Diogo Ribeiro

ESMAD - Escola Superior de Média Arte e Design

Lead Data Scientist, Mysense.ai

dfr@esmad.ipp.pt 0009-0001-2022-7072

From Linear Models to Neural Networks

The evolution from simple to complex models
Linear Regression:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

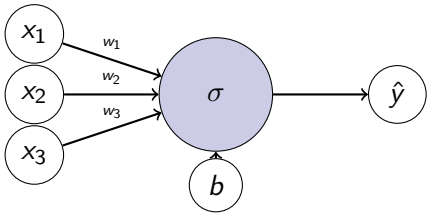
Limitations:

- Can only learn linear relationships
- Decision boundary is a hyperplane
- XOR problem unsolvable

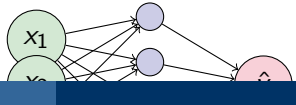
Adding Non-Linearity:
Key Insight: Compose linear transformations with non-linear activation functions!

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

Single Neuron (Perceptron):



Multi-Layer Perceptron (MLP):



Activation Functions

Non-linearities that make deep learning work

Why Activations Matter:

Without non-linearity, stacking layers is pointless:

$$f(W_2(W_1x)) = (W_2W_1)x = Wx$$

Still just linear!

Common Activations:

1. **Sigmoid:**

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Range: $(0, 1)$
Use: Output layer (binary classification)

Activation Function Comparison:

Function	Pros	Cons
Sigmoid	Smooth, interpretable	Vanishing gradient, not zero-cent
Tanh	Zero-centered	Vanishing gradient
ReLU	Fast, no vanishing gradient	Dead neurons (always 0)
Leaky ReLU	Fixes dead ReLU	Needs tuning
ELU	Smooth, negative values	Slower than ReLU
Swish	Smooth, self-gated	More computation

Modern Variants:

- **Leaky ReLU:**

$$f(z) = \begin{cases} z & z > 0 \\ 0.01z & z \leq 0 \end{cases}$$

Backpropagation: Training Neural Networks

The algorithm that makes deep learning possible

The Problem:
Given training data $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, find weights W that minimize loss:

$$\mathcal{L}(W) = \frac{1}{n} \sum_{i=1}^{n} \text{loss}(\hat{y}^{(i)}, y^{(i)})$$

The Solution: Gradient Descent

$$W \leftarrow W - \eta \nabla_W \mathcal{L}$$

But how to compute $\nabla_W \mathcal{L}$ for deep networks?

Backpropagation = Chain Rule
For a 2-layer network:

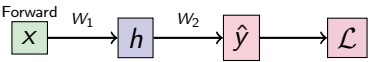
Forward Pass:

1. Compute activations layer by layer
2. Store intermediate values
3. Compute loss at output

Backward Pass:

1. Start from loss
2. Compute gradients backward
3. Update weights

Example: 1 Hidden Layer



Optimization Algorithms

Beyond vanilla gradient descent

Stochastic Gradient Descent (SGD):
Vanilla GD: Use all data

$$W \leftarrow W - \eta \nabla_W \mathcal{L}$$

SGD: Use mini-batches

$$W \leftarrow W - \eta \nabla_W \mathcal{L}_{\text{batch}}$$

Faster, more memory efficient, adds noise (helps escape local minima)

Modern Optimizers:

1. **SGD with Momentum:**

Optimizer Comparison:

Optimizer	Characteristics
SGD	Simple, requires good LR tuning
SGD+Momentum	Faster than SGD, less oscillation
RMSProp	Adaptive LR, good for RNNs
Adam	Adaptive LR, fast convergence, most popular
AdamW	Adam + weight decay (better regularization)
RAdam	Rectified Adam (warm-up built-in)

Learning Rate Schedules:

- **Step decay:** Reduce LR every N epochs
- **Exponential decay:** $\eta_t = \eta_0 e^{-kt}$
- **Cosine annealing:** Smooth reduction
- **1cycle:** Increase then decrease (fast.ai)

PyTorch Example:

Convolutional Neural Networks (CNNs)

Designed for spatial data (images, video)

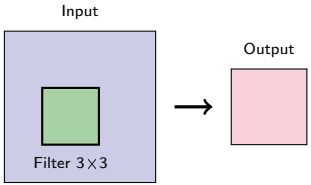
Why CNNs for Images?

