CSE 40171: Artificial Intelligence

Artificial Neural Networks: Gradient-Based Optimization
Homework #8 has been released. It is due on 11/30 at 11:59PM.
Group project presentations will happen Nov. 26 - 30. Check the course webpage for guidance.
CSE Seminar Speaker: Karl Ricanek

Co-founder and Chief Data Scientist of Lapetus Life Event Solutions

Professor of Computer Science and the University of North Carolina at Wilmington

Thursday 11/29
138 DeBartolo
3:30 - 4:45pm
How do we train a network?

A Neural network with two layers

inputs

input layer

hidden layer

output layer

outputs

A Neural network with two layers © BY-SA 3.0 Chrislb
Training Loop

1. Draw a batch of training samples $x$ and corresponding targets $y$

2. Run the network on $x$ (forward pass) to obtain predictions $y_{\text{pred}}$

3. Compute the loss of the network on the batch, a measure of the mismatch between $y_{\text{pred}}$ and $y$

4. Update the weights of the network in a way that slightly reduces the loss on this batch
Training Loop

**Step 1** is easy: just some I/O code

**Steps 2 & 3** just consist of a handful of tensor operations, also easy

**Step 4**, updating the network’s weights, is difficult

Given an individual weight coefficient in the network, how can we compute whether the coefficient should be increased or deceased, and by how much?
Naive Strategy for Updating Weights

Freeze all weights in the network except the one scalar coefficient being considered, and try different values for it. Repeat for all coefficients in the network.

Example:

- Initial value of coefficient: 0.3  
  Corresponding loss of net: 0.5
- New value of coefficient: 0.35  
  Corresponding loss of net: 0.6
- New value of coefficient: 0.25  
  Corresponding loss of net: 0.4

Why is this algorithm bad?
Gradient-Based Learning

Take advantage of the fact that all operations used in the network are **differentiable**, and compute the **gradient** of the loss with respect to the network’s coefficients.

Move coefficients in the opposite direction from the gradient, thus decreasing the loss.
Derivatives

Consider a continuous, smooth function \( f(x) = y \), mapping a real number \( x \) to a new real number \( y \).

The function is continuous: a small change in \( x \) can only result in a small change in \( y \).

If \( x \) is increased by a small factor \( \varepsilon_x \) this results in a small \( \varepsilon_y \) change to \( y \):

\[
f(x + \varepsilon_x) = y + \varepsilon_y
\]
Derivatives

\[ f(x) = y, \] is a smooth function (the curve doesn’t have any abrupt angles)

When \( \epsilon_x \) is small enough, around a certain point \( p \), it’s possible to approximate \( f \) as a linear function of slope \( a \), so that \( \epsilon_y \) becomes \( a \times \epsilon_x \):

\[
f(x + \epsilon_x) = y + a \times \epsilon_y
\]

This linear approximation is valid only when \( x \) is close enough to \( p \)
Derivative of $f$ in $p$

Local linear approximation of $f$, with a slope $a$

The slope $a$ is called the derivative of $f$ in $p$

If $a$ is negative, it means a small change of $x$ around $p$ will result in a decrease of $f(x)$

If $a$ is positive, a small change in $x$ will result in an increase of $f(x)$

Absolute value of $a$ tells you how quick this increase or decrease will happen
Differentiable functions

For every differentiable function \( f(x) \), there exists a derivative function \( f'(x) \) that maps values of \( x \) to the slope of the local linear approximation of \( f \) in those points.

**Examples:**

The derivative of \( \cos(x) \) is \(-\sin(x)\)

The derivative of \( f(x) = ax \) is \( f'(x) = a \)

The derivative completely describes how \( f(x) \) evolves as you change \( x \)

If you want to reduce the value of \( f(x) \), you just need to move \( x \) a little in the opposite direction of the derivative.
Derivative of a tensor operation: the gradient

A gradient is the generalization of the concept of derivatives to functions of multidimensional inputs.

Consider an input vector $x$, a matrix $W$, a target $y$, and a loss function $\text{loss}$. You can use $W$ to compute a target candidate $y_{\text{pred}}$ and compute the loss, or mismatch, between the target candidate $y_{\text{pred}}$ and the target $y$:

$$y_{\text{pred}} = \text{dot}(W, x)$$
$$\text{loss}_{\text{value}} = \text{loss}(y_{\text{pred}}, y)$$

If data inputs $x$ and $y$ are frozen, then:

$$\text{loss}_{\text{value}} = f(W)$$

Slide credit: F. Chollet, Deep Learning with Python
Derivative of a tensor operation: the gradient

Let’s say the current value of $W$ is $W_0$

The derivative of $f$ in $W_0$ is a tensor $\text{gradient}(f')(W_0)$ with the same shape as $W$,

- Each coefficient $\text{gradient}(f')(W_0)[i,j]$ indicates the direction and magnitude of the change in loss observed when modifying $W_0[i,j]$
- The tensor $\text{gradient}(f')(W_0)$ is the gradient of $f(W) = \text{loss\_value}$ in $W_0$

$\text{gradient}(f')(W_0)$ can be interpreted as the tensor describing the curvature of $f(W)$ around $W_0$
Derivative of a tensor operation: the gradient

You can reduce $f(W)$ by moving $W$ in the opposite direction from the gradient.

Example:

$$W_1 = W_0 - \text{step} \times \text{gradient}(f)(W_0)$$

Moves go against the curvature, which intuitively should put you lower on the curve.
Analytical Solution?

Yes: solve the equation $\text{gradient}(f)(W) = 0$ for $W$.

This is a polynomial equation of $N$ variables, where $N$ is the number of coefficients in the network.

How many coefficients are we typically dealing with in a modern neural network?
Finding the best weights: Hill Descent

How do we get to the bottom of the deepest valley?

How do we do this if we don’t have gravity?
Mini-batch Stochastic Gradient Descent (SGD)

1. Draw a batch of training samples $x$ and corresponding targets $y$
2. Run the network on $x$ to obtain predictions $y_{\text{pred}}$
3. Compute the loss of the network on the batch a measure of mismatch between $y_{\text{pred}}$ and $y$
4. Compute the gradient of the loss with respect to the network’s parameters (a backward pass)
5. Move the parameters a little in the opposite direction from the gradient, thus reducing the loss on the batch a bit
SGD down a 1D loss curve

Starting Point

$t = 0$

$t = 1$

$t = 2$

$t = 3$

Step, also called learning rate
Selecting a step value

It’s important to pick a reasonable value for the step factor

What happens if we choose a value that is too small?

What happens if we choose a value that is too large?
Too Small

Too Large

Variable (Just Right)

step = 0.1; 75 steps

step = 2; 10 steps

step = variable; 10 steps

Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data
Other SGD Variants

True SGD: draw a single sample and target at each iteration, rather than drawing a batch of data

Batch SGD: run every step on all available data
  ‣ Each update would be more accurate, but far more expensive

Mini-Batch SGD is an efficient compromise between the two strategies
Gradient Descent

Stochastic Gradient Descent

Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data
Momentum

How can we mitigating the problem off getting stuck in bad local minima?

A **momentum** term addresses two problems with SGD: local minima and convergence speed
Momentum

Imagine the optimization as a small ball rolling down the loss curve. If it has enough momentum, it won’t get stuck in a ravine and will end up at the global minimum.

Implement by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (from past acceleration).

i.e., update parameter $w$ based on the current gradient and on the previous parameter update.
Backpropagation

What’s an efficient way to calculate the gradients?

The gradient of a set of nested functions is the product of the individual derivatives:

$$\frac{\partial f_4(f_3(f_2(f_1(x))))}{\partial x} = \frac{\partial f_4}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x}$$

Gradients are matrices of all first order partial derivatives of $f_n$ (Jacobian)

$$\frac{\partial f_4(f_3(f_2(f_1(x))))}{\partial x} = \frac{\partial f_4}{\partial f_3} \cdot \left( \frac{\partial f_3}{\partial f_2} \cdot \left( \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} \right) \right)$$

Bad: For vectors of dimensionality $D$, need to propagate $D \times D$ Jacobian at each step
Backpropagation

\[
\frac{\partial f_4(f_3(f_2(f_1(x)))))}{\partial x} = \left( \left( \frac{\partial f_4}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \right) \cdot \frac{\partial f_2}{\partial f_1} \right) \cdot \frac{\partial f_1}{\partial x}
\]

Good: accumulate just a $D$-dimensional vector at each step by starting with scalar output

But... typical implementations store the entire training trajectory, $w_1...w_t$ in memory.

• Resource intensive for a single feed-forward deep network optimizing weights
Dropout

(a) Standard Neural Net

(b) After applying dropout.

Image Credit: Srivastava et al., JMLR 2014
Batch Normalization
What is the learning rule used by the brain?