Northeastern University
Silicon Valley

DeepOilFlow
Interpretable LSTM for oil production forecasting

Tiago Monteiro

Master of Science in Artificial Intelligence

Overview

- Section 1: The history of the AI field
- Section 2: The problem and solution achieved
- Section 3: The path to the solution in plain English
- Section 4: The main ideas of the LSTM
- Section 5: An Interpretability approach: SHAP
- Section 6: Final results from this study
- Section 7: Next steps

You will learn:

- Basic history of AI
- PyTorch good practices based on a project
  - Create a custom data class and why it is useful
  - How to divide data without scikit-learn
  - Why __**init**__ and **forward** are very important in defining deep learning models
  - Why organizing layers in blocks is very important
  - **In all deep learning models: Clear, Diagnose, Update**
  - How to freeze gradients when testing the model on testing data
  - Main ideas of how LSTM neural networks work **without heavy math**
    - Overview of how it works with an analogy and in simple English with code
  - What is dropout and its importance in preventing overfitting
  - Example of SHAP

Pre-requisites

- **Basic Python programming**: Variables, functions, loops, and data structures
- **Basic understanding of ML concepts**: What training/testing data means, what models do
- **Basic awareness of time series data**: Data that changes over time

Will not go in detail:

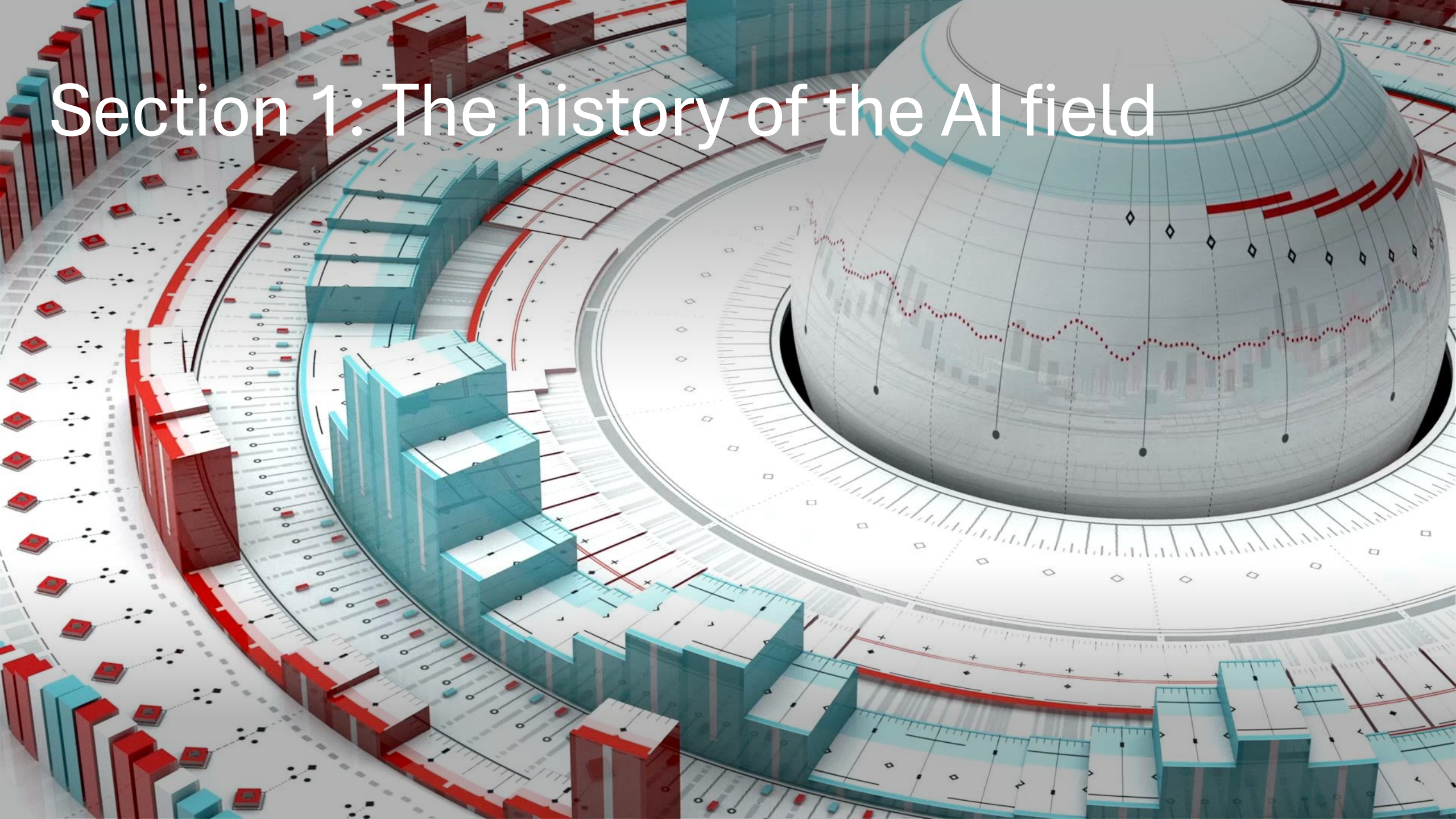
HEAVY MATH OF LSTM

MATH OF SHAP INTERPRETABILITY

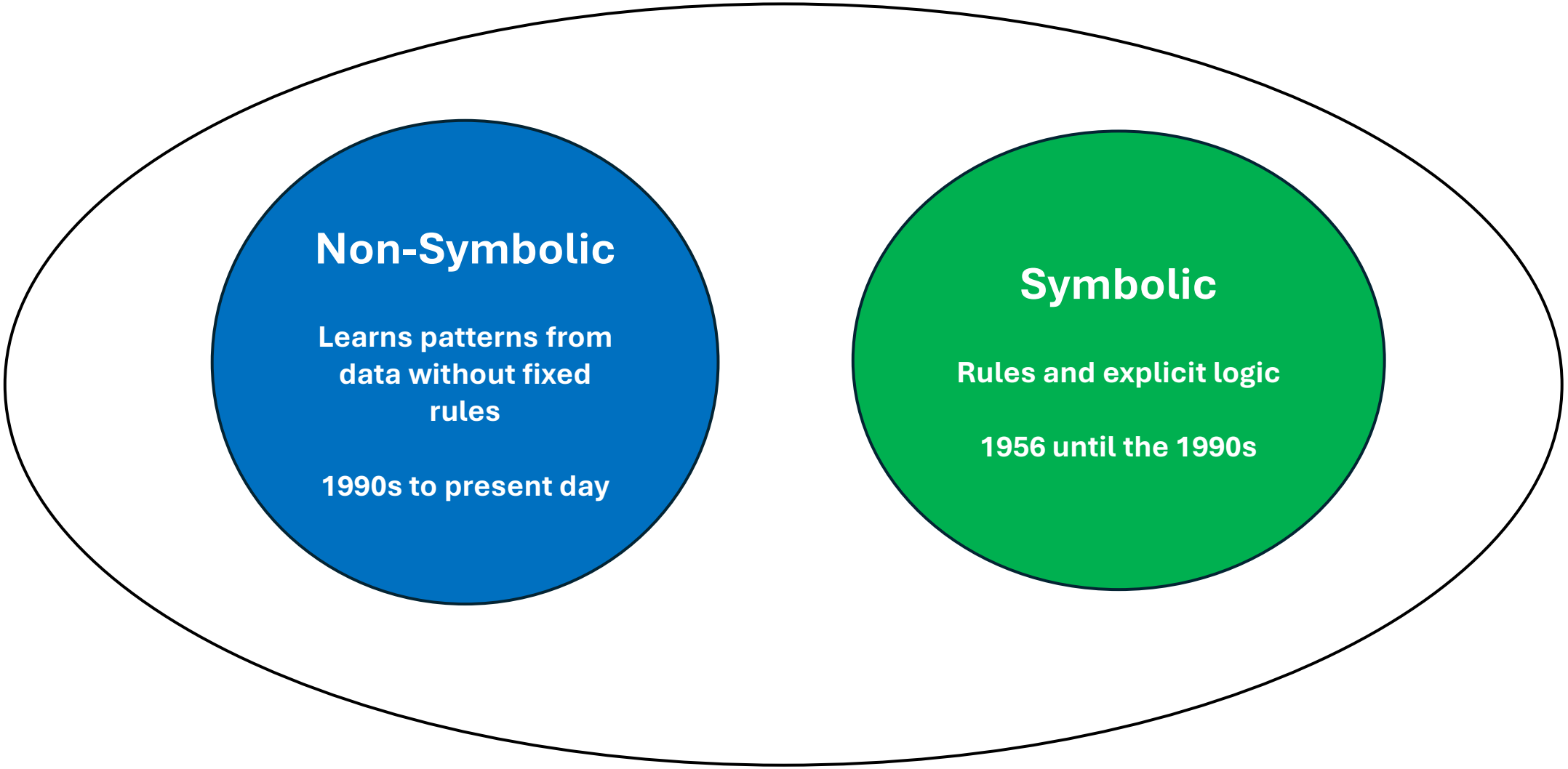
PYTORCH DYNAMIC COMPUTATIONAL GRAPH

AND OTHER ADVANCED CONCEPTS

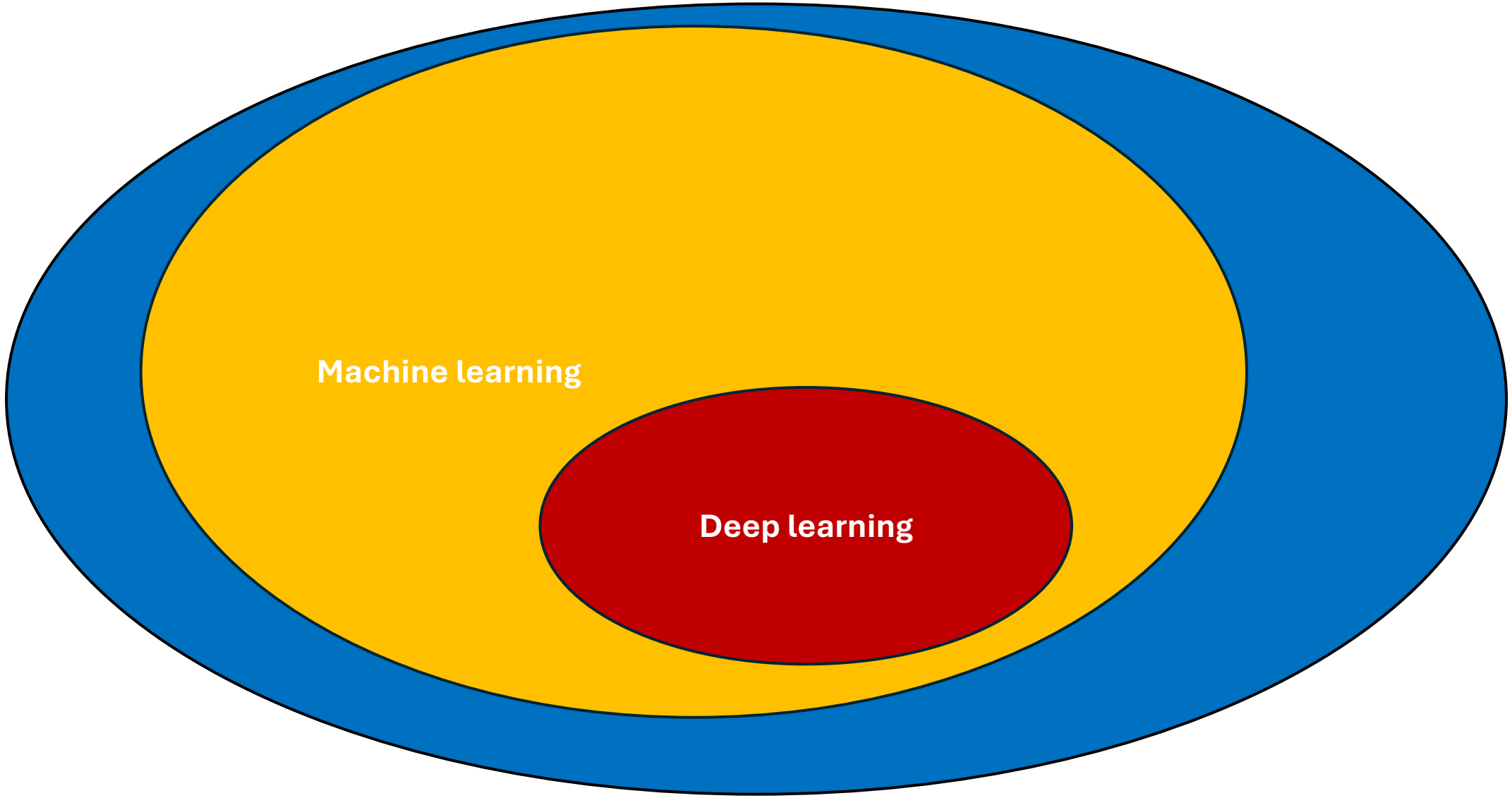
Goal: A clean, standard starting point for any NeuralAI deep learning project



