📘 Cross-Entropy: Recap & Intuition

✅ What Is Cross-Entropy?

Cross-entropy is a **loss function** commonly used in classification tasks. It measures the difference between two probability distributions:

- The **true distribution** (from the actual labels)
- The **predicted distribution** (from the model's outputs)

It answers:

> *"How well does the model's predicted distribution match the actual labels?"*

🧮 Cross-Entropy Formula

🔷 General (multi-class):

$$\text{Loss} = -\sum_i y_i \log(\hat{y}_i)$$

- (y_i): one-hot encoded true label (1 for the correct class, 0 for others)
- (\hat{y}_i): predicted probability for class (i)

Since only the correct class has (y_i = 1), this simplifies to:

$$\text{Loss} = -\log(\hat{y}_{\text{correct class}})$$

🔷 Binary Classification (two classes):

$$\text{Loss} = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

- (y \in {0, 1}) is the true label
- (\hat{y} \in [0, 1]) is the predicted probability of the positive class

💡 Intuition

- Cross-entropy **punishes wrong predictions** more harshly as they get more confident and incorrect.
- If the model assigns a high probability to the correct class, the loss is low.
- If the model assigns a low probability to the correct class, the loss is high.

📊 Example

Multi-class example:

- True class = A (index 0)
- Predicted: [0.7, 0.2, 0.1]

$$\text{Loss} = -\log(0.7) \approx 0.357$$

- If predicted [0.1, 0.2, 0.7], loss = (-\log(0.1) \approx 2.3)

📈 Why Use Log?

- Log penalizes low probabilities **exponentially**.
- Converts multiplication (likelihood) into addition (log-likelihood).
- Helps with numerical stability.

🎯 Connection to Maximum Likelihood Estimation (MLE)

Likelihood of correct prediction:

$$L = \prod_{i=1}^{n} P(y_i \mid \hat{y}_i)$$

Take log to get log-likelihood:

$$\log L = \sum_{i=1}^{n} \log P(y_i \mid \hat{y}_i)$$

Cross-entropy loss = **negative log-likelihood**:

$$\text{Loss} = -\sum_{i=1}^{n} \log(\text{model's predicted prob for true class})$$

MLE Goal:

> "Choose model parameters that make the observed labels as probable as possible."

🔁 How It Works in Gradient Descent

- Loss is computed via cross-entropy
- Gradient tells the model how to adjust weights to **increase predicted probability of the true class**
- Weight update:

$$W \leftarrow W - \eta \cdot \nabla_W \text{Loss}$$

Where:

$$\nabla_W \text{Loss} = (\hat{y} - y) \cdot x^T$$

🧱 Softmax Derivative Summary

Given:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}, \quad z_i = w_i^T x$$

Derivative of softmax:

$$\frac{\partial \hat{y}_i}{\partial z_j} = \begin{cases} \hat{y}_i(1 - \hat{y}_i) & \text{if } i = j \\ -\hat{y}_i \hat{y}_j & \text{if } i \neq j \end{cases}$$

When using **softmax + cross-entropy**, the gradient simplifies to:

$$\frac{\partial \text{Loss}}{\partial z} = \hat{y} - y$$

🧠 Entropy & Information Theory

- **Entropy (H(p))**: average uncertainty of a true distribution

$$H(p) = -\sum p_i \log p_i$$

- **Cross-entropy (H(p, q))**: how many bits needed to encode (p) using (q)

$$H(p, q) = -\sum p_i \log q_i$$

- **KL Divergence**: measures the "distance" between two distributions

$$\text{KL}(p \parallel q) = H(p, q) - H(p)$$

Minimizing cross-entropy = minimizing KL divergence = getting predicted distribution (q) close to true (p)

🎓 Use Cases

- Binary classification (with sigmoid): binary cross-entropy

- Multi-class classification (with softmax): categorical cross-entropy
- Multi-label classification: multiple binary cross-entropies (one per label)

📘 Mean Squared Error (MSE) — Study Notes

🔢 Definition

Mean Squared Error (MSE) is a loss function that measures the average of the squares of the errors — that is, the average squared difference between actual and predicted values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

- y_i is the **true value** (actual/ground truth)
- $\hat{y}_i$ is the **predicted value**
- n is the number of data points

🎯 Intuition

1. **Error**: For each prediction, calculate the error: $y_i - \hat{y}_i$
2. **Square it**: We square the error to make all differences positive and emphasize larger errors.
3. **Average**: Add up all the squared errors and divide by the number of observations.

