Neural Networks

Lecture 6

Rob Fergus
Overview

- Individual neuron
- Non-linearities (RELU, tanh, sigmoid)
- Single layer model
- Multiple layer models
- Theoretical discussion: representational power
- Examples shown decision surface for 1,2,3-layer nets
- Training models
- Backprop
- Example modules
- Special layers
- Practical training tips
- Setting learning rate
- Debugging training
- Regularization
Additional Readings
Useful books and articles

- Neural Networks for Pattern Recognition, Christopher M. Bishop, Oxford University Press 1995.
  - Red/Green cover, NOT newer book with yellow/beige cover.
- Andrej Karpathy’s CS231n Stanford Course on Neural Nets
  http://cs231n.github.io/
- Yann LeCun’s NYU Deep Learning course
Neural nets composed of layers of artificial *neurons*.

- Each layer computes some function of layer beneath.
- Inputs mapped in *feed-forward* fashion to output.
- Consider only feed-forward neural models at the moment, i.e. no cycles.
Neural nets are an example of connectionism. Connectionism [Hebb 1940s] argues that complex behaviors arise from interconnected networks of simple units. As opposed to formal operations on symbols (computationalism).

Early work in 1940’s and 1950’s by Hebb, McCulloch and Pitts on artificial neurons.


Perceptron book [Minsky and Pappert 1969]. Showed limitations of single layer models (e.g. cannot solve XOR).
Historical Overview
More recent history

- Neocognitron [Fukushima 1980]. Proto-ConvNet, inspired by [Hubel & Weisel 1959].
- Convolutional Networks [LeCun & others 1989].
- Bigger datasets, e.g. [ImageNet 2009]
- Neural Nets applied to speech [Hinton’s group 2011].
- ConvNets applied to ImageNet Challenge 2012 [Krizhevsky, Sutskever & Hinton NIPS 2012]
- Last few years, improved ConvNet architectures. Closing on human performance.
An Individual Neuron
Also known as a unit

- Input: \( x \) (\( n \times 1 \) vector)
- Parameters: weights \( w \) (\( n \times 1 \) vector), bias \( b \) (scalar)
- Activation: \( a = \sum_{i=1}^{n} x_i w_i + b \).
  Note \( a \) is a scalar.
  Multiplicative interaction between weights and input.
- Point-wise non-linear function: \( \sigma(.) \), e.g. \( \sigma(.) = \text{tanh}(.). \)
- Output:
  \( y = f(a) = \sigma(\sum_{i=1}^{n} x_i w_i + b) \)
  Can think of bias as weight \( w_0 \), connected to constant input 1:
  \( y = f(\tilde{w}^T[1, x]) \).
Single Layer Network

Multiple outputs

- Input: \( x \) (\( n \times 1 \) vector)
- \( m \) neurons
- Parameters:
  - weight matrix \( W \) (\( n \times m \))
  - bias vector \( b \) (\( m \times 1 \))
- Non-linear function \( \sigma(\cdot) \)
- Output: \( y = \sigma(Wx + b) \) (\( m \times 1 \))
Non-linearities: Sigmoid

- $\sigma(z) = \frac{1}{1+e^{-z}}$
- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +ve,-ve inputs
- Gradients go to zero
- Outputs centered at 0.5 (poor conditioning)
- Not used in practice

$y = \text{sigmoid}(x)$
Non-linearities: Tanh

- $\sigma(z) = \tanh(z)$
- Bounded in $[+1,-1]$ range
- Saturation for large +ve, -ve inputs
- Outputs centered at zero
- Preferable to sigmoid
Non-linearities: Rectified Linear (ReLU)

- \( \sigma(z) = \max(z, 0) \)
- Unbounded output (on positive side)
- Efficient to implement: 
  \[ \frac{d\sigma(z)}{dz} = \{0, 1\} \]
- Also seems to help convergence (see 6x speedup vs tanh in Krizhevsky et al.)
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models.
Non-linearities: Leaky RELU

- Leaky Rectified Linear
  \[ \sigma(z) = 1[z > 0] \max(0, x) + 1[z < 0] \max(0, \alpha z) \]
- where \( \alpha \) is small, e.g. 0.02
- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- \( \alpha \) can also be learned (see Kaiming He et al. 2015).
Multiple Layers
A bit more information about this

- Neural networks is composed of multiple layers of neurons.
- Acyclic structure. Basic model assumes full connections between layers.
- Layers between input and output are called *hidden*.
- Various names used:
  - Artificial Neural Nets (ANN)
  - Multi-layer Perceptron (MLP)
  - Fully-connected network
- Neurons typically called *units*.
By convention, number of layers is hidden + output (i.e. does not include input).
So 3-layer model has 2 hidden layers.
Parameters: weight matrices $W^1, W^2, W^3$ and bias vectors $b^1, b^2, b^3$. 

