Introduction to Backpropagation

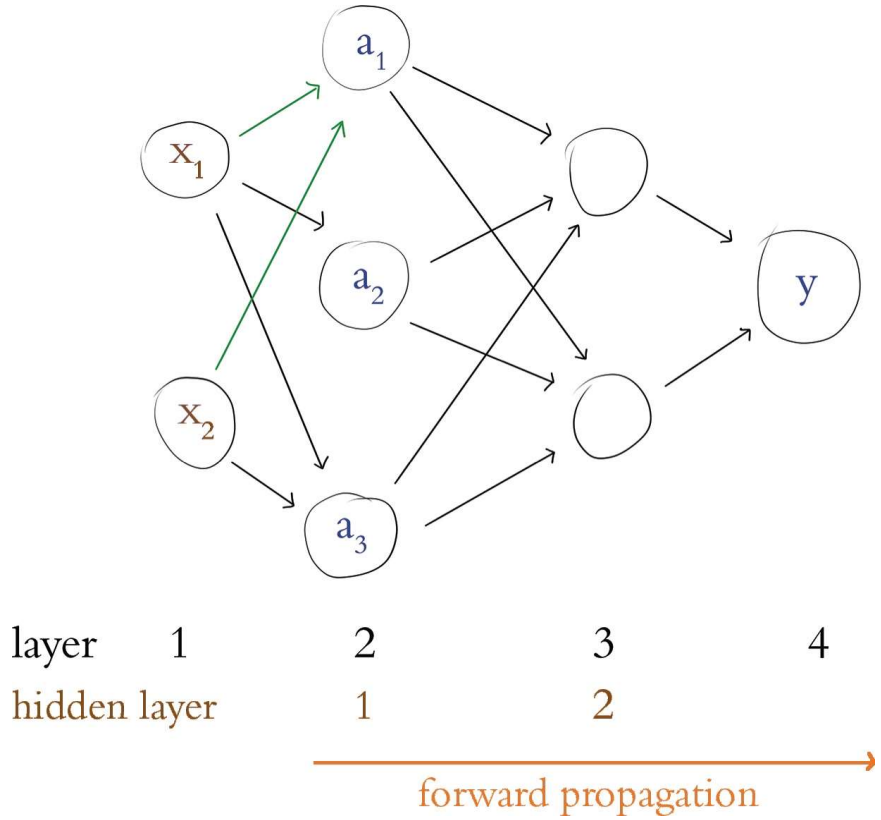
TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

Formal Neural Network Notation

- To keep discussion as straighforward as possible, we used a **shorthand** notation to identify neurons within a network.
- We now lay out a more widely used formal notation.
- This is important because:
  - We could want to possess a more precise manner for describing neurons;
  - Follow closely the backpropagation technique we will see later in this class.
- Let's take a look again at a figure that we have already seen.

A Hot Dog-Detecting Dense Network



A dense network of artificial neurons, highlighting the inputs to the neuron labeled $\mathbf{a_1}$.

Formal Neural Network Notation

- This neural network has a total of four layers.
- The first is the **input layer**.
  - It can be thought of as a collection of starting blocks for each data point to enter the network.
  - In the case of MNIST models, there are 784 starting blocks, representing each of the pixels in 28 x 28-pixel handwritten MNIST digit.
- No computation happens within an input layer.
  - It simply holds space for the input values to existe in.
  - The network knows how many values it needs to be ready to compute on in the next layer.

Formal Neural Network Notation

- The next two layers in the network are hidden layers.
  - It's where the bulk of the computation within a neural network occurs.
- The input values **x** are mathematically transformed and combined by each neuron in the hidden layer.
  - Outputting some activation value **a**.
- Because we need a way to address specific neurons in specific layers we'll use:
  - **Superscript** to define a layer, starting at the first hidden layer.
  - **Subscript** to define a neuron in that layer
- In the figure, we we'd have a^1_1, a^1_2, and a^1_3.

Formal Neural Network Notation

- We can then precisely refer to an individual neuron in a specific layer.
- For example, a^2_2 represents the second neuron in the second hidden layer.
- Because the last figure is a dense network, the neuron a^1_1 receives inputs from all of the neurons in the preceding layer.
  - Namely, the network inputs x_1 and x_2.
- Each neuron has its own bias, **b**, and we'll label that bias in exactly the same manner as the activation **a**: for example, b^1_2 is the bias or the second neuron of the in the first hidden layer.

