Neural Networks and Deep Learning

Representation learning and multi-layer networks

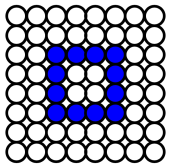
Adam Bloniarz

Department of Computer Science
adam.bloniarz@colorado.edu

January 30, 2020

University of Colorado Boulder

- Describe object in a way that makes it easy to perform a task
- "Easy" depends on the task



vs $\{(1,1),(1,-1),(-1,-1),(-1,1)\}$

Multinomial Logistic Regression

- Find representation that orthogonalizes classes.

$\rightarrow [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$

$\rightarrow [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$

- Representation needs to allow for linear (or low-complexity) separation of classes.

Representation should be invariant to appropriate transformations.

For image classification

- Translation
- Rotation (somewhat)
- Local deformation
- Background clutter / texture

The raw pixel representation provides none of these.

For speech recognition, we need invariance to

- Pitch
- Temporal dilation
- Background noise
- Voice, Accents

Many decades of research to design useful representations

- Image classification
  - SIFT
  - Fisher vector
  - Sparse coding
  - Scattering transform - provably invariant to deformations
- Speech recognition
  - Cepstral features
  - Psychophysics

These transformations often followed by a linear classifier or SVM.

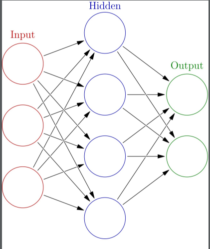
Lofty goal: let the computer learn the representation. The training data will decide what is a useful representation.

Provide a scaffolding, and let gradient descent fill in the details.

Encourage some 'inductive bias' in the scaffolding.

- The state of data as they pass through the network's hidden units are *learned features* or *representations*.
- The network must learn weights so the hidden representations/features/states maximize performance on the supervised task *at the output layer*.
- To learn good features, the supervised error must be pushed back through the network. How?

Feedforward networks



Stacked transformations

$$\mathbf{h}_{j+1} \leftarrow \sigma\left(\mathbf{W}_j \mathbf{h}_j + \mathbf{b}_j\right)$$

$$\mathbf{h}_j \in \mathbb{R}^{n_j}, \mathbf{h}_{j+1} \in \mathbb{R}^{n_{j+1}}$$

$\mathbf{W}_j$ is an $n_{j+1} \times n_j$ matrix.

$$\mathbf{b}_j \in \mathbb{R}^{n_{j+1}}$$

σ is a nonlinear activation function.

Why do we have a nonlinear activation function?

$$\begin{aligned}\mathbf{h}_{j+2} &= \mathbf{W}_{j+1}\left(\mathbf{W}_j\mathbf{h}_j + \mathbf{b}_j\right) + \mathbf{b}_{j+1} \\ &= \mathbf{W}_{j+1}\mathbf{W}_j\mathbf{h}_j + \mathbf{W}_{j+1}\mathbf{b}_j + \mathbf{b}_{j+1}\end{aligned}$$

Function space is still linear. Sometimes linear activations are useful.

Specifying a feedforward network

- Activation function
  - ReLU: $\sigma(u) = \max(0, u)$
  - Sigmoid: $\sigma(u) = \frac{e^u}{1+e^u} = \frac{1}{1+e^{-u}}$
  - Tanh: $\sigma(u) = \frac{e^{2u}-1}{e^{2u}+1}$
- Number of layers
- Width of each layer
- Loss function

Colab: fit one-dimensional analogue of XOR

Training deep networks

Goal is to minimize empirical risk w.r.t. the parameters of the network

$$\hat{\Theta} = \arg\min \frac{1}{n} \sum_{i=1}^{n} l\left(f_{\Theta}(\mathbf{x}_i), y_i\right)$$

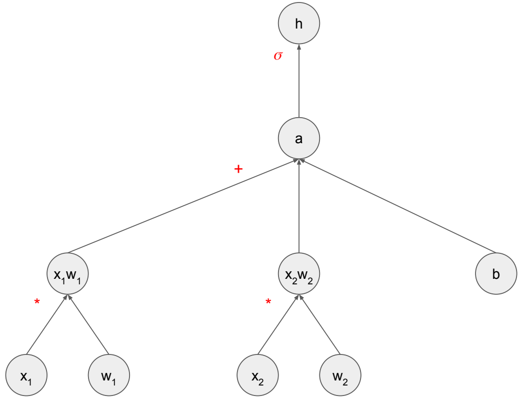
Now f_{Θ} is a complicated beast.

Use gradient descent, SGD, or mini-batch SGD as before.

Do not try to analytically calculate $\nabla_\Theta l\left(f_\Theta(\mathbf{x}_i), y_i\right)$

The neural network is a graph of computations. Use automatic differentiation.