💡 Why use the squared error?

- Squaring **penalizes larger errors** more than smaller ones.
- It's differentiable and smooth, making it great for optimization with gradient descent.
- It's the default loss function for regression problems in many machine learning models (like linear regression, neural nets doing regression, etc).

✅ Properties

- MSE is always **non-negative**
- **Lower is better**: 0 means perfect predictions
- Units are **squared** compared to the original data (if you're predicting dollars, MSE is in dollars²)

🧠 Example

Let's say we have actual vs predicted:

Actual (y)	Predicted ($\hat{y}$)
3	2
-0.5	0
2	2
7	8

Now compute errors and MSE:

1. Errors: $[1, -0.5, 0, -1]$
2. Squared: $[1, 0.25, 0, 1]$
3. Mean: $\frac{1+0.25+0+1}{4} = 0.5625$

MSE = 0.5625

🤖 MSE and Gradient Descent

Assume your model is:

$$\hat{y}_i = wx_i + b$$

MSE loss:

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

Gradient w.r.t. w:

Using the chain rule:

$$\frac{\partial L}{\partial w} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)x_i$$

Gradient w.r.t. b:

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

So yes — you compute the gradient at each data point, **sum them**, and **divide by** n to get the average gradient for updating your weights in batch gradient descent.

🤝 MSE vs MAE vs RMSE

Metric	Description	Pros	Cons
MSE	Mean of squared errors	Smooth, convex, common	Sensitive to outliers
MAE	Mean of absolute errors	Robust to outliers	Non-smooth gradient
RMSE	Square root of MSE	Same units as target	Still sensitive to outliers

---

🚫 When not to use MSE

- Classification tasks (use **cross-entropy loss** instead)
- Heavy outliers dominate the dataset (use **MAE** or **Huber loss**)

🔄 Alternatives

- **MAE** — good for robustness
- **Huber loss** — hybrid between MSE and MAE
- **Quantile loss** — for percentile prediction
- **Log-Cosh loss** — smooth and outlier-resistant

✅ Key Takeaways

- MSE is the **go-to loss** for regression tasks
- Penalizes large errors more due to squaring
- Has a smooth, differentiable gradient — perfect for deep learning optimization
- Can be skewed by outliers → know when to use alternatives like **MAE** or **Huber**

🧯 Dropout: Regularization in Deep Learning

🔍 What is Dropout?

Dropout is a regularization technique used to prevent overfitting in neural networks. During training, dropout randomly "drops" a subset of neurons by setting their output to **zero**. This forces the network to learn **redundant, robust representations** that do not rely on any specific neuron.

⚙️ Implementation Example (Pseudocode)

```
# Forward pass with dropout (PyTorch-style logic)
mask = torch.bernoulli(torch.full_like(activations, p))  # p = keep probability
output = activations * mask / p
```

- **p**: probability of keeping a neuron (e.g., 0.8 means 20% are dropped)
- Division by **p** scales activations to preserve expected value

🎓 Why Does Dropout Work?

Dropout prevents co-adaptation of neurons: they can't rely on specific other neurons being present.

It trains an ensemble of subnetworks, effectively improving generalization.

Dropout improves robustness by forcing the model to spread out learned representations.

🔬 Key Points to Remember

Feature	Description
Applied to	Activations (after ReLU, typically in fully connected layers)
Typical dropout rates	0.2 – 0.5 (common: 0.5 in dense layers)
During training	Randomly masks out activations
During inference	No dropout, optionally scale activations by keep_prob
Effect on gradient flow	Dropped neurons receive no gradient
Difference from ReLU = 0	ReLU output = 0 is computed; Dropout = 0 is overwritten

🧠 Intuition

Dropout is like training a different, smaller network each time and averaging them — but efficiently, without explicitly building all the networks.

