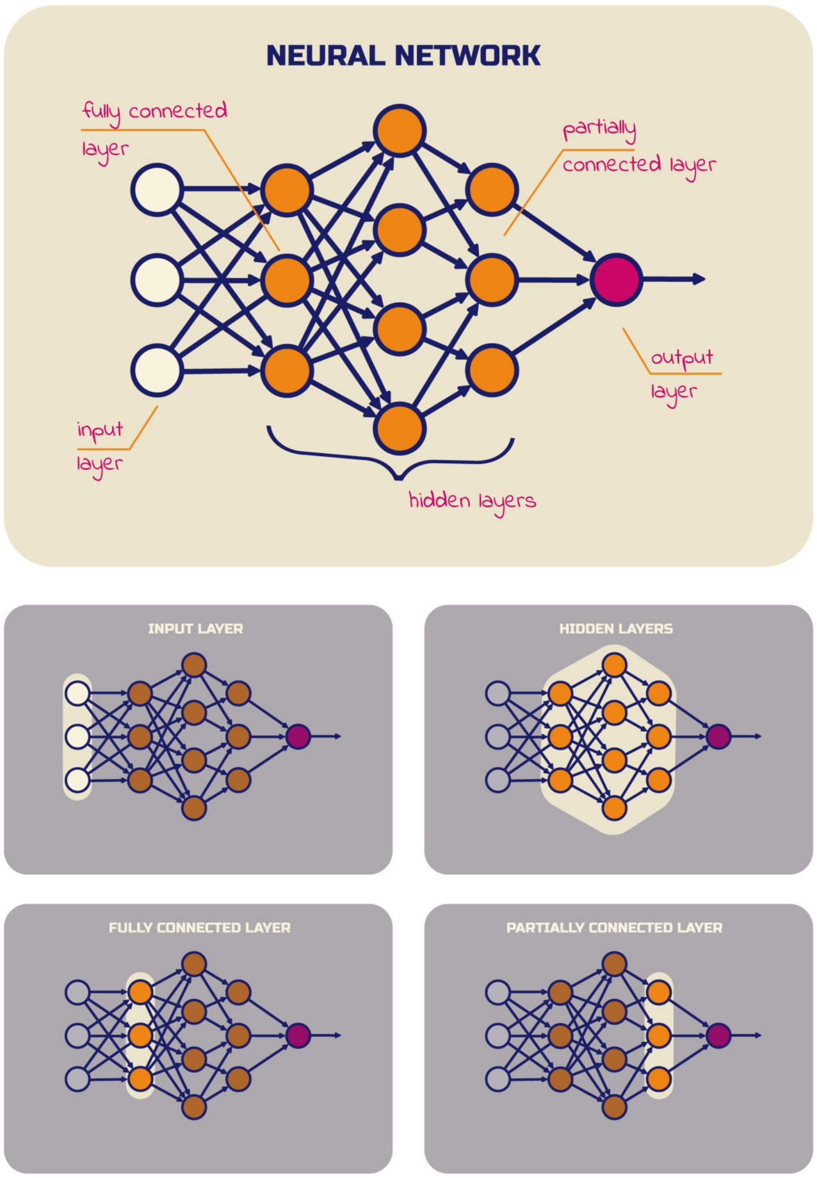


!

What is a Deep Learning Library?

Deep Learning libraries are software tools that help us:

```
•	Build neural networks
•	Train them on data
•	Test their performance
•	Visualize results
```

Without libraries, writing deep learning code from scratch would take thousands of lines of math-heavy code.

In simple words:

Deep Learning library = Math + GPU power + Ready-made tools

Why Do We Need TensorFlow, Keras, and PyTorch?

Problem	Without Library	With Library
Matrix calculations	Manual coding	Automatic
Backpropagation	Very complex	Built-in
GPU usage	Hard	Automatic
Model training	Months	Minutes

3.1 TensorFlow

- Developed by Google
- Very powerful and scalable
- Used in industry and production systems

3.2 Keras

- High-level API built **on top of TensorFlow**
- Very easy for beginners
- Focuses on simplicity

3.3 PyTorch

- Developed by Facebook (Meta)
- Very popular in research
- Code feels like normal Python

This workflow is **common everywhere**:

1. Import libraries
2. Load dataset
3. Prepare data
4. Define model
5. Compile model
6. Train model
7. Evaluate model
8. Visualize output

Keep this flow in mind. You will see it again and again.