Formal Neural Network Notation

- The green arrows in the figure represent the **mathematical transformation** that takes place during forward propagation.
- Each green arrow has its own individual weight associated with it.
- We employ the following notation: $w^1_{(1,2)}$ is the weight in the **first** hidden layer that connects neuron a^1_1 to its input x_2 in the input layer.
  - This double-barreled subscript is necessary because the network is fully connected.
  - Every neuron in a layer is connected to every neuron in the layer before it.
  - This connection carries its own weight.

Formal Neural Network Notation

- Let's generalize this weight notation:
  - The superscript is the hidden-layer number of the input-receiving neuron.
  - The first subscript is the number of the neuron receiving the input within its hidden layer.
  - The second subscript is the number of the neuron providing input from the preceding layer.
- As an example, the weight for neuron a^2_2 will be denoted $w^2_{(2,i)}$ where **i** is a neuron in the preceding layer.

Backpropagation

- We now use this formal notation to dive into the partial-derivative calculus behind the **backpropagation method**.
- Let's define some additional notation to help us along.
- Backpropagation works backwards → the notation is based on the final layer *L*.
- Earlier layers are annotated with respect to it: $L - 1$, $L - 2$, …, $L - n$.
- Weights, biases and outputs are subscripted appropriately with this notation.

Backpropagation

- The **layer activation** a^L is calculated by multiplying the **preceding layer's activation** (a^{L-1}) by the **weight** w^L and **bias** b^L terms to produce z^L and passing this through an activation function (denoted as σ).

- Also, we implement a cost function at the end.
  - Here, we are using Euclidean distance.

- For the final layer, we have:

$$z^L = w^L \cdot a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

$$C_0 = (a^L - y)^2$$

Backpropagation

- In every iteration, we need the gradient of the total error from the preceding layer: $\frac{\partial C}{\partial a^L}$
- The total error of the system is propagated backwards: δ_L.
- Backpropagation runs back-to-front → we start with the output layer.
- This layer is a special case since the error **originates** here and there no layer above it.
- δ_L is given as follows:

$$\delta_L = \frac{\partial C}{\partial a^L} = 2(a^L - y)$$

Backpropagation

- This is a special case for the initial δ value.
  - **The remaining layers will be different.**

- To update the weights in layer *L* we need to find the gradient of the cost w.r.t. the weights: $\frac{\partial C}{\partial w^L}$

- According to the chain rule, this is the product of the gradient of the cost for the layer before w.r.t its output. the gradient of the activation function w.r.t. *z*, and the gradient of *z* w.r.t. the weights w^L:

$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}$$

Backpropagation

- This can be simplified to:

$$\frac{\partial C}{\partial w^L} = \delta_L \cdot a^L(1 - a^L) \cdot a^{L-1}$$

- This value is essentialy the relative amount by which the weights at layer *L* affect the total cost.
  - We use this to update the weights at this layer.

- Our work isn't complete: we need to continue down the rest of the layers. For layer *L – 1*:

$$\delta_{L-1} = \frac{\partial C}{\partial a^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}}$$

Backpropagation

- The total error is being incorporated down the line → **backpropagated**.
- The remaining terms have derivatives, so the equation becomes:

$$\delta_{L-1} = \frac{\partial C}{\partial a^{L-1}} = \delta_L \cdot a^L(1 - a^L) \cdot w^L$$

- Now we need to find the gradient of the cost w.r.t. the weights at this layer *L – 1* as before:

$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial C}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}}$$

Backpropagation

- Once again, substituting δ_{L-1}, and taking the derivatives of the other terms, we get:

$$\frac{\partial C}{\partial w^{L-1}} = \delta_{L-1} \cdot a^{L-1}(1 - a^{L-1}) \cdot a^{L-2}$$

- **This process is repeated layer by layer all the way down to the first layer.**

