Training Neural Networks

Loss Functions

High-Level Summary

A **loss function** is a mathematical formula that measures **how wrong a model's prediction is** compared to the true label.

It provides the **signal for learning** by guiding how the model's parameters should be updated during training.

Detailed Explanation

When a model makes predictions, we need a way to **quantify the error**.

- **Loss function**: Calculates error for a single prediction.
- **Cost function**: Usually refers to the average loss across all training examples.

Loss functions differ depending on the **type of task**:

🔹 1. Regression Loss Functions (continuous outputs)

- **Mean Squared Error (MSE)**:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  - Penalizes larger errors more heavily.
  - Common in predicting prices, temperatures, etc.

- **Mean Absolute Error (MAE)**:

$$L = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

- Robust to outliers, treats all errors equally.

- **Huber Loss**:
  - Combination of MSE + MAE (less sensitive to outliers than MSE).

🔹 2. Classification Loss Functions (discrete outputs)

- **Binary Cross-Entropy (Log Loss)** (for binary classification):

$$L = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]$$

  - Used with **Sigmoid** in binary tasks.
  - Example: spam vs. not spam.

- **Categorical Cross-Entropy** (for multi-class classification):

$$L = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

  - Used with **Softmax**.
  - Example: image classification (dog, cat, car).

🔹 3. Specialized Loss Functions

- **Hinge Loss**: Used in Support Vector Machines (SVMs).
- **KL Divergence**: Measures difference between probability distributions.
- **IoU (Intersection over Union) Loss / Dice Loss**: Used in **segmentation** tasks.
- **Adversarial Loss**: Used in GANs (generator vs. discriminator).
- **Contrastive Loss / Triplet Loss**: Used in **metric learning** (face verification, embeddings).

Analogy

Think of the loss function like a **teacher's red pen**:

- Every time you answer (predict), the teacher marks how wrong you are.
- The goal is to keep adjusting (learning) until the red marks (loss) are minimized.

Mathematical Foundation

General idea:

$$(y, \hat{y}) = \text{Error between true value } y \text{ and prediction } \hat{y}$$

Training goal:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f_\theta(x_i))$$

Where:

- f_θ = model with parameters θ.
- y_i = true labels.
- $\hat{y}_i$ = predictions.

Use Case Examples

- Predicting **house prices** → MSE (regression).
- Detecting **spam emails** → Binary Cross-Entropy.
- Classifying **ImageNet images** → Categorical Cross-Entropy.
- Segmenting **medical images** → Dice Loss / IoU Loss.
- Training **GANs** → Adversarial Loss.

✅ **Key Insight**:

- **Regression → MSE / MAE**

- **Classification → Cross-Entropy**
- **Segmentation → Dice / IoU**
- **Generative / Embedding tasks → Specialized losses**

Convex Loss Function

- In optimization, we want to **minimize the loss function** to find the best model parameters.
- If the loss function is **convex**, gradient descent is guaranteed (in theory) to reach the **global minimum**.
- If it is **non-convex**, there may be many local minima/saddle points → optimization becomes harder (common in deep learning).
- A **convex loss function** is one where the error curve is shaped like a **bowl (U-shape) —** meaning it has **one global minimum** and no local minima, making optimization easier and more stable.

Examples of Convex Loss Functions

1. **Mean Squared Error (MSE)**
    - Quadratic → convex.
    - Common in regression.
2. **Hinge Loss** (used in SVMs)
    - Convex because it is piecewise linear.
3. **Log Loss / Cross-Entropy** (for classification)
    - Convex with respect to predictions.

Gradient Descent & Stochastic Gradient Descent

High-Level Summary

- **Gradient Descent (GD)** is an optimization method that updates model parameters in the **direction of the negative gradient** of the loss to minimize it.

- **Stochastic Gradient Descent (SGD)** is a faster variant that updates parameters using **one (or a few) training examples at a time**, introducing randomness that helps escape poor local minima.

Detailed Explanation

🔹 1. Gradient Descent (Batch Gradient Descent)

- Compute the loss over the **entire dataset**.
- Compute the gradient of loss w.r.t. each parameter.
- Update all parameters **once per pass (epoch)**.

Update rule:

$$\theta \ \leftarrow \ \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

Where:

- θ = parameters (weights, biases)
- η = learning rate (step size)
- $\nabla_\theta \mathcal{L}(\theta)$ = gradient of loss function

Pros: Stable, guaranteed convergence for convex losses.

