
SESSION // 03

TRAINING NEURAL NETWORKS

FACULTY OF
SCIENCE AND ENGINEERING
+++

MANCHESTER
1824
The University of Manchester



Diego Corona Lopez – AI Technical Specialist





AGENDA

Introduction to Artificial Neural Networks

- Understanding neural networks and their components
- Exploring neuron structure and activation functions

PyTorch Workflow Overview

Case Study: ARKOMA Robot Dataset

Data Preparation and Model Design

- Preprocessing and normalizing data
- Designing network architecture

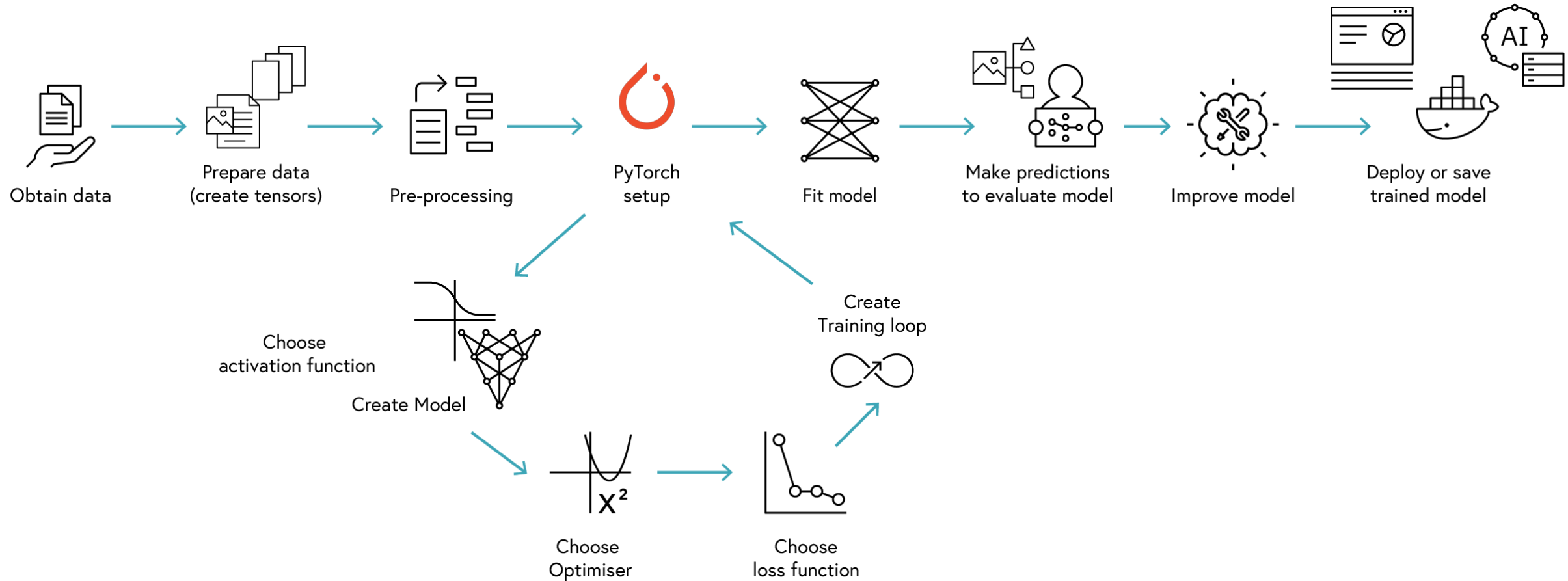
Training and Evaluation

- Implementing loss functions and optimizers
- Monitoring model performance

Results Analysis and Visualization

Challenges & Next Steps

PYTORCH WORKFLOW



ARKOMA ROBOT DATASET

Critical problem in robotics:

"How should joints move to place end-effector at desired position?"

Why Inverse Kinematics Matter:

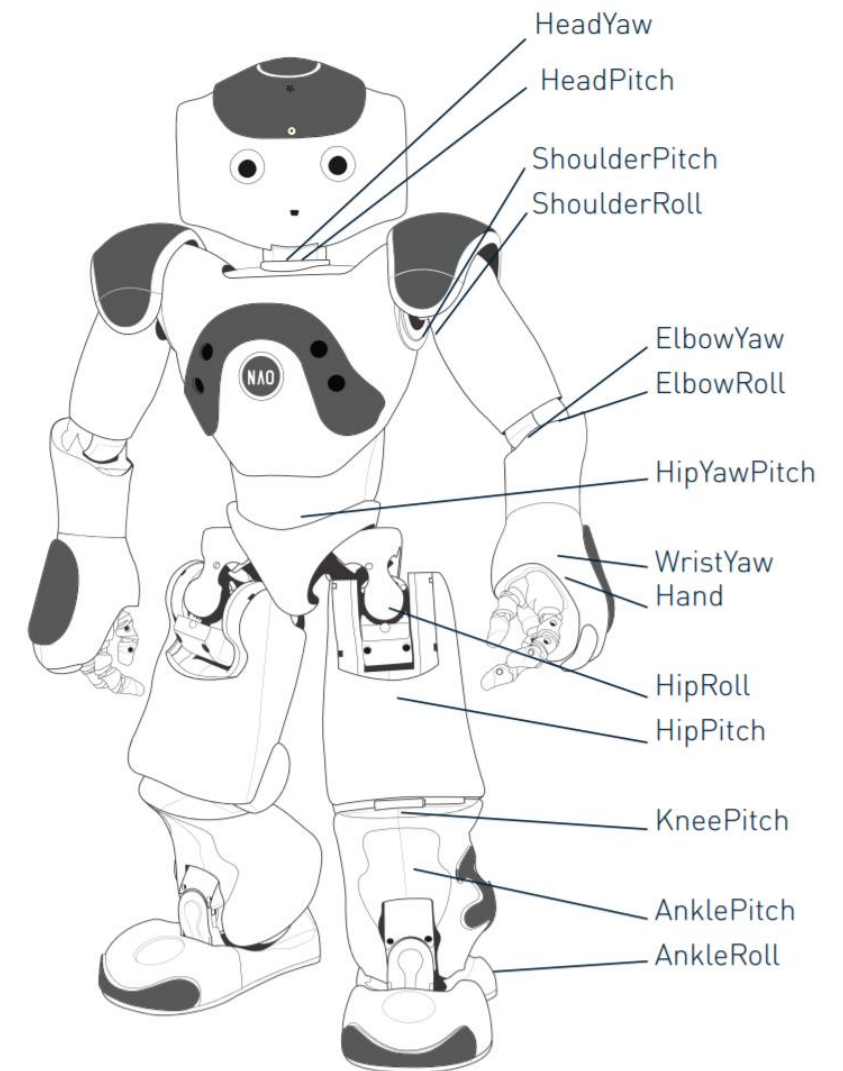
- Forward kinematics: Easy to calculate (joint angles \rightarrow position)
- Inverse kinematics: Challenging mathematical problem (position \rightarrow joint angles)

Real-world applications:

- Robot manipulation tasks (grasping objects)
- Manufacturing automation
- Surgical robotics

NAO robot inverse kinematics dataset

- 10,000 input-output pairs
 - Inputs: End-effector positions ($P_x, P_y, P_z, R_x, R_y, R_z$)
 - Outputs: Joint angles ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$)
 - We'll focus on the right arm
-



DATA PREPARATION AND PRE-PROCESSING

```
from sklearn.model_selection import train_test_split

# Load the complete dataset
X = pd.read_csv("dataset_features.csv")
y = pd.read_csv("dataset_targets.csv")

# First split: separate test set (80% train+val, 20% test)
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Second split: separate validation set (75% train, 25% val from the temp set)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.25, random_state=42
) # 0.25 * 0.8 = 0.2 of original data

print(f"Training set: {X_train.shape[0]} samples")
print(f"Validation set: {X_val.shape[0]} samples")
print(f"Test set: {X_test.shape[0]} samples")
```

Purpose of each dataset:

- Training (60-80%):
 - Training the model
- Validation (10-20%):
 - Tuning hyperparameters
- Test (10-20%):
 - Final evaluation

DATA NORMALISATION

Why normalise?

- Faster convergence
- Numerical stability
- Equal feature contribution
- Better generalization

Min-Max scaling:

$$X_{\text{norm}} = (X - X_{\text{min}}) / (X_{\text{max}} - X_{\text{min}})$$

```
from sklearn.preprocessing import MinMaxScaler

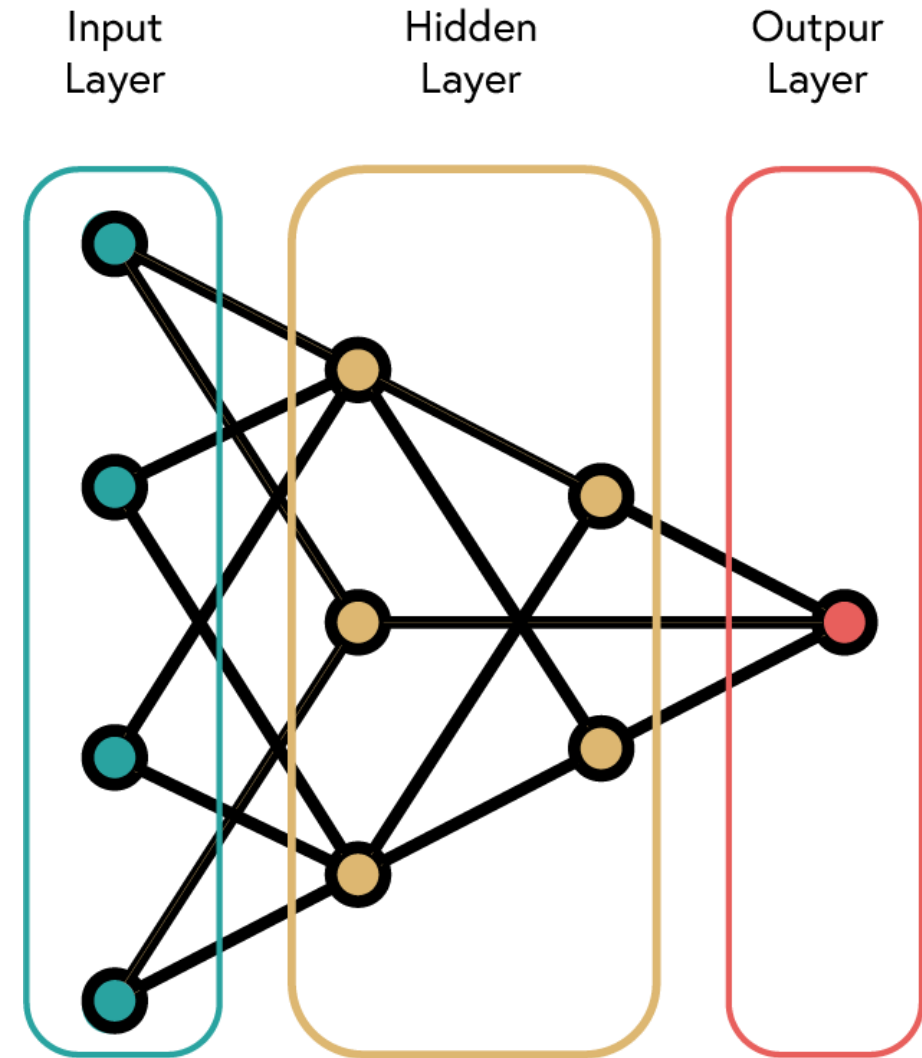
# Create and apply MinMaxScaler
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()

x_scaler.fit(X_train_tensor)
y_scaler.fit(y_train_tensor)

X_train_scaled = torch.tensor(x_scaler.transform(X_train_tensor), dtype=torch.float32)
```

BUILDING NEURAL NETWORKS

- Input Layer: Receives input features
- Hidden Layers: Process information
- Output Layer: Produces predictions
- Width (neurons per layer) vs Depth (number of layers)



