

Word Embeddings & NLP
Word Vector Encodings

Notation: x(i)=example i, |V|=vocab size, T=seq length, d=embedding dim.
One-hot: Length = |V|. Single 1 at word index, rest 0s. No semantic relationships. Dot product = 0. Sparse, high-dim, no OOV.
GloVe: Length = 300 (or 50,100,200). Dense, captures semantics via co-occurrence. Similar words -> similar vectors. Pre-trained, enables transfer learning.
Ex: |V| = 5000 -> One-hot: 5000, GloVe: 300.
Memory: 5000 words: one-hot 5000 x 5000 = 25M vs GloVe 5000 x 300 = 1.5M.
Embeddings: Dense vectors (50-300 dim); capture semantic relationships; learned from large text corpora.

Embedding Matrix: E in R^{d x |V|}. d = embedding dim; |V| = vocab size; Column j = embedding for word j; e_w = E . o_w where o_w is one-hot.
Lookup: For word 427, E . o_427 extracts column 427. Dims: (d x |V|) . (|V| x 1) = (d x 1).
Transfer Learning: (1) Learn embeddings from large corpus; (2) Transfer to new task; (3) Fine-tune (optional).

Cosine Similarity: cos(a,b) = (a.b) / (||a||.||b||) = (sum a_i b_i) / (sqrt(sum a_i^2) sqrt(sum b_i^2)). Range: [-1,1]. 1=same
dir, 0=orthog, -1=opposite. Use: word similarity, document matching, face verification thresholds.
Distance Metrics: Euclidean (L2): ||a-b||_2 = sqrt(sum (a_i - b_i)^2); Manhattan (L1): ||a-b||_1 = sum |a_i - b_i|; Cosine distance: 1 - cos(a,b).

Word Analogies
Task: Man is to Woman as King is to ?; London is to UK as Beijing is to ?.
Method: Find word w that maximizes similarity; argmax_w cos(e_king, e_king - e_man + e_woman); e_man - e_woman approx e_king - e_queen (cosine sim typically 0.7-0.9).
Intuition: Embeddings encode relationships; Directions in space have meaning.

Word2Vec
Two Algorithms: Skip-gram: predict context from center word (common); CBOW: predict center word from context (faster).
Skip-gram: Given center word w_t, predict context words w_{t+j}; Context: words within window of size c; Window size: typically 5-10 (window 5 means 5 words on each side, total context = 10 words); Each center-context pair is a training example.

Architecture: Input: one-hot vector for center word in R^{|V|}. Hidden: embedding layer E in R^{d x |V|} (Critical: No activation, linear lookup only); v_c = E . one-hot(c). Output: softmax over vocabulary.
Objective: Max log-likelihood: max 1/T sum_{t=1}^T sum_{j in [-c,c], j!=0} log P(w_{t+j}|w_t).
Softmax:

P(o|c) = (exp(u_o^T v_c) / sum_{v in V} exp(u_o^T v_c)). v_c = embed of center word; u_o = embed of output/context word.
Note: Two embedding matrices v (input/center, kept) and u (output/context, discarded). Final: use v_c or average both.

Problem: Softmax requires sum over all |V| words (expensive); For vocab of 1M words: 1M computations per example; Solution: Negative Sampling.
Negative Sampling: Don't compute full softmax; Convert to binary classification; For each positive pair, sample k negative words; Train: context word vs. random words.
Objective: log sigma(u_o^T v_c) + sum_{i=1}^k E_{w_i ~ P_noise} [log sigma(-u_w^T v_c)]. First term: maximize prob of actual context word; Second term: minimize prob of k random words; sigma = sigmoid; Binary cross-entropy loss.
Process: (1) Pick positive pair (c, o): center and actual context (label = 1); (2) Sample k negative words from P_noise (label = 0); (3) Compute binary loss for positive + k negatives; (4) Much faster: only k + 1 computations vs |V| for softmax.

Noise Distribution: P(w_i) = (f(w_i)^{3/4}) / (sum_j f(w_j)^{3/4}). f(w_i) = unigram frequency of word w_i. Power 3/4: balances common/rare words.
Ex: word with freq 100 vs freq 10. Ratio: uniform=10:1, sqrt=3.16:1, 3/4=5.6:1 (balanced). Uniform (f^1): too many common words. Square root (f^{1/2}): still imbalanced. Power 3/4: empirically works best. Typical k: 5-20 for small datasets, 2-5 for large.
Tradeoff: Higher k = slower but better gradients; lower k = faster but noisier.
Speedup: Training time reduced from O(|V|) to O(k) per example!

Convolutional Neural Networks (CNNs)
CNN Fundamentals
Notation: x^{[l]}=layer l, n=spatial size, d=feature dim, *=element-wise.
sigma=sigmoid/activation, g=activation func.
Activations: sigma(x) = 1 / (1 + exp(-x)) (sig-moid, range [0,1]); tanh(x) = (exp(x)-exp(-x)) / (exp(x)+exp(-x)) (range [-1,1]); ReLU(x) = max(0,x); Leaky ReLU(x) = max(0.01x,x); Softmax_i(x) = (exp^x_i / sum_j exp^x_j).
Losses: Binary cross-entropy: -[y log y + (1-y) log(1-y)]; Categorical cross-entropy: -sum_i y_i log y_i; MSE: 1/2 sum (y - y-hat)^2; MAE: 1/n sum |y - y-hat|.
Why CV is Hard: 64 x 64 x 3 image = 12,288 pixels. First FC layer: 64 neurons x 12,288 input = 786,432 params! 1000 x 1000 x 3 = 3M pixels.
Conv Solution: 64 filters of 3 x 3 x 3 = only 64 x 27 = 1,728 params, reused across entire image. Shares params, learns spatial hierarchies.
Conv Output Size: n_out = floor((n_in + 2p - f) / s) + 1. n_in=input, p=padding, f=filter, s=stride.
Ex: 512 x 512, f = 3, p = 1, s = 2: floor((512 + 2 - 3) / 2 + 1) = 256.
Patterns: s = 1, p = 0: shrink by f - 1. s = 2: half.

Edge Detection: CNN learns filter params via backprop. Sobel: [-1 0 1; -2 0 2; -1 0 1] (vertical).
Scharrr: [-3 0 3; -10 0 10; -3 0 3] (better - stronger central weighting). Transpose detects horizontal.
Why Padding: (1) Prevents shrinking: n x n -> (n - f + 1) x (n - f + 1) shrinks each layer. Deep nets need padding. (2) Edge pixels: without padding, corners used 1x vs center used f^2 times -> edges underrepresented.

