Systematic Trading Strategies with Machine Learning Algorithms

Supervised Learning Algorithms

May 15, 2025

Outline

Imperial College Business School

Outline

Outline

Decision Tree Algorithm for Classification

Imperial College Business School

Algorithm Decision Tree Learning Algorithm

Require: Training data $\{(F_i, y_i)\}_{i=1}^{n}$, stopping criteria

Ensure: Decision tree T

Initialize tree with single root node containing all data
while nodes can be split and stopping criteria not met **do**
    for each leaf node with region $\mathcal{R}$ **do**
        Find (j^*, τ^*) that maximizes:
        $IG(j, \tau) = I(\mathcal{R}) - \frac{|\mathcal{R}_L|}{|\mathcal{R}|} I(\mathcal{R}_L) - \frac{|\mathcal{R}_R|}{|\mathcal{R}|} I(\mathcal{R}_R)$
        Where $\mathcal{R}_L = \{F \in \mathcal{R} : F_j \leq \tau\}$ and $\mathcal{R}_R = \{F \in \mathcal{R} : F_j > \tau\}$
        Split node using rule $F_{j^*} > \tau^*$
    end for
end while
Assign prediction to each leaf node (majority class)
return T

Information Gain for Classification

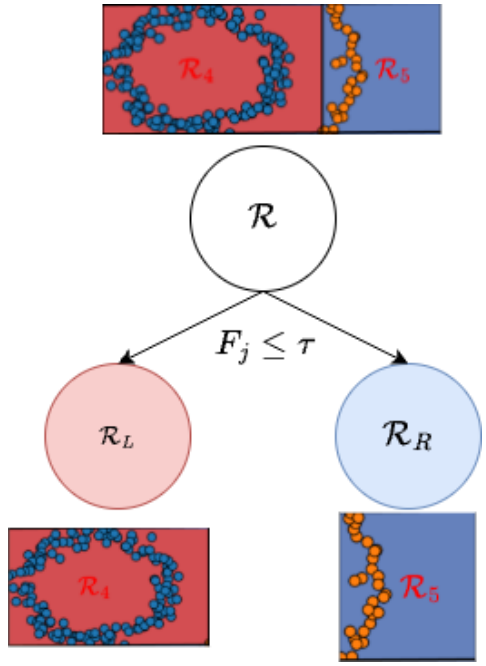
Imperial College Business School

- **Information Gain** for feature j and threshold τ:

$$IG(j,\tau) = I(\mathcal{R}) - \frac{|\mathcal{R}_L|}{|\mathcal{R}|} I(\mathcal{R}_L) - \frac{|\mathcal{R}_R|}{|\mathcal{R}|} I(\mathcal{R}_R)$$

Where:

  - $\mathcal{R}_L = \{F \in \mathcal{R} : F_j \leq \tau\}$
  - $\mathcal{R}_R = \{F \in \mathcal{R} : F_j > \tau\}$

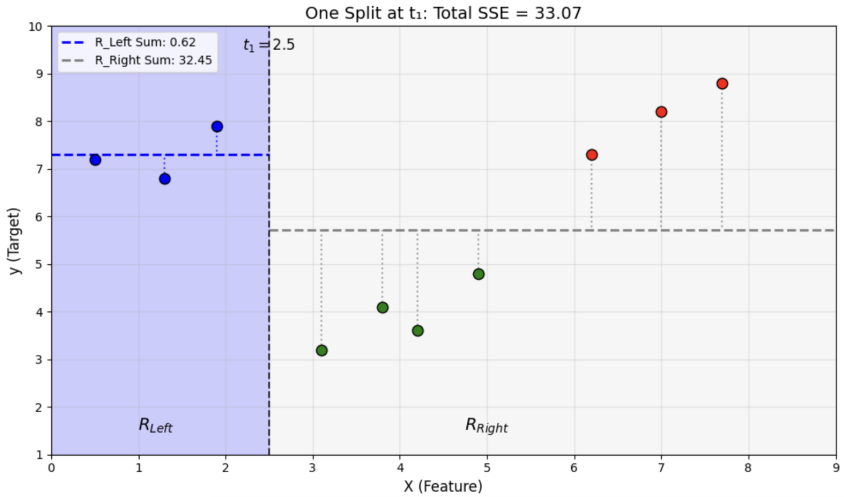


- The DT algorithm:
  1. For each feature j and possible threshold τ, compute $IG(j,\tau)$
  2. Select feature j^* and threshold τ^* that maximize IG
  3. Split node and create child regions $\mathcal{R}_L$ and $\mathcal{R}_R$
  4. Recursively apply to each child node until stopping criteria met

Finding the Optimal Split in Regression Trees

Imperial College Business School

- For a split at value t_1 on feature X:
  - Left region: $R_{Left} = \{x_i | x_i \leq t_1\}$
  - Right region: $R_{Right} = \{x_i | x_i > t_1\}$



- For each region, we compute:
  - Prediction value: average of y_i in the region
  - SSE: sum of squared errors in the region

- **Optimization objective**: Choose feature j and threshold t that minimizes:

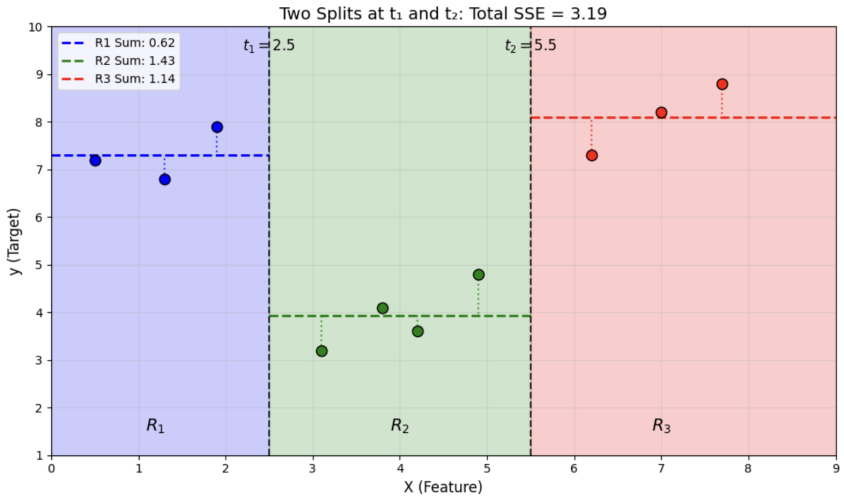
$$SSE_{total} = SSE_{Left} + SSE_{Right}$$

Growing a Regression Tree

- The prediction function for M regions $(R_m)_{1 \leq m \leq M}$ is:

$$f(x) = \sum_{m=1}^{M} c_m \mathbb{1}\{x \in R_m\}$$

- Where c_m is the average of all y_i for which $x_i \in R_m$



- After recursive splitting, we end up with multiple regions (leaves).
- Increasing the number of regions leads to lower training error
- To avoid overfitting, we need **stopping criteria**: Maximum depth, minimum samples per leaf, minimum error improvement.

Algorithm Regression Tree Learning Algorithm

Require: Training data $\{(F_i, y_i)\}_{i=1}^{n}$, stopping criteria

Ensure: Regression tree T

Initialize tree with single root node containing all data
while nodes can be split and stopping criteria not met **do**
    for each leaf node with region $\mathcal{R}$ **do**
        Find (j^*, τ^*) that minimizes:
        $SSE(j, \tau) = \sum_{i:F_i \in \mathcal{R}_L} (y_i - \bar{y}_{\mathcal{R}_L})^2 + \sum_{i:F_i \in \mathcal{R}_R} (y_i - \bar{y}_{\mathcal{R}_R})^2$
        Where $\mathcal{R}_L = \{F \in \mathcal{R} : F_j \leq \tau\}$ and $\mathcal{R}_R = \{F \in \mathcal{R} : F_j > \tau\}$
        Split node using rule $F_{j^*} > \tau^*$
    end for
end while
Assign prediction $\bar{y}_{\mathcal{R}_m}$ to each leaf node (average of y_i in the region)
return T

