

CS498: Algorithmic Engineering

Lecture 22: Neural Networks & PyTorch

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 12

Outline

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs

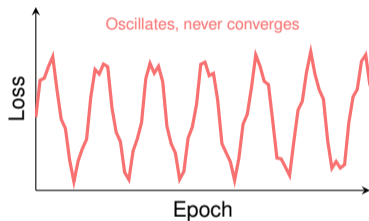
Why Not a Constant Learning Rate?

What happens if we just pick one learning rate and keep it forever?

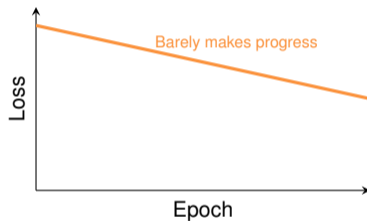
Why Not a Constant Learning Rate?

What happens if we just pick one learning rate and keep it forever?

η too high



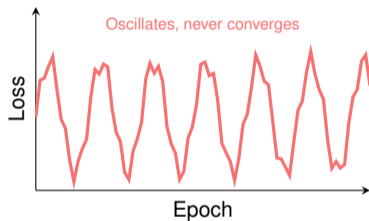
η too low



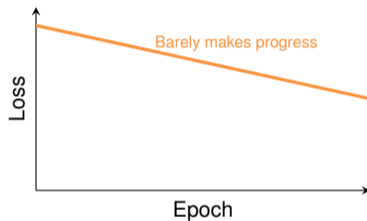
Why Not a Constant Learning Rate?

What happens if we just pick one learning rate and keep it forever?

η too high



η too low

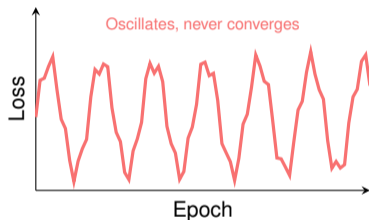


Too high: the optimizer overshoots and oscillates. Too low: the optimizer crawls and never reaches a good solution.

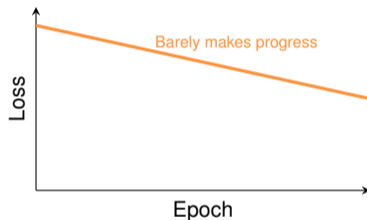
Why Not a Constant Learning Rate?

What happens if we just pick one learning rate and keep it forever?

η too high



η too low



Too high: the optimizer overshoots and oscillates. Too low: the optimizer crawls and never reaches a good solution.

Key insight: we want a learning rate that *changes* over training.

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

If we take big steps with bad gradients, the model can land in a bad region early and never recover.

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

If we take big steps with bad gradients, the model can land in a bad region early and never recover.

Linear warmup: start with $\eta \approx 0$ and linearly increase to η_{\max} over T_w steps.

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

If we take big steps with bad gradients, the model can land in a bad region early and never recover.

Linear warmup: start with $\eta \approx 0$ and linearly increase to η_{\max} over T_w steps.

After warmup, we want η to **decrease**. Early in training, big steps explore; late in training, small steps fine-tune.

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

If we take big steps with bad gradients, the model can land in a bad region early and never recover.

Linear warmup: start with $\eta \approx 0$ and linearly increase to η_{\max} over T_w steps.

After warmup, we want η to **decrease**. Early in training, big steps explore; late in training, small steps fine-tune.

Cosine decay formula:

$$\eta_t = \frac{1}{2} \eta_{\max} \left(1 + \cos \left(\frac{\pi (t - T_w)}{T - T_w} \right) \right)$$

Linear Warmup and Cosine Decay

At the start of training, weights are **random** and gradients are noisy. Adam's moment estimates need several steps to calibrate.

If we take big steps with bad gradients, the model can land in a bad region early and never recover.

Linear warmup: start with $\eta \approx 0$ and linearly increase to η_{\max} over T_w steps.

After warmup, we want η to **decrease**. Early in training, big steps explore; late in training, small steps fine-tune.

Cosine decay formula:

$$\eta_t = \frac{1}{2} \eta_{\max} \left(1 + \cos \left(\frac{\pi (t - T_w)}{T - T_w} \right) \right)$$

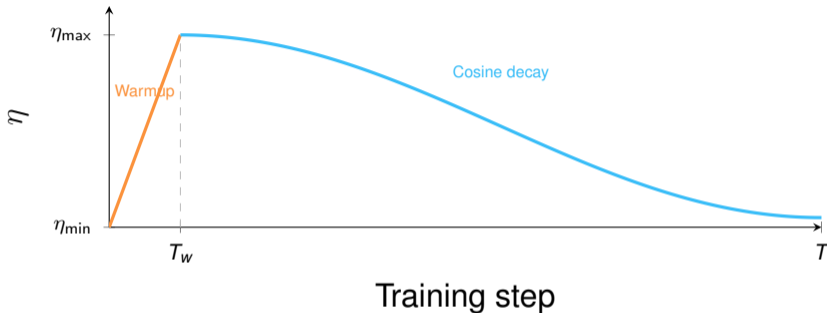
Starts at η_{\max} , smoothly decays to 0. The cosine shape gives a gradual transition.

What Does the Full Schedule Look Like?

Combine warmup + cosine decay into a single schedule:

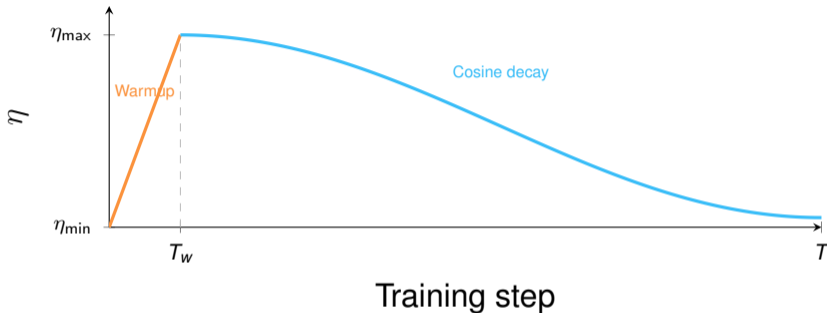
What Does the Full Schedule Look Like?

Combine warmup + cosine decay into a single schedule:



What Does the Full Schedule Look Like?

Combine warmup + cosine decay into a single schedule:

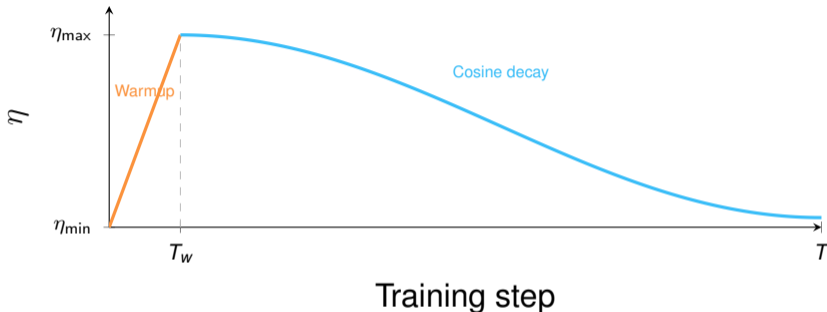


Typical values:

- Warmup: 1%–5% of total training steps.

What Does the Full Schedule Look Like?

Combine warmup + cosine decay into a single schedule:

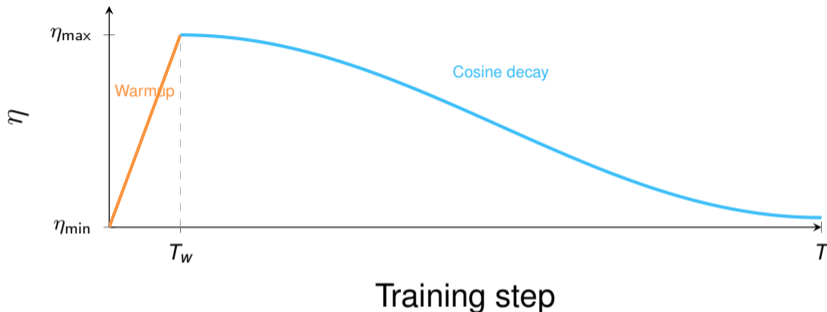


Typical values:

- Warmup: 1%–5% of total training steps.
- η_{\max} depends on model size: 10^{-3} for small models, 10^{-4} for large.