Section 1: The history of the AI field

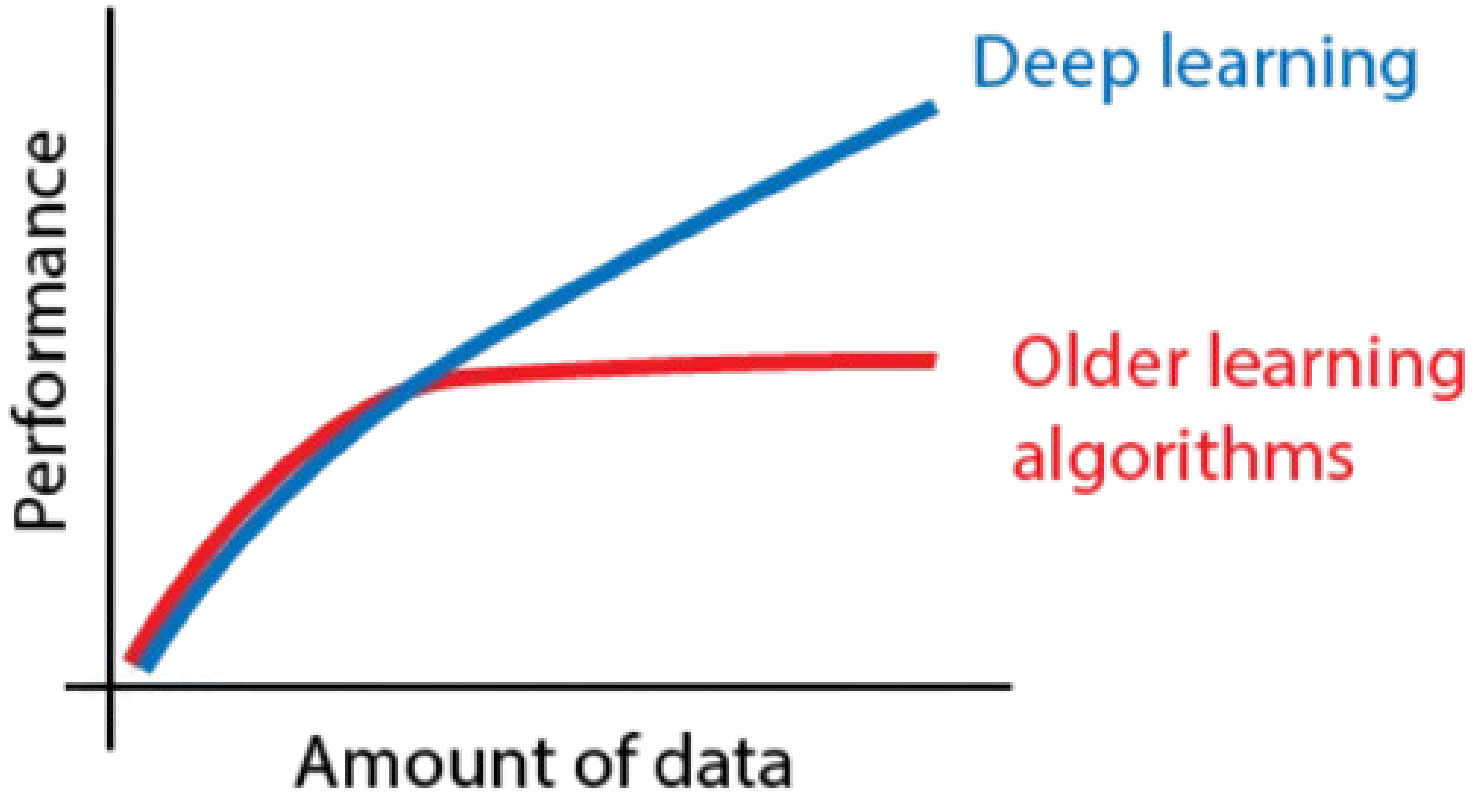


Artificial Intelligence



Non-Symbolic Artificial Intelligence

Why deep learning



Section 2: The problem and solution achieved

What is the problem and its economic value?

How to predict next month's oil production from 1 year of data?

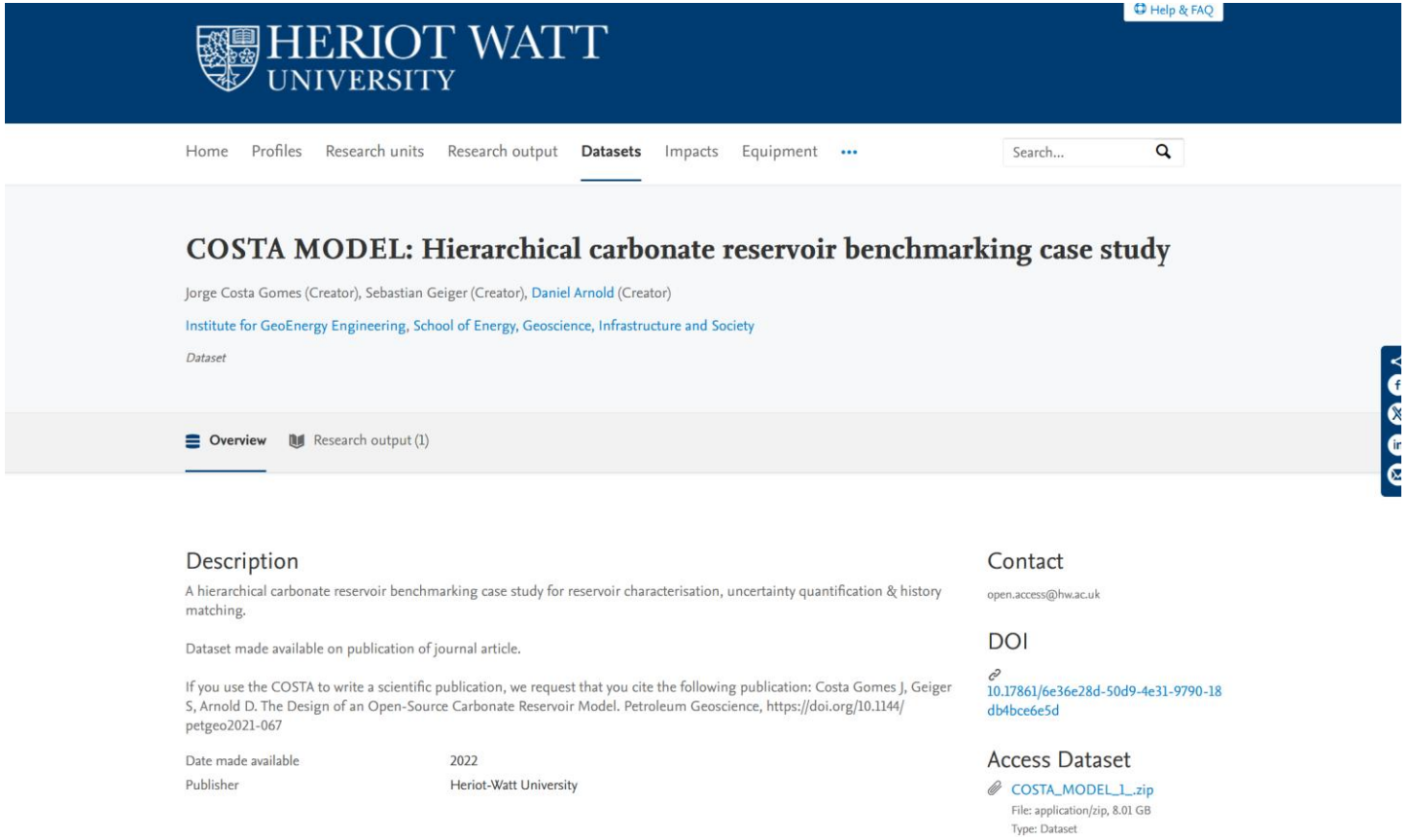
Better Production Planning

Reduced Operational Costs

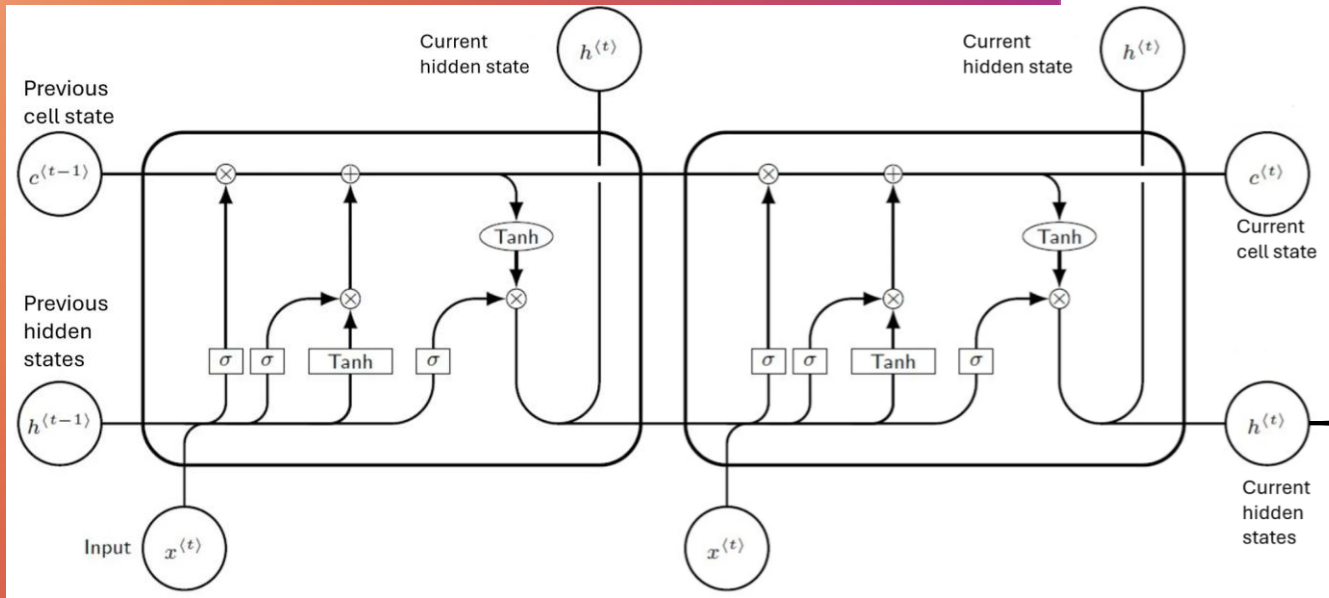
Data

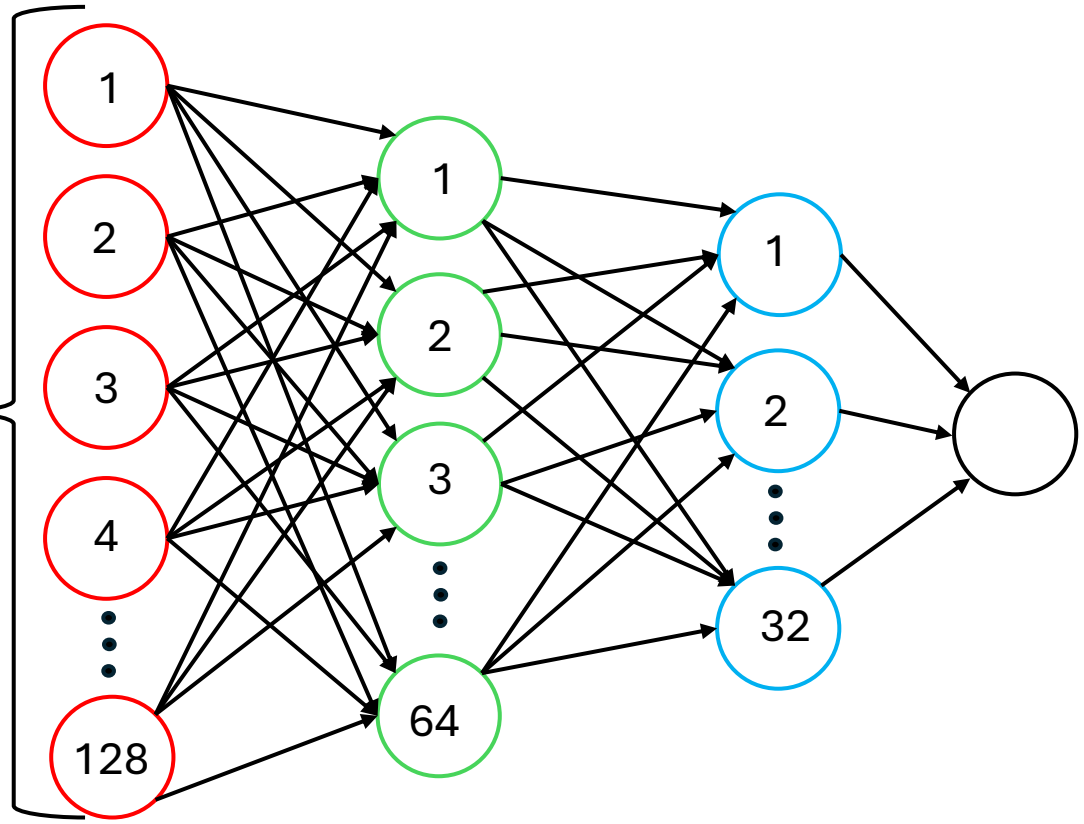
- 30 years of production data from 248 wells in the COSTA carbonate reservoir benchmark case

- For each prediction, it analyzes 12 months of historical data across 6 metrics:
  - Gas Oil Ratio
  - Gas Rate
  - Oil Rate
  - Water Cut
  - Water Rate
  - Well Bottom-hole Pressure



https://researchportal.hw.ac.uk/en/datasets/costa-model-hierarchical-carbonate-reservoir-benchmarking-case-st/





AI Solution

- An LSTM (Long Short-Term Memory) neural network connected to a feed forward layer
  - 2 LSTM cells
  - 3 feedforward nn layers
  - 212 097 parameters

Section 3: The path to the solution in plain English

PyTorch

```
import json
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shap
import torch
import torch.nn as nn
from IPython.display import HTML, display
from torch.utils.data import DataLoader, Dataset, random_split
```

Data

	Well	Target_Oil_Rate	Month_1_Date	Month_1_Gas Oil Ratio SC (ft^3/bbl)	Month_1_Gas Rate SC - Monthly (ft^3/day)	Month_1_Oil Rate SC - Monthly (bbl/day)	Month_1_Water Cut SC - %	Month_1_Water Rate SC - Monthly (bbl/day)	Month_1_Well Bottom-hole Pressure (psi)	Month_2_Date	Mont Oil
0	Well-1	5023.651982	2025-02-01	731.414978	3.567828e+06	4877.981171	0.000046	0.002222	3132.787350	2025-03-01	731
1	Well-1	5156.666858	2025-03-01	731.414978	3.662807e+06	5007.836862	0.000064	0.003183	3090.725933	2025-04-01	731
2	Well-1	4984.891998	2025-04-01	731.414978	3.626329e+06	4957.963500	0.000050	0.002470	3059.375781	2025-05-01	731
3	Well-1	5166.799378	2025-05-01	731.414978	3.740978e+06	5114.712817	0.000048	0.002431	3039.715643	2025-06-01	731
4	Well-1	4940.163238	2025-06-01	731.414978	3.657319e+06	5000.333284	0.000051	0.002528	3015.697719	2025-07-01	731
5	Well-1	4872.770900	2025-07-01	731.414978	3.592993e+06	4912.386022	0.000103	0.005059	3003.678545	2025-08-01	731
6	Well-1	4925.642330	2025-08-01	731.414978	3.680401e+06	5031.892153	0.000070	0.003539	2986.926955	2025-09-01	731
7	Well-1	5019.080507	2025-09-01	731.414978	3.593829e+06	4913.529550	0.000067	0.003279	2968.007638	2025-10-01	731
8	Well-1	4922.541903	2025-10-01	731.414978	3.534131e+06	4831.909593	0.000108	0.005200	2947.988424	2025-11-01	731
9	Well-1	5139.027491	2025-11-01	731.414978	3.603630e+06	4926.930069	0.000124	0.006090	2923.813792	2025-12-01	731
10	Well-1	5138.429237	2025-12-01	731.414978	3.648792e+06	4988.675921	0.000102	0.005107	2913.636242	2026-01-01	731
11	Well-1	4998.176506	2026-01-01	731.414978	3.766142e+06	5149.117544	0.000092	0.004745	2903.087513	2026-02-01	731
12	Well-1	5056.668043	2026-02-01	731.414978	3.674374e+06	5023.651982	0.000107	0.005387	2876.896641	2026-03-01	731
13	Well-1	5018.697825	2026-03-01	731.414978	3.771663e+06	5156.666858	0.000126	0.006480	2862.643536	2026-04-01	731

