# Deep Learning, goodfellow (2016)

A feedforward network defines a mapping $y = f(\theta, x)$ & learns the value of the parameters $\theta$ that results in the Best Func p

Convolutional networks for object recog are a spezalized kind of feedforward Network

conceptual Stepping Stone to recurrent networks

FF called networks as represented by composing many different functions

The Nets have a chain structure $f(x) = f3(f^2(f^1(x1)))$

$f^1$ = first layer $\quad f^{n/3}$ = last layer

the length of the chain is the depth of the model

$f(x)$ provides use with appox value of $y \quad y \approx f^*(x)$

Dimensionality of the network is the width

NN conceptually starts w/ linear models, i.e. linreg: logress, and adaptions are made to over come non-linear limitations

To extend linear models to rep nonlinear functions of x — we apply the lin model to a transform versn of $x \rightarrow \phi(x)$

where $\phi$ = non-linear transformation

(2)

$\phi$ Provides a new representation of $x$

How to choose the $\phi$ mapping?

(1) • use generic $\phi$ such as infinite-dimensional that is implicit used by kernal machine or RBF Kernel
  • if $\phi(x)$ is high enough dimension, can always find a fit in the ~~tr~~ train data.
  • But generalization will be poor
  • Do not include enough info to solve advanced into

(2) • manually engineer $\phi$
  • Dominant convention prior to DL
  • requires alot of Human effort for each task
  • Specific to domains w/ little knowledge transfer between domains

(3) • Deep learning strategy is to learn $\phi$
  • We have a model: $y = f(x;\theta,w) = \phi(x;\theta)^T w$
  • $\theta$ paras, $\phi$ functions, $w$ weights/mappings
  = Deep foward network
  • This gives up on convexity of problem
  • Parameterize $\phi(x;\theta)$ & use opt algo to find ~~good~~ $\theta$ that correspond to good output

# 6.2 Gradient Based Learning

training NN is not that much diff to any other ML model w/ gradient Descent

The larges DIFF between linear models & NNs is the non-linearity cause loss function to become non-convex

This means NNs are trained using iterative gradient based optimizers

rather than linear equation solvers (lin reg)

or convex opt algos w/ global convergence gaurentees used to train (log reg or SVM)

Convex optim converges from any initial parameters (in theory)

Stochastic gradient Descent has no convergence gauntees & is sensitive to starting parameters (optim applied to non-convex)

for ff Nets, MUST set initial paras to small random values

## 6.2.2 Output Units

Cost function chosen is tightly related to the output units of the NN

typically uses cross entropy of data dist & model dist

Output then determins to form of cross ent function

~~Con~~ Any Hidden unit can be ~~as~~ ~~be~~ used as an output unit

A FF NN provides hidden features given by $h = f(x; \theta)$

the role of the final output layer is to provide some additional transformation from the features to complete the network task

## 6.2.2.4 Other types of Output Units

NNs represent a function $f(x, \theta)$

the outputs are not direct predictions of $y$

instead $f(x, \theta) = w$ provides parameters for a distribution over $y$

loss func is then interpretated as: $-\log p(y; w(x))$

Therefore the model can be adapted to learn the variance of $p$

## 6.3 Hidden Units

How to choose type of hidden unit to use in the hidden layers?

At time of book, was considered to be an active area of research w/o definitive theoretical principles

Rectified linear units are a popular choice

Picking the best in advance is not usually possible. Instead requires a design process of trial & error

Not all hidden units are are differentiable at all input points

e.g. Relu $g(z) = \max\{0, z\}$ is not diff @ $z = 0$

This may seem like it invalidates g for use w/ a gradient-based learning algo

but Practice gd still works well. This is because NNs do not arrive @ the min of a loss func. — but just reduce value significantly

Hidden units that are not diff usually only exist @ a small number of points

## 6.3.1 Rectified linear units

activ func where $g(z) = \max\{0, z\}$

easy to opt as similar to linear units. only diff = 0 across half the domain (-)

Second the derivatives of the activ units (>0) are large & consistent

large Dervative =
- Small change in input = large Δ in output
- Allows for faster learning & bigger steps towards min the error
- Backprop gradients have a gradient that becomes very small in the layer — hence large gradient is a good quality to counter

