Outline

1. Computational graphs & deep learning frameworks
   • Deep neural networks as computational graphs
   • Dynamic vs. static computational graphs

2. QuantLab & quantlib
   • The deep learning development stack
   • QNNs: a HW/SW co-design problem

3. Graph editing
   • Tree traversal and leaf replacement
   • Graph morphisms and algebraic graph rewriting
QuantLab Virtual Workshop

Part 1: computational graphs & deep learning frameworks
Graph terminology - basics

• Let $V \neq \emptyset$ be a set of nodes
Graph terminology - basics

- Let $V \neq \emptyset$ be a set of nodes
- **Graph**
  - $G = (V, E \subseteq V \times V)$
  - Elements $e \in E$ are called **arcs**
Graph terminology - basics

• Let $V \neq \emptyset$ be a set of nodes

• Graph
  • $G = (V, E \subseteq V \times V)$
  • Elements $e \in E$ are called arcs

• Undirected graph
  • $(u, v) \in E \Rightarrow (v, u) \in E$
  • Elements $e \in E$ are called edges
Graph terminology - basics

- Let $V \neq \emptyset$ be a set of nodes
- Graph
  - $G = (V, E \subseteq V \times V)$
  - Elements $e \in E$ are called arcs
- Undirected graph
  - $(u, v) \in E \Rightarrow (v, u) \in E$
  - Elements $e \in E$ are called edges
- Bipartite graph
  - $V = V_A \cup V_B \mid V_A, V_B \neq \emptyset, V_A \cap V_B = \emptyset$
  - $E \subseteq ((V_A \times V_B) \cup (V_B \times V_A))$
Supervised learning: the problem

• The task is approximating an (unknown) function

\[ f^* : X \to Y \]

• How can we assess the quality of an approximation \( f \approx f^* \)?

  • Loss function:
    \[ \ell : Y \times Y \to \mathbb{R}^+_0 \]

  • Loss functional:
    \[ \mathcal{L}(f) := \int_{X \times Y} \ell(f(x), y) \, d\mu(x, y) \]
Supervised learning: the solution

• Machine learning system
  • **Hypothesis space**
    • $f : \Theta \times X \to Y$ (i.e., a collection $\{f_\theta : X \to Y \mid \theta \in \Theta\}$)
    • Rewrite $L(f) = L(\theta) = \int_{X \times Y} \ell(f(\theta, x), y) d\mu(x, y)$
  • **Data set**
    • $\mathcal{D} : X \times Y \to \mathbb{N}_0 \mid 0 < \sum_{(x,y) \in X \times Y} \mathcal{D}(x, y) = N < +\infty$
    • Approximate $\mu \approx \frac{1}{N} \sum_{(x,y) \in X \times Y} \mathcal{D}(x, y) \delta_{(x,y)}$
  • **Learning algorithm**
    • If $L$ and $f$ are differentiable, it can be gradient-based:
      $$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L(\theta_t) = \theta_t - \eta \left( \frac{1}{N} \sum_{(x,y) \in X \times Y} \mathcal{D}(x, y) \nabla_{\theta} \ell(f(\theta_t, x), y) \right)$$
Computational graphs

• Directed, bipartite graphs

• $V = V_M \cup V_K$
  • *Memory nodes* $v \in V_M$ represent operands
  • *Kernel nodes* $v \in V_K$ represent operations

• $E \subseteq ((V_M \times V_K) \cup (V_K \times V_M))$
  • Arcs $e \in E \cap (V_M \times V_K)$ represent *read/load* dependencies
  • Arcs $e \in E \cap (V_K \times V_M)$ represent *write/store* dependencies

• Each operand is the result of at most one operation:
  • $\forall v \in V_M, (u_1, v), (u_2, v) \in E \Rightarrow u_1 = u_2$
DNNs as computational graphs: an example

\[ w_2 \varsigma(b_1 + w_1 x_0) \]
DNNs as computational graphs: an example

\[ w_1 x_0 \]
DNNs as computational graphs: an example

\[ b_1 + w_1 x_0 \]
DNNs as computational graphs: an example

\[ \varsigma(b_1 + w_1 x_0) \]
DNNs as computational graphs: an example

\[ w_2 \varsigma(b_1 + w_1 x_0) \]
Three ways of performing differentiation

- **Symbolic differentiation**
  - Based on the rules of differential calculus
  - Given a function $\mathcal{L}(\theta, z)$, *pre-compute* $\nabla_\theta \mathcal{L}|_{\theta, z}$ as a function of $\theta$ and $z$.
  - Cons:
    - Computing the differential **automatically** might be **impossible for complex functions**
    - Computing the differential **by hand** can be **time-consuming and is error-prone**
Three ways of performing differentiation