Inherit dataset class to make a custom dataset class

```
class RegressionDataset(Dataset):
    def __init__(self, X, y, X_mean, X_std, y_mean, y_std):
        self.X = torch.FloatTensor(X.values if hasattr(X, 'values') else X)
        self.y = torch.FloatTensor(y.values if hasattr(y, 'values') else y)
        self.X_mean = torch.FloatTensor(X_mean)
        self.X_std = torch.FloatTensor(X_std)
        self.y_mean = torch.FloatTensor([y_mean])
        self.y_std = torch.FloatTensor([y_std])

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        #if self.X_std== 0, X normalized does not become Nan it becomes 1e-8
        X_normalized = (self.X[idx] - self.X_mean) / (self.X_std + 1e-8)
        y_normalized = (self.y[idx] - self.y_mean) / (self.y_std + 1e-8)
        return X_normalized, y_normalized

dataset = RegressionDataset(X, y, X_mean, X_std, y_mean, y_std)
```

```
import torch
import torch.nn as nn
from IPython.display import HTML, display
from torch.utils.data import DataLoader, Dataset, random_split
```

Inherit dataset class to make a custom dataset class

```python
class RegressionDataset(Dataset):
    def __init__(self, X, y, X_mean, X_std, y_mean, y_std):
        self.X = torch.FloatTensor(X.values if hasattr(X, 'values') else X)
        self.y = torch.FloatTensor(y.values if hasattr(y, 'values') else y)
        self.X_mean = torch.FloatTensor(X_mean)
        self.X_std = torch.FloatTensor(X_std)
        self.y_mean = torch.FloatTensor([y_mean])
        self.y_std = torch.FloatTensor([y_std])

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        #if self.X_std== 0, X normalized does not become Nan it becomes 1e-8
        X_normalized = (self.X[idx] - self.X_mean) / (self.X_std + 1e-8)
        y_normalized = (self.y[idx] - self.y_mean) / (self.y_std + 1e-8)
        return X_normalized, y_normalized

dataset = RegressionDataset(X, y, X_mean, X_std, y_mean, y_std)
```

```python
import torch
import torch.nn as nn
from IPython.display import HTML, display
from torch.utils.data import DataLoader, Dataset, random_split
```

Spliting dataset into training, developement(validation) and testing

```
total_size = len(dataset)
train_size = int(0.6 * total_size)
dev_size = int(0.2 * total_size)
test_size = total_size - train_size - dev_size

train_dataset, dev_dataset, test_dataset = random_split(
    dataset,
    [train_size, dev_size, test_size],
    generator=torch.Generator().manual_seed(42)  # for reproducibility
)

print(f"Train size: {len(train_dataset)}, Dev size: {len(dev_dataset)}, Test size: {len(test_dataset)}")
```

```
Train size: 26935, Dev size: 8978, Test size: 8979
```

Create the dataLoaders

```
import torch
import torch.nn as nn
from IPython.display import HTML, display
from torch.utils.data import DataLoader, Dataset, random_split
```

DataLoaders automatically batch, shuffle, and load data in parallel.

This way, making it easy to feed data to the model during training.

```
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Spliting dataset into training, developement(validation) and testing

```
total_size = len(dataset)
train_size = int(0.6 * total_size)
dev_size = int(0.2 * total_size)
test_size = total_size - train_size - dev_size

train_dataset, dev_dataset, test_dataset = random_split(
    dataset,
    [train_size, dev_size, test_size],
    generator=torch.Generator().manual_seed(42)  # for reproducibility
)

print(f"Train size: {len(train_dataset)}, Dev size: {len(dev_dataset)}, Test size: {len(test_dataset)}")
```

```
Train size: 26935, Dev size: 8978, Test size: 8979
```

Create the dataLoaders

```
import torch
import torch.nn as nn
from IPython.display import HTML, display
from torch.utils.data import DataLoader, Dataset, random_split
```

DataLoaders automatically batch, shuffle, and load data in parallel.

This way, making it easy to feed data to the model during training.

```
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Spliting dataset into training, developement(validation) and testing

```
total_size = len(dataset)
train_size = int(0.6 * total_size)
dev_size = int(0.2 * total_size)
test_size = total_size - train_size - dev_size
```

Spliting dataset into training, developement(validation) and testing

```
total_size = len(dataset)
train_size = int(0.6 * total_size)
dev_size = int(0.2 * total_size)
test_size = total_size - train_size - dev_size

train_dataset, dev_dataset, test_dataset = random_split(
    dataset,
    [train_size, dev_size, test_size],
    generator=torch.Generator().manual_seed(42)  # for reproducibility
)

print(f"Train size: {len(train_dataset)}, Dev size: {len(dev_dataset)}, Test size: {len(test_dataset)}")
```

```
Train size: 26935, Dev size: 8978, Test size: 8979
```

Architecture of LSTM model

•[16]:

```python
class LSTM_model(nn.Module):
    def __init__(self, input_size=6, hidden_size=128, num_layers=2, dropout=0.2, seq_length=12):
        super(LSTM_model, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.seq_length = seq_length

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout
        )

        self.fully_connected_block = nn.Sequential(
            nn.Linear(hidden_size, 64),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        batch_size = x.size(0)
        x = x.view(batch_size, self.seq_length, -1)  # (batch_size, 12, 6)

        h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)

        # LSTM forward pass
        lstm_out, (h_n, c_n) = self.lstm(x)

        # Use the last hidden state
        out = lstm_out[:, -1, :]  # (batch_size, hidden_size)

        # Fully connected layers
        out = self.fully_connected_block(out)

        return out
```

We will see after the code overview

LSTMs in depth in plain english

```python
def __init__(self, input_size=6, hidden_size=128, num_layers=2, dropout=0.2, seq_length=12):
    super(LSTM_model, self).__init__()

    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.seq_length = seq_length

    self.lstm = nn.LSTM(
        input_size=input_size,
        hidden_size=hidden_size,
        num_layers=num_layers,
        batch_first=True,
        dropout=dropout
    )

    self.fully_connected_block = nn.Sequential(
        nn.Linear(hidden_size, 64),
        nn.ReLU(),
        nn.Dropout(dropout),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 1)
    )
```

What is the Architecture?

```python
def forward(self, x):
    batch_size = x.size(0)
    x = x.view(batch_size, self.seq_length, -1)  # (batch_size, 12, 6)

    h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)
    c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)

    # LSTM forward pass
    lstm_out, (h_n, c_n) = self.lstm(x)

    # Use the last hidden state
    out = lstm_out[:, -1, :]  # (batch_size, hidden_size)

    # Fully connected layers
    out = self.fully_connected_block(out)

    return out
```

How data flows in the architecture?

IMPORTANT

```
self.fully_connected_block = nn.Sequential(
    nn.Linear(hidden_size, 64),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

```
out = self.fully_connected_block(out)
```

Good practice

```python
class LSTM_model(nn.Module):
    def __init__(self, input_size=6, hidden_size=128, num_layers=2, dropout=0.2, seq_length=12):
        super(LSTM_model, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.seq_length = seq_length

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout
        )

        self.fully_connected_block = nn.Sequential(
            nn.Linear(hidden_size, 64),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        batch_size = x.size(0)
        x = x.view(batch_size, self.seq_length, -1)  # (batch_size, 12, 6)

        h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size)

        # LSTM forward pass
        lstm_out, (h_n, c_n) = self.lstm(x)

        # Use the last hidden state
        out = lstm_out[:, -1, :]  # (batch_size, hidden_size)

        # Fully connected layers
        out = self.fully_connected_block(out)

        return out
```

```
        self.fully_connected_block = nn.Sequential(
            nn.Linear(hidden_size, 64),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )
```

```
        # Fully connected layers
        out = self.fully_connected_block(out)
```

Other good practices

Model architecture and total parameter count

```
print(model)
print(f"Total parameters: {sum(p.numel() for p in model.parameters())}")
```

```
LSTM_model(
  (lstm): LSTM(6, 128, num_layers=2, batch_first=True, dropout=0.2)
  (fully_connected_block): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=64, out_features=32, bias=True)
    (4): ReLU()
    (5): Linear(in_features=32, out_features=1, bias=True)
  )
)
Total parameters: 212097
```

Transformers have a lot of blocks

Direct children modules

```
for name, module in model.named_children():
    print(f"{name}: {module}")
```

```
lstm: LSTM(6, 128, num_layers=2, batch_first=True, dropout=0.2)
fully_connected_block: Sequential(
  (0): Linear(in_features=128, out_features=64, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.2, inplace=False)
  (3): Linear(in_features=64, out_features=32, bias=True)
  (4): ReLU()
  (5): Linear(in_features=32, out_features=1, bias=True)
)
```

```python
hyperparameters = {
    'learning_rate': 0.00001,
    'batch_size': 64,
    'num_epochs': 15,
    'weight_decay': 1e-5,
    'patience': 5,
    'min_delta': 1e-4
}

print("\nHyperparameters:")
for key, value in hyperparameters.items():
    print(f"  {key}: {value}")

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(
    model.parameters(),
    lr=hyperparameters['learning_rate'],
    weight_decay=hyperparameters['weight_decay']
)
```

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

https://arxiv.org/abs/1412.6980

```python
def train_one_epoch(model, train_loader, criterion, optimizer, device, train_dataset):
    """Train for one epoch and return average loss"""
    model.train()
    total_loss = 0.0

    for batch_X, batch_y in train_loader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)

        # Forward pass
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item() * batch_X.size(0)

    return total_loss / len(train_dataset)  # Using __len__ from the RegressionDataset class
```

```python
    total_loss = 0.0

    for batch_X, batch_y in train_loader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)

        # Forward pass
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item() * batch_X.size(0)

    return total_loss / len(train_dataset)  # Using __len__ from the RegressionDataset class
```

```
for batch_X, batch_y in train_loader:
    batch_X, batch_y = batch_X.to(device), batch_y.to(device)
```

Forward pass

```
# Forward pass
outputs = model(batch_X)
loss = criterion(outputs, batch_y)
```

Backward pass

```
# Backward pass
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

IMPORTANT

Clean, Diagnose, Update

```
# Backward pass
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Clean - Remove previous gradients

```
# Backward pass
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Diagnose - Calculate new gradients

```
# Backward pass
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Update - Apply gradients to parameters(weights and biases)

Training the LSTM model and saving results

```
train_losses, dev_losses, best_epoch, best_dev_loss = train_model(
    model, train_loader, dev_loader, criterion, optimizer, scheduler,
    hyperparameters, device, train_dataset, dev_dataset
)
```

```
Starting Training

Device: cpu
Training samples: 26935
Validation samples: 8978
Batches per epoch: 421
```

```
Epoch [  1/15] | Train Loss: 0.978192 | Dev Loss: 0.927809 | Patience: 0/5
> New best model saved | Improvement: inf

Epoch [  2/15] | Train Loss: 0.683201 | Dev Loss: 0.342159 | Patience: 0/5
> New best model saved | Improvement: 0.585649

Epoch [  3/15] | Train Loss: 0.213919 | Dev Loss: 0.117312 | Patience: 0/5
> New best model saved | Improvement: 0.224847

Epoch [  4/15] | Train Loss: 0.093523 | Dev Loss: 0.049060 | Patience: 0/5
> New best model saved | Improvement: 0.068252

Epoch [  5/15] | Train Loss: 0.050107 | Dev Loss: 0.027035 | Patience: 0/5
> New best model saved | Improvement: 0.022025
```

```
Epoch [  6/15] | Train Loss: 0.038273 | Dev Loss: 0.020775 | Patience: 0/5
> New best model saved | Improvement: 0.006260

Epoch [  7/15] | Train Loss: 0.033921 | Dev Loss: 0.018219 | Patience: 0/5
> New best model saved | Improvement: 0.002556

Epoch [  8/15] | Train Loss: 0.031545 | Dev Loss: 0.016771 | Patience: 0/5
> New best model saved | Improvement: 0.001448

Epoch [  9/15] | Train Loss: 0.030715 | Dev Loss: 0.016090 | Patience: 0/5
> New best model saved | Improvement: 0.000681

Epoch [ 10/15] | Train Loss: 0.029319 | Dev Loss: 0.014906 | Patience: 0/5
> New best model saved | Improvement: 0.001183
```

```
Epoch [ 11/15] | Train Loss: 0.028244 | Dev Loss: 0.014408 | Patience: 0/5
> New best model saved | Improvement: 0.000499

Epoch [ 12/15] | Train Loss: 0.026985 | Dev Loss: 0.013903 | Patience: 0/5
> New best model saved | Improvement: 0.000505

Epoch [ 13/15] | Train Loss: 0.026717 | Dev Loss: 0.013747 | Patience: 0/5
> New best model saved | Improvement: 0.000155

Epoch [ 14/15] | Train Loss: 0.026041 | Dev Loss: 0.013349 | Patience: 0/5
> New best model saved | Improvement: 0.000398

Epoch [ 15/15] | Train Loss: 0.024889 | Dev Loss: 0.012944 | Patience: 0/5
> New best model saved | Improvement: 0.000405
```

```
Epoch [  1/15] | Train Loss: 0.978192 | Dev Loss: 0.927809 | Patience: 0/5
> New best model saved | Improvement: inf

Epoch [  2/15] | Train Loss: 0.683201 | Dev Loss: 0.342159 | Patience: 0/5
> New best model saved | Improvement: 0.585649

Epoch [  3/15] | Train Loss: 0.213919 | Dev Loss: 0.117312 | Patience: 0/5
> New best model saved | Improvement: 0.224847

Epoch [  4/15] | Train Loss: 0.093523 | Dev Loss: 0.049060 | Patience: 0/5
> New best model saved | Improvement: 0.068252

Epoch [  5/15] | Train Loss: 0.050107 | Dev Loss: 0.027035 | Patience: 0/5
> New best model saved | Improvement: 0.022025

Epoch [  6/15] | Train Loss: 0.038273 | Dev Loss: 0.020775 | Patience: 0/5
> New best model saved | Improvement: 0.006260

Epoch [  7/15] | Train Loss: 0.033921 | Dev Loss: 0.018219 | Patience: 0/5
> New best model saved | Improvement: 0.002556

Epoch [  8/15] | Train Loss: 0.031545 | Dev Loss: 0.016771 | Patience: 0/5
> New best model saved | Improvement: 0.001448