Consistent Derivative:
- Stable = not wildly diff changes per unit Δ of input
- reliable converge
- Avoid oscilations = hard to converge

Second Derive of activ $z = 1$ & $z = 0 = 0$

= gradient dir is far more useful vs act funcs that intro second-order effects

ReLu applied over top of affine transform

: $h = g(W^T x + b)$

7

when initilizing the parameters, it can be good to start w/ small values of $b$, 0.1

ensures most of the ~~hidd~~ units will remain achive for most of the training

↳ Allows Differable units to flow through

There are some generalized versions of rect linear units which seek to address 0 issue

## 6.3.2 Logistic Sigmoid & Hyperbolic tangent

Prior to recln, most NNs used Sig act Fnc of

$g(z) = \sigma(z)$

or hyperbolic tangent activ
$g(z) = \tanh(z)$

related as $\tanh(z) = 2\sigma(2z) - 1$

Issue w/ Sig is that it saturates at high vs low (-) values

i.e. a big change in the input doesn't change much

only when value around 0 is the gradient significant

Saturate = learn slow or poorly

Widespread saturation makes gradient based learning difficult

for this reason, hidden sig units in FF Nets are now discouraged

Only approp when units are coupled w/ a loss function that undo sigmoid output (log $\Rightarrow$ exp)

Sig functions are common in other settings (other than FFNN) hence there inclusion may still happen

## 6.3.3 Other Hidden Units

In genal a wide variety of Hidden funcs work well

Many unpublished = just as good as popular

During research it is common to test many & find they test comparably

Newly discovered Hidden types are considered to be so common to be considered as uninteresting

Using no activation can be considered as using an identity matrix

Not all activs can be linear otherwise the output of the model would have no non-lin

Acceptable for some to be linear

## 6.4. Architecture Design

Architecture = overall structure of the network

* How many units it should have
* How they should be connected

* MOST NNs = organized into layers
* these layers are in a chain structure
  i.e. each a func of the prev func

Main considerations of network = Depth & width

Deep = less paras needed (width)
but harder to optimize

Ideal Network archi = Experiment & guide via validation set

# 6.5 Backpropigation & other Diff Algos

* ff NN accepts an input $x$ & produces $\hat{y}$
* info flows forward through the network
* initial info propagates up to the hidden units @ each layer
* training forward prop continues until we create a scalar cost $J(\theta)$

Backprop Allows the info from the cost to the flow backward through the network in order to compute the network

Generally comp an analytical expression is easy, but evaluating is comp costly

Backprop does so using simple & inexpen proc

Backprop = just the method for computing the gradient
→ other algos, i.e, SDG, Performs the learning using this gradient

We will work through how to compute the gradient for $\nabla_x f(x, y)$

$x$ is set of vars whose derivs are desired

$y$ = inputs to func but Derivs not required

the gradient we most require is that of the cost function w/ respect to the params $\nabla_\theta J(\theta)$

## 6.5.2 Chain Rule of Calculus

Backprop is the algo to compute chain Rule, with a specific order of calculations

$x = \mathbb{R}$ $\quad f, g$ : functions $\mathbb{R} \to \mathbb{R}$

$y = g(x)$ , $z = f(g(x)) = f(y)$

then chain rule: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$

If $x, y$ are vectors: $\mathbb{R}^n$ & $g: \mathbb{R}^n \to \mathbb{R}^m$ $\quad f: \mathbb{R}^m \to \mathbb{R}$

$$\frac{\partial z}{\partial x_i} = \sum \frac{\partial z}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_i}$$

in vector notation: $\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$

where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian Matrix of $g$

gradient of value $x$ obtained by multiplying Jacob Matrix by the gradient $\nabla_y z$

Backprop algo = perform Jacob-gradient product for each operation in the graph

Usually Back prop is applied to n-Dim tensor, not just vectors.

