Activation Functions

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$, $\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x))$ Output: $(0,1)$; Not zero-centered; Saturates. Not for hidden layers. 4 FLOPS/element.

Tanh: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, $\frac{d\tanh}{dx} = 1 - \tanh^2(x)$ Output: $(-1,1)$; Zero-centered; Still saturates.

ReLU: $\max(0,x)$, $\frac{d}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$ Fast converge; No saturation; May 'die' if x < 0. 1 FLOP.

Leaky ReLU: $\max(\alpha x, x)$, α small (e.g., 0.01) Fixes dying ReLU problem.

ParamReLU: LeakyReLU, but α is learnable.

ELU: $\begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$, $\alpha > 0$ ReLU benefits + mean closer to zero. Computational expensive.

GELU: $x\Phi(x)$ where Φ is normal CDF. Used in transformers.

Softmax:$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ (For multi-class output) $3 \times N$ FLOPs (exp + sum + divide)

Loss Functions

SVM Loss (Hinge):
$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$
Δ = margin (usually 1). Wants correct class score higher by Δ.

Softmax Loss (Cross-Entropy):
$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$ Maximizes probability of correct class. Comparable to SVM.

Binary Cross-Entropy:
$L = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$

Regression:
MSE: $\sum_i (y_i - \hat{y}_i)^2$, MAE: $\sum_i |y_i - \hat{y}_i|$

Optimization

SGD: $W = W - \alpha\nabla_W L$ Simple but slow convergence and noisy updates.

SGD+Momentum:

$$v = \beta v + \alpha\nabla_W L$$
$$W = W - v$$

Dampens oscillations, faster convergence.

AdaGrad (adaptive gradient): slow, saddle pt

$$c = c + (\nabla_W L)^2 \qquad W = W - \alpha\frac{\nabla_W L}{\sqrt{c} + \epsilon}$$

Per-parameter LRs; accumulation can stop learn.

RMSProp:

$$c = \beta \cdot c + (1-\beta)(\nabla_W L)^2 \quad W = W - \alpha\frac{\nabla_W L}{\sqrt{c} + \epsilon}$$

Fixes AdaGrad's diminishing LRs by using exp moving avg instead of grad accumulation.

Adam:

$$m = \beta_1 m + (1-\beta_1)\nabla_W L \quad v = \beta_2 v + (1-\beta_2)(\nabla_W L)^2$$
$$\hat{m} = \frac{m}{1-\beta_1^t}; \quad \hat{v} = \frac{v}{1-\beta_2^t} \quad W = W - \alpha\frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

Combines 1st and 2nd momentum (RMSProp) and adaptive LRs. Adds bias corrections.

LR Schedules: Step decay: $\alpha_0 \cdot \gamma^{\lfloor t/s \rfloor}$ Exp decay: $\alpha_0 \cdot e^{-kt}$ or 1/t decay: $\frac{\alpha_0}{1+kt}$

Computational Graphs & Backprop

Local Gradients:

- Add: $\frac{\partial(x+y)}{\partial x} = 1$, $\frac{\partial(x+y)}{\partial y} = 1$ (Distributor)
- Multiply: $\frac{\partial(xy)}{\partial x} = y$, $\frac{\partial(xy)}{\partial y} = x$ (Switcher)
- ReLU: $\frac{\partial\max(0,x)}{\partial x} = \mathbb{I}(x > 0)$ (Router)
- Max: $\frac{\partial\max(x,y)}{\partial x} = \mathbb{I}(x > y)$ (Selector)

Grad Check: $\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}$, h is small

Regularization

L2 Reg: $R(W) = \frac{\lambda}{2}\sum W^2$. Grad: λW. Penalizes large weights; encourages diffuse weights.

L1 Reg: $R(W) = \lambda\sum|W|$. Grad: $\lambda \cdot \text{sign}(W)$. Encourages sparse weights (many exactly zero).

Elastic Net: $R(W) = \lambda_1\sum|W| + \lambda_2\sum W^2$

Dropout:
Zero rand activations with prob p during training.

- Scale remaining output: $\frac{1}{1-p}$ (inverted drop)
- No dropout at test time
- Prevents neurons co-adaptation; can be seen as an ensemble of neural nets.

Data Augmentation: Crops, flips, rotations, color jitter, mixup, cutout

Early Stopping: Stop when val loss increases. Can prevent overfitting.

