CSC 580 Principles of Machine Learning

13 Neural networks (NN)

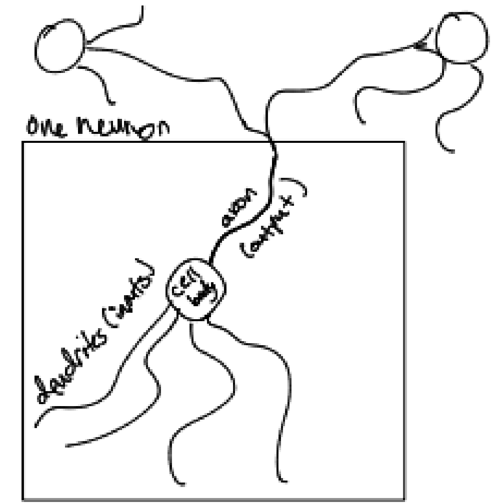
Chicheng Zhang

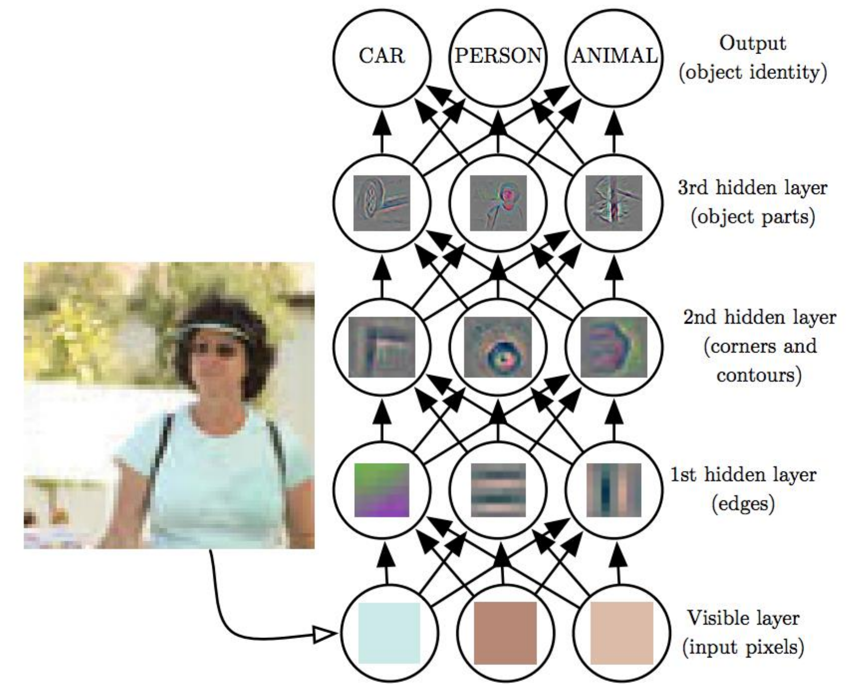
Department of Computer Science

*slides credit: built upon CSC 580 Fall 2021 lecture slides by Kwang-Sung Jun

Neural networks

- Obviously, a dominant approach these days.
- From biological inspirations
- From machine learning point of view, it is one way to train **nonlinear** functions (recall kernel methods)
- Turns out, the strength is the **representation learning**!
    - handling images / natural languages
- When the main bottleneck is not the representation, on par with competitors





https://pantelis.github.io/cs677/docs/common/lectures/deep-learning-introduction/

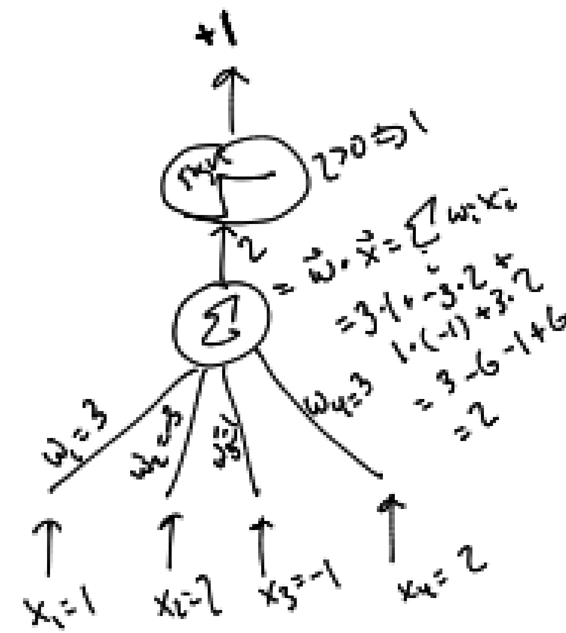
Neural networks: some learning resources