Outline

Imperial College Business School

Ensemble Learning: Bagging

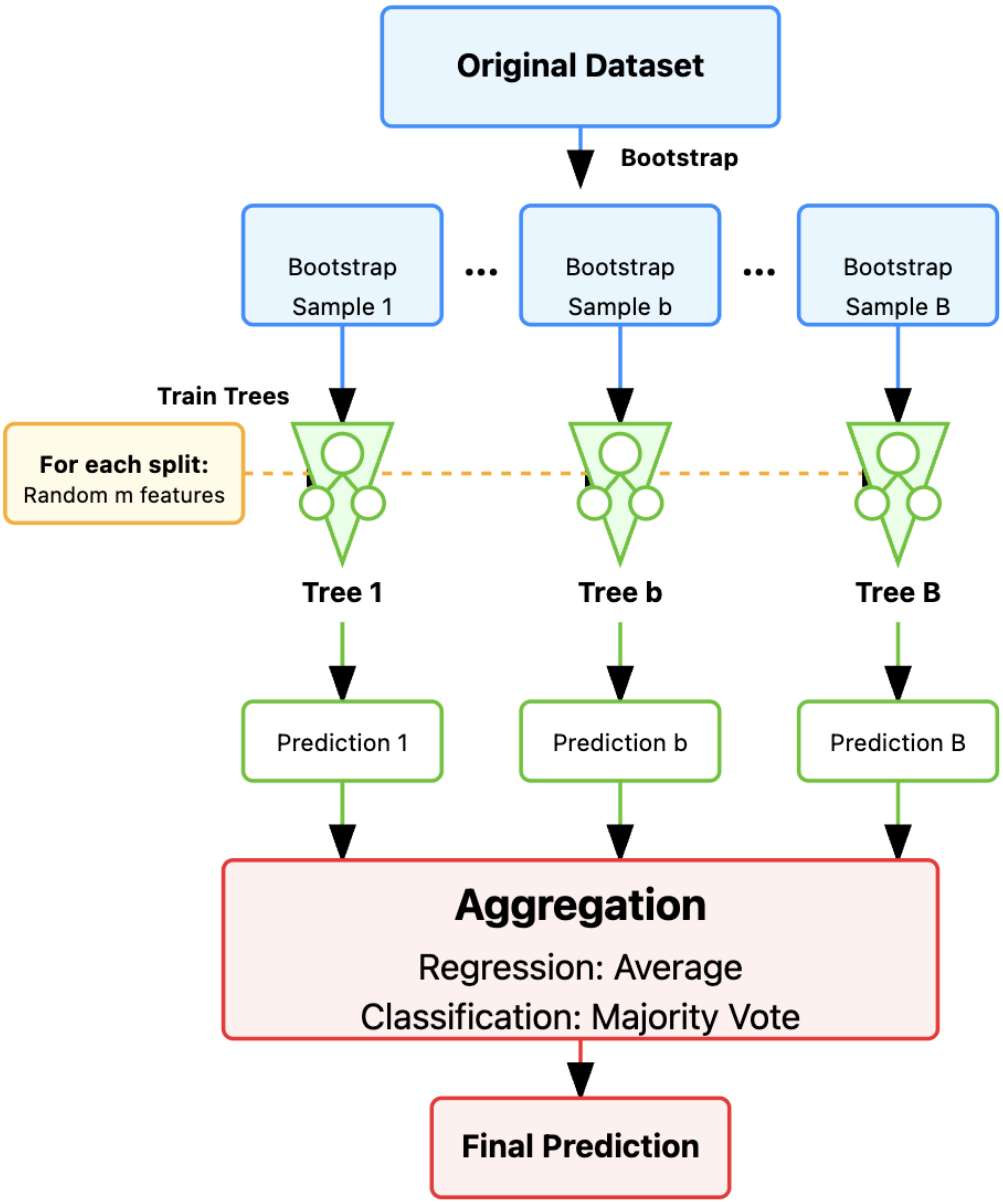
Imperial College Business School

- **Motivation of Ensemble Models**: Aggregate weak learners to build a strong learner
- **Bagging Methodology** (Bagging: Bootstrap Aggregation):
  1. **Generate bootstrap samples** $\mathcal{B}_1, \ldots, \mathcal{B}_B$:
     - Create $\mathcal{B}_b$ by picking points from $\{x_1, \ldots, x_n\}$ randomly n times
     - A particular x_i can appear in $\mathcal{B}_b$ many times.
  2. **Train a model per bootstrap**:
     - Each bootstrap sample trains an independent model
  3. **Aggregate the predictions**:
     - **Classification**: Majority vote across all models
     - **Regression**: Average of individual model predictions

Random Forest: Intuition and Overview

Imperial College Business School

- **Random Forest** is an ensemble method that improves upon bagging decision trees by introducing additional randomness at each split for each decision tree:
  - During training, at each node, only consider a subset of features
  - Result: Less correlated trees



Random Forest Training Algorithm

Imperial College Business School

Algorithm Random Forest Algorithm

Require: Training data $\{(F_i, y_i)\}_{i=1}^{n}$, number of trees B, features per split $m < p$

Ensure: Random Forest model RF

for $b = 1$ to B **do**
  Draw a bootstrap sample $\mathcal{B}_b$ of size n from the training data
  Initialize tree T_b with root node containing data from $\mathcal{B}_b$
  while nodes in T_b can be split **do**
    for each leaf node with region $\mathcal{R}$ **do**
      Randomly select m features from the available p
      Find best split (j^*, τ^*) among these m features
      Split node using rule $F_{j^*} > \tau^*$
    end for
  end while
end for
return $RF = \{T_1, T_2, \ldots, T_B\}$

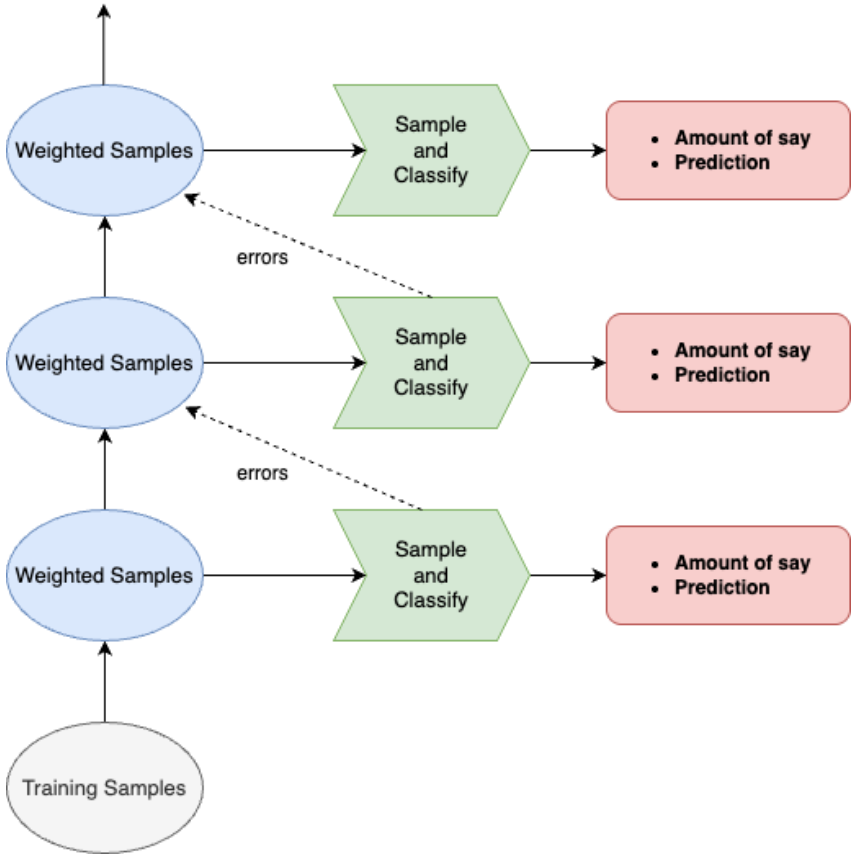
Outline

Imperial College Business School

Imperial College Business School

Adaptive Sample Weighting

- Unlike bagging, AdaBoost [2] does not use uniform sampling
- Samples are weighted based on classification difficulty
- Misclassified examples receive higher weights in subsequent iterations
- The algorithm creates models sequentially, each one focusing on correcting previous errors
- This adaptive weighting is the core mechanism behind boosting's effectiveness

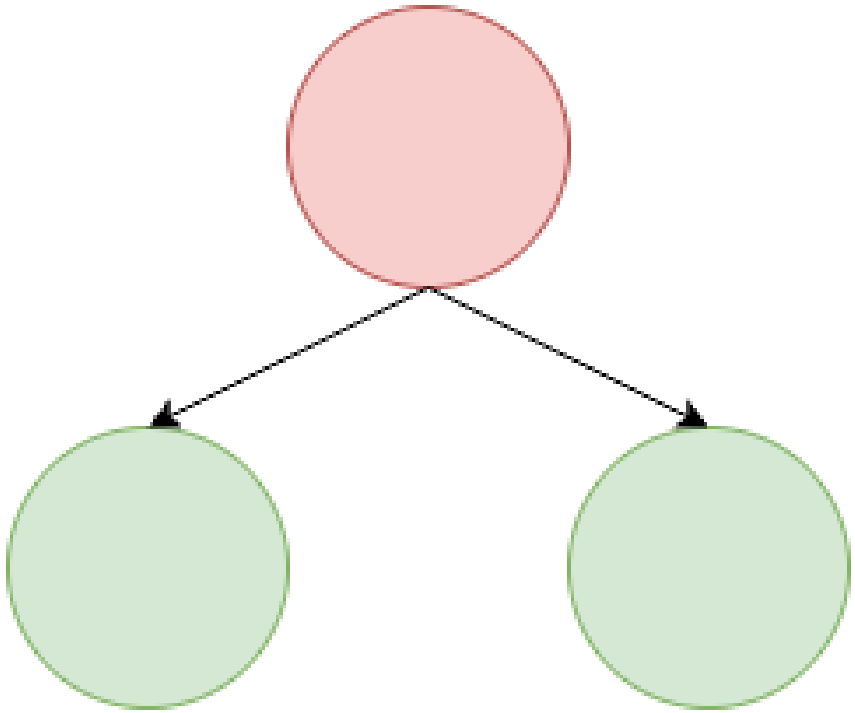


AdaBoost: Key Idea 2

Imperial College Business School

Leveraging Weak Learners (Decision Stumps)

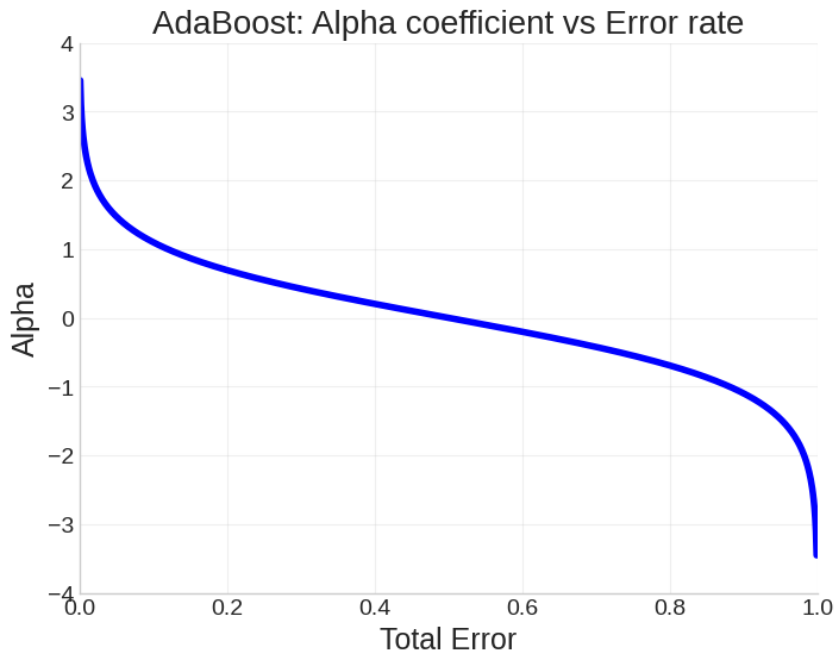
- AdaBoost builds its power from simple decision trees (typically depth=1) called **stumps**
- Individually, each model performs only slightly better than random guessing
- However, they are computationally efficient and resistant to overfitting
- The algorithm's strength comes from combining many weak models into a strong ensemble



Imperial College Business School

Performance-Based Model Weighting

- Unlike random forests where all trees contribute equally, AdaBoost assigns varying importance to each model
- The algorithm calculates an "amount of say" (α_t) for each classifier based on its accuracy
- Highly accurate classifiers receive strong positive weights
- Random-level performers (error rate $= 0.5$) receive zero weight
- Poor performers can contribute negatively by having their predictions reversed



AdaBoost: Key Algorithm Components

Imperial College Business School

1. How do we determine each model's contribution?
   - Model influence (α_t) is based on weighted error $\epsilon_t = \sum_{i=1}^{n} w_t(i)\mathbb{1}\{y_i \neq f_t(x_i)\}$
   - $\alpha_t = \frac{1}{2}\ln(\frac{1-\epsilon_t}{\epsilon_t})$ where ϵ_t is the proportion of weighted misclassifications

2. How do we adaptively weight training examples?
   - Weights are updated after each iteration: $w_{t+1}(i) \propto w_t(i) \cdot e^{\alpha_t(1-2\mathbb{1}\{y_i=f_t(x_i)\})}$
   - For binary classification where $y_i \in \{0,1\}$ and $f_t(x_i) \in \{0,1\}$
   - This means weights increase for misclassified examples and decrease for correctly classified ones
   - Weights are normalized to form a probability distribution

AdaBoost: Ensemble Prediction

Imperial College Business School

How do we make predictions with the ensemble?

