Deep Learning Neural Networks – Exam Cheat Sheet

Contents

1. Lecture 0 – Overview: Biological & Artificial Neural Networks

1.1 Biological Neurons

- **Components**:

  - Dendrites: Receive input signals
  - Soma: Cell body that processes input
  - Axon: Sends output signal
  - Synapse: Weighted connections (excitatory or inhibitory)
- Neuron fires when combined input exceeds a threshold.

1.2 Brain vs Computer

- Brain is slower (ms) but massively parallel and energy efficient.
- Computers are faster (ns) but less efficient in large-scale parallelism.

1.3 Artificial Neural Networks (ANNs)

- Inspired by biological networks.
- Two main types:
  - **Spiking** Neural Networks (SNNs)
  - **Activation-based** (used in this course)
- A neuron computes:

$$u = \sum_i w_i x_i + b, \quad y = f(u)$$

2. Lecture 1 – Fundamentals: Artificial Neurons & Activation Functions

2.1 Artificial Neuron Model

$$\begin{aligned} \mathbf{x} &= [x_1, x_2, \ldots, x_n]^T \quad \text{(input vector)} \\ \mathbf{w} &= [w_1, w_2, \ldots, w_n]^T \quad \text{(weight vector)} \\ u &= \mathbf{w}^T \mathbf{x} + b \\ y &= f(u) \end{aligned}$$

2.2 Activation Functions and Their Derivatives

- **Step (threshold)**:

$$f(u) = \begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{otherwise} \end{cases}, \quad f'(u) = 0 \text{ (undefined at } u = 0)$$

- **Linear**:

$$f(u) = u, \quad f'(u) = 1$$

- **Sigmoid / Logistic**:

$$f(u) = \frac{1}{1 + e^{-u}}, \quad f'(u) = f(u)(1 - f(u))$$

- **ReLU**:

$$f(u) = \max(0, u), \quad f'(u) = \begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Tanh**:

$$f(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}, \quad f'(u) = 1 - f(u)^2$$

- **SoftMax (for class** i):

$$f_i(u) = \frac{e^{u_i}}{\sum_{k=1}^{K} e^{u_k}}, \quad \frac{\partial f_i}{\partial u_j} = \begin{cases} f_i(u)(1 - f_i(u)) & \text{if } i = j \\ -f_i(u)f_j(u) & \text{if } i \neq j \end{cases}$$

2.2 Bias and Threshold

- Threshold θ is equivalent to bias $b = -\theta$
- Bias simplifies math and is often added to weighted sum.

2.3 Initialization of weights

- Weights should not be initialized to the same value as this might lead to exploding gradient problem. A better solution is to use random or Xavier initialization.
- n_{in} is neurons in beginning of weights, n_{out} is number of neurons in end of weights.

2.4 Xavier Normal Initialization

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

2.5 Xavier Uniform Initialization

$$w \sim \mathcal{U}\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$$

2.6 Torch Size

Position	**Size**	**Meaning**
0	2	Two outer elements (e.g., a batch of 2 items)
1	1	Each contains one sub-element
2	3	Each sub-element has three values (features or numbers)

```
import torch
tensor = torch.tensor([
    [[1, 2, 3]],
    [[4, 5, 6]]
])
print(tensor.shape)  # Output: torch.Size([2, 1, 3])
```

3. Lecture 2 – Regression & Classification with Single Neuron

3.1 Types of Tasks

- **Regression**: Continuous output (e.g., income, age)
- **Classification**: Discrete labels (e.g., digit class)

3.2 Linear Neuron

$$y = \mathbf{w}^T\mathbf{x} + b$$

Performs linear regression if output is not passed through activation.

