Fundamentals of Machine Learning:

Deep Learning and The Backpropagation Algorithm

Prof. Andrew D. Bagdanov (andrew.bagdanov AT unifi.it)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

DINFO
DIPARTIMENTO DI
INGEGNERIA
DELL'INFORMAZIONE

Outline

Introduction

Neural Networks are Universal Function Approximators

The Backpropagation Algorithm

Backpropagation Reflections

Discussion

Introduction

Lecture objectives

After this lecture you will:

- Understand how ReLU networks (with at least one hidden layer) can approximate to a desired precision any continuous function on a compact domain.
- Understand how backpropagation can be used to automatically and efficiently compute gradients of neural networks.
- Understand some of the pitfalls of the Backpropagation Algorithm and how they are mitigated in practice.

Neural Networks are Universal Function Approximators

An example function

- Consider a function $f[x, \phi]$ that maps a scalar input x to a scalar output y.
- This function has ten parameters $\phi = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$
- The definition of $f[x, \phi]$ is:

$$\begin{aligned} y &= f[x, \phi] \\ &= \phi_0 + \phi_1 \sigma[\theta_{10} + \theta_{11} x] + \phi_2 \sigma[\theta_{20} + \theta_{21} x] + \phi_3 \sigma[\theta_{30} + \theta_{31} x]. \end{aligned}$$

- The activation function σ we consider will be the Rectified Linear Unit (ReLU):

$$\sigma[z] = \text{ReLU}[z] = \max(0, z) = \begin{cases} 0 \text{ if } z < 0 \\ z \text{ otherwise} \end{cases}$$

An example function

- Let's implement this function and see what family it represents:

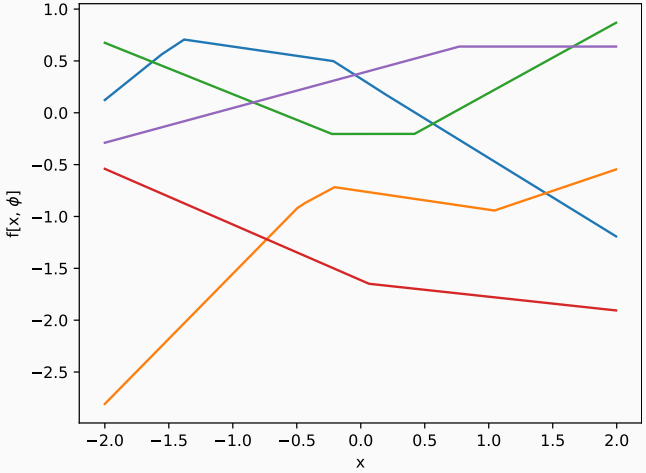
```
def random_parameters(hidden=4):
    W1 = np.random.normal(size=(hidden,1))
    b1 = np.random.normal(size=(hidden,1))
    W2 = np.random.normal(size=(hidden,1))
    b2 = np.random.normal()
    return (W1, b1, W2, b2)

def relu(z):
    return np.maximum(z, 0.0)

def f(x, W1, b1, W2, b2):
    return W2.T @ relu((W1*x.T + b1)) + b2
```

An example function

- By visualizing some random instances:



An example function

- This function represents the parameterized family of functions with at most four linear segments.
- That is, the family of piecewise linear functions with up to four linear regions.
- It is helpful to break down the computation of $f[x, \boldsymbol{\phi}]$ by first computing the hidden quantities:

$$\begin{aligned} h_1 &= \sigma[\theta_{10} + \theta_{11}x] \\ h_2 &= \sigma[\theta_{20} + \theta_{21}x] \\ h_3 &= \sigma[\theta_{30} + \theta_{31}x]. \end{aligned}$$

- We then combine these with:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3.$$

A shallow network

- Each linear region corresponds to an activation pattern in the hidden units.
- When a unit is clipped by the RelU activation, we refer to it as inactive.
- The slope of each region is determined by the original slopes θ_{*1} of the active inputs for this region and the weights ϕ_* that were subsequently applied.
- Each hidden unit contributes one joint to the function, so with three hidden units there can be four linear regions.