What TensorFlow actually does

- Handles heavy math (matrix multiplication, gradients)
- Uses CPU and GPU efficiently
- Trains very large neural networks
- Runs models in real products

Think of TensorFlow like this

TensorFlow is like a **car engine**.

- Powerful
- Efficient
- But not very friendly to touch directly

Key characteristics

- Very scalable
- Used in production systems
- Slightly complex for beginners

Keras is **NOT a separate engine**.

It is a **high-level API** that uses TensorFlow underneath.

What Keras does

- Makes TensorFlow easy to use
- Hides low-level complexity
- Lets you build models in few lines

Think of Keras like this

Keras is like the **steering wheel and dashboard** of the car.

- You drive easily
- You don’t worry about engine internals

Example difference

TensorFlow (low level):

- Many steps
- More control

Keras:

- Simple

PyTorch is another **full deep learning framework**, developed by Meta (Facebook).

What makes PyTorch special

- Code feels like normal Python
- Easy to debug
- Very popular in research

Think of PyTorch like this

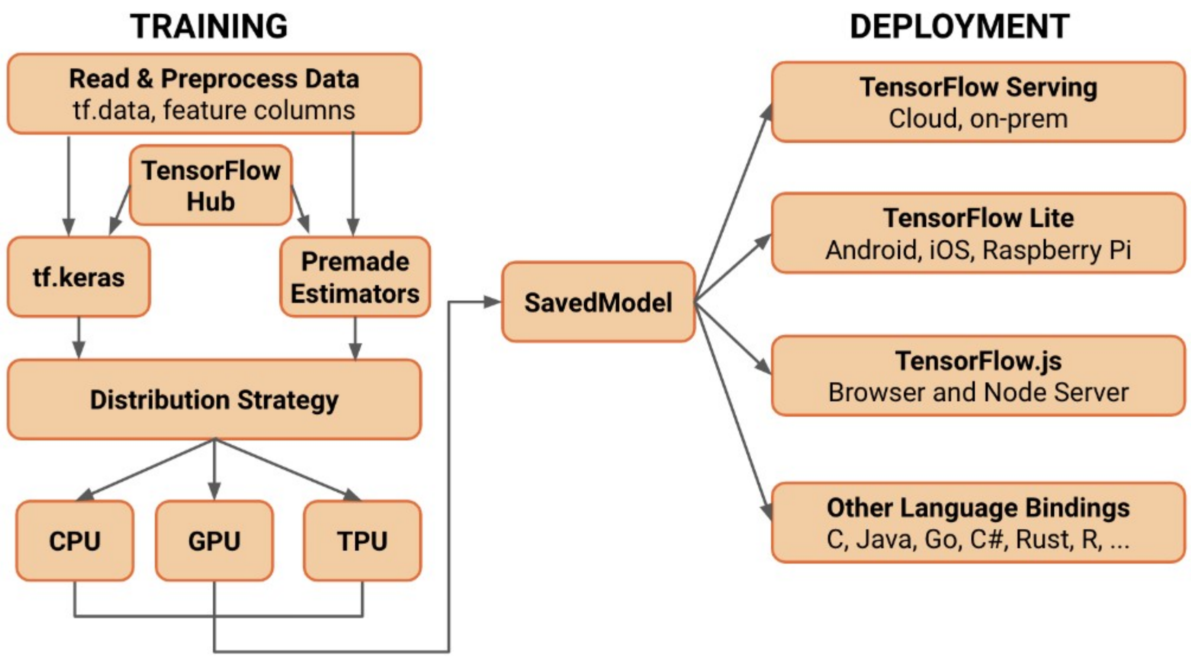
PyTorch is like a **manual sports car**.

- You control everything
- You feel what the engine is doing
- More freedom, slightly more work

Key characteristics

- Dynamic computation graphs
- Very flexible
- Loved by researchers

Question	TensorFlow	Keras	PyTorch
What is it?	Framework	API	Framework
Who made it?	Google	François Chollet	Meta
Beginner friendly?	Medium	Very High	Medium
Used for research?	Medium	Low	Very High
Used in industry?	Very High	High	High
Style	Structured	Simple	Python-like



What Does “Keras Is a High-Level API That Uses TensorFlow Underneath” Mean?

i.e.

We write our deep learning code using Keras, but the actual computation and training are performed by TensorFlow

Keras does not train models by itself. It depends on TensorFlow to do the heavy mathematical work.

What Is an API?

