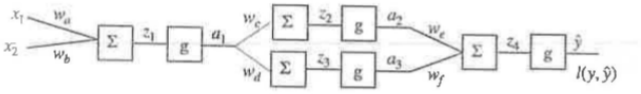
Backpropagation

Chain rule on a computation graph For each parameter w, multiply local derivatives along every path from w to the loss ℓ.



General rules

· **Chain (series):** $\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial y} \cdot \frac{\partial y}{\partial z} \cdots \frac{\partial u}{\partial w}$

· **Linear node** $z = \sum_i w_i x_i$: $\frac{\partial z}{\partial w_i} = x_i, \;\; \frac{\partial z}{\partial x_i} = w_i$

· **Activation** $a = g(z)$: $\frac{\partial a}{\partial z} = g'(z)$

Worked example Network: Output-side weight (no branching): $\frac{\partial \ell}{\partial w_e} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_4} \cdot \frac{\partial z_4}{\partial w_e}$

$\frac{\partial \ell}{\partial w_c} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_c}$

Input weight w_a (here a_1 *branches* into z_2 and z_3, so two paths sum) at a and anything past it:

$$\frac{\partial \ell}{\partial w_a} = \frac{\partial \ell}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_4} \cdot \left[\frac{\partial z_4}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} + \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial a_1} \right] \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_a}$$

where each $\frac{\partial \ell}{\partial a_k}$ continues forward through $z_4, \hat{y}, \ell$.

NN

Definitions

· ReLU: f(x) = max(0, x). Range: [0, ∞)
  · Can add/compose ReLUs to make complex shapes
  · Default for hidden layers. Fast, simple, avoids vanishing gradient . Watch for dead neurons (z ≤ 0), called dying ReLU, fixed by Leaky ReLU

· Sigmoid: $1/(1+e^{-x})$. Range: (0, 1)
  · Use in output layer for binary classification: don't use in hidden layers (vanishing gradient, σ' tiny when $|z|$ large, multiplying many of these becomes 0)

· Tanh: $(e^x - e^{-x})/(e^x + e^{-x})$. Range: (–1, 1)
  · Hidden layers when 0-centered outputs matter. Better than sigmoid but still has vanishing gradient at saturation

Ways to reduce underfitting High bias, bad train+val, Increase number of neurons in hidden layers to represent more complex patterns, reduce dropout rate *Ways to reduce overfitting* High var, good train+bad val, Introduce regularization to punish large gradients/weights, use dropout (also has the side bonus of making your network more efficient since less of it is being used), *Param calculation* For FF weights are always output by input, so for a 20input-¿50-¿100-¿3 output network, there are (20*50 + 50) + (50*100 + 100) + (100*3 + 3) parameters used in total, including biases.

L1 / L2 Regularization

Penalties added to loss:

$$L_2: \; \tfrac{\lambda}{2}\|w\|_2^2 = \tfrac{\lambda}{2}\sum_i w_i^2 \qquad L_1: \; \lambda\|w\|_1 = \lambda \sum_i |w_i|$$

Vector derivatives

$$\nabla_w \tfrac{1}{2}\|w\|_2^2 = w \qquad \nabla_w \|w\|_1 = \text{sign}(w)$$

(componentwise; subgradient at 0). Update with L_2: $w \leftarrow w - \alpha(\nabla \ell + \lambda w) = (1 - \alpha\lambda)w - \alpha \nabla \ell$ – "weight decay."

Effects

· L_2: shrinks all weights smoothly toward 0, never exactly 0. Penalizes large weights heavily (quadratic). Forms a circle

· L_1: pushes weights to *exactly* 0 ⇒ sparsity / feature selection. Penalty grows linearly. Forms a diamond

Optimizers

Initialize with uniform variance to prevent exploding gradient

Let $g_t = \nabla_w \ell(w_t)$, learning rate α.

