An Intro to Machine Learning

compiled by Yaswanth Ram Kumar and Harshith Pendela

for AnimAI - SoC 2025

"50 years ago, we thought the brain was a computer, a digital computer. But you see, the brain has no operating system. It has no programming. It has no windows. It has no CPU. 50 years later, after this wild goose chase, we now understand that the brain is a pattern-seeking neural network, a learning machine, and it learns and rewires every time it learns something new."

— Michio Kaku, CCNY

This document is compiled with pure enthusiasm by us. If there are any typos or mistakes, please write to us at 23b1277@iitb.ac.in or 23b1207@iitb.ac.in. Any suggestions or improvements are also welcomed. Thank you.

Contents

1 Introduction

Why do we need Machine Learning? Is it just about building a function, like in linear regression? In one way, that's partially true. In linear regression, we are given inputs and their corresponding outputs, and we try to observe the trend by plotting the data. If the relationship appears linear, we fit a straight line (a linear function) to approximate that trend. If the pattern seems quadratic or follows some other curve, we apply the corresponding regression technique to model it. Essentially, we try to find a mathematical function that maps inputs to outputs.

But if ML is doing the same thing, why is it considered more advanced and sophisticated? The key difference is that the functions machine learning models learn are not simple polynomials or predefined equations, they can be highly complex and flexible. Designing and training such models involves a variety of subtle challenges, such as the vanishing gradient problem during backpropagation, getting stuck in local minima, and ensuring the model generalizes well to unseen data. We will discuss these problems later in this document. These challenges aren't just mathematical, they are also intuitive and geometric in nature, and understanding them requires a shift in perspective. In many real world problems, simple functions aren't sufficient. We want our machines to learn from experience, adapt like humans, and discover hidden patterns in the data, that's what ML enables.

2 Different Types of Learning

2.1 Supervised Learning

In supervised learning, the model is trained on a labeled dataset — that means for every input, the correct output is also provided. The goal is to learn a function that maps inputs to outputs by minimizing the difference between the predicted and actual outputs.
Example: Predicting house prices based on features like size, location, number of bedrooms, etc and you have some data on collection of houses and their features helping you to proceed.
Common Algorithms: Linear Regression, Logistic Regression, Support Vector Machines, Neural Networks, Decision Trees.

2.2 Unsupervised Learning

In **unsupervised learning**, we give the model data **without any labels**. This means the model doesn't know the "correct" answers. Instead, it tries to **find patterns or groupings** in the data on its own.

Example:

A company wants to group customers based on their shopping behavior. The model analyzes purchase data and automatically forms groups (e.g., frequent buyers, discount seekers) *without being told* what those groups are.

Common Algorithms:

- **K-Means Clustering** – Groups data points into clusters based on similarity.
- **PCA (Principal Component Analysis)** – Reduces the number of features while keeping the most important information.
- **Autoencoders** – A type of neural network that learns to compress and reconstruct data, helping to find patterns or reduce noise.

2.3 Reinforcement Learning (Brief Idea)

Reinforcement Learning (RL) is about training an agent to learn from interactions with its environment to achieve a goal. It senses the environment, takes actions, and learns through trial and error based on reward feedback.

RL stands apart from supervised and unsupervised learning because it focuses on decision-making under uncertainty and delayed rewards.

Generational / Evolutionary RL methods like genetic algorithms explore solutions by evolving policies over generations. While useful in some cases, they don't learn from each individual's experience like typical RL methods do, and are generally considered less efficient.

This learning is the closest to the learning of an animal or a human.

Example: A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counterreplies—and by immediate, intuitive judgements of the desirability of particular positions and moves

Key Concepts: Agent, Environment, Actions, Rewards, Policy.

This week we are going to work with supervised and unsupervised learning. If time permits, we can have an exercise on RL, but RL itself is a very involved sub-domain in ML, so fingers crossed!

3 Neural Networks

So let's start building our core in ML. To start-off, you might have heard about neural networks. Neural networks are computing systems inspired by biological neural networks. They consist of interconnected nodes (neurons) that process information.

3.1 Neural Network Architecture

Below is a neural network with 4 inputs and 2 outputs for binary classification with two hidden layers:

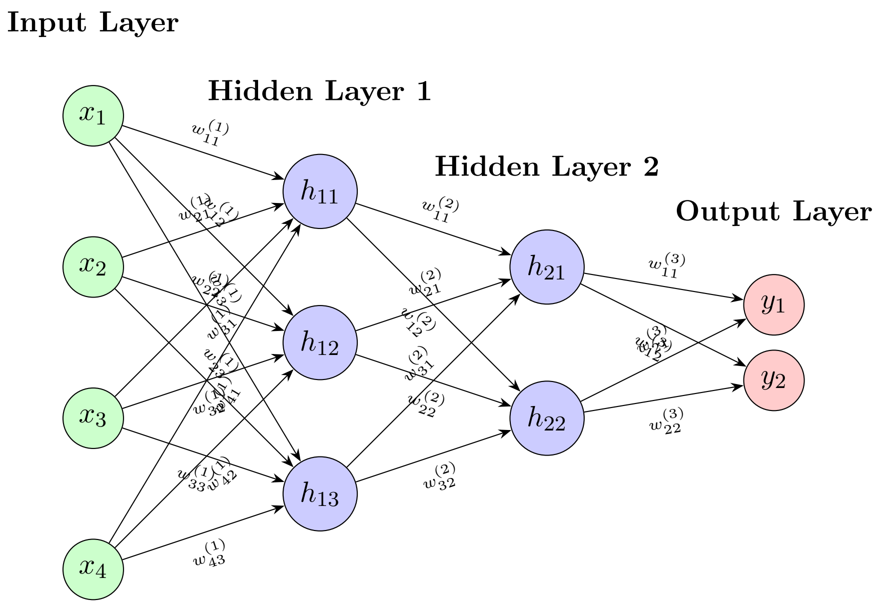


Figure 1: Neural Network Architecture with 4 inputs, 2 hidden layers, and 2 outputs

equations for each layer are:

First Hidden Layer:

$$h_{11} = \sigma\left(\sum_{i=1}^{4} w_{i1}^{(1)} x_i + b_1^{(1)}\right) \tag{1}$$

$$h_{12} = \sigma\left(\sum_{i=1}^{4} w_{i2}^{(1)} x_i + b_2^{(1)}\right) \tag{2}$$

$$h_{13} = \sigma\left(\sum_{i=1}^{4} w_{i3}^{(1)} x_i + b_3^{(1)}\right) \tag{3}$$

Second Hidden Layer:

$$h_{21} = \sigma\left(\sum_{j=1}^{3} w_{j1}^{(2)} h_{1j} + b_1^{(2)}\right) \tag{4}$$

$$h_{22} = \sigma\left(\sum_{j=1}^{3} w_{j2}^{(2)} h_{1j} + b_2^{(2)}\right) \tag{5}$$

Output Layer:

$$y_1 = \sigma\left(\sum_{j=1}^{2} w_{j1}^{(3)} h_{2j} + b_1^{(3)}\right) \tag{6}$$

$$y_2 = \sigma\left(\sum_{j=1}^{2} w_{j2}^{(3)} h_{2j} + b_2^{(3)}\right) \tag{7}$$

where σ is the activation function and b is the bias. Don't worry if these words sound new — just remember for now: the activation is a small function we apply at each step, and the bias is just an extra number we add. We will understand both of them properly very soon.

