Machine Learning Experiments for Researchers

IMT School for Advanced Studies Lucca

Nick Korbit

Idea of This Tutorial

“Tools change fast, principles remain” ⚙️

Main goal: build intuition so ML codebases make sense across tools, languages, and IDEs.

Hands-on: we construct a full research experiment pipeline.

Reality: big models, big data, accelerators, long runs, heterogeneous data, fragile hyperparameters.

Research demands: reproducibility, reporting, rigor.

Logistics

- Short **theory blocks** (slides), followed by **live coding** in the cloud (Colab notebooks) and locally (PyCharm)

- All materials are available on **Github**: https://github.com/cor3bit/mle4r-winter26

Also: https://tinyurl.com/imt-ml and **QR** (on the right)

How to follow: download locally or open slides on github and notebook in Colab

- **Q&A: ask anytime** (interrupt me, it's fine!)

About the Instructor

Nick Korbit

- MS in Computer Science
- PhD student, DYSCO Lab
- Research: large-scale second-order methods for ML
- Open-source: JAX/Optax tooling for second-order optimization
- Previously:
  - ML Engineer, robotic delivery
  - Quant, risk modeling

Agenda

1. **Big Picture** - why ML experiments are hard today
2. **Dev Setup in 2026** - how we work
3. **Training Script (Vanilla)** - the minimal loop
4. **Training Script (Research-Grade)** - scaling and logging properly
5. **(Optional) Working with Text** - mini Transformer experiment
6. **(Optional) Hardware for ML** - get the right GPUs for your project

ML Universe Today

Training a Machine Learning Model Today

Reality (Constraint)	**Engineering Response**
🌊 Big data	Data loaders, streaming, batching
🧠 Big models	Memory awareness, batch size control
⏳ Long brittle runs	Seeds, configs, checkpoints
🧰 Tool explosion	We pick a thin slice: Python + JAX + W&B

Scaling Changes the Experimental Regime

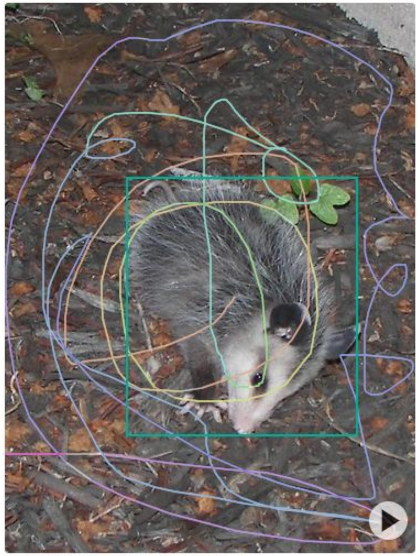
Scaling regimes: 7B → 70B → 400B+ (eg, GPT-3: 175 B (2020), Grok-1: 314B (2024), DeepSeek-V3: 671B (2024)).

At larger scale you usually can't afford: repeated full training runs, large grid searches, debugging in the main loop

Multiple tricks: gate expensive runs, extensive logging, structured configs, pilot runs

Large Models Require Large Datasets

Dataset	Domain	Approximate Size
Common Crawl	Text	~380 TB raw
FineWeb	Text	~40 TB
ImageNet (2012)	Vision	~155 GB
Open Images V7 (2022)	Vision	~18 TB


"In this image we can see an animal. There are leaves and wooden pieces on the land."

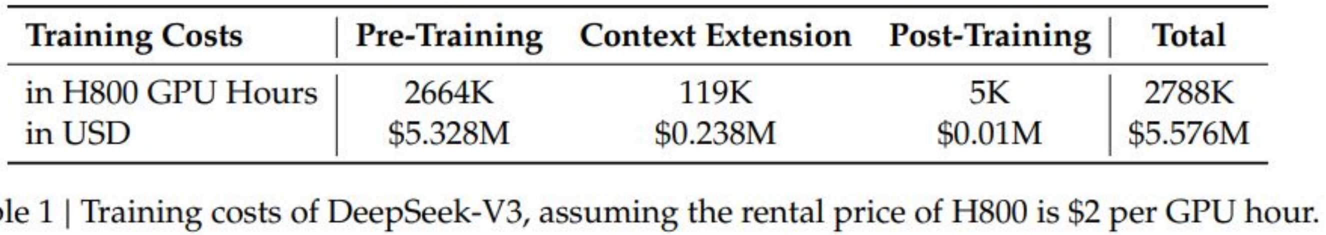
Toy: 10-50 GB (fits on disk, can "download and go")

Serious: 0.5-2 TB (needs streaming, caching, sharding)

Frontier: 10-100+ TB (distributed ingest, heavy filtering, provenance)

Training Is Long, Brittle And Costly

DeepSeek-V3



Training Costs	Pre-Training	Context Extension	Post-Training	Total
in H800 GPU Hours	2664K	119K	5K	2788K
in USD	\$5.328M	\$0.238M	\$0.01M	\$5.576M