Training Issues & Solutions

Gradient Problems

Vanishing Gradients: Causes: Sigmoid/tanh saturation, deep networks
Solutions: ReLU, residual connections, He/Xavier, batch norm, use LSTM or GRU

Exploding Gradients: Solution: Gradient clipping (if norm > threshold), good init, batch norm

Weight Initialization

Zero: Bad (neurons learn same features) **Small Random:** $W \sim \mathcal{N}(0, \sigma^2)$. Ok for small nets.
Xavier/Glorot: Good for tanh/sigmoid; preserves variance **He:** $W \sim \mathcal{N}(0, \sqrt{\frac{2}{n_{in}}})$ Better for ReLU; accounts for ReLU drop half activations

Troubleshooting

Loss not decreasing: Check gradien, adjust LR
Overfitting: ↑ regulariz, data augment, smaller model, early stop. **Underfitting:**
↑ model capacity, train longer, ↓ regularization
Loss explodes: LR too high, bad initialization
No initial progress: LR too low/high, bad init
First layer visualizations: Should show patterns (edges, blobs, textures).

Normalization Layers

Batch Normalization (BN)

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad y = \gamma\hat{x} + \beta$$

Trainable Params: $2 \times C$ (γ, β per channel)
For ConvNets: Normalizes per channel across (N,H,W) dims. **Test time:** Uses running avg of μ, σ^2 **Benefits:** Better grad flow, higher LRs, acts as regularizer, reduces sensitivity to initialization.

Layer Normalization (LN)

Normaliz across features for each sample. Consistent in train/test; effective in RNNs/Transformers

Instance Normalization

Normalizes across H×W per channel and sample.

Convolutional Neural Networks

Convolution Layer

Hyperparams:
K = # filters, F = filter size, S = stride, P = pad

Output Dimensions:

$$W_{out}(\text{or } H_{out}) = \frac{W_{in}(\text{or } H_{in}) - F + 2P}{S} + 1$$
$$D_{out} = K \qquad D_{in} = \#\text{channels input}$$

Output Volume: $W_{out} \cdot H_{out} \cdot K$
Parameters: $(F^2 \cdot D_{in} + 1) \cdot K$

FLOPs: $O(N \cdot F^2 \cdot D_{in} \cdot D_{out} \cdot H_{out} \cdot W_{out})$

1x1 Conv: Channel-wise dim reduction, adds nonlinearity

Depthwise Separable Conv:

- Depthwise: One filter per input channel
- Pointwise: 1x1 conv to mix channels
- Params: $F^2 D_{in} + D_{in} D_{out}$ vs. $F^2 D_{in} D_{out}$

Dilated Conv: Expands receptive field without ↑ params. $F_{eff} = F + (F-1)(d-1)$

Transposed Conv: For upsampling
$W_{out} = (W_{in} - 1) \cdot S + F - 2P$

Receptive Field r

Single layer: $r = F$ (kernel size)
Stacked layers:

$$r_l = r_{l-1} + (F_l - 1) \cdot \prod_{i=1}^{l-1} s_i$$

where $r_0 = 1$, F_l = kernel size at layer l, s_i = stride at layer i
Same stride/kernel: For L layers with same kernel F and same stride $s = 1$:

$$r = 1 + L \cdot (F-1)$$

Pooling Layer

F = pool size, S = stride (typically $S = F$)

Output dims: Same formula as conv with $P = 0$

Max Pooling: Takes max in window. Gradients flow only to max element.
FLOPS = $H_{out} \times W_{out} \times C$

Avg Pooling: Takes average in window. Gradients distributed. FLOPS = $H_{out} \times W_{out} \times C \times (k^2 - 1)$, for $k \times k$ kernel

Global Avg Pooling:
Averages over entire spatial dimensions.

Params: 0 (no learnable params)

Convolution Properties

- **Translation Equivariance:**
  Shift input → shift output
- **NOT Rotation/Scale Invariant**
  Requires data augmentation for this
- **Parameter Sharing:** Fewer params than FC
- **Local Connectiv:** Neurons see local regions
- **Hierarchical Features:**
  Edges → textures → patterns → objects

Parameter Count Formulas

FC: $(D_{in}+1) \times D_{out}$ and $O(N_{neuron} \cdot M_{input})$
FLOPS $= 2 \times N_{batch} \times N_{input} \times N_{output}$
with bias $(2 \times N_{input} - 1) \times N_{output} + N_{output}$

Conv2D: $(F_h \times F_w \times D_{in} + 1) \times D_{out}$
Conv3D: $(F_{depth} \times F_h \times F_w \times D_{in} + 1) \times D_{out}$
BN/LN: $2 \times C$ (γ, β trainable per channel. Otherwise $4 = 2$ trainable $+ 2$ non train)

