

# Machine Learning Experiments for Researchers

IMT School for Advanced Studies Lucca

Nick Korbit

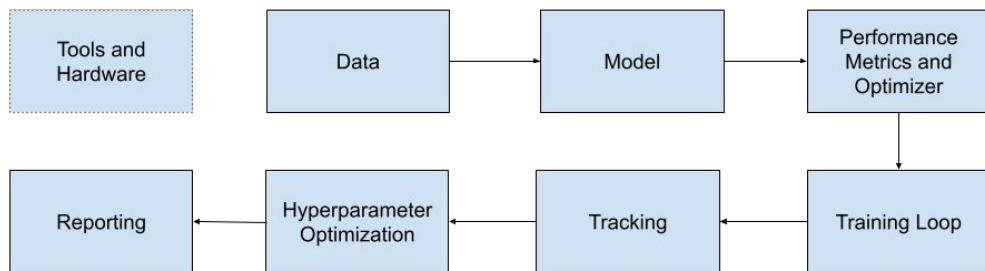
# What Is This Tutorial About

Welcome to a hands-on deep dive into running research experiments in the modern ML landscape using Python and JAX/Flax.

- **“Modern ML”**: handling large datasets and models across diverse data types, leveraging efficient hardware utilization and scalable training techniques;
- **Designed for researchers**, looking to refine their experimental workflow;
- **“Head-first” methodology**: we build the full research pipeline, from data preparation to training, evaluation, and reporting;
- **Big picture first, references later**: we prioritize intuition and understanding the “why”, while also providing pointers for deeper exploration.

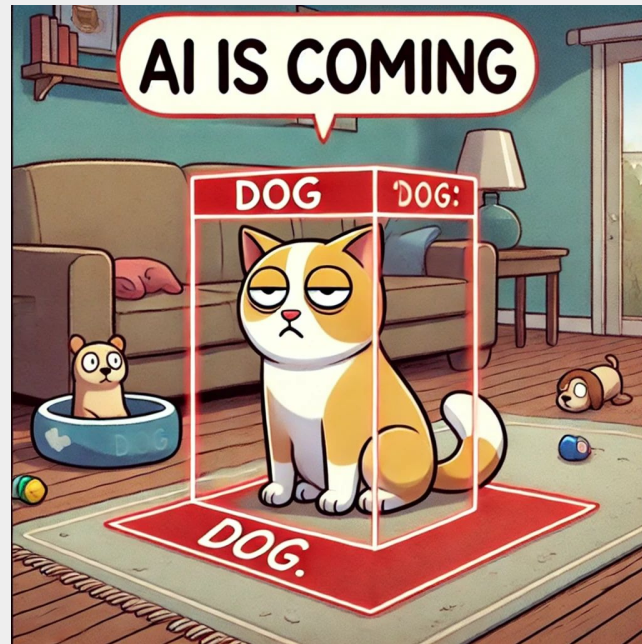
# Logistics

- Our project is to exploring working with text, and to train a mini Transformer model on a small text corpus.
- Step-by-step building the full ML research pipeline:



- A bit of the theory (slides), followed by interactive coding (Jupyter notebooks).
- The code is available at: <https://github.com/cor3bit/mle4r-winter25>
- Questions are welcome throughout the course!

# Introduction



# Training a Machine Learning Model Today

- Datasets are big;
- Models are big;
- Training is long, brittle and costly;
- Multi-modal training (data types beyond tabular: images, audio, video, medical data);
- Lots of available tools to choose from: datasets, programming languages, frameworks, models, optimizers;
- Comparison with benchmarks;
- Uncertainty estimation.