❌ When Not to Use Dropout

- Small models with little overfitting risk
- Convolutional layers (often unnecessary or harmful here)
- Recurrent networks (requires special care: e.g., `nn.Dropout` with consistent masks)
- When using other strong regularizers (e.g., heavy augmentation, weight decay)

✅ When to Use Dropout

- Deep fully connected networks
- Large datasets with risk of overfitting
- As a simple, powerful addition to your regularization toolkit

📐 Math Recap

If a neuron's activation is:

$$a = \phi(w^T x + b)$$

Then during dropout:

$$a_{\text{dropped}} = a \cdot m$$

Where:

- (m \sim \text{Bernoulli}(p)), a binary mask
- (p): the keep probability (e.g., 0.8)

During training, scale:

$$a_{\text{dropped}} = \frac{a \cdot m}{p}$$

During inference, use full (a) (optionally scaled by (p)).

🔗 References

- Srivastava et al. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting (https://jmlr.org/papers/v15/srivastava14a.html) ```

📚 Regularization & Weight Decay: Detailed Notes

🧱 L1 and L2 Regularization

✅ L1 Regularization

Adds a penalty proportional to the **absolute value** of weights:

$$\text{Loss}_{\text{total}} = \text{Loss}_{\text{data}} + \lambda \sum_i |w_i|$$

- Encourages **sparsity** (many weights go to exactly zero)
- Useful for **feature selection**
- Not smooth at 0 — harder to optimize

✅ L2 Regularization

Adds a penalty proportional to the **squared value** of weights:

$$\text{Loss}_{\text{total}} = \text{Loss}_{\text{data}} + \lambda \sum_i w_i^2$$

- Encourages **small** weights (but not exactly zero)
- Helps with **overfitting**
- Common in deep learning
- Equivalent to **weight decay** in SGD

🔍 Gradient of the Regularization Term

Loss with L2 regularization:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \cdot \frac{1}{2} \sum_i w_i^2$$

Taking the gradient w.r.t. each weight:

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial w_i} = \frac{\partial \mathcal{L}_{\text{data}}}{\partial w_i} + \lambda w_i$$

So the **gradient of the regularization term** is:

$$\nabla_w (\lambda \cdot \frac{1}{2} \|w\|_2^2) = \lambda w$$

> The summation "disappears" in the gradient because the partial derivative of the sum isolates each (w_i).

🧲 What Is Weight Decay?

Weight decay is a technique that **shrinks weights during training** to prevent them from growing too large:

- It is **mathematically equivalent** to L2 regularization in **SGD**
- It modifies the gradient update rule:

$$w \leftarrow w - \eta \cdot (\nabla_w \mathcal{L}_{\text{data}} + \lambda w)$$

> (\lambda) controls how strongly weights are pulled toward zero.

🧨 The Problem with Adam + L2 Regularization

Adam rescales gradients based on **moving averages of squared gradients**.
So when you inject L2 into the gradient:

$$g_t = \nabla_w \mathcal{L}_{\text{data}} + \lambda w$$

It gets passed into the adaptive machinery (momentum & RMSProp):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Then:

$$w \leftarrow w - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$$

> ❌ The weight decay term (\lambda w) is **scaled and distorted**, so it no longer behaves like clean L2 shrinkage.

✅ AdamW: The Fix

AdamW **decouples** weight decay from the gradient:

1. Do a regular Adam step using only the gradient of the data loss
2. Apply weight decay **afterward**, directly to the weights:

$$w \leftarrow w - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \eta \cdot \lambda w$$

> ✅ This restores proper "pull toward zero" behavior, consistent with L2 regularization in SGD.

🆚 Adam vs AdamW

Feature	Adam + L2 Regularization	AdamW
Applies decay via	Gradient injection	Directly on weights post-update
Affected by gradient scaling?	✅ Yes	❌ No
Consistent with SGD/L2?	❌ No	✅ Yes
Generalization performance	Often worse	More reliable

🔗 References

- Loshchilov & Hutter (2019). Decoupled Weight Decay Regularization (https://arxiv.org/abs/1711.05101)
- PyTorch Optimizers: `AdamW` (https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html)

🔄 Batch Normalization (BatchNorm)

🧠 Why BatchNorm?

