Lecture 8: Introduction to Deep Learning: Part 2
(More on backpropagation, and ConvNets)

Sanjeev Arora       Elad Hazan
Recap: Structure of a deep net

- "Circuit" of gates connected by wires.
- Each wire has a weight on it.
- Each gate computes a simple nonlinear function, which is applied to weighted sum of incoming signals.

Each gate first computes weighted sum of incoming signals, then applies nonlinear function on it.
Basic structure of a deep net (contd)

- "Circuit" of gates connected by wires.
- Each wire has a weight on it.
- Each gate computes a simple nonlinear function, which is applied to weighted sum of incoming signals.

More popular nonlinearities:
- Rectifier Linear Unit ("RELU").
- Sigmoid (soft threshold)
The optimization problem

- **N inputs** $x_1, x_2, \ldots, x_N$ in $\mathbb{R}^d$, labeled with values $y_1, y_2, \ldots, y_N$ in $\{0,1\}$
- Experimenter decides on # of layers, # of nodes in each, and the nonlinearity type.
- $(W, A) = \text{Vector of unknowns.}$
  (Weight of each wire, and “bias” of each node.)

$f_{W,A}(x) = \text{output of this net on input } x.$

$$\text{Minimize over } (W, A): \sum_i (f_{W,A}(x_i) - y_i)^2 + \text{Regularizer}(W, A)$$

Typical choice of regularizer = sum of squares of entries of $W.$
Minimize over \((W, A)\):

\[
\text{OBJECTIVE}(W, A) = \sum_i (f_{W,A}(x_i) - y_i)^2
\]
Gradient calculation as message passing

Imagine: On each node, one little green man doing some computation.

Desired: At the end, each edge knows \( \frac{\partial f}{\partial w} \) where \( w \) is its weight, and \( f \) is the function at the last layer.

Goal:

Work per node = \( \mathcal{O}(\text{# of adjacent edges}) \).

\( \Rightarrow \) Total work by all green men = \( \mathcal{O}(\text{Network Size}) \).

(NB: Green men = Inner loop of some program)
Simple inductive algorithm to compute $\frac{\partial y}{\partial w_1}$ for all nodes $y$ in the network.

Work per node = $O(\# \text{ incoming wires})$; 
$\Rightarrow$ Total work is $O(\text{network size})$.

Repeat for all $w_1 \Rightarrow$ Overall work becomes $O((\text{network size})^2)$

Pattern of operations is identical for different $w_i$'s: consolidate! (i.e., dynamic programming)

Main idea: Message passing (each message is a real #)
Backprop Algorithm:

Message from $v_1$ to $u_1 = \text{Sum of all its incoming messages } \times \frac{\partial v_1}{\partial u_1}$

(NB: Amount of work at $v_1 = O(\# \text{ of nodes it is adjacent to})$)

Green man at $v_1$ can compute this locally.

**Backprop. Lemma:** This rule satisfies for any node $v_i$:

$$\text{Sum of messages received by } v_i = \frac{\partial f}{\partial v_i}$$

Next few slides:
Proof of this claim: By induction on (#layers –k) (NOT k)
Main ingredient: "Network Chain Rule"

Node $f$ depends on $u$ only through $v_1$, $v_2$, ..., $v_m$.

Circuit

Every path from $u$ to $f$ passes through one of the $v_i$'s.

$$\frac{\partial f}{\partial u} = \sum_{i=1}^{m} \frac{\partial f}{\partial v_i} \frac{\partial v_i}{\partial u}$$
Proof of "Network Chain Rule"

Node $f$ depends on $u$ only through $v_1, v_2, ..., v_m$.

\[
\frac{\partial f}{\partial u} = \sum_{i=1}^{m} \frac{\partial f}{\partial v_i} \frac{\partial v_i}{\partial u}
\]

Thought experiment:

\[
u \leftarrow u + \Delta u
\]

Then for each $i$:

\[
v_i \leftarrow v_i + \Delta u \cdot \frac{\partial v_i}{\partial u}
\]

So

\[
f \leftarrow f + \sum_i \frac{\partial f}{\partial v_i} \frac{\partial v_i}{\partial u} \cdot \Delta u
\]

QED
Backprop Lemma (proof by induction)

Node $f$ depends on $u$ only through $v_1, v_2, ..., v_m$

$\nabla_f = \sum_{i=1}^{m} \frac{\partial f}{\partial v_i} \frac{\partial v_i}{\partial u}$

Inductive claim for level $k$

Sum of messages received by $v_i = \frac{\partial f}{\partial v_i}$

(Sum of all incoming messages at $v_i) \times \frac{\partial v_i}{\partial u_1} = \frac{\partial f}{\partial v_i} \frac{\partial v_i}{\partial u_1}$

Claim is true for level $k-1$!
How to connect to training objective (notes from tablet)

\[ \text{Loss} = \sum_i \text{Loss} (x_i, w, A). \]

\[ \nabla \text{Loss} = \sum_i \nabla \text{Loss} (x_i, w, A). \]

\[ \text{SGD} \Rightarrow \text{Pick random } i, \text{ and compute gradient of Loss} (x_i, w, A). \]

\[ \text{Via Back-Prop!} \]

\[ u \quad \text{Ow}\]

\[ x_i \]

\[ g(w, A) \]

\[ \frac{\partial \text{Loss}(x_i, w, A)}{\partial f} \]

\[ \text{Treat this as output!} \]

\[ \text{message} = \frac{\partial g}{\partial f} \]
(more tablet notes: how to get partial derivative wrt network parameters)

We described backprop as producing \( \frac{\partial g}{\partial v} \).

