AI Frameworks: Origins & Hello World

Silvano Piazza

2025/11/12

CONTENTS

CONTENTS

01

Birth of TensorFlow

Google's 2015 Gift to Deep Learning

TensorFlow:

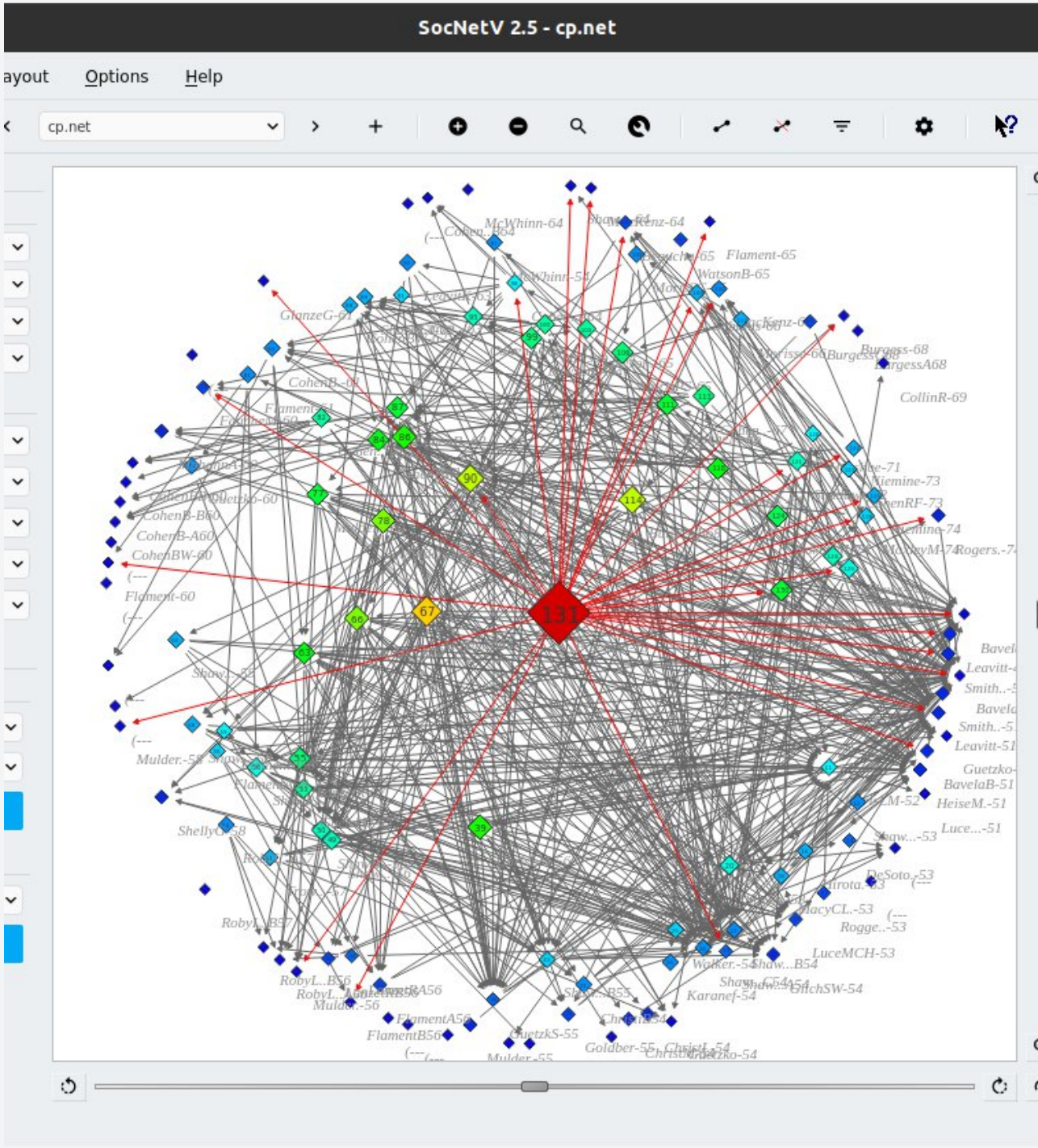
A New Paradigm for Large-Scale Machine Learning

Static Dataflow Graphs

Introduced a novel way to define computations as graphs, enabling aggressive global optimization before execution.