API (Application Programming Interface) is a set of tools or functions that allows us to interact with a system without knowing its internal complexity.

```
•    We give commands using the API
•    The internal system performs complex operations
•    We only see the final result
```

High-Level API vs Low-Level Framework

Low-Level Framework (TensorFlow)

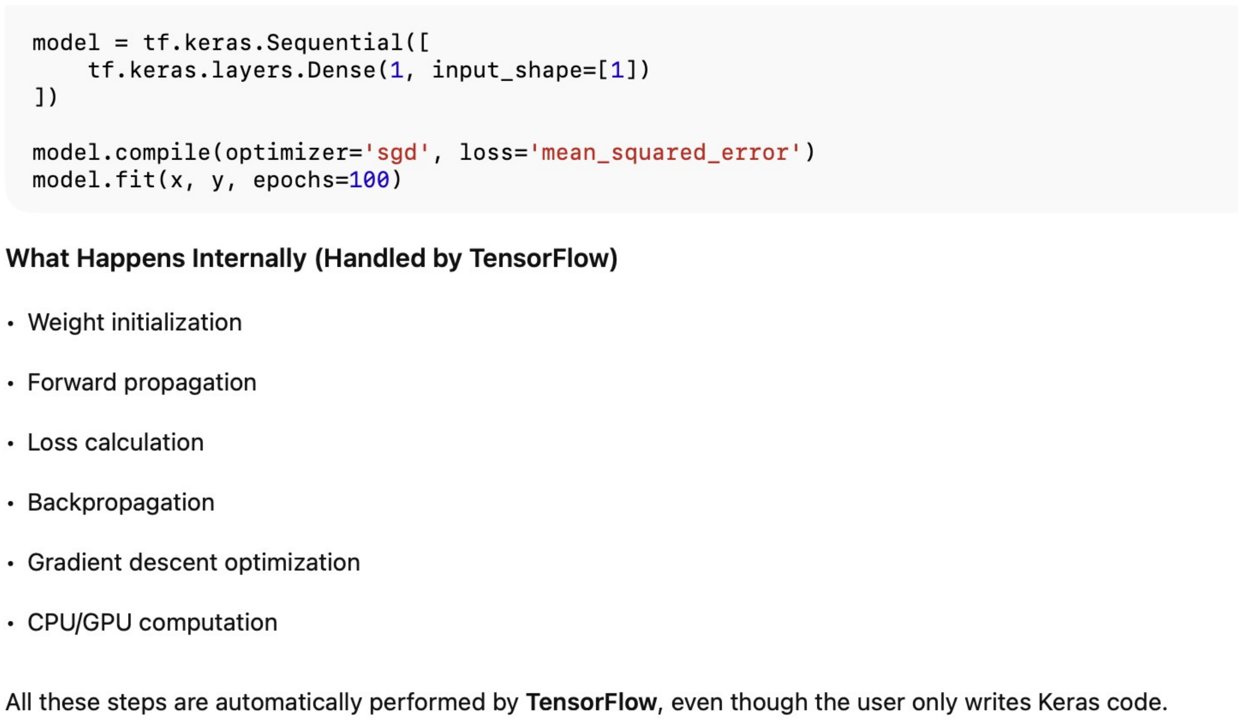
```
•    Handles mathematical operations
•    Performs gradient computation and backpropagation
•    Manages CPU/GPU execution
•    Requires more detailed and complex code
```

High-Level API (Keras)

```
•	Provides simple and readable commands
•	Hides internal complexity
•	Allows quick model creation and training
•	Ideal for beginners and rapid development
```

Relationship Between Keras and TensorFlow

```
•	TensorFlow is a complete deep learning framework
•	Keras is an interface built on top of TensorFlow
•	Keras uses TensorFlow as its backend engine
```



```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1])
])

model.compile(optimizer='sgd', loss='mean_squared_error')
model.fit(x, y, epochs=100)
```

What Happens Internally (Handled by TensorFlow)

- Weight initialization
- Forward propagation
- Loss calculation
- Backpropagation
- Gradient descent optimization
- CPU/GPU computation

All these steps are automatically performed by **TensorFlow**, even though the user only writes Keras code.

