ML Foundations

Iliad Intensive

Julian Schulz

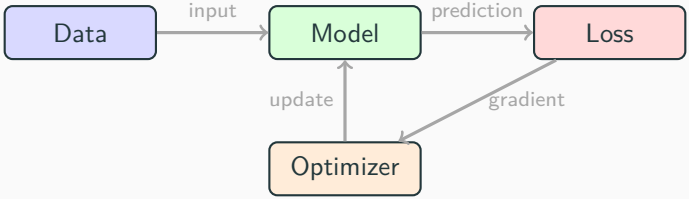
September 8, 2026

Outline

1. The Training Loop
2. Tensors and PyTorch
3. Loss Functions
4. Optimizers
5. Architectures
6. Hyperparameter Optimization
7. Transformer Architecture
8. Reinforcement Learning and RLHF

The Training Loop

Concepts in Machine Learning



Data input to a task we want an AI to do

Model gets data, outputs a prediction

Loss function quantifies how well the model did

Optimizer updates the model to do better

The Training Loop in PyTorch

```
for x, y in dataloader:
    optimizer.zero_grad()
    prediction = model(x)          # forward pass
    loss = loss_fn(prediction, y) # compute loss
    loss.backward()                # backward pass
    optimizer.step()               # update parameters
```

This is *the* loop. Everything else in ML is about making each piece better.

Terminology

Parameters / weights the numbers that represent the model itself, updated by the optimizer (θ)

Activations the numbers created as we feed data through the network

Hyperparameters choices made *before* training: learning rate, batch size, architecture

Stochasticity and Batching

We want to minimize the true loss over the full data distribution:

$$\mathcal{L}(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \left[\ell(f_\theta(x), y) \right]$$

But we only ever see a finite **batch** $\mathcal{B} \subset \mathcal{D}$ at each step:

$$\hat{\mathcal{L}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{(x,y) \in \mathcal{B}} \ell(f_\theta(x), y)$$

- Every gradient is a noisy estimate of the true gradient
- Batch size controls the noise level

Tensors and PyTorch

Tensors

The central object in PyTorch: multidimensional arrays of numbers.

We use tensors to represent: data, parameters, activations, outputs, losses.

```
x = torch.tensor([1.0, 2.0, 3.0])
M = torch.zeros(3, 4)
```

By convention, everything except parameters has the **batch dimension first**, so we can process many samples at once.

Tensor Operations

```
torch.exp(x)            # elementwise exponential
A @ B                   # matrix multiply
x.unsqueeze(0)          # add a dimension
```

Einstein notation makes index operations explicit:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
torch.einsum('ik,kj->ij', A, B)
```

Einops

Readable notation for tensor reshaping:

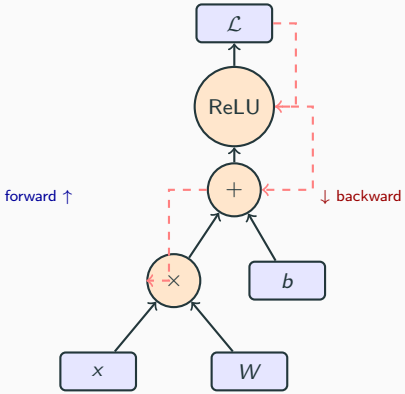
```
# PyTorch (what does this do?)
x.reshape(B, S, n_heads, d_head).permute(0, 2, 1, 3)