• Symbolic differentiation
  • Based on the rules of differential calculus
  • Given a function $\mathcal{L}(\theta, z)$, pre-compute $\nabla_\theta \mathcal{L}|_{\theta, z}$ as a function of $\theta$ and $z$.
  • Cons:
    • Computing the differential automatically might be impossible for complex functions
    • Computing the differential by hand can be time-consuming and is error-prone

• Numerical differentiation
  • Based on the definition of derivative
  • Given a function $\mathcal{L}(\theta, z)$, computing $\mathcal{L}(\theta + h, z) - \mathcal{L}(\theta, z)$ requires two evaluations.
  • Cons:
    • Computers have no notion of limit operation: numerical derivatives are usually approximations computed using small values for $\| h \|
    • Approximation errors are more likely when $\theta$ is multi-dimensional
Three ways of performing differentiation

• **Symbolic differentiation**
  • Based on the rules of differential calculus
  • Given a function $\mathcal{L}(\theta, z)$, **pre-compute** $\nabla_\theta \mathcal{L}|_{\theta, z}$ as a function of $\theta$ and $z$.
  • Cons:
    • Computing the differential **automatically** might be **impossible for complex functions**
    • Computing the differential **by hand** can be time-consuming and is error-prone

• **Numerical differentiation**
  • Based on the definition of derivative
  • Given a function $\mathcal{L}(\theta, z)$, computing $\mathcal{L}(\theta + h, z) - \mathcal{L}(\theta, z)$ requires two evaluations.
  • Cons:
    • Computers have no notion of limit operation: numerical derivatives are usually approximations computed using small values for $\| h \|$
    • Approximation errors are more likely when $\theta$ is multi-dimensional

• **Automatic differentiation**
Automatic differentiation

• Based on the chain rule
• Each operation computes the gradients with respect to its inputs
• Two modes
  • **Direct-mode**
    • Can be computed in parallel to the forward pass
    • Almost always requires recomputing tensor contractions
  • **Reverse-mode** (aka back-propagation)
    • Must wait the completion of the forward pass before beginning the gradient computation
    • Computes each product in the chain rule just once
Differentiable computational graphs

• Each operation \( v \in V_K \) is differentiable with respect to its operands \( u \in V_M \mid (u, v) \in E \)

• **Forward pass** (aka *inference pass*)

• **Backward pass**
  • This is gradient computation
  • Do not confuse it with gradient descent!

\[
\nabla_w \ell \quad \nabla_b \ell \quad \nabla_s \ell
\]
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- Static computational graph
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- Dynamic computational graph
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• **Static computational graph**
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• **Dynamic computational graph**
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time Depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation \((define-and-run)\)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow \((define-by-run)\)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (define-and-run)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (define-by-run)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

- **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

- **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• Static computational graph
  • Graph is fully defined before executing any operation (*define-and-run*)
  • Pro: the graph’s structure is clear and easy to manipulate
  • Cons: slower development cycle

• Dynamic computational graph
  • Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  • Pro: faster development cycle
  • Cons: the graph’s structure might be obscure and cumbersome to manipulate
Static vs. dynamic computational graphs

• **Static computational graph**
  - Graph is fully defined before executing any operation (*define-and-run*)
  - Pro: the graph’s structure is clear and easy to manipulate
  - Cons: slower development cycle

• **Dynamic computational graph**
  - Graph is defined at run-time depending on the script’s control flow (*define-by-run*)
  - Pro: faster development cycle
  - Cons: the graph’s structure might be obscure and cumbersome to manipulate
A thousand flavours of computational graphs

ONNX: the “assembly” of computational graphs
A thousand flavours of computational graphs

TensorFlow (v1.0 – might have changed)

• Operation “super-nodes” contain:
  • Memory nodes
    • Constants
    • Parameters
    • Hyper-parameters
    • Output features
  • Kernel nodes

• Edges can be associated to the output memory nodes contained in each “super-node”
  • “Nodes represent operations, edges represent data flowing between operations”
A thousand flavours of computational graphs

TensorFlow (v1.0 – might have changed)