- Final prediction uses a weighted majority vote:

$$f_{boost}(x) = \mathbb{1}\left\{\sum_{t=1}^{T} \alpha_t f_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t\right\}$$

- Each weak learner's vote ($f_t(x) \in \{0, 1\}$) is weighted by its performance (α_t)
- Models with lower error rates have larger influence on the final prediction
- The threshold is half the sum of all model weights
- When the weighted sum favors class 1, we predict 1; otherwise, we predict 0

Algorithm AdaBoost Algorithm

Require: Training data $\{(x_i, y_i)\}_{i=1}^n$, $x \in \mathcal{X}$, $y \in \{0, 1\}$, number of iterations T

Ensure: AdaBoost model

1: Initialize weights $w_1(i) = \frac{1}{n}$ for $i = 1 : n$
2: **for** $t = 1$ to T **do**
3: Train classifier f_t on weighted training data with weights w_t
4: Calculate weighted error: $\epsilon_t = \sum_{i=1}^n w_t(i) \mathbb{1}\{y_i \neq f_t(x_i)\}$
5: Calculate model weight: $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
6: Scale weights: $\hat{w}_{t+1}(i) = w_t(i) \cdot e^{\alpha_t (1 - 2\mathbb{1}\{y_i = f_t(x_i)\})}$
7: Normalize: $w_{t+1}(i) = \frac{\hat{w}_{t+1}(i)}{\sum_j \hat{w}_{t+1}(j)}$
8: **end for**
9: **return** Classification rule: $f_{boost}(x_0) = \mathbb{1}\{\sum_{t=1}^T \alpha_t f_t(x_0) \geq \frac{1}{2} \sum_{t=1}^T \alpha_t\}$

AdaBoost Example: First Decision Tree

Key calculations for Tree 1:

- Initial weights:

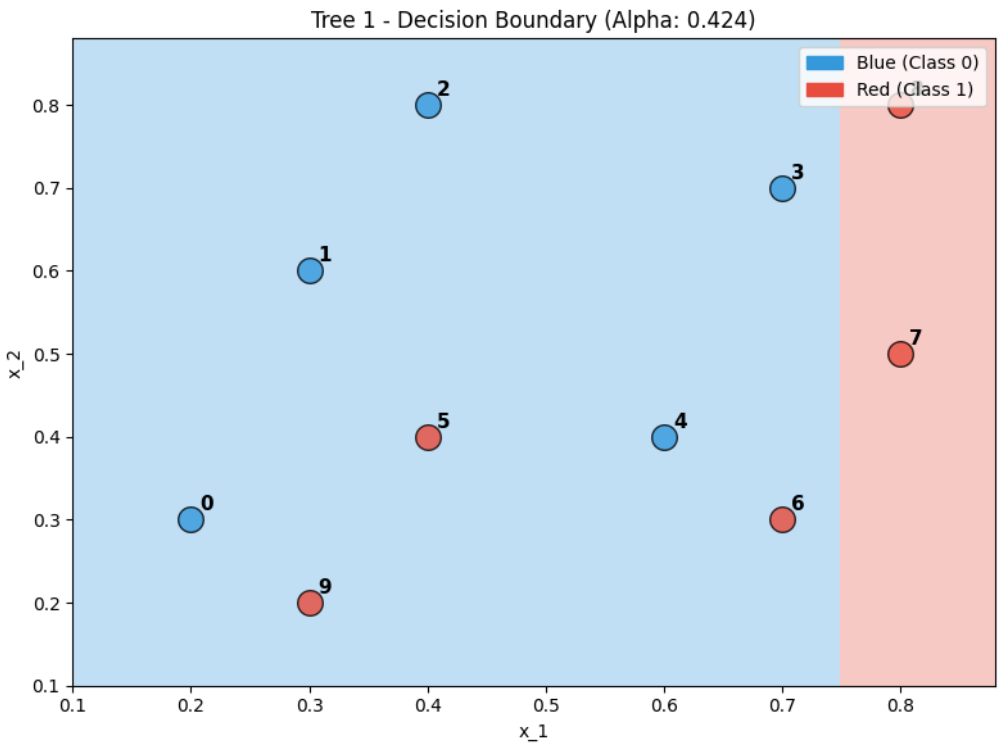
  $w_1(i) = 0.1$ for all samples

- Weighted Error:

$$\epsilon_1 = 0.300$$

- Model Contribution:

$$\begin{aligned}\alpha_1 &= \frac{1}{2}\ln\left(\frac{1-\epsilon_1}{\epsilon_1}\right) \\ &= \frac{1}{2}\ln\left(\frac{0.7}{0.3}\right) \\ &= 0.424\end{aligned}$$



Imperial College Business School

Weight Update Process:

- For misclassified:

$$w_2(i) \propto w_1(i) \cdot e^{+\alpha_1}$$
$$= 0.1 \cdot e^{0.424}$$

- For correctly classified:

$$w_2(i) \propto w_1(i) \cdot e^{-\alpha_1}$$
$$= 0.1 \cdot e^{-0.424}$$

- Weights are then normalized.

Sample	x_1	x_2	y	w_1	$f_1(x)$	w_2
0	0.2	0.3	0	0.1	0	0.071
1	0.3	0.6	0	0.1	0	0.071
2	0.4	0.8	0	0.1	0	0.071
3	0.7	0.7	0	0.1	0	0.071
4	0.6	0.4	0	0.1	0	0.071
5	**0.4**	**0.4**	**1**	**0.1**	**0**	**0.167**
6	**0.7**	**0.3**	**1**	**0.1**	**0**	**0.167**
7	0.8	0.5	1	0.1	1	0.071
8	0.8	0.8	1	0.1	1	0.071
9	**0.3**	**0.2**	**1**	**0.1**	**0**	**0.167**

Imperial College Business School

Key calculations for Tree 2:

- Initial weights:

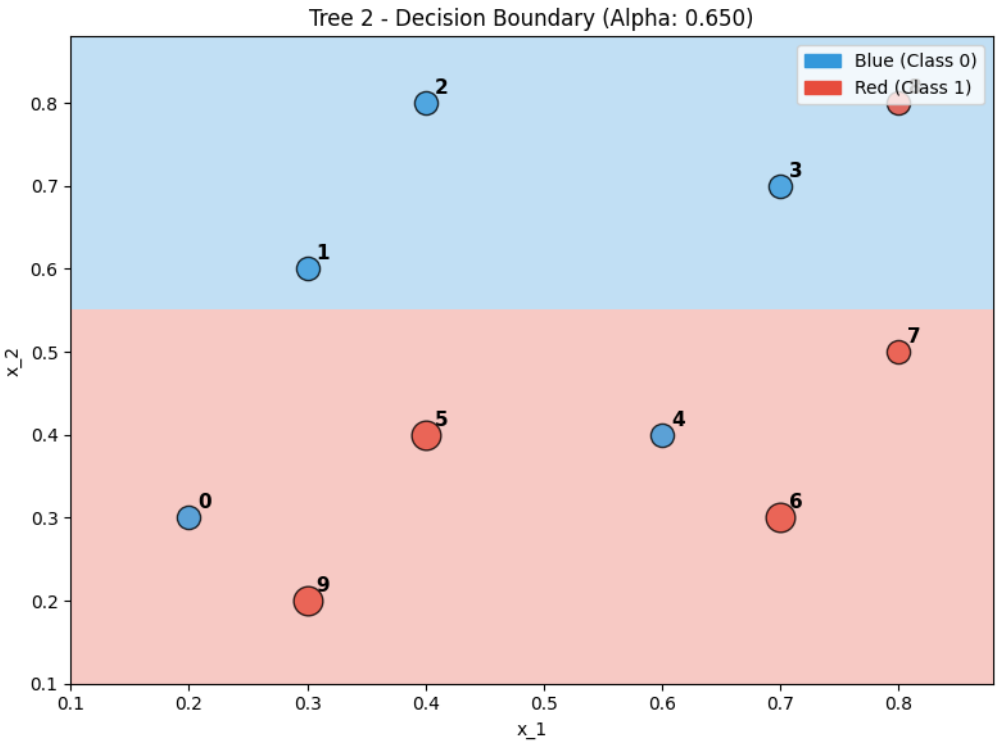
  $w_2(i)$ from previous iteration

- Weighted Error:

$$\epsilon_2 = 0.214$$

- Model Contribution:

$$\begin{aligned}\alpha_2 &= \frac{1}{2}\ln\left(\frac{1-\epsilon_2}{\epsilon_2}\right) \\ &= \frac{1}{2}\ln\left(\frac{0.786}{0.214}\right) \\ &= 0.650\end{aligned}$$



Imperial College Business School

Weight Update Process:

- For misclassified:

$$w_3(i) \propto w_2(i) \cdot e^{+\alpha_2}$$
$$= w_2(i) \cdot e^{0.650}$$

- For correctly classified:

$$w_3(i) \propto w_2(i) \cdot e^{-\alpha_2}$$
$$= w_2(i) \cdot e^{-0.650}$$

- Weights are then normalized.