- **Fully connected:**
  - 100×100 RGB image = 30K inputs
  - 1st hidden layer (1000 neurons) = 30M parameters!
- **Problems:**
  - Too many parameters
  - Ignores spatial structure
  - Not translation invariant

CNN Solutions:

1. **Local connectivity:**
   - Each neuron sees small region (receptive

Convolution Operation:



Filter slides across image:

$$(\text{img} * \text{filter})_{ij} = \sum_m \sum_n \text{img}_{i+m,j+n} \cdot \text{filter}_{mn}$$

Learned Filters Detect Features:

- Early layers: edges, corners

Modern CNN Architectures

Evolution from LeNet to EfficientNet
Classic Architectures:

1. **LeNet-5 (1998):**
   - 7 layers, digit recognition
   - First successful CNN
2. **AlexNet (2012):**
   - ImageNet winner
   - 8 layers, 60M parameters
   - ReLU, dropout, GPU training
3. **VGG (2014):**
   - 16-19 layers
   - Small 3×3 filters stacked
   - Very deep, simple architecture
4. **ResNet (2015):**
   - 50-152 layers
   - Skip connections!

Modern Developments:

Architecture	Innovation
Inception	Multiple filter sizes in parallel
ResNet	Skip connections (residual blocks)
DenseNet	Every layer connects to every other
MobileNet	Depthwise separable convs (mobile)
EfficientNet	Compound scaling (width+depth+resolution)
Vision Transformer	Self-attention instead of convolutions

Transfer Learning:
Instead of training from scratch:

1. Load pre-trained model (ImageNet)
2. Remove final layer

Recurrent Neural Networks (RNNs)

Networks with memory for sequential data

Why RNNs?

For sequences (text, time series, speech):

- Input length varies
- Temporal dependencies matter
- Order is important

RNN Idea:

Maintain hidden state h_t that captures history:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Hidden state h_t is updated at each timestep.

LSTM (Long Short-Term Memory):

Solution to vanishing gradients:

- **Cell state:** Highway for information flow
- **Gates:** Control info flow
  - Forget gate: What to forget
  - Input gate: What to update
  - Output gate: What to output

LSTM Cell:

$$f_t = \sigma(W_f[h_{t-1}, x_t]) \quad \text{(forget)}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t]) \quad \text{(input)}$$

$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t])$$

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

Transformers: Attention is All You Need

The architecture that revolutionized NLP and beyond

Limitations of RNNs/LSTMs:

- Sequential processing (slow)
- Still struggle with very long sequences
- Hard to parallelize

Transformer Innovation:

Replace recurrence with **self-attention**:

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

where:

- Q: Query (what I'm looking for)
- K: Key (what I have)

Transformer Architecture:

- **Encoder:** Process input sequence
- **Decoder:** Generate output sequence
- **Stack:** 6-12 layers typical

Famous Transformers:

Model	Task
BERT	Encoder-only, understanding
GPT	Decoder-only, generation
T5	Encoder-decoder, seq2seq
Vision Transformer (ViT)	Images as patches
CLIP	Vision-language alignment

Regularization in Deep Learning

Preventing overfitting in large models
The Overfitting Problem:
Deep networks have millions of parameters → can memorize training data!

Regularization Techniques:

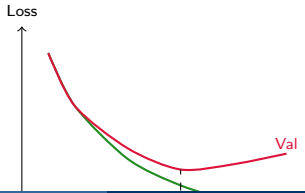
1. **L2 Regularization (Weight Decay):**

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum_i w_i^2$$

Penalizes large weights

2. **Dropout:**
   - Randomly set neurons to 0 (prob p)
   - Forces network to be robust
   - Ensemble effect
   - Typical: $p = 0.5$ for FC, $p = 0.2$ for

Early Stopping:
Monitor validation loss, stop when it starts increasing.



Knowledge Check: Deep Learning

Test your understanding

Question 1: Why don't we stack linear layers without activations?

A) It's too slow
B) **Multiple linear transformations collapse to one linear transformation**
C) It causes vanishing gradients
D) Backprop doesn't work

Knowledge Check: Deep Learning

Test your understanding

Question 1: Why don't we stack linear layers without activations?

A) It's too slow
B) **Multiple linear transformations collapse to one linear transformation**
C) It causes vanishing gradients
D) Backprop doesn't work

Explanation

Answer: B - Without non-linear activations, stacking layers is pointless: $f(W_2(W_1x)) = W_2W_1x = Wx$ is still just a linear function. Non-linearity is essential for learning complex patterns. This is why we use ReLU, sigmoid, tanh, etc. between layers.

Knowledge Check: Deep Learning

Test your understanding

Question 1: Why don't we stack linear layers without activations?

