OPTIMIZATION IN DEEP LEARNING

In machine learning, optimizers and loss functions are two fundamental components that help improve a model's performance.

- A loss function evaluates a model's effectiveness by computing the difference between expected and actual outputs. Common loss functions include log loss, hinge loss, and mean square loss.
- An optimizer improves the model by adjusting its parameters (weights and biases) to minimize the loss function value. Examples include RMSProp, ADAM, and SGD (Stochastic Gradient Descent).

The optimizer's role is to find the best combination of weights and biases that leads to the most accurate predictions.

Gradient Descent
Gradient Descent is a popular optimization method for training machine learning models. It works by iteratively adjusting the model parameters in the direction that minimizes the loss function.

Key Steps in Gradient Descent

1. Initialize parameters: Randomly initialize the model parameters.
2. Compute the gradient: Calculate the gradient (derivative) of the loss function with respect to the parameters.
3. Update parameters: Adjust the parameters by moving in the opposite direction of the gradient, scaled by the learning rate.

Formula :

$$\theta(k+1)=\theta k-\alpha\nabla J(\theta k)$$

Gradient Descent with Armijo Goldstein Condition

This variant ensures that the step size is large enough to effectively reduce the objective function, using a line search that satisfies the Armijo condition.

Condition:

$$f(xt-1+\alpha\bigtriangledown f(xt-1))-f(xt-1)\geq c\alpha||\bigtriangledown f(xt-1)||2$$

Gradient Descent with Armijo Full Relaxation Condition

Incorporates both first and second derivatives (Hessian matrix) to determine a more optimal step size for the update.

Condition:

$$f(xt-1+\alpha\nabla f(xt-1))-f(xt-1)\geq c\alpha||\nabla f(xt-1)||2+b2\alpha2\nabla f(xt-1)TH(x)\nabla f(xt-1)$$

Variants of Gradient Descent

1. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) updates the model parameters after each training example, making it more efficient for large datasets compared to traditional Gradient Descent, which uses the entire dataset for each update.

Steps:

1. Select a training example.
2. Compute the gradient of the loss function.
3. Update the model parameters.

- Advantages: Requires less memory and may find new minima.
- Disadvantages: Noisier, requiring more iterations to converge.

2. Mini Batch Stochastic Gradient Descent

Mini-batch stochastic gradient descent consists of a predetermined number of training examples, smaller than the full dataset. This approach combines the advantages of the previously mentioned variants. In one epoch, following the creation of fixed-size mini-batches, we execute the following steps:

1. Select a mini-batch.
2. Compute the mean gradient of the mini-batch.
3. Apply the mean gradient obtained in step 2 to update the model's weights.
4. Repeat steps 1 to 2 for all the mini-batches that have been created.

- Advantages: Requires medium amount of memory and less time required to converge when compared to SGD
- Disadvantage: May get stuck at local minima

3. SGD with Momentum

Momentum helps accelerate convergence by smoothing out the noisy gradients of SGD, thus reducing fluctuations and improving the speed of convergence.

$$v(t+1)=\beta*vt+(1-\beta)*\nabla J(\theta t)$$

Then, the model parameters are updated using:

$$\theta(t+1)=\theta t-\alpha*v(t+1)$$

- Advantages: Mitigates oscillations, reduces variance, and faster convergence.
- Disadvantages: Requires tuning the momentum coefficient
- β

Advanced Optimizers

1. AdaGrad

AdaGrad adapts the learning rate for each parameter based on the historical gradient information. The learning rate decreases over time, making AdaGrad effective for sparse features.

$$\theta(t+1)=\theta t-\alpha Gt+\varepsilon*\nabla J(\theta t)$$

Where:

- Gt is the sum of squared gradients.
- ε is a small constant to avoid division by zero.
- Advantages: Adapts the learning rate, improving training efficiency.
- Disadvantages: Learning rate decays too quickly, causing slow convergence.

2. RMSProp

RMSProp improves upon AdaGrad by introducing a decay factor to prevent the learning rate from decreasing too rapidly.

$$E[g2]t=\gamma*E[g2](t-1)+(1-\gamma)*(\nabla J(\theta t))2$$

$$\theta(t+1)=\theta t-\alpha E[g2]t+\varepsilon*\nabla J(\theta t)$$

Where:

- γ is the decay rate.
- E[g2]t is the exponentially moving average of squared gradients.
- Advantages: Prevents excessive decay of learning rates.
- Disadvantages: Computationally expensive due to the additional parameter.

3. Adam (Adaptive Moment Estimation)

Adam combines the advantages of Momentum and RMSProp. It uses both the first moment (mean) and second moment (variance) of gradients to adapt the learning rate for each parameter.

1. Update the first moment:
2. $mt=\beta1*m(t-1)+(1-\beta1)*\nabla J(\theta t)$
3. Update the second moment:
4. $vt=\beta2*v(t-1)+(1-\beta2)*(\nabla J(\theta t))2$

Where:

- are the decay rates for the first and second moments.
- ε is a small constant to prevent division by zero.
- Advantages: Fast convergence.
- Disadvantages: Requires significant memory due to the need to store first and second moment estimates.