Cons: Very **slow** for large datasets because each update requires scanning all examples.

🔹 2. Stochastic Gradient Descent (SGD)

- Instead of using all data, update parameters after **just one sample** (or a small random subset).
- Introduces **stochasticity (noise)** into updates.

Update rule (per sample):

$$\theta \ \leftarrow \ \theta - \eta \nabla_\theta \mathcal{L}(x_i, y_i; \theta)$$

Pros:

- Much faster for large datasets.
- Noise helps **escape saddle points and local minima**.

Cons:

- Updates are noisy, loss may oscillate.
- Convergence less stable without tricks like learning rate schedules or momentum.

🔹 3. Mini-Batch Gradient Descent (Hybrid)

- Uses a **small batch** of data (e.g., 32, 64, 128 samples).
- Compromise between **efficiency** and **stability**.
- Modern deep learning almost always uses **mini-batch SGD**.

Update rule (per mini-batch):

$$\theta \ \leftarrow \ \theta - \eta \frac{1}{m} \sum_{j=1}^{m} \nabla_\theta \mathcal{L}(x_j, y_j; \theta)$$

Where m = batch size.

Analogy

- **Batch Gradient Descent**: Like checking the **entire class's exam papers** before deciding how to improve teaching.
- **SGD**: Like checking **just one student's paper** and immediately changing teaching style.
- **Mini-batch GD**: Like checking a **small group's papers** (say 10 students) before making adjustments.

Mathematical Foundation

1. We want to solve:

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$$

1. Gradient descent updates iteratively:

$$theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

1. For mini-batches:

$$\nabla_\theta \mathcal{L}(\theta_t) \approx \frac{1}{m} \sum_{j=1}^{m} \nabla_\theta \mathcal{L}(x_j, y_j; \theta_t)$$

Use Cases

- **Batch GD**: Small datasets where full dataset fits in memory (e.g., linear regression on small data).
- **SGD**: Online learning, streaming data, very large datasets.
- **Mini-batch GD**: Standard choice for deep learning (vision, NLP, speech).

Extra Insight

- SGD is often paired with **optimizers** that improve convergence:
  - **Momentum**: accelerates in consistent gradient directions.
  - **Adam, RMSProp**: adaptive learning rates.
- In practice, when people say **“SGD” in deep learning**, they almost always mean **mini-batch SGD**.

✅ **Key Takeaway**:

- **Batch GD** = exact but slow.
- **SGD** = noisy but fast.
- **Mini-batch SGD** = sweet spot → standard in deep learning.

Vanishing & Exploding Gradients

High-Level Summary

- The **Vanishing Gradient Problem** happens when gradients become extremely **small** during backpropagation, making early layers learn very slowly (or stop learning).
- The **Exploding Gradient Problem** happens when gradients become extremely **large**, causing unstable updates and diverging weights.

Detailed Explanation

🔹 1. Why These Problems Occur

- In deep networks, backpropagation uses the **chain rule**:

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} \cdots \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[1]}}$$

- This is a **product of many derivatives**.
- If derivatives < 1 → product shrinks → **vanishing gradient**.
- If derivatives > 1 → product explodes → **exploding gradient**.

🔹 2. Vanishing Gradient

- Common with **sigmoid** or **tanh** activations, where:
  - $\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq 0.25$.
  - $\tanh'(z) \leq 1$.
- In deep nets, multiplying numbers < 1 repeatedly → gradient goes to **0**.

Effect:

- Early layers don't update → network can't learn long dependencies.
- RNNs suffered heavily from this problem before LSTM/GRU.

🔹 3. Exploding Gradient

- Opposite effect: derivatives (or weights) are large.
- Multiplying numbers > 1 repeatedly → gradient grows **exponentially**.

Effect:

- Weight updates become huge.
- Loss oscillates or goes to **NaN** (not a number).

🔹 4. Causes

- Poor weight initialization (too large/small).

- Very deep architectures.
- Certain activation functions (sigmoid, tanh are more prone).
- Recurrent networks (due to repeated multiplications over time steps).

Analogy

- **Vanishing Gradient**: Like whispering a message down a **long hallway**. By the time it reaches the first person, the signal is so faint they can't hear it.
- **Exploding Gradient**: Like shouting through a **microphone with max volume**. The signal becomes so loud that it distorts everything.

Mathematical Foundation

Suppose each layer multiplies by a Jacobian matrix J[l]J^{[l]}.