Architecture Selection for MLPs
How to pick number of layers and units/layer

- No good answer:
  - Problem has now shifted from picking good features to picking good architectures.
  - (Non-answer) Pick using validation set.
  - Hyper-parameter optimization [e.g. Snoek 2012 https://arxiv.org/pdf/1206.2944].
  - Active area of research.

- For fully connected models, 2 or 3 layers seems the most that can be effectively trained (more later).

- Regarding number of units/layer:
  - Parameters grows with (units/layer)$^2$.
  - With large units/layer, can easily overfit.
  - For classification, helps to expand towards output.
TOY EXAMPLE: SYNTHETIC DATA

1 input & 1 output
100 hidden units in each layer
TOY EXAMPLE: SYNTHETIC DATA

1 input & 1 output
3 hidden layers

10 hiddens
100 hiddens
1000 hiddens

output

input

Ranzato
Representational Power

What functions can you represent with an MLP?

- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent any function. Assuming non-trivial non-linearity.
  - Bengio 2009,
  - Bengio, Courville, Goodfellow book
    http://www.deeplearningbook.org/contents/mlp.html
  - Simple proof by M. Neilsen

- But issue is efficiency: very wide two layers vs narrow deep model?
- In practice, more layers helps.
- But beyond 3, 4 layers no improvement for fully connected layers.
Training a model: Overview

How to set the parameters

- Given dataset \( \{x, y\} \), pick appropriate cost function \( C \).
- Forward-pass (f-prop) examples through the model to get predictions.
- Get error using cost function \( C \) to compare prediction to targets \( y \).
- Use back-propagation (b-prop) to pass error back through model, adjusting parameters to minimize loss/energy \( E \).
- Back-propagation is essentially chain rule of derivatives back through the model.
- Each layer is differentiable w.r.t. to parameters and input.
- Once gradients obtained, use Stochastic Gradient Descent (SGD) to update weights.
Stochastic Gradient Descent

- Want to minimize overall loss function $E$.
- Loss is sum of individual losses over each example.
- In gradient descent, we start with some initial set of parameters $\theta^0$.
- Update parameters: $\theta^{k+1} \leftarrow \theta^k + \eta \nabla \theta$.
  - $k$ is iteration index, $\eta$ is learning rate (scalar; set semi-manually).
- Gradients $\nabla \theta = \frac{\partial E}{\partial \theta}$ computed by b-prop.
- In *Stochastic* gradient descent, compute gradient on sub-set (batch) of data.
- If batchsize=1 then $\theta$ is updated after each example.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).
Computing Gradients in a multi-stage architecture

Forward Pass

- Consider model with $N$ layers. Layer $i$ has vector of weights $W_i$.
- F-Prop (in red) takes input $x$ and passes it through each layer $F_i$: $x_i = F_i(x_{i-1}, W_i)$
- Output of each layer $x_i$; prediction $x_n$ is output of top layer.
- Cost function $C$ compares $x_n$ to $y$.
- Overall energy $E = \sum_{m=1}^{M} C(x_n^m, y^m)$, i.e. sum over all examples of $C(x_n, y)$.

