

CSE 592 - Discussion 6

Optimization & Deep Learning

Zhengjie (Jerry) Xu / Fall 26

Logistics

Today: backpropagation and autograd, optimizers (momentum $\rightarrow$ Adam $\rightarrow$ AdamW), training deep networks, convolutions, diffusion, and generalization.

Recap: backpropagation

Forward pass: compute each intermediate value in order, up to the loss. Save them.

Backward pass: start from $\partial L / \partial L = 1$ and go in reverse order.

$$\partial L / \partial(\text{input}) = \partial L / \partial(\text{output}) \times \text{local derivative}$$

Each step needs only the gradient from the layer above and one local derivative.

Backward costs about as much as forward.

Training must store activations for the backward pass, so it uses much more memory.

Backpropagation by hand

$z = W_1x$, $h = \text{ReLU}(z)$, $s = w \cdot h$, $r = s - y$, $L = r^2$. $x = 1$, $y = 1$, $W_1 = 2$, $w = -1$.

Forward: $z = 2$, $h = 2$, $s = -2$, $r = -3$, $L = 9$

Gradient	Chain rule	Value
$\partial L / \partial r$	$2r$	-6
$\partial L / \partial s$	$\partial L / \partial r \cdot 1$	-6
$\partial L / \partial w$	$\partial L / \partial s \cdot h$	-12
$\partial L / \partial h$	$\partial L / \partial s \cdot w$	6
$\partial L / \partial z$	$\partial L / \partial h \cdot 1[z > 0]$	6
$\partial L / \partial W_1$	$\partial L / \partial z \cdot x$	6

Q: After one gradient step, does w go up or down? Does W_1 ?

A: w goes up (gradient -12), W_1 goes down (gradient 6): both push s toward y .

When a value is used twice

If x feeds several nodes, add up the gradient along every edge out of x .

$$\partial L / \partial x = \sum \text{over uses } j \text{ of } \partial L / \partial f_j \cdot \partial f_j / \partial x$$

Q: $u = x^2$, $v = 3x$, $L = u + v$, at $x = 2$. What is $\partial L / \partial x$?

A: Path through u : $1 \cdot 2x = 4$. Path through v : $1 \cdot 3 = 3$. Total 7.

Check directly: $L = x^2 + 3x$, so $dL/dx = 2x + 3 = 7$.

Backprop visits nodes in reverse topological order so all uses are done first.

Gradient checking

Nudge one parameter and see how the loss changes:

$$\partial L / \partial \theta_i \approx (L(\theta + h e_i) - L(\theta)) / h$$

Example: $L(\theta) = \theta^2$ at $\theta = 3$, $h = 0.01$: $(9.0601 - 9) / 0.01 = 6.01$. Exact: 6.

Q: A network has 1 million parameters. What does this cost vs. backprop?

A: About 1 million extra forward passes, vs. roughly one forward + one backward.

Too slow to train with, but easy to get right: use it to test backprop on a few parameters.

Autograd and the PyTorch training loop

Write only the forward pass. Each operation records a node in the graph.

`loss.backward()` runs backprop on the recorded graph. Loops and ifs just work.

```
loss = ((model(x) - y) ** 2).mean()
```

```
opt.zero_grad(); loss.backward(); opt.step()
```

Q: What goes wrong if you forget `opt.zero_grad()`?

A: `.grad` accumulates (`+=`). Each step uses the sum of all past gradients.

ReLU at exactly 0 isn't differentiable: libraries just pick a value, e.g. 0.

Recap: why optimizing neural nets is harder

Linear regression: convex. One global minimum, no saddle points.

Neural networks: not convex. Local minima, saddle points, flat regions, narrow valleys.

We always use minibatch SGD: g_t = average gradient over B random examples.

Q: Why can't we initialize every weight of a network to 0?

A: Hidden units in a layer would compute the same thing, get the same gradient,

A: and stay identical forever. Random initialization breaks the symmetry.

Momentum

$$u_t = \beta u_{(t-1)} + (1 - \beta) g_t, \quad \theta_t = \theta_{(t-1)} - \eta u_t, \quad \beta = 0.9 \text{ typical}$$

Narrow valley: the steep coordinate flips sign, the flat one is steady.