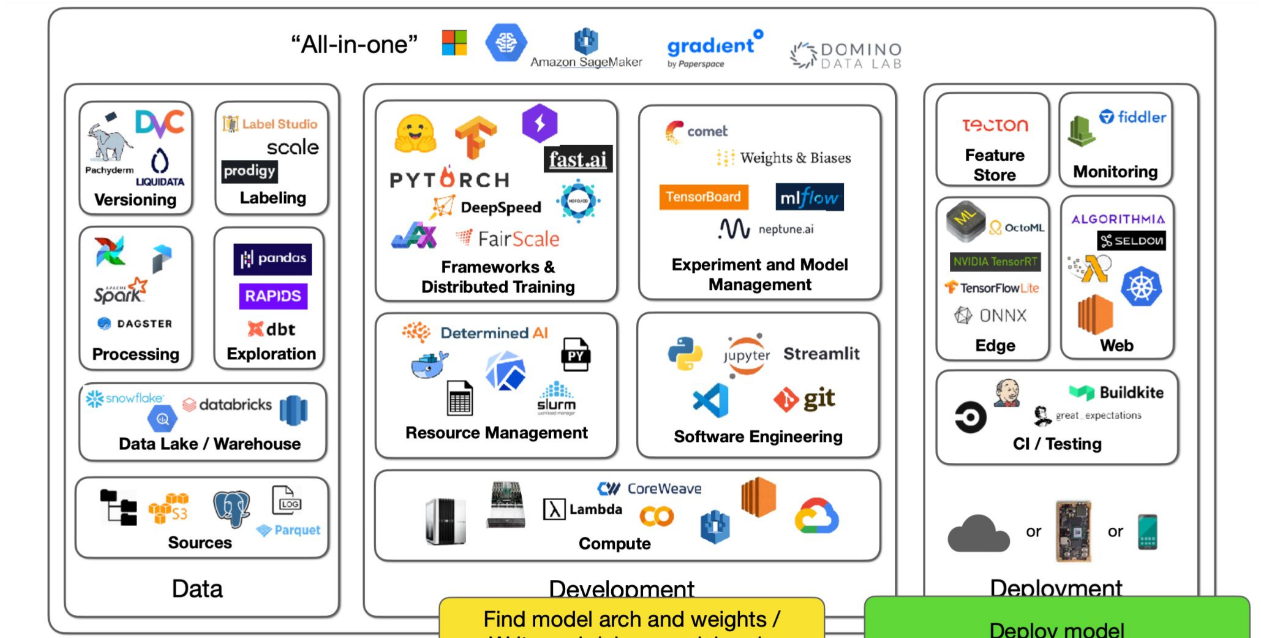
Table 1 | Training costs of DeepSeek-V3, assuming the rental price of H800 is \$2 per GPU hour.

LLaMAs: 8B - 1.46M GPU hours, 70B - 7M hours, 405B - 30.84 M hours.

GPT-4, 4o, o1, o3, 5, 5.2: Undisclosed

Why brittle? Sensitivity to hyper-parameters, stochasticity (seeding), software dependencies, bugs in code…

Instruments Are (Too) Abundant



Find model arch and weights / Write and debug model code

Deploy model

Source: FSDL 2022

Aggregate, process, clean, label, and version data

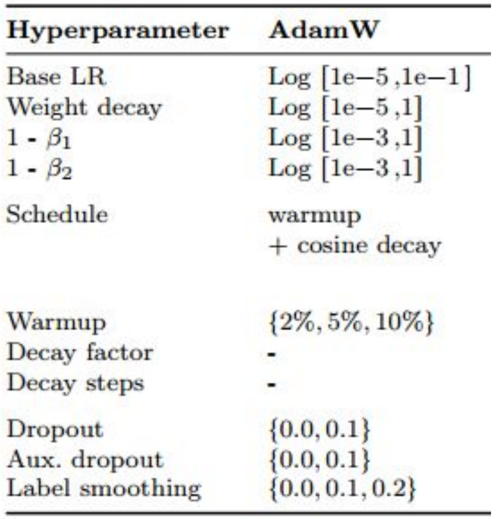
Run and review training experiments

Monitor predictions and close data flywheel loop

Benchmarks and HP Optimization

For robust evaluation several (tuned!) benchmarks should be considered. E.g. Dahl et al (2023), Schmidt et al (2021) **~1920 configurations to compare optimization algorithms**.

$$\begin{array}{cccc} \textbf{Problem} & \textbf{Optimizer} & \textbf{Tuning} & \textbf{Schedule} \\ \begin{Bmatrix} \text{P1} \\ \text{P2} \\ \dots \\ \text{P8} \end{Bmatrix}_8 \times & \begin{Bmatrix} \text{ADAM} \\ \text{NAG} \\ \dots \\ \text{SGD} \end{Bmatrix}_{15} \times & \begin{Bmatrix} \text{one-shot} \\ \text{small} \\ \text{medium} \\ \text{large} \end{Bmatrix}_4 \times & \begin{Bmatrix} \text{constant} \\ \text{cosine} \\ \text{cosine wr} \\ \text{trapez.} \end{Bmatrix}_4 . \end{array}$$



Hyperparameter	AdamW
Base LR	Log [1e−5 ,1e−1]
Weight decay	Log [1e−5 ,1]
1 - β_1	Log [1e−3 ,1]
1 - β_2	Log [1e−3 ,1]
Schedule	warmup + cosine decay
Warmup	$\{2\%, 5\%, 10\%\}$
Decay factor	-
Decay steps	-
Dropout	$\{0.0, 0.1\}$
Aux. dropout	$\{0.0, 0.1\}$
Label smoothing	$\{0.0, 0.1, 0.2\}$

