Lecture 1

Introduction

August 5, 2025

The Goal: A Functional View

What are we trying to build?

- At a high level, many tasks in AI can be seen as learning a complex function that maps a given input to a desired output.
- A neural network is a powerful tool for modeling such functions. Our goal is to find the right network that performs the specific mapping we need.

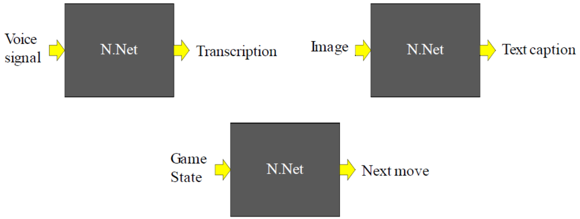


Figure 1: Neural networks act as functions mapping inputs (like voice or images) to outputs (like text or actions).

The Goal: The Formal Problem Setup

What we have and what we want

Based on the previous slides, we can formally define our task:

- **Given:**
  - The architecture of the network (e.g., number of layers and neurons).
  - A set of N training data pairs: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})$.
- **To Find:**
  - The optimal set of parameters (weights and biases, denoted collectively as W) for our network.

The Goal: The Parametric Function and Cost

Representing the Network and its Error

- We consider a neural network as a parametric function, $f(x; W)$, where W represents all learnable parameters.
- A loss function, $\text{loss}(f(x; W), y)$, penalizes the difference between the network's prediction and the desired output for a *single* training example.
- The overall Cost Function $E(W)$ is the average loss over the *entire* dataset:

$$E(W) = \frac{1}{N} \sum_{n=1}^{N} \text{loss}(f(x^{(n)}; W), y^{(n)})$$

The Goal: A Key Requirement

The Need for Differentiability

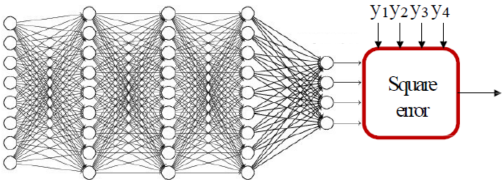
- Our goal is to minimize the cost $E(W)$. To do this with gradient-based methods, the cost function must be differentiable with respect to the weights W.
- This means we must use:
    - **Differentiable Loss Functions:** The way we measure error must be smooth.
    - **Continuous Activation Functions:** The functions inside our neurons (like Sigmoid or ReLU) must be differentiable, allowing gradients to flow through the network.

Case Study: Regression

Output and Loss for Real-Valued Targets

- For tasks where the desired output is a real number or a vector of real numbers (e.g., predicting a price).
- **Output Layer:** Typically has linear neurons (i.e., no activation function is applied).
- **Loss Function:** The most common choice is the Squared Error (or L2 loss):

$$\text{loss}(y, o) = \frac{1}{2}\|y - o\|^2 = \frac{1}{2}\sum_k (y_k - o_k)^2$$

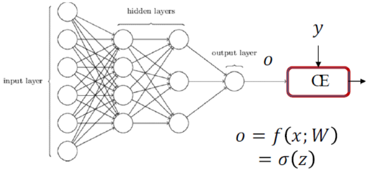


Case Study: Binary Classification

Output and Loss for Two-Class Problems

- For tasks with two classes (e.g., Cat vs. Dog), where the target y is either 0 or 1.
- **Output Layer:** A single neuron with a Sigmoid activation function. This squashes the output to a range of (0, 1), which we can interpret as a probability $P(Y = 1|x)$.
- **Loss Function:** Binary Cross-Entropy is the standard choice:

$$\text{loss}(y, o) = -y \log(o) - (1 - y) \log(1 - o)$$



Case Study: Multi-Class Classification

Setup for K > 2 Classes

- For tasks with multiple classes (e.g., MNIST digits 0-9).
- **Target Representation:** The desired output y is represented as a one-hot vector. For example, for class 3 out of 5, $y = [0, 0, 1, 0, 0]^T$.
- **Output Layer:** Must have K neurons, one for each class.
- To ensure the outputs are probabilities that sum to 1, we need a special activation function.