A) It's too slow
B) **Multiple linear transformations collapse to one linear transformation**
C) It causes vanishing gradients
D) Backprop doesn't work

Explanation

Answer: B - Without non-linear activations, stacking layers is pointless: $f(W_2(W_1x)) = W_2W_1x = Wx$ is still just a linear function. Non-linearity is essential for learning complex patterns. This is why we use ReLU, sigmoid, tanh, etc. between layers.

Question 2: What problem do ResNets solve with skip connections?

A) Overfitting
B) **Vanishing gradients in very deep networks**
C) Slow training

Knowledge Check: Deep Learning

Test your understanding

Question 1: Why don't we stack linear layers without activations?

A) It's too slow
B) **Multiple linear transformations collapse to one linear transformation**
C) It causes vanishing gradients
D) Backprop doesn't work

Explanation

Answer: B - Without non-linear activations, stacking layers is pointless: $f(W_2(W_1x)) = W_2W_1x = Wx$ is still just a linear function. Non-linearity is essential for learning complex patterns. This is why we use ReLU, sigmoid, tanh, etc. between layers.

Question 2: What problem do ResNets solve with skip connections?

A) Overfitting
B) **Vanishing gradients in very deep networks**
C) Slow training

Summary & Best Practices

Key takeaways for deep learning

Architecture Choices:

- **Images:** CNNs (ResNet, EfficientNet) or Vision Transformers
- **Sequences:** Transformers (BERT, GPT) or LSTMs for small data
- **Tabular:** Often better with XGBoost/LightGBM
- **Time series:** LSTMs, Transformers, or specialized (N-BEATS)

Training Best Practices:

1. Start simple, add complexity

Common Pitfalls:

- × Not checking for bugs (use small dataset first)
- × Wrong loss function
- × Learning rate too high/low
- × Not shuffling training data
- × Using test set for validation
- × Forgetting to normalize inputs
- × No baseline comparison

Resources:

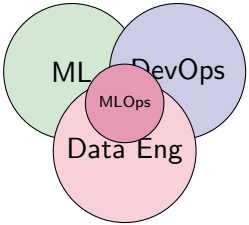
- **Courses:**
  - Andrew Ng: Deep Learning Specialization

The ML Production Gap

Why 87% of ML projects never make it to production

What is MLOps?

ML + DevOps + Data Engineering



Research vs. Production:

Research/Notebook	Production
Static dataset	Live, changing data
Batch processing	Real-time inference
Single model version	Multiple versions
Local machine	Distributed systems
Manual execution	Automated pipelines
No monitoring	24/7 monitoring
Reproduce once	Reproduce always

Challenges in Production:

1. Data drift: Input distribution changes
2. Concept drift: Relationship changes

MLOps Goals:

- **Automation**: Reduce manual work
- **Reproducibility**: Same inputs $\rightarrow$ same outputs

The ML Lifecycle

From data to deployment and back
Traditional Software:

$$\text{Code} \rightarrow \text{Build} \rightarrow \text{Deploy}$$

Machine Learning:

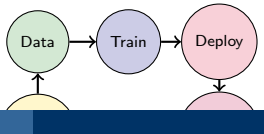
$$\text{Code} + \text{Data} + \text{Config} \rightarrow \text{Model} \rightarrow \text{Deploy}$$

More complex! Models are artifacts built from code AND data.

Complete ML Lifecycle:

1. **Data Collection**
   - Gather raw data
   - Set up pipelines
5. **Model Deployment**
   - Serve predictions
   - A/B testing
6. **Monitoring**
   - Track performance
   - Detect drift
7. **Retraining**
   - Update with new data
   - Close the loop

It's a Cycle, Not Linear!



Experiment Tracking

Managing the chaos of ML experiments

The Problem:

- Tried 50 experiments
- Which hyperparameters worked best?
- Can't reproduce last week's results
- Lost track of what changed

What to Track:

1. **Code:**
   - Git commit hash

MLflow Example:

```
import mlflow

mlflow.set_experiment("churn_prediction

with mlflow.start_run():
    # Log parameters
    mlflow.log_param("n_estimators", 10
    mlflow.log_param("max_depth", 10)

    # Train model
    model = RandomForestClassifier(
        n_estimators=100,
        max_depth=10
```

Model and Data Versioning

Git for machine learning assets

Why Version Everything?