Step t	Steep g	Flat g	Steep $\hat{u}$ (corrected)	Flat $\hat{u}$ (corrected)
1	+4	+1	4.00	1.00
2	-4	+1	-0.21	1.00
3	+4	+1	1.34	1.00
4	-4	+1	-0.21	1.00

Q: Why does momentum let us use a larger η here?

A: The flips cancel, so the steep direction stops zigzagging. The steady direction keeps full size.

Bias correction

We start at $u_0 = 0$, so early averages are pulled toward 0.

Q: $\beta = 0.9$ and the first gradient is $g_1 = 10$. What is u_1 ?

A: $u_1 = 0.9 \cdot 0 + 0.1 \cdot 10 = 1$. Only 10% of the real gradient.

The weights on real gradients sum to $1 - \beta^t$, so divide by it:

$$\hat{u}_t = u_t / (1 - \beta^t) \quad \hat{u}_1 = 1 / 0.1 = 10 \quad \checkmark$$

Step 2 with $g_2 = 10$: $u_2 = 1.9$, $\hat{u}_2 = 1.9 / 0.19 = 10$. The fix fades as t grows.

Adaptive step sizes: RMSProp

$$v_t = \beta v_{(t-1)} + (1 - \beta) g_t^2, \quad \theta \leftarrow \theta - \eta g_t / (\sqrt{v_t} + \varepsilon) \quad (\text{per coordinate})$$

Two coordinates with steady gradients 100 and 0.05:

Q: How big is each coordinate's step with SGD ($\eta = 0.01$)? With RMSProp?

A: SGD: 1 and 0.0005. RMSProp: $\sqrt{v} \approx |g|$, so both steps $\approx \eta = 0.01$.

Each coordinate is scaled by its own typical gradient size.

At a saddle point, a tiny but consistent gradient still gets a full-size step.

Adagrad (2011) sums all past g^2 instead of averaging, so its steps only shrink.

Adam = momentum + RMSProp

$$u = \beta_1 u + (1 - \beta_1)g, \quad v = \beta_2 v + (1 - \beta_2)g^2 \quad (\beta_1 = 0.9, \beta_2 = 0.999)$$

$$\hat{u} = u / (1 - \beta_1^t), \quad \hat{v} = v / (1 - \beta_2^t), \quad \theta \leftarrow \theta - \eta \hat{u} / (\sqrt{\hat{v}} + \epsilon)$$

Q: On step 1, how big is Adam's step in each coordinate?

A: $\hat{u} = g_1$ and $\hat{v} = g_1^2$, so the step is $\eta \cdot g_1 / |g_1|$: size η everywhere.

Q: What if we forgot to bias-correct v ?

A: $\sqrt{v_1} = \sqrt{0.001} \cdot |g_1| \approx 0.032 |g_1|$, so the first step is about 32× too large.

Weight decay, AdamW, and Muon

L2 penalty $(\lambda/2)\|\theta\|^2$ adds $\lambda\theta$ to the gradient. The SGD step becomes:

$$\theta \leftarrow (1 - \eta\lambda) \theta - \eta g$$

$\eta = 0.1, \lambda = 0.01$: each step multiplies θ by 0.999. After 1000 steps: ≈ 0.37 .

Q: In Adam, why not just add $\lambda\theta$ to the gradient g ?

A: It gets divided by $\sqrt{\hat{v}}$, so weights with large gradients decay less. Not the intent.

AdamW: keep the decay separate: $\theta \leftarrow \theta - \eta(\hat{u} / (\sqrt{\hat{v}} + \epsilon) + \lambda\theta)$.

Muon (2024): normalize whole weight matrices instead of each coordinate.

Learning-rate schedule: warmup, then cosine

Warmup: $\eta_t = \eta_{\max} \cdot t / T_w$ for $t \leq T_w$. Then cosine decay with $p = (t - T_w)/(T - T_w)$:

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \cdot (1 + \cos \pi p) / 2$$

$\eta_{\max} = 0.1$, $\eta_{\min} = 0$, $T_w = 100$, $T = 1100$.

Q: What is η at $t = 50, 100, 350, 600$, and 1100 ?

t	50	100	350	600	1100
η_t	0.05	0.1	≈ 0.085	0.05	0

Warmup: early gradients (and Adam's v) are noisy. Decay: big steps first, then settle.

Is deeper always better?

