1 Backpropagation

1.1 Derivation

Introduction Backpropagation is a key algorithm for training artificial neural networks by minimizing the error between predicted and target outputs using gradient descent. This derivation outlines the process of adjusting weights layer by layer.

Define the Error The total error E in the network is the sum of squared differences between target values t_k and actual outputs a_k across all output nodes k:

$$E = \frac{1}{2} \sum_k (t_k - a_k)^2$$

The factor $\frac{1}{2}$ simplifies later derivative calculations. The goal is to adjust weights to minimize E.

Weight Change Rule (Gradient Descent) To minimize E, we update each weight w by a small step in the opposite direction of the gradient of E with respect to that weight:

$$\Delta w_{jk} \propto -\frac{\partial E}{\partial w_{jk}}$$

Incorporating the learning rate ε, the weight change is:

$$\Delta w_{jk} = -\varepsilon \frac{\partial E}{\partial w_{jk}}$$

Chain Rule Breakdown Since E is not directly a function of w_{jk}, we use the chain rule:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{jk}}$$

Substitute into the weight change equation:

$$\Delta w_{jk} = -\varepsilon \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{jk}}$$

Compute Each Partial Derivative Derivative of Error with Respect to Activation ($\frac{\partial E}{\partial a_k}$) For one output node:

$$E_k = \frac{1}{2}(t_k - a_k)^2$$

Derivative with respect to a_k:

$$\frac{\partial E_k}{\partial a_k} = \frac{\partial}{\partial a_k}\left[\frac{1}{2}(t_k - a_k)^2\right] = \frac{1}{2} \cdot 2(t_k - a_k) \cdot (-1) = -(t_k - a_k)$$

This shows the error's change depends on the difference between target and output.

Derivative of Activation with Respect to Net Input ($\frac{\partial a_k}{\partial net_k}$) Assuming a sigmoid activation function $a_k = \frac{1}{1+e^{-net_k}}$:

$$\frac{\partial a_k}{\partial net_k} = \frac{\partial}{\partial net_k}\left[(1 + e^{-net_k})^{-1}\right] = \frac{e^{-net_k}}{(1 + e^{-net_k})^2}$$

Rewrite in terms of a_k:

$$1 - a_k = \frac{e^{-net_k}}{1 + e^{-net_k}}, \quad \frac{e^{-net_k}}{(1 + e^{-net_k})^2} = a_k(1 - a_k)$$

So, $\frac{\partial a_k}{\partial net_k} = a_k(1 - a_k)$.

Derivative of Net Input with Respect to Weight ($\frac{\partial net_k}{\partial w_{jk}}$) The net input is $net_k = \sum_j w_{jk} a_j$. For w_{jk}:

$$\frac{\partial net_k}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}(w_{jk} a_j) = a_j$$

This depends on the activation a_j from the hidden node.

Weight Change Rule for Hidden-to-Output Weight Substitute the derivatives:

$$\Delta w_{jk} = -\varepsilon\left[-(t_k - a_k)\right] \cdot \left[a_k(1 - a_k)\right] \cdot \left[a_j\right]$$

Simplify:

$$\Delta w_{jk} = \varepsilon(t_k - a_k) a_k (1 - a_k) a_j$$

This adjusts w_{jk} based on error, output sensitivity, and hidden activation.

Weight Change for Input-to-Hidden Weight For w_{ji}, the error at the hidden node depends on all output nodes:

$$\frac{\partial E}{\partial w_{ji}} = \sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial a_j} \cdot \frac{\partial a_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

- $\frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial net_k} = -(t_k - a_k)a_k(1 - a_k)$. - $\frac{\partial net_k}{\partial a_j} = w_{kj}$. - $\frac{\partial a_j}{\partial net_j} = a_j(1 - a_j)$. - $\frac{\partial net_j}{\partial w_{ji}} = a_i$. Combine:

$$\Delta w_{ji} = -\varepsilon \sum_k [-(t_k - a_k)a_k(1 - a_k)w_{kj}] \cdot [a_j(1 - a_j)a_i]$$

Define $\delta_k = (t_k - a_k)a_k(1 - a_k)$ and $\delta_j = \sum_k \delta_k w_{kj} a_j(1 - a_j)$:

$$\Delta w_{ji} = \varepsilon \delta_j a_i$$

Conclusion Backpropagation uses the chain rule to propagate errors backward, adjusting weights to minimize E. The derivation differs for hidden-to-output and input-to-hidden weights due to error propagation across layers.

2 Convolution and CNN

2.1 Introduction to Convolution

Convolution is a mathematical operation used in signal and image processing to extract features by applying a filter (or kernel) over an input. In the context of neural networks, this operation forms the foundation of Convolutional Neural Networks (CNNs), which are specialized for processing structured grid-like data such as images.

2.2 CNN

A Convolutional Neural Network (CNN) is a type of Deep Learning neural network architecture commonly used in Computer Vision. CNNs are particularly effective for tasks like image classification, object detection, and facial recognition due to their ability to capture spatial hierarchies and patterns. Convolutional Neural Network consists of multiple layers.

2.3 Uses of Convolutional Neural Networks

CNNs are widely applied in various domains, including:

- Image classification (e.g., identifying objects in photographs).
- Object detection (e.g., locating multiple objects within an image).
- Facial recognition (e.g., identifying individuals in images or videos).
- Image Segmentation (Dividing an image into segments to make it easier to analyze.)
- Medical image analysis (e.g., detecting abnormalities in X-rays or MRIs).
- Video Analysis (Analyzing sequences of images (frames) to detect activities or events.)