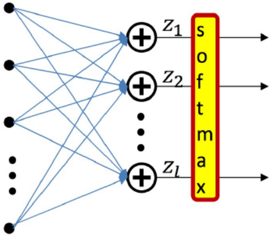
Case Study: Multi-Class Classification

The Softmax Activation

- The Softmax function is used as the activation for the output layer in multi-class problems.
- It takes a vector of raw scores (logits) z and transforms it into a probability distribution o:

$$o_i = \mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

- Each output o_i is between 0 and 1, and all outputs sum to 1, making them valid probabilities.



Case Study: Multi-Class Classification

Cross-Entropy Loss

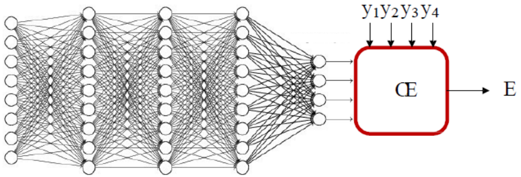
- **Loss Function:** For multi-class classification, we use the Cross-Entropy loss.

$$\text{loss}(y, o) = -\sum_{i=1}^{K} y_i \log(o_i)$$

- Since y is a one-hot vector, only one term in the sum is non-zero. If the true class is c, the formula simplifies to:

$$\text{loss}(y, o) = -\log(o_c)$$

- This means we are trying to maximize the log-probability of the correct class.



The Tool for Optimization

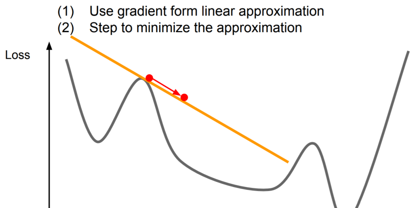
How do we find the minimum?

- We have defined our goal: find the set of weights W that minimizes the cost function $E(W)$.
- The cost function for a neural network is a complex function in a very high-dimensional space (one dimension for each parameter).
- We cannot solve for the minimum analytically. Instead, we must search for it using an iterative optimization algorithm.

The Tool: Gradient Descent

The Core Idea

- The most common optimization algorithm is gradient descent.
- The core idea is to start with an initial random guess for the weights (w^0) and take small, iterative steps in the direction of the negative gradient of the error surface.
- **Analogy:** Imagine a blindfolded hiker trying to find the bottom of a valley. At any point, they feel the slope of the ground under their feet and take a step in the steepest downhill direction. They repeat this process until they reach a point where the ground is flat.


(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Loss

The Tool: Gradient Descent

What is the Gradient?

- In a multi-dimensional space, the derivative is called the gradient, denoted as $\nabla_w J(w)$.
- The gradient is a vector that contains the partial derivative of the cost function with respect to every single weight in the network:

$$\nabla_w J(w) = \left[\frac{\partial J(w)}{\partial w_0}, \frac{\partial J(w)}{\partial w_1}, \dots, \frac{\partial J(w)}{\partial w_d} \right]^T$$

- Crucially, the gradient vector always points in the direction of the steepest ascent of the cost function. Therefore, the negative gradient points in the direction of the steepest descent.

The Tool: Gradient Descent

The Update Rule

- At each iteration t, we update the entire vector of weights according to the following rule:

$$w^{t+1} = w^t - \eta \nabla_w J(w^t)$$

- Where:
  - w^t is the vector of all weights in the network at iteration t.
  - η (eta) is the learning rate, a critical hyperparameter that controls the step size. If it's too small, learning is slow; if it's too large, the algorithm might overshoot the minimum and diverge.
  - $\nabla_w J(w^t)$ is the gradient of the cost function, evaluated at the current weights w^t.

The Tool: Gradient Descent

The Challenge: Computing the Gradient

- The update rule is simple, but how do we compute the gradient $\nabla_w J$ for a deep network with potentially millions of parameters?
- The cost function is a deeply nested composite function:

$$E(W) = \text{loss}(f(W^{[L]} \ldots f(W^{[2]} f(W^{[1]} x)) \ldots))$$

- Calculating the derivative for each weight by hand is impossible and computationally inefficient. We need an automated and efficient method.

The Tool: Gradient Descent

The Solution: Backpropagation

- The gradient $\nabla_w J$ is computed efficiently using the backpropagation algorithm.
- Backpropagation is not the training algorithm itself; it is the procedure for efficiently calculating the gradient. Gradient descent is the training algorithm that *uses* this gradient.
- It is a clever implementation of the chain rule from calculus, combined with dynamic programming to avoid re-computing intermediate values.