• Operation “super-nodes” contain:
  • Memory nodes
    • Constants
    • Parameters
    • Hyper-parameters
    • Output features
  • Kernel nodes

• Edges can be associated to the output memory
  nodes contained in each “super-node”
  • “Nodes represent operations, edges represent data
    flowing between operations”
A thousand flavours of computational graphs

PyTorch (v1.9)
• Operation “super-nodes” contain:
  • Memory nodes
    • Constants
    • Parameters
    • Hyper-parameters
  • Kernel nodes; remember: they are instantiated only at runtime!
    • Defined **explicitly** in the constructor (**__init__**) method
• Edges can be associated to the memory nodes representing features
  • Remember: they are instantiated only at runtime!
  • Defined **implicitly** in the **forward** (**__call__**) method
A thousand flavours of computational graphs

PyTorch (v1.9)

• Operation “super-nodes” contain:
  • Memory nodes
    • Constants
    • Parameters
    • Hyper-parameters
  • Kernel nodes; remember: they are instantiated only at runtime!
  • Defined explicitly in the constructor (__init__) method

• Edges can be associated to the memory nodes representing features
  • Remember: they are instantiated only at runtime!
  • Defined implicitly in the forward (__call__) method
A thousand flavours of computational graphs

PyTorch (v1.9)

• Operation “super-nodes” contain:
  • Memory nodes
    • Constants
    • Parameters
    • Hyper-parameters
  • Kernel nodes; remember: they are instantiated only at runtime!
  • Defined explicitly in the constructor (__init__) method

• Edges can be associated to the memory nodes representing features
  • Remember: they are instantiated only at runtime!
  • Defined implicitly in the forward (__call__) method
A thousand flavours of computational graphs

NASBench201 data set

- Neural architecture search (NAS) is a deep-learning-specific variant of model selection

- NASBench201
  - Inputs: genotypes, i.e., structured description of network topologies
  - Outputs: accuracies

- Genotypes are described in terms of cells
  - Nodes represent feature arrays
  - Edges represent operations and their parameters
A thousand flavours of computational graphs

NASBench201 data set

• *Neural architecture search* (NAS) is a deep-learning-specific variant of *model selection*

• NASBench201
  • Inputs: *genotypes*, i.e., structured description of network topologies
  • Outputs: accuracies

• Genotypes are described in terms of cells
  • Nodes represent feature arrays
  • Edges represent operations and their parameters
QuantLab Virtual Workshop

Part 2: QuantLab & quantlib
The deep learning development stack

**Platform-agnostic**
- **Data analysis**: how can we model the data problem?
- **DNN design**: which network topology can work best?
- **Training**: backpropagation + SGD

**Platform-specific**
- **Graph optimisation**: ONNX graph (e.g., tiling, “node fusion”)
- **Code generation**: from ONNX graph to C/C++ code
- **Compilation**: from C/C++ code to machine code
QuantLab: structure overview
QuantLab: the systems package
The **systems** package

```
<table>
<thead>
<tr>
<th>systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>CIFAR10</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>vgg.py</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>config.json</td>
</tr>
</tbody>
</table>
```
The `systems` package
The **systems** package: problem sub-package

```bash
$ bash configure/problem.sh CIFAR10
```
The **systems** package: adding problems
The **systems** package: topology sub-package

```
$ bash configure/problem.sh CIFAR10 VGG
```
The **systems** package: adding topologies
QuantLab: the **manager** package
The manager package

- **platform**: management of HW/OS aspects (e.g., GPU aspects, distributed processing)
- **flows**: the services that can be accessed from the façade
- **logbook**: the abstraction that mediates the interactions between the QuantLab flows and the disk
- **assistants**: the abstractions that assemble the components of the deep learning systems inside QuantLab flows
- **meter**: the abstractions to track statistics on parameters and features of the deep neural network being trained or tested
QuantLab flows
QuantLab *flows*: configuring an experiment

```
$ python main.py –problem=CIFAR10 –topology=VGG configure
```
QuantLab flows: configuring an experiment

$ python main.py -problem=CIFAR10 -topology=VGG configure
QuantLab *flows*: configuring an experiment

```bash
$ python main.py --problem=CIFAR10 --topology=VGG configure
```
QuantLab flows: configuring an experiment

$ python main.py –problem=CIFAR10 –topology=VGG configure

Logbook
QuantLab flows: configuring an experiment

$ python main.py --problem=CIFAR10 --topology=VGG configure
QuantLab *flows*: configuring an experiment