3.3 Cost Function

$$J = \frac{1}{2}(d - y)^2$$

- d: target label
- y: predicted output

3.4 Stochastic Gradient Descent (SGD)

$$\begin{aligned}
\nabla_{\mathbf{w}} J &= -(d - y)\mathbf{x} \\
\nabla_b J &= -(d - y) \\
\mathbf{w} &\leftarrow \mathbf{w} + \alpha(d - y)\mathbf{x} \\
b &\leftarrow b + \alpha(d - y)
\end{aligned}$$

- α: learning rate

3.5 Things to Remember for the Exam

- You may be asked to compute weight updates over multiple steps.
- Understand how gradients relate to prediction error.
- Be fluent in computing $y = \mathbf{w}^T\mathbf{x} + b$ and updating weights.

4. Lecture 3 – Neuron Layers

4.1 From Single Neuron to Layers

- A layer consists of K neurons sharing the same input vector.
- Weight matrix: $\mathbf{W} \in \mathbb{R}^{n\times K}$, each column $\mathbf{w}_k$ is for one neuron.
- Bias vector: $\mathbf{b} \in \mathbb{R}^K$

4.2 Computation for a Single Input

$$\mathbf{u} = \mathbf{W}^T\mathbf{x} + \mathbf{b}, \quad \mathbf{y} = f(\mathbf{u})$$

4.3 Binary Cross Entropy Loss

$$J = -[d\ln(y) + (1 - d)\ln(1 - y)]$$

4.4 Cross Entropy Loss

$$J = -\sum_{j=1}^{K} d_j ln(y_j)$$

where d is true label. Write as hot-one, d=2 becomes **d**=[0, 1] and y is output/prediction.

4.5 Batch Processing

$$\mathbf{U} = \mathbf{X}\mathbf{W} + \mathbf{B}$$

- $\mathbf{X} \in \mathbb{R}^{P \times n}$: batch of P input patterns.
- $\mathbf{B}$: bias vector replicated across rows.
- Output: $\mathbf{Y} = f(\mathbf{U})$

4.6 Notes

- Hidden layers typically use ReLU or sigmoid activations.
- Output layers use linear or softmax activations depending on task.

5. Lecture 4 – Deep Neural Networks (DNN)

5.1 DNN Structure

- A feedforward network with multiple hidden layers.
- Data flows from input to output without cycles.

5.2 Chain Rule (Backpropagation)

Scalar form:

$$\frac{\partial J}{\partial u_2} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial u_3} \cdot \frac{\partial u_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial u_2}$$

Most commonly, the first to terms sum to $-(d - y)$ if GD. Also note that the output derivative should always have the same dimensions as the variable we derive with respect to. So in the above example, $\frac{\partial J}{\partial u_2}$ should have the same dimensions as u_2. Do the Harvard multiplication and transpose if needed or follow the below formula.

Vector form:

$$\nabla_{\mathbf{x}} J = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} J$$

5.3 Forward Pass Equations

$$\mathbf{h} = f(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \quad \text{(hidden layer)}$$
$$\mathbf{y} = g(\mathbf{V}^T\mathbf{h} + \mathbf{c}) \quad \text{(output layer)}$$

- f: activation function (e.g., ReLU, sigmoid)
- g: linear (regression) or softmax (classification)

5.4 Network Properties

- **Depth**: number of layers.
- **Width**: number of neurons per layer.

6. Lecture 5 – Model Selection & Overfitting

6.1 Model Parameters

- Hyperparameters include learning rate, batch size, number of layers, neurons per layer.
- Each configuration defines a unique model.

6.2 Performance Metrics

For Regression:

$$\text{MSE} = \frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})^2$$

For Classification:

$$\text{Error Rate} = \sum_{p=1}^{P} \nVdash (d_p \neq y_p)$$

6.3 Error Types

- **Apparent Error**: error on training set (often optimistic).
- **True Error**: error over the true data distribution (unavailable).
- **Test Error**: estimate of true error on unseen data.

6.4 Overfitting

- Occurs when model fits training data too well, generalizes poorly.
- Characterized by low training error but high test error.

6.5 Validation Techniques

- **Holdout method**: e.g., 2/3 training, 1/3 testing split.
- ***k*-fold cross-validation**: rotate validation over k subsets.
- **Leave-one-out cross-validation**: each example gets a turn as validation.