The Universal Approximation Theorem (shallow version)

- If we generalize this to D hidden units:

$$h_d = \sigma[\theta_{d0} + \theta_{d1}x] \text{ for } d \in \{1, \ldots, D\}$$

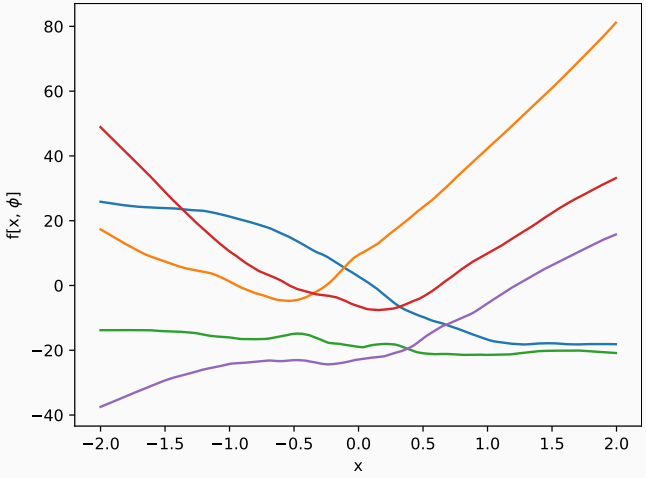
- And combine them in the same way:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

- The number of hidden units in a shallow network is a measure of the network capacity.
- With enough capacity a shallow network can describe *any continuous 1D function defined on a compact subset of the real line to arbitrary precision*.

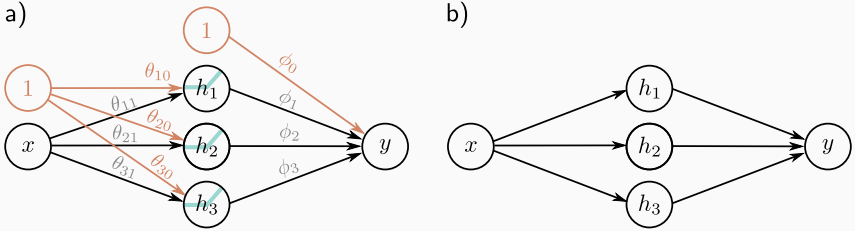
The Universal Approximation Theorem (shallow version)

- What if we increase the number of hidden unites to 1000?



The Humble Neural Network

- Of course we are describing the same architecture we already know:



- We usually think in terms of the simpler figure on the right.
- Important: The Universal Approximation Theorem only says that there exists a network with a single hidden layer with some D hidden units that approximates any continuous function f to any desired precision.

The Backpropagation Algorithm

Another toy network

- Consider the following function:

$$f[x, \boldsymbol{\phi}] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 x]\big]\Big]$$

- This is a composition of the functions $\cos[\bullet], \exp[\bullet], \sin[\bullet]$.
- With parameters $\boldsymbol{\phi} = \{\beta_0, \omega_0, \beta_1, \omega_1, \beta_2, \omega_2, \beta_3, \omega_3\}$.
- Suppose that we have a least squares loss function: $\ell_i = (f[x_i, \boldsymbol{\phi}] - y_i)^2$.
- And that we know the current values of $\beta_0, \beta_1, \beta_2, \beta_3, \omega_0, \omega_1, \omega_2, \omega_3$, x_i, and y_i.
- We could obviously calculate ℓ_i, but we instead interested in computing how small changes in the parameters $\boldsymbol{\phi}$ changes ℓ_i.

