Lecture 1

Introduction

The Challenge of the Error Surface

- The error surface of a neural network can have very different curvature along different directions.
- When the contours of the error surface are elongated ellipsoids, the problem is said to be **poorly conditioned** or **ill-conditioned**.
- This creates a scenario resembling a long, narrow valley.

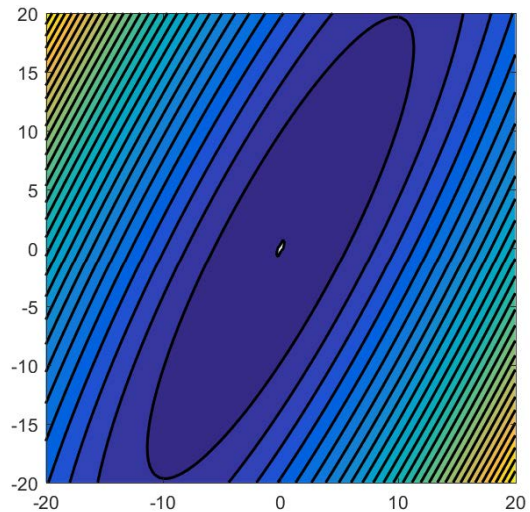


Figure 1: An ill-conditioned error surface with elliptical contour lines.

Inefficient Gradient Descent

A single learning rate struggles on an ill-conditioned surface.

- The gradient vector is always perpendicular to the contour lines.
- In a narrow valley, this means the gradient points mostly across the valley, not along it.
- **Result:** The optimization path ”zig-zags” inefficiently.
  - **Fast oscillations** across the steep direction (the valley walls).
  - **Slow progress** along the shallow direction (the valley floor).

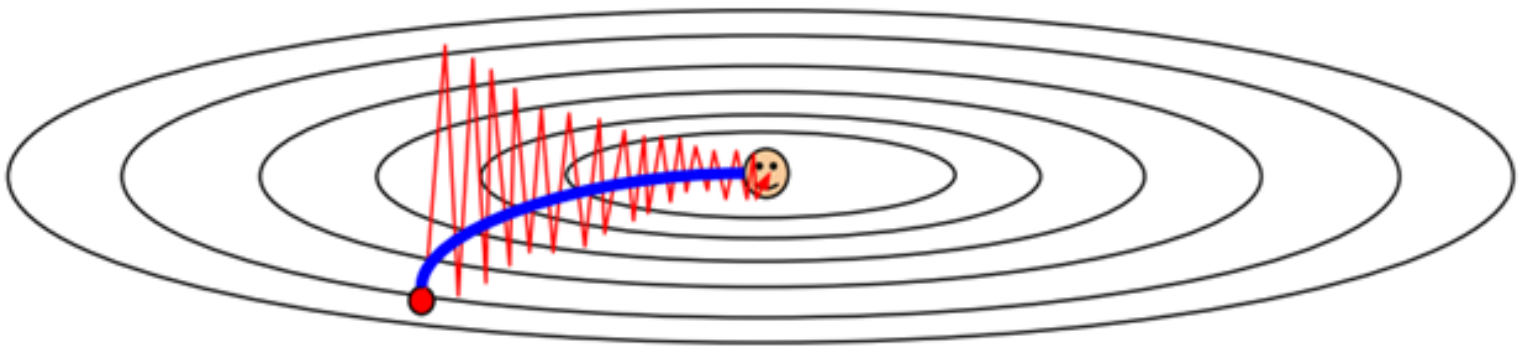


Figure 2: Gradient descent oscillates in the steep direction while making slow progress in the shallow direction.

One Step Size, Two Different Needs

The core issue is that different directions require different optimal learning rates.

- To prevent divergence, the learning rate η must be small enough for the steepest direction.
- This small learning rate is then far too small for the shallow direction, leading to extremely slow convergence.
- This conflict forces a compromise that is suboptimal for all directions.

The Motivation for Advanced Optimizers

Poor conditioning is a key reason that vanilla gradient descent is often too slow. This motivates methods that can adapt and take appropriately sized steps in different directions.

The Problem with a Single Learning Rate

- Vanilla Gradient Descent uses a single learning rate for all parameters.
- On ill-conditioned error surfaces (long, narrow valleys), this is inefficient.
- The learning rate is either too large for steep directions (causing oscillation) or too small for shallow directions (causing slow progress).
- **Question:** Can we do better by using information about the surface's curvature?

Using Curvature for Faster Convergence

Instead of a linear approximation, second-order methods use a local **quadratic approximation** of the loss function.

- This is based on the second-order Taylor expansion for loss function E(w) around the current point $w^{(k)}$:

$$E(w) \approx E(w^{(k)}) + \nabla_w E(w^{(k)})^T (w - w^{(k)}) + \frac{1}{2}(w - w^{(k)})^T H_E(w^{(k)})(w - w^{(k)})$$

- **Newton's Method** jumps directly to the minimum of this quadratic approximation in a single step.