The Tool: Gradient Descent

The Two-Step Process of Backpropagation

The algorithm works in two stages for each training example (or mini-batch):

1. **Forward Pass:**
   - The input data x is fed into the network.
   - The computation flows forward through the layers, calculating the pre-activations (z) and activations (a) at each layer, until the final output $f(x; W)$ is produced.
   - The loss between the output and the true label y is calculated.
2. **Backward Pass:**
   - The gradient of the loss is propagated backward from the output layer to the input layer.
   - At each layer, the algorithm calculates the gradient of the loss with respect to that layer's weights and biases.

The Computational Graph

Why do we need them?

- As we've seen, the output of a neural network is a deeply nested composite function.
- To calculate the gradient of the loss with respect to any weight, we must use the chain rule from calculus.
- For a network with millions of parameters, applying the chain rule manually is impossible. A computational graph is a powerful visualization tool that helps us manage this complexity and automate the process.

The Computational Graph

Structure

- A computational graph represents a function as a directed graph:
  - Nodes (Vertices): Represent operations (e.g., addition, multiplication) or variables.
  - Edges: Represent the flow of data. The output of one node becomes the input to another.

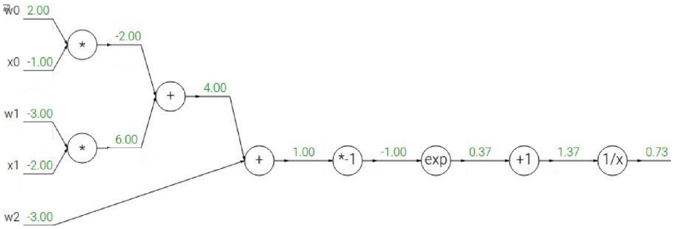


Figure 4: A computational graph breaking down a sigmoid neuron into its elementary operations.

The Computational Graph

The Forward Pass

- The forward pass is the process of evaluating the function represented by the graph.
- We start from the input nodes and move forward through the graph, applying the operation at each node to the incoming values.
- This continues until we reach the final node, which typically represents the overall loss of the network. The path is determined by the topological sort of the graph's nodes.

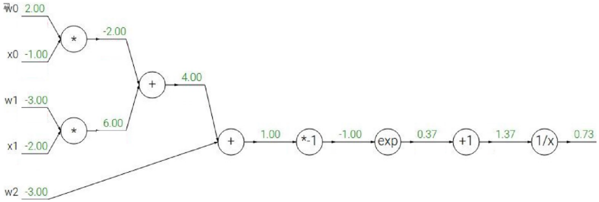


Figure 5: In the forward pass, values are computed from left to right. For example, $w_0 \times x_0 = 2.00 \times -1.00 = -2.00$.

The Computational Graph
The Backward Pass and The Chain Rule

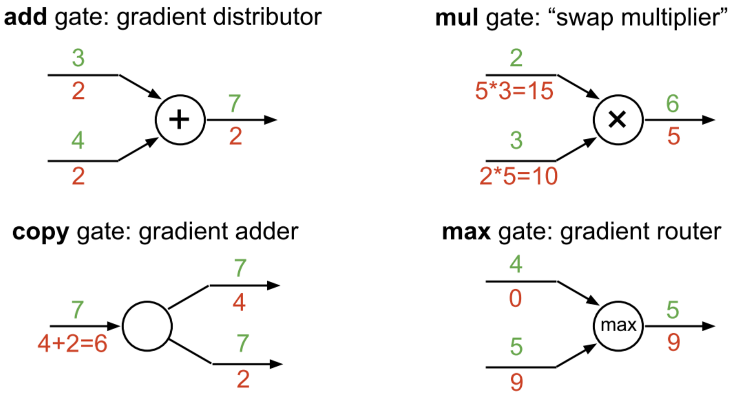
- Backpropagation is the process of calculating the gradients by moving backward through the graph.
- At each node, we use the chain rule to compute the gradient of the final loss with respect to the node's *inputs*, given the gradient with respect to its *output*.
- We define three key terms:
    - Upstream Gradient ($\frac{\partial L}{\partial z}$): The gradient coming from the nodes that follow.
    - Local Gradient ($\frac{\partial z}{\partial x}$): The derivative of the node's operation with respect to its own inputs.
    - Downstream Gradient ($\frac{\partial L}{\partial x}$): The gradient we want to compute and pass backward.