[Figure: Y. LeCun and M. Ranzato]
Computing gradients

To compute the gradients, we could start by wringing the full energy $E$ as a function of the network parameters.

$$E(\theta) = \sum_{m=1}^{M} C \left(F_n \left(F_{n-1} \left(F_2 \left(F_1 \left(x_0^m, w_1\right), w_2\right), w_{n-1}\right), w_n\right), y^m\right)$$

And then compute the partial derivatives... instead, we can use the chain rule to derive a compact algorithm: back-propagation.
Matrix calculus

- $x$ column vector of size $[n \times 1]$
  
  \[ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \]

- We now define a function on vector $x$: $y = F(x)$
- If $y$ is a scalar, then
  
  \[
  \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \ldots & \frac{\partial y}{\partial x_n} \end{bmatrix}
  \]

  The derivative of $y$ is a row vector of size $[1 \times n]$

- If $y$ is a vector $[m \times 1]$, then (Jacobian formulation):

  \[
  \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \ldots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \ldots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \ldots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}
  \]

  The derivative of $y$ is a matrix of size $[m \times n]$

  (m rows and n columns)
Matrix calculus

- If $y$ is a scalar and $x$ is a matrix of size $[n \times m]$, then

$$
\frac{\partial y}{\partial X} = \begin{bmatrix}
\frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{21}} & \ddots & \frac{\partial y}{\partial x_{n1}} \\
M & M & \ddots & M \\
\frac{\partial y}{\partial x_{1m}} & \frac{\partial y}{\partial x_{12}} & \ddots & \frac{\partial y}{\partial x_{nm}}
\end{bmatrix}
$$

The output is a matrix of size $[m \times n]$
Matrix calculus

• Chain rule:

For the function: \( z = h(x) = f(g(x)) \)

Its derivative is: \( h'(x) = f'(g(x)) \cdot g'(x) \)

and writing \( z = f(u) \), and \( u = g(x) \):

\[
\frac{dz}{dx}_{x=a} = \frac{dz}{du}_{u=g(a)} \cdot \frac{du}{dx}_{x=a}
\]

\([m \times n][m \times p][p \times n]\)

with \( p = \) length vector \( u = |u| \), \( m = |z| \), and \( n = |x| \)

Example, if \(|z| = 1\), \(|u| = 2\), \(|x| = 4\)

\( h'(x) = \)

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
Matrix calculus

- Chain rule:

For the function: $h(x) = f_n(f_{n-1}(\ldots(f_1(x)) ))$

With $u_1 = f_1(x)$
$u_i = f_i(u_{i-1})$
$z = u_n = f_n(u_{n-1})$

The derivative becomes a product of matrices:

$$\left. \frac{dz}{dx} \right|_{x=a} = \left. \frac{dz}{du_n} \right|_{u_{n-1} = f_{n-1}(u_{n-2})} \cdot \left. \frac{du_{n-1}}{du_{n-2}} \right|_{u_{n-2} = f_{n-2}(u_{n-3})} \cdot \cdots \cdot \left. \frac{du_2}{du_1} \right|_{u_1 = f_1(a)} \cdot \left. \frac{du_1}{dx} \right|_{x=a}$$

(exercise: check that all the matrix dimensions work fine)
Computing gradients

The energy $E$ is the sum of the costs associated to each training example $x^m, y^m$

$$E(\theta) = \sum_{m=1}^{M} C(x^m_n, y^m; \theta)$$

Its gradient with respect to the networks parameters is:

$$\frac{\partial E}{\partial \theta_i} = \sum_{m=1}^{M} \frac{C(x^m_n, y^m; \theta)}{\partial \theta_i}$$

is how much $E$ varies when the parameter $\theta_i$ is varied.
Computing gradients

We could write the cost function to get the gradients:

\[ C(x_n, y; \theta) = C(F_n(x_{n-1}, w_n), y) \]

with \( \theta = [w_1, w_2, \ldots, w_n] \)

If we compute the gradient with respect to the parameters of the last layer (output layer) \( w_n \), using the chain rule:

\[
\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n(x_{n-1}, w_n)}{\partial w_n}
\]

(how much the cost changes when we change \( w_n \): is the product between how much the cost changes when we change the output of the last layer and how much the output changes when we change the layer parameters.)
Computing gradients: cost layer

If we compute the gradient with respect to the parameters of the last layer (output layer) $w_n$, using the chain rule:

$$\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n(x_{n-1}, w_n)}{\partial w_n}$$

For example, for an Euclidean loss:

$$C(x_n, y) = \frac{1}{2} \|x_n - y\|^2$$

The gradient is:

$$\frac{\partial C}{\partial x_n} = x_n - y$$

Will depend on the layer structure and non-linearity.
Computing gradients: layer i