Cross-Platform Portability

Designed for seamless deployment, from large-scale server clusters to resource-constrained mobile devices.



Keras: An API Designed for Human Beings

Released by **François Chollet** in March 2015, Keras prioritized developer happiness with a declarative, modular style. Its focus on simplicity and ease of use made it the perfect high-level front-end, eventually merging into TensorFlow 2.0 as its default API.

2015
Keras Born

2019
Merged into TF 2.0

02

Rise of PyTorch

Facebook Rewrites Torch in Python

PyTorch's Dynamic Revolution

Lua-based Torch
The Predecessor

→

PyTorch (2016)
Dynamic Computation Graphs inspired by **Chainer**, allowing for intuitive debugging and rapid experimentation.

→

Research Dominance
The New Standard

By adopting dynamic graphs, PyTorch turned model debugging into standard Python `print` statements, fueling a new era of agile research and development.

fastai Democratizes PyTorch

Released by **Jeremy Howard** and **Rachel Thomas** in October 2018, fastai wraps PyTorch best practices into a layered, teachable API.

- **Layered Library:** Offers high-level abstractions while allowing deep customization by peeling back layers.
- **Free Education:** Powers the popular fast.ai course, enabling domain experts to achieve state-of-the-art results.

03

Design Philosophy

Static vs. Dynamic Graphs: A Spectrum of Choice

Static Graphs (TensorFlow 1.x)

Define, compile, then execute. This approach trades flexibility for performance and optimization.

Define → Compile → Execute

VS

Dynamic Graphs (PyTorch)

Define the graph on-the-fly during execution, offering unparalleled flexibility and debuggability.

Execute ⟳ Define

Modern frameworks are converging: TensorFlow 2.x adds eager mode, PyTorch adds TorchScript, illustrating a spectrum of choice, not a binary one.

The Layered Abstraction Stack

Modern frameworks are built in layers, allowing users to operate at the level of abstraction that best suits their needs.

High-Level APIs (Keras, fastai)

Reusable building blocks and best-practice defaults for rapid prototyping and education.

Mid-Level Engines (TensorFlow, PyTorch)

Core engines that schedule operations and manage hardware acceleration.

Low-Level Kernels (C++/CUDA)

High-performance tensor operations optimized for specific hardware.

04

Dummy Example Spec

Toy Problem Definition

A controlled experiment to compare framework syntax and workflow.

The Dataset

200 points from y = 3x + 2 with Gaussian noise.

The Goal

Learn slope (3.0) and intercept (2.0) within a 0.05 tolerance using MSE loss.

The Optimizer

200 epochs with Adam or SGD (lr=0.1).

The Control

Same random seed for fair convergence comparison.

05

Code Walk-Through

TensorFlow 2: Raw Gradient Tape

Explicit control for production pipelines.

1. Import & Define: Import `tensorflow` and create `tf.Variable` for `w` and `b`.

2. Build Dataset: Create `tf.constant` for X and calculate Y = w*X + b + noise.

3. GradientTape Loop: Inside a loop, open `tf.GradientTape()`, compute MSE loss, and apply gradients using `Adam`.

4. Result: After 200 steps, print final `w` ≈ 3.0 and `b` ≈ 2.0.

```python
import tensorflow as tf

# Data
X = tf.constant([[1.0], [2.0], [3.0]])
y = 3 * X + 2

# Initialize parameters manually
W = tf.Variable(tf.random.normal([1, 1]))
b = tf.Variable(tf.zeros([1]))

# Define the model
def model(x):
    return tf.matmul(x, W) + b

# Mean Squared Error loss
def loss_fn(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))

# Optimizer
optimizer = tf.optimizers.Adam(learning_rate=0.1)

# Training loop
for step in range(200):
    with tf.GradientTape() as tape:
        pred = model(X)
        loss = loss_fn(y, pred)

    gradients = tape.gradient(loss, [W, b])
    optimizer.apply_gradients(zip(gradients, [W, b]))

# Final parameters
print("Final W:", W.numpy())
print("Final b:", b.numpy())
```

```python
import tensorflow as tf
from tensorflow import keras

# Data
X = tf.constant([[1.0], [2.0], [3.0]])
y = 3 * X + 2

# Model
model = keras.Sequential([
    keras.layers.Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Train
model.fit(X, y, epochs=200)
```

Keras: The Sequential One-Liner

Model-centric abstraction for developer happiness.