2.4 Different Layers in CNN

A CNN comprises several distinct layers, each serving a specific purpose:

1. **Input Layer**: Receives the raw input data, such as pixel values of an image, and passes it to subsequent layers for processing.
2. **Convolution Layer**: Applies convolution operations using filters to extract features like edges, textures, or patterns from the input data.
3. **Activation Layer**: Introduces non-linearity (e.g., ReLU) to enable the network to learn complex patterns by transforming the convolved output.
4. **Pooling Layer**: Reduces the spatial dimensions (e.g., max pooling) of the feature maps, retaining important information while decreasing computational load.
5. **Fully Connected Layer**: Connects all neurons from the previous layer to produce high-level reasoning, consolidating features for final decision-making.
6. **Output Layer**: Generates the final output, such as class probabilities or a classification label, based on the processed features.

3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data by maintaining a memory of previous inputs through recurrent connections.

3.1 Uses of Recurrent Neural Networks

RNNs are applied in various domains, including:

- Natural language processing (e.g., language modeling, text generation).
- Time series prediction (e.g., stock price forecasting).
- Speech recognition (e.g., converting audio to text).
- Machine translation (e.g., translating sentences between languages).

3.2 Vanishing and Exploding Gradients Problems

During backpropagation through time (BPTT) in RNNs, the following issues arise:

- **Vanishing Gradients**: Gradients become extremely small as they are propagated back through many time steps, causing early layers to learn slowly or not at all.
- **Exploding Gradients**: Gradients grow excessively large, leading to unstable training and large weight updates that disrupt learning.

These problems stem from the repeated multiplication of gradients over time, often exacerbated by the choice of activation functions (e.g., sigmoid or tanh).

3.3 Solutions to Vanishing and Exploding Gradients

Several techniques address these issues:

- **Gradient Clipping**: Limits the gradient magnitude during backpropagation to prevent exploding gradients by capping values above a threshold.
- **Long Short-Term Memory (LSTM) Units**: Introduces memory cells and gates (input, forget, output) to preserve long-term dependencies, mitigating vanishing gradients.
- **Gated Recurrent Units (GRU)**: A simplified version of LSTM with update and reset gates, improving gradient flow and reducing vanishing gradient effects.
- **ReLU Activation**: Replaces traditional activations to avoid the saturation issues that contribute to vanishing gradients.

3.4 Architectural Differences Between LSTM and GRU

Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) are advanced recurrent neural network architectures designed to mitigate the vanishing gradient problem. This document compares their architectural differences in a tabular format.

Aspects	**LSTM**	**GRU**
Gating Mechanisms	Uses three gates: input gate, forget gate, and output gate, along with a separate cell state and hidden state to control information flow.	Uses two gates: update gate and reset gate, with a single hidden state that handles both memory and update functions.
Number of Parameters	Higher due to the separate memory cell and three gates, increasing computational complexity.	Lower as it combines the forget and input functions into the update gate, reducing computational cost.
Memory Cell	Maintains a distinct cell state to preserve long-term memory, updated through the gates.	Lacks a separate cell state; the hidden state manages both short- and long-term memory.
Complexity and Training	More complex due to additional gates, potentially better for long sequences but slower to train.	Simpler and faster to train, often performing comparably on shorter sequences.

4 Transfer Learning

4.1 Definition

Transfer learning is a machine learning technique where a model trained on a large, general dataset is fine-tuned for a specific task with a smaller dataset.

Transfer learning is a machine learning technique where knowledge gained from solving one task is leveraged to improve a model's performance on a different but related task. Instead of training a model from scratch for every new problem, transfer learning uses a pre-trained model—typically trained on a large dataset for a source task—and adapts it (often by fine-tuning) for a target task, which may have less data or slightly different requirements

4.2 Usage

Application	Description
Image Classification	Adapts pre-trained models (e.g., ResNet) for new image categories with limited data.
Natural Language Processing	Fine-tunes models like BERT for tasks such as sentiment analysis or question answering.
Medical Imaging	Utilizes pre-trained models to detect diseases in X-rays or MRIs with small labeled datasets.
Speech Recognition	Adjusts pre-trained audio models for specific languages or accents.

5 Transformer Model

5.1 Definition

The Transformer model is a deep learning architecture introduced for sequence-to-sequence tasks, relying entirely on attention mechanisms and eliminating recurrent structures.

A transformer model is a type of neural network architecture that has revolutionized the field of artificial intelligence, especially in natural language processing (NLP) and other sequential data tasks. Introduced in the 2017 paper "Attention is All You Need," transformers have become the foundation for many modern AI systems, including large language models and generative AI.

5.2 Attention Mechanism in Transformer Architecture

Aspect	Description
Self-Attention	Allows the model to weigh the importance of different words in a sequence, capturing contextual relationships.
Multi-Head Attention	Uses multiple attention mechanisms in parallel to focus on different parts of the input simultaneously.
Scaled Dot-Product Attention	Computes attention scores by scaling the dot product of query and key vectors, improving gradient stability.

5.3 Transformer vs RNN

Aspects	Transformer	RNN
Architecture	Based on attention mechanisms, no recurrence.	Relies on sequential recurrence with hidden states.
Parallelization	Highly parallelizable, faster training on large data.	Sequential processing, slower due to time dependency.
Long-Term Dependencies	Effectively captures long-range dependencies via attention.	Struggles with vanishing/exploding gradients over long sequences.
Memory Usage	Fixed memory, independent of sequence length.	Memory grows with sequence length due to recurrence.