Machine Learning 2

Gergo Nador[1] and **Frederik Thorning Bjørn**[2]

[1]A student preparing for the exam
[2]The teacher waiting to fail the students

January 2, 2025

Contents

1 Introduction

1.1 What is Deep Learning?

1.1.1 Artificial Intelligence (AI)

Artificial intelligence refers to systems that can perform tasks typically associated with human intelligence.

1.1.2 Machine Learning (ML)

Machine learning is a subset of AI, encompassing systems that derive patterns from data without being explicitly programmed.

1.1.3 Deep Learning (DL)

Deep learning is a specialized area of machine learning that employs neural networks with multiple layers to model and understand complex patterns.

1.2 Key Concepts in Neural Networks

1.2.1 Neurons, Weights, and Biases

Neurons are the basic computational units of a neural network that process input data. Each neuron performs a weighted summation of its inputs and adds a bias term before applying an activation function to produce an output. The **weights** control the influence of each input on the neuron, while the **bias** allows the neuron to shift the activation function, providing flexibility in decision boundaries. These parameters are learned during the training process to minimize the network's error.

1.2.2 The Structure of a Neural Network

A neural network consists of an interconnected arrangement of layers: an **input layer**, one or more **hidden layers**, and an **output layer**. The input layer receives raw data, the hidden layers perform intermediate computations, and the output layer generates the final prediction. Connections between neurons are governed by weights, and information flows from one layer to the next, with each neuron applying an activation function to its inputs.

1.2.3 Activation Functions

Activation functions introduce non-linearity into the computations of a neural network, enabling it to model complex patterns in data. Examples include:

1. **Sigmoid**: Maps the output between 0 and 1.
2. **ReLU (Rectified Linear Unit)**: Outputs positive values and suppresses negatives.
3. **Tanh**: Maps the output between -1 and 1.

These functions allow the network to approximate non-linear relationships and are essential for learning.

1.2.4 Gradient Descent

Gradient descent is an optimization technique used to minimize the loss function by iteratively updating the weights and biases. It calculates the gradient of the loss function with respect to each parameter and adjusts the parameters in the direction opposite to the gradient to reduce error. Variants such as **Stochastic Gradient Descent (SGD)** and **Adam** improve efficiency and convergence, especially for large datasets.

1.2.5 Loss Functions

Loss functions measure the error between the network's predictions and the true target values. Common loss functions include:

1. **Mean Squared Error (MSE)**: Used for regression tasks, it calculates the average squared difference between predicted and actual values.
2. **Cross-Entropy Loss**: Used for classification tasks, it measures the difference between predicted probability distributions and true labels.

The loss function guides the training process by providing feedback to adjust the network's weights and biases.

1.3 Gradient Descent

Gradient descent is an optimization algorithm used to minimize a given loss function $\mathcal{L}(\theta)$, where θ represents the parameters of a model. The method updates the parameters iteratively by moving in the direction of the negative gradient of the loss function.

1.3.1 Gradient of the Loss Function

The gradient of the loss function is defined as the vector of partial derivatives with respect to all parameters:

$$\nabla\mathcal{L}(\theta) = \left[\frac{\partial\mathcal{L}}{\partial\theta_0} \quad \frac{\partial\mathcal{L}}{\partial\theta_1} \quad \cdots \quad \frac{\partial\mathcal{L}}{\partial\theta_n}\right]. \tag{1}$$

1.3.2 Parameter Update Rule

The parameters are updated in the direction of the negative gradient as follows:

$$\theta \leftarrow \theta - \eta\nabla\mathcal{L}(\theta), \tag{2}$$

where η is the learning rate, a scalar hyperparameter controlling the step size.

The iterative process stops when the gradient $\nabla\mathcal{L}(\theta) \approx 0$, indicating convergence to a minimum.

1.4 Gradient Descent - Numerical Approximation

In some cases, the gradient may be approximated using finite differences:

$$\frac{\partial\mathcal{L}}{\partial\theta_j} \approx \frac{\mathcal{L}(\theta_j + \epsilon) - \mathcal{L}(\theta_j)}{\epsilon}, \tag{3}$$

where ϵ is a small constant. This method estimates the slope of the loss function based on discrete points.

1.5 Gradient Descent in Neural Networks

In neural networks, gradient descent involves optimizing the loss function across multiple parameters. For example, in a simple network with input, hidden, and output layers, the loss function may take the form of Mean Squared Error (MSE):

$$\mathcal{L} = (\hat{y} - y)^2, \tag{4}$$

where $\hat{y}$ is the predicted output and y is the true label.

To compute gradients for all parameters, the chain rule is applied. For instance:

$$\frac{\partial\mathcal{L}}{\partial w_{21}^{(1)}} = \frac{\partial\mathcal{L}}{\partial\hat{y}} \cdot \frac{\partial\hat{y}}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_{21}^{(1)}}. \tag{5}$$

1.5.1 Forward Pass Equations

In the forward pass of a neural network:

$$\hat{y} = a_1 w_{11}^{(2)} + a_2 w_{21}^{(2)} + b^{(2)}, \tag{6}$$

$$a_1 = \sigma(z_1), \quad a_2 = \sigma(z_2), \tag{7}$$

$$z_1 = x_1 w_{11}^{(1)} + x_2 w_{21}^{(1)} + b_1^{(1)}, \tag{8}$$

$$z_2 = x_1 w_{12}^{(1)} + x_2 w_{22}^{(1)} + b_2^{(1)}, \tag{9}$$

where σ is the activation function, and w and b represent weights and biases.

This process becomes more complex for larger networks with multiple paths converging on the same parameters, but the fundamental principles remain the same.

1.6 The Backpropagation Algorithm

The backpropagation algorithm is a supervised learning technique used in artificial neural networks to compute the gradient of the loss function with respect to the model parameters. It relies on the chain rule of differentiation to efficiently propagate errors backward through the network.

1.6.1 Steps in Backpropagation

The backpropagation algorithm can be divided into three main stages:

1. **Forward Pass:** During this step, data flows through the network, layer by layer. The model computes the predictions $\hat{y}$ based on the input x and the current set of parameters θ. These predictions are compared to the true labels y to compute the loss $\mathcal{L}$.

2. **Backward Pass:** The backward pass propagates the error signal through the network using the chain rule. The goal is to compute the partial derivatives of the loss function $\mathcal{L}$ with respect to each parameter in the network. For a parameter θ, this is represented as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \theta}. \tag{10}$$

   This involves calculating derivatives layer by layer, starting from the output and moving toward the input, in a process often referred to as "backpropagating the error."

3. **Parameter Update:** Once the gradients $\nabla \mathcal{L}(\theta)$ are computed, the parameters are updated using an optimization algorithm, typically gradient descent. The update rule is:

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}(\theta), \tag{11}$$

   where η is the learning rate.

1.6.2 Key Components of Backpropagation

- **Activation Functions:** Non-linear activation functions, such as ReLU or sigmoid, are applied at each neuron. Their derivatives are crucial for propagating gradients backward through the network.

- **Chain Rule:** The chain rule is used to compute the gradient of the loss function with respect to each parameter. This is done recursively from the output layer to the input layer.

- **Gradient Representation:** Gradients are often organized in matrix or vector form for computational efficiency. For example:

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} \\ \frac{\partial \mathcal{L}}{\partial w_{21}^{(1)}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial b_1^{(2)}} \end{bmatrix}. \tag{12}$$

1.6.3 Iterative Process

Backpropagation is an iterative process. After each forward and backward pass, the parameters are updated, and the process repeats until convergence. Convergence occurs when the loss function $\mathcal{L}$ reaches a minimum or when the gradient magnitude becomes sufficiently small.

1.6.4 Efficiency Considerations

Backpropagation significantly reduces computational cost by reusing intermediate results (e.g., activations and partial derivatives) during the backward pass. This makes it feasible to train large neural networks with millions of parameters.

1.6.5 Overall Workflow

The backpropagation process can be summarized as follows:

1. Perform a forward pass to compute predictions and loss.
2. Calculate gradients of the loss function with respect to each parameter through the backward pass.
3. Update parameters using an optimization method.
4. Repeat the process until the network converges.

1.7 Vanishing and Exploding Gradients

In deep neural networks, the process of backpropagation involves computing gradients of the loss function with respect to the network parameters. These gradients are then used to update the parameters. However, due to the structure and depth of the network, the gradients can either become very small (vanishing gradients) or very large (exploding gradients), leading to instability in training.

1.7.1 Vanishing Gradients

The vanishing gradient problem occurs when gradients become progressively smaller as they are propagated backward through the layers of the network. This results in:

- **Insufficient Parameter Updates:** Parameters in the earlier layers of the network are hardly updated, as their gradients approach zero.
- **Stalled Learning:** The lower layers of the network effectively stop learning, while the later layers continue to adjust.

Mathematically, this problem arises due to repeated application of the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \prod_i \frac{\partial a_i}{\partial z_i}, \tag{13}$$

where a_i are activations and z_i are pre-activations. When the derivatives $\frac{\partial a_i}{\partial z_i}$ are small (e.g., for activation functions like sigmoid), their repeated multiplication causes the gradients to shrink exponentially.

1.7.2 Exploding Gradients

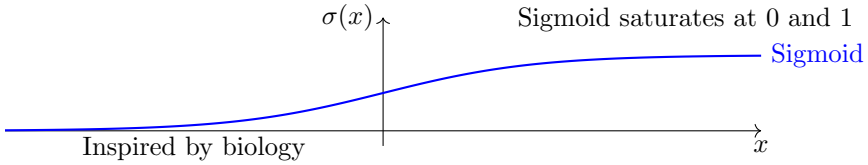
The exploding gradient problem occurs when gradients become excessively large as they are propagated backward. This results in:

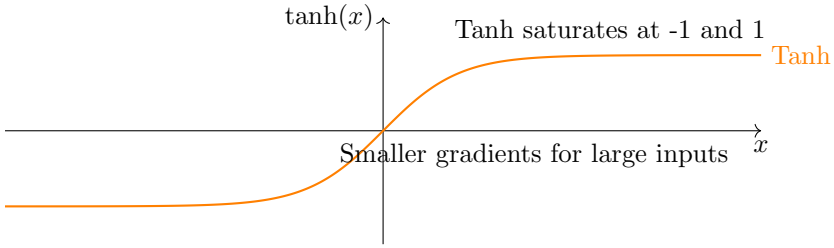
- **Unstable Training:** Parameters experience very large updates, leading to oscillations or divergence of the loss function.
- **Numerical Instability:** Extremely large gradient values can cause overflow or instability in numerical computations.

