# Complete Machine Learning & Deep Learning Notes

## Table of Contents



---

# 1. MACHINE LEARNING FUNDAMENTALS

### 1.1 What is Machine Learning?

Machine learning is about teaching computers to learn patterns from data instead of explicitly programming rules.

**Three Types:**

- **Supervised Learning**: You have labeled data (input + correct output)
- **Unsupervised Learning**: Only input data, no labels
- **Reinforcement Learning**: Agent learns by trial and error with rewards

**Example:**

- Supervised: Email spam detection (emails labeled as spam/not spam)
- Unsupervised: Customer segmentation (grouping similar customers)
- Reinforcement: Game AI (learns to play by getting points)

### 1.2 Bias-Variance Tradeoff

**Bias**: Error from wrong assumptions. High bias = underfitting (model too simple).

**Variance**: Error from sensitivity to training data. High variance = overfitting (model too complex).

**Sweet Spot**: Balance both for best generalization.

**Example:**

- High Bias: Using linear regression for clearly non-linear data (circles)
- High Variance: 10th degree polynomial that fits every noise point
- Good Balance: 2nd degree polynomial that captures the curve

### 1.3 Train/Validation/Test Split

**Why 3 splits?**

- **Training Set (60-70%)**: Model learns from this
- **Validation Set (15-20%)**: Tune hyperparameters, select best model
- **Test Set (15-20%)**: Final evaluation, never touch until the end

**Common Mistake**: Using test set multiple times leads to overfitting on test set.

### 1.4 Cross-Validation

**K-Fold Cross-Validation**: Split data into K parts, train on K-1 parts, validate on 1 part. Repeat K times.

**Why?** Get reliable performance estimate when you have limited data.

**Example:** 5-fold CV means you get 5 different performance scores, then average them.

### 1.5 Regularization

**Purpose**: Prevent overfitting by adding penalty for model complexity.

**L1 Regularization (Lasso)**:

- Penalty = $\lambda \times |\text{weights}|$
- Makes some weights exactly zero
- Good for feature selection

**L2 Regularization (Ridge)**:

- Penalty = $\lambda \times \text{weights}^2$
- Makes weights smaller but not zero
- Generally better performance

**Elastic Net**: Combines L1 + L2

**Example**: Without regularization, your model might have weights [100, -50, 200]. With L2, it might become [3, -2, 5] which generalizes better.

---

# 2. DEEP LEARNING BASICS

## 2.1 Neural Networks - The Foundation

**What is it?** Layers of interconnected neurons that transform input to output through learned weights.

**Basic Structure:**

1. Input Layer (your features)
2. Hidden Layers (learned representations)
3. Output Layer (predictions)

**Forward Pass**: Input → multiply by weights → add bias → activation → next layer

**Example Calculation:**

Input: [1, 2]
Weights: [[0.5, 0.3], [0.2, 0.8]]
Bias: [0.1, 0.2]

Layer 1: [1*0.5 + 2*0.2 + 0.1, 1*0.3 + 2*0.8 + 0.2] = [1.0, 1.9]
After ReLU: [1.0, 1.9] (no change, both positive)

## 2.2 Activation Functions

**Why needed?** Without activation, multiple layers = just one linear transformation. Activation adds non-linearity.

**Sigmoid**:

- Formula: 1/(1 + e^(-x))
- Output: (0, 1)
- Use: Binary classification output
- Problem: Vanishing gradient

**Tanh**:

- Formula: (e^x - e^(-x))/(e^x + e^(-x))
- Output: (-1, 1)
- Better than sigmoid (zero-centered)
- Problem: Still vanishing gradient

**ReLU (Most Popular)**:

- Formula: max(0, x)
- Output: [0, ∞)
- Pros: Fast, no vanishing gradient for positive values
- Cons: Dead neurons (always outputs 0)

**Leaky ReLU**:

- Formula: max(0.01x, x)
- Fixes dead ReLU problem

**GELU (Modern, used in BERT)**:

- Formula: x × Φ(x) where Φ is cumulative Gaussian distribution
- Smoother than ReLU
- Better for transformers

**When to use what?**

- Hidden layers: ReLU or GELU
- Output layer (binary): Sigmoid
- Output layer (multiclass): Softmax
- RNNs: Tanh

## 2.3 Loss Functions

**Regression Problems:**

**Mean Squared Error (MSE)**:

- Formula: $(1/n) \times \Sigma(\text{predicted} - \text{actual})^2$
- Penalizes large errors heavily
- Use: Standard regression

**Mean Absolute Error (MAE)**:

- Formula: $(1/n) \times \Sigma|\text{predicted} - \text{actual}|$
- Less sensitive to outliers
- Use: When you have outliers

**Huber Loss**:

- Combines MSE + MAE
- MSE for small errors, MAE for large errors
- Best of both worlds

**Classification Problems:**

**Binary Cross-Entropy**:

- Formula: $-[y \times \log(p) + (1-y) \times \log(1-p)]$
- Use: Binary classification

**Categorical Cross-Entropy**:

- Formula: $-\Sigma(y \times \log(p))$
- Use: Multiclass classification

**Example:** True label: Cat (class 2) Predicted probabilities: [Dog: 0.1, Bird: 0.2, Cat: 0.6, Fish: 0.1] Loss = -log(0.6) = 0.51

## 2.4 Backpropagation

**What?** Algorithm to compute gradients (how to adjust weights to reduce loss).

**How it works:**

1. Forward pass: Calculate output and loss
2. Backward pass: Calculate gradient of loss with respect to each weight
3. Update weights: weight = weight - learning_rate × gradient

**Chain Rule**: Core of backpropagation

If $y = f(u)$ and $u = g(x)$, then $dy/dx = (dy/du) \times (du/dx)$

**Example:**

```
Loss = (prediction - actual)²
prediction = weight × input
dLoss/dweight = dLoss/dprediction × dprediction/dweight
              = 2(prediction - actual) × input
```

### 2.5 Gradient Descent Variants

**Batch Gradient Descent**:

- Use entire dataset for one update
- Pros: Stable, smooth convergence
- Cons: Slow for large datasets

**Stochastic Gradient Descent (SGD)**:

- Use one sample for each update
- Pros: Fast, can escape local minima
- Cons: Noisy, unstable

**Mini-Batch Gradient Descent (Most Used)**:

- Use small batch (32, 64, 128 samples)
- Best tradeoff between speed and stability

**SGD with Momentum**:

- Adds velocity to updates
- Formula: velocity = momentum × velocity + gradient
- Helps accelerate in right direction

**Adam (Most Popular)**:

- Combines momentum + adaptive learning rates
- Automatically adjusts learning rate for each parameter
- Usually the default choice