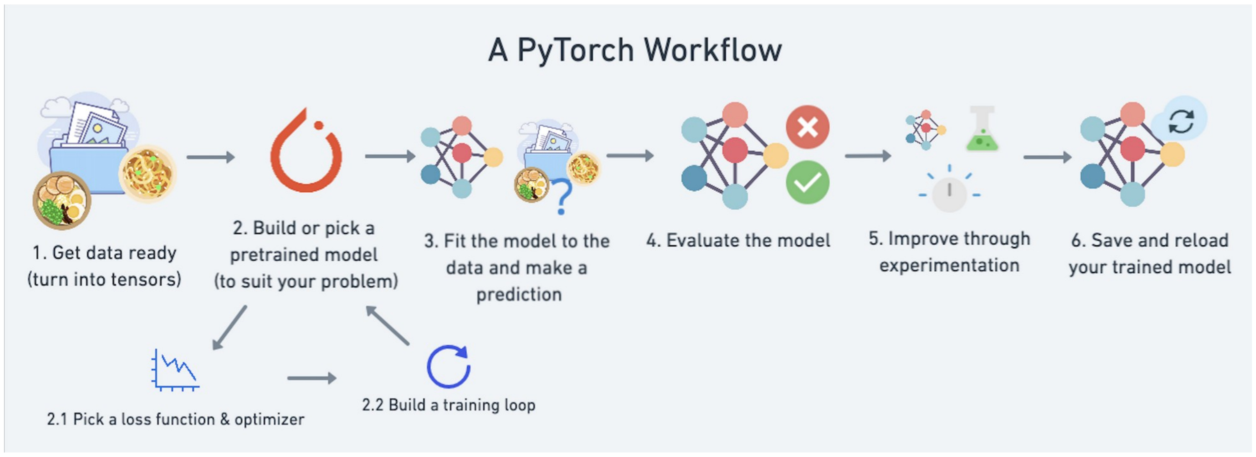
Real-World Analogy

```
•	TensorFlow: Engine of a car
•	Keras: Steering wheel and pedals
•	User: Driver
```

The driver controls the car using the steering wheel, but the engine performs the actual work.

Why Keras Is Called "High-Level"

```
•	Requires fewer lines of code
•	Easy to read and understand
•	Focuses on what to do, not how to do it
•	Suitable for beginners and teaching
```


A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model
2.1 Pick a loss function & optimizer
2.2 Build a training loop

PyTorch is a full deep learning framework used to build, train, and test neural networks.

Unlike Keras, PyTorch is not an API on top of another framework.

It directly performs all computations by itself.

PyTorch Is a Low-Level (But Friendly) Framework

PyTorch is called **low-level** because:

- You manually define training steps
- You explicitly write the training loop
- You see how learning actually happens

But it is still **beginner-friendly** because:

- Code looks like normal Python
- Easy to read and debug
- No complex syntax

What Does “Model Training” Actually Mean?

Training a neural network always involves these steps:

```
1.    Forward pass (prediction)
2.    Loss calculation
3.    Backpropagation (gradient computation)
4.    Weight update (optimizer)
```

in Keras user writes

```
model.compile(
optimizer='sgd',
loss='mse'
```

)

model.fit(x, y, epochs=100)

What Keras does internally (hidden from user)

When you call model.fit(), Keras automatically:

```
1.    Sends input through the model (forward pass)
2.    Computes the loss
3.    Computes gradients using backpropagation
4.    Updates weights using the optimizer
5.    Repeats this for every epoch
```

You do not write these steps.

Why this is called "automatic"

```
•     One function call (fit)
•     No training loop written by user
•     No manual gradient handling
•     Best for beginners and fast development
```

In Keras, you say what to train, not how to train.

PyTorch: Manual Training

What does the user write in PyTorch?

for epoch in range(100): optimizer.zero_grad() output = model(x) loss = criterion(output, y) loss.backward() optimizer.step()

Here, every step is written explicitly.

What each line does

- output = model(x)

→ Forward pass

- loss = criterion(output, y)

→ Loss calculation

- loss.backward()

→ Backpropagation (gradient computation)

- optimizer.step()

→ Weight update

Nothing is hidden.

Why this is called “manual”

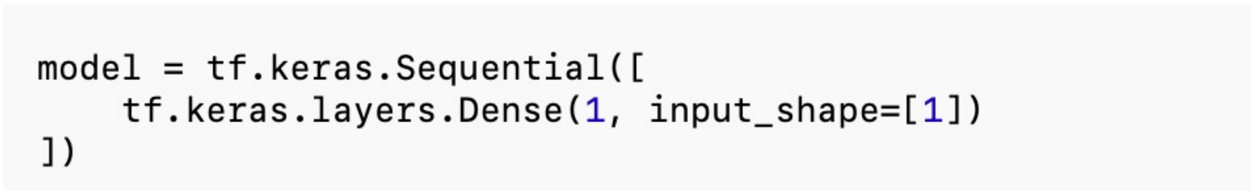
```
•     User writes the training loop
•     User controls gradient flow
•     User decides when and how to update weights
•     Better for deep understanding and research
```

In PyTorch, you control how the model learns.