$$\text{Downstream Gradient} = \text{Upstream Gradient} \times \text{Local Gradient}$$

The Computational Graph

Example ”Gates”

- Different nodes (or ”gates”) distribute the upstream gradient in unique ways based on their local gradient:



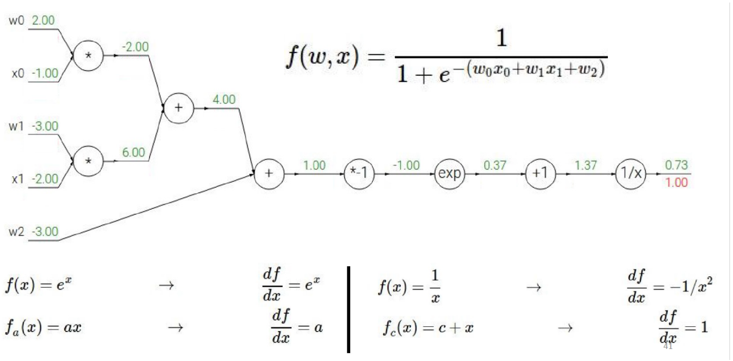
add gate: gradient distributor
3
2
4
2
+
7
2
mul gate: “swap multiplier”
2
5*3=15
3
2*5=10
×
6
5
copy gate: gradient adder
7
4+2=6
7
4
7
2
max gate: gradient router
4
0
5
9
max
5
9

Figure 6: **Add gate:** acts as a ”gradient distributor”. **Max gate:** acts as a ”gradient router”. **Multiply gate:** acts as a ”swap multiplier”.

The Computational Graph

Backpropagation Example: Step 1

- Let's trace the backward pass for our sigmoid neuron example.
- We start at the very end. The gradient of the output with respect to itself is 1. This is our initial upstream gradient.



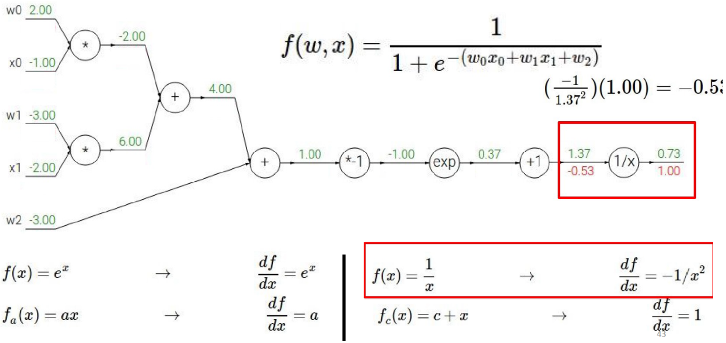
w0 2.00
x0 -1.00
w1 -3.00
x1 -2.00
w2 -3.00
*
-2.00
6.00
+
4.00
1.00
*-1
-1.00
exp
0.37
+1
1.37
1/x
0.73
1.00
$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$
$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$
$f_a(x) = ax \rightarrow \frac{df}{dx} = a$
$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$
$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$
41

Figure 7: Starting the backward pass. The initial gradient at the output is 1.

The Computational Graph

Backpropagation Example: Step 2

- The first gate is the $1/x$ gate.
- **Upstream Gradient**: 1.0
- **Local Gradient**: The derivative of $f(x) = 1/x$ is $-1/x^2$. Since the input was 1.37, the local gradient is $-1/(1.37^2) \approx -0.53$.
- **Downstream Gradient**: $1.0 \times -0.53 = -0.53$.



$$f(w,x) = \frac{1}{1 + e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\left(\frac{-1}{1.37^2}\right)(1.00) = -0.5$$

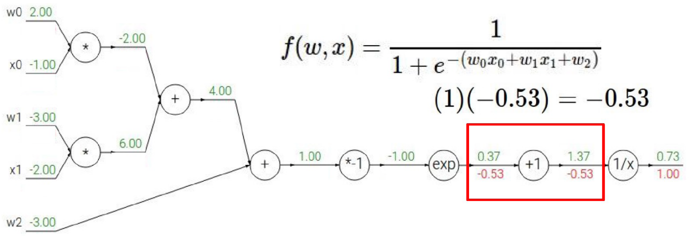
$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a \qquad f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