Sample	x_1	x_2	y	w_2	$f_2(x)$	w_3
0	**0.2**	**0.3**	**0**	**0.071**	**1**	**0.166**
1	0.3	0.6	0	0.071	0	0.045
2	0.4	0.8	0	0.071	0	0.045
3	0.7	0.7	0	0.071	0	0.045
4	**0.6**	**0.4**	**0**	**0.071**	**1**	**0.166**
5	0.4	0.4	1	0.166	1	0.106
6	0.7	0.3	1	0.166	1	0.106
7	0.8	0.5	1	0.071	1	0.045
8	**0.8**	**0.8**	**1**	**0.071**	**0**	**0.166**
9	0.3	0.2	1	0.166	1	0.106

AdaBoost Example: Third Decision Tree

Key calculations for Tree 3:

- Initial weights:

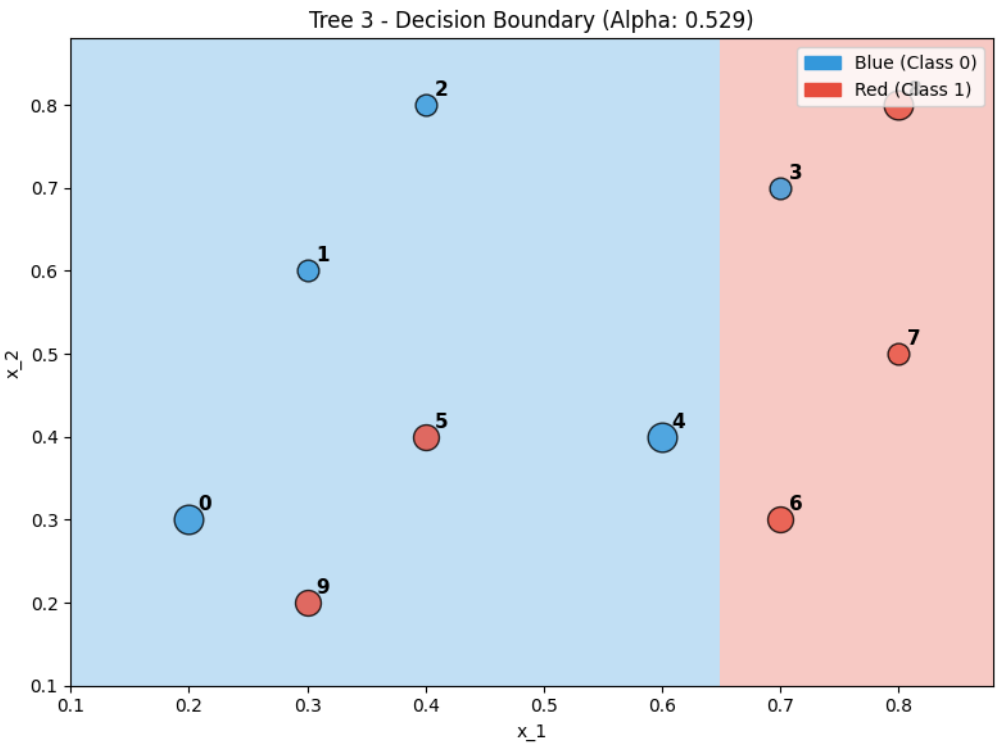
  $w_3(i)$ from previous iteration

- Weighted Error:

$$\epsilon_3 = 0.258$$

- Model Contribution:

$$\begin{aligned}\alpha_3 &= \frac{1}{2}\ln\left(\frac{1-\epsilon_3}{\epsilon_3}\right) \\ &= \frac{1}{2}\ln\left(\frac{0.742}{0.258}\right) \\ &= 0.529\end{aligned}$$



Imperial College Business School

Weight Update Process:

- For misclassified:

$$w_4(i) \propto w_3(i) \cdot e^{+\alpha_3}$$
$$= w_3(i) \cdot e^{0.529}$$

- For correctly classified:

$$w_4(i) \propto w_3(i) \cdot e^{-\alpha_3}$$
$$= w_3(i) \cdot e^{-0.529}$$

- Weights are then normalized.

Sample	x_1	x_2	y	w_3	$f_3(x)$	w_4
0	0.2	0.3	0	0.166	0	0.112
1	0.3	0.6	0	0.045	0	0.031
2	0.4	0.8	0	0.045	0	0.031
3	**0.7**	**0.7**	**0**	**0.045**	**1**	**0.088**
4	0.6	0.4	0	0.166	0	0.112
5	**0.4**	**0.4**	**1**	**0.106**	**0**	**0.206**
6	0.7	0.3	1	0.106	1	0.071
7	0.8	0.5	1	0.045	1	0.031
8	0.8	0.8	1	0.166	1	0.112
9	**0.3**	**0.2**	**1**	**0.106**	**0**	**0.206**

Imperial College Business School

Key calculations for Tree 4:

- Initial weights:

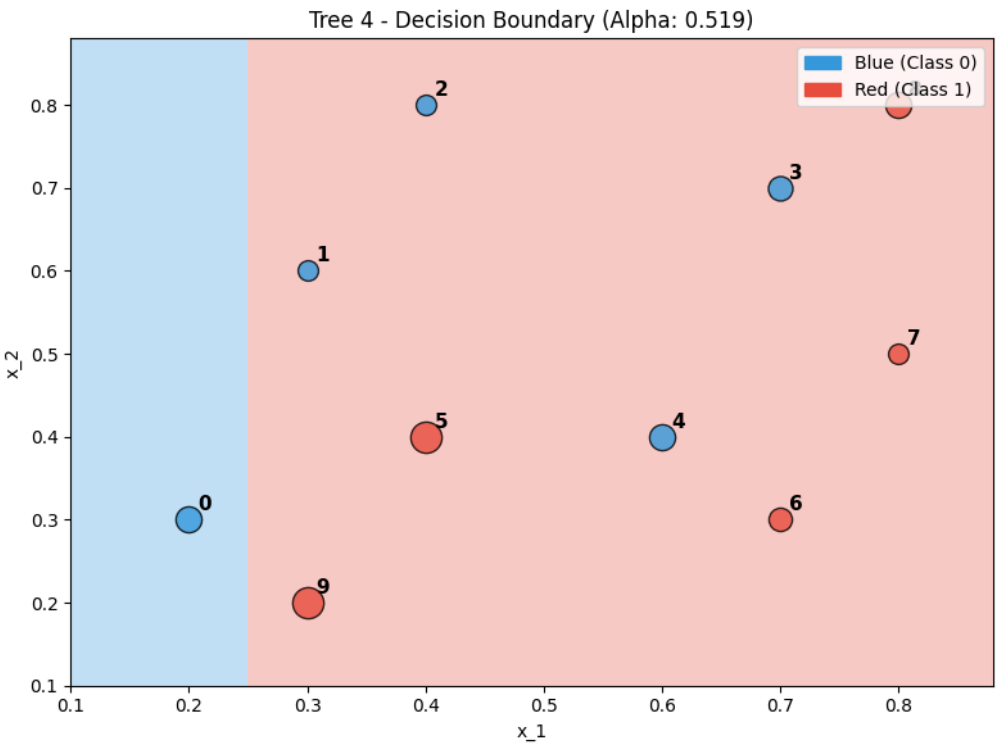
  $w_4(i)$ from previous iteration

- Weighted Error:

$$\epsilon_4 = 0.262$$

- Model Contribution:

$$\begin{aligned}\alpha_4 &= \frac{1}{2}\ln\left(\frac{1-\epsilon_4}{\epsilon_4}\right) \\ &= \frac{1}{2}\ln\left(\frac{0.738}{0.262}\right) \\ &= 0.519\end{aligned}$$



Weight Update Process:

- For misclassified:

$$w_5(i) \propto w_4(i) \cdot e^{+\alpha_4}$$
$$= w_4(i) \cdot e^{0.519}$$

- For correctly classified:

$$w_5(i) \propto w_4(i) \cdot e^{-\alpha_4}$$
$$= w_4(i) \cdot e^{-0.519}$$

- Weights are then normalized.

Sample	x_1	x_2	y	w_4	$f_4(x)$	w_5
0	0.2	0.3	0	0.112	0	0.076
1	**0.3**	**0.6**	**0**	**0.031**	**1**	**0.058**
2	**0.4**	**0.8**	**0**	**0.031**	**1**	**0.058**
3	**0.7**	**0.7**	**0**	**0.088**	**1**	**0.169**
4	**0.6**	**0.4**	**0**	**0.112**	**1**	**0.214**
5	0.4	0.4	1	0.206	1	0.139
6	0.7	0.3	1	0.071	1	0.048
7	0.8	0.5	1	0.031	1	0.021
8	0.8	0.8	1	0.112	1	0.076
9	0.3	0.2	1	0.206	1	0.139

AdaBoost: Final Ensemble Predictions

Imperial College Business School

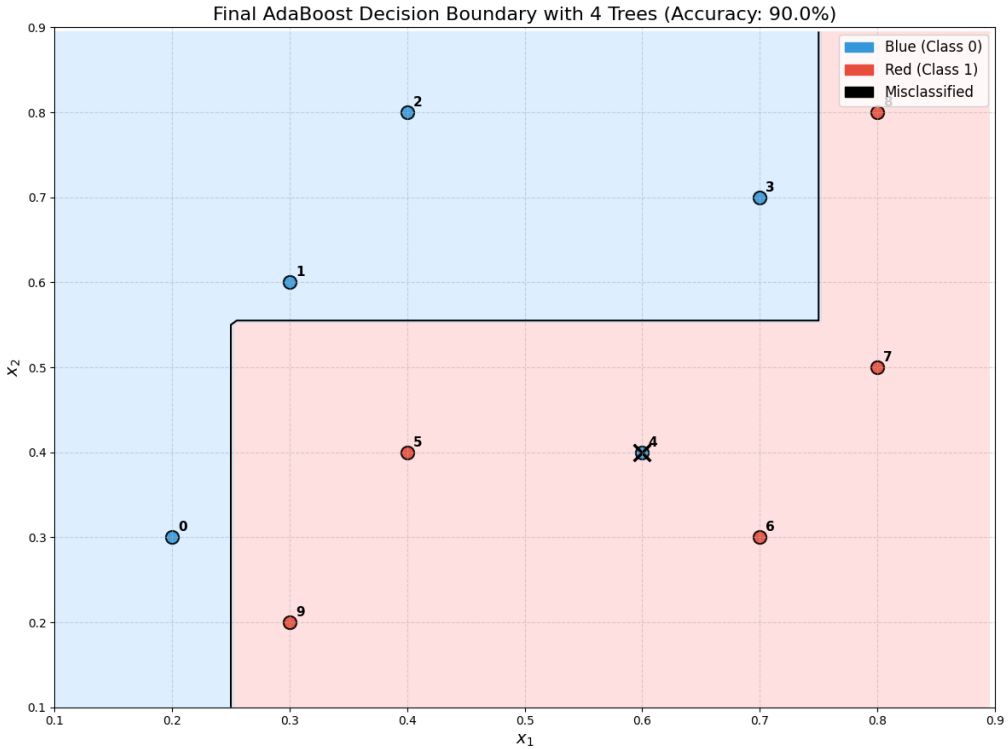
Sample	y	f_1	α_1	f_2	α_2	f_3	α_3	f_4	α_4	f_{boost}
0	0	0	0.424	1	0.650	0	0.529	0	0.519	0
1	0	0	0.424	0	0.650	0	0.529	1	0.519	0
2	0	0	0.424	0	0.650	0	0.529	1	0.519	0
3	0	0	0.424	0	0.650	1	0.529	1	0.519	0
4	0	0	0.424	1	0.650	0	0.529	1	0.519	1
5	1	0	0.424	1	0.650	0	0.529	1	0.519	1
6	1	0	0.424	1	0.650	1	0.529	1	0.519	1
7	1	1	0.424	1	0.650	1	0.529	1	0.519	1
8	1	1	0.424	0	0.650	1	0.529	1	0.519	1
9	1	0	0.424	1	0.650	0	0.529	1	0.519	1