Add 36 layers to a 20-layer ReLU MLP. The 56-layer net can copy the 20-layer one: set each extra layer to the identity ($W = I$, $b = 0$; ReLU keeps nonnegative inputs). Yet in experiments the 56-layer net has higher training and test error.

Q: Is this approximation error or estimation error?

A: Neither, really: training error went up. Good weights exist; the optimizer didn't find them.

So the problem with depth is optimization. The rest of this section fixes it.

Vanishing and exploding gradients

Backprop multiplies by a local derivative at every layer.

Simple case: each of 50 layers multiplies the gradient by the same number a .

Q: What happens to the gradient at layer 1 if $a = 0.9$? If $a = 1.1$?

A: $0.9^{50} \approx 0.005$: it vanishes. $1.1^{50} \approx 117$: it explodes.

Small differences per layer compound exponentially with depth.

In real MLPs these are matrices: some directions shrink, others grow.

Activations in the forward pass have the same problem.

Initialization: keeping the scale

One unit: $z = \sum_i w_i x_i$ over n inputs, with independent weights of mean 0.

$\text{Var}(z) = n \cdot \text{Var}(w) \cdot \text{Var}(x) \rightarrow$ keep it constant with $\text{Var}(w) = 1/n$

ReLU zeroes the negative half, so it halves the average squared size.

He initialization: $\text{Var}(w) = 2/n$

Q: $n = 512$ inputs per unit. What standard deviation should the weights have?

A: $\text{Var}(w) = 2/512 \approx 0.0039$, so std = 0.0625.

Using $1/n$ with ReLU: squared size halves each layer. After 50 layers: $0.5^{50} \approx 10^{-15}$.

BatchNorm vs. LayerNorm by hand

Normalize: subtract the mean, divide by $\sqrt{\text{variance}}$ (then learn a scale γ and offset β).

	F1	F2	F3
Ex. 1	3	3	6
Ex. 2	5	1	0
Ex. 3	1	5	3
Ex. 4	3	7	3

Q: BatchNorm on column F1? LayerNorm on row Ex. 1?

A: F1: mean 3, var 2 $\rightarrow$ [0, 1.41, -1.41, 0].

A: Ex. 1: mean 4, var 2 $\rightarrow$ [-0.71, -0.71, 1.41].

BatchNorm: per feature, across the batch. Needs running statistics at test time.

LayerNorm: per example, across its features. Same rule in training and testing.

Q: Why can't BatchNorm use batch statistics to predict on a single example?

A: With one example, each feature's variance is 0 and subtracting the mean gives 0.

Residual connections

Instead of learning the full map, learn a change to the input:

$$\text{output} = x + F(x)$$

Q: What should F output for the block to compute the identity?

A: $F(x) = 0$, which is easy: e.g. set the last layer's weights to 0.

Gradient: $\partial \text{output} / \partial x = 1 + \partial F / \partial x$. The 1 gives a direct path back from the loss.

With shortcuts, ResNet-34 trains better than ResNet-18, up to 152 layers on ImageNet.

ConvNeXt (2022) keeps convolutions + residuals, adding LayerNorm and GELU.

Dropout

Training: zero each activation with probability p ; scale the survivors by $1/(1 - p)$.

Activations $h = [3, 5, 1, 7]$, $p = 0.25$, mask $m = [1, 1, 0, 1]$.

Q: What does the next layer receive?

A: $[3, 5, 0, 7] \cdot 4/3 = [4, 6.67, 0, 9.33]$.

Scaling keeps the expected value: $E[\tilde{h}_j] = h_j$.

Q: What do we do at prediction time?

A: Use all activations, unscaled: $[3, 5, 1, 7]$. No randomness.

Recap: why convolutions for images?

A 224×224 RGB image has $224 \cdot 224 \cdot 3 = 150,528$ numbers.

An MLP layer with 1000 units on it: about 150 million weights.

And a detector learned at one position is not reused at another.

Convolution: slide one small filter over every position. Same weights everywhere.

Q: How many weights in a conv layer with 64 filters of size 3×3 on an RGB input?

A: $3 \cdot 3 \cdot 3 \cdot 64 = 1,728$ (plus 64 biases), for any image size.

A convolution by hand

2	0	1
1	3	0
0	1	2

Input (3×3)

1	-1
0	2

Filter (2×2)

8	-1
0	7

Output (stride 1)

Q: What is the output for the top-left window (blue)?