$ python main.py –problem=CIFAR10 –topology=VGG configure
QuantLab *flows*: configuring an experiment

$ python main.py –problem=CIFAR10 –topology=VGG configure
QuantLab *flows*: configuring an experiment

```bash
$ python main.py --problem=CIFAR10 --topology=VGG configure
```
QuantLab *flows*: configuring an experiment

```
$ python main.py --problem=CIFAR10 --topology=VGG configure
```
QuantLab flows: training a DNN

$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
QuantLab *flows*: training a DNN

$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
QuantLab *flows*: training a DNN

$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
QuantLab flows: training a DNN

$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
QuantLab *flows*: training a DNN

```
$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
```
QuantLab *flows*: training a DNN

```
$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
```

![Diagram showing the QuantLab flows structure]
QuantLab *flows*: training a DNN

```
$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
```
QuantLab *flows*: training a DNN

$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
QuantLab *flows*: training a DNN

$\text{python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0}$
QuantLab flows: training a DNN
QuantLab *flows*: training a DNN

```bash
$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
```
QuantLab *flows*: training a DNN

```
$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
```
QuantLab *flows*: training a DNN

```
$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
```

![Diagram](image)
QuantLab \textit{flows}: training a DNN

```bash
$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0
```
QuantLab flows: training a DNN

$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
QuantLab flows: training a DNN

Python command: $ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0
QuantLab *flows*: training a DNN

$ python main.py --problem=CIFAR10 --topology=VGG train --exp_id=0

![Diagram of QuantLab flows: training a DNN](image)
QuantLab \textit{flows}: training a DNN

\$ python main.py –problem=CIFAR10 –topology=VGG train –exp_id=0 \$
QuantLab: usage overview

• Create a problem sub-package (remember to prepare the data!)
• Create a topology sub-package
• Write the working files:
  • Data pre-processing and loading
  • Network definition
  • Output post-processing
• Write the configuration file that describes how to instantiate the system
• Run the configure flow
• Run the training flow

ITERATE UNTIL YOU ARE SATISFIED!
QNNs: a HW/SW co-design problem

Platform-agnostic
- **Data analysis**: how can we model the data problem?
- **DNN design**: which network topology can work best?
- **Training**: backpropagation + SGD

Platform-aware
- **float2fake** conversion
- **Post-training quantization** algorithm (w/o fine-tuning)
- **fake2true** conversion

Platform-specific
- **Graph optimisation**: ONNX graph (e.g., tiling, “node fusion”)
- **Code generation**: from ONNX graph to C/C++ code
- **Compilation**: from C/C++ code to machine code
QNNs: a HW/SW co-design problem

<table>
<thead>
<tr>
<th>Data analysis</th>
<th>DNN design</th>
<th>Training (FP)</th>
<th>float2fake</th>
<th>Post-training quantization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training (FP)</td>
<td>float2fake</td>
<td>Post-training quantization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNN design</td>
<td>Training (FP)</td>
<td>Graph optimisation</td>
<td>Code generation</td>
<td>Compilation</td>
</tr>
<tr>
<td>DNN design</td>
<td>Training (FP)</td>
<td>Graph optimisation</td>
<td>Code generation</td>
<td>Compilation</td>
</tr>
</tbody>
</table>

Platform-agnostic

- **Data analysis**: how can we model the data problem?
- **DNN design**: which network topology can work best?
- **Training**: backpropagation + SGD

Platform-aware

- **float2fake** conversion
- **Post-training quantization** algorithm (w/ fine-tuning)
- **fake2true** conversion

Platform-specific

- **Graph optimisation**: ONNX graph (e.g., tiling, “node fusion”)
- **Code generation**: from ONNX graph to C/C++ code
- **Compilation**: from C/C++ code to machine code
QNNs: a HW/SW co-design problem

**Platform-agnostic**
- **Data analysis**: how can we model the data problem?
- **DNN design**: which network topology can work best?
- **Training**: backpropagation + SGD

**Platform-aware**
- **float2fake** conversion
- **Quantization-aware training** algorithm
- **fake2true** conversion

**Platform-specific**
- **Graph optimisation**: ONNX graph (e.g., tiling, “node fusion”)
- **Code generation**: from ONNX graph to C/C++ code
- **Compilation**: from C/C++ code to machine code
QNNs: a HW/SW co-design problem

Platform-agnostic
- Data analysis: how can we model the data problem?
- DNN design: which network topology can work best?
- Training: backpropagation + SGD