Epoch [  9/15] | Train Loss: 0.030715 | Dev Loss: 0.016090 | Patience: 0/5
> New best model saved | Improvement: 0.000681

Epoch [ 10/15] | Train Loss: 0.029319 | Dev Loss: 0.014906 | Patience: 0/5
> New best model saved | Improvement: 0.001183

Epoch [ 11/15] | Train Loss: 0.028244 | Dev Loss: 0.014408 | Patience: 0/5
> New best model saved | Improvement: 0.000499

Epoch [ 12/15] | Train Loss: 0.026985 | Dev Loss: 0.013903 | Patience: 0/5
> New best model saved | Improvement: 0.000505

Epoch [ 13/15] | Train Loss: 0.026717 | Dev Loss: 0.013747 | Patience: 0/5
> New best model saved | Improvement: 0.000155

Epoch [ 14/15] | Train Loss: 0.026041 | Dev Loss: 0.013349 | Patience: 0/5
> New best model saved | Improvement: 0.000398

Epoch [ 15/15] | Train Loss: 0.024889 | Dev Loss: 0.012944 | Patience: 0/5
> New best model saved | Improvement: 0.000405
```

Same performance on test set?

Evaluating the best model on the test set

[29]:

```
model.eval()
test_loss = 0.0
all_predictions = []
all_actuals = []

with torch.no_grad():
    for batch_X, batch_y in test_loader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)

        outputs = model(batch_X)

        # accumulate loss
        loss = criterion(outputs, batch_y)
        test_loss += loss.item() * batch_X.size(0)

        all_predictions.append(outputs.cpu())
        all_actuals.append(batch_y.cpu())
```

Evaluating the best model on the test set

[29]:

```
test_loss = 0.0
all_predictions = []
all_actuals = []
```

Evaluating the best model on the test set

[29]:

```
with torch.no_grad():
```

Evaluating the best model on the test set

[29]:

```
    for batch_X, batch_y in test_loader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)

        outputs = model(batch_X)
```

Evaluating the best model on the test set

[29]:

```
        # accumulate loss
        loss = criterion(outputs, batch_y)
        test_loss += loss.item() * batch_X.size(0)
```

Loss comparasion between training, development and testing datasets

```
print(f"Train Loss: {train_losses[best_epoch-1]:.6f}")
print(f"Dev Loss:   {best_dev_loss:.6f}")
print(f"Test Loss:  {test_loss:.6f}")
```

```
Train Loss: 0.024889
Dev Loss:   0.012944
Test Loss:  0.013183
```

No overfitting!

Mean Squared Error, Root Mean Squared Error and Mean Absolute Error

Mean Squared Error means the average of the squared differences between predictions and actual values

Root Mean Squared Error means just the square root of MSE. This way, bringing the error back to original units

To end, Mean Absolute Error means the average of the absolute differences between predictions and actual values

```
mse = np.mean((predictions_original - actuals_original)**2)
print(f"  MSE (Mean Squared Error):  {mse:.2f}")

rmse = np.sqrt(mse)
print(f"  RMSE (Root Mean Squared Error): {rmse:.2f} bbl/day")

mae = np.mean(np.abs(predictions_original - actuals_original))
print(f"  MAE (Mean Absolute Error):  {mae:.2f} bbl/day")
```

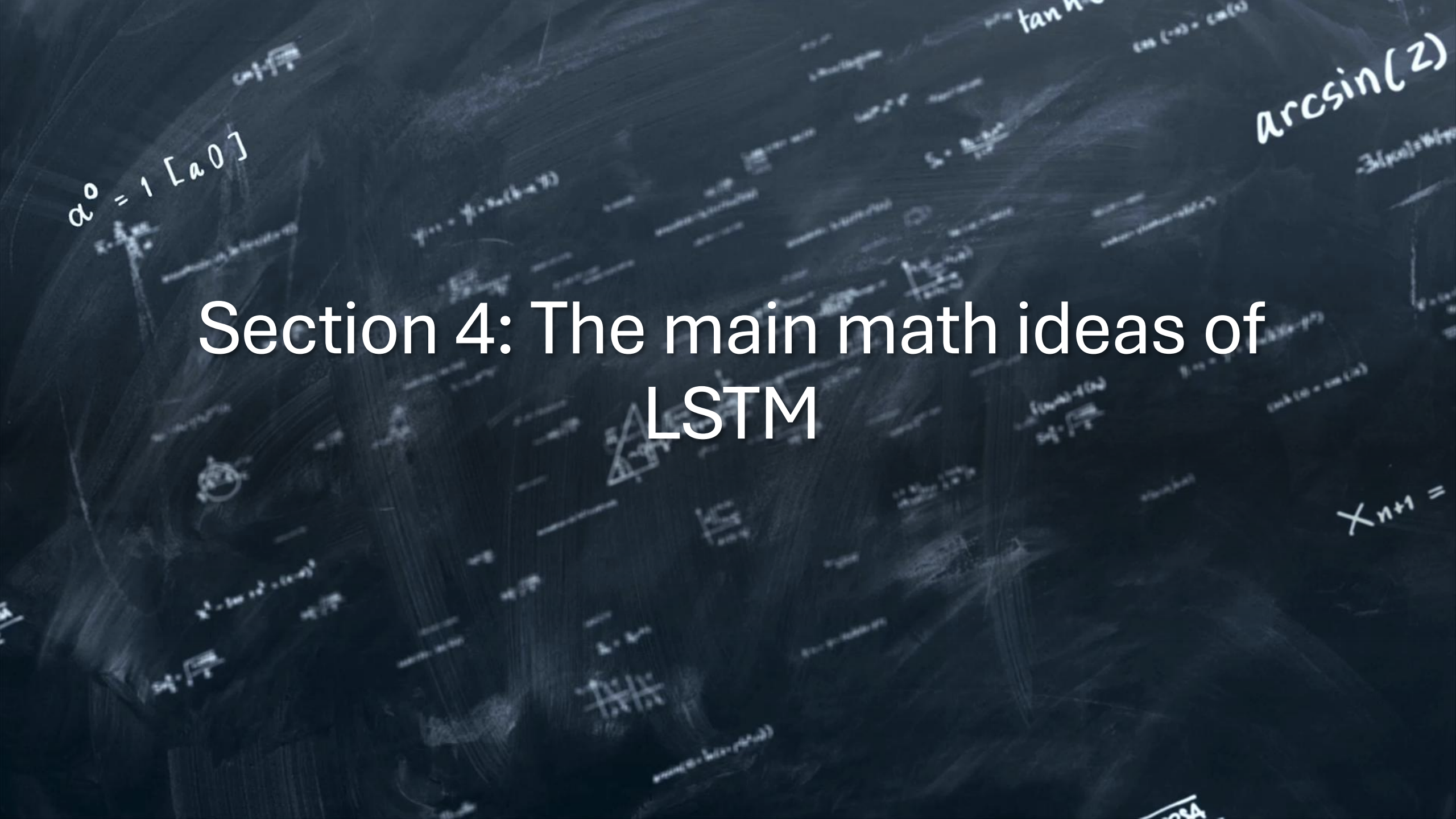
```
  MSE (Mean Squared Error):  17921.57
  RMSE (Root Mean Squared Error): 133.87 bbl/day
  MAE (Mean Absolute Error):  89.37 bbl/day
```

Coefficient of Determination

Coefficient of Determination means the proportion of variance in the actual values that is explained by the model's predictions

```
r2 = 1 - (np.sum((actuals_original - predictions_original)**2) / np.sum((actuals_original - np.mean(actuals_original))**2))
print(f"  R² (Coefficient of Determination):   {r2:.4f}")
```