Optimizer	Advantages	Disadvantages
SGD	Simple, easy to implement	Slow convergence, requires tuning
Mini-Batch SGD	Faster than SGD	Computationally expensive, stuck in local minima
SGD with Momentum	Faster convergence, reduces noise	Requires careful tuning of βββ

AdaGrad	Adaptive learning rates	Decays too fast, slow convergence
RMSProp	Prevents fast decay of learning rates	Computationally expensive
Adam	Fast, combines momentum and RMSProp	Memory-intensive, computationally expensive

Each optimizer has its own strengths and weaknesses. The choice of optimizer depends on the specific problem, dataset characteristics, and the computational resources available. Adam is often the default choice due to its robust performance, but each situation may call for a different optimizer to achieve optimal results.

NON CONVEX OPTIMIZATION IN DEEP NETWORKS

1. What Is Non-Convex Optimization?

A function $f(\theta)$ is **convex** if for all $\theta 1, \theta 2$ and $\lambda \in [0,1]$:

$$f(\lambda\theta 1+(1-\lambda)\theta 2) \leq \lambda f(\theta 1)+(1-\lambda)f(\theta 2)$$

In contrast, non-convex functions violate this inequality, often having:

- Multiple local minima
- Saddle points
- Flat regions

In deep learning, the loss function $L(\theta)$ with respect to parameters θ (weights and biases) is typically non-convex.

2. Why Deep Networks Are Non-Convex

A deep neural network is a composition of functions:

$f(x;\theta)=f(L)(f(L-1)(...f(1)(x;\theta 1)...);\theta L)$

The **loss function** becomes:

$L(\theta)=1n\sum i=1n\ell(f(xi;\theta),yi)$

Where:

- ℓ: Loss function (e.g., cross-entropy, MSE)
- xi,yix_i, y_ixi,yi: Data samples
- θ: All model parameter

Due to nonlinearities and hierarchical composition, $L(\theta)$ is **non-convex**.

3. Optimization Landscape

Deep networks have loss surfaces like:

- **Local minima**: $\nabla L(\theta*)=0$, and $\theta*$ is a minimum in a neighborhood.
- **Saddle points**: $\nabla L(\theta)=0$, but Hessian has both positive and negative eigenvalues.
- **Plateaus**: Gradient is near zero, but not a minimum.

The **Hessian matrix** helps analyze this:

$H=\nabla 2L(\theta)$

Eigenvalues of HHH:

- All positive → local minimum
- All negative → local maximum
- Mixed → saddle point

4. Gradient-Based Methods in Non-Convex Optimization

Despite non-convexity, we use **gradient descent** and its variants:

Gradient Descent (GD):

$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$

- η \eta: Learning rate
- $\nabla L(\theta)$: Gradient

Stochastic Gradient Descent (SGD):

$\theta_{t+1} = \theta_t - \eta \nabla L$

Use minibatches to estimate the gradient. Noise helps escape saddle points.

Momentum:

$v_{t+1} = \gamma v_t + \eta \nabla L(\theta_t) \theta_{t+1} = \theta_t - v_{t+1}$

- Helps move through flat/saddle regions.

5. Strategies for Handling Non-Convexity

Strategy	Description
Overparameterization	Makes the loss surface smoother; many local minima become global-like.
Batch Normalization	Stabilizes training; reduces sharp curvature.
Adaptive optimizers	Like Adam, RMSProp adjust learning rates dynamically.
Initialization	Good weight initialization avoids bad minima (e.g., Xavier, He init).
Loss smoothing	Adding regularization helps (e.g., $\lambda \|\theta\|^2$ \lambda |\theta|^2).

6. Empirical Observation

Despite non-convexity:

- Many local minima generalize well.
- **Flat minima** often yield better generalization than **sharp minima**.

7. Summary Formula Table

Concept	Formula
Loss Function	$L(\theta)=1n\sum i=1n\ell(f(xi;\theta),yi)$
Gradient Descent	$\theta t+1=\theta t-\eta\nabla L(\theta t)$
Hessian Matrix	$H=\nabla 2L(\theta)$
Saddle Point (example)	$L(\theta)=x2-y2\Rightarrow\nabla L=(2x,-2y)$

STOCHASTIC OPTIMIZATION AND GENERALISATION IN NEURAL NETWORK

Stochastic optimization plays a crucial role in training deep neural networks, especially in achieving good **generalization** — the ability of the model to perform well on **unseen data**. Below is a **detailed explanation** with **formulas and insights** into how stochastic optimization affects generalization.

