Feedforward Neural Networks and Activation Functions

Gokberk Cinbis
Where we are...

\[ s = f(x; W) = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2 \]
Where we are...

\[ s = f(x; W) = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2 \]

scores function
SVM loss
data loss + regularization
Neural Network: without the brain stuff

(Before) Linear score function:

\[ f = Wx \]
Neural Network: without the brain stuff

(Before) Linear score function:

\[ f = Wx \]

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]
Neural Network: without the brain stuff

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]
Neural Network: without the brain stuff

(Before) Linear score function: \[ f = Wx \]

(Now) 2-layer Neural Network \[ f = W_2 \max(0, W_1x) \]
Neural Network: without the brain stuff

(Before) Linear score function: \[ f = Wx \]

(Now) 2-layer Neural Network or 3-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]
sigmoid activation function

\[ f(x) = \frac{1}{1 + e^{-x}} \]
class Neuron:
    # ...
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
        return firing_rate
Be very careful with your Brain analogies:

**Biological Neurons:**
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

[Dendritic Computation. London and Hausser]
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\text{tanh} \quad \text{tanh}(x)

\text{ReLU} \quad \max(0, x)

Leaky ReLU

\[ \max(0.1x, x) \]

Maxout

\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

ELU

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Neural Networks: Architectures

“2-layer Neural Net”, or “1-hidden-layer Neural Net”

“Fully-connected” layers

“3-layer Neural Net”, or “2-hidden-layer Neural Net”
Example Feed-forward computation of a Neural Network

```python
class Neuron:
    # ...
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number ""
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))  # sigmoid activation function
        return firing_rate
```

We can efficiently evaluate an entire layer of neurons.
Example Feed-forward computation of a Neural Network

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x))  # activation function (use sigmoid)
x = np.random.randn(3, 1)  # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1)  # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)  # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3  # output neuron (1x1)
```
Setting the number of layers and their sizes

3 hidden neurons

6 hidden neurons

20 hidden neurons

more neurons = more capacity
Do not use size of neural network as a regularizer. Use stronger regularization instead:

\[ \lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1 \]
Neural Network Demo

http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html
Summary

- we arrange neurons into fully-connected layers
- the abstraction of a layer has the nice property that it allows us to use efficient vectorized code (e.g. matrix multiplies)
- neural networks are not really neural
- neural networks: bigger = better (but might have to regularize more strongly)
Training of Feedforward Neural Networks
Recap: Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Neural Turing Machine

input tape

loss
activations

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}
\]

“local gradient”

gradients

\[
\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}
\]

CS 559 Deep Learning
Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Implementation: forward/backward API

Graph (or Net) object. *(Rough psuedo code)*

```python
class ComputationalGraph(object):
    
    #...
    
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss  # the final gate in the graph outputs the loss
    
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward()  # little piece of backprop (chain rule applied)
        return inputs_gradients
```
Implementation: forward/backward API

```python
class MultiplyGate(object):
    def forward(x, y):
        z = x * y
        self.x = x  # must keep these around!
        self.y = y
        return z