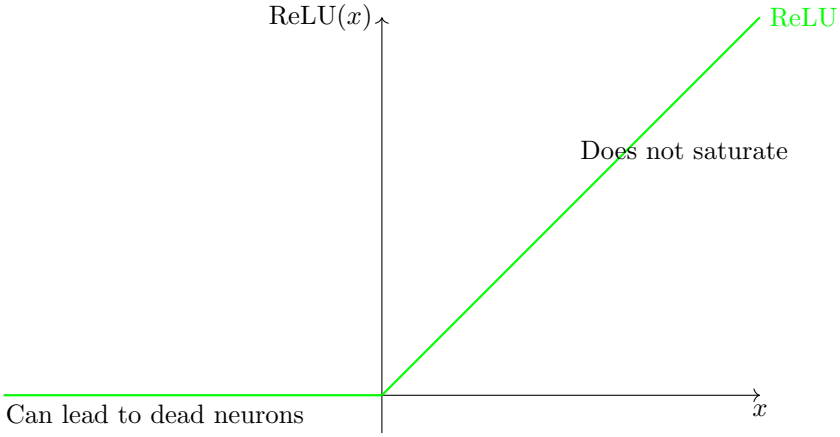
This issue is caused when the derivatives $\frac{\partial a_i}{\partial z_i}$ are consistently greater than 1, leading to exponential growth in the gradients.

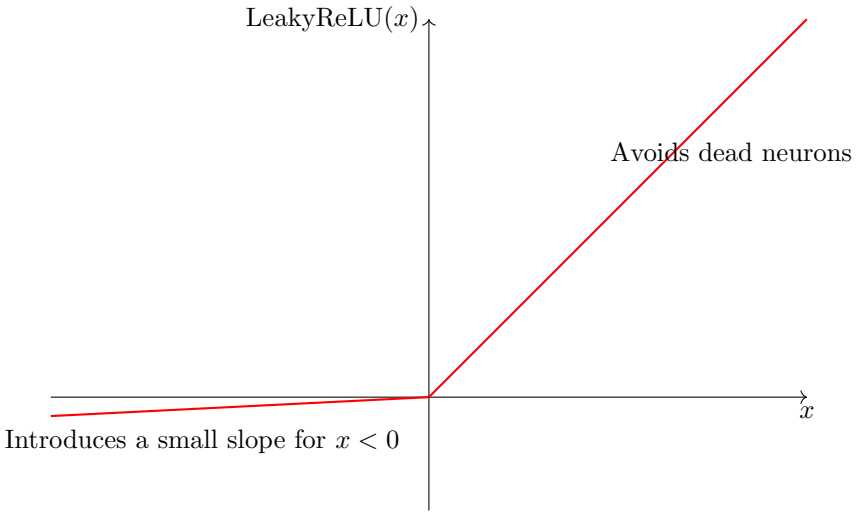
1.7.3 Impact on Training

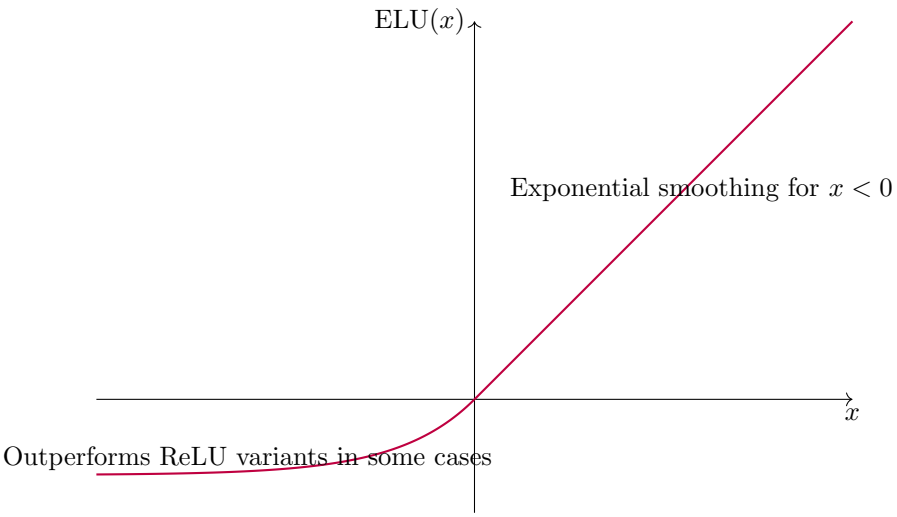
Both vanishing and exploding gradients affect the ability of deep neural networks to converge during training. The deeper the network, the more pronounced these issues become. Layers closer to the input are more susceptible to vanishing gradients, while exploding gradients can destabilize the entire training process.

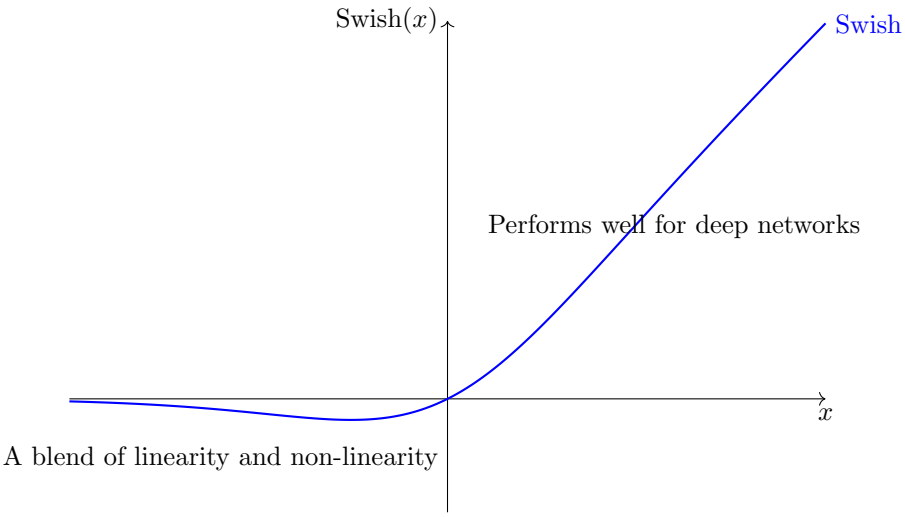












1.8 Optimization Algorithms for Machine Learning

1.8.1 Stochastic Gradient Descent (SGD)

Definition: Stochastic Gradient Descent is an optimization algorithm that updates model parameters by calculating the gradient of the loss function with respect to each parameter using a subset (or single sample) of the data at each step. While computationally efficient, its convergence can be slow without additional techniques.

1.8.2 Momentum

Definition: Momentum is an extension of SGD that accelerates convergence by considering the previous gradients' direction and applying a damping effect. It computes updates as a linear combination of the current gradient and the previously accumulated momentum, thereby reducing oscillations and improving the convergence rate. This introduces a velocity term that smoothens the updates.

$$m \leftarrow \beta m - \eta \nabla L(\theta) \tag{14}$$

$$\theta \leftarrow \theta + m \tag{15}$$

Here:

1. m: The momentum term.

2. β: The momentum coefficient, controlling the influence of past gradients.

3. η: The learning rate.

4. $\nabla L(\theta)$: The gradient of the loss function with respect to parameters θ.

1.8.3 Nesterov Accelerated Gradient (NAG)

Definition: NAG improves upon Momentum by calculating the gradient not at the current position but at the anticipated future position, offering better convergence by adapting the step direction.

$$m \leftarrow \beta m - \eta \nabla L(\theta + \beta m) \tag{16}$$
$$\theta \leftarrow \theta + m \tag{17}$$

Here:

1. The gradient is evaluated at $\theta + \beta m$, predicting the future position based on the momentum term.

2. This adjustment ensures better alignment of updates with the loss surface's true gradient direction.

1.8.4 Adaptive Gradient (AdaGrad)

Definition: AdaGrad is an adaptive learning rate optimization algorithm that adjusts the learning rate for each parameter based on its historical gradient magnitude, favoring sparse features. However, it can suffer from decaying learning rates.

Idea: Reduce the learning rate in the steepest directions to ensure more stable and efficient convergence.

$$s \leftarrow s + \nabla L(\theta) \odot \nabla L(\theta) \tag{18}$$
$$\theta \leftarrow \theta - \frac{\eta \nabla L(\theta)}{\sqrt{s} + \epsilon} \tag{19}$$

Here:

1. s: Accumulated squared gradients (elementwise).

2. $\odot$: Elementwise multiplication operator.

3. η: Learning rate.

4. $\nabla L(\theta)$: Gradient of the loss function with respect to parameters θ.

5. ϵ: A small constant added to prevent division by zero.

Note: While effective, methods like AdaGrad may scale down the learning rate excessively, potentially causing the training process to stagnate prematurely.

1.8.5 Scaling Decay (Root Mean Squared Propagation RMSProp)

Definition: Scaling decay is a technique used in optimization algorithms to add a decay factor to the accumulated scaling term. This ensures that the scaling factor does not grow excessively, which could lead to instability or premature cessation of the training process. RMSProp is an adaptive learning rate method that scales the gradients by the square root of an exponentially decaying average of their squared values, mitigating the vanishing learning rate problem in AdaGrad.

