ML Interview Cheat Sheet

Essential formulas, algorithms, and concepts for machine learning interviews

NumPy Essentials

Array Creation

```
np.array([1,2,3])        # From list
np.zeros((3,4))          # 3x4 zeros
np.ones((2,3))           # 2x3 ones
np.eye(3)                # 3x3 identity
np.arange(0,10,2)        # [0,2,4,6,8]
np.linspace(0,1,5)       # 5 points [0,1]
np.random.randn(3,4)     # Normal dist
```

Indexing & Slicing

```
arr[1:5]          # Elements 1-4
arr[::2]          # Every 2nd element
arr[-1]           # Last element
arr[arr > 0]      # Boolean indexing
arr[[0,2,4]]      # Fancy indexing
arr[1, :]         # Row 1, all columns
arr[:, 2:4]       # All rows, cols 2-3
```

NumPy Operations

Broadcasting Rules

```
(3,4) + (4,)   -> (3,4)  # OK
(3,1) * (1,4)  -> (3,4)  # OK
(3,4) + (3,)   -> Error  # Mismatch
```

Aggregations

```
np.sum(arr)            # Total sum
np.sum(arr, axis=0)    # Column sums
np.sum(arr, axis=1)    # Row sums
np.mean(), np.std()    # Statistics
np.argmax(), np.argmin()  # Indices
```

Shape Manipulation

```
arr.reshape(2, -1)     # -1 = inferred
arr.T                  # Transpose
arr.flatten()          # Copy to 1D
arr.ravel()            # View to 1D
np.expand_dims(arr, 0) # Add axis
```

Einstein Summation

Basic Patterns

```
'ij->'      # Sum all elements
'ij->i'     # Row sums
'ij->j'     # Column sums
'ij->ji'    # Transpose
'ii->i'     # Diagonal
'ii->'      # Trace
```

Matrix Operations

```
'ik,kj->ij'   # Matrix multiply
'ij,ij->ij'   # Element-wise
'ij,ij->'     # Frobenius product
'i,j->ij'     # Outer product
'i,i->'       # Dot product
```

Batch Operations

```
'bij,bjk->bik'   # Batch matmul
'bqd,bkd->bqk'   # Attention scores
'bhqk,bhkd->bhqd' # Multi-head attn
```

Letters that disappear are summed over!

Data Preprocessing

Normalization (Min-Max)

```
X_norm = (X − X_min) / (X_max − X_min)
```

```
X_norm = (X - X.min()) / (X.max() - X.min())
```

Standardization (Z-score)

```
X_std = (X − mean) / std
```

```
X_std = (X - X.mean()) / X.std()
```

One-Hot Encoding

```
n_classes = len(np.unique(labels))
one_hot = np.eye(n_classes)[labels]
```

Handle Missing Data

```
col_mean = np.nanmean(X, axis=0)
inds = np.where(np.isnan(X))
X[inds] = col_mean[inds[1]]
```

Classical Machine Learning

Linear Regression

Model

```
y = Xw + b
```

Loss (MSE)

```
L = (1/n) ∗ sum((y − y_pred)^2)
```

Gradients

```
dw = (2/n) ∗ X.T @ (y_pred − y)
```

```
db = (2/n) ∗ sum(y_pred − y)
```

Implementation

```
for _ in range(epochs):
    y_pred = X @ w + b
    error = y_pred - y
    dw = (2/n) * X.T @ error
    db = (2/n) * np.sum(error)
    w -= lr * dw
    b -= lr * db
```

Logistic Regression

Sigmoid Function

```
sigma(z) = 1 / (1 + exp(−z))
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Binary Cross-Entropy

```
L = −[y∗log(p) + (1−y)∗log(1−p)]
```

Gradients

```
dw = (1/n) ∗ X.T @ (y_pred − y)
```

```
db = (1/n) ∗ sum(y_pred − y)
```

Same gradient form as linear regression!

K-Means Clustering

Algorithm

1. Initialize K centroids randomly
2. Assign points to nearest centroid
3. Update centroids = mean of cluster
4. Repeat until convergence