What Does the Full Schedule Look Like?

Combine warmup + cosine decay into a single schedule:



Typical values:

- Warmup: 1%–5% of total training steps.
- η_{\max} depends on model size: 10^{-3} for small models, 10^{-4} for large.
- This schedule is used in most modern training run (GPT, LLaMA, ...)

A Puzzle About Model Size

In Lecture 21, the bias-variance tradeoff told us:

A Puzzle About Model Size

In Lecture 21, the bias-variance tradeoff told us:

- More complexity \rightarrow lower bias, higher variance.

A Puzzle About Model Size

In Lecture 21, the bias-variance tradeoff told us:

- More complexity \rightarrow lower bias, higher variance.
- There is a sweet spot. Past it, test error goes up.

A Puzzle About Model Size

In Lecture 21, the bias-variance tradeoff told us:

- More complexity \rightarrow lower bias, higher variance.
- There is a sweet spot. Past it, test error goes up.

But: modern neural networks have *billions* of parameters, far more than training data points, and they work spectacularly well.

A Puzzle About Model Size

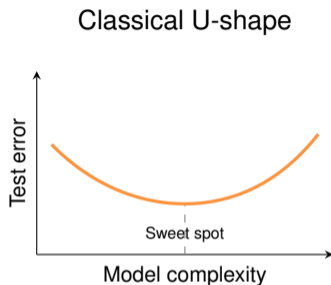
In Lecture 21, the bias-variance tradeoff told us:

- More complexity \rightarrow lower bias, higher variance.
- There is a sweet spot. Past it, test error goes up.

But: modern neural networks have *billions* of parameters, far more than training data points, and they work spectacularly well.

How is that possible? Shouldn't they overfit catastrophically?

Classical View vs. What Actually Happens

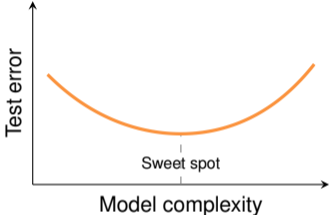


Test error goes up, then **spikes** at the interpolation threshold, then comes back **down**. This is **double descent**.

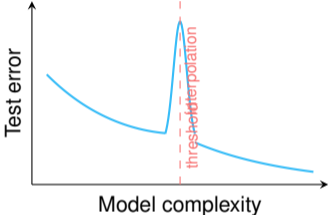
The interpolation threshold is the smallest model that can perfectly fit the training data.

Classical View vs. What Actually Happens

Classical U-shape

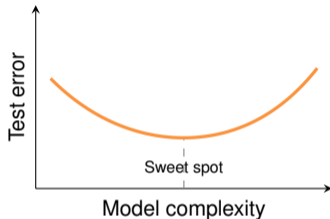


Double descent (reality)

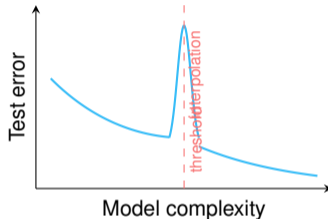


Classical View vs. What Actually Happens

Classical U-shape



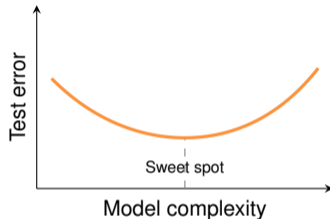
Double descent (reality)



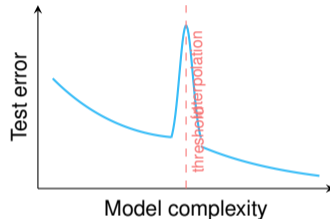
Test error goes up, then **spikes** at the interpolation threshold, then comes back **down**. This is **double descent**.

Classical View vs. What Actually Happens

Classical U-shape



Double descent (reality)



Test error goes up, then **spikes** at the interpolation threshold, then comes back **down**. This is **double descent**.

The interpolation threshold is the smallest model that can perfectly fit the training data.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Think of fitting a degree- $(n-1)$ polynomial through n noisy points: perfect interpolation but extreme oscillation.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Think of fitting a degree- $(n-1)$ polynomial through n noisy points: perfect interpolation but extreme oscillation.

In words: just enough capacity to memorize = worst generalization.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Think of fitting a degree- $(n-1)$ polynomial through n noisy points: perfect interpolation but extreme oscillation.

In words: just enough capacity to memorize = worst generalization.

Beyond the threshold, the model is **overparameterized**: many parameter settings fit the data perfectly. SGD's **implicit regularization** (from Lecture 21) selects flat, smooth solutions.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Think of fitting a degree- $(n-1)$ polynomial through n noisy points: perfect interpolation but extreme oscillation.

In words: just enough capacity to memorize = worst generalization.

Beyond the threshold, the model is **overparameterized**: many parameter settings fit the data perfectly. SGD's **implicit regularization** (from Lecture 21) selects flat, smooth solutions.

More parameters \rightarrow smoother interpolation \rightarrow better generalization.

Why Does the Spike Happen, Then Resolve?

At the interpolation threshold, the model has *just barely* enough capacity to fit every training point. It is forced to pass through every point exactly, including noisy ones, producing wild oscillations.

Think of fitting a degree- $(n-1)$ polynomial through n noisy points: perfect interpolation but extreme oscillation.

In words: just enough capacity to memorize = worst generalization.

Beyond the threshold, the model is **overparameterized**: many parameter settings fit the data perfectly. SGD's **implicit regularization** (from Lecture 21) selects flat, smooth solutions.

More parameters \rightarrow smoother interpolation \rightarrow better generalization.

Takeaway: overparameterization + SGD's implicit bias combine to produce well-generalizing solutions.

What Does Double Descent Mean in Practice?

The classical rule “more parameters = more overfitting” is **wrong** in the overparameterized regime. This is why billion-parameter models generalize well.

What Does Double Descent Mean in Practice?

The classical rule “more parameters = more overfitting” is **wrong** in the overparameterized regime. This is why billion-parameter models generalize well. Double descent was empirically demonstrated by Belkin et al. (2019).

What Does Double Descent Mean in Practice?

The classical rule “more parameters = more overfitting” is **wrong** in the overparameterized regime. This is why billion-parameter models generalize well. Double descent was empirically demonstrated by Belkin et al. (2019).

Practical implications:

- If your model is near the interpolation threshold, make it *bigger*, not smaller.

What Does Double Descent Mean in Practice?

The classical rule “more parameters = more overfitting” is **wrong** in the overparameterized regime. This is why billion-parameter models generalize well. Double descent was empirically demonstrated by Belkin et al. (2019).

Practical implications:

- If your model is near the interpolation threshold, make it *bigger*, not smaller.
- Modern practice: use very large models and rely on implicit regularization.

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity**
- 3 Neural Networks
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs

Can a Line Separate These Points?

Consider the XOR function. Four data points, two classes:

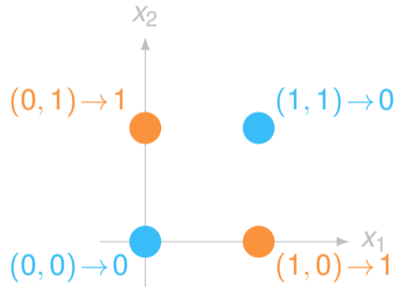
Can a Line Separate These Points?

Consider the XOR function. Four data points, two classes:



Can a Line Separate These Points?

Consider the XOR function. Four data points, two classes:

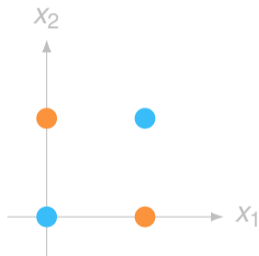


No Linear Boundary Works

Let us try three different lines:

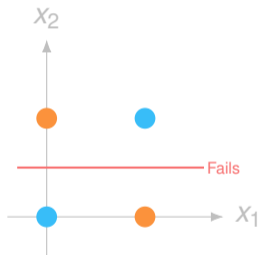
No Linear Boundary Works

Let us try three different lines:



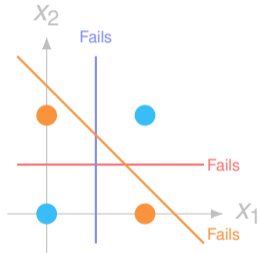
No Linear Boundary Works

Let us try three different lines:



No Linear Boundary Works

Let us try three different lines:



What If We Could Transform the Inputs?