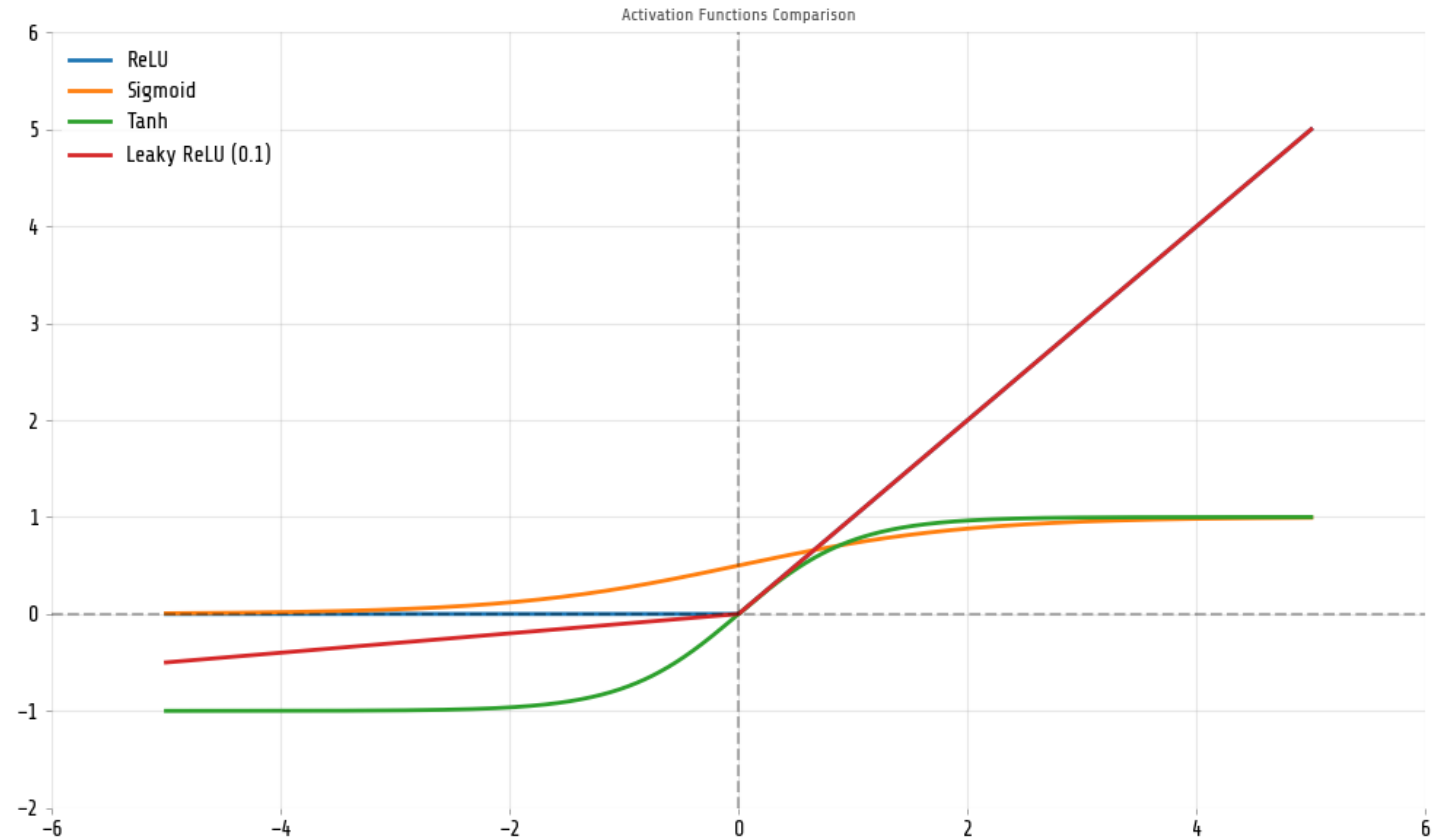
ACTIVATION FUNCTIONS

Purpose:

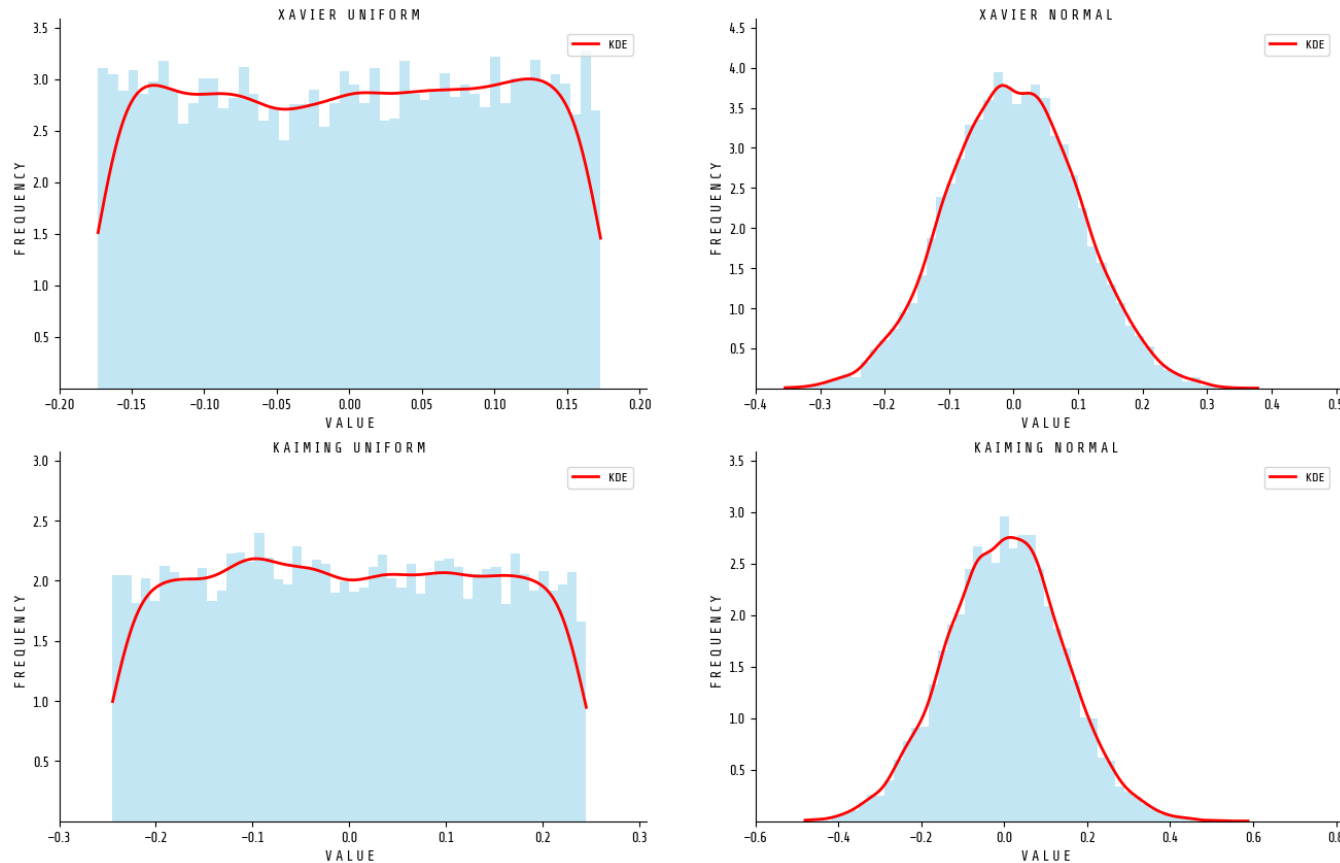
- Transform linear input to non-linear output
- Enable networks to learn complex patterns and relationships

Key properties:

- Differentiable
- Non-linear
- Computationally efficient



WEIGHT INITIALISATION



Importance of proper initialization:

- 1. Convergence Speed:** Good initialization leads to faster training
- 2. Symmetry Breaking:** Prevents neurons from learning the same features
- 3. Vanishing/Exploding Gradients:** Proper scaling helps maintain gradient flow
- 4. Training Stability:** Reduces the chance of getting stuck in poor local minima
- 5. Reproducibility:** Sets a consistent starting point for experiments


```
# Initialize weights with appropriate methods
torch.nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='relu')
torch.nn.init.zeros_(self.fc1.bias)
```

LOSS FUNCTION

- Loss functions quantify prediction errors
- We use Mean Squared Error (MSE):

$$MSE = 1/n \sum (y - \hat{y})^2$$

- Provides direction for optimization



```
def mse_loss(predictions: torch.Tensor,  
             targets: torch.Tensor) -> torch.Tensor:  
  
    squared_diff = (predictions - targets) ** 2  
    return squared_diff.mean()
```

ANN IMPLEMENTATION

- PyTorch model implementation for robotic arm
- Simple architecture to avoid overfitting

```
class RobotArmNetwork(torch.nn.Module):  
    def __init__(self, input_size, hidden_size, output_size):  
        super().__init__()  
        self.fc1 = torch.nn.Linear(input_size, hidden_size)  
        self.hidden_activation = torch.nn.ReLU()  
        self.fc2 = torch.nn.Linear(hidden_size, output_size)  
  
    def forward(self, x):  
        x = self.hidden_activation(self.fc1(x))  
        x = self.fc2(x)  
        return x
```

OPTIMISATION

Popular optimisers:

- **SGD**: Simple, works well with momentum
 - **Adam**: Adaptive learning rates, widely used
 - **RMSProp**: Good for recurrent networks
 - **AdamW**: Adam with proper weight decay
- Critical importance:

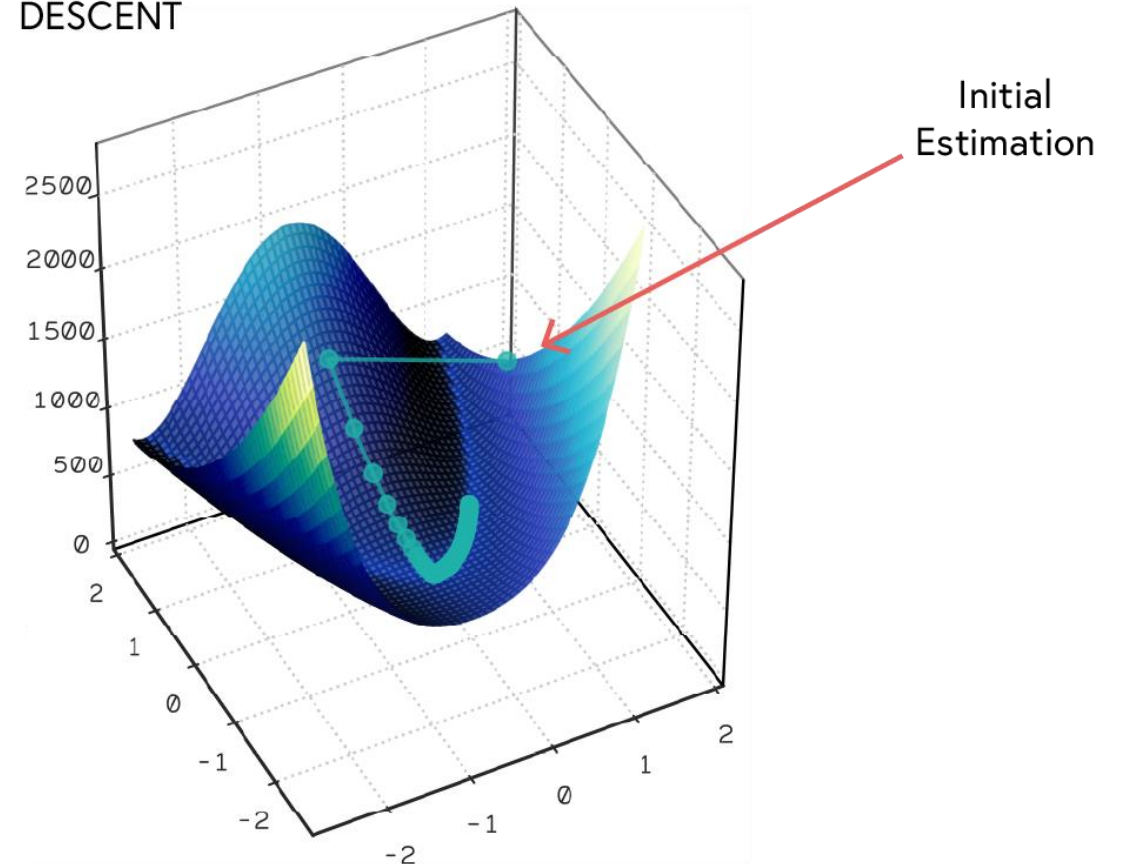
Learning rate importance:

- **Too large**: Causes unstable training, overshooting minima
- **Too small**: Results in slow convergence or getting stuck in local minima
- **"Just right"**: Efficient convergence to good solutions


Learning Rate Strategies:

- **Fixed**: Simple but rarely optimal for entire training
 - **Decay/Scheduling**: Reduce rate over time (e.g., step, exponential, cosine)
 - **Adaptive**: Adjusts automatically based on gradient history
-

GRADIENT DESCENT



PYTORCH OPTIM



```
# Fixed learning rate
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer, step_size=30, gamma=0.1) # Reduce by 10x every 30 epochs

# After each epoch
scheduler.step()
```

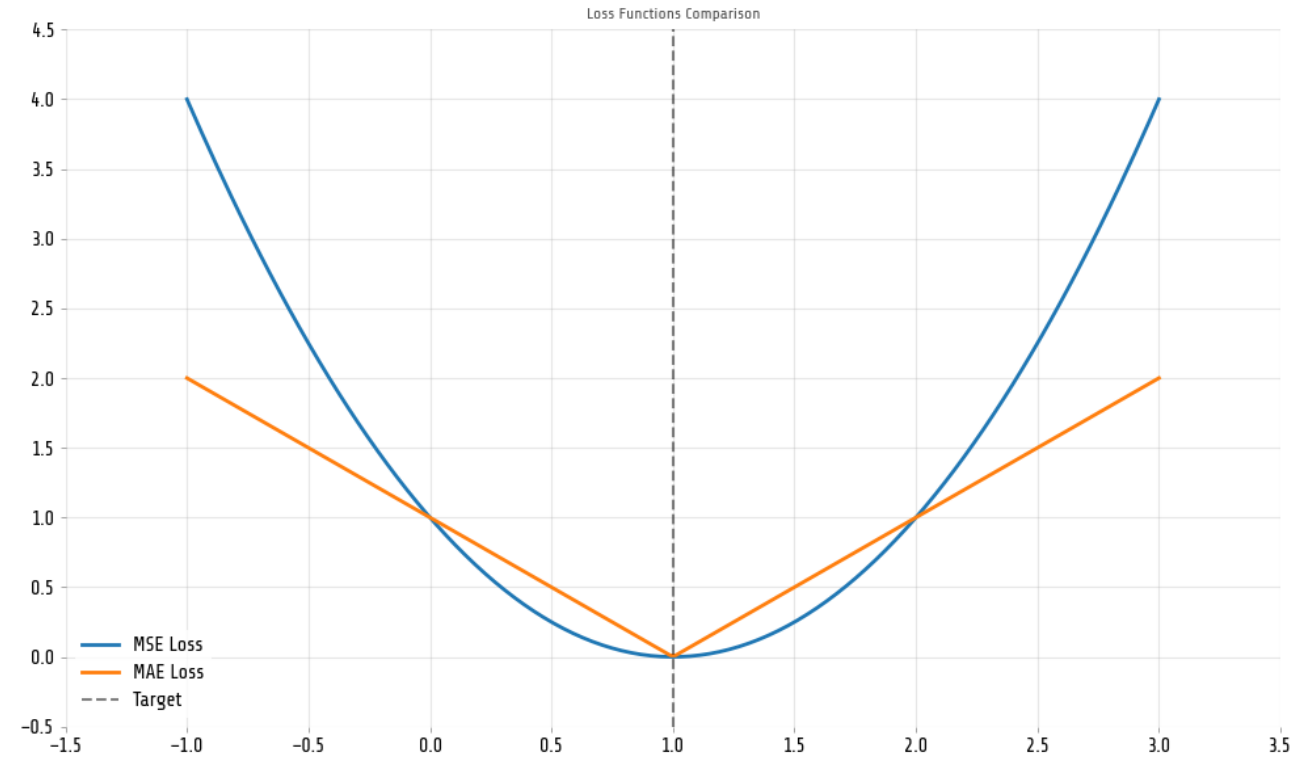
LOSS FUNCTIONS

Types of loss functions:

- **MSE**: Regression tasks
- **MAE**: Regression with less sensitivity to outliers
- **Binary Cross-Entropy**: Binary classification
- **Categorical Cross-Entropy**: Multi-class classification
- **For our regression task**: Mean Squared Error