AdaBoost: Final Decision Boundary

Imperial College Business School

The Power of Ensemble Learning

- The final decision boundary combines all four weak classifiers
- This demonstrates how AdaBoost transforms simple models into sophisticated classifiers
- Accuracy improves from 70% (individual trees) to 90% (ensemble)



Outline

Algorithm Gradient Boosting Algorithm

Require: Training data $\{(x_i, y_i)\}_{i=1}^n$, loss function L, number of trees M, learning rate η

Ensure: Gradient Boosted model

1: Initialize model with a constant: $F_0(x) = \arg\min_{\hat{y}} \sum_{i=1}^n L(y_i, \hat{y})$

2: **for** $m = 1$ to M **do**

3: Compute pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1 : n$$

4: Fit a new regression tree to the pseudo-residuals r_{im}

5: Compute optimal value for each leaf region R_{jm}:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

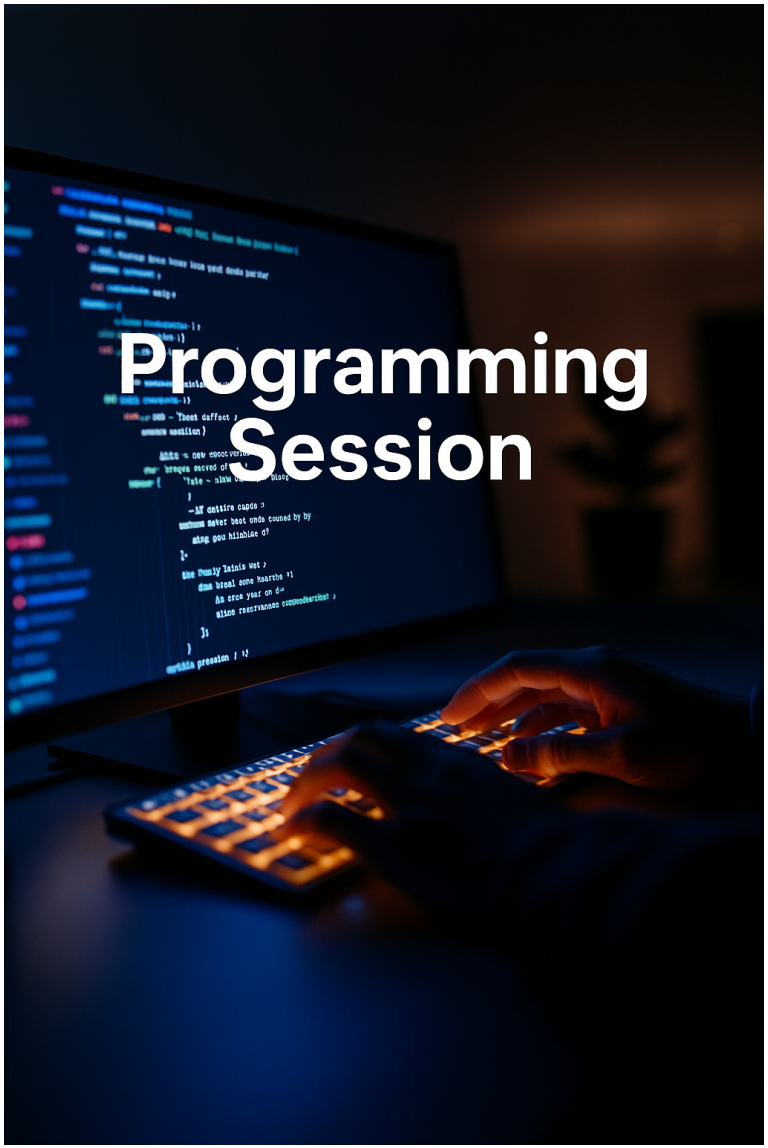
6: Update equation: $F_m(x) = F_{m-1}(x) + \eta \cdot \gamma_m$

7: **end for**

8: **return** Final model $F_M(x)$

Feedback Poll

Click here to participate in the poll



Programming Session 4: Introducing Supervised Learning Algorithms for Time Series Forecasting

- Section 1: Preprocessing the Dataset.
- Section 2: Tree based Models for Time Series Forecasting.
- *Click here to access the programming session*

Solution will be posted tonight on the GitHub page.

- *Click here to access ccess the GitHub Page*

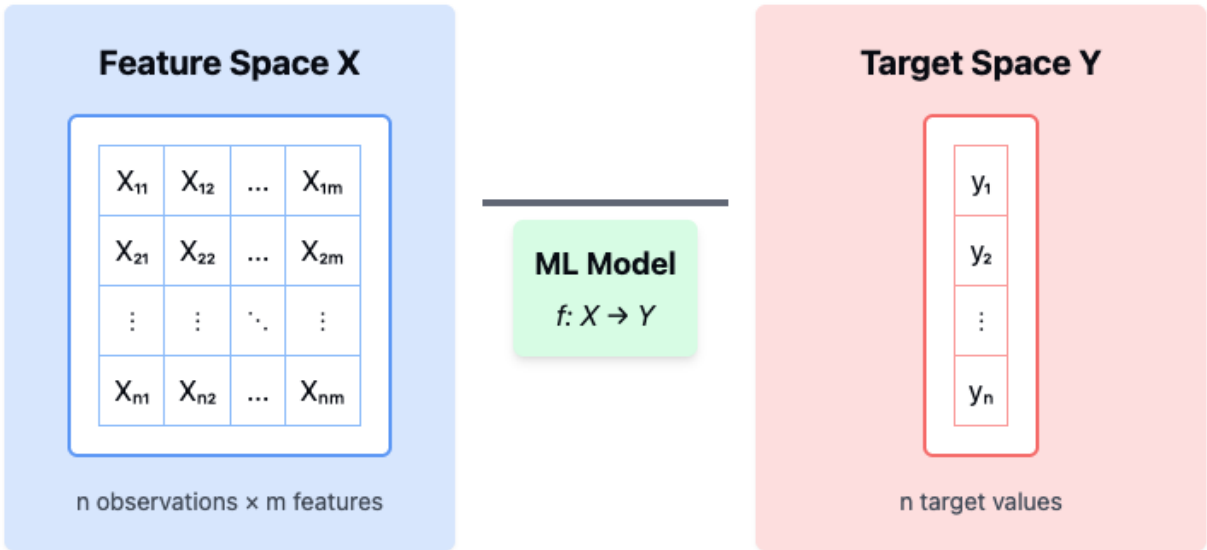
Outline

Outline

Imperial College Business School

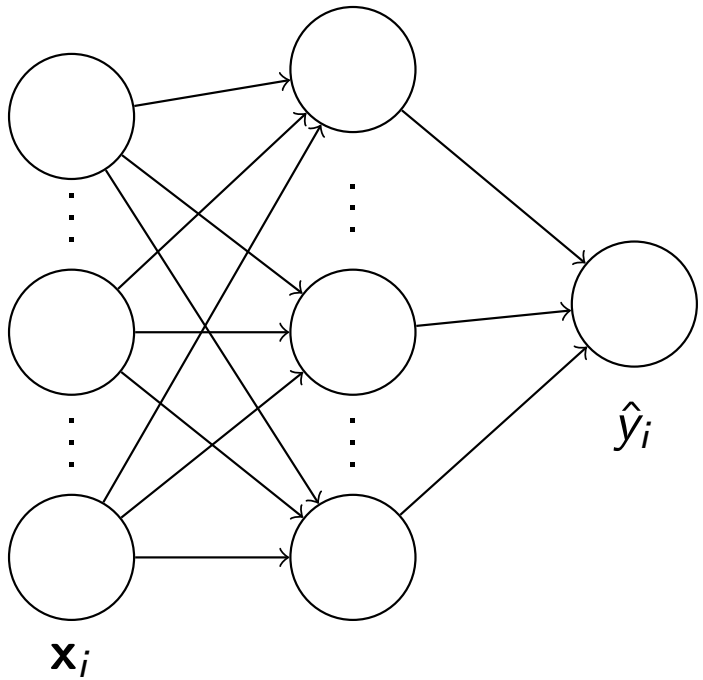
Supervised Learning:

- Learn a function $f : X \to Y$ from labeled data.
- **Feature space** X: matrix of features, n observations $\times$ m features.
- **Target space** Y: vector of n labels (or target values).



Shallow Neural Network

- **Feature vector**:
  $\mathbf{x}_i = (x_{i1}, \ldots, x_{im}) \in \mathbb{R}^m$
- **Parameters**:
  $\theta = \{(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}), (\mathbf{W}^{(2)}, \mathbf{b}^{(2)})\}$
  - $\mathbf{W}^{(1)} \in \mathbb{R}^{m \times L}$: weights connecting input to hidden layer
  - $\mathbf{b}^{(1)} \in \mathbb{R}^{L}$: biases for hidden layer
  - $\mathbf{W}^{(2)} \in \mathbb{R}^{L \times 1}$: weights connecting hidden to output
  - $\mathbf{b}^{(2)} \in \mathbb{R}$: bias for output layer
- **Model**: $f_\theta : \mathbb{R}^m \to \mathbb{R}$ mapping features to predictions
- **Output**: $\hat{y}_i$ (category for classification or continuous value for regression)



Forward Propagation

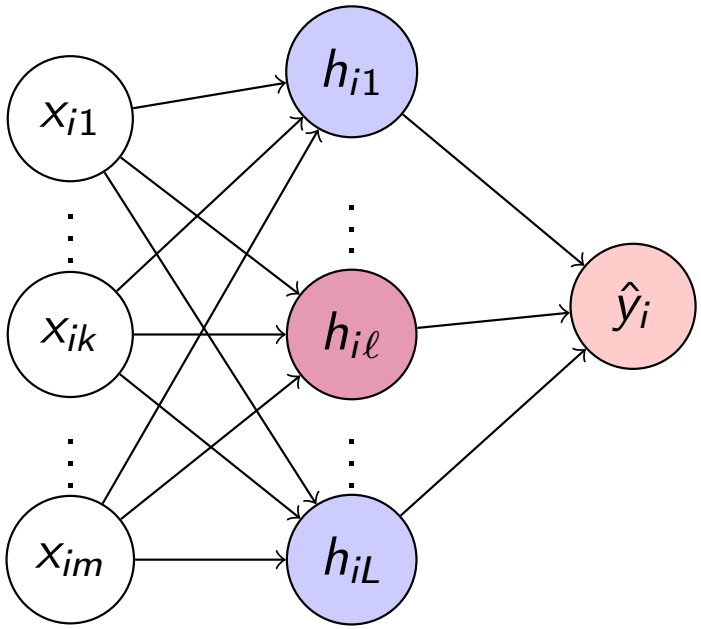
- For all $\ell \in \{1, 2, \ldots, L\}$:

$$h_{i\ell} = \sigma_1 \left(\sum_{k=1}^{m} W_{k\ell}^{(1)} \cdot x_{ik} + b_{\ell}^{(1)} \right)$$