Key insight: we can map inputs into a new space where the classes *are* separable, a linear classifier on top would work.

What If We Could Transform the Inputs?

Key insight: we can map inputs into a new space where the classes *are* separable, a linear classifier on top would work.

Example: define $h = |x_1 - x_2|$.

What If We Could Transform the Inputs?

Key insight: we can map inputs into a new space where the classes *are* separable, a linear classifier on top would work.

Example: define $h = |x_1 - x_2|$.

x_1	x_2	$h = x_1 - x_2 $	y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

What If We Could Transform the Inputs?

Key insight: we can map inputs into a new space where the classes *are* separable, a linear classifier on top would work.

Example: define $h = |x_1 - x_2|$.

x_1	x_2	$h = x_1 - x_2 $	y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Now a simple linear threshold $h > 0.5$ separates perfectly. But we hand-designed this feature.

What If We Could Transform the Inputs?

Key insight: we can map inputs into a new space where the classes *are* separable, a linear classifier on top would work.

Example: define $h = |x_1 - x_2|$.

x_1	x_2	$h = x_1 - x_2 $	y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Now a simple linear threshold $h > 0.5$ separates perfectly. But we hand-designed this feature.

What if we *parameterize* the transformation and optimize it with gradient descent?

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks**
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1\mathbf{x} + \mathbf{b}_1$.

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1\mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1\mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{y} = W_2\mathbf{h} + \mathbf{b}_2$.

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1\mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{y} = W_2\mathbf{h} + \mathbf{b}_2$.

In one line:

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{y} = W_2 \mathbf{h} + \mathbf{b}_2$.

In one line:

$$f_{\theta}(\mathbf{x}) = W_2 \sigma(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{y} = W_2 \mathbf{h} + \mathbf{b}_2$.

In one line:

$$f_{\theta}(\mathbf{x}) = W_2 \sigma(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Why the nonlinearity?

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{\mathbf{y}} = W_2 \mathbf{h} + \mathbf{b}_2$.

In one line:

$$f_{\theta}(\mathbf{x}) = W_2 \sigma(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Why the nonlinearity?

Without σ , composing two linear transforms gives another linear transform:

$$W_2(W_1 \mathbf{x}) = (W_2 W_1) \mathbf{x}.$$

What Is the Neural Network Idea?

Instead of hand-engineering features, stack **learned transformations**:

- 1 Take input \mathbf{x} .
- 2 Apply a **linear transform**: $\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1$.
- 3 Apply a **nonlinear activation**: $\mathbf{h} = \sigma(\mathbf{z})$, element-wise.
- 4 Apply another linear transform: $\hat{y} = W_2 \mathbf{h} + \mathbf{b}_2$.

In one line:

$$f_{\theta}(\mathbf{x}) = W_2 \sigma(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Why the nonlinearity?

Without σ , composing two linear transforms gives another linear transform:

$$W_2(W_1 \mathbf{x}) = (W_2 W_1) \mathbf{x}.$$

The nonlinearity breaks this collapse and lets the model represent nonlinear boundaries.

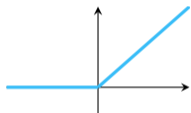
What Are the Common Activation Functions?

The nonlinearity σ is applied element-wise. Three common choices:

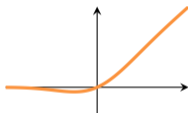
What Are the Common Activation Functions?

The nonlinearity σ is applied element-wise. Three common choices:

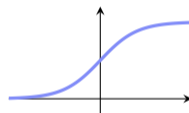
ReLU: $\max(0, z)$



GELU (smooth ReLU)



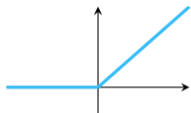
Sigmoid: $\frac{1}{1+e^{-z}}$



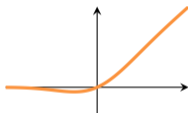
What Are the Common Activation Functions?

The nonlinearity σ is applied element-wise. Three common choices:

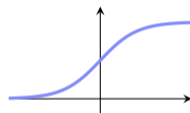
ReLU: $\max(0, z)$



GELU (smooth ReLU)



Sigmoid: $\frac{1}{1+e^{-z}}$

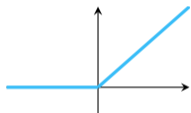


ReLU: gradient is 0 or 1. Simple, fast, widely used.

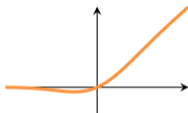
What Are the Common Activation Functions?

The nonlinearity σ is applied element-wise. Three common choices:

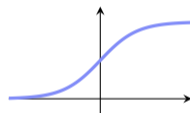
ReLU: $\max(0, z)$



GELU (smooth ReLU)



Sigmoid: $\frac{1}{1+e^{-z}}$



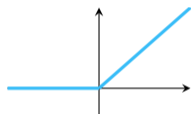
ReLU: gradient is 0 or 1. Simple, fast, widely used.

GELU: smooth approximation of ReLU. Standard in transformers (GPT, BERT).

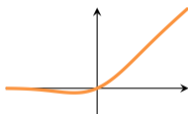
What Are the Common Activation Functions?

The nonlinearity σ is applied element-wise. Three common choices:

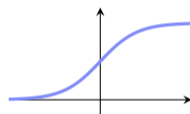
ReLU: $\max(0, z)$



GELU (smooth ReLU)



Sigmoid: $\frac{1}{1+e^{-z}}$

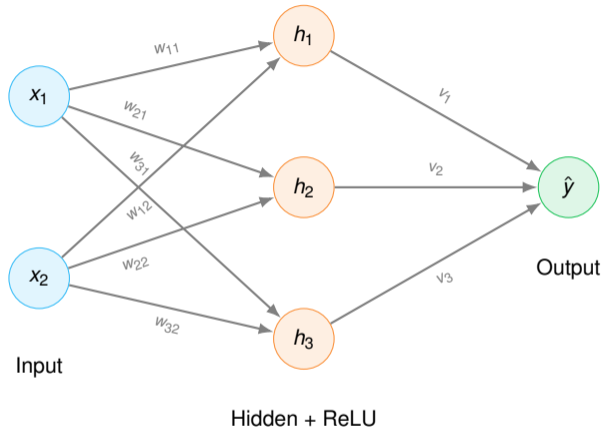


ReLU: gradient is 0 or 1. Simple, fast, widely used.

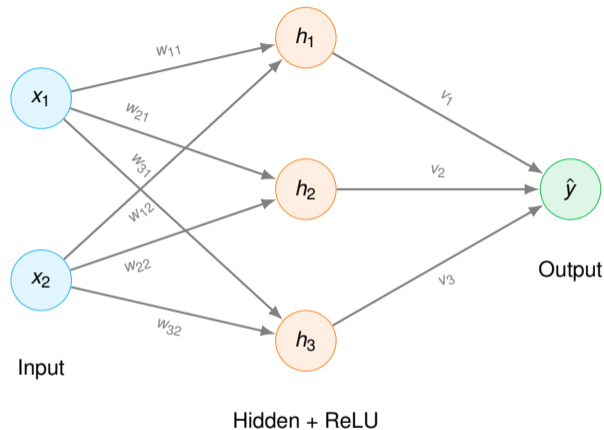
GELU: smooth approximation of ReLU. Standard in transformers (GPT, BERT).

Sigmoid: squashes to $[0, 1]$. Good for probabilities.

How Does a 2-Layer Network Look?

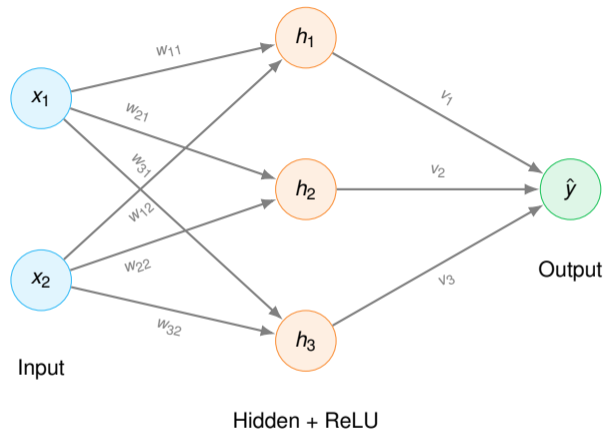


How Does a 2-Layer Network Look?