1. Define Model: Instantiate a `Sequential` model with a single `Dense(1)` layer.

2. Compile: Configure the model with `optimizer='adam'` and `loss='mse'`.

3. Fit: Train the model on the dataset for 200 epochs with a single `fit()` call.

4. Result: Extract and print the learned weight and bias, achieving the same result in just 4 statements.

PyTorch: The Dynamic Loop

Imperative style for agile research.

1. Define Model: Create `torch.tensor` for X and Y, and instantiate `nn.Linear(1,1)`.

2. Setup: Choose `MSELoss` and `torch.optim.SGD` as the optimizer.

3. Manual Loop: For 200 iterations, manually `zero_grad`, `forward` pass, `backward`, and `step`.

4. Result: Print learned parameters, converging to slope 3.0 and bias 2.0.

```python
import torch

# Data
X = torch.tensor([[1.0], [2.0], [3.0]])
y = 3 * X + 2

# Initialize parameters
W = torch.randn(1, 1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# Learning rate
lr = 0.1

# Training loop
for step in range(200):
    # Forward pass
    y_pred = X @ W + b

    # Loss: Mean Squared Error
    loss = ((y_pred - y)**2).mean()

    # Backpropagation
    loss.backward()

    # Gradient descent update
    with torch.no_grad():
        W -= lr * W.grad
        b -= lr * b.grad

    # Reset gradients
    W.grad.zero_()
    b.grad.zero_()

print("Final W:", W.detach().numpy())
print("Final b:", b.detach().numpy())
```

```python
from fastai.data.core import DataLoaders, TensorData
from fastai.tabular.all import tabular_learner
import torch

# Data
X = torch.tensor([[1.0], [2.0], [3.0]])
y = 3 * X + 2

# Build DataLoaders
dls = DataLoaders.from_dsets(TensorData(X), TensorData(y), bs=3)

# Create a simple learner with a linear model
learn = tabular_learner(dls, layers=[1])

# Train
learn.fit_one_cycle(20)
```

fastai: The High-Level Learner

Ultimate productivity with best-practice defaults.

1. DataLoaders: Wrap numpy arrays into fastai's `DataLoaders` object with a batch size.

2. Learner: Instantiate a `tabular_learner` with an empty `layers` list for a linear model.

3. Train: Use the `fit_one_cycle` policy for 20 epochs with a single training call.

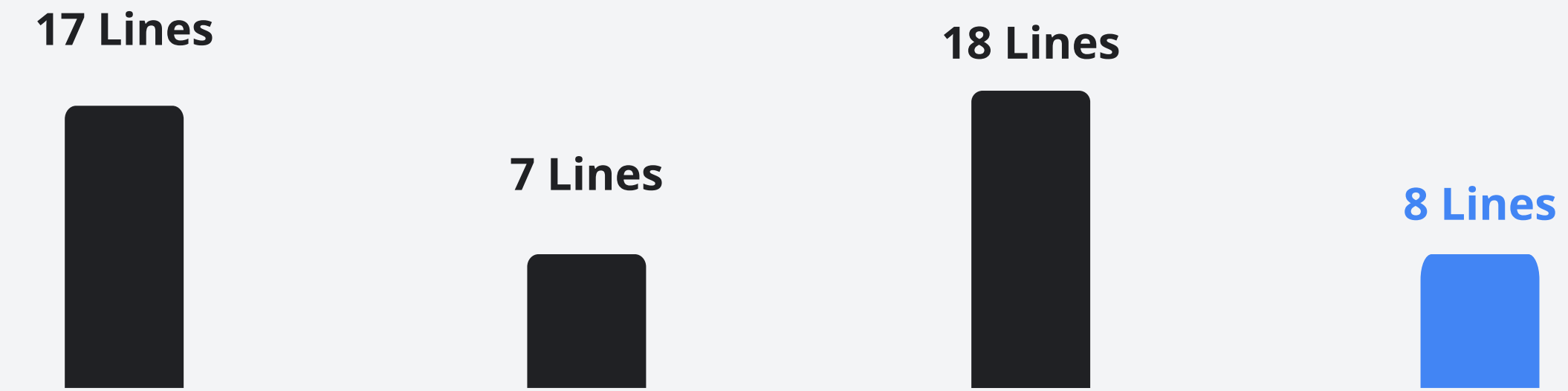
4. Result: Retrieve and print the learned parameters, demonstrating extreme conciseness.