# Einops (self-documenting)
rearrange(x, 'b s (h d) -> b h s d', h=n_heads)
```

- `rearrange`: reshape, permute, split, merge
- `reduce`: pool over dimensions
- `repeat`: tile/broadcast

Autograd and the Computation Graph

1. Build expressions from tensors with `requires_grad=True`
2. Call `.backward()` on a scalar
3. Access gradients via `.grad`

You never write derivatives by hand. The graph is rebuilt every forward pass.



Devices

- Tensors live on a **device**: CPU or GPU (`cuda:0`)
- Move with `x.to('cuda')`
- All tensors in an operation must be on the same device
- GPU memory is the main bottleneck for model size and batch size

Exercise 1: PyTorch Basics

25 minutes

Loss Functions

Loss Functions: Supervised Learning

The loss function defines *what* we are optimizing.

When we can directly verify the output:

- Predict something continuous (position of a ball) $\Rightarrow$ MSE
- Predict something discrete (class, next word) $\Rightarrow$ **cross-entropy loss**

$$\mathcal{L} = -\sum_i y_i \log \hat{y}_i$$

Minimizing cross-entropy makes the model **well-calibrated**: its output numbers can be interpreted as probabilities.

Loss Functions: Human Preferences

We do not always have the "correct label." Sometimes we only know a pattern outputs should have.

Example: rating text quality, given only pairwise human preferences ("I prefer A over B").

The preferred text should be rated higher:

$$\mathcal{L} = -\log \sigma(r_\theta(x_w) - r_\theta(x_l))$$

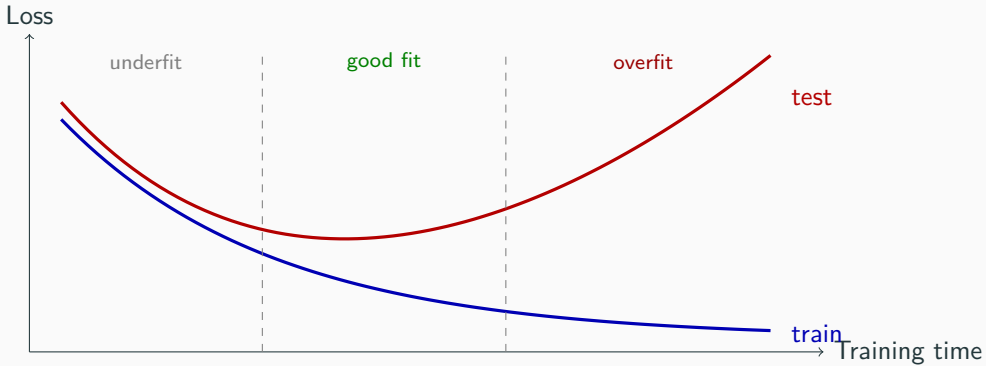
[CLB+17]

Loss Functions: Reinforcement Learning

Sometimes we cannot say what the correct output looks like, but we can identify one when we see it (e.g. winning at chess).

- RL loss functions are more complex