Key Take: We need *structured* experiments

Chaos

- Ad-hoc notebooks
- Manual hyperparameter tweaks
- No seed control
- Screenshots of results

Structure

- Parameterized scripts
- Config files
- Fixed seeds
- Logged metrics
- Comparable runs

Dev Set-up

In 2026

Tool Selection - Software

A minimalist ML setup for reproducible experiments:

- OS: Linux (Ubuntu) - best-supported for GPU-accelerated ML dev
- Shell + scripts: (schedule server-side experiments mostly)
- Programming Language: Python + JAX Ecosystem
- IDE: VSCode / PyCharm
- Version Control: Git
- Experiment Tracking: Weights & Biases

Skill Refresher: MIT Missing Semester https://missing.csail.mit.edu/

IDE Choice in 2026

What matters

🐞 Debugger + profiling

🔍 Code navigation (jump to def, find usages)

🧪 Test runner integration

🌐 Remote dev (SSH, containers, cluster)

🤖 Autocomplete + AI assist

Editors

Classic: PyCharm/VSCode + plugins

Emerging (“AI-first”): Cursor, Zed

As long as you have the features on the left, any IDE is fine ✅

Student perks 🎓 Many tools offer academic tiers (free/discounted): PyCharm, Github Copilot, cloud providers (Azure, AWS), Overleaf, W&B.
Rule: always search “academic/student tier”.

Python in One Slide

Why Python? Huge ecosystem (JAX/PyTorch, data, plotting, tooling), fast iteration + good glue language, performance comes from compiled backends (XLA/CUDA).

Nice, but…

- Python is interpreted-ish: pure Python loops are slow; vectorize / JIT when possible
- Dependencies can break runs: pin versions, avoid global installs
- Reproducibility needs discipline: record seed + package versions + git commit

Environment setup (minimum viable)

- *Basic*: venv + pip (+ requirements.txt)
- *Pro*: uv (fast installs + lockfiles)

Setup Checklist

- GitHub account
- Git installed (run `git --version`)
- IDE installed (VSCode/PyCharm)
- AI assist (optional but useful)
    - GitHub Copilot installed + signed in (VSCode/PyCharm plugin)
- Weights & Biases (W&B) account

The Anatomy of the Training Script

Training Script, Big Picture

ML is iterative refinement: we repeatedly evaluate performance and adjust parameters.

Mental model (archery analogy):

- - 🏹 Data - the arrows
- - 🎯 Performance metric - distance to the center
- - ⚙️ Model - the bow
- - 🔧 Optimizer - adjusting grip, tension, bowstring

Loop: shoot → measure → adjust → repeat

Data

Data Types and Tasks

Data types:

- Tabular 📊
- Image 🖼️
- Video 🎥
- Audio 🎵
- Text/Code 📜
- Specialized: graphs, MRI, EEG.. 🔬

Data tasks:

- Regression
- Classification
- Time Series Forecasting
- Question Answering
- Text Generation
- Summarization
- …

Data type and task determine the choice of machine learning model, preprocessing steps, and evaluation metrics.

Dataset Aggregators

🔍 "**Browsers**": given a task, explore available datasets

🪩🪩 HuggingFace Datasets: https://huggingface.co/datasets

🪩 Kaggle Datasets: https://www.kaggle.com/datasets

Google Dataset Search: https://datasetsearch.research.google.com

📦 **Python Bindings**: Python API to selected datasets

- Scikit-learn `pip install scikit-learn`
- LibSVM `pip install libsvmdata`
- TensorFlow Datasets `pip install tensorflow-datasets`
- HuggingFace Datasets `pip install datasets`
- Pytorch Vision `pip install torchvision`

Data Loaders

Why they exist: keep the accelerator busy (throughput) and make training reproducible.

- Batching: load multiple samples at once to optimize GPU usage
- Shuffling: reduce ordering bias (better SGD behavior)
- Streaming: iterate without loading all data into RAM
- Transforms: preprocessing/augmentation in the input pipeline

Examples: PyTorch DataLoader, TensorFlow tf.data.Dataset, HuggingFace datasets.

Data Processing

Processing pipeline depends heavily on the data type.

Tabular: scaling numerical features, encoding categorical features.

See Lecture 1 of the ML course and scikit-learn guide for Python implementation: https://scikit-learn.org/stable/modules/preprocessing.html

Text: “tokenizing” the features is the process of converting raw text into smaller units (tokens) that a model can process.

Rule: preprocessing is part of the experiment pipeline. Log the choices so runs are comparable.

Data Caveats (Non-goal Today) ⚠️

Diversity of data - adding code, multiple languages, images help LLMs generalize better.

Bias in data - e.g., gender bias, socioeconomic factors, ethnicity, dialects. LLMs capture “the values” of the training data snapshot. Consider de-biasing techniques: filtering the data (e.g., FineWeb), reweighting the data, loss function modifications.

Privacy concerns - LLMs can memorize the data and reveal private information, e.g, API keys, passwords, addresses, parts of the train data etc.

