AI Frameworks: Origins & Hello World – Speaker Notes

Slide 1: Title Slide

- **Title:** *AI Frameworks: Origins & Hello World*
- **Presenter:** Silvano Piazzq
- **Date:** November 12, 2025

This opening slide presents the title of the talk along with the presenter's name and the date. As the introduction, briefly welcome the audience and explain that the session will explore the evolution of major AI frameworks and a simple "Hello World" example across them. The goal is to set the stage for why understanding these frameworks (TensorFlow, PyTorch, Keras, fastai) is important for deep learning practitioners.

Slide 2: Contents (Part 1)

- **Birth of TensorFlow:** Introduction to Google's first major deep learning framework and its significance in 2015.
- **Rise of PyTorch:** How Facebook's PyTorch emerged in 2016 and rapidly became popular in research.
- **Design Philosophy:** A look at the core design differences (static vs. dynamic computation graphs and abstraction layers) among frameworks.
- **Dummy Example Specification:** Definition of a simple linear regression experiment that will be used to compare the frameworks.
- **Code Walk-Through:** Step-by-step "Hello World" implementation in each framework to illustrate their syntax and workflow.
- **Side-by-Side Review:** A comparative analysis of the frameworks in terms of code complexity and level of control vs. abstraction.

Part 1 of the presentation covers the historical context and technical foundations of each framework, leading up to a hands-on comparison using a toy example.

Slide 3: Contents (Part 2)

- **Ecosystem & Future:** Discussion of each framework's community adoption, ecosystem strengths, and emerging trends pointing to the future of AI development.
- **Key Takeaways:** Concluding insights and practical guidance for choosing frameworks and further learning resources for practitioners.

Part 2 focuses on the current landscape and future outlook of these frameworks, concluding with actionable advice and learning resources.

Slide 4: Birth of TensorFlow

- **TensorFlow's Origin:** TensorFlow was launched by Google in November 2015 (successor to an earlier system called DistBelief). It introduced the concept of representing computations as **static dataflow graphs**, which was a new paradigm for large-scale machine learning. This design allowed the framework to pre-optimize the entire computation graph before running it, enabling efficient distributed training and deployment at scale.
- **Static Dataflow Graphs:** In TensorFlow 1.x, you first define the computation graph, then compile it, and finally execute it. This static graph approach sacrifices some flexibility (you cannot easily change the graph during runtime) in exchange for performance optimizations. By compiling the graph ahead of time, TensorFlow can achieve aggressive global optimizations, making it well-suited for production environments where efficiency is crucial.
- **Cross-Platform Portability:** TensorFlow was built with portability in mind. It can run on a variety of platforms ranging from large server clusters (for heavy-duty model training) to resource-constrained mobile and embedded devices. This cross-platform compatibility helped TensorFlow gain wide adoption in both industry and academia, as models could be trained in the cloud and then deployed to devices like smartphones (e.g., via TensorFlow Lite).
- **Keras Integration:** *"Keras: an API designed for human beings"* – Keras is a high-level neural network library released by François Chollet in 2015. It emphasizes user-friendliness and a modular, declarative approach to building models. Initially independent, Keras prioritized developer experience, allowing beginners to construct neural networks with minimal code. In 2019, Keras was integrated into TensorFlow 2.0 as the official high-level API. This integration married TensorFlow's powerful execution engine with Keras's simplicity, lowering the entry barrier for new users. As a result, developers can enjoy the ease-of-use of Keras while still tapping into TensorFlow's optimized backend.

Slide 5: Rise of PyTorch

- **PyTorch's Origin:** PyTorch was developed by Facebook AI Research and released in September 2016 as a Python-based successor to the Lua-based Torch library. It introduced **dynamic computation graphs**, inspired by the Chainer framework from Japan. Unlike TensorFlow's static graphs, PyTorch builds the computational graph on-the-fly during execution. This was a revolutionary shift that resonated with researchers due to its intuitive *"define-by-run"* approach.
- **Dynamic Graphs and Flexibility:** With dynamic graphs, PyTorch allows you to modify the model's architecture or control flow during runtime. This means model debugging and development can be done using standard Python tools and techniques (for example, inserting print statements to inspect values during training). The flexibility dramatically improves ease of

use for experimentation: researchers can iterate quickly without the compile step required by static graph frameworks. PyTorch's imperative style (execute as you go) made it much easier to learn and use for many, contributing to its popularity in the research community.

