Introduction to TensorFlow, PyTorch, JAX, and Keras

CS4152 Deep Learning and Neural Networks

Dr. Jameel Ahmad

Fall 2025
Computer Science Department
SST, UMT

Oct 21, 2025

Chapter Overview

- A closer look at all major deep learning frameworks and their relationships
- Overview of how core deep learning concepts translate to code across all frameworks
- Everything you need to start doing deep learning in practice
- Three popular frameworks that can be used with Keras:
  - TensorFlow (`https://tensorflow.org`)
  - PyTorch (`https://pytorch.org/`)
  - JAX (`https://jax.readthedocs.io/`)

Brief History of Deep Learning Frameworks

- **2009**: Theano - first framework with autodiff and GPU computation
- **2013-2014**: Torch 7 (Lua-based) and Caffe (C++-based) gain popularity
- **2015**: Keras launches as higher-level library powered by Theano
- **2015**: Google launches TensorFlow
- **2016**: Meta launches PyTorch
- **2018**: Google releases JAX

Key Features of Deep Learning Frameworks

All frameworks combine three key features:

1. **Automatic Differentiation**: Compute gradients for arbitrary differentiable functions
2. **Tensor Computation**: Run tensor operations on CPUs and GPUs
3. **Distribution**: Distribute computation across multiple devices/computers

Framework Relationships

- **Keras**: High-level framework (prefabricated building kit)
- **TensorFlow, PyTorch, JAX**: Lower-level frameworks (raw materials)
- Keras can use any of the three as backend engines

Low-level vs High-level Concepts

- **Low-level**: Tensors, tensor operations, backpropagation
- **High-level**: Layers, models, loss functions, optimizers, training loops

TensorFlow: First Steps

```python
import tensorflow as tf

# Creating tensors
x = tf.ones(shape=(2, 1))
y = tf.zeros(shape=(2, 1))
z = tf.constant([1, 2, 3], dtype="float32")

# Random tensors
random_normal = tf.random.normal(shape=(3, 1))
random_uniform = tf.random.uniform(shape=(3, 1))

# Variables (modifiable state)
v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1))
    )
v.assign(tf.ones((3, 1)))
```

TensorFlow: Gradient Computation

```python
# Using GradientTape for automatic differentiation
input_var = tf.Variable(initial_value=3.0)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)

# With constant tensors
input_const = tf.constant(3.0)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

TensorFlow: Compilation for Performance

```python
# Regular eager execution
def dense(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)

# Graph mode compilation
@tf.function
def dense_compiled(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)

# XLA compilation (even faster)
@tf.function(jit_compile=True)
def dense_xla(inputs, W, b):
    return tf.nn.relu(tf.matmul(inputs, W) + b)
```

TensorFlow: Strengths and Weaknesses

Strengths

- Fast (graph mode and XLA compilation)
- Extremely feature complete (string tensors, ragged tensors)
- Mature ecosystem for production deployment
- Excellent data preprocessing with tf.data API

Weaknesses

- Sprawling API (thousands of operations)
- Numerical API inconsistent with NumPy
- Less support on Hugging Face for latest generative AI models

PyTorch: First Steps

```python
import torch

# Creating tensors
x = torch.ones(size=(2, 1))
y = torch.zeros(size=(2, 1))
z = torch.tensor([1, 2, 3], dtype=torch.float32)

# Random tensors
random_normal = torch.normal(
    mean=torch.zeros(size=(3, 1)),
    std=torch.ones(size=(3, 1))
)
random_uniform = torch.rand(3, 1)

# Parameters (trainable state)
p = torch.nn.Parameter(data=torch.zeros(size=(2, 1)))
```

PyTorch: Gradient Computation

```python
# Computing gradients with .backward()
input_var = torch.tensor(3.0, requires_grad=True)
result = torch.square(input_var)
result.backward()
gradient = input_var.grad  # tensor(6.)

# Reset gradients
input_var.grad = None

# Multiple backward calls accumulate gradients
result = torch.square(input_var)
result.backward()
input_var.grad  # tensor(12.) - accumulated!
```

PyTorch: Using torch.nn.Module

```python
class LinearModel(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.W = torch.nn.Parameter(
            torch.rand(input_dim, output_dim))
        self.b = torch.nn.Parameter(
            torch.zeros(output_dim))

    def forward(self, inputs):
        return torch.matmul(inputs, self.W) + self.b

# Usage
model = LinearModel(2, 1)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

PyTorch: Strengths and Weaknesses

Strengths

- Easy to debug (eager execution by default)
- First-class support on Hugging Face
- Popular in research community

Weaknesses

- API inconsistent with NumPy and internally inconsistent
- Slowest of major frameworks
- torch.compile() less effective than competitors