Model

Anatomy of the Neural Network Model Class

Mathematically, a function that transforms inputs into outputs of the form

$$f(w; x) = \hat{y}$$

In implementation (software) terms,

- **weights**, dictionary-style structure, stored as tensors, trainable parameters;
- **forward pass**, inference code, transforms inputs into outputs layer by layer;
- ***backward pass**, computes gradients for training;
- **metadata and configs**, stores hyperparameters, architecture, checkpoints.

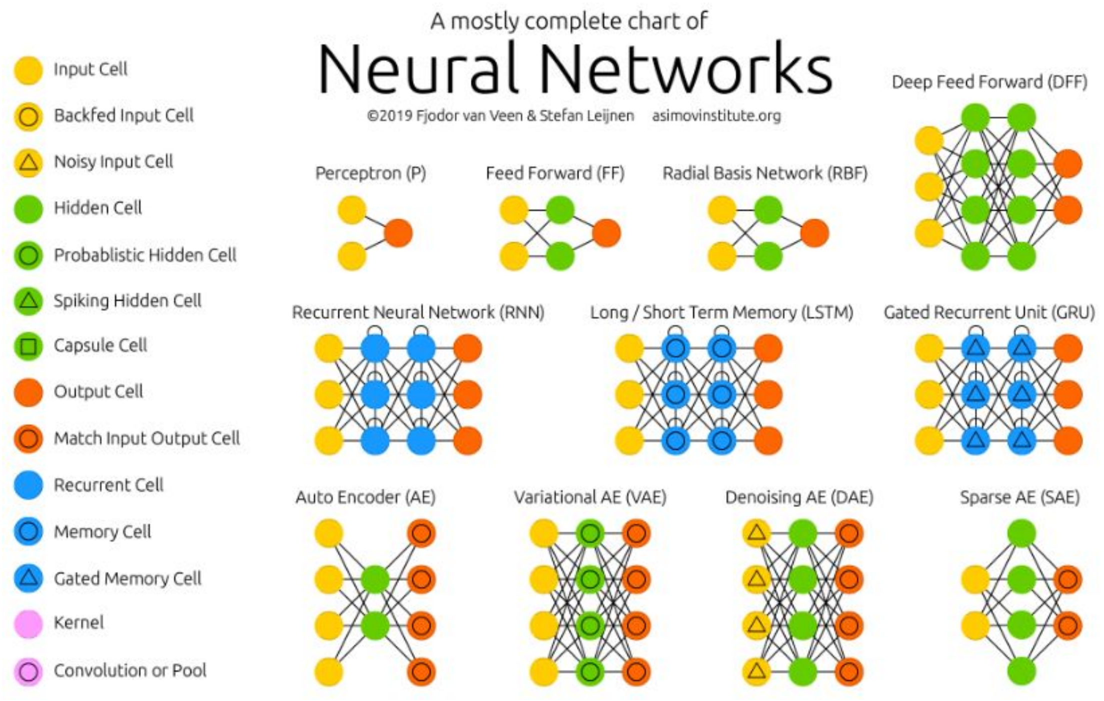
*backward pass is typically implemented automatically by the framework, such as PyTorch or JAX.

Model Selection

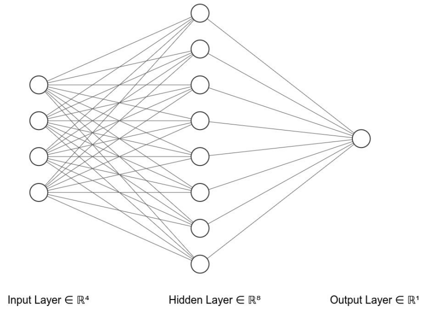
Neural Network **architecture encodes knowledge about the problem.**

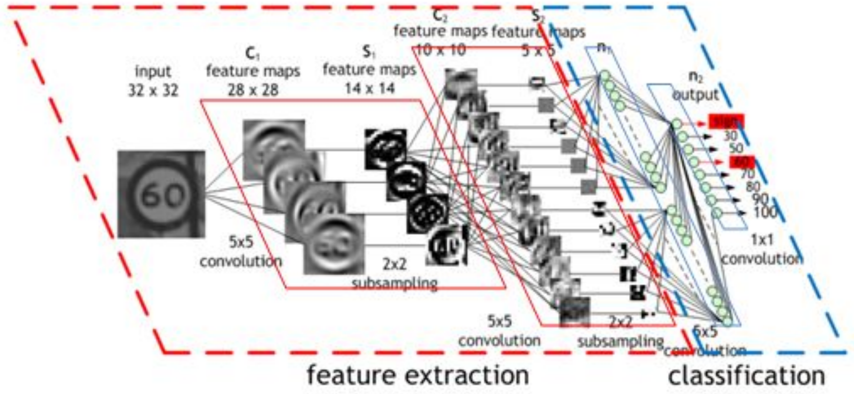
See: https://www.asimovinstitute.org/neural-network-zoo/

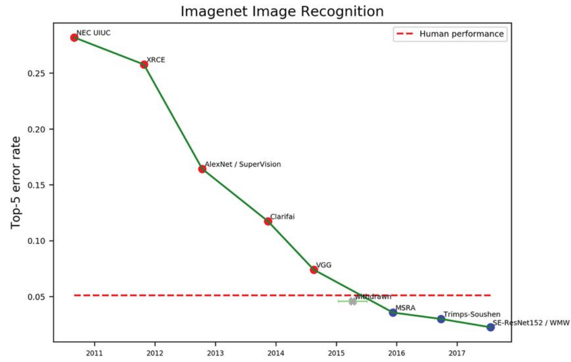
Typical architectures: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Transformer, Diffusion model, Autoencoder (AE).