Computation Graphs



Chain Rule

Given functions f and g, the derivative of their composition, $f \circ g$, is $f \circ g$ is

$$(f' \circ g) \cdot g'.$$

Equivalently, given variables z, y, and x, where z depends on y and y on x,

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

A computational graph is a directed acyclic graph (DAG) of function compositions.

Jacobian

- A vector-valued function is a mapping from $f : R^n \to R^m$.
- The Jacobian operator is a generalization of the derivative operator to vector-valued functions.
- Jacobian matrix J captures the rate of change of each component of y with respect to each component of input variable x.

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & . & . & \frac{\partial y_1}{\partial x_n} \\ . & . & . & . \\ . & . & . & . \\ \frac{\partial y_m}{\partial x_1} & . & . & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Finite Differences

- Numerical differentiation is the finite difference approximation of derivatives using original function evaluated at some sample points.

$$\frac{\partial f(x)}{\partial x_i} = \frac{f(x + he_i) - f(x)}{h}$$

- Pros - Not complicated to implement
- Cons -
  - » Numerical approximations of derivatives are inherently ill-conditioned and unstable, due to introduction of -
    - Truncation errors (caused by chosen value of x).
    - Round-off errors (caused by limited precision of computations).
  - » $O(n)$ computation complexity for a gradient in n dimensions.

Symbolic Differentiation

- Symbolic differentiation is the automatic manipulation of expressions for obtaining derivative expressions, carried out by applying transformations representing rules of differentiation such as -
$$\frac{\partial}{\partial x}(f(x)+g(x)) = \frac{\partial}{\partial x}f(x) + \frac{\partial}{\partial x}g(x)$$
- Pros - Can give valuable insight into structure of problem domain.
- Cons -
  - Careless symbolic differentiation can produce exponentially large symbolic expressions which take correspondingly long time to evaluate - *expression swell*
  - Limited expressivity

Intuition

- The insight behind AD is to apply symbolic differentiation at the elementary operation level and keep intermediate numerical results.
- AD can differentiate not only closed-form expressions, but also algorithms making use of control flow such as branching, loops, recursion, and procedure calls.

Overview

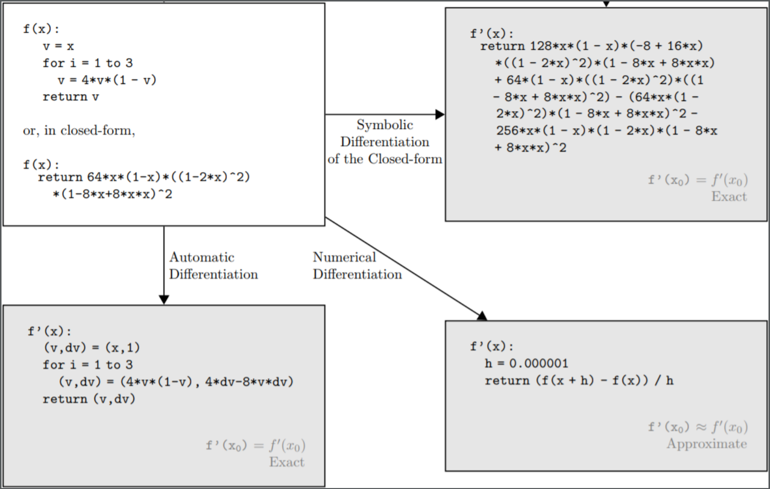
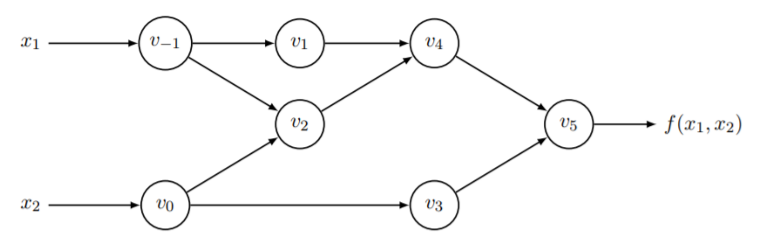


Figure taken from Automatic differentiation in machine learning: A survey

Example

$$f(x_1, x_2) = ln(x_1) + x_1 * x_2 - sin(x2)$$



Computational graph for $f(x_1, x_2)$, Figure taken from Automatic differentiation in machine learning: A survey

$$v_{-1} = x_1 \qquad v_0 = x_2$$

$$v_1 = ln(v_{-1}) \qquad v_2 = v_{-1} v_0$$

$$v_3 = sin(v_0) \qquad v_4 = v_1 + v_2$$

$$v_5 = v_4 - v_3 \qquad y = v_5$$

Intuition

- AD in forward mode is conceptually easy.
- Method
  - Apply chain rule to each elementary operation in the forward primal trace and generate corresponding tangent (derivative) trace.
  - Evaluating primals in lockstep with corresponding tangents gives the required derivative in the final variable.
- For cases $f : R^n \to R^m$, where $n \gg m$, Forward-mode becomes computationally expensive.