```
  R² (Coefficient of Determination):   0.9867
```



Section 4: The main math ideas of LSTM

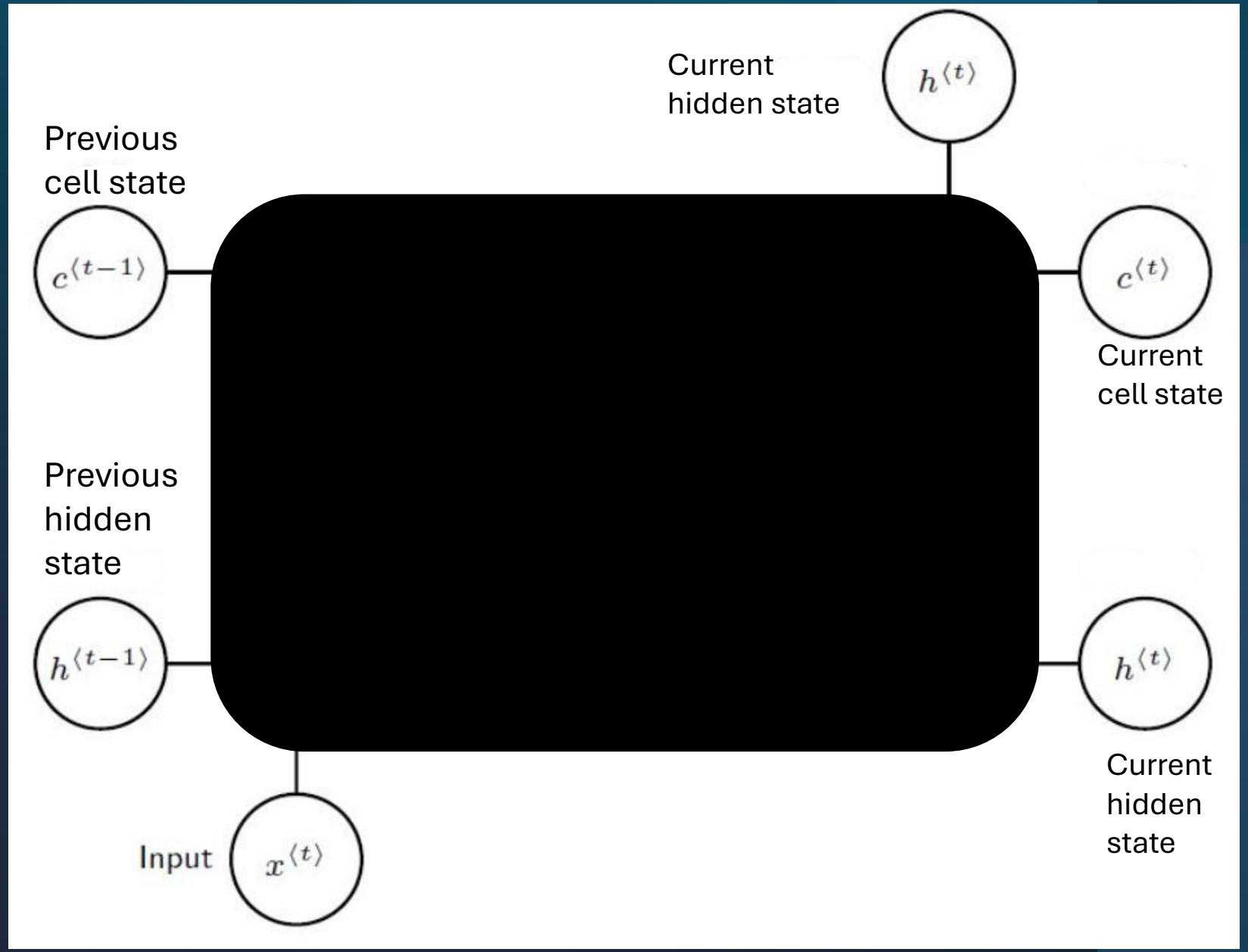
LSTM cell = Adaptable data flow mechanism.

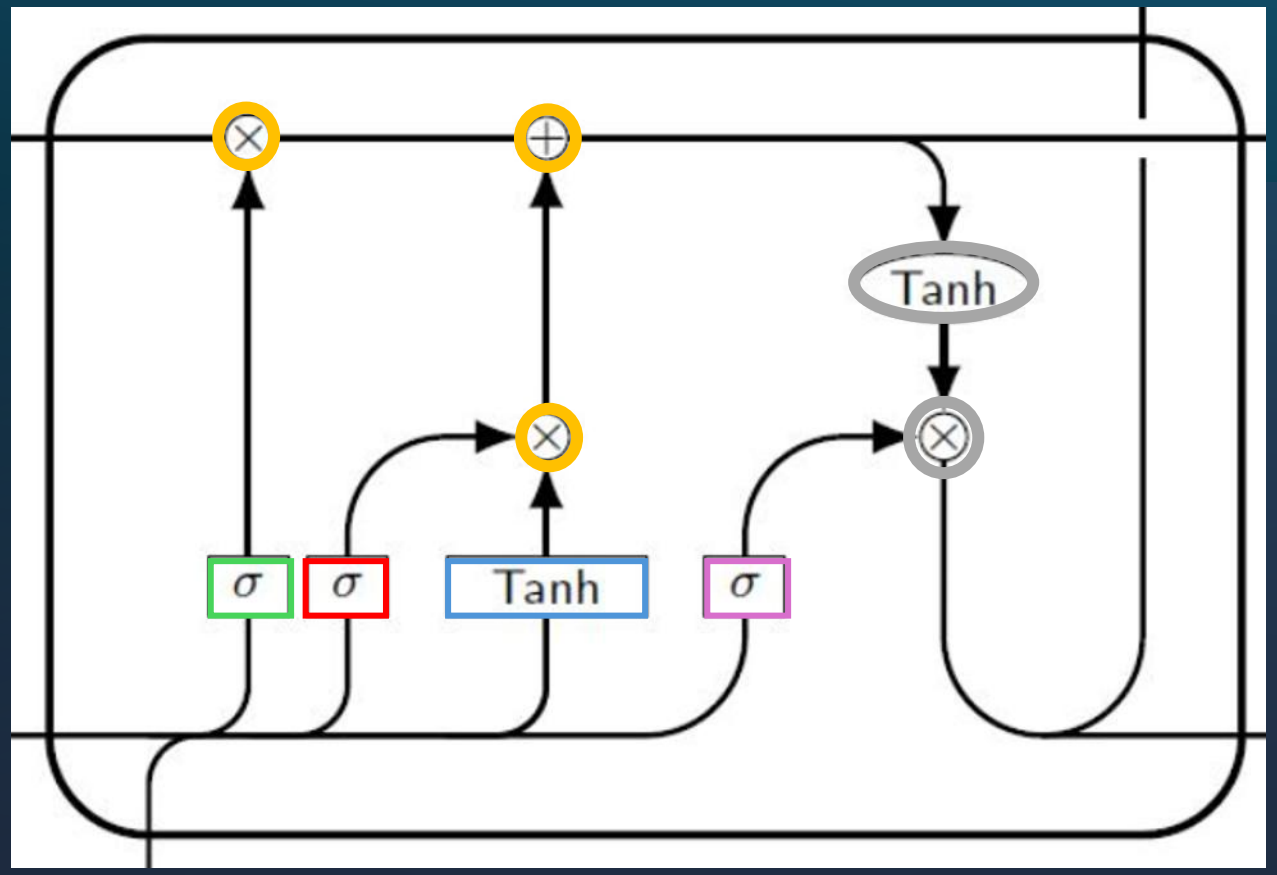
Like an adjustable faucet that controls the flow of data over time.

History and why it was created

- Created in 1997
- Very popular in the 2010s for natural language processing
- Less popular since 2017 with the paper “Attention is all you need”
  - Paper that created self-attention and the transformer neural network architecture

- https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735


Current
hidden state
$h^{\langle t \rangle}$
Previous
cell state
$c^{\langle t-1 \rangle}$
$c^{\langle t \rangle}$
Current
cell state
Previous
hidden
state
$h^{\langle t-1 \rangle}$
$h^{\langle t \rangle}$
Current
hidden
state
Input
$x^{\langle t \rangle}$



Forget Gate

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

Input Gate

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

Candidate Gate

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

Output Gate

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

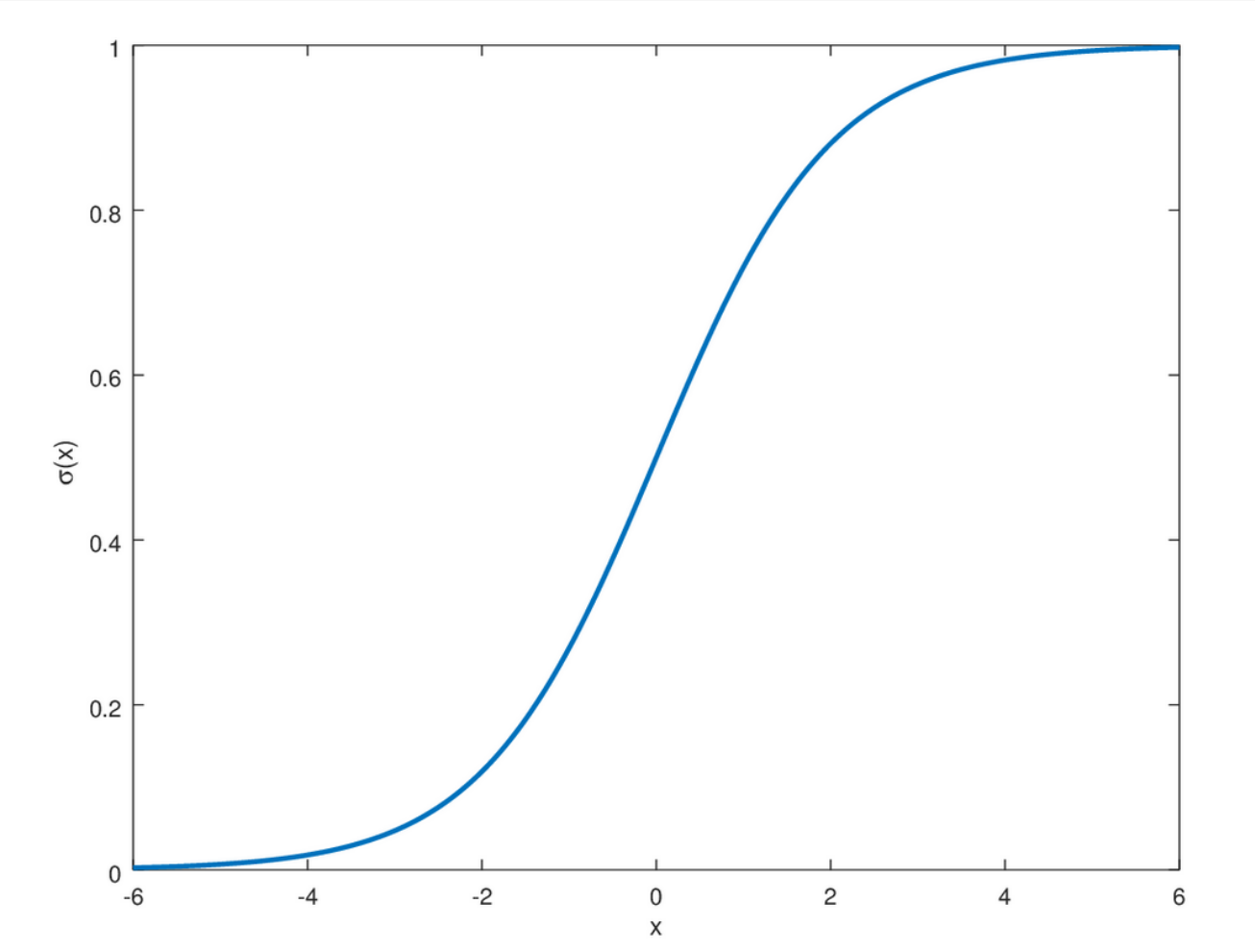
From these 4 gates, we decide the next:

Current Cell State

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

Current Hidden State

$$h_t = o_t \odot \tanh(C_t)$$



Sigmoid

- Output is a value between 0 and 1

- **Values between 0 and 1 can be interpreted as probabilities.**

Forget Gate

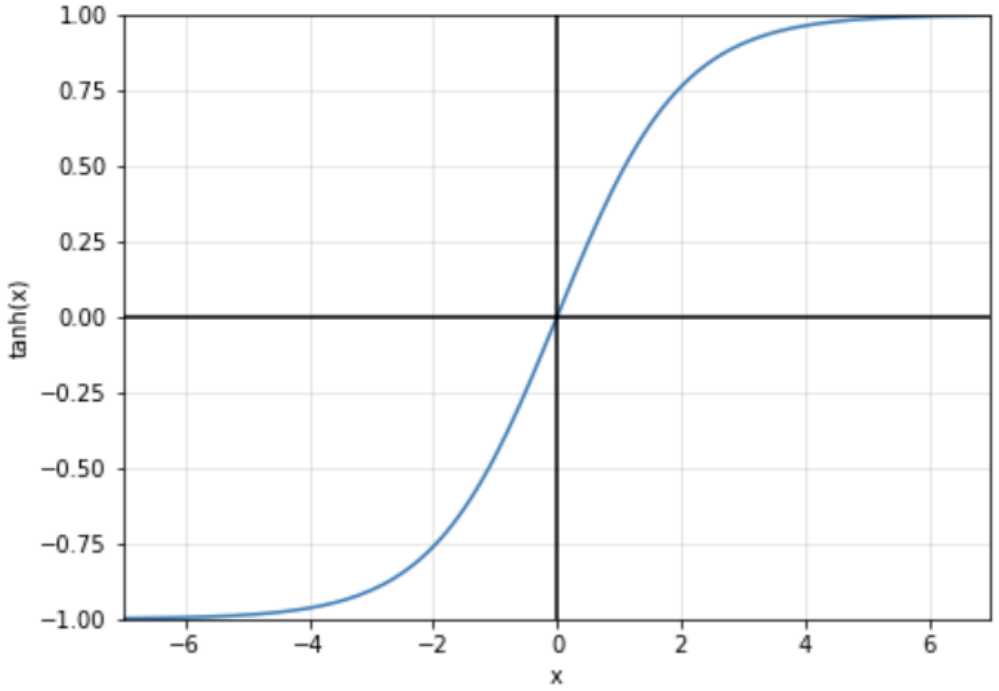
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Input Gate

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

Output Gate

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



Tanh

- The output is now -1 to 1, instead of from 0 to 1!
- **Preserves the order of magnitude**

Candidate Gate

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

Ensure nonlinearity and stable information flow

Sigmoid

- Convert parameters values between 0 and 1
- **Main idea**: Decide in a way how much data moves or not with probabilities
- **0.22 = 22% of data can go to next cell/hidden state**

Tanh

- Convert to smaller values between -1 and 1 to preserve order of magnitude
- **Main idea**: Prevent vanishing/exploding gradients

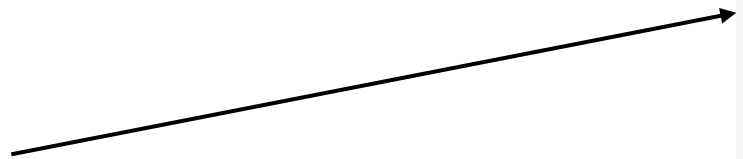
Simplifing the AI model code

```python
def __init__(self, input_size=6, hidden_size=128, num_layers=2, dropout=0.2, seq_length=12):
    super(LSTM_model, self).__init__()

    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.seq_length = seq_length

    self.lstm = nn.LSTM(
        input_size=input_size,
        hidden_size=hidden_size,
        num_layers=num_layers,
        batch_first=True,
        dropout=dropout
    )

    self.fully_connected_block = nn.Sequential(
        nn.Linear(hidden_size, 64),
        nn.ReLU(),
        nn.Dropout(dropout),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, 1)
    )
```



```python
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)
```

```python
self.fully_connected_block = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

```
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)

self.fully_connected_block = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

IMPORTANT

Row 0 of data

	Well	Target_Oil_Rate	Month_1_Date	Month_1_Gas Oil Ratio SC (ft³/bbl)	Month_1_Gas Rate SC - Monthly (ft³/day)	Month_1_Oil Rate SC - Monthly (bbl/day)	Month_1_Water Cut SC - %	Month_1_Water Rate SC - Monthly (bbl/day)	Month_1_Well Bottom-hole Pressure (psi)	Month_2_Date	Month_2_Gas Oil Ratio SC (ft³/bbl)	Mo
0	Well-1	5023.651982	2025-02-01	731.414978	3.567828e+06	4877.981171	0.000046	0.002222	3132.787350	2025-03-01	731.414978	3.66

After removing index, well name, target and date. 72 features(12 months * 6 Values per month) of data in row 0

Row 0 excluding index, well name, target, date

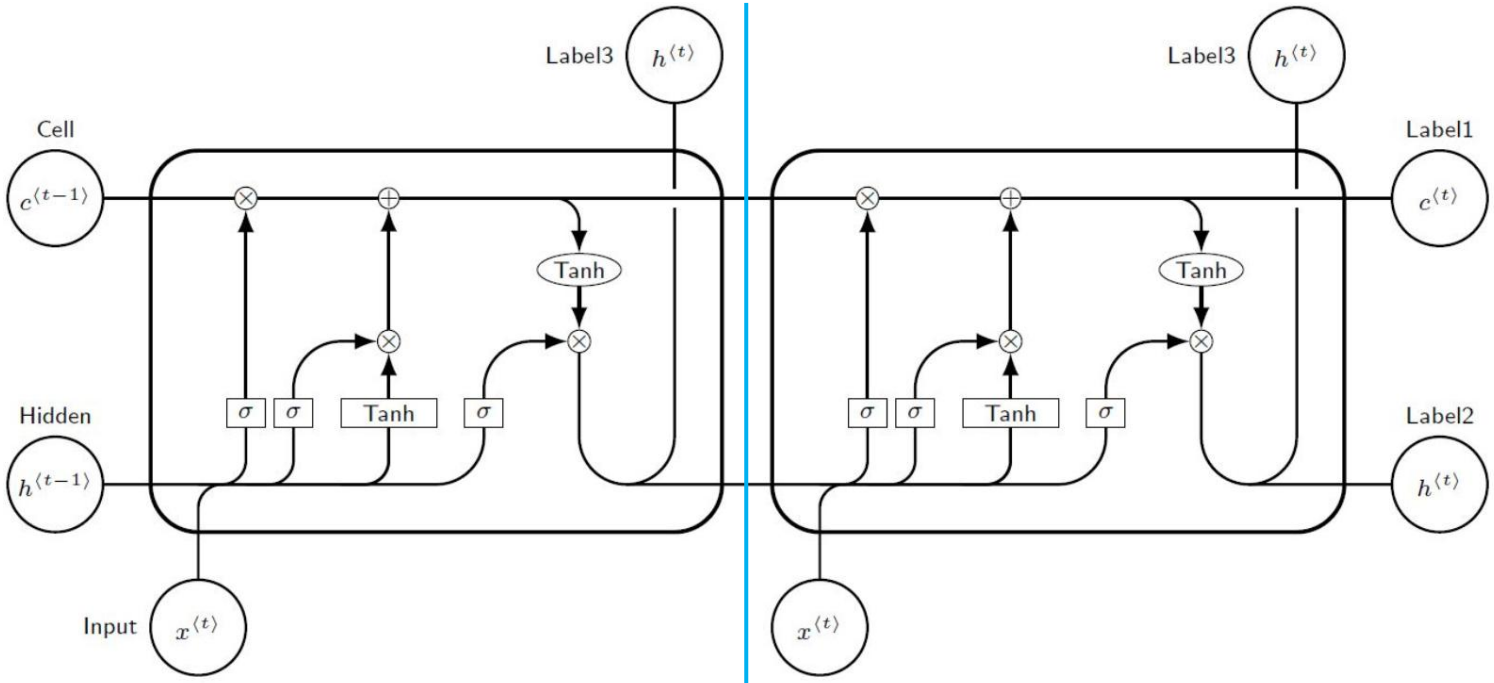
731.414978	3.567828e+06	4877.981171	0.000046	0.002222	3132.787350
731.414978	3.662807e+06	5007.836862	0.000064	0.003183	3090.725933
731.414978	3.626329e+06	4957.963500	0.000050	0.002470	3059.375781
731.414978	3.740978e+06	5114.712817	0.000048	0.002431	3039.715643
731.414978	3.657319e+06	5000.333284	0.000051	0.002528	3015.697719
731.414978	3.592993e+06	4912.386022	0.000103	0.005059	3003.678545
731.414978	3.680401e+06	5031.892153	0.000070	0.003539	2986.926955
731.414978	3.593829e+06	4913.529550	0.000067	0.003279	2968.007638
731.414978	3.534131e+06	4831.909593	0.000108	0.005200	2947.988424
731.414978	3.603630e+06	4926.930069	0.000124	0.006090	2923.813792
731.414978	3.648792e+06	4988.675921	0.000102	0.005107	2913.636242
731.414978	3.766142e+06	5149.117544	0.000092	0.004745	2903.087513

```
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)
```

__init__ from LSTM class

```
def __init__(self, input_size=6, hidden_size=128, num_layers=2, dropout=0.2, seq_length=12):
```