CNN Architecture Tips

- Prefer stacks of small filters (e.g., 3×3) over one large filter. Deeper, more non-linearities, fewer parameters for same receptive field.
- Use stride 2 for downsampling (vs pooling)
- Common: [CONV-BN-RELU]·N-POOL
- FC layers have most params; use GlobAvgPooling before FC
- Increase channels as spatial dims decrease
- Normalize inputs (subtract mean, divide by std)

CNN Architectures (chronologic)

- **AlexNet**: 1st CNN (ImgNet). ReLu, drop
- **VGG** Simple 3×3 conv stacks. Depth matters
- **GoogLeNet/Inception**: Parallel paths, different filter size, pooling.
- **ResNet**: Skip connections $O_l = I_l + F(I_l)$ Solves vanishing gradient in deep networks
- **DenseNet**: Each layer connect to all previous

Backprop in Conv Layers

Gradient w.r.t. input $\frac{\partial L}{\partial X}$ is a full convolution with flipped kernel (180° rotated). **Gradient w.r.t. filters** $\frac{\partial L}{\partial F}$ is a convolution between input X and output gradient $\frac{\partial L}{\partial Y}$.

1D conv output (forward): $z_i = \sum_j k_j x_{i+j-1} + b$
$\frac{\partial L}{\partial k_j} = \sum_{i=1}^{W_{out}} \frac{\partial L}{\partial z_i} x_{i+j-1}$ $\frac{\partial L}{\partial b} = \sum_{i=1}^{W_{out}} \frac{\partial L}{\partial z_i}$

Transformer

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

Q, K, V are query, key, value project of input X
$\sqrt{d_k}$ is scaling factor (d_k = key vector K dim)
Embed: *vocab_size* $\times D_{embedding}$
Posit Encod *seq_length* $\times D_{embedding}$

Self-Attent: X is used for Q, K, V (same source)
Masked Attention: Sets future positions to $-\infty$ (decoders) before softmax

Multi-Head Attention (MHA):

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

- Per head h: $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{in} \times d_k}$ $d_k = \frac{d_{in}}{h}$
- Output projection: $W^O \in \mathbb{R}^{d_{in} \times d_{in}}$
- **Parms:** $4d_{in}^2 + 4d_{in}$ (incl. biases)
- $O(n^2 d_{in} + n d_{in}^2)$ for seq length n

1) Multi-Head Self-Att + Add&Norm

- Residual connection: LayerNorm$(x + \text{MHA}(x))$
- LayerNorm params: $2d_{in}$ (γ and β vectors)

2) Feed-Forward Network + Add&Norm

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- $W_1 : d_{in} \to d_{ff}$, $W_2 : d_{ff} \to d_{in}$
- Typically $d_{ff} = 4 \cdot d_{in}$
- **Parms:** $2d_{in} \cdot d_{ff} + d_{ff} + d_{in} \approx 8d_{in}^2$
- Residual: LayerNorm$(x + \text{FFN}(x))$
- LayerNorm params: $2d_{in}$

Summary per Transform Block (per layer):

Tot pars: $12d_{in}^2 + 13d_{in}$ (NO embed, pos encod)
 # matmuls: 4 from Self-attention + 2 from MLP
$O(L(n^2 d_{in} + n d_{in}^2))$ for L layers

Pre-Norm variant: LayerNorm placed before self-att modules inside residual connection, more stable than the original.

K-Nearest Neighbors (KNN)

Non-parametric lazy learning algorithm.

- Classify by majority vote K closest train exmpl.
- Higher K: smoother decision boundary, more robust to outliers.
- Distance Metrics:
  - L1 (Manhattan): $d(I_1, I_2) = \sum_p |I_{1p} - I_{2p}|$ (Sensitive to coordinate system rotation)
  - L2: $d(I_1, I_2) = \sqrt{\sum_p (I_{1p} - I_{2p})^2}$ (rotat invariant)
- Training time: $O(1)$ (store data).
- Test time: $O(ND)$ to compare with N training samples of D dimensions. Faster with approximate methods.
- Curse of dimensionality: Distances become less meaningful in high dimensions.

Linear Classifiers

$f(x, W, b) = Wx + b$ (Scores for each class)

- W: weights (matrix of size [num_classes × num_features]). b: bias vector.
- Image input x often flatten into a column vect.
- Decision boundary is linear.
- Training time (e.g., SGD): $O(NKD)$ per epoch if N samples, K classes, D features. $O(KD)$ per mini-batch update.
- Params $= D_{in} + 1$
- Test time: $O(KD)$ per sample.
- Template matching if visualize weights.