(A highly incomplete list)

- ``Deep Learning'' book by Goodfellow, Bengio, Courville: https://www.deeplearningbook.org/
- U Washington CS446: https://courses.cs.washington.edu/courses/cse446/19wi/
- Stanford CS231n: https://www.youtube.com/watch?v=gYpoJMlgyXA&feature=youtu.be&t=20m54s&ab_channel=AndrejKarpathy
- PyTorch tutorials: https://pytorch.org/tutorials/
- Many more...

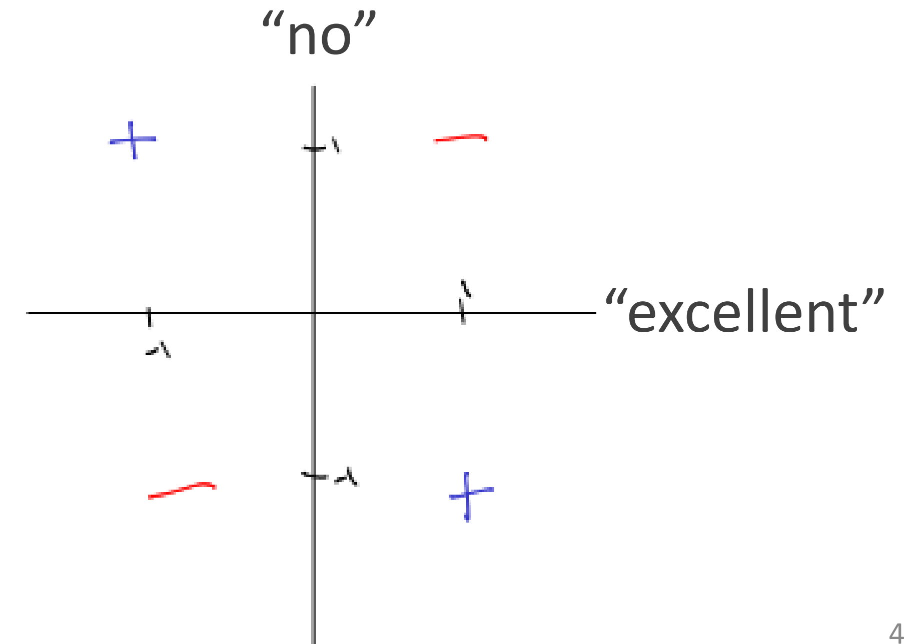
Warmup: Perceptron

- Perceptron as a neuron
  - $h_w(x) = \text{sign}(\langle w, x \rangle)$



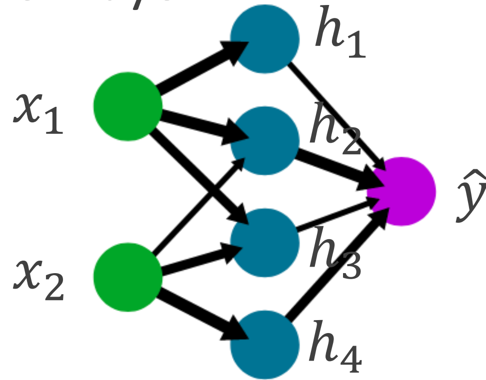
- Logistic regression
  - $h_w(x) = \sigma(\langle w, x \rangle)$

- Issues: limited expressive power of linear models
  - the XOR problem



Multi-layer perceptron (MLP)