# The World of LLMs

- **GPT-3** (2020): 175 B parameters
- **Grok-1** (2024): 314 B, ~320 GB
- **DeepSeek-V3** (2024): 671 B; **-R1**: 1.5-70 B
- **GPT-4, 4o, o1, o3**: Undisclosed
- **LLaMA** model family (below)

Multilingual

## Llama 3.1

- 8B: Light-weight, ultra-fast model you can run anywhere.
- 405B: Flagship foundation model driving widest variety of use cases

[Download models](#)

Lightweight and Multimodal

## Llama 3.2

- 1B and 3B: Light-weight, efficient models you can run everywhere on mobile and on edge devices.
- 11B and 90B: Multimodal models that are flexible and can reason on high resolution images.

Multilingual

## Llama 3.3

- 70B: Experience leading performance and quality at a fraction of the cost with our latest release.

[Download models](#)

AI

X1



MISTRAL  
AI



OpenAI



Google AI



Meta



deepseek

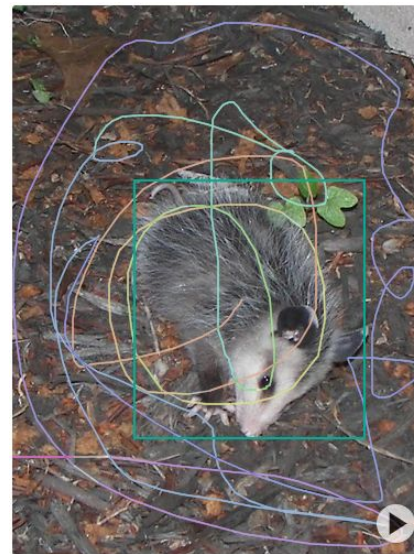


Qwen

# Large Models Require Large Datasets

Some common datasets to train large models are:

Dataset	Domain	Approximate Size
Common Crawl	Text	~380 TB raw
C4 (Colossal Clean Crawled)	Text	~750 GB
FineWeb	Text	~40 TB
OpenWebText (2019)	Text	~38 GB
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."

# 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.

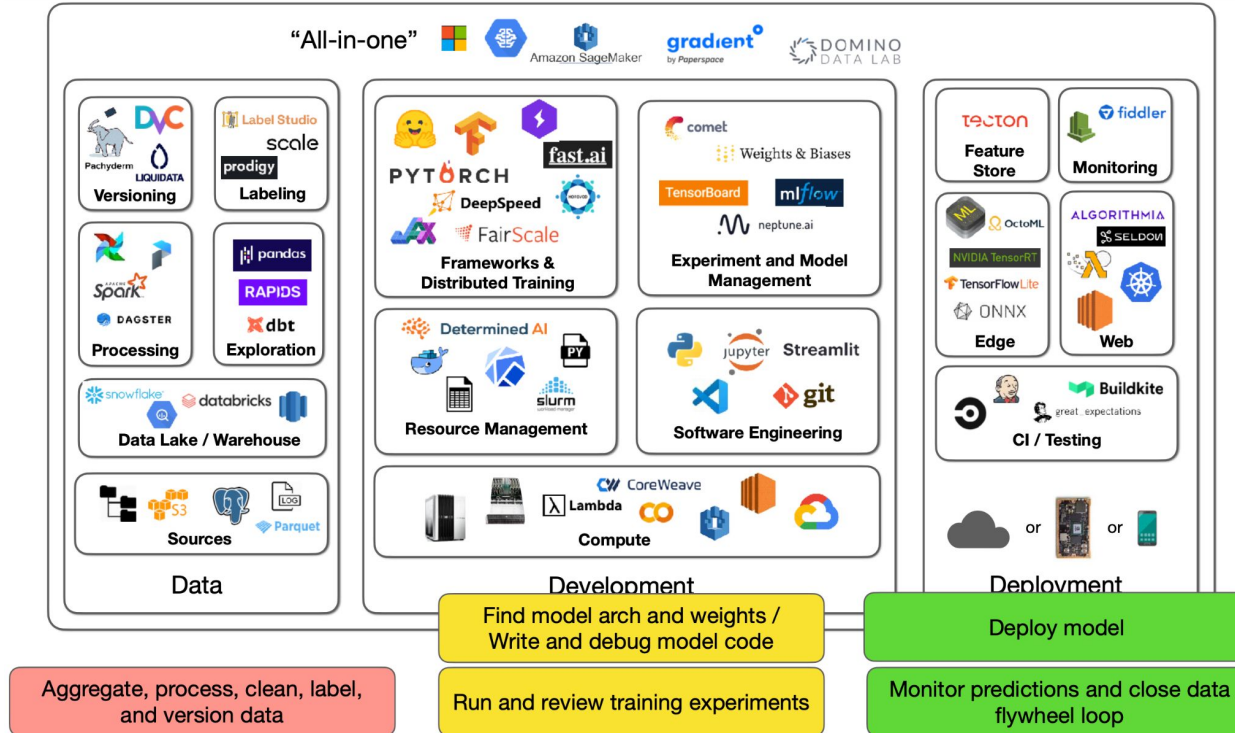
**LLaMAs:** 8B - 1.46M GPU hours, 70B - 7M hours, 405B - 30.84 M hours.