A: $2 \cdot 1 + 0 \cdot (-1) + 1 \cdot 0 + 3 \cdot 2 = 8$.

After ReLU: $[[8, 0], [0, 7]]$. 2×2 max pooling: 8.

Q: For an $n \times n$ input and $k \times k$ filter, stride 1, how big is the output?

A: $(n - k + 1) \times (n - k + 1)$. Here $3 - 2 + 1 = 2$.

Receptive fields and the CNN pipeline

Receptive field: the input region that can affect one unit.

Stacked 3×3 convolutions, stride 1: 1 layer $\rightarrow 3 \times 3$, 2 layers $\rightarrow 5 \times 5$, 3 $\rightarrow 7 \times 7$.

Q: How many stacked 3×3 layers until one unit sees a 15×15 region?

A: Each layer adds 2: $3 + 2(L - 1) = 15$, so $L = 7$.

Pooling and striding shrink the maps, so receptive fields grow faster.

LeNet-5 pattern: conv $\rightarrow$ activation $\rightarrow$ downsample, repeat, then a classifier.

Later layers: fewer positions, more channels.

From features to class probabilities

Last layer: class scores $s = W h$. Softmax turns scores into probabilities:

$$P(\text{class } k) = e^{(s_k)} / \sum_j e^{(s_j)}$$

Loss: cross-entropy, $-\log P(\text{correct class})$. Generalizes logistic regression.

Q: Scores [2, 1, 0]. What are the probabilities?

A: $e^2 \approx 7.39$, $e \approx 2.72$, $e^0 = 1$, total $\approx 11.11 \rightarrow [0.665, 0.245, 0.090]$.

Q: Cross-entropy if class 1 is correct? If class 3 is correct?

A: $-\ln 0.665 \approx 0.41$. $-\ln 0.090 \approx 2.41$.

Data augmentation, and what CNNs learn

Create extra training data by label-preserving transformations:

random crops, horizontal flips, small color changes, small rotations.

Q: Which of these are safe? (a) flipping cat photos, (b) rotating a handwritten 6

Q: by 180°, (c) horizontally flipping a “no left turn” traffic sign.

A: (a) is safe. (b) turns a 6 into a 9. (c) turns it into a “no right turn” sign.

AlexNet (2012): GPUs, ReLU, augmentation, dropout. Top-5 error 15.3% vs. 26.2%.

Learned filters: edges and colors in layer 1, then textures, then object parts.

Diffusion: learning from unlabeled images

Make a training pair from a clean image x_0 : pick noise level t , draw $\varepsilon \sim N(0, I)$,

$$x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{1 - \bar{\alpha}_t} \cdot \varepsilon. \quad \text{Input } (x_t, t), \text{ target } \varepsilon.$$

One pixel: $x_0 = 0.8$, $\varepsilon = -0.5$, $\bar{\alpha}_t = 0.64$.

Q: What is the noisy input x_t ? If the model predicts ε exactly, can we recover x_0 ?

A: $x_t = 0.8 \cdot 0.8 + 0.6 \cdot (-0.5) = 0.34$. $x_0 = (0.34 + 0.6 \cdot 0.5) / 0.8 = 0.8$.

Self-supervised: we made the noise, so we know the target.

Generate: start from pure noise, denoise step by step to an image.

Generalization, revisited

Zhang et al.: CNNs reach 0 training error even with random labels.

Q: 10 classes, random labels, 0 training error. What is the test error?

A: About 90%: chance level. There's nothing to generalize.

So the network can memorize. Counting $|F|$ can't explain why it generalizes on real labels.

Double descent: test error falls, peaks near the interpolation threshold, then falls again.

What decides which fitting function we get: architecture, data and augmentation, the optimizer and initialization, and regularization.

Summary

Backprop: forward pass saves values; backward pass applies the chain rule block by block.

Optimizers: momentum averages gradients; RMSProp rescales per coordinate;

Adam combines both with bias correction; AdamW decouples weight decay.

Schedules: warmup, then decay. Initialization: $\text{Var}(w) = 2/n$ for ReLU.

Depth: normalization (BatchNorm, LayerNorm) and residual connections.

Regularize: dropout, augmentation, weight decay.

Images: convolutions share local filters; diffusion learns to predict noise.

Next: language models and attention.