- Then the output:

$$\hat{y}_i = \sigma_2 \left(\sum_{\ell=1}^{L} W_{\ell}^{(2)} \cdot h_{i\ell} + b^{(2)} \right)$$

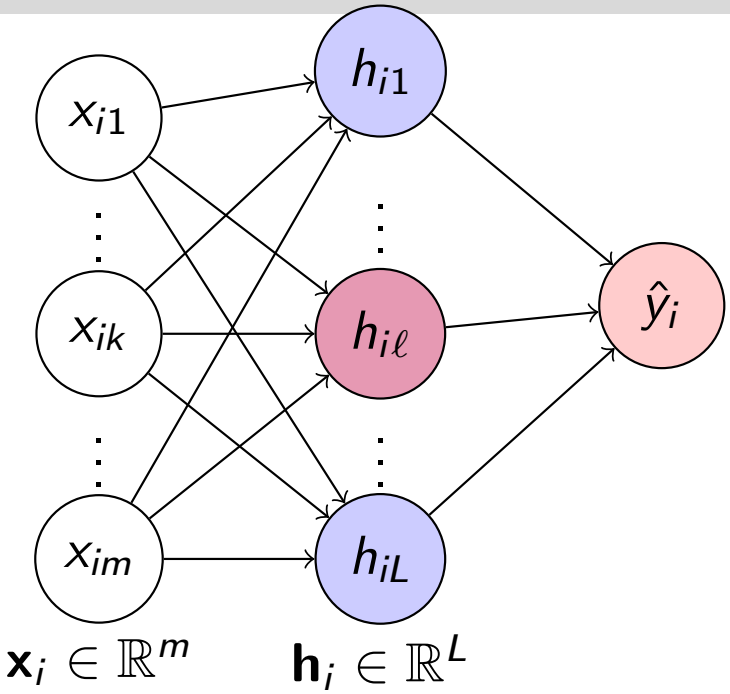
- σ_1, σ_2 are activation functions.
- Parameters:
  $\theta = \{(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}), (\mathbf{W}^{(2)}, \mathbf{b}^{(2)})\}$



Forward Propagation - Matrix Notation

- $\mathbf{x}_i \in \mathbb{R}^m$ is the input feature vector
- $\mathbf{h}_i \in \mathbb{R}^L$ is the hidden layer vector
- The model output $\hat{y}_i \in \mathbb{R}$ is calculated as follows:

$$\mathbf{h}_i = \sigma_1(\mathbf{W}^{(1)T}\mathbf{x}_i + \mathbf{b}^{(1)})$$
$$\hat{y}_i = \sigma_2((\mathbf{W}^{(2)})^T\mathbf{h}_i + b^{(2)})$$



- The final output is:

$$\boxed{\hat{y}_i = f_\theta(x_i) = \sigma_2\left((\mathbf{W}^{(2)})^T\sigma_1\left(\mathbf{W}^{(1)T}\mathbf{x}_i + \mathbf{b}^{(1)}\right) + b^{(2)}\right)}$$

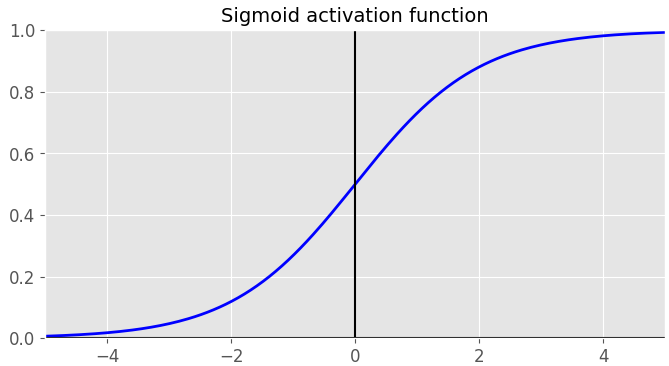
- **Next step:** Learning the parameters $\theta = \{(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}), (\mathbf{W}^{(2)}, \mathbf{b}^{(2)})\}$ from training data using a loss function

Outline

Activation Functions

Sigmoid:

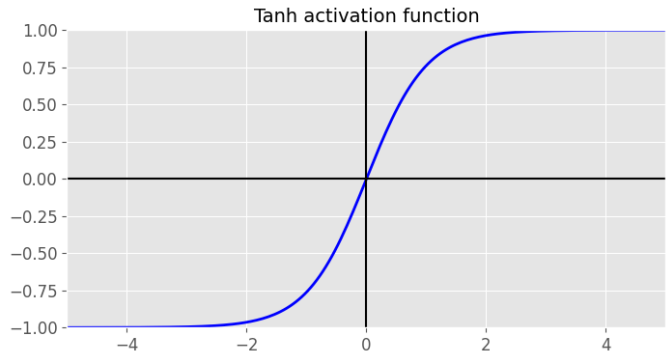
$$\sigma(z) = \frac{1}{1 + e^{-z}} \in [0, 1]$$



- Used as final activation for binary classification
- Output interpreted as probability: $\hat{y}_i = P(Y = 1|\mathbf{x}_i)$

Tanh:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \in [-1, 1]$$

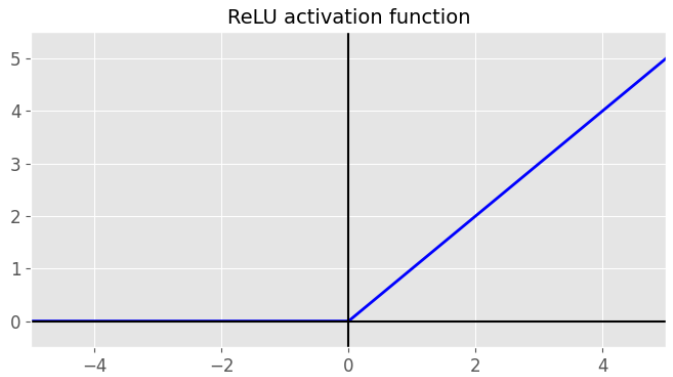


- Zero-centered, helps with convergence
- Often used in hidden layers

ReLU Activation Function

ReLU (Rectified Linear Unit):

$$\text{ReLU}(z) = \max(0, z)$$

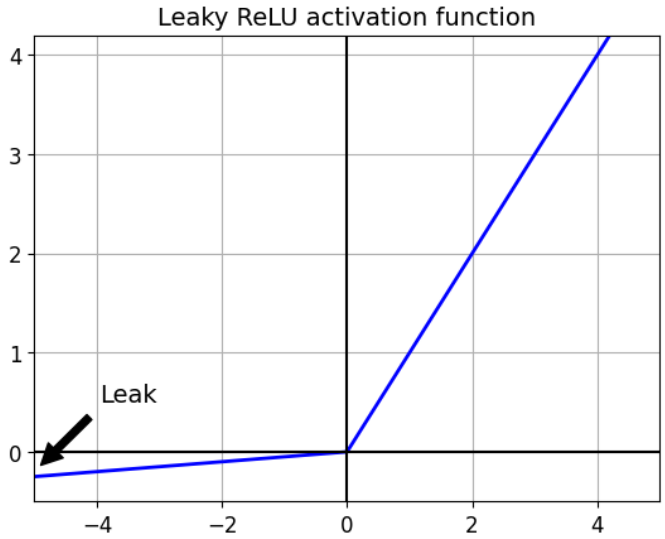


- Faster to compute, computational efficiency
- No saturation for positive values, helps with gradient flow
- Problem: "dying ReLU" (neurons can get stuck at 0)
- Most widely used in hidden layers
- Sparse activation: typically 50% of neurons inactive
- No vanishing gradient for positive inputs

Leaky ReLU Activation Function

Leaky ReLU:

$$\text{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$$

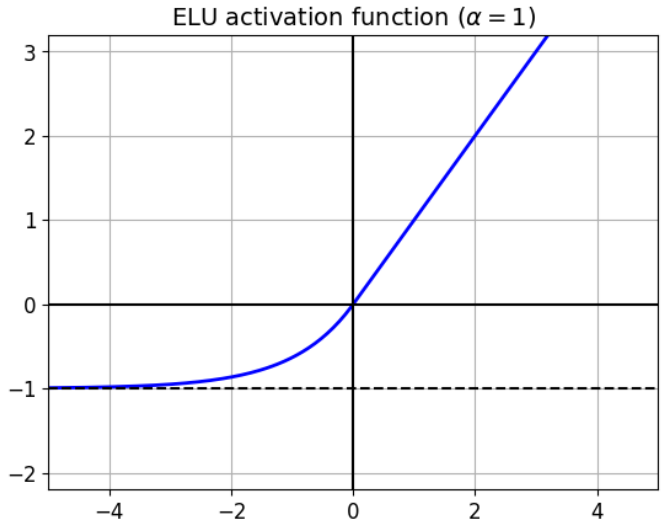


- Introduced in [4].
- Prevents "dying ReLU" problem with small slope α
- Typical values for α range from 0.01 to 0.2
- Allows small gradient flow for negative inputs
- Maintains most of the computational efficiency of ReLU
- Not always superior to ReLU in practice

ELU Activation Function

ELU (Exponential Linear Unit):

$$\text{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$$

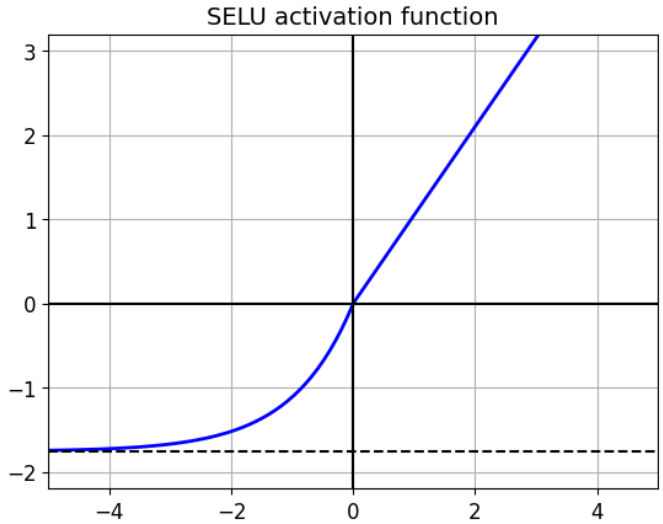


- Introduced in [1].
- Smooth curve for negative values, reducing noise
- Approaches $-\alpha$ as z becomes very negative
- Self-regularizing: can help with internal covariate shift
- More computationally expensive than ReLU
- Often produces faster convergence in training

SELU Activation Function

SELU (Scaled ELU):

$$\text{SELU}(z) = \lambda \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$$



