

# Gemini Chat

attempt to  
prompt Gemini  
to explore many  
methods (if possible) →

Perform this question by explaining many possible approaches (if there are many) and exactly how to arrive at the answer. Show all work.

Question 1: Optimizers as Penalized Linear Improvement with different norm penalties

In lecture, you saw the locally linear perspective of a neural network and the loss by Taylor expanding the loss around the current value of the parameters. This approximation is only very good in a near neighborhood of those values. One way to proceed with optimization is to consider the size of the neighborhood as a hyperparameter and to bound our update to stay within that neighborhood while minimizing our linear approximation to the loss. You saw in lecture that the choice of norm in defining that neighborhood also matters.

In this problem, you will work out for yourself a slightly different perspective. Instead of treating the norm as a constraint (with the size of the acceptable norm as a hyperparameter), we can do an unconstrained optimization with a weighted penalty that corresponds to the squared norm — where that weight is a hyperparameter.

At each iteration, we wish to maximize linear improvement of the objective (as defined by the dot-product between the gradient and the update) locally regularized by a penalty on the size of the update. This can be expressed (in traditional minimization form) as:

$$u = \operatorname{argmin}_{\Delta\theta}$$

$$g$$

$T \Delta\theta$

$| \{z\}$

Linear Improvement

+

1

$\alpha$

$d(\Delta\theta)$

$| \{z\}$

Distance Penalty

, (1)

where  $g = \nabla f(\theta)$  is the gradient of the loss,  $\alpha$  is a scalar, and  $d$  is a scalar-output distance function

$R$

$\dim(\theta) \rightarrow R$

+

(a) Let's assume Euclidean distance is the norm that captures our sense of relevant neighborhoods in parameter space. Then our objective can be:

$u = \operatorname{argmin}_{\Delta\theta}$

$g$

$T \Delta\theta +$

1

$\alpha$

$\|\Delta\theta\|$

2

2

. (2)

What is the analytical solution for  $u$  in the above problem?

What standard optimizer does this recover?

(b) Now, consider an alternative way of capturing local neighborhood size – the infinity norm over parameters.

Recall that this is defined as  $\|x\|_\infty = \max_i$

$|x_i|$

. Our objective is now:

$u = \operatorname{argmin}$

$\Delta\theta$

$g$

$T \Delta\theta +$

$1$

$\alpha$

$\|\Delta\theta\|_\infty. (3)$

What is the analytical solution for  $u$  in this case? Which optimizer does this correspond to?

- ◆ Here is a step-by-step derivation for both optimization problems. This perspective helps bridge the gap between abstract geometry and the mechanical update rules we use in deep learning.

### Part (a): Euclidean Norm ( $L_2$ )

We are asked to solve for the update  $u = \Delta\theta$  that minimizes the following objective:

$$u = \operatorname{argmin}_{\Delta\theta} \left( g^T \Delta\theta + \frac{1}{\alpha} \|\Delta\theta\|_2^2 \right)$$

Here,  $g = \nabla f(\theta)$  is the gradient vector.

**1. Expand the Terms** Recall that the squared Euclidean norm is the dot product of a vector with itself:  $\|\Delta\theta\|_2^2 = \Delta\theta^T \Delta\theta$ . The objective function  $J(\Delta\theta)$  becomes:

*defines objective* ←  $J(\Delta\theta) = g^T \Delta\theta + \frac{1}{\alpha} \Delta\theta^T \Delta\theta$   
*fn, good nl notation*

**2. Find the Derivative** To find the minimum, we take the derivative of  $J$  with respect to the vector  $\Delta\theta$  and set it to zero.

- The derivative of the linear term  $g^T \Delta\theta$  is simply  $g$ .
- The derivative of the quadratic term  $\Delta\theta^T \Delta\theta$  is  $2\Delta\theta$ .

$$\nabla_{\Delta\theta} J = g + \frac{1}{\alpha} (2\Delta\theta)$$

**3. Solve for  $\Delta\theta$**  Set the gradient to zero to find the critical point:

$$g + \frac{2}{\alpha} \Delta\theta = 0$$

Rearrange to solve for  $\Delta\theta$ :

$$\frac{2}{\alpha} \Delta\theta = -g$$

$$\Delta\theta = -\frac{\alpha}{2} g$$

simple  
and correct  
optimization

**4. Identify the Optimizer** The solution  $\Delta\theta = -\frac{\alpha}{2} g$  tells us to move in the direction of the negative gradient, scaled by a scalar factor ( $\frac{\alpha}{2}$ ).