Calculus by hand sucks

- We could compute expressions for the partial derivatives by hand:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \omega_0} = -2 \Big(\beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\big]\Big] - y_i\Big) \\ \cdot \omega_1\omega_2\omega_3 \cdot x_i \cdot \cos[\beta_0 + \omega_0 \cdot x_i] \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big] \\ \cdot \sin\left[\beta_2 + \omega_2 \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big]\right].\end{aligned}$$

- But this is a pain in the ass and – more importantly – extremely redundant!
- Let's see if we can derive a generic algorithm to compute derivatives of functions like this...

The Computational Graph of $f[x, \phi]$

- The key is in how we decompose this composition of functions:

$$f[x, \phi] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 x]\big]\Big]$$

FORWARD: Decomposing the graph of the computation

- First, we write the computation of ℓ_i as a sequence of intermediate steps:

$$
\begin{aligned}
f[x_i, \boldsymbol{\phi}] &= \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 x_i]\big]\Big] \\
f_0 &= \beta_0 + \omega_0 x_i \\
h_1 &= \sin[f_0] \\
f_1 &= \beta_1 + \omega_1 h_1 \\
h_2 &= \exp[f_1] \\
f_2 &= \beta_2 + \omega_2 h_2 \\
h_3 &= \cos[f_2] \\
f_3 &= \beta_3 + \omega_3 h_3 \;\; (\equiv f[x_i, \boldsymbol{\phi}]) \\
l_i &= (f_3 - y_i)^2
\end{aligned}
$$

BACKWARD: The chain rule

- Next, compute derivatives of ℓ_i with respect to the intermediate quantities in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}.$$

- The first of these is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y).$$

- Working backward, the second can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3} \frac{\partial \ell_i}{\partial f_3}.$$

BACKWARD: Intuition

- This first backward step is:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}.$$

- The left-hand side asks how ℓ_i changes when h_3 changes.
- The right-hand side says we can decompose this into:
  1. How ℓ_i changes when f_3 changes; and
  2. How f_3 changes when h_3 changes.
- We get a chain of events happening: h_3 changes f_3, which changes ℓ_i, and the derivatives represent the effects of this chain.
- Note that already computed f_3 and the first of these derivatives $2(f_3 - y)$.

BACKWARD: Just keep going

- We continue computing the derivatives of the output with respect to intermediate quantities:

$$
\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned}
$$

BACKWARD: But... the loss? And the forward pass?!

- We still need to compute how the loss ℓ_i changes in terms of changes to parameters β_k and ω_k!
- Once more, we apply the chain rule:

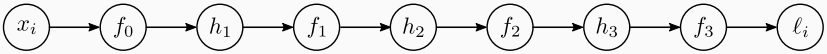
$$\begin{aligned}\frac{\partial l_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\ \frac{\partial l_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.\end{aligned}$$

  - And again: we have already computed $\frac{\partial \ell_i}{\partial f_k}$!
  - For $k > 0$ we have:

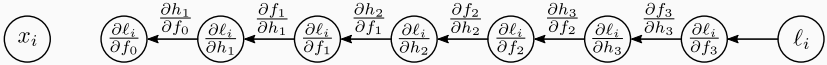
$$\frac{\partial f_k}{\partial \beta_k} = 1 \text{ and } \frac{\partial f_k}{\partial w_k} = h_k.$$

The Illustrated Backpropagation Algorithm

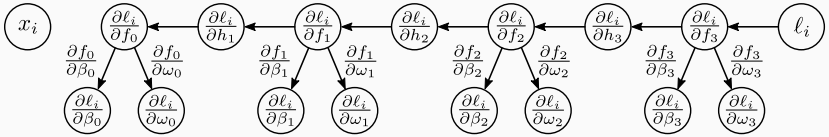
- Step 1: Compute the forward pass:



- Step 2: Backpropagate gradients of intermediate activations:



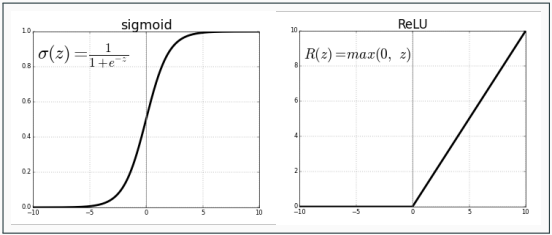
- Step 3: Compute final partial derivatives:



Backpropagation Reflections

Reflections: problems with backprop

- If the backpropagation algorithm is so “simple”, why haven’t we been using neural networks since the 1970s?
- There are a number of problems:
  - Saturating units: many activation functions are “flat” in their extremal values – this results in near zero gradients
  - Vanishing gradients: backprop creates a long chain of multiplied gradients – all of which are typically very small.
- Partial Solution: use non-saturating activation functions:



Reflections: more problems with backprop

- Another problem is overparameterization: the (often very) many parameters in neural networks can lead to easy overfitting.
- Good exercise: count the number of weights in an MLP.
- Partial solution: use regularization to control the magnitude of weights in the network.

Reflections: Stochastic Gradient Descent (SGD)

- Problem: what happens if N (the number of training samples) is very large?
- Well, we end up taking very slow steps – each iteration of gradient descent is an average over the entire dataset.
- Solution: approximate the true gradient with the gradient at a single training example:

Online Stochastic Gradient Descent

- Choose an initial vector of parameters θ and learning rate η.
- Repeat until an approximate minimum is found:
  1. Randomly shuffle training samples in D.
  2. For $(\boldsymbol{x}, y) \in D$:
     $$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\{\boldsymbol{x}, y\}; \boldsymbol{\theta})$$

Reflections: Stochastic Gradient Descent (continued)

- Another problem: evaluating the gradient on single examples leads to very noisy steps in parameter space.
- One trick to mitigate this is to use momentum: keep a running average of gradients that is slowly updated.
- Another solution is to use mini-batches: instead of a single sample, average the gradients over a small batch of samples.
- It is common to use a combination of mini-batches and momentum to stabilize training.

Reflections: Terminology

- Some useful terminology for deep learning optimization:
  - 1 epoch: one complete pass over the data.
  - 1 iteration: a single gradient step.
  - N: number of training samples.
  - B: batch size.

Algorithm	iterations per epoch
Batch gradient descent	1
Stochastic Gradient Descent	N
Mini-batch Gradient Descent	$\frac{N}{B}$

Discussion

Reflections on backpropagation

- Efficient reuse of partial computations while calculating gradients in computational graphs has been repeatedly discovered [Werbos (1974), Bryson et al. (1979), LeCun (1985), and Parker (1985)].
- The most celebrated description of this idea was by Rumelhart et al. (1985), who also coined the term “backpropagation.”
- This work kick-started a new phase of neural network research: for the first time, it was *practical to train networks with hidden layers*.
- Progress stalled due (in retrospect) to a lack of training data, limited computational power, and the use of sigmoid activations.
- Applications like NLP and Computer vision did not rely on neural network models until the remarkable image classification results of Krizhevsky et al. (2012).

Neural networks and the vicissitudes of history

- Research on neural networks was effectively stalled from about 1969 until the 1980s.
- Difficulties with training networks (even those with only a single hidden layer) prevented their widespread use and development.
- They represent, however, an extremely powerful family of functions we can use to model complex, nonlinear mappings from input to output.
- Training them still requires a delicate combination of art, mathematical background, and a lot of patience.
- Discovery of the backpropagation algorithm was a major milestone, and to effectively use (and debug) training loops, it is essential to know how it works.

The Road Forward

- In the next lecture we will look at some different architectures for Deep Neural Networks.
- Specifically, we will see how Convolutional Neural Networks (CNNs) can be used to efficiently process image data.
- And we will see how CNNs are really just Multilayer Perceptrons in disguise.
- We will also look more closely at some of the pitfalls in training deep networks, and the bag of tricks we use to avoid them (or to get out of them).