- **Reproducibility**: Recreate any experiment
- **Debugging**: What changed between v1 and v2?
- **Rollback**: Revert to previous version
- **Compliance**: Audit trail for regulations
- **Collaboration**: Team knows what's happening

Data Version Control (DVC):

DVC Benefits:

- Data stored in cloud (S3, GCS, Azure)
- Git tracks only metadata
- Lightweight repositories
- Easy collaboration
- Pipeline versioning

ML Pipeline with DVC:

```
# dvc.yaml
stages:
  preprocess:
    cmd: python preprocess.py
    deps:
      - data/raw
```

Deployment Patterns

How to serve model predictions

1. Batch Prediction:

- Process data in batches (hourly, daily)
- Generate predictions offline
- Store in database
- Serve pre-computed predictions

Use when:

✓ Don't need real-time

✓ Can predict in advance

✓ Large-scale scoring

Flask API Example:

```
from flask import Flask, request, jsoni
import joblib

app = Flask(__name__)
model = joblib.load('model.pkl')

@app.route('/predict', methods=['POST']
def predict():
    data = request.get_json()
    features = data['features']

    prediction = model.predict([feature
    probability = model.predict_proba([
```

Containerization with Docker

Package model + dependencies + code together

Why Docker?

- **"Works on my machine"** → Works everywhere!
- Reproducible environments
- Easy deployment
- Isolated dependencies
- Scalable (Kubernetes)

Dockerfile Example:

```
FROM python:3.9-slim

# Set working directory
WORKDIR /app
```

Build and Run:

```
# Build image
docker build -t my-ml-model:v1 .

# Run container
docker run -p 5000:5000 my-ml-model:v1

# Test
curl -X POST http://localhost:5000/pred
  -H "Content-Type: application/json" \
  -d '{"features": [1.0, 2.0, 3.0]}'
```

Docker Compose (Multi-Container):

```
# docker-compose.yml
```

Model Monitoring in Production

What to monitor and why

Why Monitor?

Models degrade over time!

- Data distribution shifts
- Concept drift (relationships change)
- Bugs in production pipeline
- Adversarial attacks

What to Monitor:

1. **Data Quality**
   - Missing values
   - Out-of-range values
   - Schema changes
2. **Data Drift**
5. **System Metrics**
   - Latency (p50, p95, p99)
   - Throughput (requests/sec)
   - Error rate
   - Resource usage (CPU, memory)

Example: Data Drift Detection

```
from evidently.dashboard import Dashboa
from evidently.tabs import DataDriftTab

dashboard = Dashboard(
    tabs=[DataDriftTab()]
)
```

CI/CD for Machine Learning

Continuous Integration and Deployment

Traditional CI/CD:

1. Code commit → Git
2. Run tests
3. Build artifact
4. Deploy to production

ML CI/CD Additions:

- **CT:** Continuous Training
- **CD:** Continuous Deployment
- **CM:** Continuous Monitoring

GitHub Actions Example:

```
# .github/workflows/ml-pipeline.yml
deploy:
  needs: train
  if: github.ref == 'refs/heads/main'
  runs-on: ubuntu-latest
  steps:
    - name: Build Docker image
      run: docker build -t model:${{
          github.sha }} .

    - name: Push to registry
```

Knowledge Check: MLOps

Test your understanding

Question 1: Your model was 92% accurate in development but 78% in production. What's likely the issue?

A) The model is overfitted
B) Wrong metrics used
C) **Data drift - production data differs from training data**
D) Need more training data

Knowledge Check: MLOps

Test your understanding

Question 1: Your model was 92% accurate in development but 78% in production. What's likely the issue?

A) The model is overfitted
B) Wrong metrics used
C) **Data drift - production data differs from training data**
D) Need more training data

Explanation

Answer: C - Large accuracy drop in production usually indicates data drift. The model was trained on one distribution but is seeing different data in production. Could be covariate shift (features changed), concept drift (relationships changed), or data quality issues. Monitor input distributions and retrain with recent data.

Knowledge Check: MLOps

Test your understanding

Question 1: Your model was 92% accurate in development but 78% in production. What's likely the issue?

A) The model is overfitted
B) Wrong metrics used
C) **Data drift - production data differs from training data**
D) Need more training data

Explanation

Answer: C - Large accuracy drop in production usually indicates data drift. The model was trained on one distribution but is seeing different data in production. Could be covariate shift (features changed), concept drift (relationships changed), or data quality issues. Monitor input distributions and retrain with recent data.

Question 2: For real-time fraud detection (need ¡100ms latency), which deployment pattern?

A) Batch prediction (daily)

Knowledge Check: MLOps

Test your understanding

Question 1: Your model was 92% accurate in development but 78% in production. What's likely the issue?