- **Analytical Solution:**  $u = -\frac{\alpha}{2} \nabla f(\theta)$
  - **Standard Optimizer:** This recovers **Gradient Descent** (or SGD if using mini-batches), where  $\frac{\alpha}{2}$  acts as the learning rate.
- 

### Part (b): Infinity Norm ( $L_\infty$ )

Now we consider the infinity norm, defined as the maximum absolute value among the vector's components:  $\|\Delta\theta\|_\infty = \max_i |\Delta\theta_i|$ . The objective is:

$$u = \operatorname{argmin}_{\Delta\theta} \left( g^T \Delta\theta + \frac{1}{\alpha} \|\Delta\theta\|_\infty \right)$$

The homework initially had a typo

This problem is slightly trickier because the "max" function is not differentiable everywhere in the standard sense. We can solve this by logical deduction regarding the **direction** and **magnitude** of the vectors.

**1. Analyzing the Direction (Sign Alignment)** We want to minimize the sum of two terms:

1. **Linear Improvement:**  $g^T \Delta\theta = \sum_i g_i \Delta\theta_i$
2. **Penalty:**  $\frac{1}{\alpha} \max_i |\Delta\theta_i|$

Let's look at the first term. To make the dot product  $\sum g_i \Delta\theta_i$  as negative (minimized) as possible, the sign of each update component  $\Delta\theta_i$  must be the opposite of the gradient component  $g_i$ .

- If  $g_i$  is positive,  $\Delta\theta_i$  should be negative.
- If  $g_i$  is negative,  $\Delta\theta_i$  should be positive.

Thus, we know the **direction** for every component must satisfy:

$$\Delta\theta_i = -|\Delta\theta_i| \cdot \text{sign}(g_i)$$

**2. Analyzing the Magnitude** Let  $C = \max_i |\Delta\theta_i|$  be the maximum magnitude of our update. This  $C$  is the only thing determining the penalty cost. The penalty term equals  $\frac{1}{\alpha}C$ .

Now look at the improvement term:  $\sum_i g_i \Delta\theta_i$ . Since we aligned the signs, this becomes  $\sum_i -|g_i||\Delta\theta_i|$ . To get the "most bang for our buck," we should make every  $|\Delta\theta_i|$  as large as allowed by the penalty. Since the penalty is fixed by the *maximum* element  $C$ , we can increase all other elements up to  $C$  "for free" (it doesn't increase the penalty, but it improves the dot product).

Therefore, the optimal strategy is to set the magnitude of **all** components to be equal:

$$|\Delta\theta_1| = |\Delta\theta_2| = \dots = C$$

**3. The Analytical Solution** Combining the sign alignment and the equal magnitude, the update for every component  $i$  is:

$$u_i = -C \cdot \text{sign}(g_i)$$

*Recognizes  
that the problem  
is unconstrained*

(Note on  $C$ : In the specific unconstrained form given in equation (3), minimizing  $-C\|g\|_1 + \frac{1}{\alpha}C$  is linear. If the gradient is large enough,  $C$  shoots to infinity; otherwise, it stays at 0. However, in the context of optimization algorithms, we are interested in the **direction** this geometry implies. If we used a squared penalty

$\frac{1}{2\alpha} \|\Delta\theta\|_\infty^2$ , we would find a finite specific  $C$ , but the direction remains the same.)

**4. Identify the Optimizer** The update rule  $u_i \propto -\text{sign}(g_i)$  means we ignore the magnitude of the gradient and step by a constant amount in the direction of the sign.

- **Analytical Solution:**  $u = -C \cdot \text{sign}(\nabla f(\theta))$  (where  $C$  is a scalar magnitude).
- **Standard Optimizer:** This corresponds to **SignSGD** (sometimes referred to as sign-based gradient descent or related to Rprop's update direction). It moves exclusively based on the coordinate-wise sign of the gradient, effectively moving to the "corners" of the hypercube neighborhood.