- **Community and Ecosystem:** PyTorch rapidly cultivated a vibrant open-source community. A rich ecosystem of libraries, extensions, and pre-trained models grew around PyTorch (such as torchvision for vision, Hugging Face Transformers for NLP, etc.). The strong community support and Facebook's active development have led to rapid improvements and widespread adoption. By 2020, PyTorch became the dominant framework in academic research, often the tool of choice for implementing new papers and models, due to its balance of flexibility and performance.
- **fastai's Contribution:** Built on top of PyTorch, **fastai** was introduced in 2018 by Jeremy Howard and Rachel Thomas with the mission to **"democratize deep learning"**. fastai provides a high-level API that encapsulates best practices and sensible defaults, enabling users (even those with less coding experience) to achieve advanced results with minimal code. It offers layered abstractions – you can use it at a very high level to train models in just a couple of lines, or dive deeper into PyTorch as needed. fastai also emphasizes education and practical guidance (through the popular fast.ai online courses), bringing ethical and accessible AI to a broad audience. In essence, fastai builds on PyTorch's flexibility, adding productivity and ease-of-use features so that more people can experiment with state-of-the-art models easily.

Slide 6: Design Philosophy

- **Static vs. Dynamic Graphs:** One key design consideration for deep learning frameworks is how they handle computational graphs. TensorFlow (especially in its original 1.x form) uses **static graphs**: the entire computation graph is defined and **compiled** upfront, then executed. This yields performance benefits through global graph optimization, but it makes the development process less interactive. PyTorch, on the other hand, uses **dynamic graphs**: the graph is defined on-the-fly **during execution**, which means you can execute Python control flow normally and the graph is created as you run the code. This dynamic approach offers unparalleled flexibility and debuggability (you see results of changes immediately, just like regular Python code). Modern frameworks are not strictly one or the other anymore – for example, TensorFlow 2.x introduced an *Eager Execution* mode (dynamic execution similar to PyTorch by default), and PyTorch introduced *TorchScript* to optionally create static graphs for optimization. This convergence shows that the static-vs-dynamic choice is now a spectrum rather than a binary split, with frameworks providing options for both as needed.
- **High-Level vs. Low-Level Abstractions:** Deep learning frameworks are designed in **layers of abstraction** to cater to different user needs. At the top are **high-level APIs** which prioritize ease of use:
    - **High-Level APIs (e.g., Keras, fastai):** These provide user-friendly interfaces with reusable building blocks and sensible defaults. They allow rapid prototyping and are excellent for teaching and for developers who want results quickly. High-level APIs

abstract away most of the low-level details of model building and training (such as manual gradient calculation or device management).

- **Mid-Level Engines (e.g., Core TensorFlow, PyTorch):** These are the core framework libraries that manage tensors, automatic differentiation, and hardware acceleration. At this mid-level, developers can write custom training loops or layers if needed. The mid-level offers a balance between flexibility and convenience — you still get help from the framework (like not having to write CUDA code for GPU by yourself), but you have the freedom to customize models and training processes more deeply than high-level APIs typically allow.
- **Low-Level Kernels (C++/CUDA implementations):** Under the hood, both TensorFlow and PyTorch rely on highly optimized low-level kernels (often written in C++ and CUDA for NVIDIA GPUs) for tensor operations (like matrix multiplication, convolution, etc.). These kernels are specialized for various hardware and are what actually execute the math operations. While end users rarely interact with these directly, understanding that they exist helps explain why frameworks can achieve high performance — they delegate computations to these optimized routines.