A) The model is overfitted
B) Wrong metrics used
C) **Data drift - production data differs from training data**
D) Need more training data

Explanation

Answer: C - Large accuracy drop in production usually indicates data drift. The model was trained on one distribution but is seeing different data in production. Could be covariate shift (features changed), concept drift (relationships changed), or data quality issues. Monitor input distributions and retrain with recent data.

Question 2: For real-time fraud detection (need ¡100ms latency), which deployment pattern?

A) Batch prediction (daily)

Summary & Best Practices

Building production ML systems
The MLOps Checklist:

1. **Version everything**
   - Code (Git)
   - Data (DVC)
   - Models (MLflow)
   - Environment (Docker)
2. **Automate pipelines**
   - Data preprocessing
   - Model training
   - Evaluation
   - Deployment
3. **Test rigorously**
   - Unit tests
   - Integration tests
   - Data validation
   - Model performance

Common Pitfalls:

- × Not monitoring after deployment
- × No rollback plan
- × Training-serving skew
- × No data validation
- × Ignoring latency requirements
- × Not documenting decisions

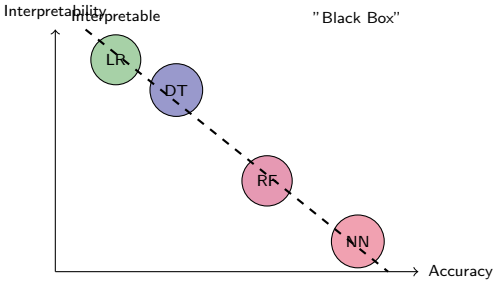
Tools Ecosystem:

Category	Tools
Experiment Tracking	MLflow, W&B
Versioning	DVC, Git LFS
Model Serving	TensorFlow Serving, TorchServe
Containerization	Docker, Kubernetes

Why Interpretability Matters

The black box problem in machine learning

The Accuracy-Interpretability Tradeoff:



Traditional View:

- Simple models = interpretable but less accurate
- Complex models = accurate but black box

Why We Need Interpretability:

1. **Trust & Adoption**
   - Doctors won't use unexplainable diagnosis
   - Banks must explain loan decisions
2. **Debugging & Improvement**
   - Identify when model fails
   - Find data leakage
   - Improve features
3. **Regulatory Compliance**
   - GDPR "right to explanation"
   - Fair lending laws
   - Medical device approval
4. **Fairness & Ethics**
   - Detect bias

Types of Interpretability

Taxonomy of explanation methods
Intrinsic vs. Post-hoc:

Intrinsic	Post-hoc
Model is inherently interpretable	Apply to any model after training
Linear regression, decision trees	SHAP, LIME
Limited complexity	Can explain black boxes

Model-Specific vs. Model-Agnostic:

Model-Specific	Model-Agnostic
Uses model internals	Treats model as black box
Attention weights, neuron activations	LIME, PDP
More accurate	Works for any model

Global vs. Local:

- **Global:** How model works overall
  - Feature importance

Levels of Interpretability:

1. **Algorithm Transparency**
   - How does the algorithm work?
   - Mathematical properties
2. **Global Model Interpretability**
   - What did the model learn overall?
   - Feature importances
   - Decision rules
3. **Local Interpretability**
   - Why this specific prediction?
   - Which features mattered for this instance?

The Interpretability Toolbox:

Method	Scope	Type

Intrinsically Interpretable Models

Models we can directly understand

Linear Regression:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

Interpretation:

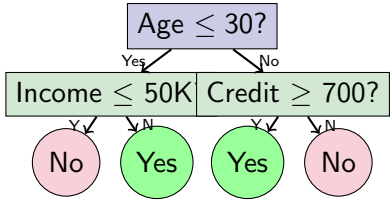
- β_j: Change in y for 1-unit change in x_j
- Sign: positive/negative relationship
- Magnitude: strength of effect

Caveats:

- Assumes linearity
- Correlation $\neq$ causation
- Multicollinearity issues

Logistic Regression:

Decision Trees:



Interpretation:

- Clear if-then rules
- Easy to visualize
- Mimics human decision-making

Limitations:

- High variance (unstable)

SHAP: Unified Framework for Interpretability

Game theory meets machine learning

The Core Idea:

How much does each feature contribute to prediction?

