Advanced Methods in Natural Language Processing

Session 2: Neural Networks, Backpropagation & Recurrent Neural Networks

Arnault Gombert

April 2025

Barcelona School of Economics

Introduction

Introduction to Deep Learning

Today's Focus: Understanding the Core of Neural Networks

- **Neural Networks Basics**: Exploring the structure and function of simple neural networks.
- **Gradient Descent and Backpropagation**: Unveiling how neural networks learn and optimize.

Advancing to Complex Models

- **Recurrent Neural Networks (RNNs)**: Delving into the handling of sequential data.
- **Long Short-Term Memory (LSTM) Networks**: Understanding how LSTMs tackle the limitations of traditional RNNs.
- **Language Models**: Introducing and exploring basic language models.

Neural Networks

Introduction to Neural Networks

- **Diverse Network Types**: Neural Networks encompass various architectures, each suited to specific tasks.
  - *Multi-layer Perceptrons (MLPs)*: Basic form of NNs.
  - *Recurrent Neural Networks (RNNs)*: Ideal for sequential data like text (Rumelhart et al., 1986).
  - *Convolutional Neural Networks (CNNs)*: Specialized in processing structured grid data like images (LeCun et al., 1989).
  - *Transformers*: NLP Revolution with attention mechanisms (Vaswani et al., 2017).
- **Understanding the Basics**: Before delving into complex models, it's crucial to grasp the foundational principles.
  - Avoiding the "black box" approach
  - Blind feature engineering without algorithmic understanding.
- **Vanilla Neural Networks**: Also known as single-layer backpropagation networks, these form the cornerstone of more complex architectures.

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.

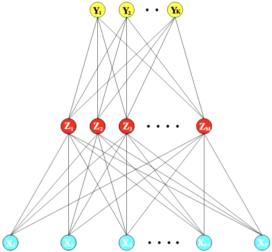


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units** Z_m:

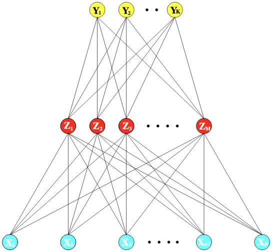


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units** Z_m:
  - Formed from a linear combination of inputs X_p.

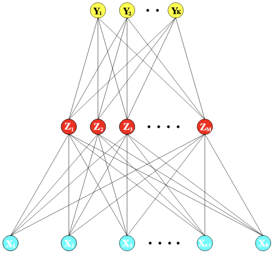


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units** Z_m:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.

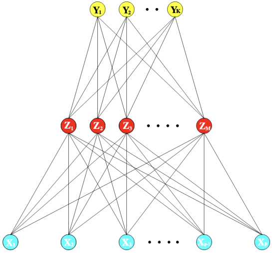


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units** Z_m:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.
- **Linear Combination for** T_k:

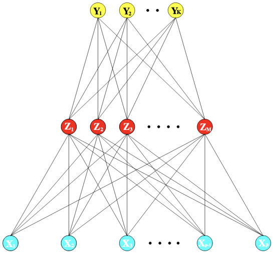


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.

- **Hidden Units** Z_m:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.

- **Linear Combination for** T_k:
  - $T_k = \beta_{0k} + \beta_k^T Z \forall k \in [1, K]$.

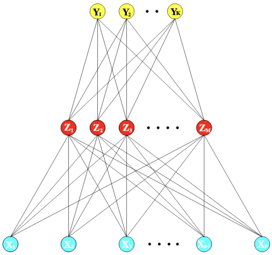


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.

- **Hidden Units** Z_m:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.

- **Linear Combination for** T_k:
  - $T_k = \beta_{0k} + \beta_k^T Z \forall k \in [1, K]$.

- **Output Function** Y_k:

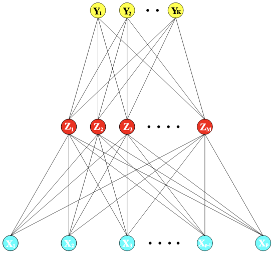


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units Z_m**:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.
- **Linear Combination for T_k**:
  - $T_k = \beta_{0k} + \beta_k^T Z \forall k \in [1, K]$.
- **Output Function Y_k**:
  - Softmax function applied to T_k.

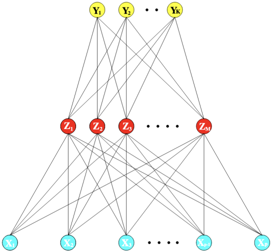


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks

- **K-Class Classification**: With K targets $Y_k, k \in [1, K]$.
- **Hidden Units Z_m**:
  - Formed from a linear combination of inputs X_p.
  - $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \forall m \in [1, M]$.
- **Linear Combination for T_k**:
  - $T_k = \beta_{0k} + \beta_k^T Z \forall k \in [1, K]$.
- **Output Function Y_k**:
  - Softmax function applied to T_k.
  - $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} \forall k \in [1, K]$.

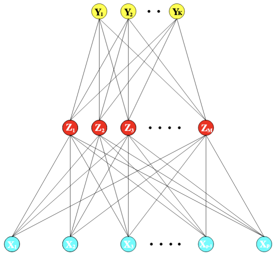


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Vanilla Neural Networks: The Role of Non-Linearity

- **Identity Function** $\sigma(v) = v$: Reduces to a linear model; typically used in output layers for regression.
- **Rectified Linear Unit (ReLU)** $\sigma(v) = \max(0, v)$: Popular for deep networks.
- **Sigmoid Function** $\sigma(v) = \frac{1}{1+e^{-v}}$: Commonly used, depicted on the right.
- **Hyperbolic Tangent** $\sigma(v) = \tanh(v)$: Similar to sigmoid but ranges from -1 to 1.
- **Others**: Various options available in deep learning libraries like Keras.

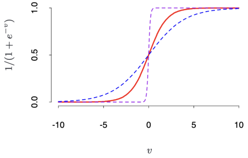


FIGURE 11.3. *Plot of the sigmoid function $\sigma(v) = 1/(1+\exp(-v))$ (red curve), commonly used in the hidden layer of a neural network. Included are $\sigma(sv)$ for $s = \frac{1}{2}$ (blue curve) and $s = 10$ (purple curve). The scale parameter s controls the activation rate, and we can see that large s amounts to a hard activation at $v = 0$. Note that $\sigma(s(v - v_0))$ shifts the activation threshold from 0 to v_0.*

Credit: Hastie et al. (2009)

Fitting the Vanilla Neural Network - Classification Problem

- Hidden Layer ($\forall m$ in [1, M]):
  $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- Output Layer ($\forall k$ in [1, K]):
  $T_k = \beta_{0k} + \beta_k^T Z$
- Softmax Output ($\forall k$ in [1, K]):
  $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = f_k(X)$

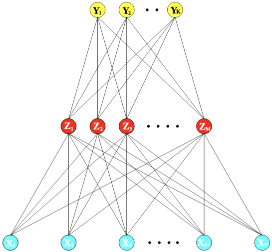


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Fitting the Vanilla Neural Network - Classification Problem