1. What Is Stochastic Optimization?

Stochastic Optimization (like **SGD**) approximates the full-batch gradient using small batches (minibatches):

$\theta t+1=\theta t-\eta\nabla\theta LB(\theta t)$

Where:

- θ: Model parameters
- η: Learning rate
- LB(θt): Empirical loss on minibatch BBB

The stochastic nature introduces **noise**, helping the model:

- Escape sharp local minima or saddle points
- Explore flatter regions that often **generalize better**

2. Generalization: Formal Definition

Given:

- Training loss:
  L^(θ)=n1i=1∑nℓ(f(xi;θ),yi)
- True (expected) loss:
  L(θ)=E(x,y)∼D[ℓ(f(x;θ),y)]

Then the **generalization gap** is:

GenGap(θ)=L(θ)−L^(θ)

Good generalization implies a **small generalization gap**.

3. Why SGD Helps Generalization

A. Implicit Regularization

- Unlike full-batch gradient descent, **SGD does not converge exactly** to the minimum of training loss.
- The noise in updates **acts like a regularizer**, favoring **flatter minima** which are often associated with better generalization.

> Flat minima are defined as points in parameter space where small perturbations to weights do not significantly increase the loss.

B. Escaping Sharp Minima

A **sharp minimum** has a steep loss landscape (large Hessian eigenvalues), leading to poor generalization.

Flat minimum example:

$\nabla 2L(\theta)$ has small eigenvalues

SGD's noise helps the model **avoid sharp minima** by not "settling" too precisely.

Property	SGD	Full-Batch Gradient Descent
Update Type	Noisy, per mini-batch	Deterministic, all data
Exploration	High	Low
Regularization Effect	Implicit	Minimal
Generalization	Better	Often worse

5. SGD Generalization Theories

A. Noise-Induced Generalization

Noise in SGD behaves like **annealed Langevin dynamics**:

$\theta t+1=\theta t-\eta\nabla\theta LB(\theta t)+2\eta T\cdot N(0,I)$

This form resembles **stochastic differential equations**, where TTT is a "temperature" parameter.

B. PAC-Bayes Bounds

Some generalization bounds (e.g., PAC-Bayes) show that **flat minima** imply tighter bounds on generalization error.

6. Practical Techniques That Help Generalization

Technique	Purpose
Dropout	Adds noise; prevents co-adaptation

Data Augmentation	Increases sample diversity
Weight Decay	Penalizes large weights
Early Stopping	Prevents overfitting
Batch Normalization	Stabilizes training, smooths landscape

7. Visualization Insight

In a 2D loss surface:

- **Sharp minima** look like steep pits.
- **Flat minima** resemble wide valleys.

SGD tends to land in **wide valleys**, helping the model generalize well.

SUMMARY

Aspect	Explanation
Stochastic Optimization	Uses randomness to update weights, mainly via mini-batch gradients
Generalization	Ability to perform well on unseen data
Key Benefit	SGD’s noise drives the model toward flat minima that generalize better

Theoretical Support	Langevin dynamics, PAC-Bayes bounds, and Hessian spectrum analysis

SPATIAL TRANSFORMER NETWORKS

Spatial Transformer Networks (STNs) are neural network modules that allow the network to learn spatial transformations of the input data during training. Introduced by Jaderberg et al., 2015, STNs are differentiable modules that can be inserted into convolutional networks to provide spatial invariance.

1. Why Use STNs?

Traditional CNNs are limited in their ability to handle geometric variations such as:

- Rotation
- Scaling
- Translation
- Perspective changes

STNs learn to spatially transform feature maps or input images to canonical forms, improving recognition accuracy and robustness.

2. STN Architecture Overview

An STN has three key components:

A. Localization Network

- Learns the parameters θ\thetaθ of a spatial transformation.
- Takes input feature map $U \in R^{H \times W \times C}$
- Outputs parameters for transformation, e.g., affine:

$$\theta = \text{LocalizationNet}(U)$$

B. Grid Generator

- Uses θ to produce a sampling grid of coordinates in the input image.
- For an affine transformation:

$$[x_s y_s] = A[x_t y_t 1], \text{where } A \in R^{2\times 3}$$

Here, (xt,yt)(x_t, y_t)(xt,yt) are target coordinates and (xs,ys)(x_s, y_s)(xs,ys) are the source coordinates.

C. Sampler

- Samples the input feature map at grid locations using bilinear interpolation:

$$V_i^c = \sum_n^H \sum_m^W U_{nm}^c \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - n|)$$

Where:

- UUU is the input feature map
- VVV is the output transformed feature map

3. Affine Transformation Matrix

The affine transform matrix used by the grid generator is:

$$A = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix}$$

This can apply:

- Translation
- Rotation
- Scaling
- Shearing

4. Integration in CNNs

STNs can be plugged anywhere in a CNN pipeline:

Input → STN → Conv Layers → Classifier

The transformation is learned via backpropagation—since all parts are differentiable.

Feature	Description
Learned Invariance	Learns to correct geometric distortions
Modularity	Can be added into existing architectures
End-to-End Training	Differentiable and trainable with rest of network

Better Performance	Especially on spatially distorted datasets (e.g., MNIST variants)

6. Use Cases

- Handwritten digit recognition (rotated MNIST)
- Scene text recognition
- Fine-grained image classification
- Object detection & localization

RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNNs) are a class of neural networks designed to model sequential data such as time series, speech, or text. Unlike feedforward networks, RNNs have cyclic connections, allowing them to maintain a memory of previous inputs using hidden states.