Formula: $\hat{y} = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_2$, where $h_i = \text{ReLU}(w_{i1} x_1 + w_{i2} x_2 + b_{1,i})$.

How Does a 2-Layer Network Look?



Formula: $\hat{y} = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_2$, where $h_i = \text{ReLU}(w_{i1} x_1 + w_{i2} x_2 + b_{1,i})$.
This is a 2-layer network with 3 hidden neurons. Let us use it to solve XOR.

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Input (0, 0), **target** $y = 0$: $\mathbf{z} = (0, 0, -1)^T$, $\mathbf{h} = (0, 0, 0)^T$, $\hat{y} = 0$. ✓

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Input (0, 0), **target** $y = 0$: $\mathbf{z} = (0, 0, -1)^T$, $\mathbf{h} = (0, 0, 0)^T$, $\hat{y} = 0$. ✓

Input (0, 1), **target** $y = 1$: $\mathbf{z} = (0, 1, 0)^T$, $\mathbf{h} = (0, 1, 0)^T$, $\hat{y} = 1$. ✓

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Input (0, 0), **target** $y = 0$: $\mathbf{z} = (0, 0, -1)^T$, $\mathbf{h} = (0, 0, 0)^T$, $\hat{y} = 0$. ✓

Input (0, 1), **target** $y = 1$: $\mathbf{z} = (0, 1, 0)^T$, $\mathbf{h} = (0, 1, 0)^T$, $\hat{y} = 1$. ✓

Input (1, 0), **target** $y = 1$: $\mathbf{z} = (1, 0, 0)^T$, $\mathbf{h} = (1, 0, 0)^T$, $\hat{y} = 1$. ✓

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Input (0, 0), **target** $y = 0$: $\mathbf{z} = (0, 0, -1)^T$, $\mathbf{h} = (0, 0, 0)^T$, $\hat{y} = 0$. ✓

Input (0, 1), **target** $y = 1$: $\mathbf{z} = (0, 1, 0)^T$, $\mathbf{h} = (0, 1, 0)^T$, $\hat{y} = 1$. ✓

Input (1, 0), **target** $y = 1$: $\mathbf{z} = (1, 0, 0)^T$, $\mathbf{h} = (1, 0, 0)^T$, $\hat{y} = 1$. ✓

Input (1, 1), **target** $y = 0$: $\mathbf{z} = (1, 1, 1)^T$, $\mathbf{h} = (1, 1, 1)^T$, $\hat{y} = 1 + 1 - 2 = 0$. ✓

Solving XOR: Setting Up and Testing Weights

Choose

$$W_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \quad b_2 = 0.$$

Compute

$$\mathbf{z} = W_1 \mathbf{x} + \mathbf{b}_1, \quad \mathbf{h} = \text{ReLU}(\mathbf{z}), \quad \hat{y} = \mathbf{v}^T \mathbf{h} + b_2.$$

Input (0, 0), **target** $y = 0$: $\mathbf{z} = (0, 0, -1)^T$, $\mathbf{h} = (0, 0, 0)^T$, $\hat{y} = 0$. ✓

Input (0, 1), **target** $y = 1$: $\mathbf{z} = (0, 1, 0)^T$, $\mathbf{h} = (0, 1, 0)^T$, $\hat{y} = 1$. ✓

Input (1, 0), **target** $y = 1$: $\mathbf{z} = (1, 0, 0)^T$, $\mathbf{h} = (1, 0, 0)^T$, $\hat{y} = 1$. ✓

Input (1, 1), **target** $y = 0$: $\mathbf{z} = (1, 1, 1)^T$, $\mathbf{h} = (1, 1, 1)^T$, $\hat{y} = 1 + 1 - 2 = 0$. ✓

All four inputs are classified correctly.

Solving XOR: Why This Works

Solving XOR: Why This Works

Key insight (On Binary Inputs):

$$h_1 = \text{ReLU}(x_1) = x_1, \quad h_2 = \text{ReLU}(x_2) = x_2, \quad h_3 = \text{ReLU}(x_1 + x_2 - 1) = x_1 x_2.$$

On binary inputs, $h_3 = 1$ exactly when both inputs are 1, so

$$\hat{y} = h_1 + h_2 - 2h_3 = x_1 + x_2 - 2x_1x_2,$$

which is exactly XOR.

Solving XOR: Why This Works

Key insight (On Binary Inputs):

$$h_1 = \text{ReLU}(x_1) = x_1, \quad h_2 = \text{ReLU}(x_2) = x_2, \quad h_3 = \text{ReLU}(x_1 + x_2 - 1) = x_1 x_2.$$

On binary inputs, $h_3 = 1$ exactly when both inputs are 1, so

$$\hat{y} = h_1 + h_2 - 2h_3 = x_1 + x_2 - 2x_1x_2,$$

which is exactly XOR.

Neural Networks can learn the transformations automatically. The goal is to find good parameters using an optimizer.

What Region of the Input Space Maps to Class 1?

Our network computes

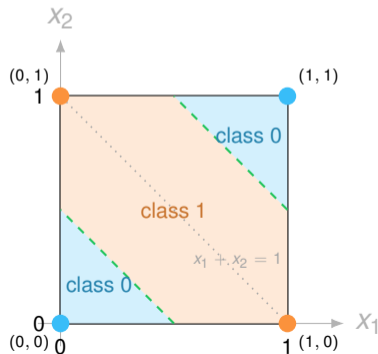
$$\hat{y}(x_1, x_2) = \text{ReLU}(x_1) + \text{ReLU}(x_2) - 2 \text{ReLU}(x_1 + x_2 - 1).$$

What Region of the Input Space Maps to Class 1?

Our network computes

$$\hat{y}(x_1, x_2) = \text{ReLU}(x_1) + \text{ReLU}(x_2) - 2 \text{ReLU}(x_1 + x_2 - 1).$$

On the domain $(x_1, x_2) \in [0, 1]^2$, if we predict class 1 when $\hat{y} > \frac{1}{2}$, then



Going Deeper and Universal Approximation

Stack more layers, each with its own learned transform and nonlinearity:

Going Deeper and Universal Approximation

Stack more layers, each with its own learned transform and nonlinearity:

$$\mathbf{h}_1 = \sigma_1(W_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}_2 = \sigma_2(W_2 \mathbf{h}_1 + \mathbf{b}_2)$$

$$\vdots$$

$$\hat{y} = W_{L+1} \mathbf{h}_L + \mathbf{b}_{L+1}$$

Going Deeper and Universal Approximation

Stack more layers, each with its own learned transform and nonlinearity:

$$\begin{aligned}\mathbf{h}_1 &= \sigma_1(W_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma_2(W_2\mathbf{h}_1 + \mathbf{b}_2) \\ &\vdots \\ \hat{y} &= W_{L+1}\mathbf{h}_L + \mathbf{b}_{L+1}\end{aligned}$$

Each layer refines the representation. Early layers learn simple features, later layers compose them into complex ones.

Going Deeper and Universal Approximation

Stack more layers, each with its own learned transform and nonlinearity:

$$\begin{aligned}\mathbf{h}_1 &= \sigma_1(W_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma_2(W_2\mathbf{h}_1 + \mathbf{b}_2) \\ &\vdots \\ \hat{y} &= W_{L+1}\mathbf{h}_L + \mathbf{b}_{L+1}\end{aligned}$$

Each layer refines the representation. Early layers learn simple features, later layers compose them into complex ones.

This is a **Multi-Layer Perceptron** (MLP), aka a feedforward neural network.

Going Deeper and Universal Approximation

Universal Approximation Theorem (Cybenko 1989, Hornik 1991):

Going Deeper and Universal Approximation

Universal Approximation Theorem (Cybenko 1989, Hornik 1991):

Statement

A neural network with a single hidden layer of sufficient width can approximate any continuous function on a compact domain to arbitrary precision.

Going Deeper and Universal Approximation

Universal Approximation Theorem (Cybenko 1989, Hornik 1991):

Statement

A neural network with a single hidden layer of sufficient width can approximate any continuous function on a compact domain to arbitrary precision.

This is an *existence* result: it says the model *can* represent the function, not that SGD will find those weights.

