Back-Propagation

Lecture 0 2

Yann Le Cun
Facebook AI Research,
Center for Data Science, NYU
Courant Institute of Mathematical Sciences, NYU
http://yann.lecun.com
Discovering the Hidden Structure in High-Dimensional Data

The manifold hypothesis

Learning Representations of Data:

- **Discovering & disentangling the independent explanatory factors**

The Manifold Hypothesis:

- Natural data lives in a low-dimensional (non-linear) manifold
- Because variables in natural data
Example: all face images of a person
- 1000x1000 pixels = 1,000,000 dimensions
- But the face has 3 cartesian coordinates and 3 Euler angles
- And humans have less than about 50 muscles in the face
- Hence the manifold of face images for a person has <56 dimensions

The perfect representations of a face image:
- Its coordinates on the face manifold
- Its coordinates away from the manifold

We do not have good and general methods to learn functions that turns an image into this kind of representation
Disentangling factors of variation

The Ideal Disentangling Feature Extractor

Pixel n → Ideal Feature Extractor → Pixel 2 → Expression → View
Azimuth-Elevation manifold. Ignores lighting. [Hadsell et al. CVPR 2006]
**Basic Idea for Invariant Feature Learning**

- Embed the input non-linearly into a high(er) dimensional space
  - In the new space, things that were non separable may become separable

- Pool regions of the new space together
  - Bringing together things that are semantically similar. Like pooling.

```
Input  -->  Non-Linear Function  -->  Pooling Or Aggregation  -->  Stable/invariant features
        ^ high-dim Unstable/non-smooth features
```
Non-Linear Expansion $\rightarrow$ Pooling

Entangled data manifolds

Non-Linear Dim Expansion, Disentangling

Pooling, Aggregation
Use clustering to break things apart, pool together similar things

Clustering, Quantization, Sparse Coding → Pooling, Aggregation
Overall Architecture:
Normalization → Filter Bank → Non-Linearity → Pooling

Stacking multiple stages of
[Normalization → Filter Bank → Non-Linearity → Pooling].

Normalization: variations on whitening
- Subtractive: average removal, high pass filtering
- Divisive: local contrast normalization, variance normalization

Filter Bank: dimension expansion, projection on overcomplete basis

Non-Linearity: sparsification, saturation, lateral inhibition....
- Rectification (ReLU), Component-wise shrinkage, tanh, winner-takes-all

Pooling: aggregation over space or feature type
- \(X_i\); \(L_p: \sqrt[p]{X_i^p}\); \(PROB: \frac{1}{b} \log \left( \sum_i e^{bX_i} \right)\)
Deep Supervised Learning
(modular approach)
Complex learning machines can be built by assembling modules into networks.

Simple example: sequential/layered feed-forward architecture (cascade).

Forward Propagation:

Let $X = X_0$,

$$X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n]$$

$$E(Y, X, W) = C(X_n, Y)$$
Each module is an object
- Contains trainable parameters
- Inputs are arguments
- Output is returned, but also stored internally
- Example: 2 modules m1, m2

**Torch7 (by hand)**
- hid = m1:forward(in)
- out = m2:forward(hid)

**Torch7 (using the nn.Sequential class)**
- model = nn.Sequential()
- model:add(m1)
- model:add(m2)
- out = model:forward(in)
Computing the Gradient in Multi-Layer Systems

To train a multi-module system, we must compute the gradient of $E(W, Y, X)$ with respect to all the parameters in the system (all the $W_k$).

Let’s consider module $i$ whose fprop method computes $X_k = F_k(X_{k-1}, W_k)$.

Let’s assume that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector $X_k$ we know how much $E$ would wiggle if we wiggled that component of $X_k$. 
Computing the Gradient in Multi-Layer Systems

- We can apply chain rule to compute \( \frac{\partial E}{\partial W_k} \) (how much \( E \) would wiggle if we wiggled each component of \( W_k \)):

\[
\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}
\]

\[
[1 \times N_w] = [1 \times N_x].[N_x \times N_w]
\]

- \( \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k} \) is the Jacobian matrix of \( F_k \) with respect to \( W_k \).

\[
\left[ \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k} \right]_{pq} = \frac{\partial [F_k(X_{k-1}, W_k)]_p}{\partial [W_k]_q}
\]

- Element \((p, q)\) of the Jacobian indicates how much the \( p \)-th output wiggles when we wiggle the \( q \)-th weight.
Computing the Gradient in Multi-Layer Systems

Using the same trick, we can compute $\frac{\partial E}{\partial X_{k-1}}$. Let’s assume again that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector $X_k$ we know how much $E$ would wiggle if we wiggled that component of $X_k$.