7. Lecture 6 – Convolutional Neural Networks I (CNN Basics)

7.1 Motivation

- Fully connected layers are inefficient for high-dimensional inputs (e.g., images).
- They destroy spatial locality and require many parameters.

7.2 Key Concepts

- **Convolution**: Sparse local connectivity + weight sharing.

$$u(i,j) = \sum_{l=-a}^{a} \sum_{m=-b}^{b} x(i+l, j+m) \cdot w(l,m) + \text{bias}$$

- **Receptive field**: Local region processed by a filter.
- **Feature map**: Activation output from applying a filter.

7.3 Own Notes for CNN

- A filter always extends the depth of the input. So if input is RGB image of size 32 (i.e 3x32x32) the filter must have size 3xNxM
- We are computing the dot products!
- General form: NxDxHxW, Number of layers, depth, height and width N=H=W
- Stride is the factor with which the output is subsampled - i.e number of steps.
- Outputs size of NxN input, FxF kernel, S stride (eesult must be integer):

$$\frac{N-F}{S} + 1$$

- Will not shrink if we use zero-padding (i.e add zeros around the input, so NxN input becomes (N+2)x(N+2), where top, bottom, right and left are all zeros). With padding, output size becomes (P size of padding per size):

$$\frac{N-F+2P}{S} + 1$$

- Common to use padding

$$P = (F-1)/2$$

- Number of parameters

$$n * d * F * F + n$$

- If input is 12x55x55 and we want to use filters of 6x12x7x7 we.
  - Depthwise is independent of input depth. Number of filters and size becomes: 12 independent $1x7x7$ filters
- If next layer is 6x49x49 and we want to use filters of 4x6x3x3 we.
  - Pointwise becomes: 4 independent $12x1x1$ filters.

7.4 Own Notes for Pooling

- Operates over each activation map independently.
- Either Max or Average pooling is used
- Pooling is good because the position of a xcertain feature becomes less important. We care more about that the feature exists than where the feature is located.

7.5 Own Notes Others

- If sigmoid activation function is used, output becomes:

$$f(U) = \frac{1}{1 + e^{-U}} =$$

- Padding=SAME means output size should be same as input size. Padding=VALID means no zero-padding.
- General rules: number of multiplications is

$$F^2$$

while number of additions is

$$F^2 - 1$$

- **Standard convolution cost:**

$$\text{FLOPs}_{\text{standard}} = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

- **Depthwise convolution cost:**

$$\text{FLOPs}_{\text{depthwise}} = (2 \times F^2) \times D_1 \times H_2 \times W_2$$

- **Pointwise convolution cost:**

$$\text{FLOPs}_{\text{pointwise}} = (2 \times D_1) \times D_2 \times H_2 \times W_2$$

Reduces the number of depth channels.
- Receptive Field Formula:

$$\boxed{\text{RF}_k = \text{RF}_{k-1} + (f_k - 1) \cdot \prod_{i=1}^{k-1} s_i}$$

$RF_0 = 1$, k is layer and f_k is filter size at layer k.The receptive field is then RF_kxRF_k.

7.6 Standard, depthwise + pointwise conv.

The dimension reductions reduces the number of FLOPs through D1 becoming 1. For pointwise we instead get F=1. The reduction ratio becomes:

$$\frac{1}{D_1} + \frac{1}{F^2}$$

7.7 Typical Layer Types

- **Convolution layer**: Extracts spatial features.
- **Pooling layer**: Reduces spatial resolution (e.g., max pooling).
- **Fully connected layer**: Used at final stages for classification/regression.

8. Lecture 7 – CNN II (Architectures & Tricks)

8.1 Classic Architectures

- **AlexNet**: 8 layers, used ReLU, dropout, data augmentation.
- **VGG**: Deep network using small 3×3 filters.
- **GoogLeNet**: Inception modules, multi-scale filters.
- **ResNet**: 152 layers with residual (skip) connections.

8.2 Efficiency & Engineering

- **FLOPs**: Floating-point operations used to quantify layer cost.
- **Pointwise Convolution**: 1×1 filters for dimension reduction.
- **Depthwise Separable Convolution**: Used in MobileNet for efficiency.