**Learning Rate Scheduling**:

- Start high, gradually decrease
- Techniques: Step decay, Exponential decay, Cosine annealing

---

# 3. NEURAL NETWORK ARCHITECTURES

### 3.1 Convolutional Neural Networks (CNNs)

**Purpose**: Process grid-like data (images, videos).

**Key Components:**

**1. Convolutional Layer**:

- Applies filters/kernels to detect features
- Filter: small matrix (3×3, 5×5) that slides over image
- Each filter learns one pattern (edges, textures, shapes)

**Example:**

```
3×3 edge detection filter:
[-1, -1, -1]
[ 0,  0,  0]
[ 1,  1,  1]

Detects horizontal edges
```

**2. Pooling Layer**:

- Reduces spatial dimensions
- Max pooling: Take maximum in each region
- Reduces computation, adds translation invariance

**Example:**

```
4×4 input → 2×2 max pooling (2×2 regions):
[1,2,3,4]     [6,8]
[3,6,5,8]  →  [9,9]
[2,4,7,3]
[9,1,2,9]
```

**3. Fully Connected Layer**:

- At the end, flatten and classify
- Regular neural network layer

**Famous CNN Architectures:**

**LeNet-5 (1998)**: First CNN, digit recognition

**AlexNet (2012)**:

- 8 layers, 60M parameters
- Won ImageNet, started deep learning revolution
- Used ReLU, Dropout, Data augmentation

**VGGNet (2014)**:

- Very deep (16-19 layers)
- Only 3×3 filters
- Showed depth matters

**ResNet (2015)**:

- 152 layers using skip connections
- Skip connections: x + F(x) instead of just F(x)
- Solves vanishing gradient in very deep networks

**Inception/GoogLeNet (2014)**:

- Multiple filter sizes in parallel
- 1×1 convolutions for dimension reduction
- More efficient than VGG

**MobileNet (2017)**:

- Depthwise separable convolutions
- Fast and lightweight for mobile devices

**EfficientNet (2019)**:

- Systematically scales depth, width, resolution
- Best accuracy vs computation tradeoff

## 3.2 Recurrent Neural Networks (RNNs)

**Purpose**: Process sequential data (text, time series, audio).

**Key Idea**: Maintain hidden state that gets updated at each time step.

**Formula:**

```
h_t = tanh(W_hh × h_(t-1) + W_xh × x_t + b)
```

**Problem: Vanishing/Exploding Gradients**

- Long sequences → gradients become tiny or huge
- Can't learn long-term dependencies

## 3.3 Long Short-Term Memory (LSTM)

**Solution to RNN problems**: Gates that control information flow.

**Components:**

**1. Forget Gate**: What to remove from cell state

```
f_t = sigmoid(W_f × [h_(t-1), x_t] + b_f)
```

**2. Input Gate**: What new information to store

```
i_t = sigmoid(W_i × [h_(t-1), x_t] + b_i)
C̃_t = tanh(W_C × [h_(t-1), x_t] + b_C)
```

**3. Cell State Update**:

```
C_t = f_t * C_(t-1) + i_t * C̃_t
```

**4. Output Gate**: What to output

```
o_t = sigmoid(W_o × [h_(t-1), x_t] + b_o)
h_t = o_t * tanh(C_t)
```

**Why LSTM works**: Cell state acts like memory highway, gradients flow better.

## 3.4 Gated Recurrent Unit (GRU)

**Simpler than LSTM**: Only 2 gates instead of 3.

**Gates:**

1. Update gate (combines forget + input gate)
2. Reset gate

**Advantages over LSTM:**

- Fewer parameters
- Faster to train
- Often similar performance

**When to use:**

- GRU: Smaller datasets, need speed
- LSTM: Larger datasets, complex patterns

---

# 4. TRANSFORMERS & ATTENTION MECHANISMS

## 4.1 Attention Mechanism

**The Big Idea**: Instead of compressing entire sequence into fixed vector, let model focus on relevant parts.

**Problem with RNNs**:

- Fixed-size context vector = bottleneck
- Long sequences lose information

**Attention Solution**:

- For each output, look at all inputs
- Calculate importance (attention weights) for each input
- Weighted sum of inputs

**Example (Translation):**

```
English: "The cat sat on the mat"
French: "Le chat"

When generating "chat", pay high attention to "cat"
Attention weights: [0.1, 0.7, 0.1, 0.05, 0.03, 0.02]
```

**Attention Score Calculation:**

1. Query (what we're looking for)
2. Key (what each input represents)
3. Value (actual content)

**Formula:**

```
Attention(Q, K, V) = softmax((Q × K^T) / √d_k) × V
```

**Why divide by √d_k?** Prevents softmax from becoming too sharp (better gradients).

## 4.2 Self-Attention

**Difference from Attention**: Sequence attends to itself, not to another sequence.

**Purpose**: Each word understands context from all other words.

**Example:**

```
Sentence: "The animal didn't cross the street because it was too tired"

For "it", self-attention looks at all words:
- High attention on "animal" (it refers to animal)
- Low attention on "street" (not referring to street)
```

## 4.3 Multi-Head Attention

**Idea**: Run multiple attention mechanisms in parallel, each learning different relationships.

**Why?**

- Head 1 might learn syntactic relationships
- Head 2 might learn semantic relationships
- Head 3 might learn positional relationships

**Implementation:**

```
8 heads × 64 dimensions = 512 total dimensions
Each head: Q, K, V are 64-dimensional
Concatenate all heads, apply linear layer
```

## 4.4 Transformer Architecture

**Revolutionary**: No recurrence, only attention. Parallelizable, scalable.

**Encoder Structure:**

1. Input Embedding + Positional Encoding
2. Multi-Head Self-Attention
3. Add & Normalize (residual connection)
4. Feed-Forward Network
5. Add & Normalize
6. Repeat 6-12 times

**Decoder Structure:**

1. Output Embedding + Positional Encoding
2. Masked Multi-Head Self-Attention (can't see future)
3. Add & Normalize

4. Multi-Head Attention (attend to encoder)
5. Add & Normalize
6. Feed-Forward Network
7. Add & Normalize
8. Repeat 6-12 times

**Positional Encoding:**

- Transformers have no notion of order
- Add sinusoidal position information to embeddings

**Formula:**

```
PE(pos, 2i) = sin(pos / 10000^(2i/d))
PE(pos, 2i+1) = cos(pos / 10000^(2i/d))
```

**Why Transformers Win:**

- Parallelizable (RNNs process sequentially)
- Better at long-range dependencies
- Scalable to billions of parameters