**GPT-4, 4o, o1, o3:** Undisclosed

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



# Instruments Are (Too) Abundant



Source: FSDL 2022

# 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.

# 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.

$$\left\{ \begin{matrix} \text{P1} \\ \text{P2} \\ \dots \\ \text{P8} \end{matrix} \right\}_8 \times \left\{ \begin{matrix} \text{ADAM} \\ \text{NAG} \\ \dots \\ \text{SGD} \end{matrix} \right\}_{15} \times \left\{ \begin{matrix} \text{one-shot} \\ \text{small} \\ \text{medium} \\ \text{large} \end{matrix} \right\}_4 \times \left\{ \begin{matrix} \text{constant} \\ \text{cosine} \\ \text{cosine wr} \\ \text{trapez.} \end{matrix} \right\}_4 .$$

Hyperparameter	AdamW
Base LR	Log [1e-5, 1e-1]
Weight decay	Log [1e-5, 1]
$1 - \beta_1$	Log [1e-3, 1]
$1 - \beta_2$	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}

# Tools and Hardware



# Tool Selection - Software

A (somewhat subjective) minimalist machine learning set-up:

- Bash scripting (schedule server-side experiments mostly)
- Language: Python + JAX/Flax
- IDE: VSCode / PyCharm
- Experiment Tracking: Weights & Biases

**Refresher:** MIT Missing Semester <https://missing.csail.mit.edu/>

# Hardware For ML

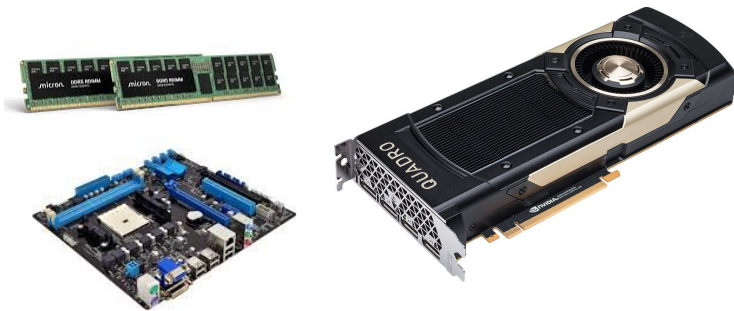
Assuming a non-distributed training:

**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

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

② Find 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).