Recurrent Neural Networks

Vanilla RNN:

Process sequential data by storing a hidden state.

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

Params: $d_h^2 + d_x d_h + d_h d_y + d_h + d_y$
Complex: $O(nd^2)$ for seq length n, hidden size d
Issue: Vanish/explod grad over long sequences

LSTM

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{(forget gate)}$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{(input gate)}$$
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(output gate)}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \text{(cell state)}$$
$$h_t = o_t \odot \tanh(C_t) \quad \text{(hidden state)}$$

Params: $4 \times ((d_h + d_x) \times d_h + d_h)$
Solves vanish grad through gating mechanisms
Complexity: $O(nd^2)$

GRU (Gated Recurrent Unit)
Simpler alternative to LSTM with fewer parms.

Efficient Conv Implementation

- **im2col:** Input image patches are rearranged into columns of a matrix. Filters are also arranged as rows. Convolution becomes a single large matrix multiplication (GEMM).
- **GEMM (General Matrix Multiplication)**
- **FFT-based convs:** Efficient for large filters, uses $O(N \log N)$ complexity. FastFourierTrans.

Vision Transformer (ViT)

- Split img to patches, project to embedding dim
- Add positional embed + prepend CLS token
- Process with transformer encoder with $O(n^2)$
- No CNN operationss - relies on self-attention
- Classificat from CLS token or pooled featuresn

MoE: E experts per layer, activ $A < E$ per token

- **Filter Visualizat:** Direct viz of learned filters
- **Saliency Maps:** Compute gradient of class score $\nabla_x S_c(x)$ - which pixels matter
- **Class Activation Mapping CAM:** $M_c(x,y) = \sum_k w_{k,c} f_k(x,y)$ requires GlobAvgPool; CNNs with GAP before final FC layer
- **Grad-CAM:** General CAM for any CNN.

1. Semantic Segmentation
Label each pixel with semantic category
Sliding window → Fully convolutional (FCN)
U-Net: Encoder-decoder with skip connections
Upsampling: Transposed conv, max unpool

2. Object Detection
Classify + locate objects with bounding boxes
R-CNN: Extract regions → classify each independ. **Fast R-CNN:** Shared conv features + RoI pooling. **Faster R-CNN:** Add Region Proposal Network (RPN) with anchor points. **YOLO:** Single-stage detector, grid-based approach

3. Instance Segmentation
Mask R-CNN: Faster + mask prediction branch
RoI Align for better feature alignment

Transfer Learning

- Pretrain CNN, then remove original classifier
- **Feature Extraction:** Freeze pretrained, train only new classifier
- **Fine-tuning:** Train whole net with small LR
- Early layers=general featur, later=task-specific
- Effective for small target datasets

3D Conv Networks for Video

3D Convolution (T= No. frames):

- Input: Video clip of shape $C \times T \times H \times W$
- Kernel: 3D filter of shape $C_{out} \times C_{in} \times T_{kernel} \times H_{kernel} \times W_{kernel}$
- Temporal stride controls temporal downsampl
- Parameters: $(T_{kernel} \times H_{kernel} \times W_{kernel} \times C_{in} + 1) \times C_{out}$
- Complexity: $O(T_{out} \times H_{out} \times W_{out} \times C_{out} \times C_{in} \times T_{kernel} \times H_{kernel} \times W_{kernel})$

Video Understanding Approaches:

- **2D CNN (Single-frame):** Process each frame independently
- **Early Fusion:** Stack frames along channel dimension at input. **Late Fusion:** Process frames separately, fuse features later
- **3D CNN:** Use 3D convs throughout network
- **(2+1)D CNN:** Factorize 3D conv into spatial 2D + temporal 1D

Common Architectures:

- C3D: 3D version of VGG
- I3D: Inflated 3D nets from 2D pretrain models
- SlowFast: 2-branch net for slow and fast motion
- Video Transfmr: Apply self-att to video tokens

METRICS

- **Accuracy:** Correct / Total
- **Precision:** TP / (TP + FP) - Of predicted positive, how many correct?
- **Recall:** TP / (TP + FN) - Of actual positive, how many found?
- **F1:** $2 \times (Prec \times Recall)/(Prec + Recall)$
- **IoU:** $\frac{Overlap}{Union}$ (for detection/segmentation)
- **mAP:** Mean Average Precision across classes