- We can apply chain rule to compute $\frac{\partial E}{\partial X_{k-1}}$ (how much $E$ would wiggle if we wiggled each component of $X_{k-1}$):

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- $\frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$ is the Jacobian matrix of $F_k$ with respect to $X_{k-1}$.

- $F_k$ has two Jacobian matrices, because it has to arguments.

- Element $(p, q)$ of this Jacobian indicates how much the $p$-th output wiggles when we wiggle the $q$-th input.

- The equation above is a recurrence equation!
Jacobians and Dimensions

- derivatives with respect to a column vector are line vectors (dimensions: $[1 \times N_{k-1}] = [1 \times N_k] \ast [N_k \times N_{k-1}]$)

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- (dimensions: $[1 \times N_{wk}] = [1 \times N_k] \ast [N_k \times N_{wk}]$):

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W}$$

- we may prefer to write those equation with column vectors:

$$\frac{\partial E'}{\partial X_{k-1}}' = \frac{\partial F_k(X_{k-1}, W_k)'}{\partial X_{k-1}} \frac{\partial E'}{\partial X_k}$$

$$\frac{\partial E'}{\partial W_k} = \frac{\partial F_k(X_{k-1}, W_k)'}{\partial W} \frac{\partial E'}{\partial X_k}$$
To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for $\frac{\partial E}{\partial X_k}$:

\[
\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n,Y)}{\partial X_n}
\]

\[
\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_{n}(X_{n-1},W_n)}{\partial X_{n-1}}
\]

\[
\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_{n}(X_{n-1},W_n)}{\partial W_n}
\]

\[
\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2},W_{n-1})}{\partial X_{n-2}}
\]

\[
\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2},W_{n-1})}{\partial W_{n-1}}
\]

... etc, until we reach the first module.

we now have all the $\frac{\partial E}{\partial W_k}$ for $k \in [1, n]$. 
Computing Gradients by Back-Propagation

\[ C(X, Y, \Theta) \]

- **Cost**
  - \( W_n \) \( dC/dW_n \)
  - \( dC/dX_i \) \( X_i \)
  - \( W_i \) \( dC/dW_i \)
  - \( dC/dX_i-1 \) \( X_{i-1} \)

- **A practical Application of Chain Rule**
  - Backprop for the state gradients:
    - \( dC/dX_i-1 = dC/dX_i \cdot dX_i/dX_i-1 \)
    - \( dC/dX_i-1 = dC/dX_i \cdot dF_i(X_{i-1}, W_i)/dX_i-1 \)
  - Backprop for the weight gradients:
    - \( dC/dW_i = dC/dX_i \cdot dX_i/dW_i \)
    - \( dC/dW_i = dC/dX_i \cdot dF_i(X_{i-1}, W_i)/dW_i \)

\[ F_n(X_{n-1}, W_n) \]

\[ F_i(X_{i-1}, W_i) \]

\[ F_1(X_0, W_1) \]

\( X \) (input) \( Y \) (desired output)
Complex learning machines can be built by assembling modules into networks.

- **Linear Module**
  - $\text{Out} = W \cdot \text{In} + B$

- **ReLU Module (Rectified Linear Unit)**
  - $\text{Out}_i = 0 \text{ if } \text{In}_i < 0$
  - $\text{Out}_i = \text{In}_i \text{ otherwise}$

- **Cost Module: Squared Distance**
  - $C = \|\|\text{In}_1 - \text{In}_2\|\|^2$

- **Objective Function**
  - $L(\Theta) = \frac{1}{p} \sum_k C(X^k, Y^k, \Theta)$
  - $\Theta = (W_1, B_1, W_2, B_2, W_3, B_3)$
Building a Network by Assembling Modules

- All major deep learning frameworks use modules (inspired by SN/Lush, 1991)
  - Torch7, Theano, TensorFlow...

\[
C(X,Y,\Theta) \quad \text{NegativeLogLikelihood}
\]

```
-- sizes
ninput = 28*28  -- e.g. for MNIST
nhidden1 = 1000
noutput = 10

-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax())

-- cost module
cost = nn.ClassNLLCriterion()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)
```
- Torch7 example
- Gradtheta contains the gradient

\[ C(X, Y, \Theta) \]

```
net = nn.Sequential()
net.add(nn.Linear(ninput, nhidden))
net.add(nn.ReLU())
net.add(nn.Linear(nhidden, noutput))
net.add(nn.LogSoftMax())

-- cost module
cost = nn.ClassNLLCriterion()

-- gather the parameters in a vector
theta, gradtheta = net:getParameters()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)

-- run backprop
gradtheta:zero()
gradoutput = cost:backward(output, target)
net:backward(input, gradoutput)
```
### Module Classes

