Dynamic Models in PyTorch

Introduction

Dynamic models in PyTorch represent a fundamental shift in macine leanring models are conceptualized and designed, particularly in comparison to traditional, static approaches. These models derive their name from their ability to define and modify their computational graph dynamically—during runtime—unlike static models that rely on a fixed computation graph defined before execution. This approach is particularly powerful in research, prototyping, and applications involving non-uniform input structures or dynamic control flows, as it provides the flexibility and expressiveness required to implement complex behavior directly in the forward pass.

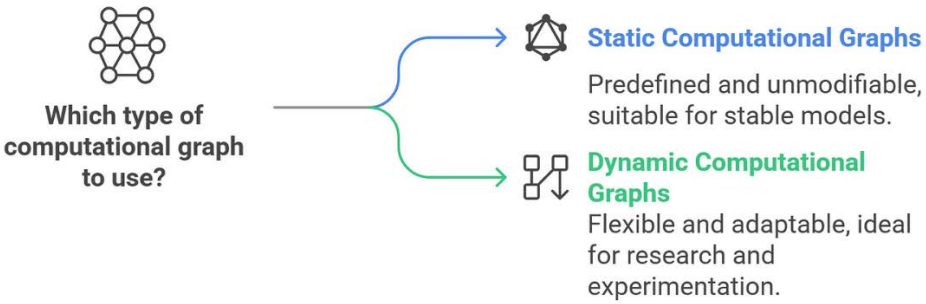
At the core of PyTorch's dynamic modeling capabilities lies its define-by-run approach. In contrast to the define-and-run style, where the entire computation graph is first declared and then executed, define-by-run means the graph is built on-the-fly as operations are executed. Each forward pass constructs a new graph, reflecting exactly the control flow taken and the operations applied. This means that every model forward call can be different, adapting to the specific structure or content of the input.

This flexibility is highly advantageous when dealing with variable-length data, recursive structures, conditional logic, or any scenario where the flow of computation cannot be predetermined. For example, models may need to perform loops whose number of iterations depends on the input data, or may apply different operations depending on certain runtime conditions. In a dynamic model, such logic is implemented using native Python control flow constructs, such as `if` statements, `for` and `while` loops, which seamlessly integrate into the model's execution. This makes dynamic models particularly intuitive to write, debug, and understand, as the modeling code follows familiar imperative programming patterns.

Another key aspect of dynamic models is that they facilitate a more transparent and interactive development cycle. Because the computation graph exists only during the forward execution, it is directly observable and modifiable during runtime. This is highly-useful for debugging: one can insert print statements, use debuggers, or explore intermediate values interactively without requiring specialized tools or abstractions. The model behaves like standard Python code, and any runtime errors or inconsistencies can be caught and understood in the context of normal program execution.

Despite their runtime flexibility, dynamic models in PyTorch fully support automatic differentiation. As the computation graph is built dynamically, PyTorch records all operations

involving tensors with `requires_grad=True` in a structure called the dynamic computation graph or autograd graph. This graph tracks the sequence of operations and their gradients, allowing PyTorch to compute derivatives during the backward pass. Once the backward pass is complete, the graph is discarded, freeing memory and preparing for the construction of a new graph in the next iteration.



Dynamic Model Execution

An important implication of dynamic modeling is the reduced separation between the model definition and its execution semantics. In many static frameworks, defining the model involves using a domain-specific configuration or a graph-building API that abstracts the actual control flow. This often leads to a disconnect between how a model is described and how it behaves. In contrast, PyTorch's dynamic models are written as standard Python classes and functions. The same code that defines the model also dictates its behavior during execution, leading to a more coherent and unified development experience.