Modified Formula (AdaGrad) for RMSProp:

$$s \leftarrow \rho s + (1 - \rho) \nabla L(\theta) \odot \nabla L(\theta) \tag{20}$$
$$\theta \leftarrow \theta - \frac{\eta \nabla L(\theta)}{\sqrt{s} + \epsilon} \tag{21}$$

Here:

1. ρ: Decay rate, typically set to 0.9.

2. s: Exponentially decayed accumulation of squared gradients.

3. η: Learning rate.

4. ϵ: A small constant added for numerical stability.

5. $\odot$: Elementwise multiplication.

This modification prevents the scale factor from exploding, thus maintaining the stability and efficiency of the optimization process.

1.8.6 Adaptive Moment Estimation (Adam)

1.8.7 Adam with Weight Decay (AdamW)

Definition: AdamW introduces weight decay regularization into Adam to address overfitting, ensuring the parameter update considers both optimization and regularization effects.

Definition: Weight decay is a regularization technique applied during the training process to discourage large weight values. At each iteration, all parameters are multiplied by a constant factor slightly less than 1 (e.g., 0.99). This introduces an implicit penalty term to the loss function, effectively reducing overfitting and improving generalization.

Formula:

$$\theta \leftarrow \lambda\theta - \eta\nabla L(\theta) \tag{22}$$

Here:

1. λ: Decay factor, typically close to 1 (e.g., 0.99).

2. η: Learning rate.

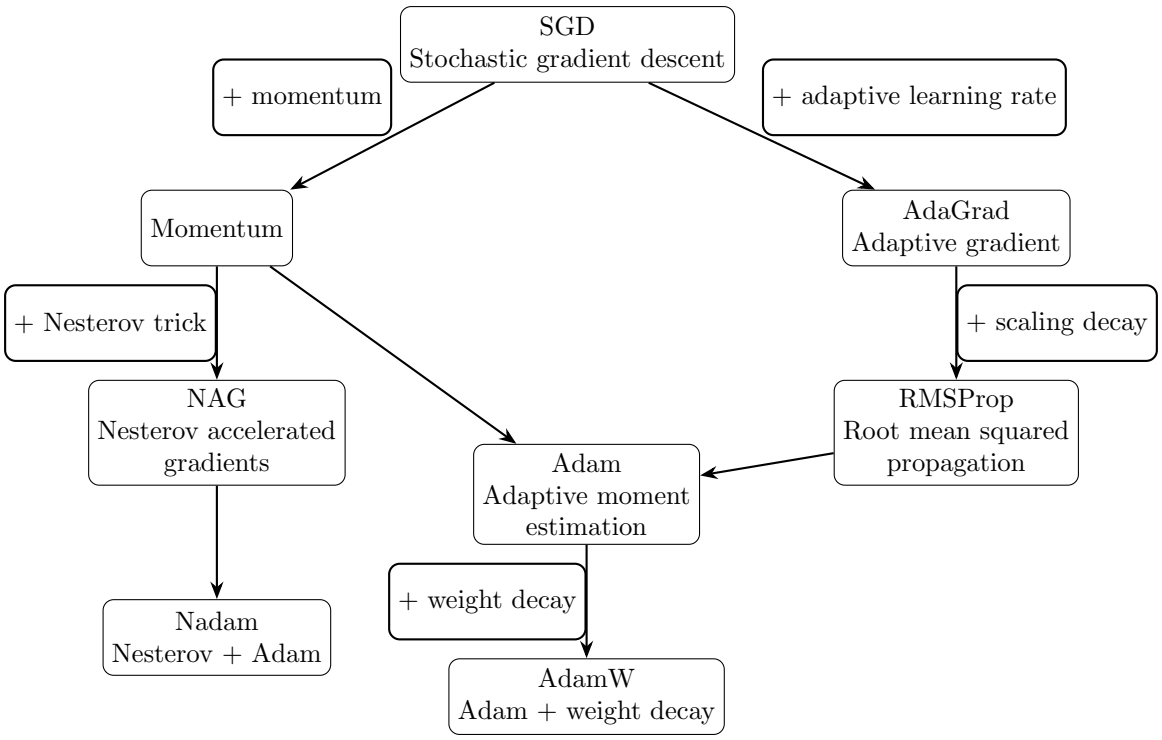
3. $\nabla L(\theta)$: Gradient of the loss function with respect to parameters θ.

By applying weight decay, the model is regularized, mitigating the risk of overfitting to the training data.

1.8.8 Nadam

Definition: Nadam incorporates the Nesterov trick into Adam, further improving its convergence properties in certain scenarios.

1.8.9 Visual Overview of Optimizers



1.9 Learning Rate Scheduling

Definition: Learning rate scheduling is a technique used to adjust the learning rate during training to improve convergence and performance. The learning rate is critical in determining how quickly a model converges and whether it converges effectively at all.

Observations:

1. If the learning rate (η) is **too high**, the loss may diverge, leading to instability.
2. If η is **too low**, the training process will be excessively slow.
3. If η is **slightly too high**, it may oscillate around the optimal point without converging.
4. A **well-tuned learning rate** will result in steady and effective convergence.

1.9.1 Types of Learning Rate Schedules

1. **Power Scheduling:**

$$\eta(t) = \eta_0 \frac{1}{1 + t/s} \tag{23}$$

   - η_0: Initial learning rate.
   - t: Current iteration number.
   - s: A hyperparameter controlling the rate of decay.

   Reduces the learning rate gradually based on the iteration number.

2. **Exponential Scheduling:**

$$\eta(t) = \eta_0 \cdot 0.1^{t/s} \tag{24}$$

   - Reduces the learning rate by a constant factor (e.g., 10) every s steps.

3. **Piecewise Constant Scheduling:**

- The learning rate is manually adjusted at fixed intervals.
- Example:
  (a) $\eta = 0.1$ for the first 5 epochs.
  (b) $\eta = 0.01$ for the next 10 epochs.

4. **Performance-based Scheduling:**
   - The learning rate is adjusted based on validation performance.
   - For example:
     (a) Measure the validation error every N steps.
     (b) Reduce η by a factor λ if the error stagnates.

Conclusion: All the above methods are effective for different scenarios. However, performance-based scheduling is often a good default choice due to its adaptive nature.

1.10 Regularization

Definition: Regularization techniques like L1 and L2 are used to prevent overfitting by penalizing large parameter values. These penalties are added as regularization terms to the loss function, thereby encouraging simpler models with smaller coefficients.

1.10.1 L2 Regularization (Ridge Regression):

$$\text{Loss} = \text{Original Loss} + \alpha \sum_i \theta_i^2 \tag{25}$$

- Penalizes the sum of squared weights (θ_i^2).
- Encourages small, distributed weights across parameters.

1.10.2 L1 Regularization (Lasso Regression):

$$\text{Loss} = \text{Original Loss} + \alpha \sum_i |\theta_i| \tag{26}$$

- Penalizes the sum of absolute weights ($|\theta_i|$).
- Encourages sparsity, setting some parameters to zero.

Note: L2 regularization is commonly used with Adam optimizers, but for more effective weight decay, AdamW is recommended.

1.10.3 Dropout Regularization

Definition: Dropout is a regularization technique where, at each training step, a random subset of neurons is ignored with a certain probability p. This prevents the model from relying too heavily on specific neurons, resulting in a more robust and generalizable model.

Mathematical Representation:

- During training, each neuron is dropped with probability p.
- At test time, all neurons are active, and their outputs are scaled by $1 - p$.

Implementation:

```
tf.keras.layers.Dropout(rate=0.2)
```

1.10.4 Early Stopping

Definition: Early stopping is a regularization technique where training is halted once the validation performance stops improving. This prevents overfitting by ensuring the model does not over-optimize for the training data.

Key Observations:

- Training loss typically decreases continuously.
- Validation loss starts increasing when the model begins to overfit.

1.11 General Suggestions

Hyperparameter	Recommendations
Activation function	Use ReLU for shallow networks. For deeper networks, use Swish or Leaky ReLU as a faster alternative.
Optimizer	Start with Adam or AdamW. If these do not work, switch to NAG. Avoid using SGD or AdaGrad.
Learning rate schedule	Performance scheduling is recommended as it requires minimal hyperparameter tuning. However, other schedules may yield better results if carefully optimized.
Regularization	Use EarlyStopping and weight decay as the primary methods to mitigate overfitting. Explore other techniques if overfitting persists. Avoid using L1/L2 regularization with Adam-type optimizers.

Table 1: General Suggestions for Hyperparameters

2 Convolutional Neural Networks

2.1 What is an Image?

An image can be described as a visual representation composed of a combination of simple structures. These structures, when aggregated, form increasingly complex patterns. This concept is foundational in the design of new architectures for neural networks.

2.2 Convolutional Layers

Convolutional layers are a fundamental component of Convolutional Neural Networks (CNNs). In these layers:

1. Each hidden neuron is connected to only a subset of the input neurons. This subset is referred to as the **local receptive field**.
2. The local receptive field allows the model to capture spatial hierarchies of features by focusing on localized regions of the input.

2.2.1 Convolutional Neural Networks (CNNs)

A **Convolutional Neural Network (CNN)** is a type of artificial neural network specifically designed to process structured data, such as images or time-series data. CNNs utilize convolutional layers to extract spatial features and learn hierarchical patterns in the input data.

2.2.2 Structure of Convolutional Layers

The structure of convolutional layers can be summarized as follows:

1. The **input neurons** are arranged in a grid-like fashion, reflecting the spatial structure of the data (e.g., pixels in an image).
2. Each neuron in the first hidden layer is connected to a local region of the input neurons, emphasizing spatial locality.
3. Connections are sparse compared to fully connected layers, enabling the model to focus on localized patterns.

2.2.3 Operation of Convolutional Layers

In convolutional layers:

1. **Local connections**: Each neuron computes a weighted sum of its local receptive field, followed by the addition of a bias term.
2. **Shared weights**: The same set of weights (kernel) is applied across different regions of the input, enabling the detection of similar features in various locations.
3. **Hierarchical feature extraction**: Higher layers combine simpler features detected in earlier layers, constructing increasingly complex representations.