BatchNorm was introduced to solve a key training problem in deep networks:
internal covariate shift.

> Internal covariate shift = The distribution of inputs to each layer keeps changing during training because the layers before it are also updating.

This "shifting target" makes it harder for deeper layers to learn — like trying to hit a moving bullseye.

✅ What Does BatchNorm Do?

For each feature/channel in a layer, **per mini-batch**:

1. Compute batch mean and variance:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

2. Normalize the activations:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

3. Apply learned scale and shift:

$$y_i = \gamma \cdot \hat{x}_i + \beta$$

🧩 What Are γ and β?

- γ: learnable **scale** parameter
- β: learnable **shift** parameter

Without them, the layer output would always have mean 0 and variance 1 — limiting expressiveness.

> BatchNorm normalizes for training stability,
> but (\gamma) and (\beta) allow the network to **learn any useful distribution** again if needed.

🔀 Behavior During Training vs Inference

🏋️ Training Mode

- Compute mean and variance **from the current mini-batch**
- Normalization is noisy → helps regularize and stabilize training

🧪 Inference Mode

- Use **running averages** of mean and variance collected during training:

$$\mu_{\text{running}} \leftarrow \rho \cdot \mu_{\text{running}} + (1 - \rho) \cdot \mu_B$$

- Prevents randomness at test time
- Ensures consistent output even when batch size is 1

⚙️ Where to Use BatchNorm

Typically placed:

- After linear/convolution layers
- Before activation functions

Example:

```
nn.Sequential(
    nn.Linear(128, 64),
    nn.BatchNorm1d(64),
    nn.ReLU()
)
```

✅ Benefits of BatchNorm

- Allows **higher learning rates**
- Reduces sensitivity to initialization
- Mitigates vanishing/exploding gradients
- Adds mild **regularization effect** (from batch noise)
- Enables training of **very deep networks**

! Best Practices

- Always call `.eval()` before inference to switch BN to test mode
- Works best with reasonably sized mini-batches (e.g., ≥32)
- Be cautious when combining with Dropout — BatchNorm already acts as a regularizer

🔗 References

- Ioffe & Szegedy (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (https://arxiv.org/abs/1502.03167) ```

🧠 Layer Normalization (LayerNorm)

📌 What Is LayerNorm?

LayerNorm is a normalization technique that normalizes activations **across the features** of each individual sample.

Unlike BatchNorm, it does **not use batch statistics**, which makes it ideal for:

- Recurrent models (RNNs, LSTMs)
- Transformers (BERT, GPT, etc.)
- Situations with small or variable batch sizes
- Autoregressive generation (batch size = 1)

🔍 How Does It Work?

Given a vector of activations (x = [x_1, x_2, \dots, x_H]) for a single sample, LayerNorm computes:

Mean and variance over the features:

$$\mu = \frac{1}{H}\sum_{i=1}^{H} x_i, \quad \sigma^2 = \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2$$

Normalize:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Apply learnable scale and shift:

$$y_i = \gamma \cdot \hat{x}_i + \beta$$

Where:

- (\gamma): learnable scale parameter
- (\beta): learnable shift parameter
- (\epsilon): small constant to prevent division by zero

🧠 Why Normalize Across Features?

In NLP and Transformers:

- Each **token** is represented by a feature vector (embedding + positional info)
- You **don't want to normalize across tokens** — each token should be treated independently
- Instead, normalize **within each token**, across its features

> This preserves each token's position and semantic identity.

📐 Input Shapes and Normalization Axes

Input Shape	LayerNorm Normalizes Over
`(batch_size, features)`	The `features` axis (per sample)
`(batch_size, seq_len, embed_dim)`	The `embed_dim` axis (per token)

Each row or token vector is normalized **independently**.

🆚 LayerNorm vs BatchNorm

Feature	BatchNorm	LayerNorm
Normalizes over	Features **across** batch	Features **within** each sample
Requires batch stats	✅ Yes	❌ No
Consistent for batch size=1	❌ No	✅ Yes
Good for CNNs	✅ Yes	👎 Not common

Feature	BatchNorm	LayerNorm
Good for RNNs/Transformers	❌ No	✅ Yes
Used in GPT/BERT/etc	❌ No	✅ Yes
Needs `.train()` / `.eval()`	✅ Yes (depends on mode)	❌ No (mode-agnostic)

📦 Where Is LayerNorm Used?