We could write the full cost function to get the gradients:

\[
C(x_n, y; \theta) = C\left(F_n\left(F_{n-1}\left(F_2\left(F_1(x_0, w_1), w_2\right), w_{n-1}\right), w_n\right), y\right)
\]

If we compute the gradient with respect to \(w_i\), using the chain rule:

\[
\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot \frac{\partial x_{i+1}}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i}
\]

\[
\frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}
\]

And this can be computed iteratively! This is easy.
Backpropagation

\[
\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot K \cdot \frac{\partial x_{i+1}}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i}
\]

If we have the value of \( \frac{\partial C}{\partial x_i} \) we can compute the gradient at the layer below as:

\[
\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_{i-1}}
\]

Gradient layer i-1

Gradient layer i

\( \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}} \)
Backpropagation: layer $i$

- Layer $i$ has two inputs (during training):
  \[ x_{i-1}, \frac{\partial C}{\partial x_i} \]

- For layer $i$, we need the derivatives:
  \[ \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}, \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i} \]

- We compute the outputs:
  \[ x_i = F_i(x_{i-1}, w_i) \]
  \[ \frac{\partial C}{\partial x_i} = \frac{\partial C}{\partial x_{i-1}} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}} \]

- The weight update equation is:
  \[ \frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i} \]

\[ w_i^{k+1} \leftarrow w_i^k + \eta \frac{\partial E}{\partial w_i} \] (sum over all training examples to get $E$)
Backpropagation: summary

• Forward pass: for each training example. Compute the outputs for all layers
  \[ x_i = F_i(x_{i-1}, w_i) \]

• Backwards pass: compute cost derivatives iteratively from top to bottom:
  \[ \frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}} \]

• Compute gradients and update weights.
Linear Module

- **Forward propagation:** $x_{out} = F(x_{in}, W) = W x_{in}$

- **Backprop to input:**

$$\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial x_{in}}$$

If we look at the $j$ component of output $x_{out}$, with respect to the $i$ component of the input, $x_{in}$:

$$\frac{\partial x_{out,i}}{\partial x_{in,j}} = W_{ij} \quad \Rightarrow \quad \frac{\partial F(x_{in}, W)}{\partial x_{in}} = W$$

Therefore:

$$\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot W$$
Linear Module

- **Forward propagation:** \( x_{out} = F(x_{in}, W) = Wx_{in} \)

- **Backprop to weights:**
  \[
  \frac{\partial C}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial W}
  \]

If we look at how the parameter \( W_{ij} \) changes the cost, only the \( i \) component of the output will change, therefore:

\[
\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{x_{out_i}}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial W_{ij}} = x_{in_j}
\]

And now we can update the weights (by summing over all the training examples):

\[
W_{ij}^{k+1} \leftarrow W_{ij}^k + \eta \frac{\partial E}{\partial W_{ij}} \quad \text{(sum over all training examples to get E)}
\]
Linear Module

\[ x_{\text{out}} = W x_{\text{in}} \]

\[ \frac{\partial C}{\partial x_{\text{in}}} = \frac{\partial C}{\partial x_{\text{out}}} \cdot W \]

\[ \frac{\partial C}{\partial W} = x_{\text{in}} \cdot \frac{\partial C}{\partial x_{\text{out}}} \]

Weight updates

\[ W^{k+1} \leftarrow W^k + \eta_t \left( \frac{\partial E}{\partial W} \right)^T \]
Pointwise function

- **Forward propagation:**
  \[ x_{out_i} = h(x_{in_i} + b_i) \]
  \( h = \) an arbitrary function, \( b_i \) is a bias term.

- **Backprop to input:**
  \[ \frac{\partial C}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i) \]

- **Backprop to bias:**
  \[ \frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i) \]

  We use this last expression to update the bias.