Going Deeper and Universal Approximation

Universal Approximation Theorem (Cybenko 1989, Hornik 1991):

Statement

A neural network with a single hidden layer of sufficient width can approximate any continuous function on a compact domain to arbitrary precision.

This is an *existence* result: it says the model *can* represent the function, not that SGD will find those weights.

It also says nothing about how wide the layer needs to be. The required width could be exponentially large.

Going Deeper and Universal Approximation

Universal Approximation Theorem (Cybenko 1989, Hornik 1991):

Statement

A neural network with a single hidden layer of sufficient width can approximate any continuous function on a compact domain to arbitrary precision.

This is an *existence* result: it says the model *can* represent the function, not that SGD will find those weights.

It also says nothing about how wide the layer needs to be. The required width could be exponentially large.

The Universal Approximation Theorem says one wide layer suffices in theory. **In practice:** deeper networks are far more parameter-efficient.

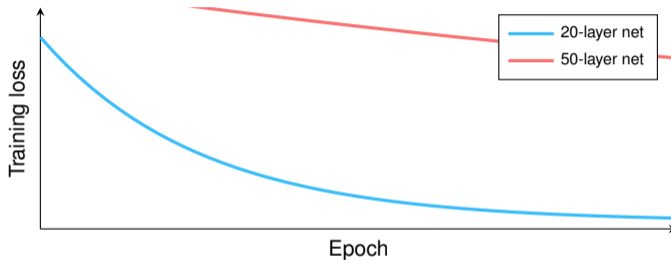
- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks
- 4 Key Building Blocks**
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs

What Goes Wrong in Deep Networks?

Let us try training very deep networks:

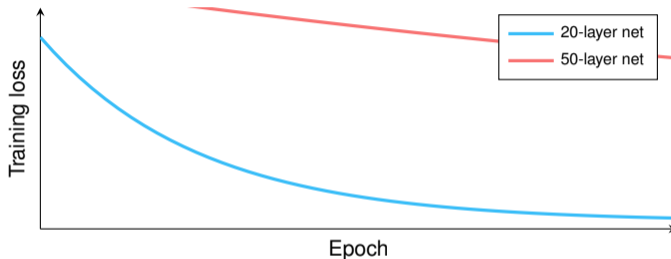
What Goes Wrong in Deep Networks?

Let us try training very deep networks:



What Goes Wrong in Deep Networks?

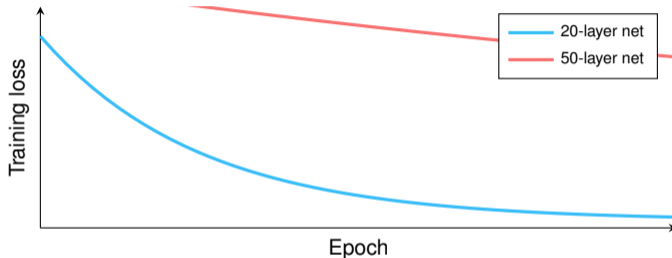
Let us try training very deep networks:



The 50-layer net trains **worse**. This should not happen: it contains all 20-layer nets as a special case (set extra layers to identity).

What Goes Wrong in Deep Networks?

Let us try training very deep networks:



The 50-layer net trains **worse**. This should not happen: it contains all 20-layer nets as a special case (set extra layers to identity).

The question is: what prevents the deeper network from learning?

Why Do Gradients Vanish?

By the chain rule, the gradient passes through every layer:

Why Do Gradients Vanish?

By the chain rule, the gradient passes through every layer:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \cdot \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial W_1}$$

Why Do Gradients Vanish?

By the chain rule, the gradient passes through every layer:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \cdot \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial W_1}$$

This is a product of L Jacobian matrices. If each has spectral norm < 1 :

Why Do Gradients Vanish?

By the chain rule, the gradient passes through every layer:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \cdot \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial W_1}$$

This is a product of L Jacobian matrices. If each has spectral norm < 1 :

$$\left\| \frac{\partial \mathcal{L}}{\partial W_1} \right\| \leq \prod_{k=1}^L \left\| \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \right\| \cdot \left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \right\|$$

Why Do Gradients Vanish?

By the chain rule, the gradient passes through every layer:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \cdot \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial W_1}$$

This is a product of L Jacobian matrices. If each has spectral norm < 1 :

$$\left\| \frac{\partial \mathcal{L}}{\partial W_1} \right\| \leq \prod_{k=1}^L \left\| \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \right\| \cdot \left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_L} \right\|$$

If each factor is 0.9, then with 50 layers: $0.9^{50} \approx 0.005$. **The gradient shrinks exponentially with depth.** Early layers barely update. This is the **vanishing gradient problem**.

What Are Residual Connections?

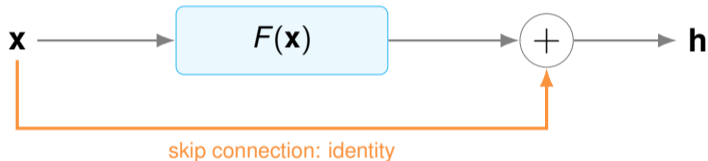
Instead of learning $\mathbf{h} = F(\mathbf{x})$, learn the **residual**:

$$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$$

What Are Residual Connections?

Instead of learning $\mathbf{h} = F(\mathbf{x})$, learn the **residual**:

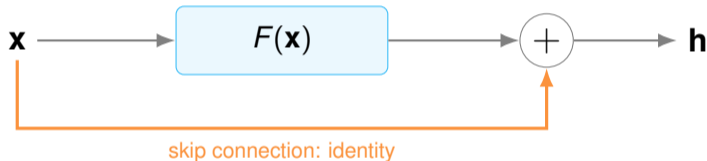
$$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$$



What Are Residual Connections?

Instead of learning $\mathbf{h} = F(\mathbf{x})$, learn the **residual**:

$$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$$

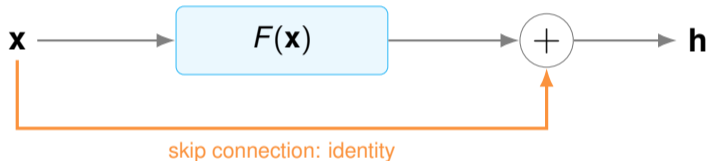


The gradient: $\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} + I$

What Are Residual Connections?

Instead of learning $\mathbf{h} = F(\mathbf{x})$, learn the **residual**:

$$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$$



The gradient: $\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} + I$

The $+I$ means the gradient is pushed away from zero. It flows directly through the skip connection.

Why Do Residual Connections Work So Well?

The network only needs to learn the *residual* $F(\mathbf{x}) = \text{desired output} - \mathbf{x}$.

Why Do Residual Connections Work So Well?

The network only needs to learn the *residual* $F(\mathbf{x}) = \text{desired output} - \mathbf{x}$.
This is the key innovation of ResNet (He et al., 2015).

Why Do Residual Connections Work So Well?

The network only needs to learn the *residual* $F(\mathbf{x}) = \text{desired output} - \mathbf{x}$.

This is the key innovation of ResNet (He et al., 2015).

Before ResNet: training networks deeper than ~ 20 layers was very difficult.

Why Do Residual Connections Work So Well?

The network only needs to learn the *residual* $F(\mathbf{x}) = \text{desired output} - \mathbf{x}$.

This is the key innovation of ResNet (He et al., 2015).

Before ResNet: training networks deeper than ~ 20 layers was very difficult.

After ResNet: networks with 100+ layers trained successfully.

Why Do Residual Connections Work So Well?

The network only needs to learn the *residual* $F(\mathbf{x}) = \text{desired output} - \mathbf{x}$.

This is the key innovation of ResNet (He et al., 2015).

Before ResNet: training networks deeper than ~ 20 layers was very difficult.

After ResNet: networks with 100+ layers trained successfully.

Today, almost **every** major architecture uses residual connections: transformers, modern CNNs, diffusion models.

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Layer Normalization normalizes across features at each layer:

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Layer Normalization normalizes across features at each layer:

$$\text{LayerNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Layer Normalization normalizes across features at each layer:

$$\text{LayerNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where μ, σ^2 are the mean and variance of \mathbf{h} , and γ, β are **learned** scale/shift parameters (same dimension as \mathbf{h}).