# Training Script, Big Picture

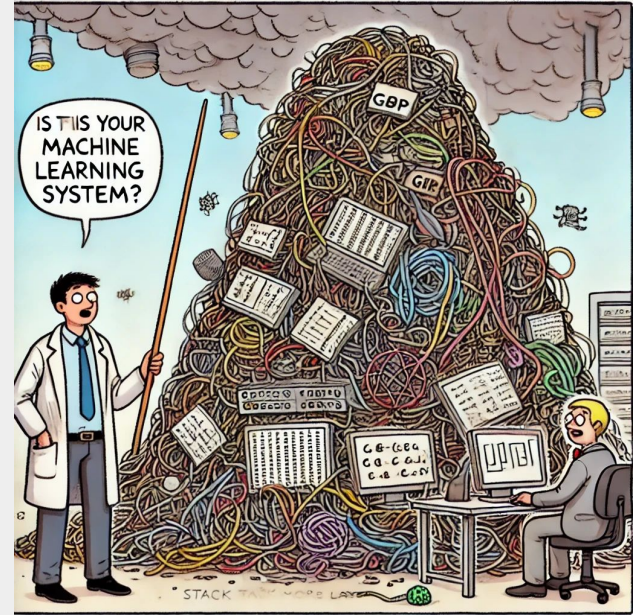
Shooting the target and seeing what happens.

Key parts:

- Data (~arrows)
- Model (~bow)
- Performance metric (~distance to the center of the target)
- Optimization algorithm (~adjust the grip, tune the bowstring)







In ML, can be done infinitely, just with `while True`.

# 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



PapersWithCode:

<https://paperswithcode.com/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

Additional considerations while working with data:

- Batching: load multiple samples at once to optimize GPU usage;
- Shuffling: prevent model overfitting to specific sequences;
- Streaming: load large datasets efficiently from disk;
- Preprocessing on the Fly: apply transformations while loading.

**Data Loader** is a specific class that automates loading, batching, shuffling, and preprocessing of datasets, ensuring efficient training.

Examples are PyTorch's `DataLoader`, TensorFlow's `tf.data.Dataset`, and Hugging Face's `datasets.load_dataset`.

# 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.

Main types: Character-level, Word-level, Subword-based (BPE, WordPiece, SentencePiece)

# Other Issues

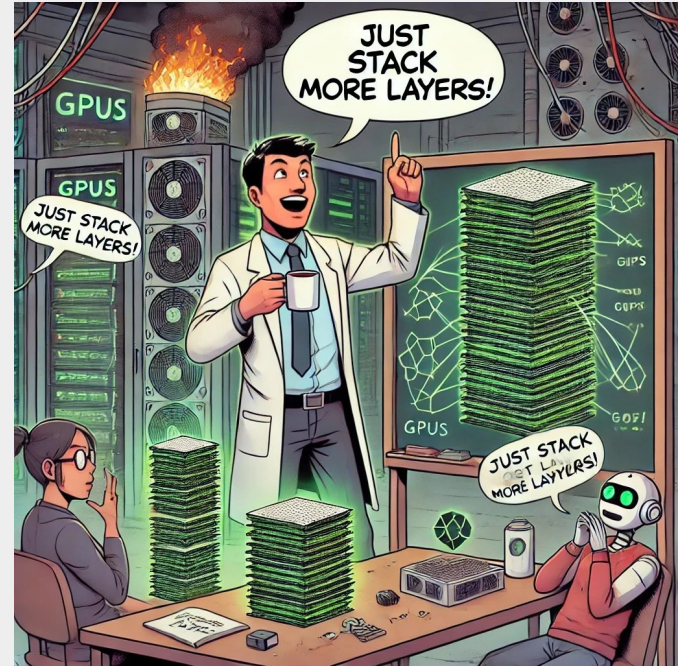
**Diversity of data** - adding code, multiple languages, images help LLMs generalize better.