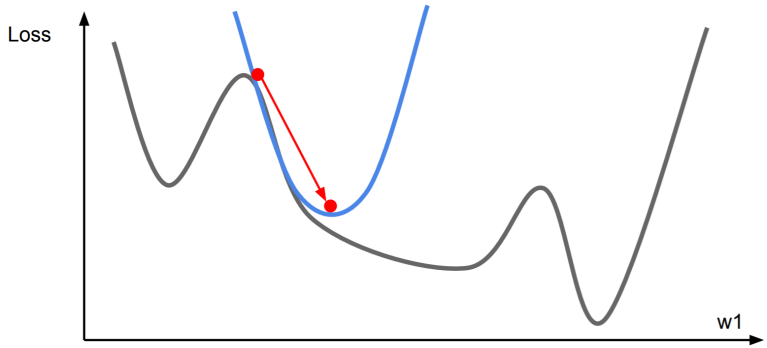


Figure 3: Newton's method forms a local quadratic approximation (blue) and jumps to its minimum.

Intuition Behind Newton's Method

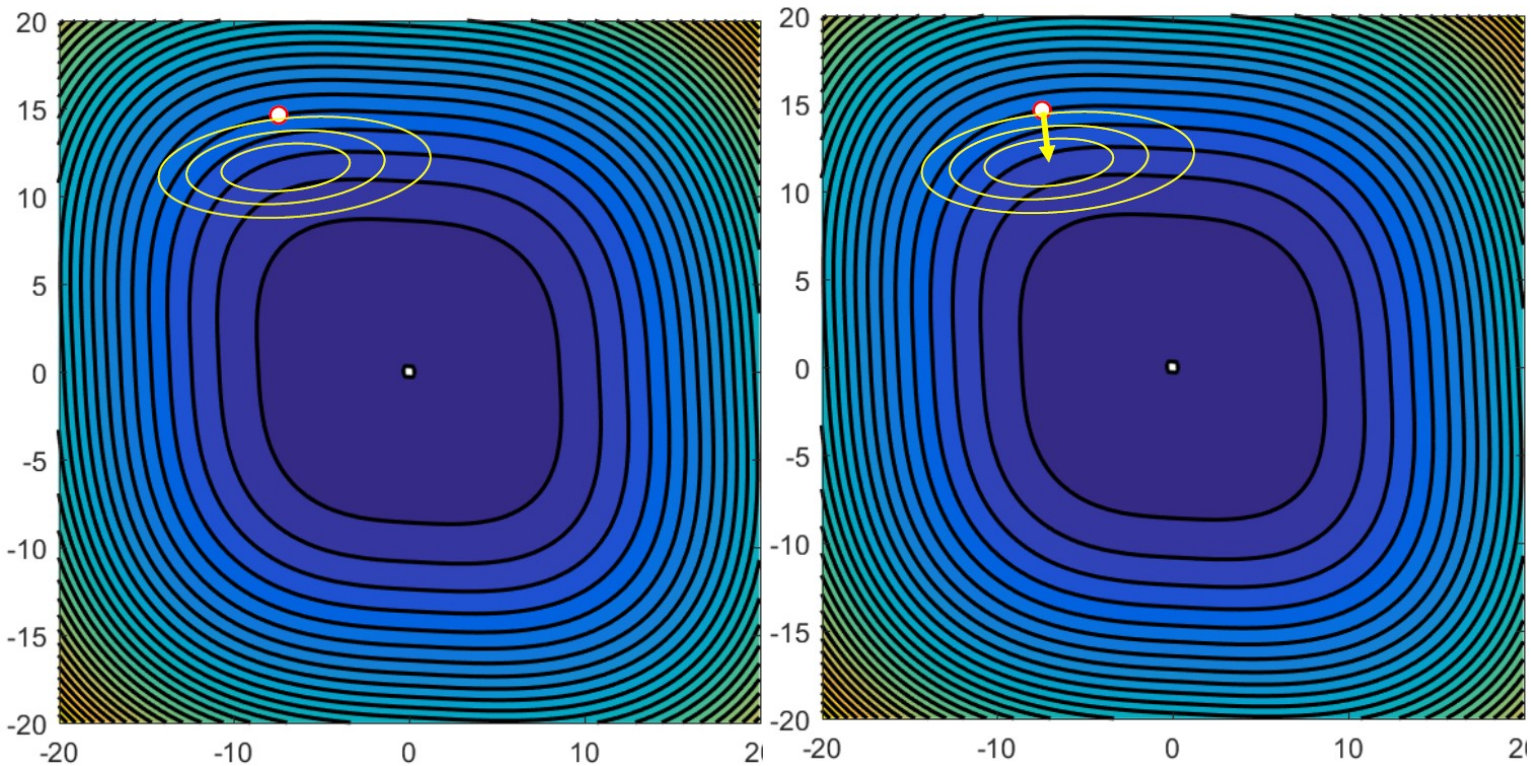


Figure 4: Fit a quadratic at each point and find the minimum of that quadratic

Incorporating the Hessian

The update rule incorporates the **Hessian matrix (H)**, the matrix of second derivatives, which describes the local curvature.