- Example: one hidden layer



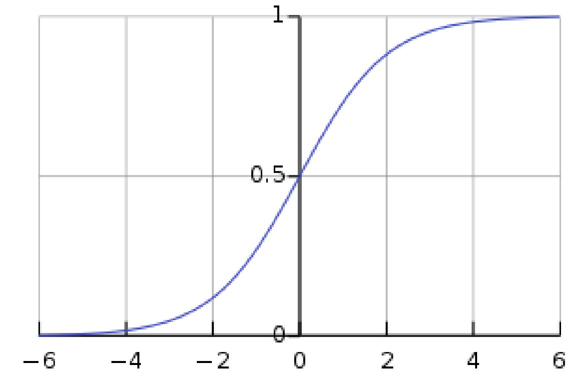
$$h_i = \sigma\left(\sum_{i=1}^{d} w_{ij}\, x_j\right), i = 1,2,3,4$$

$$\hat{y} = \sigma\left(\sum_{i=1}^{4} v_i\, h_i\right)$$

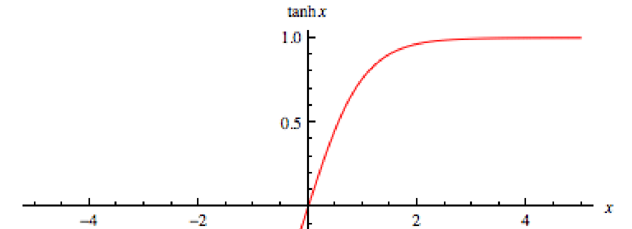
- In matrix form: $h = \sigma(Wx)$, $\hat{y} = \sigma(vh)$
  - here σ denotes elementwise application of scalar function σ
  - In short: $\hat{y} = \sigma\big(v \cdot \sigma(Wx)\big)$
- Activation function σ matters:
  - Nonlinear σ results in nonlinear models with good expressive power (complicated decision boundaries)
  - if $\sigma(z) = z$ => $\hat{y} = vWx$ => it's still linear in the input (just overparameterized)
  - Need it to be *differentiable* for gradient-based training

recommended simulation: https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

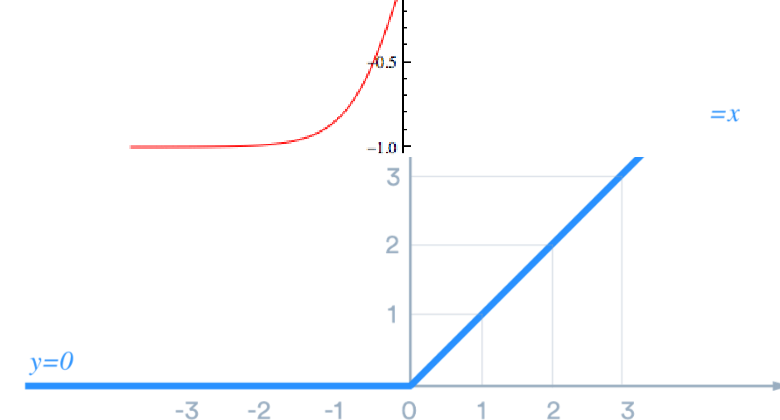
Activation functions



- Sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$



- tanh(z) = $2\sigma(2z) - 1$
  - tanh is preferred (Section 4.4 of LeCun et al., Efficient BackProp, 1998.)
  - Faster convergence when the variables are centered.



- Both above: Vanishing gradient problem
- Solution ReLU(z) = max{z,0}   // REctified Linear Unit
  - Still, outputs are not centered => batch normalization helps.