1. Basic RNN Architecture

An RNN processes a sequence one step at a time, maintaining a hidden state that captures information from previous time steps.

Forward Pass Equations:

Given:

- Input at time step ttt: xt
- Hidden state at time step ttt: ht
- Output at time step ttt: yt

The basic RNN equations:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Where:

- Wxh: input-to-hidden weights

- Whh: hidden-to-hidden weights
- Why: hidden-to-output weights
- bh,by: biases

2. Unrolling an RNN

An RNN over time can be visualized as a unrolled network:

x_1 → [h_1] → y_1

x_2 → [h_2] → y_2

x_3 → [h_3] → y_3

↑

(hidden state flows through time)

The hidden state at each step depends on both the current input and the previous hidden state.

3. Loss Function

For a sequence of length TTT, the total loss is often the sum over time steps:

$L=\sum t=1T\ell(yt,y^t)$

Where y^t is the target at time step ttt, and ℓ\ell$\ell$ is a loss function like cross-entropy or MSE.

4. Backpropagation Through Time (BPTT)

To train an RNN, gradients are computed over all time steps using Backpropagation Through Time (BPTT).

$\partial W\partial L=t=1\sum T\partial ht\partial L\partial W\partial ht$

BPTT can suffer from:

- Vanishing gradients (when gradients shrink)
- Exploding gradients (when gradients grow too large)

5. Variants of RNNs

A. LSTM (Long Short-Term Memory)

B. GRU (Gated Recurrent Unit)

6. Applications of RNNs

Task	Input	Output
Language Modeling	Text sequence	Next word
Sentiment Analysis	Sentence	Sentiment label
Machine Translation	Sentence	Translated sentence
Speech Recognition	Audio sequence	Text transcript
Time Series Forecasting	Past data	Future values

7. Limitations of Basic RNNs

Problem	Explanation
Vanishing gradients	Long-term dependencies are hard to learn
Limited memory	Only short-term memory without LSTM/GRU
Slow training	Sequential processing limits parallelism

LSTM

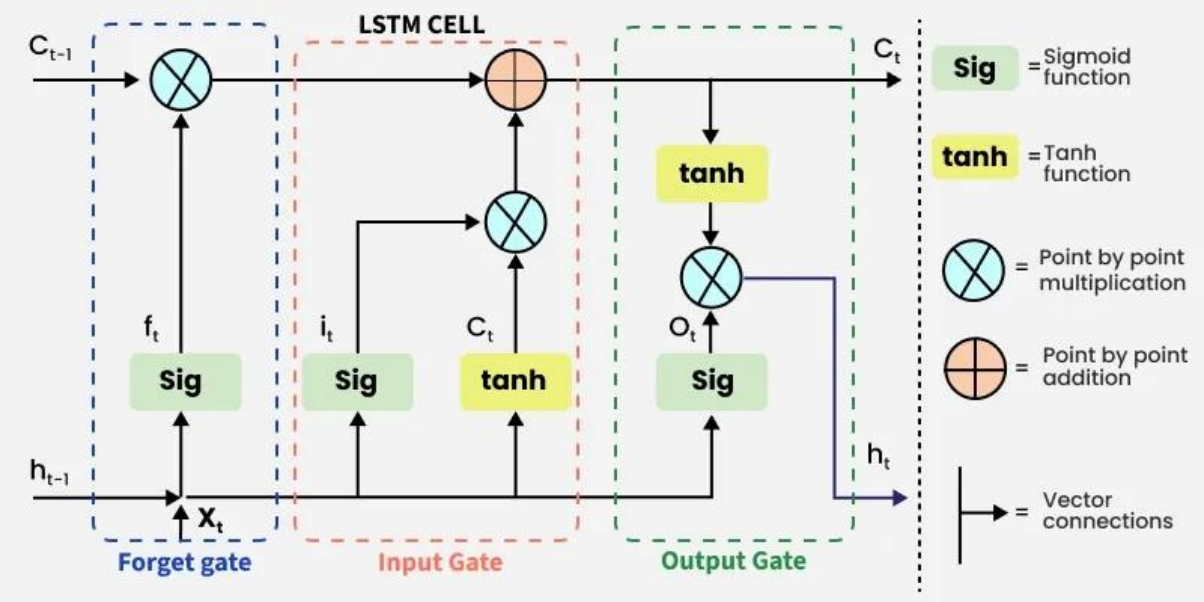
LSTM architectures involves the memory cell which is controlled by three gates: the *input gate*, the *forget gate* and the *output gate*. These gates decide what information to add to, remove from and output from the memory cell.

- Input gate: Controls what information is added to the memory cell.

- Forget gate: Determines what information is removed from the memory cell.
- Output gate: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Working of LSTM
LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



LSTM Model

Information is retained by the cells and the memory manipulations are done by the gates. There are three gates –

Forget Gate

The information that is no longer useful in the cell state is removed with the forget gate. Two inputs *xt* (input at the particular time) and *ht-1* (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.