3.2 Typical Structure

3.2.1 Input Layer

- Receives raw data (like pixel values of an image)
- Each neuron represents one feature of your data
- No computation happens here — just data entry

3.2.2 Hidden Layer(s)

- Where the "magic" happens — patterns are detected
- Can have multiple hidden layers (that's "deep" learning!)
- Each neuron combines inputs and applies mathematical transformations

3.2.3 Output Layer

- Produces final predictions
- For binary classification: 1 neuron (yes/no)
- For multi-class: multiple neurons (one per class)

3.3 Forward Propagation

Forward propagation is the process of passing input data through the neural network to generate an output. Each neuron performs a weighted sum of its inputs, adds a bias, and passes the result through an activation function:

$$h = \sum_i w_i x_i + b, \quad a = \sigma(h)$$

where:

- x_i are the inputs,
- w_i are the corresponding weights,
- b is the bias term,
- σ is the activation function (like ReLU, Sigmoid, Tanh),
- a is the neuron's output.

This process continues layer by layer, from input to hidden to output.

3.4 Activation Functions

Activation functions are the key to neural networks' power. They transform the weighted sum of inputs into the neuron's output, introducing crucial non-linearity into the system.

3.4.1 Why Do We Need Non-linearity?

Consider a neural network without activation functions. Each layer would simply perform a linear transformation:

$$\text{Layer 1: } \mathbf{h_1} = \mathbf{W_1 x} + \mathbf{b_1} \tag{8}$$

$$\text{Layer 2: } \mathbf{h_2} = \mathbf{W_2 h_1} + \mathbf{b_2} = \mathbf{W_2}(\mathbf{W_1 x} + \mathbf{b_1}) + \mathbf{b_2} \tag{9}$$

$$= \mathbf{W_2 W_1 x} + \mathbf{W_2 b_1} + \mathbf{b_2} \tag{10}$$

This can be simplified to a single linear transformation: $\mathbf{y} = \mathbf{W_{combined} x} + \mathbf{b_{combined}}$. No matter how many layers you stack, you're still just performing linear regression! This severely limits the network's ability to model complex, non-linear relationships.

Activation functions break this linearity, allowing networks to:

- **Model complex patterns:** Non-linear functions can approximate any continuous function (Universal Approximation Theorem)
- **Learn hierarchical features:** Early layers detect simple patterns, deeper layers combine them into complex concepts
- **Introduce decision boundaries:** Create regions in input space for classification

3.4.2 Common Activation Functions

- **Sigmoid:** Squashes values between 0 and 1

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, maximum value is 0.25

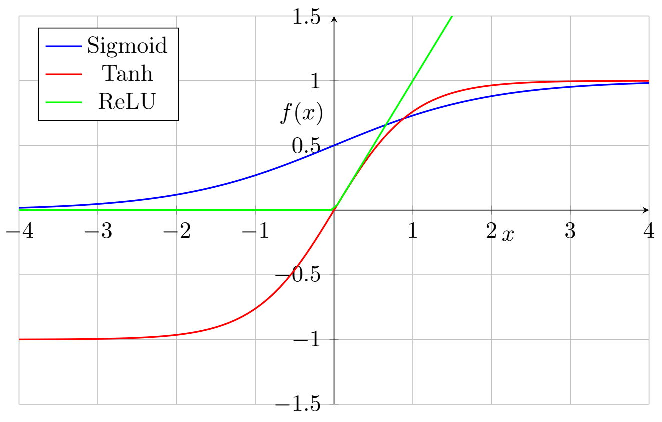


Figure 2: Comparison of Common Activation Functions

- **Tanh (Hyperbolic Tangent):** Squashes values between -1 and 1

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{12}$$

  Derivative: $\tanh'(x) = 1 - \tanh^2(x)$, maximum value is 1

- **ReLU (Rectified Linear Unit):** Returns input if positive, 0 if negative

$$\text{ReLU}(x) = \max(0, x) \tag{13}$$

  Derivative: $\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$

3.5 Loss Functions

To measure the performance of a neural network, we define a loss function $L(y, \hat{y})$ that quantifies the difference between the predicted output $\hat{y}$ and the true output y. A few commonly used loss functions include:

- **Mean Squared Error (MSE)** for regression:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- **Binary Cross-Entropy** for binary classification:

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})]$$

- **Categorical Cross-Entropy** for multi-class classification:

$$L(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

3.6 Backpropagation

Backpropagation is the process of calculating gradients of the loss with respect to each weight in the network using the chain rule of calculus.

1. **Forward Pass:** Compute the output $\hat{y} = f(x; \theta)$.

2. **Compute Loss:** Evaluate the loss function $L(y, \hat{y})$.
3. **Backward Pass:** Compute the gradients $\frac{\partial L}{\partial w}$ using chain rule:
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$
4. **Update Weights:** Update each weight w using a learning rate η:
$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

Don't worry if math seems like a sudden jump, it is fairly intuitive when you understand it. The learning rate η, is a small number that decides how big each step should be when a model updates its parameters during training. It plays an important role in how quickly or slowly a model learns. If the learning rate is too large, the steps may overshoot the best solution, causing the training to become unstable or fail. If it's too small, the model learns very slowly and may get stuck without finding the best answer. So, choosing a good learning rate is important to make sure the training is both fast and effective.

3.7 Optimization Methods

Once we compute the gradients using backpropagation, we need to use them to improve our model. This is done by updating the weights in the network to reduce the loss. But how exactly do we update the weights? There are different strategies, each with its pros and cons:

- **Batch Gradient Descent (BGD):** In BGD, we use the entire training dataset to compute the gradient and then update the weights. This gives accurate updates but is very slow, especially for large datasets.
- **Stochastic Gradient Descent (SGD):** In SGD, we update the weights after every single data point. This is much faster and introduces randomness, which can help the model escape local minima (bad solutions that are not the best possible). But this randomness can also make the path to the best solution a bit shaky.
- **Mini-Batch Gradient Descent:** This is a middle ground, we split the dataset into small batches (like 32 or 64 samples) and update weights for each batch. It combines the speed of SGD with the stability of BGD. Most modern ML models use mini-batch training.

Why randomness helps? In complex problems, the loss surface (the curve that shows how good or bad your model is) can have many local dips (local minima). Pure BGD can get stuck in these, more on this in Section 4.

Please see the below illustration for a better understanding, The loss function used for this illustration is:
$$\mathcal{L}(\boldsymbol{\theta} \mid \hat{y}, y) = \frac{1}{2}(\theta_1^2 + \theta_2^2) + 0.3\sin(3\theta_1)\cos(3\theta_2) + 2 \tag{14}$$
where $\boldsymbol{\theta} = [\theta_1, \theta_2]^T$ represents the parameter vector, $\hat{y}$ denotes predicted values, and y represents actual target values. Here $\mid$ means the loss function dependent on the parameter vector $\boldsymbol{\theta}$ given the true and predicted values.

