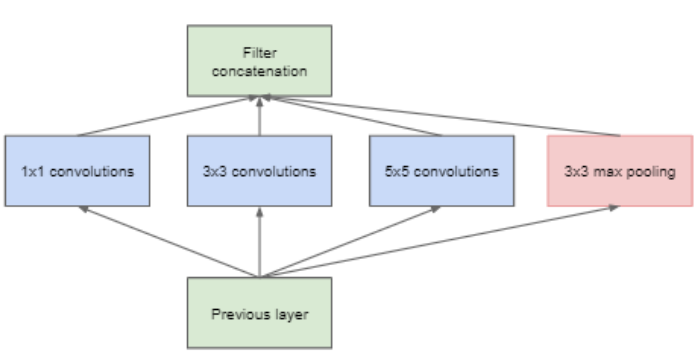
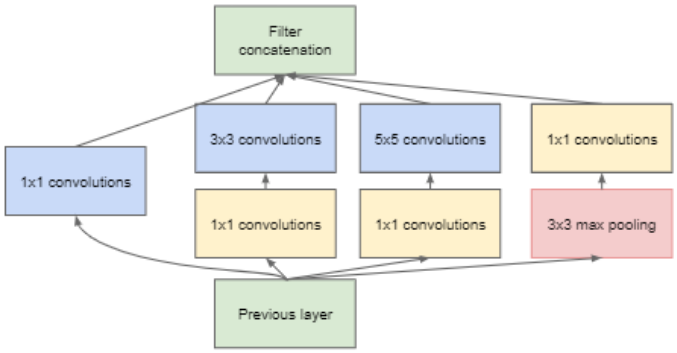
Unit-3

1 Draw the Architecture of GoogleNet and identify the kernel sizes used in GoogleNet?



(a) Inception module, naïve version



(b) Inception module with dimension reductions

1. **GoogleNet (Inception V1) Architecture:**
   - GoogleNet, also known as Inception V1, was proposed by researchers at Google in 2014. It won the **ILSVRC 2014** image classification challenge and significantly outperformed previous architectures like AlexNet and VGG.
   - Key features of GoogleNet:
     - **1×1 Convolutions**: GoogleNet uses 1×1 convolutions to reduce the number of parameters and increase the depth of the architecture.
     - **Global Average Pooling**: Instead of fully connected layers, it employs global average pooling, reducing trainable parameters and improving accuracy.
     - **Inception Module**: The core building block, combining multiple convolutional filters of different sizes.
     - **Auxiliary Classifiers**: Intermediate classifiers used during training to improve learning.
   - The architecture is 22 layers deep and can process RGB images of size **224×224**.
2. **Inception Module**:
   - The Inception module performs parallel convolutions with different kernel sizes and concatenates their outputs.
   - It includes:
     - 1×1 convolutions
     - 3×3 convolutions
     - 5×5 convolutions
     - 3×3 max pooling
   - The idea is that filters of different sizes handle objects at multiple scales better.
3. **Kernel Sizes**:
   - The kernel sizes used in the Inception module are:
     - 1×1 convolutions
     - 3×3 convolutions
     - 5×5 convolutions
4. **Answer for 8 Marks (Detailed)**:
   - GoogleNet, also known as Inception V1, revolutionized deep learning architectures. It introduced several key innovations:

        - **1×1 Convolutions**: These reduce parameters while increasing depth. For example, a 5×5 convolution with 48 filters can be replaced by a 1×1 convolution with 16 filters followed by a 5×5 convolution with 48 filters, significantly reducing computation.
        - **Global Average Pooling**: Instead of fully connected layers, GoogleNet uses global average pooling. It takes a 7×7 feature map and averages it to 1×1, reducing parameters and improving accuracy.
        - **Inception Module**: The heart of GoogleNet, it combines parallel convolutions of different sizes (1×1, 3×3, 5×5) and max pooling. This allows the network to handle objects at various scales.
        - **Auxiliary Classifiers**: These intermediate classifiers are used during training. They consist of average pooling, 1×1 convolutions, fully connected layers, and softmax classification. They help regularize the network.
    - The architecture processes RGB images of size 224×224 and achieved a top-5 error rate of 6.67% in the ILSVRC 2014 challenge.
    - In summary, GoogleNet's innovative design, including 1×1 convolutions, global average pooling, and the Inception module, significantly improved image classification performance.

2.Distinguish the features learned by the 1x1 convolutions and the other traditional convolutions of kernel size greater than or equal to 2.

Distinguishing Features Learned by 1x1 Convolutions vs. Traditional Convolutions

1. 1x1 Convolutions:

- **Purpose**:
    - **Dimensionality Reduction**: 1x1 convolutions reduce the number of feature maps while retaining essential features.
    - **Efficient Embeddings**: They create efficient low-dimensional embeddings.
    - **Non-Linearity**: Can apply non-linearity after other convolutions.
- **Operation**:
    - Operates on individual pixels of the input image.
    - Applies a linear transformation independently across channels.
    - Acts like channel-wise pooling.
    - Can increase or decrease the number of feature maps.
- **Use Cases**:
    - **Network Compression**: Reducing model complexity.
    - **Transition Layers**: Used between convolutional blocks.
    - **Bottleneck Layers**: Common in architectures.
- **Example**: GoogleNet (Inception V1) effectively employs 1x1 convolutions.
- **Google net diagram**

2. Traditional Convolutions (Kernel Size ≥ 2):

- **Purpose**: Capture spatial patterns and hierarchical features.
- **Operation**:
    - Slides a larger filter (e.g., 3x3, 5x5) across the input.
    - Considers spatial neighborhoods.

    - Learns complex features.
- **Use Cases**:
    - **Feature Extraction**: Capturing local patterns (edges, textures).
    - **Hierarchical Features**: Combining low-level features.
    - **Spatial Invariance**: Handling variations.
- **Examples**: VGGNet, ResNet, etc., rely on traditional convolutions.
- **Resnet diagram** …

Summary:

1x1 convolutions manage complexity, while traditional convolutions handle spatial features and hierarchies. Both are crucial in deep learning architectures.

3.What are skip connections in CNNs? Explain how ResNet has exploited skip connections in reducing Top 5% error in ImageNet classification.

Skip Connections (Residual Connections)

Skip connections, also known as **residual connections**, play a crucial role in training very deep neural networks. Here's an in-depth explanation:

1. **Degradation Problem**:
    - When we increase the depth of neural networks, their performance often **degrades**. This phenomenon is known as the **degradation problem**.
    - Surprisingly, this degradation occurs even when we add more layers to the network without overfitting or gradient issues.
2. **Vanishing Gradient and Identity Mappings**:
    - The vanishing gradient problem can hinder the training of deep networks. Even with proper weight initialization and batch normalization, deeper layers struggle to learn meaningful representations.
    - Consider a deep neural network with an initial shallow part (blue layers) followed by additional layers (green layers). If we set the weights of the added layers as **identity mappings**, the deeper network should perform at least as well as the shallow one.
    - However, experiments show that deeper networks produce higher training error, indicating that not all systems are equally easy to optimize.
    - The issue arises due to random weight initialization and regularization (L1, L2), which cause the weights to hover around zero. Consequently, deeper layers fail to learn identity mappings effectively.
3. **Skip Connections to the Rescue**:
    - ResNet introduces **skip connections** (also called **residual connections**).
    - The idea is to add **shortcut connections** that skip one or more layers, allowing the network to learn the **residual** (difference) between the input and output of those layers.
    - Instead of directly learning the desired underlying mapping, ResNet learns the **difference** between the input and output. This enables the network to handle very deep architectures.
4. **Architecture of ResNet**:
    - ResNet-50, introduced by Microsoft Research, is a popular architecture.
    - It consists of **residual blocks**, each containing multiple convolutional layers.
    - Within each residual block, a **skip connection** adds the input directly to the output after passing through the convolutional layers.
    - The residual block structure allows gradients to flow more easily during backpropagation, mitigating the vanishing gradient problem.

5. **Benefits of Skip Connections**:
    - **Vanishing Gradient Mitigation**: Skip connections ensure that gradients can flow directly from the input to the output, bypassing intermediate layers. This facilitates training of very deep networks.
    - **Efficient Learning of Residuals**: By learning residuals, the network focuses on capturing the deviations from the identity mapping. This enables efficient training and better generalization.
    - **State-of-the-Art Performance**: ResNet-50 and its variants have achieved remarkable results in various image-related tasks, including object detection, image classification, and segmentation.

In summary, skip connections in ResNet allow the network to learn residual mappings effectively, leading to improved performance and reduced error rates. ResNet's success lies in its ability to handle deep architectures while maintaining gradient flow and stability.

---

4. What modifications are necessary to CNNs used for image classification, so that they work well for semantic segmentation also?

Certainly! Let's explore the necessary modifications to adapt Convolutional Neural Networks (CNNs) from image classification to semantic segmentation. I'll provide a detailed answer to help you impress the examiner.

Adapting CNNs for Semantic Segmentation

Semantic segmentation involves assigning a class label to each pixel in an image, enabling fine-grained understanding of object boundaries and regions. To make CNNs work effectively for semantic segmentation, we need specific modifications:

1. **Transition from Classification to Segmentation**:
    - In image classification, CNNs predict a single label for the entire image. For segmentation, we need per-pixel predictions.
    - Modify the last layer(s) of the CNN to output pixel-wise class probabilities or logits.
2. **Spatial Information Preservation**:
    - CNNs typically downsample feature maps to reduce spatial dimensions. However, for segmentation, we need precise localization.
    - Use **deconvolutional layers** (also known as **transposed convolution** or **upsampling layers**) to upsample feature maps back to the original resolution.
    - These layers reverse the downsampling process, allowing the network to recover spatial details.
3. **Fully Convolutional Networks (FCNs)**:
    - FCNs are designed specifically for semantic segmentation.
    - Replace fully connected layers with convolutional layers.
    - FCNs maintain spatial information throughout the network, enabling pixel-wise predictions.
4. **Skip Connections (U-Net Architecture)**:
    - To capture both local and global context, incorporate skip connections.
    - The **U-Net architecture** uses an encoder-decoder structure with skip connections.
    - Skip connections connect corresponding layers in the encoder and decoder, allowing the network to combine high-level features with fine-grained details.
5. **Dilated (Atrous) Convolutions**:
    - Dilated convolutions increase the receptive field without downsampling.
    - They allow the network to capture context at multiple scales.
    - Use dilated convolutions in intermediate layers to improve segmentation performance.
6. **Class Imbalance Handling**:
    - In semantic segmentation, some classes may be rare (e.g., small objects).
    - Address class imbalance by using weighted loss functions or oversampling minority classes during training.

7. **Post-processing Techniques**:
    - After obtaining segmentation masks, apply post-processing steps:
        - **Crisp Boundary Refinement**: Smooth boundaries to avoid jagged edges.
        - **Conditional Random Fields (CRFs)**: Refine predictions based on spatial coherence.
        - **Connected Component Analysis**: Group adjacent pixels with the same label.
8. **Transfer Learning and Pretrained Models**:
    - Fine-tune CNNs pretrained on large image classification datasets (e.g., ImageNet).
    - Transfer learned features can benefit semantic segmentation tasks.

Example Architectures:

- **FCN**: Replace fully connected layers with 1x1 convolutions to maintain spatial information.
- **U-Net**: Encoder-decoder architecture with skip connections.
- **DeepLab**: Combines dilated convolutions and CRFs for accurate segmentation.

5.What makes U-Net different from regular CNN? Explain in detail.

Aspect	Regular CNN	U-Net
Architecture	- Multiple layers (convolutional, pooling, fully connected) - Encoder-decoder architecture (no specific symmetry).	- Symmetric architecture with encoder (contracting path) and decoder (expansive path). - Explicit skip connections.
Purpose	- General-purpose (classification, detection). - Not optimized for precise segmentation.	- Primarily for **semantic segmentation** (pixel-wise labeling).
Input/Output	- Single output (e.g., class probabilities).	- Pixel-wise segmentation map (per-class labels).
Skip Connections	- None. Information flows sequentially.	- Utilizes skip connections for preserving fine details.
Contextual Info	- Local context within receptive fields.	- Combines local and global context (due to skip connections).
Training Data	- Requires large labeled data.	- Effective with limited data (due to skip connections).
Loss Function	- Cross-entropy loss.	- Often uses Dice coefficient or Jaccard index.

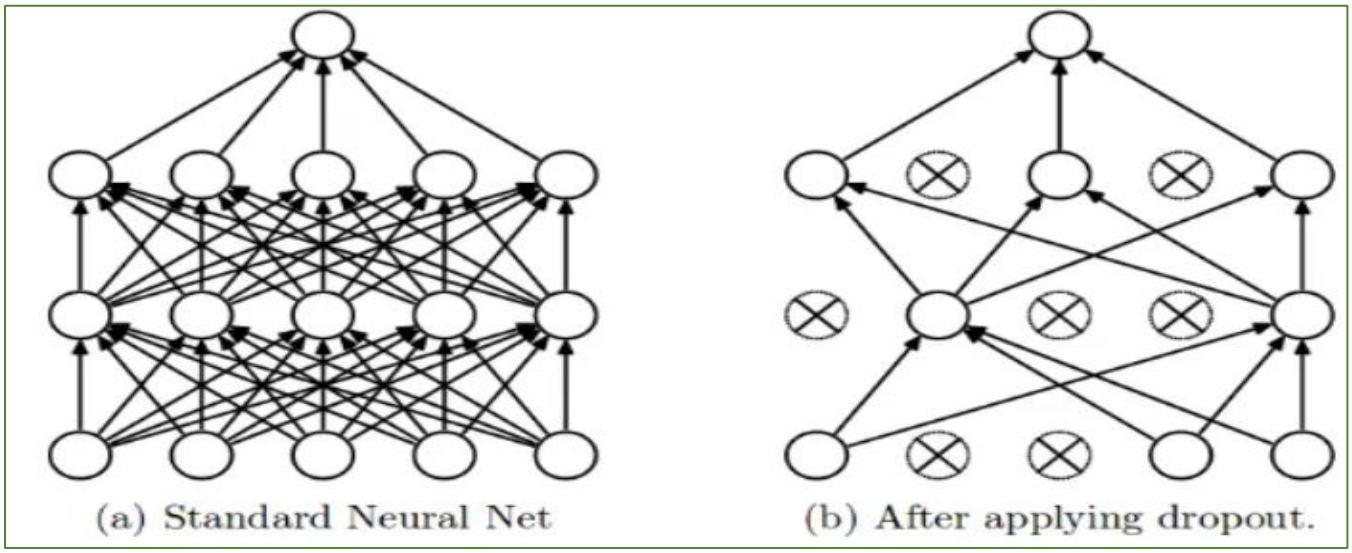
Aspect	Regular CNN	U-Net
Applications	- Image classification, object detection.	- Biomedical image segmentation, satellite imagery, etc.