```
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)
```



h(t) - 128 units - Hidden state is the "output" designed for other layers

c(t) - 128 units - Cell state is the "internal memory" meant to stay within the LSTM

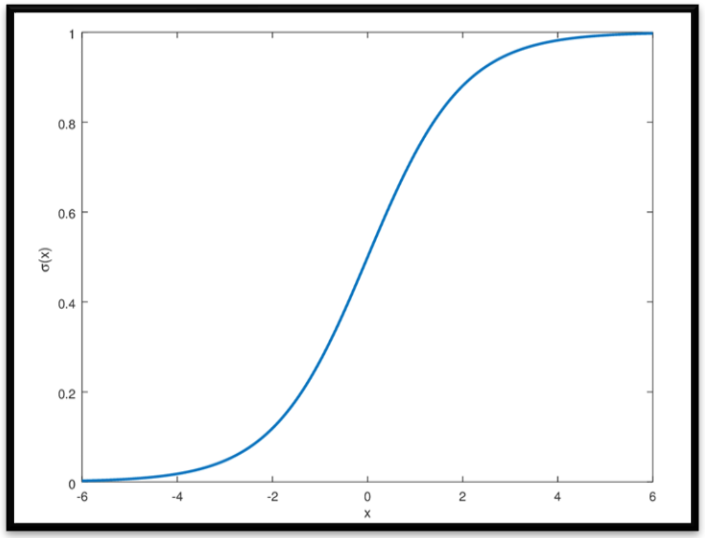
Dropout (20%)

Randomly disables 20% of h(t) connections during training to prevent overfitting (~25 of 128 units).

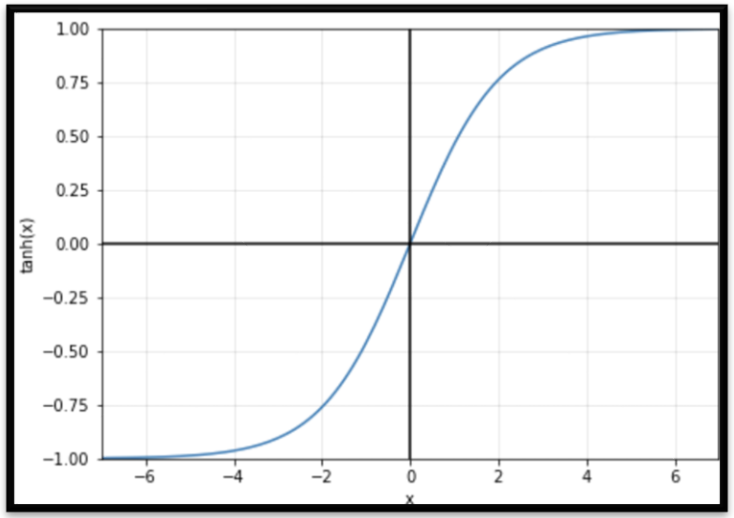
Gates will learn how to process the flow of data the right way

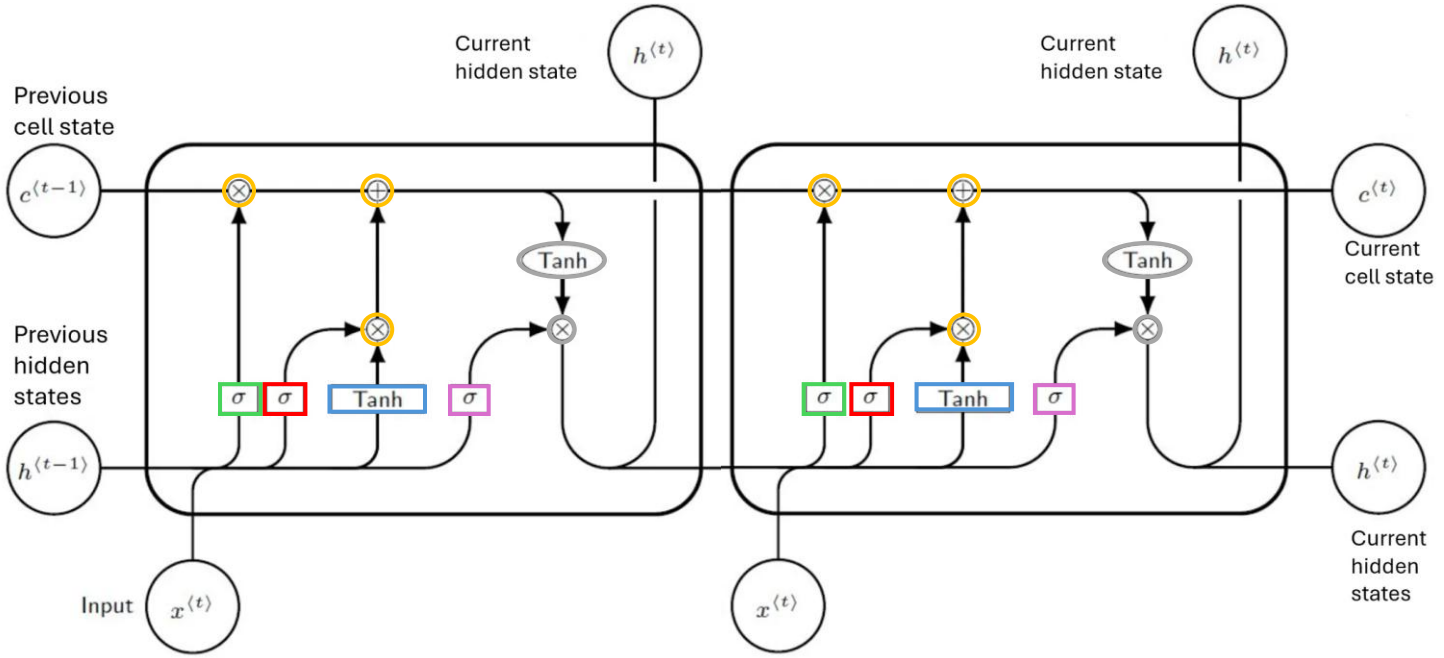
IMPORTANT

Sigmoid function – Between 0 and 1



Tanh function – Between -1 and 1





Forget Gate

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

Input Gate

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

Candidate Gate

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

Output Gate

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

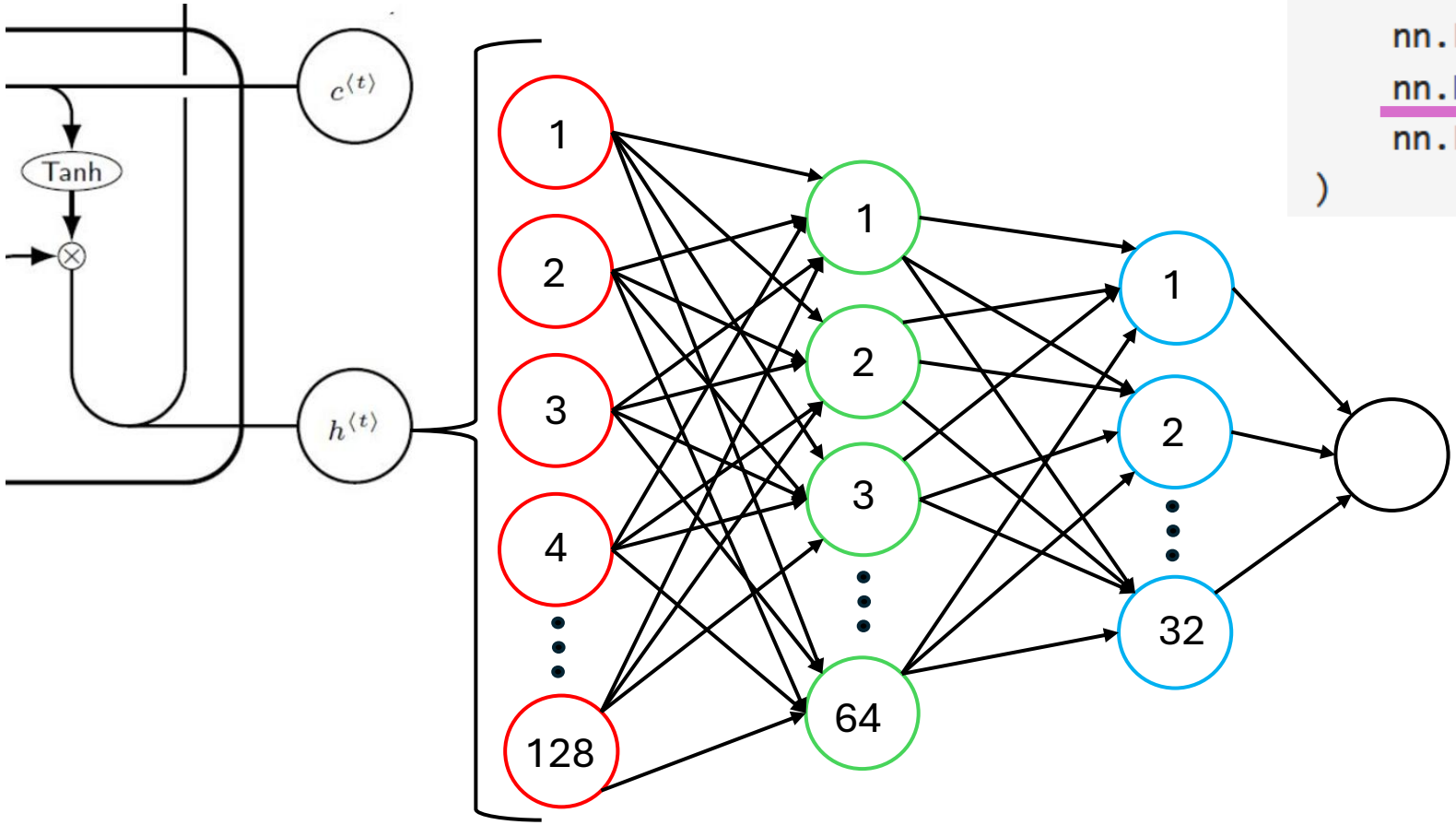
Current Cell State

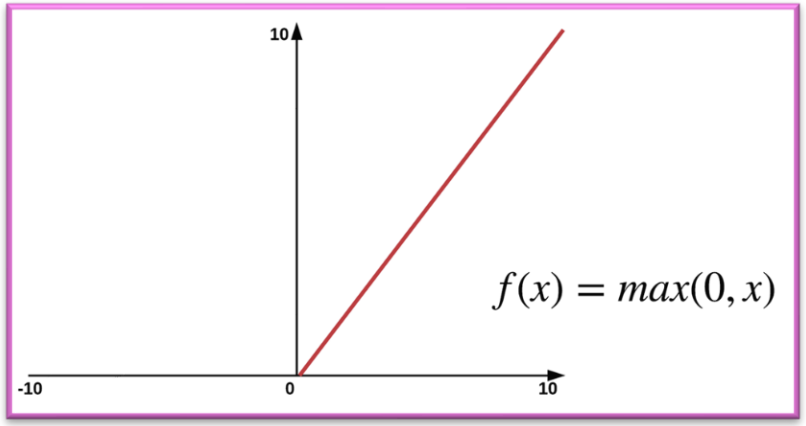
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Current Hidden State

$$h_t = o_t \odot \tanh(C_t)$$

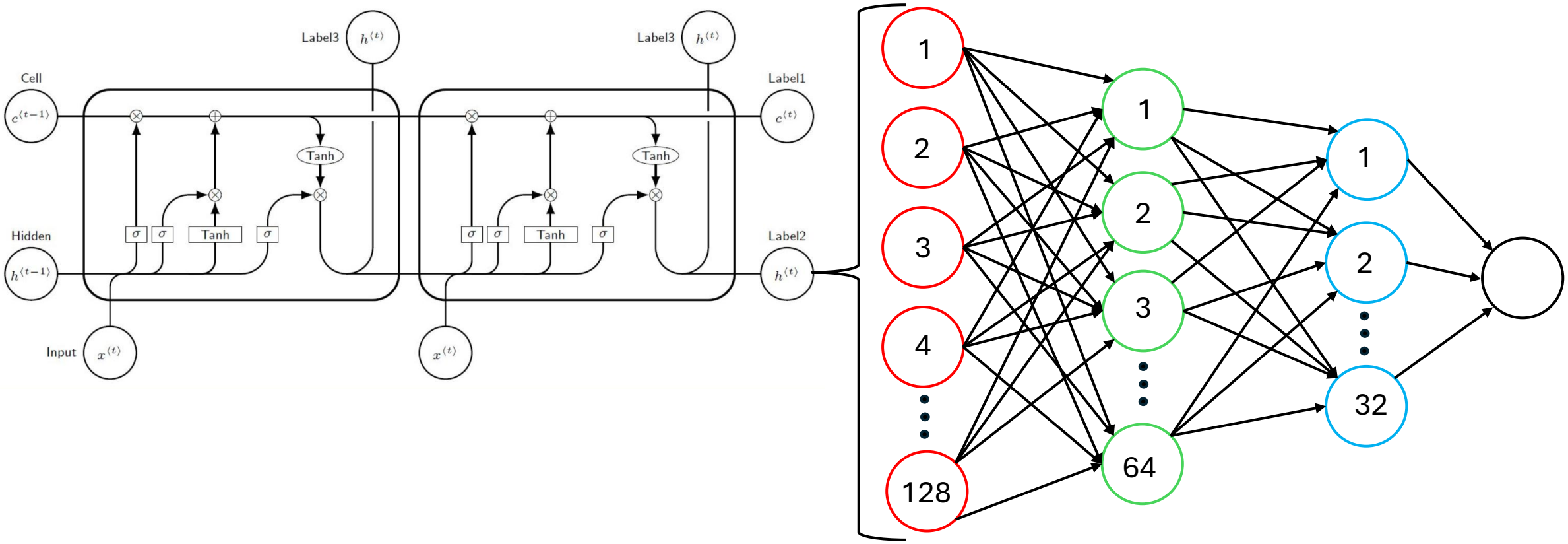
```
self.fully_connected_block = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