Coding Differences Between Keras and PyTorch

```
•     Keras automates most of the training process.
•     PyTorch requires the user to explicitly write each training step.
```

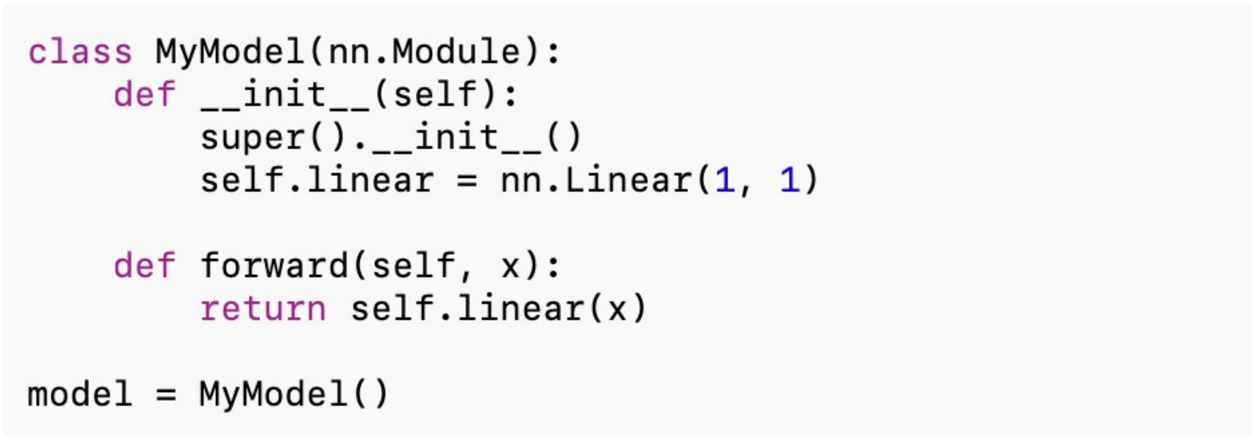
Keras Model Definition:



```python
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1])
])
```

```
•     Model is defined using a simple layer stack
•     No need to define a forward function
•     Less code, more abstraction
```

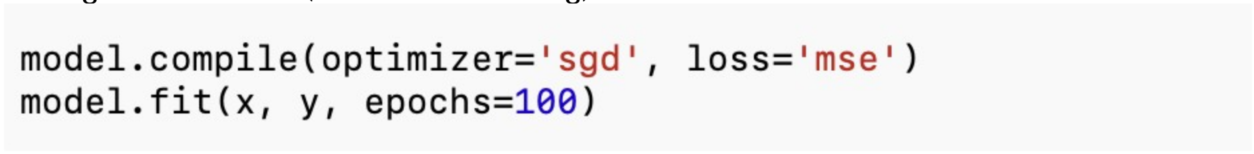
PyTorch Model Definition:



```python
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

model = MyModel()
```

```
•    Model must be defined as a class
•    Forward pass is explicitly written
•    More control over model behavior
```

1. Training Process Keras (Automatic Training)



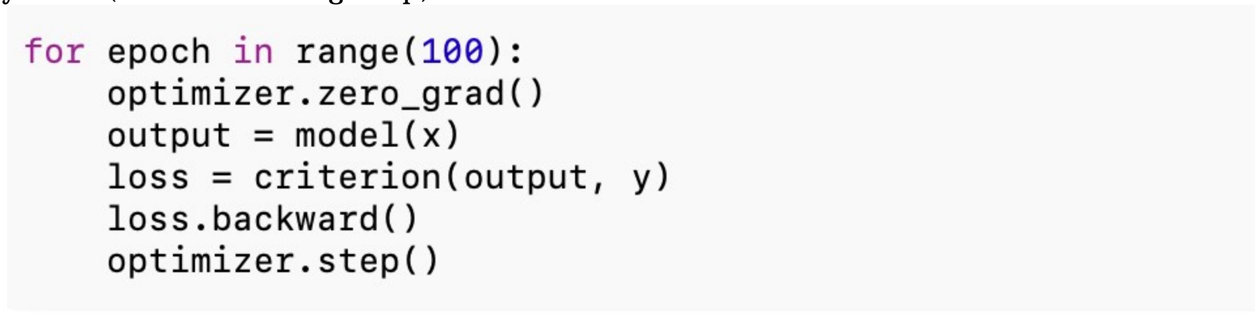
```python
model.compile(optimizer='sgd', loss='mse')
model.fit(x, y, epochs=100)
```

Internally, Keras automatically performs:

• Forward propagation • Loss computation • Backpropagation • Weight updates

The user does not see these steps.