In summary, U-Net's symmetric design, skip connections, and focus on precise segmentation make it powerful for tasks requiring pixel-level accuracy. Regular CNNs lack this specialization

Unit-4

6.Define dropout. What are different types of dropouts? Explain how dropout is helpful in the regularization of the neural network model.

Dropout Regularization in Neural Networks



Dropout is a powerful technique used to prevent overfitting in deep neural networks. It involves randomly deactivating a chosen proportion of neurons (and their connections) within a layer during training. By temporarily removing these neurons, dropout prevents any single neuron from becoming too specialized or overly dependent on specific features in the training data.

Here are the key points about dropout:

1. **Definition**:
   - Dropout randomly ignores or "drops out" some layer outputs during training.
   - It is implemented per-layer in various types of layers, including dense fully connected, convolutional, and recurrent layers (excluding the output layer).
2. **How Dropout Works**:
   - During training, dropout randomly deactivates a fraction of neurons in each layer.
   - The deactivated neurons are chosen at random for each training iteration.
   - To account for the deactivated neurons, the outputs of the remaining active neurons are scaled up by a factor equal to the probability of keeping a neuron active.

    - This randomness prevents overfitting by ensuring that no single neuron dominates the learning process.
3. **Different Types of Dropouts:**

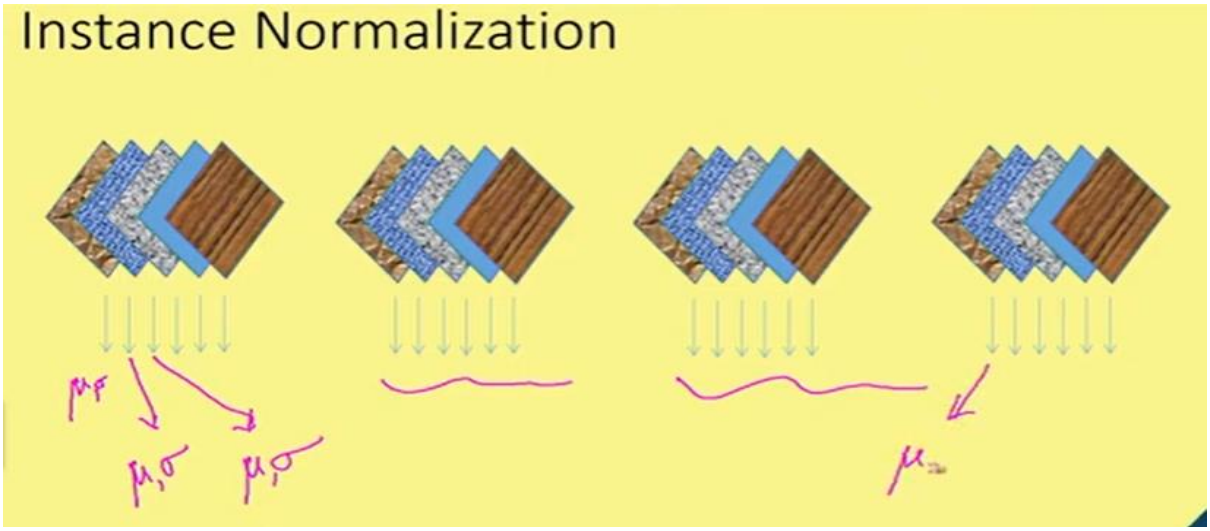
- **Standard Dropout**: Randomly drops neurons during training.
- **Spatial Dropout**: Applies dropout to entire 2D feature maps (commonly used in convolutional neural networks).
- **Alpha Dropout**: Uses a different scaling factor for the remaining active neurons.
- **Variational Dropout**: Adds noise to the dropout mask to improve robustness.
- **Zoneout**: Randomly drops entire hidden states in recurrent neural networks (RNNs).

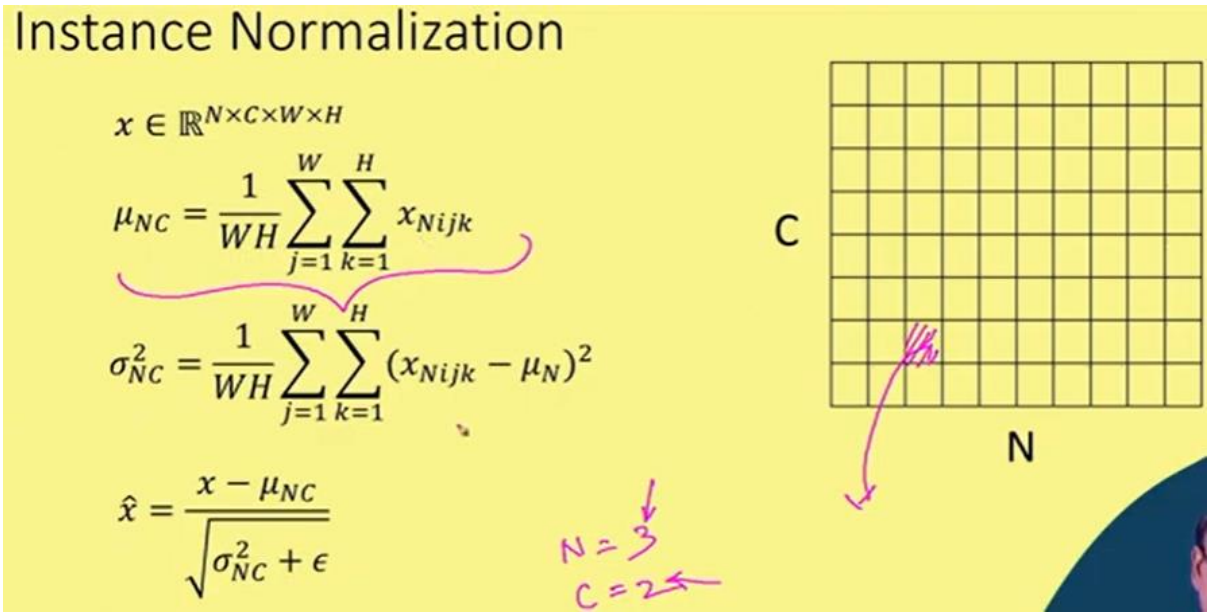
4. **Benefits of Dropout**:
    - **Prevents Overfitting**: By randomly disabling neurons, the network cannot overly rely on specific connections, leading to better generalization.
    - **Ensemble Effect**: Dropout acts like training an ensemble of smaller neural networks with varying structures during each iteration, improving the model's ability to generalize to unseen data.
    - **Enhances Data Representation**: Dropout introduces noise, generates additional training samples, and improves model effectiveness during training.
5. **Implementation**:
    - Dropout is typically added as a separate layer after a fully connected layer in the neural network architecture.
    - The dropout rate (probability of dropping a neuron) is a hyperparameter that needs tuning.
    - Start with a dropout rate of around 20% and adjust based on model performance.

In summary, dropout regularization helps prevent overfitting by randomly deactivating neurons, leading to better generalization and improved neural network performance. It's a simple yet effective technique that enhances model robustness and reliability.

_____----------_____

7.Explain about Instance normalization in the context of CNNs.





Instance Normalization in Convolutional Neural Networks

Instance Normalization (also known as contrast normalization) is a normalization technique used in Convolutional Neural Networks (CNNs). It is typically used between convolutional layers and nonlinearities, such as ReLU layers [1].

Working of Instance Normalization

Instance Normalization works by normalizing the features in each individual example in a batch over both spatial dimensions, independently, in contrast to Batch Normalization which normalizes features jointly across all examples in a batch [2]. The formula for Instance Normalization is given by:

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}$$

where:

- x_{tijk} is the input to the layer
- y_{tijk} is the output
- μ_{ti} is the mean of the input over spatial dimensions
- σ_{ti}^2 is the variance of the input over spatial dimensions
- ϵ is a small constant for numerical stability [2]

Benefits of Instance Normalization

Instance Normalization helps improve the convergence of training the CNN and reduces the sensitivity to network hyperparameters [1]. It also ensures the same behavior at train and test times [3].

Diagram

```
Input Image
    |
Convolutional Layer
    |
Instance Normalization
    |
ReLU Activation
    |
... (further layers)
```

In the above diagram, the Instance Normalization layer is placed after a Convolutional layer and before a ReLU Activation layer, which is a common practice in CNN architectures [3].

Training and Testing:
- During training, the mean and variance are computed per instance (sample) within a batch.
- During testing, IN behaves as a linear operator and can be fused with the previous layer (e.g., fully connected or convolutional).

Benefits of Instance Normalization:
- **Improved Gradient Flow**: IN helps gradients flow more smoothly during training.
- **Higher Learning Rates**: It allows higher learning rates, leading to faster convergence.
- **Robust Initialization**: Networks become more robust to initialization.
- **Regularization**: Acts as implicit regularization during training.

8.Define Early Stopping and explain how it prevents the overfitting.

Early stopping is an approach used during the training of complex machine learning models to **prevent overfitting**.

Early stopping is a technique used during the training of machine learning models, particularly neural networks, to prevent overfitting. It involves monitoring the model's performance on a validation set during training and stopping the training process when certain conditions are met.

Let's delve into the details:

1. **Overfitting and Regularization**:
   - **Overfitting** occurs when a model performs exceptionally well on the training dataset but fails to generalize to unseen data (test/validation sets). Essentially, it captures noise and idiosyncrasies specific to the training data.
   - **Regularization techniques** help mitigate overfitting. One effective method, especially for neural networks, is **early stopping**.
2. **How Early Stopping Works**:
   - After each iteration of the training algorithm, we evaluate the model on both the training and validation sets.

- The **training curve** shows decreasing errors as the model fits the training data better.
- The **validation curve** reveals how well the model generalizes to unseen data. It typically starts with larger errors that gradually decrease but eventually may increase.
- The **inflection point** in the validation curve marks where the model starts incorporating noise from the training set, leading to performance deterioration on the validation set.
- **Early stopping aims to stop training before reaching this inflection point**.

3. **Pseudocode for Early Stopping**:
   - We define a parameter, say `n`, which represents the number of consecutive iterations in which validation losses should increase.
   - During training:
     - If the validation loss increases for `n` consecutive iterations, we stop training.
     - We save the current model because it's the best one found up to that point.
     - If the validation loss doesn't increase, we reset the counter.
   - This approach not only avoids the inflection point but also stops training if no validation-score improvement occurs.
4. **Setting the Parameter**:
   - Choosing the right value for `n` is crucial:
     - A large `n` requires stronger proof of overfitting, leading to longer training.
     - A small `n` may stop training prematurely, resulting in an unsatisfactory model.
   - Balancing these factors is essential.

WORKING

Here's how it works:

1. **Training and Validation Curves**:
   - During model training, we compute both the training loss (error) and the validation loss at each iteration.
   - The **training curve** shows how well the model fits the training data over time. It typically decreases as the model learns.
   - The **validation curve** reflects how well the model generalizes to unseen data (validation set). Initially, the validation loss decreases, but after a point, it may start increasing due to overfitting.

2. **The Inflection Point**:
   - Early stopping aims to identify the **inflection point** in the validation curve.

- This point occurs when the model begins to incorporate noise from the training data, leading to worse performance on the validation set.
- Beyond this point, the model overfits the training data and fails to generalize well.

3. **Stopping Criteria**:
   - We set a parameter, say `n`, which represents the number of consecutive iterations with increasing validation loss.
   - During training:
     - If the validation loss increases for `n` consecutive iterations, we stop training.
     - We save the current model as the best one found so far.
     - If the validation loss doesn't increase, we reset the counter.

Remember, early stopping helps strike a balance between fitting the training data well and generalizing effectively to new data. It prevents the model from learning noise and ensures better performance on unseen examples.

9.Explain how the Data Augmentation Technique improves the performance of the Model.

Data Augmentation: Enhancing Model Performance

Data augmentation is a powerful technique in machine learning that significantly improves model performance.
Data augmentation is the process of increasing the amount and diversity of data. We do not collect new data; rather we transform the already present data. I will be talking specifically about image data augmentation in this article. So we will look at various ways to transform and augment the image data. Let's delve into the details:

1. **What is Data Augmentation?**
   - Data augmentation artificially expands the training dataset by creating modified copies of existing data.
   - It involves making minor alterations to the dataset or generating new data points using deep learning methods.
   - Although commonly used for images, data augmentation can also be applied to audio, text, and other data types.
2. **Why Use Data Augmentation?**
   - **Preventing Overfitting**:
     - By introducing artificial variations, data augmentation prevents the model from memorizing specific instances.
     - This forces the model to learn more invariant representations, ultimately improving generalization.
   - **Increasing Model Accuracy**:
     - A larger, diverse training set allows models to learn better features and patterns.
   - **Reducing Labeling Costs**:
     - Instead of manually labeling more data, augmentation generates additional examples.
3. **Techniques for Different Data Types**:
   - **Image Augmentation**:
     - Geometric transformations (flipping, cropping, rotation, zooming).
     - Color space transformations (changing RGB channels, contrast, brightness).