2.2.4 Filters and Feature Maps

Filters are trained sets of weights that respond to specific patterns in the input data:

1. A filter slides over the input data, computing a weighted sum of the inputs it overlaps with.
2. If the weights match the pattern in the input, the output (neuron activation) is large, effectively highlighting the presence of the pattern.

The output of applying a filter is called a **feature map**, which captures the presence and location of specific patterns in the input.

2.2.5 Feature Extraction with Filters

1. Different filters extract different patterns from the input, such as vertical or horizontal edges.
2. The feature maps generated by these filters emphasize specific patterns while suppressing irrelevant information.

2.2.6 Stacking Feature Maps

In convolutional neural networks, **stacking feature maps** is a key approach to enhance the representational power of the model:

1. Each **feature map** focuses on detecting and enhancing a specific pattern or feature within the input data.
2. By stacking multiple feature maps, the network can represent and process diverse patterns simultaneously.
3. These stacked feature maps serve as the input for subsequent layers, facilitating the hierarchical learning of features at different levels of abstraction.

This methodology ensures that the network captures a comprehensive understanding of the input data by leveraging multiple feature detectors in parallel.

2.3 Other Layers

2.3.1 Pooling Layers

Pooling layers are a crucial component of convolutional neural networks, responsible for summarizing and reducing the spatial dimensions of feature maps:

1. **Purpose**: Pooling layers aim to downsample the feature maps, summarizing the most significant information within small spatial regions.
2. **Operation**: Each region in the feature map is replaced with a single value, typically determined by one of the following methods:
   - **Max pooling**: Selects the maximum value within the region, emphasizing the strongest activation.
   - **Average pooling**: Computes the average value within the region, providing a smoothed representation.
3. **Advantages**:
   (a) Reduces computational complexity by decreasing the spatial dimensions of the feature maps.
   (b) Introduces spatial invariance, making the model robust to small translations in the input.
4. **Built-in regularization**: Pooling inherently simplifies the representation, acting as a regularization mechanism that prevents overfitting.

2.4 A Typical CNN Structure

A Convolutional Neural Network (CNN) typically consists of the following sequence of layers:

1. **Input layer**: Accepts the original image or input data in its raw form.
2. **Convolutional layers**:
   - Detects simple shapes, such as lines or edges, in the initial layers.
   - Later layers detect more complex patterns (e.g., corners) and, eventually, very intricate features (e.g., whiskers in an animal image).
   - Use the **ReLU activation function** to introduce non-linearity, ensuring the model can learn complex mappings.

3. **Pooling layers**:
    - Downsample the feature maps to reduce spatial dimensions and computational requirements.
    - Introduce translational invariance, making the model robust to small movements in the input.
4. **Fully connected layers**:
    - Flatten the feature maps and connect all neurons, combining the learned features to make predictions.
    - Serve as the output layer, mapping the input to specific categories (e.g., "cat" or "dog").

2.5 Object Localization

Object localization involves training a neural network to simultaneously classify and locate objects within an image. This is achieved by combining classification and regression tasks:

1. **Classification task**: Determines the class or category of the object within the image (e.g., car, traffic light).
2. **Regression task**: Predicts the bounding box coordinates (w_1, w_2, h_1, h_2) that specify the location of the object in the image.

2.5.1 Key Concepts

1. **Multi-task learning**: The model is designed to predict both the class of an object and the spatial coordinates of its bounding box simultaneously.
2. **Loss functions**: The network uses separate loss functions for classification and localization tasks:
    - **Categorical cross-entropy** for classification.
    - **Mean squared error (MSE)** for bounding box regression.
3. **Output representation**: The final layer of the model generates two outputs:
    - A softmax probability distribution for class predictions.
    - A set of four numbers representing the bounding box coordinates.
4. **Loss weighting**: The losses for classification and regression tasks are weighted to ensure balanced training, prioritizing both tasks appropriately.

By incorporating both classification and localization, object localization models are capable of accurately identifying objects and their spatial locations within an image.

3 Autoencoders

Autoencoders are a type of neural network architecture designed to learn an efficient representation of data, often referred to as "codings" or "latent space representations." They are composed of two main components:

1. **Encoder:** The encoder maps the input data to a lower-dimensional representation.
2. **Decoder:** The decoder reconstructs the original input from the lower-dimensional representation.

3.1 Loss Function in Autoencoders

The objective of an autoencoder is to minimize the reconstruction error, which quantifies the difference between the input and the reconstructed output. This can be expressed as:

$$\text{Loss} = \|X - \hat{X}\|^2, \tag{27}$$

where X is the input, and $\hat{X}$ is the reconstructed output.

3.2 Undercomplete Autoencoder

An undercomplete autoencoder is characterized by a bottleneck in its architecture, where the latent representation has fewer dimensions than the input data. This forces the model to capture the most salient features of the input.

To reconstruct X, the bottleneck layer must encode as much information as possible about the original input. The encoder and decoder work together to achieve this compression and reconstruction.

3.3 Deep Autoencoders

Deep autoencoders are an extension of basic autoencoders that utilize multiple hidden layers in both the encoder and decoder. They are capable of learning complex, non-linear mappings between input and latent space.

These models are often symmetric, with the encoder and decoder having mirror-like architectures. The codings learned by deep autoencoders can represent intricate patterns and relationships in the data.

However, the transformation from codings to reconstructed data may be challenging to interpret due to its complexity.

3.4 Denoising with Autoencoders

Denoising autoencoders are an extension of basic autoencoders that aim to reconstruct the original input data from a corrupted version of it. This process enforces the model to learn robust features that are resilient to noise or distortions in the input.

The primary objective is to minimize the reconstruction error while handling input with added noise. The loss function can be expressed as:

$$\text{Loss} = \|X - \hat{X}\|^2, \tag{28}$$

where X is the clean input, and $\hat{X}$ is the reconstructed output from the noisy input.

3.5 Applications of Autoencoders

Autoencoders have diverse applications in various fields due to their ability to learn efficient representations of data. Some key applications include:

1. **Image Denoising:** Autoencoders can be trained to remove noise or artifacts from images, enhancing their quality.
2. **Super-Resolution:** They can transform low-resolution data into high-resolution versions by learning mappings between different quality levels of input data.

3. **Anomaly Detection:** Autoencoders are effective in identifying anomalies by measuring reconstruction errors that deviate significantly from normal patterns.

4. **Data Compression:** By encoding data into a lower-dimensional latent space, autoencoders serve as powerful tools for compressing information while retaining essential features.

5. **Image colorization** is a challenging task in computer vision that involves converting grayscale images into their corresponding colored versions. Autoencoders can be employed for this purpose by learning a mapping between grayscale input images and their colorized counterparts.

4 Generative Adversarial Networks

4.1 Overview

Generative Adversarial Networks (GANs) consist of two neural networks that are trained in opposition to one another. These networks are known as the **generator** and the **discriminator**. The generator creates data samples, while the discriminator evaluates whether a given sample is real or generated.

4.2 Training Goals

1. **Discriminator (D):** The objective of the discriminator is to become proficient at distinguishing between real and generated (fake) data samples.
2. **Generator (G):** The objective of the generator is to generate data samples that can successfully deceive the discriminator.

4.2.1 Phase I: Training the Discriminator

In this phase, the discriminator (D) is trained to differentiate between real and fake data samples. The generator (G) is frozen during this step.

1. Generate a batch of fake samples using the generator (G) from noise input.
2. Combine the fake samples with real data, maintaining a 50:50 ratio.
3. Label the fake samples as "fake" and real samples as "real."
4. Train the discriminator to classify the combined batch as accurately as possible.

4.2.2 Phase II: Training the Generator

In this phase, the generator (G) is optimized to improve its ability to generate samples that deceive the discriminator (D). The discriminator is frozen during this step.

1. Generate fake samples using the generator (G).
2. Label the fake samples as "real" (misleading labels).
3. Train the generator to optimize its performance based on the discriminator's feedback, improving its ability to fool the discriminator.

Note: By alternating between these two phases, the generator and discriminator improve their respective tasks iteratively.

4.2.3 Explanation for misleading labels (Source)

When training GANs, the training steps for the generator and discriminator are separate:

1. There is a training stage for the discriminator, where it is presented with a mix of generated and real data, all correctly labelled. It is important at this stage to not update the generator (otherwise it will get worse by helping to make the fake data look more fake).
2. There is a training stage for the generator, where the discriminator is presented with only generated data, all labelled incorrectly as if it were real. It is important at this stage to not update the discriminator.

Typically you will alternate between the two stages frequently, making small updates to discriminator and generator separately. Some GANs use metrics to decide how much of each to do, because you don't want either the generator or discriminator to win outright and stop progress - at least at the start.

So yes there is a stage where you deliberately "fool" the discriminator, because being able to do so is the goal of the generator. However, one key detail of this stage is that the discriminator weights are not updated from that faked data. Instead the gradients from that stage are used only to update and improve the generator.

It may help if you don't think of the false labels as being "fool the discriminator", but instead they are "measure how well the generator is fooling the discriminator".

4.3 Solutions to Mode Collapse

Mode collapse occurs when the generator produces limited and repetitive outputs, failing to capture the diversity of the target data distribution. The following methods can help address this issue:

4.3.1 Experience Replay

- Store generated samples in a replay buffer.
- Use these stored samples during training to diversify the data seen by the discriminator.

4.3.2 Mini-Batch Discrimination

- Measure the similarity among samples in a generated batch.
- Provide this information to the discriminator to ensure it can reject an entire batch of overly similar samples.

4.4 Other Convergence Challenges

4.4.1 Imbalance in Training

If the discriminator (D) becomes too strong, it may easily classify fake data, making it difficult for the generator (G) to learn. This can lead to:

- D achieving perfect accuracy early in training.
- G failing to improve because it receives minimal feedback from D.

