: Repeat the following steps until the model converges or reaches the maximum number of iterations:
  - Shuffle the training dataset to introduce randomness.
  - Iterate over each training example (or a small batch) in the shuffled order.
  - Compute the gradient of the loss function with respect to the model parameters using the current training example (or batch).
  - Update the model parameters by taking a step in the direction of the negative gradient, scaled by the learning rate.
  - Evaluate the convergence criteria, such as the difference in the loss function between iterations of the gradient.
- Return Optimized Parameters: Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with a significantly shorter training time.



One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minimum, because of the randomness in its descent. Even though it requires a higher number of iterations to reach the minimum than typical Gradient Descent, it is still



computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

#### **Reference**

<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>

<https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85>

<https://karpathy.github.io/2015/05/21/rnn-effectiveness/>

<https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>

<https://jramapuram.github.io/ramblings/rnn-backprop/>

<https://willwolf.io/2016/10/18/recurrent-neural-network-gradients-and-lessons-learned-therein/>

<https://joshvarty.github.io/VisualizingRNNs/>