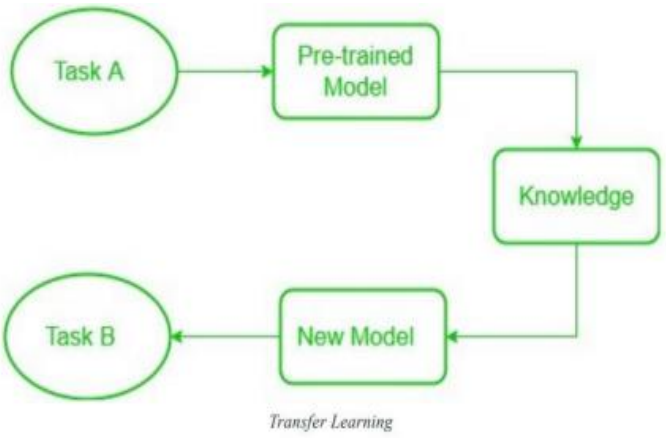
        - Kernel filters (adjusting sharpness or blurring).
    - **Audio Augmentation**:
        - Noise injection (adding random noise).
        - Shifting (changing audio position).
        - Speed and pitch changes.
    - **Text Augmentation**:
        - Word shuffling, replacement, syntax-tree manipulation.
        - Random word insertion or deletion.
4. **Data Augmentation in Deep Learning**:
    - **Need for Data Augmentation**:
        - Crucial in deep learning due to the following reasons:
            - **Increasing Dataset Size**:
                - Collecting diverse data can be challenging; data augmentation expands the dataset without new data collection.
            - **Introducing Variability**:
                - Transforming existing data introduces variations, making the model more robust and better at generalizing.
    - **Operations in Data Augmentation**:
        - Rotation: Rotates the image by a specified degree (useful for handling object orientation variations).
        - Shearing: Changes the image orientation (useful for introducing perspective changes).
        - Zooming: Zooms in or out of the image (helps handle different scales).
        - Cropping: Selects a specific area from an image (useful for focusing on relevant features).
        - Flipping: Horizontally or vertically flips the image (be cautious with vertical flipping for facial recognition).
        - Brightness Adjustment: Helps combat illumination changes (robustness to varying lighting conditions).
    - **Operations: Explanation**
        - **Rotation**:
            - Rotates the image by a specified degree.
            - Useful for handling variations in object orientation.
            - For instance, if you're training an object detection model to recognize cars, rotation augmentation helps the model learn to identify cars from different angles.
        - **Shearing**:
            - Changes the orientation of the image.
            - Useful for introducing perspective changes.
            - Imagine a street scene: shearing can simulate the effect of viewing it from a slightly different angle, enhancing the model's ability to generalize.
        - **Zooming**:
            - Zooms in or out of the image.
            - Helps the model handle different scales.
            - For example, if your dataset contains images of various resolutions, zoom augmentation ensures the model adapts to these variations.
        - **Cropping**:
            - Selects a specific area from an image.
            - Useful for focusing on relevant features.
            - In facial recognition, cropping can help the model concentrate on critical facial landmarks.
        - **Flipping**:
            - Horizontally or vertically flips the image.

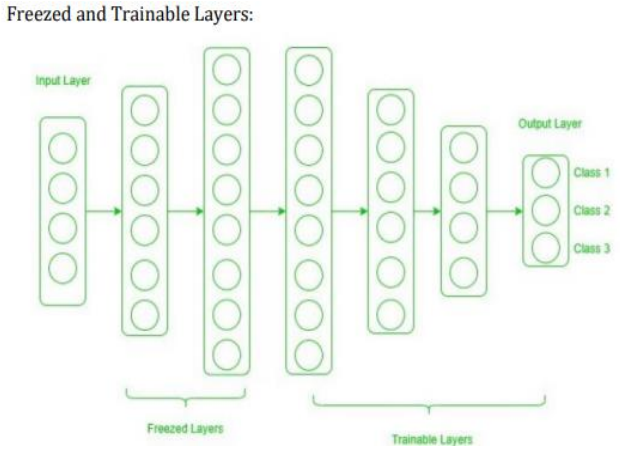
        - Be cautious with vertical flipping (e.g., for facial recognition).
        - Flipping augmentation introduces mirror images, which aids the model in recognizing objects regardless of their orientation.
    - **Brightness Adjustment**:
        - Helps combat illumination changes.
        - Useful for making the model robust to varying lighting conditions.
        - Brightness augmentation ensures the model performs consistently under different lighting scenarios.
- **Data Augmentation in Keras**:
    - Use the `ImageDataGenerator` class in Keras.
    - Specify augmentation parameters (e.g., rotation range, brightness range).
    - The generator applies these transformations during training, improving the model's ability to handle real-world variations.

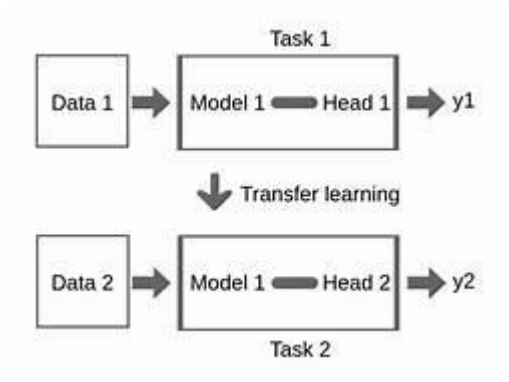
Remember that data augmentation is a powerful technique, but it should be thoughtfully tailored to the specific problem domain and dataset characteristics.

---

10. Why do we need Transfer Learning? Explain.







Transfer Learning: Leveraging Pre-trained Models

Transfer learning is a powerful technique in machine learning where a model trained on one task is used as a starting point for a model on a second, related task. It offers several advantages and is particularly useful when the second task has limited data available or shares similarities with the first task. Here's why we need transfer learning:

1. **Scarcity of Data**:
    - Collecting large amounts of labeled data for training deep neural networks can be challenging and resource-intensive.

- Transfer learning allows us to leverage pre-existing models trained on massive datasets, even if our target dataset is small.

2. **Generalization and Feature Extraction**:
   - In the early layers of deep networks, models learn low-level features such as edges, colors, and textures.
   - These features are not specific to any particular dataset or task; they are general and reusable.
   - Transfer learning enables us to inherit these learned features, which can significantly boost performance on related tasks.
3. **Faster Convergence and Better Initialization**:
   - Starting from pre-trained weights provides a better initialization point for the model.
   - Fine-tuning the model on the new task allows it to adapt more quickly, as it doesn't need to learn everything from scratch.
4. **Preventing Overfitting**:
   - Transfer learning helps prevent overfitting by leveraging features that are already relevant to the new task.
   - The model has already learned general patterns, reducing the risk of fitting noise in the new dataset.
5. **Domain Adaptation**:
   - When the source and target domains share some underlying structure (e.g., images of different animals), transfer learning helps adapt the model.
   - By fine-tuning, we adjust the model's parameters to align with the target domain while retaining useful features from the source domain.

How Transfer Learning Works:

1. **Pre-trained Model**:
   - Start with a model that has been pre-trained on a large dataset (e.g., ImageNet for image classification).
   - This base model has already learned general features relevant to various tasks.
2. **Base Model:**
   - The model that has been pre-trained is known as the base model. It is made up of layers that have utilized the incoming data to learn hierarchical feature representations.
3. **Transfer Layers**:
   - Identify a set of layers in the pre-trained model that captures generic information.
   - These layers are usually found near the top of the network.
   - They learn features like edges, textures, and basic shapes.
4. **Fine-tuning**:
   - Retrain the chosen layers using the new dataset (target task).
   - Preserve the knowledge from pre-training while allowing the model to adapt to the new task.
   - Adjust the model's parameters to fit the demands of the current assignment.

Freeze and Trainable Layers:

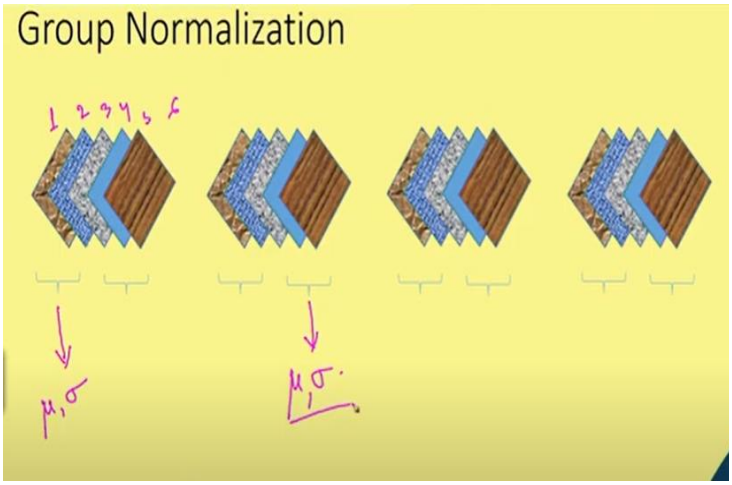
- **Freeze Layers**:
  - Some layers (e.g., early layers) are fixed and not updated during fine-tuning.
  - These layers retain general features.
  - Useful when the target dataset is similar to the base network dataset.
- **Trainable Layers**:
  - Other layers (e.g., fully connected layers) are trained on the new dataset.
  - These layers adapt to the specific task.
  - Useful when the target dataset is different from the base network dataset.
- **The target dataset is small and similar to the base network dataset**
- **The target dataset is large and similar to the base training dataset:**

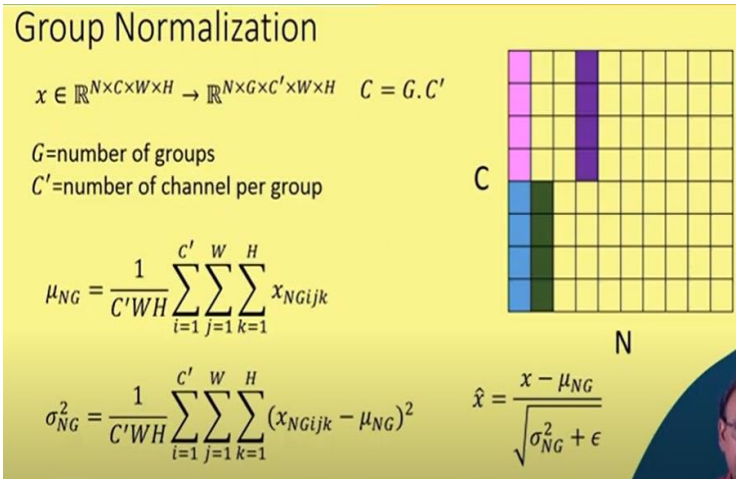
- **The target dataset is small and different from the base network dataset**
- **The target dataset is large and different from the base network dataset:**

Transfer learning is efficient, effective, and widely used in practice. It allows us to build accurate models even with limited data, making it a valuable tool in deep learning.

---

11. Explain about Group normalization in the context of CNNs.





Group Normalization is a normalization layer that divides channels into groups and normalizes the features within each group. GN does not exploit the batch dimension, and its computation is independent of batch sizes. In the case where the group size is 1, it is equivalent to Instance Normalization.
As motivation for the method, many classical features like SIFT and HOG had *group-wise* features and involved *group-wise normalization*. For example, a HOG vector is the outcome of several spatial cells where each cell is represented by a normalized orientation histogram.
Group Normalization (GN) is a technique used in Convolutional Neural Networks (CNNs) to normalize the activations within a neural network layer.
Unlike **Batch Normalization (BN)**, which normalizes across the entire batch of data, GN divides the channels (feature maps) into groups and normalizes each group independently.
Let's delve into the details:

1. Motivation:

- **Batch Normalization (BN)** is widely used to normalize activations in neural networks. However, BN has limitations, especially for small batch sizes or varying batch sizes.
- GN aims to address these limitations by normalizing activations within smaller groups of channels (feature maps) rather than across the entire batch.

2. **Procedure:**
    - Given an input tensor with shape `(N, C, H, W)`:
    - `N`: Batch size
    - `C`: Number of channels (feature maps)
    - `H`: Height of feature maps
    - `W`: Width of feature maps
- GN divides the `C` channels into `G` groups (where `G` is a hyperparameter).

Formally, Group Normalization is defined as:

$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k$$

$$\sigma_i^2 = \frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2$$

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Here x is the feature computed by a layer, and i is an index. Formally, a Group Norm layer computes μ and σ in a set $\mathcal{S}_i$ defined as: $\mathcal{S}_i = \{k \mid k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor \}$.

Here G is the number of groups, which is a pre-defined hyper-parameter ($G = 32$ by default). C/G is the number of channels per group. $\lfloor$ is the floor operation, and the final term means that the indexes i and k are in the same group of channels, assuming each group of channels are stored in a sequential order along the C axis.

3. Benefits of Group Normalization:

- **Stability**: GN is less sensitive to batch size variations.
- **Robustness**: Works well even for small batch sizes.
- **Independence**: Each group operates independently, allowing better generalization.

4. Comparison with Batch Normalization:

- BN normalizes across the entire batch, which can be problematic for small batches.
- GN normalizes within smaller groups, making it more robust.
- BN introduces additional parameters (mean and variance), while GN introduces fewer parameters (group-specific scaling and shifting).

---

12. What is normalization? How is it useful in neural network training?

Normalization in Neural Networks

Normalization techniques play a crucial role in training deep neural networks. They help stabilize and accelerate the training process, improve convergence, and enhance generalization. Let's dive into the details of normalization and its usefulness:

1. **What is Normalization?**
    - **Normalization** refers to the process of transforming input features or intermediate activations in a neural network to have specific properties.
    - The goal is to make the data distribution more suitable for training by reducing internal covariate shift (changes in the distribution of activations during training).
2. **Why is Normalization Useful?**
    - **Stabilizing Training:**
        - Without normalization, neural networks can suffer from slow convergence or even fail to converge.
        - Normalization helps stabilize the training process by ensuring that gradients don't explode or vanish during backpropagation.
    - **Accelerating Convergence:**

        - Normalized features allow the optimization algorithm (e.g., gradient descent) to converge faster.
        - It helps the model learn more efficiently by preventing large weight updates.
    - **Generalization Improvement:**
        - Normalization reduces overfitting by preventing the model from memorizing the training data.
        - It encourages better generalization to unseen examples.
    - **Robustness to Input Variations:**
        - Normalization makes the model less sensitive to variations in input scale, translation, or rotation.
3. **Types of Normalization Techniques:**
    - **Batch Normalization (Batch-Norm):**
        - Normalizes activations across a mini-batch.
        - Helps stabilize training by reducing internal covariate shift.
        - Applied after the linear transformation (before activation function).
        - Includes learnable parameters (scale and shift) for flexibility.