Regularization: L2 (Ridge): L = L_data + lambda/2 sum w_i^2 (penalizes large weights, smooth).
L1 (Lasso): L = L_data + lambda sum |w_i| (drives weights to 0, sparse).
Dropout: During training: randomly set neurons to 0 with prob p (typically 0.5). Scale outputs by 1/(1 - p) to maintain expected value. At test: use all neurons (no dropout). Prevents co-adaptation of features.
Early Stopping: Monitor validation loss. Stop when validation loss stops improving.
Data Aug: Create variations (flip, rotate, crop). Expands effective dataset size.
Batch Normalization: Normalize batch: x-hat = (x - mu_B) / sqrt(sigma_B^2 + epsilon) where mu_B = 1/m sum x_i, sigma_B^2 = 1/m sum (x_i - mu_B)^2. Then scale/shift: y = gamma-hat x + beta-hat. Learnable gamma, beta. Benefits: faster training, higher LR, reduces internal covariate shift. At test: use running avg of mu, sigma from training.
Optimizers: SGD: w = w - alpha grad L (simple, noisy). Momentum: v = beta v + (1 - beta) grad L; w = w - alpha (smooth updates, beta approx 0.9). RMSprop: s = beta s + (1 - beta) (grad L)^2; w = w - alpha sqrt(s / (sqrt(s) + epsilon)) (adaptive LR, beta approx 0.999). Adam: Combines momentum + RMSprop: m = beta_1 m + (1 - beta_1) grad L; v = beta_2 v + (1 - beta_2) (grad L)^2; m-hat = m / (1 - beta_1^t), v-hat = v / (1 - beta_2^t) (bias correction); w = w - alpha sqrt(v-hat / (sqrt(v-hat) + epsilon)) (most popular, beta_1 = 0.9, beta_2 = 0.999, alpha = 0.001).
LR Schedules: Step decay: alpha = alpha_0 . gamma^floor(epoch / k) (e.g., gamma = 0.5, k = 10: halve every 10 epochs). Exp decay: alpha = alpha_0 exp(-kt). 1/t decay: alpha = alpha_0 / (1 + kt). Cosine annealing: alpha_t = alpha_min + 1/2 (alpha_max - alpha_min) (1 + cos(floor(pi * t / T))) (smooth decay, good for SGD).

CNN Components: Conv: Extracts features w/ learnable filters. Each filter: f x f x c_in params + 1 bias. Output channels = # filters. Preserves spatial structure via local connectivity & param sharing. Pooling: Reduces spatial dims. Max: max in window. Avg: mean in window. No learnable params. Provides translation invariance. Common: 2 x 2, s = 2 (halves dims). FC: Flattens spatial features. Every neuron to all inputs. Final classification. Params: weight matrix (n_in x n_out) + bias vector (n_out).
Conv Types: Valid: p = 0, n_out = n_in - f + 1. Same (s=1): Pad so n_out = n_in. p = (f - 1)/2. Requires f odd. Quick ref: f = 3 -> p = 1, f = 5 -> p = 2, f = 7 -> p = 3. Dilated Conv: Dilation rate r: insert r - 1 zeros between filter weights. Receptive field: (f - 1) . r + 1. Ex: f = 3, r = 2: receptive field = 2 . 2 + 1 = 5 (same as f = 5 but fewer params). Captures larger context without increasing params.
Layer l Notation: Input: n^{[l-1]} x n^{[l-1]} x c^{[l-1]}. Output: n^{[l]} x n^{[l]} x c^{[l]} where n^{[l]} = floor((n^{[l-1]} + 2p^{[l]} - f^{[l]}) / s^{[l]} + 1). Filters: f^{[l]} x f^{[l]} x c^{[l-1]} (per filter). # filters = s^{[l]} floor(f^{[l]} x f^{[l]} x c^{[l-1]} x c^{[l-1]}. Bias: c^{[l]}. Receptive Field: Layer 1: f. Layer 2: f + (f - 1) . s. Layer l: f_l + sum_{i=1}^{l-1} (f_i - 1) . prod_{j=i+1}^l s_j.
Ex: Two 3 x 3 conv, s = 1: receptive field = 3 + (3 - 1) . 1 = 5. Equivalent to one 5 x 5 but fewer params: 2 . (3^2 . c^2) < 5^2 . c^2 when c > 2.5 (always true).

CNN Architectures
LeNet-5 (1998): MNIST digits (28 x 28). AlexNet (2012): Won ImageNet ILSVRC. GPU training breakthrough. VGG-16 (2014): 16 layers. Insight: Stack of small filters > large filter (2 x 3 x 3 = 5 x 5 receptive field but fewer params). ResNet (2015): Very deep nets (50, 101, 152 layers). Key innovation: Skip connections (residual blocks). Solves vanishing gradient in deep nets. Identity mapping allows gradient to flow directly. Bottleneck block: 1 x 1 conv (reduce dim) -> 3 x 3 conv -> 1 x 1 conv (restore dim). Reduces params: 1 . 1 . 256 . 64 + 3 . 3 . 64 . 64 + 1 . 1 . 64 . 256 = 69,632 vs direct 3 . 3 . 256 . 256 = 589,824 (88% reduction). Transfer Learning: Reuse pre-trained model for new task. Process: Download pre-trained. Replace final layer (1000-unit softmax to C-unit). Train on new task. Strategies: Feature Extractor: Freeze all, train final layer only. Small data (<1k). Fine-tune: Freeze early, train later. Moderate data (1k-100k). Lower LR for pre-trained (e.g., 0.001 vs 0.01 for new layer). Full: Pre-trained init, train all. Large data (>100k).

Object Detection
Sliding Windows: Naive: Slide window across image, run classifier each position, try multiple scales. Very expensive. Conv Impl: Replace FC w/ conv layers (FC with 400 units -> conv with 400 filters of size 1 x 1). Process entire image in 1 forward pass. Speedup: Instead of 100 windows x 100ms = 10s, process once in 100ms. Localization: Predict car brand + bounding box (bbox). 7 classes: other, Ford, GMC, Chevrolet, Toyota, Honda, Tesla. Y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, c_4, c_5, c_6, c_7]. p_c: obj presence (1=car, 0=no car). (b_x, b_y): center coords. (b_h, b_w): height/width normalized 0-1. c_1, ..., c_7: one-hot classes. If p_c = 0, ignore rest. Total: 1 + 4 + 7 = 12. Formula: 1 + 4 + C. Ex: [1, 0.5, 0.6, 0.3, 0.4, 0, 1, 0, 0, 0, 0, 0] = car at (0.5, 0.6), size (0.3, 0.4).
Ford. Loss: L(y, y-hat) = 1/2 (y1 - y1-hat)^2 + ... + 1/2 (y12 - y12-hat)^2 p_c = 1. If object present p_c = 0.