---

# 5. BERT & LANGUAGE MODELS

## 5.1 Word Embeddings

**Idea**: Represent words as dense vectors where similar words are close together.

**Word2Vec (2013)**:

**Two Architectures:**

**1. CBOW (Continuous Bag of Words)**:

- Predict center word from context
- Example: "The [?] sat on the mat" → predict "cat"

**2. Skip-gram**:

- Predict context from center word
- Example: Given "cat", predict ["The", "sat", "on", "the"]
- Better for rare words

**Training Trick**: Negative sampling

- Instead of predicting from entire vocabulary
- Predict: Is this a real context word? (Yes/No)
- Sample negative examples randomly

**GloVe (2014)**:

- Matrix factorization on word co-occurrence
- Combines local context (Word2Vec) + global statistics

**FastText (2016)**:

- Character n-grams
- Can handle out-of-vocabulary words
- Example: "running" = ["run", "runn", "unni", "nnin", "ning"]

## 5.2 ELMo (2018)

**Key Innovation**: Context-dependent embeddings.

**Before**: "bank" always has same embedding **ELMo**: "river bank" vs "bank account" get different embeddings

**How?**

- Bi-directional LSTM
- Train on language modeling
- Use hidden states as embeddings

## 5.3 BERT (Bidirectional Encoder Representations from Transformers)

**Game Changer (2018)**: Pre-train then fine-tune paradigm.

**Key Innovations:**

**1. Bidirectional Context**:

- Previous models: left-to-right or right-to-left
- BERT: Sees both directions simultaneously

**2. Training Tasks:**

**Task 1: Masked Language Modeling (MLM)**:

- Randomly mask 15% of words
- Predict masked words
- Example: "The [MASK] sat on the [MASK]" → predict "cat" and "mat"

**Masking Strategy:**

- 80%: Replace with [MASK]
- 10%: Replace with random word
- 10%: Keep original

**Why?** Prevents model from only learning [MASK] token.

**Task 2: Next Sentence Prediction (NSP)**:

- Given two sentences A and B
- Predict if B follows A
- Helps understand sentence relationships

**BERT Sizes:**

- BERT-Base: 12 layers, 768 hidden, 12 heads, 110M parameters

- BERT-Large: 24 layers, 1024 hidden, 16 heads, 340M parameters

**Input Format:**

```
[CLS] sentence A [SEP] sentence B [SEP]
- [CLS]: Classification token (use for sentence-level tasks)
- [SEP]: Separator
```

**Fine-Tuning BERT:**

1. Classification: Use [CLS] token output
2. Named Entity Recognition: Use each token's output
3. Question Answering: Predict start and end positions

**Example:**

```
Question: Where is the cat?
Context: The cat is on the mat.
BERT predicts: start=4 (on), end=6 (mat)
Answer: "on the mat"
```

## 5.4 GPT (Generative Pre-trained Transformer)

**Key Difference from BERT**: Unidirectional (left-to-right), generative.

**Training**: Predict next word (language modeling).

**GPT vs BERT:**

- GPT: Better for generation tasks
- BERT: Better for understanding tasks

**Evolution:**

- GPT-1 (2018): 117M parameters
- GPT-2 (2019): 1.5B parameters
- GPT-3 (2020): 175B parameters
- GPT-4 (2023): ~1.7T parameters (estimated)

## 5.5 Other Important Models

**RoBERTa (2019)**:

- BERT trained better

- Remove NSP
- Larger batches, more data
- Beats BERT

**ALBERT (2019)**:

- Lighter BERT
- Parameter sharing across layers
- Factorized embeddings
- 18x fewer parameters, similar performance

**DistilBERT (2019)**:

- 40% smaller than BERT
- 60% faster
- Retains 97% performance
- Uses knowledge distillation

**T5 (Text-to-Text Transfer Transformer) (2019)**:

- Everything is text-to-text
- Translation: "translate English to German: hello" → "hallo"
- Classification: "classify: great movie" → "positive"

**XLNet (2019)**:

- Permutation language modeling
- Best of BERT (bidirectional) + GPT (autoregressive)

**ELECTRA (2020)**:

- Replaced token detection instead of MLM
- More efficient pre-training
- Better with less compute

---

# 6. COMPUTER VISION MODELS

## 6.1 Object Detection

**Problem**: Find and classify all objects in an image.

**R-CNN (2014)**:

1. Selective search: Generate ~2000 region proposals
2. CNN on each region
3. Classify each region

- Slow: CNN runs 2000 times per image

**Fast R-CNN (2015)**:

1. CNN on entire image once
2. ROI pooling to extract features for each region

- 10x faster

**Faster R-CNN (2015)**:

1. Region Proposal Network (RPN) to generate proposals
2. Shares conv features with detection

- 10x faster than Fast R-CNN

**YOLO (You Only Look Once) (2016)**:

- Single neural network
- Divide image into grid
- Each cell predicts bounding boxes + classes
- Real-time (45 fps)

**YOLO Evolution:**

- YOLOv1-v4: Incremental improvements
- YOLOv5: Faster, easier to use
- YOLOv8 (2023): Current state-of-the-art

**SSD (Single Shot Detector)**:

- Multiple feature maps at different scales
- Good balance of speed and accuracy

**When to use:**

- Faster R-CNN: Highest accuracy, slower
- YOLO: Real-time applications
- SSD: Middle ground

## 6.2 Semantic Segmentation

**Task**: Label every pixel with a class.

**FCN (Fully Convolutional Network) (2015)**:

- Replace FC layers with conv layers
- Upsampling to restore resolution

**U-Net (2015)**:

- Encoder-decoder with skip connections
- Very popular for medical imaging

**Structure:**

Encoder (downsampling) → Bottleneck → Decoder (upsampling)
Skip connections from encoder to decoder at each level

**DeepLab**:

- Atrous (dilated) convolutions
- Increases receptive field without losing resolution
- ASPP (Atrous Spatial Pyramid Pooling)

## 6.3 Image Generation

**GANs (Generative Adversarial Networks) (2014)**:

**Two Networks:**

1. Generator: Creates fake images
2. Discriminator: Distinguishes real from fake

**Training**: Adversarial game

- Generator tries to fool discriminator
- Discriminator tries to catch generator

**Loss:**

```
min_G max_D V(D,G) = E[log D(x)] + E[log(1 - D(G(z)))]
```

**Problems:**

- Training instability
- Mode collapse (generates only few types)

**Improvements:**

- DCGAN: Deep convolutional GANs
- WGAN: Wasserstein loss (more stable)
- StyleGAN: Control style at different scales
- StyleGAN2: Even better quality