SGD $w_{t+1} = w_t - \alpha g_t$, using particular datapoints. minibatch if done over B datapoints, where $g_t = \frac{1}{b}\sum_b g_i$ of their gradients

Momentum (heavy ball) accumulate exponentially-decaying avg of past gradients; dampens oscillation across ravines, accelerates along consistent directions.

$$v_{t+1} = \beta v_t + (1-\beta) g_t, \qquad w_{t+1} = w_t - \alpha v_{t+1}$$

Adam (Adaptive Moment Estimation)

$m = b1 * m + (1 - b_1) * g$[1st moment, $b1 = 0.9$]

$v = b2 * v + (1 - b_2) * g^2$[2nd moment, $b2 = 0.999$]

$m_{hat} = m/(1 - b_1^t), v_{hat} = v/(1 - b_2^t)$[bias corr]

$w = w - eta * m_{hat}/(sqrt(v_hat) + eps)$

RMSprop: like Adam but without 1st moment (no m).

Convolution / Pooling Dimensions

Conv output spatial size input $n \times n$, filter $f \times f$, stride s, channels C, padding p, and K output filters, so a convolution layer of $f \times f \times C \times K$:, produces output $n_{out} \times n_{out} \times K$

$$\boxed{n_{\text{out}} = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1}$$

Conv parameter count input has C_{in} channels, layer has C_{out} filters of size $f \times f$:

$$\#\text{weights} = f \cdot f \cdot C_{\text{in}} \cdot C_{\text{out}}$$

$$\#\text{params} = f \cdot f \cdot C_{\text{in}} \cdot C_{\text{out}} + C_{\text{out}} \text{ (biases)}$$

Max pool 0 learnable parameters *FC layer* from flat d inputs to N outputs: #params $= dN + N$.

BatchNorm per channel: $2C$ learnable $(\gamma, \beta) + 2C$ running stats (non-learnable). Goal is to mitigate interal covariance, leading to faster convergence

Padding finder for "same padding" To find padding given you want the output size to be equal to input size, set size formula equal to n and solve for p.

Inception Modules Parallel paths with different receptive field sizes and operations to capture sparse patterns of correlations in the stack of feature maps; Use 1x1 convolutions for dimensionality reduction before expensive convolutions; May be useful to increase # of parallel sets inside block

im2col scans input image patches (e.g., 3×3) and flattens them into individual columns in a new matrix. If a filter has C channels, each column will have $C \times$ patch_height $\times$ patch_width elements

Object Detection

Confusion-matrix metrics

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

Precision: "of predicted positives, how many were correct." Recall: "of actual positives, how many did we find."

Intersection over Union (IoU) for predicted box A and ground-truth box B:

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Detection counts as TP if IoU ≥ threshold (commonly 0.5) *and* class matches.

Pipeline (mAP) for each class: sort predictions by confidence, sweep threshold to get precision-recall curve, AP = area under curve. mAP = mean over classes (sometimes also over IoU thresholds, e.g. COCO uses 0.5:0.05:0.95).

Sliding window, usually 1 stage slide a window across the image and evaluate a detection model at each location, housands of windows to evaluate; efficiency and low false positive rates are essential, difficult to extend to a large range of scales/aspect ratios. EX: overfeat, yolo (conv feature maps to 2 fc layers), single-shot multi-box (Like YOLO, predict bding boxes directly from conv maps, Unlike YOLO, don't use FC layers and predict different size boxes from conv maps at different resolutions, uses anchors or predefined reference shapes)

Proposal-driven detection Generate and evaluate a few hundred region proposals, Take advantage of low-level perceptual organization clues, Can be category-specific or category-independent, Classifier can be slower but more powerful. Extract "blobby" features as regions, commonly used with r-cnn