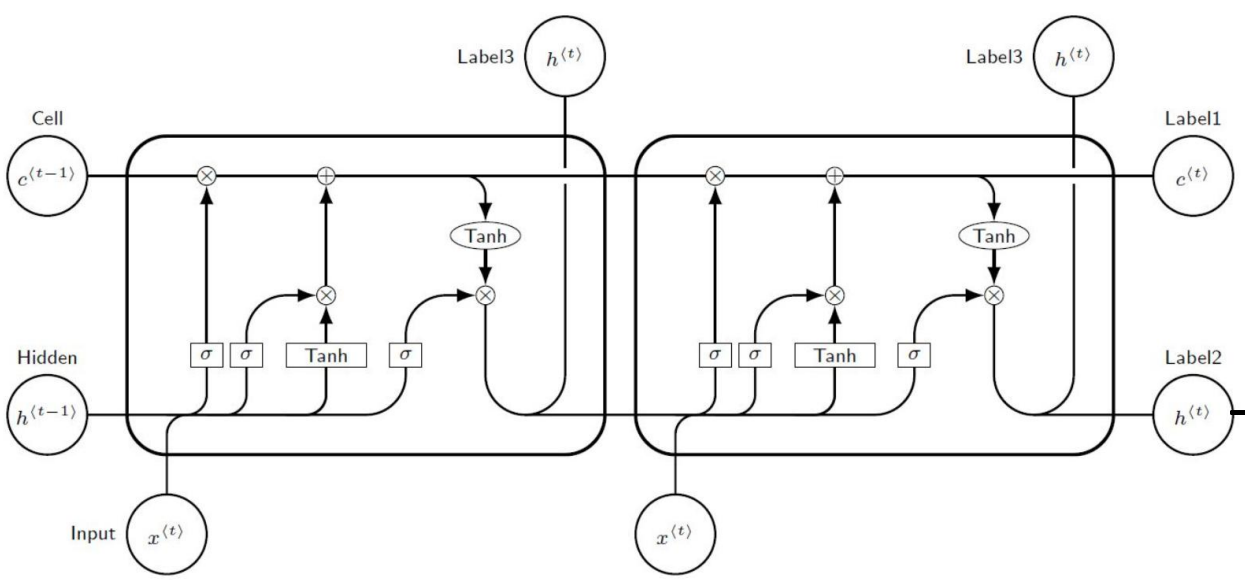


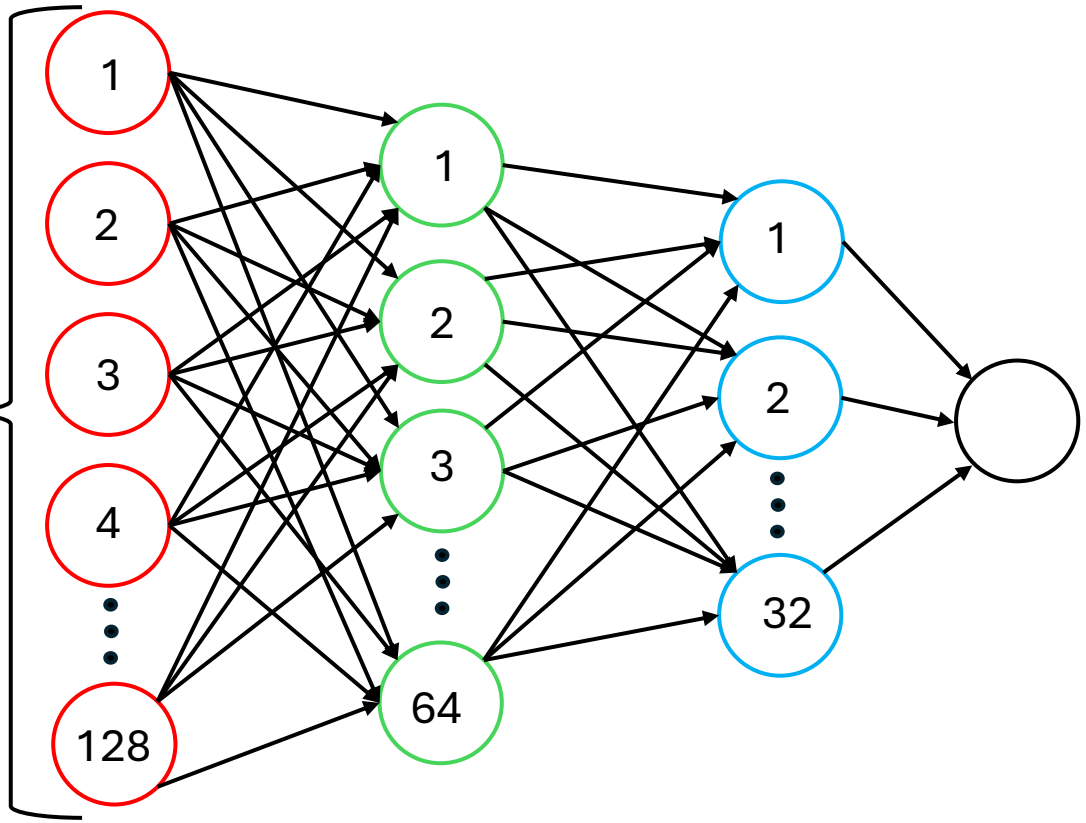


h(t) - 128 units - Hidden state is the "output" designed for other layers

c(t) - 128 units - Cell state is the "internal memory" meant to stay within the LSTM


Label3
$h^{\langle t \rangle}$
Label3
$h^{\langle t \rangle}$
Cell
$c^{\langle t-1 \rangle}$
Label1
$c^{\langle t \rangle}$
Tanh
Tanh
σ
σ
Tanh
σ
σ
σ
Tanh
σ
Hidden
$h^{\langle t-1 \rangle}$
Label2
$h^{\langle t \rangle}$
Input
$x^{\langle t \rangle}$
$x^{\langle t \rangle}$
1
2
3
4
128
1
2
3
64
1
2
32

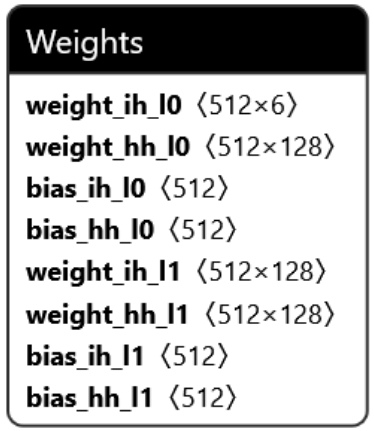




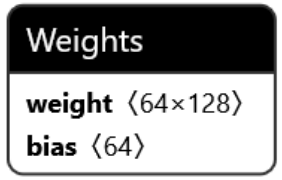

```
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)

self.fully_connected_block = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

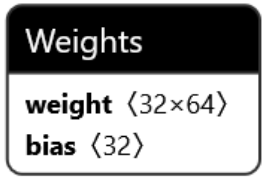
Complete architecture!



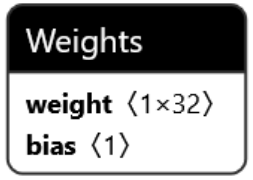
Add small blocks



Add red circles



Add green circles



Add blue circles

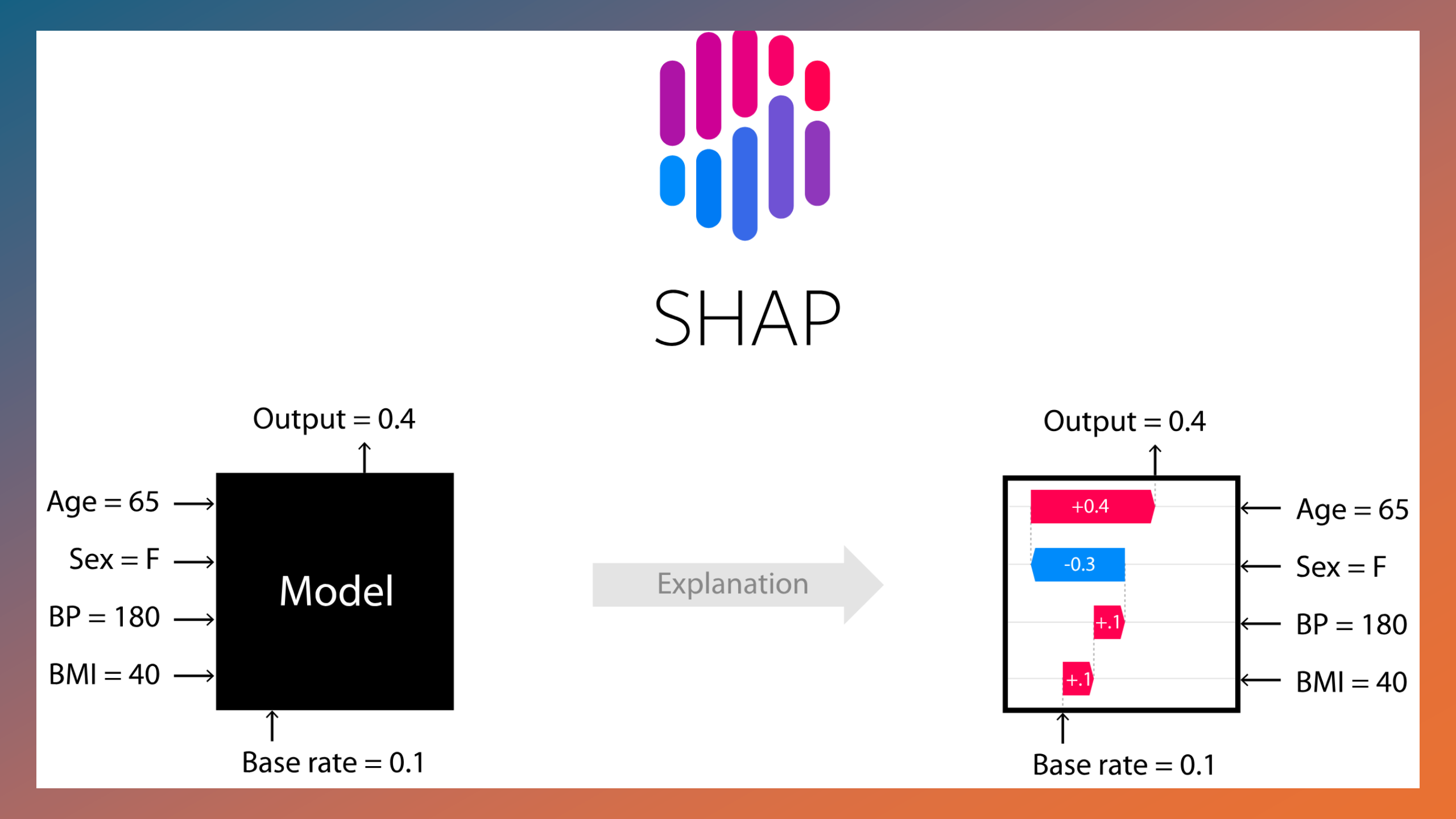
```python
self.lstm = nn.LSTM(
    input_size=6,
    hidden_size=128,
    num_layers=2,
    dropout=0.2
)