**VAE (Variational Autoencoder)**:

- Encoder: Image → latent distribution
- Decoder: Sample from latent → reconstruct image
- Can generate new images by sampling latent space
- More stable than GANs but blurrier

**Diffusion Models (2020-2023)**:

- Start with noise, gradually denoise
- DALL-E 2, Stable Diffusion, Midjourney
- Current state-of-the-art for generation

## 6.4 Vision Transformers (ViT)

**Key Idea**: Apply transformers to images.

**How:**

1. Split image into patches (16×16)
2. Flatten each patch
3. Linear projection to embedding
4. Add positional embeddings
5. Transformer encoder

**Results**: Matches or beats CNNs with enough data.

**When to use:**

- Large datasets: ViT
- Small datasets: CNNs (better inductive bias)

**Hybrid Models:**

- Swin Transformer: Hierarchical, windows
- Better efficiency
- Current best for many vision tasks

---

# 7. OPTIMIZATION & TRAINING

## 7.1 Batch Normalization

**Problem**: Internal covariate shift (distribution of inputs to layers changes during training).

**Solution**: Normalize inputs to each layer.

**How:**

1. Calculate mean and variance of batch
2. Normalize: x_norm = (x - mean) / √(variance + ε)
3. Scale and shift: y = γ × x_norm + β

**γ and β are learnable parameters (allows model to undo normalization if needed).

**Benefits:**

- Faster training
- Higher learning rates possible
- Less sensitive to initialization
- Acts as regularization

**Where to put**: After linear layer, before activation.

## 7.2 Dropout

**Problem**: Overfitting.

**Solution**: Randomly set some neurons to zero during training.

**How:**

- During training: Drop each neuron with probability p (typically 0.5)
- During inference: Use all neurons, scale by (1-p)

**Why it works:**

- Prevents co-adaptation
- Ensemble effect (different subnetworks each time)

**Where to use**: After fully connected layers, not in conv layers usually.

## 7.3 Learning Rate Schedules

**Why**: Large LR early (fast progress), small LR later (fine-tune).

**Step Decay**:

- Reduce LR by factor every N epochs
- Example: LR × 0.1 every 30 epochs

**Exponential Decay**:

- LR = LR_0 × e^(-kt)

**Cosine Annealing**:

- LR follows cosine curve
- Smooth decrease

**Warm-up**:

- Start with tiny LR, gradually increase
- Prevents instability early in training

**One Cycle Policy**:

- Increase LR then decrease
- Used by fast.ai
- Very effective

## 7.4 Weight Initialization

**Why matters**: Bad initialization → vanishing/exploding gradients.

**Xavier/Glorot Initialization**:

- Variance = 2 / (fan_in + fan_out)
- For Tanh, Sigmoid

**He Initialization**:

- Variance = 2 / fan_in
- For ReLU
- Most common choice

**Rule**: Match initialization to activation function.

## 7.5 Data Augmentation

**Purpose**: Artificially increase dataset size, improve generalization.

**Image Augmentation:**

- Flip (horizontal/vertical)
- Rotation
- Crop
- Color jitter (brightness, contrast, saturation)
- Cutout/RandomErasing (set random patches to zero)
- Mixup (blend two images)
- CutMix (cut and paste patches)

**Text Augmentation:**

- Synonym replacement
- Random insertion
- Random swap
- Random deletion
- Back-translation (translate to another language and back)

**Advanced (AutoAugment):**

- Learn augmentation policy using RL
- Find best augmentations for your dataset

## 7.6 Transfer Learning

**Idea**: Use pre-trained model, adapt to your task.

**Why**: Pre-trained models learned general features.

**Strategies:**

**1. Feature Extraction:**

- Freeze pre-trained layers
- Only train new classification layer
- Use when: Little data, similar domain

**2. Fine-Tuning:**

- Train entire model with small LR
- Use when: More data, somewhat different domain

**3. Fine-Tuning Last Layers:**

- Freeze early layers, train last few
- Middle ground approach

**Example:**

```
ResNet trained on ImageNet → Fine-tune for medical images
- Early layers: edges, textures (keep frozen)
- Middle layers: object parts (maybe fine-tune)
- Last layers: specific objects (definitely fine-tune)
```

---

# 8. EVALUATION METRICS

## 8.1 Classification Metrics

**Confusion Matrix:**

```
                Predicted
              Pos      Neg
Actual Pos    TP       FN
       Neg    FP       TN
```

**Accuracy**: (TP + TN) / Total

- Use when: Balanced classes
- Problem: Misleading for imbalanced data

**Precision**: TP / (TP + FP)

- "Of all predicted positive, how many are actually positive?"
- Use when: False positives are costly (spam detection)

**Recall/Sensitivity**: TP / (TP + FN)

- "Of all actual positive, how many did we catch?"
- Use when: False negatives are costly (disease detection)

**F1 Score**: 2 × (Precision × Recall) / (Precision + Recall)

- Harmonic mean of precision and recall
- Use when: Need balance

**Example:**

Disease detection: 100 samples, 5 actually sick
Model predicts 10 as sick, 4 are actually sick

Accuracy: (4 + 89) / 100 = 93% (misleading!)
Precision: 4 / 10 = 40% (many false alarms)
Recall: 4 / 5 = 80% (caught most sick people)
F1: 2 × (0.4 × 0.8) / (0.4 + 0.8) = 53%

**ROC Curve**: Plot TPR vs FPR at different thresholds

- TPR = Recall
- FPR = FP / (FP + TN)

**AUC (Area Under ROC Curve)**:

- 1.0 = Perfect classifier
- 0.5 = Random guessing
- Use when: Need single metric, care about all thresholds

**PR Curve**: Precision vs Recall

- Better than ROC for imbalanced data

## 8.2 Regression Metrics

**MSE (Mean Squared Error)**:

- $(1/n) \times \Sigma(y - \hat{y})^2$
- Penalizes large errors heavily

**RMSE (Root Mean Squared Error)**:

- √MSE
- Same units as target
- Easier to interpret

**MAE (Mean Absolute Error)**:

- $(1/n) \times \Sigma|y - \hat{y}|$
- Less sensitive to outliers

**$R^2$ (R-squared)**:

- 1 - (SS_res / SS_tot)
- Range: (-∞, 1]
- 1 = Perfect fit
- 0 = As good as mean
- Negative = Worse than mean

**When to use:**

- MSE/RMSE: Standard choice
- MAE: Outliers present

- $R^2$: Want percentage of variance explained

## 8.3 NLP Metrics

**Perplexity**:

- e^(average negative log-likelihood)
- Lower is better
- Measures how "surprised" model is
- Use for: Language models