- Often involve a learned reward model (from the previous slide) combined with a policy objective

Train Loss vs. Test Loss



- **Underfitting**: both losses high (model too simple or undertrained)
- **Overfitting**: train loss low, test loss rises (model memorizes)
- Regularization (L2, dropout) constrains the model to delay overfitting

Optimizers

Optimizers

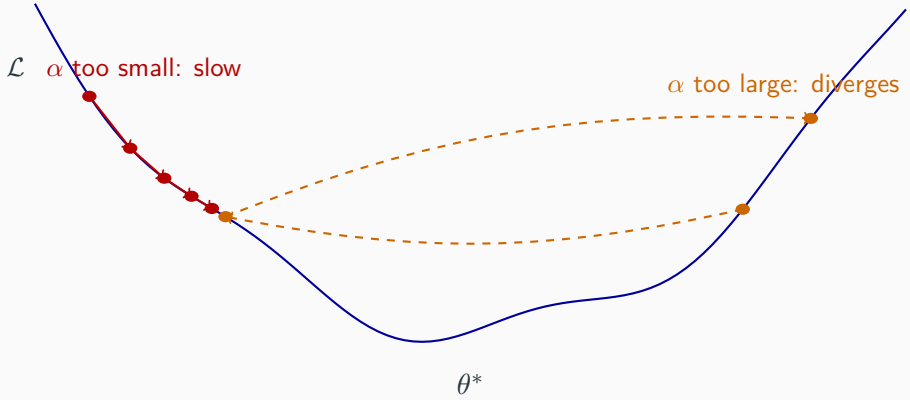
We have a gradient. Which direction do we step?

Stochastic Gradient Descent

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}$$

- α: learning rate (hyperparameter)
- Simple and robust
- Can be slow: oscillates across narrow valleys, crawls along them

Learning Rate



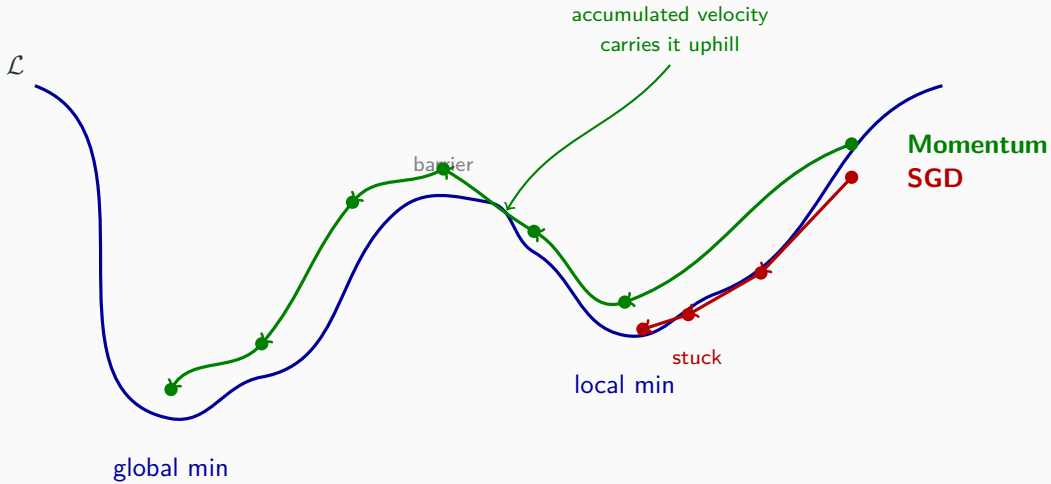
Momentum [SMDH13]

$$v_{t+1} = \beta\, v_t + (1 - \beta)\, \nabla_\theta \mathcal{L}$$

$$\theta_{t+1} = \theta_t - \alpha\, v_{t+1}$$

- Keeps a running velocity, like a ball rolling downhill
- Accelerates through flat regions
- Dampens oscillations in steep directions
- Can roll over small local minima

Momentum Escapes Local Minima



Adam = Momentum + RMSProp [KB14]

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t \qquad v_{t+1} = \beta_2 v_t + (1-\beta_2) g_t^2$$

$$\hat{m} = \frac{m_{t+1}}{1-\beta_1^{t+1}} \qquad \hat{v} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

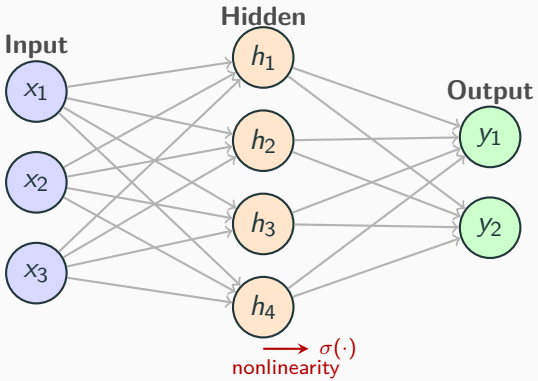
- Adaptive per-parameter learning rates + momentum
- Default choice for most deep learning
- Typical: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Exercise 2: Implement Optimizers

25 minutes

Architectures

The MLP



- Without a hidden layer, we can only represent linear functions
- The hidden layer with a nonlinearity lets us express richer functions

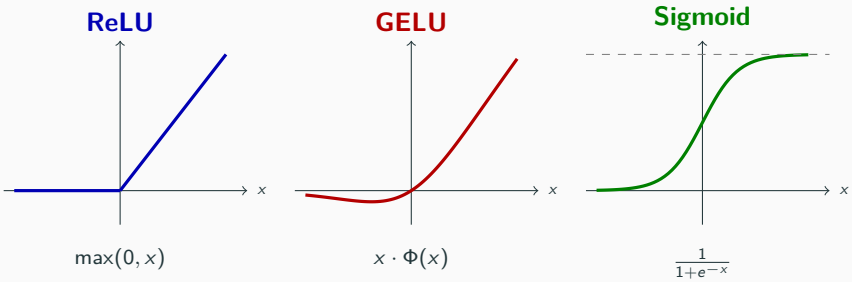
The MLP in PyTorch