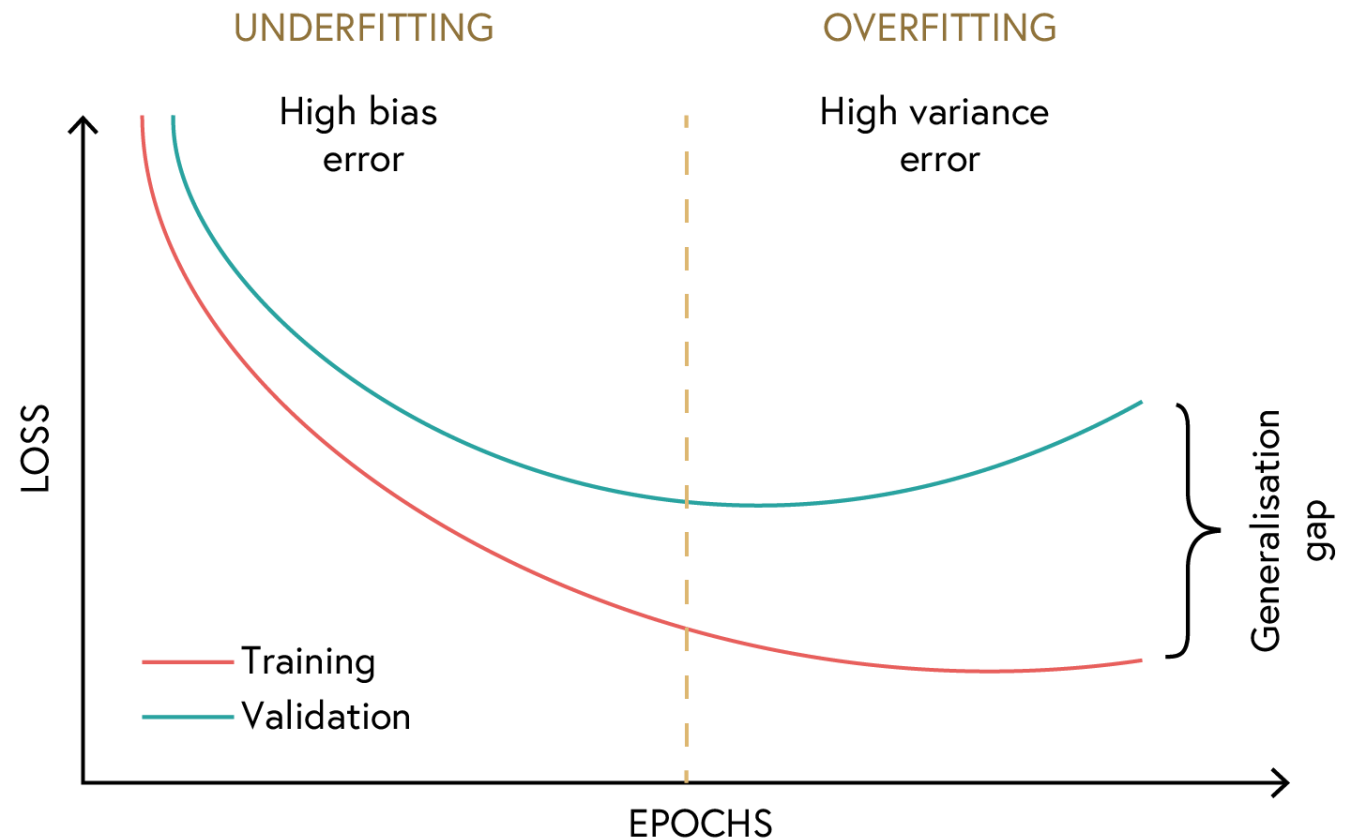


```
loss_function = torch.nn.MSELoss()
```



OVERFITTING AND UNDERFITTING

- **Underfitting:** Model too simple, high bias
- **Overfitting:** Model too complex, high variance
- **Finding the right balance**




MODEL EVALUATION

Testing on unseen data

Metrics for regression:

- **Mean Squared Error**
- **R-squared score**

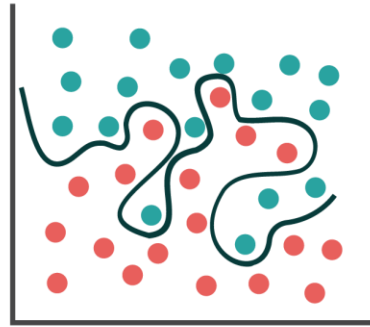


```
model.eval()  
with torch.no_grad():  
    test_predictions = model(X_test_scaled)  
    test_loss = loss_function(test_predictions, y_test_scaled)
```