Solution: Introduce noise to the discriminator's input labels to prevent it from becoming overly confident:

$$\text{labels} \mathrel{+}= 0.05 \cdot \text{tf.random.uniform(tf.shape(labels))} \tag{29}$$

4.4.2 Fleeting Convergence

GAN training can be unstable, with alternating periods where:

- The generator improves significantly while the discriminator becomes weaker, or vice versa.
- Convergence appears temporary and oscillatory.

4.4.3 General Challenges in GAN Training

In general, training GANs is unstable and often requires significant effort to fine-tune hyperparameters, as small changes can lead to divergence or failure to converge.

Note: Careful hyperparameter tuning and monitoring during training are critical for achieving stable convergence.

4.5 Deep Convolutional Generative Adversarial Networks (DCGANs)

4.5.1 Overview

Deep Convolutional GANs (DCGANs) utilize convolutional neural networks (CNNs) for both the generator and discriminator. This is particularly effective when working with image data. However, training DCGANs can often be unstable, requiring specific architectural considerations.

4.5.2 Guidelines for DCGANs

- Replace pooling layers with strided convolutions in the discriminator (D) and transposed convolutions in the generator (G).
- Use batch normalization in both G and D, except in the output layer of G and the input layer of D.
- Avoid fully connected hidden layers for deeper architectures.
- Use ReLU activation in G for all layers except the output layer, which should use the `tanh` activation.
- Use leaky ReLU activation in all layers of D.

4.6 Latent Vector Arithmetic in GANs

4.6.1 Concept

Latent vectors, sampled from a random noise distribution, serve as input to the generator in GANs. Despite their randomness, these vectors are trained to represent meaningful features in the data.

4.6.2 Latent Space Arithmetic

- Latent vectors allow for arithmetic operations that correspond to meaningful transformations in the generated data.
- For instance, subtracting the vector representation of a man without glasses from that of a man with glasses, and adding the representation of a woman without glasses, can produce a woman with glasses.

Note: These operations demonstrate the ability of GANs to capture structured and interpretable features in the latent space, mapping them to complex transformations in pixel space.

4.7 Conditional Generative Adversarial Networks (cGANs)

4.7.1 Overview

Conditional GANs (cGANs) are an extension of GANs where additional information is provided to both the generator and discriminator in the form of a condition, such as a label or class. This allows for controlled generation of data samples based on the specified condition.

4.7.2 Architecture

- The generator (G) takes both a noise vector and a label as input. The noise vector represents randomness, while the label specifies the desired class of the generated sample.
- The discriminator (D) receives both the generated or real image and the corresponding label as input. It evaluates whether the input image is real or fake, while also considering the provided label.

4.7.3 Functionality

1. Labels are encoded into a one-hot vector and concatenated with the noise vector before being passed into the generator.
2. The generator creates a fake image based on the input noise and label.
3. The discriminator evaluates pairs of images and labels, identifying whether the image is real or fake while ensuring the label matches the image.

Example: In the case of digit generation:

- Input label "5" prompts the generator to create an image resembling the digit 5.

- The discriminator verifies whether the generated or provided image matches the label "5."
- Mismatches (e.g., generating a "6" when the label is "5") are flagged as fake.

4.8 Text-to-Image Generation

4.8.1 Overview

Text-to-image generation involves creating realistic images based on textual descriptions. This is achieved by conditioning the generative adversarial network (GAN) on text embeddings derived from a language model.

4.8.2 Architecture

- **Language Model:** A language model converts textual prompts into descriptive embeddings. These embeddings capture the semantic meaning of the input text.
- **Generator (G):** Takes noise and the text embedding as input to generate an image that corresponds to the given description.
- **Discriminator (D):** Evaluates whether the input image is real or fake, considering the associated text embedding for consistency.

4.8.3 Functionality

1. A text prompt is passed into the language model, which outputs an embedding representing the textual description.
2. The embedding is combined with random noise and passed into the generator to produce a synthetic image.
3. The discriminator receives both real and generated images, along with their respective text embeddings, and determines whether each image-text pair is consistent and real.

Potential Applications:

- Generating artwork or illustrations based on textual descriptions.
- Synthesizing visual content for storyboarding or conceptual design.
- Assisting creative projects by automating visual representation of ideas.

Note: Text-to-image generation poses significant challenges due to the complexity of mapping high-dimensional textual representations to realistic image outputs.

5 Recurrent Neural Networks

5.1 Recurrent Neurons

Recurrent neurons are a type of neural network unit designed to process sequential data by maintaining a memory of previous inputs. This is achieved by using the output of the neuron at a previous time step as an input for the current time step.

5.1.1 Core Concepts

1. **Hidden State (h):** Represents the memory or context from prior time steps, enabling the network to retain information over time.
2. **Time-Dependent Input ($x(t)$):** The input to the neuron at each time step t.
3. **Output ($y(t)$):** The output of the neuron at each time step t, which is computed using the hidden state and the current input.

5.1.2 Mathematical Representation

The computation in a recurrent neuron can be expressed as:

$$h_t = f(W_h h_{t-1} + W_x x_t + b_h) \tag{30}$$
$$y_t = g(V h_t + b_y) \tag{31}$$

where:

1. h_t: Hidden state at time t.
2. h_{t-1}: Hidden state from the previous time step.
3. x_t: Input at time t.
4. W_h, W_x, V: Weight matrices for the hidden state, input, and output, respectively.
5. b_h, b_y: Bias terms for the hidden state and output, respectively.
6. f: Activation function for the hidden state (e.g., tanh or ReLU).
7. g: Activation function for the output.

5.2 Recurrent Layers

A recurrent layer is an extension of recurrent neurons applied to a sequence of inputs over time. It enables the model to process temporal data by maintaining a hidden state that evolves across time steps.

5.2.1 Characteristics of Recurrent Layers

1. The layer processes a sequence of inputs $\{x(0), x(1), \ldots, x(T)\}$.
2. The hidden state $h(t)$ propagates through time, capturing temporal dependencies.
3. The outputs $\{y(0), y(1), \ldots, y(T)\}$ are computed at each time step, dependent on the hidden states.

5.3 Backpropagation Through Time

Backpropagation Through Time (BPTT) is the gradient-based optimization algorithm used to train recurrent neural networks. It extends standard backpropagation to account for temporal dependencies by unrolling the network across time.

5.3.1 Key Steps in BPTT

1. The network is unrolled over all time steps, forming a sequence of layers.
2. Gradients are computed for each time step and accumulated across the sequence.
3. The weight updates are applied simultaneously for all time steps, using the same shared parameters.

5.3.2 Challenges of BPTT

Recurrent neural networks face the following issues during training:

1. **Vanishing Gradients:** Gradients may shrink exponentially over long sequences, hindering the ability to learn long-term dependencies.
2. **Exploding Gradients:** Gradients may grow exponentially, causing instability in optimization.

To mitigate these problems, activation functions such as tanh or ReLU are typically employed.

5.4 The Versatility of Recurrent Neural Networks

Recurrent Neural Networks (RNNs) exhibit remarkable versatility and adaptability to various tasks involving sequential data. They can be configured into different architectures depending on the input and output requirements.

5.4.1 Sequence-to-Sequence Networks

Sequence-to-sequence networks process input sequences and generate corresponding output sequences.

- **Application:** Time-series forecasting with shifted input-output pairs.
- **Example:** Predicting future values from past observations.

5.4.2 Sequence-to-Vector Networks

Sequence-to-vector networks take an input sequence and produce a single output vector, summarizing the entire sequence.

- **Application:** Sentiment analysis or classification tasks.
- **Example:** Determining whether a review is positive or negative.

5.4.3 Vector-to-Sequence Networks

Vector-to-sequence networks start with a fixed input vector and generate an output sequence.

- **Application:** Image captioning, where an image vector is translated into a descriptive text sequence.
- **Example:** Generating captions such as "A man holding a kite" from an image input.

5.4.4 Encoder-Decoder Networks

Encoder-decoder networks consist of two components:

1. **Encoder:** Encodes the input sequence into a fixed-length vector representation.
2. **Decoder:** Decodes the vector representation to produce an output sequence.

- **Application:** Machine translation, where sentences in one language are translated into another.
- **Example:** Translating "Bonjour" to "Hello".

5.5 Short-Term Memory

Recurrent neural networks (RNNs) inherently struggle with maintaining long-term dependencies due to the gradual loss of information over time. This phenomenon occurs because:

- At every time step, only a portion of the hidden state is carried forward.
- As a result, information from earlier time steps is progressively diluted or lost.

This limitation makes standard RNNs more suited for tasks requiring short-term memory, as virtually all information beyond a certain number of time steps becomes inaccessible.

5.6 Long Short-Term Memory (LSTM) Cells

LSTM cells are a specialized type of RNN architecture designed to address the limitations of standard RNNs in learning long-term dependencies. They introduce mechanisms for explicitly controlling what information to remember or forget over time.

5.6.1 Key Components of an LSTM Cell

1. **Forget Gate (f_t):** Decides what information to discard from the cell state.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{32}$$

2. **Input Gate (i_t):** Decides what new information to store in the cell state.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{33}$$

3. **Cell State Update ($\tilde{C}_t$):** Represents new candidate values to add to the cell state.

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{34}$$

4. **Output Gate (o_t):** Determines the output from the LSTM cell.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{35}$$

5.6.2 Overall Cell State and Output Update

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{36}$$

$$h_t = o_t \cdot \tanh(C_t) \tag{37}$$

Here:

- C_t: Cell state at time t.
- h_t: Hidden state or short-term memory at time t.
- σ: Sigmoid activation function.
- tanh: Hyperbolic tangent activation function.

5.7 Gated Recurrent Unit (GRU) Cells

GRU cells are a simplified variant of LSTM cells, combining certain gates to reduce computational complexity while maintaining performance.

5.7.1 Key Components of a GRU Cell

1. **Update Gate (z_t):** Controls the amount of information to retain from the past.