R-CNN Regions w CNN features, extract region proposals, compute cnn features, linear classification with SVM. But this isn't an end-to-end system, with different training regimes for each of the different losses (spt for classification, sft for detection, log loss for softmax, hinge loss for svms, least squares for bounding-box). Also slow! Pros: Can be plugged into deep arch, accurates *Fast R-CNNS* Convnet the whole thing, propose regions, region-of-interest pool (pool together regions) - predicting probs for c+1 classes (0 is background) and drawing c bounding boxes, FF then softmax and bounding box. Entire task is trainable at once! Speeds up CNN computation - instead of once per proposal, it's over the entire image Only possible disadvantage: downsampling may hurt detection for small objects. *Faster R-CNN* Speed up generating region proposals through a region proposal network, Slide a small window (3x3) over the conv5 layer, Predict object/no object, Regress bounding box coordinates with reference to anchors (3 scales x 3 aspect ratios) **RNNs and LSTMs**

Vanilla RNN hidden state recurrence:

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b), \quad y_t = W_{hy} h_t$$

Same weights at every timestep (parameter sharing). Trained with backprop through time (BPTT): unroll, apply chain rule. Weight updates are computed for each copy in the unfolded net, then summed (or avged) and applied to the RNN weights

Problem Advantages: Can process any length input, computation canuse info from many steps back, model size doesn't increase, same weights applied on every timestep Disadvantages: Slow, difficult to access, in practice hard to access info from steps back, repeated multiplication by W_{hh} over many steps causes **vanishing** (eigenvalues < 1) or **exploding** (> 1) gradients $\Rightarrow$ can't learn long-range dependencies. Exploding: clip gradients. Vanishing: use gates.

LSTM adds a cell state c_t with additive updates (gradient highway). Three gates control flow:

$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$	forget, what to remove
$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$	input, control what to store
$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$	step 2 of store, candidate
$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$	update, forget (lhs) and add new(rhs)
$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$	output p1, decide what from prev
$h_t = o_t \odot \tanh(c_t)$	output p2, cell state

Forget gate of 1 + input of 0 $\Rightarrow c_t = c_{t-1}$ exactly (no decay), so gradients pass unchanged.

GRU simpler: combines forget+input into one gate, merges h and c. Fewer params, often comparable.

Attention & Transformers

Scaled dot-product attention given queries $W_q, W_k, W_v \in \mathbb{R}^{x \times d_k}$, $X \in \mathbb{R}^{n \times x}$, $Q, K, V, Z \in \mathbb{R}^{n \times d_k}$, where $Q = XW_q$ and so on, Q = decoder output, K, V = encoder output, so if encoder outputs 4x512 and decoder 2x512, Q = 2x64, K=V=4x64:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \in \mathbb{R}^{n \times d_k}$$

$\sqrt{d_k}$ prevents dot products from growing large $\Rightarrow$ softmax saturating $\Rightarrow$ vanishing gradients.

Self-attention Q, K, V all from same input X: $Q = XW^Q$, $K = XW^K$, $V = XW^V$. Each token attends to every other.

Multi-head run h attention ops in parallel with different projections, concat, project:

$$\text{MHA}(X) = [\text{head}_1; \ldots; \text{head}_h]W^O$$

Lets the model attend to different subspaces / relations simultaneously.

So with h heads, $Z_i \in \mathbb{R}^{n \times d_k}, Z \in \mathbb{R}^{n \times hd_k}, W_o \in \mathbb{R}^{x \times xh}$

Transformer block

1. Multi-head self-attention $\rightarrow$ residual + LayerNorm
2. Position-wise feed-forward (2 linear layers w/ ReLU/GELU) $\rightarrow$ residual + LayerNorm

Positional encoding attention is permutation-equivariant; add sinusoidal or learned positional vectors to embeddings so order matters.

What teli did: given position $p, i : 0 - 255$, return matrix pos(p, 2i), pos(p, 2i + 1), where pos(p, 2i) = $sin(p/(100000^(2i/512)))$, pos(p, 2i+1) = $cos(p/(100000^(2i + 1/512)))$, produces matrix of size 512. Using 2 words as context produces $Z \in \mathbb{R}^{2 \times 512}$. Note that the entire block doesn't change X dims or anything else like that

Masking decoder self-attention masks future positions ($-\infty$ before softmax) so token t only attends to $\leq t$.