- **MLPs**: `Linear → LayerNorm → Activation`
- **Transformers**: applied around residual blocks
  - **PreNorm**: `x + Sublayer(LayerNorm(x))`
  - **PostNorm**: `LayerNorm(x + Sublayer(x))`
- **RNNs / LSTMs**: normalize input and/or hidden state per time step

🧠 What's a "Token" in NLP?

- A **token** is a unit of text (word, subword, character, etc.)
- After embedding, each token becomes a **vector** (e.g., 768-dim)
- Positional encoding is **added** to the token vector
- That final vector is what gets normalized by LayerNorm

So yes — in NLP, a token is **represented as a feature vector**, and LayerNorm ensures that vector has:

- Mean ≈ 0
- Std ≈ 1
- ✅ Without touching or depending on other tokens or the batch

✅ Summary

- LayerNorm helps with stable, consistent training
- Doesn't depend on batch statistics
- Perfect for sequence models, small batches, and autoregressive decoding
- Normalizes **per token**, **across features**
- Keeps token-level information clean and separate

🔗 Reference

- Ba et al. (2016). Layer Normalization (https://arxiv.org/abs/1607.06450)

⚙️ Core Hyperparameters for LLMs / GenAI Models

These are the key hyperparameters that control training stability, convergence, and generalization in large models like Transformers and LLMs.

Hyperparameter	Why It Matters
Learning Rate	Most critical hyperparameter — controls step size of optimization. Use schedules (e.g. cosine, linear decay).
Batch Size	Affects gradient stability and memory use. Larger = smoother gradients. Smaller = noisier but faster updates.
Weight Decay	Penalizes large weights to reduce overfitting. Usually used with AdamW.
Dropout Rate	Randomly drops activations for regularization. Common values: 0.1–0.3
Gradient Clipping	Caps gradient norm to prevent instability/exploding gradients.
Warmup Steps	Gradually increase LR early in training to avoid overshooting.
LR Scheduler	Adjusts learning rate throughout training.
Max Sequence Length	Controls how much context the model sees. Higher = more memory/computation.

✂️ Gradient Clipping (Deep Dive)

🔍 What it does:

Clips the **L2 norm** of the total gradient **after** backpropagation and batch aggregation:

$$g = [g_1, g_2, \ldots, g_n], \quad \|g\|_2 = \sqrt{\sum_i g_i^2}$$

If $\|g\|_2 > \text{max\_norm}$, then:

$$g \leftarrow g \cdot \frac{\text{max\_norm}}{\|g\|_2}$$

✅ Intuition:

> Scale the "macro" gradient (aggregated over the batch), not each individual datapoint's gradient.

- Preserves gradient direction
- Controls gradient magnitude
- Prevents large, unstable weight updates