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Layer Normalization normalizes across features at each layer:

$$\text{LayerNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where μ, σ^2 are the mean and variance of \mathbf{h} , and γ, β are **learned** scale/shift parameters (same dimension as \mathbf{h}).

RMSNorm (used in LLaMA, Qwen, Gemma) is a simpler variant that skips the mean subtraction:

$$\text{RMSNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h}}{\text{RMS}(\mathbf{h})}, \quad \text{RMS}(\mathbf{h}) = \sqrt{\frac{1}{d} \sum_i h_i^2}$$

Normalization: LayerNorm and RMSNorm

As data flows through many layers, activation distributions can shift dramatically, making training unstable.

Layer Normalization normalizes across features at each layer:

$$\text{LayerNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where μ, σ^2 are the mean and variance of \mathbf{h} , and γ, β are **learned** scale/shift parameters (same dimension as \mathbf{h}).

RMSNorm (used in LLaMA, Qwen, Gemma) is a simpler variant that skips the mean subtraction:

$$\text{RMSNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h}}{\text{RMS}(\mathbf{h})}, \quad \text{RMS}(\mathbf{h}) = \sqrt{\frac{1}{d} \sum_i h_i^2}$$

RMSNorm is simpler, slightly faster, and works just as well.

How Does Dropout Prevent Overfitting?

With millions of parameters, a network can **memorize** the training data: near-zero training loss, high test loss.

How Does Dropout Prevent Overfitting?

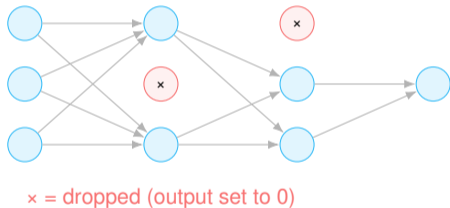
With millions of parameters, a network can **memorize** the training data: near-zero training loss, high test loss.

Dropout (Srivastava et al., 2014): during training, randomly set each neuron's output to 0 with probability p .

How Does Dropout Prevent Overfitting?

With millions of parameters, a network can **memorize** the training data: near-zero training loss, high test loss.

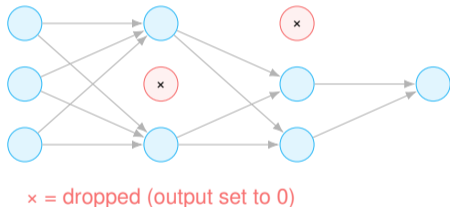
Dropout (Srivastava et al., 2014): during training, randomly set each neuron's output to 0 with probability p .



How Does Dropout Prevent Overfitting?

With millions of parameters, a network can **memorize** the training data: near-zero training loss, high test loss.

Dropout (Srivastava et al., 2014): during training, randomly set each neuron's output to 0 with probability p .

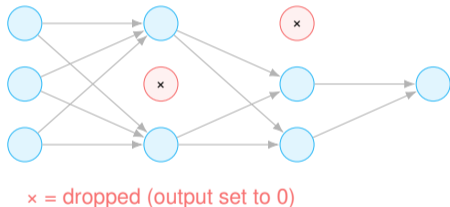


Each forward pass uses a different random subnetwork. No single neuron can be too important.

How Does Dropout Prevent Overfitting?

With millions of parameters, a network can **memorize** the training data: near-zero training loss, high test loss.

Dropout (Srivastava et al., 2014): during training, randomly set each neuron's output to 0 with probability p .



Each forward pass uses a different random subnetwork. No single neuron can be too important.

At test time: use the full network (no dropout).

Building Blocks Summary

Problem	Solution	Mechanism
Vanishing gradients	Residual connections	$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$

Building Blocks Summary

Problem	Solution	Mechanism
Vanishing gradients	Residual connections	$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$
Unstable activations	LayerNorm / RMSNorm	Normalize each layer

Building Blocks Summary

Problem	Solution	Mechanism
Vanishing gradients	Residual connections	$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$
Unstable activations	LayerNorm / RMSNorm	Normalize each layer
Overfitting	Dropout	Random neuron zeroing

Building Blocks Summary

Problem	Solution	Mechanism
Vanishing gradients	Residual connections	$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$
Unstable activations	LayerNorm / RMSNorm	Normalize each layer
Overfitting	Dropout	Random neuron zeroing

Every modern neural network uses some combination of these three. Transformers use all three: residual connections around every block, layer normalization before each block, and dropout on attention weights and hidden states (although dropout is becoming less popular).

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)**
- 6 Training on NCSA Delta GPUs

How Do We Define a Neural Network in PyTorch?

Every neural network in PyTorch inherits from `nn.Module`.

How Do We Define a Neural Network in PyTorch?

Every neural network in PyTorch inherits from `nn.Module`.

Two methods: `__init__` declares layers, `forward` defines the computation.

How Do We Define a Neural Network in PyTorch?

Every neural network in PyTorch inherits from `nn.Module`.

Two methods: `__init__` declares layers, `forward` defines the computation.

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x
```

How Do We Define a Neural Network in PyTorch?

Every neural network in PyTorch inherits from `nn.Module`.

Two methods: `__init__` declares layers, `forward` defines the computation.

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x
```

PyTorch automatically tracks all parameters declared in `__init__` for `.backward()`.

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

```
model = nn.Sequential(  
    nn.Flatten(),           # 28x28 image -> 784 vector  
    nn.Linear(784, 256),    # first hidden layer  
    nn.ReLU(),  
    nn.Linear(256, 128),    # second hidden layer  
    nn.ReLU(),  
    nn.Linear(128, 10)     # output: 10 digit classes  
)
```

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

```
model = nn.Sequential(  
    nn.Flatten(),           # 28x28 image -> 784 vector  
    nn.Linear(784, 256),   # first hidden layer  
    nn.ReLU(),  
    nn.Linear(256, 128),  # second hidden layer  
    nn.ReLU(),  
    nn.Linear(128, 10)    # output: 10 digit classes  
)
```

Use `nn.Sequential` for simple architectures; use `nn.Module` for more complex networks.

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

```
model = nn.Sequential(  
    nn.Flatten(),           # 28x28 image -> 784 vector  
    nn.Linear(784, 256),   # first hidden layer  
    nn.ReLU(),  
    nn.Linear(256, 128),   # second hidden layer  
    nn.ReLU(),  
    nn.Linear(128, 10)     # output: 10 digit classes  
)
```

Use `nn.Sequential` for simple architectures; use `nn.Module` for more complex networks.

```
total = sum(p.numel() for p in model.parameters())  
print(f"Total parameters: {total:,}") # 235,146
```

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

```
model = nn.Sequential(  
    nn.Flatten(),           # 28x28 image -> 784 vector  
    nn.Linear(784, 256),   # first hidden layer  
    nn.ReLU(),  
    nn.Linear(256, 128),   # second hidden layer  
    nn.ReLU(),  
    nn.Linear(128, 10)     # output: 10 digit classes  
)
```

Use `nn.Sequential` for simple architectures; use `nn.Module` for more complex networks.

```
total = sum(p.numel() for p in model.parameters())  
print(f"Total parameters: {total:,}") # 235,146
```

nn.Sequential and Model Inspection

For simple stacked architectures, use `nn.Sequential`:

```
model = nn.Sequential(  
    nn.Flatten(),           # 28x28 image -> 784 vector  
    nn.Linear(784, 256),   # first hidden layer  
    nn.ReLU(),  
    nn.Linear(256, 128),   # second hidden layer  
    nn.ReLU(),  
    nn.Linear(128, 10)     # output: 10 digit classes  
)
```

Use `nn.Sequential` for simple architectures; use `nn.Module` for more complex networks.

```
total = sum(p.numel() for p in model.parameters())  
print(f"Total parameters: {total:,}") # 235,146
```

Breakdown: Layer 1: $784 \times 256 + 256 = 200,960$. Layer 2: $256 \times 128 + 128 = 32,896$. Layer 3: $128 \times 10 + 10 = 1,290$. **Total: 235,146.**

How Do We Load Data?

MNIST: 60,000 training images of handwritten digits (28×28 pixels, 10 classes).

How Do We Load Data?