This coherence also makes dynamic models more accessible for experimentation. Changes to the model architecture or behavior can be made with immediate effect, without the need for recompiling graphs or reconfiguring execution plans. However, the benefits of dynamism come with trade-offs. One concern is that the dynamic nature of model construction can introduce variability in execution time and memory usage across iterations, depending on the input data and control flow. This can complicate optimization and deployment, particularly in environments where predictability and efficiency are important. For example, exporting models for inference in production, where execution must be consistent and fast, may require transforming the dynamic model into a more static representation. This transformation process imposes restrictions on the kinds of Python features that can be used and may necessitate refactoring parts of the model to conform to export-friendly constructs.

Another challenge involves performance optimization. While dynamic graphs are highly flexible, they may not be as easily optimized by compilers and hardware accelerators as static graphs. Static models allow for extensive graph-level optimizations, such as operation fusion, memory preallocation, and execution planning, which are more difficult to apply when the structure of the graph is not known in advance. PyTorch continues to evolve in this direction with intermediate representations and ahead-of-time compilation strategies, but dynamic models generally prioritize flexibility over raw performance.

Despite these considerations, dynamic models represent a modern approach to deep learning model development, since they emphasze runtime flexibility, native control flow, and seamless integration with the Python programming language. This makes them an adequate choice for tasks requiring adaptive computation or experimental modeling.

Curriculum Learning

Curriculum learning is a training strategy in machine learning that draws inspiration from the way humans and animals naturally learn: starting with simpler concepts and progressively handling more complex ones. The core idea behind curriculum learning is to organize the training data or tasks in a meaningful order that gradually increases in difficulty, allowing models to learn more effectively and often leading to faster convergence, improved generalization, and greater robustness.

In a traditional machine learning setup, all training examples are typically presented to the model in a random order, regardless of their complexity or difficulty. While this random exposure may work in many cases, it can also lead to inefficient learning. The model may struggle to learn complex examples without first mastering simpler, underlying patterns. Curriculum learning addresses this by deliberately structuring the learning process: initially exposing the model to easy examples, and then incrementally introducing more difficult ones as the model's understanding improves.

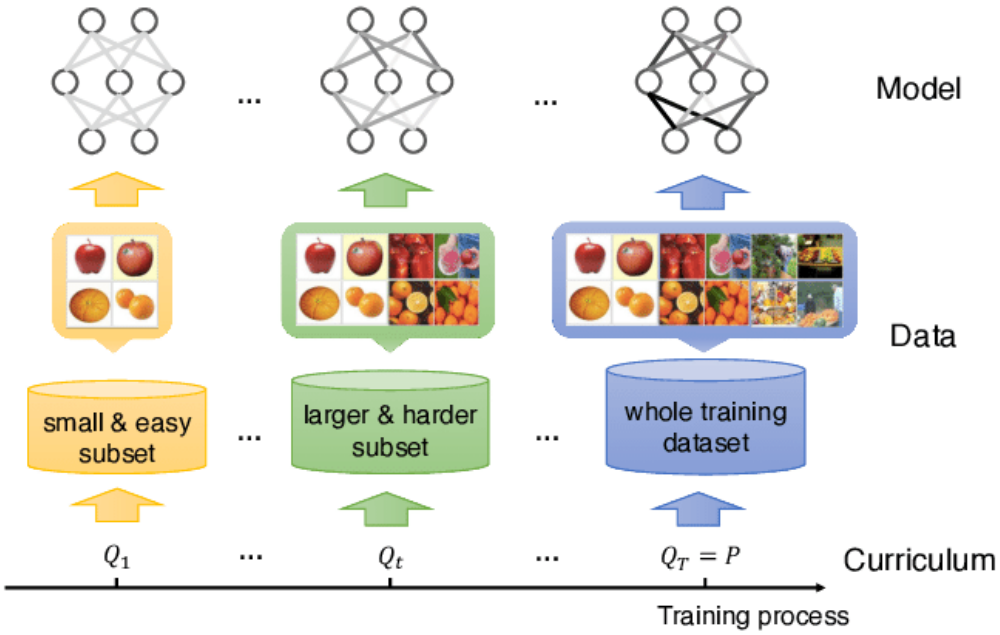
This approach has several theoretical motivations. From an optimization perspective, easier examples often provide stronger and clearer gradients, which can guide the model towards favorable regions of the parameter space early in training. Once the model has reached a region where it has learned simple structures well, it can better handle the noise, variability, and subtle patterns present in harder examples. Curriculum learning can thus be seen as a way to smooth the optimization landscape, making it easier for the model to find good solutions. Moreover, curriculum learning can serve as a form of regularization. By guiding the learning