- The update rule is:

$$w^{(k+1)} = w^{(k)} - H_E(w^{(k)})^{-1} \nabla_w E(w^{(k)})$$

- By multiplying the gradient by the **inverse Hessian**, the update is rescaled. This effectively transforms the elliptical contours of an ill-conditioned problem into circular ones, allowing for a direct path to the minimum.
- For a quadratic function, the optimal η is 1 (which is exactly Newton's method). Therefore, this method has no learning rate/hyperparameter to tune; the optimal step size is effectively 1.

The Effect of Preconditioning

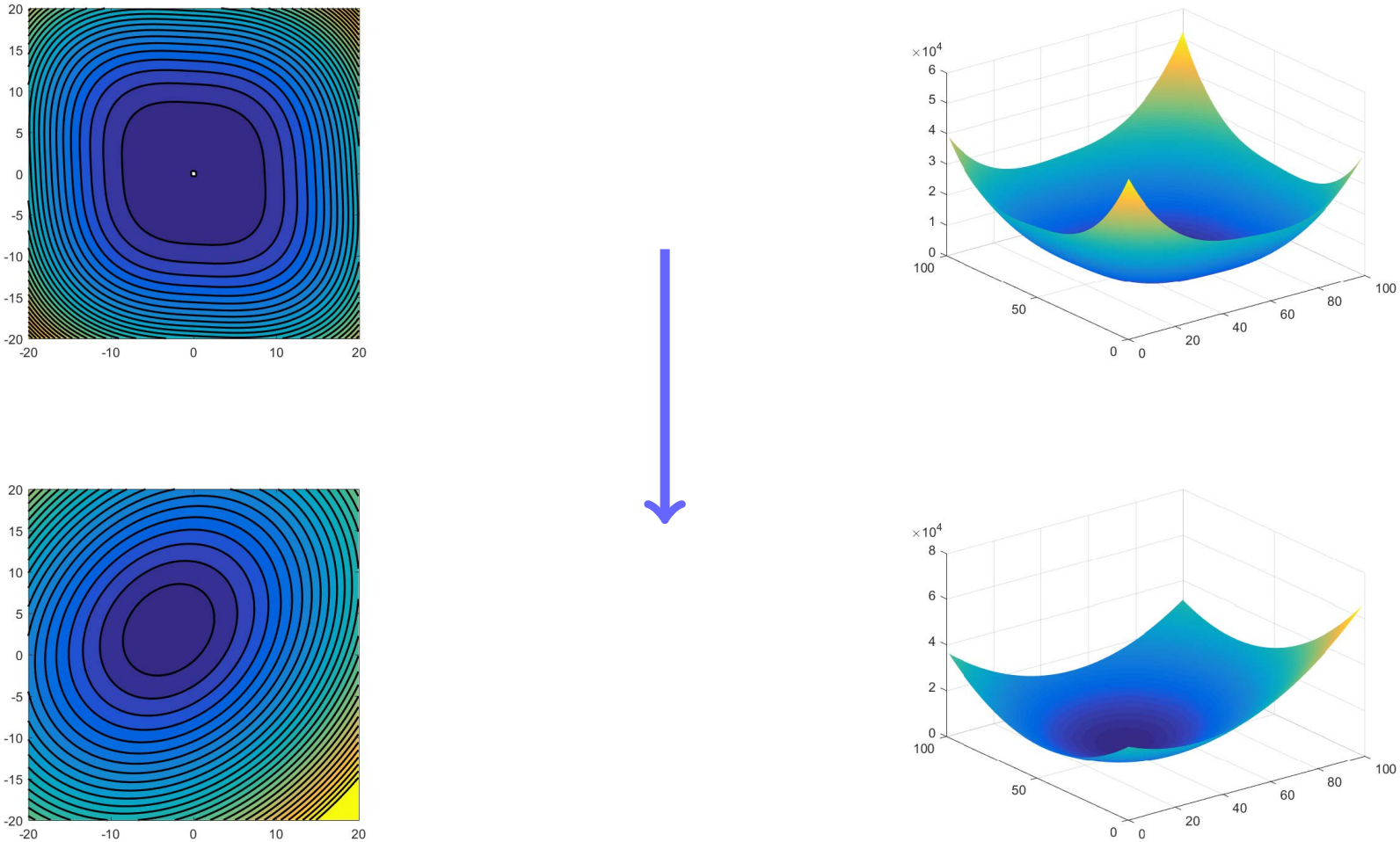


Figure 5: Visualizing the error surface before (top) and after (bottom) applying second-order information.

Why It's Impractical for Deep Learning

Despite its rapid convergence, Newton's method is not used for large-scale deep learning.

- **Prohibitive Computational Cost**:
  - For a network with n parameters, the Hessian matrix has n^2 elements.
  - Calculating the Hessian takes $O(n^2)$ time.
  - Inverting it takes $O(n^3)$ time. For millions of parameters, this is infeasible.
- **Stability Issues on Non-Convex Surfaces**:
  - For non-convex functions, the Hessian may not be positive definite. This can cause the update to move towards a saddle point or maximum instead of a minimum.

Approximating the Inverse Hessian

- **Quasi-Newton methods**, like BFGS, avoid the full computation of the inverse Hessian.
- They iteratively build up an approximation of the inverse Hessian using only first-order gradient information.
- **L-BFGS (Limited-memory BFGS)** is a popular variant that uses only the last few gradient updates, making it more memory-efficient.
- However, L-BFGS works best in a full-batch, deterministic setting and does not transfer well to the noisy, mini-batch updates common in deep learning.

Next Steps

Since second-order methods are too costly, we look for more sophisticated first-order methods that can tackle the conditioning problem without computing the Hessian.

Accelerating Gradient Descent

- Momentum methods are designed to accelerate learning, especially in deep, narrow valleys where vanilla gradient descent is slow.
- **The Core Idea:** Keep track of past gradients to build up ”velocity” in consistent directions and dampen oscillations in steep directions.
- This is analogous to a heavy ball rolling down the error surface; it builds up momentum and is less affected by small bumps and changes in gradient direction.

Intuition Behind Momentum

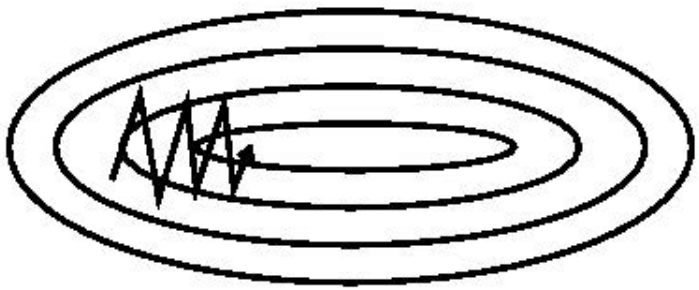


Figure 6: Plain gradient update

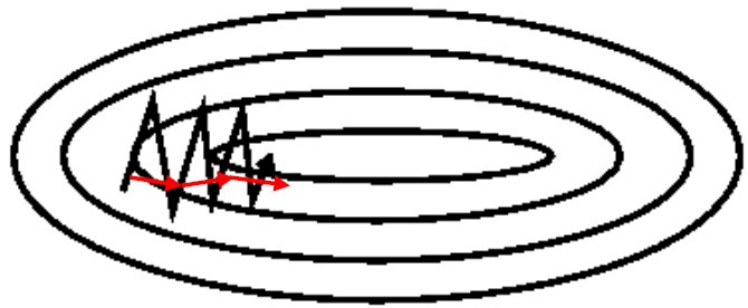


Figure 7: With momentum

Momentum helps accelerate in consistent directions and dampens oscillations.

The Momentum Update Rule

This method adds a fraction of the previous update vector to the current gradient step.

- A velocity vector, v, accumulates an exponentially decaying moving average of past gradients.
- The update rule is:

$$v^{(k)} = \beta v^{(k-1)} + \eta \nabla_w E(W^{(k-1)})$$

$$W^{(k)} = W^{(k-1)} - v^{(k)}$$

- β is the momentum term, typically set to a value like 0.9
  It determines how much of the past velocity is retained.

A Smarter Momentum

Nesterov Accelerated Gradient (NAG) is a slightly different, and often more effective, version of the momentum update.

- **The Idea:** Instead of calculating the gradient at the current position, NAG "looks ahead" by calculating the gradient at a point where the previous velocity would have taken it.
- This allows the optimizer to "correct" its course sooner if the gradient is changing, leading to faster convergence.
- The update rule is:

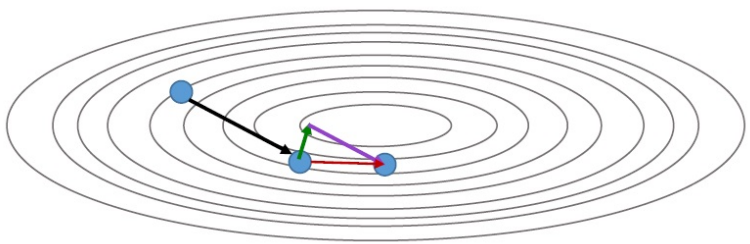
$$v^{(k)} = \beta v^{(k-1)} + \eta \nabla_w E(W^{(k-1)} - \beta v^{(k-1)})$$

$$W^{(k)} = W^{(k-1)} - v^{(k)}$$

The ”Look-Ahead” Advantage

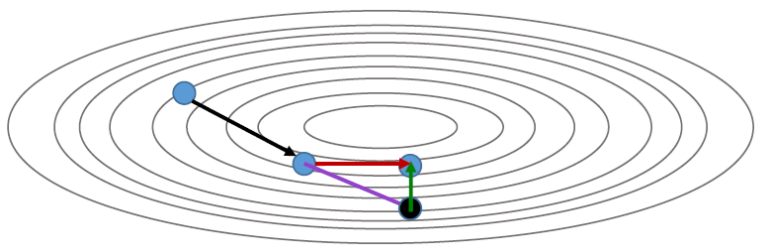
Standard Momentum

1. Computes gradient at current position.
2. Adds scaled previous velocity.



NAG

1. First, takes a step in direction of previous velocity.
2. Computes gradient at this ”look-ahead” position and makes a correction.



One Learning Rate to Rule Them All?

- We've seen that Momentum helps us move faster by building velocity.
- **But there's a problem:** It's like driving a car where all four wheels are forced to turn at the same angle. On a winding road (our ill-conditioned error surface), this is clumsy.
- Some parameters (wheels) need to turn sharply (small learning rate), while others need to go straight (large learning rate).
- **The Goal:** Give each parameter its own "smart" learning rate that adapts automatically to the terrain it's on.

Every Gradient Leaves a Mark

Adagrad was a pioneering idea that gave each parameter a personal learning rate.

- **The Strategy:** Keep a running sum of the squares of all past gradients for each parameter.
- Parameters that have seen large gradients in the past will have their learning rate aggressively decreased.
- The update rule:

$$s^{(k)} = s^{(k-1)} + (\nabla_w E)^2$$

$$W^{(k+1)} = W^{(k)} - \frac{\eta}{\sqrt{s^{(k)} + \epsilon}} \nabla_w E$$

- **The Fatal Flaw:** The denominator, $s^{(k)}$, is a sum that only ever grows, it eventually stops the learning process entirely as the learning rate vanishes.

Remembering Only the Recent Past

RMSProp solves Adagrad's problem by forgetting the distant past and focusing only on recent gradient history.

- Instead of a sum, it uses an **exponentially decaying moving average** of squared gradients.
- This prevents the denominator from growing indefinitely, allowing learning to continue.
- The update rule:

$$\overline{(\nabla_w E^2)}^{(k)} = \beta \overline{(\nabla_w E^2)}^{(k-1)} + (1-\beta)(\nabla_w E)^2$$

$$W^{(k+1)} = W^{(k)} - \frac{\eta}{\sqrt{\overline{(\nabla_w E^2)}^{(k)} + \epsilon}} \nabla_w E$$

- It effectively normalizes each parameter's update by the magnitude of its recent gradients.

Adaptive Moment Estimation

Adam is the most popular optimizer because it combines the best ideas we've seen so far.

- It's a hybrid of **Momentum** and **RMSProp**.
- It uses a moving average of the gradient itself (like Momentum) to track the direction of travel (the *first moment*).
- It uses a moving average of the squared gradient (like RMSProp) to adapt the learning rate for each parameter (the *second moment*).

The Result

An optimizer that knows both where it's going and how fast it should get there, for every single parameter.

Putting It All Together

Adam maintains two moving averages, m (for momentum) and v (for variance scaling), and includes a bias-correction step.

1. First moment (Momentum):

$$m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1) g^{(k)}$$

2. Second moment (RMSProp):

$$v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)(g^{(k)})^2$$

3. Bias Correction (to counteract initialization at zero):

$$\hat{m}^{(k)} = \frac{m^{(k)}}{1 - \beta_1^k} \quad , \quad \hat{v}^{(k)} = \frac{v^{(k)}}{1 - \beta_2^k}$$

4. Final Update Rule:

$$W^{(k+1)} = W^{(k)} - \frac{\eta}{\sqrt{\hat{v}^{(k)}} + \epsilon} \hat{m}^{(k)}$$

Common hyperparameters: η (needs tuning), $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

The ”Cold Start” Problem

- Adam’s moment vectors, m and v, are initialized to zero.
- In the first few iterations, the moving averages are heavily weighted by these initial zeros, making them biased towards zero.
- This ”cold start” causes the optimizer to take very small steps at the beginning of training, when it should be making the most progress.

The Solution: Bias Correction

The bias correction step divides the moment estimates by a factor that is initially small and approaches 1 over time. This scales up the early, biased estimates, giving the optimizer a ”warm start” and allowing it to take meaningful steps from the very beginning.

Is Newer Always Better?

- While Adam is a fantastic default, it's not a silver bullet.
- Recent research has shown that adaptive methods can sometimes converge to "sharper" minima that generalize more poorly than the "flatter" minima found by well-tuned SGD with Momentum.
- There is no single optimizer that dominates across all possible problems.
- **The Takeaway:** Adam is a great starting point, but if you're aiming for state-of-the-art results, it can be worth trying to carefully tune SGD+Momentum as well.

Overfitting and the Failure to Generalize

- A model that performs perfectly on training data but fails on new, unseen data has **overfitted**.
- It's like a student who memorizes the answers to past exams but hasn't learned the underlying concepts.
- This "generalization gap" is what we aim to reduce.

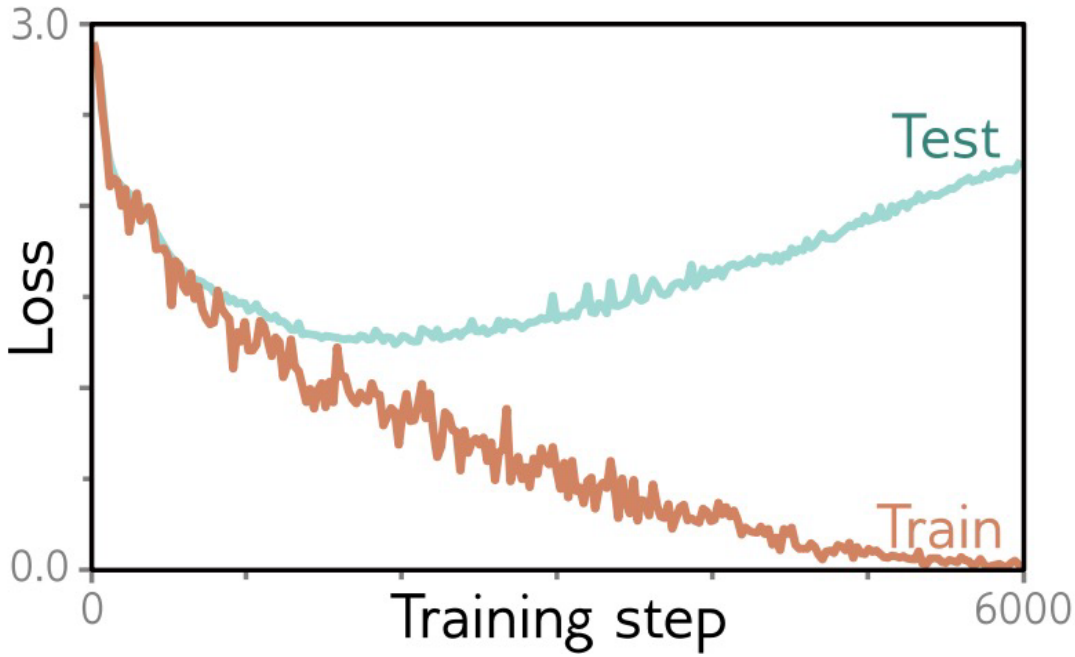


Figure 8: A classic sign of overfitting: training error continues to decrease while test (validation) error begins to rise.

Introducing Regularization

How do we encourage our model to learn the true underlying pattern instead of just memorizing the data points?

- We can add a penalty term $R(W)$ to our loss function.
- This penalty discourages the model from becoming too complex.
- Our new objective is to balance two competing goals:
  1. Fit the training data well (minimize the original loss).
  2. Keep the model simple (minimize the regularization penalty).
- The new cost function is:

$$J(W) = \underbrace{\sum_{n=1}^{N} \text{loss}(y^{(n)}, f(x^{(n)}; W))}_{\text{Data Fit}} + \underbrace{\lambda R(W)}_{\text{Complexity Penalty}}$$

L2 Regularization (Weight Decay)

L2 regularization penalizes the squared magnitude of the weights. It encourages the network to use small weights.

- The penalty term is the squared L2 norm of the weights:

$$R(W) = ||W||_2^2 = \sum_k \sum_l W_{k,l}^2$$

- The new cost function becomes:

$$J(W) = \text{Loss}(W) + \lambda ||W||_2^2$$

- This leads to a modified gradient update rule called **Weight Decay**:

$$W \leftarrow W - \alpha \nabla_W \text{Loss}(W) - 2\lambda W$$

$$W \leftarrow (1 - 2\lambda)W - \alpha \nabla_W \text{Loss}(W)$$

- At each step, the weights are multiplicatively shrunk before the gradient update.

Why Do Smaller Weights Generalize Better?

- **Smoother Functions:** Penalizing large weights forces the network to learn smoother, less complex functions.
- **Linear Regime:** With tanh or sigmoid activations, small weights keep neurons in their linear range. Since linear neurons can only learn simple linear functions, L2 regularization promotes this low-complexity behavior.

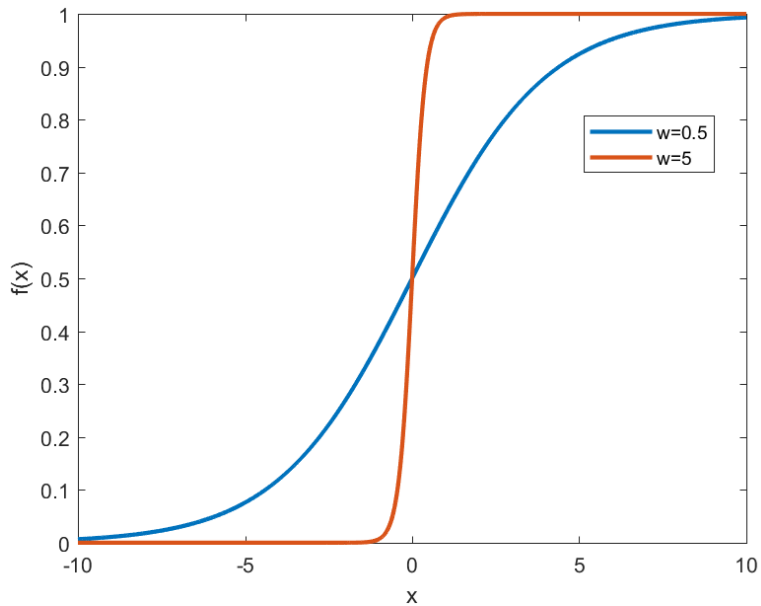


Figure 9: As weight ‘w‘ increases, the sigmoid function becomes much steeper and more non-linear, allowing for more complex responses.

Finding the Right Balance

The hyperparameter λ controls the trade-off between fitting the data and keeping the weights small.