```
for _ in range(max_iters):
    # Assign points
    dists = np.linalg.norm(
        X[:, None] - centroids, axis=2)
    labels = np.argmin(dists, axis=1)

    # Update centroids
    for k in range(K):
        centroids[k] = X[labels == k].mean(0)
```

PCA

Steps

1. Center data: X = X - mean
2. Covariance: C = (1/n) * X.T @ X
3. Eigendecomposition of C
4. Project onto top k eigenvectors

```
X_centered = X - X.mean(axis=0)
cov = X_centered.T @ X_centered / n
eigvals, eigvecs = np.linalg.eigh(cov)
# Sort by descending eigenvalue
idx = np.argsort(eigvals)[::-1]
components = eigvecs[:, idx[:k]]
X_reduced = X_centered @ components
```

Decision Trees

Gini Impurity

`Gini = 1 - sum(p_i^2)`

```
def gini(y):
    _, counts = np.unique(y, return_counts=True)
    probs = counts / len(y)
    return 1 - np.sum(probs ** 2)
```

Information Gain

`IG = H(parent) - weighted_avg(H(children))`

Entropy

`H = -sum(p_i * log2(p_i))`

Evaluation Metrics

Classification

`Precision = TP / (TP + FP)`

`Recall = TP / (TP + FN)`

`F1 = 2 * (P * R) / (P + R)`

`Accuracy = (TP + TN) / Total`

Regression

`MSE = (1/n) * sum((y - y_pred)^2)`

`MAE = (1/n) * sum(|y - y_pred|)`

`R^2 = 1 - SS_res / SS_tot`

Use F1 for imbalanced classes!

Deep Learning

Activation Functions

Name	Formula	Range
ReLU	max(0, x)	[0, inf)
Sigmoid	1/(1+e^-x)	(0, 1)
Tanh	(e^x-e^-x)/(e^x+e^-x)	(-1, 1)
LeakyReLU	max(0.01x, x)	(-inf, inf)
GELU	x*Phi(x)	(-inf, inf)

Softmax

```
softmax(x)_i = exp(x_i) / sum(exp(x_j))
```

```
def softmax(x):
    e_x = np.exp(x - np.max(x))  # Stability
    return e_x / e_x.sum(axis=-1, keepdims=True)
```

Loss Functions

Cross-Entropy (Multi-class)

```
L = -sum(y_true * log(y_pred))
```

```
def cross_entropy(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred + 1e-8))
```

Binary Cross-Entropy

```
L = -[y*log(p) + (1-y)*log(1-p)]
```

MSE Loss

```
L = (1/n) * sum((y - y_pred)^2)
```

Softmax + Cross-Entropy Gradient

```
dL/dz = y_pred - y_true
```

Simple gradient when combined!

Backpropagation

Chain Rule

```
dL/dw = dL/dy * dy/dz * dz/dw
```

2-Layer MLP Gradients

```
# Forward
z1 = X @ W1 + b1
a1 = relu(z1)
z2 = a1 @ W2 + b2
y_pred = softmax(z2)

# Backward
dz2 = y_pred - y_true
dW2 = a1.T @ dz2
db2 = dz2.sum(axis=0)

da1 = dz2 @ W2.T
dz1 = da1 * (z1 > 0)  # ReLU grad
dW1 = X.T @ dz1
db1 = dz1.sum(axis=0)
```

Weight Initialization

Xavier/Glorot (tanh, sigmoid)

```
W ~ N(0, sqrt(2/(n_in + n_out)))
```

```
std = np.sqrt(2 / (n_in + n_out))
W = np.random.randn(n_in, n_out) * std
```

He/Kaiming (ReLU)

```
W ~ N(0, sqrt(2/n_in))
```

```
std = np.sqrt(2 / n_in)
W = np.random.randn(n_in, n_out) * std
```

He for ReLU, Xavier for others

Batch Normalization

Forward Pass