Case of CNNs







CNN: local connectivity (~pixels near each other are more related than distant ones) + parameter sharing (the same filter is applied across the whole image).

Image: https://developer.nvidia.com/discover/convolutional-neural-network

Model Card

A **Model Card** 📝 is a structured report for a model: what it is, how it was trained, and how to interpret results. Goal: transparency + reproducibility + responsible use.

Minimum contents (what we care about):

- Model: architecture + parameters / tokenizer (if text)
- Training: data sources + preprocessing + objective + compute + key hyperparameters
- Evaluation: benchmarks, metrics, and settings (splits, prompts, decoding)

Examples:

- LLaMA 3 llama3/MODEL_CARD.md at main
- Gemini 1.5 Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context
- DeepSeek R1 deepseek-ai/DeepSeek-R1 · Hugging Face

Model Zoo

A **Model Zoo** 🦁 is a repository of **pre-trained ML models** that can be downloaded and used for various tasks without training from scratch.

🪩🪩🪩 HuggingFace Hub https://huggingface.co/models

🪩 TorchVision https://pytorch.org/vision/0.20/models.html

🪩 Kaggle Hub https://www.kaggle.com/models/

Also, for language models: https://huggingface.co/docs/transformers/index

Model Bookkeeping

Configurations (Hyperparameters & Settings):

- Defines model architecture, optimizer settings, batch size, learning rate, etc;
- Stores configs in YAML, JSON, or Python dictionaries for easy reloading.

Load pre-trained weights / Save weights (checkpointing)

- Enables fine-tuning & resuming training;
- Prevents loss of work & supports reproducibility.

Optimizer and Loss

Performance Metrics

Given a batch of data (x,y) and a model f(x;w), the **loss function** L(w) tells how well the model's predictions match the ground truth.

The choice of loss functions is wide:
https://scikit-learn.org/stable/modules/model_evaluation.html

For regression, **MSE**: $\mathcal{L}(y, \hat{y}) = \frac{1}{b} \sum_{i=1}^{b} (y_i - \hat{y}_i)^2$

For classification, **Cross-entropy**: $\mathcal{L}(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$

Note: The choice of the loss function matters! See: https://arxiv.org/abs/2204.12511

Gradient-based Optimization

Make a small step in the direction where loss is decreasing. The basic form is Stochastic Gradient Descent (SGD):

$$w_{t+1} \leftarrow w_t - \alpha \nabla \mathcal{L}$$

What can be done to **accelerate the descent**?

- Momentum (temporal averaging);
- Normalization of the gradient;
- Gradient clipping;
- Line search for the learning rate;
- Penalizing large weights…

Automatic Differentiation

Key idea: write a scalar loss function, get derivatives “for free”.

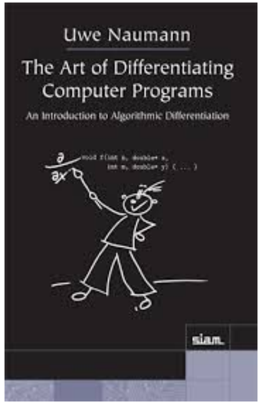
AD computes exact derivatives (up to floating-point) by tracing the computation and applying the chain rule.

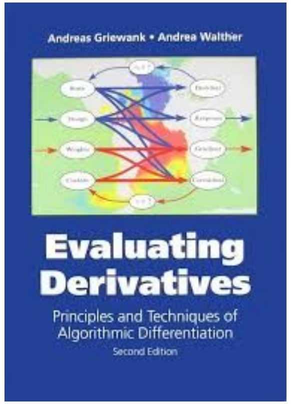
Why to use the AD+ML framework?

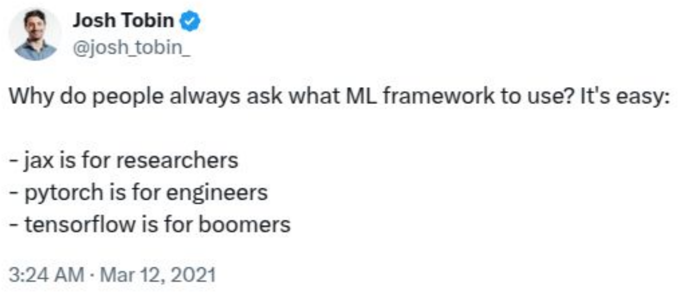
- Calculate gradients and Hessians
- Support for hardware accelerators: GPU/TPU

In this course: we use JAX/Flax.

