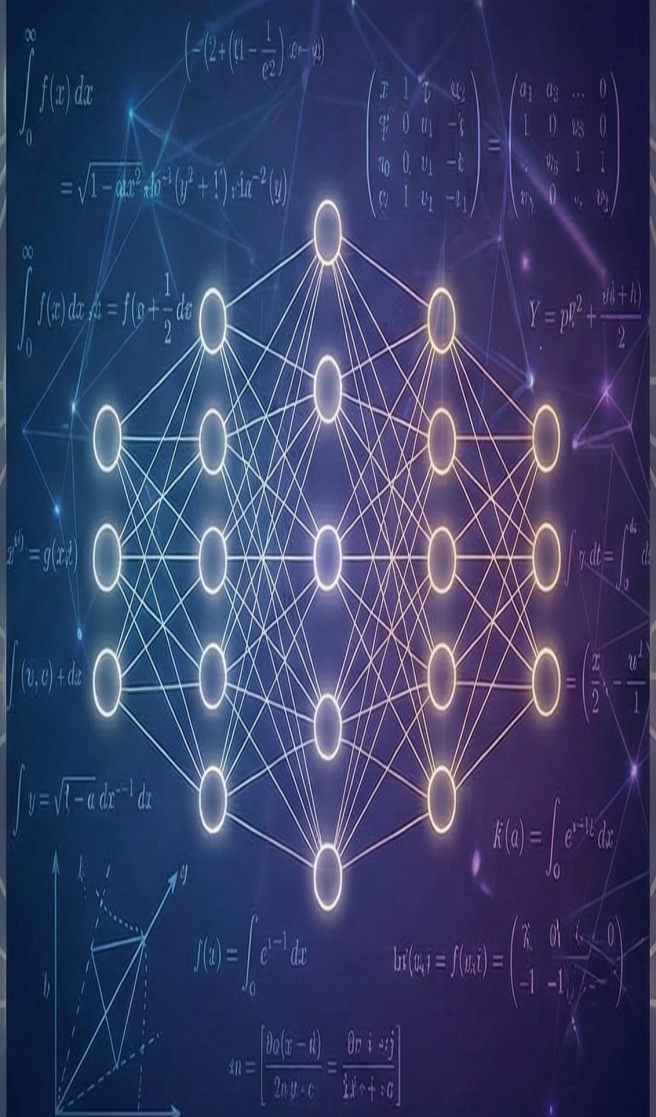
The Mathematical Intuition Behind Deep Learning



Alex Punnen

The Maths of Deep Learning

Alex Punnen

Table of contents

1 The Maths behind Neural Networks

Alex Punnen

1.1 Contents

2 The simplest Neural Network - Perceptron using Vectors and Dot Products

Even the most complex Neural network is based on vectors and matrices, and it uses the concept of a cost function and algorithms like gradient descent to find a reduced cost. Then, it propagates the cost back to all constituents of the network proportionally via a method called back-propagation.

Have you ever held an integrated circuit or chip in your hand or seen one? It looks overwhelmingly complex. But its base is the humble transistor and Boolean logic. To understand something complex, we need to understand the simpler constituents.

2.1 The Magic of Representation - Vector Space and Hyperplane

Most people are familiar with neural networks, cost functions, gradient descent, and backpropagation. However, beyond these building blocks is the magic of representations.

Features live in a multidimensional universe where the concept of a **hyperplane** classifies or clusters similar features together.

This idea applies equally to the simplest neural networks and to modern architectures such as Transformers.

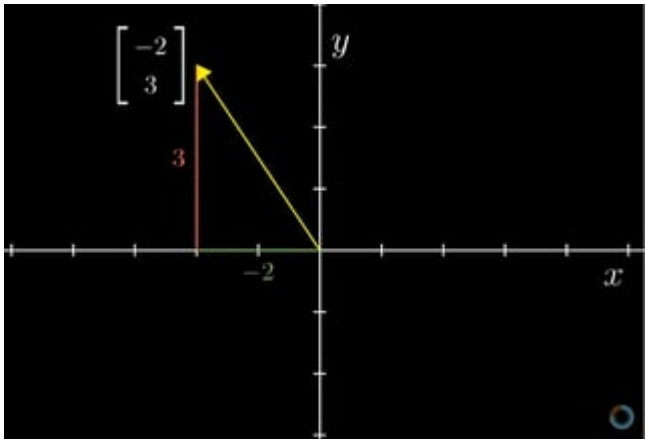
One of the earliest neural networks, **Rosenblatt's Perceptron**, introduced the idea of representing inputs as **vectors** and using the **dot product** to define a decision boundary — a hyperplane that separates input feature vectors.

First a short refresher.

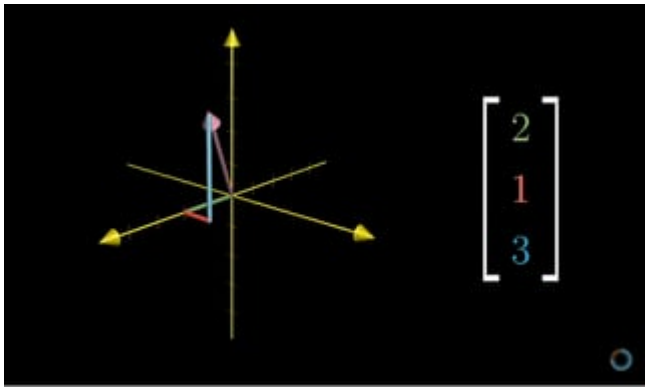
2.1.1 Vectors

A vector is an object that has both a magnitude and a direction. Example Force and Velocity. Both have magnitude as well as direction.

However we need to specify also a context where this vector lives -Vector Space. For example when we are thinking about something like Force vector, the context is usually 2D or 3D Euclidean world.



2Dvector



3Dvector

(Source: 3Blue1Brown)

The easiest way to understand the Vector is in such a geometric context, say 2D or 3D cartesian coordinates, and then extrapolate it for other Vector spaces which we encounter but cannot really imagine.

2.1.2 Matrices - A way to represent Vectors (and Tensors)

Vectors are represented as matrices. A Vector is a one dimensional matrix.A matrix is defined to be a rectangular array of numbers. Example here is a Euclidean Vector in three-dimensional Euclidean space (or R^3) with some magnitude and direction (from (0,0,0) origin in this case).

A vector is represented either as column matrix (m*1)or as a row matrix (1*m).

$$a = \begin{matrix} a_1 \\ a_2 \\ a_3 \end{matrix} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$

a_1, a_2, a_3 are the component scalars of the vector. A vector is represented as $\overrightarrow{a}$ in the **Vector notation** and as a_i in the **Index Notation**.

2.1.3 Two dimensional matrices can be thought of as one dimensional vectors stacked on top of each other.

This intuition is especially helpful when we use dot products on neural network weight matrices.

2.1.4 Dot Product

This is a very important concept in linear algebra and is used in many places in machine learning.

Algebraically, the dot product is the sum of the products of the corresponding entries of the two sequences of numbers.

if $\overrightarrow{a} = \langle a_1, a_2, a_3 \rangle$ and $\overrightarrow{b} = \langle b_1, b_2, b_3 \rangle$, then

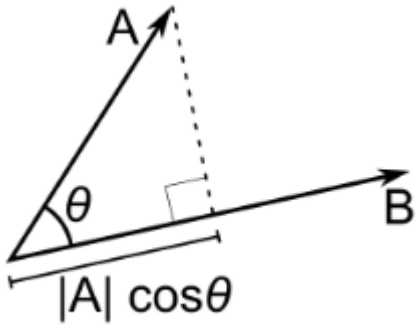
$$\overrightarrow{a} \cdot \overrightarrow{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 = a_i b_i \quad \text{in index notation}$$

In Matrix notation,

$$\overrightarrow{a} \cdot \overrightarrow{b} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{matrix} b_1 \\ b_2 \\ b_3 \end{matrix} = a_i b_i$$

Geometrically, it is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them

$$\overrightarrow{a} \cdot \overrightarrow{b} = \left| \overrightarrow{a} \right| \left| \overrightarrow{b} \right| \cos\theta$$



dotproduct

Note- These definitions are equivalent when using Cartesian coordinates (Ref 8, 9)

If two vectors point in roughly the same direction, their dot product is positive. If they point in opposite directions, the dot product is negative.

This simple geometric fact becomes a powerful computational tool.

Imagine a problem where we want to classify whether a leaf is healthy or diseased based on certain features. Each leaf is represented as a feature vector in a two-dimensional space (for simplicity).

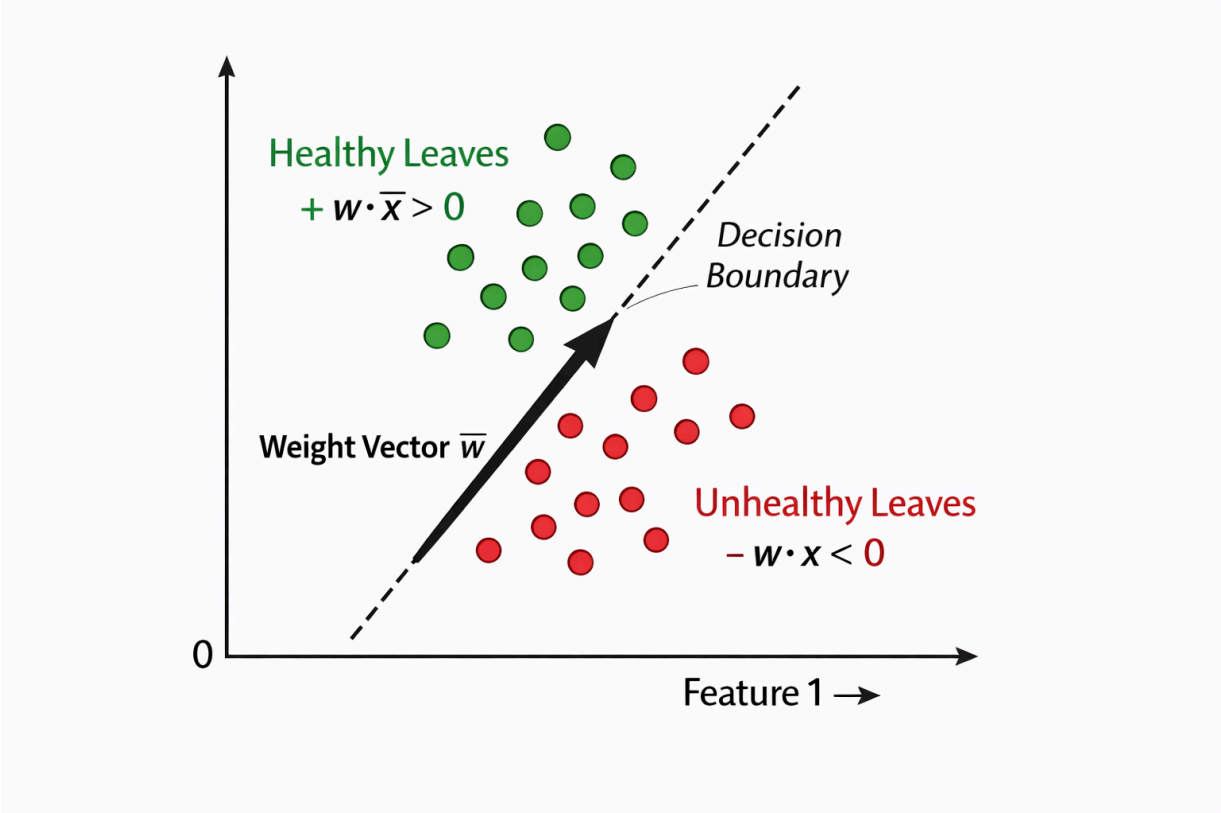
If we can find a weight vector such that:

Its dot product with healthy leaf vectors is positive

Its dot product with diseased leaf vectors is negative

then that weight vector defines a hyperplane that splits the feature space into two regions.

This is exactly how first artificial neuron-the Perceptron performs classification.

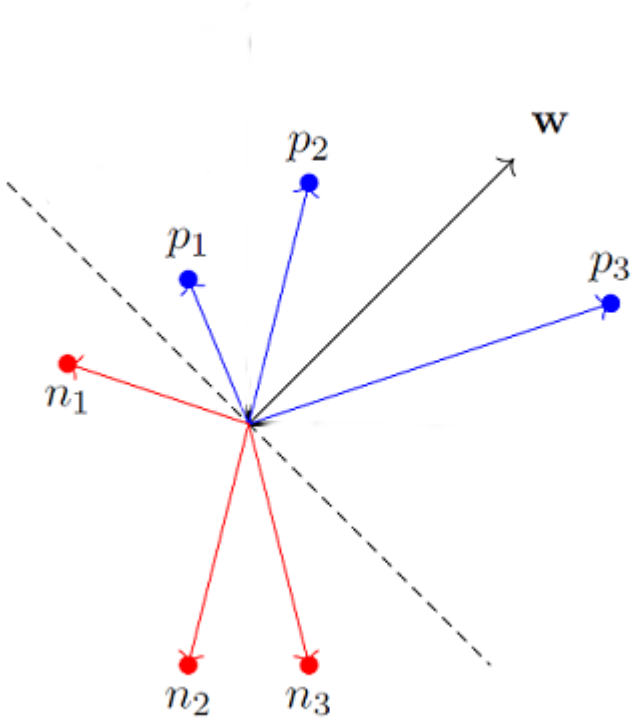


hyperplane1

Imagine we have a problem of classifying if a leaf is healthy or not based on certain features of the leaf. For each leaf we have some feature vector set assume it is a 2D vector space with say color as the feature for simplicity.

For any **input feature vector** in that vector space, if we have a **weight vector**, whose dot product with one feature vector of the set of input vectors of a certain class (say leaf is healthy) is positive, and with the other set is negative, then that weight vector is splitting the feature vector hyper-plane into two.

Or in a better way, which shows the vectors properly



weightvector

In essence, we are using the weight vectors to split the hyper-plane into two distinctive sets.

For any new leaf, if we only extract the same features into a feature vector; we can *dot product* it with the *trained* weight vector and find out if it falls in healthy or deceased class.

Here is a Colab notebook to play around with this.14

Summary

What we have seen so far is that we can represent real world features as vectors residing in some N dimensional space.

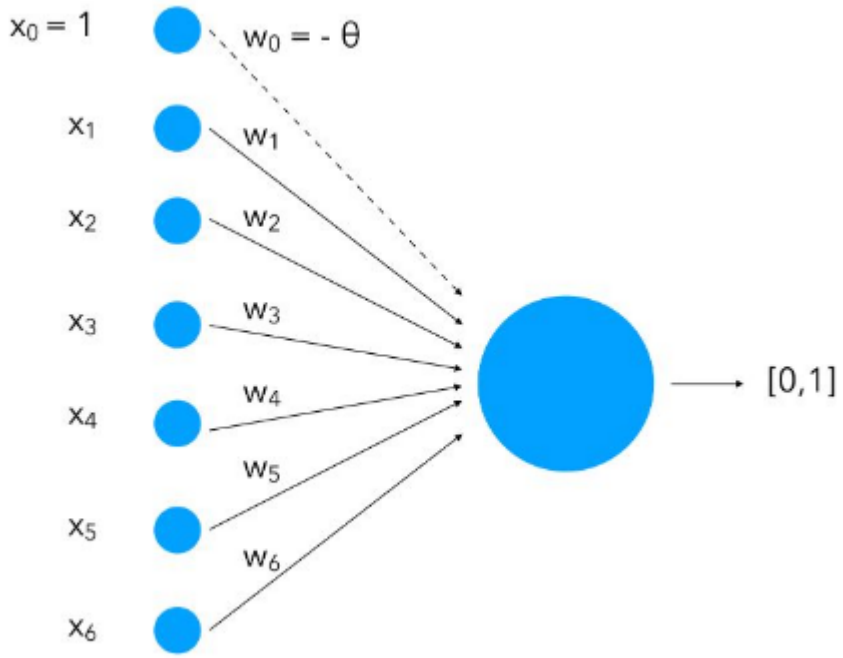
We can then use the concept of hyperplane to split the feature space into two distinctive sets. This is the magic of Representation

Next Perceptron Training

2.2 References

- Vector Space - Wikipedia
- Introduction to Vectors - MathCentre
- Euclidean Vector - Wikipedia
- Dot Product - Paul’s Online Math Notes
- Vector Math - MathOfNeuralNetworks
- Minsky’s And-Or Theorem
- The Perceptron - Mael Fabien
- Rosenblatt’s Perceptron Article
- Colab Notebook
- A Primer on Index Notation - John Crimaldi

3 Perceptron Training via Feature Vectors & HyperPlane split



perceptron

Let’s follow from the previous chapter of the Perceptron neural network.

We have seen how the concept of splitting the hyper-plane of feature set separates one type of feature vectors from other.

3.1 How are the weights learned?

You may have heard about **Gradient Descent**, which is the backbone of training modern neural networks. However, for the classic Perceptron, the learning algorithm is much simpler and relies on a geometric intuition.

The goal is to find a weight vector w that defines a hyperplane separating the two classes of data (e.g., Positive and Negative).

Note this term **hyperplane** is used in the context of feature vector space and is used throughout neural network learning.

3.1.1 The Intuition: Nudging the Vector

Imagine the weight vector w as a pointer. We want this pointer to be oriented such that: 1. It points generally in the same direction as **Positive** examples. 2. It points away from **Negative** examples.

We start with a random weight vector. Then, we iterate through our training data and check how the current w classifies each point.

- **If the classification is correct**: We do nothing. The weight vector is already doing its job for this point.
- **If the classification is wrong**: We need to "nudge" or rotate the weight vector to correct the error.

3.1.2 The Update Rules

Let's say we have an input vector x.

Case 1: False Negative The input x is a **Positive** example ($y = 1$), but our current w classified it as negative (dot product $w \cdot x < 0$). * **Action**: We need to rotate w *towards* x. * **Update**: $w_{new} = w_{old} + x$ * **Result**: Adding

x to w makes the new vector more aligned with x, increasing the dot product for the next time.

Case 2: False Positive The input x is a **Negative** example ($y = 0$ or -1), but our current w classified it as positive (dot product $w \cdot x > 0$). * **Action**: We need to rotate w *away* from x. * **Update**: $w_{new} = w_{old} - x$ * **Result**: Subtracting x from w pushes it in the opposite direction, decreasing the dot product.

3.1.3 The Formal Algorithm

We can combine these rules into a single update equation. We often introduce a **learning rate** η (a small number like 0.1) to make the updates smoother, preventing the weight vector from jumping around too wildly.

For each training example (x, y_{target}): 1. Compute prediction: $\hat{y} = \text{step\_function}(w \cdot x)$ 2. Calculate error: $error = y_{target} - \hat{y}$ 3. Update weights:

$$w = w + \eta \cdot error \cdot x$$

This is known as the **Perceptron Learning Rule**.

$$\Delta w_j = \eta (y_{target} - \text{prediction}) x_j$$

> **Note**: This is distinct from Gradient Descent. Gradient Descent requires a differentiable activation function to compute gradients (slope). The Perceptron uses a “step function” (hard 0 or 1) which is not differentiable. However, this simple rule is guaranteed to converge if the data is linearly separable.

A more rigorous explanation of the proof can be found in the book Neural Networks by R.Rojas or this article.

Next: Gradient Descent and Optimization

3.2 References

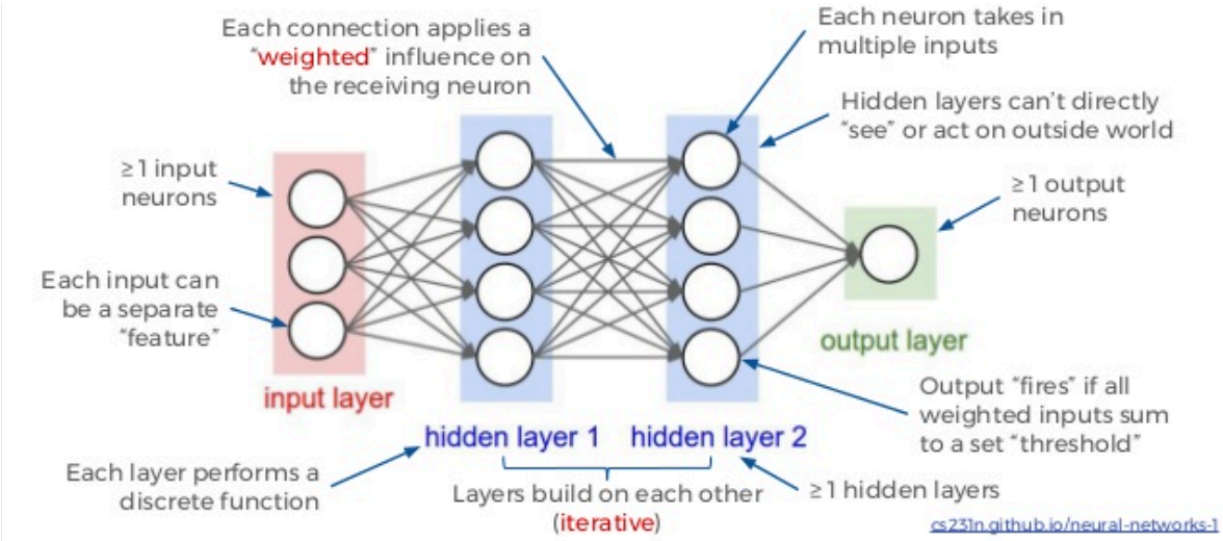
- Neural Networks - R. Rojas (Chapter 4)
- Perceptron Learning Algorithm - Towards Data Science
- Gradient Search - Alvarez

4 Gradient Descent

4.1 Neural Network as a Chain of Functions

To understand deep learning, we need to understand the concept of a neural network as a chain of functions.

A Neural Network is essentially a chain of functions. It consists of a set of inputs connected through ‘weights’ to a set of activation functions, whose output becomes the input for the next layer, and so on.



neuralnetwork

4.1.1 The Forward Pass

Let’s consider a simple two-layer neural network.

- x: Input vector
- y: Output vector (prediction)
- L: Number of layers
- w^l, b^l: Weights and biases for layer l
- a^l: Activation of layer l (we use sigmoid σ here)

The flow of data (Forward Pass) can be represented as:

$$x \to a^1 \to \ldots \to a^L \to y$$

For any layer l, the activation a^l is calculated as:

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

where $a^0 = x$ (the input).

The linear transformation:

$z = w^T x + b$

defines a hyperplane (decision boundary) in the feature space.

The activation function then introduces *non-linearity*, allowing the network to combine **multiple such hyperplanes into complex decision boundaries.**

4.1.1.1 Why Non-Linearity Is Non-Negotiable

Without activation:

$$f\left(x\right) = W_L W_{L-1} \ldots W_1 x$$

This collapses to:

$$f\left(x\right) = Wx$$

Still one big linear transformation and hence one hyperplane; the problems of not able to separate features will come. Only because of non-linearity, we

can get multiple hyperplanes and hence a composable complex decision boundaries that can separate features.

So the concept of Vectors, Matrices and Hyperplanes remain the same as before. Let us explore the chain of functions part here

A neural network with L layers can be represented as a nested function:

$$f\left(x\right) = f_L\left(...f_2\left(f_1\left(x\right)\right)...\right)$$

Each “link” in the chain is a layer performing a linear transformation followed by a non-linear activation and cascading to the final output.

4.1.2 The Cost Function (Loss Function)

To train this network, we need to measure how “wrong” its predictions are compared to the true values. We do this using a **Cost Function** (or Loss Function).

A simplest Loss function is just the difference between the predicted output and the true output ($ y(x) - a^L(x) $.)

But usually we use the square of the difference to make it a non-negative function.

A common choice is the **Mean Squared Error (MSE)**:

$$C = \frac{1}{2n} \sum_x \parallel y\left(x\right) - a^L\left(x\right) \parallel^2$$

- n: Number of training examples
- $y\left(x\right)$: The true expected output (label) for input x
- $a^L\left(x\right)$: The network’s predicted output for input x

4.1.3 The Goal of Training

The goal of training is to find the set of weights w and biases b that minimize this cost C.

This means that we need to optimise each component of the function $f(x)$ to reduce the cost proportional to its contribution to the final output. The method to do this is called **Backpropagation**. It helps us calculate the **gradient** of the cost function with respect to each weight and bias.

Once the gradient is calculated, we can use **Gradient Descent** to update the weights in the opposite direction of the gradient.

Gradient descent is a simple optimization algorithm that works by iteratively updating the weights in the opposite direction of the gradient.

However neural network is a composition of vector spaces and linear transformations. Hence gradient descent acts on a very complex space.

There are two or three facts to understand about gradient descent:

1. It does not attempt to find the **global minimum**, but rather follows the **local slope** of the cost function and converges to a local minimum or a flat region. **Saddle point** is a good optimisation point.

2. Gradients can **vanish or explode**, leading to slow or unstable convergence. The practical solution to control this is to use **learning rate** and using **adaptive learning rate** methods like **Adam** or **RMSprop**.

3. **Batch Size matters**: Calculating the gradient over the entire dataset (Batch Gradient Descent) is computationally expensive and memory-intensive. In practice, we use **Stochastic Gradient Descent (SGD)** (one example at a time) or, more commonly, **Mini-batch Gradient Descent** (a small batch of examples). This introduces noise into the gradient estimate, which paradoxically helps the optimization process escape shallow local minima and saddle points.

4.2 Optimization: Gradient Descent — Take 1

Gradient Descent is a simple yet powerful optimization algorithm used to minimize functions by iteratively updating parameters in the direction that reduces the function's output.

For basic scalar functions (e.g., $\left(f\left(x\right) = x^2\right)$), the update rule is straightforward:

$$x \leftarrow x - \eta \frac{df}{dx}$$

where $\left(\eta\right)$ is the learning rate.

However, **neural networks are not simple scalar functions**. They are **composite vector-valued functions** — layers of transformations that take in high-dimensional input vectors and eventually output either vectors (like logits) or scalars (like loss values).

Understanding how to optimize these complex, high-dimensional functions requires us to extend basic calculus: - The **gradient vector** helps when the function outputs a scalar but takes a vector input (e.g., a loss function w.r.t. weights). - The **Jacobian matrix** becomes important when both the input and the output are vectors (e.g., when computing gradients layer by layer in backpropagation).

We'll build up to this step by step — starting with scalar gradients, then moving to vector calculus, Jacobians, and how backpropagation stitches it all together.

Let's take it one layer at a time.

4.3 Gradient Descent for Scalar Functions

Consider this simple system that composes two functions:

$$L = g\left(f\left(x, w_1\right), w_2\right)$$

Where: - x is your input (fixed, given by your data) - w_1 and w_2 are **parameters you can adjust** (like weights in a neural network) - f is the first function (think: first layer) - g is the second function (think: second layer) - L is the final output

Let’s make this concrete with simple linear functions:

$$f\left(x, w_1\right) = x \cdot w_1 + b_1$$

$$g\left(z, w_2\right) = z \cdot w_2 + b_2$$

So the full composition is:

$$L = g\left(f\left(x, w_1\right), w_2\right) = \left(x \cdot w_1 + b_1\right) \cdot w_2 + b_2$$

4.3.1 Running the Numbers: A Real Example

Let’s pick actual values and see what happens:

Fixed values: - Input: $x = 2.0$ - Bias terms: $b_1 = 1.0$, $b_2 = 0.5$

Current parameter values: - $w_1 = 0.5$ - $w_2 = 1.5$

Step 1: Compute intermediate result from first function:

$$z = f\left(x, w_1\right) = 2.0 \times 0.5 + 1.0 = 2.0$$

Step 2: Compute final output from second function:

$$L = g\left(z, w_2\right) = 2.0 \times 1.5 + 0.5 = 3.5$$

The problem: Suppose we want $L_{\text{target}} = 5.0$ instead!

Our current error is:

$$E = \frac{1}{2}(L - L_{\text{target}})^2 = \frac{1}{2}(3.5 - 5.0)^2 = \frac{1}{2}(-1.5)^2 = 1.125$$

The million-dollar question: How should we change w_1 and w_2 to reduce this error?

4.3.2 The Adjustment Problem: Which Direction? How Much?

Here's what we need to know:

1. **Should we increase or decrease w_1?** (Which direction?)
2. **How sensitive is L to changes in w_1?** (How much?)
3. **Same questions for w_2.**

This is where derivatives come in! Specifically, we need:

$$\frac{\partial L}{\partial w_1} \quad \text{and} \quad \frac{\partial L}{\partial w_2}$$

These tell us:

- **Sign**: Positive means "increase w increases L", negative means the opposite
- **Magnitude**: Larger absolute value means L is more sensitive to changes in w

But there's a complication: w_1 doesn't directly affect L. It affects f, which then affects *g*, which then affects L. This is a **composition**, and we need to trace the effect through multiple steps.

This is where the "Chain Rule" of Calculus comes into play.

4.3.3 The Chain of Effects

Let's visualize how changes propagate:

```
Change w₁ → Affects f → Changes z → Affects g → Changes L
      ↓             ↓          ↓                ↓
      Δw₁         ∂f/∂w₁      Δz      ∂g/∂z      ΔL
```

Similarly for w_2 (but w_2 directly affects g):

```
Change w₂ → Affects g → Changes L
↓ ↓ ↓
Δw₂ ∂g/∂w₂ ΔL
```

The key insight: **To find how w_1 affects L, we need to multiply the effects at each step.**

This is the **chain rule** in action!

4.3.4 The Solution: Applying the Chain Rule

For our composition $L = g\left(f\left(x, w_1\right), w_2\right)$, let's introduce a shorthand: call $z = f\left(x, w_1\right)$ the intermediate value.

Then:

$$L = g\left(z, w_2\right)$$

Computing $\frac{\partial L}{\partial w_1}$:

By the chain rule of calculus:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

Let's compute each piece:

Part 1: How does L change with z?

$$\frac{\partial L}{\partial z} = \frac{\partial}{\partial z}(z \cdot w_2 + b_2) = w_2 = 1.5$$

Part 2: How does z change with w_1?

$$\frac{\partial z}{\partial w_1} = \frac{\partial}{\partial w_1}(x \cdot w_1 + b_1) = x = 2.0$$

Putting it together:

$$\frac{\partial L}{\partial w_1} = 1.5 \times 2.0 = 3.0$$

Interpretation: If we increase w_1 by 0.1, then L increases by approximately $3.0 \times 0.1 = 0.3$.

Computing $\frac{\partial L}{\partial w_2}$:

This is simpler because w_2 directly affects g:

$$\frac{\partial L}{\partial w_2} = \frac{\partial}{\partial w_2}(z \cdot w_2 + b_2) = z = 2.0$$

Interpretation: If we increase w_2 by 0.1, then L increases by approximately $2.0 \times 0.1 = 0.2$.

4.3.5 Making the Update: Gradient Descent

Now we can adjust our parameters! Since we want to **increase** L from 3.5 to 5.0, and both gradients are positive, we should increase both w_1 and w_2.

Using gradient descent with learning rate $\alpha = 0.2$:

$$w_1^{\text{new}} = w_1 + \alpha \cdot \frac{\partial L}{\partial w_1} = 0.5 + 0.2 \times 3.0 = 0.5 + 0.6 = 1.1$$

$$w_2^{\text{new}} = w_2 + \alpha \cdot \frac{\partial L}{\partial w_2} = 1.5 + 0.2 \times 2.0 = 1.5 + 0.4 = 1.9$$

Note: We're adding (not subtracting) because we want to increase L. Normally in machine learning, we minimize error, so we'd use $w - \alpha \cdot \frac{\partial E}{\partial w}$.

4.3.6 Verification: Did It Work?

Let's recompute with the new weights:

Step 1: New intermediate value:

$$z^{\text{new}} = x \cdot w_1^{\text{new}} + b_1 = 2.0 \times 1.1 + 1.0 = 3.2$$

Step 2: New output:

$$L^{\text{new}} = z^{\text{new}} \cdot w_2^{\text{new}} + b_2 = 3.2 \times 1.9 + 0.5 = 6.58$$

Progress check: - Before: $L = 3.5$ (error from target = 1.5) - After: $L = 6.58$ (error from target = -1.58) - We overshot! But that's okay - we moved in the right direction

With a smaller learning rate (say $\alpha = 0.1$), we'd get: - $w_1^{\text{new}} = 0.8$, $w_2^{\text{new}} = 1.7$ - $z^{\text{new}} = 2.6$, $L^{\text{new}} = 4.92$ - Much closer to our target of 5.0!

This is how Gradient Descent works in a nutshell. The same concepts carry over in deep learning with some added complexity.

4.4 Gradient Descent for a Two-Layer Neural Network (Scalar Form)

Let's apply this to a simple neural network with one hidden layer. We have: * **Input**: x * **Hidden Layer**: 1 neuron with weight w_1, bias b_1, activation σ

* **Output Layer**: 1 neuron with weight w_2, bias b_2, activation σ * **Target**: y

Forward Pass: 1. $z_1 = w_1 x + b_1$ 2. $a_1 = \sigma(z_1)$ 3. $z_2 = w_2 a_1 + b_2$ 4. $a_2 = \sigma(z_2)$ (This is our prediction $\hat{y}$)

Loss Function: We use the Mean Squared Error (MSE) for this single example:

$$C = \frac{1}{2}(y - a_2)^2$$

Goal: Find $\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial b_2}$ to update the weights.

Backward Pass (Deriving Gradients):

Layer 2 (Output Layer): We want how C changes with w_2.

$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

- $\frac{\partial C}{\partial a_2} = -(y - a_2)$ (Derivative of $\frac{1}{2}(y-a)^2$)
- $\frac{\partial a_2}{\partial z_2} = \sigma'(z_2)$ (Derivative of activation)
- $\frac{\partial z_2}{\partial w_2} = a_1$

So,

$$\frac{\partial C}{\partial w_2} = -(y - a_2)\sigma'(z_2)a_1$$

Let's define the "error term" for layer 2 as $\delta_2 = -(y - a_2)\sigma'(z_2)$. Then:

$$\frac{\partial C}{\partial w_2} = \delta_2 a_1$$

$$\frac{\partial C}{\partial b_2} = \delta_2 \cdot 1 = \delta_2$$

Layer 1 (Hidden Layer): We want how C changes with w_1. The path is longer: $w_1 \to z_1 \to a_1 \to z_2 \to a_2 \to C$.

$$\frac{\partial C}{\partial w_1} = \underbrace{\frac{\partial C}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2}}_{\delta_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

- We know the first part is δ_2.
- $\frac{\partial z_2}{\partial a_1} = w_2$
- $\frac{\partial a_1}{\partial z_1} = \sigma\prime\left(z_1\right)$
- $\frac{\partial z_1}{\partial w_1} = x$

So,

$$\frac{\partial C}{\partial w_1} = \delta_2 \cdot w_2 \cdot \sigma\prime\left(z_1\right) \cdot x$$

Let's define the error term for layer 1 as $\delta_1 = \delta_2 w_2 \sigma\prime\left(z_1\right)$. Then:

$$\frac{\partial C}{\partial w_1} = \delta_1 x$$

$$\frac{\partial C}{\partial b_1} = \delta_1$$

The Update:

$$w_1 \leftarrow w_1 - \eta\delta_1 x$$

$$w_2 \leftarrow w_2 - \eta\delta_2 a_1$$

This pattern—calculating an error term δ at the output and propagating it back using the weights—is why it's called **Backpropagation**.

Note that we are using here scalar form of gradient descent and not directly applicable to real neural networks. But this gives us the intuition of how backpropagation works.

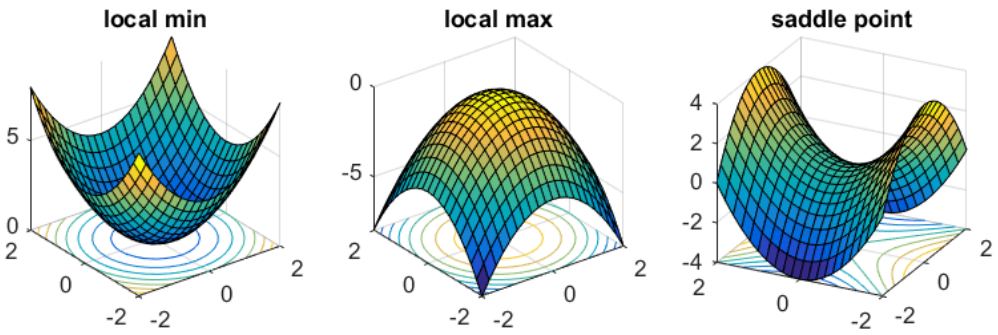
4.5 Some other notes related to Gradient Descent

The Loss/Cost function is a scalar function of the weights and biases.

The loss/error is a scalar function of all weights and biases.

In simpler Machine Learning problems like linear regression with MSE, the loss is a convex quadratic in the parameters, so optimization is well-behaved (a bowl-shaped surface)(e.g. see left in picture).

In deep learning, the loss becomes non-convex because it is the result of composing many nonlinear transformations. This creates a complex landscape with saddle points, flat regions, and multiple minima (e.g. see right in picture).



costfunction

How will Gradient Descent work in this case - non convex function?

Gradient descent does not attempt to find the **global minimum**, but rather follows the local slope of the cost function and converges to a local minimum or a flat region.

The Loss function is differentiable almost everywhere*. At any point in parameter space, the gradient indicates the direction of steepest local increase, and moving in the opposite direction reduces the cost. During optimization, the algorithm may encounter local minima or saddle points.

(*The function is not differentiable at the point where the function is zero ex ReLU. This is not a problem in practice, as optimization algorithms handle such points using subgradients)

In practice, deep learning works well despite non-convexity, partly because modern networks have millions of parameters and their loss landscapes contain many saddle points and wide, flat minima rather than poor isolated local minima.

Also we rarely use full-batch gradient descent. Instead, we use variants such as Stochastic Gradient Descent (SGD) or mini-batch gradient descent that acts as form of sampling.

In these methods, gradients are computed using a single training example or a small batch of examples rather than the entire dataset.

The resulting gradient is an average over the batch and serves as a noisy approximation of the true gradient. This stochasticity helps the optimizer escape saddle points and sharp minima, enabling effective training in practice.

Next: Backpropagation with Scalar Calculus

4.6 References

- Neural Networks and Deep Learning - Michael Nielsen
- Gradient Descent - Wikipedia

5 Backpropagation with Scalar Calculus

In this chapter lets deep dive a bit more into the technique of Back Propagation

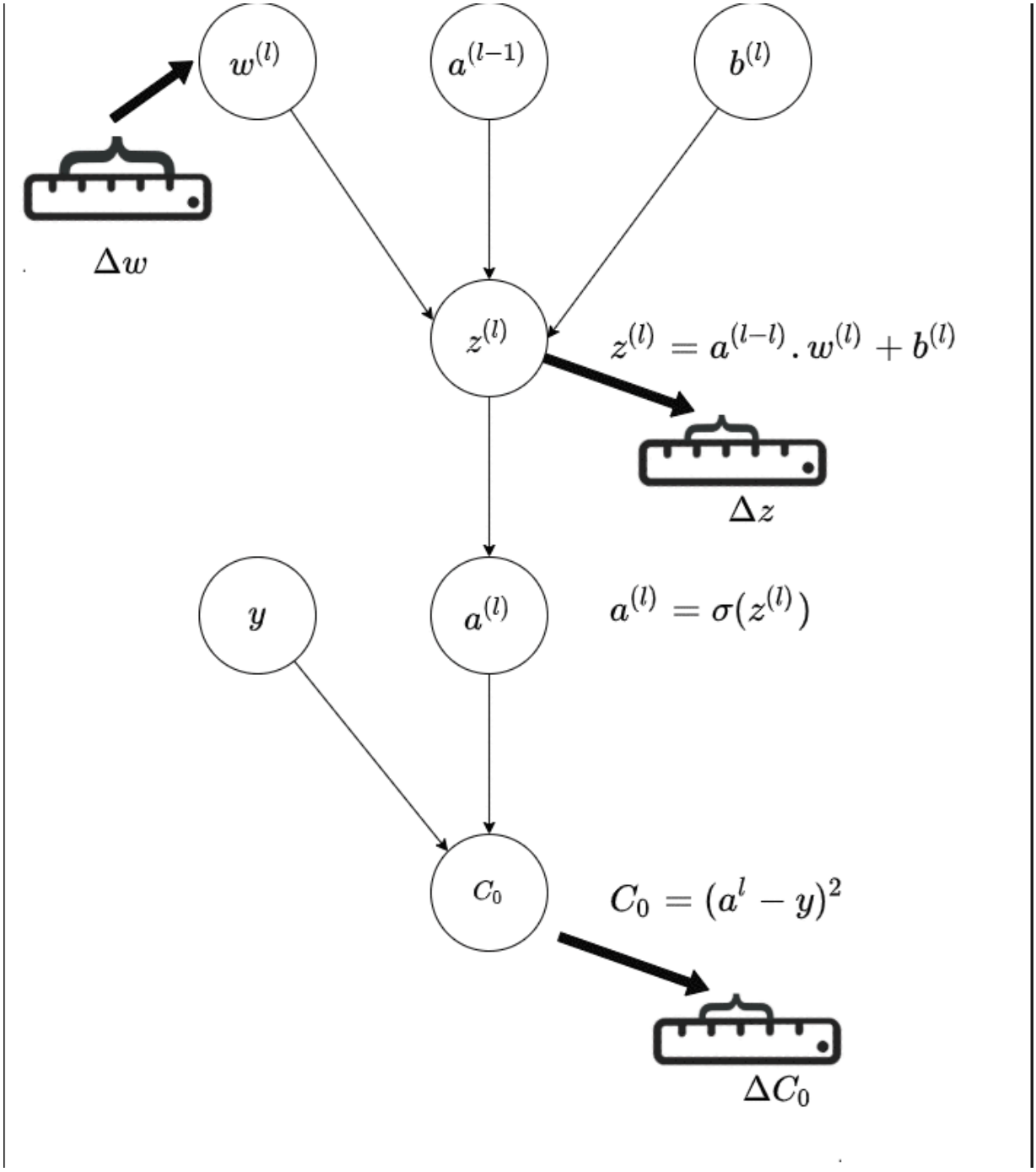
5.1 How Backpropagation Works

Consider a neural network with multiple layers. The weight of layer l is W^l . And for the previous layer it is $W^{(l-1)}$.

The best way to understand backpropagation is visually and by the way it is done by the tree representation of 3Blue1Brown video linked here.

The below GIF is a representation of a single path in the last layer(l of a neural network; and it shows how the connection from previous layer - that is the activation of the previous layer and the weight of the current layer is affecting the output; and thereby the final Cost.

The central idea is how a **small change** in weight in the previous layer affects the final output of the network.



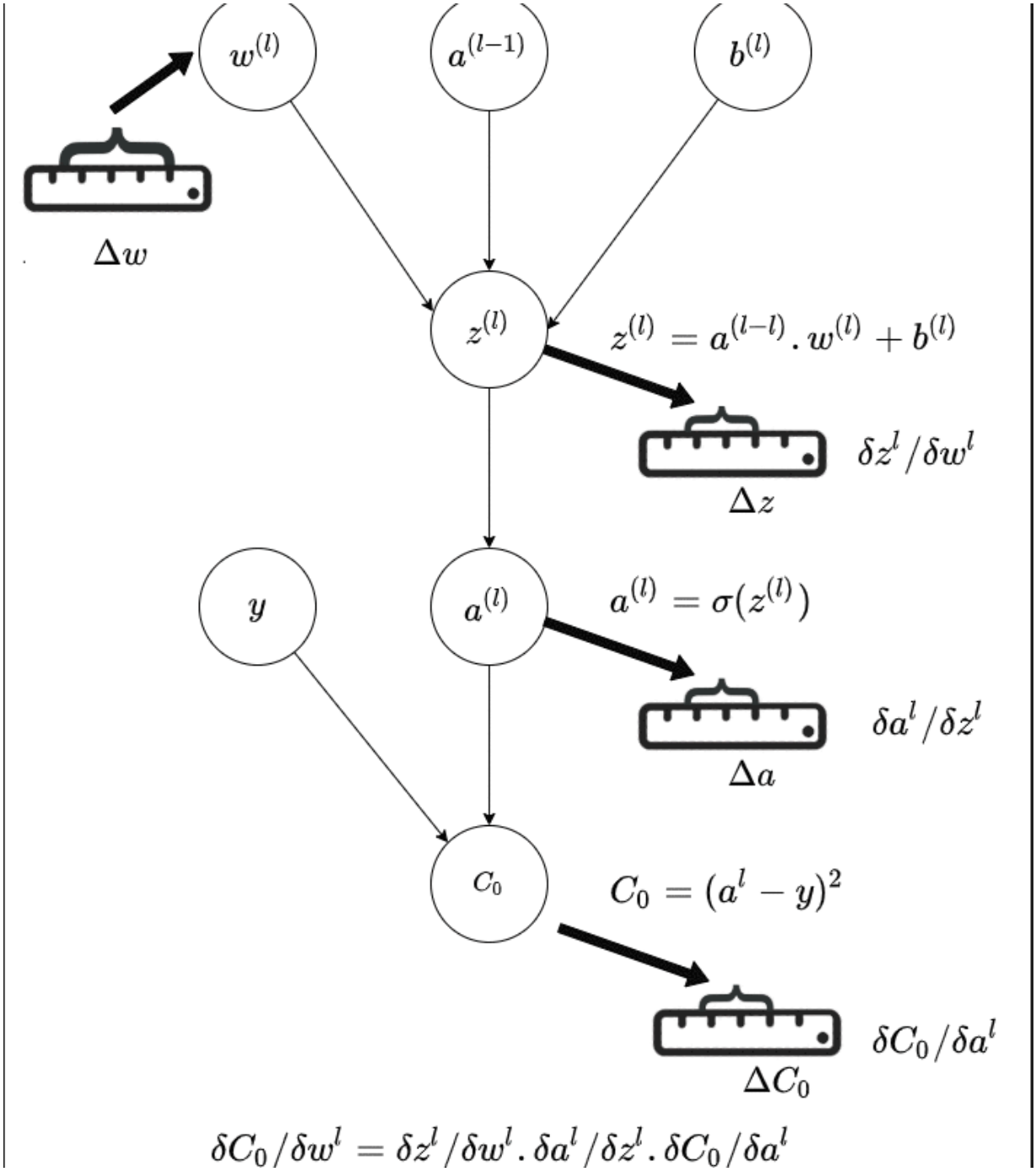
Source : Author

5.2 Writing This Out as Chain Rule

Here is a more detailed depiction of how the small change in weight adds through the chain to affect the final cost, and **how much** the small change of weight in an inner layer affect the final cost.

This is the **Chain Rule** of Calculus and the diagram is trying to illustrate that visually via a chain of activations, via a **Computational Graph**

$$\frac{\partial C_0}{\partial W^l} = \frac{\partial C_0}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial W^l}$$



Source : Author

Next part of the recipe is adjusting the weights of each layers, depending on how they contribute to the Cost. We have already seen this in the previous chapter.

The weights in each layer are adjusted in proportion to how each layers weights affected the Cost function.

This is by calculating the new weight by following the negative of the gradient of the Cost function - basically by gradient descent.

$$W^l_{new} = W^l_{old} - \eta \cdot \frac{\partial C_0}{\partial W^l}$$

For adjusting the weight in the $(l-1)$ layer, we do similar

First calculate how the weight in this layer contributes to the final Cost or Loss

$$\frac{\partial C_0}{\partial W^{l-1}} = \frac{\partial C_0}{\partial a^{l-1}} \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} \cdot \frac{\partial z^{l-1}}{\partial W^{l-1}}$$

and using this. Basically we are using Chain rule to find the partial differential using the partial differentials calculated in earlier steps.

$$W^{l-1}_{new} = W^{l-1}_{old} - \eta \cdot \frac{\partial C_0}{\partial W^{l-1}}$$

5.3 Neural Net as a Composition of Vector Functions

Lets first look at a neural network as a composition of vector functions.

Imagine a simple neural network with 3 layers. It is essentially a composition of three functions:

A neural network is a composition of vector-valued functions, followed by a scalar-valued cost function:

$$ C = (a_3)

a_3 = L_3(L_2(L_1(x))) $$

Where L_1, L_2 and L_3 are the three layers of the network and

Each layer is defined as:

$$z_i = W_i a_{i-1} + b_i, \quad a_i = \sigma(z_i)$$

And gradient descent is defined as:

$$W_{i_{new}} = W_{i_{old}} - \eta \cdot \partial C / \partial W_i$$

Problem is to find the partial derivative of the loss function with respect to the weights at each layer.

To calculate how a change in the first layer's weights (W_1) affects the final Cost (C), we have to trace the "path of influence" all the way through the network.

A nudge in W_1 changes the output of Layer 1. The change in Layer 1 changes the input to Layer 2. The change in Layer 2 changes the input to Layer 3. The change in Layer 3 changes the final Cost.

Mathematically, we multiply the derivatives (Linear Maps) of these links together:

We need to update weights of three layers

$$W_{1_{new}} = W_{1_{old}} - \eta \cdot \partial C / \partial W_1$$

$$W_{2_{new}} = W_{2_{old}} - \eta \cdot \partial C / \partial W_2$$

$$W_{3_{new}} = W_{3_{old}} - \eta \cdot \partial C / \partial W_3$$

And for that we need to find $ C / W_1 $, $ C / W_2 $, $ C / W_3 $.

Lets write down the chain rule for each layer:

$$\frac{\partial C}{\partial W_1} = \frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2} \cdot \frac{\partial L_2}{\partial L_1} \cdot \frac{\partial L_1}{\partial W_1}$$

$$\frac{\partial C}{\partial W_2} = \frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2} \cdot \frac{\partial L_2}{\partial W_2}$$

$$\frac{\partial C}{\partial W_3} = \frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial W_3}$$

Why is this written this way? By the **chain rule**, **the derivative of a composition of functions is the product of the derivatives of the functions**. It is thus easy to calculate the gradient of the loss with respect to the weights of each layer.

Lets calculate the gradient of the loss with respect to the weights of the first layer.

Notice something interesting?

- To calculate $\frac{\partial C}{\partial W_3}$, we need $\frac{\partial C}{\partial L_3}$.
- To calculate $\frac{\partial C}{\partial W_2}$, we need $\frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2}$.
- To calculate $\frac{\partial C}{\partial W_1}$, we need $\frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2} \cdot \frac{\partial L_2}{\partial L_1}$.

We are re-calculating the same terms over and over again!

If we start from the **Output** (Layer 3) and move **Backwards**: 1. We calculate $\frac{\partial C}{\partial L_3}$ once. We use it to find the update for W_3.

2. We pass this value back to find $\frac{\partial C}{\partial L_2}$ (which is $\frac{\partial C}{\partial L_3} \cdot \frac{\partial L_3}{\partial L_2}$). We use it to find the update for W_2.
3. We pass *that* value back to find $\frac{\partial C}{\partial L_1}$. We use it to find the update for W_1.

This avoids redundant calculations and is why it's called **Backpropagation**.

It is essentially **Dynamic Programming** applied to the Chain Rule.

5.3.1 The Backpropagation Algorithm Step-by-Step

Step 1: The Output Layer (L_3)

We want to find the gradient $\frac{\partial C}{\partial W_3}$. Using the Chain Rule:

$$\frac{\partial C}{\partial W_3} = \frac{\partial C}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial W_3}$$

Let's break it down term by term:

1. **Derivative of Cost w.r.t Activation** ($\frac{\partial C}{\partial a_3}$): For MSE $C = \frac{1}{2}(a_3 - y)^2$:

$$\frac{\partial C}{\partial a_3} = (a_3 - y)$$

2. **Derivative of Activation w.r.t Input** ($\frac{\partial a_3}{\partial z_3}$): Since $a_3 = \sigma(z_3)$:

$$\frac{\partial a_3}{\partial z_3} = \sigma'(z_3)$$

3. **Derivative of Input w.r.t Weights** ($\frac{\partial z_3}{\partial W_3}$): Since $z_3 = W_3 a_2 + b_3$:

$$\frac{\partial z_3}{\partial W_3} = a_2$$

Combining them: We define the "error" term δ_3 at the output layer as:

$$\delta_3 = \frac{\partial C}{\partial z_3} = (a_3 - y) \odot \sigma'(z_3)$$

Note on $\odot$ (Hadamard Product): We use element-wise multiplication here because both $(a_3 - y)$ and $\sigma\prime\left(z_3\right)$ are vectors of the same size.

The Jacobian of an element-wise activation σ is a diagonal matrix:

$$\frac{\partial a}{\partial z} = \operatorname{diag}\left(\sigma\prime\left(z\right)\right)$$

So multiplying by it is the same as a Hadamard product:

$$\operatorname{diag}\left(\sigma\prime\left(z\right)\right) v = v \odot \sigma\prime\left(z\right)$$

We will see the Jacobian and Gradient Vector later.

So the gradient for the weights is:

$$\frac{\partial C}{\partial W_3} = \delta_3 \cdot a_2^T$$

Note on Transpose (a_2^T): In backprop, we push gradients through a linear map $z = Wa + b$. The Jacobian w.r.t. a is W, so the chain rule gives:

$$\frac{\partial C}{\partial a} = W^T \frac{\partial C}{\partial z}$$

The transpose appears because we're applying the transpose (adjoint) of the Jacobian to move gradients backward.

Result: We have the update for W_3.

$$W_{3_{new}} = W_{3_{old}} - \eta \cdot \partial C \,/\, \partial W_3$$

Step 2: Propagate Back to L_2

Now we need to find the gradient for the second layer weights: $\frac{\partial C}{\partial W_2}$. Using the Chain Rule, we can reuse the error from the layer above:

$$\frac{\partial C}{\partial W_2} = \frac{\partial C}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = \delta_2 \cdot a_1^T$$

But what is δ_2 (the error at layer 2)?

$$\delta_2 = \frac{\partial C}{\partial z_2} = \frac{\partial C}{\partial z_3} \cdot \frac{\partial z_3}{\partial z_2}$$

We know $\frac{\partial C}{\partial z_3} = \delta_3$. And since $z_3 = W_3\sigma\left(z_2\right) + b_3$:

$$\frac{\partial z_3}{\partial z_2} = W_3 \cdot \sigma\prime\left(z_2\right)$$

So, we can calculate δ_2 by "backpropagating" δ_3:

$$\delta_2 = \left(W_3^T \cdot \delta_3\right) \odot \sigma\prime\left(z_2\right)$$

The Update Rule for Layer 2:

$$\frac{\partial C}{\partial W_2} = \delta_2 \cdot a_1^T$$

Result: We have the update for W_2.

$$W_{2_{new}} = W_{2_{old}} - \eta \cdot \frac{\partial C}{\partial W_2}$$

Step 3: Propagate Back to L_1

We repeat the exact same process to find the error at the first layer δ_1.

$$\delta_1 = \left(W_2^T \cdot \delta_2\right) \odot \sigma\prime\left(z_1\right)$$

The Update Rule for Layer 1:

$$\frac{\partial C}{\partial W_1} = \delta_1 \cdot x^T$$

(Recall that $a_0 = x$, the input).

Result: We have the update for W_1.

$$W_{1_{new}} = W_{1_{old}} - \eta \cdot \frac{\partial C}{\partial W_1}$$

5.3.2 Summary

So, Backpropagation is the efficient execution of the Chain Rule by utilizing the linear maps of each layer in reverse order. * It computes the local linear map (Jacobian) of a layer. * It takes the incoming gradient vector from the future layer. * It performs a Vector-Jacobian Product to pass the gradient to the past layer.

Next Backpropagation with Matrix Calculus

5.4 References

- Neural Networks and Deep Learning - Michael Nielsen
- A Step by Step Backpropagation Example - Matt Mazur

6 Backpropagation with Matrix Calculus

The previous chapters we used a Scalar derivation of the Back Propagation formula to implement it in a simple two layer neural network. What we have done is is to use Hadamard product and matrix transposes with scalar derivation alignment.

But we have not really explained why we use Hadamard product and matrix transposes with scalar derivation alignment.

This is due to Matrix Calculus which is the real way in which we should be deriving the Back Propagation formula.

Lets explore this in this chapter. Note that we are still not using a Softmax activation function in the output layer as is usually the case with Deep Neural Networks. Deriving the Back Propagation formula with Softmax activation function is bit more complex and we will do that in a later chapter.

Let’s take the previous two layered simple neural network,with a Mean Square Error Loss function, and derive the Back Propagation formula with Matrix Calculus now.

Let’s write the equation of the following neural network

```
x is the Input
y is the Output.
```

```
l is the number of layers of the Neural Network.
a is the activation function ,(we use sigmoid here)
```

$$x \to a^{l-1} \to a^{l} \to y$$

Where the activation a^l is

$$a^l = \sigma\left(W^l a^{l-1} + b^l\right).$$

and

$$a^l = \sigma\left(z^l\right) \quad where \quad z^l = W^l a^{l-1} + b^l$$

Our two layer neural network can be written as

$$\mathbf{a^0} \to \mathbf{a^1} \to \mathbf{a^2} \to \mathbf{y}$$

(a^2 does not denote the exponent but just that it is of layer 2)

6.1 Some Math Intuition

The concept of a Neural Network as a composition of functions remains central.

In our network, most layers represent functions that map a **Vector to a Vector** ($\mathbb{R}^n \to \mathbb{R}^m$). For example, the hidden layers take an input vector and produce an activation vector.

However, the final step—calculating the Loss—is different. It maps the final output vector a^L (and the target y) to a single **Scalar** value, the Cost C ($\mathbb{R}^n \to \mathbb{R}$).

6.1.1 Gradient Vector

When we take the derivative of a scalar-valued function (like the Cost C) with respect to a vector (like the weights w), the result is a vector of the same size as w. This is called the **Gradient Vector**.

$$\nabla_w C = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

Why is it called a “gradient”?

Because the gradient points in the direction of steepest increase of the function.

Moving a tiny step along $+\nabla_w C$ increases the cost the fastest.

Moving a tiny step against it, i.e. along $-\nabla_w C$, decreases the cost the fastest.

That’s exactly why gradient descent updates parameters like this:

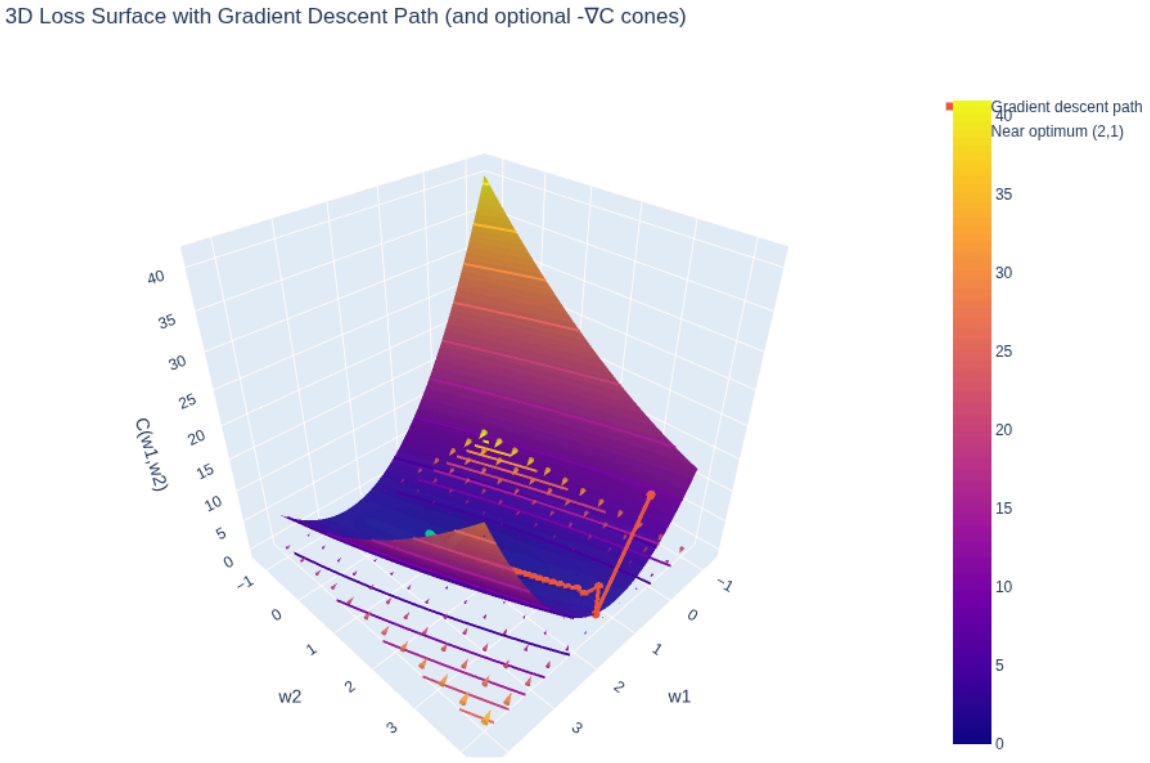
$$w \leftarrow w - \eta \nabla_w C$$

where η is the learning rate.

So the gradient vector is more than a list of derivatives—it’s the local direction that tells us how to change parameters to reduce the loss.

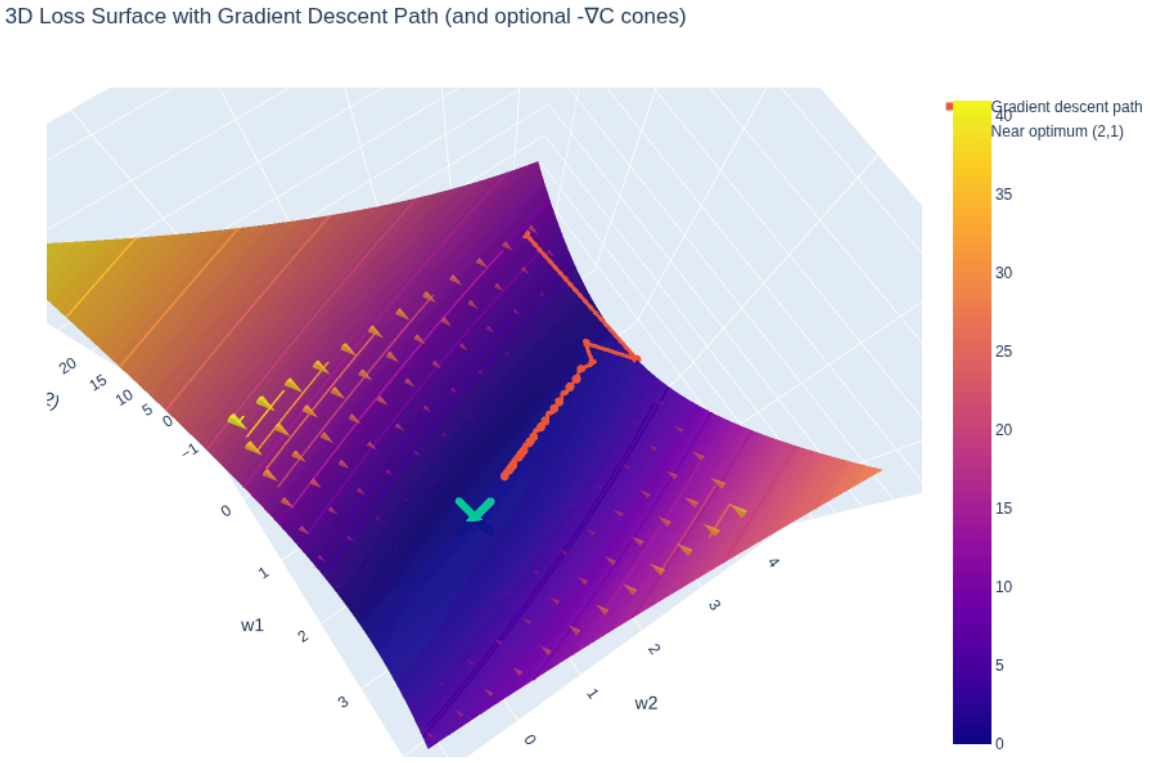
See the Colab 7_1 for a generated visualization of this.

The first image is the plotting of the Cost function



gradient_vector

The second image where you see the cones are the gradient vector of the Cost function wrto weights plotted in 3D space.



gradient_vector

6.2 Jacobian Matrix

The second key concept is the **Jacobian Matrix**.

As mentioned earlier, in our network, most layers represent functions that map a **Vector to a Vector** ($\mathbb{R}^n \to \mathbb{R}^m$). For example, a hidden layer takes an input vector x and produces an activation vector a.

What is the derivative of a vector-valued function with respect to a vector input? This is where the Jacobian comes in.

For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ that maps an input vector x of size n to an output vector y of size m, the derivative is an $m \times n$ matrix called the Jacobian Matrix J.

The entry J_{ij} is the partial derivative of the i-th output component with respect to the j-th input component:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

6.2.1 The Chain Rule with Matrices

The beauty of the Jacobian is that it allows us to generalize the chain rule.

For scalar functions, the chain rule is just multiplication:
$(f \circ g)\prime(x) = f\prime(g(x)) \cdot g\prime(x)$.

For vector functions, the chain rule becomes **Matrix Multiplication** of the Jacobians:

If we have a composition of functions $y = f(g(x))$, and we let A be the Jacobian of f and B be the Jacobian of g, then the Jacobian of the composition is simply the matrix product $A \cdot B$.

$$(A \cdot B)_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$$

So, the Jacobian is a matrix of partial derivatives that represents the local linear approximation of a vector function. When we say the Jacobian represents a “local linear approximation,” we mean:

$$\text{Change in Output} \approx \text{Jacobian Matrix} \cdot \text{Change in Input}$$

$$\Delta y \approx J \cdot \Delta x$$

It tells us: “If I nudge the input vector by a tiny vector Δx, the output vector will change by roughly the matrix-vector product $J \cdot \Delta x$.”

6.2.2 Backpropagation Trick - VJP (Vector Jacobian Product) and JVP (Jacobian Vector Product)

There is one more trick that we can use to make backpropogation more efficient.

Let me explain with an example.

Suppose we have a chain of functions: $y = f\left(g\left(h\left(x\right)\right)\right)$. To find the derivative $\frac{\partial y}{\partial x}$, the chain rule tells us to multiply the Jacobians:

$$J_{total} = J_f \cdot J_g \cdot J_h$$

If x, h, g, f are all vectors of size 1000, then each Jacobian is a 1000×1000 matrix. Multiplying them is expensive ($O\left(N^3\right)$).

However, in Backpropagation, we always start with a scalar Loss function. The final derivative $\frac{\partial C}{\partial y}$ is a row vector (size $1 \times N$).

So we are computing:

$$\nabla C = \underbrace{\frac{\partial C}{\partial y}}_{1\times N} \cdot \underbrace{J_f}_{N\times N} \cdot \underbrace{J_g}_{N\times N} \cdot \underbrace{J_h}_{N\times N}$$

Notice the order of operations matters! 1. **Jacobian-Matrix Product**: If we multiply the matrices first ($J_f \cdot J_g$), we do expensive matrix-matrix multiplication. 2. **Vector-Jacobian Product (VJP)**: If we multiply from left to right: * $v_1 = \frac{\partial C}{\partial y} \cdot J_f$ (Vector $\times$ Matrix $\rightarrow$ Vector) * $v_2 = v_1 \cdot J_g$ (Vector $\times$ Matrix $\rightarrow$ Vector) * $v_3 = v_2 \cdot J_h$ (Vector $\times$ Matrix $\rightarrow$ Vector)

We **never** explicitly compute or store the full Jacobian matrix. We only compute the product of a vector with the Jacobian. This is much faster ($O\left(N^2\right)$) and uses less memory.

This is the secret sauce of efficient Backpropagation!

6.2.3 Hadamard Product

Another important operation we use is the **Hadamard Product** (denoted by $\odot$ or sometimes $\circ$). This is simply **element-wise multiplication** of two vectors or matrices of the same size.

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 \\ a_2 \cdot b_2 \end{bmatrix}$$

It is different from the dot product (which sums the results to a scalar) and matrix multiplication. In backpropagation, it often appears when we apply the chain rule through an activation function that operates element-wise (like sigmoid or ReLU).

6.3 Backpropagation Derivation

6.3.1 The 2-Layer Neural Network Model

For this derivation, we use a simple 2-layer network (one hidden layer, one output layer):

$$x \xrightarrow{W^1, b^1} a^1 \xrightarrow{W^2, b^2} a^2$$

Forward Pass Equations: 1. **Hidden Layer:**

$$z^1 = W^1 x + b^1$$

$$a^1 = \sigma\left(z^1\right)$$

2. **Output Layer:**

$$z^2 = W^2 a^1 + b^2$$

$$a^2 = \sigma\left(z^2\right)$$

We use the **Mean Squared Error (MSE)** loss function:

$$C = \frac{1}{2} \parallel y - a^2 \parallel^2$$

6.3.2 Gradient Vector/2D-Tensor of Loss Function in Last Layer

$$C = \frac{1}{2} \parallel y - a^2 \parallel^2 = \frac{1}{2} \sum_j \left(y_j - a_j^2\right)^2$$

Where:

$$a^2 = \sigma\left(z^2\right) \quad \text{and} \quad z^2 = W^2 a^1 + b^2$$

We want to find $\frac{\partial C}{\partial W^2}$. Using the Chain Rule:

$$\frac{\partial C}{\partial W^2} = \frac{\partial C}{\partial z^2} \cdot \frac{\partial z^2}{\partial W^2}$$

Let's define the **error term** δ^2 as the derivative of the cost with respect to the pre-activation z^2:

$$\delta^2 \equiv \frac{\partial C}{\partial z^2} = \frac{\partial C}{\partial a^2} \odot \frac{\partial a^2}{\partial z^2}$$

1. $\frac{\partial C}{\partial a^2} = \left(a^2 - y\right)$
2. $\frac{\partial a^2}{\partial z^2} = \sigma\prime\left(z^2\right)$

So, using the Hadamard product ($\odot$) for element-wise multiplication:

Note that none of these terms are exponents but super scripts.!

Hadamard product or Element-wise multiplication

The confusion usually lies in this term:

$$\frac{\partial a}{\partial z}$$

Since a is a vector and z is a vector, the derivative of one with respect to the other is technically a Jacobian Matrix, not a vector.However, because the activation function σ is applied element-wise (i.e., a_i depends only on z_i, not on z_j - That is - activation function in one layer is just dependent of the output of only the previous layer and no other layers), all off-diagonal elements of this Jacobian are zero.

$$J = \frac{\partial a}{\partial z} = \begin{bmatrix} \sigma\prime\left(z_1\right) & 0 & \dots \\ 0 & \sigma\prime\left(z_2\right) & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

When you apply the chain rule, you are multiplying the gradient vector $\nabla_a C$ by this diagonal matrix J - VJP (Vector-Jacobian Product).

Key Identity: Multiplying a vector by a diagonal matrix is mathematically identical to taking the Hadamard product of the vector and the diagonal elements and this is why we use Hadamard product in backpropogation.

$$\delta^2 = \left(a^2 - y\right) \odot \sigma\prime\left(z^2\right)$$

Now for the second part, we need to find how the Cost changes with respect to the weights W^2.

We know that $z^2 = W^2 a^1$. In index notation, for a single element z_i^2:

$$z_i^2 = \sum_k W_{ik}^2 a_k^1$$

We want to find $\frac{\partial C}{\partial W_{ik}^2}$. Using the chain rule:

$$\frac{\partial C}{\partial W_{ik}^2} = \frac{\partial C}{\partial z_i^2} \cdot \frac{\partial z_i^2}{\partial W_{ik}^2}$$

1. We already defined $\frac{\partial C}{\partial z_i^2} = \delta_i^2$.
2. From the linear equation $z_i^2 = \ldots + W_{ik}^2 a_k^1 + \ldots$, the derivative with respect to W_{ik}^2 is simply a_k^1.

So:

$$\frac{\partial C}{\partial W_{ik}^2} = \delta_i^2 \cdot a_k^1$$

If we organize these gradients into a matrix, the element at row i and column k is the product of the i-th element of δ^2 and the k-th element of a^1.

Let's visualize the matrix of gradients ∇W:

$$\nabla W = \begin{bmatrix} \frac{\partial C}{\partial W_{11}} & \frac{\partial C}{\partial W_{12}} \\ \frac{\partial C}{\partial W_{21}} & \frac{\partial C}{\partial W_{22}} \end{bmatrix}$$

Substitute the result from step 3 ($\delta_i \cdot a_k$):

$$\nabla W = \begin{bmatrix} \delta_1 a_1 & \delta_1 a_2 \\ \delta_2 a_1 & \delta_2 a_2 \end{bmatrix}$$

This is exactly the definition of the **Outer Product** $\otimes$ of two vectors:

$$\frac{\partial C}{\partial W^2} = \delta^2 \otimes a^1 = \delta^2 \left(a^1\right)^T \quad \rightarrow (Eq\ 3)$$

This gives us the gradient matrix for the last layer weights.

6.4 Jacobian of Loss Function in Inner Layer

Now let's do the same for the inner layer (W^1).

$$\frac{\partial C}{\partial W^1} = \frac{\partial C}{\partial z^1} \cdot \frac{\partial z^1}{\partial W^1} = \delta^1 \left(a^0\right)^T$$

We need to find $\delta^1 = \frac{\partial C}{\partial z^1}$. We can backpropagate the error δ^2 from the next layer:

$$\delta^1 = \frac{\partial C}{\partial z^1} = \left(\left(W^2\right)^T \delta^2\right) \odot \sigma\prime\left(z^1\right)$$

Explanation: 1. We propagate δ^2 backwards through the weights $\left(W^2\right)^T$.
2. We multiply element-wise by the derivative of the activation function $\sigma\prime\left(z^1\right)$.

Putting it all together:

$$\frac{\partial C}{\partial W^1} = \left(\left(W^2\right)^T \delta^2 \odot \sigma\prime\left(z^1\right)\right)\left(a^0\right)^T \quad \rightarrow (Eq\ 5)$$

6.4.1 Summary of Backpropagation Equations

1. **Compute Output Error:**

$$\delta^L = \left(a^L - y\right) \odot \sigma\prime\left(z^L\right)$$

2. **Backpropagate Error:**

$$\delta^l = \left(\left(W^{l+1} \right)^T \delta^{l+1} \right) \odot \sigma\prime \left(z^l \right)$$

3. **Compute Gradients:**

$$\frac{\partial C}{\partial W^l} = \delta^l \left(a^{l-1} \right)^T$$

6.4.2 Summary of Backpropagation Equations in Terms of Numpy

Here is how these equations translate to Python code using NumPy, assuming standard column vectors (shape `(N, 1)`).

```
# Forward pass context:
# x, a1, a2 are column vectors
# W1, W2 are weight matrices
# sigmoid_prime(z) is the derivative of activation

# 1. Compute Output Error (Hadamard Product)
# '*' operator in numpy is element-wise multiplication (Hadamard)
delta2 = (a2 - y) * sigmoid_prime(z2)

# 2. Gradient for W2 (Outer Product)
# We need shape (n_out, 1) @ (1, n_hidden) -> (n_out, n_hidden)
dC_dW2 = np.matmul(delta2, a1.T)

# Alternative using einsum for outer product:
# dC_dW2 = np.einsum('i,j->ij', delta2.flatten(), a1.flatten())

# 3. Backpropagate Error to Hidden Layer
# Matrix multiplication (W2.T @ delta2) followed by Hadamard
product
delta1 = np.matmul(W2.T, delta2) * sigmoid_prime(z1)

# 4. Gradient for W1 (Outer Product)
dC_dW1 = np.matmul(delta1, x.T)
```

6.5 Using Gradient Descent to Find the Optimal Weights to Reduce the Loss Function

With equations (3) and (5) we can calculate the gradient of the Loss function with respect to weights in any layer - in this example

$$\frac{\partial C}{\partial W^1}, \frac{\partial C}{\partial W^2}$$

We now need to adjust the previous weight, by gradient descent.

So using the above gradients we get the new weights iteratively like below. If you notice this is exactly what is happening in gradient descent as well; only chain rule is used to calculate the gradients here. Backpropagation is the algorithm that helps calculate the gradients for each layer.

$$\mathbf{W_{new}^{l-1} = W_{old}^{l-1} - \eta \cdot \frac{\partial C}{\partial W^{l-1}}}$$

Where η is the learning rate.

Next: Backpropagation with Softmax and Cross Entropy

6.6 References

- Back Propagation - Srihari (SUNY Buffalo)
- Neural Networks and Deep Learning - Michael Nielsen
- Colab Visualization

7 Backpropagation with Softmax and Cross Entropy

Let's think of a l layered neural network whose input is $x = a^0$ and output is a^l.In this network we will be using the **sigmoid (σ)** function as the activation function for all layers except the last layer l. For the last layer we use the **Softmax activation function**. We will use the **Cross Entropy Loss** as the loss function.

This is how a proper Neural Network should be.

7.1 The Neural Network Model

I am writing this out, without index notation, and with the super script representing just the layers of the network.

$$\mathbf{a^0} \rightarrow \mathbf{hidden\ layers} \underset{\mathbf{a^{l-2}}}{} \rightarrow \mathbf{W^{l-1}a^{l-2}+b^{l-1}} \underset{\mathbf{z^{l-1}}}{} \rightarrow \boldsymbol{\sigma}\left(\mathbf{z^{l-1}}\right) \underset{\mathbf{a^{l-1}}}{} \rightarrow \mathbf{W^{l}a^{l-1}+b^{l}} \underset{\mathbf{z^{l}/logits}}{} \rightarrow \mathbf{P}\left(\mathbf{z^{l}}\right) \underset{\mathbf{\vec{P}/softmax/a^{l}}}{} \rightarrow \mathbf{L}\left(\mathbf{\vec{P}, \vec{Y}}\right) \underset{\mathbf{CrossEntropy Loss}}{}$$

\begin{aligned}

a^0 \rightarrow

\underbrace{\text{hidden layers}}_{a^{l-2}}

\,\rightarrow

\underbrace{W^{l-1} a^{l-2}+b^{l-1}}_{z^{l-1} }

\,\rightarrow

\underbrace{\sigma(z^{l-1})}_{a^{l-1}}

\,\rightarrow

\underbrace{W^l a^{l-1}+b^l}_{z^{l}/logits }

\,\rightarrow

\underbrace{P(z^l)}_{\vec P/ \text{softmax} /a^{l}}

\,\rightarrow

\underbrace{L (\vec P, \vec Y)}_{\text{CrossEntropyLoss}}

\end{aligned}

}

Y is the target vector or the Truth vector. This is a one hot encoded vector, example $Y = [0, 1, 0]$, here the second element is the desired class.The training is done so that the CrossEntropyLoss is minimized using Gradient

Descent algorithm.

P is the Softmax output and is the activation of the last layer a^l. This is a vector. All elements of the Softmax output add to 1; hence this is a probability distribution unlike a Sigmoid output.The Cross Entropy Loss L is a Scalar.

Note the Index notation is the representation an element of a Vector or a Tensor, and is easier to deal with while deriving out the equations.

Softmax (in Index notation) Below I am skipping the superscript part, which I used to represent the layers of the network.

$$p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}

\end{aligned}

This represent one element of the softmax vector, example $\overrightarrow{P} = [p_1, p_2, p_3]$

Cross Entropy Loss (in Index notation) Here y_i is the indexed notation of an element in the target vector Y.

$$L = -\sum_j y_j \log p_j$$

L = -\sum_j y_j \log p_j

\end{aligned}

There are too many articles related to Back propagation, many of which are very good.However many explain in terms of index notation and though it is illuminating, to really use this with code, you need to understand how it translates to Matrix notation via Matrix Calculus and with help form StackOverflow related sites.

7.1.1 CrossEntropy Loss with Respect to Weights in Last Layer

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{\mathbf{l}}} = \frac{\partial \mathbf{L}}{\partial \mathbf{z}^{\mathbf{l}}} \cdot \frac{\partial \mathbf{z}^{\mathbf{l}}}{\partial \mathbf{W}^{\mathbf{l}}} \rightarrow \quad \mathbf{EqA1}$$

\frac {\partial L}{\partial W^l}

= \color{red}{\frac {\partial L}{\partial z^l}} \cdot \color{green}{\frac {\partial z^l}{\partial W^l}} \rightarrow \quad EqA1

}

Where

$$L = -\sum_k y_k \log p_k \quad \text{and} \quad p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

If you are confused with the indexes, just take a short example and substitute. Basically i,j,k etc are dummy indices used to illustrate in index notation the vectors.

I am going to drop the superscript l denoting the layer number henceforth and focus on the index notation for the softmax vector P and target vector Y

From Derivative of Softmax Activation -Alijah Ahmed

$$ {

$$\frac{\partial L}{\partial z_i} = \frac{\partial(-\sum_k y_k \log p_k)}{\partial z_i}$$

$$\text{taking the summation outside}$$

$$= -\sum_k y_k \frac{\partial(\log p_k)}{\partial z_i}$$

$$\text{since } \frac{d}{dx}(f(g(x))) = f\prime(g(x))g\prime(x)$$

$$= -\sum_k y_k \cdot \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

\frac {\partial L}{\partial z_i} = \frac {\partial ({-\sum_k y_k \log {p_k})}}{\partial z_i}

\ \ \text {taking the summation outside} \ \ = -\sum_k y_k\frac {\partial ({ \log {p_k})}}{\partial z_i}

\ \ \color{grey}{\text {since }

\frac{d}{dx} (f(g(x))) = f'(g(x))g'(x) }

\ \

= -\sum_k y_k \cdot \frac {1}{p_k} \cdot \frac {\partial { p_k}}{\partial z_i}

\end{aligned}

} $$

The last term $\frac{\partial p_k}{\partial z_i}$ is the derivative of Softmax with respect to its inputs also called logits. This is easy to derive and there are many sites that describe it. Example [Derivative of SoftMax Antoni Parellada]. The more rigorous derivative via the Jacobian matrix is here The Softmax function and its derivative-Eli Bendersky

$$\frac{\partial p_i}{\partial z_i} = p_i(\delta_{ij} - p_j)$$

$$\delta_{ij} = 1 \text{ when i =j}$$
$$\delta_{ij} = 0 \text{ when i} \neq \text{j}$$

{

\begin{aligned}

\frac {\partial { p_i}}{\partial z_i} = p_i(\delta_{ij} -p_j) \ \

\delta_{ij} = 1 \text{ when i =j}

\

\delta_{ij} = 0 \text{ when i} \ne \text{j}

\end{aligned}

}

Using this above and from Derivative of Softmax Activation -Alijah Ahmed

$$ {

$$\frac{\partial L}{\partial z_i} = -\sum_k y_k \cdot \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

$$= -\sum_k y_k \cdot \frac{1}{p_k} \cdot p_i \left(\delta_{ij} - p_j\right)$$

$$\text{these i and j are dummy indices and we can rewrite this as}$$

$$= -\sum_k y_k \cdot \frac{1}{p_k} \cdot p_k \left(\delta_{ik} - p_i\right)$$

$$\text{taking the two cases and adding in above equation}$$

$$\delta_{ik} = 1 \text{ when i =k and } \delta_{ik} = 0 \text{ when i} \neq \text{k}$$

$$= \left[-y_i \cdot \frac{1}{p_i} \cdot p_i \left(1 - p_i\right)\right] + \left[-\sum_{k \neq i} y_k \cdot \frac{1}{p_k} \cdot p_k \left(0 - p_i\right)\right]$$

$$= \left[-y_i \cdot \frac{1}{p_i} \cdot p_i \left(1 - p_i\right)\right] + \left[-\sum_{k \neq i} y_k \cdot \frac{1}{p_k} \cdot p_k \left(0 - p_i\right)\right]$$

$$= \left[-y_i \left(1 - p_i\right)\right] + \left[-\sum_{k \neq i} y_k \cdot \left(0 - p_i\right)\right]$$

$$= -y_i + y_i.p_i + \sum_{k \neq i} y_k.p_i$$

$$= -y_i + p_i \left(y_i + \sum_{k \neq i} y_k\right)$$

$$= -y_i + p_i \left(\sum_k y_k\right)$$

$$\text{note that } \sum_k y_k = 1 \text{ as it is a One hot encoded Vector}$$

$$= p_i - y_i$$

$$\frac{\partial L}{\partial z^l} = p_i - y_i \rightarrow \quad \text{EqA1.1}$$

\frac {\partial L}{\partial z_i} = -\sum_k y_k \cdot \frac {1}{p_k} \cdot \frac {\partial { p_k}}{\partial z_i}

\ \

=-\sum_k y_k \cdot \frac {1}{p_k} \cdot p_i(\delta_{ij} -p_j) \ \ \text{these i and j are dummy indices and we can rewrite this as}

\ \

=-\sum_k y_k \cdot \frac {1}{p_k} \cdot p_k(\delta_{ik} -p_i) \ \ \text{taking the two cases and adding in above equation } \ \ \delta_{ik} = 1 \text{ when i =k} \text{ and }

\delta_{ik} = 0 \text{ when i} \ne \text{k}

\ \

= [- y_i \cdot \frac {1}{p_i} \cdot p_i(1 -p_i)]+[-\sum_{k \ne i} y_k \cdot \frac {1}{p_k} \cdot p_k(0 -p_i)]

\ \

= [- y_i \cdot \frac {1}{p_i} \cdot p_i(1 -p_i)]+[-\sum_{k \ne i} y_k \cdot \frac {1}{p_k} \cdot p_k(0 -p_i)]

\ \

= [- y_i(1 -p_i)]+[-\sum_{k \ne i} y_k \cdot (0 -p_i)]

\ \

= -y_i + y_i.p_i + \sum_{k \ne i} y_k.p_i \ \

= -y_i + p_i(y_i + \sum_{k \ne i} y_k) \ \

= -y_i + p_i(\sum_{k} y_k)

\ \

\text {note that } \sum_{k} y_k = 1 \, \text{as it is a One hot encoded Vector}

\ \

= p_i - y_i

\ \

\frac {\partial L}{\partial z^l} = p_i - y_i \rightarrow \quad \text{EqA1.1}

\end{aligned}

} $$

We need to put this back in $EqA1$. We now need to calculate the second term, to complete the equation

$$\frac{\partial L}{\partial W^l} = \frac{\partial L}{\partial z^l} \cdot \frac{\partial z^l}{\partial W^l}$$

$$z^l = \left(W^l a^{l-1} + b^l\right)$$
$$\frac{\partial z^l}{\partial W^l} = \left(a^{l-1}\right)^T$$

Putting all together

$$\frac{\partial L}{\partial W^l} = (p - y) \cdot \left(a^{l-1}\right)^T \quad \rightarrow \quad \mathbf{EqA1}$$

\frac {\partial L}{\partial W^l}

= \color{red}{\frac {\partial L}{\partial z^l}} \cdot \color{green}{\frac {\partial z^l}{\partial W^l}}

\ \

z^{l} = (W^l a^{l-1}+b^l)

\

\color{green}{\frac {\partial z^l}{\partial W^l} = (a^{l-1})^T}

\ \ \text{Putting all together} \ \

\frac {\partial L}{\partial W^l} = (p - y) \cdot (a^{l-1})^T \quad \rightarrow \quad \mathbf {EqA1}

\end{aligned}

7.2 Gradient Descent

Using Gradient descent we can keep adjusting the last layer like

$$W^l = W^l - \alpha \cdot \frac{\partial L}{\partial W^l}$$

Now let’s do the derivation for the inner layers

7.3 Derivative of Loss with Respect to Weights in Inner Layers

The trick here is to find the derivative of the Loss with respect to the inner layer as a composition of the partial derivative we computed earlier. And also to compose each partial derivative as partial derivative with respect to either z^x or w^x but not with respect to a^x. This is to make derivatives easier and intuitive to compute.

$$\frac{\partial L}{\partial W^{l-1}} = \frac{\partial L}{\partial z^{l-1}} \cdot \frac{\partial z^{l-1}}{\partial W^{l-1}} \rightarrow \text{EqA2}$$

\frac {\partial L}{\partial W^{l-1}}

= \color{blue}{\frac {\partial L}{\partial z^{l-1}}} \cdot \color{green}{\frac {\partial z^{l-1}}{\partial W^{l-1}}} \rightarrow \text{EqA2}

\end{aligned}

We represent the first part in terms of what we computed earlier ie $\frac{\partial L}{\partial z^l}$

$$

$$\frac{\partial L}{\partial z^{l-1}} = \frac{\partial L}{\partial z^l} \cdot \frac{\partial z^l}{\partial a^{l-1}} \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} \rightarrow \text{ Eq with respect to Prev Layer}$$

$$\frac{\partial L}{\partial z^l} = (p_i - y_i) \text{ from the previous layer (from EqA1.1)}$$

$$z^l = w^l a^{l-1} + b^l \text{ which makes } \frac{\partial z^l}{\partial a^{l-1}} = w^l$$
$$\text{and } a^{l-1} = \sigma\left(z^{l-1}\right) \text{ which makes } \frac{\partial a^{l-1}}{\partial z^{l-1}} = \sigma\prime\left(z^{l-1}\right)$$

$$\text{Putting together we get the first part of Eq A2}$$

$$\frac{\partial L}{\partial z^{l-1}} = \left(\left(W^l\right)^T \cdot (p - y)\right) \odot \sigma\prime\left(z^{l-1}\right) \rightarrow \text{EqA2.1}$$

$$z^{l-1} = W^{l-1} a^{l-2} + b^{l-1} \text{ which makes } \frac{\partial z^{l-1}}{\partial W^{l-1}} = \left(a^{l-2}\right)^T$$

$$\frac{\partial L}{\partial W^{l-1}} = \frac{\partial L}{\partial z^{l-1}} \cdot \frac{\partial z^{l-1}}{\partial W^{l-1}} = \left(\left(\left(W^l\right)^T \cdot (p - y)\right) \odot \sigma\prime\left(z^{l-1}\right)\right) \cdot \left(a^{l-2}\right)^T$$

\color{blue}{\frac {\partial L}{\partial z^{l-1}}} =

\color{blue}{\frac {\partial L}{\partial z^{l}}}.

\frac {\partial z^{l}}{\partial a^{l-1}}.

\frac {\partial a^{l-1}}{\partial z^{l-1}} \rightarrow \text{ Eq with respect to Prev Layer}

\ \

\color{blue}{\frac {\partial L}{\partial z^{l}}} = \color{blue}{(p_i- y_i)}

\text{ from the previous layer (from EqA1.1) }

\ \

z^l = w^l a^{l-1}+b^l

\text{ which makes }

{\frac {\partial z^{l} }{\partial a^{l-1}} = w^l} \ \text{ and }

a^{l-1} = \sigma (z^{l-1}) \text{ which makes }

\frac {\partial a^{l-1}}{\partial z^{l-1}} = \sigma \color{red}{'} (z^{l-1})

\ \ \text{ Putting together we get the first part of Eq A2 }

\\

\color{blue}{\frac {\partial L}{\partial z^{l-1}}} = \left((W^l)^T \cdot \color{blue}{(p- y)} \right) \odot \sigma \color{red}{'} (z^{l-1}) \rightarrow \text{EqA2.1 }

\ \

z^{l-1} = W^{l-1} a^{l-2}+b^{l-1}

\text{ which makes }

\color{green}{\frac {\partial z^{l-1}}{\partial W^{l-1}}=(a^{l-2})^T}

\ \

\frac {\partial L}{\partial W^{l-1}}

= \color{blue}{\frac {\partial L}{\partial z^{l-1}}} \cdot \color{green}{\frac {\partial z^{l-1}}{\partial W^{l-1}}} = \left(\left((W^l)^T \cdot \color{blue}{(p- y)} \right) \odot \sigma '(z^{l-1}) \right) \cdot \color{green}{(a^{l-2})^T}

\end{aligned}

$$

7.4 Some Implementation Details

For a detailed explanation of the Matrix Calculus, Jacobian, and Hadamard product used here, please refer to **Chapter 6: Back Propagation - Matrix Calculus**.

From Index Notation to Matrix Notation

The equations above use index notation for clarity. In practice, we use Matrix Notation which involves Transposes and Hadamard products as explained in the previous chapter.

7.4.1 Implementation in Python

Here is an implementation of a relatively simple Convolutional Neural Network to test out the forward and back-propagation algorithms given above https://github.com/alexcpn/cnn_in_python. The code is well commented and you will be able to follow the forward and backward propagation with the equations above.

7.5 Gradient Descent

Using Gradient descent we can keep adjusting the inner layers like

$$W^{l-1} = W^{l-1} - \alpha \cdot \frac{\partial L}{\partial W^{l-1}}$$

Next: Neural Network Implementation

7.6 References

- Supervised Deep Learning - Marc'Aurelio Ranzato (DeepMind) - Easier to follow (without explicit Matrix Calculus) though not really correct
- Notes on Backpropagation - Peter Sadowski - Easy to follow but lacking in some aspects
- The Softmax function and its derivative - Eli Bendersky - Slightly hard to follow using the Jacobian
- Backpropagation In Convolutional Neural Networks - Jefkine - More difficult to follow with proper index notations (I could not) and probably correct
- A Primer on Index Notation - John Crimaldi
- The Matrix Calculus You Need For Deep Learning - Terence Parr & Jeremy Howard
- Neural Networks and Deep Learning - Michael Nielsen
- Derivative of Softmax Activation - Alijah Ahmed

8 Neural Network Implementation

With the derivative of the Cost function derived from the last chapter, we can code the network We will use matrices to represent input and weight matrices.

> [!NOTE] **Important Note on Matrix Conventions:** In the previous chapters (Matrix Calculus), we derived equations assuming **Column Vectors** (where input x is $N \times 1$). However, in standard Python/NumPy implementations (like the one below), we typically use **Batch Processing** where inputs are **Row Vectors** (where input X is $BatchSize \times Features$).
>
> This means: * Input X has shape `(BatchSize, InputFeatures)` * Weight W has shape `(InputFeatures, OutputFeatures)` * Forward pass is $Z = X \cdot W$ (instead of $W \cdot x$) * This effectively transposes the standard mathematical notation. The code below follows this "Row Vector/Batch" convention.

```
x = np.array(
    [
        [0,0,1],
        [0,1,1],
        [1,0,1],
        [1,1,1]
    ])
```

This is a 4*3 matrix. Note that each row is an input. lets take all this 4 as 'training set'

```
y = np.array(
  [
      [0],
      [1],
      [0],
      [1]
  ])
```

Note you can change the output and try to train the Neural network This is a 4*1 matrix that represent the expected output. That is for input [0,0,1] the output is [0] and for [0,1,1] the output is 1 etc.

A neural network is implemented as a set of matrices representing the weights of the network.

Let's create a two layered network. Before that please note the formula for the neural network So basically the output at layer l is the dot product of the weight matrix of layer l and input of the previous layer.

Now let's see how the matrix dot product works based on the shape of matrices.

```
[m*n].[n*x] = [m*x]
[m*x].[x*y] = [m*y]
```

We take the $[m*n]$ as the input matrix this is a $[4*3]$ matrix.

Similarly the output y is a $[4*1]$ matrix; so we have $[m*y] = [4*1]$

So we have

```
m=4
n=3
x=?
y=1
```

Lets then create our two weight matrices of the above shapes, that represent the two layers of the neural network.

```
w0 = x
w1 = np.random.random((3,4))
w2 = np.random.random((4,1))
```

We can have an array of the weights to loop through, but for the time being let’s hard-code these. Note that ‘np’ stands for the popular numpy array library in Python.

We also need to code in our non-linearity. We will use the Sigmoid function here.

```
def sigmoid(x):
    return 1/(1+np.exp(-x))

# derivative of the sigmoid
def derv_sigmoid(x):
   return sigmoid(x)*(1-sigmoid(x))
```

With this we can have the output of first, second and third layer, using our equation of neural network forward propagation.

```
a0 = x
a1 = sigmoid(np.dot(a0,w1))

a2 = sigmoid(np.dot(a1,w2))
```

a2 is the calculated output from randomly initialized weights. So lets calculate the error by subtracting this from the expected value and taking the MSE.

$$C = \frac{1}{2} \parallel y - a^l \parallel^2$$

```
c0 = ((y-a2)**2)/2
```

Now we need to use the back-propagation algorithm to calculate how each weight has influenced the error and reduce it proportionally.

We use this to update weights in all the layers and do forward pass again, re-calculate the error and loss, then re-calculate the error gradient $\frac{\partial C}{\partial w}$ and repeat

$$

$$w^2 = w^2 - \left(\frac{\partial C}{\partial w^2}\right) * learningRate$$

$$w^1 = w^1 - \left(\frac{\partial C}{\partial w^1}\right) * learningRate$$

w^2 = w^2 - (\frac {\partial C}{\partial w^2})*learningRate \ \

w^1 = w^1 - (\frac {\partial C}{\partial w^1})*learningRate

\end{aligned}

$$

Let’s update the weights as per the formula derived in the previous chapter:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{w^1}} = \boldsymbol{\sigma\prime}\left(\mathbf{z^1}\right) * \left(\mathbf{a^0}\right)^{\mathbf{T}} * \boldsymbol{\delta^2} * \mathbf{w^2} . \boldsymbol{\sigma\prime}\left(\mathbf{z^2}\right) \quad \rightarrow \mathbb{E}\mathbf{q}\ (\mathbf{5})$$

\frac {\partial C}{\partial w^1} = \sigma'(z^1) * (a^{0})^T*\delta^{2}*w^2.\sigma'(z^2) \quad \rightarrow \mathbb Eq \; (5)
}

$$\delta^2 = \left(a^2 - y\right)$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{w^2}} = \boldsymbol{\delta^{2}}*\boldsymbol{\sigma'}\left(\mathbf{z^2}\right)*\left(\mathbf{a^1}\right)^{\mathbf{T}} \quad \rightarrow \mathbb{E}\mathbf{q}\ (\mathbf{3})$$

\frac {\partial C}{\partial w^2}= \delta^{2}*\sigma^{'}(z^2) * (a^{1})^T \quad \rightarrow \mathbb Eq \; (3) }

8.1 A Two layered Neural Network in Python

Below is a two layered Network; I have used the code from http://iamtrask.github.io/2015/07/12/basic-python-network/ as the basis. With minor changes to fit into how we derived the equations.

```
import numpy as np
# seed random numbers to make calculation deterministic
np.random.seed(1)

# pretty print numpy array
np.set_printoptions(formatter={'float': '{: 0.3f}'.format})

# let us code our sigmoid function
def sigmoid(x):
    return 1/(1+np.exp(-x))

# let us add a method that takes the derivative of x as well
def derv_sigmoid(x):
   return sigmoid(x)*(1-sigmoid(x))

#-------------------------------------------------------------

# Two layered NW. Using from (1) and the equations we derived as
explanations
# (1) http://iamtrask.github.io/2015/07/12/basic-python-network/
#-------------------------------------------------------------

# set learning rate as 1 for this toy example
learningRate = 1
```

```
# input x, also used as the training set here
x = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])

# desired output for each of the training set above
y = np.array([[0,1,1,0]]).T

# Explanation - as long as input has two ones, but not three,
output is One
"""
Input [0,0,1]  Output = 0
Input [0,1,1]  Output = 1
Input [1,0,1]  Output = 1
Input [1,1,1]  Output = 0
"""

# Randomly initialized weights
weight1 =  np.random.random((3,4))
weight2 =  np.random.random((4,1))

# Activation to layer 0 is taken as input x
a0 = x

iterations = 1000
for iter in range(0,iterations):

  # Forward pass - Straight Forward
  z1= np.dot(x,weight1)
  a1 = sigmoid(z1)
  z2= np.dot(a1,weight2)
  a2 = sigmoid(z2)
  if iter == 0:
    print("Initial Output \n",a2)

  # Backward Pass - Backpropagation
  delta2  = (a2-y)
  #-----------------------------------------------------------------
  # Calculating change of Cost/Loss wrto weight of 2nd/last layer
  # Eq (A) ---> dC_dw2 = delta2*derv_sigmoid(z2)*a1.T
  #-----------------------------------------------------------------

  dC_dw2_1  = delta2*derv_sigmoid(z2)
  dC_dw2  = a1.T.dot(dC_dw2_1)

  #-----------------------------------------------------------------
```

```
  # Calculating change of Cost/Loss wrto weight of 2nd/last layer
  # Eq (B)---> dC_dw1 =
derv_sigmoid(z1)*delta2*derv_sigmoid(z2)*weight2*a0.T
  # dC_dw1 = derv_sigmoid(z1)*dC_dw2*weight2_1*a0.T
  #--------------------------------------------------------------

  dC_dw1 =  np.dot(dC_dw2_1, weight2.T) * derv_sigmoid(z1)
  dC_dw1 = a0.T.dot(dC_dw1)

  #--------------------------------------------------------------
  #Gradient descent
  #--------------------------------------------------------------

  weight2 = weight2 - learningRate*(dC_dw2)
  weight1 = weight1 - learningRate*(dC_dw1)


print("New output",a2)

#--------------------------------------------------------------
# Training is done, weight2 and weight2 are primed for output y
#--------------------------------------------------------------

# Lets test out, two ones in input and one zero, output should be
One
x = np.array([[1,0,1]])
z1= np.dot(x,weight1)
a1 = sigmoid(z1)
z2= np.dot(a1,weight2)
a2 = sigmoid(z2)
print("Output after Training is \n",a2)
```

Output

```
Initial Output


 [[ 0.758]


 [ 0.771]
```

```
 [ 0.791]

 [ 0.801]]

New output [[ 0.028]

 [ 0.925]

 [ 0.925]

 [ 0.090]]

Output after Training is

 [[ 0.925]]
```

We have trained the NW for getting the output similar to y; that is [0,1,0,1]

8.2 References

- A Neural Network in 11 lines of Python - I Am Trask
- Colab Notebook

Index