(p_c = 1): all components matter. If no object (p_c = 0): only p_c matters.
Landmark Detection: Extension of localization: instead of 4 bbox coords, output multiple specific (x, y) points. Output: Y = [p_c, l_1x, l_1y, l_2x, l_2y, ..., l_nx, l_ny] where n=# landmarks. Coords normalized 0-1. Ex: Face with 64 landmarks: Y in R^{129} (1 for p_c + 128 for coords). Used in Snapchat filters, pose estimation.
Loss: Same as localization. YOLO Dimensions: S x S grid, B anchors/cell. Anchors = pre-defined bbox shapes/aspect ratios. Each anchor: p_c (objectness) + 4 bbox coords + C class probs. Given: 3 x 3 grid (9 cells), 4 anchors, 6 regular classes (elephant, giraffe, zebra, tiger, lion, monkey; excl. background). Per anchor: 1 + 4 + 6 = 11 values. (b_x, b_y) rel to cell (offset 0-1), (b_h, b_w) rel to anchor (scale factors). dims = S^2 x B x (1 + 4 + C) = 3^2 x 4 x 11 = 396. Shape: (3, 3, 4, 11) or flat 396. Background inferred when all p_c values low. YOLO: You Only Look Once - single pass. Divide image into S x S grid. Each cell predicts bboxes + class probs. Output/cell: p_c (obj prob), (b_x, b_y) (center), (b_h, b_w) (dims), c_1, ..., c_n (classes). Advantages: Very fast (real-time), single end-to-end network, sees full image (better context = reduces background false positives). Anchor boxes: Pre-defined shapes. Multiple preds/cell. Handle overlapping objs, different aspect ratios.

Non-Max Suppression & IoU: IoU: Measure bbox overlap. IoU = Area of Intersection / Area of Union. Range: [0,1]. 1=perfect overlap, 0=no overlap. Non-Max Suppression: Eliminate duplicate detections. Algorithm: (1) Discard boxes with p_c < threshold (e.g., 0.6); (2) Pick box with highest p_c; (3) Discard boxes with IoU >= 0.5 with picked box; (4) Repeat steps 2-3 for remaining boxes. Apply per class separately. Purpose: YOLO outputs many bboxes/object. NMS keeps only best one. R-CNN Family: R-CNN: Selective search generates 2000 region proposals. Run CNN on each. SVM classifier. Slow. Fast R-CNN: Single CNN for full image. ROI pooling (extract cell c-hat) (fixed-size feature map). End-to-end training. Much faster (~10x speedup). Faster uses a for hidden state, c for cell (unlike GRU which uses c for both). R-CNN: Region Proposal Network (RPN) learns proposals. Shares conv features w/ Gamma = sigma(W_f[a<t-1>, x<t>]) + b_f (forget gate: decides what info to discard from

person. If d > tau: different. Works w/ 1 example/person. Siamese Network: Two identical networks (shared weights). Siamese (twin in Thai) = identical architecture. Both take image input, output embedding f(x) in R^{128} (typically 128-256 dim). Process: img1 -> f(x^{(1)}), img2 -> f(x^{(2)}) (same net). Distance: d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(2)})||^2. Similar faces -> similar encodings. Triplet Loss: Train on triplets (A,P,N): A=anchor (ref), P=positive (same person), N=negative (diff person). Goal: ||f(A) - f(P)||^2 + alpha <= ||f(A) - f(N)||^2. L(A,P,N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + alpha, 0). ||f(A) - f(P)||^2 = dist to pos (want small). ||f(A) - f(N)||^2 = dist to neg (want large). alpha=margin (0.2-0.5), ensures separation. max(..., 0) = hinge loss (0 when satisfied). Ex: ||f(A) - f(P)||^2 = 0.1, ||f(A) - f(N)||^2 = 0.8, alpha = 0.2: Loss = max(0.1 - 0.8 + 0.2, 0) = 0. Training: Need multi imgs/person. Choose hard triplets! Hard pos: diff pose/lighting. Hard neg: looks similar but diff person. Semi-hard: d(A,P) < d(A,N) < d(A,P) + alpha. Random triplets often satisfied (no learning). Batch hard: for each anchor in batch find hardest positive (furthest same-person) and hardest negative (closest different person) within that batch. Total loss: avg over all triplets. Face Verification Binary: Siamese net to embeddings. Feed diff to sigmoid. Out: prob same person (y^{(i)} - f^{(j)})^2 / (f^{(i)} + f^{(j)})^2.
Notation: x<t> = time t, T=seq length, n_x=input dim, n_a=hidden dim. Long-Range Dependencies: Ranking (worst to best): (1) RNN (least effective), (2) Bi-RNN, (3) GRU, (4) LSTM. Why RNN fails: Vanishing gradient (grad -> 0) or exploding (grad -> inf). Hidden state xW repeatedly. Eigenval(W) < 1: gradient vanishes exponentially (0.9100 approx 0); > 1: explodes (1.1100 approx 13780). Info from distant past lost. Solutions: LSTM: additive cell state c<t> = c<t-1> + new (addition preserves gradients), 3 gates (forget,input,output), maintains 100+ steps. GRU: simpler, 2 gates (reset,update), fewer params. Bi-RNN: process fwd+pwd, needs full seq. Transformers > LSTM via attention. RNN Parameters (NER): a<t> = g_1(w_a[a<t-1>, x<t>] + b_a); y-hat<t> = g_2(w_y[a<t> + b_y]). g_1, g_2 = activation functions (typically tanh, sigmoid). Why RNN: Variable input/output lengths. Parameter sharing across time steps (same w_a, w_y for all t). Loss: L<t> = -y<t> log y<t> - (1 - y<t>) log(1 - y<t>). Total: L = sum_{t=1}^T L<t>. Given n_x = 5000, n_a = 128: x in R^{5000}, a in R^{128}, y in R (scalar NER). [a,x] in R^{5128} (concatenation). Count: w_a in R^{128 x 5128} (output x input dims): 656,384. w_y in R^{128}: 128. b_a, b_y: 1 each. Total: 656384 + 128 + 1 + 1 = 656,514. Formula: n_a(n_a + n_x) + n_a + 2. Params shared across all timesteps t = 1, ..., T. RNN Architectures & Language Modeling Many-to-Many (Equal): T_x = T_y. Ex: NER. Each input word to classification Many-to-Many (Diff): Encoder-Decoder. Encoder reads input sequence into single vector (context). Decoder expands context into output sequence. Ex: Machine translation. T_x != T_y. Many-to-One: Seq input to single output. Ex: Sentiment analysis. One-to-Many: Single input to seq output. Ex: Music gen. Feed output back as next input (y<t-1> becomes x<t>); teacher forcing during training (use ground truth), generated output during inference (use model's predictions). Language Modeling: Estimate P(y<1>, ..., y<T_y>). Predict next word given previous. P(y<1>, ..., y<T_y>) = P(y<1>) x P(y<2>|y<1>) x ... x P(y<T_y>|y<1>, ..., y<T-1>). Training: Words w/ zero vector or <EOS> token. Sample from y-hat<1>. Feed sampled as next input. Continue until <EOS> or max len (e.g., 50 words). Beam Search: At each step, keep k most likely candidates (beam width k, typically 3-10). Greedy search: k = 1 (fast but suboptimal). Large k: better quality, slower. Normalize by length: 1/T_x sum_{t=1}^T log P(y<t>|x, y<1:t-1>) where a approx 0.7 (length penalty, prevents favoring short sequences). Gradient Clipping: Prevents exploding gradients in RNNs. Clip by value: g = max(min(g, threshold), -threshold). Clip by norm: If ||g|| > threshold: g = g-threshold / ||g|| (scales down). Typical thresh old: 5-10. BLEU Score: Evaluates machine translation quality. Precision-based metric. BLEU = BP . exp(sum_{n=1}^N w_n log p_n) where p_n = modified n-gram precision, BP = min(1, e^{1-r/c}) (brevity penalty, r=ref length, c=candidate length), typically N = 4, w_n = 1/N. Range: [0,1]. Higher = better. Perplexity: PP = 2^{-1/N} sum log_2 P(w_i) = exp(L) where L is cross-entropy loss. Lower = better. Measures avg branching factor.