Based on Shapley Values from Game Theory:

Consider prediction as "coalition game":

- Players = features
- Payoff = prediction
- Fair distribution of credit

SHAP Value for Feature i:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{i\}) - f(S)]$$

SHAP in Practice:

```
import shap

# Train model
model = XGBClassifier()
model.fit(X_train, y_train)

# Create explainer
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_t

# Visualize
shap.summary_plot(shap_values, X_test)
shap.force_plot(explainer.expected_valu
```

LIME: Local Interpretable Model-Agnostic Explanations

Explain any model locally with simple approximations

The LIME Approach:

Key Idea: Approximate complex model locally with interpretable model

Algorithm:

1. Select instance to explain
2. Generate perturbed samples nearby
3. Get predictions from black box
4. Train simple model (linear/tree) on perturbed data
5. Use simple model to explain

LIME Implementation:

```
from lime import lime_tabular

# Create explainer
explainer = lime_tabular.
    LimeTabularExplainer(
        X_train,
        feature_names=feature_names,
        class_names=['No', 'Yes'],
        mode='classification'
    )

# Explain instance
explanation = explainer.explain_instanc
```

Feature Importance Methods

Understanding what the model learned globally

1. Permutation Importance:

Shuffle feature j, measure performance drop:

```
from sklearn.inspection import
    permutation_importance

result = permutation_importance(
    model, X_test, y_test,
    n_repeats=10
)
```

Advantages:

- Model-agnostic
- Reliable
- Accounts for interactions

4. Partial Dependence Plots (PDP):

Show marginal effect of feature:

$$\hat{f}_S(x_S) = \mathbb{E}_{x_C}[\hat{f}(x_S, x_C)]$$

```
from sklearn.inspection import
    PartialDependenceDisplay

PartialDependenceDisplay.from_estimator
    model, X, features=[0, 1, (0,1)]
)
```

Shows:

- Feature effect on average

Counterfactual Explanations

"What would need to change for a different outcome?"

The Counterfactual Question:
"Why was my loan denied?"
Bad Answer: "Credit score was too low (SHAP value = -0.3)"
Better Answer (Counterfactual): "If your credit score were 680 instead of 620, you would be approved"

Properties of Good Counterfactuals:

1. **Proximity:** Close to original instance

Methods:

1. **Optimization-Based:**
   - Minimize distance subject to prediction constraint
   - Can use genetic algorithms
2. **DiCE** (Diverse Counterfactual Explanations):
   - Generate multiple diverse counterfactuals
   - Gives user options

Implementation:

```
import dice_ml

# Create DiCE explainer
dice = dice_ml.Dice(data, model)
```

Knowledge Check: Explainable AI

Test your understanding

Question 1: What is the key difference between SHAP and LIME?

A) SHAP is for trees only, LIME is model-agnostic
B) **SHAP has theoretical guarantees (Shapley values), LIME uses local approximation**
C) LIME is faster
D) They are the same

Knowledge Check: Explainable AI

Test your understanding

Question 1: What is the key difference between SHAP and LIME?

A) SHAP is for trees only, LIME is model-agnostic
B) **SHAP has theoretical guarantees (Shapley values), LIME uses local approximation**
C) LIME is faster
D) They are the same

Explanation

Answer: B - SHAP is based on game-theoretic Shapley values with mathematical guarantees (local accuracy, consistency). LIME approximates the model locally with a simpler model, which is fast but doesn't have the same theoretical properties. Both are model-agnostic, but SHAP provides more consistent explanations.

Knowledge Check: Explainable AI

Test your understanding

Question 1: What is the key difference between SHAP and LIME?

A) SHAP is for trees only, LIME is model-agnostic
B) **SHAP has theoretical guarantees (Shapley values), LIME uses local approximation**
C) LIME is faster
D) They are the same

Explanation

Answer: B - SHAP is based on game-theoretic Shapley values with mathematical guarantees (local accuracy, consistency). LIME approximates the model locally with a simpler model, which is fast but doesn't have the same theoretical properties. Both are model-agnostic, but SHAP provides more consistent explanations.

Question 2: When would you use counterfactual explanations over feature importance?

A) When you need global model understanding
B) **When users need actionable advice on how to change outcomes**

Knowledge Check: Explainable AI

Test your understanding

Question 1: What is the key difference between SHAP and LIME?

A) SHAP is for trees only, LIME is model-agnostic
B) **SHAP has theoretical guarantees (Shapley values), LIME uses local approximation**
C) LIME is faster
D) They are the same

Explanation

Answer: B - SHAP is based on game-theoretic Shapley values with mathematical guarantees (local accuracy, consistency). LIME approximates the model locally with a simpler model, which is fast but doesn't have the same theoretical properties. Both are model-agnostic, but SHAP provides more consistent explanations.

Question 2: When would you use counterfactual explanations over feature importance?