```
import torch.nn as nn

mlp = nn.Sequential(
    nn.Linear(3, 128),    # input -> hidden (weights + bias)
    nn.ReLU(),            # nonlinearity
    nn.Linear(128, 2),    # hidden -> output
)

y = mlp(x)  # x: (batch, 3) -> y: (batch, 2)
```

- `nn.Linear`: $y = Wx + b$ (learnable W, b)
- `nn.Sequential`: chain layers into a pipeline

Activation Functions

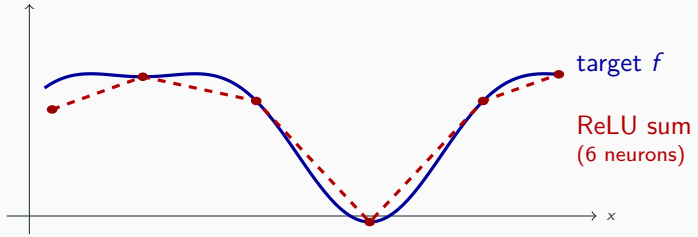
Without nonlinearities, stacking linear layers just gives another linear layer.



- **ReLU**: simple, effective, standard for MLPs
- **GELU**: smooth, used in transformers
- **Sigmoid**: saturates at extremes, mostly historical

Universal Approximation

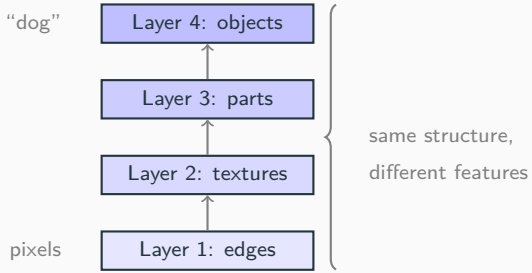
Theorem (Cybenko, Hornik et al.): A single hidden layer can approximate any continuous function on a compact set.



- Each ReLU neuron contributes a "hinge"; their weighted sum approximates f
- Existence result, not a recipe: says nothing about how many neurons you need
- In practice, deeper is much more efficient than wider

[Cyb89, HSW89]

Depth: Stacking Layers



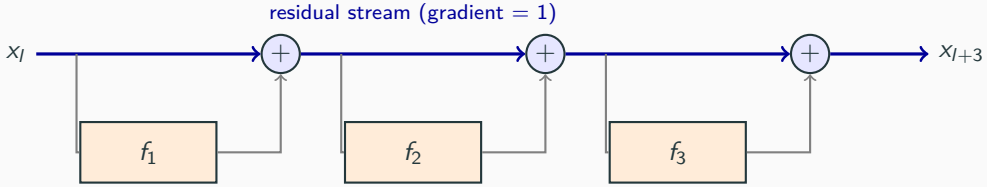
- Each layer builds on the previous one's representations
- The deep learning recipe: repeat a simple block many times

Over/underparameterization:

- Too few parameters $\Rightarrow$ cannot represent the function
- Too many $\Rightarrow$ can memorize, but modern networks often still generalize

Residual Connections

Deep networks suffer from **vanishing gradients**: gradients shrink to zero through many layers.



$$x_{l+1} = x_l + f(x_l)$$

- Gradient flows unimpeded through the skip path (identity)
- Each block only needs to learn the *residual* correction $f(x_l)$

[HZRS15]

Residual Connections in PyTorch

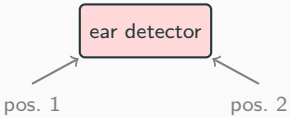
```python
class ResidualBlock(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, d_model),
        )

    def forward(self, x):
        return x + self.layers(x)  # skip connection!
```

- The `x +` is the entire trick

Architecture Encodes Symmetry

The right architecture exploits structure in the data.

CNN: one detector

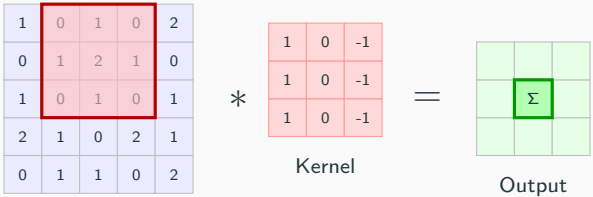


MLP: one detector per position

. . . (one for every pixel)

An architecture that respects a symmetry needs **fewer parameters** and **less data**.

Convolutional Neural Networks



- **Same kernel** slides across every position (weight sharing)
- Same features detected regardless of location $\Rightarrow$ translation invariance
- Hierarchy: edges $\rightarrow$ textures $\rightarrow$ parts $\rightarrow$ objects

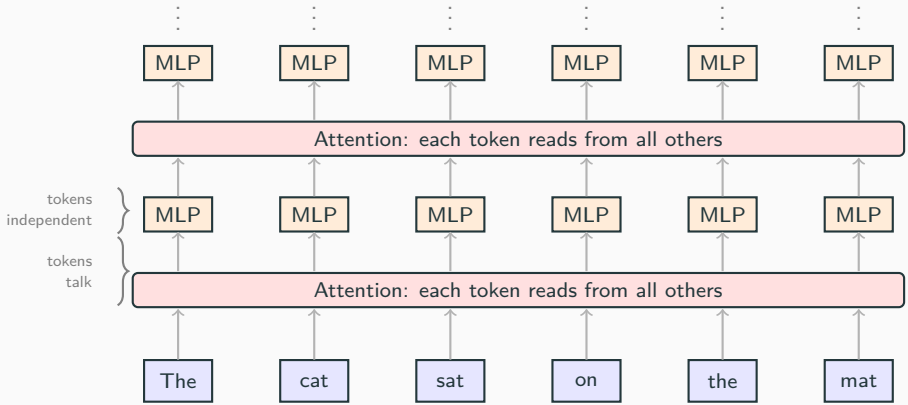
[LBBH98]

CNNs in PyTorch