Some useful derivatives:

For hyperbolic tangent:
\[ \tanh'(x) = 1 - \tanh^2(x) \]

For ReLU:
\[ h(x) = \max(0,x) \quad h'(x) = 1 \text{ [}x>0\text{]} \]
Pointwise function

\[ x_{out_i} = h(x_{in_i} + b_i) \]

\[ \frac{\partial C}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i) \]

\[ \frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i) \]

Weight updates

\[ b_i^{k+1} \leftarrow b_i^k + \eta_i \frac{\partial E}{\partial b_i} \]
Euclidean cost module

\[
C = \frac{1}{2} \| x_{in} - y \|^2
\]

\[
\frac{\partial C}{\partial x_{in}} = x_{in} - y
\]

\[
\frac{\partial C}{\partial C} = 1
\]
Back propagation example

Learning rate = -0.2 (because we used positive increments)

Euclidean loss

Training data:

<table>
<thead>
<tr>
<th>input</th>
<th>desired output</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 1: 1.0</td>
<td>node 5: 0.5</td>
</tr>
<tr>
<td>node 2: 0.1</td>
<td></td>
</tr>
</tbody>
</table>

Exercise: run one iteration of back propagation
Back propagation example

After one iteration (rounding to two digits):
Toy Code: Neural Net Trainer in MATLAB

% F-PROP
for i = 1 : nr_layers - 1
    [h{i} jac{i}] = logistic(W{i} * h{i-1} + b{i});
end
h{nr_layers-1} = W{nr_layers-1} * h{nr_layers-2} + b{nr_layers-1};
prediction = softmax(h{l-1});

% CROSS ENTROPY LOSS
loss = - sum(sum(log(prediction) .* target));

% B-PROP
dh{l-1} = prediction - target;
for i = nr_layers - 1 : -1 : 1
    Wgrad{i} = dh{i} * h{i-1}';
    bgrad{i} = sum(dh{i}, 2);
    dh{i-1} = (W{i}' * dh{i}) .* jac{i-1};
end

% UPDATE
for i = 1 : nr_layers - 1
    W{i} = W{i} - (lr / batch_size) * Wgrad{i};
b{i} = b{i} - (lr / batch_size) * bgrad{i};
end
TOY EXAMPLE: SYNTHETIC DATA

1 input & 1 output
3 hidden layers, 1000 hiddens
Regression of cosine

Ranzato
TOY EXAMPLE: SYNTHETIC DATA

1 input & 1 output
3 hidden layers, 1000 hiddens
Regression of cosine
Alternate Topologies

- Models with complex graph structures can be trained by backprop.
- Each node in the graph must be differentiable w.r.t. parameters and inputs.
- If no cycles exist, then b-prop takes a single pass.
- If cycles exist, we have a recurrent network which will be discussed in subsequent lectures.

[Figure: Y. LeCun and M. Ranzato]
Plus module has $K$ inputs $x_1, \ldots, x_K$. Output is sum of inputs: $x_{out} = \sum_{k=1}^{K} x_k$

Plus B-prop: $\frac{\partial E}{\partial x_k} = \frac{\partial E}{\partial x_{out}} \forall k$

Branch module has a single input, but $K$ outputs $x_1, \ldots, x_K$ that are just copies of input: $x_k = x_{in} \forall k$

Branch B-prop: $\frac{\partial E}{\partial x_{in}} = \sum_{k=1}^{K} \frac{\partial E}{\partial x_k}$.
Softmax Module

- Single input $x$. Normalized output vector $z$, i.e. $\sum_i z_i = 1$.
- F-Prop: $z_i = \frac{\exp(-\beta x_i)}{\sum_k \exp(-\beta x_k)}$
- $\beta$ is "temperature", usually set to 1.
- B-prop:
  - If $i = j$, then $\frac{\partial z_i}{\partial x_j} = z_i(1 - z_i)$.
  - If $i \neq j$, then $\frac{\partial z_i}{\partial x_j} = -z_i z_j$.
- Often combined with cross-entropy cost function:
  $E = -\sum_{c=1}^{C} y_c \log(z_i)$.
- Conveniently, this yields b-prop: $\frac{\partial E}{\partial x_i} = x_i - y_i$. 

Fergus, Miller, Puhrsch
Introduction to Deep Learning
Practical Tips for Backprop
[from M. Ranzato and Y. LeCun]

- 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). More on this later.
- 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
- See also [LeCun et al. Efficient Backprop 1998]
Setting the Learning Rate

- Learning rate $\eta$ has dramatic effect on resulting model.
- Pretend energy surface is quadratic bowl (in reality, much more complex).
- Gradient descent direction is just local, so if surface is highly elliptical then easy to have learning rate too large and oscillate.
- Difficult to have single learning rate that works for all dimensions.