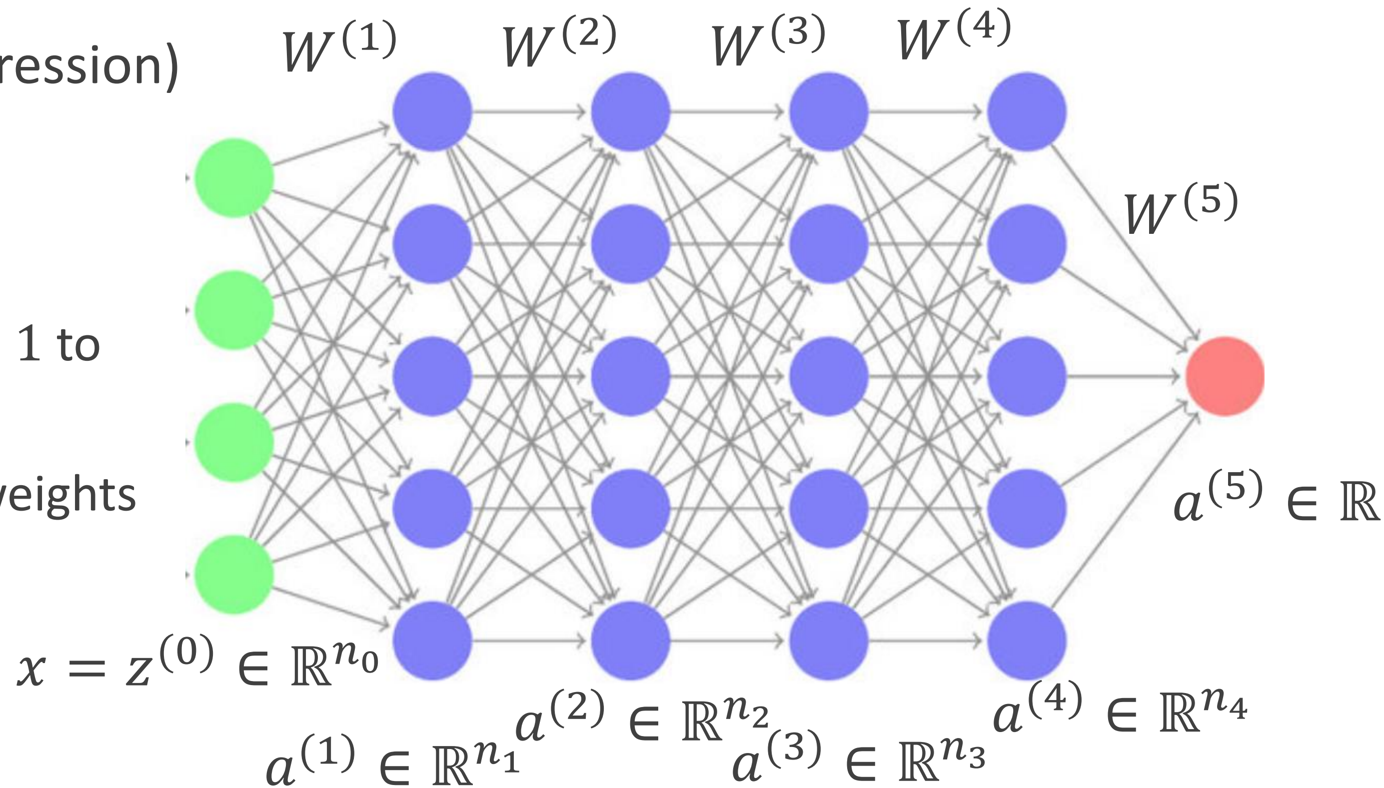
- Note: Section 6.3.2. of Deep Learning book

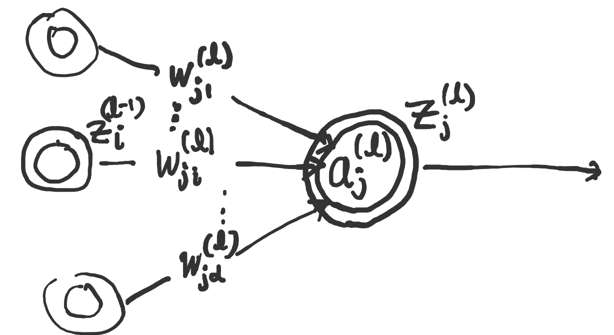
  "Sigmoidal activation functions are more common in settings other than feed-forward networks. Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation."

https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

Multi-layer perceptron (MLP): general case

- L: #hidden layers; $n_0 = d, n_{L+1} = 1$ (for regression)
- $z_i^{(0)} := x_i$
- For layer $l \in \{1, \dots, L+1\}$
  - $w_{ji}^{(l)} \in \mathbb{R}$: weight from neuron i from layer $l-1$ to neuron j in layer l
  - $w_j^{(l)} = (w_{j1}^{(l)}, \dots, w_{jn_{l-1}}^{(l)}) \in \mathbb{R}^{n_{l-1}}$: j-th neuron weights
  - $a_j^{(l)} \in \mathbb{R}$: j-th neuron output before activation
  - $z_j^{(l)} \in \mathbb{R}$: j-th neuron output after activation
- $W^{(l)} = \begin{pmatrix} - w_1^{(l)} - \\ \dots \\ - w_{n_l}^{(l)} - \end{pmatrix} \in \mathbb{R}^{n_l \times n_{l-1}}$
- In matrix form: $\hat{y} = W^{(L+1)} \sigma\big(W^{(L)} \dots \sigma(W^{(2)} \sigma\big(W^{(1)} x\big))\big)$