Platform-aware
- float2fake conversion
- Quantization-aware training algorithm
- fake2true conversion

Platform-specific
- Graph optimisation: ONNX graph (e.g., tiling, “node fusion”)
- Code generation: from ONNX graph to C/C++ code
- Compilation: from C/C++ code to machine code

TODAY WE WILL NOT DEAL WITH THESE STEPS
QNNs: a HW/SW co-design problem

Platform-agnostic
- **Data analysis**: how can we model the data problem?
- **DNN design**: which network topology can work best?
- **Training**: backpropagation + SGD

Platform-aware
- **float2fake** conversion
- **Quantization-aware training** algorithm
- **fake2true** conversion

Platform-specific
- **Graph optimisation**: ONNX graph (e.g., tiling, “node fusion”)
- **Code generation**: from ONNX graph to C/C++ code
- **Compilation**: from C/C++ code to machine code

TODAY WE WILL FOCUS ON THESE STEPS
QuantLab: the `quantlib` package
QuantLab: the quantlib package
The *quantlib* package: overview
The **quantlib** package: overview

- Data analysis
- DNN design
- **float2fake**
- Quantization-aware training
- **fake2true**
- Graph optimisation
- Code generation
- Compilation
The `quantlib` package: overview

- Data analysis
- DNN design
- Quantization-aware training
- Graph optimisation
- Code generation
- Compilation
- `float2fake`
- `fake2true`
The **quantlib** package: overview

- **Data analysis**
- **DNN design**
- **float2fake**
  - Quantization-aware training
- **fake2true**
- **Graph optimisation**
- **Code generation**
- **Compilation**
The quantlib package: overview
The **quantlib** package: overview

TODAY’S EXERCISES WILL FOCUS ON THESE TOOLS
The **quantlib** package: overview
Extending topology sub-packages
Extending topology sub-packages

- systems
  - utils
  - CIFAR10
  - ILSVRC12
  - ...
  - data
  - VGG
  - ...
  - logs
  - preprocess
  - vgg.py
  - postprocess
  - quantize
  - config. json
QuantLab: usage overview

• Create a problem sub-package (remember to prepare the data!)
• Create a topology sub-package
• Write the working files:
  • Data pre-processing and loading
  • Network definition
  • Output post-processing

• Write the configuration file that describes how to instantiate the system
• Run the configure flow
• Run the training flow

ITERATE UNTIL YOU ARE SATISFIED!
QuantLab: usage overview

• Create a problem sub-package (remember to prepare the data!)
• Create a topology sub-package
• Write the working files:
  • Data pre-processing and loading
  • Network definition
  • Output post-processing
  • Quantization recipes and network controllers creators (quantize namespace)
• Write the configuration file that describes how to instantiate the system
• Run the configure flow
• Run the training flow
• Perform fake2true conversion
• Generate code for your platform (warning: this is has not been automated yet!)

ITERATE UNTIL YOU ARE SATISFIED!
QuantLab: present and future
QuantLab: present and future

Existing features:

• Configuration-based training flows
• Multi-GPU and multi-process support
• Integration with TensorBoard
• \textit{float2fake} conversion
• Quantization-aware training algorithms (STE, INQ, RPR, ANA, PACT, SAWB)
• \textit{fake2true} conversion
QuantLab: present and future

Existing features:
- Configuration-based training flows
- Multi-GPU and multi-process support
- Integration with TensorBoard
- `float2fake` conversion
- Quantization-aware training algorithms (STE, INQ, RPR, ANA, PACT, SAWB)
- `fake2true` conversion

Planned features:
- Data and network initialisation seeding
- PyTorch code generation for true-quantized networks
- Post-training quantization
- More quantization-aware training algorithms
- Mixed-precision support
QuantLab Virtual Workshop

Part 3: graph editing
Graph editing in **quantlib**

By *graph editing* we refer to a collection of techniques to modify graphs:

- Tree traversal and leaf replacement
  - *float2fake* conversions

- Graph morphisms and algebraic graph rewriting
  - *fake2true* conversions
Tree traversal and leaf replacement

- **Tree**: a directed graph $G$ whose associated undirected version is connected and acyclic

- **Rooted tree**: a tree where a node has been designated to be the *root*; nodes with no incoming edges are called *leaves* (we assume that the natural orientation of arcs is towards the root)

- **Tree traversal**: the process by which, starting from the root of a rooted tree, all leaves are identified