- Introduced in [3].
- Fixed parameters: $\alpha \approx 1.67$ and $\lambda \approx 1.05$
- Scaling factor λ enables self-normalization
- Automatically preserves mean and variance of inputs
- Helps training deep networks without batch normalization
- Requires "SELU initialization" (LeCun Normal)

Outline

- **Why do we need loss functions?**
  - Quantify how well our model f_θ performs on data
  - Provide a differentiable objective to optimize
  - Guide the learning of parameters θ
- **From predictions to learning:**
  - Forward propagation gives us: $\hat{y}_i = f_\theta(\mathbf{x}_i)$
  - Loss function measures: $\mathcal{L}(\hat{y}_i, y_i)$ - the discrepancy between predictions and true values
- **Overall objective:**

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f_\theta(\mathbf{x}_i), y_i)$$

Binary Cross Entropy Loss Function - MLE

Imperial College Business School

- **Dataset**: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $y_i \in \{0, 1\}$
- **Probabilistic view**: Neural network outputs represent probabilities
- **For binary classification:**
  $\forall i \in \{1, \ldots, n\} \quad \hat{y}_i = f_\theta(\mathbf{x}_i) = p_\theta(Y = 1|\mathbf{x}_i)$
- **Log-likelihood for all data:**

$$\begin{aligned}\log \mathcal{L}(\theta) &= \sum_{i=1}^{n} \log p_\theta(Y = y_i|\mathbf{x}_i) \\ &= \sum_{i=1}^{n} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]\end{aligned}$$

- **The loss function is the normalized negative Log-Likelihood:**

$$\min_\theta -\frac{1}{n} \log \mathcal{L}(\theta) \iff \min_\theta -\frac{1}{n} \sum_{i=1}^{n} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Categorical Cross Entropy Loss Function - MLE

Imperial College Business School

- **Dataset**: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $y_i \in \{1, 2, \ldots, K\}$
- **Probabilistic view**: Neural network outputs represent probability distribution over K classes
- **For multi-class classification:**
  $\forall i \in \{1, \ldots, n\} \quad \hat{\mathbf{y}}_i = f_\theta(\mathbf{x}_i) = (p_\theta(Y = 1|\mathbf{x}_i), \ldots, p_\theta(Y = K|\mathbf{x}_i))$
- **One-hot encoding of target:** $\mathbf{y}_i = (y_{i1}, y_{i2}, \ldots, y_{iK})$ where $y_{ik} = \mathbb{1}_{[y_i = k]}$
- **Log-likelihood for all data:**

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log p_\theta(Y = y_i|\mathbf{x}_i) = \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

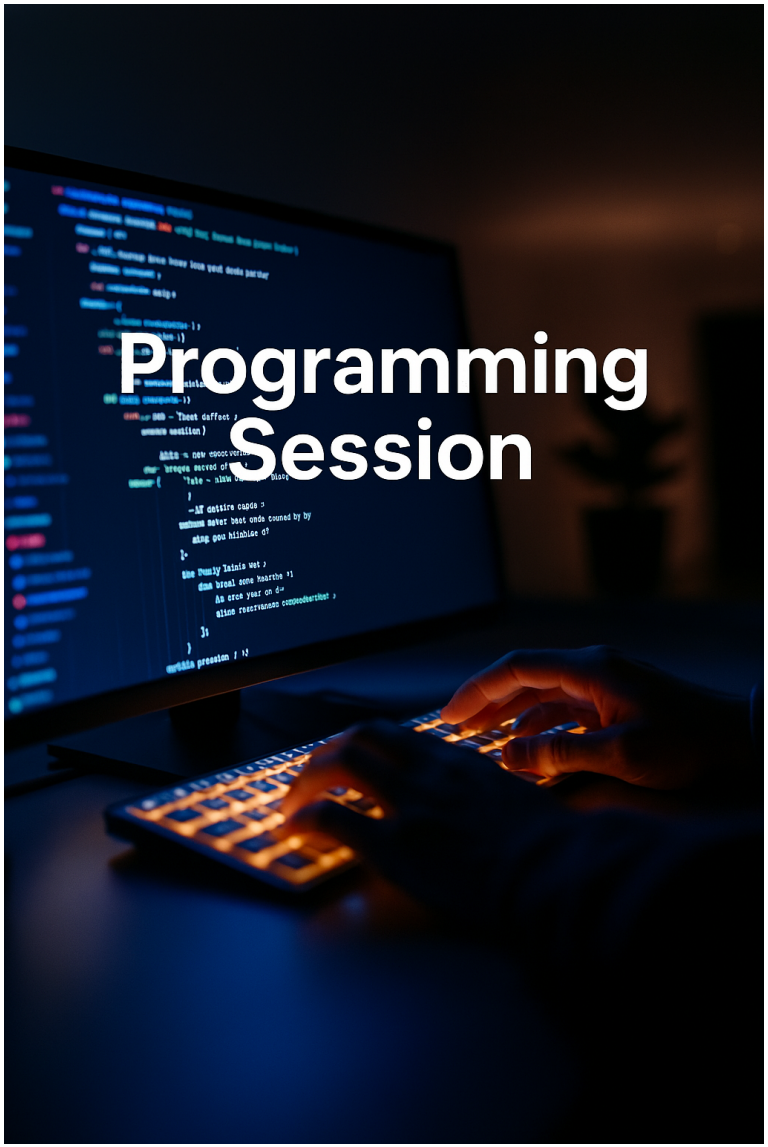
- **The loss function is the normalized negative Log-Likelihood:**

$$\min_\theta -\frac{1}{n} \log \mathcal{L}(\theta) \iff \min_\theta -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

Regression Loss Functions Overview

Imperial College Business School

Loss	Formula	Key Properties
MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	▶ Differentiable everywhere ▶ Sensitive to outliers
MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$	▶ Less sensitive to outliers ▶ Non-differentiable at zero
MAPE	$\frac{1}{n}\sum_{i=1}^{n}\frac{\lvert y_i - \hat{y}_i\rvert}{\lvert y_i\rvert}$	▶ Scale-independent



Optional Programming Session: The Custom Huber Loss function

- *Click here to access the programming session*

Content:

- Tensors and operations in TensorFlow.
- Computing Gradients with Autodiff.
- Custom Loss Function: The Huber Loss.

- **For classification:**
  - **Binary classification**: Binary Cross-Entropy with sigmoid

$$\mathcal{L}_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

  - **Multi-class classification**: Categorical Cross-Entropy with softmax

$$\mathcal{L}_{CCE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

  - **Multi-label classification**: Binary Cross-Entropy per label

$$\mathcal{L}_{ML} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}[y_{ik} \log \hat{y}_{ik} + (1-y_{ik})\log(1-\hat{y}_{ik})]$$

Loss Function Guidelines - Regression

Imperial College Business School

- **For regression:**
  - **Clean data, normal distribution**: MSE

$$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  - **Data with outliers**: MAE

$$\mathcal{L}_{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

  - **Data with outliers (alternative)**: Huber Loss

$$\mathcal{L}_{\delta} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{otherwise}\end{cases}$$

  - **Relative error important**: MAPE

$$\mathcal{L}_{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{|y_i|} \times 100\%$$

Outline

Algorithm Gradient Descent Algorithm

Require: Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, loss function $\mathcal{L}$, learning rate α, iterations T

Ensure: Optimized parameters θ

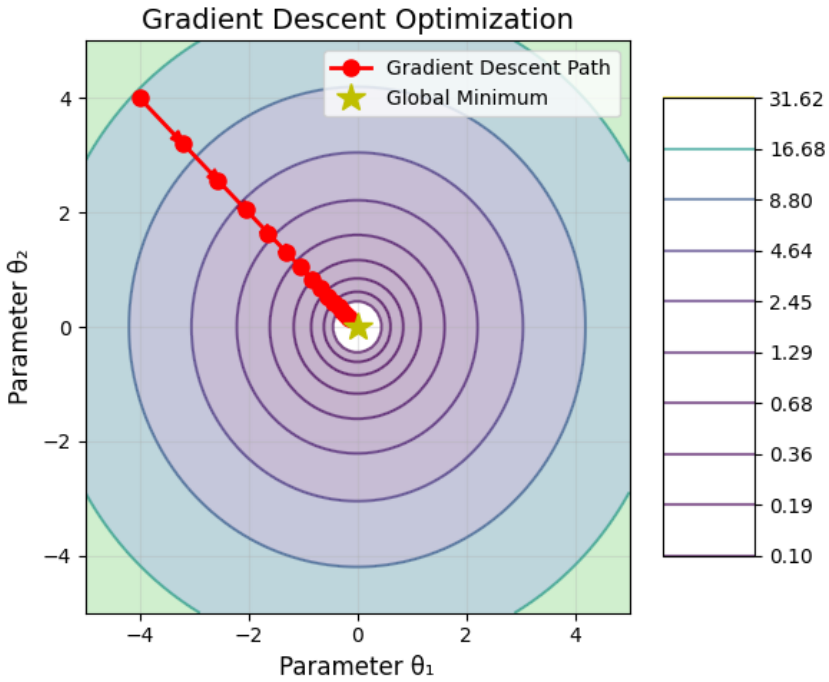
1: Initialize parameters $\theta^{(0)}$ randomly
2: **for** $t = 1$ to T **do**
$$\theta^{(t)} = \theta^{(t-1)} - \alpha \cdot \nabla_\theta \mathcal{L}(\theta^{(t-1)})$$
3: **if** Convergence criteria met **then**
4: **break**
5: **end if**
6: **end for**
7: **return** Final parameters $\theta^{(T)}$

Visualizing Gradient Descent

Imperial College Business School

Key aspects of Gradient Descent:

- **Intuition**: Move downhill in the direction of steepest descent
- **Learning rate** α controls step size:
  - Too small: slow convergence
  - Too large: overshooting or divergence
- For neural networks, loss landscapes are typically **non-convex**, so we often converge to a **local minimum**