process along a trajectory from simple to complex, it can prevent the model from overfitting early to noisy or anomalous hard examples. This gradual exposure mirrors processes in cognitive science, where humans develop skills sequentially, mastering basic elements before integrating them into complex reasoning and problem-solving.

In practice, implementing curriculum learning involves two main components:

- defining "difficulty"
- scheduling the progression.

Difficulty can be determined in many ways: manually based on domain knowledge, automatically through heuristics (for example, sentence length in language tasks, distance to decision boundary in classification), or adaptively based on the model's current performance. Scheduling strategies dictate when and how to introduce harder examples. Simple approaches may use a pre-set schedule based on training epochs, while more sophisticated methods might adapt the curriculum dynamically, responding to the model's learning progress.



Curriculum Learning with Dynamic Models

The relationship between curriculum learning and dynamic models is particularly important because dynamic models offer the flexibility needed to fully exploit curriculum strategies. Since dynamic models define their computation graph at runtime, they can naturally adapt to different levels of input complexity without being constrained by a fixed architecture or computation path.

In a dynamic model setting, the curriculum does not have to be just about presenting simpler inputs first; it can also involve modifying the model's behavior/structure during training. For instance, a dynamic model might:

- Change its architecture based on input complexity (e.g., using deeper processing pipelines for harder examples).
- Dynamically adjust how much computation is applied to each input (e.g., by skipping layers, branching conditionally, or varying recursion depth).
- Incorporate adaptive mechanisms, where the model decides how to process an input based on its estimated difficulty at runtime.

This close interaction is possible because dynamic models inherently support varying computational flows. As the curriculum progresses and examples become harder, the model can change how it processes them: more layers, different pathways, increased attention to specific featuresm without needing to redefine the entire network structure ahead of time.

Additionally, dynamic models can evaluate the difficulty of inputs on-the-fly. They can include auxiliary modules that estimate how confident the model is about a prediction or how complex an input appears to be, and then adjust their behavior accordingly. This capability enables *self-paced learning*, a form of curriculum learning where the model itself decides the pace at which it moves from easier to harder examples. Curriculum learning can be naturally applied by ordering inputs not just by difficulty but by structural complexity, and dynamic models can fully accommodate these varying structures without redesign.In research and real-world applications, combining curriculum learning with dynamic modeling often leads to more resilient models that generalize better and are less unstable when faced with new, complex inputs.

Tasks

For completing the tasks, the code samples that are provided with this documentation can be used as starting points.

Implement a dynamic PyTorch model that uses a curriculum-learning-style approach for training. Details are as follows:

- The dynamic neural network starts out very simple (i.e. a single convolutional layer and a single fully-connected layer) and increases in complexity during training. For example, after certain predefined epochs, other layers may be added (additional

convolutional, pooling, dropout and/or fully-connected layers). The neural network is a dynamic model in the sense that its architecture and computational graph change dynamically as training occurs.

- During training, in earlier stages we use easier instances of the training data set. As training progresses and the neural network increases in complexity, mode difficult instances are introduced to the model.
- We aim to design a dynamic model and compare it to a static version (one that remains unchanged during training)
    - Try different versions of the dynamic model, with different architecture variations during training.
    - Evaluate the model on the easy, medium and hard instances, as well as on the entire data set.
    - Compare the dynamic model(s) with the static one (in terms of accuracy)
- Note your findings – the primary aim is to come up with a dynamic model that trains faster and performs better that the equivalent static one.