🧪 PyTorch example:

```
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

🧪 When and Why to Use:

🔷 Weight Decay (L2 Regularization)

- Shrinks weights over time: (\nabla_w \mathcal{L} + \lambda w)
- Encourages smoother, lower-capacity models
- ✅ Common with **AdamW** (decoupled implementation)

🔷 L1 Regularization

- Encourages **sparse** weights (many exactly zero)
- Not commonly used in LLMs
- ⚠️ Adds optimization challenges (non-differentiable at zero)

🔷 Dropout

- Randomly zeros out activations during training
- Acts like an ensemble of subnetworks
- ✅ Used in fully connected and attention layers

- ⚠️ Can conflict with BatchNorm (both inject noise)

🔷 BatchNorm

- Normalizes **across batch dimension**
- Helps CNNs and small feedforward nets train faster
- ❌ Not used in Transformers/LLMs (conflicts with small/variable batch sizes)

🔷 LayerNorm

- Normalizes **within each sample**, across features
- Independent of batch size
- ✅ Default normalization for **Transformers / LLMs**
- Used before or after residual connections

🧠 Summary: When to Use What

Technique	Use Case	When to Avoid
Dropout	FC layers, attention, FFNs	May conflict with BatchNorm
Weight Decay	Almost always (use AdamW)	Don't use naive L2 with Adam
L2 Regularization	SGD or AdamW	Not with Adam (use AdamW instead)
L1 Regularization	Sparse models / feature selection	Not typical for LLMs
BatchNorm	CNNs, ResNets	Not for sequences / Transformers
LayerNorm	Transformers, LLMs, RNNs	Rarely used in CNNs
Gradient Clipping	Transformers, deep models	Set to 1.0 or 0.5 (very safe default)

🔗 References

- Loshchilov & Hutter (2019). Decoupled Weight Decay Regularization (AdamW) (https://arxiv.org/abs/1711.05101)
- Ba et al. (2016). Layer Normalization (https://arxiv.org/abs/1607.06450)
- Ioffe & Szegedy (2015). BatchNorm (https://arxiv.org/abs/1502.03167)
- Srivastava et al. (2014). Dropout (https://jmlr.org/papers/v15/srivastava14a.html) ```

🔗 Regularization & Normalization Compatibility

✅ Good Combinations

Combo	Works Well?	Why
Dropout + Weight Decay	✅ Yes	Dropout adds noise during training, weight decay shrinks unused weights
BatchNorm + Weight Decay	✅ Yes	BatchNorm normalizes activations, weight decay still keeps weights small
LayerNorm + Dropout	✅ Yes	Common in Transformers — stable + regularized training
LayerNorm + Weight Decay	✅ Yes	Works perfectly, especially with AdamW
AdamW + Weight Decay	✅ Yes	Decoupled weight decay avoids distortion of gradient scaling

⚠️ Combos to Use with Caution

Combo	Warning	Why
Dropout + BatchNorm	⚠️ Be careful	Both add stochasticity → can conflict or cause instability
Dropout + Heavy Augmentation + Label Smoothing	⚠️ Can over-regularize	Too much noise may hurt convergence or underfit
L2 Regularization + Adam (not AdamW)	❌ Bad idea	L2 gets distorted by adaptive gradient scaling — use AdamW instead

❌ Combos to Avoid

Combo	Why Not?
BatchNorm + LayerNorm	Redundant — both normalize but along different axes, confusing signal
BatchNorm in Transformers	Needs batch stats — breaks in variable-length or autoregressive settings
L1 Regularization in LLMs	Too harsh for dense models, rarely needed outside sparse setups

✅ Recommended "Starter Stack" for LLMs / Transformers

Use this setup in modern LLM/Transformer training:

- **LayerNorm** for normalization (pre/post residual)
- **Dropout** after attention and feedforward blocks (e.g., 0.1–0.3)
- **Weight Decay** via **AdamW optimizer**
- **Gradient Clipping** to stabilize large updates (e.g., max norm = 1.0)
- **No BatchNorm**
- **No direct L2 regularization** (already handled via weight decay)

🔍 Weight Decay Reminder

With AdamW, weight decay is applied separately:

$$w \leftarrow w - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \eta \cdot \lambda w$$

This avoids distorting the adaptive gradient and ensures clean decay.

🔍 Gradient Clipping Reminder

Clip the **L2 norm** of the full gradient vector across all parameters:

If:

$$\|g\|_2 > \text{max\_norm}$$

Then rescale:

$$g \leftarrow g \cdot \frac{\text{max\_norm}}{\|g\|_2}$$

This stabilizes training, especially in deep networks and Transformers.

📊 Monitoring & Debugging Training with Curves

Understanding and monitoring **learning curves** is essential for training deep learning models effectively. These curves help identify issues like underfitting, overfitting, unstable training, or misconfigured hyperparameters.