MNIST: 60,000 training images of handwritten digits (28×28 pixels, 10 classes).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),           # PIL image -> tensor
    transforms.Normalize((0.1307,), (0.3081,)) # mean, std
])

train_data = datasets.MNIST('.', train=True, download=True,
                             transform=transform)
test_data = datasets.MNIST('.', train=False, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1000, shuffle=False)
```

How Do We Load Data?

MNIST: 60,000 training images of handwritten digits (28×28 pixels, 10 classes).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),          # PIL image -> tensor
    transforms.Normalize((0.1307,), (0.3081,)) # mean, std
])

train_data = datasets.MNIST('.', train=True, download=True,
                             transform=transform)
test_data = datasets.MNIST('.', train=False, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1000, shuffle=False)
```

DataLoader handles:

- **Batching**: groups data into mini-batches of size 64.

How Do We Load Data?

MNIST: 60,000 training images of handwritten digits (28×28 pixels, 10 classes).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),           # PIL image -> tensor
    transforms.Normalize((0.1307,), (0.3081,)) # mean, std
])

train_data = datasets.MNIST('.', train=True, download=True,
                             transform=transform)
test_data = datasets.MNIST('.', train=False, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1000, shuffle=False)
```

DataLoader handles:

- **Batching**: groups data into mini-batches of size 64.
- **Shuffling**: randomizes order each epoch (important for SGD).

How Do We Load Data?

MNIST: 60,000 training images of handwritten digits (28×28 pixels, 10 classes).

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),          # PIL image -> tensor
    transforms.Normalize((0.1307,), (0.3081,)) # mean, std
])

train_data = datasets.MNIST('.', train=True, download=True,
                             transform=transform)
test_data = datasets.MNIST('.', train=False, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1000, shuffle=False)
```

DataLoader handles:

- **Batching**: groups data into mini-batches of size 64.
- **Shuffling**: randomizes order each epoch (important for SGD).
- **Parallel loading**: can use multiple workers to load data in the background.

What Does the Training Loop Look Like?

The core of training is just six lines inside the inner loop:

What Does the Training Loop Look Like?

The core of training is just six lines inside the inner loop:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
```

```
for epoch in range(10):  
    for images, labels in train_loader:  
        logits = model(images)           # 1. Forward pass  
        loss = nn.functional.cross_entropy(logits, labels) # 2. Compute loss  
        optimizer.zero_grad()           # 3. Zero old gradients  
        loss.backward()                  # 4. Backprop  
        optimizer.step()                 # 5. Update weights
```

What Does the Training Loop Look Like?

The core of training is just six lines inside the inner loop:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
```

```
for epoch in range(10):  
    for images, labels in train_loader:  
        logits = model(images)           # 1. Forward pass  
        loss = nn.functional.cross_entropy(logits, labels) # 2. Compute loss  
        optimizer.zero_grad()           # 3. Zero old gradients  
        loss.backward()                  # 4. Backprop  
        optimizer.step()                 # 5. Update weights
```

Line by line:

- 1 Forward pass: run the batch through the model, get logits.

What Does the Training Loop Look Like?

The core of training is just six lines inside the inner loop:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)

for epoch in range(10):
    for images, labels in train_loader:
        logits = model(images)           # 1. Forward pass
        loss = nn.functional.cross_entropy(logits, labels) # 2. Compute loss
        optimizer.zero_grad()          # 3. Zero old gradients
        loss.backward()                 # 4. Backprop
        optimizer.step()                # 5. Update weights
```

Line by line:

- 1 Forward pass: run the batch through the model, get logits.
- 2 Compute loss. Then zero old gradients (PyTorch accumulates by default).

What Does the Training Loop Look Like?

The core of training is just six lines inside the inner loop:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)

for epoch in range(10):
    for images, labels in train_loader:
        logits = model(images)           # 1. Forward pass
        loss = nn.functional.cross_entropy(logits, labels) # 2. Compute loss
        optimizer.zero_grad()          # 3. Zero old gradients
        loss.backward()                 # 4. Backprop
        optimizer.step()                # 5. Update weights
```

Line by line:

- 1 Forward pass: run the batch through the model, get logits.
- 2 Compute loss. Then zero old gradients (PyTorch accumulates by default).
- 3 Backpropagation: compute gradients. Optimizer step: update parameters.

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

Softmax converts logits to probabilities:

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

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

Softmax converts logits to probabilities:

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

Cross-entropy measures how far the prediction is from the true label.

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

Softmax converts logits to probabilities:

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

Cross-entropy measures how far the prediction is from the true label.

Example: logits $\mathbf{z} = (2.0, 0.5, -1.0)$, true class is 0.

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

Softmax converts logits to probabilities:

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

Cross-entropy measures how far the prediction is from the true label.

Example: logits $\mathbf{z} = (2.0, 0.5, -1.0)$, true class is 0.

Softmax:

(0.786, 0.175, 0.039)

What Is Cross-Entropy Loss?

The model outputs **logits** z_1, \dots, z_n : raw, unnormalized scores for each class.

Softmax converts logits to probabilities:

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

Cross-entropy measures how far the prediction is from the true label.

Example: logits $\mathbf{z} = (2.0, 0.5, -1.0)$, true class is 0.

Softmax:

$$(0.786, 0.175, 0.039)$$

Cross-entropy:

$$-\ln p_{\text{true}} = -\ln(0.786) \approx 0.241$$

How Do We Evaluate the Model?

Switch to evaluation mode and disable gradient computation:

How Do We Evaluate the Model?

Switch to evaluation mode and disable gradient computation:

```
model.eval() # switches off dropout, changes BatchNorm behavior
correct = 0
total = 0
```

```
with torch.no_grad(): # saves memory: no gradient tracking
    for images, labels in test_loader:
        logits = model(images)
        preds = logits.argmax(dim=1) # highest-scoring class
        correct += (preds == labels).sum().item()
        total += labels.size(0)
```

```
accuracy = correct / total
print(f"Test accuracy: {accuracy:.1%}") # ~98.0%
```

How Do We Evaluate the Model?

Switch to evaluation mode and disable gradient computation:

```
model.eval() # switches off dropout, changes BatchNorm behavior
correct = 0
total = 0
```

```
with torch.no_grad(): # saves memory: no gradient tracking
    for images, labels in test_loader:
        logits = model(images)
        preds = logits.argmax(dim=1) # highest-scoring class
        correct += (preds == labels).sum().item()
        total += labels.size(0)
```

```
accuracy = correct / total
print(f"Test accuracy: {accuracy:.1%}") # ~98.0%
```

Key points:

- `model.eval()`: critical for correct dropout/BatchNorm behavior.

How Do We Evaluate the Model?

Switch to evaluation mode and disable gradient computation:

```
model.eval() # switches off dropout, changes BatchNorm behavior
correct = 0
total = 0
```

```
with torch.no_grad(): # saves memory: no gradient tracking
    for images, labels in test_loader:
        logits = model(images)
        preds = logits.argmax(dim=1) # highest-scoring class
        correct += (preds == labels).sum().item()
        total += labels.size(0)
```

```
accuracy = correct / total
print(f"Test accuracy: {accuracy:.1%}") # ~98.0%
```

Key points:

- `model.eval()`: critical for correct dropout/BatchNorm behavior.
- `torch.no_grad()`: saves memory by not building the computation graph.

How Do We Evaluate the Model?

Switch to evaluation mode and disable gradient computation:

```
model.eval() # switches off dropout, changes BatchNorm behavior
correct = 0
total = 0
```

```
with torch.no_grad(): # saves memory: no gradient tracking
    for images, labels in test_loader:
        logits = model(images)
        preds = logits.argmax(dim=1) # highest-scoring class
        correct += (preds == labels).sum().item()
        total += labels.size(0)
```

```
accuracy = correct / total
print(f"Test accuracy: {accuracy:.1%}") # ~98.0%
```

Key points:

- `model.eval()`: critical for correct dropout/BatchNorm behavior.
- `torch.no_grad()`: saves memory by not building the computation graph.
- `.argmax(dim=1)`: picks the class with the highest logit.

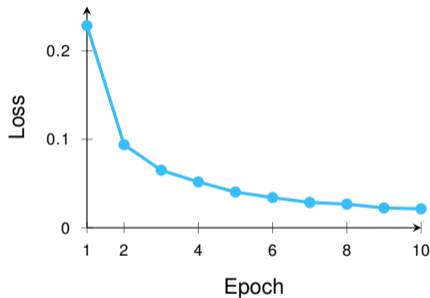
What Do the Training Curves Look Like?