This **layered abstraction stack** means you can choose the level of abstraction appropriate for your task: beginners and those seeking quick results can stay at the high-level, while experts with specialized needs can drop down to mid-level or even write custom low-level code. The design philosophy of frameworks is to provide multiple entry points — you can get started easily, but you're not boxed in if you need more control.

Slide 7: Dummy Example Spec

(Toy Problem Definition for Framework Comparison)

This slide defines a simple linear regression problem that will be used as a common benchmark across all frameworks. The idea is to solve a straightforward task in each framework and compare how each one approaches it. The elements of the experiment are:

- **The Dataset:** We generate 200 data points based on the linear function $y = 3x + 2$ and add some Gaussian noise to the outputs. This synthetic dataset is simple enough for all frameworks to handle, and the true underlying model (slope 3 and intercept 2) is known.
- **The Goal:** Train a model to recover the underlying linear relationship. In other words, we expect the learned parameters (weight and bias) to be close to 3.0 and 2.0, respectively. We will consider the training successful if the learned slope and intercept are within a tolerance (about ±0.05) of the true values. We will use mean squared error (MSE) as the loss function to measure the difference between predictions and true values.
- **The Optimizer:** We will train for a fixed number of epochs (200) using a standard optimization algorithm. In this example, we can use **Adam** (a popular adaptive optimizer) or stochastic gradient descent (SGD) with a learning rate of 0.1. The choice of 200 epochs and the learning

rate is to ensure the model has plenty of opportunity to converge to the correct solution in each framework.

- **The Control:** To make fair comparisons, we use the same random seed and initialization for each framework's experiment. By controlling randomness (in data shuffling, initialization, etc.), we ensure that any differences in outcomes are due to the frameworks and not due to random chance. This way, we can directly compare how each framework performs on the *exact* same problem under similar conditions.

(By setting up this toy problem consistently, we create a level playing field to evaluate and demonstrate the differences in code and workflow between TensorFlow, Keras, PyTorch, and fastai.)

Slide 8: Code Walk-Through

In the next part of the presentation, we walk through a "Hello World" linear regression implementation in each of the four frameworks. Each sub-section of this slide outlines the approach in one framework, highlighting how much code is needed and how the workflow differs. The problem setup is the same for all (the dummy dataset defined earlier). Below are the key steps for each framework:

- **TensorFlow 2: Raw Gradient Tape**
    1. **Import & Define:** Begin by importing TensorFlow and setting up trainable variables for the weight (`w`) and bias (`b`). Also generate the input data (`X`) as a constant tensor and compute the noisy outputs `Y = 3*X + 2 + noise`.
    2. **Build Dataset:** (In this simple example, data is already in memory as tensors; for larger problems one might build a `tf.data.Dataset`). Ensure `X` and `Y` are prepared for training.
    3. **GradientTape Loop:** Use TensorFlow's low-level automatic differentiation API. For each training iteration (we run 200 iterations/steps), open a `tf.GradientTape()` context, compute the predicted `Y_pred = w*X + b`, calculate the MSE loss between `Y_pred` and true `Y`, and then use the tape to compute gradients of the loss with respect to `w` and `b`. Apply an optimizer (Adam with learning rate 0.1) to adjust `w` and `b` using those gradients.
    4. **Result:** After 200 training steps, output the learned parameters. We expect `w ≈ 3.0` and `b ≈ 2.0`. Indeed, TensorFlow's final values for `w` and `b` should be very close to these targets, demonstrating that the model has learned the correct linear relationship. *This low-level exercise shows TensorFlow's core capabilities (like automatic differentiation via GradientTape) and gives maximum control over the training loop.*
- **Keras: The Sequential One-Liner**
    1. **Define Model:** Using Keras (which is integrated in TensorFlow 2.x), define a very simple model. For example, use `keras.Sequential` to create a model with a single dense layer (1 input, 1 output) which will learn the weight and bias.