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{38}$$

2. **Reset Gate (r_t):** Determines how much of the previous state to forget.

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{39}$$

5.7.2 State Updates

$$\tilde{h}_t = \tanh(W_h[r_t \cdot h_{t-1}, x_t] + b_h) \tag{40}$$

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t \tag{41}$$

Here:

- h_t: Current hidden state.
- $\tilde{h}_t$: Candidate hidden state.
- r_t: Reset gate activation.
- z_t: Update gate activation.

6 Small Language Models

6.1 Character Recurrent Neural Networks (Char-RNNs)

A Char-RNN is a type of recurrent neural network that functions as a **standard classifier**, where each character in a sequence is treated as a class. Given an input sequence of characters, the Char-RNN processes the sequence one character at a time and predicts the subsequent character based on learned patterns.

6.1.1 Hidden States and Predictions

The hidden state at each time step h_t is updated based on the input character x_t and the previous hidden state h_{t-1}. The prediction at each time step $\hat{y}_t$ represents the most likely next character.

$$h_t = f(h_{t-1}, x_t) \tag{42}$$
$$\hat{y}_t = g(h_t) \tag{43}$$

Here, f represents the update function for the hidden state, and g represents the function that generates the output predictions.

6.2 Representation of Letters

Letters can be represented in various forms for input into machine learning models. Two commonly used methods are described below.

6.2.1 One-Hot Encoding

One-hot encoding is a technique where each character is represented as a binary vector. The vector has a dimension equal to the size of the alphabet, with all elements set to zero except for the index corresponding to the character, which is set to one.

1. "a" is represented as $[1, 0, 0, 0, \ldots]$
2. "b" is represented as $[0, 1, 0, 0, \ldots]$
3. "c" is represented as $[0, 0, 1, 0, \ldots]$
4. "d" is represented as $[0, 0, 0, 1, \ldots]$

Although simple, one-hot encoding is not efficient for capturing relationships between characters.

6.2.2 Embeddings

Embeddings are trainable vector representations for characters. An embedding layer learns to map each character into a dense, continuous vector space, where semantically related characters are placed closer to one another. This approach often produces more meaningful representations.

$$\text{Embedding}(x) \to \mathbb{R}^d \tag{44}$$

The embedding space can also be interpreted and manipulated through arithmetic operations, revealing relationships such as:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

$$\text{Copenhagen} - \text{Denmark} + \text{Germany} \approx \text{Berlin}$$

6.3 Char-RNN: Writing Like Tolkien

Char-RNNs are structured with multiple layers, each serving a specific purpose in processing and generating sequences. The architecture can be described as follows:

6.3.1 Embedding Layer

The embedding layer converts each input character into a dense vector representation. This transformation enables the model to work with continuous-valued vectors rather than discrete symbols, capturing relationships between characters in the process.

6.3.2 RNN Layer

The recurrent neural network (RNN) layer, which may use GRU (Gated Recurrent Unit) or LSTM (Long Short-Term Memory) units, processes the sequence of embeddings. It is designed to model temporal dependencies and sequential patterns, making it well-suited for tasks involving time series or sequential data.

6.3.3 Fully Connected Output Layer

The fully connected output layer uses a softmax activation function to produce a probability distribution over possible output classes. The number of neurons in this layer corresponds to the total number of characters in the dataset. The model predicts the next character in the sequence by selecting the class with the highest probability.

$$P(y_t|x_{1:t}) = \text{softmax}(Wh_t + b) \tag{45}$$

Here, h_t represents the hidden state at time t, W and b are learnable parameters, and $P(y_t|x_{1:t})$ gives the probability of the next character given the input sequence up to t.

6.4 Two Considerations for Embeddings and Text Generation

6.4.1 Do the Embeddings Represent Meaningful Information?

To evaluate whether the embeddings represent meaningful relationships, dimensionality reduction techniques such as Principal Component Analysis (PCA) can be applied. For example, PCA can reduce a multi-dimensional embedding space to a 2-dimensional space for visualization, where distinct groupings of characters (e.g., letters, sentence starters, and sentence stoppers) may emerge.

6.4.2 Challenges in Generating Long Text Strings

When attempting to generate long text sequences, repetitive patterns can occur if the model consistently selects the most probable character at each step. This limitation arises because deterministic approaches may overfit to the highest likelihood.

6.4.3 Sampling from the Probability Distribution

Instead of always choosing the character with the highest probability, the model can sample characters according to their probability distribution. For example, given a probability distribution:

$$\begin{bmatrix} P(a) & P(b) & P(c) & P(d) & P(e) & P(f) & \dots \\ 0.31 & 0.01 & 0.03 & 0.02 & 0.12 & 0.01 & \dots \end{bmatrix}$$

The character "a" would be chosen 31% of the time. This approach introduces controlled randomness and diversity into the generated text, mitigating repetitive outputs.

6.5 Softmax and Temperature Adjustment

6.5.1 Softmax Function

The softmax function is used to convert raw scores z_i into probabilities p_i over possible classes:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{46}$$

6.5.2 Temperature-Adjusted Softmax

Temperature adjustment modifies the softmax function to control the randomness of predictions. The temperature-adjusted softmax is defined as:

$$p_i' = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} \tag{47}$$

Here, T represents the temperature:

- When T is low, z_i/T becomes large, leading to exaggerated differences and more deterministic predictions.
- When T is high, z_i/T becomes small, and differences between probabilities are minimized, making predictions more random.

6.5.3 Temperature Impact on Probabilities

1. **Low Temperature (T small)**: Probability distribution becomes peaked, with one class dominating:
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & \dots \end{bmatrix}$$
2. **High Temperature (T large)**: Probability distribution becomes uniform:
$$\begin{bmatrix} 0.04 & 0.04 & 0.04 & \dots \end{bmatrix}$$

6.6 Detailed Calculation of Softmax with Temperature

The logarithmic form of the softmax is derived as follows:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{48}$$

$$\ln p_i = z_i - \ln\left(\sum_j e^{z_j}\right) \tag{49}$$

Dividing by the temperature T, we have:

$$\frac{\ln p_i}{T} = \frac{z_i}{T} - \frac{\ln\left(\sum_j e^{z_j}\right)}{T} \tag{50}$$

The temperature-adjusted softmax can be expressed as:

$$p_i' = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} \tag{51}$$

This adjustment enables fine-grained control over the randomness of the output probabilities, balancing determinism and diversity in model predictions.

6.7 Use Cases of Recurrent Neural Networks (RNNs)

6.7.1 Text Classification

RNNs can be employed for text classification tasks, such as detecting spam messages. The architecture processes the input text sequentially, leveraging an embedding layer for vector representation, an RNN layer for capturing sequential dependencies, and a fully connected output layer for class prediction (e.g., "spam").

6.7.2 Music Generation

RNNs are capable of generating music by predicting the next note in a sequence. The model learns patterns from existing compositions and outputs notes sequentially, potentially considering aspects such as note length and emphasis to produce coherent melodies.

7 Large Language Models

7.1 Encoder-Decoder Networks for Translation

The encoder-decoder network is a framework commonly used in machine translation tasks. The primary objective is to transform an input sequence into a meaningful representation (hidden state), which is then used to produce the corresponding output sequence.

7.1.1 Mathematical Formulation

Let $x^{(0)}, x^{(1)}, \ldots, x^{(n)}$ represent the input sequence, and $\hat{y}^{(0)}, \hat{y}^{(1)}, \ldots, \hat{y}^{(m)}$ represent the predicted output sequence. The encoder maps the input sequence into a fixed-length hidden representation h, which the decoder uses to generate the output sequence:

$$h = f_{\text{encoder}}(x^{(0)}, x^{(1)}, \ldots, x^{(n)}) \tag{52}$$

$$\hat{y}^{(t)} = f_{\text{decoder}}(h, \hat{y}^{(0)}, \hat{y}^{(1)}, \ldots, \hat{y}^{(t-1)}) \tag{53}$$

7.1.2 Key Concept

The hidden state h must encapsulate the essential information of the input sequence, enabling accurate reconstruction or translation.

7.2 Attention Mechanism

Attention mechanisms allow the network to focus on specific parts of the input sequence during decoding, addressing the limitations of fixed-length representations.

7.2.1 Mathematical Formulation of Attention

Let h_i represent the hidden state of the encoder for the i-th input token. Attention weights α_{ij} are computed as:

$$\alpha_{ij} = \text{softmax}(e_{ij}) \tag{54}$$

$$e_{ij} = f_{\text{score}}(h_i, s_{j-1}) \tag{55}$$

Here, s_{j-1} is the decoder's hidden state at time step $j-1$. The context vector c_j is then computed as:

$$c_j = \sum_i \alpha_{ij} h_i \tag{56}$$

7.3 Types of Attention

7.3.1 Self-Attention in the Encoder

Self-attention in the encoder computes relationships between input tokens by comparing all tokens against each other. This mechanism allows the model to capture dependencies within the input sequence.

The computation involves:

1. Generating three vectors: Query (Q), Key (K), and Value (V) for each input token using learned weight matrices:

$$Q = XW_Q, \tag{57}$$

$$K = XW_K, \tag{58}$$

$$V = XW_V, \tag{59}$$

where $X \in \mathbb{R}^{n \times d}$ is the input matrix, $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ are learned weight matrices, n is the sequence length, d is the input dimension, and d_k is the dimensionality of the query/key vectors.

2. Computing the attention scores using the scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{60}$$

3. Normalizing the attention scores with the softmax function and applying them to the value matrix V.

7.3.2 Masked Self-Attention in the Decoder

Masked self-attention in the decoder focuses on relationships between output tokens, ensuring that only tokens up to the current position are considered. This is achieved by applying a mask to the attention mechanism.

The process is similar to self-attention in the encoder, but with an additional step:

1. A mask is applied to the attention scores to prevent the model from attending to future tokens:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V, \tag{61}$$

where M is a masking matrix with $-\infty$ in positions corresponding to future tokens and 0 elsewhere.

This ensures that the model processes the sequence in an autoregressive manner.

7.3.3 Cross-Attention in the Decoder

Cross-attention in the decoder connects the output tokens to the input tokens, enabling the decoder to focus on relevant parts of the input sequence while generating the current output token.

The computations involve:

1. Queries (Q) are generated from the decoder's hidden states, while Keys (K) and Values (V) are derived from the encoder's output:

$$Q_{\text{decoder}} = H_d W_Q, \tag{62}$$
$$K_{\text{encoder}} = H_e W_K, \tag{63}$$
$$V_{\text{encoder}} = H_e W_V, \tag{64}$$

where H_d is the decoder hidden state, and H_e is the encoder output.

2. Attention scores are computed as:

$$\text{Attention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}) = \text{softmax}\left(\frac{Q_{\text{decoder}} K_{\text{encoder}}^\top}{\sqrt{d_k}}\right)V_{\text{encoder}}. \tag{65}$$

This mechanism allows the decoder to selectively attend to relevant parts of the encoded input sequence for generating accurate output tokens.

7.4 Transformers

Transformers remove the sequential dependency of RNNs by relying entirely on attention mechanisms, significantly improving parallelism during training.

7.4.1 Overview of the Encoder-Decoder Structure

The transformer architecture is composed of two main components:

1. **Encoder:** Processes the input sequence to produce a context-aware representation.
2. **Decoder:** Generates the output sequence while attending to the encoder's output.

Each component consists of a stack of identical layers:

1. **Encoder Layers:**

(a) **Self-Attention:** Captures dependencies within the input sequence.

(b) **Feed-Forward Networks:** Processes the output of the attention mechanism.

2. **Decoder Layers:**

(a) **Masked Self-Attention:** Ensures that only past tokens are considered.

(b) **Cross-Attention:** Aligns the decoder's output with the encoder's representation.

(c) **Feed-Forward Networks:** Processes the output of the attention mechanisms.

The architecture applies residual connections and layer normalization around each sub-layer.

7.4.2 Encoder-Decoder Formulation

The encoder-decoder mechanism can be formally described as:

1. The encoder maps the input sequence $X = \{x_1, x_2, \ldots, x_n\}$ to a set of context vectors $H = \{h_1, h_2, \ldots, h_n\}$.

2. The decoder generates the output sequence $Y = \{y_1, y_2, \ldots, y_m\}$, using:

$$h = \text{Encoder}(X), \tag{66}$$

$$y_t = \text{Decoder}(Y_{<t}, h), \tag{67}$$

where $Y_{<t}$ is the sequence of tokens generated before time step t.

7.5 Generative Pretrained Transformers (GPTs)

7.5.1 Key Characteristics of GPT

Generative Pretrained Transformers (GPTs) are a family of models built upon the transformer decoder stack. Key attributes include:

1. A stack of 12 transformer modules.

2. **Masked Self-Attention:** Ensures autoregressive behavior by only attending to previous tokens.

3. **Pretraining Objective:** Predicts the next token in a sequence based on context, using:

$$\mathcal{L} = -\sum_{t=1}^{m} \log P(y_t | Y_{<t}), \tag{68}$$

where $Y_{<t}$ represents tokens up to time step t.

4. **Fine-Tuning:** Tailored for tasks like entailment, similarity detection, and question answering.

7.6 Vision Transformers

7.6.1 Overview

Transformers, initially developed for natural language processing, have been adapted to computer vision tasks. Vision transformers process images by splitting them into patches and treating these patches as analogous to words in a sentence.

7.6.2 Key Steps in Vision Transformers

1. **Image Splitting:** The input image is divided into fixed-size patches (e.g., 16×16), each treated as an independent unit.

2. **Patch Embedding:** Each patch is flattened into a vector and projected into a higher-dimensional embedding space.

3. **Positional Encoding:** Since transformers are permutation invariant, positional encodings are added to the patch embeddings to retain spatial information.

4. **Transformer Encoder:** The sequence of patch embeddings is processed by a transformer encoder, using self-attention to capture global dependencies between patches.

5. **Output Representation:** The final output can be used for tasks such as classification, object detection, or caption generation.

7.6.3 Mathematical Representation

Let an image I of size $H \times W \times C$ be divided into N patches, where each patch has size $P \times P \times C$. The patch embedding process is represented as:

$$X_p = \text{Flatten}(I_p), \quad p \in \{1, 2, \ldots, N\}, \tag{69}$$

$$E_p = X_p W_E + P_E, \tag{70}$$

where W_E is the embedding projection matrix, and P_E is the positional encoding.

7.6.4 Analogy with NLP

Each image patch is treated as equivalent to a word in a sentence. The embeddings for these patches are processed through a transformer, similar to how word embeddings are handled in language models.

7.7 Attention and Explainability

7.7.1 Role of Attention Mechanisms

Attention mechanisms in transformers allow the model to focus on specific parts of the input image, enabling explainability. For example, when generating captions, the model highlights regions of the image most relevant to the current output token.

7.7.2 Ethics and Transparency

The ethical deployment of AI systems requires transparency and explainability. The level of transparency should align with the application context, balancing against other principles such as privacy, safety, and security.

8 Reinforcement Learning

Reinforcement Learning is a framework in which an **agent** interacts with an **environment** by performing **actions**, observing **states**, and receiving **rewards**. The goal of the agent is to learn an optimal strategy, based on a **policy**, to maximize cumulative rewards over time.

8.1 Agent-Environment Interaction

The interaction between the agent and the environment occurs in discrete time steps:

1. At time t, the agent observes the current state s_t of the environment.
2. Based on its policy π, the agent selects an action a_t.
3. The environment transitions to a new state s_{t+1} and provides a reward r_t as feedback to the agent.

The process can be mathematically modeled as:

$$s_{t+1} \sim P(s_{t+1}|s_t, a_t), \tag{71}$$
$$r_t = R(s_t, a_t), \tag{72}$$

where P represents the state transition probability, and R is the reward function.

8.2 Q-Learning

A policy π is a mapping that determines the action a to be taken by an agent when in a given state s. The objective of the policy is to guide the agent towards maximizing its cumulative reward.

In Q-Learning, the policy is derived based on the *quality* of a state-action pair (s, a). The quality, represented as $Q(s, a)$, measures the expected cumulative reward starting from state s, taking action a, and following the optimal policy thereafter.

The quality function is defined as:

$$Q(s, a) = \sum_{t=0}^{\infty} \gamma^t r_{t+1}, \tag{73}$$

where:

- γ is the discount factor, $0 \leq \gamma < 1$, which prioritizes immediate rewards over distant ones.
- r_{t+1} is the reward received at time step $t + 1$.

The optimal action a for a given state s is chosen to maximize the Q value:

$$\pi(s) = \arg\max_a Q(s, a). \tag{74}$$

8.2.1 Q-Learning Update Rule

The quality function $Q(s, a)$ can be updated iteratively to find the optimal policy. The equation for $Q(s, a)$ is derived as follows:

$$Q(s, a) = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{75}$$
$$= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots \tag{76}$$
$$= r + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+2} \tag{77}$$
$$= r + \gamma \max_{a'} Q(s', a'), \tag{78}$$

where s' is the next state and a' is the action taken in that state. This is referred to as the **iterative updating algorithm**.

8.2.2 Incorporating a Learning Rate

In practice, directly using the above update rule can lead to divergence. To mitigate this, a **learning rate** α is introduced to control the update step size:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') \right], \tag{79}$$

where:

- α is the learning rate, with $0 < \alpha \leq 1$.
- γ is the discount factor prioritizing immediate rewards.

This ensures gradual convergence to the optimal Q values over time.

8.2.3 Deep Q-Learning

Deep Q-Learning extends the concept of Q-Learning by leveraging a neural network to approximate the quality function $Q(s,a)$. Instead of explicitly calculating the Q-value for every possible state-action pair, the neural network maps a given state to corresponding Q-values for all actions.

Neural Network Structure

- The **input layer** represents the current state, often encoded as a vector (e.g., the positions of entities in a game).
- The **hidden layers** process this information to learn complex representations of the state.
- The **output layer** produces Q-values corresponding to each possible action.

The best action for a given state is selected using the *argmax* function:

$$\text{Best action: } a^* = \arg\max_a Q(s,a). \tag{80}$$

Training the Neural Network The neural network is trained to minimize the error between the predicted Q-values and the target Q-values obtained from the Bellman equation:

$$\text{Loss: } \mathcal{L} = \left(Q(s,a) - \left[r + \gamma \max_{a'} Q(s',a') \right] \right)^2, \tag{81}$$

where:

- $Q(s,a)$ is the predicted Q-value for a state-action pair.
- $r + \gamma \max_{a'} Q(s',a')$ is the target Q-value obtained from the environment.

This approach allows the agent to generalize across similar states, making it scalable to high-dimensional state spaces.

8.3 Implementation of Deep Q-Learning

8.3.1 Basic Policy

A **basic policy** is a straightforward rule-based approach used by an agent to determine its actions based on observations. For example, the agent may choose to:

- Move left when the system leans left.
- Move right when the system leans right.

The goal is to maximize rewards over multiple episodes by maintaining a balance or upright state.

8.3.2 Epsilon-Greedy Policy

The **Epsilon-Greedy Policy** is a method for balancing exploration and exploitation:

- With probability ϵ, the agent explores by selecting a random action.
- Otherwise, the agent exploits its current knowledge by choosing the action with the highest Q-value:

$$a^* = \arg\max_a Q(s, a). \tag{82}$$

8.3.3 Deep Q-Learning Training

Replay Buffer: A replay buffer stores experiences (s (state), a (action), r (reward), s' (next state)) for training the model. This helps to break correlations in the data and allows for more efficient use of experiences.

