ML Foundations

Iliad Intensive

Julian Schulz

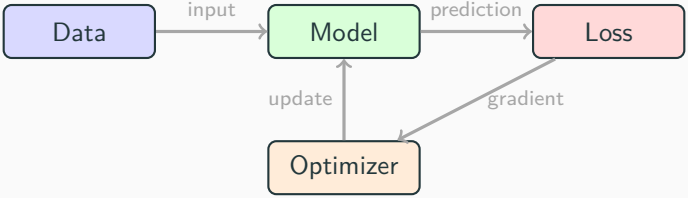
April 16, 2026

Outline

1. The Training Loop
2. Tensors and PyTorch
3. Loss Functions
4. Optimizers
5. Architectures
6. Hyperparameter Optimization

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:

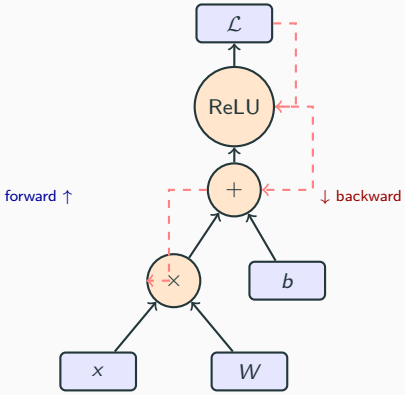
```
# PyTorch (what does this do?)
x.reshape(B, S, n_heads, d_head).permute(0, 2, 1, 3)

# 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:

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

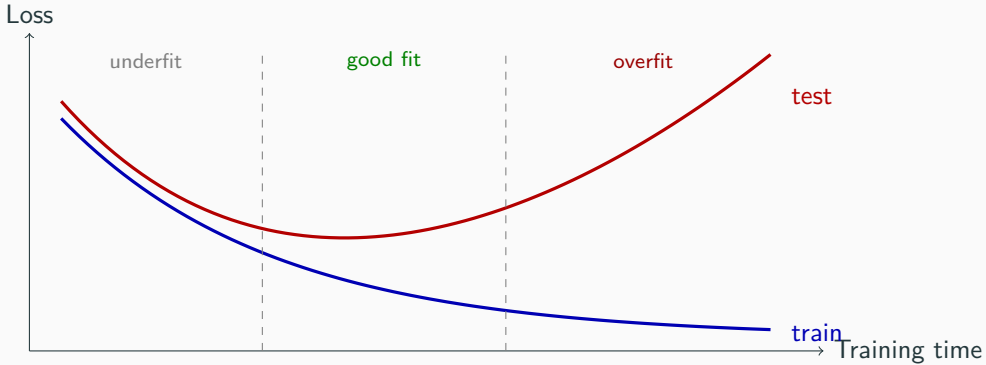
[Christiano et al. 2017]

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

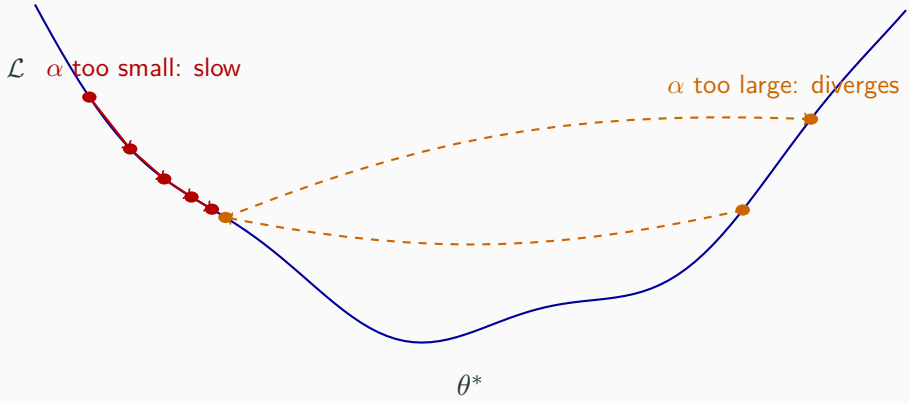
We have a gradient. Which direction do we step?

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



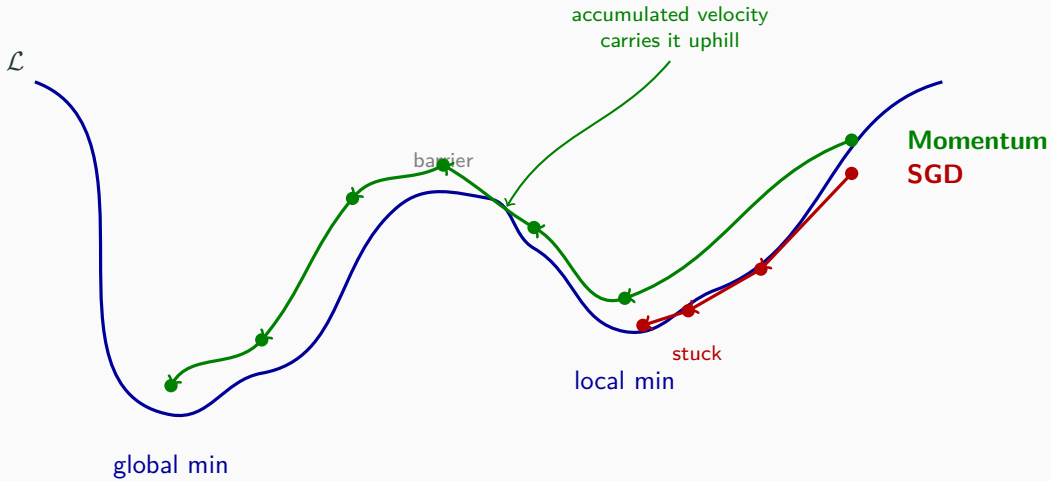
Momentum [Sutskever et al. 2013]

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

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

- 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 [Kingma et al. 2014]

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

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

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

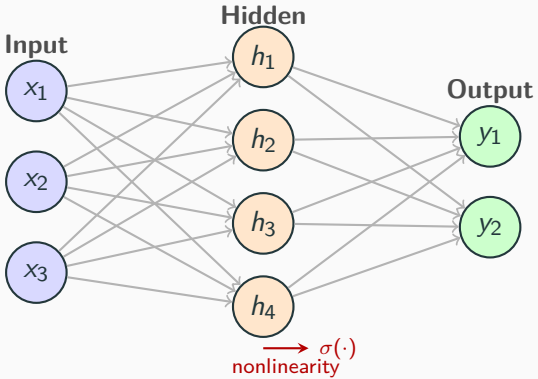
- Adaptive per-parameter learning rates + momentum
- Default choice for most deep learning
- Typical: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

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

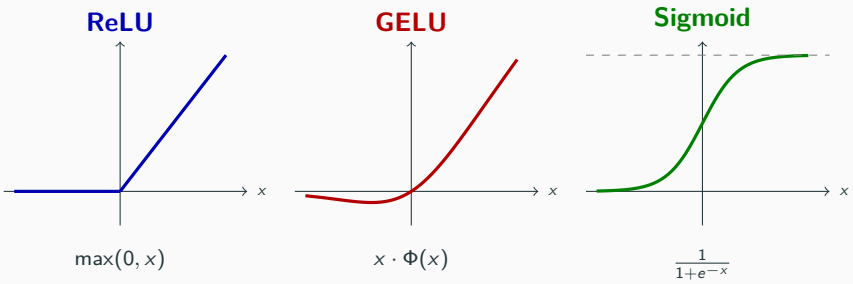
The MLP in PyTorch

```
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

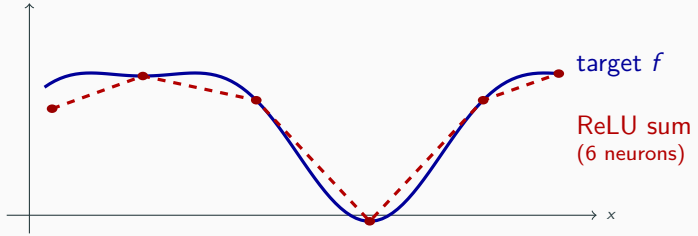
Without nonlinearities, stacking linear layers just gives another linear layer.



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

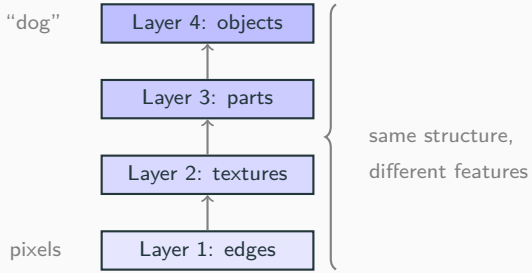
Universal Approximation

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

Depth: Stacking Layers



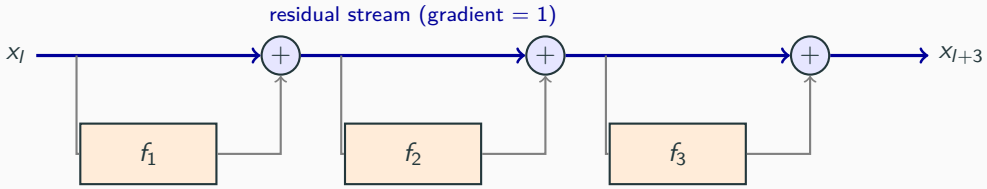
- Each layer builds on the previous one's representations
- The deep learning recipe: repeat a simple block many times

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 $f(x_l)$

Residual Connections in PyTorch

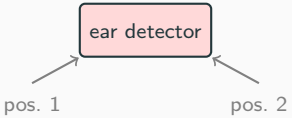
```python
class ResidualBlock(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, d_model),
        )

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

- The `x +` is the entire trick
- Stack many of these: `nn.Sequential(*[ResidualBlock(d) for _ in range(n)])`

Architecture Encodes Symmetry

The right architecture exploits structure in the data.

CNN: one detector

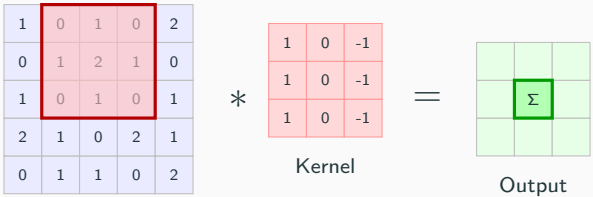


MLP: one detector per position

. . . (one for every pixel)

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

Convolutional Neural Networks



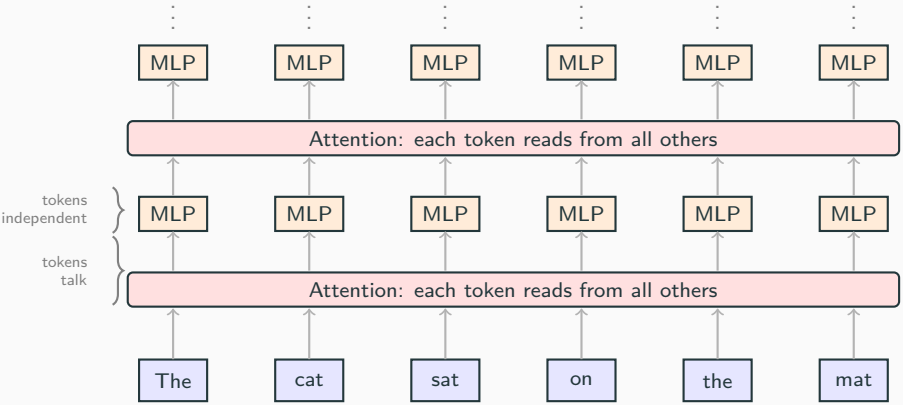
- **Same kernel** slides across every position (weight sharing)
- Same features detected regardless of location $\Rightarrow$ translation invariance
- Hierarchy: edges $\rightarrow$ textures $\rightarrow$ parts $\rightarrow$ objects

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
- Parameters: only the kernel values, shared across all positions

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

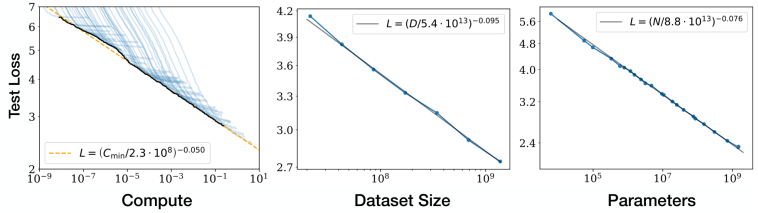
Any property not set by gradient descent is set by the experimenter:

- 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 [Kaplan et al. 2020]



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

Exercise 4: Hyperparameter Tuning

Until lunch

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

References i

- Christiano, Paul, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei [June 2017]. *Deep Reinforcement Learning from Human Preferences*. URL: https://arxiv.org/abs/1706.03741.
- Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei [Jan. 2020]. *Scaling Laws for Neural Language Models*. URL: https://arxiv.org/abs/2001.08361.
- Kingma, Diederik P. and Jimmy Ba [Dec. 2014]. *Adam: A Method for Stochastic Optimization*. URL: https://arxiv.org/abs/1412.6980.
- Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton [2013]. "On the Importance of Initialization and Momentum in Deep Learning". In: ICML. URL: https://proceedings.mlr.press/v28/sutskever13.html.