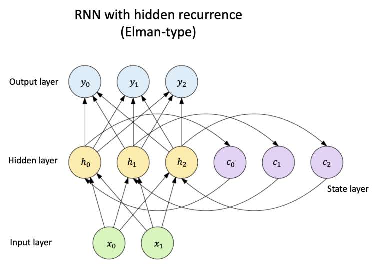
8.3 Optimization Techniques

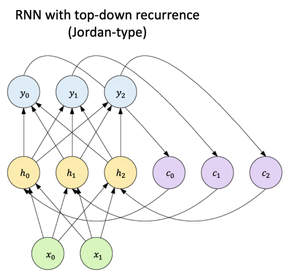
- **Batch Normalization**: Improves training stability and speed.
- **Data Augmentation**: Enhances generalization by generating variations.
- **Transfer Learning**: Use pretrained weights to accelerate training.

9. Lecture 8 – Recurrent Neural Networks (RNN)

9.1 Sequential Modeling

- RNNs process data sequentially. This is things like sound (think of soundwaves).
- The next data point is dependent on the previous data point(s).
- Hidden state maintains context over time.





9.2 Formulas for RNN - The below formulas seem to miss transpose

$$\mathbf{h}_t = \phi(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{b})$$
$$\mathbf{y}_t = \sigma(\mathbf{V}\mathbf{h}_t + \mathbf{c})$$

- ϕ: typically tanh or ReLU
- σ: softmax (classification) or linear (regression)
- The activation of the Elman RNN with one hidden layer is:

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

where U is weight vector that transforms raw inputs to hidden layer, W weight layer vector connecting previous hidden wlayer outputs to hidden layer, V is weight vector of the output layer, b is bias connected to hidden layer and c bias for output layer. ϕ is tanh.

9.3 Formulas for RNN - Jordan type

$$h(t) = \phi(U^T x(t) + W^T y(t-1) + b)$$
$$y(t) = \sigma(V^T h(t) + c)$$

Given two input sequences, where the Nth vector corresponds to the Nth time frame, we can batch them like the following:

$$\mathbf{x}_1 = \left(\begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 3 \end{bmatrix} \right), \quad \mathbf{x}_2 = \left(\begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \begin{bmatrix} 3 \\ -1 \end{bmatrix} \right)$$

We represent this as a batch of sequences:

$$\mathbf{X} = \left(\begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ 2 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 3 \\ 3 & -1 \end{bmatrix} \right)$$

We then use the following formulas, where ϕ is tanh and σ is sigmoid:

$$\mathbf{H}(t) = \phi\left(\mathbf{X}(t)\mathbf{U} + \mathbf{Y}(t-1)\mathbf{W} + \mathbf{B}\right)$$
$$\mathbf{Y}(t) = \sigma\left(\mathbf{H}(t)\mathbf{V} + \mathbf{C}\right)$$

Initially:

$$\mathbf{Y}(0) = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$$

9.4 Types of RNNs

- **Elman-type**: Recurrence in hidden state ($\mathbf{h}_{t-1} \to \mathbf{h}_t$)
- **Jordan-type**: Feedback from previous output ($\mathbf{y}_{t-1} \to \mathbf{h}_t$)

9.5 Unfolding in Time/Backpropagation

- RNNs are "unfolded" across time steps for training using backpropagation through time (BPTT).
- The propagation starts from the end of the sequence from two directions.
- Top down direction
- Reverse sequence direction
- This can result in exploding or vanishing gradients, since the gradient is multiplied many times by the weight matrix associated with W. However exploding gradients can easily be stopped through clipping at threshold level. The solution to these problems is Gated RNNs (Long Short-Term Memory, Gated Recurrent Units).

10. Lecture 9 – Transformers & Attention

Transformers are a competitive alternative to CNN. It uses encoders and decoders.

10.1 Attention Motivation

- Not all inputs are equally important. Example is the word bank (river bank or cash bank). It is dependent on the context.
- Attention dynamically focuses on relevant input parts.