Target Q-Value: The target Q-value for training is calculated using:

$$Q_{\text{target}} = r + \gamma \max_{a'} Q(s', a'), \tag{83}$$

where:

- r is the immediate reward.
- γ is the discount factor.
- $Q(s', a')$ represents the estimated Q-value of the next state-action pair.

Loss Function: The loss function measures the error between the predicted Q-value and the target Q-value:

$$\mathcal{L} = \left(Q(s, a) - Q_{\text{target}}\right)^2. \tag{84}$$

Model Optimization: The model parameters are updated using gradient descent to minimize the loss function:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}, \tag{85}$$

where α is the learning rate and θ represents the model parameters.

8.3.4 Neural Network Architecture

The neural network used for approximating Q-values consists of:

- An **input layer** representing the state vector (e.g., four-dimensional input for a cart-pole system).
- **Hidden layers** with non-linear activation functions, such as Exponential Linear Unit (ELU).
- An **output layer** with as many neurons as the number of possible actions, each outputting the Q-value for a corresponding action.

9 AI Ethics

9.1 Ethical Frameworks

9.1.1 Rights-Based Ethics

The ethical principle in rights-based ethics is grounded in the idea that what is ethical aligns with the inherent rights of all individuals. Actions are evaluated based on whether they respect or violate these fundamental rights.

9.1.2 Deontology

Deontology asserts that the ethicality of an action is determined by whether the action itself is intrinsically right or wrong, independent of its consequences.

9.1.3 Virtue Ethics

Virtue ethics focuses on the cultivation of good character traits. Ethical behavior is achieved by nurturing virtues such as honesty, courage, and compassion.

9.1.4 Care-Based Ethics

Care-based ethics emphasizes the importance of care, empathy, and interpersonal relationships. What is ethical is defined by the emphasis on understanding and addressing the needs of others.

9.1.5 Utilitarianism

Utilitarianism determines the ethicality of an action based on its outcomes. The central tenet is to maximize the greatest good for the greatest number of people.

9.2 Rights-Based Ethics in the Context of AI

Rights-based ethics emphasizes that ethical principles align with universal human rights, ensuring that actions respect the rights of all individuals. In the context of AI, this framework is closely tied to foundational documents such as:

1. **The Universal Declaration of Human Rights**: A seminal document that outlines fundamental human rights universally recognized and protected.
2. **The European Convention on Human Rights**: A regional treaty that guarantees human rights and freedoms.

9.2.1 Key Rights in AI Ethics

The ethical integration of AI must address the following rights to ensure its development and use are aligned with human rights principles:

1. **Right to Privacy**: AI systems must safeguard individual privacy, preventing unauthorized access to personal data.
2. **Right to Non-Discrimination**: AI technologies must be designed and implemented to avoid biases and ensure equitable treatment for all.
3. **Right to a Fair Trial**: AI applications in legal or judicial contexts must uphold fairness and transparency, ensuring just processes and outcomes.

9.3 The COMPAS Algorithm and Machine Bias

9.3.1 Introduction to Machine Bias

Machine bias refers to the systemic biases embedded in algorithms, often reflecting societal inequalities. The COMPAS (Correctional Offender Management Profiling for Alternative Sanctions) algorithm is used to predict the likelihood of individuals committing future crimes. However, investigations have revealed that it exhibits racial biases, disproportionately categorizing individuals based on race.

9.3.2 Case Study: COMPAS Predictions

1. **Vernon Prater (white):**
    - **Prior Offenses:** Two armed robberies and one attempted armed robbery.
    - **Subsequent Offenses:** One instance of grand theft.
    - **Risk Assessment:** Rated as low risk with a score of 3.
2. **Brisha Borden (black):**
    - **Prior Offenses:** Four juvenile misdemeanors.
    - **Subsequent Offenses:** None.
    - **Risk Assessment:** Rated as high risk with a score of 8.

9.3.3 Key Ethical Principles in AI

To address biases in AI systems like COMPAS, the following ethical principles are essential:

1. **Fairness and Non-Discrimination:** AI systems must promote social justice, fairness, and inclusivity, ensuring that their benefits are accessible to all without reinforcing discrimination.
2. **Transparency and Explainability:** The ethical deployment of AI systems requires transparency in their decision-making processes and explainability to users, balancing privacy and security.
3. **Human Oversight and Determination:** AI systems should not replace ultimate human responsibility and accountability, ensuring human oversight in critical decision-making contexts.

9.4 Uber Self-Driving Car Fatality

9.4.1 Incident Overview

In 2018, an Uber self-driving car was involved in the first fatal autonomous vehicle crash, resulting in the death of a pedestrian. The incident raised significant questions regarding the safety and accountability of autonomous systems:

1. The test driver of the vehicle was charged with endangerment in the pedestrian's death case.
2. It was noted that while driverless cars are generally safer than human drivers, they exhibit weaknesses in certain scenarios, such as making turns.

9.4.2 Key Ethical Considerations

This case highlights the following critical ethical principles for AI systems:

1. **Safety and Security:** AI systems must avoid unwanted harms and address vulnerabilities to safety risks and security breaches. Proactive measures are necessary to mitigate potential hazards.
2. **Responsibility and Accountability:** AI systems must be auditable and traceable. Oversight, impact assessments, audits, and due diligence mechanisms are essential to prevent conflicts with human rights norms and ensure compliance with ethical standards.

9.5 Clearview AI Facial Recognition

9.5.1 Overview of Clearview AI

Clearview AI is a facial recognition technology company that has garnered significant controversy for its practices. The company creates a database by scraping publicly available images from the internet, which are then used to identify individuals. Critics argue that this practice poses severe risks to privacy and can lead to a dystopian future.

9.5.2 Ethical Issues and Consequences

1. **Privacy Concerns:** Clearview AI's practices threaten to undermine privacy rights by collecting and utilizing images without consent. Such actions are widely regarded as unethical and potentially illegal.
2. **Legal Actions:** Clearview AI has faced substantial fines, such as a $33 million penalty, for maintaining an illegal database of facial images.

9.5.3 Key Ethical Principles for Facial Recognition

1. **Right to Privacy and Data Protection:** Privacy must be upheld throughout the AI lifecycle. Adequate frameworks for data protection must be implemented to safeguard individuals' rights.
2. **Proportionality and Do No Harm:** The application of AI technologies should not exceed the necessity of achieving a legitimate aim. Risk assessments must be conducted to prevent harm resulting from their use.

9.6 Chatbots and Ethical Challenges

9.6.1 Overview of Chatbot-Related Incidents

The increasing deployment of chatbots has led to ethical concerns, particularly when they interact with vulnerable individuals. Some documented incidents include:

1. A Snapchat chatbot reportedly provided detailed advice to children on self-harm.
2. A man ended his life after allegedly being encouraged by an AI chatbot to sacrifice himself for climate change.
3. Questions have arisen about whether AI systems can be held accountable for influencing tragic decisions, such as a teen's suicide.

9.6.2 Key Ethical Principles in Chatbot Deployment

To mitigate harm and ensure the responsible use of chatbots, the following principles are critical:

1. **Proportionality and Do No Harm:** AI systems should not go beyond what is necessary to achieve a legitimate purpose. Risk assessments must be conducted to prevent harm resulting from chatbot interactions.
2. **Responsibility and Accountability:** AI systems should be auditable and traceable, with mechanisms in place to ensure oversight and compliance with human rights norms. Proper impact assessments and due diligence are necessary to address potential risks.

9.6.3 Resource Intensity of AI Technologies

The development and deployment of advanced AI systems like ChatGPT involve significant resource consumption:

1. **Water Usage:** Large-scale AI technologies require extensive water resources for cooling data centers. For instance, the infrastructure supporting ChatGPT was constructed in Iowa, utilizing a substantial amount of water.
2. **Energy Consumption:** The energy demands of AI systems are increasing exponentially. It is projected that AI technologies could eventually consume as much electricity as an entire country.

9.6.4 Sustainability Challenges in AI

The proliferation of generative AI, including ChatGPT, has raised concerns regarding its impact on global sustainability. These technologies are seen as contributors to sustainability challenges due to their resource-intensive nature.

9.6.5 Key Ethical Principle: Sustainability

1. **Sustainability:** AI systems should be evaluated for their sustainability impact, adhering to evolving goals such as those outlined in the United Nations' Sustainable Development Goals. Efforts should focus on minimizing resource consumption and environmental harm.

9.7 The UNESCO principles

To ensure the ethical development and deployment of AI systems, the following principles are essential:

1. **Proportionality and Do No Harm:** AI systems should not exceed the scope necessary to achieve legitimate aims. Risk assessments are crucial to prevent unintended harm from such technologies.
2. **Safety and Security:** AI systems must avoid unwanted harms (safety risks) and address vulnerabilities to attacks (security risks).
3. **Right to Privacy and Data Protection:** Privacy should be safeguarded throughout the AI lifecycle, supported by robust data protection frameworks.
4. **Multi-Stakeholder and Adaptive Governance & Collaboration:** Respect for international law and national sovereignty in data usage is vital. Diverse stakeholder participation is necessary for inclusive AI governance.
5. **Responsibility and Accountability:** AI systems should be auditable and traceable. Oversight, impact assessments, audits, and due diligence are required to align AI systems with human rights and environmental standards.
6. **Transparency and Explainability:** AI systems must exhibit transparency and explainability (T&E) appropriate to their context, balancing it with other principles such as privacy and security.
7. **Human Oversight and Determination:** AI systems should not replace human responsibility. Member states must ensure mechanisms for retaining ultimate human accountability.
8. **Sustainability:** AI technologies should align with evolving sustainability goals, including those outlined in the United Nations' Sustainable Development Goals.
9. **Awareness and Literacy:** Open and accessible education on AI and data, including ethics training, civic engagement, and digital skills, should be promoted to enhance public understanding.
10. **Fairness and Non-Discrimination:** AI actors must foster social justice and fairness while ensuring non-discrimination. An inclusive approach is necessary to make AI's benefits accessible to all.