Batch Normalization

Batch normalization is a method that normalizes activations in a network across the mini-batch of definite size. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation.

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

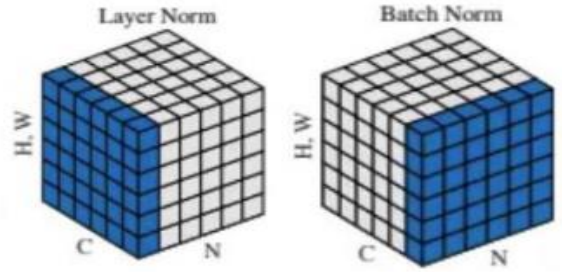
Problems associated with Batch Normalization

1. **Variable Batch Size →** If batch size is of 1, then variance would be 0 which doesn't allow batch norm to work. If we have small mini-batch size then it becomes too noisy and training might affect.
2. **Distributed Training:** if you are computing in different machines then you have to take same batch size because otherwise γ and β will be different for different systems.
3. **Recurrent Neural Network →** In an RNN, we have to fit a separate batch norm layer for each time-step. This makes the model more complicated and space consuming because it forces us to store the statistics for each time-step during training.

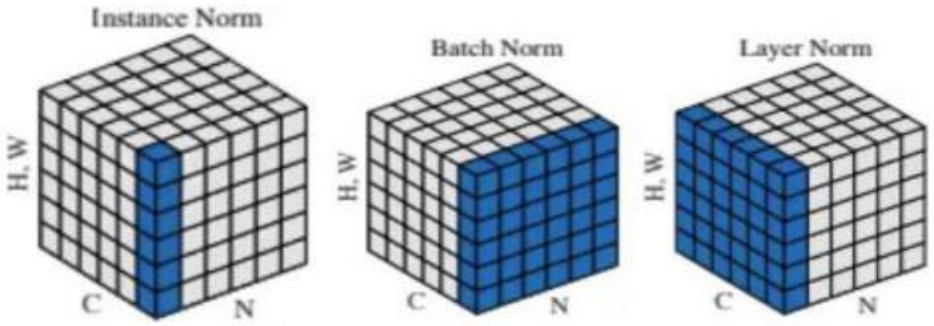
- **Layer Normalization (Layer-Norm):**
  - Normalizes activations across features (channels) for each example.
  - Useful for recurrent neural networks (RNNs) and transformers.
  - Applied after the linear transformation (before activation function).

Layer Normalization

Layer normalization normalizes input across the features instead of normalizing input features across the batch dimension in batch normalization.



- **Instance Normalization (Instance-Norm):**
  - Normalizes activations across features (channels) for each example independently.
  - Commonly used in style transfer and image-to-image translation tasks.
  - Applied after the linear transformation (before activation function).



$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - mu_{ti})^2.$$

Here, $\mathbf{x} \in \mathbb{R}^{T \times C \times W \times H}$ be an input tensor containing a batch of T images. $\mathbf{x}_{tijk}$ denote its tijk-th element

where k and j span spatial dimensions(**H**eight and **W**idth of the image) i is the feature channel (color channel if the input is an RGB image) t is the index of the image in the batch.

$$\mu_i = \frac{1}{m}\sum_{k\in\mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathcal{S}_i}(x_k - \mu_i)^2 + \epsilon},$$

$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i). \qquad y_i = \gamma\hat{x}_i + \beta,$$

Here, **x** is the feature computed by a layer, and i is an index.

- **Group Normalization (Group-Norm):**
  - Divides channels into groups and normalizes each group independently.
  - Useful when batch size is small or when Batch-Norm is not suitable.
  - Applied after the linear transformation (before activation function).

4. **Early Stopping:**
   - Although not a direct normalization technique, early stopping is related to regularization.
   - Early stopping prevents overfitting by monitoring the validation performance during training.
   - Training stops when the validation performance starts deteriorating (indicating overfitting).
5. **Visualization of Normalization Techniques:**
   - Normalization Techniques{:target="_blank"}
   - The diagram shows the differences between Batch-Norm, Layer-Norm, and Instance-Norm.

Weight Normalization

We **normalize weights of a layer** instead of normalizing the activations directly Weight normalization reparameterizes the weights (**ω**) as :

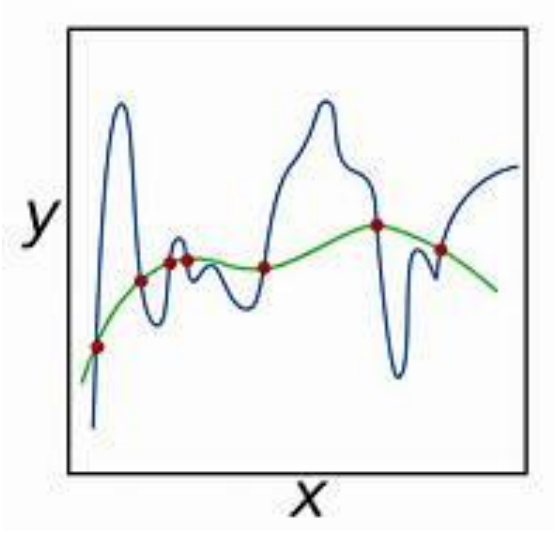
$$\boldsymbol{w} = \frac{g}{\|\boldsymbol{v}\|}\boldsymbol{v}$$

6. **Conclusion:**
   - Normalization techniques are essential for effective neural network training.
   - Choose the appropriate normalization method based on the problem, architecture, and available data.
   - By incorporating normalization, neural networks become more robust, converge faster, and generalize better.

Remember that normalization is a powerful tool, but its effectiveness depends on the specific context and problem.

---

13. What is the objective of regularization? How the dropout mechanism fulfills that objective?



Regularization in machine learning aims to prevent overfitting by adding a penalty term to the model's objective function during training. The primary objectives of regularization are:

1. **Complexity Control**: Regularization helps control model complexity by discouraging the model from assigning too much importance to individual features or coefficients. This results in better generalization to new data.
2. **Reducing Overfitting**: Overfitting occurs when a model learns to perform exceptionally well on the training data but fails to generalize to unseen data. Regularization helps prevent overfitting by adding constraints to the model's complexity.
3. **Improving Generalization**: A well-regularized model generalizes better to new data. It strikes a balance between fitting the training data well and avoiding excessive complexity.
4. **Preventing Overfitting**: Overfitting occurs when a model memorizes the training data instead of learning its underlying patterns. Regularization penalizes large coefficients, constraining their magnitudes and preventing overly complex models.
5. **Balancing Bias and Variance**: Regularization strikes a balance between model bias (underfitting) and model variance (overfitting), leading to improved performance.
6. **Feature Selection**: Some regularization methods, like L1 regularization (Lasso), promote sparse solutions by driving some feature coefficients to zero. This automatically selects important features while excluding less relevant ones.
7. **Handling Multicollinearity**: When features are highly correlated (multicollinearity), regularization stabilizes the model by reducing coefficient sensitivity to small data changes.
8. **Generalization**: Regularized models learn underlying patterns for better generalization to new data.

Dropout, specifically, fulfills these objectives as follows:

- Dropout randomly ignores certain nodes (units) in a neural network during training. This creates a "thinned" network with unique combinations of units dropped at different points in time.
- By training multiple thinned networks and merging them into one, dropout reduces overfitting by preventing units from becoming codependent.
- The process encourages robust learning and helps the model generalize better to unseen data[1][2].
- In essence, dropout acts as an efficient and effective regularization technique for deep neural networks.

Dropout Mechanism

Dropout is a specific regularization technique commonly used in neural networks. Let's explore how it fulfills the objectives:

1. **How Dropout Works**:
    - Dropout randomly "drops out" (deactivates) a fraction of neurons during training. These dropped neurons do not contribute to forward or backward passes.

    - During each training iteration, dropout randomly selects which neurons to drop with a specified probability (usually around 0.5).
    - At test time (inference), dropout is turned off, and all neurons are active.
2. **Objective Fulfillment**:
    - **Reducing Overfitting**:
        - By dropping neurons during training, dropout prevents co-adaptation of neurons. It forces the network to learn more robust features that do not rely on specific combinations of neurons.
        - The model becomes less sensitive to individual neurons, leading to better generalization.
    - **Improving Generalization**:
        - Dropout acts as implicit regularization. It encourages the network to learn diverse representations.
        - The model becomes less reliant on any single neuron, making it more adaptable to unseen data.
3. **Visual Comparison**:
    - Let's visualize the effect of dropout:
        - !Standard Neural Net <!-- Placeholder image -->
        - (a) Standard Neural Net: All neurons are active during training.
        - (b) After Applying Dropout: Randomly dropped neurons create a more robust network.
4. **Benefits and Limitations**:
    - **Benefits**:
        - Reduces overfitting.
        - Improves generalization.
        - Requires less training data.
        - Efficient and simple to implement.
    - **Limitations**:
        - Risk of underfitting if stopped too early.
        - May not benefit all types of models.
        - Proper validation set selection is crucial.

Conclusion

In summary, dropout is a powerful regularization technique that enhances generalization by promoting diverse feature learning. Its controlled randomness during training ensures better model performance on unseen data.

UNIT-5

14.

Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW's architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C="Hey, this is sample corpus using only one context word." Let us construct the training data.

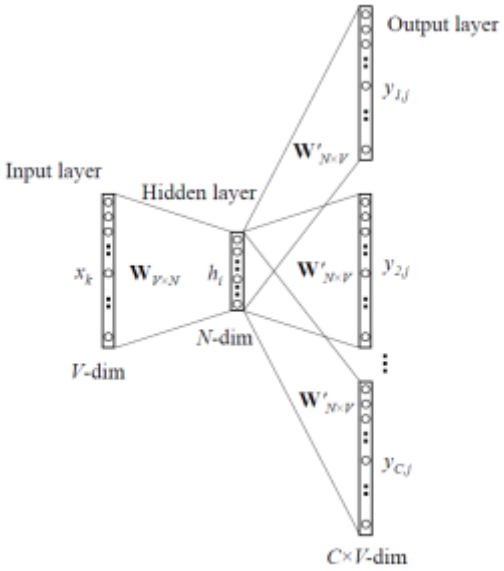
Input	Output(Context1)	Output(Context2)
Hey	this	<padding>
this	Hey	is
is	this	sample
sample	is	corpus
corpus	sample	corpus
using	corpus	only
only	using	one
one	only	context
context	one	word
word	context	<padding>

The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be "two" one hot encoded target variables and "two" corresponding outputs as can be seen by the blue section in the image.

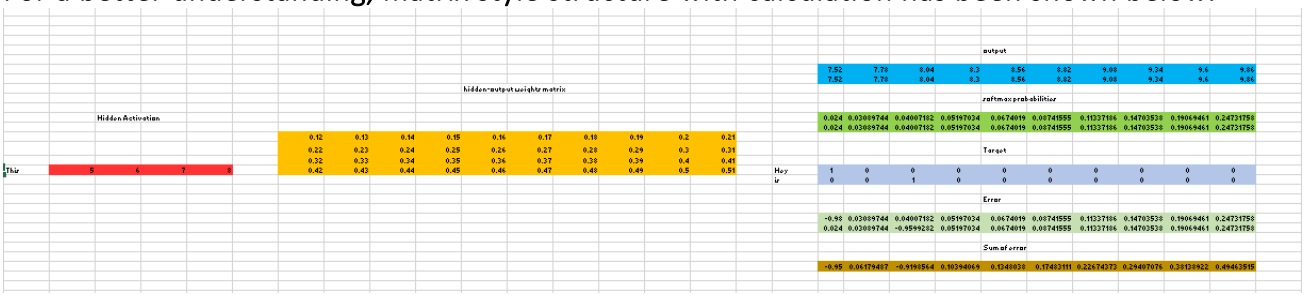
Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.

The weights between the input and the hidden layer are taken as the word vector representation after training. The loss function or the objective is of the same type as of the CBOW model.

The skip-gram architecture is shown below.



For a better understanding, matrix style structure with calculation has been shown below.



Let us break down the above image.

Input layer size – [1 X V], Input hidden weight matrix size – [V X N], Number of neurons in hidden layer – N, Hidden-Output weight matrix size – [N X V], Output layer size – C [1 X V]

In the above example, C is the number of context words=2, V= 10, N=4

1. The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically the corresponding row of input-hidden matrix copied.
2. The yellow matrix is the weight between the hidden layer and the output layer.
3. The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.
4. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
5. The grey matrix contains the one hot encoded vectors of the two context words(target).
6. Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for n target context words, we will have n error vectors.
7. Element-wise sum is taken over all the error vectors to obtain a final error vector.
8. This error vector is propagated back to update the weights.

Advantages of Skip-Gram Model:

1. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.
2. Skip-gram with negative sub-sampling outperforms every other method generally.

→ Discuss the process of learning skipgram Embeddings.

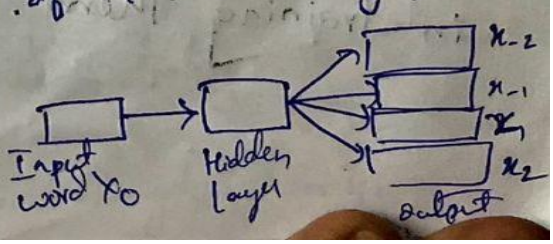
A: The process of learning skipgram embeddings involves training a neural network to predict the context words surrounding a given target word.

The process is

1) Training data preparation:
We start with large corpus of text data. This could be news articles, books or any other form of text
The text is pre-processed to clean it & segment it into sentences & words.
These pairs are inputted into the skipgram model for training.

2) Neural Networks Architecture:
The skipgram model typically includes an input layer, a hidden layer & output layer.
It uses a feedforward neural networks with a single hidden layer.
Hidden layer represents the learned embeddings of input words
Output layer for predicting context words.
Each word in the vocabulary is assigned a unique vector in a low dimensional space.

3) Objective Function:
Skipgrams ideal/objective is to predict the context words of a target word by capturing words within a specific window.
The model aims to learn the weights of the hidden layer, which represents the word vectors.