- Hidden Layer ($\forall m$ in [1, M]):
  $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- Output Layer ($\forall k$ in [1, K]):
  $T_k = \beta_{0k} + \beta_k^T Z$
- Softmax Output ($\forall k$ in [1, K]):
  $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = f_k(X)$

Dimensionality and Loss Function

- $\alpha_{0m} \in \mathbb{R}^M$, $\alpha_m \in \mathbb{R}^{M \times p}$, $\beta_{0k} \in \mathbb{R}^K$, $\beta_k \in \mathbb{R}^{M \times K}$.
- Total Weights to Optimize (θ):
  $M(p+1) + K(M+1)$.
- $L(\theta) = -\sum_{k=1}^{K} y_k log(f_k(x))$
- $L(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(f_k(x_i))$

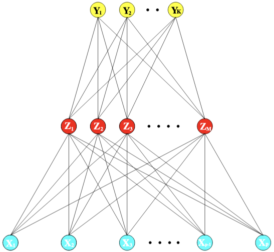


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Credit: Hastie et al. (2009)

Gradient Descent Algorithm

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."
- **Objective**: Minimizes loss function $L(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."
- **Objective**: Minimizes loss function $L(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.
- **Types of Gradient Descent**:

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."
- **Objective**: Minimizes loss function $L(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.
- **Types of Gradient Descent**:
  - *Batch Gradient Descent (Vanilla)*: Uses the entire dataset for each update.

Introduction to Gradient Descent

- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."
- **Objective**: Minimizes loss function $L(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.
- **Types of Gradient Descent**:
  - *Batch Gradient Descent (Vanilla)*: Uses the entire dataset for each update.
  - *Stochastic Gradient Descent*: Updates parameters for each training example.

Introduction to Gradient Descent

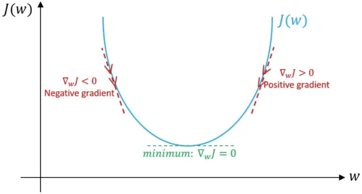
- **Popular Optimization Algorithm**: Widely used for function optimization, for logistic regression or NNs.
- **Neural Network Optimization**: As highlighted by Ruder (2020), it's the most common method for optimizing neural networks.
- **Widespread Implementation**: Integral to many libraries like TensorFlow, Keras, PyTorch, and Caffe, often utilized as "black-boxes."
- **Objective**: Minimizes loss function $L(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.
- **Types of Gradient Descent**:
  - *Batch Gradient Descent (Vanilla)*: Uses the entire dataset for each update.
  - *Stochastic Gradient Descent*: Updates parameters for each training example.
  - *Mini-batch Gradient Descent*: Strikes a balance using subsets of the dataset.

Batch Gradient Descent (Vanilla)

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

with η as the **learning rate**.



Credit: Imad Daburra

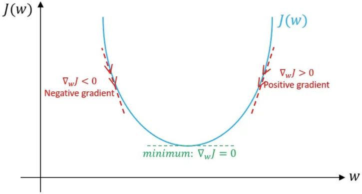
Batch Gradient Descent (Vanilla)

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

with η as the **learning rate**.

Direction of Update: Updates parameters in the **opposite direction** to the gradient of the objective function $\nabla_\theta L(\theta)$.



Credit: Imad Daburra

Batch Gradient Descent (Vanilla)

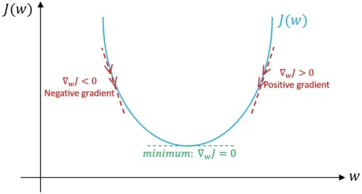
Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

with η as the **learning rate**.

Direction of Update: Updates parameters in the **opposite direction** to the gradient of the objective function $\nabla_\theta L(\theta)$.

Analogy: Similar to descending a mountain to reach a valley, considering the slope to find the optimal path.



Credit: Imad Daburra

Batch Gradient Descent (Vanilla)

Update Rule:

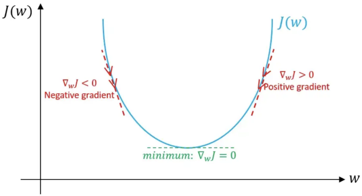
$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

with η as the **learning rate**.

Direction of Update: Updates parameters in the **opposite direction** to the gradient of the objective function $\nabla_\theta L(\theta)$.

Analogy: Similar to descending a mountain to reach a valley, considering the slope to find the optimal path.

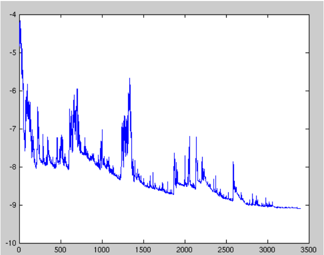
Limitation: Requires processing **the entire dataset** for each update, problematic for large datasets due to memory constraints.



Credit: Imad Daburra

Stochastic Gradient Descent

Update for Each Observation: Applies the gradient descent update rule for **each observation** individually, chosen randomly.



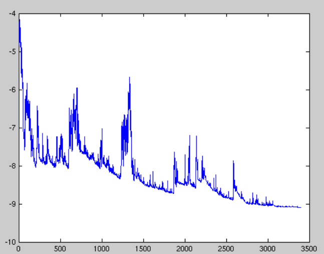
Credit: Wikipedia

Stochastic Gradient Descent

Update for Each Observation: Applies the gradient descent update rule for **each observation** individually, chosen randomly.

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_i, y_i)$$



Credit: Wikipedia

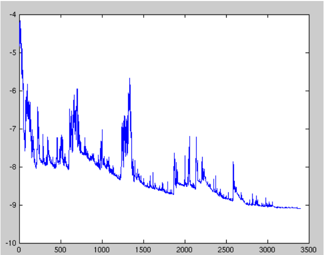
Stochastic Gradient Descent

Update for Each Observation: Applies the gradient descent update rule for **each observation** individually, chosen randomly.

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_i, y_i)$$

Advantages: Faster updates, suitable for online learning, and better exploration of minima compared to batch gradient descent.



Credit: Wikipedia

Stochastic Gradient Descent

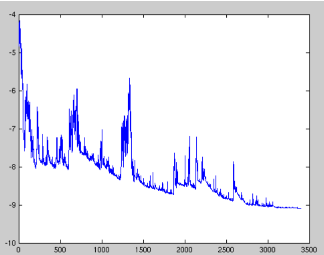
Update for Each Observation: Applies the gradient descent update rule for **each observation** individually, chosen randomly.

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_i, y_i)$$

Advantages: Faster updates, suitable for online learning, and better exploration of minima compared to batch gradient descent.

Challenge: Tendency to oscillate around or even overshoot minima. Reducing η over time can mitigate this issue.



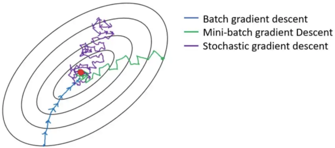
Credit: Wikipedia

Mini-batch Gradient Descent

Combining the Best of Both Worlds!