**BLEU (Bilingual Evaluation Understudy)**:

- Measures n-gram overlap with reference
- Range: 0-1
- Use for: Machine translation

**ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**:

- Similar to BLEU but focuses on recall
- ROUGE-N: N-gram overlap
- ROUGE-L: Longest common subsequence
- Use for: Summarization

**METEOR**:

- Considers synonyms and stemming
- Better correlation with human judgment than BLEU

**BERTScore**:

- Uses BERT embeddings
- Semantic similarity instead of exact match
- More robust

## 8.4 Object Detection Metrics

**IoU (Intersection over Union)**:

- (Area of Overlap) / (Area of Union)
- Measures bounding box accuracy
- Threshold: Usually 0.5

**mAP (mean Average Precision)**:

- Average precision across all classes
- Gold standard for object detection
- mAP@0.5: IoU threshold of 0.5
- mAP@0.5:0.95: Average over multiple thresholds

---

# 9. ADVANCED TOPICS

## 9.1 Few-Shot Learning

**Problem**: Learn from very few examples (1-5 samples per class).

**Approaches:**

**1. Meta-Learning (Learning to Learn)**:

- Train on many tasks
- Learn how to quickly adapt to new tasks
- MAML (Model-Agnostic Meta-Learning)

**2. Prototypical Networks**:

- Learn embedding space
- Classify based on distance to class prototypes
- Prototype = mean of support set embeddings

**3. Siamese Networks**:

- Learn similarity function
- Two inputs → same network → compare embeddings
- One-shot face recognition

**4. Matching Networks**:

- Attention-based
- Weighted nearest neighbor in embedding space

**Real-World Example:**

Few-shot image classification:
Support set: 3 images of a new animal species
Query: Is this image the same species?
Model outputs similarity score

## 9.2 Self-Supervised Learning

**Idea**: Create supervision signal from data itself, no labels needed.

**Computer Vision Methods:**

**Contrastive Learning (SimCLR, MoCo)**:

1. Take an image
2. Create two augmented versions (different crops/colors)
3. Train to make them similar
4. Make them different from other images

**Loss**: InfoNCE (Noise Contrastive Estimation)

**Rotation Prediction**:

- Rotate image by 0°, 90°, 180°, 270°
- Predict rotation

**Jigsaw Puzzles**:

- Split image into patches
- Shuffle
- Predict correct order

**NLP Methods:**

**Masked Language Modeling**: BERT approach

**Next Sentence Prediction**: Predict if sentences are consecutive

**Contrastive Learning (SimCSE)**:

- Same sentence with different dropout = positive pair
- Different sentences = negative pairs

**Why Self-Supervised?**

- Leverage unlimited unlabeled data
- Learn better representations
- Transfer to downstream tasks

## 9.3 Knowledge Distillation

**Goal**: Transfer knowledge from large model (teacher) to small model (student).

**Process:**

1. Train large teacher model
2. Teacher generates soft targets (probabilities, not just labels)
3. Train student to match teacher's outputs

**Loss Function:**

```
Loss = α × CE(student, true_labels) + (1-α) × KL(student, teacher)
```

**Temperature Scaling**: Softens probabilities

$$p_i = e^{(z_i/T)} / \Sigma e^{(z_j/T)}$$

Higher $T \rightarrow$ softer probabilities

**Why Soft Targets?**

- More information than hard labels
- Example: Dog image
  - Hard label: [0, 1, 0, 0] (only dog)
  - Soft label: [0.05, 0.85, 0.08, 0.02] (mostly dog, bit like wolf and cat)

**Applications:**

- Deploy large models to mobile
- DistilBERT from BERT
- 60% faster, 40% smaller, 97% performance

## 9.4 Neural Architecture Search (NAS)

**Goal**: Automatically design neural network architectures.

**Methods:**

**1. Reinforcement Learning**:

- Controller RNN generates architecture
- Train architecture, get accuracy
- Accuracy = reward signal
- Update controller

**2. Evolution**:

- Start with population of architectures
- Mutate and crossover
- Select best performers

**3. Differentiable NAS (DARTS)**:

- Make architecture search differentiable
- Gradient-based optimization
- Much faster

**Famous Results:**

- EfficientNet: NAS-designed, state-of-the-art
- MobileNet: Designed for mobile efficiency

**Problem**: Extremely expensive (thousands of GPU hours).

**Solution**: Once-for-all networks, weight sharing.

## 9.5 Explainability & Interpretability

**Why Needed**: Understand what model learned, build trust, debug.

**Methods:**

**1. Saliency Maps**:

- Gradient of output with respect to input
- Shows which pixels matter most
- Simple but noisy

**2. Grad-CAM (Gradient-weighted Class Activation Mapping)**:

- For CNNs
- Weighted combination of feature maps
- Shows where model looks

**Steps:**

1. Get gradients of target class with respect to conv layer
2. Global average pooling of gradients
3. Multiply feature maps by weights
4. Sum and apply ReLU

**3. Integrated Gradients**:

- Path from baseline to input
- Accumulate gradients along path
- More robust than simple gradients

**4. LIME (Local Interpretable Model-Agnostic Explanations)**:

- Perturb input
- Train simple interpretable model (linear) on perturbations
- Explains prediction locally

**5. SHAP (SHapley Additive exPlanations)**:

- Game theory approach
- Each feature's contribution to prediction
- Works for any model

**6. Attention Visualization**:

- For transformers
- Plot attention weights
- See what words model focuses on

**Example:**

Sentiment analysis: "The movie was not bad"
SHAP values: {"not": -0.3, "bad": -0.4, "not bad": +0.7}
Model learned "not bad" is positive phrase

## 9.6 Adversarial Examples

**Problem**: Small imperceptible changes to input cause wrong predictions.

**Example:**

```
Panda image + tiny noise = Gibbon (99% confidence)
Noise is invisible to human eye
```

**Attacks:**

**FGSM (Fast Gradient Sign Method)**:

```
x_adv = x + ε × sign(∇_x Loss(θ, x, y))
```

- Single step, fast
- ε controls perturbation size

**PGD (Projected Gradient Descent)**:

- Iterative FGSM
- Stronger attack

**C&W (Carlini & Wagner)**:

- Optimization-based
- Minimal perturbation
- Very effective

**Defenses:**

**Adversarial Training**:

- Include adversarial examples in training
- Makes model robust
- Expensive

**Defensive Distillation**:

- Train with soft labels
- Reduces gradient magnitude

**Input Transformations**:

- JPEG compression
- Random resizing

- Removes some perturbations

**Certified Defenses**:

- Randomized smoothing
- Provable robustness guarantees

## 9.7 Continual Learning

**Problem**: Learn new tasks without forgetting old ones (catastrophic forgetting).

**Approaches:**

**1. Regularization-Based:**

- EWC (Elastic Weight Consolidation)
- Identify important weights for old tasks
- Penalize changes to those weights