Example

Forward Primal Trace

$$
\begin{array}{lll}
v_{-1} & = x_1 & = 2 \\
v_0 & = x_2 & = 5 \\
\hline
v_1 & = ln(v_{-1}) & = ln2 \\
v_2 & = v_{-1} v_0 & = 10 \\
v_3 & = sin(v_0) & = sin5 \\
v_4 & = v_1 + v_2 & = 10.693 \\
v_5 & = v_4 - v_3 & = 11.652 \\
\hline
y & = v_5 & = 11.652
\end{array}
$$

Forward Tangent Trace

$$
\begin{array}{lll}
\dot{v}_{-1} & = \dot{x}_1 & = 1 \\
\dot{v}_0 & = x_2 & = 0 \\
\hline
\dot{v}_1 & = \dot{v}_{-1} / v_{-1} & = 1/2 \\
\dot{v}_2 & = \dot{v}_{-1} v_0 + \dot{v}_0 v_{-1} & = 5 \\
\dot{v}_3 & = \dot{v}_0 cos(v_0) & = 0 \\
\dot{v}_4 & = \dot{v}_1 + \dot{v}_2 & = 5.5 \\
\dot{v}_5 & = \dot{v}_4 - \dot{v}_3 & = 5.5 \\
\hline
\dot{y} & = \dot{v}_5 & = 5.5
\end{array}
$$

Forward mode AD example to compute $\frac{\partial y}{\partial x_1}$.The original evaluation of primals on the left is augmented by tangent operations on the right.

Complexity

- Forward mode is efficient and straightforward for functions $f : R \to R^m$, as all derivatives of $\frac{\partial y_i}{\partial x}$ can be computed in one forward pass.
- For $f : R^n \to R$, forward mode requires n evaluations to compute the gradient

$$\nabla f = \left(\frac{\partial y}{\partial x_1}, \dots, \frac{\partial y}{\partial x_n}\right)$$

which corresponds to a $1 \times n$ Jacobian matrix built one column at a time.

- AD in reverse-mode corresponds to a generalized back propagation algorithm, in that it propagates derivatives backward from a given output.
- This is done by complementing each intermediate variable v_i with an adjoint $\frac{\partial y_j}{\partial v_i}$, which represents the sensitivity of a considered output y_j w.r.t changes in v_i.
- Method -
  - » Original function code is run forward, populating intermediate variables v_i and recording dependencies in the computational graph in a book-keeping procedure
  - » Derivatives are calculated by propagating adjoints in reverse, from outputs to inputs
- For cases $f : R^n \to R^m$, where $n \gg m$, reverse mode AD performs well computationally.

Example

Forward Primal Trace

$$\begin{array}{lll}
v_{-1} & = x_1 & = 2 \\
v_0 & = x_2 & = 5 \\
\hline
v_1 & = ln(v_{-1}) & = ln2 \\
v_2 & = v_{-1} v_0 & = 10 \\
v_3 & = sin(v_0) & = sin5 \\
v_4 & = v_1 + v_2 & = 10.693 \\
v_5 & = v_4 - v_3 & = 11.652 \\
\hline
y & = v_5 & = 11.652
\end{array}$$

Reverse Adjoint Trace

$$\begin{array}{lll}
\bar{x}_1 & = \bar{v}_{-1} & = 5.5 \\
\bar{x}_2 & = \bar{v}_0 & = 1.716 \\
\hline
\bar{v}_{-1} & = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} & = 5.5 \\
\bar{v}_0 & = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} & = 1.716 \\
\bar{v}_{-1} & = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} & = 5 \\
\bar{v}_0 & = \bar{v}_3 \frac{\partial v_3}{\partial v_0} & = -0.284 \\
\bar{v}_2 & = \bar{v}_4 \frac{\partial v_4}{\partial v_2} & = 1 \\
\bar{v}_1 & = \bar{v}_4 \frac{\partial v_4}{\partial v_1} & = 1 \\
\bar{v}_3 & = \bar{v}_5 \frac{\partial v_5}{\partial v_3} & = -1 \\
\bar{v}_4 & = \bar{v}_5 \frac{\partial v_5}{\partial v_4} & = 1 \\
\hline
\bar{v}_5 & = \bar{y} & = 1
\end{array}$$

Computational

- For $f : R^n \rightarrow R$, one application of reverse mode is sufficient to calculate the full gradient.

$$\nabla f = \left(\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n} \right)$$

- Machine learning principally involves gradient of a scalar-valued objective w.r.t a large number of parameters, making reverse AD the mainstay technique in form of back propagation.
- In general, for $f : R^n \rightarrow R^m$, if operation count to evaluate original function is $ops(f)$, the time taken to calculate an $m \times n$ Jacobian for forward mode is $n\ c\ ops(f)$, whereas reverse AD takes $m\ c\ ops(f)$ (Typically, $n \gg m$).
- However, the advantage comes at the cost of increased storage.