To get \( \frac{\partial g}{\partial w} \), use

\[
\frac{\partial g}{\partial w} = \frac{\partial g}{\partial v} \cdot \frac{\partial v}{\partial w},
\]

\[
\frac{\partial v}{\partial w} = \delta v
\]

NB: These are computed using value of these nodes right now (computed by forward pass).

Can similarly compute partial derivative wrt bias parameter.
Some Implementation details

Minimize over \((W, A)\):
\[
\sum_i (f_{W, A}(x_i) - y_i)^2 + \text{Regularizer}(W, A)
\]

- Recall SGD: use random index \(i\) and use that \(x_i\) to compute estimate of gradient.
- Better: In each iteration, estimate gradient using random sample of \(d\) inputs. “Batch” (typically power of 2 in [16, 256]).

Motivations:
- (Theoretical) Avg. of a few samples gives more accurate sample of gradient (lower variance).
- (Practical) On many architectures ---eg GPUs--- doing \(d=2^k\) identical operations is not much more expensive than a single one.

- Let the “learning rate” drop a bit each iteration.

\(w \leftarrow w - \nu \cdot \nabla F\) \((F \text{ is objective})\)
The output layer

• If desired output is 0/1 then use sigmoid gate at the output

• If output is in \{1, 2, 3, \ldots, m\} (i.e., multiclass classification) then use \(m\) sigmoids at the output layer.

Now output of deep net = \(f_{W,A}(x) = \text{a vector in } [0,1]^m\)

Possible training objective:

\[ \sum_i (f_{W,A}(x_i) - y_i)^2 \]

Where \(y_i\) is a unary representation of output. (Number \(i\) is represented as \(000010...000\))

Better:

\[ \sum_i \text{cross-entropy}(f_{W,A}(x_i), y_i) \]

\[ \text{cross-entropy}(z, y) = \sum_j (y_j \ln z_j + (1 - y_j) \ln(1 - z_j)) \]

\(y\) is 0/1 vector; \(z\) is real-valued
Convolutional Neural Nets (aka “Convnet”)

----- useful in image recognition, language models, etc.

[LeCun et al’98]

Generic way to reduce # of parameters in the neural net
(leverages special structure of images, text etc.)
(Motivated by neuroscience studies of Visual Cortex V1)

Key component in nearly all successful deep learning in recent years.
Sanity Check: Why is it good to reduce # of network parameters?

Better generalization! (Training uses fewer samples)

Ptolemaic model of solar system (with complicated “epicycles”)

Correct model with Sun at center; planets in elliptical orbits. [Copernicus, Kepler]

MORAL: WITH SUFFICIENT # PARAMETERS, INCORRECT MODEL FITS DATA TOO.
Main Idea in Convolution Net: A Local Filter/Feature

“Multiply each pixel value by -4 and add to it values of neighboring pixels.”

Nonzeroes wherever neighboring pixels have very different values (“Edge Detector”)

Many other useful filters were designed (“AI by introspection”)

ConvNets try to learn filters from data directly. (Above filter is 3x3 matrix; only 9 parameters!)
The philosophy

• A layer consists of M types of filters, each of which is k x k (e.g., k = 5) Filter is applied in every k x k window. (Sometimes, every 2\textsuperscript{nd} or every 3\textsuperscript{rd} window; determined by “stride” parameter.)

• Inputs to each layer are outputs of filters of prev. layer.

• And so on...

At the end of the day, it is just a deep net with special connection structure; trained using backprop.

Convolution v. fast on GPUs.
Convolution: two dimensional case

**Input**

\[
\begin{array}{cccc}
    a & b & c & d \\
    e & f & g & h \\
    i & j & k & l \\
\end{array}
\]

**Kernel/filter**

\[
\begin{array}{cc}
    w & x \\
    y & z \\
\end{array}
\]

**Feature map**

Obtained by sliding same filter across all 2x2 “windows”

we + fx + iy + jz  
wa + bx + ey + fz  
bw + cx + fy + gz

(Stride =2 ➔ Apply filter every 2\textsuperscript{nd} pixel.)
Apply to all \( k \times k \) “windows”.

“Max Pooling”

Divide into \( 2 \times 2 \) windows; replace each by \text{max} of its 4 values.

Shrink feature map by \( 4 \times 4 \).

Same architecture at all layers. (Sometimes throw in some fully connected layers at the top.)
Clarification on how to use backprop to train convnets

Main Issue: Filter parameter appears many times in the network.

This formula shows how to “pool” the gradient from the multiple occurrences of this parameter.
Going further in deep nets.

• Mechanisms to allow many layers; even 100+. (Gradient gets noisier as it is backpropagated through more layers!)

• Modifications of gradient descent that allow deep nets with feedback connections (output of higher layer feeding into lower layers) “Recurrent Neural Nets”

• Deep nets with memory: e.g. Long-Short Term Memory nets (LSTMs)

• Ways to compose different deep nets automatically; use backprop to propagate gradient across the interface.

• Practical tools such as autograd, tensorflow, Caffe,..

(Look for ugrad deep nets course in spring.)