GRU & LSTM
GRU: Address vanishing grad. Simpler than LSTM (2 gates, faster). Often approx LSTM performance. Gamma = sigma(W_gamma[c<t-1>, x<t>] + b_gamma); Gamma_u = sigma(W_u[c<t-1>, x<t>] + b_u); c-hat<t> = tanh(W_c[Gamma_gamma * c<t-1>, x<t>] + b_c); c<t> = Gamma_u * c-hat<t> + (1 - Gamma_u) * c<t-1>. * = element-wise. Reset Gamma_r = 0? ignore past, s=1 use past. Update Gamma_u = 1? new, s=0 keep old. Acts like forget+input combined. Gates allow grad flow, maintain info many timesteps. Additive update: c-hat<t> = Gamma_u * c-hat<t> + (1 - Gamma_u) * c<t-1> provides path for gradient to flow unchanged (like ResNet), prevents vanishing. GRU Parameters: Given n_x (input dim), n_h (hidden dim). W_r, W_u, W_c in R^{n_h x (n_h + n_x + 1)}. Ex: n_x = 100, n_h = 128: 3 . 128 . (128 + 100 + 1) = 87,936 params. LSTM: Separate cell c-hat<t> (long-term memory) from hidden a<t> (short-term output). Note: LSTM uses a for hidden state, c for cell (unlike GRU which uses c for both). Equations: c-hat<t> = sigma(W_f[a<t-1>, x<t>]) + b_f (forget gate: decides what info to discard from cell); Gamma_u = sigma(W_u[a<t-1>, x<t>] + b_u) (input/update gate: decides what new info to store); Gamma_o = sigma(W_o[a<t-1>, x<t>] + b_o) (output gate: decides what to output based on cell); c-hat<t> = tanh(W_c[a<t-1>, x<t>] + b_c) (candidate cell state: new info to potentially add); c<t> = Gamma_f * c<t-1> + Gamma_u * c-hat<t> (cell update: forget old x keep fraction + add new x input fraction); a<t> = Gamma_o * tanh(c<t>) (hidden state output: filtered version of cell). All gates in (0,1) via sigma (near 0 = close, near 1 = open). Why works: Cell has additive updates (not multiplicative). Additive = gradient of 1 preserved dur-

cal). Scharrr: [-3 0 3; -10 0 10; -3 0 3] (better - stronger central weighting). Transpose detects horizontal.
Why Padding: (1) Prevents shrinking: n x n -> (n - f + 1) x (n - f + 1) shrinks each layer. Deep nets need padding. (2) Edge pixels: without padding, corners used 1x vs center used f^2 times -> edges underrepresented.
Regularization: L2 (Ridge): L = L_data + lambda/2 sum w_i^2 (penalizes large weights, smooth).
L1 (Lasso): L = L_data + lambda sum |w_i| (drives weights to 0, sparse).
Dropout: During training: randomly set neurons to 0 with prob p (typically 0.5). Scale outputs by 1/(1 - p) to maintain expected value. At test: use all neurons (no dropout). Prevents co-adaptation of features.
Early Stopping: Monitor validation loss. Stop when validation loss stops improving.
Data Aug: Create variations (flip, rotate, crop). Expands effective dataset size.
Batch Normalization: Normalize batch: x-hat = (x - mu_B) / sqrt(sigma_B^2 + epsilon) where mu_B = 1/m sum x_i, sigma_B^2 = 1/m sum (x_i - mu_B)^2. Then scale/shift: y = gamma-hat x + beta-hat. Learnable gamma, beta. Benefits: faster training, higher LR, reduces internal covariate shift. At test: use running avg of mu, sigma from training.
Optimizers: SGD: w = w - alpha grad L (simple, noisy). Momentum: v = beta v + (1 - beta) grad L; w = w - alpha (smooth updates, beta approx 0.9). RMSprop: s = beta s + (1 - beta) (grad L)^2; w = w - alpha sqrt(s / (sqrt(s) + epsilon)) (adaptive LR, beta approx 0.999). Adam: Combines momentum + RMSprop: m = beta_1 m + (1 - beta_1) grad L; v = beta_2 v + (1 - beta_2) (grad L)^2; m-hat = m / (1 - beta_1^t), v-hat = v / (1 - beta_2^t) (bias correction); w = w - alpha sqrt(v-hat / (sqrt(v-hat) + epsilon)) (most popular, beta_1 = 0.9, beta_2 = 0.999, alpha = 0.001).
LR Schedules: Step decay: alpha = alpha_0 . gamma^floor(epoch / k) (e.g., gamma = 0.5, k = 10: halve every 10 epochs). Exp decay: alpha = alpha_0 exp(-kt). 1/t decay: alpha = alpha_0 / (1 + kt). Cosine annealing: alpha_t = alpha_min + 1/2 (alpha_max - alpha_min) (1 + cos(floor(pi * t / T))) (smooth decay, good for SGD).