See: [PML Book 1, Section 13.3]()


Uwe Naumann
The Art of Differentiating Computer Programs
An Introduction to Algorithmic Differentiation
siam


Andreas Griewank • Andrea Walther
Evaluating Derivatives
Principles and Techniques of Algorithmic Differentiation
Second Edition


Josh Tobin
@josh_tobin_

Why do people always ask what ML framework to use? It's easy:

- jax is for researchers
- pytorch is for engineers
- tensorflow is for boomers

3:24 AM · Mar 12, 2021

Optimizer Implementation

JAX Optax API, a sequence of transformations to the gradient:

1. Initialization of the optimizer class

```
optimizer = optax.adam(learning_rate=0.001)
```

2. Initialization of the state (~internal parameters of the solver)

```
opt_state = optimizer.init(params)
```

3. Update function

```
updates, new_opt_state = optimizer.update(grads, opt_state)

new_params = optax.apply_updates(params, updates)
```

Training Loop

Training Loop Pseudocode

A **training loop** integrates the **data**, **model**, **loss function**, and **optimizer** into a structured iteration process.

```
Until good_fit():
    Sample a batch from the dataset
    # Forward pass
    Compute predictions: y_pred = model(x_batch, params)
    Compute loss: loss = loss_fn(y_pred, y_batch)
    # Backward pass
    Compute gradients: grads = ∂L/∂w   # Backpropagation or AD
    Update parameters: params = params + f(grads)
```

Training Script

Research Grade

The Standards For ML Research Are High

Formal requirements for running ML experiments - “NeurIPS checklist”:

https://neurips.cc/public/guides/PaperChecklist

Very demanding! Cannot submit a paper unless you checkbox all the requirements, including:

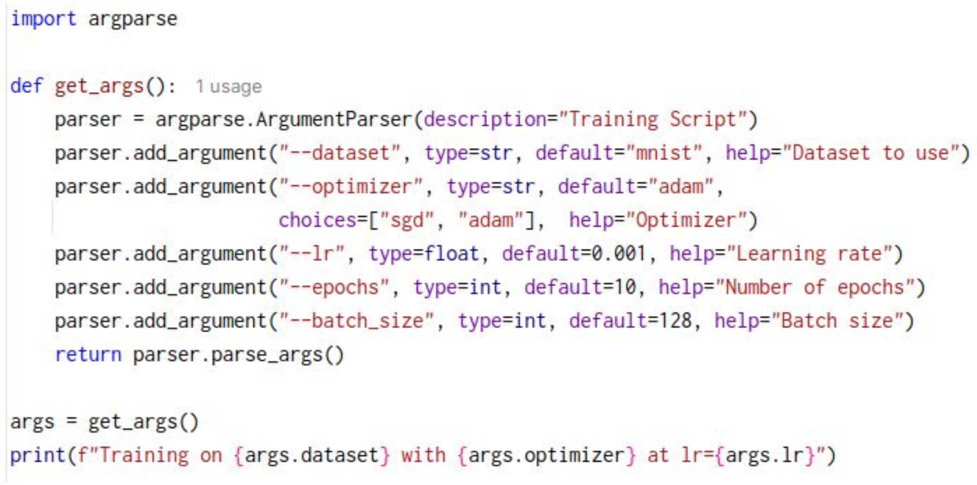
- Experimental Result Reproducibility - describe the architecture and algorithm;
- Open Access to Data and Code;
- Experimental Setting/ Details - describe in full the set-up;
- Experiment Statistical Significance - method and error bars;
- Experiments Compute Resource - describe the hardware and total compute.

From Training Loop To Training Script

Vanilla loop: single run, single config, single device.

Research-grade: parameterized, tracked, repeatable, comparable runs.

Ideally, we want to run several versions of the training loop in parallel, testing several configurations (different datasets, models, optimization hyper-parameters).



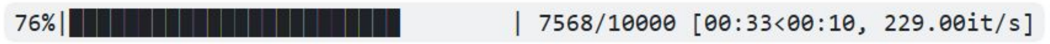
```python
import argparse

def get_args():  1 usage
    parser = argparse.ArgumentParser(description="Training Script")
    parser.add_argument("--dataset", type=str, default="mnist", help="Dataset to use")
    parser.add_argument("--optimizer", type=str, default="adam",
                        choices=["sgd", "adam"],  help="Optimizer")
    parser.add_argument("--lr", type=float, default=0.001, help="Learning rate")
    parser.add_argument("--epochs", type=int, default=10, help="Number of epochs")
    parser.add_argument("--batch_size", type=int, default=128, help="Batch size")
    return parser.parse_args()


args = get_args()
print(f"Training on {args.dataset} with {args.optimizer} at lr={args.lr}")
```

Experiment Tracking

Effective experiment tracking ensures reproducibility, helps analyze model performance, and accelerates hyperparameter tuning.

Logging: print, `logging` module.

Progress bar: print, tqdm.



```
76%|███████████████████████████        | 7568/10000 [00:33<00:10, 229.00it/s]
```

Tracking performance metrics:

- *Local* - files, DB, TensorBoard;
- *Remote tracking* - many proprietary solutions (e.g., Weights & Biases, MLFlow, Neptune AI, Comet ML)

🪩🪩🪩 Our choice: minimal print() + tqdm + TensorBoard + W&B.

Hyperparameter Optimization (HPO) Matters

Hyperparameters (HPs) control the learning process. Optimizing HPs can **significantly** boost model accuracy and efficiency.