The gradient is:

$\frac{\partial \mathcal{L}}{\partial x} = \prod_{l=1}^{L} J^{[l]}$

- If eigenvalues of $J^{[l]}$ are < **1**, the product tends to **0** → vanishing gradient.
- If eigenvalues are > **1**, the product tends to ∞ → exploding gradient.

Solutions

✅ **For Vanishing Gradient**

- Use **ReLU** (derivative is 0 or 1, not shrinking like sigmoid).
- Use **Batch Normalization**.
- Use **Residual Connections (ResNets)**.
- Use **LSTMs/GRUs** in RNNs (gating helps preserve gradient flow).
- Careful weight initialization (Xavier/He init).

✅ **For Exploding Gradient**

- **Gradient Clipping** (cap gradients at a threshold).
- Smaller learning rates.

- Careful initialization.

Use Case

- In training **deep CNNs**, vanishing gradients can stall learning in early layers → ResNet skip connections solved this.
- In **RNNs**, exploding gradients caused instability → gradient clipping + gated RNNs fixed it.

✅ **Key Takeaway**:

- **Vanishing = network can't learn early features.**
- **Exploding = network updates blow up, training diverges.**

Backpropagation (Forward & Backward Propagation)

High-Level Summary

Backpropagation is the algorithm that **efficiently computes gradients** of a loss with respect to all network parameters by applying the **chain rule** backward through the network; the **forward pass** computes predictions and loss, the **backward pass** propagates error signals to obtain gradients used to update weights.

Detailed Explanation

1) Forward Propagation (compute predictions & loss)

For an LL-layer feedforward network (MLP/CNN head notation):

- Inputs: $a^{[0]} = x$
- For each layer $l = 1, \ldots, L$:
    - **Affine/conv step**: $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
        - (For CNNs, $W^{[l]}$ applies a convolution instead of a matrix multiply.)

- **Nonlinearity**: $a^{[l]} = \phi^{[l]}(z^{[l]})$

- **Loss** (example: cross-entropy for classification): $\mathcal{L} = \ell(a^{[L]}, y)$

You also **cache** intermediates $\{z^{[l]}, a^{[l]}\}$ for the backward pass.

---

2) Backward Propagation (compute gradients)

Start from the loss and move **right→left** (output→input).

Define the **error signal** (a.k.a. **delta**) at layer l:

$$\delta^{[l]} \ \triangleq \ \frac{\partial \mathcal{L}}{\partial z^{[l]}}$$

Then, for $l = L, L-1, \ldots, 1$:

1. **Local gradient at layer output**

$$\delta^{[l]} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \ \odot \ \phi'^{[l]}(z^{[l]})$$

   where $\odot$ is the elementwise product.

2. **Parameter gradients**

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \delta^{[l]} \, (a^{[l-1]})^\top, \qquad \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \mathrm{sum}(\delta^{[l]})$$

   (sum over batch and spatial axes as appropriate).

3. **Back-propagate to previous activations**

$$\frac{\partial \mathcal{L}}{\partial a^{[l-1]}} = (W^{[l]})^\top \delta^{[l]}$$

   (for CNNs, this is a convolution with flipped kernels).

Finally, update parameters (e.g., SGD):

$$W^{[l]} \leftarrow W^{[l]} - \eta \, \frac{\partial \mathcal{L}}{\partial W^{[l]}}, \quad b^{[l]} \leftarrow b^{[l]} - \eta \, \frac{\partial \mathcal{L}}{\partial b^{[l]}}$$

Batch setting: replace outer products/sums by their batch-averaged counterparts.

Analogy

Think of a **factory line**:

- **Forward pass**: raw material (input) flows through stations (layers) to produce a product (prediction) and a quality score (loss).
- **Backward pass**: a **quality inspector** walks backward, station by station, telling each what adjustments reduce defects. Those adjustments are the **gradients**.

Mathematical Foundation

A) Chain Rule (multivariate)

For composed mappings $x \xrightarrow{f_1} u \xrightarrow{f_2} v \xrightarrow{f_3} \mathcal{L}$,

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u} \frac{\partial u}{\partial x}$$

Backprop is **reverse-mode autodiff**: compute a single scalar loss gradient wrt **all** parameters in time proportional to a few forward passes.

B) Layerwise derivatives (dense layer)

For $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \;\; a^{[l]} = \phi^{[l]}(z^{[l]})$:

- Activation derivative:

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot \phi'^{[l]}(z^{[l]})$$

- Weights, bias:

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \left(\frac{\partial \mathcal{L}}{\partial z^{[l]}} \right) (a^{[l-1]})^\top, \qquad \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \mathrm{sum}\left(\frac{\partial \mathcal{L}}{\partial z^{[l]}} \right)$$

- Previous activations:

$$\frac{\partial \mathcal{L}}{\partial a^{[l-1]}} = (W^{[l]})^\top \left(\frac{\partial \mathcal{L}}{\partial z^{[l]}}\right)$$

C) Common activation derivatives

$$\text{ReLU: } \phi'(z) = \mathbf{1}[z > 0];$$

$$\text{Sigmoid: } \sigma'(z) = \sigma(z)(1 - \sigma(z));$$

$$\text{Tanh: } \tanh'(z) = 1 - \tanh^2(z)$$

D) Softmax + Cross-Entropy (clean top-layer gradient)

Let $\hat{y} = \mathrm{softmax}(z^{[L]})$, one-hot y, and $\mathcal{L} = -\sum_c y_c \log \hat{y}_c$.

Then a key simplification:

$$\boxed{\delta^{[L]} \equiv \frac{\partial \mathcal{L}}{\partial z^{[L]}} = \hat{y} - y}$$

This is why softmax-CE is numerically and analytically convenient.

E) Convolution layers (intuition of gradients)

- $z^{[l]} = W^{[l]} * a^{[l-1]} + b^{[l]}$($*$ = convolution)
- Weight gradient: **correlate** input activations with $\delta^{[l]}$.
- Input gradient: **convolve** $\delta^{[l]}$ with **flipped** kernels (transpose conv).
- Bias gradient: sum $\delta^{[l]}$ over batch and spatial positions.

F) Stability tricks

- Use **cached forward values** (e.g., $\sigma(z)$ to compute $\sigma'(z)$).
- **BatchNorm**, **residual connections**, and good inits (He/Xavier) stabilize gradients.
- **Gradient checking** (finite differences) can verify an implementation.

Use Case

Training an image classifier:

1. Forward: images → CNN → logits → softmax → cross-entropy loss.

2. Backward: compute $\hat{y} - y$ at the output, propagate through FC, conv, pool, and activations to get $\{\partial\mathcal{L}/\partial W, \partial\mathcal{L}/\partial b\}$ for **every layer**.
3. Update params with SGD/Adam. Repeat until convergence.

Invariance vs. Equivariance

High-Level Summary

- A function is **invariant** to a transformation if the **output doesn't change** when the input changes in a specific way.
- A function is **equivariant** if the **output changes in a predictable way** when the input changes.

Detailed Explanation

🔹 1. Invariance

- A model is **invariant** if certain transformations of the input leave the output unchanged.
- Example: A classifier should still predict **"cat"** whether the cat image is **shifted, rotated, or resized**.

Formally:

f(T(x)) = f(x)

Where:

- T = transformation (e.g., translation, rotation).
- f = model.

Interpretation: The model **ignores irrelevant variations**.

🔹 2. Equivariance

- A model is **equivariant** if the transformation of the input leads to a **corresponding transformation of the output**.

- Example: In semantic segmentation, if you shift the input image by 10 pixels, the **segmentation mask** should also shift by 10 pixels.

Formally:

$f(T(x)) = T(f(x))$

Interpretation: The model **tracks transformations consistently**.

🔹 3. Examples in Deep Learning

- **CNNs**:
  - Convolutions are **translation equivariant** (shifting input shifts feature map).
  - Pooling layers add **translation invariance** (small shifts don't change output).
- **Classification networks**: We want **invariance** (e.g., rotated "3" is still a "3").
- **Detection/Segmentation networks**: We want **equivariance** (shifted object → shifted bounding box/mask).

Analogy

- **Invariance**: Imagine a **face-recognition system** → No matter how you rotate or slightly shift your head, it still recognizes *you*.
- **Equivariance**: Imagine **Google Maps arrows** → if the map rotates, the arrows rotate **accordingly**, preserving orientation.

Mathematical Foundation

- **Invariance**:

  $f(T(x)) = f(x)$

  (Output doesn't change under transformation).

- **Equivariance**:

  $f(T(x)) = T(f(x))$

  (Output transforms the same way as input).

Use Case

- **Invariance**:
  - Classification (digit recognition, face ID, object recognition).
- **Equivariance**:
  - Segmentation, detection, pose estimation (where location/orientation matters).

✅ **Key Takeaway**:

- **Invariance = ignore transformations.**
- **Equivariance = respect transformations.**