Object Detection
Sliding Windows: Naive: Slide window across image, run classifier each position, try multiple scales. Very expensive. Conv Impl: Replace FC w/ conv layers (FC with 400 units -> conv with 400 filters of size 1 x 1). Process entire image in 1 forward pass. Speedup: Instead of 100 windows x 100ms = 10s, process once in 100ms. Localization: Predict car brand + bounding box (bbox). 7 classes: other, Ford, GMC, Chevrolet, Toyota, Honda, Tesla. Y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, c_4, c_5, c_6, c_7]. p_c: obj presence (1=car, 0=no car). (b_x, b_y): center coords. (b_h, b_w): height/width normalized 0-1. c_1, ..., c_7: one-hot classes. If p_c = 0, ignore rest. Total: 1 + 4 + 7 = 12. Formula: 1 + 4 + C. Ex: [1, 0.5, 0.6, 0.3, 0.4, 0, 1, 0, 0, 0, 0, 0] = car at (0.5, 0.6), size (0.3, 0.4).
Ford. Loss: L(y, y-hat) = 1/2 (y1 - y1-hat)^2 + ... + 1/2 (y12 - y12-hat)^2 p_c = 1. If object present p_c = 0.

(p_c = 1): all components matter. If no object (p_c = 0): only p_c matters.
Landmark Detection: Extension of localization: instead of 4 bbox coords, output multiple specific (x, y) points. Output: Y = [p_c, l_1x, l_1y, l_2x, l_2y, ..., l_nx, l_ny] where n=# landmarks. Coords normalized 0-1. Ex: Face with 64 landmarks: Y in R^{129} (1 for p_c + 128 for coords). Used in Snapchat filters, pose estimation.
Loss: Same as localization. YOLO Dimensions: S x S grid, B anchors/cell. Anchors = pre-defined bbox shapes/aspect ratios. Each anchor: p_c (objectness) + 4 bbox coords + C class probs. Given: 3 x 3 grid (9 cells), 4 anchors, 6 regular classes (elephant, giraffe, zebra, tiger, lion, monkey; excl. background). Per anchor: 1 + 4 + 6 = 11 values. (b_x, b_y) rel to cell (offset 0-1), (b_h, b_w) rel to anchor (scale factors). dims = S^2 x B x (1 + 4 + C) = 3^2 x 4 x 11 = 396. Shape: (3, 3, 4, 11) or flat 396. Background inferred when all p_c values low. YOLO: You Only Look Once - single pass. Divide image into S x S grid. Each cell predicts bboxes + class probs. Output/cell: p_c (obj prob), (b_x, b_y) (center), (b_h, b_w) (dims), c_1, ..., c_n (classes). Advantages: Very fast (real-time), single end-to-end network, sees full image (better context = reduces background false positives). Anchor boxes: Pre-defined shapes. Multiple preds/cell. Handle overlapping objs, different aspect ratios.

Non-Max Suppression & IoU: IoU: Measure bbox overlap. IoU = Area of Intersection / Area of Union. Range: [0,1]. 1=perfect overlap, 0=no overlap. Non-Max Suppression: Eliminate duplicate detections. Algorithm: (1) Discard boxes with p_c < threshold (e.g., 0.6); (2) Pick box with highest p_c; (3) Discard boxes with IoU >= 0.5 with picked box; (4) Repeat steps 2-3 for remaining boxes. Apply per class separately. Purpose: YOLO outputs many bboxes/object. NMS keeps only best one. R-CNN Family: R-CNN: Selective search generates 2000 region proposals. Run CNN on each. SVM classifier. Slow. Fast R-CNN: Single CNN for full image. ROI pooling (extract cell c-hat) (fixed-size feature map). End-to-end training. Much faster (~10x speedup). Faster uses a for hidden state, c for cell (unlike GRU which uses c for both). R-CNN: Region Proposal Network (RPN) learns proposals. Shares conv features w/ Gamma = sigma(W_f[a<t-1>, x<t>]) + b_f (forget gate: decides what info to discard from

detector. Near real-time.
Face Recognition
Verification vs Recognition: Verification (1:1): Given: image + claimed ID. Q: store; O: sigma(W_o[a<t-1>, x<t>] + b_o) (output gate: decides what to output based on cell); c-hat<t> = tanh(W_c[a<t-1>, x<t>] + b_c) (candidate cell state: new info to potentially add); c<t> = Gamma_f * c<t-1> + Gamma_u * c-hat<t> (cell update: forget old x keep fraction + add new x input fraction); a<t> = Gamma_o * tanh(c<t>) (hidden state output: filtered version of cell). All gates in (0,1) via sigma (near 0 = close, near 1 = open). Why works: Cell has additive updates (not multiplicative). Additive = gradient of 1 preserved dur-

cal). Scharrr: [-3 0 3; -10 0 10; -3 0 3] (better - stronger central weighting). Transpose detects horizontal.
Why Padding: (1) Prevents shrinking: n x n -> (n - f + 1) x (n - f + 1) shrinks each layer. Deep nets need padding. (2) Edge pixels: without padding, corners used 1x vs center used f^2 times -> edges underrepresented.
Regularization: L2 (Ridge): L = L_data + lambda/2 sum w_i^2 (penalizes large weights, smooth).
L1 (Lasso): L = L_data + lambda sum |w_i| (drives weights to 0, sparse).
Dropout: During training: randomly set neurons to 0 with prob p (typically 0.5). Scale outputs by 1/(1 - p) to maintain expected value. At test: use all neurons (no dropout). Prevents co-adaptation of features.
Early Stopping: Monitor validation loss. Stop when validation loss stops improving.
Data Aug: Create variations (flip, rotate, crop). Expands effective dataset size.
Batch Normalization: Normalize batch: x-hat = (x - mu_B) / sqrt(sigma_B^2 + epsilon) where mu_B = 1/m sum x_i, sigma_B^2 = 1/m sum (x_i - mu_B)^2. Then scale/shift: y = gamma-hat x + beta-hat. Learnable gamma, beta. Benefits: faster training, higher LR, reduces internal covariate shift. At test: use running avg of mu, sigma from training.
Optimizers: SGD: w = w - alpha grad L (simple, noisy). Momentum: v = beta v + (1 - beta) grad L; w = w - alpha (smooth updates, beta approx 0.9). RMSprop: s = beta s + (1 - beta) (grad L)^2; w = w - alpha sqrt(s / (sqrt(s) + epsilon)) (adaptive LR, beta approx 0.999). Adam: Combines momentum + RMSprop: m = beta_1 m + (1 - beta_1) grad L; v = beta_2 v + (1 - beta_2) (grad L)^2; m-hat = m / (1 - beta_1^t), v-hat = v / (1 - beta_2^t) (bias correction); w = w - alpha sqrt(v-hat / (sqrt(v-hat) + epsilon)) (most popular, beta_1 = 0.9, beta_2 = 0.999, alpha = 0.001).
LR Schedules: Step decay: alpha = alpha_0 . gamma^floor(epoch / k) (e.g., gamma = 0.5, k = 10: halve every 10 epochs). Exp decay: alpha = alpha_0 exp(-kt). 1/t decay: alpha = alpha_0 / (1 + kt). Cosine annealing: alpha_t = alpha_min + 1/2 (alpha_max - alpha_min) (1 + cos(floor(pi * t / T))) (smooth decay, good for SGD).