Category	**Examples**	**Impact**
Model Architecture	Number of layers, hidden size, attention heads, sequence size	Controls model capacity and expressivity
Optimization	Learning rate, optimizer type (SGD, Adam, AdamW)	Determines speed and stability of training
Batching & Data	Batch size, data augmentation, tokenization method	Affects training stability and generalization
Regularization	Dropout, weight decay	Prevents overfitting

HPO Approaches

- **Grid Search** → Simple but expensive: tests all combinations;

- **Random Search** → More efficient: samples random values;

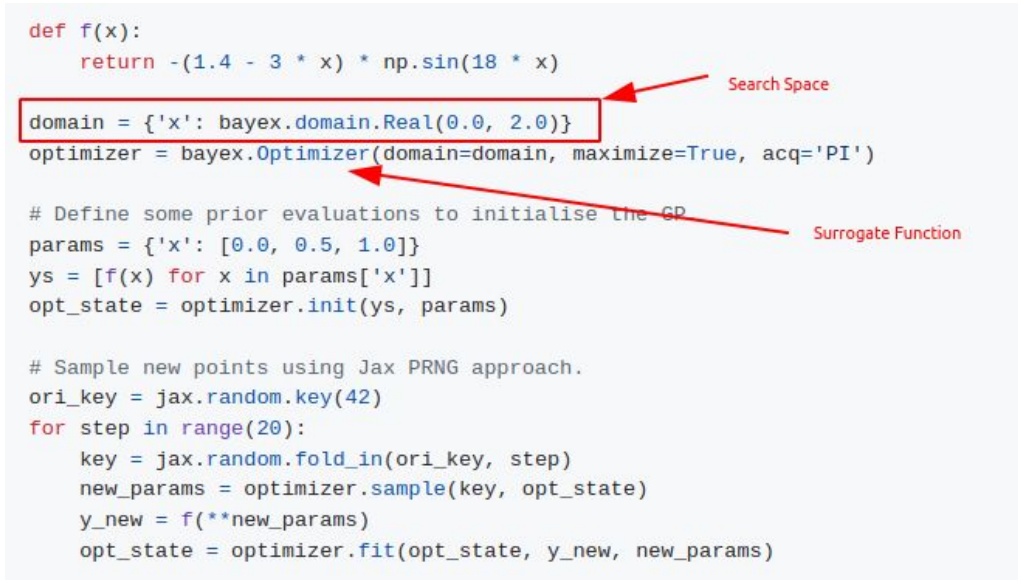
- **Bayesian Optimization** → Learns from previous runs, focuses on promising regions;

- **Population-Based Training (PBT)** → Dynamic tuning based on evolving populations.

👍 Rule of thumb: try simple methods first (grid/random), then explore smarter methods (BO/PBT).

HPO In Practice

1) Schedule several runs in bash, compare the results in W&B (~grid search like);
2) Use bayex, GPJax, NumpPyro for Bayesian Optimization.



```
def f(x):
    return -(1.4 - 3 * x) * np.sin(18 * x)

domain = {'x': bayex.domain.Real(0.0, 2.0)}
optimizer = bayex.Optimizer(domain=domain, maximize=True, acq='PI')

# Define some prior evaluations to initialise the GP.
params = {'x': [0.0, 0.5, 1.0]}
ys = [f(x) for x in params['x']]
opt_state = optimizer.init(ys, params)

# Sample new points using Jax PRNG approach.
ori_key = jax.random.key(42)
for step in range(20):
    key = jax.random.fold_in(ori_key, step)
    new_params = optimizer.sample(key, opt_state)
    y_new = f(**new_params)
    opt_state = optimizer.fit(opt_state, y_new, new_params)
```

Training a (mini) Transformer

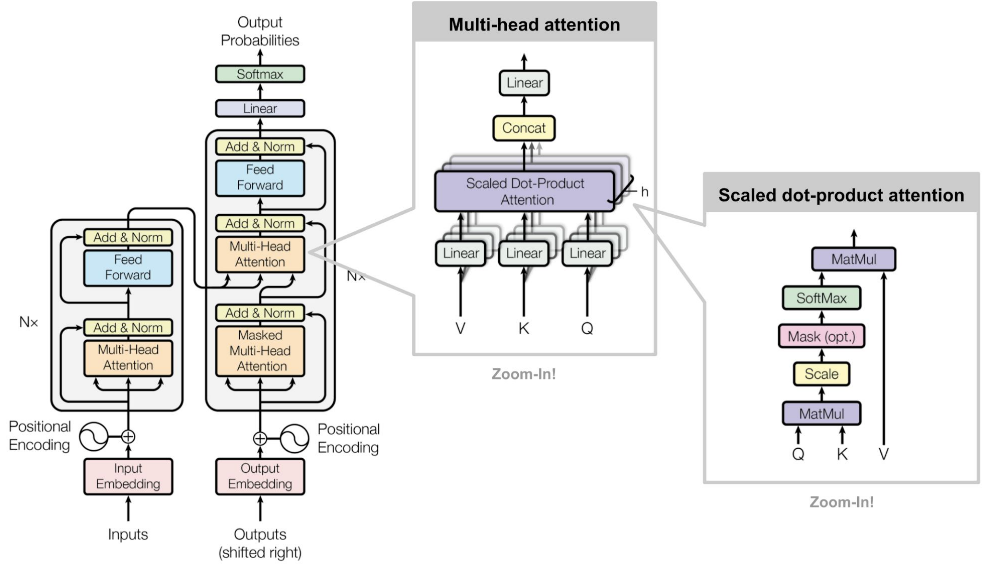
Optional Material

Attention and Transformer

Uses an **attention mechanism** to process input data all at once, rather than one piece at a time like previous sequence models (e.g., RNNs). Steps:

1. Chop data into small bits (tokens);
2. Make tokens “talk” to each other (attention);
3. Learn several patterns at once (multi-head);
4. Combine the outputs with the Dense layer.