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$



Credit: Imad Daburra

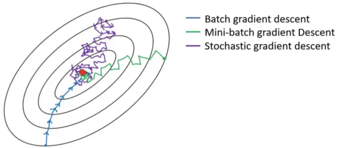
Mini-batch Gradient Descent

Combining the Best of Both Worlds!

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

Reduced Variance: Balances the variance of updates: more stable convergence than SGD.



Credit: Imad Daburra

Mini-batch Gradient Descent

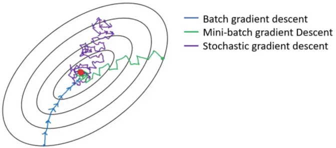
Combining the Best of Both Worlds!

Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

Reduced Variance: Balances the variance of updates: more stable convergence than SGD.

Efficiency: Tends to be faster than Batch Gradient Descent, particularly for large datasets.



Credit: Imad Daburra

Mini-batch Gradient Descent

Combining the Best of Both Worlds!

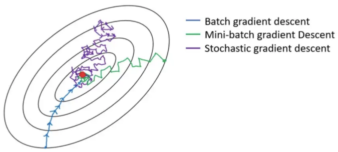
Update Rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

Reduced Variance: Balances the variance of updates: more stable convergence than SGD.

Efficiency: Tends to be faster than Batch Gradient Descent, particularly for large datasets.

Widely Adopted: Often the preferred choice in practical applications and deep learning frameworks.



Credit: Imad Daburra

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:
  - Individual learning rates for different parameters.

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:
  - Individual learning rates for different parameters.
  - Techniques: Adagrad (Duchi et al., 2011), RMSprop (Hinton et al.), Adam (Kingma et al., 2015).

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:
  - Individual learning rates for different parameters.
  - Techniques: Adagrad (Duchi et al., 2011), RMSprop (Hinton et al.), Adam (Kingma et al., 2015).
- **Non-Convex Functions**:

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:
  - Individual learning rates for different parameters.
  - Techniques: Adagrad (Duchi et al., 2011), RMSprop (Hinton et al.), Adam (Kingma et al., 2015).
- **Non-Convex Functions**:
  - Gradient descent can struggle with complex, non-convex functions typical in deep neural networks.

Main Challenges in Gradient Descent

- **Optimizing Learning Rate**:
  - *Too Low*: Slow convergence.
  - *Too High*: May lead to divergence.
- **Adaptive Learning Rates**:
  - *Dynamic Adjustments*: Varying the rate during exploration and refinement phases.
  - Examples: Momentum (Qian, 1999), Nesterov accelerated gradient (Nesterov, 1983).
- **Parameter-Specific Learning Rates**:
  - Individual learning rates for different parameters.
  - Techniques: Adagrad (Duchi et al., 2011), RMSprop (Hinton et al.), Adam (Kingma et al., 2015).
- **Non-Convex Functions**:
  - Gradient descent can struggle with complex, non-convex functions typical in deep neural networks.
  - Issue: Getting trapped in local minima.

Apply Gradient Descent Algorithm

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik} \log(f_k(x_i))$$

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik}\log(f_k(x_i))$$

- **Optimization Algorithm**:

$$\theta = \theta - \eta\nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik} \log(f_k(x_i))$$

- **Optimization Algorithm**:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

- **Parameter Dimensions**:

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik}\log(f_k(x_i))$$

- **Optimization Algorithm**:

$$\theta = \theta - \eta\nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

- **Parameter Dimensions**:
  - $\alpha_{0m} \in \mathbb{R}^M$, $\alpha_m \in \mathbb{R}^{M\times p}$

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik}\log(f_k(x_i))$$

- **Optimization Algorithm**:

$$\theta = \theta - \eta\nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

- **Parameter Dimensions**:
  - $\alpha_{0m} \in \mathbb{R}^M$, $\alpha_m \in \mathbb{R}^{M\times p}$
  - $\beta_0 \in \mathbb{R}^K$, $\beta_k \in \mathbb{R}^{M\times K}$

Applying the Gradient Descent Algorithm

- **Loss Function**:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik}\log(f_k(x_i))$$

- **Optimization Algorithm**:

$$\theta = \theta - \eta \nabla_\theta L(\theta, x_{i:i+n}, y_{i:i+n})$$

- **Parameter Dimensions**:
  - $\alpha_{0m} \in \mathbb{R}^M$, $\alpha_m \in \mathbb{R}^{M\times p}$
  - $\beta_0 \in \mathbb{R}^K$, $\beta_k \in \mathbb{R}^{M\times K}$
- **Understanding the Chain Rule**:

$$\frac{\partial f(x)}{\partial z} = \frac{\partial f(x)}{\partial t}\frac{\partial t}{\partial z}$$

Key to computing gradients for backpropagation.

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- For each output unit k in $[1, K]$: $T_k = \beta_{0k} + \beta_k^T Z$

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- For each output unit k in $[1, K]$: $T_k = \beta_{0k} + \beta_k^T Z$
- For the softmax output: $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = g_k(T) = f_k(X)$

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- For each output unit k in $[1, K]$: $T_k = \beta_{0k} + \beta_k^T Z$
- For the softmax output: $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = g_k(T) = f_k(X)$
- Loss function:

$$L(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} \log(f_k(x_i))$$

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- For each output unit k in $[1, K]$: $T_k = \beta_{0k} + \beta_k^T Z$
- For the softmax output: $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = g_k(T) = f_k(X)$
- Loss function:

$$L(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} \log(f_k(x_i))$$

Application to a Single-Layer Neural Network

Classification Problem Formulation:

- For each hidden unit m in $[1, M]$: $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$
- For each output unit k in $[1, K]$: $T_k = \beta_{0k} + \beta_k^T Z$
- For the softmax output: $Y_k = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} = g_k(T) = f_k(X)$
- Loss function:

$$L(\theta) = -\sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik} \log(f_k(x_i))$$

Applying the Chain Rule to Compute Gradients for β_k:

- Gradient of the loss function with respect to β_k:

$$\frac{\partial L(\theta)}{\partial \beta_k} = -\frac{\partial}{\partial \beta_k}\sum_{j=1}^{K}\sum_{i=1}^{N} y_{ij} \log(f_j(x_i))$$

Derivatives of β_k - Part 1

$$
\begin{aligned}
\frac{\partial L(\theta)}{\partial \beta_k} &= -\sum_{j=1}^{K} Y_j \frac{\partial \log(\hat{Y}_j)}{\partial \beta_k} \\
&= -\sum_{j=1}^{K} Y_j \left(\frac{\partial T_j}{\partial \beta_k} - \frac{\partial \log(\sum_{l=1}^{K} e^{T_l})}{\partial \beta_k} \right) \\
&= -\sum_{j=1}^{K} Y_j \left(\mathbf{1}_{j=k} Z^T - \frac{e^{T_k} Z^T}{\sum_{l=1}^{K} e^{T_l}} \right) \\
&= -\sum_{j=1}^{K} Y_j \left(\mathbf{1}_{j=k} Z^T - \hat{Y}_k Z^T \right)
\end{aligned}
$$