[Figures: G. Hinton]
Annealing of Learning Rate

- Constant learning rate $\eta$ typically not optimal.
- Start with largest value that for which training loss decreases, e.g. 0.1.
- Then train until validation error flatens out.
- Divide $\eta$ by, say, 0.3.
- Repeat.
Automatic Adjustment of Learning Rate

- Smart way of adjusting $\eta$ automatically?
- Active area of research in optimization community.
- **ADAGRAD:** $\Delta \theta^k = - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k} (\nabla \theta)^2}} (\nabla \theta)^k$
  
  [Duchi et al., Adaptive subgradient methods for online learning and stochastic optimization, in COLT, 2010].

- **ADADELTA:** $\Delta \theta^k = - \frac{RMS[\Delta \theta]^{k-1}}{RMS[\nabla \theta]^k} (\nabla \theta)^k$
  

- **RMSProp** (similar to **ADADELTA**):
  
  $E[(\nabla \theta)^2] = 0.9E[(\nabla \theta)^2] + 0.1(\nabla \theta)^2$

  $\Delta \theta^k = - \frac{\eta}{\sqrt{E[(\nabla \theta)^2] + \epsilon}} (\nabla \theta)^k$


- **ADAM** uses different learning rate for each parameter
  
  ADAM: A method for Stochastic Optimization, Kingma and Ba, ICLR 2015].

Fergus, Miller, Puhrsch | Introduction to Deep Learning
Momentum
To speed convergence

- Add momentum term to the weight update.
- Encourages updates to keep following previous direction.
- Damps oscillations in directions of high curvature.
- Builds up speed in directions with gentle but consistent gradient.
- Usually helps speed up convergence.

\[ \theta^{k+1} \leftarrow \theta^k + \alpha (\Delta \theta)^{k-1} - \eta \nabla \theta \]

- \(\alpha\) typically around 0.9.

[Slide: G. Hinton]
Nesterov Momentum

- Simple idea.
- Update weights with momentum vector.
- Then measure gradient and take step.
- This is opposite order to regular momentum.

[Figure: G. Hinton]

**Input:** Values of $x$ over a mini-batch: $B = \{x_1 \ldots x_m\}$; Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

\[
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i & \text{// mini-batch mean} \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 & \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} & \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) & \text{// scale and shift}
\end{align*}
\]

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

![Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.](image)
Local Minima
Non-convexity of energy surface

- Non-convexity means there are multiple minima.
- Gradient descent is a local method: the minima you fall into depends on your initial starting point.
- Maybe some minima have much lower energy than others?
- The Loss Surfaces of Multilayer Networks Choromanska et al. 
DropOut

- Fully connected layers only.
- Randomly set activations in layer to zero
- Gives ensemble of models
- Similar to bagging [Breiman94], but differs in that parameters are shared
Debugging Training
What to do when it's not working

- Training diverges:
  - Learning rate may be too large  decrease learning rate
  - BPROP is buggy  numerical gradient checking
- Parameters collapse / loss is minimized but accuracy is low
  - Check loss function:
  - Is it appropriate for the task you want to solve?
  - Does it have degenerate solutions? Check pull-up term.
- Network is underperforming
  - Compute flops and nr. params. if too small, make net larger
  - Visualize hidden units/params  fix optimization
- Network is too slow
  - Compute flops and nr. params. GPU,distrib. framework, make net smaller
Debugging Training (2)
What to do when it's not working

- Inspect hidden units.
- Should be sparse across samples and features (left).
- In bad training, strong correlations are seen (right), and also units ignore input.

[Figures: M. Ranzato]
Debugging Training (3)
What to do when its not working

- Visualize weights

**Good training:** learned filters exhibit structure and are uncorrelated.

[Figure: M. Ranzato]
Regularization

The Intuition

Better to have big model and regularize, than unfit with small model.
Regularizing the model
Help to prevent over-fitting

- Weight sharing (greatly reduce the number of parameters)
- Data augmentation (e.g., jittering, noise injection, etc.)
- Dropout.
- Weight decay (L2, L1).
- Sparsity in the hidden units.
- Multi-task learning.