

1. The Memory Cost of an Optimizer

Training a network with N parameters requires the GPU to hold more than the weights $\theta \in \mathbb{R}^N$. At every step t , backpropagation produces a gradient $g_t = \nabla f_t(\theta_{t-1}) \in \mathbb{R}^N$, and the optimizer may need to remember extra vectors from one step to the next. Those vectors are the *optimizer state*, and each of them is as large as θ itself. In this problem we count these vectors and turn the count into gigabytes. We ignore activations, which scale with batch size and sequence length rather than directly with N (and can be traded for compute by recomputing them), and count only what scales with N . Every number is stored in fp32, which takes 4 bytes, and $1 \text{ GB} = 10^9$ bytes.

- (a) **Fill in the table below.** For each optimizer, write the update rule, list the vectors in $\mathbb{R}^N$ that must be carried in memory from step $t-1$ to step t (the optimizer state), and count the numbers per parameter that the GPU holds during a step, including θ and g_t but not temporaries such as g_t^2 . Use η for the learning rate and β_1, β_2 for the averaging constants.

(Hint: momentum keeps an exponential running average m of the gradient and steps along it; RMSPprop keeps a running average v of the squared gradient and divides the step by $\sqrt{v}$; Adam does both.)

Optimizer	Update rule	Optimizer state	Numbers per parameter
Gradient descent			
SGD			
SGD with momentum			
RMSProp	$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$ $\theta_t \leftarrow \theta_{t-1} - \eta g_t / \sqrt{v_t}$	v	3
Adam			

Solution:

Optimizer	Update rule	State	Numbers/param.
Gradient descent	$\theta_t \leftarrow \theta_{t-1} - \eta g_t$, $g_t = \text{gradient over the whole training set}$	none	2
SGD	$\theta_t \leftarrow \theta_{t-1} - \eta g_t$, $g_t = \text{gradient over a minibatch}$	none	2
SGD with momentum	$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$, $\theta_t \leftarrow \theta_{t-1} - \eta m_t$	m	3
RMSProp	$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$, $\theta_t \leftarrow \theta_{t-1} - \eta g_t / \sqrt{v_t}$	v	3
Adam	m_t, v_t as above, $\theta_t \leftarrow \theta_{t-1} - \eta m_t / \sqrt{v_t}$	m, v	4

The count is θ (one number per parameter) plus g_t (one) plus the state. Three remarks:

- GD and SGD store exactly the same thing. The optimizer cannot tell whether g_t came from one example or from the whole dataset. The difference between them is how much data goes through the forward and backward pass per step, which shows up as activation memory and wall-clock time, not as optimizer state. (Full-batch GD on a large dataset is implemented by accumulating minibatch gradients into the very buffer g we already counted.)
 - Momentum and RMSProp cost the same: one extra vector each. Adam is the union of the two, which is why it is the most expensive of the standard optimizers.
 - In PyTorch these vectors are `p.grad` and, for Adam, the tensors `exp_avg` and `exp_avg_sq` stored in `optimizer.state[p]`. Each has the same shape as `p`.
- (b) A 7-billion-parameter model ($N = 7 \times 10^9$) is trained with every vector stored in fp32. **How many GB does each optimizer need for θ , g_t , and its state? Which of them fit on a single GPU with an 80 GB memory budget?** Remember that the activations still have to fit on top of this.

Solution: One fp32 vector of size N costs $7 \times 10^9 \times 4 \text{ bytes} = 28 \text{ GB}$.

Optimizer	Numbers/param.	Memory	Fits in 80 GB?
GD, SGD	2	56 GB	yes, 24 GB left for activations
SGD with momentum, RMSProp	3	84 GB	no
Adam	4	112 GB	no

With Adam the weights are only a quarter of the footprint; the other three quarters are the gradient and the two moments. This is before any activation is stored, and activations are the other large consumer of memory (they scale with batch size times sequence length).

2. Weight Decay and Adam

Because Adam can use different learning rates for different weights, it can have different implicit regularization behavior than traditional SGD or SGD with momentum. Consequently, it can interact differently when combined with other approaches for explicit regularization — even if those other approaches are equivalent with traditional SGD.

Two such regularization approaches are putting an L2 penalty on weights in the loss function itself (in the style of explicit ridge regularization) and alternatively, adding a weight-decay term during learning updates. For a simplified variant of Adam (without momentum), we have:

- **L2 regularization** can be expressed as: $\theta_{t+1} \leftarrow \theta_t - \alpha \mathbf{M}_t (\nabla f_t(\theta_t) + \lambda_w \theta_t)$
- **Weight decay** can be expressed as: $\theta_{t+1} \leftarrow (1 - \lambda_r) \theta_t - \alpha \mathbf{M}_t \nabla f_t(\theta_t)$

Here $\mathbf{M}_t$ denotes the adaptive term in the Adam update step — think of this as a diagonal matrix.

(a) **When will the updates for L2 regularization and weight decay be identical?**

(Hint: your answer should involve things like α , λ_w , λ_r , and $\mathbf{M}_t$ in some way.)