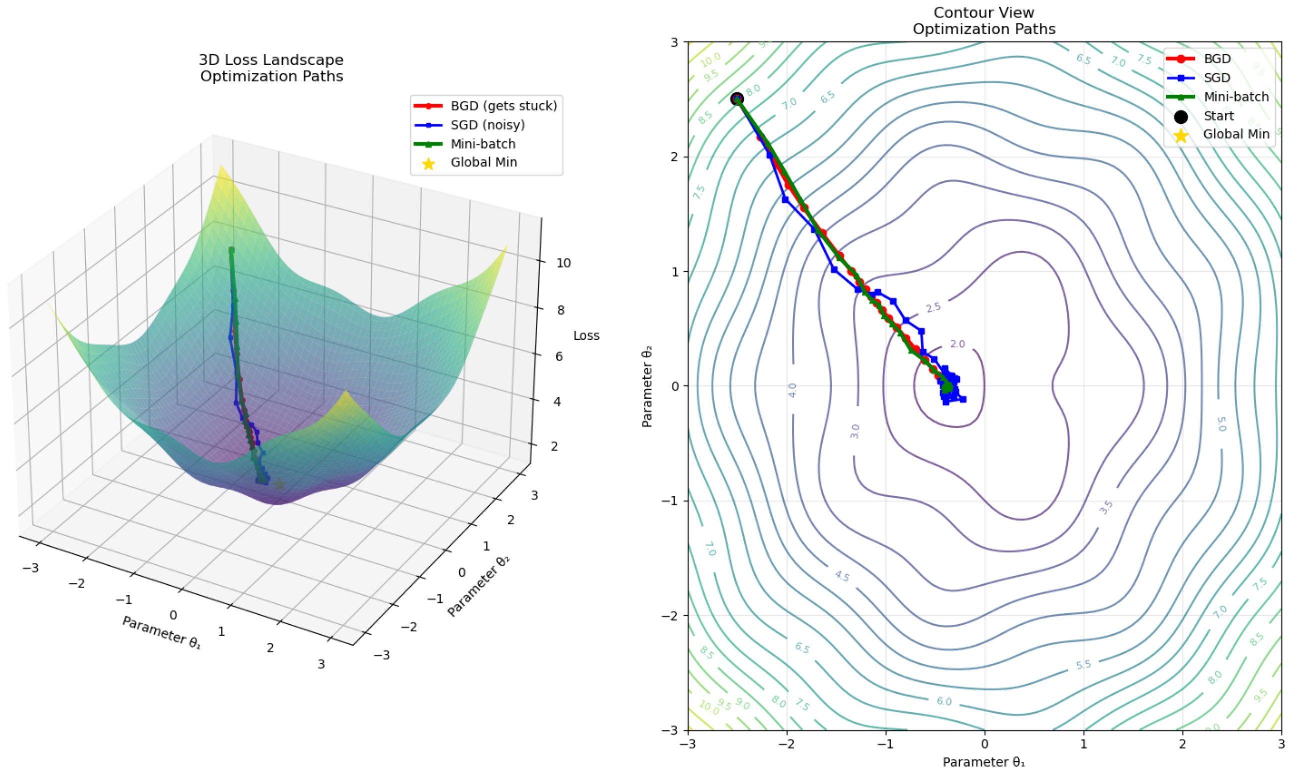


Figure 3: Applying all three strategies to the loss function $\mathcal{L}(\boldsymbol{\theta} \mid \hat{y}, y)$

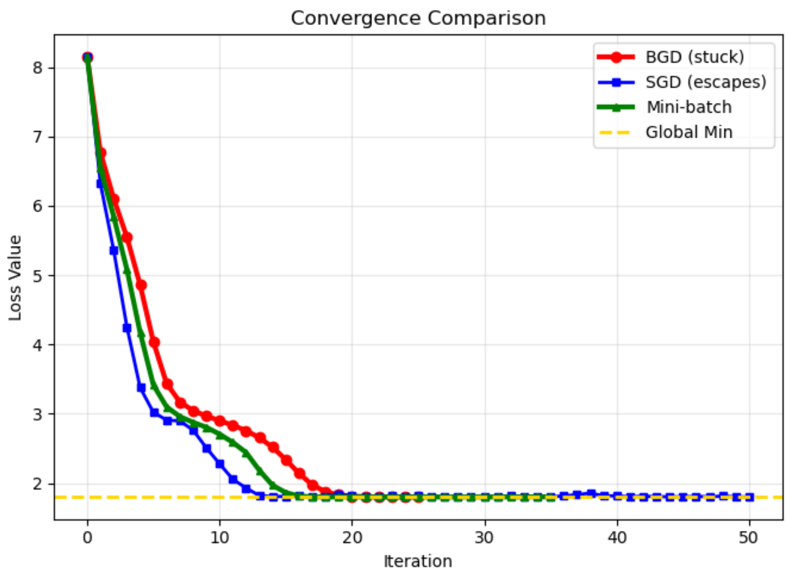


Figure 4: Comparing convergence in all three strategies to the loss function $\mathcal{L}(\boldsymbol{\theta} \mid \hat{y}, y)$

Note: There are more advanced optimizers like Adam, which combine benefits of SGD with momentum and adaptive learning rates. We will not go into the math here, but we use Adam in our projects, it is available directly in TensorFlow. You are encouraged to read about it yourself.

From the plots, we can observe that Batch Gradient Descent (BGD) provides smooth and stable convergence but is relatively slow. Stochastic Gradient Descent (SGD), on the other hand, converges faster initially but exhibits significant fluctuations due to its high variance. Mini-batch Gradient Descent achieves a balance between the two, offering faster convergence than BGD and more stability than SGD. Overall, mini-batch methods provide a practical trade-off between computational efficiency and convergence stability.

```
BGD: Position (-0.385, 0.007)
     Loss: 1.800

SGD: Position (-0.376, 0.026)
     Loss: 1.801

Mini-batch: Position (-0.390, -0.008)
            Loss: 1.800

Global Min: Position (-0.385, 0.006)
            Loss: 1.800
```

Listing 1: Optimization Results

4 Escaping the Local Minima Problem with Stochastic Gradient Descent

One of the most fascinating challenges in optimization is the tendency for deterministic algorithms to become trapped in suboptimal solutions. Consider a scenario where our optimization landscape contains multiple valleys, each seemingly promising a solution, yet only one represents the true global optimum.

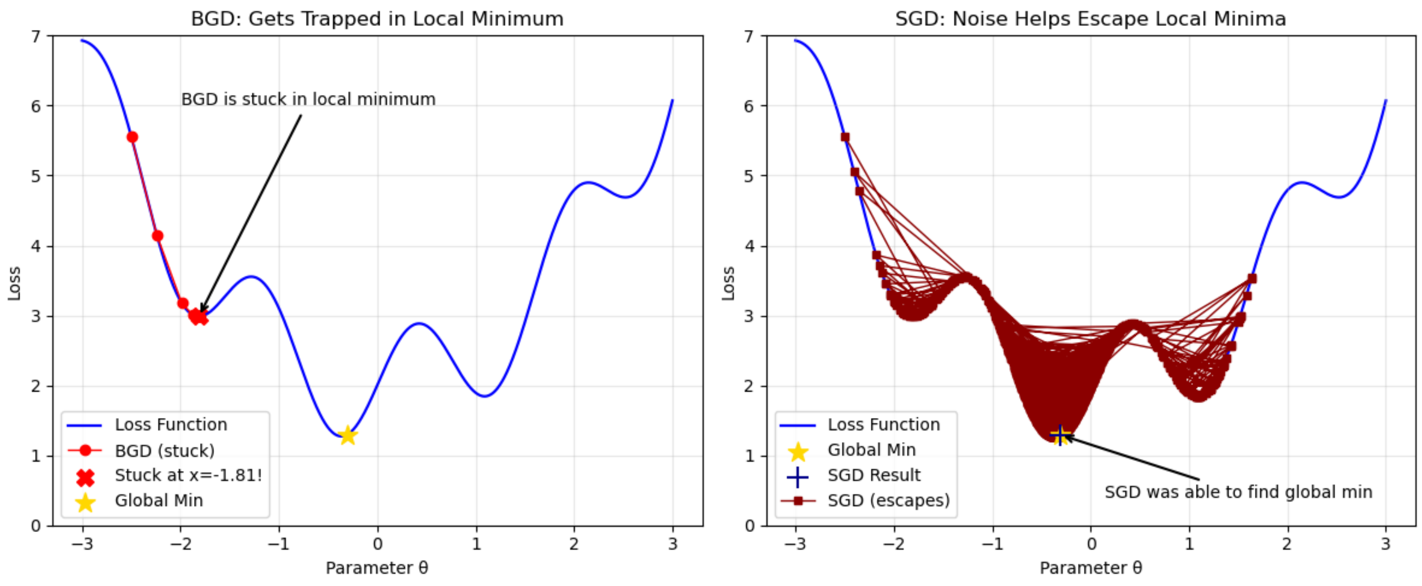


Figure 5: Optimization landscape showing local minima traps and global minimum

The loss function used for this illustration is:

$$\mathcal{L}(\theta) = 0.5\theta^2 + 0.8\sin(4\theta) + 2 \tag{15}$$

This function creates a landscape with multiple local minima, mimicking real-world optimization challenges where the objective function is non-convex.