Cell has additive updates (not multiplicative). Additive = gradient of 1 preserved dur-

(vs multiplication compounds: $(0.9)^{100} \approx 0$). **Advantages:** Easy to train. **Disadvantages:** Slow. **Implementation:** Add mask to attention scores; Set future positions to $-\infty$; After $3n_h(n_h + n_x + 1)$ (33% more params, slower but more expressive). **Peephole LSTM:** softmax, these become 0.

Gates also depend on cell state: $G_f = \sigma(W_f[a<t-1>, x<t>, c<t-1>] + b_f)$ (adds $c<t-1>$ to input). Better for tasks requiring precise timing.

Bidirectional RNN

Motivation: Standard RNN only uses past context; Many tasks need future context too; Example: "Teddy bears are on sale" vs "Teddy Roosevelt". **Architecture:** Two RNN layers; Forward RNN: left to right; Backward RNN: right to left; Combine both representations. **Forward:** $\vec{a}<t> = g(W_{\vec{a}}[\vec{a}<t-1>, x<t>] + b_{\vec{a}})$. **Backward:** $\overleftarrow{a}<t> = g(W_{\overleftarrow{a}}[\overleftarrow{a}<t+1>, x<t>] + b_{\overleftarrow{a}})$. **Output:** $\hat{y}<t> = g(W_y[\vec{a}<t>, \overleftarrow{a}<t>] + b_y)$. If $\vec{a}, \overleftarrow{a} \in \mathbb{R}^n$, then concat $\in \mathbb{R}^{2n}$. **Disadvantage:** Need entire sequence before processing; Not suitable for real-time applications.

Transformers & Attention Mechanisms

Self-Attention

Motivation: RNN processes sequentially (slow, no parallelization); Hard to capture long-range dependencies (vanishing gradient); Self-attention: parallel processing of entire sequence; Computational complexity: $O(n^2 \cdot d)$ vs RNN's $O(n \cdot d^2)$ where n =seq length, d =hidden dim; **Ex:** $n = 1000, d = 512$: attention= $1000^2 \times 512 \approx 512M$ vs RNN= $1000 \times 512^2 \approx 262M$. For short sequences ($n < 512$), attention wins!

Components & Queries, Keys, Values: Input: $X \in \mathbb{R}^{n \times d}$ (n tokens, d dim each); $Q = XW^Q$ where $W^Q \in \mathbb{R}^{d \times d_k}$; $K = XW^K$ where $W^K \in \mathbb{R}^{d \times d_k}$; $V = XW^V$ where $W^V \in \mathbb{R}^{d \times d_v}$; Usually: $d_k = d_v = d/h$ where h = num heads.

Attention Formula: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$.

Process (step-by-step): (1) Compute $QK^T \in \mathbb{R}^{n \times n}$: similarity matrix (all pairs). (2) Scale by $\sqrt{d_k}$: prevents saturation of softmax. Why? Dot products grow with dimension. **Ex:** Without scaling, if $d_k = 64$ and values are unit normal, QK^T has std $\approx \sqrt{64} = 8$. Softmax on values with std 8 \rightarrow nearly one-hot (saturated), gradient ≈ 0 . (3) Softmax (row-wise): normalize to probability distributions. Each row sums to 1 (attention weights). (4) Multiply by V : weighted sum of values. Output: $\in \mathbb{R}^{n \times d_v}$. **Intuition:** **Query:** what am I looking for? (from current position); **Key:** what do I contain? (from all positions); **Value:** what do I actually pass on? (content); Each position attends to all positions simultaneously; Attention weights show which positions are relevant.

Example: The animal didn't cross the street because it was too tired" Query "it" attends strongly to "animal" (high attention weight); Learns contextual relationships automatically.

Multi-Head Attention

Motivation: Single attention limited to one representation subspace; Different heads learn different patterns/relationships; **Ex patterns:** Head 1 learns syntax (subject-verb), Head 2 learns semantics (word meaning), Head 3 learns position (nearby words), Head 4 learns coreference (pronoun-noun); Allows attending to different aspects simultaneously.

Formula: $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$.

Each head: $\text{head}_i = \text{Attn}(QW_i^Q, KW_i^K, VW_i^V)$.

Dimensions: h = number of heads (typically 8 in Transformer); d_{model} = model dimension (typically 512); Each head dimension: $d_k = d_v = d_{model}/h$ (e.g., $512/8 = 64$); $W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ for each head i ; $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ for each head i ; $W^O \in \mathbb{R}^{hd_v \times d_{model}}$: final output projection.

Process: (1) Split: project Q,K,V into h heads (each d_k dim). (2) Compute: attention for each head in parallel. (3) Concatenate: stack all head outputs. (4) Project: linear layer W^O to get final output.

Benefits: Parallel computation across heads; Different representation subspaces learned; Richer, more robust representations; Same total parameters as single-head with full dimension.

Key insight: h heads of size d_k vs 1 head of size d_{model} has same computation but better performance! **Computation:** h heads $\times d_k^2$ each $= h \times (d_{model}/h)^2 = d_{model}^2/h$ vs single head: d_{model}^2 . With h parallel, wall-clock time similar but captures h different patterns!

Positional Encoding & Layer Normalization

Positional Encoding:

Problem: Self-attention has no notion of position; Order of words matters in language; Need to inject positional information. **Solution:** Add positional encoding to input embeddings. **Formula:** $PE_{(pos, 2i)} = \sin(\frac{pos}{10000^{2i/d_{model}}})$; Where: pos = position in sequence; i =

dimension index; d_{model} = embedding dimension.

Properties: Unique encoding for each position; Model can learn relative positions; Sinusoidal pattern allows extrapolation (continues smoothly beyond training sequence length, allowing model to handle longer sequences at test time); Same dimension as word embeddings. **Ex:** For $pos = 0, i = 0, d_{model} = 512$: $PE_{(0,0)} = \sin(0/10000^0) = 0$.

For $pos = 1, i = 0$: $PE_{(1,0)} = \sin(1/10000^0) = \sin(1) \approx 0.84$.

Layer Normalization:

Purpose: Stabilize training of deep networks; Normalize across features for each example; Used in Transformers.

Formula: $\text{LayerNorm}(x) = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$. Where: $\mu = \frac{1}{d} \sum_{i=1}^d x_i$ (mean);

$\sigma^2 = \frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2$ (variance); γ, β = learned parameters; ϵ = small constant for stability (typically 10^{-5}), prevents division by zero when $\sigma^2 \approx 0$ (all features equal).