Solution:

Solving the equality between weight decay and L2 regularization would yield the correct result.

$$\theta_t - \alpha \mathbf{M}_t (\nabla f_t(\theta_t) + \lambda_w \theta_t) = (1 - \lambda_r) \theta_t - \alpha \mathbf{M}_t \nabla f_t(\theta_t) \quad (1)$$

$$-\alpha \mathbf{M}_t \lambda_w \theta_t = -\lambda_r \theta_t \quad (2)$$

$$\alpha \mathbf{M}_t \lambda_w = \lambda_r \mathbf{I} \quad (3)$$

$$\mathbf{M}_t = \frac{\lambda_r}{\lambda_w \alpha} \mathbf{I} \quad (4)$$

What does this tell us? For L2 regularization to be equivalent to the weight decay, $\mathbf{M}_t$ has to be a scaled identity. This does not tend to happen in practice since $\mathbf{M}_t$ changes over every iteration in each component.

This is why AdamW exists — it allows one to just use straightforward weight-decay with Adam.

(b) **Where should you add $\lambda_r \theta_{t-1}$ to the Adam code below to get explicit weight decay?****Algorithm 1** Adam Optimizer (without bias correction)

```

1: Given  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ 
2: Initialize time step  $t \leftarrow 0$ , parameter  $\theta_{t=0} \in \mathbb{R}^n$ ,  $m_{t=0} \leftarrow \mathbf{0}$ ,  $v_{t=0} \leftarrow \mathbf{0}$ 
3: Repeat
4:    $t \leftarrow t + 1$ 
5:    $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$ 
6:    $g_t \leftarrow \nabla f_t(\theta_{t-1}) + \text{---(A)---}$ 
7:    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1)(g_t + \text{---(B)---})$ 
8:    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)g_t^2 + \text{---(C)---}$ 
9:    $\theta_t \leftarrow \theta_{t-1} - (\alpha \cdot \frac{m_t}{\sqrt{v_t}} + \text{---(D)---})$ 
10: Until the stopping condition is met

```

- ☐ **A** on line 6 above
- ☐ **B** on line 7 above
- ☐ **C** on line 8 above
- ☐ **D** on line 9 above

Solution: **Option D** is correct. Adding $\lambda_r \theta_{t-1}$ on line 9 explicitly decreases the magnitude of θ so as to perform weight decay. Option A is the place for L2 regularization, which is almost never equivalent to weight decay for Adam in practical settings, as shown in part (a).

3. Coding: ReLU Networks with Different Optimizers

This notebook trains the same one-hidden-layer ReLU network as in the elbows notebook, but with three optimizers: SGD, SGD with momentum, and Adam, across several widths and random seeds. The choice of optimizer changes how well the fit goes, and it matters most when the network is small.

Work through `q_relu_optim.ipynb`. Run all of the active cells. Leave the optional training cells commented out; the notebook loads pretrained checkpoints, so you do not have to retrain anything. Read the “Bug Hunt” cell without executing it. Then answer the questions below.

- (a) **How do the hidden layer width and the choice of optimizer affect the learned function and the test error?** Say something about the error bars as well as the medians.

Solution: Wider hidden layers give lower test error under every optimizer. Both Adam and SGD with momentum are consistently better than plain SGD, and Adam generally converges fastest, though whether it ends up better than momentum depends on tuning. The gap between optimizers is largest at small width and narrows as capacity grows. Dots show median test error; bars span the 25th–75th percentiles across runs. Compare both typical error and spread at each width—no optimizer has the smallest spread in every setting. Medians and quartiles are useful summaries, but a logarithmic axis does not require using the median.

- (b) **Compare the first and last saved checkpoints for the example run. How do the elbow locations change relative to the target’s kinks under each optimizer?**

Solution: Under all three optimizers the elbows migrate toward the kinks of the target function. It is also common to see several ReLU units converge to nearly the same elbow location.

- (c) The notebook shows a training loop with a bug in it (the “Bug Hunt” cell). **Identify the bug and explain why it can make training unstable. If the first two batch gradients are g_1 and g_2 , what is in each parameter’s `.grad` field after the second `loss.backward()` call? Does `optimizer.step()` clear `.grad`? Write the corrected loop.**

Solution: The bug is that `optimizer.zero_grad()` sits outside the loop, so it runs once instead of once per batch. PyTorch *accumulates* into `.grad`, so after the second `loss.backward()` each parameter holds $g_1 + g_2$ rather than g_2 . `optimizer.step()` does not clear `.grad`; it only reads it to update the parameters. The effective gradient is therefore a running sum over the epoch rather than the current batch gradient. The sum does not necessarily grow, since gradients from different batches can cancel, but each update no longer tracks the current batch, which can make training unstable. The fix is to zero the gradients once per batch:

```
for inputs, labels in training_loader:
    optimizer.zero_grad()
    predictions = model(inputs)
    loss = loss_fn(predictions, labels)
    loss.backward()
    optimizer.step()
```

(Accumulating deliberately is a legitimate technique for simulating a larger batch than fits in memory, but then one steps and zeroes every k batches, and usually scales the loss by $1/k$.)

Contributors:

- Matteo Guarrera.
- Mert Cemri.
- Sizhe Chen.
- Gireeja Ranade.
- Kevin Li.

- Anant Sahai.
- Kumar Krishna Agrawal.
- Naman Jain.