Expressive power of neural networks

- (Cybenko, 1989; Hornik et al, 1989)

Theorem 10 (Two-Layer Networks are Universal Function Approximators). *Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate F arbitrarily well. Namely, for all $\boldsymbol{x}$ in the domain of F, $\left|F(\boldsymbol{x}) - \hat{F}(\boldsymbol{x})\right| < \epsilon$.*

- Does this mean that there is no benefit in learning deeper networks?
  - No (Eldan and Shamir, 2015; Telgarsky, 2016)

The Power of Depth for Feedforward Neural Networks

Ronen Eldan, Ohad Shamir

We show that there is a simple (approximately radial) function on $\reals^d$, expressible by a small 3-layer feedforward neural networks, which cannot be approximated by any 2-layer network, to more than a certain constant accuracy, unless its width is exponential in the dimension. The result holds for virtually all known activation functions, including rectified linear units, sigmoids and

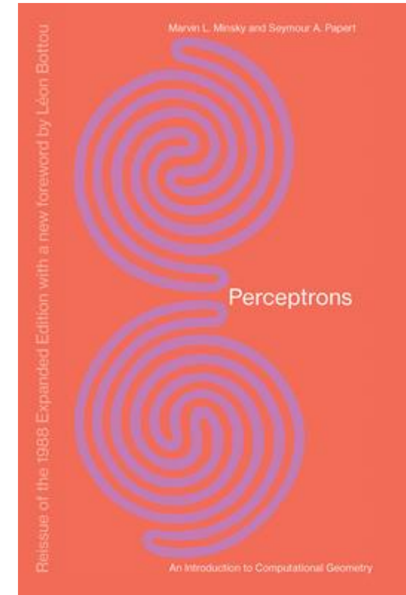
Benefits of depth in neural networks

Matus Telgarsky

For any positive integer k, there exist neural networks with $\Theta(k^3)$ layers, $\Theta(1)$ nodes per layer, and $\Theta(1)$ distinct parameters which can not be approximated by networks with $\mathcal{O}(k)$ layers unless they are exponentially large --- they must possess $\Omega(2^k)$ nodes. This result is proved here for a class of nodes termed "semi-algebraic gates" which includes the common choices of ReLU, maximum,

Neural network: some history

- 60’s: early interest in perceptron, but the XOR problem..
  - Minsky-Papert “Perceptrons”, 1969



- MLP was a way to get around, but people did not know how to train it
- Werbos’74 breakthrough: backpropagation (but still hard to get people back)
- NNs became popular again in ‘86 with McClelland, Rumelhart, and Hinton on training large-scale neural nets.
- Since 2010: NNs took off, with well-developed high-performance computing infrastructure (GPUs) and easy-to-use programming framework (Tensorflow / Torch / Keras /..)

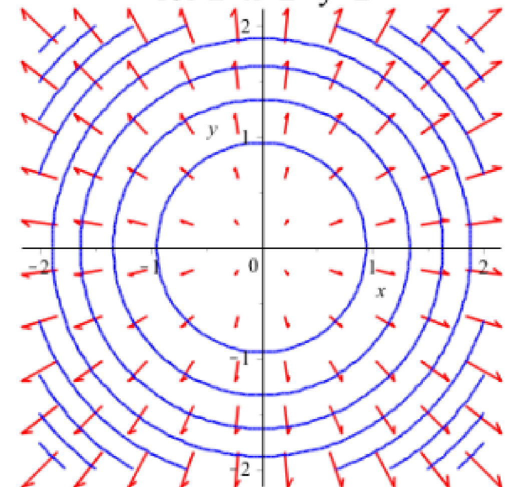
Training neural networks

Recap: partial derivatives & gradients

- Given $F(x_1, \dots, x_d)$,

$$\frac{\partial F}{\partial x_i} = \lim_{\Delta \to 0} \frac{F(x_1, \dots x_i + \Delta, \dots, x_d) - F(x_1, \dots, x_i, \dots, x_d)}{\Delta}$$

