TensorFlow vs PyTorch for PINN

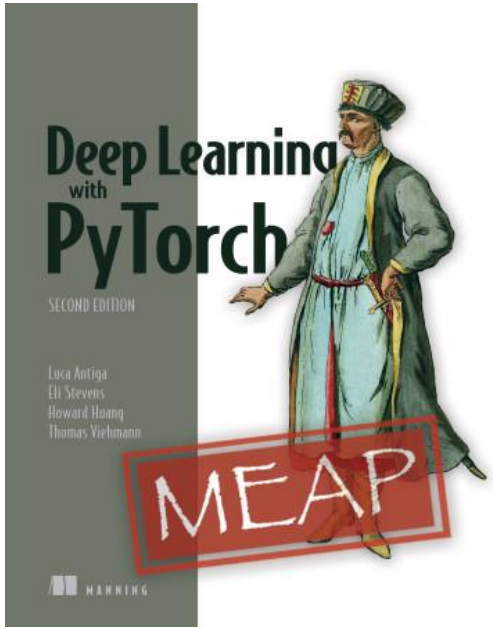
Panchatcharam Mariappan

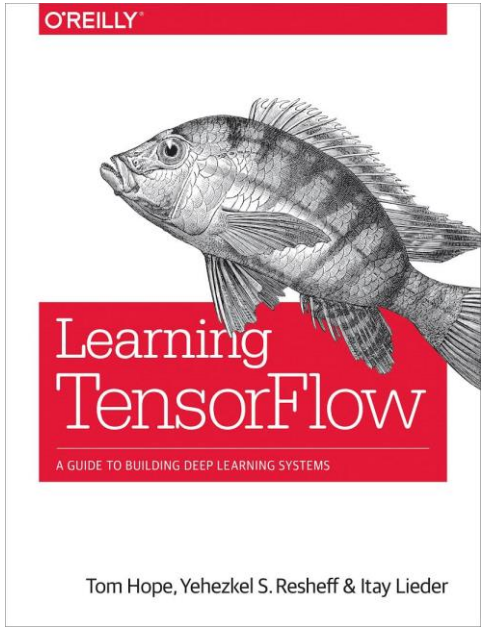
Associate Professor

Department of Mathematics and Statistics,
IIT Tirupati

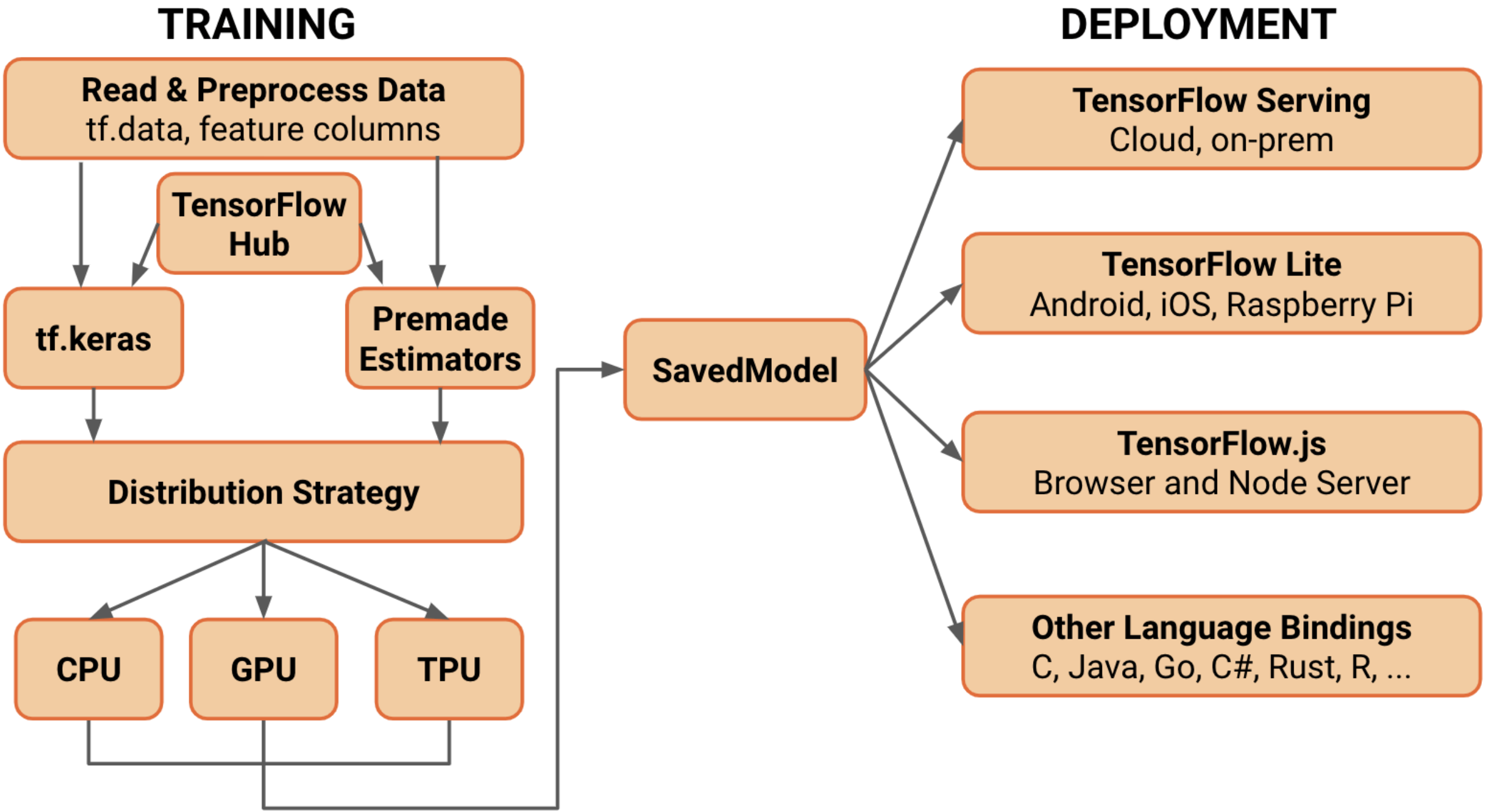
References

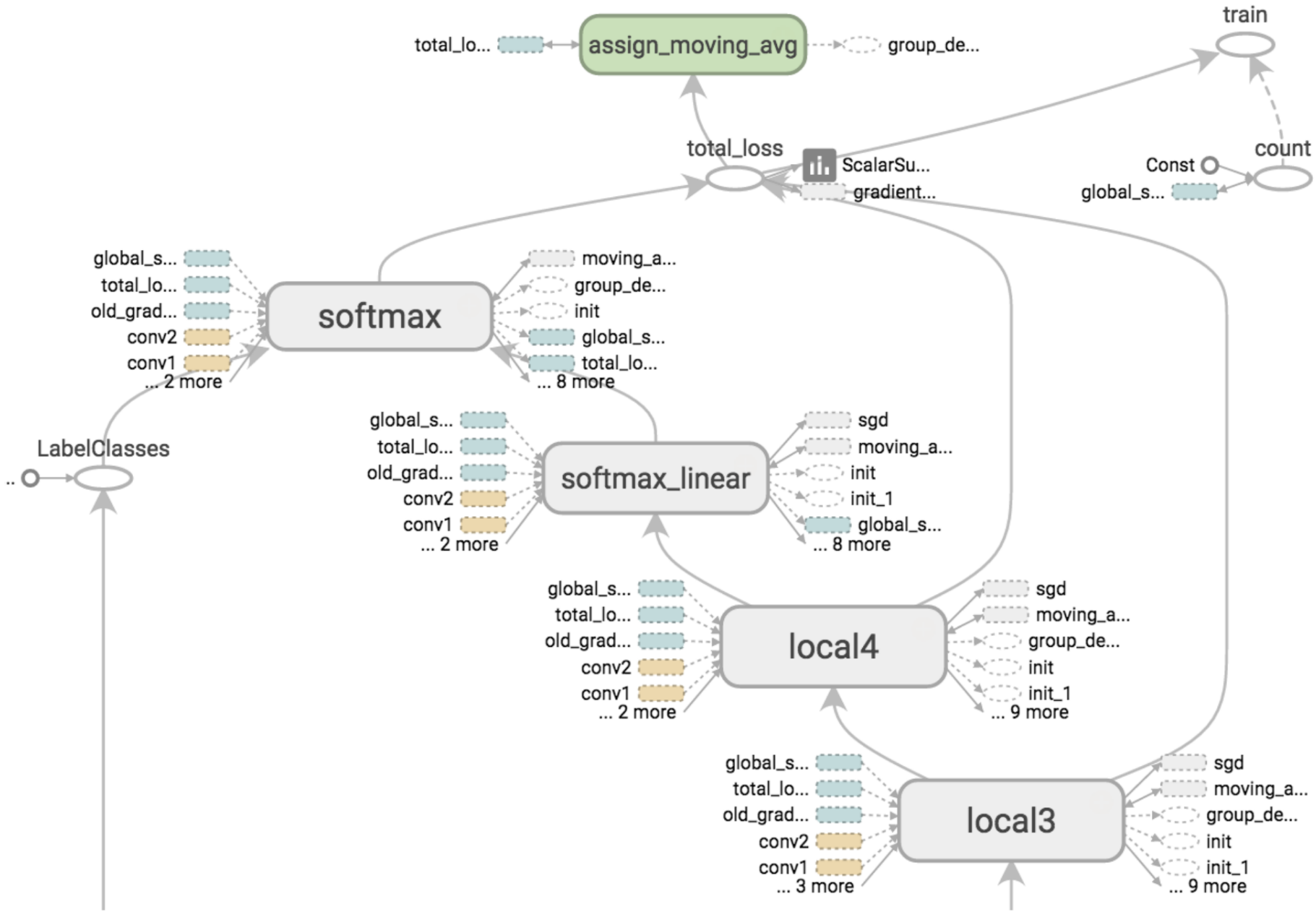
- Deep Learning with PyTirch, Luca, et al
- Learning Tensorflow, Tom Hope, et al
- https://www.tensorflow.org/tutorials
- https://docs.pytorch.org/docs/stable/index.html





TensorFlow


TRAINING
Read & Preprocess Data
tf.data, feature columns
TensorFlow Hub
tf.keras
Premade Estimators
Distribution Strategy
CPU
GPU
TPU
SavedModel
DEPLOYMENT
TensorFlow Serving
Cloud, on-prem
TensorFlow Lite
Android, iOS, Raspberry Pi
TensorFlow.js
Browser and Node Server
Other Language Bindings
C, Java, Go, C#, Rust, R, ...


train
total_lo...
assign_moving_avg
group_de...
total_loss
ScalarSu...
gradient...
Const
count
global_s...
global_s...
total_lo...
old_grad...
conv2
conv1
... 2 more
softmax
moving_a...
group_de...
init
global_s...
total_lo...
... 8 more
LabelClasses
..
global_s...
total_lo...
old_grad...
conv2
conv1
... 2 more
softmax_linear
sgd
moving_a...
init
init_1
global_s...
... 8 more
global_s...
total_lo...
old_grad...
conv2
conv1
... 2 more
local4
sgd
moving_a...
group_de...
init
init_1
... 9 more
global_s...
total_lo...
old_grad...
conv2
conv1
... 3 more
local3
sgd
moving_a...
group_de...
init
init_1
... 9 more

TensorFlow

Tensors = multidimensional arrays.

TensorFlow (TF) is a **numerical computation library** based on *dataflow graphs*.

A TensorFlow program describes **how data moves between operations**, each represented as a node in a graph.

Everything in TensorFlow is represented as a tensor and manipulated through computational graphs.

TensorFlow

Tensors = multidimensional arrays.

TensorFlow is an **open-source machine learning framework** developed by **Google Brain**. It is used for building and deploying machine learning (ML) and deep learning (DL) models at scale.

Everything in TensorFlow is represented as a tensor and manipulated through computational graphs.

Features

Computational Graphs

- Static graph (TF1.x)
- Eager execution (TF2.x)
- Optimized execution through `tf.function()`

Multi-Device Execution

TensorFlow automatically uses:

- CPU
- GPU
- TPU (Tensor Processing Unit)
- Device placement is automatic but can be controlled manually.

Keras

TF ships with **Keras**, a user-friendly tool for building neural networks.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

Automatic Differentiation

Through `tf.GradientTape()` TensorFlow automatically computes derivatives.

Distributed Training

Supports:

- Data parallelism
- Model parallelism
- Multi-GPU / Multi-TPU clusters

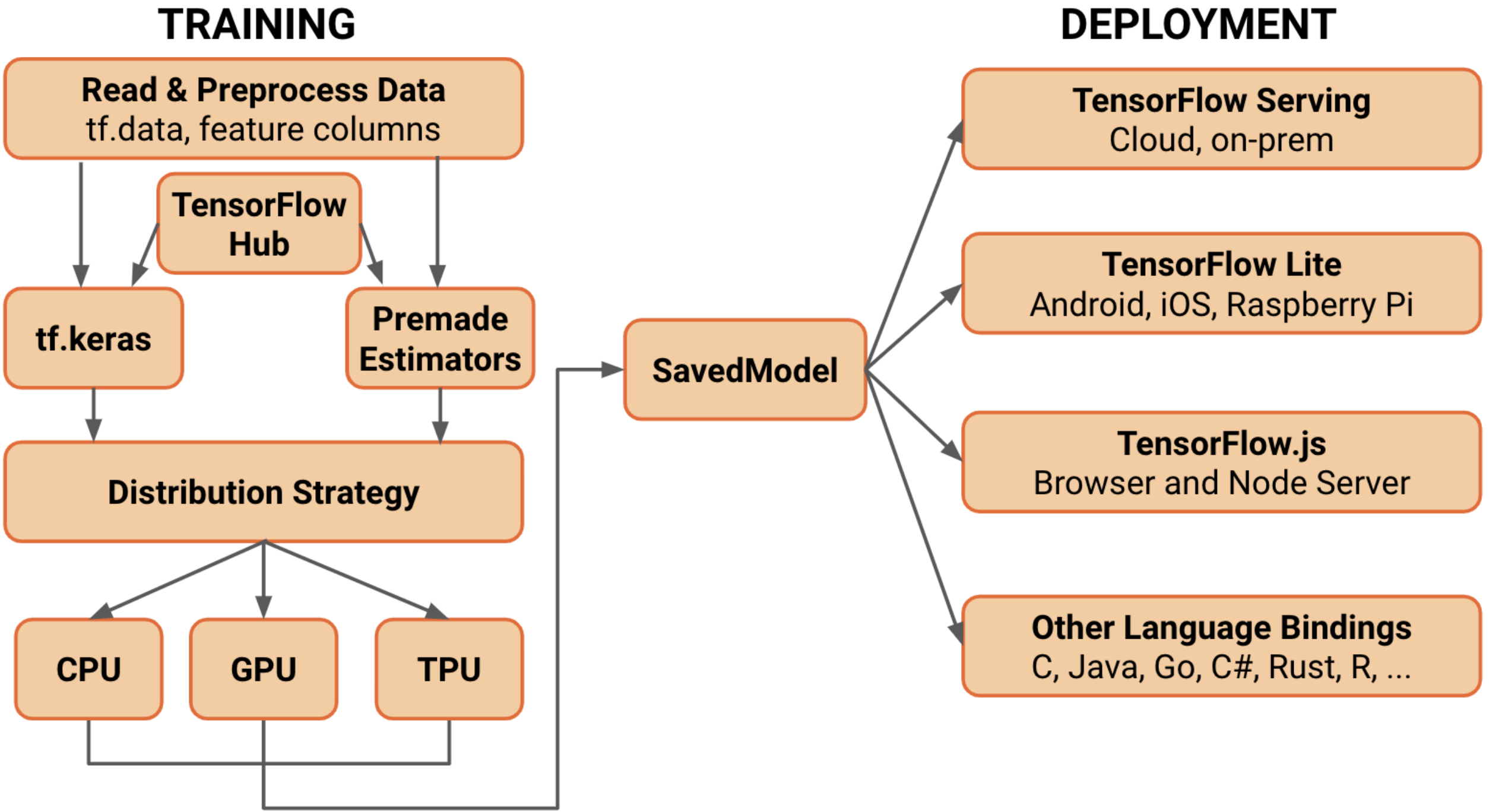
Frameworks:

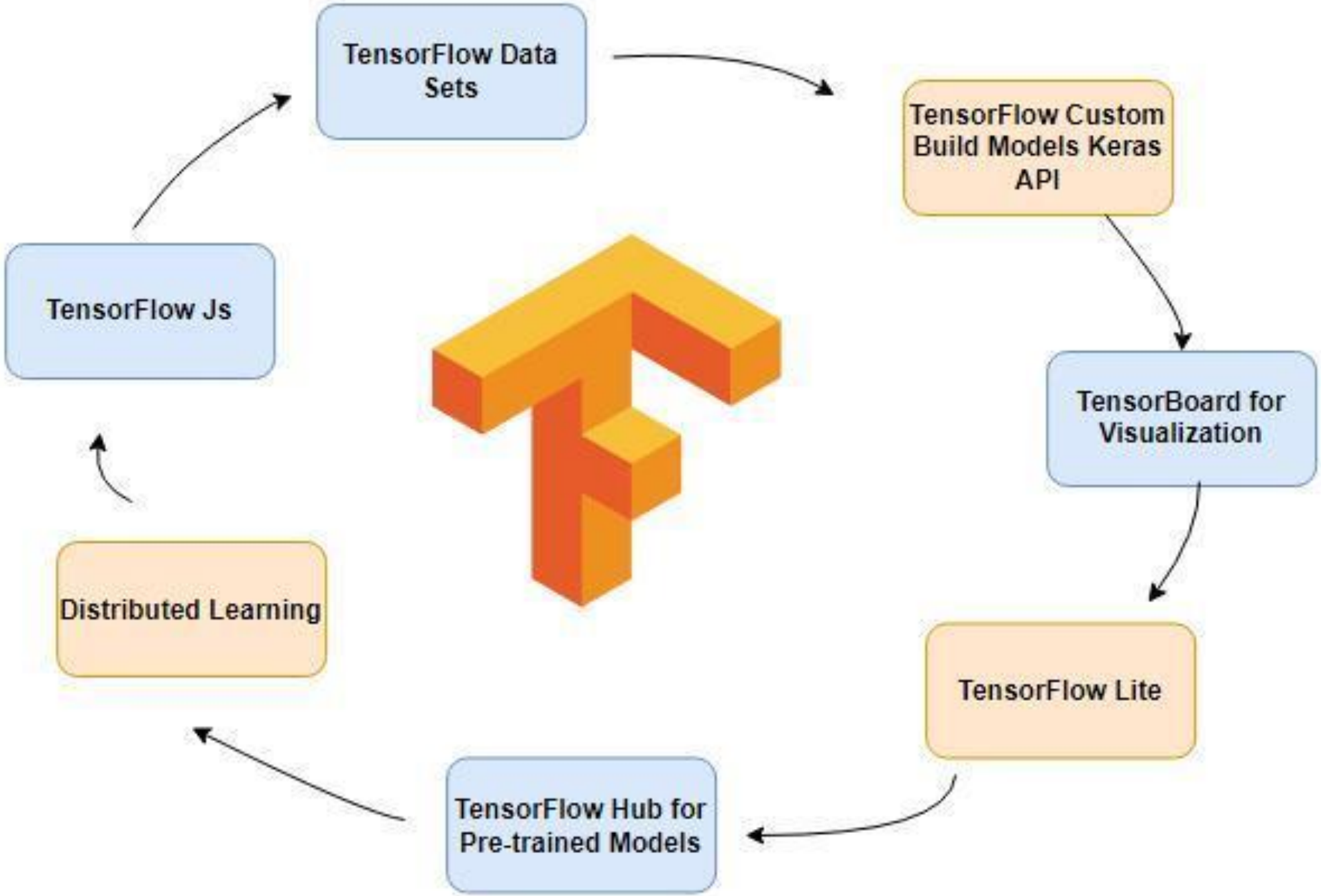
- MirroredStrategy
- MultiWorkerMirroredStrategy

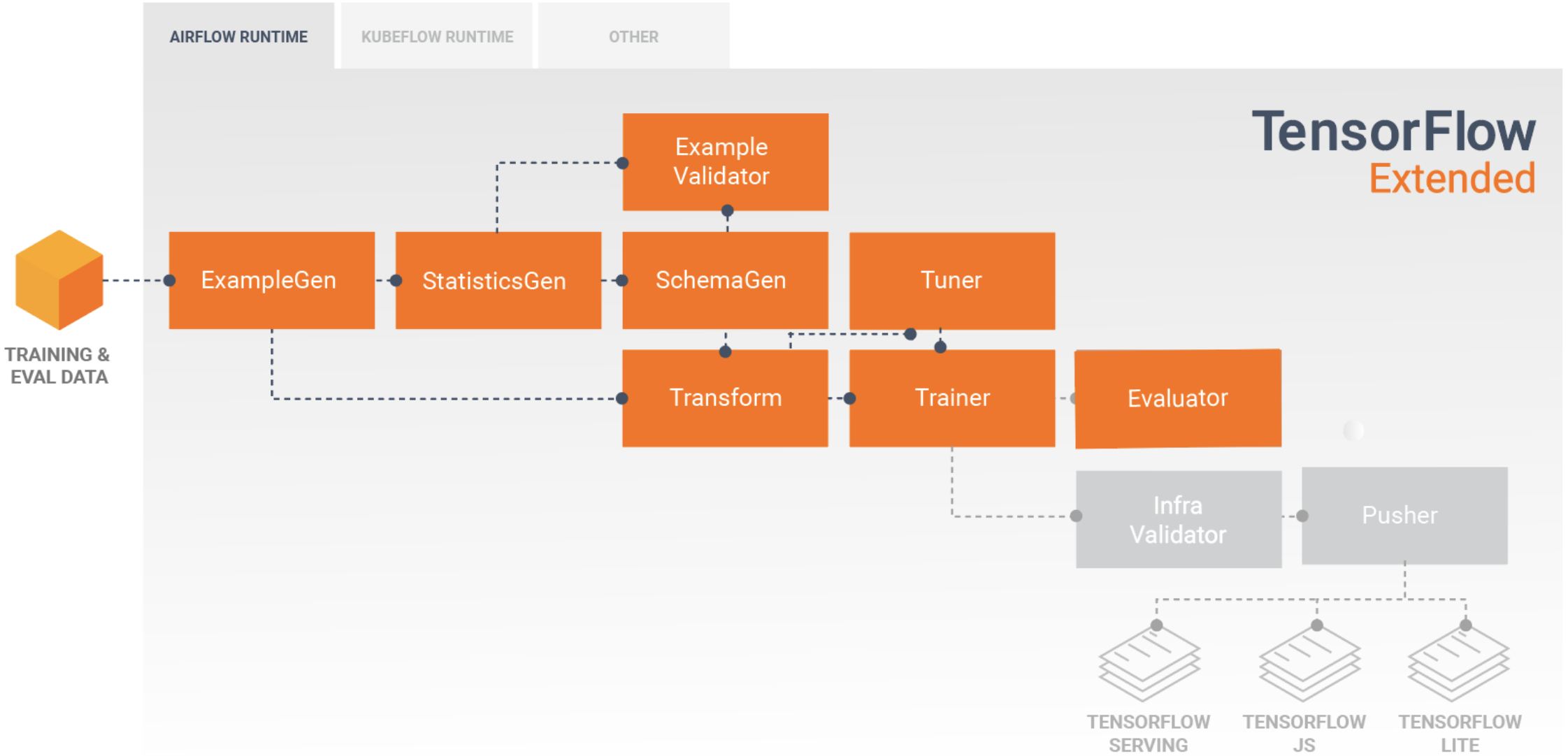
Deployment Ecosystem

- TensorFlow Lite → mobile/IoT
- TensorFlow.js → browser
- TensorFlow Serving → production models
- TFX pipelines → end-to-end ML workflow

TensorFlow Architecture


TRAINING
Read & Preprocess Data
tf.data, feature columns
TensorFlow Hub
tf.keras
Premade Estimators
Distribution Strategy
CPU
GPU
TPU
SavedModel
DEPLOYMENT
TensorFlow Serving
Cloud, on-prem
TensorFlow Lite
Android, iOS, Raspberry Pi
TensorFlow.js
Browser and Node Server
Other Language Bindings
C, Java, Go, C#, Rust, R, ...


TensorFlow Data Sets
TensorFlow Custom Build Models Keras API
TensorBoard for Visualization
TensorFlow Lite
TensorFlow Hub for Pre-trained Models
Distributed Learning
TensorFlow Js


AIRFLOW RUNTIME
KUBEFLOW RUNTIME
OTHER
TensorFlow
Extended
TRAINING &
EVAL DATA
ExampleGen
StatisticsGen
Example
Validator
SchemaGen
Tuner
Transform
Trainer
Evaluator
Infra
Validator
Pusher
TENSORFLOW
SERVING
TENSORFLOW
JS
TENSORFLOW
LITE

How TensorFlow Works (Step-by-Step)

Step 1: **Define a computation graph**

Nodes = operations
Edges = tensors flowing between operations

Step 2: **Execute graph**

Backends like XLA compile and optimize the graph.

Step 3: **Automatic differentiation**

Used for backpropagation in neural networks.

TensorFlow: Programming Model

Tensors

```
x = tf.constant([[1.0, 2.0], [3.0, 4.0]])
```

Variables

```
w = tf.Variable(tf.random.normal([2, 2]))
```

Gradient Tape

```
with tf.GradientTape() as tape:
    y = model(x)
grads =
tape.gradient(y,model.trainable_variables)
```

TensorFlow vs PyTorch

Feature	TensorFlow	PyTorch
Execution	Graph + Eager	Eager by default
Deployment	Strong (TFX, Lite)	Improving (TorchServe)
Mobile	TensorFlow Lite	PyTorch Mobile
TPU support	Excellent	Limited
Research flexibility	Moderate	Very flexible

TensorFlow is strong in **production**;
PyTorch is strong in **research**.

TensorFlow for PINNs and PDEs

TensorFlow is commonly used for:

- Physics-Informed Neural Networks (PINNs)
- Auto-differentiation of PDE residuals
- High-order derivatives
- PDE solvers via

```
tf.GradientTape(persistent=True)
```

TensorFlow for PINNs and PDEs

```
with tf.GradientTape(persistent=True) as t2:
    with tf.GradientTape(persistent=True) as t1:
        u = model(x)
    du_dx = t1.gradient(u, x)
d2u_dx2 = t2.gradient(du_dx, x)
```

When to Use TensorFlow

- You want mobile/web deployment
- You need TPU acceleration
- You are developing production-grade models
- You want strong pipeline support (TFX)
- You work with PINNs or differential programming

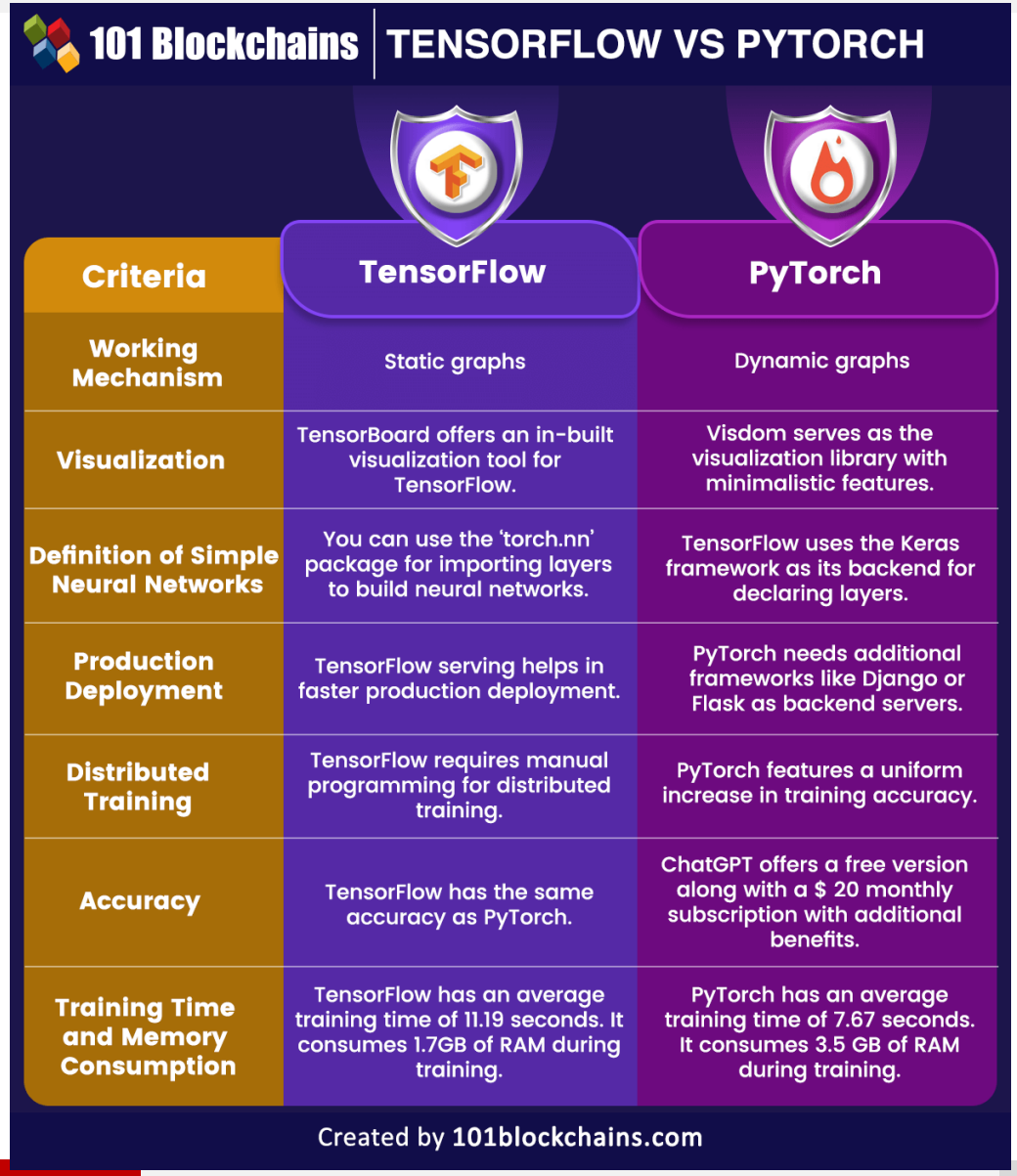
When Not to Use TensorFlow

Consider PyTorch if:

- You do rapid prototyping
- You want very flexible autograd
- Your workflow is purely research (academia)

TensorFlow vs PyTorch

TensorFlow vs PyTorch



TensorFlow vs PyTorch

- Originally designed for production-scale deployment.
- Uses static graphs (TF1) + eager execution (TF2).
- Strong integration with Google's ecosystem (TPU, TFX, TensorFlow Lite).

TensorFlow

- Designed for research flexibility.
- Uses eager execution by default (define-by-run).
- Most popular in academia and rapid experimentation.

PyTorch

TensorFlow → production-first
PyTorch → research-first

Programming: TensorFlow vs PyTorch

- Graph-based;
- optimized execution
- Functions wrapped with @tf.function
- Requires more boilerplate

TensorFlow

- Pythonic, easy-to-read code
- Dynamic graph: operations executed immediately
- Very intuitive for debugging

PyTorch

PyTorch is easier to write and debug.
TensorFlow is more optimized for execution.

Model: TensorFlow vs PyTorch

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1)
])
```

TensorFlow

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        return self.fc2(F.relu(self.fc1(x)))
```

Keras = simpler
PyTorch = more control & transparency

PyTorch

AutoGrad: TensorFlow vs PyTorch

- Uses `tf.GradientTape`
- Supports higher-order derivatives
- Sometimes harder to compute complex derivatives in PDE problems

TensorFlow

torch.autograd is extremely flexible
Better for custom loss functions, PDEs, PINNs

PyTorch

PyTorch autograd is more flexible and intuitive
TensorFlow autograd is more optimized for hardware

Community and Popularity: TensorFlow vs PyTorch

Industry:
TensorFlow widely used by Google, Airbnb, Twitter, etc.

TensorFlow

Research papers:
PyTorch dominates (>80% of deep learning papers)

PyTorch

Performance: TensorFlow vs PyTorch

TensorFlow

- **Faster for static graphs**
- **XLA optimizations**
- **Better for large-scale production**

PyTorch

Slightly slower (dynamic execution)
But JIT improves performance
PyTorch 2.0 uses TorchDynamo + AOTAutograd → significant speedups

Performance gap is shrinking; both are comparable.

PINN: TensorFlow vs PyTorch

TensorFlow

- For PINNs in productionTPU acceleration for large
- PDE parameter sweeps
- TF2 + Keras = clean high-level API

PyTorch

- **Easy autograd for PDE residuals**
- **Better control over backprop graph**
- **Faster prototyping for research**
- **JAX/PyTorch preferred in recent SciML papers**

Research PINNs → PyTorch
Industrial PINNs / large simulations → TensorFlow or JAX

Summary

Feature	TensorFlow	PyTorch
Execution	Static + Eager	Dynamic
Ease of use	Moderate	Very easy
Debugging	Harder	Very easy
Autograd	Strong but complex	Best-in-class
Deployment	Excellent (TFX, Lite, JS)	Good (TorchServe, ONNX)
TPU support	Best	Moderate
Research adoption	Moderate	Very High
Production adoption	Very High	Moderate
PINN suitability	Good	Excellent for research

Which One Should I Choose?

Choose PyTorch if:

- You are doing research
- You work on PINNs, PDEs, SciML
- You want clean, Pythonic code
- You need fast debugging

Choose TensorFlow if:

- You want end-to-end ML pipelines
- You deploy to mobile or web
- You want TPU acceleration
- You work in production environments

TensorFlow CheatSheet

Install and Import

```
pip install tensorflow
```

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

Tensor Creations and Operations

```
x = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
x = tf.Variable(tf.random.normal([3, 3]))

# Basic ops
y = x + 5
z = tf.matmul(x, tf.transpose(x))
```

Build Models

```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)
])
```

Functional

```
inputs = keras.Input(shape=(10,))
h = keras.layers.Dense(32,
activation="relu")(inputs)
outputs = keras.layers.Dense(1)(h)

model = keras.Model(inputs, outputs)
```

Compile and Train

```
model.compile(
    optimizer='adam',
    loss='mse',
    metrics=['mae']
)

model.fit(X_train, y_train, epochs=10,
batch_size=32)
```

Evaluate and Predict

```
model.evaluate(X_test, y_test)
```

```
y_pred = model.predict(X_new)
```

Optimizers

```
opt = tf.keras.optimizers.Adam(learning_rate=0.001)
opt = tf.keras.optimizers.SGD(0.01, momentum=0.9)
```

Activation Functions

```
tf.nn.relu(x)
tf.nn.sigmoid(x)
tf.nn.tanh(x)
tf.nn.softmax(x)
```

```
keras.layers.Dense(64, activation='relu')
```

Loss Functions

```
keras.losses.MeanSquaredError()
keras.losses.BinaryCrossentropy()
keras.losses.CategoricalCrossentropy()
keras.losses.MeanAbsoluteError()
```

Dataset and Data Pipeline

```
dataset = tf.data.Dataset.from_tensor_slices((X, y))
dataset = dataset.shuffle(1000).batch(32).prefetch(1)
```

```
raw_ds = tf.data.TFRecordDataset('data.tfrecord')
```

Automatic Differentiation

```
x = tf.Variable(3.0)

with tf.GradientTape() as tape:
    y = x**2 + 2*x + 1

dy_dx = tape.gradient(y, x)
```

```
    grads = tape.gradient(loss,
    model.trainable_variables)
```

Training Loop

```
for epoch in range(10):
    for x_batch, y_batch in dataset:
        with tf.GradientTape() as tape:
            y_pred = model(x_batch, training=True)
            loss = tf.reduce_mean(tf.square(y_batch - y_pred))

        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

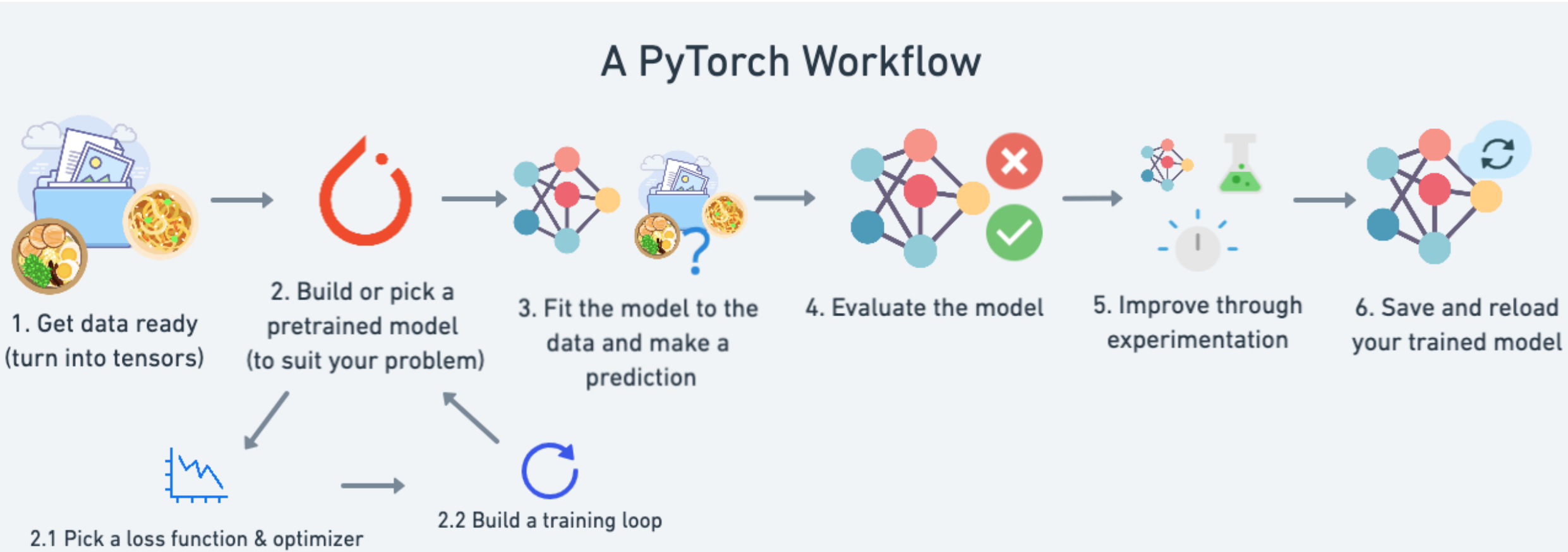
Saving and Loading

```
model.save("model.h5")
model = keras.models.load_model("model.h5")
```

Useful One-Liners

```
tf.random.set_seed(42)
tf.keras.utils.plot_model(model, show_shapes=True)
tf.reduce_sum(x)
tf.argmax(pred, axis=1)
tf.reshape(tensor, new_shape)
tf.cast(x, tf.float32)
```

PyTorch CheatSheet


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

Install and Import

```
pip install torch torchvision torchaudio
```

```
import torch
import torch.nn as nn
import torch.optim as optim
```

Tensor Creations and Operations

```
x = torch.tensor([1, 2, 3])
x = torch.zeros((3, 3))
x = torch.ones((2, 2))
x = torch.randn((4, 5))
```

```
y = x + 5
z = torch.matmul(x, x.T)
x = x.reshape(9)
```

Build Models

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        return self.fc2(torch.relu(self.fc1(x)))

model = Net()
```

Common Layers

```
nn.Linear(in_features, out_features)
nn.Conv2d(in_channels, out_channels, kernel_size)
nn.MaxPool2d(kernel_size)
nn.Dropout(p=0.5)
nn.BatchNorm1d(num_features)
```

Compile and Train

```
for epoch in range(10):
    for X, y in dataloader:
        optimizer.zero_grad()
        pred = model(X)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()

    print(epoch, loss.item())
```

Evaluate and Predict

```
model.evaluate(X_test, y_test)
```

```
y_pred = model.predict(X_new)
```

Optimizers

```
optimizer = optim.Adam(model.parameters(), lr=0.001)

optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optim.Adam(model.parameters(), lr=0.001)
optim.RMSprop(model.parameters(), lr=0.0005)
```

Activation Functions

```
nn.ReLU()
nn.Sigmoid()
nn.Tanh()
nn.Softmax(dim=1)
```

Loss Functions

```
criterion = nn.MSELoss()
nn.CrossEntropyLoss()       # multi-class
nn.BCELoss()                # binary
nn.BCEWithLogitsLoss()      # sigmoid + BCE
```

Dataset and Data Pipeline

```
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

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

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
loader = DataLoader(MyDataset(X, y),
batch_size=32, shuffle=True)
```

Automatic Differentiation

```
x = torch.randn(3, requires_grad=True)
y = x**2 + 3*x
y.sum().backward()
print(x.grad)
```

Saving and Loading

```
torch.save(model.state_dict(), "model.pth")
model = Net()
model.load_state_dict(torch.load("model.pth"))
model.eval()
```