**2. Replay-Based:**

- Store subset of old data
- Replay when learning new task
- Generative replay: Use GAN to generate old data

**3. Architecture-Based:**

- Progressive Neural Networks
- Add new columns for new tasks
- Old columns frozen

**4. Meta-Learning:**

- Learn to learn without forgetting
- OML (Online Meta-Learning)

## 9.8 Graph Neural Networks (GNNs)

**Purpose**: Learn on graph-structured data (social networks, molecules, knowledge graphs).

**Key Idea**: Aggregate information from neighbors.

**Basic GNN Layer:**

```
h_v^(k+1) = σ(W × Σ(h_u^(k) / |N(v)|) for u in N(v))
```

**Types:**

**1. GCN (Graph Convolutional Network)**:

- Convolutional operation on graphs
- Aggregate neighbor features

**2. GraphSAGE**:

- Sample neighbors (for scalability)
- Different aggregators (mean, LSTM, pooling)

**3. GAT (Graph Attention Network)**:

- Attention mechanism
- Learn importance of different neighbors

**4. GIN (Graph Isomorphism Network)**:

- Most expressive GNN
- Based on Weisfeiler-Lehman test

**Applications:**

- Social network analysis
- Drug discovery (molecule property prediction)
- Recommendation systems
- Traffic prediction

**Example:**

Molecule: Predict if it's toxic
Graph: Atoms = nodes, Bonds = edges
GNN aggregates information across molecule structure

## 9.9 Multimodal Learning

**Goal**: Learn from multiple modalities (text + image, audio + video).

**Challenges:**

- Different representations
- Different noise patterns
- Missing modalities

**Approaches:**

**1. Early Fusion**:

- Concatenate features early
- Simple but may not capture interactions

**2. Late Fusion**:

- Separate networks for each modality
- Combine predictions at end

**3. Cross-Attention**:

- Modalities attend to each other
- CLIP, DALL-E

**CLIP (Contrastive Language-Image Pre-training)**:

- Train image encoder and text encoder jointly
- Contrastive loss: match image-text pairs
- Zero-shot classification

**How CLIP works:**

1. Image encoder: Vision Transformer
2. Text encoder: Text Transformer
3. Learn shared embedding space
4. Image of cat close to text "a cat" in embedding space

**Applications:**

- Image captioning
- Visual question answering
- Text-to-image generation
- Video understanding

## 9.10 Efficient Deep Learning

**Problem**: Models too large for deployment.

**Solutions:**

**1. Pruning**:

- Remove unimportant weights
- Magnitude-based: Remove smallest weights
- Structured pruning: Remove entire filters/channels
- Can reduce 90% weights with minimal accuracy loss

**2. Quantization**:

- Reduce precision (FP32 → INT8)
- Post-training quantization: After training
- Quantization-aware training: During training
- 4x smaller, 2-4x faster

**3. Low-Rank Factorization**:

- Decompose weight matrices
- $W = U \times V$ where U and V are smaller

**4. Efficient Architectures**:

- MobileNet: Depthwise separable convolutions
- EfficientNet: Compound scaling
- SqueezeNet: Fire modules

**5. Mixed Precision Training**:

- FP16 for most operations

- FP32 for critical operations
- Faster, less memory
- Needs gradient scaling

**6. Gradient Checkpointing**:

- Trade computation for memory
- Don't store all activations
- Recompute during backward pass

---

# 10. INTERVIEW QUESTIONS & ANSWERS

## 10.1 Conceptual Questions

**Q: Explain bias-variance tradeoff with an example.**

A: Bias is error from wrong assumptions (underfitting), variance is error from sensitivity to training data (overfitting). Imagine predicting house prices. Linear regression has high bias (assumes linear relationship) but low variance (stable predictions). A 20-degree polynomial has low bias (fits any curve) but high variance (wildly different on new data). Goal is finding the sweet spot where total error = bias² + variance is minimized.

**Q: Why do we need non-linear activation functions?**

A: Without them, stacking layers is pointless. Multiple linear transformations collapse into a single linear transformation. If h1 = W1×x and h2 = W2×h1, then h2 = W2×W1×x = W_combined×x. One layer is enough. Non-linearity lets us approximate any function (universal approximation theorem) and learn complex patterns like XOR, which linear models can't solve.

**Q: Explain vanishing gradient problem.**

A: In deep networks, gradients get multiplied at each layer during backprop. With sigmoid/tanh (gradients < 1), repeated multiplication makes gradients exponentially tiny. Early layers barely update, can't learn. Solution: ReLU (gradient 1 for positive inputs), residual connections (skip gradient through), or batch normalization.

**Q: Difference between bagging and boosting?**

A: Both are ensemble methods but different philosophies. Bagging (Random Forest): train many models in parallel on different data subsets, average predictions. Reduces variance. Boosting (XGBoost, AdaBoost): train models sequentially, each correcting previous errors. Reduces bias. Bagging is more robust, boosting often more accurate but can overfit.

**Q: Why does batch normalization work?**

A: Multiple reasons. Main one: reduces internal covariate shift (changing distributions across layers). Also allows higher learning rates, acts as regularization, makes network less sensitive to weight initialization. During training, normalizes using batch statistics. During inference, uses running averages computed during training.

**Q: Explain dropout. Why does it prevent overfitting?**

A: Randomly drops neurons during training (typically 50%). Prevents co-adaptation where neurons rely on specific other neurons. Forces redundant representations. Also acts like training ensemble of exponentially many sub-networks. At test time, use all neurons but scale outputs by keep probability. Result: more robust, generalizable model.

## 10.2 Architecture Questions

**Q: When would you use CNN vs Transformer for vision?**

A: CNNs have inductive bias (locality, translation invariance) → better with small datasets. Transformers are more flexible → better with huge datasets (need 100M+ images). For most practical applications with moderate data, CNNs or hybrid models (like Swin Transformer) are still better. Use ViT when you have data like Google/Facebook scale.

**Q: Explain ResNet skip connections. Why do they help?**

A: Skip connections: x + F(x) instead of just F(x). Solves vanishing gradients by creating gradient highway. During backprop, gradient flows directly through skip connection. Also helps optimization: easier to learn residual F(x) than entire transformation. Enables training 152+ layer networks. Without them, networks degrade (training error increases) after ~20 layers.