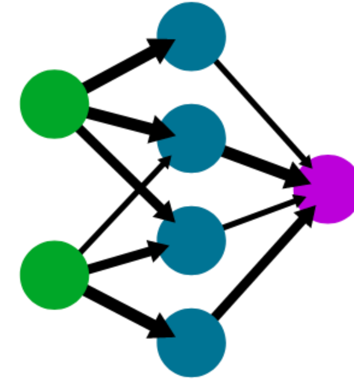
- Measures the *sensitivity* of F with respect to input x_i

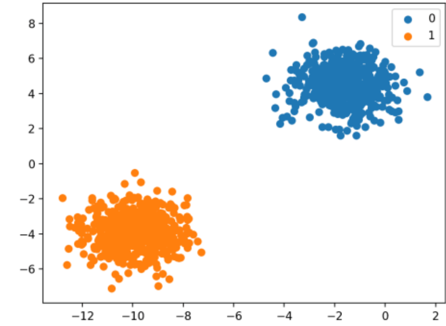


- $\nabla F(x) = \left(\frac{\partial F}{\partial x_1}, \dots, \frac{\partial F}{\partial x_d}\right)$
- The effect of each coordinate x_i on $F(x)$ is *additive*:
  - For differentiable F: $F(x + \Delta x) \approx F(x) + \langle \nabla F(x), \Delta x \rangle$

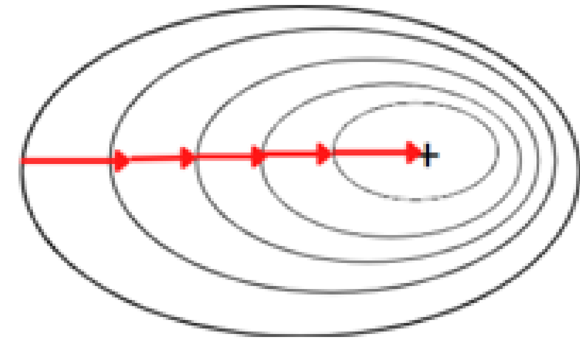
Gradient descent revisited

- Minimizing empirical loss $F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w),\, w \in \mathbb{R}^d$

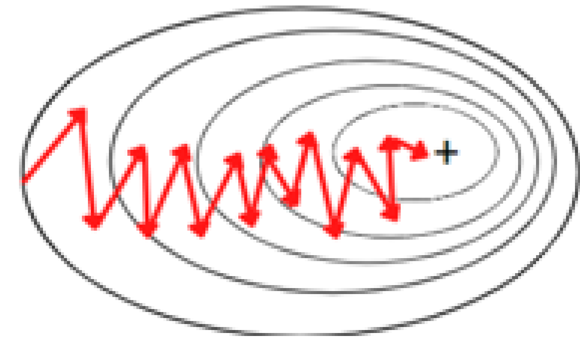




- Gradient descent (GD)
- For $t = 1, \dots, T$:
  - $w_t \leftarrow w_{t-1} - \eta_t \cdot \nabla F(w_{t-1})$



- Mini-batch stochastic gradient descent (SGD)
- For $t = 1, \dots, T$:
  - Randomly sample a size-k subset $S_t \subset \{1, \dots, T\}$
  - $F(w) = \frac{1}{k}\sum_{i \in S_t} f_i(w)$
  - $w_t \leftarrow w_{t-1} - \eta_t \cdot \nabla F^{(t)}(w_{t-1})$



Optimization methods: a comparison

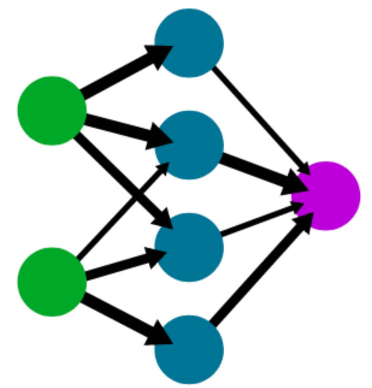
Algorithms	Number if iterations until convergence	Time complexity per iteration
Newton's method	Very small	nd^3
LBFGS	small	nmd
Gradient descent (GD)	large	nd
Stochastic gradient descent (SGD)	Very large	d