Derivatives of β_k - Part 2

$$\frac{\partial L(\theta)}{\partial \beta_k} = -\sum_{j=1}^{K} Y_j \left(\mathbf{1}_{j=k} Z^T - \hat{Y}_k Z^T \right)$$

$$= \left(\sum_{j=1}^{K} Y_j \hat{Y}_k - \sum_{j=1}^{K} Y_j \mathbf{1}_{j=k} \right) Z^T$$

$$= \left(\hat{Y}_k \sum_{j=1}^{K} Y_j - Y_k \right) Z^T$$

$$= \left(\hat{Y}_k - Y_k \right) Z^T$$

$$\beta_k^{r+1} = \beta_k^r - \eta \frac{\partial L(\theta)}{\partial \beta_k}$$

$$\beta_k^{r+1} = \beta_k^r - \eta \left(\hat{Y}_k - Y_k \right) Z^T$$

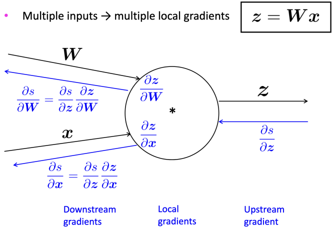
Backpropagation: Understanding the Chain Rule

The Chain Rule in Neural Networks:

- Fundamental to backpropagation:

$$\frac{\partial f(x)}{\partial z} = \frac{\partial f(x)}{\partial s}\frac{\partial s}{\partial z}$$

- **downstream gradient** $=$ **upstream gradient** $\times$ **local gradient**.
- This principle encounters challenges:
  - *Vanishing Gradient*: Gradients become very small, hindering learning.
  - *Exploding Gradient*: Gradients grow too large, leading to unstable learning.
- It can prevent the model from learning!



Credit: Christopher Manning

Understanding Vanishing Gradient

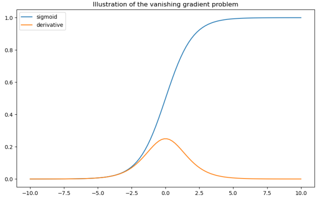
- When gradients become increasingly small as they are propagated back through the layers.
- Especially in networks with many layers.

Illustrative Example:

- Consider a deep NN with sigmoid.
- Sigmoid gradients in $(0, 0.25]$.
- Multiplying many such small values (chain rule!) makes the gradient increasingly smaller.

Consequence:

- Lower layers of the network learn very slowly, making training ineffective.



Sigmoid and its derivative

Understanding Exploding Gradient

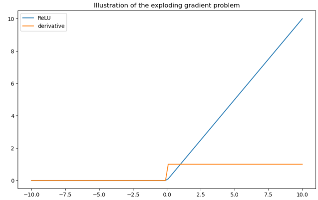
- When gradients become excessively large: model weights oscillate wildly.
- Often seen in NN with improper initialization or high learning rates.

Illustrative Example:

- NN with large weight values and high learning rates.
- Small changes in input lead to large changes in the output.
- Gradients can grow exponentially during backpropagation through layers.

Consequence:

- Results in unstable training: weights diverge and NN fail to converge.



ReLU and its derivative

Complex Models

Recurrent Neural Networks

Introduction to Recurrent Neural Networks (RNNs)

Overview of RNNs:

- RNNs, introduced by Rumelhart et al. (1986), are powerful networks for sequential data processing.
- Key Models: Vanilla RNNs and Long Short-Term Memory (LSTM) networks.
- State-of-the-art in various NLP tasks (e.g., machine translation, text generation) before the advent of Transformers and BERT models.

Introduction to Recurrent Neural Networks (RNNs)

Motivation for Using RNNs:

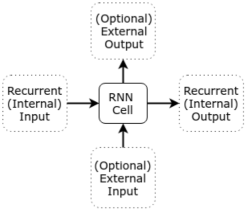
- **Sequential Data Processing**:
  - Traditional feed-forward networks are not optimized for sequential data like text or time series.
  - RNNs are designed to handle data where variables are interlinked sequentially.
- **Example - Text Analysis**:
  - For a word like "mathematics," tokenized as "m, a, t, h, e, m, a, t, i, c, s," RNNs can capture the sequence's inherent dependencies.
  - This sequential understanding is crucial for tasks like language modeling and translation.

Recurrent Neural Networks - General

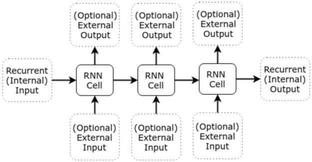
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.



Single RNN Cell



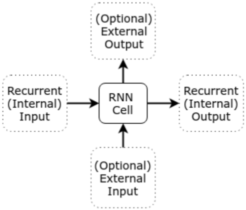
Several RNN Cells

Credits: R2Rt blog

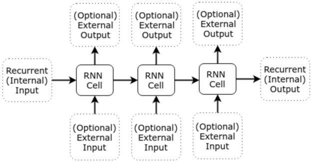
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:



Single RNN Cell



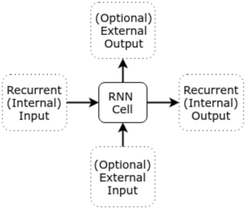
Several RNN Cells

Credits: R2Rt blog

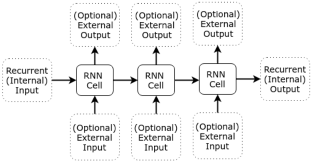
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
  - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.



Single RNN Cell



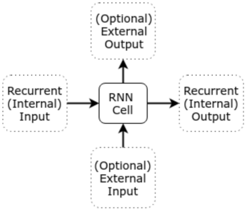
Several RNN Cells

Credits: R2Rt blog

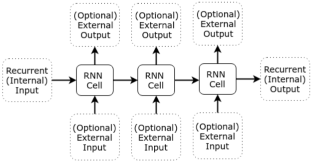
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
  - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.
  - *Internal Input*: The state output from the previous cell.



Single RNN Cell



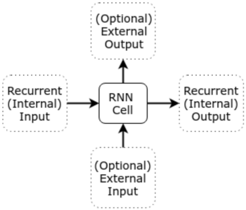
Several RNN Cells

Credits: R2Rt blog

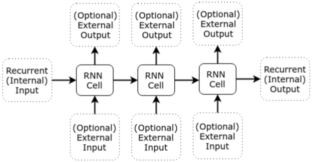
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
  - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.
  - *Internal Input*: The state output from the previous cell.
- **Outputs for Each Cell**:



Single RNN Cell



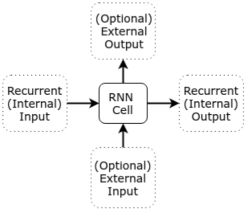
Several RNN Cells

Credits: R2Rt blog

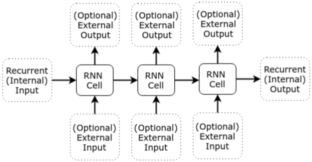
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
  - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.
  - *Internal Input*: The state output from the previous cell.
- **Outputs for Each Cell**:
  - *External Output*: Can be used or ignored depending on the application.