10.2 Some important information

- Token is the input to the transformer, it's a set of vectors of dimensionality D. This could correspond to a word within a sentence, an amino acid within a protein etc.
- The amount of attention is quantified by learned weights and thus the output is usually formed as a weighted average.
- Self-attention steps:
- Each token, think word, is converted into a an embedding vector. These embeddings all have the same dimension d.
- We then create three new vectors for each token, q_i, k_i, v_i. These vectors are created through multiplying the input embeddings with trained weight matrices $Q_i, K_i and V_i$. These matrices are learned during training.
- The second step is to calculate a score for each word. This score shows how much to attend to each word. The score is calculated by taking the dot product of the query vector and the key vector.
- The third step is to divide the scores by $\sqrt{D_k}$ where D_k is the dimension of the key vectors.
- The forth step is to use SoftMax. This normalizes the scores so that they are all positive and add up to 1.
- The fifth step is to multiply the value vector with the softmax scores.
- The sixth step is to sum up the weighted value vectors to get the output of the self-attention

10.3 Attention Formula

$$y_n = \sum_{m=1}^{N} a_{nm} x_m$$

- a_{nm}: attention weights learned via dot-product of queries and keys.
- Output y_n is a weighted combination of all inputs.

$$Y = Attention(Q, K, V) = SoftMax(\frac{QK^T}{\sqrt{D_k}})V$$

- Softmax makes every row add up to 1.

10.4 Multi-head self-attention

- In multi-headed self-attention, the model runs multiple independent attention heads in parallel. This results in multiple outputs instead of just one.
- This is important as there might be multiple patterns that are relevant at the same time (single head can average these effects).
- Identically structured copies of the single head, with independent learnable parameters.

- H number of heads are first concatenated into a single matrix, they are then linearly transformed using a matrix $W^{(o)}$

$$Y(X) = Concat(H_1, H_2, ..., H_H)W^{(o)}$$

- To improve performance we can introduce residual connections that bypass the multi-headed structure. The resulting transformations can be written as

$$Z = LayerNorm(Y(X) + X)$$

-

10.5 Positional Encoding

- Self-attention doesn't know the order of the vectors it is processing. We can introduce prositional vectors of the same size as the input embedding, which hold information about the relative or absolute position of the tokens. These are added to the original embeddings to create the tokens.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

10.6 Decoder in sequence to sequence transformer

- Cross attention - An encoder transformer is used to map the input token to a suitable internal representation.

-

10.7 Vision Transformer

- Do not have decoder

- We firstly reshape the image into sequences of flattened 2d patches (flatten subimages). (H, W) original image size, C number of channels, (P, P) resolution of image patch, N=HW/P$^2 isresultingnumberofpatches.$

10.8 Transformer Model

- No recurrence; allows parallel computation.
- Built from:
  - Scaled dot-product attention
  - Multi-head attention
  - Feedforward layers
  - Positional encoding

10.9 Applications

- Natural Language Processing (NLP): BERT, GPT
- Vision Transformers (ViT)
- Multimodal learning

11. Lecture 10 – Autoencoders

Autoencoders are unsupervised neural networks that learn to compress input data into a lower-dimensional latent representation and then reconstruct the original input from it. They consist of an encoder and a decoder and are trained by minimizing the reconstruction error between input and output. Since they do not require labeled data, autoencoders are a form of unsupervised learning.

Autoencoders (AE)

- Unsupervised neural networks that learn to copy input $\mathbf{x}$ to output $\mathbf{y}$.
- Two main components:
  - **Encoder:** $\mathbf{h} = \phi(\mathbf{x})$
  - **Decoder:** $\mathbf{y} = \psi(\mathbf{h})$
- Useful when $\mathbf{y} \approx \mathbf{x}$ but ϕ is forced to learn compressed or structured representations.
- We must constrain the mapping to not allow direct copies of the original data and to not use tied weights, i.e use the transpose of the encoder weight matrix as the decoder weight matrix.