### Summary Table

Norm Penalty	Geometry	Update Rule	Optimizer
$L_2$ (Euclidean)	Sphere	$u \propto -g$	Gradient Descent
$L_\infty$ (Max)	Cube	$u \propto -\text{sign}(g)$	SignSGD

offers useful  
next prompt  
to explore Lasso

Would you like to explore how an  $L_1$  norm penalty (Lasso) would affect the optimizer (hint: it leads to sparsity)?

Question 2: Optimizers and their convergence

Consider O: a simplified Adam-style optimizer without weight decay that has iterates

$$\theta_{t+1} \leftarrow \theta_t - \alpha t M_t \nabla f_t(\theta_t) \quad (4)$$

where  $f_t$

is the loss at iteration  $t$  and  $\alpha$

is the step size (learning rate).

Further suppose that the adaptive scaling matrix  $M_t$  is recomputed over each epoch of training and just consists of a diagonal populated by the inverses of the square roots of the mean squared value for the gradients during the epoch for that specific coordinate.

For this part, we have exactly  $n = 1$  training point corresponding to the single equation

$$[1, 0.1, 0.01]\theta = 1 \quad (5)$$

with a 3-dimensional learnable parameters  $\theta$ . Suppose that we start with  $\theta_0 = 0$  and use squared loss

$$f(\theta) = (1 - [1, 0.1, 0.01]\theta)^2$$

2

.

- (a) What specific vector  $\theta$  would standard vanilla SGD (i.e. (4) with  $M_t = I$  and  $a_t = a$ ) converge to assuming  $a > 0$  was small enough to give convergence?
- (b) What specific vector  $\theta$  would the simplified version of Adam O converge to assuming appropriate step-sizes  $a_t > 0$  to give convergence?
- (c) Consider a learning approach that first did training input feature rescaling (so that each feature had unit second-moment), then ran SGD to convergence, and then converted the solution for the rescaled problem back to the original units. What specific vector  $\theta$  would it give as its final solution (for use in original coordinates)?

- ◆ This problem explores **Implicit Regularization**: the tendency of different optimization algorithms to converge to specific solutions among the infinite valid solutions available in underdetermined systems (like neural networks with more parameters than data).

Here is the step-by-step derivation for each case.

### Problem Setup

We have a single data point  $(x, y)$  where  $x = [1, 0.1, 0.01]^T$  and  $y = 1$ . We are minimizing the squared loss  $f(\theta) = (1 - x^T \theta)^2$  starting from  $\theta_0 = [0, 0, 0]^T$ . Since there are 3 parameters and only 1 constraint ( $x^T \theta = 1$ ), there is an entire plane of solutions where the loss is 0. The optimizer determines which specific point on that plane we reach.

---

### (a) Vanilla SGD

**The Concept:** When initialized at zero, SGD (stochastic gradient descent) updates the weights  $\theta$  only in the direction of the data vector  $x$ . Consequently, the final solution must lie in the span of  $x$ . Among all solutions that satisfy  $x^T \theta = 1$ , SGD finds the one with the **minimum Euclidean ( $L_2$ ) norm**.

**Derivation:**

1. **Update Direction:** The gradient  $\nabla f(\theta) = -2(1 - x^T \theta)x$ .  
The update is always a multiple of  $x$ .
2. **Form of Solution:**  $\theta = c \cdot x = c[1, 0.1, 0.01]^T$  for some scalar  $c$ .
3. **Condition:** We need the loss to be zero, so  $x^T \theta = 1$ .

*Effective breakdown of steps*

$$x^T(c \cdot x) = 1$$

$$c\|x\|_2^2 = 1 \implies c = \frac{1}{\|x\|_2^2}$$

4. **Calculation:**

$$\|x\|_2^2 = 1^2 + 0.1^2 + 0.01^2 = 1 + 0.01 + 0.0001 = 1.0101$$

$$c = \frac{1}{1.0101} \approx 0.99$$

**Answer:**

*→ correct answer*

$$\theta_{SGD} = \frac{1}{1.0101} \begin{bmatrix} 1 \\ 0.1 \\ 0.01 \end{bmatrix} \approx \begin{bmatrix} 0.99 \\ 0.099 \\ 0.0099 \end{bmatrix}$$