06

Side-by-Side Review

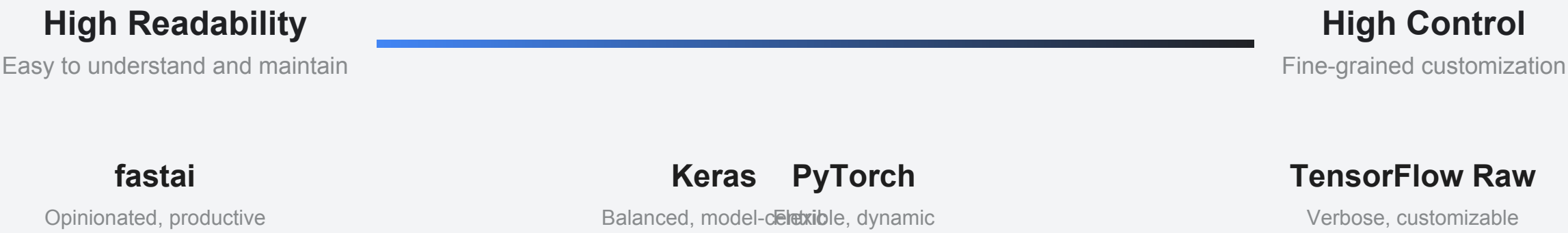
The Lines-to-Accuracy Trade-off

All frameworks solve the same problem with the same accuracy, but with vastly different levels of abstraction.



The choice is a spectrum of **control** vs. **readability**.

Readability vs. Control Axis



The optimal choice depends on your project constraints, team skill, and the balance between deadline and future maintenance.

07

Ecosystem & Future

Ecosystem & Community Momentum

Each framework has carved out a dominant niche, making the choice as much about the ecosystem as the API.

- **TensorFlow:** Dominates production serving, mobile (TF Lite), and Google Cloud integration.
- **PyTorch:** Rules academic research and powers the Hugging Face ecosystem.
- **Keras:** Educates millions via Coursera and remains the standard entry point for beginners.
- **fastai:** Champions ethical AI and free, practical education for domain experts.

Convergence and Specialization

The lines between frameworks are blurring, leading to a future of multiple front-ends and optimized backends.

Convergence

TF has eager mode, PyTorch has TorchScript. Both support ONNX. JAX introduces new paradigms.

Specialization

Frameworks expose multiple front-ends (dynamic for research, static for deployment).

Compilation

Compilers like XLA, TVM, and TorchDynamo automate target-specific optimization.

The future is a **multi-framework**, **multi-backend** ecosystem, letting users focus on model design, not graph mechanics.

08

Key Takeaways

Practitioner Decision Map

1. **Start High:** Begin with high-level APIs (Keras, fastai) to validate ideas quickly. They are the fastest path to results.
2. **Descend Gradually:** Dive lower only when customization or performance demands it. Don't optimize prematurely.
3. **Master One, Know Others:** Develop deep expertise in one ecosystem, but keep an experimental eye on the others.
4. **Focus on Math:** Frameworks are just tools. Solid mathematical understanding is the true foundation.

Further Learning Resources

Accelerate your informed framework selection with hands-on practice.

Official Docs & Notebooks

Start with the source. Reproduce the "Hello World" example in each framework.

Profile & Compare

Measure speed, memory usage, and ease of debugging for a small real-world dataset.

Join the Community

Engage with courses like Coursera (TF) or fast.ai, and follow Papers With Code.