Difference from Batch Norm: Batch Norm: normalizes across batch; Layer Norm: normalizes across features; Layer Norm works better for sequences; No batch dependence = works with batch_size = 1, behavior identical in training/inference (unlike BatchNorm where test uses running stats).

Transformer Architecture

Masked Multi-Head Attention:

Purpose: Used in decoder during training; Prevent attending to future positions; Maintain autoregressive property (each output depends only on previous outputs, like lan-

guage modeling: predict word t using words $1..t-1$).

Implementation: Add mask to attention scores; Set future positions to $-\infty$; After $3n_h(n_h + n_x + 1)$ (33% more params, slower but more expressive). **Peephole LSTM:** softmax, these become 0.

Masking: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + M)V$ where $M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$.

Ex (3 words): $M = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}$. Word 1 sees only itself, word 2 sees 1-2, word 3 sees all.

Why Needed: At position t , can only use positions $\leq t$; Simulates sequential generation at inference; Prevents "cheating" during training.

Transformer Encoder-Decoder:

Encoder: Stack of N identical layers (typically 6). Each layer has two sub-layers: (1) Multi-head self-attention, (2) Position-wise feed-forward network. Residual connection + layer norm around each (output = LayerNorm(x + Sublayer(x)), allows gradient flow).

Decoder: Stack of N identical layers. Each layer has three sub-layers: (1) Masked multi-head self-attention, (2) Multi-head attention over encoder output, (3) Position-wise feed-forward network. Residual connection + layer norm around each.

Cross-Attention: Queries from decoder; Keys and Values from encoder; Decoder attends to encoder output.

Feed-Forward Network: $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$. Two linear layers with ReLU between them. Same W_1, W_2 applied to each position independently (position-wise = process each token separately, like 1×1 conv). Different W_1, W_2 for each encoder layer / decoder layer. Typically expands dimension then contracts: $d_{model} \rightarrow 4d_{model} \rightarrow d_{model}$ (e.g., $512 \rightarrow 2048 \rightarrow 512$).

Graph Neural Networks (GNNs)

Graph Fundamentals

Notation: $G = (V, E)$, $N(v)$ =neighbors, $h^{(l)}$ =layer l embedding, $|V|$ =num nodes.

Graph Types:

Homogeneous Graph: Single node type; Single edge type; Example: Social network (all users).

Heterogeneous Graph: Multiple node types; Multiple edge types; Example: Citation network (papers, authors, venues).

Graph with Auxiliary Information: Nodes have features; Example: User profiles in social network.

Graph from Non-relational Data: Construct graph from data; Example: KNN (K-Nearest Neighbors) graph from feature vectors - connect each point to k closest points by feature distance; converts tabular data to graph for GNN processing.

Graph Proximity:

First-Order Proximity: Local pairwise similarity; Direct connection between nodes; Edge weight indicates similarity; $s_1(i, j) = w_{ij}$ where w_{ij} = edge weight (1 for unweighted graphs, or strength value).

Second-Order Proximity: Similarity of neighborhood structures; Nodes similar if neighbors are similar; Even without direct edge; $s_2(i, j)$ = similarity($N(i), N(j)$) = cosine similarity of neighborhood vectors, or Jaccard index: $|N(i) \cap N(j)| / |N(i) \cup N(j)|$.

Example: Two papers cite same references (2nd order); May not directly cite each other (no 1st order).

Homophily vs Structural Equivalence:

Homophily: "Birds of a feather flock together"; Connected nodes tend to be similar; Similar nodes have similar embeddings; Common in social networks; Example: Friends have similar interests.

Structural Equivalence: Nodes with similar structural roles; May not be directly connected; Similar position in network; Example: Two airport hubs (not connected but similar role).

Implications for Embeddings: Need to capture both properties; Homophily: local neighborhood; Structural: global network position.

Graph Embeddings

Graph Embedding Types:

Node Embedding: Map each node to vector; Preserve graph structure; Used for node classification, link prediction.

Edge Embedding:

Given node embeddings $f(u), f(v) \in \mathbb{R}^d$, edge embedding operators:

Average: $\frac{f(u)+f(v)}{2}$ (simple combination, symmetric, smooth); Hadamard: $f(u) * f(v)$ (element-wise product, captures feature interactions); Weighted-L1: $|f(u) - f(v)|$ (distance-based, emphasize differences for link prediction); Weighted-L2: $|f(u) - f(v)|^2$ (squared distance).

Subgraph Embedding: Represent substructures; Community detection; Motif finding.

Hybrid Embedding: Combine multiple embedding types; Example: node + edge + graph features; Richer representation.

Whole-Graph Embedding: Single vector per graph; Graph classification; Molecular property prediction. Methods: sum/mean/max pooling over node embeddings, or hierarchical pooling (e.g., DiffPool).

Random Walk Embeddings

DeepWalk: Random walks on graph; Treat walks as sentences; Apply Word2Vec; Uniform random walks.

node2Vec: Biased random walks; Two parameters: p and q ; p : return parameter (go back); q : in-out parameter (explore).

node2Vec Walk Strategy: p : return parameter (probability of returning to previous node; high p discourages backtracking); q : in-out parameter (ratio of exploring outward vs inward; low q favors BFS (Breadth-First Search)/stay local, high q favors DFS (Depth-First Search)/explore far). $q < 1$: BFS-like (local, homophily); $q > 1$: DFS-like (global, structural); Interpolate between both properties. **Ex:** $p = 1, q = 0.5$: local exploration; $p = 1, q = 2$: structural roles.

Objective: $\max_f \sum_{u \in V} \log P(N_S(u) | f(u))$. $N_S(u)$: neighborhood from random walks; $f(u)$: embedding of node u .

Negative Sampling: Approximate softmax normalization: $\log \sigma(\mathbf{z}_u^T \mathbf{z}_v)$ - $\sum_{i=1}^k \log \sigma(\mathbf{z}_u^T \mathbf{z}_{n_i})$ where $n_i \sim P_V$ (noise distribution over nodes, typically uniform or degree-based; random negative samples). Higher k = more robust.

GNN Aggregation & Architecture

GNN Aggregation Strategies:

Graph Notation: $G = (V, E, X)$ where $X \in \mathbb{R}^F \times |V|$ is node feature matrix; $N(v)$: neighbors of node v .

Why Not Naive Approach: Concatenating adjacency matrix + features into NN fails: $O(|V|)$ parameters; sensitive to node ordering; breaks when new nodes arrive.

Image vs Graph Convolution: Image: fixed grid, slide kernel. Graph: no ordering, aggregate neighbors. Image is special case of graph with fixed structure.