A) When you need global model understanding
B) **When users need actionable advice on how to change outcomes**

Summary & Best Practices

Building interpretable ML systems

The Interpretability Workflow:

1. **Model Selection:**
   - Start with interpretable baselines
   - Document accuracy-interpretability tradeoff
   - Justify complexity if needed
2. **Global Understanding:**
   - Permutation importance
   - Partial dependence plots
   - SHAP summary plots
3. **Local Explanations:**
   - SHAP force plots for key predictions

Common Mistakes:

- × Trusting explanations blindly
- × Using only one method
- × Ignoring feature correlations
- × Not validating with experts
- × Over-interpreting complex models

Tools & Resources:

- **Python Libraries:**
  - SHAP, LIME, ELI5
  - InterpretML (Microsoft)
  - AIX360 (IBM)
  - Captum (PyTorch)
- **Books:**

What is Time Series Data?

Data ordered by time with temporal dependencies

Definition:

A time series is a sequence of observations:

$$\{y_t : t = 1, 2, \ldots, T\}$$

where y_t is the value at time t.

Key Characteristic:

- Temporal ordering matters!
- Observations are *not* independent
- Past values influence future values
- Cannot shuffle observations

Examples:

- Stock prices (daily)

Time Series vs. Cross-Sectional:

Time Series	Cross-Sectional
Ordered by time	No time ordering
Temporal dependence	Independent obs
Cannot shuffle	Can shuffle
Train-test split: by time	Random split OK
Example: Daily stock price of AAPL	Example: House prices in a city

Types of Time Series:

1. **Univariate:** Single variable over time
   - Daily temperature
2. **Multivariate:** Multiple variables over time
   - Temperature, humidity, pressure

Time Series Components

Decomposing time series into interpretable parts

Additive Decomposition:

$$y_t = T_t + S_t + C_t + I_t$$

where:

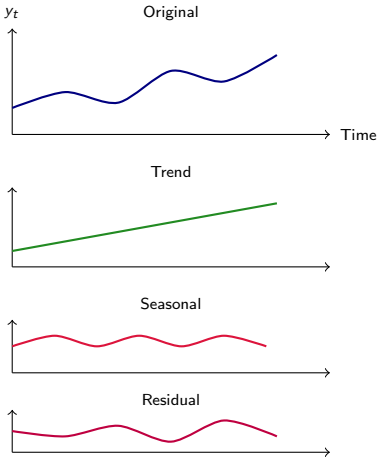
- T_t: **Trend** - Long-term direction
- S_t: **Seasonal** - Regular pattern
- C_t: **Cyclical** - Non-periodic fluctuations
- I_t: **Irregular** - Random noise

Multiplicative Decomposition:

$$y_t = T_t \times S_t \times C_t \times I_t$$

Use when seasonal variation increases with

Example: Airline Passengers



Stationarity: The Foundation

Why stationarity matters for time series modeling

Definition:
A time series is **strictly stationary** if:

$$P(y_{t_1}, \ldots, y_{t_k}) = P(y_{t_1+h}, \ldots, y_{t_k+h})$$

for all $t_1, \ldots, t_k$ and h.

Weak (Covariance) Stationarity:

1. $\mathbb{E}[y_t] = \mu$ (constant mean)
2. $\text{Var}[y_t] = \sigma^2$ (constant variance)
3. $\text{Cov}[y_t, y_{t-k}] = \gamma_k$ (depends only on lag k)

Testing for Stationarity:

Augmented Dickey-Fuller (ADF) Test

Tests null hypothesis of unit root:

$$H_0 : \text{Non-stationary (unit root)}$$

$$H_1 : \text{Stationary}$$

Reject H_0 if p-value < 0.05

Making Series Stationary:

1. **Differencing:**

$$\Delta y_t = y_t - y_{t-1}$$

ARIMA Models

AutoRegressive Integrated Moving Average
ARIMA(p, d, q) Components:

1. **AR(p):** AutoRegressive

$$y_t = \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} + \epsilon_t$$

   Uses past values

2. **I(d):** Integrated (Differencing)

$$\Delta^d y_t$$

   Makes stationary

3. **MA(q):** Moving Average

$$y_t = \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}$$

Model Selection:

Tool	Purpose
ACF	Autocorrelation function - identifies MA(q)
PACF	Partial autocorrelation - identifies AR(p)
AIC/BIC	Information criteria - compare models
Auto-ARIMA	Automated search over (p,d,q) space

Seasonal ARIMA:

SARIMA(p,d,q)(P,D,Q)$_s$

Adds seasonal components with period s:

- (P, D, Q): Seasonal AR, I, MA
- s: Seasonal period (12 for monthly)

Python Implementation:

```
from statsmodels.tsa.arima.model import
```

Knowledge Check: Time Series Fundamentals

Multiple Choice Questions

Question 1: You have daily sales data for a retail store. Which decomposition would you likely use?