Single RNN Cell



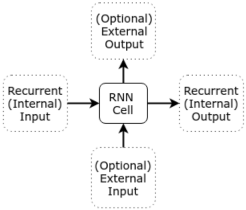
Several RNN Cells

Credits: R2Rt blog

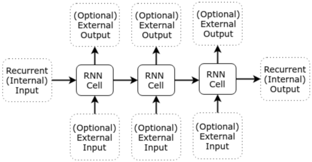
What is a Recurrent Neural Network?

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
  - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.
  - *Internal Input*: The state output from the previous cell.
- **Outputs for Each Cell**:
  - *External Output*: Can be used or ignored depending on the application.
  - *Internal Output*: The state passed to the next cell.



Single RNN Cell

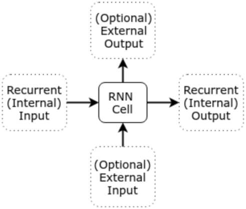


Several RNN Cells

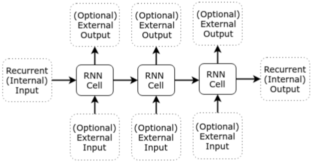
Credits: R2Rt blog

Characteristics of RNNs:

- Composed of identical units resembling feed-forward neural networks.
- **Inputs for Each Cell**:
    - *External Input* (optional): For ex., characters in a word like *p,h,o,n,e*.
    - *Internal Input*: The state output from the previous cell.
- **Outputs for Each Cell**:
    - *External Output*: Can be used or ignored depending on the application.
    - *Internal Output*: The state passed to the next cell.
- Functions by passing states from one cell to the next in a sequence.



Single RNN Cell



Several RNN Cells

Credits: R2Rt blog

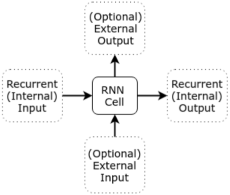
Mathematical Description of a Recurrent Neural Network

Mathematical Formulation:

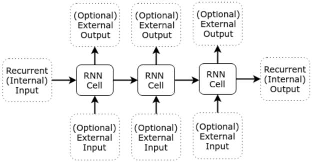
$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f\left(\begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix}\right)$$

Where:

- s_t and s_{t-1} are the current and previous states, respectively.



Single RNN Cell



Several RNN Cells

Credits: R2Rt blog

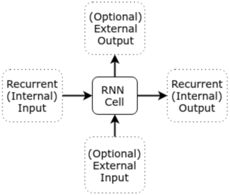
Mathematical Description of a Recurrent Neural Network

Mathematical Formulation:

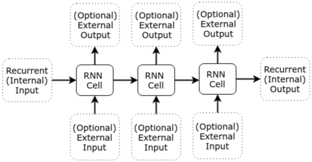
$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f\left(\begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix}\right)$$

Where:

- s_t and s_{t-1} are the current and previous states, respectively.
- o_t is the output at time t.



Single RNN Cell



Several RNN Cells

Credits: R2Rt blog

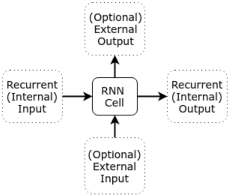
Mathematical Description of a Recurrent Neural Network

Mathematical Formulation:

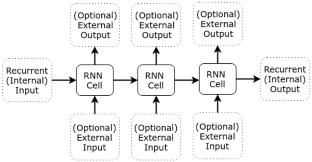
$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f\left(\begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix}\right)$$

Where:

- s_t and s_{t-1} are the current and previous states, respectively.
- o_t is the output at time t.
- x_t is the current input (optional).



Single RNN Cell



Several RNN Cells

Credits: R2Rt blog

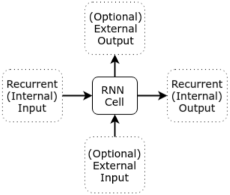
Mathematical Description of a Recurrent Neural Network

Mathematical Formulation:

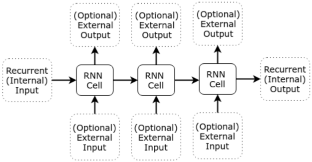
$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f\left(\begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix}\right)$$

Where:

- s_t and s_{t-1} are the current and previous states, respectively.
- o_t is the output at time t.
- x_t is the current input (optional).
- f represents the recurrent function, defining how the next state and output are computed.



Single RNN Cell

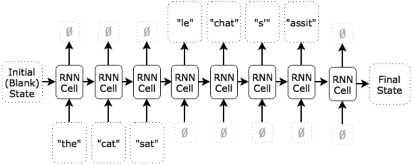


Several RNN Cells

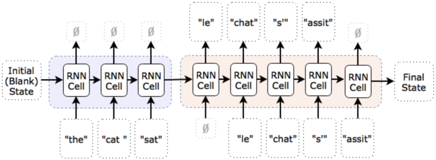
Credits: R2Rt blog

RNNs in Translation Tasks

- RNNs are particularly effective in sequence-to-sequence tasks like language translation.
- They process sequential inputs and generate sequential outputs, capturing the nuances of language patterns.



RNN for Translation - Example 1



RNN for Translation - Example 2

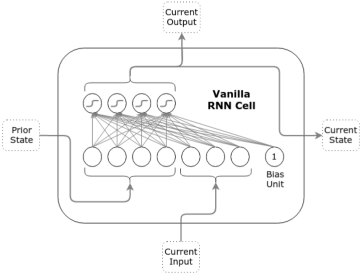
Credit: R2Rt blog

Vanilla Recurrent Neural Network

The Vanilla RNN

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.



The Vanilla RNN.

Credit: R2Rt blog

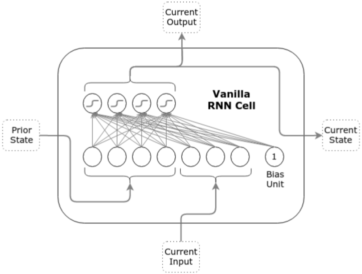
Mathematical Description:

The Vanilla RNN

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.
- Prior and current states have the same dimension.

Mathematical Description:



The Vanilla RNN.
Credit: R2Rt blog

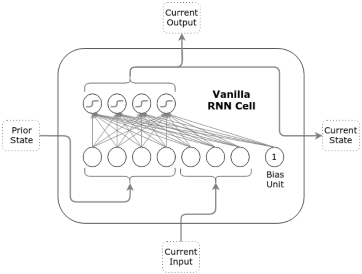
The Vanilla RNN

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.
- Prior and current states have the same dimension.

Mathematical Description:

- State Update:
  $s_t = \phi(Ws_{t-1} + Ux_t + b)$



The Vanilla RNN.
Credit: R2Rt blog

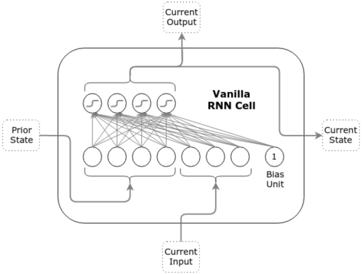
The Vanilla RNN

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.
- Prior and current states have the same dimension.

Mathematical Description:

- State Update: $s_t = \phi(Ws_{t-1} + Ux_t + b)$
- Activation Function: ϕ



The Vanilla RNN. Credit: R2Rt blog

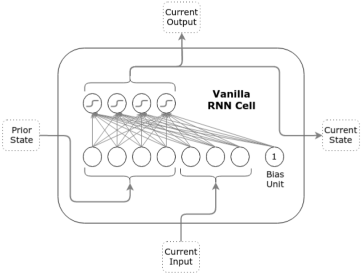
The Vanilla RNN

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.
- Prior and current states have the same dimension.

Mathematical Description:

- State Update:
  $s_t = \phi(W s_{t-1} + U x_t + b)$
- Activation Function: ϕ
- Dimensions: $s_t, s_{t-1} \in \mathbb{R}^n$, $x_t \in \mathbb{R}^m$



The Vanilla RNN.

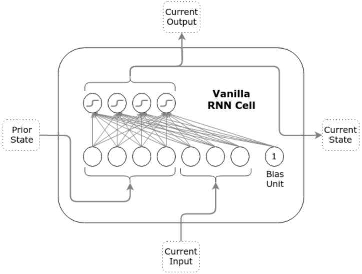
Credit: R2Rt blog

Characteristics of the Vanilla RNN:

- Features a single layer with identical current output and current state.
- Prior and current states have the same dimension.

Mathematical Description:

- State Update:
  $s_t = \phi(W s_{t-1} + U x_t + b)$
- Activation Function: ϕ
- Dimensions: $s_t, s_{t-1} \in \mathbb{R}^n$, $x_t \in \mathbb{R}^m$
- Weights: $W \in \mathbb{R}^{n \times n}$, $U \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$



The Vanilla RNN.

Credit: R2Rt blog

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.
- Known as the *degradation problem* (He et al., 2015).

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.
- Known as the *degradation problem* (He et al., 2015).

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.
- Known as the *degradation problem* (He et al., 2015).

Exploding Gradients: Can prevent model training; mitigated by limiting gradient values (Mikolov, 2012).

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.
- Known as the *degradation problem* (He et al., 2015).

Exploding Gradients: Can prevent model training; mitigated by limiting gradient values (Mikolov, 2012). **Vanishing Gradients**:

- Challenge in learning long-term dependencies.

Main Limitations of RNNs

Information Morphing:

- **State Transformation**: Information (s_t) changes from one state to another, potentially losing key information from the distant past.
- **Dual Learning Challenge**:
  - Learning to **read** the previous state.
  - Learning to **use** the current state effectively.
- Known as the *degradation problem* (He et al., 2015).

Exploding Gradients: Can prevent model training; mitigated by limiting gradient values (Mikolov, 2012). **Vanishing Gradients**:

- Challenge in learning long-term dependencies.
- Mitigation strategies: regularization and specific weight initialization (Pascanu et al., 2013; Xavier-Glorot, Glorot and Bengio, 2010).

Long Short Term Memory

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by ”writing down” information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by ”writing down” information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:
  - Challenge: Ensuring that only relevant changes are captured.

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:
  - Challenge: Ensuring that only relevant changes are captured.
  - *Selection Mechanisms*:

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:
  - Challenge: Ensuring that only relevant changes are captured.
  - *Selection Mechanisms*:
    - **Write Gate**: Determines what to update in the memory.

Principles of LSTM Networks

Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:
  - Challenge: Ensuring that only relevant changes are captured.
  - *Selection Mechanisms*:
    - **Write Gate**: Determines what to update in the memory.
    - **Read Gate**: Controls what part of the memory to consider for the current output.

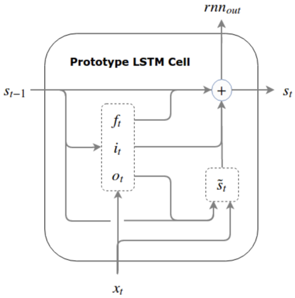
Overcoming Information Morphing:

- **Persistent Memory**: How to retain important information through time steps?
  - Solution: Introduce a **memory mechanism by "writing down" information** (Hochreiter & Schmidhuber, 1997).
  - Instead of replacing states, the model incrementally updates (writes) them: $s_{t+1} = s_t + \Delta s_{t+1}$.
- **Selective Memory Updates**:
  - Challenge: Ensuring that only relevant changes are captured.
  - *Selection Mechanisms*:
    - **Write Gate**: Determines what to update in the memory.
    - **Read Gate**: Controls what part of the memory to consider for the current output.
    - **Forget Gate**: Decides which parts of the memory may no longer be relevant.

Long Short Term Memory - Gate Functions

LSTM Gate Functions:

- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$

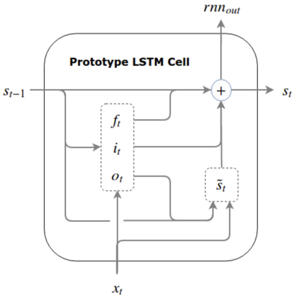


figurePrototype LSTM Cell.
Credit: R2Rt blog

Long Short Term Memory - Gate Functions

LSTM Gate Functions:

- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$
- **Read Gate** (o_t): Controls what to output based on cell state. $o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$

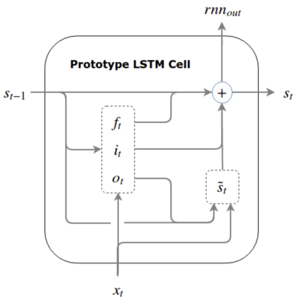


figurePrototype LSTM Cell. Credit: R2Rt blog

Long Short Term Memory - Gate Functions

LSTM Gate Functions:

- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$
- **Read Gate** (o_t): Controls what to output based on cell state. $o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$
- **Forget Gate** (f_t): Decides what to discard from the cell state. $f_t = \sigma(W_f s_{t-1} + U_f x_t + b_f)$

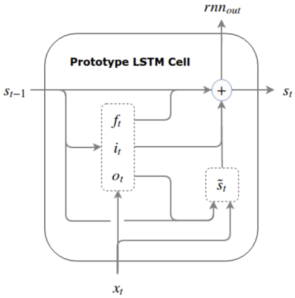


figurePrototype LSTM Cell. Credit: R2Rt blog

Long Short Term Memory - Gate Functions

LSTM Gate Functions:

- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$
- **Read Gate** (o_t): Controls what to output based on cell state. $o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$
- **Forget Gate** (f_t): Decides what to discard from the cell state. $f_t = \sigma(W_f s_{t-1} + U_f x_t + b_f)$
- **Cell State Update**:

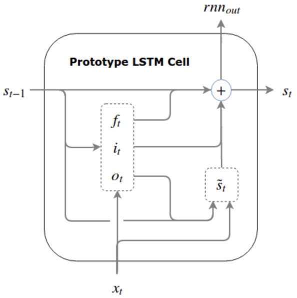


figurePrototype LSTM Cell. Credit: R2Rt blog

Long Short Term Memory - Gate Functions

LSTM Gate Functions:

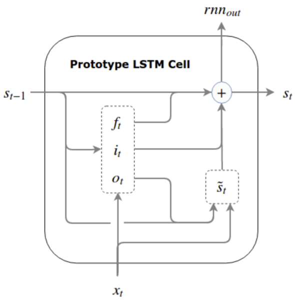
- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$
- **Read Gate** (o_t): Controls what to output based on cell state. $o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$
- **Forget Gate** (f_t): Decides what to discard from the cell state. $f_t = \sigma(W_f s_{t-1} + U_f x_t + b_f)$
- **Cell State Update**:
  - New candidate values: $\tilde{s}_t = \phi(W_c(o_t \odot s_{t-1}) + U_c x_t + b_c)$



figurePrototype LSTM Cell.
Credit: R2Rt blog

LSTM Gate Functions:

- **Write Gate** (i_t): Determines new information to be stored in the cell state. $i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$
- **Read Gate** (o_t): Controls what to output based on cell state. $o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$
- **Forget Gate** (f_t): Decides what to discard from the cell state. $f_t = \sigma(W_f s_{t-1} + U_f x_t + b_f)$
- **Cell State Update**:
  - New candidate values: $\tilde{s}_t = \phi(W_c(o_t \odot s_{t-1}) + U_c x_t + b_c)$
  - Final cell state: $s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$



figurePrototype LSTM Cell.

Credit: R2Rt blog

State-of-the-Art Applications of LSTM (or extentions)

1. **Sentiment Analysis**:
   - LSTM networks, often in combination with word embeddings, have set new benchmarks in sentiment analysis tasks.
   - SST-2 dataset (Radford et al., 2017), IMDb (Gray et al., 2017)

2. **Machine Translation (MT)**:
   - LSTM-based models were pivotal in advancing the performance of neural machine translation systems.
   - English - German (Luong et al., 2015), English-French (Cho et al., 2014)

3. **Language Modelling**:
   - LSTMs have been successfully applied in language modelling, reducing text perplexity substantially.
   - WikiText-103 dataset (Rae et al., 2018), TreeBank dataset (Aharoni et al., 2015)

LSTM for IMDb classification (1/3)

Generating a Classificaiton model with LSTM architecture

Using Python's keras library to apply a LSTM-based model.

Python Code, source: Keras

```
import numpy as np
import keras
from keras import layers

max_features = 20000  # Only consider the top 20k words
maxlen = 200  # Only consider the first 200 words of each movie review
```

LSTM for IMDb classification (2/3)

```
# Input for variable-length sequences of integers
inputs = keras.Input(shape=(None,), dtype="int32")
# Embed each integer in a 128-dimensional vector
x = layers.Embedding(max_features, 128)(inputs)
# Add 2 bidirectional LSTMs
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
x = layers.Bidirectional(layers.LSTM(64))(x)
# Add a classifier
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, None)	0
embedding (Embedding)	(None, None, 128)	2,560,000
bidirectional (Bidirectional)	(None, None, 128)	98,816
bidirectional_1 (Bidirectional)	(None, 128)	98,816
dense (Dense)	(None, 1)	129

LSTM for IMDb classification (3/3)

Python Code to train, source: Keras

```
(x_train, y_train), (x_val, y_val) = keras.datasets.imdb.load_data(
num_words=max_features
)

# Use pad_sequence to standardize sequence length:
# this will truncate sequences longer than 200 words
# and zero-pad sequences shorter than 200 words.
x_train = keras.utils.pad_sequences(x_train, maxlen=maxlen)
x_val = keras.utils.pad_sequences(x_val, maxlen=maxlen)

model.compile(optimizer="adam", loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=32, epochs=2,
          validation_data=(x_val, y_val))
```

```
Epoch 1/2
 782/782 ━━━━━━━━━━━━━━━━━━━━ 61s 75ms/step - accuracy: 0.7540 - loss: 0.4697 - val_accu
Epoch 2/2
 782/782 ━━━━━━━━━━━━━━━━━━━━ 54s 69ms/step - accuracy: 0.9151 - loss: 0.2263 - val_accu