Stochastic */stoh-KAS-tik/ adj.* having a random probability distribution or pattern that may be analysed statistically but may not be predicted precisely.

To understand this problem setup, it is **again** important to see the main difference between Batch Gradient Descent and Stochastic Gradient Descent. This difference lies in how they compute and use gradient information, which affects how well they can escape from local minima.

Batch Gradient Descent calculates the gradient using the **entire dataset** at each step. This gives the exact direction of steepest descent and leads to stable updates. However, once it reaches a local minimum where the gradient becomes zero, it gets stuck there since it has no randomness to help it move away.

Stochastic Gradient Descent uses **individual data points** or small subsets to compute the gradient. This introduces **randomness** in the updates. The gradient at each step is only an estimate, and it changes with every data point.

The important idea is that this randomness helps SGD escape local minima. When SGD reaches a local minimum, the noisy updates caused by using random samples can push it out and allow it to keep exploring the loss surface.

What seems like a disadvantage (noisy gradients) becomes helpful. This randomness gives SGD the ability to explore complex and nonconvex loss functions like Equation 15 better than batch methods. It helps the algorithm reach better solutions that batch gradient descent might miss, which is exactly what we can see in Figure 5.

5 Vanishing Gradient Problem

In training deep neural networks, especially those with many layers, one major challenge is the **vanishing gradient problem**. This typically occurs during backpropagation, where we compute the gradient (i.e., how much we should adjust the weights) by applying the chain rule through many layers.

Most commonly, the problem is caused by activation functions like the **sigmoid** we've seen earlier, Its derivative always lies between 0 and 0.25. So, when backpropagating through multiple layers, each layer multiplies the previous gradient by a number less than 1. After several layers, the gradient becomes extremely small, nearly zero and this causes the earlier (input-side) layers to learn very slowly, or not at all. This is the **vanishing gradient problem**.

Let us look at a small example to observe this. Consider a feed-forward neural network with 4 layers (including input and output), trained using sigmoid activation. We'll track the update to a specific weight in each layer after 5 epochs.

Layer	**Initial Weight**	**Weight After 1 Epoch**	**Weight After 5 Epochs**
Input $\rightarrow$ Layer 1	0.45	0.45001	0.45002
Layer 1 $\rightarrow$ Layer 2	0.30	0.3025	0.3078
Layer 2 $\rightarrow$ Layer 3	0.60	0.612	0.648
Layer 3 $\rightarrow$ Output	0.80	0.815	0.860

Table 1: Observation of Weight Updates Across Layers in a Network with Sigmoid Activation

As you can observe, the weights close to the output layer (Layer 3 to Output) change significantly compared to the weights close to the input layer which barely change at all. This shows how gradients effectively vanish as we go backward, halting learning in early layers.

Why Sigmoid Is a Problem Here?

Sigmoid compresses inputs to a small range: $(0, 1)$. For large positive or negative input values, the slope becomes very flat (i.e., derivative ≈ 0). During backpropagation, gradients shrink as they are multiplied by the derivatives. In deep networks, this leads to nearly zero gradients in early layers.

The simplest fix we could employ for situations like this is to use **ReLU** instead of sigmoid, **Why would ReLU work? think for yourself!** there are other techniques like batch normalization, which we will explore in later stages if needed.

Now we will see some technically important data analysis related stuff.

6 Data Preprocessing

Data preprocessing is like preparing ingredients before cooking. Raw data is often messy and needs to be cleaned and organized before we can feed it to our machine learning models. Think of it as getting your data ready for the algorithm to understand and learn from.

6.1 Why Do We Need Data Preprocessing?

Real world data is rarely perfect. It might have missing values, different scales, or formats that machines cannot understand directly. For example, if we have ages ranging from 18 to 80 and salaries ranging from 30,000 to 200,000, the salary numbers are much larger and might dominate the learning process.

6.2 Common Preprocessing Steps

6.2.1 Handling Missing Data

Sometimes our dataset has empty spots where information is missing. We need to decide what to do with these gaps, like we did in Assignment-01.

Listing 2: Handling Missing Data

```python
import pandas as pd
import numpy as np

# Load data
data = pd.read_csv('dataset.csv')

# Check for missing values
print("Missing values per column:")
print(data.isnull().sum())

# Option 1: Remove rows with missing values
data_cleaned = data.dropna()

# Option 2: Fill missing values with average
data['age'].fillna(data['age'].mean(), inplace=True)

# Option 3: Fill missing values with most common value
data['category'].fillna(data['category'].mode()[0], inplace=True)
```

6.2.2 Data Normalization and Scaling

When features have very different ranges, we need to bring them to similar scales so that no single feature dominates others.

Listing 3: Data Scaling

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# StandardScaler: converts data to have mean=0 and std=1
scaler = StandardScaler()
```

```
data_scaled = scaler.fit_transform(data)

# MinMaxScaler: converts data to range [0,1]
minmax_scaler = MinMaxScaler()
data_normalized = minmax_scaler.fit_transform(data)
```

6.2.3 Categorical Data Encoding

Machines work with numbers, not text. We need to convert categorical data (like colors, categories) into numerical format.

1. One Hot Encoding:

One hot encoding converts each category into a binary vector. Each category gets its own column with 1 for presence and 0 for absence.

Why Not Just Use Numbers?

You might wonder: why not just assign numbers to categories like this?

$$\text{red} = 1 \tag{16}$$
$$\text{blue} = 2 \tag{17}$$
$$\text{green} = 3 \tag{18}$$

The problem is that this creates **fake mathematical relationships**. When we assign numbers like this, we accidentally tell the machine that:

- Green (3) is "bigger" than blue (2)
- Blue (2) is "twice as much" as red (1)
- Green (3) minus red (1) equals blue (2)

But these relationships are meaningless! Colors do not have a natural order or mathematical relationship.

Why Orthogonal Vectors Work Better

One hot encoding creates vectors where each category lives in its own dimension:

$$\text{red} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{19}$$
$$\text{blue} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{20}$$
$$\text{green} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{21}$$

These vectors are called **orthogonal** because they are perpendicular to each other. This means:

Equal Distance Property: The distance between any two categories is always the same:

$$\|\text{red} - \text{blue}\|_2 = \sqrt{(1-0)^2 + (0-1)^2 + (0-0)^2} = \sqrt{2} \tag{22}$$
$$\|\text{red} - \text{green}\|_2 = \sqrt{(1-0)^2 + (0-0)^2 + (0-1)^2} = \sqrt{2} \tag{23}$$
$$\|\text{blue} - \text{green}\|_2 = \sqrt{(0-0)^2 + (1-0)^2 + (0-1)^2} = \sqrt{2} \tag{24}$$

This equal distance means no category is "closer" to another category, which is exactly what we want for categorical data.

No False Mathematical Relationships: With one hot encoding:

- You cannot add colors: red + blue ≠ green
- You cannot say one color is "bigger" than another
- Each category is completely independent

Listing 4: One Hot Encoding

```python
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Example: Converting colors to numbers
colors = ['red', 'blue', 'green', 'red', 'blue']