- **Leaf replacement**: the process by which a leaf is replaced by another leaf, or by a rooted tree whose root takes the place of the leaf
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Tree traversal and leaf replacement
Graph terminology - advanced

• **Source** and **target** of an arc:
  • \( s_G : E \rightarrow V, \ s_G((u, v)) := u \)
  • \( e_G : E \rightarrow V, \ e_G((u, v)) := v \)

• Let \( \Lambda \neq \emptyset \) denote a set of **labels**
• Let \( * \in \Lambda \) denote an **undefined** label

• **Attributed graphs**
  • Node labelling \( l_G : V \rightarrow \Lambda \)
  • Arc labelling \( m_G : E \rightarrow \Lambda \)
Functions between graphs

• Let \( L = (V_L, E_L) \), \( H = (V_H, E_H) \) be graphs

• Since a graph is a pair of sets, a function between graphs \( L, H \) is a pair \( g = (g_V, g_E) \) of functions
  • \( g_V : V_L \to V_H \)
  • \( g_E : E_L \to E_H \)
Preserving the information flow: morphisms

• Preserve the *structural* flow:
  1. \( s_H(g_E(e)) = g_V(s_L(e)), \forall e \in E_L \)
  2. \( t_H(g_E(e)) = g_V(t_L(e)), \forall e \in E_L \)

• Preserve the *semantic* flow:
  3. \( l_H(g_V(v)) = l_L(v), \forall v \in V_L \)
  4. \( m_H(g_E(e)) = m_L(e), \forall e \in E_L \)

• A function between graphs \( L, H \) that satisfies 1., 2., 3., 4. is called a **morphism**

• Can you think of a function between graphs which is not a morphism?
Algebraic graph rewriting

- **Graph rewriting rule:**
  - Context graph
  - Template graph and template core
  - Replacement graph and replacement core

- **Derivation:** recursive definition: application or sequence of derivations

- **Application point:** a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

• **Graph rewriting rule:**
  • Context graph
  • Template graph and template core
  • Replacement graph and replacement core

• **Derivation:** recursive definition: application or sequence of derivations

• **Application point:** a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

- **Graph rewriting rule:**
  - **Context graph**
  - **Template graph and template core**
  - **Replacement graph and replacement core**

- **Derivation**: recursive definition: application or sequence of derivations

- **Application point**: a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

• **Graph rewriting rule:**
  • Context graph
  • Template graph and template core
  • Replacement graph and replacement core

• **Derivation:** recursive definition: application or sequence of derivations

• **Application point:** a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

- **Graph rewriting rule:**
  - Context graph
  - Template graph and *template core*
  - Replacement graph and *replacement core*

- **Derivation**: recursive definition: application or sequence of derivations

- **Application point**: a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

• **Graph rewriting rule:**
  • Context graph
  • Template graph and template core
  • Replacement graph and replacement core

• **Derivation**: recursive definition: application or sequence of derivations

• **Application point**: a morphism; in practice we use type-checked isomorphisms
Algebraic graph rewriting

• **Graph rewriting rule:**
  • Context graph
  • Template graph and template core
  • Replacement graph and replacement core

• **Derivation:** recursive definition: application or sequence of derivations

• **Application point:** a morphism; in practice we use type-checked isomorphisms
Elevating a JIT graph to a PyTorch graph
Elevating a JIT graph to a PyTorch graph

Light blue nodes are identified

Working memory node
Elevating a JIT graph to a PyTorch graph

This graph is not acyclic!

Light blue nodes are identified

Working memory node
Elevating a JIT graph to a PyTorch graph

Light blue nodes are identified

Working memory node

Prune working memory nodes

This graph is not acyclic!
Projecting a computational graph
Projecting a computational graph

To memory partition

To kernel partition

We can work on simpler graphs!
Some last notes

• QuantLab and quantlib are released under the Apache 2.0 License
• This is a beta release: your feedback is our goal!
• Address communications to spmatteo@iis.ee.ethz.ch
Some last notes

• QuantLab and quantlib are released under the Apache 2.0 License
• This is a beta release: your feedback is our goal!
• Address communications to spmatteo@iis.ee.ethz.ch

Special thanks...
  ... for assisting with the development and proofreading the notebooks:
    Georg Rutishauser, Moritz Scherer
  ... for helping with the licensing and publication process:
    Manuel Eggimann, Frank Kagan Gürkaynak
We hope to see you at the next edition!