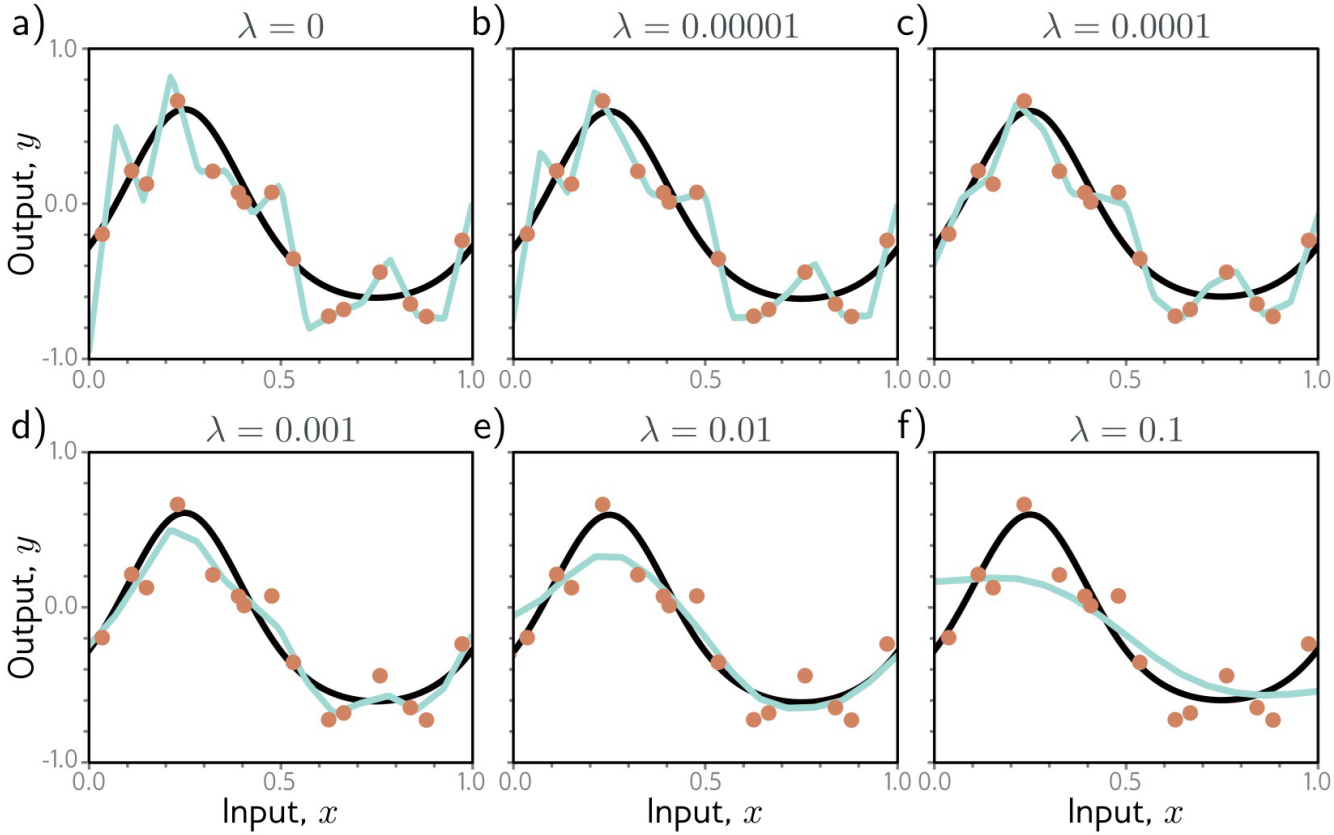


Figure 10: As λ increases, the learned function becomes smoother and less prone to overfitting, but too high a value can lead to underfitting.

Geometric Intuition of regularizations

L1 regularization penalizes the absolute value of the weights.

- The penalty term is the L1 norm of the weights:

$$R(W) = \|W\|_1 = \sum_k \sum_l |W_{k,l}|$$

- **Key Property:** L1 regularization is known for producing sparse weight vectors. It encourages many weights to be exactly zero.
- This can be interpreted as a form of automatic feature selection, as the network learns to ignore irrelevant inputs.

L1 vs. L2

Use L2 regularization as a default. L1 can be useful if you suspect many of your input features are irrelevant and you want a sparse, more interpretable model.

Geometric intuition of regularizations

L2 Regularization

- Circular constraint
- Smooth shrinkage

L1 Regularization

- Diamond constraint
- Corners hit first

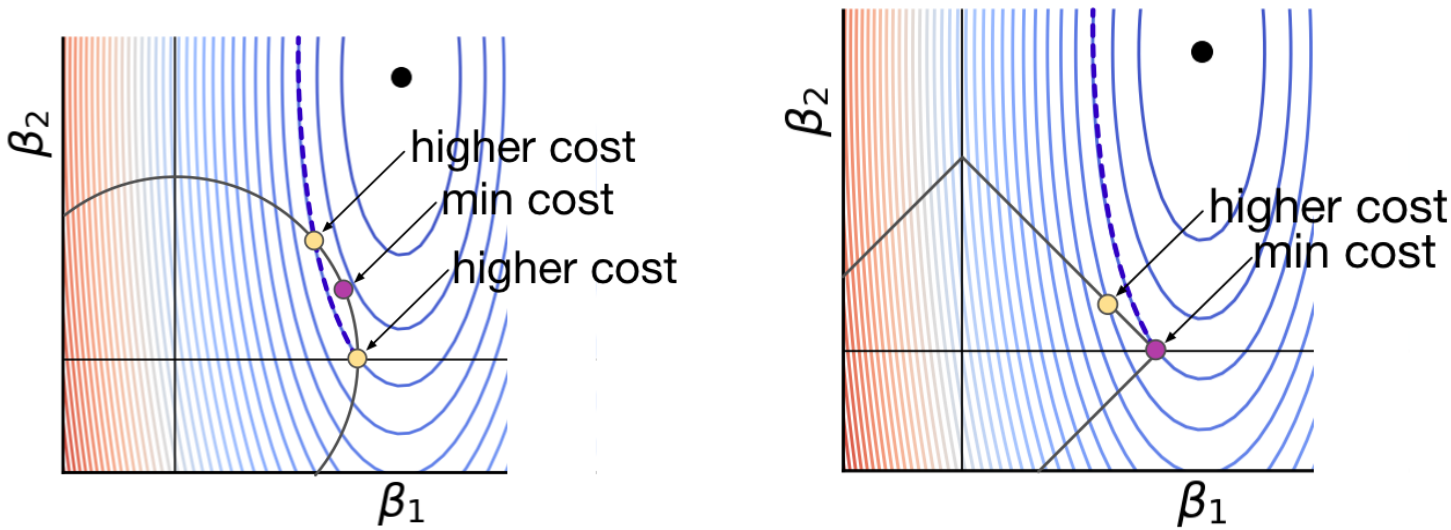


Figure 11: L1 and L2 constraint regions get different coefficient locations, on the diamond and circle, for the same loss function. Keep in mind that there are an infinite number of contour lines and there is a contour line exactly meeting the L2 purple dot.

Elastic Net: The Best of Both Worlds

What if you want both sparsity from L1 and the smoothing effect of L2?

- Elastic Net regularization is a hybrid approach that combines both L1 and L2 penalties.
- The penalty term is a simple weighted sum of the L1 and L2 norms:

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

- It can produce sparse models like L1.
- It is more stable and often performs better than L1 when features are highly correlated.