Why transformers beat RNNs

- Parallelizable across sequence (no sequential $h_{t-1} \rightarrow h_t$).
- $O(1)$ path length between any two positions (vs. $O(n)$ for RNN) $\Rightarrow$ long-range deps easier.
- Cost: $O(n^2 d)$ per layer in sequence length.

Worked Examples

P3. CNN shapes / params Fill activation shapes (H, W, C) and # params per layer. Input 32×32×3. *CONVx-N = conv with N filters of size x×x; POOL-n = n×n max-pool, stride n, padding 0. Default $p = 1, s = 1$ for conv unless stated.*

CONV3-N with $p = 1, s = 1$ preserves H, W and outputs N channels. POOL-2 halves H, W.

Layer	Shape	# params
Input	$(32, 32, 3)$	0
CONV3-8	$(32, 32, 8)$	$3{\cdot}3{\cdot}3{\cdot}8 + 8 = 224$
LeakyReLU	$(32, 32, 8)$	0
POOL-2	$(16, 16, 8)$	0
BatchNorm	$(16, 16, 8)$	$2{\cdot}8 = 16$
CONV3-16	$(16, 16, 16)$	$3{\cdot}3{\cdot}8{\cdot}16 + 16 = 1168$
LeakyReLU	$(16, 16, 16)$	0
POOL-2	$(8, 8, 16)$	0
FLATTEN	$(1024,)$	0
FC-10	$(10,)$	$1024{\cdot}10 + 10 = 10250$

Conv calculations

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}, K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, P = 1, S = 2$$

$$X_{\text{pad}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 0 \\ 0 & 5 & 6 & 7 & 8 & 0 \\ 0 & 9 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Position (0,0) — kernel covers X_{pad} rows 0–2, cols 0–2:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 5 & 6 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$= 0 + 0 - 0 + 0 + 0 - 2 + 0 + 0 - 6 = -8$$

Position (0,1) — slide right by stride 2; rows 0–2, cols 2–4:

$$\begin{bmatrix} 0 & 0 & 0 \\ 2 & 3 & 4 \\ 6 & 7 & 8 \end{bmatrix} \odot K$$

$$= 0 + 0 - 0 + 2 + 0 - 4 + 6 + 0 - 8 = -4$$

Position (1,0) — slide down by stride 2; rows 2–4, cols 0–2:

$$\begin{bmatrix} 0 & 5 & 6 \\ 0 & 9 & 1 \\ 0 & 4 & 5 \end{bmatrix} \odot K$$

$$= 0 + 0 - 6 + 0 + 0 - 1 + 0 + 0 - 5 = -12$$

Position (1,1) — rows 2–4, cols 2–4:

$$\begin{bmatrix} 6 & 7 & 8 \\ 1 & 2 & 3 \\ 5 & 6 & 7 \end{bmatrix} \odot K$$

$$= 6 + 0 - 8 + 1 + 0 - 3 + 5 + 0 - 7 = -6$$

$$Y = \begin{bmatrix} -8 & -4 \\ -12 & -6 \end{bmatrix}$$

Random formulas

Perceptron: Only if linearly seperable, $\hat{y} = w^T x + b$. If $\hat{y}y \leq 0, w = w + yx, b = b + y$

Linloss = $L = \|Xw - Y\|_2^2, w = (X^T X)^{-1} X^T Y$

RAGs: Technique that improves a language model's output by fetching external info before generating a response. Retrieval: Searches a knowledge base for relevant text. Generation: Feeds retrieved text into LM such as GPT. Benefits: Access custom/proprietary data, reduces hallucination, enables factual, domain-specific responses, no fine-tuning. Challenges: Requires good chunking/embedding of docs, good retrieval strategy, retrieved context must be relevant and concise.

RL: Learn through sequential trial error, RLHF if human-in-the-loop.

Active learning: algo with a human-in-the-loop

Semi: need clean labels, but only a portion needs to be labeled

Weakly: noisy, wrong labels that are still task-related

Self-supervised: learn from patterns obvious in data, ie text colorization