JAX: First Steps

```python
from jax import numpy as jnp
import jax

# Creating arrays (identical to NumPy API)
x = jnp.ones(shape=(2, 1))
y = jnp.zeros(shape=(2, 1))
z = jnp.array([1, 2, 3], dtype="float32")

# Stateless random number generation
seed_key = jax.random.key(1337)
random_normal = jax.random.normal(seed_key, shape=(3,))

# Array modification (creates new array)
x = jnp.array([1, 2, 3], dtype="float32")
new_x = x.at[0].set(10)  # [10, 2, 3]
```

JAX: Gradient Computation (Metaprogramming)

```python
# Define loss function
def compute_loss(input_var):
    return jnp.square(input_var)

# Get gradient function
grad_fn = jax.grad(compute_loss)

# Compute gradient
input_var = jnp.array(3.0)
gradient = grad_fn(input_var)  # 6.0

# Get both value and gradient
value_and_grad_fn = jax.value_and_grad(compute_loss)
value, gradient = value_and_grad_fn(input_var)
```

JAX: Compilation and Training

```python
@jax.jit  # XLA compilation
def training_step(inputs, targets, W, b):
    def compute_loss(state, inputs, targets):
        W, b = state
        predictions = jnp.matmul(inputs, W) + b
        loss = jnp.mean(jnp.square(targets - predictions))
        return loss

    grad_fn = jax.value_and_grad(compute_loss)
    loss, grads = grad_fn((W, b), inputs, targets)
    grad_wrt_W, grad_wrt_b = grads

    W = W - grad_wrt_W * learning_rate
    b = b - grad_wrt_b * learning_rate
    return loss, W, b
```

JAX: Strengths and Weaknesses

Strengths

- Fastest of all frameworks for most models
- NumPy-compatible API
- Best for TPU training and large-scale models
- Functional, stateless design enables better compilation

Weaknesses

- Harder to debug (metaprogramming + compilation)
- More verbose low-level training loops
- Steeper learning curve

Keras: High-Level Deep Learning API

- Released in March 2015 (oldest among the four)
- Used by Google, Netflix, Uber, NASA, Waymo, etc.
- Provides convenient way to define and train deep learning models
- Supports multiple workflows for different user profiles
- **Pluggable backends**: TensorFlow, PyTorch, or JAX

Backend Configuration

- Set environment variable: `KERAS_BACKEND=jax`
- Or edit config file: `~/.keras/keras.json`
- Code compatible with all backends

Keras: Building Layers

```python
import keras

class SimpleDense(keras.Layer):
    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(
            shape=(input_dim, self.units),
            initializer="random_normal"
        )
        self.b = self.add_weight(
            shape=(self.units,),
            initializer="zeros"
        )

    def call(self, inputs):
        y = keras.ops.matmul(inputs, self.W) + self.b
```

Keras: Model Configuration and Training

```
from keras import models, layers

# Build model
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(1)
])

# Configure learning process
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
    loss=keras.losses.MeanSquaredError(),
    metrics=[keras.metrics.BinaryAccuracy()]
)

# Train model
history = model.fit(
    inputs, targets,
    epochs=5,
    batch_size=128,
```

Keras: Key Concepts

- **Layers**: Fundamental building blocks (Dense, Conv2D, LSTM, etc.)
- **Models**: Graphs of layers (Sequential, Functional API, Subclassing)
- **Loss Functions**: Quantity to minimize during training
- **Optimizers**: How network updates based on loss (SGD, Adam, RMSprop)
- **Metrics**: Measures of success to monitor (accuracy, precision, recall)
- **Training Loop**: Mini-batch gradient descent (handled by fit())

Choosing the Right Framework

TensorFlow

- Production deployment, mobile/embedded systems
- Large-scale distributed training
- When you need extensive ecosystem tools

PyTorch

- Research and experimentation
- When you need access to latest models on Hugging Face
- When ease of debugging is priority

JAX

- Maximum performance, especially on TPUs
- Large-scale models and distributed computing

Summary

- **TensorFlow, PyTorch, JAX**: Low-level frameworks for numerical computation and autodifferentiation
- **Keras**: High-level API for building and training neural networks
- All frameworks provide: Automatic differentiation, tensor computation, distribution
- Choose framework based on: Use case, performance needs, ecosystem requirements
- Keras provides backend flexibility and high-level abstractions
- Understanding all frameworks makes you a more versatile deep learning practitioner

Next Chapter Preview: Classification and Regression

- Real-world machine learning workflows
- Binary classification: Classifying movie reviews as positive/negative
- Categorical classification: Classifying news wires by topic
- Scalar regression: Estimating house prices
- Data preprocessing, model architecture, evaluation