📉 What to Track During Training

Curve Type	Description	Purpose
Training Loss	Error on the training data	Measures how well the model is fitting
Validation Loss	Error on unseen data (validation set)	Measures generalization
Training Accuracy	Classification accuracy on training set	Useful for classification tasks
Validation Accuracy	Accuracy on validation set	Monitors real performance

🧠 What to Look For

✅ Healthy Training

- **Train and Val loss both decrease** and plateau gradually
- **Validation accuracy increases**
- Generalization gap is **small or steady**

🚨 Overfitting

- **Training loss keeps going down**
- **Validation loss goes up**
- **Training accuracy >> Validation accuracy**

✅ Fix:

- Add **Dropout**
- Add **Weight Decay**
- Use **Data Augmentation**
- Apply **Early Stopping**

⚠️ Underfitting

- **Both losses remain high**
- Model struggles to fit training data

✅ Fix:

- Increase model capacity (depth, width)
- Increase learning rate
- Improve data preprocessing

📐 Generalization Gap

$$\text{Generalization Gap} = \text{Validation Loss} - \text{Training Loss}$$

- Small gap → model generalizes well ✅
- Large gap → overfitting 🚨
- Shrinking gap → improvements in generalization 🧠

🔁 Curve Usage by Task

Task Type	Curve to Prioritize	Notes
Classification	Accuracy + Loss curves	Accuracy is easy to interpret
Regression	Loss curves (MSE, MAE)	Accuracy is not meaningful
Sequence models	Loss curves + BLEU/ROUGE	Use task-specific metrics as needed

🧪 Tools for Tracking

- **TensorBoard**: Visualize loss/accuracy in real time
- **Weights & Biases (wandb)**: Powerful experiment tracking
- **Matplotlib / Seaborn**: Local plotting for custom training scripts

🧠 Final Reminders

- Always compare **Training vs Validation**
- Look at **curves over time**, not just final metrics
- Use them to guide **early stopping**, **scheduler triggers**, and **regularization tuning**

📉 Deep Learning Curve Interpretation — Advanced Tips

1️⃣ Noisy Loss Curves? → Check Learning Rate or Batch Size

If your loss curve is **bouncy or erratic**:

- Your learning rate may be too high
- Your batch size may be too small
- Data pipeline may not be shuffling properly

✅ Fix:

- Smooth with moving average
- Lower learning rate
- Increase batch size

2️⃣ Plateaued Training Loss Early? → LR May Be Too Low or Layers Still Frozen

If training loss **flattens too early**, but you suspect underfitting:

- Learning rate might be too low
- In transfer learning, some layers might still be frozen

✅ Fix:

- Use a **learning rate finder**
- Gradually unfreeze layers during training

3️⃣ Accuracy vs. Loss (Especially for Classification)

Sometimes:

- **Validation loss increases** while
- **Validation accuracy still improves**

This can happen when:

- The model gets more confident (sharper softmax), which hurts loss
- But predictions are more often correct

✅ Plot both **accuracy** and **loss** to get the full picture

4️⃣ Generalization Gap

Define:

$$\text{Gap} = \text{Validation Loss} - \text{Training Loss}$$

- Small gap → model generalizes well ✅
- Large gap → model is overfitting 🧯
- Gap growing over time → model needs more regularization

✅ Use:

- Dropout
- Weight decay
- Data augmentation
- Early stopping

5️⃣ Watch Out for Divergence or NaNs

If you suddenly see:

- Loss = NaN
- Loss spikes massively
- Accuracy goes to 0 or 100%

It could mean:

- Exploding gradients

- Numerical instability (division by zero, log(0), etc.)

✅ Fix:

- Use **gradient clipping**
- Reduce learning rate
- Check for numerical bugs (especially with custom layers or mixed precision)

6️⃣ Compare Across Experiments

Use experiment tracking tools like:

- **TensorBoard**
- **Weights & Biases (wandb)**
- Save logs + versioned runs

✅ Look at curves **side-by-side** to validate improvements

7️⃣ Don't Stop Too Early — Deep Models Take Time

- Just because training loss plateaus doesn't mean the model is done learning
- LLMs and deep nets often need:
  - LR decay (cosine, linear, step)
  - Patience
  - 10s or 100s of epochs to converge

✅ Combine **early stopping** with **LR scheduling** for best results

---

🧠 Final Tip: Don't Chase Zero Training Loss

- It's not the goal
- A model that fits the training data **perfectly** might generalize **poorly**
- Instead:
  - Optimize for **low validation loss**
  - Minimize generalization gap
  - Monitor **both accuracy and loss curves**

In [ ]: