MODULE-2

Regularization for Deep Learning: Parameter Norm Penalties, Norm Penalties as Constrained Optimization, Regularization and Under-Constrained Problems, Dataset Augmentation, Noise Robustness, Semi- Supervised Learning, Multi-Task Learning, Early Stopping, Parameter Tying and Parameter Sharing, Sparse Representations.

Optimization for Training Deep Models: How Learning Differs from Pure Optimization, Basic Algorithms. Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates.

REGULARIZATION FOR DEEP LEARNING

- It's a method to prevent overfitting — when your model memorizes the training data too well but fails to generalize to new data.
- Main idea is to limit the complexity of the model.

PARAMETER NORM PENALTIES

It is a technique used to prevent neural networks from becoming too complex or overfitting the training data.

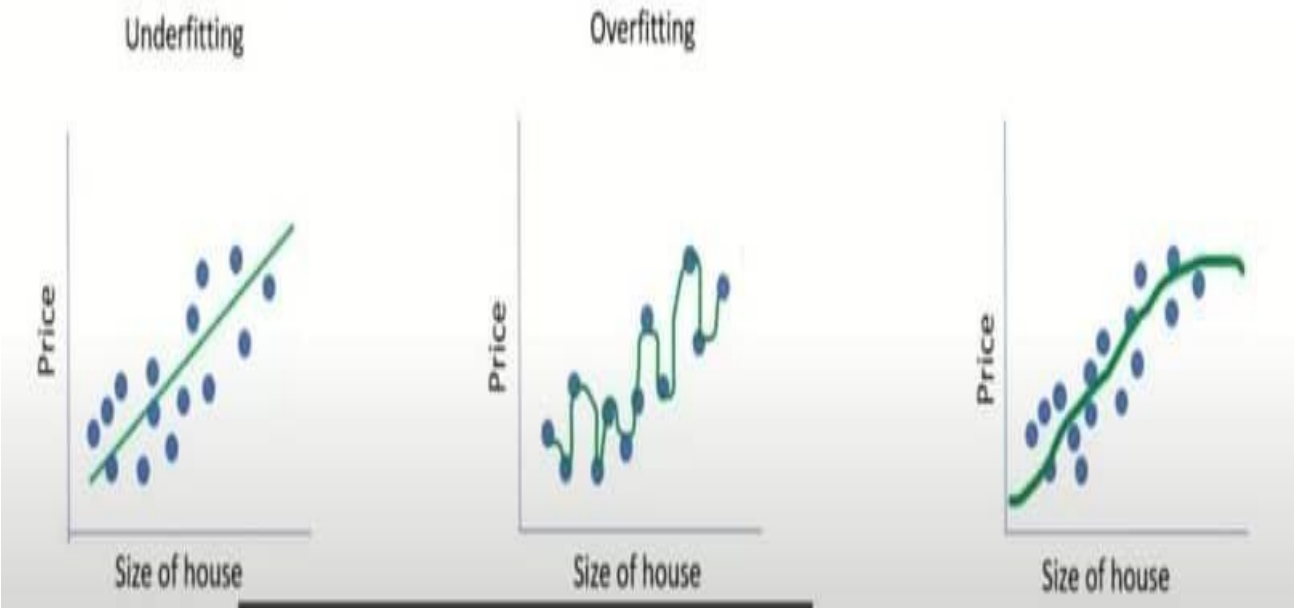
PARAMETERS: These are weights and biases associated with each connection between neurons.

A model's complexity refers to how "flexible" it is in fitting data.

- A high complexity model can wiggle a lot to fit even noisy training data (risk of overfitting).
- A low complexity model is more rigid and captures only the main trend (risk of underfitting).

How do weights relate to complexity?

- Large weights (w) → the model output changes a lot for small changes in input.
    - Example: In linear regression y=w1x, if w1=100 even tiny noise in x makes predictions jump drastically.
- Small weights → smoother predictions, less sensitivity to noise.
- Big weights = more complex model, small weights = simpler model.



The **original objective function** $J(\theta; X, y)$ is basically the **measure of how well your model is doing** on the training data **before** adding any regularization.

Here's what each part means:

- θ → the parameters of your model (e.g., weights in linear regression or a neural network)
- X → the input data (features)
- y → the true target values (labels)
- J → a function that calculates the "loss" (error) between the model's predictions and the true values

---

Examples:

- **Linear Regression:**

$$J(\theta; X, y) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

↓

Here, J is the **Mean Squared Error** (MSE).

◆ Example 2: Student exam score

Let's predict a student's exam score using:

- x_1: hours studied
- x_2: hours spent on phone

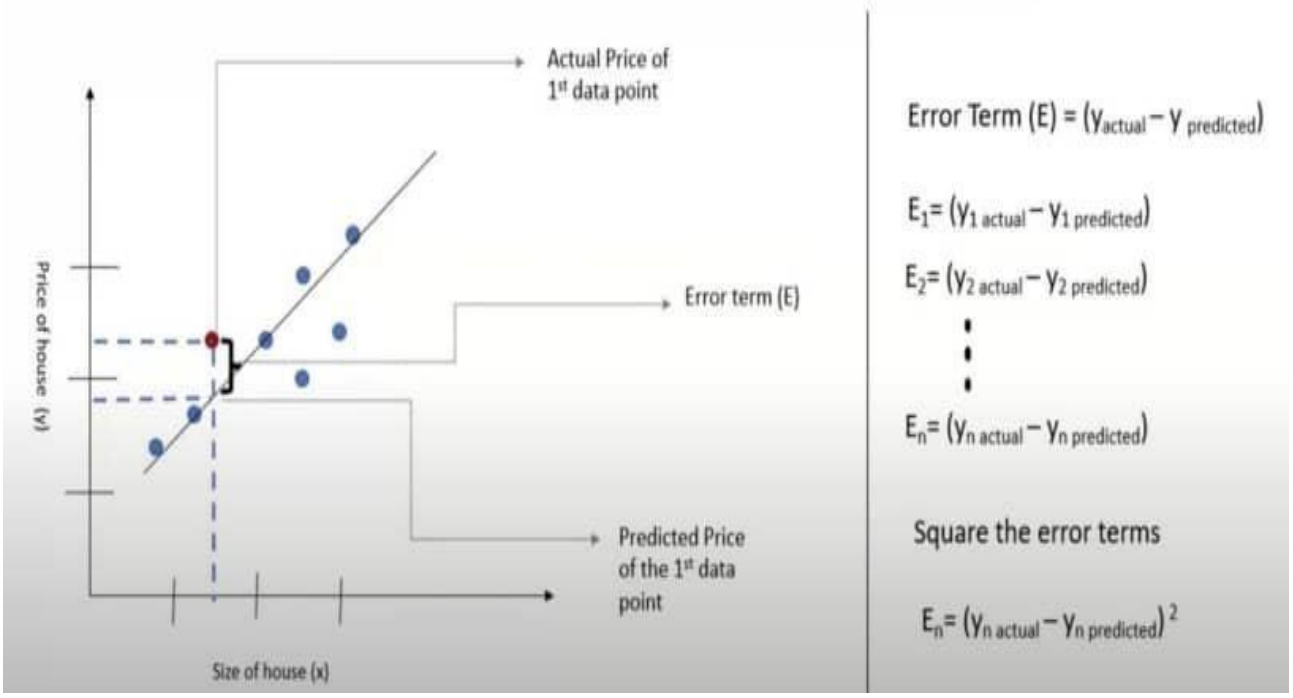
Model:

$$\text{score} = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

Suppose:

- $w_1 = +5$ → every extra study hour **adds 5 points**.
- $w_2 = -2$ → every extra phone hour **reduces 2 points**.

👉 Here, weights act like **importance multipliers** that increase or decrease the output.



- Before regularization: J focuses only on fitting the data well.
- After regularization: J = fit the data well + keep parameters small to avoid overfitting.

How This Works

You start with your **original objective function** $J(\theta; X, y)$, which measures how well the model fits the training data.

- Example: In linear regression, J could be the mean squared error.

You then **add a penalty term** $\Omega(\theta)$ that depends on the size of the model's parameters θ.

- Example: $\Omega(\theta)$ could be the sum of squared weights.

- Penalty term means how large the model's weights (parameters)are and penalize big values in the loss function.

Why measure the size of parameters?

- Large weights mean the model is very sensitive to small changes in input → it can memorize noise in training data (overfitting).
- If we keep weights smaller, the model becomes simpler and more likely to generalize well.

It's like telling the model:

"Not only should you fit the data well, but also, don't use huge numbers for your weights. If you do, I'll increase your loss."

3. Mathematical Form

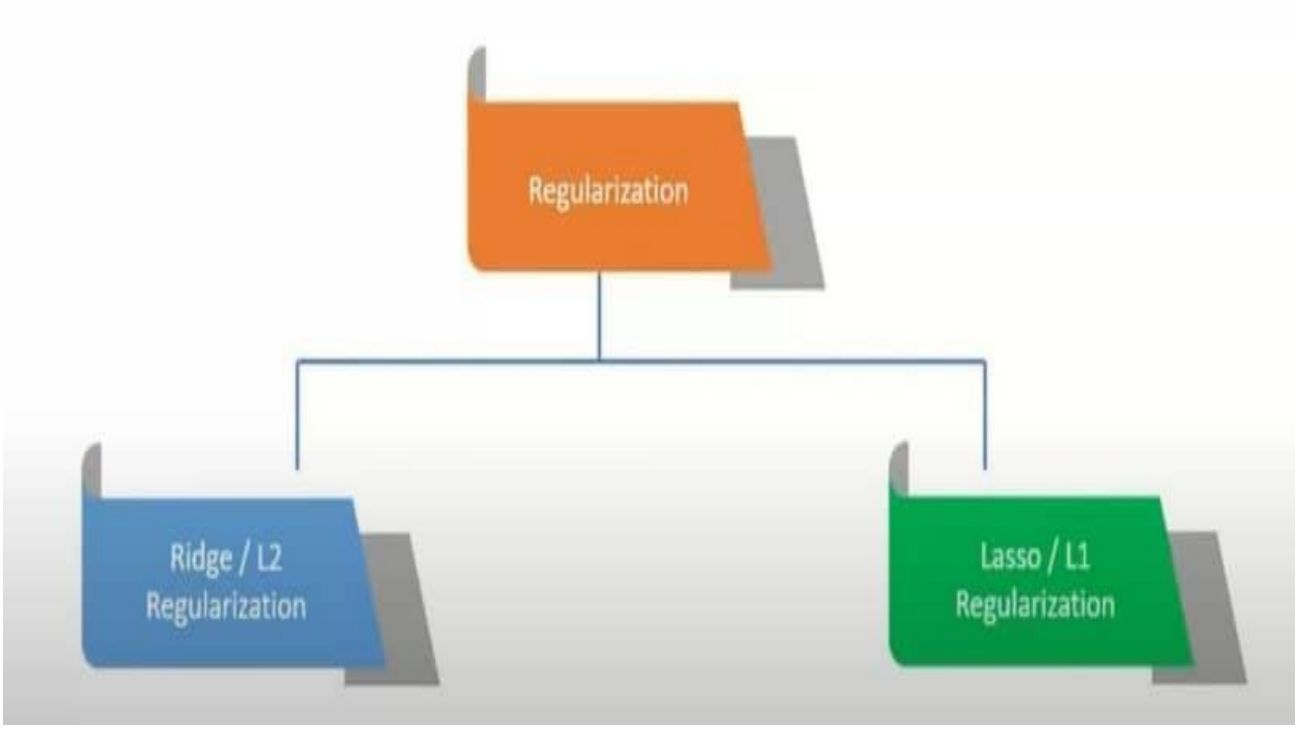
The **new objective function** becomes:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\,\Omega(\theta)$$

- $J(\theta; X, y)$: Original loss (data fitting part)
- $\Omega(\theta)$: Parameter norm penalty (keeps weights small)
- α: **Hyperparameter** controlling how much penalty matters
  - $\alpha = 0$ → no regularization
  - Larger α → stronger regularization (simpler model)

Why this helps:

- Large weights make models **more complex** and prone to overfitting.
- Adding the penalty discourages large weights → simpler models → better generalization.



L^2 Parameter Regularization

1. What it is

- **L^2 regularization = weight decay = Ridge regression** (in linear models) = **Tikhonov regularization.**
- You add a penalty term:

$$\Omega(\theta) = \frac{1}{2}\|w\|_2^2$$

to the normal objective function.

This encourages weights (w) to be **small**, pushing them toward the origin.

- Objective function /Loss Function
- J(W)=it measures how well the model fits the data and also minimizes by adjusting the weights W

◆ Adding the Penalty Term

Now, we add an extra term:

$$\Omega(\theta) = \frac{1}{2}\|w\|_2^2 = \frac{1}{2}\sum_i w_i^2$$

- $\|w\|_2^2$ = sum of squared weights (the L^2 norm squared).
- The factor $\frac{1}{2}$ is just for math convenience (it cancels when taking derivatives).

$$\tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y)$$

◆ **Terms Explained:**

1. $J(w; X, y)$
   - This is the **original loss function** (e.g., Mean Squared Error for regression, Cross-Entropy for classification).
   - It measures how well your model predicts the training data.
2. $w^\top w$
   - This is the **squared L^2 norm of the weights** (sum of squares of all weights).
   - Example: If $w = [w_1, w_2]$, then $w^\top w = w_1^2 + w_2^2$.
   - It represents the "size" (magnitude) of the weights.

3. $\frac{\alpha}{2} w^\top w$
   - This is the **penalty term** added to discourage large weights.
   - α controls how strong the penalty is.
   - Dividing by 2 is just for convenience when differentiating.
4. $\tilde{J}(w; X, y)$
   - This is the **new, regularized loss function**.
   - It balances two things:
     - Fit the data well ($J(w; X, y)$).
     - Keep weights small ($\frac{\alpha}{2} w^\top w$).

So the new **regularized loss function** becomes:

$$\tilde{J}(w) = J(w) + \alpha \cdot \frac{1}{2}\|w\|_2^2$$

Gradient=Slope

- If I change this weight a little, how much will the loss (error) go up or down?

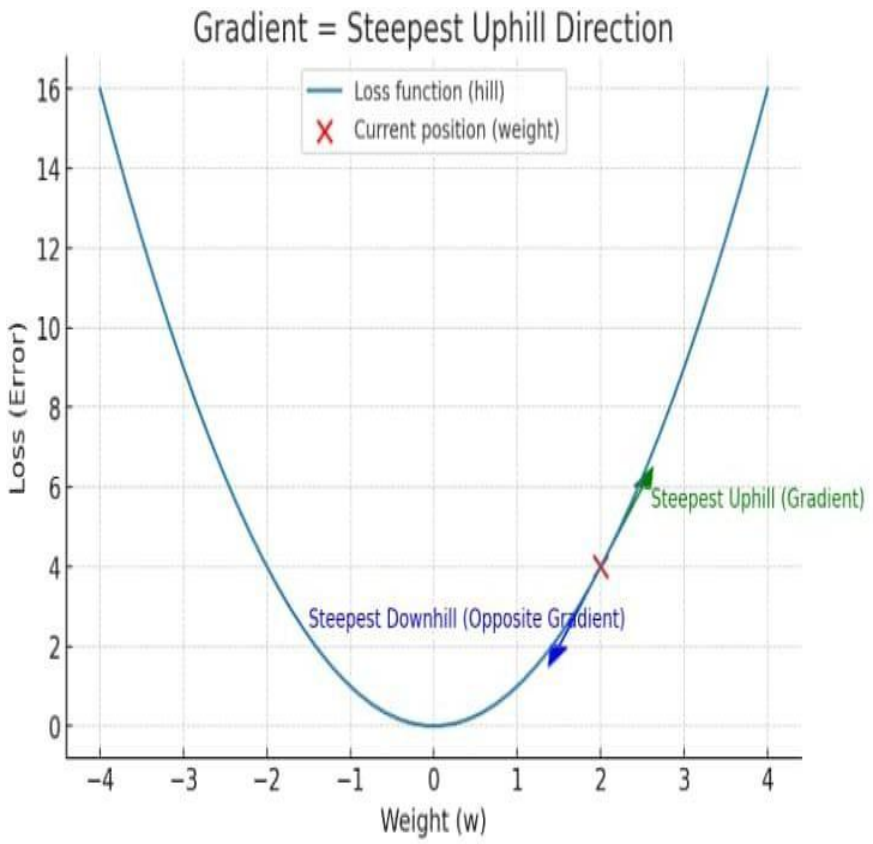
Imagine climbing a hill

- You're standing on the side of a hill.
- If you look around, you can walk in many directions.
- Some directions are almost flat → you won't climb much.
- But one direction is the **steepest uphill** → that's the direction where the hill rises the fastest.

That direction = **gradient**.

In ML terms

- The "hill" = **loss function (error)**.
- The **steepest uphill** = direction where the error increases fastest.
- But since we want to **reduce error**, we walk **opposite** to that direction (downhill).



- The **U-shape curve** is the "hill" (loss function).
- The **red dot** is where your model currently is.
- **Green arrow** = steepest uphill (gradient).
- **Blue arrow** = steepest downhill (opposite gradient → what we follow in gradient descent).

This is why we always move **downhill** to reduce error.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

with the corresponding parameter gradient

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

Step 1: Recall what derivative (gradient) means

- Taking derivative means: **find how the function changes when we change the weights.**
- For vectors, this is called a **gradient** (∇).

Step 2: Break into two parts

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \nabla_w \left(\frac{\alpha}{2} w^\top w\right)$$

- First term: $\nabla_w J(w; X, y)$ → already known from the original loss.
- Second term: derivative of regularization part.

Step 3: Focus on $\frac{\alpha}{2} w^\top w$

- $w^\top w = w_1^2 + w_2^2 + \ldots + w_n^2$ (sum of squares of all weights).
- Multiply by $\frac{\alpha}{2}$:

$$\frac{\alpha}{2}(w_1^2 + w_2^2 + \ldots + w_n^2)$$

Step 4: Derivative of each term

- Derivative of w_i^2 w.r.t. w_i = $2w_i$.
- So derivative of $\frac{\alpha}{2} w_i^2$ = αw_i.
- Collecting for all i:

$$\nabla_w \left(\frac{\alpha}{2} w^\top w\right) = \alpha w$$

Step 5: Final result

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \alpha w$$

(7) Take a small step.

If current weights are w, we update them like this,

$$w \leftarrow w - \epsilon \cdot \nabla_w \tilde{J}(w; x, y)$$

where, ϵ (learning rate) = step size (how big a move we take)

minus sign ensures we go downhill.

This is the [standard gradient descent update rule]

so,

$$w \leftarrow w - \epsilon(\alpha w + \nabla_w J(w; x, y))$$

$-\epsilon \nabla_w J(w; x, y)$ = [Gradient descent] fit data

$-\epsilon \alpha w$ = [shrink weights towards 0] weight decay.

Take a gradient descent step, but also reduce weights a little each time.

$$w \leftarrow w - \epsilon \alpha w - \epsilon \nabla_w J(w; x, y)$$

$$w \leftarrow \underbrace{(1 - \epsilon \alpha)}\, w - \epsilon \nabla_w J(w; x, y)$$

Shrink the weight a little bit every step.

Why do we need to do this?
Imagine you are packing a school bag.
Gradient descent :- putting in all the books you think you need
Regularization: Someone reminds you "don't carry unnecessary heavy books" → so you reduce extra weight.

$(1-\epsilon\alpha)$
↳ It prevents weights from growing too big.
Keep the model simpler
helps in avoid overfitting.

When we add L^2 penalty, the gradient picks up an extra term αw, which pulls weights toward zero each update.

Objective Function (Loss Function)

- In ML/DL, we have a **loss function** J(w).
- w = weights (the numbers our model learns).
- J(w) = how wrong the model is for those weights. Goal: **find weights that make J(w) as small as possible**.

2 Minimum Point w^*

- $w^* = \arg\min_w J(w)$ means:
  → The weight value w^* that gives the **smallest loss**.
- Think of it like:
  - Throwing a ball into a bowl
  - The ball settles at the **lowest point** (that's w^*).

- w∗ is the **special weight value** where the loss J(w) is the **lowest possible**.

In math, we write:

$$w^* = \arg\min_w J(w)$$

; just means: *"the weights w that give the minimum loss."*

Neighborhood Around w∗

- Near this minimum point, the curve looks like a **U-shape (parabola)**.
- So instead of studying the full complicated curve, we can say: "Close to the minimum, the loss function behaves like a simple **quadratic (squared) function**."

That's why we do a **quadratic approximation**.

Why Quadratic Approximation?

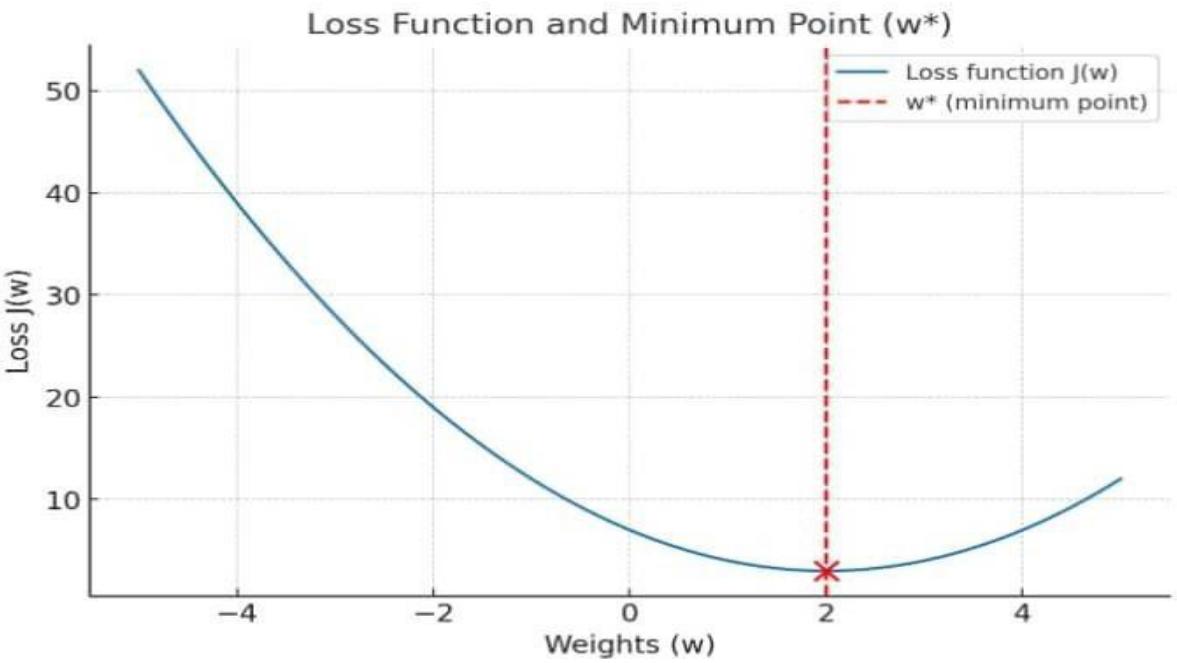
- Because quadratic functions are **easy to work with** (they're just like ax2+bx+c).
- Around w*, this approximation is good enough to analyze training.

Example:

If your real loss curve is wiggly, near the bottom it still looks like a smooth **U-shape**.

<u>simple terms:</u>

Instead of dealing with a messy loss curve everywhere, we zoom in near the **best weights w***. There, the loss function looks like a **parabola (quadratic)**. We use that simpler quadratic shape to analyze and understand training.



Here's the **loss function plot**:

- The **blue curve** is the loss J(w).
- The **red dot** at w∗=2 is the **minimum point**.
- This means: when the weight w is **2**, the loss is **lowest possible** (best fit for data).

The phrase **"quadratic approximation near w*"** means: Close to this minimum, the curve looks like a parabola (U-shape). So, instead of studying the full complicated function, we approximate it with this nice parabola around w*.

If the objective function is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect.

1. Objective function

In ML, the **objective function** (or **loss function**) is what we want to minimize.

Example: in **linear regression**, the loss is **Mean Squared Error (MSE)**, which is a **quadratic function** (a parabola).

$$J(w) = \frac{1}{N}\sum_i (y_i - wx_i)^2$$

This is already a quadratic in w.

2. Approximation

When a function is complicated, we often **approximate** it with a simpler function that is "close enough" near the minimum w^*.

We use a **quadratic approximation** because near the minimum, most functions look like a bowl (parabola).

$$\hat{J}(w) \approx J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

This is just a parabola (quadratic shape).

3. "If the function is truly quadratic..."

- If $J(w)$ is already a parabola (like MSE in linear regression),
  then the **approximation = the actual function.**
- That's why they say the approximation is **perfect** → because we didn't simplify, we just wrote the exact same parabola again.

The approximation Ĵ is given by,

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

Here:

- $J(w^*)$ = the value of the loss at the minimum.
- H = Hessian matrix (captures the curvature, i.e., how steep or flat the loss surface is).
- $(w - w^*)$ = how far you are from the minimum.
- The "hat" $\hat{J}$ means this is the **approximate version** of J.

The **Hessian** is a matrix of second derivatives of the loss function $J(w)$.

Second derivative tells us about the **curvature** (bending) of the function:

- Is it curving **upward** (like a smile 😃 bowl)?
- Or curving **downward** (like a frown ☹️ hill)?

What does "positive semidefinite" mean?

It's math language to say: the function **does not bend downward** at that point.

In simple terms: all directions around the minimum are either **flat** or **curving upward**.

So the shape is like a **bowl** (never an upside-down bowl).

Saying **"H is positive semidefinite"** is just a fancy way of saying: *Around the best weights w*, the loss function curves like a bowl (upward), so we're really at a minimum, not at a maximum or weird flat point.*

- There is no first-order term in this quadratic approximation, because $w*$ is defined to be a minimum, where the gradient vanishes.

We are talking about the gradient (slope) of the quadratic approximation of the loss function.

The formula is:

$$\nabla_w \hat{J}(w) = H(w - w^*)$$

Step-by-step meaning:

1. w^* → this is the "best weight" (the point where the loss is smallest).
2. $w - w^*$ → this means "how far are you from the best weight."
   - If you are exactly at w^*, then $w - w^* = 0$.
3. H → this is the Hessian, which tells us how curved the function is (like how steep the bowl is).
4. **Gradient** $\nabla_w \hat{J}(w)$ → this is the slope.
   - It's basically: **slope = curvature × distance from the minimum.**

Now what happens?

1. **If the marble is at the bottom ($w = w^*$)**
   - It doesn't move.
   - Gradient = 0 (no slope).
2. **If the marble is away from the bottom**
   - The slope pushes it back toward the center.
   - The farther the marble is, the stronger the pull.
   - This is what $H(w - w^*)$ means → **curvature × distance from the bottom.**
3. **The Hessian H**
   - Decides how steep the bowl is.
   - A very steep bowl = marble rolls back fast.
   - A flat bowl = marble rolls back slowly.

👉 So the equation is just saying:

Gradient (force pushing the marble) = Steepness of bowl × Distance from center.

1. **Without weight decay**

   Earlier we had:

$$\nabla_w \hat{J}(w) = H(w - w^*)$$

   The minimum happens at $w = w^*$.

2. **With weight decay**

   Weight decay adds a penalty term (αw) to discourage large weights.

   So the equation becomes:

$$\alpha \tilde{w} + H(\tilde{w} - w^*) = 0$$

   👉 Here, $\tilde{w}$ is the new optimal weight after adding regularization.

Step 3: Take derivative (gradient) w.r.t. w

The gradient of $\hat{J}_{\text{reg}}(w)$ is:

$$\nabla \hat{J}_{\text{reg}}(w) = H(w - w^*) + \alpha w$$

Step 4: Set gradient = 0 (minimum condition)

At the minimum, gradient must vanish:

$$H(\tilde{w} - w^*) + \alpha \tilde{w} = 0$$

👉 This is the equation you saw (Eq. 7.8).

Step 5: Rearrange terms

Move things around:

$$H\tilde{w} - Hw^* + \alpha \tilde{w} = 0$$

$$(H + \alpha I)\tilde{w} = Hw^*$$

👉 That's Eq. 7.9.

Step 6: Solve for $\tilde{w}$

Multiply both sides by $(H + \alpha I)^{-1}$:

$$\tilde{w} = (H + \alpha I)^{-1} H w^*$$

Step 2: How do we "free" w?

Think of it like simple algebra:

- If you have $3x = 12$, you divide both sides by 3 → $x = 12/3 = 4$.
- In matrix algebra, instead of dividing, we multiply by the **inverse matrix**.

So:

$$(H + \alpha I)w = Hw^*$$

Multiply both sides by $(H + \alpha I)^{-1}$ (the inverse of the matrix $H + \alpha I$):

$$(H + \alpha I)^{-1}(H + \alpha I)w = (H + \alpha I)^{-1}Hw^*$$

Step 3: Simplify

- On the **left side**, $(H + \alpha I)^{-1}(H + \alpha I) = I$ (identity matrix).
- And $Iw = w$.

So it becomes:

$$w = (H + \alpha I)^{-1}Hw^*$$

As α approaches 0, the regularized solution $\tilde{\boldsymbol{w}}$ approaches $\boldsymbol{w}^*$. But what happens as α grows? Because $\boldsymbol{H}$ is real and symmetric, we can decompose it into a diagonal matrix $\boldsymbol{\Lambda}$ and an orthonormal basis of eigenvectors, $\boldsymbol{Q}$, such that $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Applying the decomposition to equation 7.10, we obtain:

Step 1: Eigen-decomposition of H

- H is **real and symmetric** → it can be decomposed as:

$$H = Q\Lambda Q^\top$$

  - Q = orthonormal matrix of eigenvectors (like "rotation").
  - Λ = diagonal matrix of eigenvalues.
  - $Q^\top = Q^{-1}$ (since Q is orthonormal).

So instead of working with a messy H, we work with Λ (which is diagonal and easy to handle).

Step 2: Plug decomposition into the main equation

Substitute $H = Q\Lambda Q^\top$:

$$\tilde{w} = (Q\Lambda Q^\top + \alpha I)^{-1} Q\Lambda Q^\top w^*.$$

This is equation **(7.11)** in your image.

Step 3: Factorization trick

Notice:

$$Q\Lambda Q^\top + \alpha I = Q(\Lambda + \alpha I)Q^\top.$$

This is true because multiplying by Q and $Q^\top$ just rotates the diagonal matrix Λ.

So equation becomes:

$$\tilde{w} = \left[Q(\Lambda + \alpha I)Q^\top\right]^{-1} Q\Lambda Q^\top w^*$$

Step 4: Inverse of a rotated diagonal

Since Q is orthonormal,

$$\left[Q(\Lambda + \alpha I)Q^\top\right]^{-1} = Q(\Lambda + \alpha I)^{-1} Q^\top.$$

So:

$$\tilde{w} = Q(\Lambda + \alpha I)^{-1} Q^\top Q\Lambda Q^\top w^*$$

But $Q^\top Q = I$.

So it simplifies to:

$$\tilde{w} = Q(\Lambda + \alpha I)^{-1} \Lambda Q^\top w^*.$$

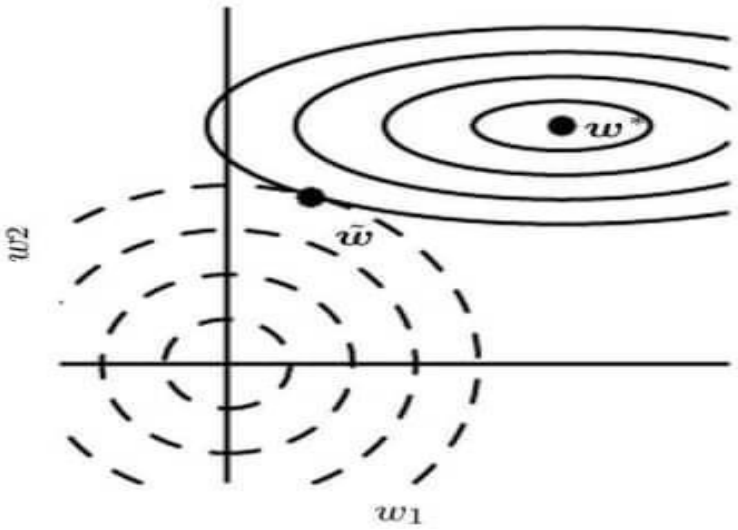
That's equation **(7.13)**.

Step 5: Why is this useful?

- Because Λ is diagonal, $(\Lambda + \alpha I)^{-1}$ is also diagonal and super easy to compute (just take reciprocal of each diagonal element).
- This shows how **regularization** (the α) **shrinks weights along eigen-directions.**
  - If eigenvalue is λ, shrink factor is $\frac{\lambda}{\lambda+\alpha}$.
  - As $\alpha \to 0$, shrink factor → 1, so $\tilde{w} \to w^*$.
  - As $\alpha \to \infty$, shrink factor → 0, so $\tilde{w} \to 0$.

Whole derivation is just:

- Start with quadratic approx.
- Add weight decay penalty.
- Take derivative.
- Set gradient = 0.
- Rearrange → gives the formula.



Solid ellipses (the oval shapes)

- Think of these as the "hills" or "valleys" of your original loss function (without regularization).
- The smallest ellipse (centered at w^* (is the **best place** the model found if we don't use regularization.
- Moving away from w^*makes the loss larger, but **not equally in all directions**:
  - Along w_1)horizontal), the ellipses are stretched → the loss doesn't change much (flat).
  - Along w_2)vertical), the ellipses are narrow → the loss changes quickly (steep).

Dotted circles (the round shapes)

- These represent the **L² penalty** (weight decay).
- The penalty cares only about the **distance from the origin (0,0)**, not about the shape of the ellipses.
- That's why they are circles — all directions are treated equally by the regularizer.
- The "goal" of the regularizer is to **pull weights toward zero** (the origin).

Two directions: w_1 vs w_2

- Along the w_1-**axis** (horizontal):
  - The ellipses are **wide** → means the loss does not change much if we move left or right.
  - In math: the Hessian eigenvalue is **small** → "flat" direction.
  - Because the loss doesn't care much, the regularizer dominates and pulls w_1 strongly toward 0.
- Along the w_2-**axis** (vertical):
  - The ellipses are **narrow** → means the loss increases quickly if we move up or down.
  - In math: the Hessian eigenvalue is **large** → "steep" direction.
  - Here, the loss dominates and the regularizer has little effect on w_2.

Weight decay shrinks parameters more in directions where the loss function is flat (low curvature), and shrinks less in directions where the loss is steep (high curvature).

L1 REGULARIZATION

While L2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L1 regularization.

Formally, L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as:

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|,$$

that is, as the sum of absolute values of the individual parameters.[2]

This is about **L¹ weight decay** (also called **Lasso regularization**).

- Just like L² weight decay adds a penalty $\alpha\|w\|_2^2$,
- L¹ weight decay adds a penalty $\alpha\|w\|_1$.

👉 $\|w\|_1$ just means:

$$\|w\|_1 = |w_1| + |w_2| + \cdots + |w_n|$$

(sum of absolute values of all weights).

So the new loss function is:

$$\tilde{J}(w; X, y) = \alpha\|w\|_1 + J(w; X, y)$$

- $J(w; X, y)$: the original loss (e.g., mean squared error, cross-entropy).
- $\alpha\|w\|_1$: the **regularization penalty**. ↓
- α: controls how strong the penalty is.

Why use L¹ regularization?

L¹ regularization tends to make many weights exactly **zero**.

This means it performs **feature selection** (keeping only the most important inputs).

That's why L¹ is used in sparse models like Lasso regression.

What about the gradient?

ıe gradient (technically, a **subgradient** because of the absolute value) is:

$$\nabla_w \tilde{J}(w; X, y) = \alpha \operatorname{sign}(w) + \nabla_w J(X, y; w)$$

- $\nabla_w J(X, y; w)$: normal gradient of the original loss.
- $\alpha \operatorname{sign}(w)$: extra "push" from the L^1 penalty.

- What is $\operatorname{sign}(w)$?

- If $w > 0$, $\operatorname{sign}(w) = +1$.
- If $w < 0$, $\operatorname{sign}(w) = -1$.
- If $w = 0$, it's undefined, but we can choose any value between $[-1, 1]$ (that's why we call it a **subgradient**).

Step 3: Derivative of $\|w\|_1$

$$\|w\|_1 = |w_1| + |w_2| + \cdots + |w_n|$$

- Derivative of $|w|$ is **sign(w)**:

$$\frac{d}{dw}|w| = \begin{cases} +1 & \text{if } w > 0 \\ -1 & \text{if } w < 0 \\ \text{undefined (any value between -1 and 1)} & \text{if } w = 0 \end{cases}$$

So:

$$\nabla_w \|w\|_1 = \operatorname{sign}(w)$$

$\nabla_w J(w; X, y)$: tells weights how to reduce prediction error.

$\alpha\,\mathrm{sign}(w)$: tells weights to **move toward zero**.

- If $w > 0$, subtract α → weight decreases.
- If $w < 0$, add α → weight increases toward zero.
- If $w = 0$, gradient is uncertain → model might leave it at zero → sparsity.

That's why L^1 makes many weights exactly zero.

They are comparing L^1 vs L^2 regularization

L^2 **(Ridge)**: Gradient contribution is proportional to w.

$$\nabla_w(\|w\|_2^2) = 2w$$

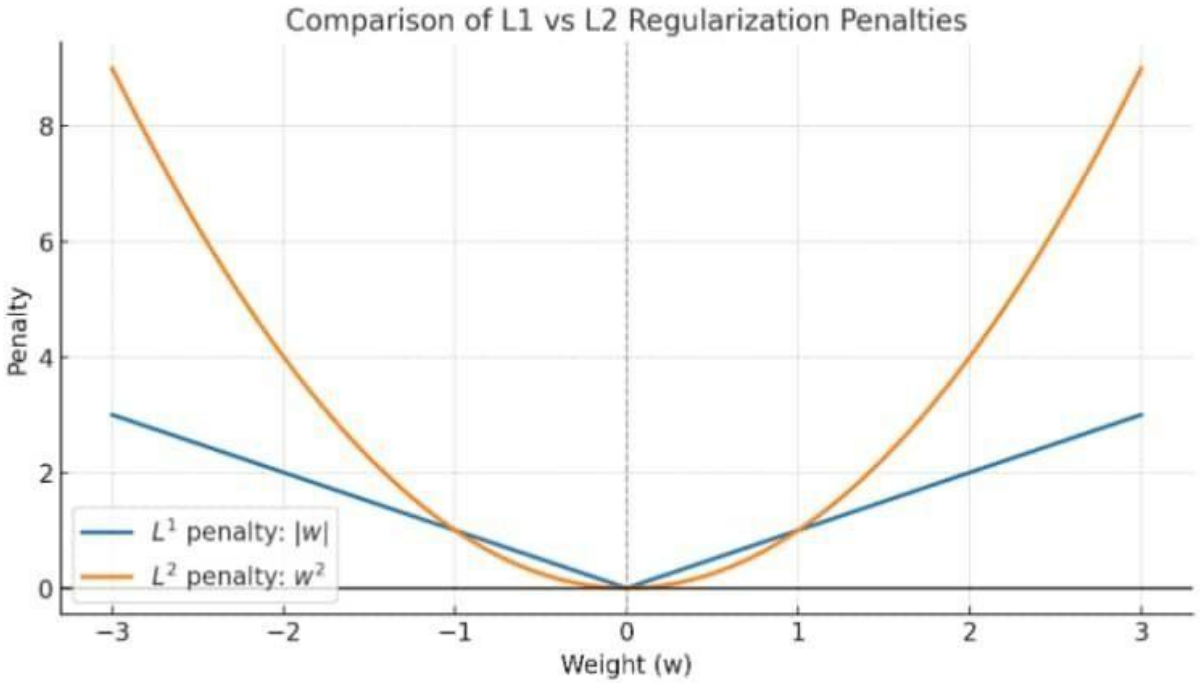
→ Bigger weights → bigger penalty.

L^1 **(Lasso)**: Gradient contribution is just **sign(w)** (±1 or 0).

$$\nabla_w(\|w\|_1) = \mathrm{sign}(w)$$

→ Always a **constant push toward zero**, no matter how big or small the weight is.

- That's why L^1 leads to sparsity (some weights go exactly to zero), while L^2 just shrinks weights smoothly.



3. The quadratic cost function + Taylor series

They are simplifying the **loss function** $J(w)$ near its minimum using a quadratic approximation (like a parabola).

This is a **Taylor expansion** around the optimal weights w^*:

$$J(w) \approx J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

Here:

- H = Hessian (matrix of 2nd derivatives = curvature of the loss).
- w^* = optimal weights without regularization.

4. Gradient of this quadratic approximation

Take derivative wrt w:

$$\nabla_w \hat{J}(w) = H(w - w^*)$$

This is equation (7.21).

- If $w = w^*$, gradient = 0 (minimum).
- If w is away from w^*, the gradient pulls it back, scaled by the curvature H.

✅ **In simple words:**

- L^2 pushes weights down smoothly in proportion to their size.
- L^1 pushes weights toward zero with a constant force (regardless of size).
- Because of that, L^1 doesn't have neat algebraic solutions → needs iterative methods.
- To analyze it, they approximate the cost function as a quadratic (parabola) around the minimum, where gradient = $H(w - w^*)$. ↓

L1and L2

<u>L2 Regularization (Weight Decay)</u>

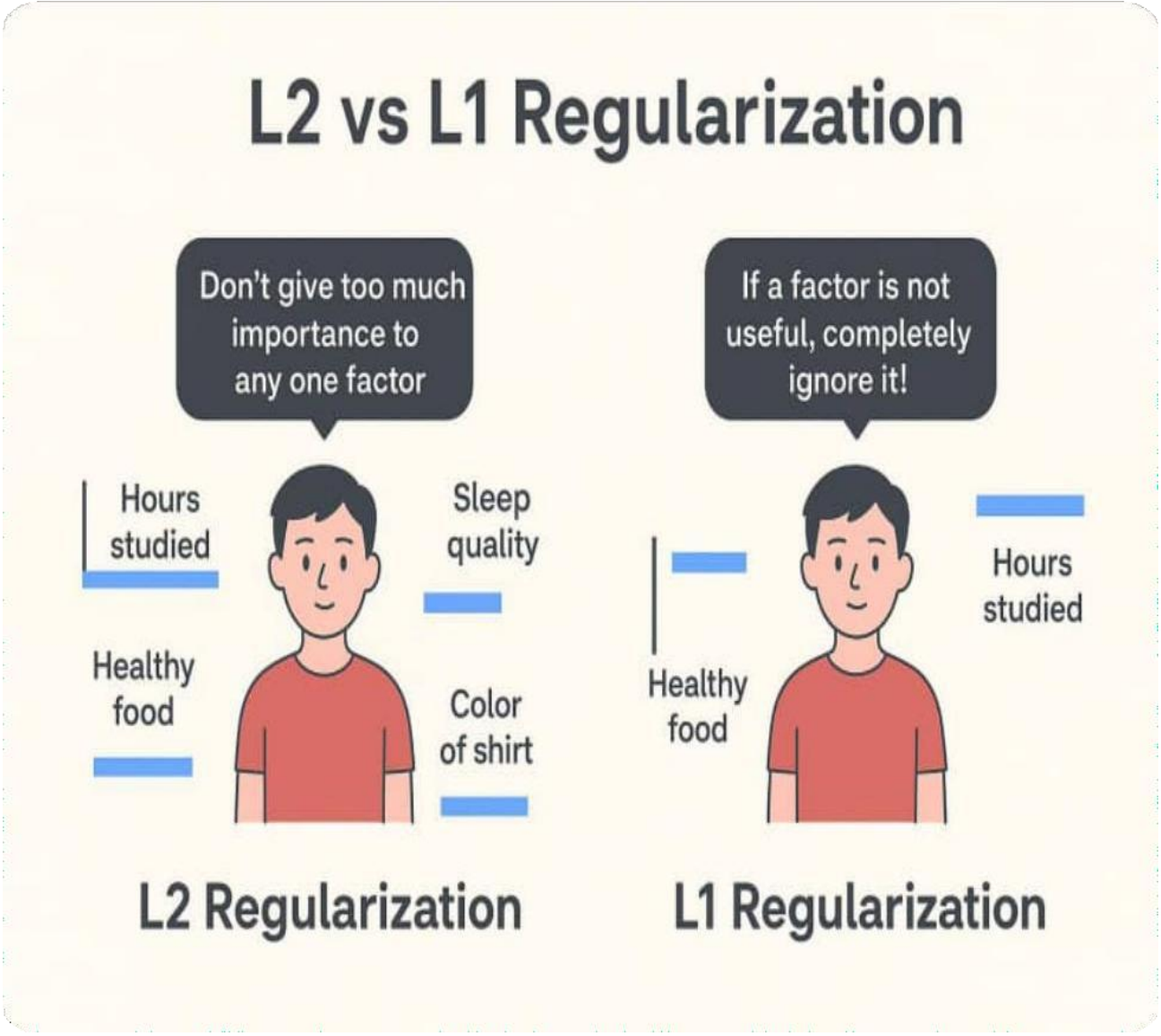
- Think of it like a teacher saying: *"Don't give too much importance to any one factor. Spread out the importance fairly."*
- The model will **shrink all factors a bit**, even the unimportant ones.
- So, every feature keeps *some role*, but weaker.

Effect: All features contribute, but with smaller influence.

<u>L1 Regularization (Sparsity)</u>

- Think of it like a strict teacher saying: *"If a factor is not useful, completely ignore it!"*
- The model will **eliminate some features entirely** (e.g., shirt color, pencils).
- Only the important ones (hours studied, sleep) stay.

Effect: The model automatically **chooses the most useful features**.



- **L2 = soft shrinkage** (everyone contributes, but less).
- **L1 = hard selection** (some features completely cut off).
-

Norm Penalties as Constrained Optimization

Step 1: Regularized cost function

Equation (7.25):

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\,\Omega(\theta)$$

- $J(\theta; X, y)$: the normal cost function (like mean squared error, cross-entropy, etc.).
- $\Omega(\theta)$: penalty on the parameters (e.g., L1 norm or L2 norm).
- α: controls how strong the penalty is.

STEP2: Thinking as a constrained problem:

- Instead of just adding the penalty, we can think of it like this:

"Minimize the cost J, but **only allow parameters** 迦ÿ**whose penalty** Ω(迦ÿ)**is less than some constant** k**.**"

That's a **constraint**.

Instead of adding a punishment, we can think like this:

👉 "Minimize the error $J(\theta)$, **but only allow solutions where the weights aren't too big.**"

That "not too big" means:

$$\Omega(\theta) \leq k$$

- $\Omega(\theta)$ is the size of the weights (like L1 or L2 norm).
- k is some limit (a boundary).

Suppose you are picking a car.

- Your goal: minimize cost (J).
- But you also add a rule: the car's fuel consumption (Ω) must be less than 10 liters/100km (k).

So now you're not free to pick any car—you only pick cars that **stay within the constraint**.

- Regularization can be viewed not just as "adding punishment" but also as "forcing parameters to stay inside a safe boundary."

STEP3:Lagrange function trick

Imagine you're solving a problem like this:

- You want to **minimize your cost** (make the model error small).
- But you also have a **rule/constraint**: don't let the parameters get too big (stay inside the red circle we discussed).

Problem: Handling "rules" directly in optimization is hard.

The Lagrangian trick makes it easier:

Instead of keeping the rule separate, we **add the rule into the cost function itself** with a penalty.

- If the model stays within the rule → small penalty (or none).
- If the model breaks the rule → big penalty is added.
- So Instead of solving *two problems* (minimize cost + check rule), we solve *one problem*:
  a **new cost function** = original cost + penalty.
- Example:

1. **Your Goal (Cost Function)**:
   You want to eat as much tasty food as possible in the hostel mess (this is like minimizing error in ML).
2. **Constraint**:
   The warden says: "You cannot spend more than ₹2000 per month."

- **Two Ways to Handle It**:
- **Constraint directly**: You must plan meals so your monthly bill ≤ ₹2000.
- **Lagrange Trick**: Instead of forcing the rule, you change your *goal* to:
- *Enjoyment of food +fine for breaking the rule*
- So, if you spend more than ₹2000, the warden gives you a fine. The fine increases as you spend more.

3. **Why is this useful?**

Now, you don't need to check every time if you cross ₹2000.

The new "goal + fine" system automatically pushes you toward the best balance:

- Eat enough to be happy,
- But not too much, or else fines make it worse.

$$\mathcal{L}(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k)$$

Here:

- α is called a **Lagrange multiplier.**
- It punishes the solution if the constraint is violated ($\Omega(\theta) > k$).

Equation (7.27):

$$\theta^* = \arg\min_{\theta} \max_{\alpha \geq 0} \mathcal{L}(\theta, \alpha)$$

This means:

1. For a given parameter setting θ, check if the constraint is satisfied.
2. If not, the multiplier α will increase the penalty.
3. We then find the parameter θ^* that balances minimizing the cost **and** respecting the constraint.

Imagine you're choosing food with two goals:

- You want to minimize **cost** (J).
- But you also want calories (Ω(θ)) to be less than some limit (k).

If calories go above the limit, you get a **penalty** (through α).

So the final decision balances **cheap food** and **healthy food** → that's what constrained optimization with regularization does.

Dataset Augmentation

- The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.
- This approach is easiest for classification. A classifier needs to take a compli cated, high dimensional input x and summarize it with a single category identity y. This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new (x,y) pairs easily just by transforming x the inputs in our training set
- This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimate on problem.

How do we create fake data?

1. **Transform the original data:**

   For example, if we are training an image classifier, we can:

   - Shift the picture a little to the left or right.
   - Rotate it slightly.
   - Zoom in or out.
   - Add some brightness or noise.

When we train a machine learning model, we usually want **more data** to make it learn better. But often, we don't have enough real data. So, the trick is: **create new fake data** from the existing one — this is called **dataset augmentation**.

Where is this useful?

- Works really well for **classification tasks**, especially **image recognition** (like detecting cats, cars, or numbers).
- It's also useful for **speech recognition** — changing the tone, speed, or adding noise in audio can create more examples.
- But it's **not always useful** for other tasks (like density estimation), because creating fake data is harder there.

Not every transformation is safe. For example, in **text recognition** (like reading letters or numbers):

- If you flip 'b', it may look like 'd'.
- If you flip '6', it may look like '9'. That would **confuse the model** instead of helping it.

Sometimes, we can also add **random noise** (like small disturbances) into the input.

- Imagine a blurry or slightly noisy image of a cat.
- If the model still recognizes it as a cat → it becomes stronger (robust).

This is also a form of **augmentation**.

Researchers found that **adding noise carefully** during training can make the model:

- More general (works in different conditions).
- Less sensitive to small changes.

This idea connects with another concept called **Dropout** (removing some parts of the network randomly during training), which also works like creating "new inputs."

NOISE ROBUSTNESS

Noise and regularization (controlling weights)

- If you add very small random noise to the inputs, it turns out to be mathematically similar to adding a **penalty on the weights** of the model.
- This penalty prevents the weights from becoming too large, which is a form of **regularization** (helps avoid overfitting).
- This idea was discussed by **Bishop (1995)**.

Noise injection is more powerful

- Noise injection is not just about shrinking weights.
- Adding noise **inside the hidden layers** (not just inputs) can be even more effective.
- It forces the network to be more robust and generalize better.

What is Dropout?

- Dropout is a **regularization technique** used in neural networks.
- During training, **random neurons are "dropped out"** (ignored/turned off) with some probability (say 50%).
- This forces the network to not depend too heavily on any one neuron.

1. Suppose we have a small network:
   - Input → Hidden Layer (4 neurons) → Output
2. With **dropout rate = 0.5**:
   - In each training step, half the neurons in the hidden layer are randomly set to 0 (inactive).
   - Example:
     - Step 1: Neurons [2, 4] dropped.
     - Step 2: Neurons [1, 3] dropped.
     - Step 3: Different random drop, and so on.

So the network never relies on a fixed set of neurons.

Dropout (special case of noise injection)

- Adding noise to hidden units is so important that it deserves its own discussion.
- **Dropout** is the main technique based on this idea.
  - In dropout, random hidden units are "dropped" (set to zero) during training.
  - This is like adding noise, and it prevents the network from relying too much on specific neurons.

- Adding noise helps models generalize better.
- Adding tiny noise at inputs = weight regularization.
- Adding noise at hidden layers = even stronger, leads to dropout.

Noise added to weights

- Another way to regularize a model (prevent overfitting) is by adding **noise directly to the weights** instead of inputs or hidden units.
- This was used in **recurrent neural networks (RNNs)**.

This can be interpreted as a stochastic implementation of Bayesian inference over the weights. The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty.

2. Connection to Bayesian inference

- Think of weights in a neural network: normally they're fixed values after training.
- But in **Bayesian learning**, weights are not just single values; instead, they are considered **uncertain** and described by a **probability distribution** (e.g., "this weight is most likely around 0.5, but it could vary").
- Adding noise to weights is like simulating this uncertainty — instead of one fixed weight, the model "samples" slightly different weights each time.

- Imagine you're guessing a student's math exam score.
- Based on practice tests, you think the student usually scores **80 marks**.

1. Traditional (non-Bayesian) view

- You say: "Okay, the score is **exactly 80**."
- Just one fixed number — no uncertainty.

2. Bayesian view

- You say: "I'm not 100% sure. Maybe the score will be between **75 and 85**."
- So instead of one number, you describe the score with a **probability distribution**:

- Most likely around 80
- Some chance of 78, 82, etc.

Adding noise (practical way)

- Instead of always using "80," you sometimes try **79, 82, 77, 83…**
- Each time you add a little **random noise** to your guess.

This way, you're not overconfident, and you're **capturing the uncertainty**.

Injecting Noise at the Output Targets

The problem

- In real datasets, **labels (y)** are not always correct.
- If the model always tries to make predictions with **100% confidence** in those labels, it can **overfit** to mistakes.
- Example: A picture of a **cat** wrongly labeled as a **dog** → the model will learn wrong things.

The solution → Add noise to labels

- Instead of saying "this label is 100% correct," we say:
  - With **high probability (1 − ε)** → the label is correct.
  - With **small probability (ε)** → it could be one of the other labels.

◆ Label Smoothing (main method)

- Normally, for classification, the true label is written as:
  - Cat = `[1, 0, 0]` (hard target → 100% Cat, 0% others)
- With **label smoothing**:
  - Cat = `[0.9, 0.05, 0.05]` (soft target → mostly Cat, but small chance of Dog or Rabbit).

So the targets are not just 0 or 1, but "softened."

Label smoothing = making labels a little uncertain (soft, not hard 0/1), so the model doesn't become overconfident and generalizes better.

Suppose we are classifying animals, and there are **k = 3 classes**:

- Cat
- Dog
- Rabbit

Now let's say the true label is **Dog**.
We'll use **ε = 0.1** (10% noise).

Step 1: Correct label probability (1 − ε)

$$1 - \varepsilon = 1 - 0.1 = 0.9$$

So → the model assumes **90% chance the label Dog is correct**.

Step 2: Distribute the noise ε among the other (k − 1) classes

There are **k − 1 = 2 other classes** (Cat and Rabbit).

Each of them gets:

$$\frac{\varepsilon}{k-1} = \frac{0.1}{2} = 0.05$$

So → Cat = 0.05, Rabbit = 0.05.

Step 3: Final "soft" label distribution (after label smoothing)

- Dog (correct class) → **0.9**
- Cat → **0.05**
- Rabbit → **0.05**

Semi Supervised Learning

In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, \mathbf{y})$ are used to estimate $P(\mathbf{y} \mid \mathbf{x})$ or predict $\mathbf{y}$ from $\mathbf{x}$.

- x → input (example / features, like an image, text, etc.)
- y → output (label, like "cat" or "dog")
- P(x) → distribution of only the inputs (unlabeled data)
- P(x, y) → distribution of inputs with labels (labeled data)
- P(y | x) → probability of the label y given input x (what we want the model to learn).

Representation $h = f(x)$

- In deep learning, instead of directly mapping input x to output y, we often first **transform x into a representation** h(a vector that captures the important features).
- Example: An image of a cat → deep network learns h (like "has fur", "has whiskers", "has pointed ears").

GOAL: Learn [Same-class examples should have similar representations]

- If two inputs belong to the same class (say, two cat pictures), then their learned representations h should be **close together**.
- If one is a cat and another is a dog, their representations should be **far apart**.

Linear classifier on top of representation

- Once a good representation h is learned, even a **simple classifier** (like logistic regression or a linear SVM) can separate the classes effectively.
- Why?
- Because the hard work (grouping, clustering) has already been done in the representation space.

1. Two types of models:

- **Discriminative model**: Directly predicts label from input

$$P(y|x) \quad \text{(classification model, e.g., logistic regression, neural net classifier)}$$

- **Generative model**: Models how the data is generated

$$P(x) \quad \text{or} \quad P(x, y) \quad \text{(describes distribution of inputs or joint distribution of inputs + labels)}$$

2. Shared parameters between them

Instead of training these separately, we can let both **share the same representation** $h = f(x)$.

- The **discriminative part** uses h to predict y.
- The **generative part** uses h to reconstruct or explain x (and sometimes y too).

This way, unlabeled data ($P(x)$) also helps improve the learned representation.

Imagine a dataset of coins:

- Input x: coin color (gold, silver, bronze).
- Label y: coin type (₹1, ₹2, ₹5).

1. **Generative model $P(x)$:**
   Learns that 70% of coins are silver, 20% gold, 10% bronze.
   → Doesn't know which type of coin, just knows what coins look like.
2. **Generative model $P(x, y)$:**
   Learns the joint pattern, like:
   - (silver, ₹2) → 60%
   - (gold, ₹5) → 15%
   - (bronze, ₹1) → 10%

     → Now it knows **which label goes with which input.**

- **P(x)** = "what inputs look like" (unlabeled data).
- **P(x,y)** = "what inputs and labels look like together" (labeled data).

3. Trade-off between objectives

We combine two losses:

- **Supervised loss:** $-\log P(y|x)$ (classification error on labeled data).
- **Unsupervised/generative loss:** $-\log P(x)$ or $-\log P(x, y)$ (fit to unlabeled data).

The training objective =

$$L = \alpha \cdot (-\log P(y|x)) + (1 - \alpha) \cdot (-\log P(x))$$

where α balances the two.

---

4. Why this helps?

- Purely **supervised**: Needs many labels.
- Purely **generative**: Learns structure, but may not classify well.
- **Combination (semi-supervised)**: Uses structure of $P(x)$ from unlabeled data to improve the classifier $P(y|x)$.

MULTI-TASK LEARNING

- **Multi-task learning** (introduced by Caruana, 1993) is a method where a single model is trained to do **multiple related tasks at the same time**.
- This helps the model **generalize better** (perform well on new unseen data).
- Why? Because when tasks are related, sharing information between them acts like a **constraint** that prevents the model from overfitting to just one task.
- **Multi-task learning = one model, multiple tasks, better generalization**.

Imagine you are teaching a student:

- **Task 1:** Learn to recognize cars.
- **Task 2:** Learn to recognize trucks.
- **Task 3:** Learn to recognize buses.

If you train **three separate models**, each only sees one type of data. But if you train **one shared model** for all three tasks:

- The model will learn **common features** (like wheels, headlights, body shapes).
- These shared features make the model **stronger** and better at generalizing.

Train one model for multiple related tasks.

When we build an MTL model, it usually has **two types of parameters (parts):**

1. **Task-specific parameters**
   - These are **unique** to each task.
   - They help only that particular task.
   - In a neural network: these are the **upper layers** (the last few layers).
   - Example:
     - If Task A = detect cars, Task B = detect trucks, the final output layers for each task are different.
2. **Generic (shared) parameters**
   - These are **common** across all tasks.
   - They capture general knowledge useful for every task.
   - In a neural network: these are the **lower layers** (the first few layers).
   - Example:
     - Features like edges, shapes, colors are useful whether detecting cars, trucks, or buses → so they are **shared**.

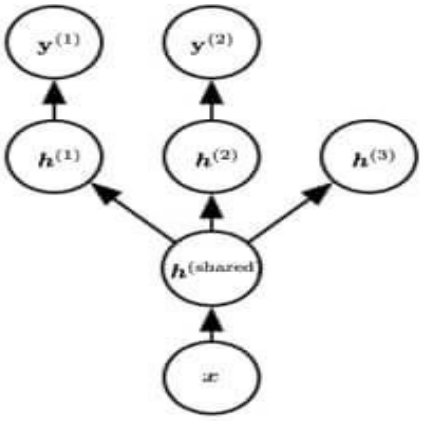


Figure 7.2: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $h^{(1)}$ and $h^{(2)}$) can be learned on top of those yielding a shared representation $h^{(shared)}$. The underlying assumption is that there exists a common

Architecture of Multi-Task Learning (MTL) in a neural network.

1. Input (x):

- The same input is given for all tasks.
- Example: an image of a face.

2. **Shared Layers (h(shared)):**

- The first few layer's extract **generic features** (edges, colors, shapes, embeddings).
- These are **common for all tasks**.
- Example: features like eyes, nose, hair texture.

3. Task-Specific Layers:

- After the shared part, the network **branches out** into different "heads," each solving a specific task.
- Example:
  - Task 1 head predicts **age**.
  - Task 2 head predicts **gender**.
  - Task 3 head predicts **emotion**.

4. Outputs (y^1, y^2, …):

- Each branch produces its own result.
- Even though tasks are different, they all **benefit from the shared lower features**.

EARLY STOPPING

- Early stopping is a regularization technique used to prevent overfitting while training neural networks.
- The idea:

- During training, the model's training loss keeps decreasing (it memorizes the training data).
- But the validation loss (performance on unseen data) first decreases, then starts increasing (overfitting starts).
- Early stopping means: stop training when validation loss starts getting worse, even if training loss is still improving.

1. Split data into **training set** and **validation set**.
2. Train the model epoch by epoch.
3. After each epoch:
   - Compute **training loss** and **validation loss**.
4. Keep track of the **best validation loss** so far.
5. If validation loss does not improve for a set number of epochs (called **patience**), stop training.

Best Parameters Tracking

- During training, we don't just stop and take the model from the *last* iteration.
- Instead, we keep a copy of the parameters (weights) whenever the validation error improves.
  That way, we always know which set of weights gave the lowest validation error.
  Returning the Best Model
- Once training ends (because of early stopping), we return the best recorded parameters, not the final parameters.
- This ensures we get the model with the best generalization performance.
- **Patience (Pre-specified number of iterations)**
- The text also explains that we don't stop immediately when the validation error fails to improve once.
- Instead, we wait for a pre-specified number of iterations/epochs (called *patience*).
- If there is no improvement in that patience window, training stops.

Early stopping isn't just about *when* to stop; it's also about *which model parameters to keep*. We save the weights that performed best on the validation set to ensure the final model generalizes well.

Suppose you train a neural network for 50 epochs:

- Training loss keeps decreasing.
- Validation loss decreases until epoch 15, then starts increasing slowly.
- With **early stopping + patience (say patience = 5 epochs)**:
  - The algorithm keeps track of the best parameters (at epoch 15).
  - If by epoch 20 the validation error hasn't improved, training stops.
  - Finally, the parameters from **epoch 15 (the best point)** are returned — not the ones from epoch 20.

- n → Number of steps (or epochs) between checking validation error.
- p (patience) → How many times we allow the validation error to get worse before stopping.
- θ → Model parameters (weights).
- θ* → Best parameters found so far (the ones that gave lowest validation error).
- v → Best validation error so far.
- i → Current step (how far training has gone).
- i* → Step at which the best validation error was found.

STEPT1

1.θ ← θ_0

- θ_0 are the initial parameters (weights) of the model before training begins.
- θ will keep changing during training as the model learns.

2.i ← 0

- i is the counter for the number of training steps (or epochs).
- Since we haven't started training, it's 0.

3.j ← 0

- j is the **patience counter**.
- It counts how many times the validation error got worse in a row.
- Starts at 0 because no bad epochs have happened yet.

4.v ← ∞

- v is the "best validation error so far."
- At the start, we haven't seen any error yet, so we set it to infinity (∞), meaning *anything we see will be better*.

5.θ* ← θ

- θ* stores the **best model parameters found so far**.
- Initially, this is just the starting parameters.

6.i* ← i

- i* is the training step where the best parameters were found.
- At the start, that's just 0.

<u>Step 2</u>: Training Loop (while j < p do …)

This step is the **core of early stopping**. We keep training the model and checking validation error **until the patience limit is exceeded**.

1.Update θ by running the training algorithm for n steps.

- We don't check validation after every tiny update (that would be inefficient).
- Instead, we run training for **n steps/epochs** before checking again.
- After this update, θ changes (the model has learned more).

2.i ← i + n

- Increase the training step counter.
- If we trained for n=5 steps, and i was 0, now i = 5.

3. v' ← ValidationSetError(θ)
   - Compute the error on the validation set using the current parameters θ.
   - This tells us how well the model is generalizing right now.
4. Check if validation improved:
   - If v' < v (new validation error is better than the best so far):
     - Reset patience counter j = 0 (because we saw improvement).
     - Save the best parameters θ* = θ (since this is the best model yet).
     - Save the step i* = i (the step number where best happened).
     - Update best validation error v = v'.
   - Else (validation error got worse or stayed the same):
     - Increase patience counter j = j + 1 (we lost one chance).

- **Stopping Condition:**

- The loop continues as long as j < p.
- That means:
  - If validation keeps improving, training continues.
  - If validation stops improving for **p times in a row**, training stops.

- θ = student's current knowledge after practicing for n more days.
- i = total number of practice days so far.
- v' = score in the mock test today.
- v = best score seen so far.
- θ* = snapshot of their best knowledge so far.
- j = how many bad scores in a row the student has had.
- p = how many bad scores we'll tolerate before saying "Stop practicing, you were already best earlier."

Example:

- Student takes practice tests every 5 days.
- If their scores keep improving → reset patience counter.
- If scores get worse for 3 tests in a row (p = 3) → stop.
- Teacher then says: "The best you did was on day X, so let's take that as your final preparation."

Step 3: Output (after loop ends)

- When the patience counter **j reaches p**, the loop stops. At this point:
- **θ*** = the parameters that gave the **lowest validation error**.
- **i*** = the step/epoch at which that lowest error occurred.
- These are returned as the final model, instead of the parameters from the very last iteration.

Early stopping watches validation performance during training and keeps the best model parameters seen so far (call that the *checkpoint*).

- To do that you must **store a copy** of those best parameters somewhere (memory or disk). That storage requirement is the "additional cost" .
- Cost is usually **negligible** because you can save the best parameters in slower storage (like host RAM or disk) instead of the fast-but-limited GPU memory.
- **Early stopping needs a validation set — tradeoff.**
- We must hold out some data (validation) to monitor when to stop. That means the model trains on slightly less data during the early-stopping run.
- To make use of that held-out data (so nothing is wasted), we use an extra training step after you finish early stopping, using *all available labeled data*.

The two ways to do that extra training:

- Strategy A — **Reinitialize and retrain on the full dataset**
- Strategy B — **Restore the best parameters and continue / fine-tune on the full dataset.**

STRATEGY A

- Split your labelled training data into **subtrain** and **validation**.
- Run **early stopping** on the subtrain (use the validation only to decide when to stop). This gives you the *best number of training steps* i∗.
- **Reinitialize** the model weights (start fresh).
- Train the model **on the entire training set** (subtrain + validation) for i* steps. Save that final model.

Strategy A — Retrain from scratch on all data (Algorithm 7.2)

1.Use subtrain/valid to find the best number of steps i* via early stopping.
2.Reinitialize model parameters randomly.
3.Train on the **entire training set** (now no hold-out) for exactly i* steps.

Pros:

•Uses all labeled data for final model (better data usage).
•Deterministic: same stopping budget as found before.

Cons :

•Ambiguous whether i* means the same number of *parameter updates* or same number of *epochs* (since training set size changed).
•Extra cost of a full re-training run from scratch.

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $X^{(\text{train})}$ and $y^{(\text{train})}$ be the training set.
Split $X^{(\text{train})}$ and $y^{(\text{train})}$ into $(X^{(\text{subtrain})}, X^{(\text{valid})})$ and $(y^{(\text{subtrain})}, y^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $X^{(\text{subtrain})}$ and $y^{(\text{subtrain})}$ for training data and $X^{(\text{valid})}$ and $y^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $X^{(\text{train})}$ and $y^{(\text{train})}$ for i^* steps.

Activa

1. **Input:** Training set (X^{train}, y^{train}).
2. **Split data** → into
   - Subtrain set $(X^{subtrain}, y^{subtrain})$ → used for actual training.
   - Validation set (X^{valid}, y^{valid}) → used for early stopping decisions.
3. **Run Early Stopping (Algorithm 7.1):**
   - Train only on subtrain data.
   - Stop at the best iteration.
   - Return i^* (optimal number of training steps).
4. **Reinitialize Parameters θ:**
   - Start from random weights again (don't reuse).
5. **Retrain on Full Dataset (X^{train}, y^{train}):**
   - Train for exactly i^* steps.

1. **Full Training Dataset**
   We start with the complete dataset (X_{train}, y_{train}).
2. **Split into Subtrain + Validation**
   - **Subtrain set:** Used for actual model training.
   - **Validation set:** Used only to monitor performance and detect overfitting.

   This ensures the model generalizes instead of just memorizing.
3. **Run Early Stopping**
   - Train the model on $(X_{subtrain}, y_{subtrain})$.
   - After each epoch, evaluate validation error.
   - Stop training when validation error stops improving → gives optimal iteration i^*.

   Output: i^* (the best number of training steps).
4. **Retrain on Full Dataset**
   - Now discard the old model.
   - Train a fresh model from scratch on the **entire dataset** (X_{train}, y_{train}).
   - Run it for exactly i^* steps (not more).

   This way, we use **all available data** but avoid overfitting.
5. **Final Model**
   ✅ Final model = trained on all the data + with the right amount of training (i* steps).

- **Dataset**: Suppose we have a training set with 10 data points: $X^{(\text{train})} = [x_1, x_2, \ldots, x_{10}]$ and $y^{(\text{train})} = [y_1, y_2, \ldots, y_{10}]$, where x_i are features and y_i are labels (e.g., 0 or 1 for binary classification).
- **Split**: Randomly split the data into:
  - $X^{(\text{subtrain})} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]$ and $y^{(\text{subtrain})} = [y_1, y_2, y_3, y_4, y_5, y_6, y_7]$ (7 points, 70%).
  - $X^{(\text{valid})} = [x_8, x_9, x_{10}]$ and $y^{(\text{valid})} = [y_8, y_9, y_{10}]$ (3 points, 30%).
- **Model Parameters**: Start with random initial parameters $\theta = [0.1, 0.2]$ (e.g., weights for a linear model).
- **Training Step Size**: Let $n = 2$ steps between validation checks (e.g., two gradient descent updates).
- **Patience**: Set $p = 2$ (stop if validation error doesn't improve for 2 consecutive checks).
- **Loss Function**: Use a simple mean squared error (MSE) as $J(\theta, X, y) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$, where $\hat{y}_i = \theta \cdot x_i$ (linear prediction) and m is the number of data points.
- **Validation Error**: Computed on the validation set after every n steps.

Step-by-Step Example

Step 1: Run Early Stopping (Algorithm 7.1)

- **Initialize**: $\theta = [0.1, 0.2]$, $i = 0$ (total steps), $j = 0$ (patience counter), $v = +\infty$ (best validation error), $\Theta^* = [0.1, 0.2]$, $i^* = 0$.
- **Loop While $j < p$ (i.e., $j < 2$)**:
  - **Update θ for $n = 2$ steps**:
    - Simulate 2 gradient descent steps on $X^{(\text{subtrain})}$ and $y^{(\text{subtrain})}$. For simplicity, assume each step adjusts θ based on the gradient of MSE. Let's say after 2 steps (e.g., using a learning rate of 0.1), θ updates to $[0.3, 0.4]$ (this depends on the gradient, which we'll approximate).
    - $i \leftarrow i + n = 0 + 2 = 2$.
  - **Compute Validation Error $v' = J(\theta, X^{(\text{valid})}, y^{(\text{valid})})$**:
    - Suppose $v' = 0.8$ (e.g., average squared error on the 3 validation points).
  - **If $v' < v$ (0.8 < $+\infty$)**:
    - $j \leftarrow 0, \Theta^* \leftarrow [0.3, 0.4], i^* \leftarrow 2, v \leftarrow 0.8$.

- **Next Iteration:**
  - Update θ for 2 more steps. Suppose θ becomes $[0.5, 0.6]$.
  - $i \leftarrow 2 + 2 = 4$.
  - $v' = J(\theta, X^{(\text{valid})}, y^{(\text{valid})}) = 0.6$ (improves).
  - $v' < v$ (0.6 < 0.8): $j \leftarrow 0$, $\Theta^* \leftarrow [0.5, 0.6]$, $i^* \leftarrow 4$, $v \leftarrow 0.6$.
- **Next Iteration:**
  - Update θ to $[0.7, 0.8]$.
  - $i \leftarrow 4 + 2 = 6$.
  - $v' = 0.7$ (worsens).
  - $v' \geq v$ (0.7 ≥ 0.6): $j \leftarrow 1$.
- **Next Iteration:**
  - Update θ to $[0.9, 1.0]$.
  - $i \leftarrow 6 + 2 = 8$.
  - $v' = 0.8$ (worsens).
  - $v' \geq v$ (0.8 ≥ 0.6): $j \leftarrow 2$.
- **Exit Loop:** $j = 2 \geq p = 2$, so stop. $i^* = 4$, $\Theta^* = [0.5, 0.6]$.

Step 2: Reset and Retrain on Full Data

- **Set θ to random values again:** Let $\theta = [0.1, 0.2]$ (same initial value for simplicity).
- **Train on $X^{(\text{train})}$ and $y^{(\text{train})}$ for $i^* = 4$ steps:**
  - Perform 4 gradient descent steps on the full 10-point dataset. Suppose after 4 steps, θ updates to $[0.52, 0.62]$ (slightly different due to using all data).

Result

- **Final Parameters:** $\theta = [0.52, 0.62]$.

Strategy B — Continue training from θ* using all data

- After early stopping, **keep the learned θ*** (don't reinitialize).
- Continue training but now include the whole training set.
- Stop when validation loss falls below the target value that was achieved at the end of the early-stopping run (or use some other rule).

Pros

- Saves time (no full retrain from scratch).

- Uses the good initialization already found.

Cons

- No guarantee it will ever reach the target value — may not terminate or may diverge.
- Less principled; behavior depends on learning rate / optimization dynamics.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
- **end while**

Start with your **full training set** and split it into **subtrain** and **validation** (just like before).

- Run early stopping on the **subtrain** (monitoring validation) until it stops. Keep the **final parameters** it produced; call them θ.
- Compute ε = the **training objective (loss)** evaluated on the *subtrain* at θ. (This is the loss value where early stopping stopped.)
- **Continue training** the model from θ, but now train on the **entire training set** (subtrain + validation).
- After each chunk of training (say every n steps), evaluate the .
- Stop **when the validation loss falls to or below ε**
- . In words: keep training until the validation error becomes as small (or smaller) than the training loss you observed at the end of the first pass.

Step-by-Step Explanation

1. **Start with training data**
   - You have a training set: $X^{(train)}, y^{(train)}$.
2. **Split training into two parts**
   - $X^{(subtrain)}, y^{(subtrain)}$: the subset of training data used to actually train the model.
   - $X^{(valid)}, y^{(valid)}$: a validation set, used only to detect overfitting.

   > This split is temporary — we'll later train on the full training set again.

---

3. **Run early stopping (Algorithm 7.1)**
   - Train on the sub-training set $(X^{(subtrain)}, y^{(subtrain)})$.
   - Keep monitoring the validation error $J(\theta, X^{(valid)}, y^{(valid)})$.
   - Stop when validation error stops improving.
   - This gives us the "best" parameters θ.
4. **Record the error at that point**
   - Compute

$$\epsilon \leftarrow J(\theta, X^{(subtrain)}, y^{(subtrain)})$$

   This is the training error (on subtrain) at the point where validation error was minimized.

---

5. **Final training on the full training set**
   - Now, instead of discarding part of the training data, we continue training on the **entire training set** $(X^{(train)}, y^{(train)})$.
   - But we stop when the error on the validation set becomes worse than ϵ:

$$\text{while } J(\theta, X^{(valid)}, y^{(valid)}) > \epsilon \quad \text{do train on } (X^{(train)}, y^{(train)})$$

   - In other words: we continue training on the full dataset until we reach the same level of overfitting that we previously detected.

- **Dataset**: Suppose we have a training set with 10 data points: $X^{(\text{train})} = [x_1, x_2, \ldots, x_{10}]$ and $y^{(\text{train})} = [y_1, y_2, \ldots, y_{10}]$, where x_i are features and y_i are labels (e.g., 0 or 1 for binary classification).
- **Split**: Randomly split the data into:
  - $X^{(\text{subtrain})} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]$ and $y^{(\text{subtrain})} = [y_1, y_2, y_3, y_4, y_5, y_6, y_7]$ (7 points, 70%).
  - $X^{(\text{valid})} = [x_8, x_9, x_{10}]$ and $y^{(\text{valid})} = [y_8, y_9, y_{10}]$ (3 points, 30%).
- **Model Parameters**: Start with random initial parameters $\theta = [0.1, 0.2]$ (e.g., weights for a linear model).
- **Training Step Size**: Let $n = 2$ steps between validation checks (e.g., two gradient descent updates).
- **Patience**: Set $p = 2$ (stop if validation error doesn't improve for 2 consecutive checks).
- **Loss Function**: Use a simple mean squared error (MSE) as $J(\theta, X, y) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$, where $\hat{y}_i = \theta \cdot x_i$ (linear prediction) and m is the number of data points.
- **Validation Error**: Computed on the validation set after every n steps.

Step-by-Step Example

Step 1: Run Early Stopping (Algorithm 7.1)

- **Initialize**: $\theta = [0.1, 0.2], i = 0, j = 0, v = +\infty, \Theta^* = [0.1, 0.2], i^* = 0.$
- **Loop While $j < p$ (i.e., $j < 2$)**:
  - **Update θ for $n = 2$ steps**:
    - Simulate 2 gradient descent steps on $X^{(\text{subtrain})}$ and $y^{(\text{subtrain})}$ (learning rate = 0.1). Suppose θ updates to $[0.3, 0.4]$.
    - $i \leftarrow 0 + 2 = 2.$
  - **Compute Validation Error $v' = J(\theta, X^{(\text{valid})}, y^{(\text{valid})})$**:
    - Suppose $v' = 0.8.$
  - **If $v' < v$ (0.8 < $+\infty$)**:
    - $j \leftarrow 0, \Theta^* \leftarrow [0.3, 0.4], i^* \leftarrow 2, v \leftarrow 0.8.$
  - **Next Iteration**:
    - Update θ to $[0.5, 0.6]$.
    - $i \leftarrow 2 + 2 = 4.$
    - $v' = 0.6.$
    - $v' < v$ (0.6 < 0.8): $j \leftarrow 0, \Theta^* \leftarrow [0.5, 0.6], i^* \leftarrow 4, v \leftarrow 0.6.$

- **Next Iteration:**
  - Update θ to $[0.7, 0.8]$.
  - $i \leftarrow 4 + 2 = 6$.
  - $v' = 0.7$.
  - $v' \geq v$ (0.7 ≥ 0.6): $j \leftarrow 1$.
- **Next Iteration:**
  - Update θ to $[0.9, 1.0]$.
  - $i \leftarrow 6 + 2 = 8$.
  - $v' = 0.8$.
  - $v' \geq v$ (0.8 ≥ 0.6): $j \leftarrow 2$.
- **Exit Loop:** $j = 2 \geq p = 2$, stop. Final $\theta = [0.9, 1.0]$.

Step 2: Compute Overfitting Threshold

- $\epsilon \leftarrow J(\theta, X^{(\text{subtrain})}, y^{(\text{subtrain})})$:
  - Compute training error with $\theta = [0.9, 1.0]$ on the subtraining set.
  - Suppose $J = 0.5$ (e.g., average MSE on the 7 subtrain points).

Step 3: Train on Full Dataset Until Threshold

- **Initialize**: Reset $\theta = [0.1, 0.2]$ (random again).
- **While** $J(\theta, X^{(\text{valid})}, y^{(\text{valid})}) > \epsilon$ **(i.e.,** > 0.5**) do:**
  - **Initial Validation Error**: $J(\theta, X^{(\text{valid})}, y^{(\text{valid})}) = 0.9$ (high since θ is random).
  - **Train on** $X^{(\text{train})}$ **and** $y^{(\text{train})}$ **for** $n = 2$ **steps:**
    - Update θ to $[0.3, 0.4]$.
    - $J(\theta, X^{(\text{valid})}, y^{(\text{valid})}) = 0.7$ (still > 0.5).
  - **Next Iteration:**
    - Update θ to $[0.5, 0.6]$.
    - $J = 0.6$ (still > 0.5).
  - **Next Iteration:**
    - Update θ to $[0.7, 0.8]$.
    - $J = 0.55$ (still > 0.5).
  - **Next Iteration:**
    - Update θ to $[0.9, 1.0]$.
    - $J = 0.5$ (equals ϵ).
  - **Exit Loop**: $J = 0.5 \leq \epsilon$, stop. Final $\theta = [0.9, 1.0]$.

Sparse Representations:

- Sparse representations aim to express a signal, image, or dataset using a minimal number of non-zero elements in a chosen basis or dictionary.
- Instead of representing data with all possible features, sparsity assumes that only a few are necessary to capture the essence of the data.
- This is often achieved by transforming the data into a domain (e.g., frequency or wavelet domain) where its representation is naturally sparse.

Definition: A representation is sparse if most of its coefficients are zero, and only a few are non-zero.

- In **sparse representation**, we want to express a signal (like an image, sound, or data vector) as a **combination of a few "building blocks"**. These building blocks are called **basis vectors**, and the whole collection is called a **dictionary**.
    - Think of a dictionary in language: it has many words.
    - Similarly, in math, a **dictionary** has many basis vectors.
    - Any signal can be written as a "sentence" using a small number of these vectors.
- The **dictionary** can be **predefined** (Fourier basis, wavelets) or **learned** from data (like in dictionary learning, K-SVD).

Sparse in the context of neural networks

- In the context of neural networks, this idea extends to the **activations** of units in a layer (i.e., the output of neurons, often denoted as h) rather than just the model parameters (weights, denoted as θ). The goal is to make the **representation** (the activation vector h) sparse,

The two types of sparsity:

- **Parameter Sparsity**: Achieved by penalizing the model parameters (e.g., weights) directly, often using an L1-norm penalty to drive many parameters to zero.
- **Representational Sparsity**: Achieved by penalizing the activations of units in a neural network, encouraging the representation h to have few non-zero elements.

- **Parameter Sparsity**: Involves making the weight matrix sparse. For example, in the linear regression model $y = Ax$, the matrix A (as shown in the first equation) has many zero entries, indicating a sparse parameterization. This reduces the number of active parameters, simplifying the model.

$$y = Ax, \quad \text{where} \quad A \in \mathbb{R}^{m \times n} \text{ is sparse (many zeros).}$$

Example from the text:

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}$$

Here, A is sparse (many zeros), reducing the number of parameters used.

- **Representational Sparsity**: Involves making the representation h (the output of a transformation of the input x) sparse. In the second example, $y = Bh$, where h is a sparse vector (few non-zero entries), representing the input data x in a compact way.

$$y = Bh, \quad \text{where} \quad h \in \mathbb{R}^{n} \text{ is sparse (many zeros).}$$

Example from the text:

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

Here, h is sparse (only two non-zero entries: 2 and -3), indicating that the representation of the data is compact.

Regularization for Representational Sparsity:

Representational sparsity is achieved by adding a penalty term to the loss function that encourages the activation vector h to be sparse. The regularized loss function is:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(h)$$

- where: J(θ;X,y) is the standard loss (e.g., mean squared error for regression or cross-entropy for classification).
- Ω(h) is a penalty on the representation h.
- α≥0 controls the strength of the regularization (larger α means more sparsity).

Common Penalties for Sparsity

L_1-Norm Penalty

$$\Omega(h) = \|h\|_1 = \sum_i |h_i|.$$

Hard Constraints for Sparsity:

Instead of a penalty, sparsity can be enforced with a hard constraint on the number of non-zero entries in h h h. For example, **Orthogonal Matching Pursuit (OMP)** solves:

$$\arg\min_{h, \|h\|_0 < k} \|x - Wh\|_2^2,$$

- where: ‖h‖0 is the number of non-zero entries in h.
- k is the maximum number of non-zero entries allowed.
- W is an orthogonal dictionary (basis).

Why Use Representational Sparsity

Efficiency: Sparse activations reduce the number of active neurons, lowering computational and memory requirements.

Robustness: Sparse representations focus on the most important features, making models less sensitive to noise or irrelevant inputs.

Interpretability: Sparse activations highlight which units are most relevant for a given input, aiding in understanding the network's behavior.

Generalization: By preventing overfitting (similar to weight decay), sparsity encourages simpler, more generalizable models.

Connection to Weight Decay

- The excerpt contrasts representational sparsity with **weight decay**, which directly penalizes model parameters (e.g., weights θ) using an L_2-norm (or sometimes L_1 -norm) penalty: $\tilde{J}(\theta; X, y) = J(\theta; X, y) + \lambda \|\theta\|_2^2.$
- Weight decay shrinks parameters toward zero but doesn't necessarily make them exactly zero (unless using L_1).
- Representational sparsity, by penalizing activations h , indirectly constrains the parameters because the activations depend on the weights and inputs h = f(Wx + b). This creates a more complex, data-dependent regularization effect compared to weight decay.

Chapter8: Optimization for Training Deep Models

How Learning Differs from Pure Optimization, Basic Algorithms. Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates. **Chapter 8 (8.1,8.3,8.4,8.5)**

What is Optimization in Deep Learning?

- **Optimization** refers to the process of adjusting the parameters (weights and biases) of a neural network so that it can minimize a given objective function, usually the **loss function** and maximize the **accuracy**.

Loss function measures how far the model's predictions are from the true outputs.

Optimizer is the algorithm that updates the weights to reduce the loss.

Common optimization algorithms:

- **Gradient Descent (GD)** → basic method.
- **Stochastic Gradient Descent (SGD)** → updates with small batches for efficiency.
- **Momentum** → remembers past gradients for faster convergence.
- **Adam** → combines momentum + adaptive learning rates.
- **RMSProp** → scales learning rate with moving average of squared gradients.

Optimization = "How do I learn the best weights to reduce training loss?"

Regularization = "How do I make sure my learned weights don't just memorize training data, but generalize to new data?"

Optimization focuses on **fitting**.
Regularization focuses on **generalizing**.

How Learning Differs from Pure Optimization

Traditional optimization:

- Optimization is about directly minimizing or maximizing a function J.
- Example: minimize cost of production, maximize profit.

Machine learning optimization:

- In ML, goal is **good performance on new (test) data**.

- That performance measure is usually called P (like accuracy, F1-score, AUROC).
- But the problem: we *cannot directly optimize P*, because it depends on unseen test data.

What we actually do:

- Instead, we optimize a **proxy function** (a cost/loss function J(θ) on training data.
- Example: cross-entropy loss, mean squared error.
- The hope is that by minimizing J(θ), the model's performance P on test data will also improve.

- **Pure optimization** = directly solving the function you care about.
- **ML optimization** = you *can't optimize test accuracy directly*, so you optimize a loss function on training data, hoping it leads to better test performance.

Cost function on training data (Eq. 8.1):

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} L(f(x;\theta), y)$$

- L = loss for a single example (how wrong prediction is).
- $f(x;\theta)$ = model's prediction for input x with parameters θ.
- $\hat{p}_{data}$ = distribution of the training dataset (the finite set of examples we actually have).
- $J(\theta)$ = average loss on the training data.

What we *really want* (Eq. 8.2):

$$J^*(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x;\theta), y)$$

- Here, instead of $\hat{p}_{data}$ (training set), we use p_{data}, the **true data distribution** (all possible real-world examples).
- $J^*(\theta)$ = expected loss over the *true distribution*.
- This corresponds to *generalization performance* (how well the model will do on unseen test data).

The gap:

- We only have access to the finite training data ($\hat{p}_{data}$), not the full true distribution (p_{data}).
- So we train by minimizing $J(\theta)$, and hope it leads to minimizing $J^*(\theta)$.

8.1.1 Empirical Risk Minimization

Goal of machine learning:

We want to reduce the **expected generalization error** → this means making the model perform well on **new/unseen data**.

- This expected error is called the **risk**.

- **Training error:** How many mistakes it makes on the training pictures you already gave it.
- **Generalization error:** How many mistakes it makes on **brand new cat and dog pictures** it has never seen before.
- **Expected generalization error:** The **average number of mistakes** it would make if you could test it on **all cats and dogs in the world** (not just a few test images).

If we knew the true data distribution $p_{data}(x, y)$:

- Then minimizing risk would be a normal optimization problem (easy to define mathematically).
- Example: If we had *all possible exam questions*, we could just directly train for them.

But reality:

- We don't know the true distribution of all data in the world.
- We only have a **finite training dataset** (a small sample).
- That's why machine learning is tricky → we're solving an optimization problem with only limited data.

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. We now minimize the **empirical risk**

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}) \tag{8.3}$$

where m is the number of training examples.

The training process based on minimizing this average training error is known as empirical risk minimization.

However, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. In many cases, empirical risk minimization is not really feasible. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we actually optimize is even more different from the quantity that we truly want to optimize.

Surrogate Loss Functions and Early Stopping

- Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier
- In such situations, one typically optimizes a surrogate loss function instead, which acts as a proxy but has advantages.
- For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation

Batch

- **Gradient Descent in Deep Learning**

When training a neural network, we update weights using **gradient descent**. But the way we compute the gradient depends on how much data we use at once.

- Optimization algorithms that use the entire training set are called **batch** or deterministic gradient methods, because they process all of the training examples simultaneously in a large batch.
- **Example**: If you have 1 million images, you calculate the loss and gradient for **all 1 million** before updating weights.

Mini-Batch Gradient Descent

- Splits the dataset into **small batches** (e.g., 32, 64, 128 samples).
- For each mini-batch: compute gradient → update weights.
- Most commonly used in deep learning.
- Pros :
  - Faster training than full batch.
  - More stable than using single samples.
  - Works well with GPUs (parallel processing).
- Cons : Still approximate, so updates can be noisy.
- Example: With 1 million images and batch size = 64 → you update weights after every 64 images.

Stochastic Gradient Descent (SGD)

- Special case of mini-batch, where batch size = 1.
- Update weights after **every single training example**.

Minibatch sizes are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect (Wilson and Martinez, 2003), perhaps due to the noise they add to the learning process. Generalization

8.3 **BASIC ALGORITHMS**

8.3.1 Stochastic Gradient Descent

- **Gradient descent** = method to train models by adjusting weights step by step to reduce loss.
- **Stochastic gradient descent (SGD)** is a version where:
  - Instead of using the *whole dataset* to compute the update (which is slow),
  - We use just **a few samples** (a mini-batch) to estimate the gradient and update weights.

- Stochastic Gradient Descent (SGD) trains a model by updating weights using small random subsets of data (mini-batches) instead of the whole dataset, making training much faster and efficient.

1. **SGD is the most popular optimization algorithm**
   - Especially in deep learning, almost everyone uses SGD or its improved versions (like Adam, RMSProp).
2. **How it works**
   - You take a random mini-batch of training examples (say 32 images).
   - Compute the gradient of the loss just for those examples.
   - Update the model weights.
   - Repeat with another mini-batch.
3. **Why it works**
   - Even though you don't use the entire dataset, the mini-batch gradient is still a **good estimate** of the true gradient.

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Inputs needed:

- **Learning rate** (ϵ_k → (how big a step we take in each update.
- **Initial parameters** (迪ÿ) → the starting weights of the model.

Loop until stopping condition is met:

- "Stopping condition" could be:
  - number of iterations reached,
  - loss becomes very small,
  - or validation accuracy stops improving.

Steps inside the loop:

1. Pick a random mini-batch of m training examples $(x^{(1)}, \ldots, x^{(m)})$ with their labels $(y^{(i)})$.
2. Compute the gradient estimate $(\hat{g})$:
   - Instead of using the whole dataset, we only use this mini-batch.
   - Formula:

$$\hat{g} = \frac{1}{m} \sum_i \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

   - This is the "average slope" of the loss for this mini-batch.
3. Update the parameters (θ):
   - Move the weights in the opposite direction of the gradient:

$$\theta \leftarrow \theta - \epsilon \hat{g}$$

Repeat until the model has trained enough.

- **^g** is the **average of the gradients computed for each example in the chosen mini-batch**.

What it me

- 迿ÿ= the parameters (weights) of the model.
- ϵ= learning rate (step size).
- =the gradient estimate (direction of steepest increase of the loss, computed from the mini-batch).
- The update rule says: Take the **current parameter values**, and **subtract a small step in the direction of the gradient**.

Why subtract?

- The gradient points in the direction of *increasing* the loss.
- We want to *decrease* the loss, so we go in the **opposite direction**.

Learning rate is crucial in SGD

- The learning rate (ϵ) controls **how big a step** the algorithm takes when updating weights.
- If it's too large → the algorithm may overshoot and never converge.
- If it's too small → learning becomes very slow.
- Before, we assumed $\epsilon$was constant (same step size at every iteration).
- It's usually better to **start with a larger learning rate** (so the model learns quickly in the beginning),
- Then **gradually decrease it** as training progresses (to fine-tune and converge smoothly).
- That's why the learning rate at iteration $k$is written as:
- instead of a fixed ϵ.

8.3.2 Momentum

- Imagine you're trying to roll a ball down a hill to reach the very bottom, where there's a prize. The hill isn't smooth—it has bumps, dips, and flat spots that can slow you down or even trap the ball.
- If you just push the ball gently each time, it might get stuck in a small dip or keep bouncing back and forth without going far.
- Momentum is like giving the ball a stronger, steadier push that keeps it moving in one direction, even if it hits a bump. It helps the ball ignore the small dips and wiggles, so it can roll faster and smoother toward the prize at the bottom
- In deep learning, this "ball" is like the computer trying to find the best solution, and momentum helps it move past obstacles and get to the goal faster!

- Small dips (local minima) where the ball could get stuck.
- Flat spots (saddle points) where the ball might slow down.
- Steep curves (high curvature) where the ball could bounce back and forth.
- One deep valley at the far right (the global minimum, where the prize is).

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

- **Learning rate (ε)**: This is like the size of the steps the computer takes to adjust its guesses. A small step means careful changes; a big step means bolder changes.
- **Momentum parameter (α)**: This is like a memory factor that helps the computer keep moving in the right direction, even if it hits a bump.
- **Initial parameter (θ)**: This is the computer's starting guess for solving the problem.
- **Initial velocity (v)**: This is like the initial push or speed, starting at zero, that will help guide the adjustments.

- In physics, **velocity** = speed + direction of movement.
- In optimization, **velocity (v)** = a **running average of past gradients**.

It's like "memory" of where the parameters were moving in previous steps.

Each time you train (one iteration) do:

1. **Sample a mini-batch** of `m` training examples
   → you'll compute the update from only this small subset (fast & scalable).
2. **Compute the gradient estimate**

$$g \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

This `g` is the average gradient for the mini-batch (same as in plain SGD).

3. Update the velocity

$$v \leftarrow \alpha v - \epsilon g$$

- αv = carry-forward of the previous velocity (keeps past direction).
- $-\epsilon g$ = the new push based on the current gradient (negative because we want to *go down* the loss).

So v is a mix of old momentum and the new gradient push.

4. Update the parameters

$$\theta \leftarrow \theta + v$$

We add v to θ because v already points (roughly) in the negative-gradient direction.

Nesterov Momentum

Nesterov Momentum, also known as Nesterov Accelerated Gradient **(NAG),** is an advanced optimization technique that improves upon traditional momentum in gradient descent by incorporating a "look-ahead" step to better anticipate future parameter updates, leading to faster and more stable convergence.

How Nesterov Momentum Works

- Traditional momentum updates parameters by adding a fraction of the past update (velocity) to the current gradient step, which helps accelerate convergence and smooth out oscillations.
- Nesterov Momentum modifies this by first making a look-ahead step using the current velocity (momentum), then calculating the gradient at this look-ahead position.
- This lets the optimizer **"peek"** where it is going and adjust the update based on this future position, effectively acting with more foresight.

Basic Idea: Moving Down a Hill

- Imagine you're rolling a ball down a curvy hill to find the bottom (the best solution for your model). In basic gradient descent, you push the ball based on the slope where it is right now. But if the hill has flat spots or small bumps, the ball might slow down or wiggle a lot.
- Momentum adds a "push" from the direction the ball was already moving, helping it keep going even on flat areas.
- Nesterov Momentum takes this further by being smarter about where it looks before pushing.

- Instead of just checking the slope at the current spot, **it peeks ahead to where the ball is about to go**, then decides how to push based on that future spot. This makes the movement more accurate and efficient.
-

Given:

- θ: model parameters
- v: velocity (momentum term)
- α: learning rate
- γ: momentum coefficient (usually close to 1, such as 0.9)

The update is done as:

1. Compute look-ahead position:
   $\theta_{lookahead} = \theta - \gamma v$
2. Calculate gradient at look-ahead:
   $g = \nabla f(\theta_{lookahead})$
3. Update velocity with gradient:
   $v = \gamma v + \alpha g$
4. Update parameters:
   $\theta = \theta - v$

This anticipatory update allows the optimizer to better adjust its course and reduce overshooting or slowdowns.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Key Benefits

- **Faster convergence:** Especially effective in loss surfaces with flat or curved regions.
- **Greater stability:** By correcting the step before the full update, it smooths oscillations more effectively than classical momentum.
- **Better responsiveness:** The look-ahead gradient allows the method to avoid poor local minima with more agility.

Monentun vs Nesterov nonentun

Aspect	Traditional Momentum	Nesterov Momentum
Gradient Evaluation Point	Computes gradient at current parameters θ_t	Computes gradient at look-ahead position $\theta_t - \gamma v_t$
Update Step	Updates velocity and then parameters using current gradient	Looks ahead by moving along velocity, computes gradient there, updates velocity and parameters accordingly
Formula	$v_{t+1} = \gamma v_t + \alpha \nabla_\theta J(\theta_t)$ $\theta_{t+1} = \theta_t - v_{t+1}$	$v_{t+1} = \gamma v_t + \alpha \nabla_\theta J(\theta_t - \gamma v_t)$ $\theta_{t+1} = \theta_t - v_{t+1}$
Intuition	Momentum accumulates past gradients to smooth updates	Momentum anticipates future position to correct update earlier
Convergence Speed	Faster than standard gradient descent but slower than Nesterov	Generally faster convergence and better correction than basic momentum
Handling Oscillations	Helps reduce oscillations but may lag behind true gradient	Reduces oscillations more effectively due to look-ahead gradient
Sensitivity to Gradient Changes	May overshoot or lag behind sudden changes	Adapts better to gradient changes, more responsive update
Use Case	Good for general acceleration	Preferred for faster and more stable training in deep learning
Complexity	Simpler to implement	Slightly more complex due to look-ahead step

Parameter Initialization Strategies

- Deep learning models are trained with **iterative algorithms** (like gradient descent).
- These algorithms **start from an initial point** (initial weights & biases).
- The **choice of initialization** affects:
  - Whether training converges at all.
  - How fast it converges.
  - Whether it finds a good solution (low cost, good generalization).
- Bad initialization can cause:
  - **Numerical instability** → algorithm crashes.
  - **Slow convergence** → learning drags.
  - **Poor generalization** → model works well on training but fails on new data.

Why Initialization Matters

- Deep learning training algorithms are iterative and sensitive to the initial parameter values.
- The initial point influences:
  - Convergence: Poor initialization may cause slow or no convergence.
  - Numerical stability: Bad initial points can cause exploding or vanishing gradients.
  - Final solution quality: Different initializations can lead to different local minima with varying generalization error.
- A key requirement is to break symmetry: units with the same activation receiving the same inputs must start with different parameters to learn distinct features.

Common Initialization Strategies

- Random Initialization
  - Most weights are initialized randomly from Gaussian or uniform distributions.
  - Often biases are initialized to zero or a small constant.
  - The scale of the distribution is crucial; too large causes exploding values, too small causes vanishing gradients.
- Heuristic Scales
  - Xavier (Glorot) Initialization: Scales weights depending on the number of input and output connections, suitable for sigmoid/tanh activations.
  - He Initialization: Specifically designed for ReLU activations; weights are drawn with variance scaled by 2 divided by the number of inputs.
  - Both approaches aim to maintain activation variance and gradient variance through layers.
- **Orthogonal Initialization**
- Use random **orthogonal matrices** for weight initialization.
- Ensures different units start with independent directions.
- **Sparse Initialization**
- Only a few weights per neuron are non-zero at start.
- Helps with diversity but can slow early learning if wrong weights are chosen.
- **Data-Based Initialization**
- Set biases/vances using training data statistics.
- Example: in autoencoders, bias = mean of input distribution.
- **Pretraining**
- Initialize weights from another model:

- Unsupervised pretraining (e.g., autoencoder).
- Supervised pretraining on related or even unrelated task (transfer learning).

Algorithms with Adaptive Learning Rates

- Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance.
- **Learning rate** difficulty**:** The learning rate is a crucial hyperparameter, difficult to set because it strongly impacts training performance.
- **Sensitivity** in parameter space**:** Some directions in parameter space affect the loss a lot (high sensitivity), while others do not.
- **Momentum** helps but adds complexity**:** Momentum smooths training but introduces another hyperparameter.
- **Separate learning rate** per parameter**:** If sensitivities are axis-aligned, it's logical to use different learning rates for each parameter.

Delta-bar-delta algorithm:

- An early heuristic from Jacobs (1988) that adjusts individual learning rates dynamically.
- If the gradient's sign for a parameter remains consistent, the learning rate for that parameter increases—to make faster progress.
- If the gradient's sign flips, it indicates overshooting or oscillation, so the learning rate decreases.

Full batch only: This algorithm generally applies to full batch gradient descent (using the full dataset at each update).

Iteration	Gradient	Sign Compared to Last	Action	New Learning Rate
1	-2	N/A	no change	0.10
2	-1.8	Same	increase	0.12
3	-1.7	Same	increase	0.13
4	+0.5	Different	decrease	0.08

AdaGrad (Adaptive Gradient) algorithm

- The AdaGrad algorithm is an adaptive optimization technique that adjusts the learning rate for each individual parameter based on past gradients.

- Individually adapts the learning rates for each model parameter by scaling them **inversely proportional** to the **square root** of the **sum of all** historical **squared gradients** for that parameter.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

How AdaGrad Works

- For every parameter, AdaGrad keeps a running sum of all previous squared gradients.
- When updating, the learning rate for a parameter is divided by the square root of its accumulated squared gradient.
- If a parameter has a large gradient often, its learning rate drops quickly, **slowing updates in steep directions**.
- If a parameter has small gradient (partial derivatives), its learning rate decreases more slowly, allowing **bigger steps in flat regions** of the loss landscape

Effects

- Rapid decrease for large gradients: Parameters with large gradients quickly get a much smaller learning rate, preventing overshooting.
- Gentle progress in flat areas: Parameters with small gradients keep relatively larger learning rates, encouraging faster movement through flat directions.
- This makes AdaGrad effective for problems with sparse data, or where different features are important at different times (e.g., NLP or recommendation systems)

Limitations in Deep Learning

- constant accumulation of squared gradients **causes** the learning rate to shrink too much over time—sometimes becoming so small that learning nearly stops before reaching a good solution.
- This means AdaGrad may stall prematurely and not always work well for deep models.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Activate Windows
Go to Settings to activate

- AdaGrad shrinks the learning rate for parameters that have large gradients frequently, but keeps it higher for those with smaller gradients.
- This means the algorithm moves faster in flatter directions and slower in steeper directions, helping especially in sparse data scenarios.
- Over time, the learning rate can become very small for some parameters, causing the effective updates to stop for those directions

RMSProp(Root Mean Square Propagation) algorithm

The RMSProp algorithm is an adaptive learning rate optimization method designed to improve deep learning training, especially on non-convex problems.

RMSProp vs. AdaGrad

- AdaGrad adapts the learning rate for each parameter by dividing it by the square root of the running sum of all its past squared gradients.
    - Problem: In deep or non-convex models, this running sum grows too large, causing the learning rate to shrink excessively. Training may slow down or even stop before finding a good solution.

RMSProp solves this by using an exponentially weighted moving average (EWMA) instead of a simple sum:

- Only recent squared gradients matter most, so old gradient information fades away.
- This keeps the learning rate from decaying too quickly and allows the optimizer to adapt to new regions in parameter space.

RMSprop

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta+\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

AdaGrad

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

Why is this important?

- RMSProp helps maintain a balanced learning rate as optimization progresses, enabling effective learning in deep, non-convex settings where AdaGrad often fails due to learning rate collapse.
- By focusing on recent gradients, RMSProp can navigate complex, changing loss landscapes better than AdaGrad, which only works best for convex functions or sparse features.

summary:

RMSProp is designed for deep learning by using a decaying average of squared gradients, preserving effective learning rates and ensuring the optimizer remains adaptable throughout the training process.

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Adam (Adaptive Moment Estimation) Algorithm

Adam is an adaptive learning rate optimization algorithm widely used in deep learning. It combines the benefits of two other **algorithms:RMSProp and Momentum**

Key Concepts in Adam

- Momentum (First-order moment):
  - Adam keeps track of an exponentially weighted moving average of past gradients, smoothing the gradient updates.
  - This helps accelerate convergence by reducing oscillations.

- RMSProp (Second-order moment):
  - Adam also tracks an exponentially weighted moving average of the squared gradients.
  - This rescales the learning rate individually per parameter, adjusting for gradient magnitudes.
- Bias correction:
  - Because both first and second moment estimates start at zero, Adam includes bias-correction terms to counteract initial bias, improving early training stability.

How Adam Works (Summary of the Algorithm)

- For each parameter, Adam estimates:
  - **First moment** (mean of gradients): m_t
  - **Second moment** (mean of squared gradients): v_t
- These are updated with decay rates β_1 and β_2 (usually 0.9 and 0.999).
- Bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$ are computed.
- Parameter updates scale the gradient by $\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ times the learning rate.

Why Adam is Effective

- By combining momentum and adaptive learning rates, Adam allows faster convergence and more stable training, especially for complex, high-dimensional neural networks.
- Bias correction ensures accurate moment estimates early on, making Adam robust to hyperparameter choices.
- This makes Adam a popular default optimizer for many deep learning tasks.

Comparison with RMSProp

- RMSProp also adapts learning rates based on squared gradients but lacks momentum incorporation and bias correction.
- Adam explicitly models both first and second moments with corrections, providing a theoretically stronger foundation and often better performance.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while