**Q: LSTM vs GRU - when to use which?**

A: LSTM has 3 gates (forget, input, output), GRU has 2 (reset, update). GRU is simpler, faster, fewer parameters. Use GRU for smaller datasets, need speed, or similar performance to LSTM. Use LSTM for larger datasets, longer sequences, or when you need the extra flexibility. In practice, try both, they often perform similarly.

**Q: Why is self-attention better than RNN?**

A: Three key advantages. (1) Parallelization: RNN processes sequentially, attention processes all tokens simultaneously. (2) Long-range dependencies: RNN path length grows linearly with distance, attention is O(1). (3) Interpretability: Can visualize attention weights. Downside: Quadratic memory complexity in sequence length, but techniques like sparse attention help.

**Q: Explain encoder-decoder architecture.**

A: Encoder processes input into context representation, decoder generates output. Used in seq2seq tasks (translation, summarization). Encoder might be bidirectional (sees full input), decoder is autoregressive (generates one token at a time, only sees previous outputs). Cross-attention lets decoder attend to encoder outputs. Transformers use this: 6 encoder layers, 6 decoder layers in original paper.

## 10.3 Training Questions

**Q: Your model is overfitting. What do you do?**

A: Multiple options, try in order: (1) Get more data or use data augmentation. (2) Add regularization (L2, dropout). (3) Reduce model complexity (fewer layers, smaller width). (4) Early stopping (stop when validation loss increases). (5) Batch normalization. (6) Ensemble models. Check if you're actually overfitting (validation loss increasing while training loss decreasing) before applying solutions.

**Q: Training loss decreasing but validation loss increasing. Why?**

A: Classic overfitting. Model memorizing training data instead of learning generalizable patterns. Solutions: regularization, more data, simpler model, early stopping. Could also be data leakage (validation data somehow in training) or train/val distributions are very different.

**Q: Explain learning rate warmup.**

A: Start with tiny learning rate, gradually increase to target over N steps (typically 1000-10000). Why? Large LR at start can destabilize training when weights are random. Particularly important for large batch sizes and adaptive optimizers like Adam. After warmup, typically use cosine annealing or step decay.

**Q: Your model is underfitting. What to do?**

A: Model is too simple or not training long enough. Solutions: (1) Increase model capacity (more layers, wider layers). (2) Train longer. (3) Reduce regularization. (4) Better features or feature engineering. (5) Different architecture. (6) Check if data has signal (is task learnable?). (7) Tune hyperparameters, especially learning rate.

**Q: Batch size - how to choose?**

A: Tradeoff between speed and generalization. Small batch (32): noisy gradients, better generalization, slower. Large batch (512+): smooth gradients, faster, may generalize worse, needs LR scaling. Practical choice: largest that fits in GPU memory while maintaining performance. For transformers, often 64-256. Use gradient accumulation if GPU memory is limited.

**Q: Class imbalance problem. How to handle?**

A: Many approaches: (1) Oversample minority class (SMOTE creates synthetic examples). (2) Undersample majority class. (3) Class weights in loss function (penalize minority errors more). (4) Focal loss (focuses on hard examples). (5) Different metrics (F1, not accuracy). (6) Anomaly detection framing. (7) Generate synthetic minority examples with GANs. Try weighted loss first, it's simplest.

## 10.4 BERT & Transformers Questions

**Q: How does BERT handle subword tokenization?**

A: Uses WordPiece tokenization. Vocabulary of 30K subword units. Frequent words stay whole ("cat"), rare words split into subwords ("embeddings" → "em", "##bed", "##dings"). Benefits: handles out-of-vocabulary words, reduces vocabulary size, captures morphology. ## indicates continuation of previous token.

**Q: Difference between BERT and GPT?**

A: BERT: bidirectional (sees full context), masked language modeling, encoder-only, better for understanding tasks (classification, NER, QA). GPT: unidirectional (left-to-right), next token prediction, decoder-only, better for generation. BERT can't generate naturally because it's trained on masked reconstruction, not autoregressive generation.

**Q: Why does BERT use [CLS] token?**

A: [CLS] (classification token) is added at start. Its final hidden state aggregates sequence information, used for sequence-level tasks. During pre-training (Next Sentence Prediction), [CLS] learns to encode sentence-pair relationship. For fine-tuning classification, add linear layer on top of [CLS] output.

**Q: How does positional encoding work in Transformers?**

A: Transformers have no recurrence, so no position information. Positional encoding adds position signal to embeddings. Uses sin/cos functions of different frequencies: PE(pos, 2i) = sin(pos/10000^(2i/d)). Why sin/cos? Model can learn to attend by relative positions. Learned positional embeddings (BERT) also work well.

**Q: Explain attention masking in Transformer decoder.**

A: Prevents positions from attending to future positions (causal masking). During training, we have full target sequence but want autoregressive generation. Mask sets attention weights to -inf for future positions, so after softmax they're 0. Example: when generating word 3, can only see words 1 and 2, not 4 onwards.

**Q: What is teacher forcing?**

A: During seq2seq training, use ground truth tokens as decoder input instead of model's predictions. Faster training, more stable. Problem: exposure bias (at test time, model sees its own predictions, different distribution). Solutions: scheduled sampling (gradually use more model predictions during training) or curriculum learning.

## 10.5 Practical/Coding Questions

**Q: How would you debug a model that's not learning (loss not decreasing)?**

A: Systematic debugging: (1) Overfit single batch (if can't, something broken). (2) Check data pipeline (labels match inputs, correct preprocessing). (3) Check loss function (correct formula, no numerical issues). (4) Check learning rate (too high/low, try 1e-3, 1e-4, 1e-5). (5) Check weight initialization. (6) Check gradients (use gradient clipping if exploding). (7) Simplify model (make sure simple version works). (8) Check for data leakage or label errors.

**Q: Your model works on training data but fails on real-world data. Why?**

A: Distribution shift. Training data doesn't represent real world. Solutions: (1) Collect more representative data. (2) Domain adaptation techniques. (3) Data augmentation matching real-world variations. (4) Test-time augmentation. (5) Ensemble models. (6) Active learning (label hard real-world examples). (7) Check preprocessing (same for train and real?). Always validate on realistic data, not just random splits.

**Q: How to detect and fix data leakage?**

A: Data leakage: information from test set leaking into training. Detection: performance too good to be true, validation >> test. Types: (1) Using future information (stock prices, include tomorrow's data). (2) Duplicates across train/test. (3) Feature derived from target. (4) Preprocessing on full dataset before split. Fix: rigorous train/test split, temporal validation for time series, check feature creation process, use proper cross-validation.