<keras.src.callbacks.history.History at 0x7f3efd663850>
```

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.

Extensions and Variants:

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

- **Normalized Prototype & GRU** (Cho et al., 2014): Introduce bounds to prevent saturation, simplifying the architecture.

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

- **Normalized Prototype & GRU** (Cho et al., 2014): Introduce bounds to prevent saturation, simplifying the architecture.
- **LSTM Variants**:

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

- **Normalized Prototype & GRU** (Cho et al., 2014): Introduce bounds to prevent saturation, simplifying the architecture.
- **LSTM Variants**:
  - *Basic LSTM*: Standard implementation in frameworks like Keras, TensorFlow, or PyTorch.

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

- **Normalized Prototype & GRU** (Cho et al., 2014): Introduce bounds to prevent saturation, simplifying the architecture.
- **LSTM Variants**:
  - *Basic LSTM*: Standard implementation in frameworks like Keras, TensorFlow, or PyTorch.
  - *LSTM hiccup*: to limit states saturation in the basic LSTM.

Main Limitations of LSTM & Related Works

Limitations of LSTM:

- **Lack of Coordination**: Forget and write gates may lack coordination, leading to unnecessarily large state sizes.
- **Unbounded State**: Gates and candidate states can become saturated, affecting the model's performance.

Extensions and Variants:

- **Normalized Prototype & GRU** (Cho et al., 2014): Introduce bounds to prevent saturation, simplifying the architecture.
- **LSTM Variants**:
  - *Basic LSTM*: Standard implementation in frameworks like Keras, TensorFlow, or PyTorch.
  - *LSTM hiccup*: to limit states saturation in the basic LSTM.
  - *LSTM with Peepholes* (Graves, 2013): Incorporates peephole connections to enhance the model's memory capability.

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.

Formal Definition:

Importance of Language Modeling:

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.
- Serving as the foundation for various applications like text generation, machine translation, and speech recognition.

Formal Definition:

Importance of Language Modeling:

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.
- Serving as the foundation for various applications like text generation, machine translation, and speech recognition.

Formal Definition:

- Given a sequence of words $w_1, w_2, ..., w_n$, a LM computes the probability $P(w_1, w_2, ..., w_n)$.

Importance of Language Modeling:

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.
- Serving as the foundation for various applications like text generation, machine translation, and speech recognition.

Formal Definition:

- Given a sequence of words $w_1, w_2, ..., w_n$, a LM computes the probability $P(w_1, w_2, ..., w_n)$.
- With probability's chain rule: $P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_1, w_2, ..., w_{i-1})$.

Importance of Language Modeling:

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.
- Serving as the foundation for various applications like text generation, machine translation, and speech recognition.

Formal Definition:

- Given a sequence of words $w_1, w_2, ..., w_n$, a LM computes the probability $P(w_1, w_2, ..., w_n)$.
- With probability's chain rule: $P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_1, w_2, ..., w_{i-1})$.

Importance of Language Modeling:

- Enables NLP systems in generating human-like language.

Introduction to Language Modeling

What is Language Modeling?

- The task of predicting the probability of a sequence of words.
- Serving as the foundation for various applications like text generation, machine translation, and speech recognition.

Formal Definition:

- Given a sequence of words $w_1, w_2, ..., w_n$, a LM computes the probability $P(w_1, w_2, ..., w_n)$.
- With probability's chain rule: $P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_1, w_2, ..., w_{i-1})$.

Importance of Language Modeling:

- Enables NLP systems in generating human-like language.
- Used to train BERT and GPT-like models.

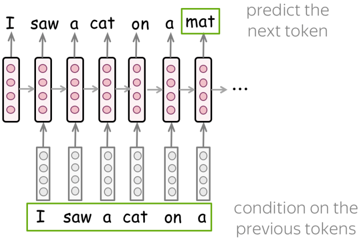
Language Model as Next Token Prediction

Next Token Prediction:

- $P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_1, w_2, ..., w_{i-1})$
- Focus on **Next Token Prediction**, $P(w_i | w_1, w_2, ..., w_{i-1})$: predict the next word given previous ones.

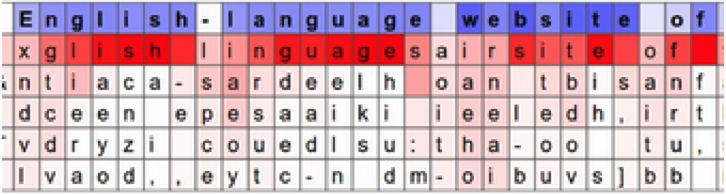
With RNNs:

- **Input**: Sequence of tokens. "I saw a cart on a", the model receives "I", "saw", "a", "cat", "on", "a" as input one after the other.
- **Output**: At each step, the RNN predicts the probability distribution of the next token. Here "mat"



Credit: Lena Voita

Next Token Prediction with top-5 proposition when training a model:



Credit: Karpathy

From Language Modeling to Word Embeddings with RNN (1/2)

Language Modeling with RNN:

- RNNs are a powerful tool for language modeling, capturing the sequential nature and dependencies between words in text data.

From Language Modeling to Word Embeddings with RNN (1/2)

Language Modeling with RNN:

- RNNs are a powerful tool for language modeling, capturing the sequential nature and dependencies between words in text data.
- Traditionally, words were represented as one-hot vectors, where each word is represented as a vector of the size of the vocabulary with all zeros except for a single one at the index of the word.

From Language Modeling to Word Embeddings with RNN (1/2)

Language Modeling with RNN:

- RNNs are a powerful tool for language modeling, capturing the sequential nature and dependencies between words in text data.
- Traditionally, words were represented as one-hot vectors, where each word is represented as a vector of the size of the vocabulary with all zeros except for a single one at the index of the word.

From Language Modeling to Word Embeddings with RNN (1/2)

Language Modeling with RNN:

- RNNs are a powerful tool for language modeling, capturing the sequential nature and dependencies between words in text data.
- Traditionally, words were represented as one-hot vectors, where each word is represented as a vector of the size of the vocabulary with all zeros except for a single one at the index of the word.

Limitations of One-Hot Representations:

- **Sparsity**: One-hot vectors are sparse and do not capture any semantic/contextual information.

From Language Modeling to Word Embeddings with RNN (1/2)

Language Modeling with RNN:

- RNNs are a powerful tool for language modeling, capturing the sequential nature and dependencies between words in text data.
- Traditionally, words were represented as one-hot vectors, where each word is represented as a vector of the size of the vocabulary with all zeros except for a single one at the index of the word.

Limitations of One-Hot Representations:

- **Sparsity**: One-hot vectors are sparse and do not capture any semantic/contextual information.
- **Dimensionality**: The dimension of one-hot vectors grows with the size of the vocabulary, leading to scalability issues.

From Language Modeling to Word Embeddings with RNN (2/2)

Transition to Dense Word Embeddings:

- RNNs, coupled with language modeling, can be used to learn dense word vectors, also known as word embeddings.

From Language Modeling to Word Embeddings with RNN (2/2)

Transition to Dense Word Embeddings:

- RNNs, coupled with language modeling, can be used to learn dense word vectors, also known as word embeddings.
- **Richer Representations**: Word embeddings capture more than just the identity of words; they encode semantic meaning and context.

From Language Modeling to Word Embeddings with RNN (2/2)

Transition to Dense Word Embeddings:

- RNNs, coupled with language modeling, can be used to learn dense word vectors, also known as word embeddings.
- **Richer Representations**: Word embeddings capture more than just the identity of words; they encode semantic meaning and context.
- **Efficiency**: Embeddings are lower-dimensional and dense, addressing the issues of sparsity and high dimensionality in one-hot representations.

From Language Modeling to Word Embeddings with RNN (2/2)

Transition to Dense Word Embeddings:

- RNNs, coupled with language modeling, can be used to learn dense word vectors, also known as word embeddings.
- **Richer Representations**: Word embeddings capture more than just the identity of words; they encode semantic meaning and context.
- **Efficiency**: Embeddings are lower-dimensional and dense, addressing the issues of sparsity and high dimensionality in one-hot representations.

From Language Modeling to Word Embeddings with RNN (2/2)

Transition to Dense Word Embeddings:

- RNNs, coupled with language modeling, can be used to learn dense word vectors, also known as word embeddings.
- **Richer Representations**: Word embeddings capture more than just the identity of words; they encode semantic meaning and context.
- **Efficiency**: Embeddings are lower-dimensional and dense, addressing the issues of sparsity and high dimensionality in one-hot representations.

Upcoming Session: We will delve deeper into the world of word embeddings, exploring how they revolutionize the understanding and representation of words in NLP models.

QA

Open Discussion

- Feel free to ask questions or share your thoughts about today's topics.
- Any insights, experiences, or perspectives you'd like to discuss are welcome.

Summary of Key Takeaways

- **Neural Networks**: Explored the fundamentals of Neural Networks, including Vanilla Networks, Backpropagation, and Gradient Descent.
- **Gradient issues**: Illustrated the the issues of vanishing and exploding gradients and gave some paths to avoid it.
- **RNNs**: Discussed the significance of RNNs in handling sequential data and their applications in tasks like language modeling and machine translation.
- **LSTM:** Introduced the concept of gates (Write, Read, Forget) to control the flow of information.
- **Language Modeling**: Introduced it with RNNs: how are used for language modeling, emphasizing their ability to capture long-term dependencies.