Backpropagation

- To recap, **we first find δ_L which is the error of the cost function**.
- We use that value in the equation for the **derivative of the cost function w.r.t. the weights in layer *L***.
- In the next layer, **we find δ_{L-1} (the gradient of the cost w.r.t. the output layer *L - 1*)**.
- As before, this is used to find **calculate the gradient of the cost function w.r.t. the weights in layer *L – 1***.
- And so on: **backpropagation continues until we reach the model inputs.**

Backpropagation

- We have dealt with a network with single input, single hidden neurons, and single output.
- We know that, in practice, deep learning models are never this simple.
- This math scales straightforwardly given multiple neurons in a layer and multiple inputs and outputs.
- Let's consider the case where there are multiple output classes, such as when you're classifying MNIST digits.
  - There are 10 output classes (n = 10).
- For each class, the model provides a probability that a given input image belongs to that class.

Backpropagation

- To find the total cost, we find the sum of the cost over all the classes:

$$C_0 = \sum_{n=1}^{n} (a_n^L - y_n)^2$$

- In this equation, a^L and y are vectors, each containing n elements.

- To calculate $\frac{\partial C}{\partial w^L}$, we must account for the fact that there may be many

- neurons in the final hidden layer, each of them connected to each output neuron.

Backpropagation

- Let's switch the notation slightly: let the final hidden layer be *i* (which is the layer *L-1*) and the output layer be *j* (which is the layer *L*).
- We have a matrix of weights that can be accessed with:
  - a row for each output neuron.
  - and a column for each hidden-layer neuron.
- **Each weight can then be denoted w_{ji}.**
- We can now find the gradient on each weight:

$$\frac{\partial C}{\partial w_{ji}^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ji}^L}$$

Backpropagation

- **We calculate this for every single layer, creating the gradient vector for the weights of size *i x j*.**

- This is essentially the same as our single-neuron-per-layer backprop, the equation for the **gradient of the cost w.r.t. the preceding layer's output a_{L-1} *will change*.**

- This gradient is composed of the **partial derivatives of the current layer's inputs and weights**.
  - There are now multiple of those → we need to sum everything up.

$$\delta_{L-1} = \frac{\partial C}{\partial a_i^{L-1}} = \sum_{j=0}^{n_j - 1} \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial a_i^{L-1}}$$

Backpropagation

- In simple terms: the equations changed in the sense that **instead of calculating the gradient on a single weight, we need to calculate the gradient on multiple weights.**

- In order to calculate the gradient on any given weight, we need that δ value:
  - which itself it is composed of the error over a number of connections in the preceding layer.
  - **so we calculate the sum over all these errors.**

Backpropagation

- **Exercise** (3.10.1 from "Neural Networks and Deep Learning"): Consider the following recurrence:

$$(x_{t+1}, y_{t+1}) = (f(x_t, y_t), g(x_t, y_t))$$

Here, *f*() and *g*() are multivariate functions.

(a) Derive an expression for $\frac{\partial x_{t+2}}{\partial x_t}$ in terms of only x_t and y_t.

(b) Can you draw a architecture of a neural network corresponding to the above recursion for *t* varying from 1 to 5? Assume that the neurons can compute any function you want.

Backpropagation

- **Exercise** (3.10.2 from "Neural Networks and Deep Learning"): Consider a two-input neuron that multiplies its two inputs x_1 and x_2 to obtain the output o. Let L be the loss function that is computed at o. Suppose that you know that $\frac{\partial L}{\partial o} = 5$, $x_1 = 2$, and $x_2 = 3$. Compute the values of $\frac{\partial L}{\partial x1}$ and $\frac{\partial L}{\partial x2}$.

Backpropagation

- **Exercise** (3.10.3 from "Neural Networks and Deep Learning"): Consider a neural network with three layers including an input layer. The first (input) layer has four inputs x_1, x_2, x_3 and x_4. The second layer has six hidden units corresponding to all pairwise multiplications. The output node o simply adds the values in the six hidden units. Let L be the loss at the output node. Suppose that you know that $\frac{\partial L}{\partial o} = 2$, and $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, and $x_4 = 4$. Compute $\frac{\partial L}{\partial xi}$ for each i.

Backpropagation

- **Exercise** (3.10.4 from “Neural Networks and Deep Learning”): How does your answer to the previous question change when the output *o* is computed as a maximum of its six inputs rather than its sum?

Referências

1. Apêndices A e B do livro *“Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence”* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.