Basic Idea: $h_v^{(k)} = \sigma(W^{(k)} \cdot \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)}\}_{u \in N(v)}))$

Mean: $h_v^{(k)} = \sigma(W^{(k)} \cdot \text{MEAN}(\{h_u^{(k-1)}\}_{u \in N(v)}))$. **Sum:** $h_v^{(k)} = \sigma(W^{(k)} \cdot \sum_{u \in N(v)} h_u^{(k-1)})$. **Max:** $h_v^{(k)} = \sigma(W^{(k)} \cdot \max_{u \in N(v)} h_u^{(k-1)})$.

GraphSAGE: Includes self-connection: $h_v^{(k)} = \sigma(W^{(k)} \cdot \text{CONCAT}(h_v^{(k-1)}, \text{AGG}(\{h_u^{(k-1)}\}_{u \in N(v)})))$. **AGG** = any aggregation (MEAN, SUM, MAX, or learned LSTM aggregator). **GAT (Graph Attention Networks):** Learns importance of neighbors via attention. **Attention coef:** $e_{ij} = \text{LeakyReLU}(a^T [W_h i || W_h j])$ (measures importance of node j to node i , $||$ = concatenation). Normalize: $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ik})}$ (softmax over neighbors).

Aggregate: $h_i' = \sigma(\sum_{j \in N(i)} \alpha_{ij} W_j h_j)$ (weighted sum by attention). **Multi-head:** Like transformers, use K attention heads: $h_i' = ||_{k=1}^K \sigma(\sum_{j \in N(i)} \alpha_{ij}^k W^k h_j)$ (concatenate outputs). Benefits: different heads learn different neighborhood patterns. More expressive than fixed aggregation. **Inductive:** Attention weights computed on-the-fly generalizes to new nodes/graphs.

GCN Parameters: $h_v^{(l+1)} = \sigma(W_l [\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}])$. $h_v^{(l)} \in \mathbb{R}^d$, $N(v)$ =neighbors of node v , $W_l \in \mathbb{R}^{d \times d}$ (neighbor agg), $B_l \in \mathbb{R}^d$ (self-connection vector), σ =activ (e.g., ReLU). Per layer: W_l : d^2 params. B_l : d params (vector). Total/layer: $d^2 + d$. Total = $L \times (d^2 + d)$. **Ex:** $L = 3, d = 64$: $3 \times (64^2 + 64) = 3 \times 4160 = 12,480$.

Key: Params inside of graph size! Only depend on d, L . Shared across all nodes (inductive = generalizes to new unseen nodes without retraining).

GNN Depth and Receptive Field:

K-Layer GNN: Embedding uses K-hop neighborhood; Layer 1: immediate neighbors; Layer 2: 2-hop neighbors; Layer K: K-hop neighbors.

Receptive Field Growth: Grows exponentially with layers; Average degree d : receptive field $\sim d^K$; Deep GNNs can be expensive.

Over-smoothing Problem: Too many layers: all nodes aggregate from entire graph \rightarrow all embeddings become similar (lose node identity); Embeddings become indistinguishable; Typical depth: 2-4 layers.

Solutions: Residual connections (add input to output); Jumping knowledge networks (concatenate all layer outputs); Layer-wise attention (weight different layers).

GNN Training & Learning

GNN Training:

Supervised Training: Direct labels for nodes/graphs; Classification: cross-entropy loss; Regression: MSE loss; Backprop through aggregation.

Loss for Node Classification:

$\mathcal{L} = - \sum_{v \in V_{\text{train}}} \sum_{c=1}^C y_{v,c} \log(\hat{y}_{v,c})$.

Unsupervised Training: No explicit labels; Preserve graph structure; Similar nodes similar embeddings.

Graph Reconstruction Loss: $\mathcal{L} = ||A - \hat{A}||_F^2$ where $|| \cdot ||_F$ = Frobenius norm = $\sqrt{\sum_{i,j} A_{ij}^2}$ (element-wise squared differences), \hat{A}_{ij} = decoder(z_i, z_j) = $\sigma(z_i^T z_j)$ (predict edge probability from node embeddings).

Contrastive Loss: Positive pairs: connected nodes; Negative pairs: random nodes; Maximize agreement for positives.

GNN Inductive Learning:

Problem: New nodes arrive after training; Cannot retrain entire model; Need to generalize to unseen nodes.

Solution: Share aggregation parameters across all nodes; Same $W^{(l)}$ for all nodes at layer l ; Generate embeddings "on-the-fly".

Process: (1) Define aggregation function (with parameters). (2) Define loss function on embeddings. (3) Train on batch of nodes. (4) Apply same function to new nodes.

Node-Level Inductive: Train on snapshot of graph; New node appears; Generate embedding using learned parameters.

Graph-Level Inductive: Train on one set of graphs; Generalize to completely new graphs; Example: Train on small molecules, test on large.

GNN Tasks & Evaluation

Graph Learning Tasks:

Node Classification: Predict category of node; Semi-supervised setting common; Example: Classify protein function.

Link Prediction: Predict if edge exists; Score: $\text{score}(u, v) = z_u^T z_v$ (dot product). Other variants: concatenate+MLP, Hadamard product, or learned similarity; Example: Friend recommendation.

Graph Classification: Classify entire graph; Need graph-level embedding; Pooling over node embeddings (global mean/max/sum over all nodes, or learnable hierarchical pooling like DiffPool, SAGPool); Example: Molecule property prediction.

Node Recommendation: Find top-k similar nodes; Use embedding similarity; Example: Product recommendation.

Visualization Techniques:

PCA (Principal Component Analysis): Linear dimensionality reduction; Preserves global structure; Projects to principal components; Fast but may lose local structure.

t-SNE: Non-linear dimensionality reduction; Preserves local neighborhoods; Better for visualization; Slower than PCA; Stochastic (uses random initialization); different runs give slightly different visualization (unlike deterministic PCA).

Guidance: Use PCA for quick exploration, large datasets. Use t-SNE for publication figures, understanding clusters (smaller datasets <10k points).

Use Cases: Visualize word embeddings; Check clustering quality; Understand learned representations; Debug embedding models.

Performance Measures:

Node Classification: Accuracy, F1-score; Precision, Recall; Confusion matrix.

Link Prediction: AUC-ROC: area under ROC curve (plots TPR=True Positive Rate vs FPR=False Positive Rate as threshold varies; 1.0=perfect, 0.5=random); AUC-PR: area under precision-recall curve; MRR: mean reciprocal rank = $\frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$ where

rank_i =position of first correct answer (e.g., if correct link is ranked 3rd, contributes 1/3); Hits@K: proportion in top K (fraction of queries where correct answer appears in top K results).

Graph Classification: Same as node classification; Cross-validation important; Small dataset: careful evaluation.

Embedding Quality: Reconstruction error; Downstream task performance; Visual inspection (t-SNE).