# One hot encoding
encoder = OneHotEncoder()
colors_encoded = encoder.fit_transform(colors.reshape(-1, 1))

# Result:
# red   -> [1, 0, 0]
# blue  -> [0, 1, 0]
# green -> [0, 0, 1]

# Verify equal distances
red = np.array([1, 0, 0])
blue = np.array([0, 1, 0])
green = np.array([0, 0, 1])

print("Distance red to blue:", np.linalg.norm(red - blue))    #  2
print("Distance red to green:", np.linalg.norm(red - green))  #  2
print("Distance blue to green:", np.linalg.norm(blue - green)) #  2
```

Key Takeaway: One hot encoding uses orthogonal vectors because it preserves the categorical nature of data without creating false mathematical relationships between categories.

2. Handling Image Pixels, for example MNIST Image Flattening:

Image flattening is a crucial preprocessing step that transforms 2D pixel matrices into 1D feature vectors suitable for machine learning algorithms. In the case of MNIST handwritten digits, each grayscale image is represented as a 28×28 pixel grid where each pixel contains an intensity value between 0 (black) and 255 (white). The flattening process systematically converts this 2D structure into a single row vector by concatenating pixel values row by row, resulting in a 784-dimensional feature vector ($28 \times 28 = 784$). This transformation preserves all pixel information while making it compatible with traditional ML models that expect flat input vectors. Each position in the flattened vector corresponds to a specific pixel location in the original image, maintaining spatial relationships through positional encoding. The resulting feature vector can then be normalized (typically to [0,1] range) and fed directly into neural networks, SVMs, or other algorithms for classification tasks.

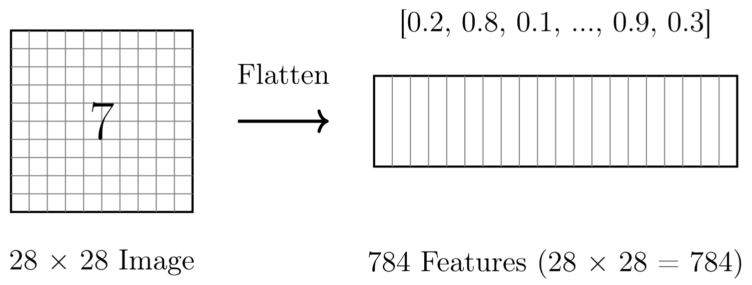


Figure 6: Converting 28×28 MNIST Image to 784 Feature Vector

6.3 Data Splitting

Understanding datasets is fundamental to machine learning - a dataset is simply a collection of examples that we use to teach our algorithms. Think of it like a textbook filled with practice problems and their solutions. Famous datasets include MNIST (handwritten digits 0-9, `Assignment-02` is based on this dataset only), CIFAR-10 (small color images of objects like cars, birds, cats), ImageNet (millions of labeled photographs), and Titanic (passenger data for survival prediction). Just as you wouldn't give a student the same exact problems on their final exam that they practiced with, we split our dataset into distinct portions to properly evaluate our model's true learning ability.

The training set is like giving students practice problems with answer keys - our model learns patterns, relationships, and rules from these examples. The testing set acts as the final exam with completely unseen problems, allowing us to measure how well the model generalizes beyond memorization. This split is crucial because a model that simply memorizes training data (called overfitting) will fail miserably on new, real-world data, much like a student who only memorized specific problems without understanding underlying concepts. A typical split allocates 70-80% of data for training and 20-30% for testing, ensuring we have enough examples to learn from while maintaining a substantial evaluation set. This approach gives us confidence that our model will perform reliably when deployed in real applications, where it will encounter entirely new data it has never seen before.

7 Overfitting and Underfitting

The fundamental challenge in machine learning is finding the right balance between model simplicity and complexity. Consider preparing for an exam: memorizing only practice questions without understanding concepts leads to failure on new questions (overfitting), while insufficient study results in poor performance even on familiar material (underfitting). Machine learning models face this identical dilemma.

Definition 1 (Underfitting)**.** *A model exhibits **underfitting** when it is too simple to capture the underlying patterns in the data. The model performs poorly on both training and test datasets due to high bias.*

Definition 2 (Overfitting)**.** *A model exhibits **overfitting** when it learns the training data too well, including noise and random fluctuations. It achieves excellent training performance but fails to generalize to new, unseen data due to high variance.*

Definition 3 (Good Fit)**.** *The optimal balance where a model captures the underlying patterns without memorizing noise, achieving good performance on both training and test data.*

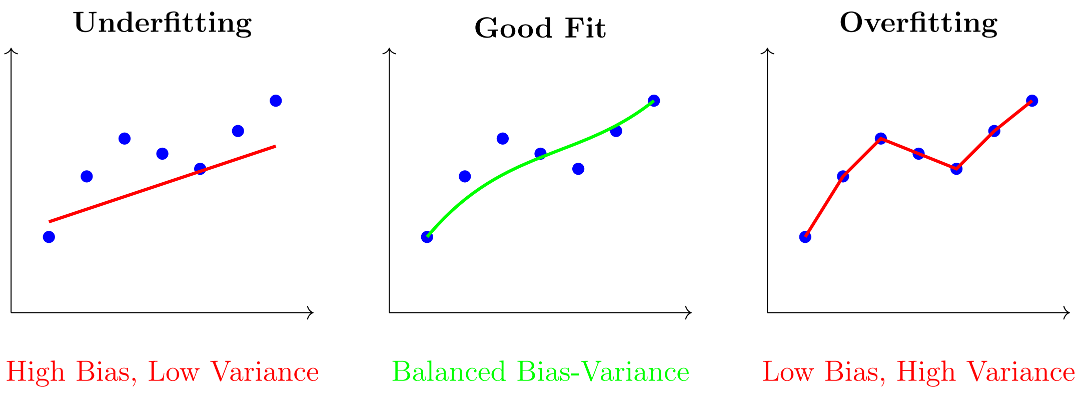


Figure 7: Visual comparison of underfitting, good fit, and overfitting scenarios

7.1 The Bias-Variance Tradeoff

The bias-variance tradeoff is fundamental to understanding overfitting and underfitting:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error} \tag{25}$$

- **Bias**: Error from oversimplified assumptions in the learning algorithm
- **Variance**: Error from sensitivity to small fluctuations in the training set
- **Irreducible Error**: Noise inherent in the problem

7.2 Training vs Validation Performance

The relationship between training and validation performance reveals the model's fitting behavior:

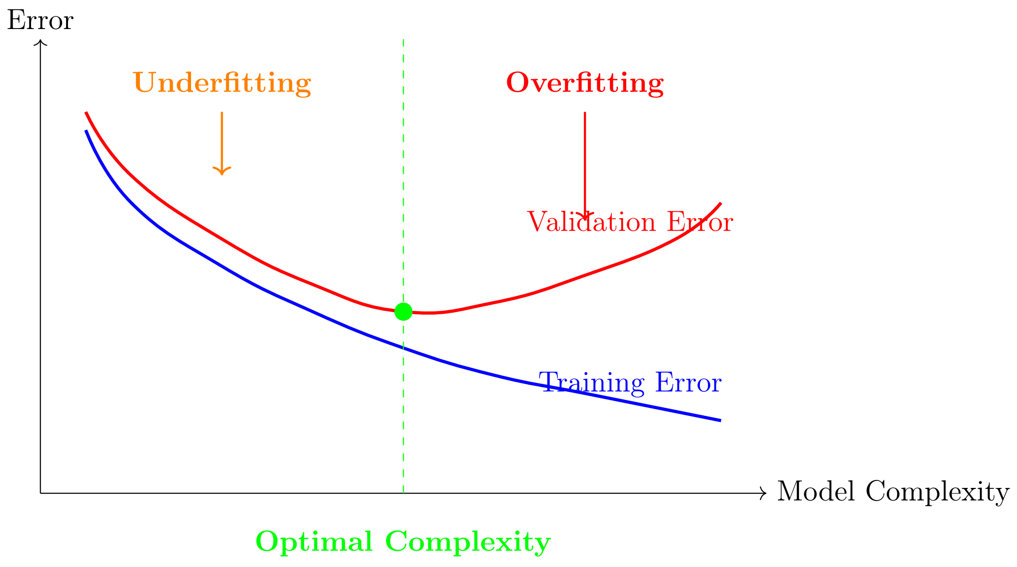


Figure 8: The relationship between model complexity and error rates

7.3 Detection Methods

7.3.1 Identifying Overfitting

1. **Performance Gap**: Large difference between training and validation accuracy
2. **Training Trajectory**: Training loss continues decreasing while validation loss increases
3. **Generalization Failure**: Excellent performance on training data, poor performance on test data

7.3.2 Identifying Underfitting

1. **Poor Performance**: Both training and validation accuracies remain low
2. **Plateau Effect**: Learning curves flatten early at suboptimal performance levels
3. **Consistent Underperformance**: Model fails to capture obvious patterns in the data

7.4 Regularization Techniques

7.4.1 L2 Regularization (Ridge Regression)

L2 regularization adds a penalty term to the loss function that discourages large weights:

$$\mathcal{L}_{\text{regularized}} = \mathcal{L}_{\text{original}} + \lambda \sum_{i=1}^{n} w_i^2 \tag{26}$$

$$= \mathcal{L}_{\text{original}} + \lambda \|\mathbf{w}\|_2^2 \tag{27}$$

where:

- λ is the regularization parameter (controls penalty strength)
- $\mathbf{w}$ represents the model weights
- $\|\mathbf{w}\|_2^2$ is the L2 norm (sum of squared weights)

Mathematical Intuition: L2 regularization encourages weight decay by penalizing large weights. During gradient descent, the weight update becomes:

$$w_{\text{new}} = w_{\text{old}} - \alpha \left(\frac{\partial \mathcal{L}}{\partial w} + 2\lambda w_{\text{old}} \right) \tag{28}$$

$$= (1 - 2\alpha\lambda) w_{\text{old}} - \alpha \frac{\partial \mathcal{L}}{\partial w} \tag{29}$$

The term $(1 - 2\alpha\lambda)$ acts as a decay factor, shrinking weights toward zero at each update.

7.4.2 Other Regularization Methods

To help neural networks generalize better and avoid overfitting, several regularization techniques are commonly used:

Dropout Dropout is a method where, during training, we randomly "turn off" (i.e., set to zero) some neurons in the network. This means that during each training step, the network looks a little different. This prevents any one neuron from relying too much on another and encourages the network to learn more robust, general patterns. At test time, all neurons are used, but with adjusted weights.

Early Stopping When training a model, we often observe that the performance on the training set keeps improving, but after a point, the performance on the validation set starts getting worse. Early stopping monitors this validation performance and stops training at the point where the model performs best on unseen data. This helps avoid overfitting.

Data Augmentation Instead of collecting more data, we can artificially increase the size of the training dataset by applying small changes or transformations to existing data. For images, this could include rotating, flipping, zooming, or adding noise. This helps the model see more variety and generalize better to new data.

Batch Normalization During training, the inputs to each layer can change, which makes learning harder. Batch normalization helps by normalizing the inputs to each layer — that is, it adjusts them to have zero mean and unit variance. This makes the training more stable and can sometimes act as a regularizer.

The key to successful machine learning lies in finding the optimal balance between bias and variance, achieved through careful model selection, appropriate regularization, and systematic validation strategies.

8 Overfitting Detection Exercise

Understanding overfitting is crucial for building reliable machine learning models. Let's work through a concrete example using a polynomial regression problem where we deliberately create and then solve an overfitting scenario.

Problem Setup

We are building a model to predict house prices based on house size in square feet. We have a simple dataset with just 50 houses, but we want to use a very flexible polynomial model to capture the relationship between size and price. This small dataset with a complex model creates perfect conditions for observing overfitting.

Listing 5: Problem Setup