The Computational Graph

Backpropagation Example: Step 3

- The next gate is the ‘+1‘ gate.
- **Upstream Gradient**: -0.53
- **Local Gradient**: The derivative of $f(x) = x + c$ is 1.
- **Downstream Gradient**: $-0.53 \times 1 = -0.53$.



$$f(w,x) = \frac{1}{1 + e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$(1)(-0.53) = -0.53$$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

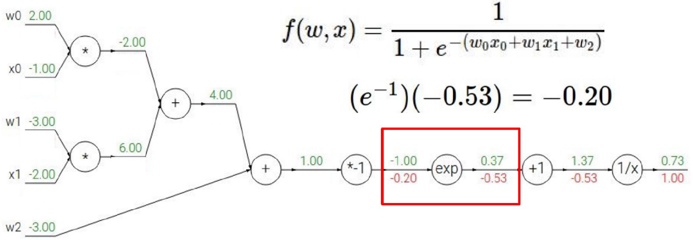
The Computational Graph

Backpropagation Example: Step 4

- Next is the ‘exp‘ gate.
- **Upstream Gradient**: -0.53
- **Local Gradient**: The derivative of $f(x) = e^x$ is e^x. Since the input was -1.0, the local gradient is $e^{-1} \approx 0.37$.
- **Downstream Gradient**: $-0.53 \times 0.37 \approx -0.20$.

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

$$(e^{-1})(-0.53) = -0.20$$



$$\boxed{f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x}$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

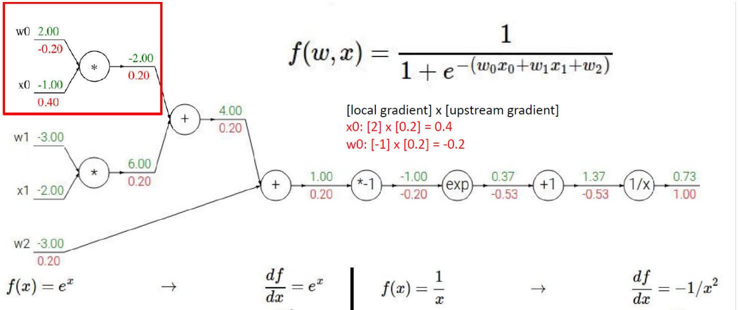
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

The Computational Graph

Backpropagation Example: Final Steps

- The gradient is propagated backward through the remaining gates (‘*-1‘, ‘+‘, ‘*‘).
- For example, at the top-most ‘*‘ gate (for w_0, x_0):
  - **Upstream Gradient**: 0.20 (from the final ‘+‘ gate).
  - **Local Gradient** for w_0: The other input, $x_0 = -1.00$.
  - **Downstream Gradient** for w_0: $0.20 \times -1.00 = -0.20$.
- This process is repeated until we have the gradient for every weight and input.



$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

[local gradient] x [upstream gradient]
x0: [2] x [0.2] = 0.4
w0: [-1] x [0.2] = -0.2

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$ | $f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

From Scalars to Vectors

The Need for Efficiency

- While the computational graph with scalar operations is great for understanding, it's extremely inefficient in practice.
- Modern deep learning libraries (like PyTorch or TensorFlow) do not compute gradients one by one. They rely on vector and matrix operations.
- These operations are highly optimized to run in parallel on GPUs (Graphics Processing Units), which can perform thousands of multiplications and additions simultaneously. This is the key to training large networks in a reasonable amount of time.

From Scalars to Vectors

The Jacobian Matrix

- When dealing with vector functions, the concept of a simple derivative is extended to the Jacobian matrix.
- The Jacobian is a matrix containing all possible partial derivatives of each output element with respect to each input element.
- For element-wise activation functions like Sigmoid or ReLU, the Jacobian is a simple diagonal matrix, which

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_m}{\partial x_2} \\ \dots & \dots & \ddots & \dots \\ \frac{\partial y_!}{\partial x_d} & \frac{\partial y_2}{\partial x_d} & \dots & \frac{\partial y_m}{\partial x_d} \end{bmatrix}$$

Figure 9: Structure of a Jacobian matrix for a vector function $y = f(x)$.