```
cnn = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=3, padding=1),  # 1 channel -> 16
    nn.ReLU(),
    nn.MaxPool2d(2),                    # downsample 2x
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Flatten(),                       # spatial -> vector
    nn.Linear(32 * 7 * 7, 10),          # classify into 10 classes
)
```

- `Conv2d`: the sliding kernel (learnable weights)
- `MaxPool2d`: downsample by taking the max in each region

Transformers (Preview)



- **Attention**: tokens mix information (the only place they interact)
- **MLP**: each token processed independently (same weights, different data)
- We will go much deeper in the afternoon session

Exercise 3: Build Simple Architectures

15 minutes

Hyperparameter Optimization

Hyperparameters

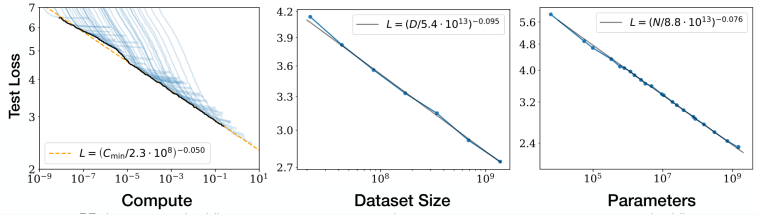
Any property not set by gradient descent is set by the experimenter:

- Model architecture (depth, width, . . .)
- Batch size, number of epochs
- Learning rate, momentum, and other optimizer settings

Finding Good Hyperparameters

- Some we have figured out once and reuse (e.g. $\beta_1 = 0.9$)
- Some we sweep over: run many times, pick the best (typically learning rate)
- Some follow **scaling laws**: cheap small experiments predict large-scale performance

Scaling Laws



Performance follows smooth power laws across > 6 orders of magnitude in compute, data, and parameters. Cheap small experiments predict large-scale performance.

[KMH+20, HBM+22]

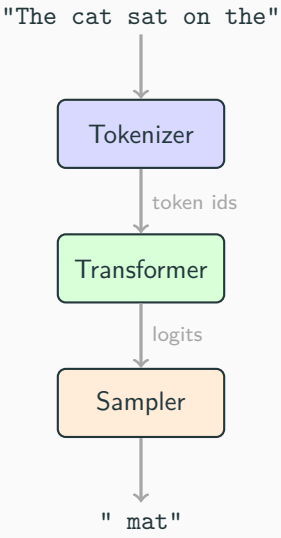
Exercise 4: Hyperparameter Tuning

Until lunch

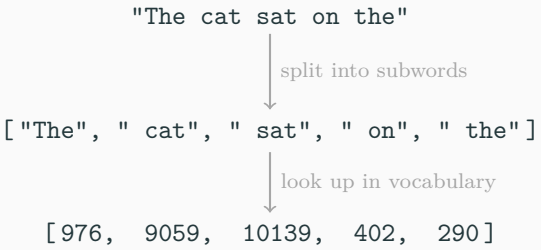
`https://playground.tensorflow.org/`

Transformer Architecture

A Top-Down View



Tokenizer


"The cat sat on the"
split into subwords
["The", " cat", " sat", " on", " the"]
look up in vocabulary
[976, 9059, 10139, 402, 290]

Building the Tokenizer: Byte Pair Encoding

Start with the vocabulary as all single characters, then **repeat**:

1. Find the most frequent adjacent pair.
2. Merge it into a new symbol; add it to the vocabulary.

Stop at the target vocabulary size.

Sampler

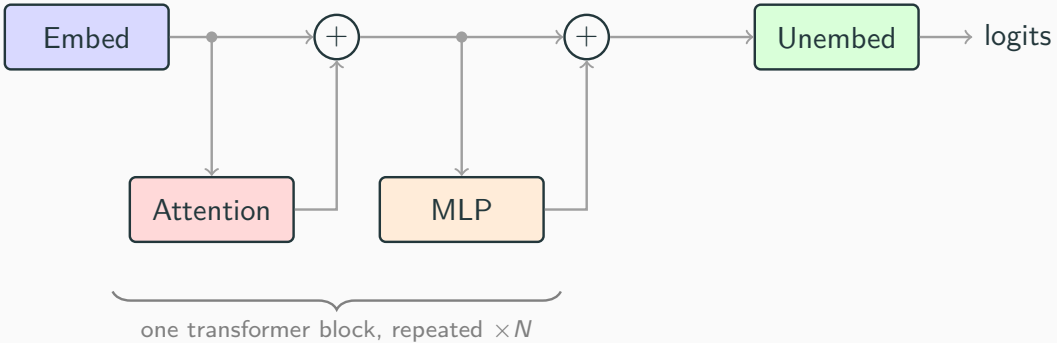
The transformer outputs one **logit** per vocabulary token: $z \in \mathbb{R}^{|V|}$.

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

Temperature T: as $T \to 0$ this recovers greedy ($\arg\max$); $T < 1$ sharpens the distribution, $T > 1$ flattens it.

- **Top-k**: keep the k highest-probability tokens, renormalize, then sample.
- **Top-p** (nucleus): keep the smallest set of tokens whose probabilities sum to at least p, renormalize, then sample.

Inside the Transformer



- **Attention head**: moves information between token positions.
- **MLP**: stores / computes information.

Embedding

A lookup table $W_E \in \mathbb{R}^{|V| \times d}$: each token id selects one row, a learned vector carrying its meaning.

$$\text{token id } t \longmapsto x_t = (W_E)_t \in \mathbb{R}^d$$

The vectors are **learned**, so directions in this space come to encode semantic relationships.