```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split

# Generate synthetic house price data
np.random.seed(42)
n_samples = 50

# House sizes from 1000 to 3000 square feet
house_sizes = np.linspace(1000, 3000, n_samples)
# True relationship: price increases with size but with some noise
true_prices = 100 + 0.1 * house_sizes + 0.00002 * house_sizes**2
# Add realistic noise
noise = np.random.normal(0, 20, n_samples)
prices = true_prices + noise

# Reshape for sklearn compatibility
X = house_sizes.reshape(-1, 1)
y = prices

print(f"Dataset size: {n_samples} houses")
print(f"House size range: {house_sizes.min():.0f} to {house_sizes.max():.0f} sq
    ft")
print(f"Price range: ${prices.min():.0f} to ${prices.max():.0f}")
```

Creating the Overfitting Scenario

We design a neural network that is deliberately too complex for our small dataset. We create polynomial features up to degree 10, giving us 10 input features from our single house size measurement. Then we build a network with two hidden layers containing 50 neurons each. This gives us thousands of parameters to fit just 50 data points, creating a recipe for overfitting.

Listing 6: Creating Overfitting Scenario

```python
# Create polynomial features (degree 10 creates overfitting)
poly_features = PolynomialFeatures(degree=10, include_bias=False)
X_poly = poly_features.fit_transform(X)

# Split into train/test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_poly, y, test_size=0.3, random_state=42
)

```

```python
# Build overly complex model
overfit_model = tf.keras.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(10,)),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(1)
])

overfit_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Train the overfitting model
overfit_history = overfit_model.fit(
    X_train, y_train,
    epochs=200,
    batch_size=8,
    validation_data=(X_test, y_test),
    verbose=0
)

print(f"Training set size: {len(X_train)}")
print(f"Model parameters: {overfit_model.count_params()}")
print(f"Parameters per data point: {overfit_model.count_params()/len(X_train)
    :.1f}")
```

Identifying Overfitting Symptoms

As training progresses, we observe clear signs of overfitting. The training loss decreases steadily throughout training, eventually reaching very low values. However, the validation loss follows a different pattern: it decreases initially but then starts increasing after a certain point. This divergence between training and validation performance is the hallmark of overfitting.

Listing 7: Detecting Overfitting

```python
# Extract training and validation losses
train_loss = overfit_history.history['loss']
val_loss = overfit_history.history['val_loss']

# Find the epoch where validation loss starts increasing
best_epoch = np.argmin(val_loss)
print(f"Best validation loss at epoch: {best_epoch}")
print(f"Training loss at best epoch: {train_loss[best_epoch]:.2f}")
print(f"Validation loss at best epoch: {val_loss[best_epoch]:.2f}")
print(f"Final training loss: {train_loss[-1]:.2f}")
print(f"Final validation loss: {val_loss[-1]:.2f}")

# Calculate overfitting indicator
overfitting_ratio = val_loss[-1] / train_loss[-1]
print(f"Overfitting ratio (val_loss/train_loss): {overfitting_ratio:.2f}")
```

Solution One: Regularization and Early Stopping

Our first approach combines multiple overfitting prevention techniques. We add L2 regularization to penalize large weights, implement dropout to prevent the model from relying too heavily on specific neurons, and use early stopping to halt training when validation performance stops improving. This approach keeps the complex model but constrains its learning.

Listing 8: Solution One - Regularization

```python
# Build regularized model
regularized_model = tf.keras.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(10,),
                            kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(50, activation='relu',
                            kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(1)
])

regularized_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=20, restore_best_weights=True
)

# Train with regularization
reg_history = regularized_model.fit(
    X_train, y_train,
    epochs=200,
    batch_size=8,
    validation_data=(X_test, y_test),
    callbacks=[early_stopping],
    verbose=0
)

print(f"Training stopped at epoch: {len(reg_history.history['loss'])}")
print(f"Final training loss: {reg_history.history['loss'][-1]:.2f}")
print(f"Final validation loss: {reg_history.history['val_loss'][-1]:.2f}")
```

Solution Two: Model Simplification

Our alternative approach focuses on reducing model complexity to match our limited data. We use fewer polynomial features and build a much simpler neural network. We also implement cross validation to get more reliable performance estimates from our small dataset.

Listing 9: Solution Two - Simplification

```
# Create simpler polynomial features (degree 3)
simple_poly = PolynomialFeatures(degree=3, include_bias=False)
X_simple = simple_poly.fit_transform(X)

X_train_simple, X_test_simple, y_train, y_test = train_test_split(
    X_simple, y, test_size=0.3, random_state=42
)

# Build simple model
simple_model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(3,)),
    tf.keras.layers.Dense(1)
])

simple_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Train simple model
simple_history = simple_model.fit(
    X_train_simple, y_train,
    epochs=100,
    batch_size=8,
    validation_data=(X_test_simple, y_test),
    verbose=0
)

```