4) Negative Sampling:

Predicting all possible context words is computationally expensive. Negative sampling is a technique used to make the training process more ~~effective~~ efficient.

The model iterates through the training data for a specified no of epochs, adjusting the weights to minimize prediction errors.

5) Word Embeddings Extraction:

After training, the word embeddings are extracted from the weights of the embedding layer.

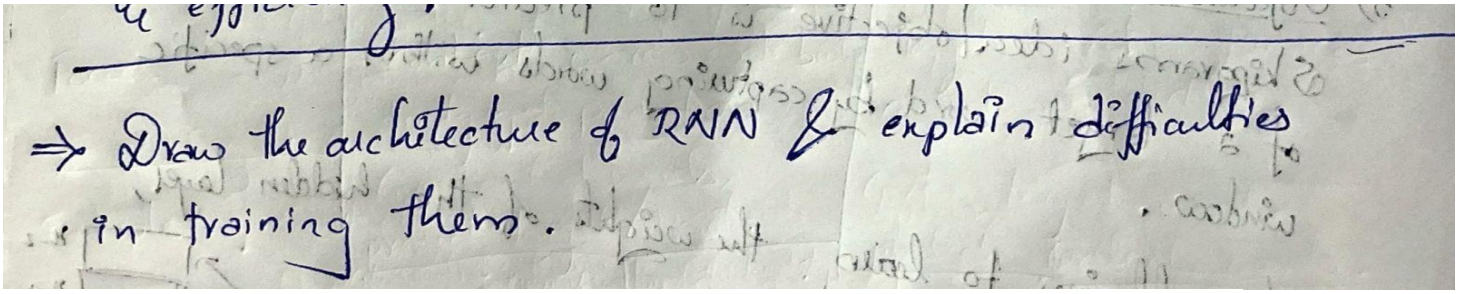
Each word in the vocabulary has unique embedding vector

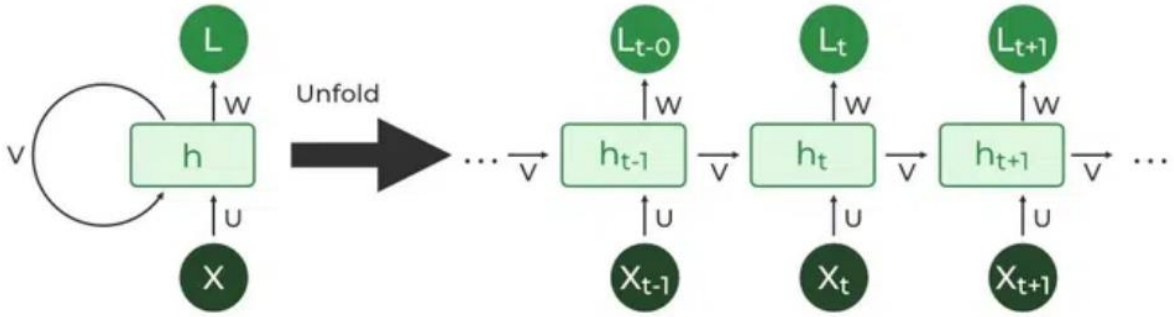
6) Applications

Skipgram Embeddings are a powerful tool for various NLP tasks, such as sentiment analysis, machine ~~Applications~~ translation & named entity recognition.

Skipgram embeddings enhance text processing accuracy & efficiency.

15.





Recurrent Neural Network:

Recurrent Neural Network (RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

Recurrent Neural Network Architecture:

RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in RNN, the weight across the network remains the same. It calculates state hidden state H_i for every input X_i .By using the following formulas:

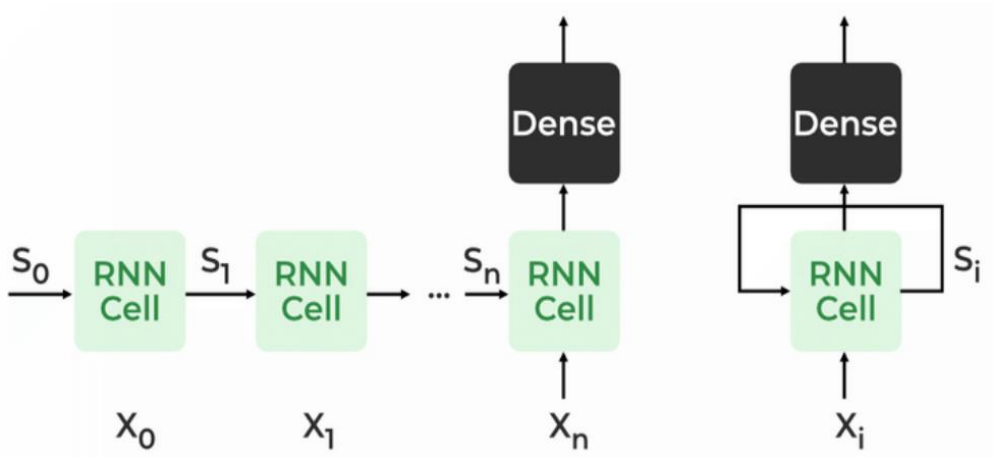
$$h = \sigma(UX + Wh_{-1} + B)$$

$$Y = O(Vh + C)$$

Hence

$$Y = f(X, h, W, U, V, B, C)$$

Here S is the State matrix which has element si as the state of the network at timestep i The parameters in the network are W, U, V, c, b which are shared across timestep.



The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-

The formula for calculating the current state:

$h_t = f(h_{t-1}, x_t)$

where,

- h_t -> current state
- h_{t-1} -> previous state
- x_t -> input state

Formula for applying Activation function(tanh)

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

where,

- w_{hh} -> weight at recurrent neuron
- w_{xh} -> weight at input neuron

The formula for calculating output:

$y_t = W_{hy}h_t$

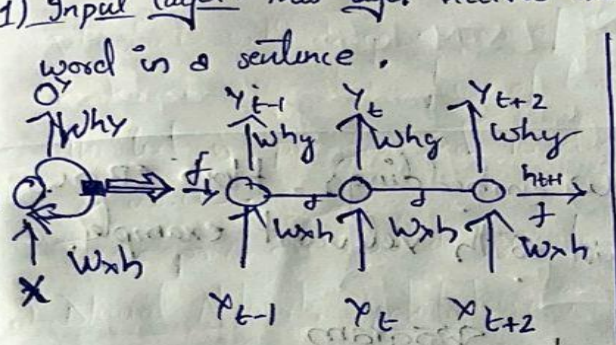
- Y_t -> output
- W_{hy} -> weight at output layer

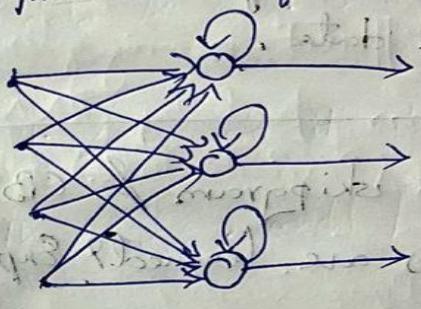
These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.

Recurrent Neural Network is a type of Neural Network where the output from the previous step is fed as input to the current step. RNN's are designed to handle the sequential data like text, speech, or time series data.

The architecture breakdown of RNN

1) Input layer: This layer receives the first element of the sequence, like a word in a sentence.





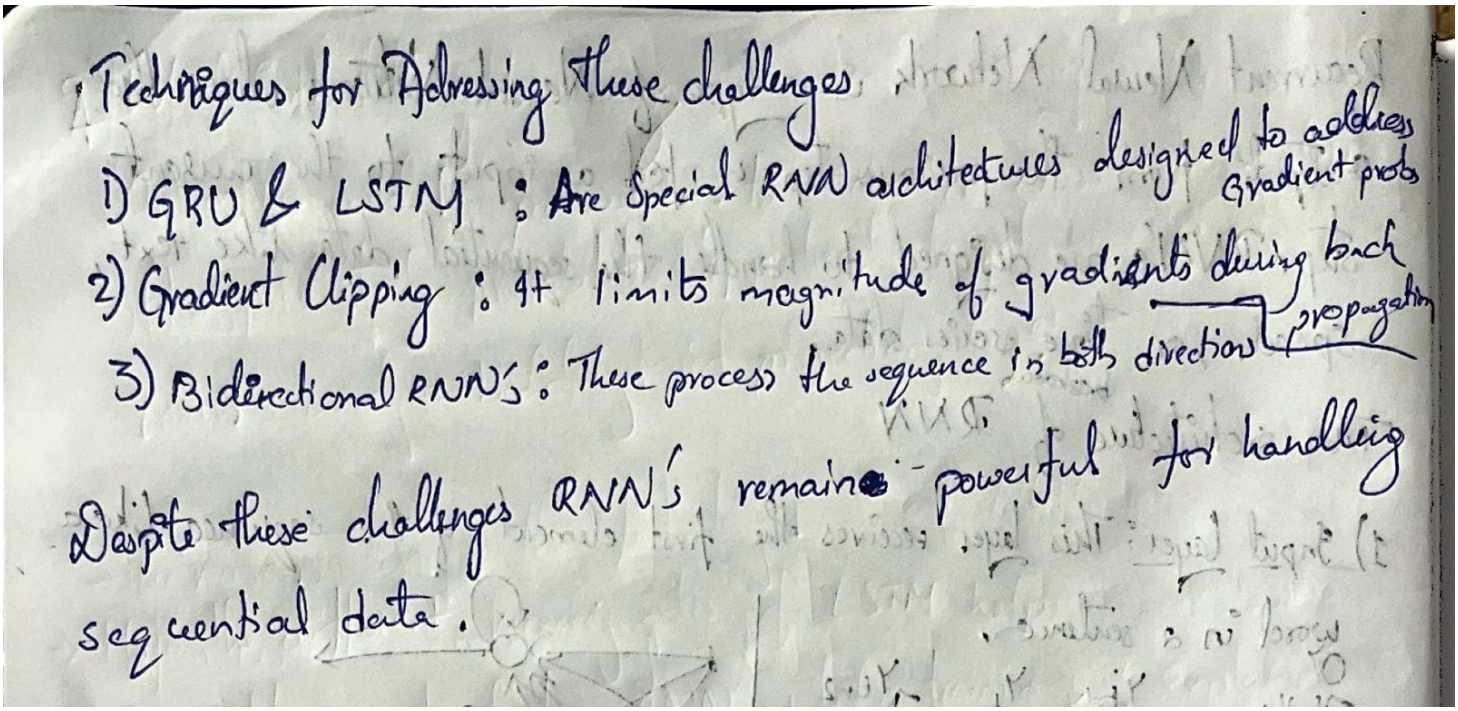
2) Hidden layer: This is the core layer of RNN. It contains recurrent neurons that have connections not only to the previous layers but also back to themselves

3) Output layer: This layer produces output based on the current input & the info stored.

Difficulties in Training RNN's

1) Exploding Gradient Problem:
The gradients can become very large during backpropagation causing the weights to update drastically.

2) Vanishing Gradient Problem:
This problem occurs when ~~long term~~ dealing with long sequences, the gradients become very small or vanish entirely as they are backpropagated through network.



16.

CBOW(Continuous bag of words) and Skip-gram model. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations. Let us discuss both these methods separately and gain intuition into their working.

CBOW (Continuous Bag of words)

The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words. But for simplicity, I will take a single context word and try to predict a single target word.

Suppose, we have a corpus C = “Hey, this is sample corpus using only one context word.” and we have defined a context window of 1. This corpus may be converted into a training set for a CBOW model as follow. The input is shown below. The matrix on the right in the below image contains the one-hot encoded from of the input on the left.

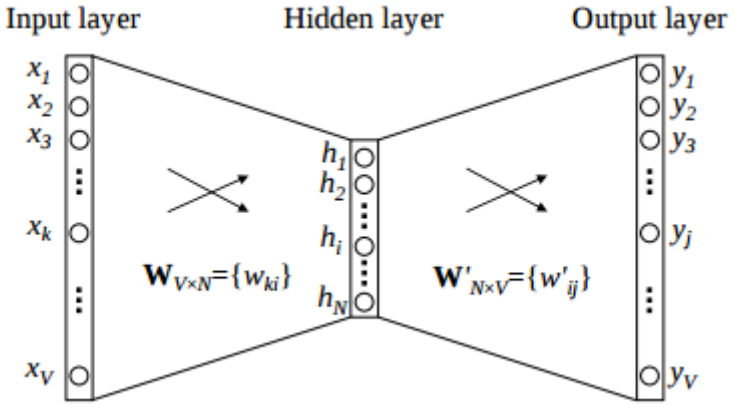
Input	Output			Hey	This	is	sample	corpus	using	only	one	context	word
Hey	this		Datapoint 1	1	0	0	0	0	0	0	0	0	0
this	hey		Datapoint 2	0	1	0	0	0	0	0	0	0	0
is	this		Datapoint 3	0	0	1	0	0	0	0	0	0	0
is	sample		Datapoint 4	0	0	1	0	0	0	0	0	0	0
sample	is		Datapoint 5	0	0	0	1	0	0	0	0	0	0
sample	corpus		Datapoint 6	0	0	0	1	0	0	0	0	0	0
corpus	sample		Datapoint 7	0	0	0	0	1	0	0	0	0	0
corpus	using		Datapoint 8	0	0	0	0	1	0	0	0	0	0
using	corpus		Datapoint 9	0	0	0	0	0	1	0	0	0	0
using	only		Datapoint 10	0	0	0	0	0	1	0	0	0	0
only	using		Datapoint 11	0	0	0	0	0	0	1	0	0	0
only	one		Datapoint 12	0	0	0	0	0	0	1	0	0	0
one	only		Datapoint 13	0	0	0	0	0	0	0	1	0	0
one	context		Datapoint 14	0	0	0	0	0	0	0	1	0	0
context	one		Datapoint 15	0	0	0	0	0	0	0	0	1	0
context	word		Datapoint 16	0	0	0	0	0	0	0	0	1	0
word	context		Datapoint 17	0	0	0	0	0	0	0	0	0	1

The target for a single datapoint say Datapoint 4 is shown as below

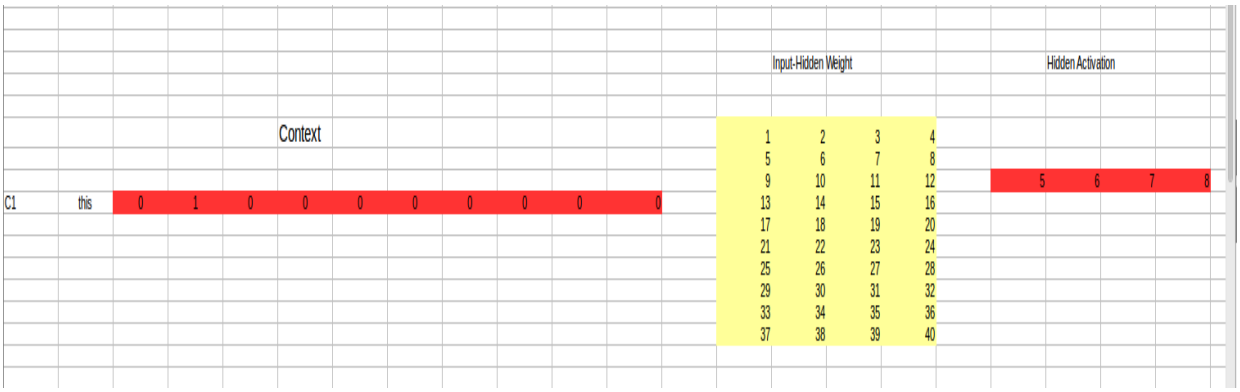
Hey	this	is	sample	corpus	using	only	one	context	word
0	0	0	1	0	0	0	0	0	0

This matrix shown in the above image is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. Now let us see how the forward propagation will work to calculate the hidden layer activation.