Training

- Common loss functions:
  - Mean squared error (MSE): $J = \frac{1}{P}\sum_{p=1}^{P} \|\mathbf{y}_p - \mathbf{x}_p\|^2$
  - Cross-entropy: $J = -\sum_{p=1}^{P} [\mathbf{x}_p \log \mathbf{y}_p + (1 - \mathbf{x}_p)\log(1 - \mathbf{y}_p)]$
  - P is number of inputs, K is dimension of each input, x_{pk} is true value and y_{pk} is predicted value
- Learn via gradient descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J, \quad \mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$

Variants of Autoencoders

Undercomplete AE

- Hidden size $M < n$ (input dim): forces model to compress.
- Learns most salient structure (like PCA).

Overcomplete AE

- Hidden size $M > n$: risk of learning identity.
- Needs regularization (e.g., sparsity).

Sparse AE (SAE)

- Both over- and undercomplete AE need constraints in order to make them useful.
- Adds sparsity constraint:

$$J_{\text{sparse}} = J + \beta \sum_{j=1}^{M} \text{KL}(\rho \,\|\, \hat{\rho}_j)$$

- ρ: desired average activation (e.g., 0.05)
- KL divergence enforces $\hat{\rho}_j \approx \rho$

Denoising AE (DAE)

- Corrupt input: $\tilde{\mathbf{x}} \sim \text{Noise}(\mathbf{x})$
- Train to reconstruct clean input from noisy input.
- Types of noise:
  - Additive: $\tilde{x}_i = x_i + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$
  - Multiplicative: $\tilde{x}_i = \epsilon x_i$, $\epsilon \sim \text{Bernoulli}(p)$

Variational AE (VAE)

- Learns latent distribution instead of point: $\mathbf{h} \sim \mathcal{N}(\mu, \sigma^2)$
- Sample from latent space: $z = \mu + \sigma \odot \epsilon$, $\epsilon \sim \mathcal{N}(0, 1)$
- Loss:
$$\text{VAE Loss} = \text{Reconstruction Loss} + \text{KL}\left[\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0, 1)\right]$$
- Promotes smooth, continuous latent space (generative modeling).

12. Lecture 11 – Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are unsupervised models where a generator creates fake data and a discriminator tries to detect it. They train in opposition: the generator improves to fool the discriminator, while the discriminator learns to detect fakes. Over time, the generator learns to produce realistic data.

GAN Basics (Goodfellow et al., 2014)

- GANs learn a generative distribution $p_g(x)$ to match real data distribution $p_{\text{data}}(x)$
- Two-player game:
  - **Generator** $G(z)$: generates fake samples from noise $z \sim p_z(z)$
  - **Discriminator** $D(x)$: outputs probability that x is real
- Objective (minimax):

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

GAN Training Procedure

- Alternate training:
  - Update D to maximize ability to distinguish real vs. fake
  - Update G to fool D
- Generator loss (non-saturating form):

$$\min_G -\mathbb{E}_{z\sim p_z}[\log D(G(z))]$$

- Challenges:
  - Training instability
  - Mode collapse (generator produces limited diversity)

DCGAN (Deep Convolutional GAN)

- Uses CNN layers for both G and D
- Tricks:
  - Batch normalization
  - Fractionally-strided (transpose) convolutions in generator
  - Adam optimizer with low learning rate (e.g., 0.0002)

Advanced Concepts

- **Conditional GANs (cGAN):**
  - Condition both G and D on label y: $G(z|y), D(x|y)$
- **Mode Collapse:**
  - Generator learns to produce few modes (repetitive outputs)
  - Solutions: minibatch discrimination, unrolled GANs, diversity penalties
- **Latent Space Arithmetic:**
  - Interpolate or manipulate z (e.g., $z_{\text{new}} = mz_1 + (1-m)z_2$) to change generated outputs smoothly

Advantages and Challenges

- **Pros:**
  - No need for explicit density estimation
  - High-quality, sharp image generation
  - Sampling is efficient
- **Cons:**
  - Hard to train (non-convergence)
  - Evaluation is nontrivial
  - No explicit likelihood function