Positional Encoding

Learned (absolute)

A second lookup table, one learned vector per position:

$$P \in \mathbb{R}^{L \times d}, \qquad \text{pos} \mapsto P_{\text{pos}}$$

$$x_{\text{pos}} = \text{embed}(\text{token}) + P_{\text{pos}}$$

The obvious default; capped at a maximum length L, and position is **absolute**.

Rotary (RoPE)

Rotate each 2D pair of query/key dims by an angle $\propto$ position:

$$\tilde{q}_m = R(m\theta)\, q, \qquad \tilde{k}_n = R(n\theta)\, k$$

$$R(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

$$\langle \tilde{q}_m, \tilde{k}_n \rangle = g(q, k,\, m - n)$$

Score depends only on the **relative** offset $m - n$; applied every layer.

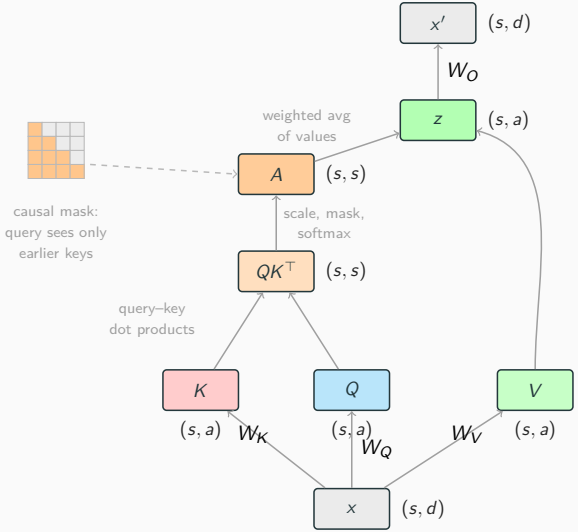
Unembedding

Not the inverse of embedding.

$$z = W_U x, \qquad W_U \in \mathbb{R}^{|V| \times d}$$

- Reads the final residual stream and projects it back to vocabulary size.
- Produces one logit per token, ready for the sampler.
- Uses its own learned weights W_U, separate from W_E.

Anatomy of an Attention Head



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{a}}\right)V$$

s: seq length, d: model dim, a: head dim

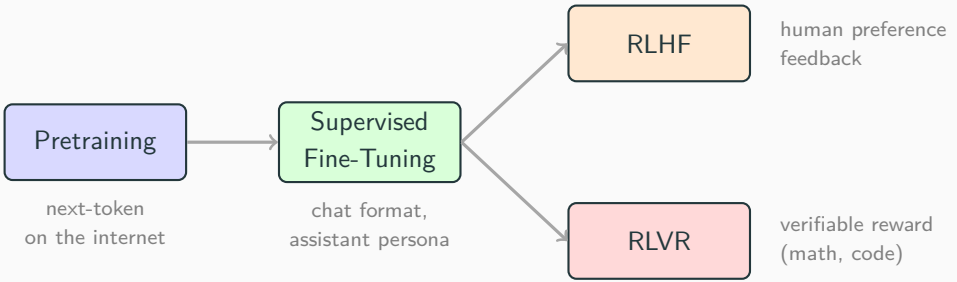
Exercises

MLP

Attention

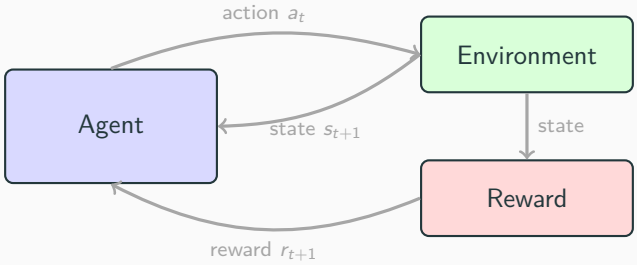
Reinforcement Learning and RLHF

The Model Lifecycle



- **Pretraining**: next-token prediction, negative log-likelihood, on trillions of tokens. Most of the compute.
- **SFT**: a little high-quality <human> / <assistant> data instills the chat format.
- **RLHF**: align to human preference judgments.
- **RLVR**: reinforcement learning on verifiable rewards (math, code).

What is Reinforcement Learning?



- The agent **chooses actions**; the environment returns a new state.
- A **reward** process assigns a score to the state.
- The agent's goal is to **maximize long-term reward**.

The Markov Decision Process

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$$

- $\mathcal{S}$: states
- $\mathcal{A}$: actions
- $P(s' \mid s, a)$: transition probabilities
- $R(s, a)$: reward
- $\gamma \in [0, 1)$: discount factor

Markov property: the future depends only on the present state, not the full history.

Return and Discounting

The agent maximizes the **expected return**: total reward extending into the future.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Small γ: myopic, values immediate reward.
- Large γ: farsighted, values future reward almost as much as present.

Policies

A **policy** $\pi(a \mid s)$ gives the probability of each action in a state.

Simple: a lookup table.

	a_1	a_2
s_1	0.5	0.5
s_2	0.9	0.1
s_3	0.2	0.8

Deep: a network's weights.

$$\pi_\theta(a \mid s) = \mathrm{softmax}(f_\theta(s))$$

For an LLM, the state s is the context and the action is the next token.

Value Functions

How good is a state, or a state-action pair, under policy π?

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \qquad \text{(state value)}$$

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s,\, A_t = a] \qquad \text{(action value)}$$

Value functions estimate expected return, which is what most RL algorithms learn to predict.

The Core Problem

We want to maximize the expected return:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

But no gradient can flow from the reward back to the model.

The reward depends on the model's output, but the path is not differentiable:

- the environment's transition dynamics,
- the reward-generating process,
- the discrete action sampler.

Solution 1: Q-Learning

Learn to predict total reward, then act greedily.

$$Q(s,a) \leftarrow \underbrace{r}_{\text{this step}} + \underbrace{\gamma \max_{a'} Q(s',a')}_{\text{best future}}$$

- Trained by **supervised consistency**: the temporal-difference error.
- Then pick the action with the highest predicted reward.

Solution 2: Policy Gradients

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \int p_\theta(\tau)\, R(\tau)\, d\tau$$

The parameter θ sits inside the distribution we sample from, not inside an expression we can evaluate and differentiate.

Optimize the policy network *directly* to maximize expected reward.

- We cannot differentiate the environment or the reward process.
- The **policy gradient theorem** estimates the gradient from samples alone.
- Idea: push up the log-probability of actions that led to high return.

Policy Gradient: The Objective

Expected return over trajectories τ:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \int p_\theta(\tau)\, R(\tau)\, d\tau$$

Trajectory distribution:

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t)\, p(s_{t+1} \mid s_t, a_t)$$

The middle term, the environment dynamics, is unknown and non-differentiable.

Policy Gradient: The Log-Derivative Trick

Identity:

$$\nabla_\theta \, p_\theta(\tau) = p_\theta(\tau) \, \nabla_\theta \log p_\theta(\tau)$$

Applied to the objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log p_\theta(\tau) \cdot R(\tau)]$$

The gradient of an integral becomes an expectation of a gradient: estimable by sampling.

Policy Gradient: The Environment Drops Out

Log of the trajectory distribution:

$$\log p_\theta(\tau) = \log p(s_0) + \sum_t \log \pi_\theta(a_t \mid s_t) + \sum_t \log p(s_{t+1} \mid s_t, a_t)$$

Only the policy depends on θ:

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

The environment terms vanish, since they do not depend on θ.

Policy Gradient Theorem

Final result:

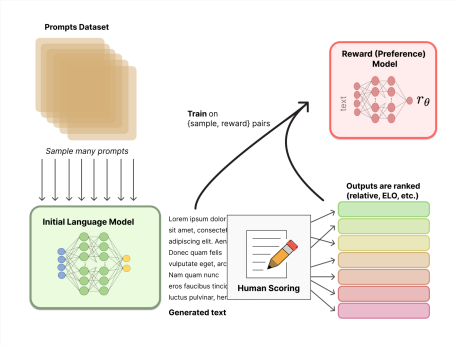
$$\boxed{\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot R(\tau)\right]}$$

Per-step form with advantage $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot A^\pi(s_t, a_t)\right]$$

Same gradient in expectation, far lower variance: weight each action by how much it beat the average for its state, not by the whole episode's return.

Reward Model from Human Preferences

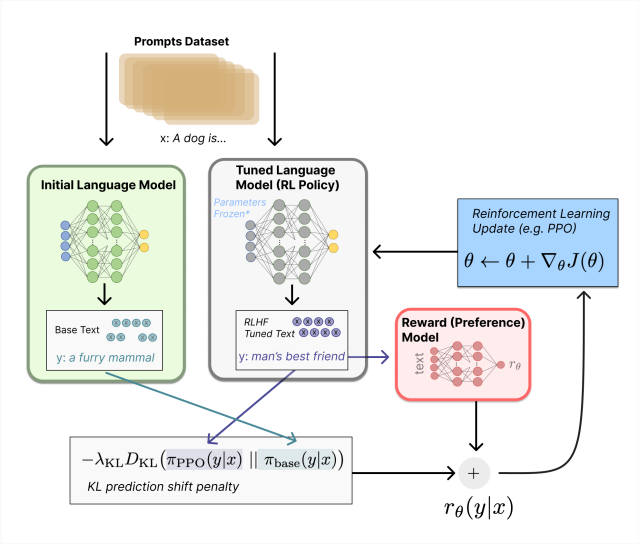


Humans rank model outputs; a **reward model** r_θ learns to predict the preferred one:

$$\mathcal{L} = -\log \sigma\big(r_\theta(x_w) - r_\theta(x_l)\big)$$

From [LCvH22]

RLHF: Reinforcement Learning from Human Feedback



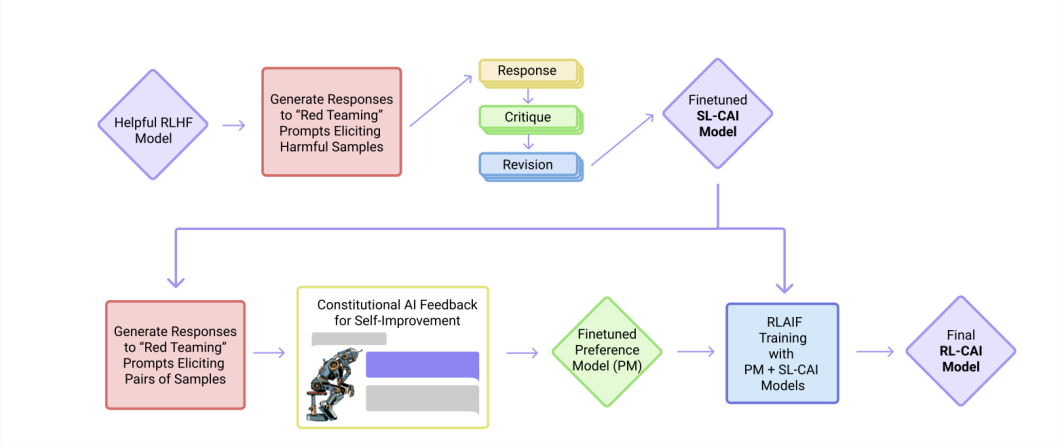
Proximal Policy Optimization

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} \widehat{A}^{\pi_{\theta_k}}(s, a),\ \ \mathrm{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right) \widehat{A}^{\pi_{\theta_k}}(s, a)\right)$$

$$\widehat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1} + \gamma^{T-t} V(s_T)$$

Constitutional AI

Replace much of the human feedback with feedback from the model itself, guided by a written **constitution**: the model **critiques** and **revises** its own responses (RLAIF).



Exercise 6: RLHF

Fine-tune a policy from a reward signal

References i


Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan.
Constitutional AI: Harmlessness from AI Feedback, December 2022.

URL: https://arxiv.org/abs/2212.08073, arXiv:2212.08073.

Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei.
Deep reinforcement learning from human preferences, June 2017.
URL: https://arxiv.org/abs/1706.03741, arXiv:1706.03741.

George Cybenko.
Approximation by Superpositions of a Sigmoidal Function.
1989.
URL: https://doi.org/10.1007/BF02551274.


Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre.
Training Compute-Optimal Large Language Models, March 2022.
URL: `https://arxiv.org/abs/2203.15556`, `arXiv:2203.15556`.


Dan Hendrycks and Kevin Gimpel.
Gaussian Error Linear Units (GELUs), June 2016.
URL: `https://arxiv.org/abs/1606.08415`, `arXiv:1606.08415`.

References iv

Kurt Hornik, Maxwell Stinchcombe, and Halbert White.
Multilayer Feedforward Networks are Universal Approximators.
1989.
URL: https://doi.org/10.1016/0893-6080(89)90020-8.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun.
Deep Residual Learning for Image Recognition, December 2015.
URL: https://arxiv.org/abs/1512.03385, arXiv:1512.03385.

Diederik P. Kingma and Jimmy Ba.
Adam: A Method for Stochastic Optimization, December 2014.
URL: https://arxiv.org/abs/1412.6980, arXiv:1412.6980.

References v

- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei.
  Scaling Laws for Neural Language Models, January 2020.
  URL: https://arxiv.org/abs/2001.08361, arXiv:2001.08361.
- Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner.
  Gradient-Based Learning Applied to Document Recognition.
  1998.
  URL: https://doi.org/10.1109/5.726791.
- Nathan Lambert, Louis Castricato, Leandro von Werra, and Alex Havrilla.
  Illustrating Reinforcement Learning from Human Feedback (RLHF), December 2022.
  URL: https://huggingface.co/blog/rlhf.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 2014.

URL: `https://jmlr.org/papers/v15/srivastava14a.html`.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton.

On the importance of initialization and momentum in deep learning. ICML, 2013.

URL: `https://proceedings.mlr.press/v28/sutskever13.html`.


Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin.
Attention Is All You Need, June 2017.
URL: https://arxiv.org/abs/1706.03762, arXiv:1706.03762, doi:10.48550/arXiv.1706.03762.