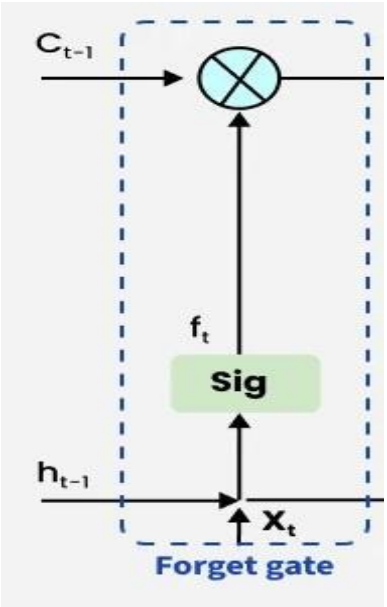
The equation for the forget gate is:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where:

- W_f represents the weight matrix associated with the forget gate.

- [h_t-1, x_t] denotes the concatenation of the current input and the previous hidden state.
- b_f is the bias with the forget gate.
- σ is the sigmoid activation function.



Forget Gate

Input gate

The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs *ht-1* and *xt*. . Then, a vector is created using *tanh* function that gives an output from -1 to +1, which contains all the possible values from ht-1 and *xt*. At last, the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

$$it=\sigma(Wi\cdot[ht-1,xt]+bi)$$

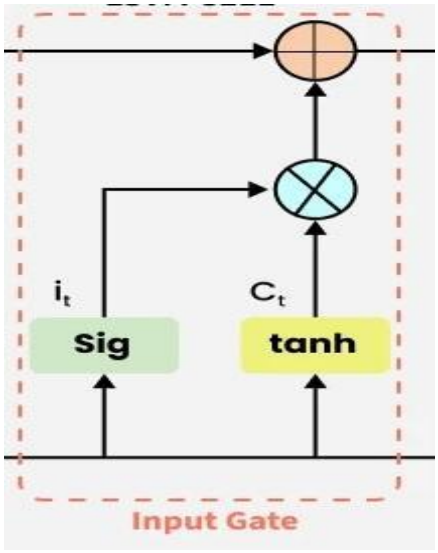
$$\hat{C}t=\tanh(Wc\cdot[ht-1,xt]+bc)$$

We multiply the previous state by ft, disregarding the information we had previously chosen to ignore. Next, we include it∗Ct. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

$$Ct=ft\odot Ct-1+it\odot \hat{C}t$$

where

- ⦿ denotes element−wise multiplication
- tanh is tanh activation function



Input Gate

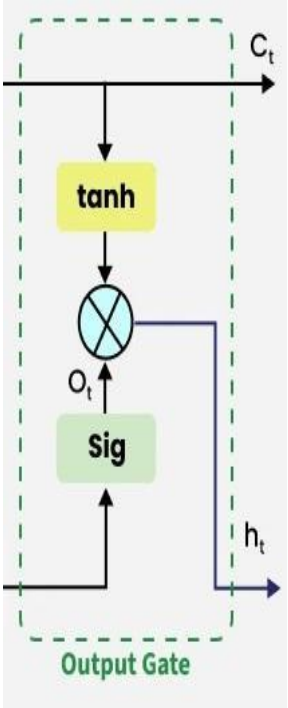
Output gate

The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs

Ht−1 and xt

. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell. The equation for the output gate is:

ot=σ(Wo·[ht−1,xt]+bo)



Output Gate

Bidirectional LSTM Model

Bidirectional LSTM (Bi LSTM/ BLSTM) is a variation of normal LSTM which processes sequential data in both forward and backward directions. This allows Bi LSTM to learn longer-range dependencies in sequential data than traditional LSTMs which can only process sequential data in one direction.

- Bi LSTMs are made up of two LSTM networks one that processes the input sequence in the forward direction and one that processes the input sequence in the backward direction.
- The outputs of the two LSTM networks are then combined to produce the final output.

LSTM models including Bi LSTMs have demonstrated state-of-the-art performance across various tasks such as machine translation, speech recognition and text summarization.

LSTM networks can be stacked to form deeper models allowing them to learn more complex patterns in data. Each layer in the stack captures different levels of information and time-based relationships in the input.

Applications of LSTM

Some of the famous applications of LSTM includes:

- Language Modeling: Used in tasks like language modeling, machine translation and text summarization. These networks learn the dependencies between words in a sentence to generate coherent and grammatically correct sentences.

- Speech Recognition: Used in transcribing speech to text and recognizing spoken commands. By learning speech patterns they can match spoken words to corresponding text.
- Time Series Forecasting: Used for predicting stock prices, weather and energy consumption. They learn patterns in time series data to predict future events.
- Anomaly Detection: Used for detecting fraud or network intrusions. These networks can identify patterns in data that deviate drastically and flag them as potential anomalies.
- Recommender Systems: In recommendation tasks like suggesting movies, music and books. They learn user behavior patterns to provide personalized suggestions.
- Video Analysis: Applied in tasks such as object detection, activity recognition and action classification. When combined with Convolutional Neural Networks (CNNs) they help analyze video data and extract useful information.