2. **Compile:** Specify the learning process by compiling the model: choose the optimizer (Adam with lr=0.1) and the loss function (mean squared error). Compilation in Keras configures the model for training in one call.
3. **Fit:** Provide the input and output data to the model's `fit()` method and train for 200 epochs. Keras handles the training loop internally – you don't have to manually compute gradients or update weights.
4. **Result:** After training, use the model to output the learned weight and bias (for a single-layer linear model, these are the kernel and bias of the dense layer). The Keras model should also find values close to 3 and 2. Notably, this approach only took a few lines of code and no explicit loops – it showcases how Keras's high-level abstraction makes the task straightforward, while still achieving the same outcome as the raw TensorFlow approach.

- **PyTorch: The Dynamic Loop**
  1. **Setup Model and Data:** Import PyTorch. Create tensors for `X` (inputs) and `Y` (outputs) using the same data as before. Initialize a simple linear model (`torch.nn.Linear`) with 1 input and 1 output node. This model internally creates a weight and bias, initialized randomly.
  2. **Optimizer and Loss Function:** Choose a loss function (PyTorch's `nn.MSELoss` for mean squared error) and an optimizer (e.g., `torch.optim.SGD` with lr=0.1 or Adam). These will be used to compute gradients and update the model's parameters.
  3. **Training Loop:** For 200 iterations, perform the training manually: zero out previous gradients (`optimizer.zero_grad()`), do a forward pass through the model to get `Y_pred` from `X`, compute the loss (`loss_fn(Y_pred, Y)`), call `loss.backward()` to compute gradients of the loss w.r.t. the model's parameters, and then call `optimizer.step()` to update the parameters. This explicit loop gives fine-grained control and is a typical training pattern in PyTorch when not using higher-level trainers.
  4. **Result:** After the loop, print out the learned weight and bias from the model. We expect them to be ~3.0 (weight) and ~2.0 (bias), and PyTorch should achieve that. The result confirms that the dynamic graph approach successfully learned the same function. *This example highlights PyTorch's imperative style – the training procedure is written as standard Python code – which many find intuitive for debugging and experimenting.*
- **fastai: The High-Level Learner**
  1. **Data Preparation:** Use fastai to simplify data handling. Wrap the training data (`X` and `Y` arrays or tensors) into a `fastai.data.core.DataLoaders` object. fastai's DataLoaders can quickly create batches and manage data transforms, but for this simple case it's essentially just holding our small dataset.

2. **Define Learner:** fastai provides high-level APIs to create models with minimal code. Use `fastai.tabular_learner` (since it's a simple tabular data problem) or a similar function to create a `Learner`. We specify the architecture (a linear model with no hidden layers, effectively) and the optimization parameters. Under the hood, this sets up a PyTorch model and everything needed for training.
3. **Training Policy:** Invoke fastai's training routine with one line. For example, call the `learner.fit_one_cycle(20, lr=0.1)` to train for 20 epochs using the 1-cycle learning rate policy. fastai automates the training loop, applying best practices like learning rate scheduling, etc., unless customized.
4. **Result:** After training, retrieve the learned model parameters. fastai will have found a weight and bias close to 3 and 2, respectively, after just a few lines of code. *This demonstrates fastai's focus on productivity: we achieved the same outcome as before, but with extremely concise code. The fastai library handles the details, allowing practitioners to get results quickly.*

(By stepping through each framework, the audience can see how the coding experience varies – from low-level manual training in TensorFlow and PyTorch to high-level abstraction in Keras and fastai – even though all end up solving the same simple problem.)