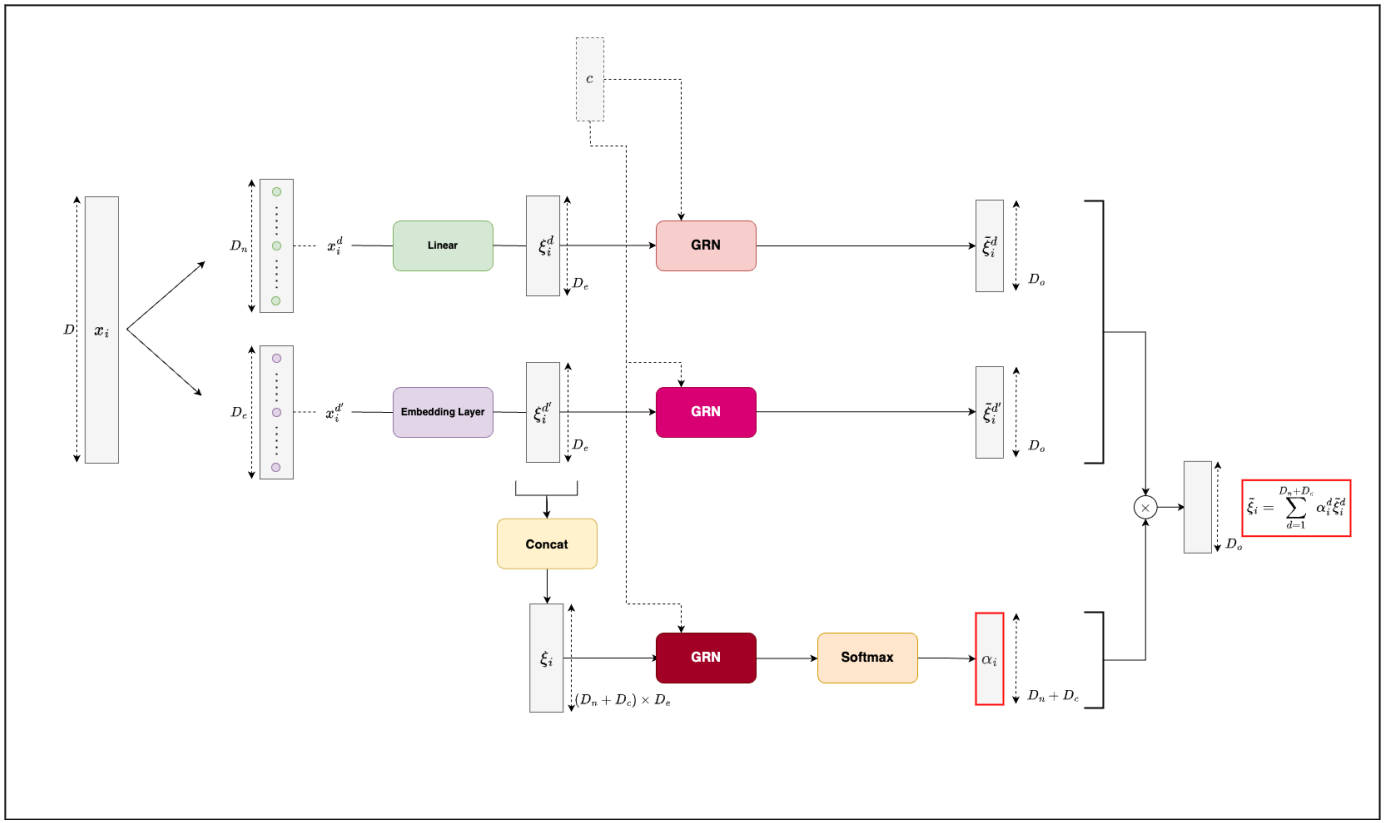


- In this figure, we reach the **global minimum** since the function is **convex**

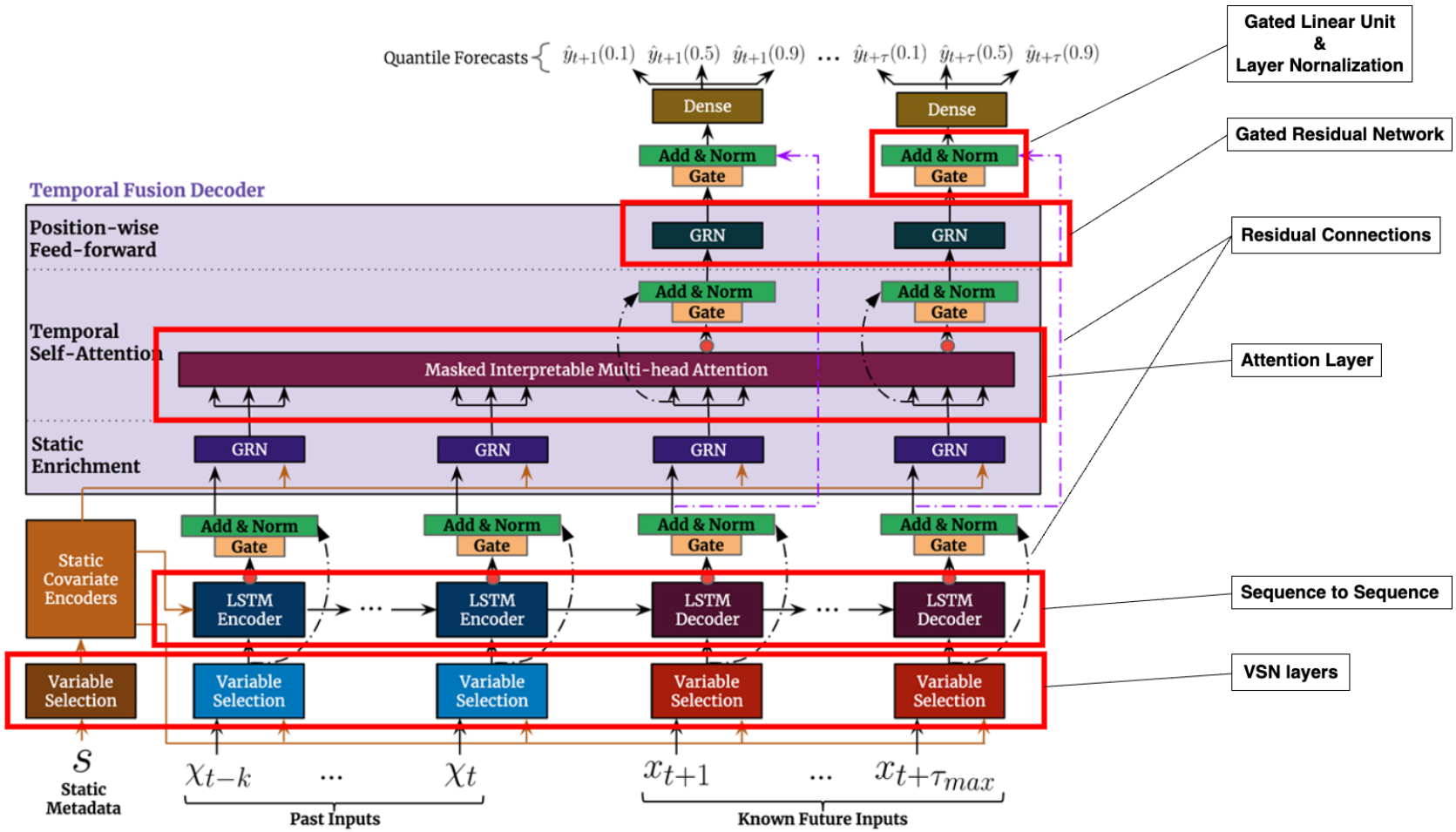
Outline

Variable Selection Network (VSN)

- We will explore the Variable Selection Network (VSN) in Lecture 7 as part of the Temporal Fusion Transformer Architecture.

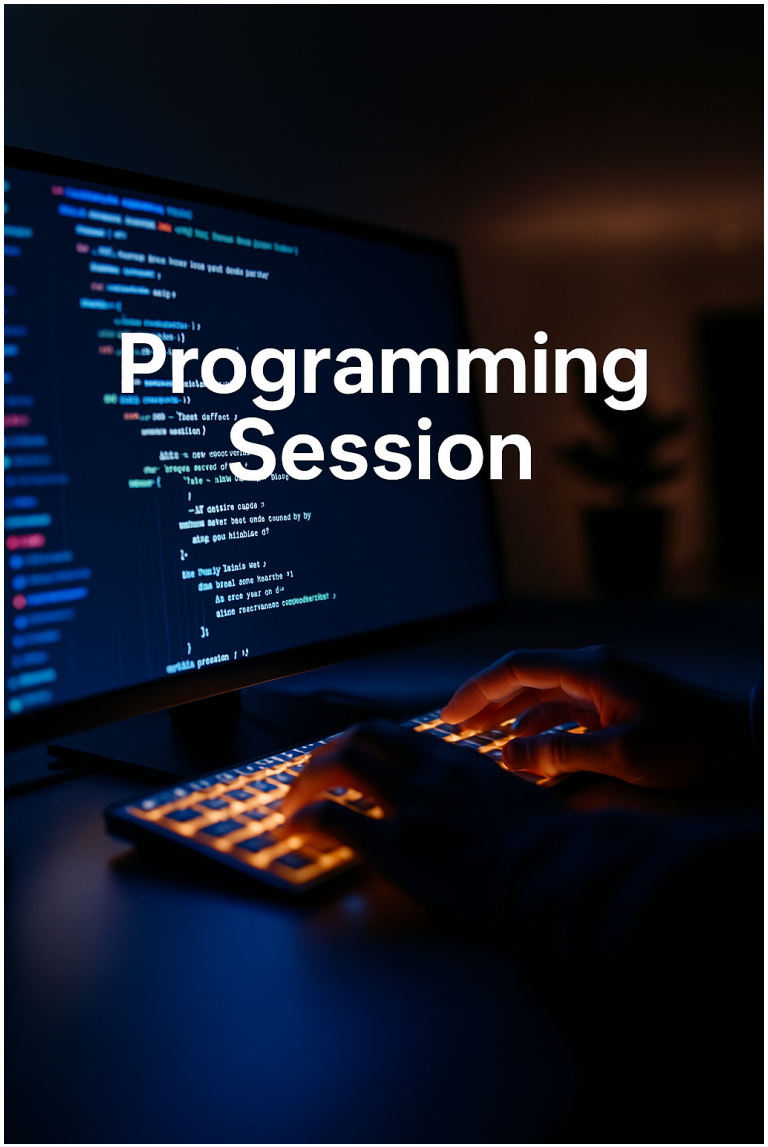


TFT Architecture: High-Level View



Feedback Poll

Click here to participate in the poll



Programming Session 4: Introducing Supervised Learning Algorithms for Time Series Forecasting

- Section 3: Neural Networks.
- Section 4: Performance Analysis.
- *Click here to access the programming session*

Solution will be posted tonight on the GitHub page.

- *Click here to access ccess the GitHub Page*

Quiz Time!

Click here to take the quiz

Thank you for your attention

References I

[1] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[2] Yoav Freund, Robert E Schapire, et al. "Experiments with a new boosting algorithm". In: *icml*. Vol. 96. Citeseer. 1996, pp. 148–156.

[3] Günter Klambauer et al. "Self-normalizing neural networks". In: *Advances in neural information processing systems* 30 (2017).

[4] Bing Xu et al. "Empirical evaluation of rectified activations in convolutional network". In: *arXiv preprint arXiv:1505.00853* (2015).