self.fully_connected_block = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1)
)
```

- https://netron.app/

Section 5: An Interpretability approach: SHAP

SHAP: SHapley Additive exPlanations


SHAP
Output = 0.4
Age = 65
Sex = F
BP = 180
BMI = 40
Model
Base rate = 0.1
Explanation
Output = 0.4
+0.4
-0.3
+.1
+.1
Age = 65
Sex = F
BP = 180
BMI = 40
Base rate = 0.1

A Unified Approach to Interpreting Model Predictions

Scott M. Lundberg
Paul G. Allen School of Computer Science
University of Washington
Seattle, WA 98105
slund1@cs.washington.edu

Su-In Lee
Paul G. Allen School of Computer Science
Department of Genome Sciences
University of Washington
Seattle, WA 98105
suinlee@cs.washington.edu

Abstract

Understanding why a model makes a certain prediction can be as crucial as the prediction's accuracy in many applications. However, the highest accuracy for large modern datasets is often achieved by complex models that even experts struggle to interpret, such as ensemble or deep learning models, creating a tension between *accuracy* and *interpretability*. In response, various methods have recently been proposed to help users interpret the predictions of complex models, but it is often unclear how these methods are related and when one method is preferable over another. To address this problem, we present a unified framework for interpreting predictions, SHAP (SHapley Additive exPlanations). SHAP assigns each feature an importance value for a particular prediction. Its novel components include: (1) the identification of a new class of additive feature importance measures, and (2) theoretical results showing there is a unique solution in this class with a set of desirable properties. The new class unifies six existing methods, notable because several recent methods in the class lack the proposed desirable properties. Based on insights from this unification, we present new methods that show improved computational performance and/or better consistency with human intuition than previous approaches.

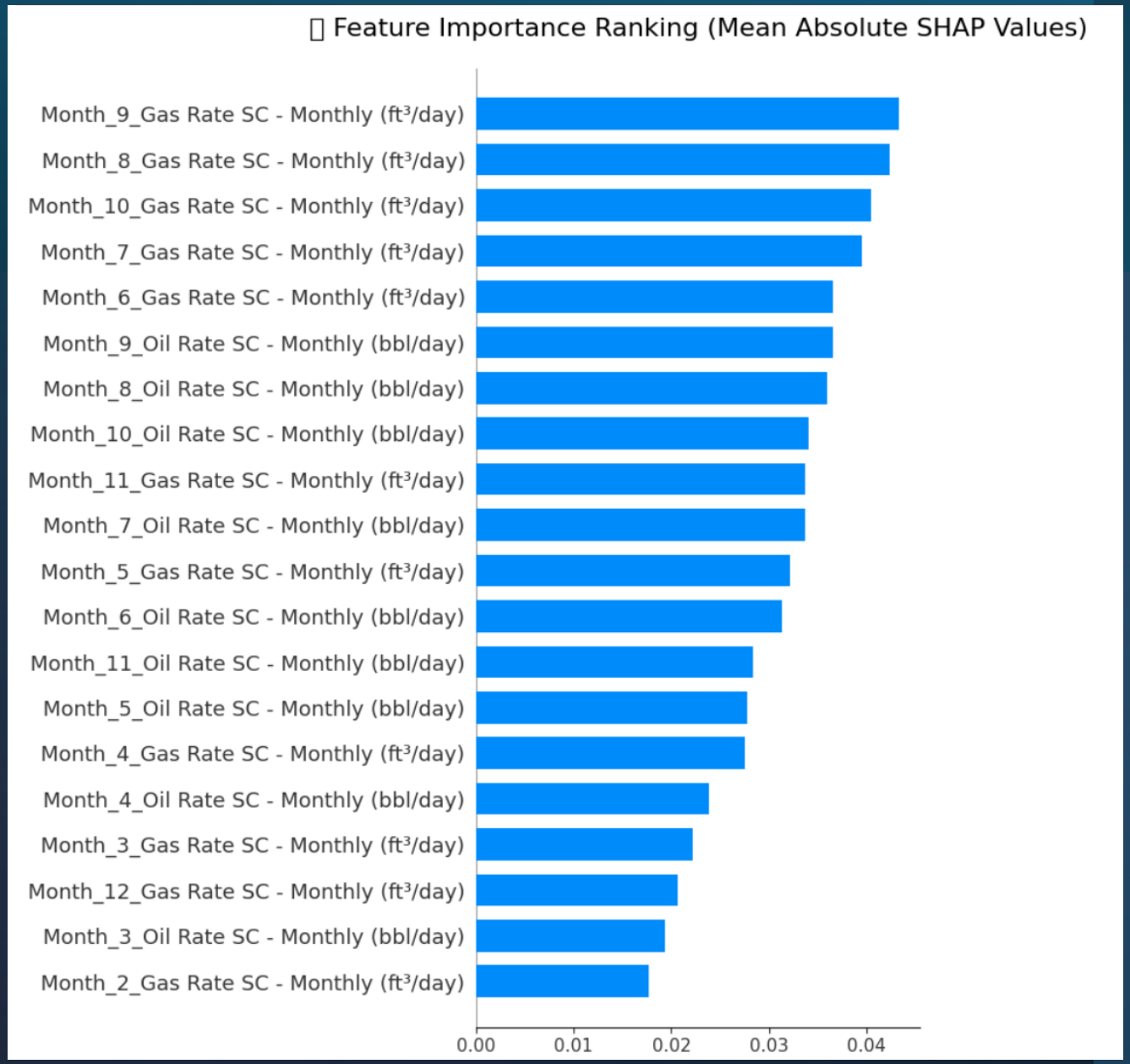
- SHAP (SHapley Additive exPlanations) is a game theory approach to explain the output of any machine learning model

https://dl.acm.org/doi/10.5555/3295222.3295230

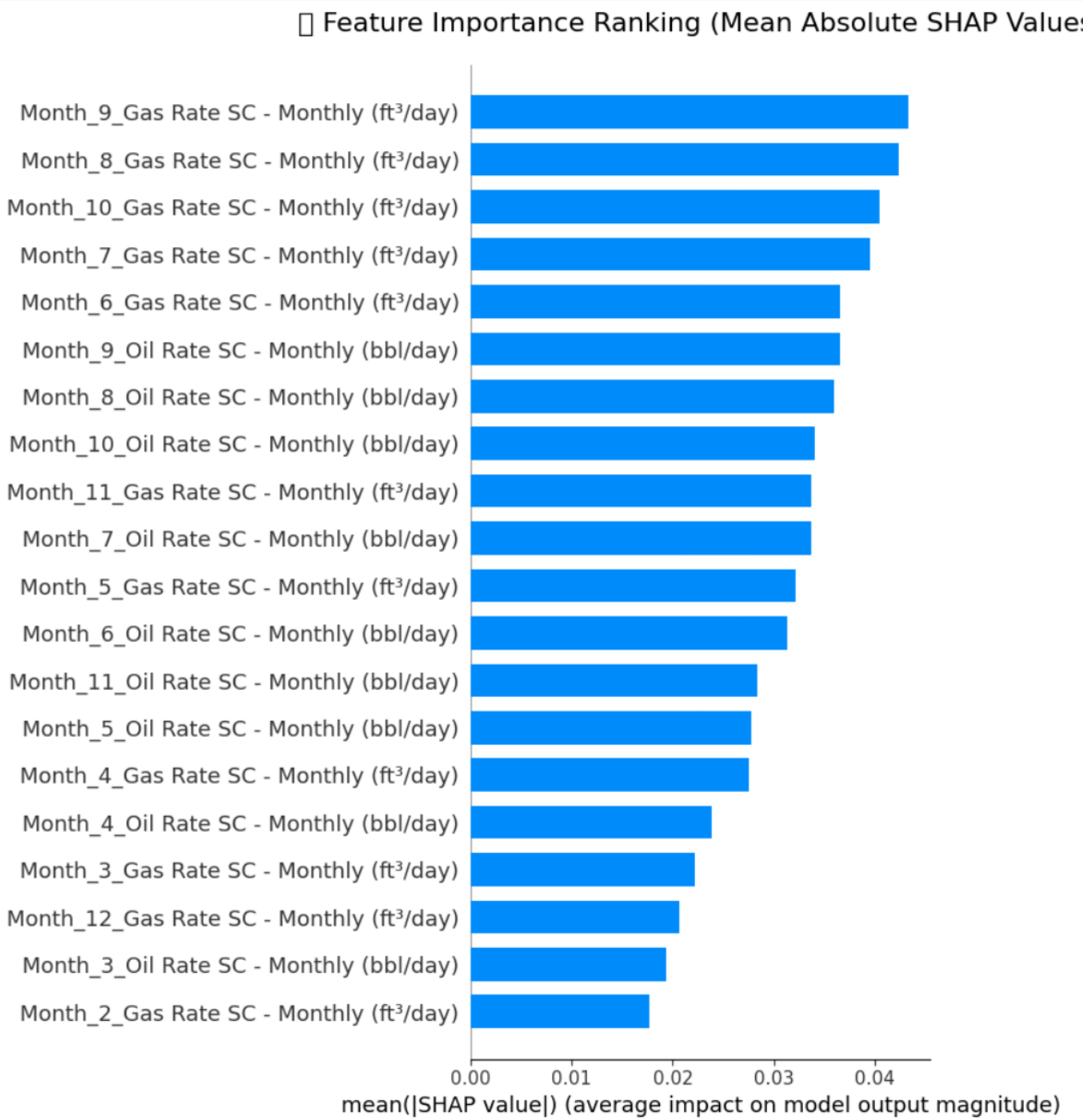
- SHAP

- Math technique to quantify how much each feature contributes to a model’s prediction.

Section 6: Final results from this study


Feature Importance Ranking (Mean Absolute SHAP Values)
Month_9_Gas Rate SC - Monthly (ft³/day)
Month_8_Gas Rate SC - Monthly (ft³/day)
Month_10_Gas Rate SC - Monthly (ft³/day)
Month_7_Gas Rate SC - Monthly (ft³/day)
Month_6_Gas Rate SC - Monthly (ft³/day)
Month_9_Oil Rate SC - Monthly (bbl/day)
Month_8_Oil Rate SC - Monthly (bbl/day)
Month_10_Oil Rate SC - Monthly (bbl/day)
Month_11_Gas Rate SC - Monthly (ft³/day)
Month_7_Oil Rate SC - Monthly (bbl/day)
Month_5_Gas Rate SC - Monthly (ft³/day)
Month_6_Oil Rate SC - Monthly (bbl/day)
Month_11_Oil Rate SC - Monthly (bbl/day)
Month_5_Oil Rate SC - Monthly (bbl/day)
Month_4_Gas Rate SC - Monthly (ft³/day)
Month_4_Oil Rate SC - Monthly (bbl/day)
Month_3_Gas Rate SC - Monthly (ft³/day)
Month_12_Gas Rate SC - Monthly (ft³/day)
Month_3_Oil Rate SC - Monthly (bbl/day)
Month_2_Gas Rate SC - Monthly (ft³/day)
0.00
0.01
0.02
0.03
0.04

- Recent Data = More impact
- Gas rates outweigh oil rates as predictors
- Gas features occupy 4 of top 5 positions, suggesting strong gas-oil production correlation



Saved: shap_bar_plot.png

LSTM trained well on the data

Loss comparasion between training, development and testing datasets

```
print(f"Train Loss: {train_losses[best_epoch-1]:.6f}")
print(f"Dev Loss:   {best_dev_loss:.6f}")
print(f"Test Loss:  {test_loss:.6f}")
```

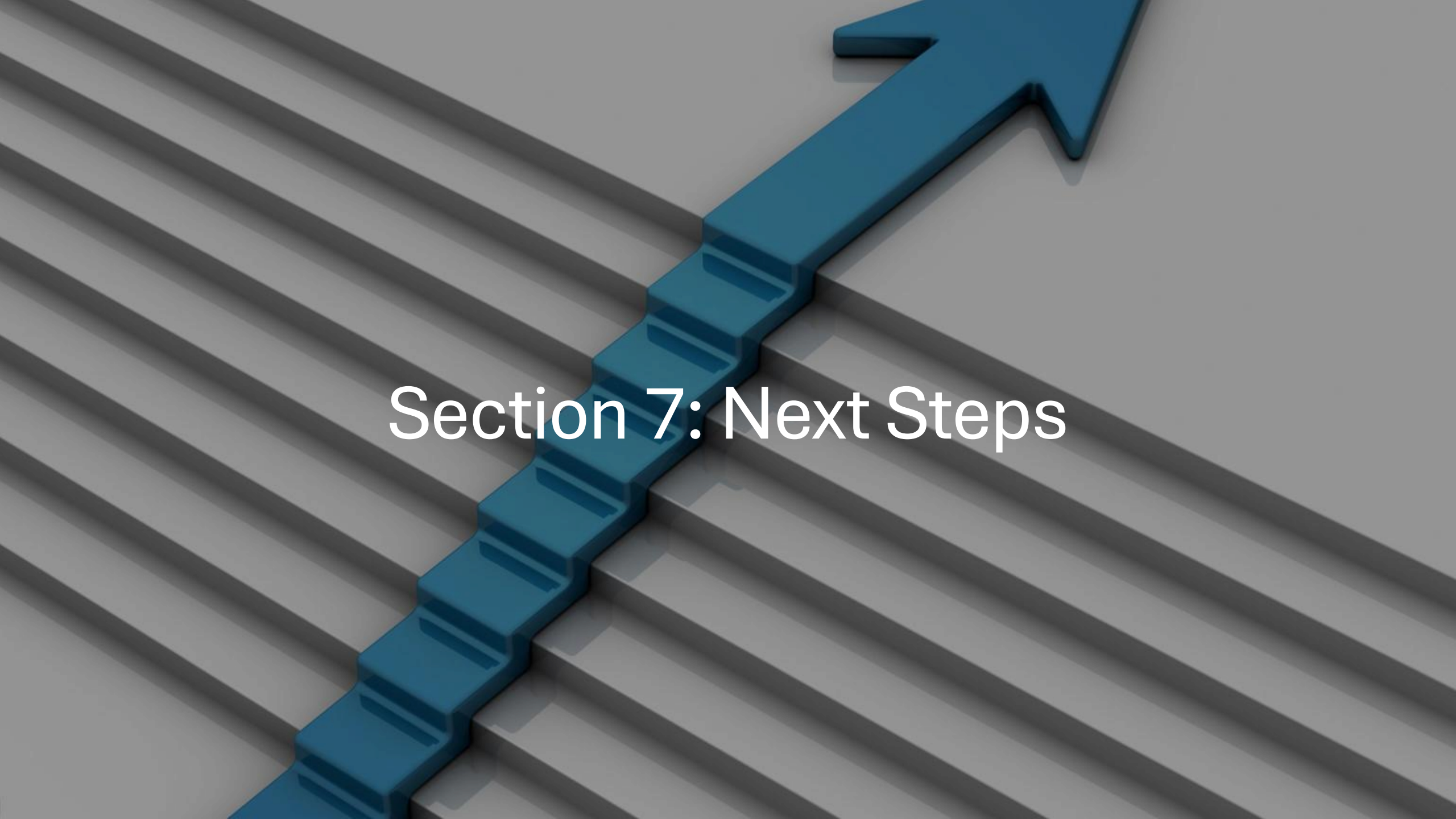
```
Train Loss: 0.024889
Dev Loss:   0.012944
Test Loss:  0.013183
```


Why know which features
are most important?

Feature importance reveals what drives predictions

Allows smarter data collection and this way faster AI models

Better business decisions



Section 7: Next Steps

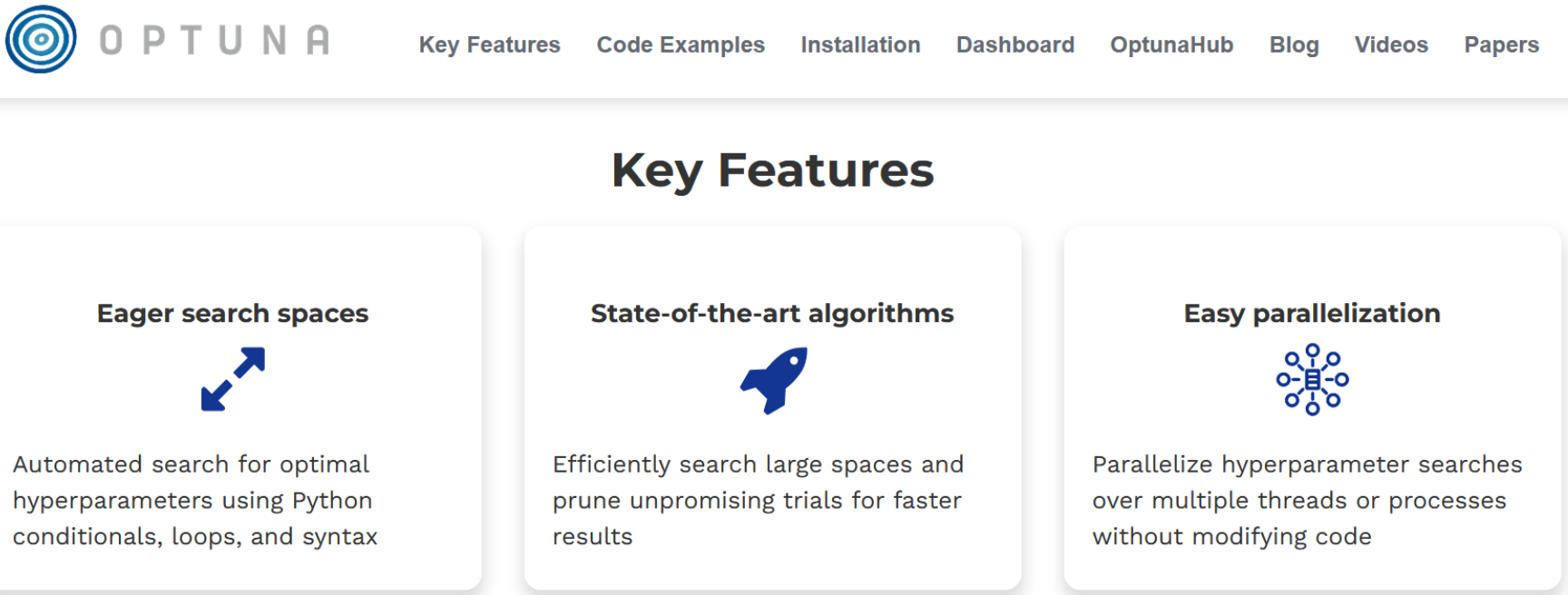
1. Use other models capable of modeling temporal dynamics, like gated recurrent units, to confirm interpretability results.

2. Use this Jupyter notebook to experiment with different ML/DL models using the top features identified by SHAP.

3. Create 10 AI models or more and compile all findings into a research paper.

Use Optuna - a Python library for fast hyperparameter optimization of AI models

https://optuna.org/



Takeaways

LSTM accurately predicted oil production from 248 wells using 12 months of data with no overfitting

Training pattern: Clear → Diagnose → Update (zero gradients, calculate loss, update weights)

LSTM uses 4 gates to control data flow with sigmoid (0-1) and tanh (-1 to 1) functions

Recent gas rates (months 8-10) were top predictors per SHAP analysis

Full code

https://github.com/tiagomonteiro0715/DeepOilFlow

Thanks for your attention