**Linear**
- \( Y = W.X \); \( \frac{dC}{dX} = W^T \cdot \frac{dC}{dY} \); \( \frac{dC}{dW} = \frac{dC}{dY} \cdot X^T \)

**ReLU**
- \( y = \text{ReLU}(x) \); if \( x < 0 \) \( \frac{dC}{dx} = 0 \) else \( \frac{dC}{dx} = \frac{dC}{dy} \)

**Duplicate**
- \( Y_1 = X, Y_2 = X \); \( \frac{dC}{dX} = \frac{dC}{dY_1} + \frac{dC}{dY_2} \)

**Add**
- \( Y = X_1 + X_2 \); \( \frac{dC}{dX_1} = \frac{dC}{dY} \); \( \frac{dC}{dX_2} = \frac{dC}{dY} \)

**Max**
- \( y = \max(x_1, x_2) \); if \( x_1 > x_2 \) \( \frac{dC}{dx_1} = \frac{dC}{dy} \) else \( \frac{dC}{dx_1} = 0 \)

**LogSoftMax**
- \( Y_i = X_i - \log \left[ \sum_j \exp(X_j) \right] \); .....
<table>
<thead>
<tr>
<th>Module Class</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$Y = W \cdot X$ ; $\frac{dC}{dX} = W^T \cdot \frac{dC}{dy}$ ; $\frac{dC}{dW} = \frac{dC}{dy} \cdot X^T$</td>
</tr>
<tr>
<td>ReLU</td>
<td>$y = \text{ReLU}(x)$ ; if $(x&lt;0)$ $\frac{dC}{dx} = 0$ else $\frac{dC}{dx} = \frac{dC}{dy}$</td>
</tr>
<tr>
<td>Duplicate</td>
<td>$Y_1 = X$, $Y_2 = X$ ; $\frac{dC}{dX} = \frac{dC}{dY_1} + \frac{dC}{dY_2}$</td>
</tr>
<tr>
<td>Add</td>
<td>$Y = X_1 + X_2$ ; $\frac{dC}{dX_1} = \frac{dC}{dy}$ ; $\frac{dC}{dX_2} = \frac{dC}{dy}$</td>
</tr>
<tr>
<td>Max</td>
<td>$y = \text{max}(x_1, x_2)$ ; if $(x_1 &gt; x_2)$ $\frac{dC}{dx_1} = \frac{dC}{dy}$ else $\frac{dC}{dx_1} = 0$</td>
</tr>
<tr>
<td>LogSoftMax</td>
<td>$Y_i = X_i - \log\left(\sum_j \exp(X_j)\right)$ ; .....</td>
</tr>
</tbody>
</table>
the internal state of the network will be kept in a “state” class that contains two scalars, vectors, or matrices: (1) the state proper, (2) the derivative of the energy with respect to that state.
Linear Module

- fprop: $X_{out} = WX_{in}$
- bprop to input:
  $$\frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} W$$
- by transposing, we get column vectors:
  $$\frac{\partial E}{\partial X_{in}}' = W' \frac{\partial E}{\partial X_{out}}'$$
- bprop to weights:
  $$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{outi}} \frac{\partial X_{outi}}{\partial W_{ij}} = X_{inj} \frac{\partial E}{\partial X_{outi}}$$
- We can write this as an outer-product:
  $$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial X_{out}}' X_{in}'$$
Tanh module (or any other pointwise function)

- $f_{\text{prop}}$: $(X_{\text{out}})_i = \tanh((X_{\text{in}})_i + B_i)$
- $b_{\text{prop}}$ to input:
  \[ (\frac{\partial E}{\partial X_{\text{in}}})_i = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'( (X_{\text{in}})_i + B_i ) \]
- $b_{\text{prop}}$ to bias:
  \[ \frac{\partial E}{\partial B_i} = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'( (X_{\text{in}})_i + B_i ) \]
- $\tanh(x) = \frac{2}{1+\exp(-x)} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}$
Euclidean Distance Module (Squared Error)

- fprop: $X_{out} = \frac{1}{2} \| X_{in} - Y \|^2$
- bprop to $X$ input: $\frac{\partial E}{\partial X_{in}} = X_{in} - Y$
- bprop to $Y$ input: $\frac{\partial E}{\partial Y} = Y - X_{in}$
- The PLUS module: a module with $K$ inputs $X_1, \ldots, X_K$ (of any type) that computes the sum of its inputs:

$$X_{out} = \sum_k X_k$$

back-prop: $\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{out}} \quad \forall k$

- The BRANCH module: a module with one input and $K$ outputs $X_1, \ldots, X_K$ (of any type) that simply copies its input on its outputs:

$$X_k = X_{in} \quad \forall k \in [1..K]$$

back-prop: $\frac{\partial E}{\partial in} = \sum_k \frac{\partial E}{\partial X_k}$
A module with $K$ inputs $X_1, \ldots, X_K$ (of any type) and one additional discrete-valued input $Y$.