Let us first see a diagrammatic representation of the CBOW model.



The matrix representation of the above image for a single data point is below.

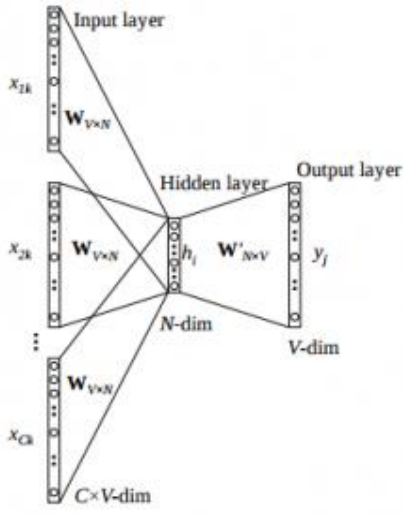


The flow is as follows:

1. The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10 in the above example.
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer.
   Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.
3. There is a no activation function between any layers.(More specifically, I am referring to linear activation)

4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.
5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust the weights.
7. The weight between the hidden layer and the output layer is taken as the word vector representation of the word.

We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.



Below is a matrix representation of the above architecture for an easy understanding.

Context

C1	this	0	1	0	0	0	0	0	0	0	0
C2	corpus	0	0	0	0	1	0	0	0	0	0
C3	context	0	0	0	0	0	0	0	0	1	0

Input-Hidden Weight

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	40

Hidden Activation

5	6	7	8
17	18	19	20
33	34	35	36

Average hidden Activation

18.33333333	19.33333333	20.33333333	21.33333333

The image above takes 3 context words and predicts the probability of a target word. The input can be assumed as taking three one-hot encoded vectors in the input layer as shown above in red, blue and green. So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.

The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix. We can understand this with the above figure. The average vector calculated becomes the hidden activation. So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation.

In both a single context word and multiple context word, I have shown the images till the calculation of the hidden activations since this is the part where CBOW differs from a simple MLP network. The steps after the calculation of hidden layer are same as that of the MLP.

Advantages of CBOW:

1. Being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally).
2. It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

Disadvantages of CBOW:

1. CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies.
2. Training a CBOW from scratch can take forever if not properly optimized.

Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW's architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C="Hey, this is sample corpus using only one context word." Let us construct the training data.

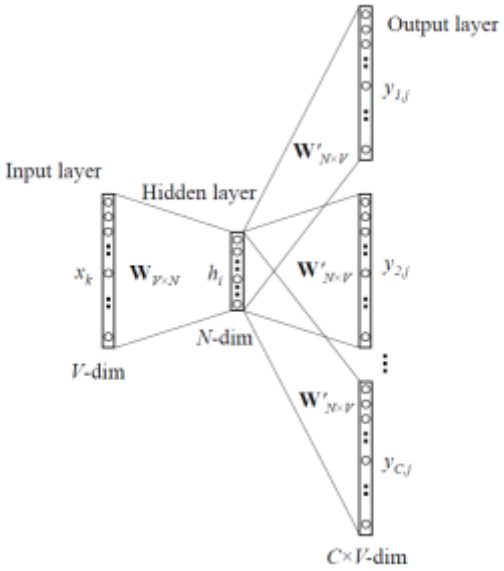
Input	Output(Context1)	Output(Context2)
Hey	this	<padding>
this	Hey	is
is	this	sample
sample	is	corpus
corpus	sample	corpus
using	corpus	only
only	using	one
one	only	context
context	one	word
word	context	<padding>

The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be "two" one hot encoded target variables and "two" corresponding outputs as can be seen by the blue section in the image.

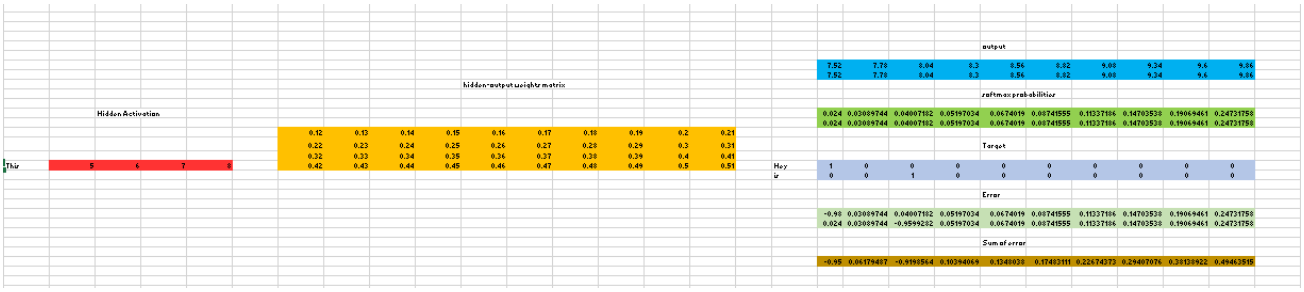
Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.

The weights between the input and the hidden layer are taken as the word vector representation after training. The loss function or the objective is of the same type as of the CBOW model.

The skip-gram architecture is shown below.



For a better understanding, matrix style structure with calculation has been shown below.



Let us break down the above image.

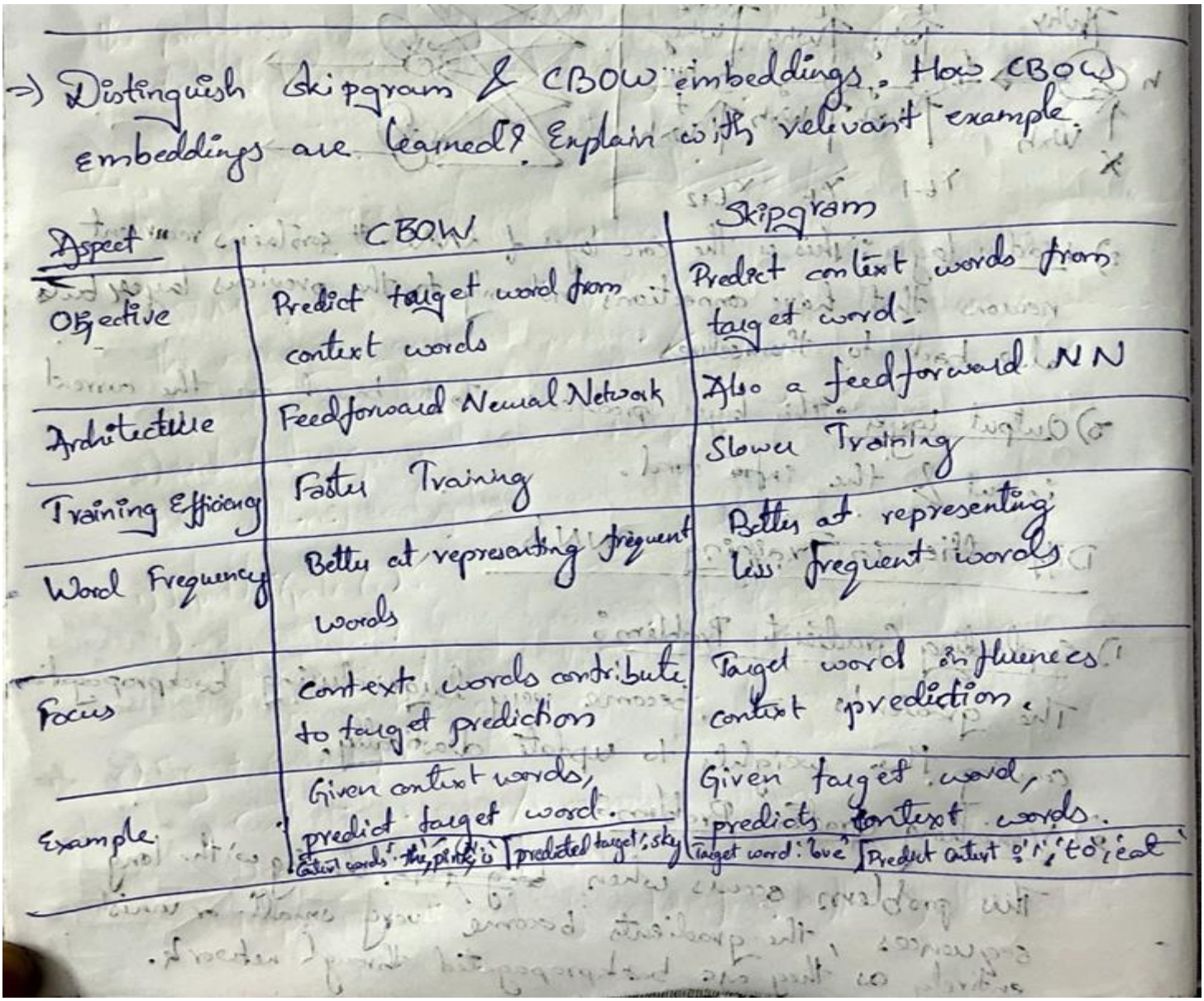
Input layer size – [1 X V], Input hidden weight matrix size – [V X N], Number of neurons in hidden layer – N, Hidden-Output weight matrix size – [N X V], Output layer size – C [1 X V]

In the above example, C is the number of context words=2, V= 10, N=4

9. The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically the corresponding row of input-hidden matrix copied.
10. The yellow matrix is the weight between the hidden layer and the output layer.
11. The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.
12. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
13. The grey matrix contains the one hot encoded vectors of the two context words(target).
14. Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for n target context words, we will have n error vectors.
15. Element-wise sum is taken over all the error vectors to obtain a final error vector.
16. This error vector is propagated back to update the weights.

Advantages of Skip-Gram Model:

3. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.
4. Skip-gram with negative sub-sampling outperforms every other method generally.



→ Distinguish Skipgram & CBOW embeddings. How CBOW embeddings are learned? Explain with relevant example.

Aspect	CBOW	Skipgram
Objective	Predict target word from context words	Predict context words from target word.
Architecture	Feedforward Neural Network	Also a feedforward NN
Training Efficiency	Faster Training	Slower Training
Word Frequency	Better at representing frequent words	Better at representing less frequent words
Focus	Context words contribute to target prediction	Target word influences context prediction.
Example	Given context words, predict target word. Context words: 'the', 'plane', 'is' → Predicted target: 'sky'	Given target word, predicts context words. Target word: 'love' → Predict output: 'I', 'to', 'eat'

Unlike skip-gram, CBOW focuses on predicting the center word based on its surrounding context words.

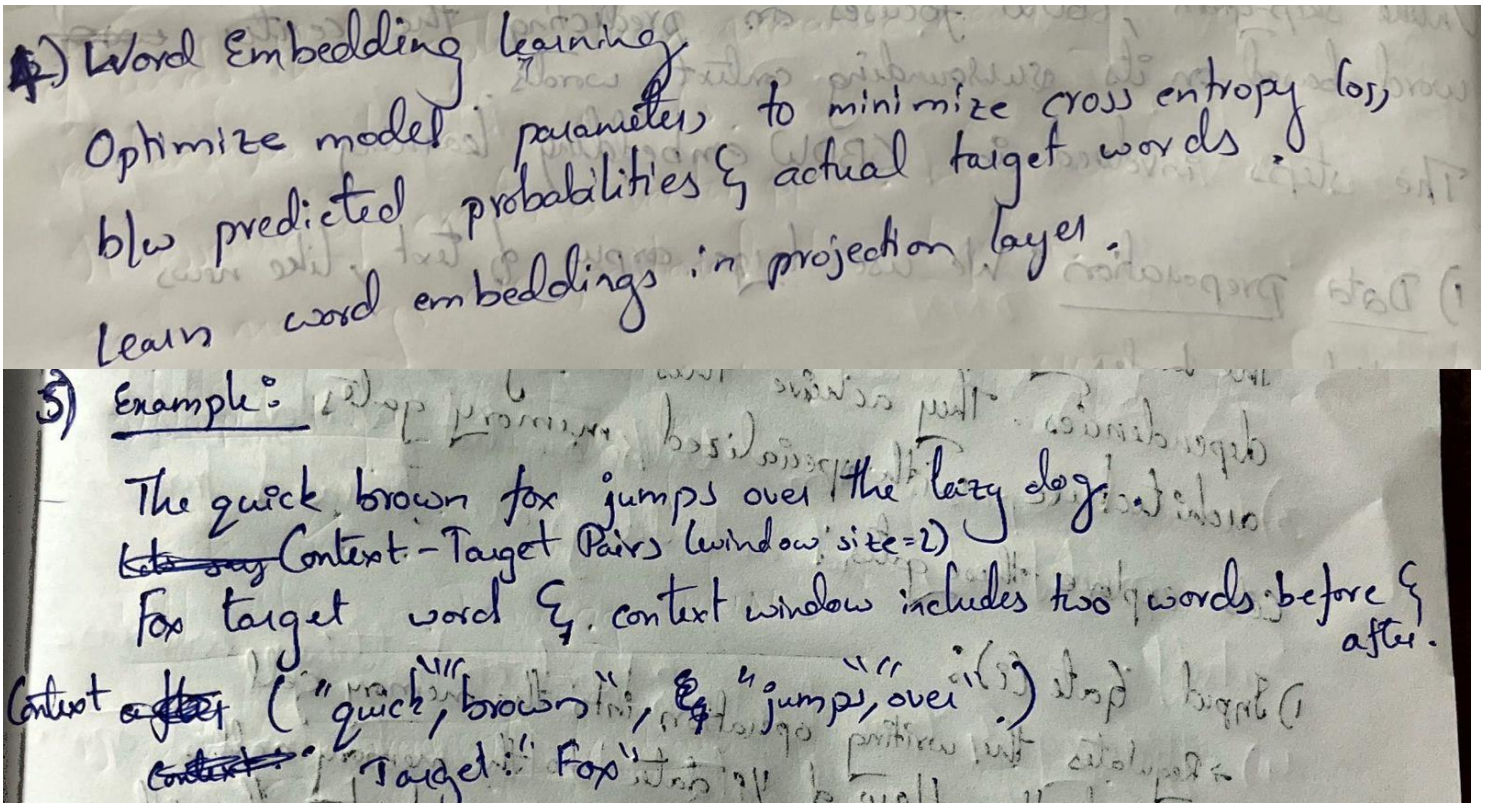
The steps involved in CBOW embedding learning

1) Data preparation: We use large corpus of text, like news articles or books
This text is preprocessed to clean it & break it down into sentences & individual words.