Why important? Parallel processing, can be applied to any sequence (audio, video, image, time series, decisions.. All together?)



Transformer model chart by Lilian Weng: https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/transformer.png

More visualization: https://bbycroft.net/llm

More on LLMs, Attention and Transformers

More references:

Transformers from Scratch: https://e2eml.school/transformers.html

Transformer Model Family:
https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/

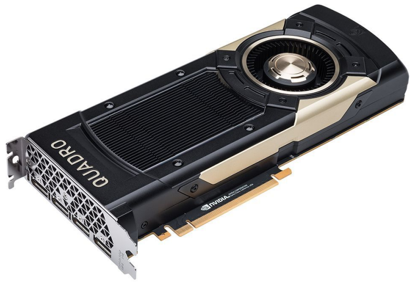
Attention Visualization (video): https://youtu.be/eMlx5fFNoYc

Coding a GPT-2 with Karpathy (video): https://youtu.be/kCc8FmEb1nY

Hardware for ML

Optional Material

Hardware For ML



Assuming a non-distributed training:

Disk - stores the full dataset.

CPU w/ RAM - OS processes, IDE, orchestration and scheduling, data preparation and transformation, (hyper-)parameter storage, configs, checkpointing. 16+ cores, 64GB–128GB+

GPU w/ VRAM - computes the gradients, does the optimization loop; stores a batch of data, model weights, optimizer states and gradients. Other considerations: FLOPS, VRAM bandwidth, connector speed, mixed precision. 8–12GB (small scale ML), 24GB+

Other - connectors (eg. RAM - GPU), power supply.

How to Assess the Hardware Requirements

1️⃣Start with a problem (Task + Dataset), ~min Disk, RAM, VRAM size

2️⃣Find a benchmark (Model + Optimizer) ~GPU parameters

eg. CIFAR100+ResNet18 trained with SGD.

Disk, ~160 MB dataset, so that disk space is not a bottleneck.

CPU/RAM, ~16GB DDR4, not a bottleneck, preload and transform the full dataset.

GPU/VRAM, more complicated: model weights ~40 MB (11M params * 4B each), activations x4, gradients 40 MB, optimizer states 40MB, overheads (!) 2GB. ~3GB overall.

If Your Training Is Slow

✅ **Optimize data loading**: Use SSDs, increase num_workers, enable prefetch().

✅ **Ensure GPU is fully utilized**: Check where the data is stored and calculations are computed (JAX device), increase batch size if possible.

✅ **Look at precision**: Check whether the GPU natively supports mixed precision, consider switching from FP32, to FP16, FP8 for some operations.

✅ **Reduce CPU bottlenecks**: Use parallelized data loading (num_workers>0).

✅ **Check VRAM usage**: Reduce batch size if out-of-memory errors occur.

✅ **Try a different optimizer**: e.g., AdamW, second-order solvers.

✅ **Scale on multi-GPU**: Use DistributedDataParallel framework.

Research Idea Funnel

“Why I should **not** conduct this experiment?”, the “red flags” checklist. Given the requirements, before conducting an experiment, better to assess the viability of the idea.

❌ Is the running time of a single experiment run too long?

❌ Do I lack the computational resources (e.g., GPUs) required to scale the experiment appropriately?

❌ Am I relying on proprietary tools, datasets, or code that others cannot easily access?

❌ Not aligned with state-of-the-art (SOTA): if the experiment ignores recent breakthroughs, results may be outdated or irrelevant. (Browse literature first!)

❌ Hyperparameter tuning is unfeasible: does finding the right learning rate, batch size, optimizer require excessive compute?

❌ Too much custom engineering required: if setting up the experiment requires writing an entire new framework, it may be an overreach (personal experience).

Thank you!

Nick Recommends: Theory

- PML by Kevin Murphy

https://probml.github.io/pml-book/book1.html

https://probml.github.io/pml-book/book2.html

- Math for ML by Deisenroth, A. Aldo Faisal, and Cheng Soon Ong

https://mml-book.github.io/

- Probabilistic Numerics by Philipp Hennig, Michael A. Osborne, Hans Kersting

https://www.probabilistic-numerics.org/textbooks/

- Lilian Weng, Lil’Log

https://lilianweng.github.io/

Nick Recommends: Practice

Andrej Karpathy, “from Scratch” approach: https://github.com/karpathy/nanochat

Sebastian Rashka: https://magazine.sebastianraschka.com/

Chip Huyen: https://github.com/chiphuyen/aie-book/blob/main/resources.md

MIT Missing Semester: https://missing.csail.mit.edu/

HF Learn: https://huggingface.co/learn