Neural Networks: Core Concepts

Cascades of Logistic Regressions

In traditional statistics or machine learning, you typically work with a single **logistic regression** and define its input features. In neural networks, you build cascades of these regressions. The network learns what would be useful inputs to downstream neurons, allowing for self-organization of intermediate levels of representation. This is what makes neural networks more powerful than other forms of machine learning.

Layered Structure and Matrix Operations

Neural networks utilize a regular structure of layers. The output of neurons in one layer is fed, with weights, to produce inputs for the next layer. This process involves:

1. **Matrix Multiplication**: Calculating WX of the inputs.
2. **Bias Addition**: Adding a vector of biases.
3. **Nonlinearity/Activation Function**: Applying a scalar function to each component of the vector.

Example: Location Prediction

Consider predicting whether a word in the middle of an input window is a location. After matrix multiplication and applying a nonlinearity, a dot product is performed and fed into a sigmoid function to predict yes or no.

Activation Functions: The Slope Matters

Threshold Units

In the earliest neural network implementations (1940s), a threshold was used to determine if a neuron should fire.

> If the activation is greater than θ, output 1; otherwise, output 0.

However, since there is no slope in threshold units, learning becomes much harder.

Gradient-Based Learning

The secret to building effective neural networks lies in gradient-based learning. Having slopes allows us to determine the steeper directions and optimize the function more quickly.

Sigmoid and TanH

Sigmoidal logistic functions were an early attempt to add slopes. The next popular choice was the **TanH function**.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

TanH is essentially a rescaled logistic function, stretched by two and moved down by one.

Hard TanH

To reduce computational cost, a "hard TanH" was considered with a slope of one in the middle and flat regions outside that area.

ReLU (Rectified Linear Unit)

ReLU is defined as:

$$f(x) = \max(0,x)$$

It is zero for negative values and $y = x$ for positive values. Despite being "dead" in the negative region, ReLU became popular because it allowed for specialization and easy backward flow of gradients.

Nonlinearities in Neural Networks

After ReLU, researchers developed other functions that are similar but not technically "dead."

Leaky ReLU

In Leaky ReLU, the negative half is a straight line with a very minor slope.

Parametric ReLU

A variant of Leaky ReLU, Parametric ReLU has an extra parameter that determines the slope of the negative part.

Swish and GeLU

Recently, nonlinearities like **Swish** and **GeLU** are commonly used in Transformer models. They are similar to y = x for all intents and purposes, but have a "funky" curve that provides a bit of slope.

Why Nonlinearities?

Neural networks approximate complex functions, such as mapping text to meaning or interpreting visual scenes.

> A matrix multiply of a vector is a linear transform. If you're doing multiple matrix multiplies, you're doing multiple linear transforms, but they compose.

Therefore, multilayer networks that are just matrix multiplies give you no additional representational power. However, they do have some learning power. To learn more complex functions, we need more than linear transforms. **Activation functions** give us nonlinear functions.

Gradient-Based Learning and Backpropagation

We use **gradient-based learning** with the stochastic gradient descent equation:

$\nabla J(\theta)$

where the upside-down triangle symbol is the gradient.

Our goal is to calculate gradients for an arbitrary function. We will do this by hand mathematically and then computationally using the **backpropagation algorithm**, which automates the math. The math involves **Matrix calculus**.

> The mantra to have in your head is multivariable calculus is just like single variable calculus except you're using matrices.

We use vectors, matrices, and higher-order tensors for **vectorized gradients**, which are a fast and efficient way to perform operations in neural networks.

Math 51 at Stanford

Math 51 at Stanford covers linear algebra, multivariate calculus, and modern applications, including neural networks and the multivariable chain rule in appendix G.

Single Variable Derivative

If $f(x) = x^3$, then the single variable derivative is $3x^2$. This derivative indicates the slope of the function.

For example:

- At x = 1, the slope is approximately $3 * 1^2 = 3$. If we change the input from 1 to 1.01, the output increases by about 0.03.
- At x = 4, the derivative is $3 * 4^2 = 48$. If we change the input from 4 to 4.01, the output increases from 64 to approximately 64.48, magnified 48 times.

Function with n Inputs

If we have a function with n inputs, we can work out its gradient which is its partial derivative with respect to each input. The gradient will be a vector of the same size as the number of inputs.

Function with n Inputs and m Outputs

For a function with n inputs and m outputs, the gradient is the **Jacobian**, which is a matrix of partial derivatives. For each output and each input, the partial derivative between the component of the input and the output is calculated.

$$\frac{\partial f_i}{\partial x_j}$$

Composition of Functions

In neural networks, multi-level computations correspond to the composition of functions.

Matrix Calculus and Tensors

Matrix calculus extends single variable calculus using tensors, progressing from scalars to vectors to matrices, and further to multi-dimensional arrays.

> **Tensor:** A generalization of scalars, vectors, and matrices to an arbitrary number of indices.

Jacobian Matrices for Multiple Variable Functions

For multiple variable functions, we multiply **Jacobians**, which are matrices of partial derivatives. For example, given a function WX + B composed with a nonlinearity F to get H, we can compute the derivatives as a product of Jacobians.

Element-wise Activation Functions

When a vector is computed as the activation function of a previously computed quantity, i.e., $h_i = F(z_i)$, where F is the activation function, we're dealing with component-wise operations. For a layer with n outputs and n inputs, the **Jacobian** is an n × n matrix.

The Jacobian is defined as:

$$J_{i,j} = \frac{\partial h_i}{\partial z_j}$$

In this specific case:

- If i = j, then h_j depends on z_j.
- If i ≠ j, the change in value doesn't affect the output because the output only depends on the corresponding index, resulting in a value of zero.

Thus, the Jacobian of activation functions is a matrix with zeros everywhere except on the diagonal, where the diagonal terms correspond to the derivative of the activation function.

Partial Derivatives in Neural Networks

Consider a layer in a neural network calculating WX + B. The partial derivative of this with respect to X is:

$$\frac{WX+B)}{\partial X} = \partial(\; W$$

Similarly, the partial derivative with respect to B is the identity matrix I:

$$\frac{WX+B)}{\partial B} = \partial(\; I$$

Dot Product Example

For a dot product of vectors U and V , if we compute the Jacobian, we obtain the transpose of the other vector.

Neural Network Partial Derivatives

Consider a neural network layer where we calculate Z = WX + B, then apply an activation function. To compute the partial derivative of the score S with respect to the bias B (i.e., ∂∂BS), we break down the equation into component pieces. We apply the chain rule:

$$= \frac{\partial S}{\partial H} * \frac{\partial H}{\partial Z} * \frac{\partial Z}{\partial B}$$

$\frac{\partial S}{\partial B}$ where:

- ∂∂HS is U T
-
- ∂∂Hz is the diagonal matrix of the derivative of F(Z)

∂∂BZ is the identity matrix I

Hadamard Product

The result simplifies to U ^{T} times the element-wise derivative of F, denoted as:

> **Hadamard Product:** Element-wise multiplication of two vectors, resulting in another vector of the same type.

Calculating Partial Derivatives

In a neural network, we want to calculate partial derivatives with respect to all variables (e.g., X, W, B, U) to optimize the model.

To calculate $\frac{\partial S}{\partial W}$, we use the same chain rule:

$$\frac{\partial S}{\partial W} = \frac{\partial S}{\partial H} * \frac{\partial H}{\partial Z} * \frac{\partial Z}{\partial W}$$

Notice that $\frac{\partial S}{\partial H}$ and $\frac{\partial H}{\partial Z}$ are the same as before.

Upstream Gradient

We can define Δ (Delta) as the upstream gradient or error signal, which is the shared part from the beginning ($\frac{\partial S}{\partial H} * \frac{\partial H}{\partial Z}$). This allows us to reuse computations.

Shape Convention

When computing $\frac{\partial S}{\partial W}$, we encounter a challenge. W is an n × m matrix, and S is a scalar. Math dictates that the result should be a 1 × (n · m) Jacobian (a long row vector). However, for engineering convenience, we use the **shape convention**:

> **Shape Convention:** Reshape Jacobians to match the shape of the parameters, allowing for easy subtraction during gradient descent.

We represent $\frac{\partial S}{\partial W}$ as an n × m matrix.

Calculating `dsdw`

`dsdw` is calculated as follows:

$Dsdw = \Delta T * X_T$

Where:

- ΔT is the **upstream gradient**.
- X_T is the **local input signal**.

This involves taking the **transpose** of the two vectors and calculating their **outer product**, resulting in the gradient.

Shape of Derivatives

There's a tension between:

- **Jacobian form**: Necessary for the chain rule.
- **Shape convention**: Used for storing tensors and stochastic gradient descent.

For assignments, adhere to the **shape convention**, ensuring derivatives with respect to a matrix are shaped like the original matrix.

Back Propagation Algorithm

The back propagation algorithm consists of two components:

1. Utilizing the **chain rule** for calculus of complex functions.
2. **Storing intermediate results** to prevent redundant computations.

Computational Graph

Functions can be represented as a graph:

- **Source nodes**: Inputs.
- **Interior nodes**: Operations.
- **Edges**: Pass along the results of each operation.

Forward Pass

The forward pass involves calculating functions. For example, to determine whether a word at the center is a location:

1. Take the input vector X.

2. Multiply it by W.
3. Add B.
4. Apply a nonlinearity.
5. Compute the dot product with vector U to get S.

Backward Pass

The backward pass involves calculating gradients for gradient-based learning.

1. Start with dS/dS = 1.
2. Work backward to compute dS/dz, dS/db, dS/dw, and dS/dx.

Single Node Gradient Calculation

For a node where H = F(x):

1. We have an **upstream gradient** dS/dH.
2. We want to calculate the **downstream gradient** dS/dz.

   dS/dz = (dS/dH) ∗ (dH/dz)

   Downstream gradient = Upstream gradient * Local gradient

Multiple Input Gradient Calculation

When a node has multiple inputs (e.g., calculating W_x):

1. We have a single **upstream gradient**.
2. We calculate the **downstream gradient** with respect to each input.
3. We compute the **local gradient** with respect to each input.
4. Multiply the **upstream gradient** by the **local gradient** with respect to each input.

Example: `f(x, y, z) = x + y * max(y, z)`

Given f(x,y,z) = x + y ∗ max(y,z) with current values x = 1, y = 2, and z = 0:

Forward Propagation

1. Compute max(2,0) = 2.

2. Compute 1 + 2 = 3.
3. Compute 3 ∗ 2 = 6. So, f(x,y,z) = 6.

Backward Propagation

1. Compute local gradients:
    - da/dx = 1, da/dy = 1 where a = x + y
    - For max function, the gradient is 1 for the largest input, 0 for the smallest.
    - For the product, df/da = b and df/db = a.
2. Calculate derivatives:
    - df/df = 1
    - Multiply by local gradients for A and B.
    - For the max function, use upstream * 1 for the largest, 0 for the smallest.
    - For addition, send the gradient down in both directions.

Results

Derivative	Calculation	Value
df/dx	Gradient through addition	2
df/dy	Gradient through addition + gradient through max	5
df/dz	z has no effect on output when $y > z$, so the gradient of df/dz is 0	0

Estimating Gradients Numerically

To estimate gradients, we can make small changes to the inputs and observe the effect on the output. For instance, consider a function:

f(x,y) = (x + 2) ∗ max(y,1)

Let's say x = 1 and y = 2. Thus, f(1,2) = (1 + 2) ∗ max(2,1) = 3 ∗ 2 = 6.

- If we change x to 1.1, then f(1.1,2) = (1.1 + 2) ∗ 2 = 3.1 ∗ 2 = 6.2. A change of 0.1 in x results in a change of 0.2 in the output, so the gradient with respect to x is approximately 2.

  - If we change y to 2.1, then f(1,2.1) = (1 + 2) ∗ max(2.1,1) = 3 ∗ 2.1 = 6.3.

  Actually, f(1,2.1) = (1 + 2) ∗ 2.1 = 3 ∗ 2.1 = 6.3. A change of 0.1 in y results in a change of 0.3 in the output, so the gradient with respect to y is approximately 3.

Gradient Behavior Through Common Operations

When running backpropagation, the way gradients behave through different operations is as follows:

- **Plus Operation:** The gradient is distributed equally to each input. The upstream gradient is simply passed to each input.

- **Max Operation:** Acts like a gradient router. It sends the gradient to the input that was the maximum and zero to the other inputs.

- **Multiplication Operation:** This is a bit more complex. You switch the forward coefficients and multiply them by the upstream gradient to get the downstream gradient.

Efficient Backpropagation

The key to efficient backpropagation is to avoid redundant calculations. Instead of calculating the gradients for each variable independently, we calculate the shared parts of the computation only once.

General Algorithm for Forward and Backward Propagation

The general algorithm for forward and backward propagation can work with arbitrary computation graphs, provided they are directed acyclic graphs (DAGs).

1. **Topological Sort:** Sort the variables so that each variable only depends on variables to its left.
2. **Forward Propagation:** Calculate the variables in the order determined by the topological sort.
3. **Backward Propagation:**

  - Initialize the output gradient as 1.
  - Visit the nodes in reverse order.
  - Compute the gradient at each node by using the upstream gradient and the local gradient to compute the downstream gradient.

The computational complexity of calculating the gradients should be the same (in Big O terms) as the forward calculation. If the backward path is more computationally intensive, it indicates inefficiency due to recomputation of work.

Automatic Differentiation

> Automatic differentiation is the idea that if you have the symbolic form of what you're calculating with your forward pass, you should be able to automatically work out the backward pass for you.

Early deep learning frameworks like Theano attempted this, but modern frameworks have largely moved away from fully automated symbolic differentiation.

Modern Deep Learning Frameworks

Modern deep learning frameworks manage the computation graph and handle forward and backward propagation, but require you to define the local derivatives.

- You need to provide the forward computation and the local gradient for each layer or function.
- The framework then uses this information to compute the gradients automatically.

Implementing Forward and Backward Passes

To implement a new layer or function, you need to implement both a forward pass and a backward pass.

- **Forward Pass:** Compute the output of the layer given the inputs.

- **Backward Pass:** Compute the gradients of the output with respect to the inputs, given the upstream gradient.

A common trick is to store the values of the inputs during the forward pass so that they are available during the backward pass for computing the gradients.

Manual Gradient Checking

To ensure the **backward calculation** is correct, it's essential to verify the **derivative** of the function. The standard approach is **manual gradient checking** using **numeric gradients**.

Here's how to do it:

1. At a specific value x, estimate the gradient.
2. Pick a small value h.
    - A typical value for neural networks is around 10^{-4}, but it varies depending on the function.
3. Compute the function value at $x + h$ and $x - h$.
4. Estimate the slope using the formula: slope = ———— $f(x+h)2-hf(x-h)$
    - This should approximate the slope that the **backward pass** calculates.
5. Compare the estimated slope with the result from the **backward pass**.
    - If they are within approximately 10^{-2} of each other, the gradient calculation is likely correct.
    - If they differ significantly, there might be an error in the implementation.

It is better to use a **two-sided estimate** using both sides of h as opposed to the onesided estimate, $f(x + h)$, because it's more accurate and stable.

Why Not Always Use Numeric Gradients?

Although this method is effective, it's **incredibly slow** because the computation must be repeated for every parameter of the model. **Backpropagation** algorithms provide significant speed improvements.

- Numeric gradients are valuable for verifying the correctness of new layer implementations or custom functions.

Backpropagation: The Core of Neural Networks

Key aspects to remember:

- **Backpropagation:** The **chain rule** applied efficiently.
- **Forward Pass:** Function application.
- **Backward Pass:** Chain rule applied efficiently.

Modern Deep Learning Frameworks

Modern frameworks simplify the process by providing pre-defined layer types that can be easily combined.

The Importance of Understanding the Underpinnings

Understanding the math and processes behind neural networks allows for a deeper understanding of complex concepts and troubleshooting issues. For example, grasping the underlying mechanisms aids in understanding phenomena like **exploding** and **vanishing gradients** in **recurrent neural networks**.