The value of the discrete input determines which of the $N$ inputs is copied to the output.

\[ X_{\text{out}} = \sum_k \delta(Y - k)X_k \]

\[ \frac{\partial E}{\partial X_k} = \delta(Y - k) \frac{\partial E}{\partial X_{\text{out}}} \]

the gradient with respect to the output is copied to the gradient with respect to the switched-in input. The gradients of all other inputs are zero.
SoftMax Module (should really be called SoftArgMax)

- Transforms scores into a discrete probability distribution
  - Positive numbers that sum to one.

- Used in multi-class classification

\[
p_k = \frac{e^{\beta x_k}}{\sum_j e^{\beta x_j}}
\]

\[
y_k = \frac{\text{softmax}(x_k)}{\sum_j \text{softmax}(x_j)}
\]
SoftMax Module: Loss Function for Classification

- **-LogSoftMax:**
  \[- \frac{1}{p} \log p_h = -x_h + \frac{1}{p} \log \sum_j e^{\beta x_j}\]

- Maximum conditional likelihood
- Minimize -log of the probability of the correct class.

Network

Switch

SoftMax

-Log

X

Y=3
LogSoftMax Module

- Transforms scores into a discrete probability distribution
- LogSoftMax = Identity - LogSumExp

\[ y_k = x_k - \frac{1}{p} \log \sum_j e^{\beta x_j} \]
LogSumExp Module

- Log of normalization term for SoftMax

\[ X_{out} = \frac{1}{\beta} \sum_j e^{\beta x_j} \]

- Fprop

\[ \frac{\partial E}{\partial x_k} = \frac{\partial E}{\partial x_{out}} \cdot \frac{e^{\beta x_k}}{\sum_j e^{\beta x_j}} \]

- Bprop

\[ \frac{\partial E}{\partial x_k} = \frac{\partial E}{\partial x_{out}} \cdot P_k \]

\[ P_k = \frac{e^{\beta x_k}}{\sum_j e^{\beta x_j}} \]
Backprop works through any modular architecture

- **Any connection is permissible**
  - Networks with loops must be “unfolded in time”.

- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.
Use ReLU non-linearities (tanh and logistic are falling out of favor)

Use cross-entropy loss for classification

Use Stochastic Gradient Descent on minibatches

Shuffle the training samples

Normalize the input variables (zero mean, unit variance)

Schedule to decrease the learning rate

Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs

Use “dropout” for regularization

Lots more in [LeCun et al. “Efficient Backprop” 1998]

Example: building a Neural Net in Torch7

Net for SVHN digit recognition
10 categories
Input is 32x32 RGB (3 channels)

1500 hidden units

Creating a 2-layer net
Make a cascade module
Reshape input to vector
Add Linear module
Add tanh module
Add Linear Module
Add log softmax layer
Create loss function module

Noutputs = 10;
nfeats = 3; Width = 32; height = 32
ninputs = nfeats*width*height
nhiddens = 1500

-- Simple 2-layer neural network
model = nn.Sequential()
model:add(nn.Reshape(ninputs))
model:add(nn.Linear(ninputs,nhiddens))
model:add(nn.Tanh())
model:add(nn.Linear(nhiddens,noutputs))
model:add(nn.LogSoftMax())
criterion = nn.ClassNLLCriterion()
Example: Training a Neural Net in Torch7

```plaintext
for t = 1, trainData:size(), batchSize do
  inputs, outputs = getNextBatch()
  local feval = function(x)
    parameters:copy(x)
    gradParameters:zero()
    local f = 0
    for i = 1, #inputs do
      local output = model:forward(inputs[i])
      local err = criterion:forward(output, targets[i])
      f = f + err
      local df_do = criterion:backward(output, targets[i])
      model:backward(inputs[i], df_do)
    end
    gradParameters:div(#inputs)
    f = f/#inputs
    return f, gradParameters
  end  -- of feval
  optim.sgd(feval, parameters, optimState)
end
```
Torch7 is based on the Lua language
- Simple and lightweight scripting language, dominant in the game industry
- Has a native just-in-time compiler (fast!)
- Has a simple foreign function interface to call C/C++ from Lua

Torch7 is an extension of Lua with
- A multidimensional array engine with CUDA and OpenMP backends
- A machine learning library that implements multilayer nets, convolutional nets, unsupervised pre-training, etc
- Various libraries for data/image manipulation and computer vision
- A quickly growing community of users
- Used by Facebook, DeepMind, Twitter, NYU, lots of startups
- Supports NVIDIA's cuDNN library.

Torch Resource
- Main website: http://torch.ch
- Cheatsheet: https://github.com/torch/torch7/wiki/Cheatsheet
- Learn Lua in 15 minutes: http://tylerneylon.com/a/learn-lua/