Conceptually, this is the same as vectors but rearranging

Backprop = Multi Jacob by gradients

## 6.5.3 Recursively applying the Chain rule to obtain backprop

Use chain rule to obtain gradient of a scalar (loss func) w/ respect to any node in the graph — that produced scalar

Computed gradients may be stored to reduce comp load

Start w/ version of Backprop that specifys the actual gradient directly

In the order that the traversed graph & chain rule ~~choo~~ direct

Consider a ~~fing~~ Comp graph desc how to comp a single scalar — $u^{(n)}$

the scalar = gradient we want to obtain w/ respect to the $n_i$ input nodes $u^{(i)}$ to $u^{(n_i)}$

we wish to calc: $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ For all $i \in \{1, 2, \ldots, n_i\}$

$u^{(1)}$ to $u^{(n_i)}$ : Parameters of the model

Assume graph allows recursive calcs

Start @ $u^{(n_i+1)}$ and go upto $u^{(n)}$

(13)

ff Calculates a graph G

Backprop creates a "new" graph B w/ one node per node of G

B order = Traverse of G

@ each node of B we calc deriv $\frac{\partial u^{(n)}}{\partial u^{(i)}}$

assoc w/ ff node $u^{(i)}$

this is the chain rule w/ respect to scalar Out $u^{(n)}$

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i : j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(j)}} \qquad \text{(Algo 6.2)}$$

↑ edge

B = one edge from node $u^{(i)}$ to $u^{(j)}$ of G

Computation for Backprop scales to linear w/ number of edges in G

where the comp of each edge correponds to the partial derivative compute w/ respect to one of its ~~parameters~~ Parents

## Algo 1 fast forward

Proc to perform mapping $n_i$ inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$

This defines a comp graph where each node computes numberical value $u^{(i)}$ by applying a function $f(i)$

to a set of args $A^{(i)}$ that comprsis the values of the prev nodes $u^{(j)} : j < i$

the input of graph is vector $x$

and is set into the first $n_i$ nodes $u^{(1)}$ to $u^{(n_i)}$

the output is read from the last node $u^{(n)}$

---

x vector = single data point

Input
for $i = 1, \ldots n_i$ do
  $u^{(i)} \leftarrow x^i$
end for

Dimensional value holder
($u^{(i)}$ = ith input node)

Hidden
for $i = n_i + i, \ldots, n$ do
$A^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
$u^{(i)} \leftarrow f^{(i)}(A^{(i)})$
end for

Set of value from parent nodes

Parent nodes of $u^{(i)}$

function applied to A vector

Output
return $u^{(n)}$

# Algo 2 Backprop

* Calculate derivatives of $u^{(n)}$
* w/ respect to variables in the graph
* Simple example where all variables are scalars (not vec)
* Compute derivs w/ respect to $u^{(1)}, \ldots, u^{(n_i)}$
* Computes Dev of all nodes in graph
* Comp cost = proportional to num of edges
* Assumes partial derivative w/ each edge requires const time
* Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a func of the parents $u^{(j)}$ of $u^{(i)}$

---

- Start with FF to obtain network - Algo 1
- init grad table — A data structure to store Devs that have been calced. GT$[u^{(i)}]$ = value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$

grad_table$[u^{(n)}] \leftarrow 1$

(using stored values)

for $j = n-1$ down to 1

Compute $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i : j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(j)}}$

grad_table$[u^{(j)}] \leftarrow \sum_{i : j \in Pa(u^{(i)})}$ grad_table$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return $\{$grad_table$[u^{(i)}] \mid i = 1, \ldots, n_i\}$

## 6.5.4 Backprop in full connect MLP

### Algo 6.3 forward Prop to cost func

Req Network depth $l$
Req $W^{(i)}$, $i \in \{1, \ldots, l\}$ weight mats of model
Req $b^{(i)}$, $i \in \{1, \ldots, l\}$
Req $x$ input
Req $y$ target output

$h^{(0)} = x$
For $k = 1, \ldots, l$ do
$a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
$h^{(k)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

## 6.4 Algo Backprop for Deep NN

- Comp yeilds the gradients on the activations $a^{(k)}$ for each layer k
- Start from output layer and go Backwards to 1st hidden layer
- Gradients = how each layer should change to reduce error
- from these, obtain the gradient of the parts of each layer
- the grads on the w & b can be immediately used as part of a SGD update

---

After forward comp, compute gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} L = \nabla_{\hat{y}} L(\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ do

Convert the grad on the layers output into a grad on the pre-linearity activation

$$g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)})$$

Comp grads on weights & biases

$$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$$
$$\nabla_{W^{(k)}} J = g\, h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$$

Propagate the grads to the next lower level layer activis

$$g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)T} g$$

end for