Recurrent Neural Network Language Models

An RNN-LM extends the idea of RNNs to language modeling. It processes input sequences one word at a time, updating its hidden state at each step based on the current word and the information stored in the hidden state from the previous steps. This enables the model to capture contextual information and dependencies within a sequence of words, making it adept at tasks like text generation, sentiment analysis, and machine translation.

Architecture

The architecture of an RNN-LM typically consists of three main components:

1.Embedding Layer

The Embedding Layer is the initial component of an RNN Language Model, responsible for transforming individual words into continuous, dense vector representations. Each word in the vocabulary is assigned a unique vector, and these vectors are learned during the training process. This embedding process allows the model to capture semantic relationships between words and helps in understanding the context of a given word within a sequence.

Suppose we have a vocabulary with three words: “cat,” “dog,” and “fish.” The embedding layer might assign the following vectors:

- Embedding(“cat”) = [0.2, 0.8, 0.5]
- Embedding(“dog”) = [0.7, 0.3, 0.2]
- Embedding(“fish”) = [0.9, 0.5, 0.1]

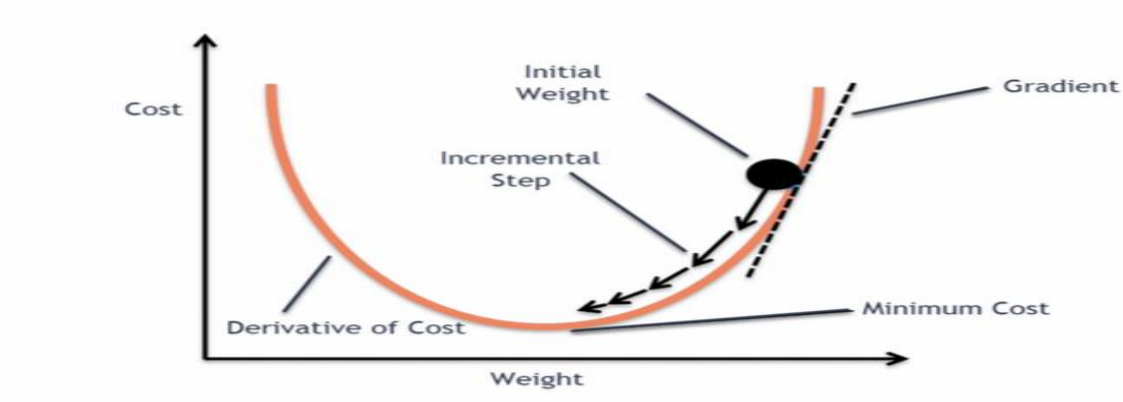
The values assigned to the word vectors in the embedding layer are learned during the training process of the neural network. The goal is to find vector representations that capture meaningful semantic relationships between words based on the context in which they appear.

Here’s a simplified explanation of how these values might be calculated:

1. **Initialization:**
   - Initially, the word vectors are randomly initialized with small values.

2.Objective Function:

- The neural network has an objective function (often a loss function) that quantifies how well the model is performing on a given task (e.g., language modeling, sentiment analysis).



Gradient Descent

3.Backpropagation:

- During training, the model processes input sequences, computes predictions, and compares them to the actual target values using the objective function.

4.Gradient Descent:

- The gradients of the objective function with respect to the parameters (including word vectors) are computed through backpropagation

5.Parameter Updates:

- The model adjusts the word vectors using optimization algorithms like gradient descent to minimize the objective function.

For each word in the vocabulary (e.g., “cat,” “dog,” “fish”), the corresponding word vector is updated based on how it contributes to the overall performance of the model on the given task.

Let’s consider an oversimplified example:

```
New Embedding(“cat”)=Old Embedding(“cat”)−Learning
Rate×Gradient(Objective Function with respect to “cat”)
```

This process is repeated iteratively for multiple epochs until the model converges to representations that effectively capture the semantics and context of the words.

2.Recurrent Layer

The Recurrent Layer is the core of the RNN Language Model, responsible for capturing sequential dependencies in the input data. It maintains a hidden state that evolves over time as the model processes each word in a sequence. This hidden state retains information about the context of previous words, enabling the model to consider the entire input sequence.

How it works:

Hidden State:

- The hidden state is updated at each time step based on the current input word and the previous hidden state.
- Captures information about the context of the sequence.
- RNNs can theoretically capture long-term dependencies by allowing information to persist in the hidden state throughout the sequence.

Example:

Given the word embeddings from the embedding layer, the recurrent layer updates the hidden state at each time step, incorporating information from previous steps.

$$ht = f(Whh \cdot ht-1 + Wxh \cdot xt)$$

- *ht* is the hidden state at time *t*.
- *Whh* and *Wxh* are weight matrices.
- *ht*`−1 is the previous hidden state.`
- *xt* is the input vector at time *t*.

3.Output Layer

The Output Layer is the final component of the RNN Language Model, responsible for producing the probability distribution over the vocabulary. It takes the information from the hidden state and generates a probability distribution, allowing the model to predict the likelihood of the next word in the sequence.

How it works:

- Softmax Activation:
- The output layer typically uses a softmax activation function to convert raw scores into probabilities.Produces a probability distribution over the entire vocabulary.

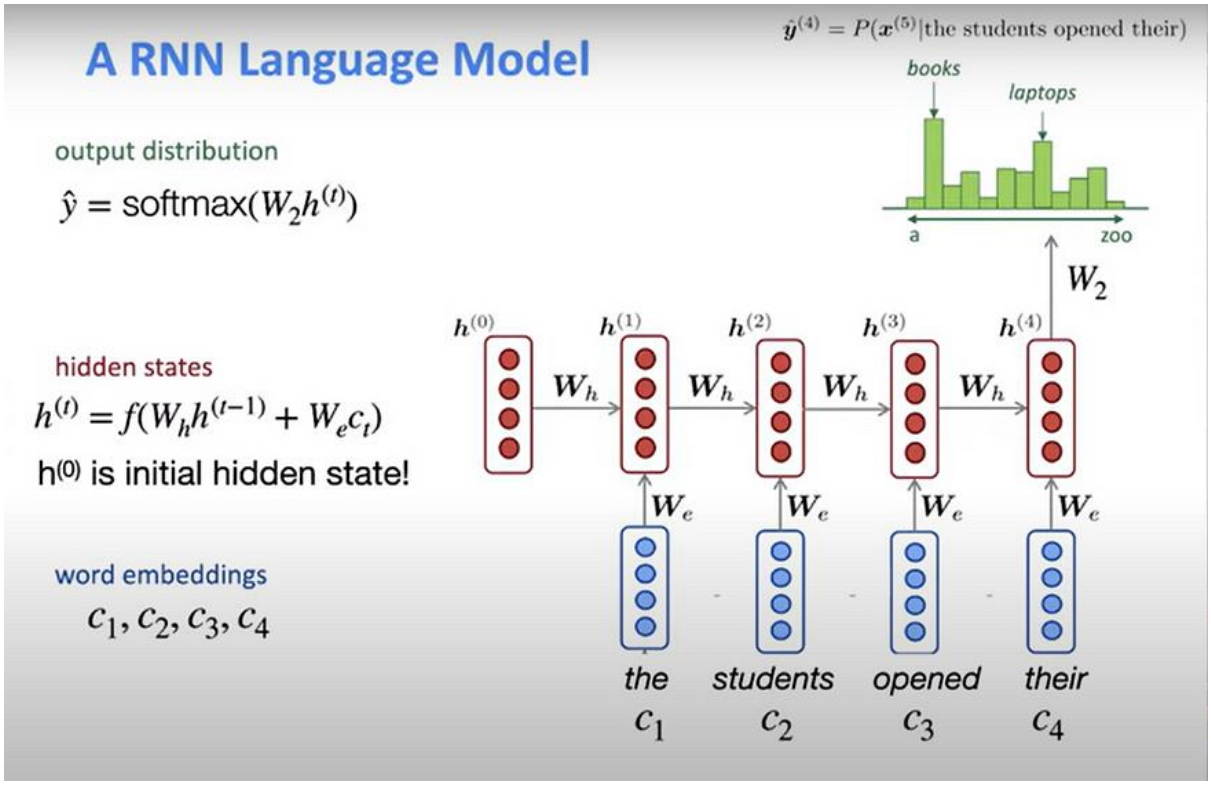
Example:

Using the hidden state *ht* from the recurrent layer, the output layer calculates the unnormalized scores *ut* for each word in the vocabulary.

- $ut = Who \cdot ht$
- The softmax activation then converts these scores into probabilities.

$$P(y_t|x_{1:t}) = \frac{\exp(u_t)}{\sum_i \exp(u_i)}$$

- $P(yt \mid x1{:}t)$ is the probability of the next word *yt* given the input sequence $x1{:}t$.



WORD-LEVEL RNNs & DEEP REINFORCEMENT LEARNING

Part 1: Word-Level Recurrent Neural Networks (RNNs)

What Are They?

Word-level RNNs treat words as atomic units rather than characters. They're commonly used in language modeling, text generation, and translation, where the input/output is a sequence of words.

1. Input Representation

- Each word is converted into a one-hot vector or more efficiently, an embedding vector:

xt=Embedding(wt)

Where:

- wtw_twt: word at time step ttt
- $x_t \in \mathbb{R}^d$: word embedding

2. RNN Forward Pass

At each time step:

$$h_t = \tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h)$$

$$y_t = \text{softmax}(W_{hy} h_t + b_y)$$

- y_t is a probability distribution over the vocabulary
- Used to predict the next word $\hat{w}_{t+1}$

3. Loss Function

$$L = -\sum_{t=1} T \log P(w_{t+1} | w_1, \cdots, w_t)$$

- Trained using cross-entropy loss over predicted vs. actual words

4. Use Cases

- Text generation
- Machine translation (e.g., seq2seq with attention)
- Speech recognition
- Next-word prediction in language models (GPT-style models use transformer variants)

Part 2: Deep Reinforcement Learning (Deep RL)

What Is It?