```
x_norm = (x - mean) / sqrt(var + eps)
```

```
y = gamma * x_norm + beta
```

```
mean = x.mean(axis=0)
var = x.var(axis=0)
x_norm = (x - mean) / np.sqrt(var + 1e-8)
out = gamma * x_norm + beta
```

Key Points

- Normalize per feature across batch
- Learnable gamma (scale) and beta (shift)
- Use running stats at inference

Dropout

Training

```
mask = np.random.rand(*x.shape) > p
out = x * mask / (1 - p)  # Inverted
```

Inference

```
out = x  # No dropout
```

Key Points

- Randomly zero out neurons
- Scale by 1/(1-p) during training
- Prevents co-adaptation
- Disabled during evaluation

CNN Operations

Output Size Formula

```
out = (in - kernel + 2*pad) / stride + 1
```

Conv2D

```
for i in range(out_h):
    for j in range(out_w):
        h_start = i * stride
        w_start = j * stride
        region = input[h_start:h_start+k,
                       w_start:w_start+k]
        out[i,j] = np.sum(region * kernel)
```

Max Pooling

```
for i in range(out_h):
    for j in range(out_w):
        region = input[i*s:i*s+k, j*s:j*s+k]
        out[i,j] = np.max(region)
```

CNN Architecture

Typical Structure

```
Input Image
     |
Conv2D + ReLU
     |
MaxPool (reduce size)
     |
Conv2D + ReLU
     |
MaxPool
     |
Flatten
     |
Dense + ReLU
     |
Dense + Softmax
     |
Output (class probs)
```

Parameter Count

```
Conv: k*k*C_in*C_out + C_out
```

```
Dense: in_feat*out_feat + out_feat
```

Attention Mechanism

Scaled Dot-Product Attention

```
Attention(Q,K,V) = softmax(QK^T / sqrt(d_k)) V
```

```
def attention(Q, K, V, mask=None):
    d_k = Q.shape[-1]
    scores = Q @ K.transpose(-2, -1)
    scores = scores / np.sqrt(d_k)
    if mask is not None:
        scores += mask * -1e9
    weights = softmax(scores, axis=-1)
    return weights @ V
```

Scale by sqrt(d_k) to prevent vanishing gradients in softmax!

Multi-Head Attention

```
def multi_head_attention(x, W_q, W_k, W_v, W_o):
    # Project to multiple heads
    Q = x @ W_q  # (B, T, n_heads * d_k)
    K = x @ W_k
    V = x @ W_v

    # Reshape: (B, n_heads, T, d_k)
    Q = Q.reshape(B, T, n_heads, d_k).transpose(0,2,1,3)
    K = K.reshape(B, T, n_heads, d_k).transpose(0,2,1,3)
    V = V.reshape(B, T, n_heads, d_k).transpose(0,2,1,3)

    # Attention per head
    out = attention(Q, K, V)

    # Concat and project
    out = out.transpose(0,2,1,3).reshape(B, T, -1)
    return out @ W_o
```

Positional Encoding

Sinusoidal Encoding

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

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

```
def positional_encoding(seq_len, d_model):
    pos = np.arange(seq_len)[:, None]
    i = np.arange(d_model)[None, :]
    angle = pos / (10000 ** (2*(i//2) / d_model))

    pe = np.zeros((seq_len, d_model))
    pe[:, 0::2] = np.sin(angle[:, 0::2])
    pe[:, 1::2] = np.cos(angle[:, 1::2])
    return pe
```

Transformer Components

Layer Normalization

```
LN(x) = gamma * (x - mean) / sqrt(var + eps) + beta
```

```
def layer_norm(x, gamma, beta, eps=1e-5):
    mean = x.mean(axis=-1, keepdims=True)
    var = x.var(axis=-1, keepdims=True)
    return gamma * (x - mean) / np.sqrt(var + eps) + beta
```

Causal Mask

```
mask = np.triu(np.ones((T, T)), k=1)
# Apply: scores += mask * -1e9
```

Encoder Block

```
x = x + MultiHeadAttn(LN(x))
x = x + FFN(LN(x))
```

Generative Models

VAE (Variational Autoencoder)

Architecture

```
Input x
    |
Encoder -> mu, log_var
    |
Sample z (reparameterization)
    |
Decoder -> x_reconstructed
```

Reparameterization Trick

```
z = mu + sigma * epsilon
```

```
epsilon ~ N(0, 1)
```

```
def reparameterize(mu, log_var):
    std = np.exp(0.5 * log_var)
    eps = np.random.randn(*mu.shape)
    return mu + std * eps
```

VAE Loss (ELBO)

Loss Components

```
L = Reconstruction + KL Divergence
```

Reconstruction Loss

```
L_rec = ||x - x_recon||^2 or BCE
```

KL Divergence

```
KL = -0.5 * sum(1 + log_var - mu^2 - exp(log_var))
```

```
def kl_divergence(mu, log_var):
    return -0.5 * np.sum(
        1 + log_var - mu**2 - np.exp(log_var)
    )
```

KL regularizes latent space toward N(0,1)

Diffusion Models (DDPM)

Noise Schedule

```
beta_t: Linear from beta_start to beta_end
```

```
alpha_t = 1 - beta_t
```

```
alpha_bar_t = prod(alpha_1:t)
```

```
betas = np.linspace(1e-4, 0.02, T)
alphas = 1 - betas
alpha_bars = np.cumprod(alphas)
```

Forward Process (Add Noise)

```
x_t = sqrt(alpha_bar_t) * x_0 + sqrt(1 - alpha_bar_t) * eps
```

```
def forward(x_0, t, alpha_bars):
    noise = np.random.randn(*x_0.shape)
    sqrt_ab = np.sqrt(alpha_bars[t])
    sqrt_1_ab = np.sqrt(1 - alpha_bars[t])
    return sqrt_ab * x_0 + sqrt_1_ab * noise, noise
```

Training Objective

```
L = ||eps - eps_theta(x_t, t)||^2
```

- Sample x_0 from data
- Sample t uniformly from [1, T]
- Sample noise eps ~ N(0, I)
- Compute x_t from x_0 and eps
- Train network to predict eps from x_t

Reverse Process (Denoise)

```
for t in reversed(range(T)):
    eps_pred = model(x_t, t)
    x_t = denoise_step(x_t, eps_pred, t)
```

KL Divergence

Definition

```
KL(P||Q) = sum(P(x) * log(P(x)/Q(x)))
```

Properties

- KL >= 0 (always non-negative)
- KL = 0 iff P = Q
- Not symmetric: KL(P||Q) != KL(Q||P)

```
def kl_divergence(p, q, eps=1e-10):
    p = p + eps
    q = q + eps
    return np.sum(p * np.log(p / q))
```

Quick Reference

Optimizer Updates

SGD

```
w = w - lr * grad
```

SGD + Momentum

```
v = beta * v + grad
```

```
w = w - lr * v
```

Adam

```
m = b1*m + (1-b1)*grad
```

```
v = b2*v + (1-b2)*grad^2
```

```
m_hat = m / (1-b1^t)
```

```
v_hat = v / (1-b2^t)
```

```
w = w - lr * m_hat / (sqrt(v_hat) + eps)
```

Common Dimensions

Symbol	Meaning
B	Batch size
T, L	Sequence length
D, d_model	Model/embedding dim
H	Number of heads
d_k, d_v	Key/value dim per head
C	Channels (CNN)
H, W	Height, Width
K	Kernel size

Interview Tips

Common Questions

- Implement gradient descent from scratch
- Explain backprop through a network
- Why scale attention by sqrt(d_k)?
- Difference: BatchNorm vs LayerNorm?
- Why reparameterization trick in VAEs?
- Explain vanishing/exploding gradients
- When to use which activation?

Key Concepts

- Bias-variance tradeoff
- Regularization (L1, L2, Dropout)
- Overfitting vs underfitting
- Train/val/test splits

ML Coding Lab Cheat Sheet | itzsid.github.io/ml-coding-lab