---

### (b) Simplified Adam (Sign Descent)

**The Concept:** The problem describes an optimizer where the update is scaled by the inverse magnitude of the gradient (similar

to Adam or RMSProp). With  $n = 1$ , the "mean squared gradient" is just the squared gradient of that single point.

$$M_t \approx \text{diag} \left( \frac{1}{|g_1|}, \frac{1}{|g_2|}, \frac{1}{|g_3|} \right)$$

The update becomes  $M_t g = \text{sign}(g)$ . This optimizer ignores the magnitude of the features and moves by a constant amount in each coordinate direction. This geometry finds the solution with the **minimum Infinity ( $L_\infty$ ) norm** (the solution where the maximum component is minimized).

#### Derivation:

1. **Update Direction:** The gradient signs are determined by  $x$  (since error starts positive). Since  $x$  has all positive entries, the negative gradient pushes all parameters positively.
2. **Trajectory:** The optimizer adds the same scalar amount to every parameter at every step.

$$\theta \propto [1, 1, 1]^T$$

3. **Form of Solution:**  $\theta = k[1, 1, 1]^T$ .
4. **Condition:**  $x^T \theta = 1$ .

$$[1, 0.1, 0.01] \cdot \begin{bmatrix} k \\ k \\ k \end{bmatrix} = 1$$

$$k(1 + 0.1 + 0.01) = 1$$

$$1.11k = 1 \implies k = \frac{1}{1.11} \approx 0.9009$$

#### Answer:

*→ correct answer*

$$\theta_{Adam} = \frac{1}{1.11} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 0.9009 \\ 0.9009 \\ 0.9009 \end{bmatrix}$$

---

### (c) Feature Rescaling + SGD

**The Concept:** This approach "whitens" the feature space so that all features have equal importance (unit variance/second-moment). SGD in this transformed space treats all dimensions equally (like Adam did in the original space). However, when we transform back to the original space to get  $\theta$ , the result will look very different.

**Derivation:**

1. **Rescaling Matrix  $S$ :** We want the new feature  $x' = Sx$  to have unit squared value ( $x'_i = 1$ ).

$$x = [1, 0.1, 0.01]^T \xrightarrow{\text{scale}} x' = [1, 1, 1]^T$$

The scaling matrix  $S$  is diagonal with entries equal to the inverse of the feature values:

$$S = \text{diag}(1, 10, 100)$$

2. **SGD in Transformed Space:** We solve for weights  $\phi$  such that  $x'^T\phi = 1$ . Using the logic from part (a), SGD finds the min-norm  $\phi \propto x'$ .

$$\phi = c[1, 1, 1]^T$$

$$[1, 1, 1] \cdot c[1, 1, 1]^T = 1 \implies 3c = 1 \implies c = 1/3$$

$$\phi = [1/3, 1/3, 1/3]^T$$

3. **Convert Back to  $\theta$ :** The prediction is  $y = x^T\phi = (Sx)^T\phi = x^T(S\phi)$ . Thus, the original parameters are  $\theta = S\phi$ .

$$\theta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 100 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 10/3 \\ 100/3 \end{bmatrix}$$

Answer:

↗ again correct

$$\theta_{Rescaled} = \begin{bmatrix} 1/3 \\ 3.33... \\ 33.33... \end{bmatrix}$$

### Comparison of Results

Notice how the choice of optimizer drastically changes the solution  $\theta$ :

- **SGD** relies heavily on the first feature ( $x_1 = 1$ ) because it is the "loudest" signal.  $\theta_1$  is large,  $\theta_3$  is tiny.
- **Adam** distributes the weight equally, ignoring feature scale. All  $\theta$  are equal.
- **Rescaling** forces the smallest feature ( $x_3 = 0.01$ ) to do just as much work as the large one. Because  $x_3$  is so small physically, its weight  $\theta_3$  must be huge to compensate.

Would you like to examine how **Weight Decay** ( $L_2$  regularization) would alter the result for the Adam optimizer?

#### Question 5: Analyzing Distributed Training

For real-world models trained on lots of data, the training of neural networks is parallelized and accelerated

by running workers on distributed resources, such as clusters of GPUs. In this question, we will explore three popular distributed training paradigms:

All-to-All Communication: Each worker maintains a copy of the model parameters (weights) and processes a subset of the training data. After each iteration, each worker communicates with every other worker

and updates its local weights by averaging the gradients from all workers.