1. PyTorch (Manual Training Loop)



```python
for epoch in range(100):
    optimizer.zero_grad()
    output = model(x)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()
```

In PyTorch, the user explicitly writes:'

• Forward pass • Loss calculation • Backward pass • Parameter updates

Handling Loss and Accuracy

Keras

```
model.compile(
    optimizer='sgd',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

```
•    Metrics are defined once
•    Stored automatically in history
```

PyTorch

```
pred = torch.round(torch.sigmoid(output))
correct = (pred == y).sum()
accuracy = correct / y.size(0)
```

```
•    Metrics are manually calculated
•    Full flexibility in metric definition
```

# Debugging Style • Keras hides many internal operations, which can make debugging harder • PyTorch follows normal Python execution, making step-by-step debugging easier

```
# pip install tensorflow torch matplotlib numpy
```

Predict output based on input numbers (y = 2x)

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Explanation:

```
•    tensorflow: Deep learning engine
•    numpy: For numerical data
•    matplotlib: For visual output

x = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([2, 4, 6, 8, 10], dtype=float)
```

Explanation:

```
•     x = input
•     y = expected output
•     Relationship: y = 2x
```

Step 3: Build Neural Network Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1])
])
```

Explanation:

```
•     Sequential: Model with layers in sequence
•     Dense: Fully connected layer
•     units=1: One neuron
•     input_shape=[1]: One input value
```

This neuron learns :

output = weight × input + bias

Step 4: Compile the Model

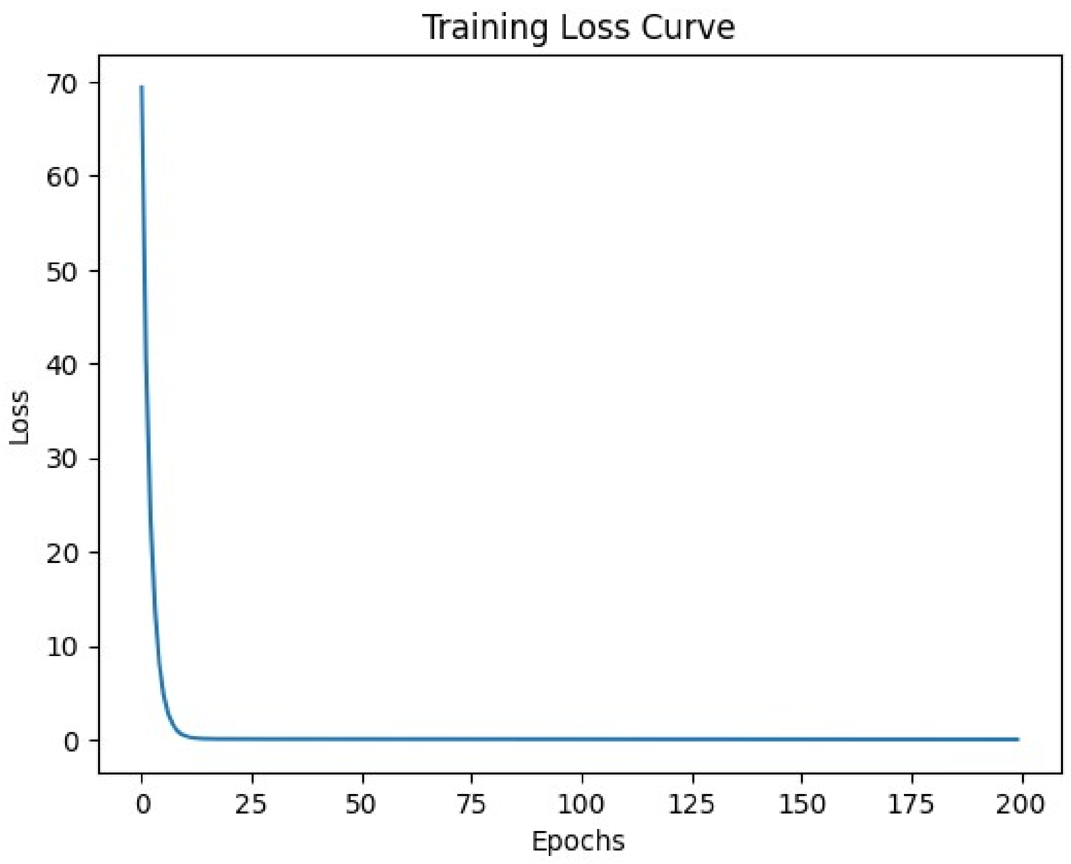
```
model.compile(
    optimizer='sgd',
    loss='mean_squared_error'
)
```