```python
print(f"Simple model parameters: {simple_model.count_params()}")
print(f"Parameters per data point: {simple_model.count_params()/len(
    X_train_simple):.1f}")
print(f"Final training loss: {simple_history.history['loss'][-1]:.2f}")
print(f"Final validation loss: {simple_history.history['val_loss'][-1]:.2f}")
```

Comparing Solutions and Key Insights

Both solutions successfully address overfitting but through different approaches. Solution One keeps the complex model but constrains its learning through regularization techniques and early stopping. Solution Two simplifies the model architecture to better match the available data. The regularized model achieves similar training and validation performance, while the simple model shows even less overfitting. The key insight is that overfitting occurs when model complexity exceeds what the data can support, and successful solutions either constrain the complex model or reduce model complexity to match data availability.

Listing 10: Comparing Solutions

```python
# Evaluate all models
models = [
    ('Overfitting Model', overfit_model, X_test),
    ('Regularized Model', regularized_model, X_test),
    ('Simple Model', simple_model, X_test_simple)
]

print("Model Comparison:")
print("-" * 50)
for name, model, test_data in models:
    train_pred = model.predict(X_train if 'Simple' not in name else
        X_train_simple, verbose=0)
    test_pred = model.predict(test_data, verbose=0)

    train_mse = np.mean((y_train - train_pred.flatten())**2)
    test_mse = np.mean((y_test - test_pred.flatten())**2)

    print(f"{name}:")
    print(f"  Training MSE: {train_mse:.2f}")
    print(f"  Test MSE: {test_mse:.2f}")
    print(f"  Overfitting ratio: {test_mse/train_mse:.2f}")
    print()
```

9 Hyperparameter Tuning

Hyperparameters are the settings we choose before training our model. Think of them as the dials and knobs on a machine that we need to adjust to get the best performance. Unlike regular parameters (like weights), hyperparameters are not learned by the algorithm but set by us.

9.1 Common Hyperparameters

9.1.1 Learning Rate

Controls how big steps the optimizer takes when updating weights. Too high and the model might overshoot the optimal solution. Too low and training will be very slow.

9.1.2 Batch Size

Number of samples processed before updating the model weights. Smaller batches give noisier but more frequent updates. Larger batches give smoother but less frequent updates.

9.1.3 Number of Epochs

How many times the model sees the entire training dataset. Too few and the model might not learn enough. Too many and it might overfit.

9.1.4 Model Architecture

Number of layers, neurons per layer, activation functions, etc.

9.2 Hyperparameter Tuning Strategies

Lookout for yourself!

10 Support Vector Machine (SVM)

10.1 Problem Setup

Support Vector Machine (SVM) is a supervised learning algorithm primarily used for binary classification. The objective is to find a hyperplane that best separates data points of different classes. The key idea is not just to separate the classes but to do so with the maximum possible margin , i.e., to keep the boundary as far away as possible from the nearest points of both classes.

Let the training dataset be:

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^n, \quad \text{where } \mathbf{x}_i \in \mathbb{R}^d, \quad y_i \in \{-1, +1\}$$

10.2 Approach

We aim to find a hyperplane:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

that maximally separates the two classes. The vector $\mathbf{w}$ is orthogonal to the hyperplane, and b determines its offset from the origin.

The distance from any point $\mathbf{x}$ to the hyperplane is:

$$\text{Distance} = \frac{|\mathbf{w}^T\mathbf{x} + b|}{\|\mathbf{w}\|}$$

To ensure good generalization, we want to:

1. Correctly classify all points: $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$
2. Maximize the margin, which is $\frac{2}{\|\mathbf{w}\|}$

The data points that lie closest to the hyperplane (i.e., exactly on the margin boundaries) are called **support vectors**. These are the most "influential" points because the hyperplane is entirely defined by them. All other points have no direct impact on the decision boundary.

10.3 Mathematical Formulation

The optimization problem is:

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i$$

For non-linearly separable data, we allow some violations using slack variables $\xi_i \geq 0$:

$$\min_{\mathbf{w},b,\xi} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

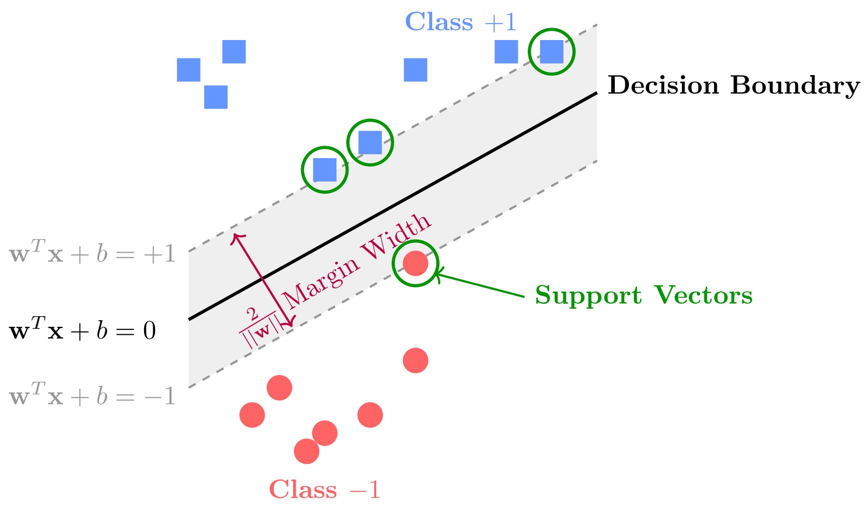
$$\text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

Here, C is a regularization parameter balancing margin size and classification errors.

10.4 Essential Configurations

- **Kernel Function**: For non-linear separation, use kernels like RBF (Gaussian), polynomial, or sigmoid to project data into a higher-dimensional space.
- **C Parameter**: Controls the trade-off between margin width and classification error.
- **Gamma**: In RBF kernels, it defines how far the influence of a single training example reaches.

Support Vector Machine (SVM)


Class $+1$
Decision Boundary
$\mathbf{w}^T\mathbf{x} + b = +1$
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$
Margin Width
$\frac{2}{\|\mathbf{w}\|}$
Support Vectors
Class -1

11 Regularized Regression

11.1 Problem Setup

Regularized regression addresses the problem of overfitting in standard linear regression. When we have many features or limited data, ordinary least squares can create models that fit the training data perfectly but perform poorly on new data. Regularization adds a penalty term to prevent the model from becoming too complex.

Given a dataset with n training examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ are the input features and $y_i \in \mathbb{R}$ are the continuous target values, we want to find parameters $\mathbf{w}$ that predict y accurately while keeping the model simple.

11.2 Approach

The regularized regression approach modifies the standard least squares objective by adding a penalty term that controls model complexity. There are two main types of regularization:

1. **Ridge Regression (L2 regularization)**: Penalizes the sum of squared parameters 2. **Lasso Regression (L1 regularization)**: Penalizes the sum of absolute values of parameters

The linear model predicts:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b$$

where $\mathbf{w}$ are the weights and b is the bias term.

11.3 Mathematical Formulation

11.3.1 Ordinary Least Squares (OLS):

$$\min_{\mathbf{w}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

11.3.2 Ridge Regression (L2 Regularization):

$$\min_{\mathbf{w}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda ||\mathbf{w}||_2^2$$

where $||\mathbf{w}||_2^2 = \sum_{j=1}^{d} w_j^2$ and $\lambda \geq 0$ is the regularization parameter.
The closed-form solution for Ridge regression is:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$$

where $\mathbf{X}$ is the design matrix and $\mathbf{I}$ is the identity matrix.

11.3.3 Lasso Regression (L1 Regularization):

$$\min_{\mathbf{w}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda ||\mathbf{w}||_1$$

where $||\mathbf{w}||_1 = \sum_{j=1}^{d} |w_j|$.

11.3.4 Elastic Net (Combined L1 + L2):

$$\min_{\mathbf{w}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda_1 ||\mathbf{w}||_1 + \lambda_2 ||\mathbf{w}||_2^2$$

Effect of Regularization:

- **Ridge**: Shrinks coefficients towards zero but keeps all features
- **Lasso**: Can set coefficients exactly to zero, performing feature selection
- **Elastic Net**: Combines both effects

11.4 Essential Configurations

- **Regularization Parameter (λ)**: Controls strength of regularization, typically chosen by cross-validation
- **Feature Scaling**: Important to standardize features before applying regularization
- **Regularization Type**: Choose L1, L2, or Elastic Net based on problem needs
- **Cross-Validation**: Use to select optimal λ value

12 Simple Neural Network Implementation Example

Let's walk through a complete binary classification example using the classic Iris flower dataset. We will build a system that can automatically classify iris flowers as either Setosa or Not Setosa based on their petal and sepal measurements.

Problem Setup

Our flower classification system needs to examine iris flowers and output either "Setosa" (1) or "Not Setosa" (0). The Iris dataset contains measurements of 150 iris flowers with four features: sepal length, sepal width, petal length, and petal width. Originally this is a three class problem (Setosa, Versicolor, Virginica), but we simplify it to binary classification by grouping Versicolor and Virginica together as "Not Setosa". This makes it perfect for understanding binary classification fundamentals.

Data Loading and Exploration

We start by examining our dataset structure. Each flower is represented as a vector of 4 measurements plus one label. For example, Flower 1 might have measurements [5.1, 3.5, 1.4, 0.2] with label 1 (Setosa), while Flower 2 might be [6.2, 2.8, 4.8, 1.8] with label 0 (Not Setosa). We check the distribution of classes and examine feature ranges to understand our data better.

Listing 11: Data Loading and Exploration