Parameter Server: A dedicated server, called the parameter server, stores the global model parameters.

The workers compute gradients for a subset of the training data and send these gradients to the parameter server. The server then updates the global model parameters and sends the updated weights back to the workers.

Ring All-Reduce: Arranges  $n$  workers in a logical ring and updates the model parameters by passing messages in a circular fashion. Each worker computes gradients for a subset of the training data, splits the gradients into  $n$  equally sized chunks and sends a chunk of the gradients to their neighbors in the ring. Each worker receives the gradient chunks from its neighbors, updates its local parameters, and passes the updated gradient chunks along the ring. After  $n-1$  passes, all gradient chunks have been aggregated across workers, and the aggregated chunks are passed along to all workers in the next  $n - 1$  steps. This is illustrated in Fig.

1.

For each of the distributed training paradigms, fill in the total number of messages sent and the size of each message. Assume that there are  $n$  workers and the model has  $p$  parameters, with  $p$  divisible by  $n$ .

- ◆ Here is the breakdown of the communication costs for each distributed training paradigm.

We assume:

- $n$  = Number of workers
  - $p$  = Number of parameters (size of the gradient vector)
- 

## 1. All-to-All Communication

In this naive approach, every worker acts as an independent entity that must synchronize with every other entity.

- **How it works:** Each of the  $n$  workers sends its calculated gradients to every other  $(n - 1)$  worker to perform the averaging.
- **Total Number of Messages:** Each worker sends  $(n - 1)$  messages. Since there are  $n$  workers, the total is  $n(n - 1)$ .
- **Size of Each Message:** Each message contains the full gradient vector.

Result:

- **Total Messages:**  $n(n - 1)$
- **Size per Message:**  $p$

*Sneaky explanation  
and correct answer*

---

## 2. Parameter Server

In this centralized topology, workers do not talk to each other; they only talk to the central server.

- **How it works:**
  1. **Push:** All  $n$  workers send their gradients to the server ( $n$  messages).
  2. **Update:** The server aggregates them and updates the weights.
  3. **Pull:** The server sends the updated weights back to all  $n$  workers ( $n$  messages).
- **Total Number of Messages:**  $n$  (upload) +  $n$  (download) =  $2n$

*↳ properly breaks  
it down*

- **Size of Each Message:** Both the gradients sent up and the weights sent down are the size of the full model.

#### Result:

- **Total Messages:**  $2n$   **connect**
  - **Size per Message:**  $p$
- 

### 3. Ring All-Reduce

This method is designed to maximize bandwidth utilization by breaking the model into chunks. The process is described in the prompt as having two phases: the "Scatter-Reduce" (first  $n - 1$  steps) and "All-Gather" (next  $n - 1$  steps).

- **How it works:**
  - The model is split into  $n$  chunks.
  - There are a total of  $2(n - 1)$  timesteps (iterations).
  - In **every** timestep, **every** worker sends exactly one chunk to its neighbor.
- **Total Number of Messages:**
  - Total timesteps:  $2(n - 1)$
  - Messages per timestep:  $n$  (since all  $n$  workers send a message simultaneously).
  - Total:  $2n(n - 1)$ .
- **Size of Each Message:** The algorithm explicitly splits the gradients into  $n$  equally sized chunks.

#### Result:

- **Total Messages:**  $2n(n - 1)$   **again connect**
  - **Size per Message:**  $\frac{p}{n}$
- 

### Summary Table

Paradigm	Total Messages Sent	Size of Each Message	Note
All-to-All	$n(n - 1)$	$p$	Communication explodes as $O(n^2)$ .
Parameter Server	$2n$	$p$	Server becomes a bandwidth bottleneck.
Ring All-Reduce	$2n(n - 1)$	$p$	Bandwidth efficient (constant load per node).



**Next Step:** Would you like to analyze the **Time Complexity** (latency vs. bandwidth cost) to see why Ring All-Reduce is

mathematically preferred for large clusters despite having a higher  
 You've reached your Thinking limit

total message count than the Parameter Server?

Responses will use other models until it resets on Dec 8, 9:58 PM. Upgrade for higher limits and more.