Training our 235K-parameter MLP on MNIST for 10 epochs:

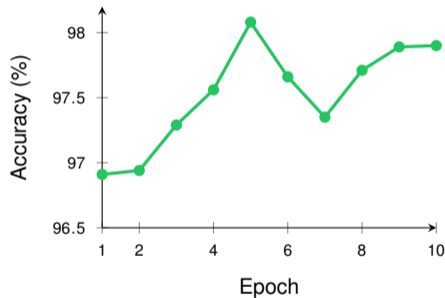
What Do the Training Curves Look Like?

Training our 235K-parameter MLP on MNIST for 10 epochs:

Training Loss



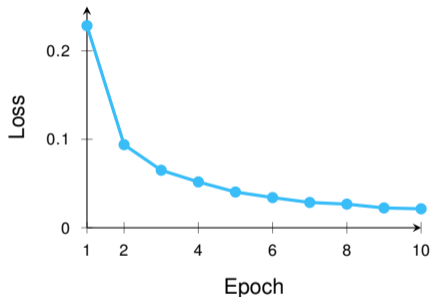
Test Accuracy



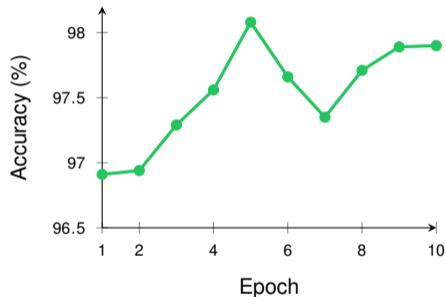
What Do the Training Curves Look Like?

Training our 235K-parameter MLP on MNIST for 10 epochs:

Training Loss



Test Accuracy

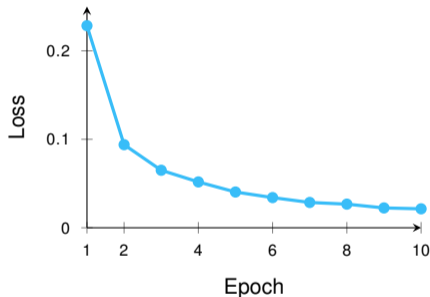


Training loss falls steadily from 0.2284 to 0.0214. Test accuracy reaches about 98%, with the best value 98.08% at epoch 5.

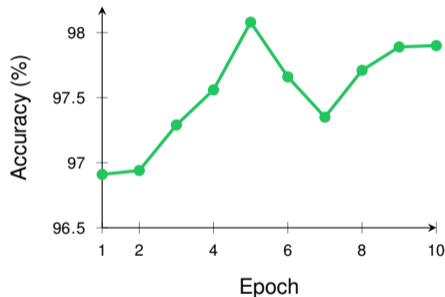
What Do the Training Curves Look Like?

Training our 235K-parameter MLP on MNIST for 10 epochs:

Training Loss



Test Accuracy



Training loss falls steadily from 0.2284 to 0.0214. Test accuracy reaches about 98%, with the best value 98.08% at epoch 5.

The MLP flattens images to a vector, losing all spatial structure. A 2-pixel shift

- 1 Finishing Up: LR Schedules & Double Descent
- 2 The Problem with Linearity
- 3 Neural Networks
- 4 Key Building Blocks
- 5 PyTorch in Practice (MNIST)
- 6 Training on NCSA Delta GPUs**

Why GPUs?

NN training is dominated by **matrix multiplications**, which can be parallelized.

Why GPUs?

NN training is dominated by **matrix multiplications**, which can be parallelized.

CPU

8–16 fast cores
Sequential, high per-core speed

GPU

Thousands of slower cores
Parallel, massive throughput

Why GPUs?

NN training is dominated by **matrix multiplications**, which can be parallelized.

CPU

8–16 fast cores
Sequential, high per-core speed

GPU

Thousands of slower cores
Parallel, massive throughput

An A100 GPU performs ~ 312 TFLOPS in FP16.

Why GPUs?

NN training is dominated by **matrix multiplications**, which can be parallelized.

CPU

8–16 fast cores
Sequential, high per-core speed

GPU

Thousands of slower cores
Parallel, massive throughput

An A100 GPU performs ~ 312 TFLOPS in FP16.
Training that takes hours on CPU takes minutes on GPU.

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

`ssh ncsa_netid@dt-login03.delta.ncsa.illinois.edu` (not necessarily netid).

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

`ssh ncsa_netid@dt-login03.delta.ncsa.illinois.edu` (not necessarily netid).

You land on a login node. **Do not run training on the login node** (shared, no GPU).

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

`ssh ncsa_netid@dt-login03.delta.ncsa.illinois.edu` (not necessarily netid).

You land on a login node. **Do not run training on the login node** (shared, no GPU).

`cd /projects/bgvu/ncsa_netid` (bgvu is this class project).

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

`ssh ncsa_netid@dt-login03.delta.ncsa.illinois.edu` (not necessarily netid).

You land on a login node. **Do not run training on the login node** (shared, no GPU).

`cd /projects/bgvu/ncsa_netid` (bgvu is this class project).

Submit a **SLURM job** to request a compute node with a GPU.

What Is NCSA Delta?

NCSA Delta is the GPU cluster for this course.

Your allocation: 500 A100 GPU-hours per student (can extend to 1k). Plenty for homeworks and the final project.

How to connect:

`ssh ncsa_netid@dt-login03.delta.ncsa.illinois.edu` (not necessarily netid).

You land on a login node. **Do not run training on the login node** (shared, no GPU).

`cd /projects/bgvu/ncsa_netid` (bgvu is this class project).

Submit a **SLURM job** to request a compute node with a GPU.

SLURM is a job scheduler: you describe what resources you need and what to run, and SLURM handles the rest.

Writing and Submitting a SLURM Job

After SSH-ing into Delta and cd into project directory: Create a file `train.sh`:

Writing and Submitting a SLURM Job

After SSH-ing into Delta and cd into project directory: Create a file `train.sh`:

```
#!/bin/bash
#SBATCH --account=bgvu-delta-gpu
#SBATCH --partition=gpuA100x4
#SBATCH --nodes=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=16g
#SBATCH --time=00:05:00
#SBATCH --job-name=mnist_train
#SBATCH --output=mnist_%j.out
module purge
module reset
module load pytorch-conda/2.8
nvidia-smi && python train_mnist.py
```

Submit, monitor, and cancel:

Writing and Submitting a SLURM Job

After SSH-ing into Delta and cd into project directory: Create a file train.sh:

```
#!/bin/bash
#SBATCH --account=bgvu-delta-gpu
#SBATCH --partition=gpuA100x4
#SBATCH --nodes=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=16g
#SBATCH --time=00:05:00
#SBATCH --job-name=mnist_train
#SBATCH --output=mnist_%j.out
module purge
module reset
module load pytorch-conda/2.8
nvidia-smi && python train_mnist.py
```

Submit, monitor, and cancel:

```
sbatch train.sh           # Output sth like: Submitted batch job 17437859
squeue -u $USER          # check status
scancel 17437859         # cancel a job with id 17437859
cat mnist_17437859.out   # read output of job 17437859
```

The GPU Training Script

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_data = datasets.MNIST('.', train=True, download=True,
                             transform=transform)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
```

The GPU Training Loop

```
device = torch.device('cuda') # use GPU
model = nn.Sequential(
    nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
    nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 10)
).to(device) # move model to GPU

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)

for epoch in range(10):
    total_loss = 0
    for images, labels in train_loader:
        images = images.to(device) # move data to GPU
        labels = labels.to(device) # move labels to GPU

        logits = model(images)
        loss = nn.functional.cross_entropy(logits, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

print(f"Epoch {epoch}: loss={total_loss/len(train_loader):.4f}")
```

What's the damage?

```
[eharb@dt-login03 eharp]$ jobcharge
Charges for user eharb and account all from 2026-03-08-23:48:11 through 2026-04-08-23:48:11.
Account          Charge (SU)
-----
bgvu-delta-gpu    0.20
```

```
[eharb@dt-login03 eharp]$ accounts
Project Summary for User 'eharb':
```

Account	Balance(Hours)	Deposited(Hours)	Project
-----	-----	-----	-----
bgvu-delta-gpu	2999	3000	gpu-accelerated llm...

For more details, see

<https://docs.ncsa.illinois.edu/systems/delta/en/latest/index.html>