From Scalars to Vectors

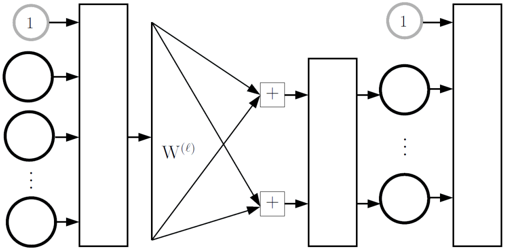
Vectorized Forward Pass

- For a full layer l, we can compute the pre-activations $z^{[l]}$ for all neurons in that layer with a single matrix-vector multiplication:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

- Then, the activations $a^{[l]}$ are computed by applying the activation function f element-wise to the vector $z^{[l]}$:

$$a^{[l]} = f(z^{[l]})$$



From Scalars to Vectors

Vectorized Backward Pass: The Sensitivity Vector

- In the backward pass, our goal is to compute the gradient of the loss with respect to all parameters.
- The key quantity we propagate backward is the sensitivity vector (often called ”error” or ”delta”), defined as the gradient of the loss with respect to the pre-activation vector z of a layer:

$$\delta^{[l]} = \frac{\partial \mathsf{Loss}}{\partial z^{[l]}}$$

- This vector tells us how sensitive the final loss is to small changes in the input of the activation function for each neuron in layer l.

From Scalars to Vectors

Vectorized Backward Pass: The Chain Rule

- We can compute the sensitivity of layer $l - 1$ from the sensitivity of layer l using the vectorized chain rule:

$$\delta^{[l-1]} = \frac{\partial z^{[l]}}{\partial a^{[l-1]}} \frac{\partial a^{[l-1]}}{\partial z^{[l-1]}} \frac{\partial \text{Loss}}{\partial z^{[l]}}$$

- This simplifies to a clean, efficient update rule:

$$\delta^{[l-1]} = (W^{[l]T} \delta^{[l]}) \odot f'(z^{[l-1]})$$

where $\odot$ is the element-wise product (Hadamard product). This formula is the core of backpropagation in practice.

From Scalars to Vectors

Vectorized Gradient Calculation

- Once we have the sensitivity vector $\delta^{[l]}$ for a given layer, the gradients for the *entire* weight matrix $W^{[l]}$ and bias vector $b^{[l]}$ of that layer can be computed with single vector operations:

$$\frac{\partial \text{Loss}}{\partial W^{[l]}} = \delta^{[l]}(a^{[l-1]})^T$$

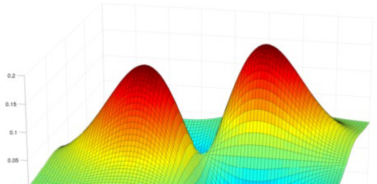
$$\frac{\partial \text{Loss}}{\partial b^{[l]}} = \delta^{[l]}$$

- This vectorized approach is significantly more efficient than looping through individual weights and is the reason deep learning is computationally feasible today.

The Landscape of Optimization

A Complex Surface

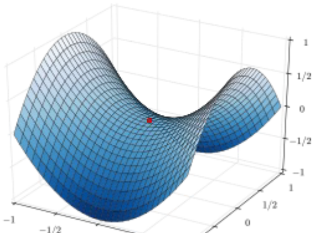
- The error surface $E(W)$ for deep networks is extremely complex and non-convex. This means it's not a simple bowl shape, but rather a landscape filled with hills, valleys, and plateaus.
- This non-convexity introduces significant challenges for our gradient descent algorithm. The two main challenges are:
  - **Local Minima:** Points that look like the bottom of a valley locally, but are not the true lowest point of the entire landscape.
  - **Saddle Points:** Points where the gradient is zero, but they are not minima.



The Landscape: Challenges

Saddle Points

- A saddle point is a point where the gradient is zero, but it is a minimum along some dimensions and a maximum along others.
- Think of the shape of a horse's saddle: if you move forward or backward you go up, but if you move side to side you go down.
- For vanilla gradient descent, the gradient at a saddle point is zero (or very close to it), causing the algorithm to slow down dramatically or even get permanently "stuck".



The Landscape: Challenges

Saddle Points vs. Local Minima

- A key insight in modern deep learning is that in very high-dimensional spaces (networks with many parameters), saddle points are exponentially more common than local minima.
- For a point to be a local minimum, the curvature must be positive in *all* dimensions. The probability of this happening simultaneously in millions of dimensions is extremely small.
- This is actually good news! It means that most of the points with zero gradient that our optimizer finds are saddle points, which advanced optimizers (like those with momentum) are better at escaping, rather than ”bad” local minima that would permanently trap the algorithm.