Slide 9: Side-by-Side Review

- **Abstraction vs. Lines of Code:** All four frameworks successfully solve the linear regression task with essentially the same result, but the amount of code and level of abstraction varies greatly. For example, implementing the solution took on the order of **8 lines** of code in low-level TensorFlow, about **4 lines** in Keras, roughly **10 lines** in pure PyTorch, and as little as **2 lines** using fastai's high-level API. Fewer lines generally indicate a higher-level, more abstracted approach (which simplifies development), whereas more lines of code indicate a lower-level approach that, while verbose, offers more manual control. This side-by-side comparison highlights the **lines-to-accuracy trade-off** – even though all approaches reach the same accuracy on this simple problem, the effort and complexity required differ. Each framework sits at a different point on the spectrum between **maximum control** and **maximum convenience**.
- **Readability vs. Control:** When choosing a framework, one must balance ease-of-use against flexibility. Frameworks like **fastai** and **Keras** emphasize readability and quick development; they handle many details automatically, which makes the code easier to understand and maintain (especially for beginners or in rapid prototyping scenarios). On the other hand, **TensorFlow (low-level)** and raw **PyTorch** give the developer fine-grained control over every step of the computation and training process, which can be crucial for complex research experiments or performance tuning in production. However, this comes at the cost of writing more code and potentially more complexity. The optimal choice of framework depends on the project's requirements and the team's expertise: if deadline speed and simplicity are priorities, a higher-level approach might be best, but if custom behavior or absolute performance is needed,

a lower-level approach could be justified. In practice, many teams even mix these tools – for instance, starting a project with Keras or fastai for quick prototyping, and then later moving to TensorFlow or PyTorch for fine-tuning and deployment once the concept is proven.

Slide 10: Ecosystem & Future

- **TensorFlow's Production Dominance:** TensorFlow has established itself as a go-to framework in industry and production environments. Its ecosystem includes **TensorFlow Serving** for deploying models at scale, **TensorFlow Lite** for mobile and IoT deployment, and tight integration with **Google Cloud AI** services. This makes TensorFlow particularly strong when you need to take a model from research to a reliable, optimized production service. Over the years, Google's backing has ensured TensorFlow is well-supported for long-term deployment, and many enterprise tools are built around it.
- **PyTorch's Research Leadership:** PyTorch has become the preferred framework for **academic research and cutting-edge development**. Its intuitive Pythonic interface and dynamic graph nature mean researchers can write experiments in a very flexible way, which has fueled innovation. PyTorch also powers popular open-source hubs – for example, many models in the **Hugging Face** NLP repository and other community libraries are built on PyTorch. The framework's growth is heavily community-driven, and it's increasingly seeing adoption in industry as well (especially in AI startups and labs) as its capabilities expand.
- **Keras's Educational Impact: Keras** is known for its motto "an API designed for human beings" and indeed has brought countless newcomers into deep learning. Through platforms like Coursera and university courses, Keras became a standard teaching tool for introducing neural networks. It provides a clean, simple way to build models which lowers the barrier to entry. Even though Keras is now integrated with TensorFlow, it remains a distinct high-level interface. Its influence is seen in how newer frameworks emphasize user experience. Keras's legacy is that it made powerful concepts accessible to non-experts and emphasized the importance of developer productivity and simplicity in AI frameworks.
- **fastai's Democratization of AI:** The **fastai** library and the broader fast.ai movement (including its free courses) focus on making state-of-the-art deep learning **accessible to everyone**. fastai's ecosystem provides prepackaged solutions for common tasks and promotes best practices (like the "learning rate finder" or the one-cycle training policy) by default. It also has a strong ethos around ethical AI and community (for instance, encouraging participants to share their work and focus on using AI for good). fastai has carved a niche among practitioners who want results quickly without needing to reinvent the wheel; it builds on PyTorch's foundation but adds a layer that often encapsulates months or years of expert knowledge in its API design.
- **Convergence and Future Trends:** We are seeing frameworks influence each other and the lines between them blurring. TensorFlow's adoption of eager execution in version 2 was a response to the popularity of PyTorch's dynamic approach, while PyTorch's development of TorchScript (and more recently TorchCompile) acknowledges the benefits of graph optimization for deployment. **Interoperability** is improving too – projects like **ONNX (Open Neural**