Explanation:

```
•     optimizer: How model improves (SGD = Gradient Descent)
•     loss: How wrong the prediction is

history = model.fit(x, y, epochs=200, verbose=0)
```

Explanation:

```
•     epochs: Number of times model sees data
•     history: Stores loss values

plt.plot(history.history['loss'])
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Loss Curve")
plt.show()
```



```
# plt.plot(history.history['accuracy'])
# plt.xlabel("Epoch")
# plt.ylabel("Accuracy")
# plt.title("Training Accuracy")
# plt.show()

import numpy as np

print(model.predict(np.array([[10]])))

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
[[19.41911]]
```

PyTorch

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

x = torch.tensor([[1.0],[2.0],[3.0],[4.0],[5.0]])
y = torch.tensor([[2.0],[4.0],[6.0],[8.0],[10.0]])
```

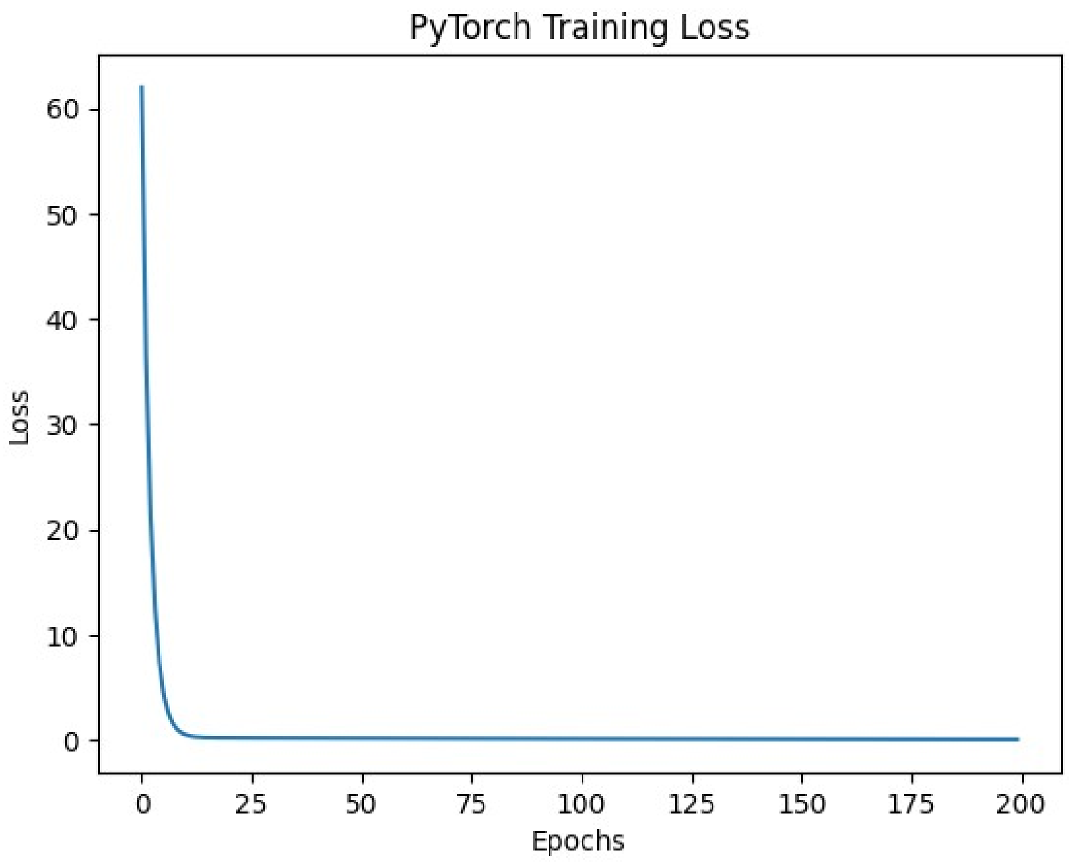
```
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1,1)

    def forward(self, x):
        return self.linear(x)

model = SimpleModel()
```

Explanation:

```
•     nn.Module: Base class for models
•     Linear(1,1): One neuron

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

losses = []

for epoch in range(200):
    optimizer.zero_grad()
    output = model(x)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
```

Explanation:

```
•     backward(): Calculates gradients
•     step(): Updates weights

plt.plot(losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("PyTorch Training Loss")
plt.show()
```



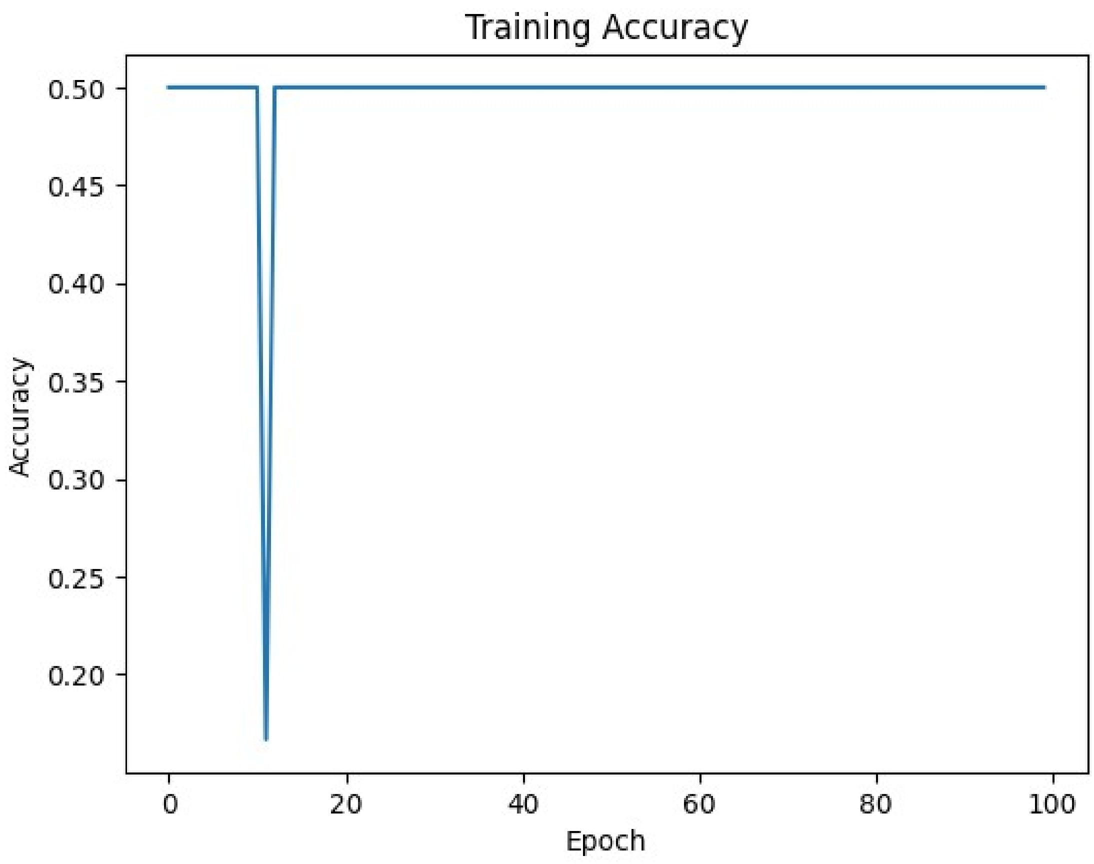
```
# plt.plot(history.history['accuracy'])
# plt.xlabel("Epoch")
# plt.ylabel("Accuracy")
# plt.title("Training Accuracy")
# plt.show()

print(model(torch.tensor([[10.0]])))
```

```
tensor([[18.9556]], grad_fn=<AddmmBackward0>)
```

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Data (classification)
x = np.array([1,2,3,6,7,8], dtype=float)
y = np.array([0,0,0,1,1,1], dtype=float)

# Model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, activation='sigmoid', input_shape=[1])
])

# Compile (IMPORTANT)
```

```python
model.compile(
    optimizer='sgd',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Train
history = model.fit(
    x, y,
    epochs=100,
    verbose=0
)

# Plot Accuracy
plt.plot(history.history['accuracy'])
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Training Accuracy")
plt.show()
```

```
c:\Users\devansh\OneDrive\Desktop\Lib\site-packages\keras\src\layers\
core\dense.py:92: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```


Training Accuracy
Accuracy
Epoch
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
20
40
60
80
100