A) Additive, because sales don't vary
B) **Multiplicative, because seasonal variation grows with sales level**
C) Neither, sales data doesn't have seasonality
D) Use differencing instead

Knowledge Check: Time Series Fundamentals

Multiple Choice Questions

Question 1: You have daily sales data for a retail store. Which decomposition would you likely use?

A) Additive, because sales don't vary
B) **Multiplicative, because seasonal variation grows with sales level**
C) Neither, sales data doesn't have seasonality
D) Use differencing instead

Explanation

Answer: B - Retail sales typically show seasonal patterns (holidays, weekends) that grow proportionally with the overall sales level. For example, a 20% Black Friday boost means $200 extra when baseline is $1000, but $2000 extra when baseline is $10,000. This multiplicative relationship requires multiplicative decomposition.

Knowledge Check: Time Series Fundamentals

Multiple Choice Questions

Question 1: You have daily sales data for a retail store. Which decomposition would you likely use?

A) Additive, because sales don't vary
B) **Multiplicative, because seasonal variation grows with sales level**
C) Neither, sales data doesn't have seasonality
D) Use differencing instead

Explanation

Answer: B - Retail sales typically show seasonal patterns (holidays, weekends) that grow proportionally with the overall sales level. For example, a 20% Black Friday boost means $200 extra when baseline is $1000, but $2000 extra when baseline is $10,000. This multiplicative relationship requires multiplicative decomposition.

Question 2: ADF test gives p-value $= 0.23$. What should you do?

Knowledge Check: Time Series Fundamentals

Multiple Choice Questions

Question 1: You have daily sales data for a retail store. Which decomposition would you likely use?

A) Additive, because sales don't vary
B) **Multiplicative, because seasonal variation grows with sales level**
C) Neither, sales data doesn't have seasonality
D) Use differencing instead

Explanation

Answer: B - Retail sales typically show seasonal patterns (holidays, weekends) that grow proportionally with the overall sales level. For example, a 20% Black Friday boost means $200 extra when baseline is $1000, but $2000 extra when baseline is $10,000. This multiplicative relationship requires multiplicative decomposition.

Question 2: ADF test gives p-value $= 0.23$. What should you do?

Modern ML Methods for Time Series

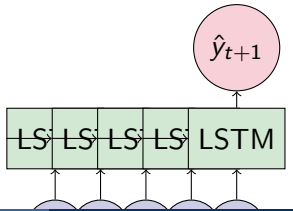
Beyond classical statistical approaches
Machine Learning Approaches:

1. **Feature Engineering + ML**
   - Lag features: $y_{t-1}, y_{t-2}, \ldots$
   - Rolling statistics: mean, std
   - Time features: day of week, month
   - Train XGBoost, Random Forest
2. **Prophet (Facebook)**
   - Additive model with trend + seasonality
   - Handles missing data, outliers
   - Easy to use, interpretable
3. **Neural Networks**
   - LSTM: Long Short-Term Memory
   - GRU: Gated Recurrent Unit
   - Transformer models
   - N-BEATS: Pure deep learning

When to Use What:

Method	Best For
ARIMA	Univariate, stationary, short-term
Exp Smoothing	Trends + seasonality, simple
Prophet	Business time series, holidays
XGBoost	Many features, non-linear
LSTM	Long sequences, complex patterns
Transformer	Multivariate, attention needed

Example: LSTM Architecture



Summary & Best Practices

Key takeaways for time series analysis

Core Principles:

1. **Temporal ordering matters**
   - Never shuffle time series data
   - Use time-based train/test splits
2. **Check stationarity**
   - ADF test before modeling
   - Difference if needed
3. **Understand your data**
   - Decompose into components
   - Identify seasonality, trend
   - Check for outliers, missing values
4. **Start simple, add complexity**

Common Pitfalls:

- × Using random train/test split
- × Ignoring non-stationarity
- × Data leakage from future
- × Over-differencing
- × Ignoring domain knowledge
- × Not handling seasonality
- × Using only RMSE (check MAE, MAPE too)

Resources:

- **Books:**
  - "Forecasting: Principles and Practice" (Hyndman & Athanasopoulos)