**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. Data filtering (again!)



# 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/>

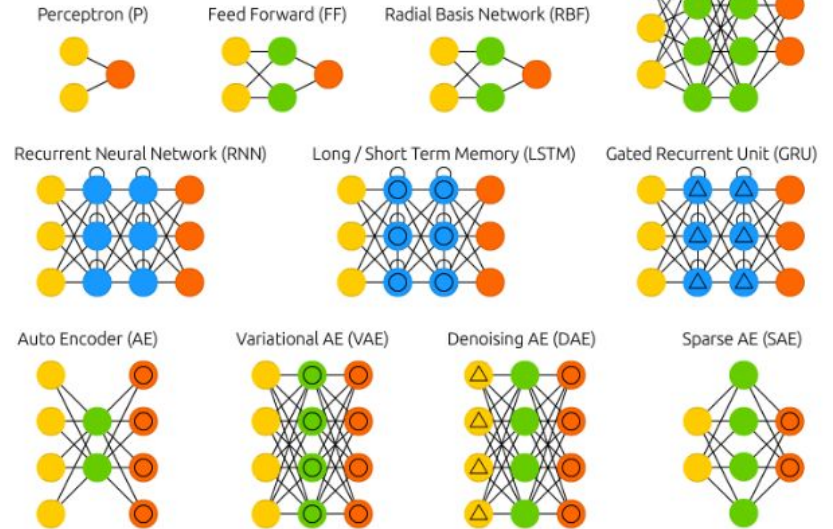
## Typical architectures:

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

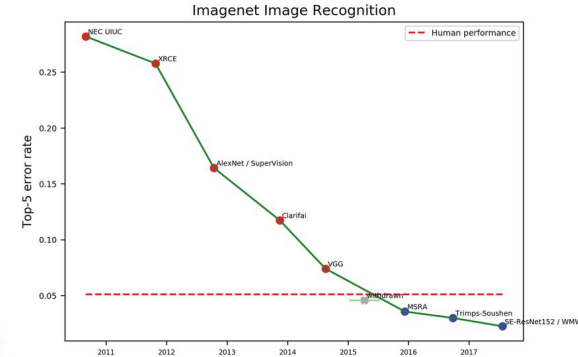
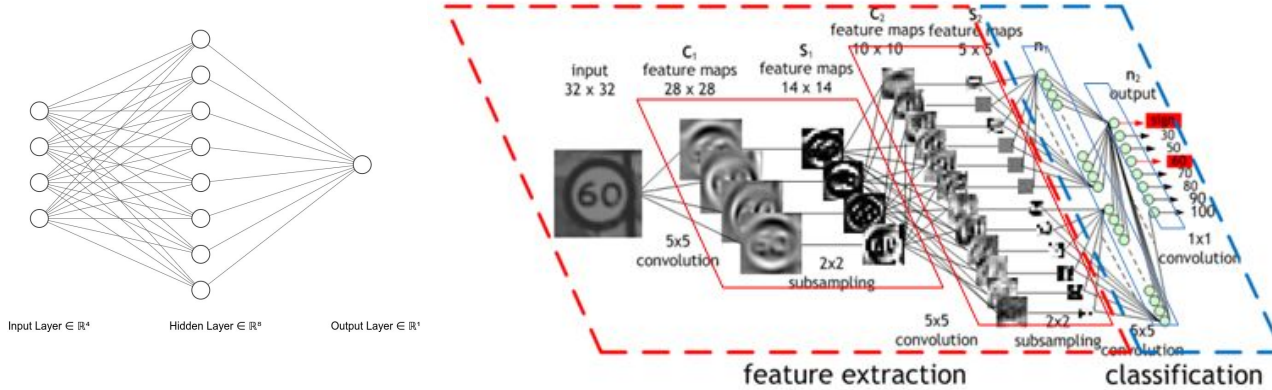


## A mostly complete chart of Neural Networks

©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org




# 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**  provides a structured summary of a model's details, performance, limitations, and ethical considerations. Proposed in the "Model Cards for Model Reporting" (2018) to improve transparency, reproducibility, and responsible AI usage. We want:

- Architecture;
- Weights (training details, how weights were obtained, data processing pipelines);
- Performance and benchmarks.

Real-world examples:

- LLaMA 3 [https://github.com/meta-llama/llama3/blob/main/MODEL\\_CARD.md](https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md)
- Gemini 1.5  
[https://storage.googleapis.com/deepmind-media/gemini/gemini\\_v1\\_5\\_report.pdf#page=105](https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf#page=105)
- DeepSeek R1 <https://huggingface.co/deepseek-ai/DeepSeek-R1>

# 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)**

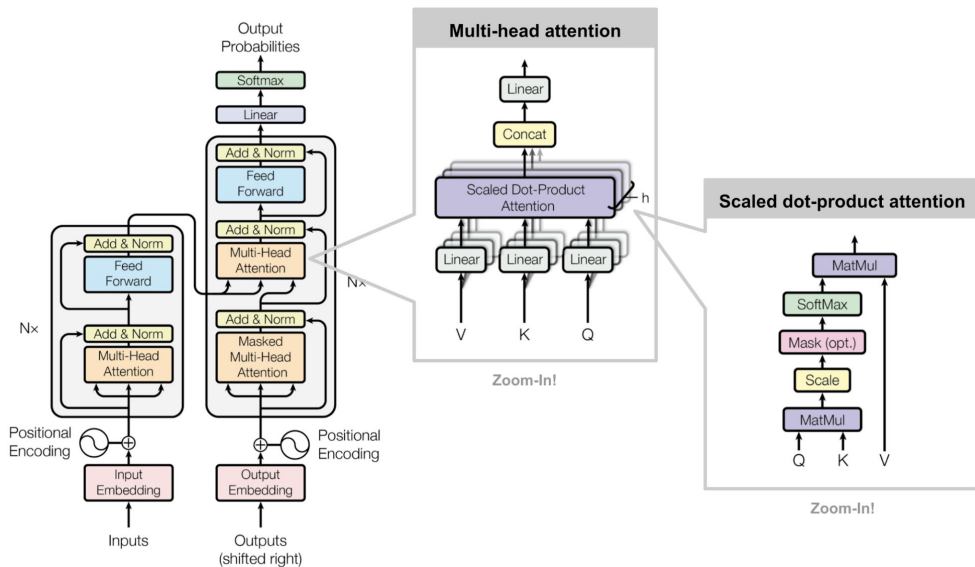
- Enables fine-tuning & resuming training;
- Prevents loss of work & supports reproducibility.

# 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>

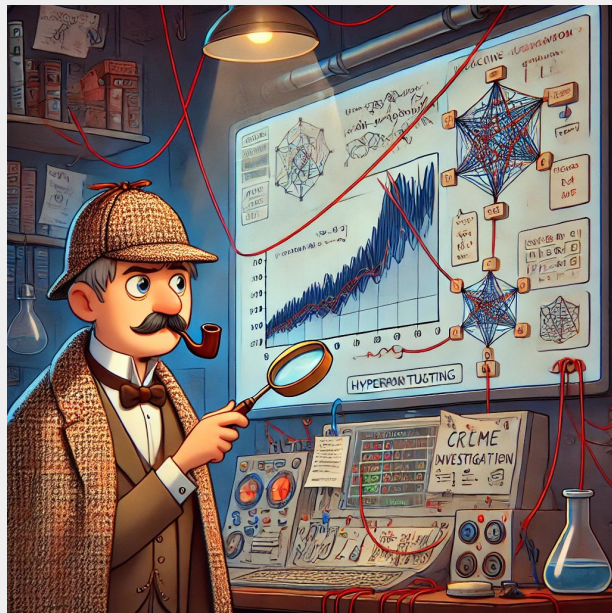
Transformer Model Family:

<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>

# Optimizer and Performance Metrics



# 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](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

But how to evaluate the derivatives? Automatically!

**Automatic Differentiation (AD)** automates the process of differentiation, allowing efficient computation of derivatives by tracking all computations with a computational graph. Knowing the derivatives to simple functions, the total derivative can be computed efficiently.

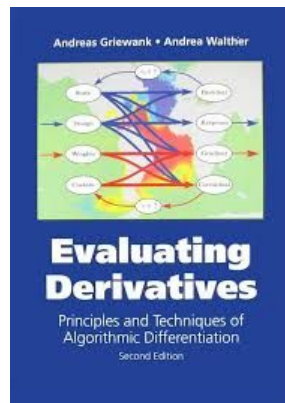
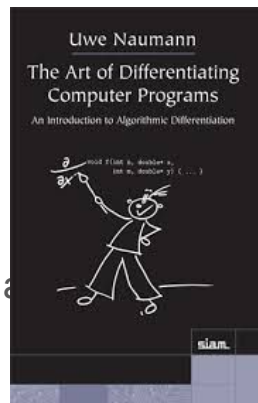
## Why to use the AD+ML framework?

- Calculate gradients and Hessians;
- All the tools for ML (layers, optimizers, loss functions);
- Support for hardware accelerators: GPU/TPU.



Our choice is JAX/Flax.

See: Section 13.3 of PML: <https://probml.github.io/pml-book/book1.1.html>



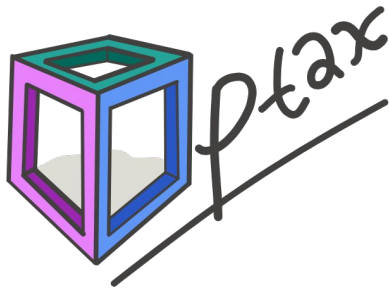
Josh Tobin  
@josh\_tobin\_

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 and Tracking



# 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:  $y_{\text{pred}} = \text{model}(x_{\text{batch}}, \text{params})$ 
```

```
    Compute loss:  $\text{loss} = \text{loss\_fn}(y_{\text{pred}}, y_{\text{batch}})$ 
```

```
    # Backward pass
```

```
    Compute gradients:  $\text{grads} = \partial L / \partial w$     # Backpropagation or AD
```

```
    Update parameters:  $\text{params} = \text{params} + f(\text{grads})$ 
```



# Experiment Tracking

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

## Logging: print, logging module.

## Progress bar: print, tqdm.



## 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()` + `tgdm` + `TensorBoard` + `W&B`.

# From Training Loop To Training Script

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

We transform the training loop with the module `argparse` (see on the right).

Then on the server we need to run a command:

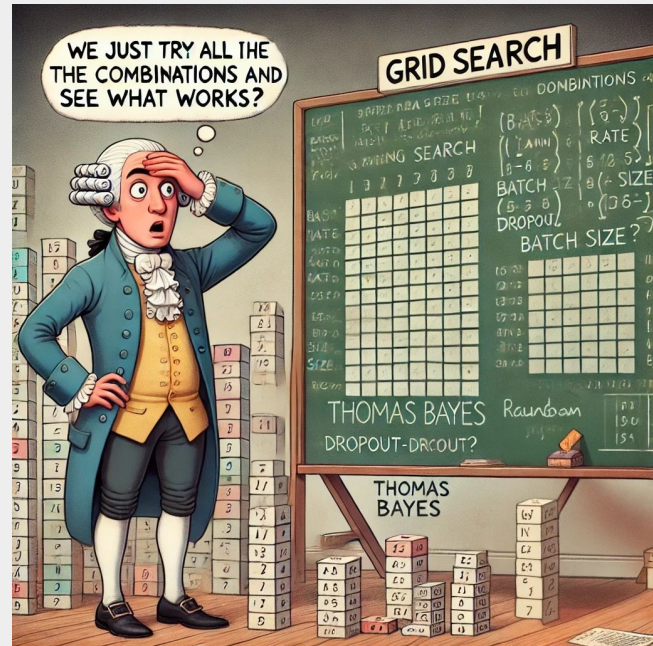
```
$ python3 train.py -dataset  
mnist -optimizer adam -lr 0.0003
```

```
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}")
```

# Hyperparameter Optimization



# 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, Numpyro 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)
```

Search Space

Surrogate Function

Thank you!