Deep Reinforcement Learning combines Reinforcement Learning (RL) with deep neural networks to make decisions in complex environments.

1. Reinforcement Learning Basics

- Agent: Learns to take actions
- Environment: Gives feedback
- State s_t
- Action a_t
- Reward r_t
- Policy $\pi(a_t | s_t)$
- Return $R_t = \sum_{k=0} \infty \gamma^k r_{t+K}$

Goal:
Maximize expected cumulative reward:

$$J(\theta) = E_{\pi\theta}[\sum_{t=0} T \gamma^t r_t]$$

2. Deep Q-Networks (DQN)

Used for discrete action spaces. Approximate the Q-value:

$Q(st,at)=rt+\gamma max_{a'}Q(st+1,a')Q$

Use a neural network $Q(s,a;\theta)Q(s, a; \theta)Q(s,a;\theta)$ and train using temporal difference (TD) loss:

$L(\theta)=(rt+\gamma max_{a'}Q(st+1,a';\theta-)-Q$

Where θ is the target network's parameter (updated less frequently).

3. Policy Gradient Methods

Directly optimize the policy $\pi\theta(a|s)\pi_\theta(a \mid s)\pi\theta(a|s)$ via gradient ascent:

$\nabla\theta J(\theta)=E\pi\theta[\nabla\theta log\pi\theta(at|st)Rt]$

Popular algorithms:

- REINFORCE
- A3C (Asynchronous Advantage Actor-Critic)
- PPO (Proximal Policy Optimization)
- DDPG, SAC (for continuous actions)

4. Deep Networks in RL

Task	Deep RL Use
Atari games (pixels → actions)	DQN, Dueling DQN
Robotics control	DDPG, PPO, SAC
Text-based games	RNNs + Policy Gradients
Multi-agent systems	MADDPG, QMIX

Combining RNNs and RL

In environments with partial observability (like dialogue systems or text-based games), RNNs are used to encode history:

$$h_t = RNN(h_{t-1}, o_t)$$

$$\pi(a_t|h_t) = PolicyNetwork(h_t)$$

This enables the agent to remember past information and act accordingly.

COMPUTATIONAL AND ARTIFICIAL NEUROSCIENCE

Computational Neuroscience

Computational neuroscience is the science of using mathematical models, computer simulations, and theoretical analysis to understand how the brain and nervous system work. It tries to bridge biology and computation by modeling how neurons, synapses, and neural circuits behave.

Key Concepts:

1. Neuron Modeling:
   Scientists model how real neurons process signals. This includes how a neuron responds to electrical input and how it decides to send a signal to another neuron (a spike).

2. Synaptic Transmission:
   The connection between neurons is called a synapse. Computational neuroscience models how signals are transmitted and modified at these synapses.

3. Plasticity:
   This refers to how the strength of connections between neurons changes over time, which is essential for learning and memory.

4. Neural Coding:
   Researchers try to understand how the brain represents information. For example, does it encode by firing faster, in a pattern, or across multiple neurons?

5. Simulations:
   Using tools like NEURON or NEST, researchers simulate parts of the brain to study how groups of neurons behave and interact.

Artificial Neuroscience (Artificial Neural Networks)

Artificial neuroscience, also known as artificial neural networks (ANNs), draws inspiration from how biological brains work but simplifies and adapts it for machine learning tasks like image recognition, language translation, and robotics.

Key Concepts:

1. Artificial Neurons:
   These are simplified models of biological neurons. They take in inputs, do some computation, and produce an output—just like real neurons.

2. Network Structure:
   Artificial neurons are connected in layers. There are input layers (which receive data), hidden layers (which process data), and output layers (which make predictions or decisions).

3. Learning Process:
   Instead of changing based on chemical or electrical signals like the brain, artificial networks learn by adjusting internal settings (weights) through a process called training.

4. Types of Networks:
   - Feedforward Networks: Data flows one way. Used for basic tasks.
   - Convolutional Neural Networks (CNNs): Great for image recognition.
   - Recurrent Neural Networks (RNNs): Handle sequences, like speech or text.
   - Transformers: Modern networks used in models like ChatGPT.

5. Differences from the Brain:
   Artificial networks are not biologically accurate. They don't model spikes or complex chemical processes. They're just inspired by the brain's structure and function.
   Bridging Brain and Machine

While computational neuroscience tries to understand the brain, artificial neuroscience tries to build smart systems inspired by the brain. The two fields increasingly overlap in areas like:

- Spiking Neural Networks (SNNs): Artificial models that try to mimic the spiking behavior of real neurons.
- Neuromorphic Computing: Special hardware designed to behave more like the brain.
- Brain-Computer Interfaces (BCIs): Systems that let brains interact with machines.