    def backward(dz):
        dx = self.y * dz  # [dz/dx * dL/dz]
        dy = self.x * dz  # [dz/dy * dL/dz]
        return [dx, dy]
```

(x, y, z are scalars)
Example: Torch Layers

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Neural Network: without the brain stuff

(Before) Linear score function:

\[ f = Wx \]

(Now) 2-layer Neural Network or 3-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]
impulses carried toward cell body

dendrites

nucleus

cell body

branches of axon

impulses carried away from cell body

axon

axon terminals

\[ f \left( \sum_i w_i x_i + b \right) \]

activation function

output axon

\[ w_0 x_0 \]

axon from a neuron

dendrite

\[ w_1 x_1 \]

class Neuron:

# ...
def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))  # sigmoid activation function
    return firing_rate

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Neural Networks: Architectures

“2-layer Neural Net”, or “1-hidden-layer Neural Net”

“Fully-connected” layers

“3-layer Neural Net”, or “2-hidden-layer Neural Net”
Training Neural Networks

A bit of history...
A bit of history

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

recognized letters of the alphabet

\[ f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases} \]

update rule:

\[ w_i(t + 1) = w_i(t) + \alpha (d_j - y_j(t)) x_{j,i} \]

*Frank Rosenblatt, ~1957: Perceptron*
A bit of history

Widrow and Hoff, ~1960: Adaline/Madaline
A bit of history

Rumelhart et al. 1986: First time back-propagation became popular

\[ E_p = \frac{1}{2} \sum (u_p - o_p)^2 \]

be our measure of the error on input/output pattern \( p \) and let \( E = \sum E_p \) be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in \( E \) when the units are linear. We will proceed by simply showing that

\[ \frac{\delta E_p}{\delta w_{ji}} = \delta_{ji} b_i, \]

which is proportional to \( \Delta w_{ji} \) as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivatives. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weights.

\[ \frac{\delta E_p}{\delta w_{ji}} = \frac{\delta E_p}{\delta o_j} \frac{\delta o_j}{\delta w_{ji}}. \]

The first part tells how the error changes with the output of the \( j \)th unit and the second part tells how much changing \( w_{ji} \) changes that output. Now, the derivatives are easy to compute. First, from Equation 2

\[ \frac{\delta E_p}{\delta o_j} = - (o_j - o_p) = - \delta_p. \]

Not surprisingly, the contribution of unit \( ij \) to the error is simply proportional to \( \delta_p \).

Moreover, since we have linear units,

\[ o_{pj} = \sum_{i} w_{pj} i_i, \]

from which we conclude that

\[ \frac{\delta o_{pj}}{\delta w_{ji}} = b_i. \]

Thus, substituting back into Equation 3, we see that

\[ \frac{\delta E_p}{\delta w_{ji}} = \delta_p b_i. \]
A bit of history

[Hinton and Salakhutdinov 2006]

Reinvigorated research in Deep Learning
First strong results

**Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition**
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

**Imagenet classification with deep convolutional neural networks**
Overview

1. One time setup
   activation functions, preprocessing, weight initialization, regularization, gradient checking

2. Training dynamics
   babysitting the learning process, parameter updates, hyperparameter optimization

3. Evaluation
   model ensembles
Activation Functions
Activation Functions

\[ f \left( \sum_i w_i x_i + b \right) \]

- **Axon from a neuron**: \( w_0 x_0 \)
- **Dendrite**: \( w_1 x_1 \)
- **Synapse**: \( w_2 x_2 \)
- **Cell body**
- **Output axon**
- **Activation function**
Activation Functions

**Sigmoid**

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

**tanh**  \( \tanh(x) \)

**ReLU**  \( \max(0, x) \)

**Leaky ReLU**

\[ \max(0.1x, x) \]

**Maxout**  \( \max(w_1^T x + b_1, w_2^T x + b_2) \)

**ELU**

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]
Activation Functions

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?

$$
\sigma(x) = \frac{1}{1 + e^{-x}}
$$

$$
\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}
$$

$$
\frac{\partial L}{\partial \sigma}
$$
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
Consider what happens when the input to a neuron (x) is always positive:

\[
f \left( \sum_i w_i x_i + b \right)
\]

What can we say about the gradients on \( \mathbf{w} \)?
Consider what happens when the input to a neuron (x) is always positive:

What can we say about the gradients on w?

Answer: All positive or all negative
Consider what happens when the input to a neuron is always positive...

$$f \left( \sum_i w_i x_i + b \right)$$

What can we say about the gradients on $w$?
Always all positive or all negative :(
(this is also why you want zero-mean data!)
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. \( \exp() \) is a bit compute expensive

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Activation Functions

- Squashes numbers to range \([-1,1]\)
- zero centered (nice)
- still kills gradients when saturated :(

\[ \text{tanh}(x) \]

[LeCun et al., 1991]
Activation Functions

- Computes $f(x) = \max(0,x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions

ReLU (Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?

\[ \sigma(x) = \max(0, x) \]
DATA CLOUD

active ReLU

dead ReLU
will never activate
=> never update
=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

dead ReLU will never activate

=> never update
Activation Functions

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

\[ f(x) = \max(\alpha x, x) \]

backprop into \( \alpha \) (parameter)

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Exponential Linear Units (ELU)

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires $\exp()$

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Maxout “Neuron”
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

\[
\max(w_1^T x + b_1, w_2^T x + b_2)
\]

Problem: doubles the number of parameters/neuron :(  

[Goodfellow et al., 2013]
TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid
What comes next?

- Data preprocessing
- Initialization
- Babysitting the learning process
- Hyperparameter optimization