2) Neural Network Architecture:
A simple neural network with 3 layers is used
1) Input layer: Represents the context words
2) Hidden layer: Learns the relationship btw words by combining context word embeddings
3) Output layer: Predicts the probability of each word in vocabulary being the target word.

3) Training Process

Each context word is converted to its embedding vector & then are averaged to create single vector.
This context vector is fed into hidden layer which tries to learn a representation that captures meaning of target word.
Finally o/p layer predicts the probability of each word in vocabulary



17.

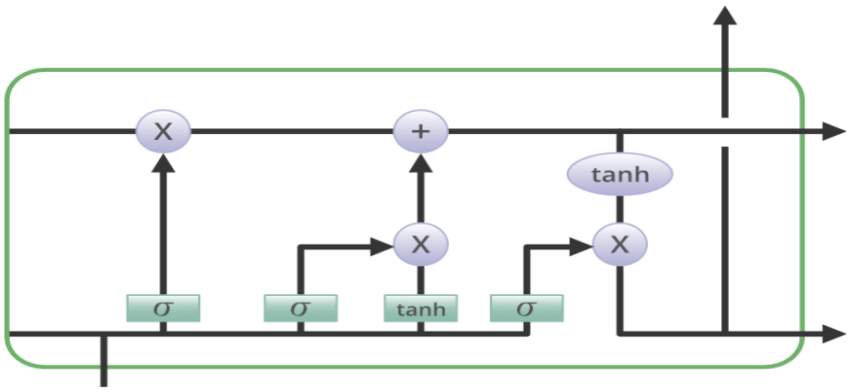
Long Short Term Memory:

Long Short-Term Memory is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber. LSTM is well-suited for sequence prediction tasks and excels in capturing long-term dependencies. Its applications extend to tasks involving time series and sequences. LSTM's strength lies in its ability to grasp the order dependence crucial for solving intricate problems, such as machine translation and speech recognition.

A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting. LSTMs can also be used in combination with other neural network architectures, such as Convolutional Neural Networks (CNNs) for image and video analysis. The memory cell 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. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the 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.

Architecture and Working of LSTM:

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



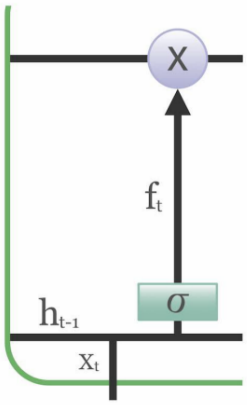
Information is retained by the cells and the memory manipulations are done by the gates. There are three gates –

Forget Gate: The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x (input at the particular time) and h (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:

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

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.



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 h_{t-1} and x_t . Then, a vector is created using tanh function that gives an output from -1 to +1, which contains all the possible

values from h_{t-1} and x_t . 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:

$$i_t = (W_i[h_{t-1}, x_t] + b_i)$$

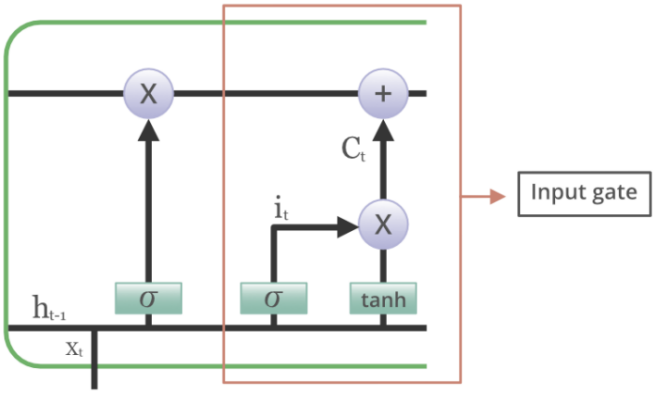
$$\hat{C}_t = tanh(W_c[h_{t-1}, x_t] + b_c)$$

We multiply the previous state by f_t , disregarding the information we had previously chosen to ignore. Next, we include $i_t * C_t$. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

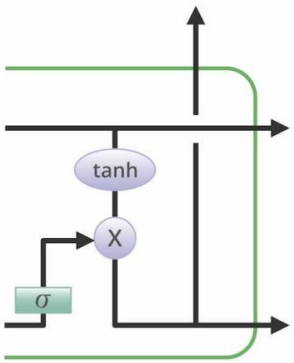
where

- $\odot$ denotes element-wise multiplication
- tanh is tanh activation function



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 h_{t-1} and x_t . 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:

$$o_t = (W_o[h_{t-1}, x_t] + b_o)$$



⇒ How the different memory gates of LSTM helps in addressing the problems faced by RNN's while training? Explain in detailed architecture diagram of LSTM.

A) RNN processes sequential data by considering the order & time dependencies of input data. ~~Unlike feed~~

LSTM's are a variant of RNN's designed to address the vanishing gradient problem & capture long-term dependencies. They achieve this through a more complex architecture with specialized memory gates.

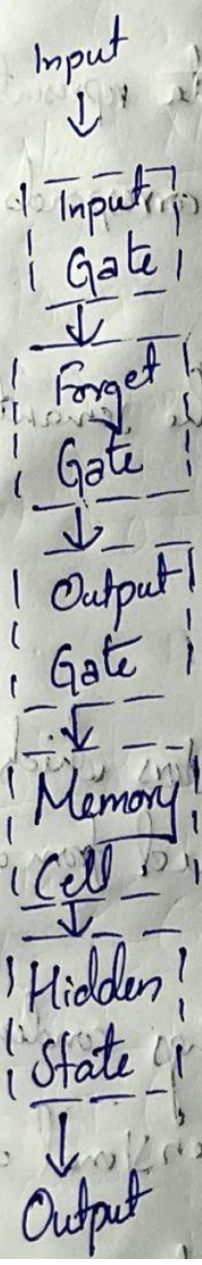
Lets explore these gates!

1) Input Gate (i):
   - → Regulates the writing operation into the memory cell
   - → Controls the flow of i/p data into the memory cell.
   - → Determines how much new info to write to the cell

2) Forget Gate (f):
   - Conducts a erase/remember option.
   - Decides what info to discard from the cell memory.
   - Helps prevent outdated info from affecting the model.

3) Output Gate (O):
→ Determines what o/p to generate from cell memory.
→ Regulates the stream of info from memory cell to LSTM blocks.
→ Influences the next hidden state.

LSTM Architecture

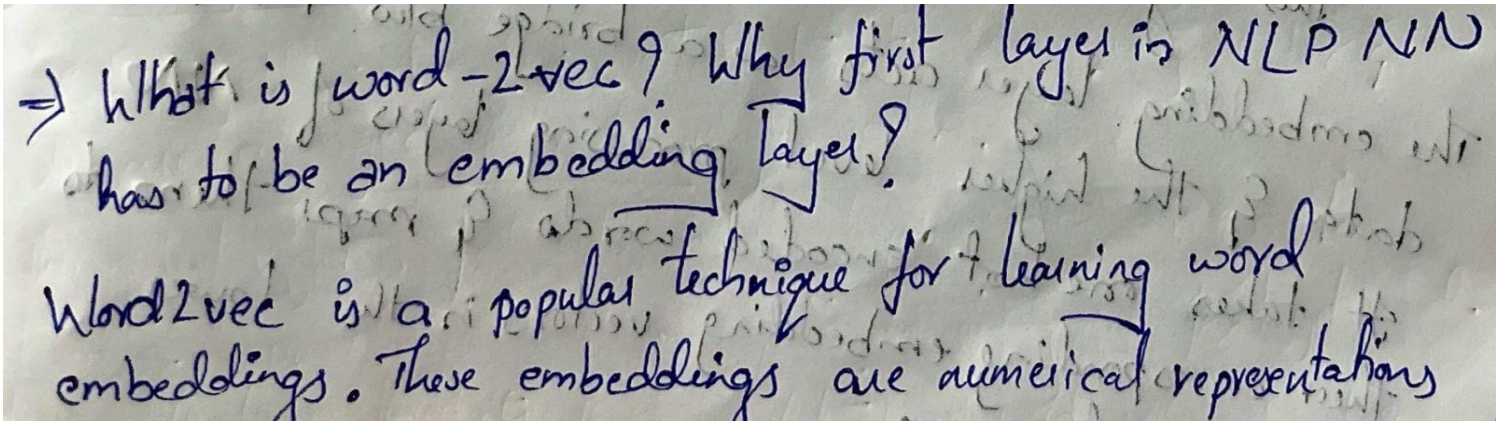


Controls the flow of new data into memory cell.

Manages the cell's memory by erasing/retaining relevant info

Determines the next hidden state & influences the final output.

18.



⇒ What is word-2-vec? Why first layer in NLP NN has to be an embedding layer?

Word2vec is a popular technique for learning word embeddings. These embeddings are numerical representations

of words that capture their semantic meaning & relationships to other words.
It essentially creates a vector space where words with similar meanings are positioned close together.

→The reasons why the first layer in an NLP neural network often uses an embedding layer:

1) High Dimensionality of Words: Raw text data represents words as one-hot vectors, where each dimension corresponds to a unique word in vocabulary.

2) Semantic: One-hot vectors don't capture the semantic relationship b/w words.
Eg King & Queen would be far apart

3) Learned Representation: Unlike one-hot vectors which are predefined, word embeddings are learned from data itself.

4) Efficiency & Generalization: Word embeddings condense this high-dimensional data into lower-dimensional space.
The embedding layer acts as a bridge b/w the raw text data & the higher level processing layers of a NN.
It takes one hot encoded words & maps them to their corresponding embedding vectors in the lower dimensional space.

19.Explain LSTM working principles along with the all the equations(same 17Q).

Unveiling the Power of LSTMs: A Deep Dive with Key Equations (8 Marks)

LSTMs (Long Short-Term Memory) networks are a powerful type of Recurrent Neural Network (RNN) designed to conquer the limitations of standard RNNs in handling long-term dependencies within sequential data. Unlike vanilla RNNs, LSTMs excel at remembering and leveraging information from distant past inputs, making them ideal for tasks like machine translation, speech recognition, and time series forecasting."

Core LSTM Architecture:

At the heart of an LSTM lies a complex cell that controls information flow through three critical gates:

1. **Forget Gate (ft):** This gate acts as a selective filter, deciding what information from the previous cell state (ct-1) to retain. It analyzes both the current input (xt) and the previous hidden state (ht-1) to determine which parts of the past memory are still relevant.
   Equation: ft = σ(Wf • [xt, ht-1] + bf)
   Here, σ represents the sigmoid activation function (squashes values between 0 and 1), Wf is the forget gate weight matrix, bf is the forget gate bias vector, and • denotes element-wise multiplication.

2. **Input Gate (it):** This gate controls the flow of new information from the current input (xt) into the cell state. It also interacts with the previous hidden state (ht-1) to decide what new information is valuable to store.
   Equation: it = σ(Wi • [xt, ht-1] + bi)
   Wi represents the input gate weight matrix and bi is the input gate bias vector.

3. **Output Gate (ot):** This gate determines what information from the current cell state (ct) is ultimately exposed as the output (ht) of the LSTM unit. It considers both the current cell state and the previous hidden state.
   Equation: ot = σ(Wo • [ct, ht-1] + bo)
   Wo represents the output gate weight matrix and bo is the output gate bias vector.

Cell State Update:

1. **Candidate Cell State (Ct'):** This step creates a temporary version of the new cell state based on the current input and the forget gate's filtered previous cell state.
   Equation: Ct' = tanh(Wc • [xt, ht-1] + bc)
   Wc represents the candidate cell state weight matrix and bc is the candidate cell state bias vector. tanh is the hyperbolic tangent activation function (outputs range between -1 and 1).

2. **New Cell State (ct):** The actual update of the cell state combines the filtered past information (ft * ct-1) with the newly created candidate information (it * Ct').
   Equation: ct = ft * ct-1 + it * Ct'

Output Generation:
The final output (ht) of the LSTM unit is derived from the current cell state (ct) passed through a sigmoid function and then multiplied element-wise with the output of the output gate (ot).
Equation: ht = ot * tanh(ct)

---

20. Vanishing gradient problem makes it difficult to train RNNs. How the LSTM dealt with that problem?

The Vanishing Gradient problem is a challenge in training Recurrent Neural Networks (RNNs) due to the backpropagation algorithm. As the gradients propagate through time, they can become very small, making it difficult for the network to learn long-term dependencies. This is particularly problematic for tasks such as language translation

and speech recognition, where understanding the context of a sequence is crucial.