```python
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features: sepal length, sepal width, petal length, petal width
y_original = iris.target  # Original classes: 0=Setosa, 1=Versicolor, 2=
    Virginica

# Convert to binary classification: Setosa (1) vs Not Setosa (0)
y = (y_original == 0).astype(int)

print(f"Dataset shape: {X.shape}")
print(f"Feature names: {iris.feature_names}")
print(f"Setosa flowers: {np.sum(y)} out of {len(y)} ({100*np.mean(y):.1f}%)")
print(f"Sample flower measurements: {X[0]}")
print(f"Sample label: {y[0]}")
```

Data Preprocessing

Before feeding data to our neural network, we perform essential preprocessing steps. First, we normalize our features so they all have similar scales, preventing features with larger values from dominating the learning process. Next, we split our 150 flowers into 105 for training and 45 for testing. The training flowers will teach our network to recognize Setosa patterns, while testing flowers will evaluate how well it learned.

Listing 12: Data Preprocessing

```python
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Normalize features to have mean 0 and standard deviation 1
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print(f"Training set size: {X_train_scaled.shape[0]}")
print(f"Testing set size: {X_test_scaled.shape[0]}")
print(f"Feature ranges after scaling (min, max):")
print(f"  Training: ({X_train_scaled.min():.2f}, {X_train_scaled.max():.2f})")
```

Neural Network Architecture Design

Our binary classification network has a simple but effective structure. The input layer accepts our 4 flower measurements. We add one hidden layer with 8 neurons, each using the ReLU activation function to capture non linear relationships between petal and sepal measurements. The output layer contains just one neuron with a sigmoid activation function, which converts any input into a probability between 0 and 1. Values closer to 1 indicate Setosa, while values closer to 0 suggest Not Setosa.

Listing 13: Neural Network Architecture

```python
# Build the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=(4,)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Configure the model for training
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Display model architecture
model.summary()
```

Training Configuration

We configure our network for optimal learning. The loss function is binary cross entropy, which measures how far our predicted probabilities are from the true labels. We use the Adam optimizer, which automatically adjusts learning speed during training. Our learning rate is set automatically by Adam. We plan to train for 100 epochs, meaning the network will see the entire training dataset 100 times.

Training Process

During training, our network gradually learns to distinguish Setosa from other iris varieties. In early epochs, predictions are essentially random, with accuracy around 50 percent. As training progresses, the network identifies patterns like "flowers with very small petal length and width tend to be Setosa." The network learns that Setosa flowers have distinctly different measurements compared to Versicolor and Virginica.

Listing 14: Model Training

```python
# Train the model
history = model.fit(
    X_train_scaled, y_train,
    epochs=100,
    batch_size=16,
    validation_split=0.2,
    verbose=1
)

# Display training progress
print(f"Final training accuracy: {history.history['accuracy'][-1]:.3f}")
print(f"Final validation accuracy: {history.history['val_accuracy'][-1]:.3f}")
```

Evaluation and Results

After training completes, we test our network on the 45 unseen flowers. The network typically achieves very high accuracy on this test set, often 100 percent, because Setosa flowers are quite distinct from the other varieties. We examine both the probability predictions and the final classifications to understand how confident our model is in its decisions.

Listing 15: Model Evaluation

```python
# Evaluate on test set
test_loss, test_accuracy = model.evaluate(X_test_scaled, y_test, verbose=0)
print(f"Test accuracy: {test_accuracy:.3f}")

# Make predictions
predictions = model.predict(X_test_scaled)
predicted_classes = (predictions > 0.5).astype(int)

# Show some example predictions
print("\nSample predictions:")
for i in range(5):
    print(f"Flower {i+1}: Probability={predictions[i][0]:.3f}, "
          f"Predicted={'Setosa' if predicted_classes[i][0] else 'Not Setosa'}, "
          f"Actual={'Setosa' if y_test[i] else 'Not Setosa'}")
```

Practical Deployment Insights

This end to end example demonstrates how neural networks solve real world classification problems. The key insight is that networks learn by finding mathematical boundaries in feature space that separate different classes. Our iris classifier learned that certain combinations of petal and sepal measurements reliably indicate Setosa flowers. This same approach works for medical diagnosis, image recognition, fraud detection, and countless other classification tasks.

Recommended Resources for Some More Key Machine Learning Concepts (optional)

We recommend exploring the following resources to build strong foundational understanding:

- **Support Vector Machines (SVM):**
  `https://www.youtube.com/watch?v=efR1C6CvhmE&t=743s`
- **Ridge Regression (Regularized Linear Regression):**
  `https://www.youtube.com/watch?v=Q81RR3yKn30&t=85s`
- **Logistic Regression:**
  `https://www.youtube.com/watch?v=yIYKR4sgzI8&t=189s`
- **K-means Clustering:**
  `https://www.youtube.com/watch?v=4b5d3muPQmA`
- **Principal Component Analysis (PCA):**
  `https://www.youtube.com/watch?v=FgakZw6K1QQ`
- **Naive Bayes Classifier:**
  `https://www.youtube.com/watch?v=O2L2Uv9pdDA`
- **Decision Trees and Random Forests:**
  `https://www.youtube.com/watch?v=7VeUPuFGJHk`
- **Gradient Descent Optimization:**
  `https://www.youtube.com/watch?v=sDv4f4s2SB8`
- **Neural Networks and Deep Learning (Enthusiasts):**
  3Blue1Brown's Intuitive Deep Learning Series:
  `https://www.youtube.com/watch?v=aircAruvnKk&list=PLcCe-ymWq77ow42k4-ZrLzlM3F7Ha7smT`

Note: These resources cover both theoretical and intuitive perspectives, and are well-suited for beginners as well as intermediate learners aiming to strengthen their ML concepts. We recommend you to try attempting `Assignment-02` alongside this documentation, it will give you some better practical idea. If you are a person who is not so interested in mathematics of ML, you can skip them, but it is good to have better idea on what exactly happening, so we recommend you to go through above resources which do a much more mathematically involved discussion on this topics, they are very standard and most tested in industry for interviews if you are looking at this project from that point of view.

Thank You