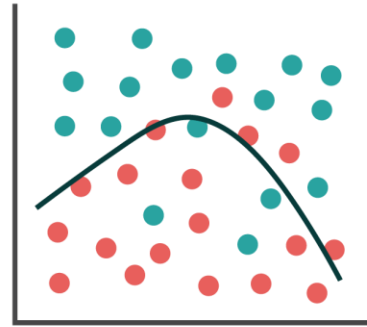
MODEL EVALUATION

CLASSIFICATION

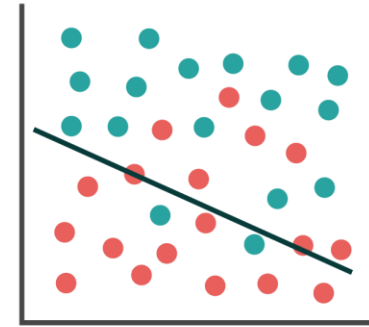
OVERFITTING



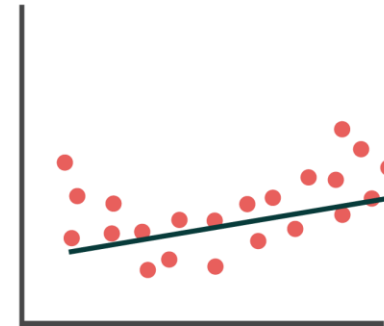
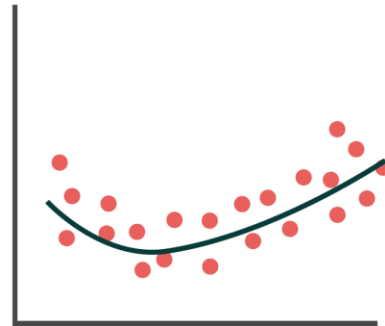
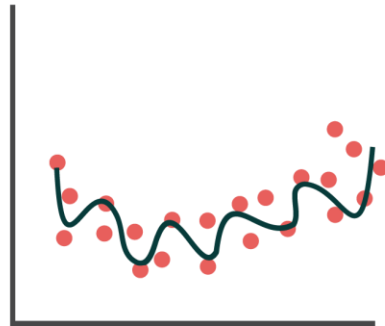
GOOD FIT

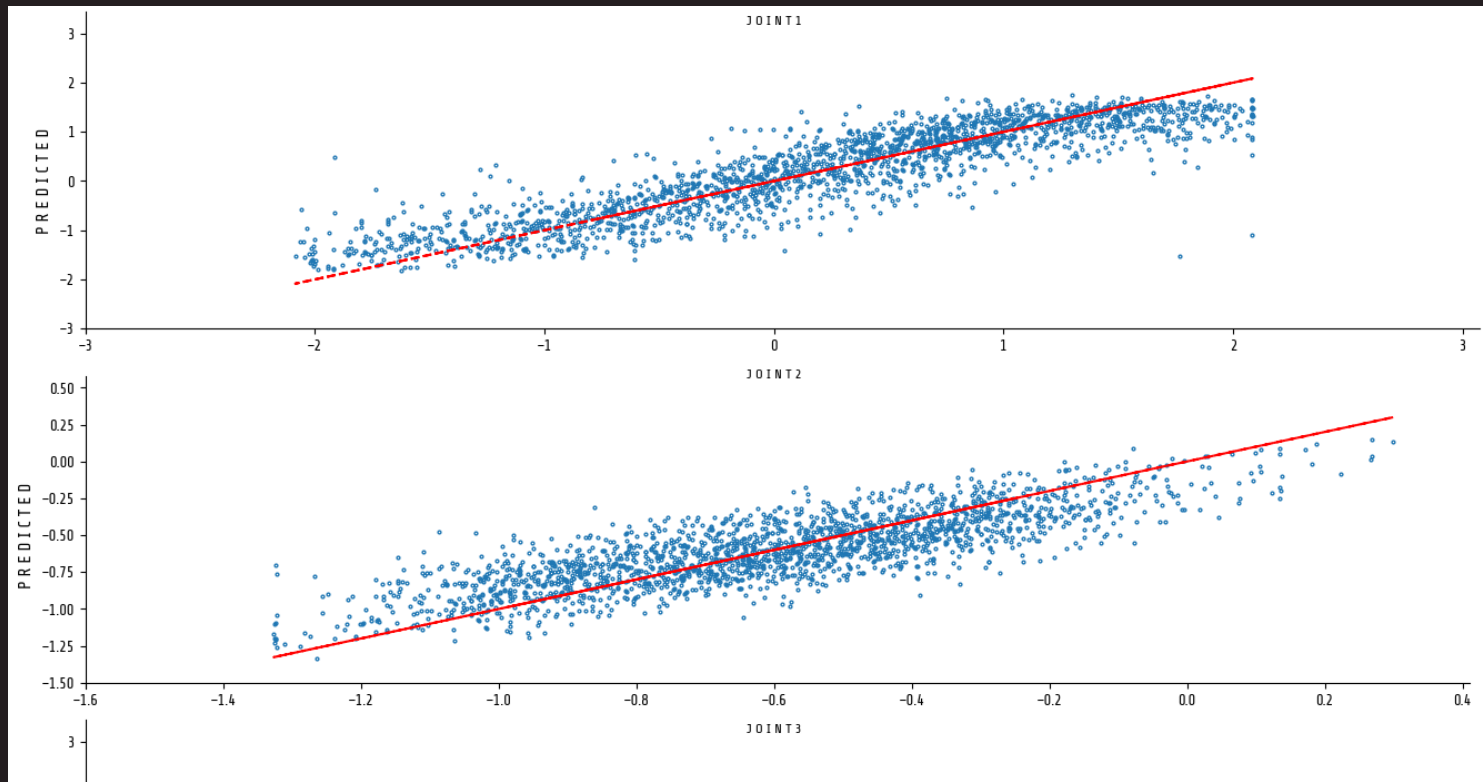


UNDERFITTING



REGRESSION





IMPROVING THE MODEL

- Hyperparameter tuning
- Deeper/wider networks
- Regularisation techniques
- Advanced architectures