Long Short-Term Memory (LSTM) networks were developed to address the Vanishing Gradient problem by introducing a memory cell and three gates: the input gate `i_t`, the forget gate `f_t`, and the output gate `o_t`. These gates help control the flow of information in the network, allowing it to maintain a hidden state that can capture long-term dependencies.

The LSTM cell state `c_t` is updated as follows:

```
c_t = f_t * c_{t-1} + i_t * W_c * h_{t-1}
```

where `c_t` is the cell state at time `t`, `f_t` is the forget gate, `c_{t-1}` is the previous cell state, `i_t` is the input gate, `W_c` is the weight matrix for the cell state, and `h_{t-1}` is the hidden state at the previous time step.

The hidden state `h_t` is then calculated as:

```
h_t = o_t * tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h)
```

where `h_t` is the hidden state at time `t`, `o_t` is the output gate, `W_h` is the weight matrix for the hidden state, `U_h` is the input weight matrix, `h_{t-1}` is the hidden state at the previous time step, and `b_h` is the bias term.

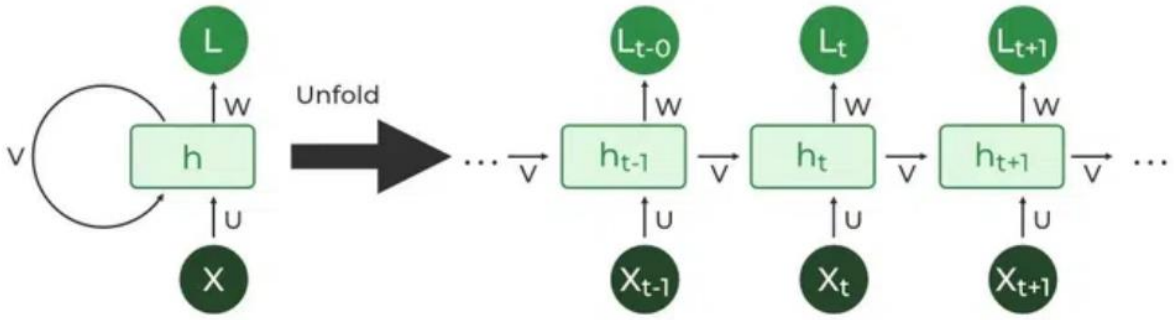
By using these gates and cell state updates, LSTM networks can effectively capture long-term dependencies in sequential data and overcome the Vanishing Gradient problem.

The Vanishing Gradient Problem

The vanishing gradient problem arises in deep neural networks during training. Gradients, which guide weight adjustments to minimize the gap between predicted and actual outcomes, tend to weaken as they traverse through layers. This weakening can slow down or even halt learning in the initial layers. In the context of Recurrent Neural Networks (RNNs), which process sequential data, this issue becomes pronounced.

Recurrent Neural Networks (RNNs)

RNNs maintain a hidden state that captures information from previous time steps within a sequence. Unlike traditional feedforward neural networks, where information flows linearly from input to output, RNNs loop back on themselves. The computation of the hidden state at a given time step follows this equation:



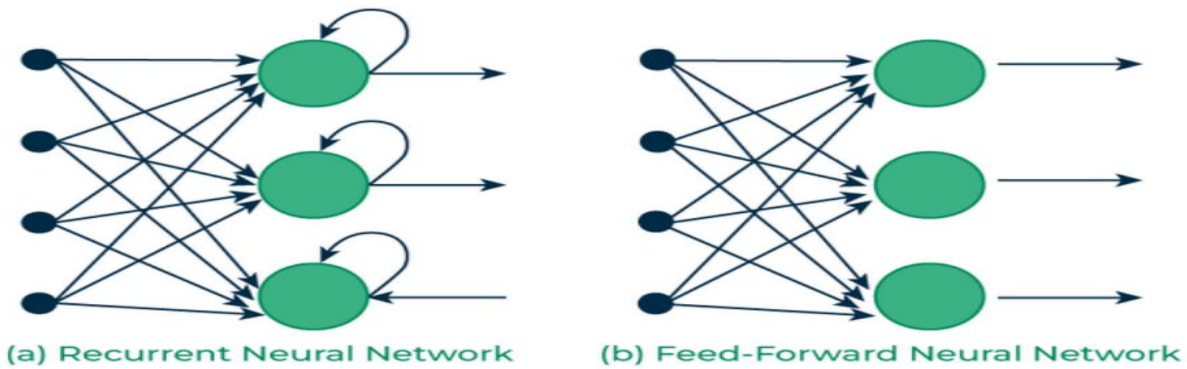
However, RNNs face challenges with gradient vanishing and computational efficiency when dealing with long sequences.

How LSTM Addresses the Problem

LSTMs, introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997, were specifically designed to overcome the vanishing gradient problem. Here's how they do it:

1. **Memory Cells and Gating Mechanisms**:
    - LSTMs use memory cells to retain information over longer sequences.
    - They incorporate gating mechanisms (input, output, and forget gates) to control the flow of information and gradients.
    - These gates allow LSTMs to selectively update and retain information, preventing the vanishing gradient issue.
2. **Constant Error Carousel (CEC)**:
    - The loop around the cell in an LSTM corresponds to the CEC.
    - This part can be seen as a sort of identity function, ensuring that the derivative remains close to 1. Consequently, the gradient doesn't vanish due to this component.
3. **Activation Functions**:
    - While LSTMs use sigmoid activation functions for input, output, and forget gates (whose derivatives are at most 0.25), they also use the hyperbolic tangent (tanh) activation function for other components (traditionally (g) and (h)).
    - Backpropagating through these functions ensures that the gradient doesn't vanish exponentially.

In summary, LSTMs avoid the vanishing gradient problem by using memory cells, gates, and activation functions designed to preserve gradients during backpropagation. This makes them efficient at remembering long-term dependencies and robust against the issue.



The vanishing gradient problem occurs when the gradients during backpropagation become very small, making it difficult for the network to learn long-term dependencies. This is because the gradients are calculated by multiplying the gradients of the previous time steps, and as the number of time steps increases, the gradients can become very small, effectively vanishing.

To overcome the vanishing gradient problem, LSTMs were introduced. LSTMs are a type of RNN that uses a memory cell and three gates: the input gate `i_t`, the forget gate `f_t`, and the output gate `o_t`. These gates help control the flow of information in the network, allowing it to maintain a hidden state that can capture long-term dependencies.

The LSTM cell state `c_t` is updated as follows:

```
c_t = f_t * c_{t-1} + i_t * W_c * h_{t-1}
```

where `c_t` is the cell state at time `t`, `f_t` is the forget gate, `c_{t-1}` is the previous cell state, `i_t` is the input gate, `W_c` is the weight matrix for the cell state, and `h_{t-1}` is the hidden state at the previous time step.

The hidden state `h_t` is then calculated as:

```
h_t = o_t * tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h)
```

where `h_t` is the hidden state at time `t`, `o_t` is the output gate, `W_h` is the weight matrix for the hidden state, `U_h` is the input weight matrix, `h_{t-1}` is the hidden state at the previous time step, and `b_h` is the bias term.

By using these gates and cell state updates, LSTM networks can effectively capture long-term dependencies in sequential data and overcome the vanishing gradient problem.

---

21. List the architectures capable of learning from sequential data. What makes them different from others?

Long short-term memory
Type of recurrent neural network
Definition
A recurrent neural network (RNN) designed to address the vanishing gradient problem in traditional RNNs.
Advantage
Relative insensitivity to gap length, making it superior to other RNNs and sequence learning methods.
Function

Provides short-term memory for RNN that can last thousands of timesteps, applicable to classification, processing, predicting time series data.
Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs) are architectures capable of learning from sequential data. They differ from other architectures due to their unique ability to maintain a hidden state or memory, which allows them to capture sequential dependencies by remembering previous inputs while processing.
RNNs, LSTMs, and GRUs are all designed to handle sequential data by using the same parameters for each input and performing the same task on all inputs or hidden layers to produce the output. They are particularly useful for tasks such as predicting the next word in a sentence, where the previous words are required.
LSTMs and GRUs are improved versions of RNNs that enhance their ability to handle long-term dependencies. LSTMs introduce a memory cell, which is a container that can hold information for an extended period, making them well-suited for sequence prediction tasks. They excel in capturing long-term dependencies, which is crucial for solving complex problems such as machine translation and speech recognition.
Compared to feedforward neural networks, which have no looping nodes and do not retain previous inputs, RNNs, LSTMs, and GRUs are more suitable for sequential data analysis. They maintain a hidden state that remembers previous inputs, allowing them to capture sequential dependencies and handle long-term dependencies more effectively.
Here are some key equations and concepts related to these architectures:
RNNs: RNNs use the following recurrence relation to update the hidden state at each time step:
h_t = f(W * h_{t-1} + U * x_t + b)

where h_t is the hidden state at time t, f is an activation function, W is the weight matrix, U is the input weight matrix, x_t is the input at time t, and b is the bias term.
LSTMs: LSTMs use three gates to control the flow of information: the input gate i_t, the forget gate f_t, and the output gate o_t. The cell state c_t is updated as follows:
c_t = f_t * c_{t-1} + i_t * W_c * h_{t-1}

The hidden state h_t is then calculated as:
h_t = o_t * tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h)

where W_c, U_h, and b_h are weight and bias matrices for the LSTM cells.
GRUs: GRUs use a reset gate r_t and an update gate z_t to control the flow of information. The cell state c_t is updated as:
c_t = (1 - z_t) * c_{t-1} + z_t * W_c * h_{t-1}

The hidden state h_t is then calculated as:
h_t = tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h) * (1 - r_t) + r_t * h_{t-1}

where W_c, U_h, and b_h are weight and bias matrices for the GRU cells.
These architectures are designed to handle sequential data, making them suitable for tasks such as language translation, speech recognition, and time series prediction. They maintain a hidden state that remembers previous inputs, allowing them to capture sequential dependencies and handle long-term dependencies more effectively than feedforward neural networks.

Architectures capable of learning from sequential data include Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs). These architectures differ from others due to their unique ability to maintain a hidden state or memory, allowing them to capture sequential dependencies by remembering previous inputs while processing. This is particularly useful for tasks such as predicting the next word in a sentence, where the previous words are required.

RNNs have a hidden state, also referred to as the memory state, which remembers some information about a sequence. They are designed to handle sequential data by using the same parameters for each input and performing the same task on all inputs or hidden layers to produce the output. LSTMs and GRUs are improved versions of RNNs that enhance their ability to handle long-term dependencies.

LSTMs introduce a memory cell, which is a container that can hold information for an extended period, making them well-suited for sequence prediction tasks. They excel in capturing long-term dependencies, which is crucial for solving complex problems such as machine translation and speech recognition.

Compared to feedforward neural networks, which have no looping nodes and do not retain previous inputs, RNNs, LSTMs, and GRUs are more suitable for sequential data analysis. They maintain a hidden state that remembers previous inputs, allowing them to capture sequential dependencies and handle long-term dependencies more effectively.

Long short-term memory

Type of recurrent neural network

More

Definition	A recurrent neural network (RNN) designed to address the vanishing gradient problem in traditional RNNs.
Advantage	Relative insensitivity to gap length, making it superior to other RNNs and sequence learning methods.
Function	Provides short-term memory for RNN that can last thousands of timesteps, applicable to classification, processing, predicting time series data.

Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs) are architectures capable of learning from sequential data. They differ from other architectures due to their unique ability to maintain a hidden state or memory, which allows them to capture sequential dependencies by remembering previous inputs while processing.

RNNs, LSTMs, and GRUs are all designed to handle sequential data by using the same parameters for each input and performing the same task on all inputs or hidden layers to produce the output. They are particularly useful for tasks such as predicting the next word in a sentence, where the previous words are required.

LSTMs and GRUs are improved versions of RNNs that enhance their ability to handle long-term dependencies. LSTMs introduce a memory cell, which is a container that can hold information for an extended period, making them well-suited for sequence prediction tasks. They excel in capturing long-term dependencies, which is crucial for solving complex problems such as machine translation and speech recognition.

Compared to feedforward neural networks, which have no looping nodes and do not retain previous inputs, RNNs, LSTMs, and GRUs are more suitable for sequential data analysis. They maintain a hidden state that remembers previous inputs, allowing them to capture sequential dependencies and handle long-term dependencies more effectively.

Here are some key equations and concepts related to these architectures:

- **RNNs:** RNNs use the following recurrence relation to update the hidden state at each time step:

```
h_t = f(W * h_{t-1} + U * x_t + b)
```

where `h_t` is the hidden state at time `t`, `f` is an activation function, `W` is the weight matrix, `U` is the input weight matrix, `x_t` is the input at time `t`, and `b` is the bias term.

- **LSTMs:** LSTMs use three gates to control the flow of information: the input gate `i_t`, the forget gate `f_t`, and the output gate `o_t`. The cell state `c_t` is updated as follows:

```
c_t = f_t * c_{t-1} + i_t * W_c * h_{t-1}
```

The hidden state `h_t` is then calculated as:

```
h_t = o_t * tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h)
```

where `W_c`, `U_h`, and `b_h` are weight and bias matrices for the LSTM cells.

- **GRUs:** GRUs use a reset gate `r_t` and an update gate `z_t` to control the flow of information. The cell state `c_t` is updated as:

```
c_t = (1 - z_t) * c_{t-1} + z_t * W_c * h_{t-1}
```

The hidden state `h_t` is then calculated as:

```
h_t = tanh(W_h * h_{t-1} + U_h * h_{t-1} + b_h) * (1 - r_t) + r_t * h_{t-
```

where `W_c`, `U_h`, and `b_h` are weight and bias matrices for the GRU cells.

These architectures are designed to handle sequential data, making them suitable for tasks such as language translation, speech recognition, and time series prediction. They maintain a hidden state that remembers previous inputs, allowing them to capture sequential dependencies and handle long-term dependencies more effectively than feedforward neural networks.