**Q: Implement attention mechanism in NumPy.**

A:

python

```python
import numpy as np

def attention(Q, K, V):
    """
    Q: (batch, seq, dim) - Query
    K: (batch, seq, dim) - Key
    V: (batch, seq, dim) - Value
    """
    d_k = Q.shape[-1]

    # Attention scores: Q @ K^T
    scores = np.matmul(Q, K.transpose(0, 2, 1))

    # Scale
    scores = scores / np.sqrt(d_k)

    # Softmax
    attention_weights = np.exp(scores - np.max(scores, axis=-1, keepdims=True))
    attention_weights /= np.sum(attention_weights, axis=-1, keepdims=True)

    # Weighted sum of values
    output = np.matmul(attention_weights, V)

    return output, attention_weights
```

**Q: How would you deploy a large language model with limited resources?**

A: Multiple strategies: (1) Quantization: INT8 or even INT4 (bitsandbytes library). (2) Pruning: Remove unimportant weights. (3) Distillation: Train smaller model from large one. (4) LoRA: Low-rank adaptation, only train small adapter matrices. (5) Model sharding: Split across devices. (6) Efficient attention: Flash attention, sparse attention. (7) Batching and caching: KV-cache for generation. (8) Use smaller model variants (7B instead of 70B). Combine multiple techniques for best results.

## 10.6 Math & Theory Questions

**Q: Derive gradient of sigmoid function.**

A:

σ(x) = 1/(1 + e^(-x))

dσ/dx = d/dx [1 + e^(-x)]^(-1)
= -1 × [1 + e^(-x)]^(-2) × (-e^(-x))
= e^(-x) / (1 + e^(-x))^2
= [1/(1 + e^(-x))] × [e^(-x)/(1 + e^(-x))]
= σ(x) × (1 - σ(x))

Nice property: derivative computed from function value!

**Q: Explain backpropagation with chain rule example.**

A: Simple network: x → w1 → h → w2 → y, loss L.

Forward: h = w1 × x, y = w2 × h, L = (y - target)^2

Backward (chain rule):
dL/dw2 = dL/dy × dy/dw2 = 2(y - target) × h
dL/dh = dL/dy × dy/dh = 2(y - target) × w2
dL/dw1 = dL/dh × dh/dw1 = [2(y - target) × w2] × x

Gradient flows backward, multiplying local gradients.

**Q: Why divide by √d_k in attention?**

A: Prevents dot products from getting too large. Large dot products → extreme softmax values (close to 0 or 1) → tiny gradients. Math: for random Q, K with mean 0, variance 1, Q·K has variance d_k. Dividing by √d_k normalizes variance back to 1. Keeps gradients healthy, especially important for large dimensions (512, 1024).

**Q: Explain softmax temperature.**

A:

Softmax with temperature T:
p_i = e^(x_i/T) / Σ e^(x_j/T)

T = 1: Normal softmax
T → 0: Sharper (picks max)
T → ∞: Uniform distribution

Example: [1, 2, 3]
T=1: [0.09, 0.24, 0.67]
T=0.5: [0.02, 0.12, 0.86] (sharper)
T=2: [0.19, 0.29, 0.52] (softer)

Use in: Knowledge distillation (high T), inference sampling (control randomness)

**Q: What is the universal approximation theorem?**

A: Neural network with single hidden layer can approximate any continuous function to arbitrary precision (given enough neurons). Doesn't mean it's easy to train or that more layers don't help. In practice, deep networks are much better: fewer neurons needed, learn hierarchical features, easier to optimize. Theorem proves networks are powerful enough, not that they're optimal.

---

# FINAL TIPS FOR INTERVIEWS

## What Interviewers Look For:

1. **Understanding over Memorization**: Explain concepts in your own words, use analogies.
2. **Practical Intuition**: Know when to use what, tradeoffs, real-world considerations.
3. **Problem-Solving**: How you approach problems, debugging skills.
4. **Math Comfort**: Not deriving everything, but understanding gradients, loss functions.
5. **Latest Trends**: Know current SOTA (Transformers, Diffusion Models, LLMs).

## Study Strategy:

1. **Master Fundamentals**: Backprop, loss functions, optimization. Everything builds on these.
2. **Code from Scratch**: Implement neural network, attention, basic models. Understanding deepens.
3. **Read Papers**: At least abstracts and conclusions of major papers (BERT, GPT, ResNet).
4. **Hands-On Projects**: Train models, debug issues, deploy. Real experience matters most.
5. **Mock Interviews**: Practice explaining concepts out loud.

## Common Mistakes to Avoid:

1. **Over-complicating**: Start simple, add complexity if asked.
2. **Jargon Overload**: Use terms correctly but explain them.
3. **Ignoring Tradeoffs**: Everything has pros/cons, mention both.
4. **Not Asking Clarifying Questions**: Understand the problem first.
5. **Giving Up**: Think out loud, interviewers want to see your thought process.

### Quick Reference Checklist:

**Must Know:**

- ✓ Forward/backward propagation
- ✓ Common loss functions and when to use
- ✓ Activation functions (ReLU, sigmoid, softmax)
- ✓ Batch normalization, dropout
- ✓ Adam optimizer
- ✓ Overfitting vs underfitting
- ✓ CNN basics (convolution, pooling)
- ✓ Transformer architecture
- ✓ Attention mechanism
- ✓ BERT pre-training tasks
- ✓ Common evaluation metrics

**Good to Know:**

- ResNet, LSTM/GRU details
- Different optimizers (SGD, Adam, AdaGrad)
- Advanced regularization techniques
- Transfer learning strategies
- Model compression techniques
- Recent models (GPT-3/4, DALL-E, Stable Diffusion)

**Nice to Have:**

- NAS, meta-learning
- GNNs, graph algorithms
- Adversarial robustness
- Theoretical guarantees
- Distributed training

---

# Resources for Deeper Learning:

**Courses:**

- Stanford CS231n (Computer Vision)
- Stanford CS224n (NLP)
- Fast.ai (Practical Deep Learning)
- DeepLearning.AI Specialization

**Books:**

- "Deep Learning" by Goodfellow, Bengio, Courville
- "Dive into Deep Learning" (d2l.ai)

**Papers:**

- "Attention Is All You Need" (Transformers)
- "BERT: Pre-training of Deep Bidirectional Transformers"
- "Deep Residual Learning" (ResNet)
- Read survey papers for overviews

**Practice:**

- Kaggle competitions
- GitHub implementations
- Reproduce paper results

---

**Good luck with your interviews! Master these fundamentals, build real projects, and you'll crush it. Remember: understanding the "why" is more important than memorizing the "what."**