Network Exchange) allow models to be converted between frameworks. In addition, all frameworks are expanding to cover both research and production needs (for example, JAX from Google is an emerging system focusing on composable function transformations which influence future framework design). We also see specialization for different use cases: frameworks might offer multiple front-ends (a high-level API vs. a low-level one) or specialized variants for mobile, browser (TensorFlow.js), etc. Another important trend is the rise of **compiler optimizations and hardware-specific accelerators**. Tools like **XLA (Accelerated Linear Algebra compiler)** in TensorFlow, **TVM**, and PyTorch's **TorchDynamo** aim to optimize computations by compiling parts of the model for specific hardware (GPUs, TPUs, specialized AI chips). This means that the future will likely consist of a **multi-framework, multi-backend ecosystem** – developers will have the flexibility to use high-level interfaces, while under the hood these systems automatically choose the best low-level execution path. Ultimately, deep learning frameworks are converging in capabilities and will continue to evolve, but this diversity and competition have been healthy for the field, driving rapid improvements and new ideas.

Slide 11: Key Takeaways

In conclusion, here are the key practical takeaways and tips for practitioners when navigating AI frameworks:

1. **Start High:** Whenever you tackle a new project or idea, begin with high-level APIs (such as Keras or fastai) to prototype quickly. These frameworks let you obtain results with minimal effort, which is great for proof-of-concept experiments. Early on, your priority should be testing the viability of an idea rather than optimizing performance or writing low-level code. High-level tools are the fastest path to a working model.
2. **Descend Gradually:** Only drop down to lower-level frameworks like core TensorFlow or raw PyTorch when necessary. If you find that the high-level abstraction is limiting (for example, you need a custom training loop or a novel layer that isn't readily available), then it's time to "descend" to a lower level. Avoid premature optimization – do not write extensive low-level code before it's clear that you need that extra control. Use the simplest tool that gets the job done, and only increase complexity when the situation demands it.
3. **Master One, Know the Others:** It's beneficial to become deeply proficient in one framework (whichever aligns best with your work or preferences), but also stay curious about the others. Each framework has unique strengths, and ideas often cross-pollinate between communities. By having a working knowledge of multiple frameworks, you can more easily pick the right tool for each project and you'll be able to learn new libraries faster. The deep learning landscape changes quickly, so being adaptable is important.
4. **Focus on Fundamentals:** Remember that frameworks are just tools to implement the underlying mathematics and concepts of machine learning. A strong grasp of the fundamentals – linear algebra, calculus, neural network architecture, and optimization concepts – is essential and will serve you regardless of which framework you use. If you understand what the model is

doing conceptually, you can translate that to any framework's syntax. Don't become too attached to one library's way of doing things; instead, invest in understanding the principles that underlie them all.

Further Learning Resources: To continue improving your skills and stay up-to-date with the fast-moving world of AI frameworks, consider these resources and activities:

- **Official Documentation & Tutorials:** Start with the official docs and beginner tutorials for each framework. Reimplement the basic examples ("Hello World" models like a simple MNIST classifier or the linear regression shown here) in TensorFlow, PyTorch, Keras, and fastai. The official guides often include best practices and will familiarize you with each library's patterns.
- **Hands-on Projects to Compare Frameworks:** Try building the same small project (e.g., a simple image classifier or text classifier) in multiple frameworks. Measure and compare things like training speed, memory usage, and ease of debugging. This will give you a concrete sense of the trade-offs. There are also open-source benchmarks and community projects that compare frameworks on various tasks – studying those can be enlightening.
- **Community Engagement:** Join the communities and courses that surround these frameworks. For TensorFlow, Google's Coursera specialization or the TensorFlow Developer Certificate materials are great for structured learning. For PyTorch, check out the official PyTorch tutorials and perhaps the fast.ai course (which uses PyTorch under the hood). The fast.ai community is very welcoming and has forums where you can ask questions. Follow repositories like **Papers With Code** to see the latest research and which frameworks are being used. Engaging on forums (Stack Overflow, Reddit's r/MachineLearning, specialized Slack/Discord groups) will keep you current on best practices and new releases.

By understanding the strengths of each framework and continuously learning, you'll be well-equipped to choose the right tool for each machine learning project and adapt as the technology evolves. The key is to remain curious and flexible – the landscape of AI frameworks is dynamic, and staying informed will help you make the best decisions in your work.