Mini-batch SGD: kd
Friendly for GPUs

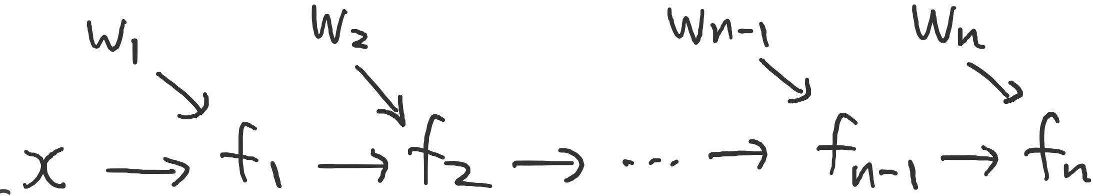
- n: #training examples
- d: dimensionality of optimization variable w
- m: LBFGS's memory hyperparameter

- But how to obtain $\nabla F(w)$?

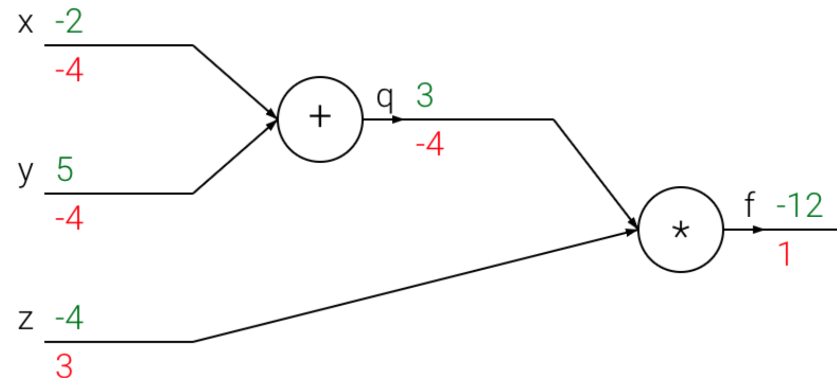
Key tool: Computation graph



- A DAG that describes the order of computation in a general computational process
  - Nodes: variables
  - Edges: dependency of the variables
  - $f_1 = F_1(w_1, x), \ldots, f_n = F_n(w_n, f_{n-1})$
- More general than neural networks' prediction processes

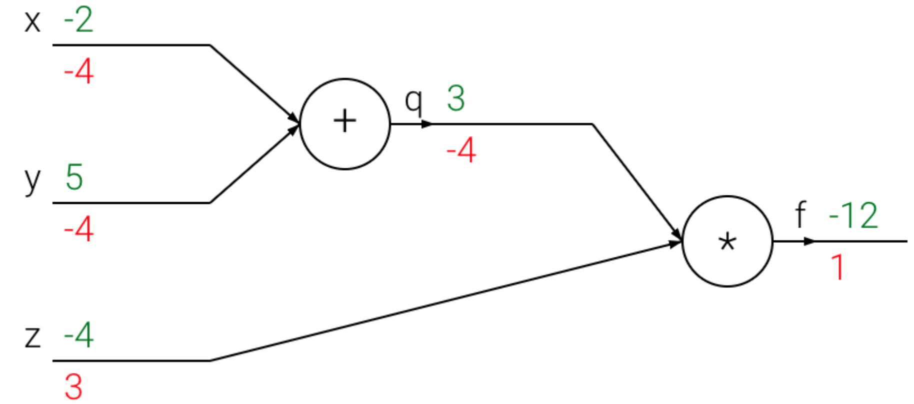


- Sometimes useful to highlight the *operation* that computes each variable as well
- E.g. $q = F_q(x, y) = x + y; f = F_f(q, z) = q \cdot z$



Chain rule in computation graphs: an example

- $q = F_1(x, y) = x + y; f = F_2(q, z) = q \cdot z$
- Representing function $F(x, y, z) = (x + y) \cdot z$



- How to calculate $\nabla F = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$?
- Interpretation of $\frac{\partial v}{\partial u}$: if node u is changed by 1 unit *independently*, how much does v change?

- Using *reverse topological order*, go over all variables in the graph
- $\frac{\partial f}{\partial q} = -4, \quad \frac{\partial f}{\partial z} = 3$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = -4$
- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = -4$

green: node values
red: derivatives

figure from Stanford cs231n

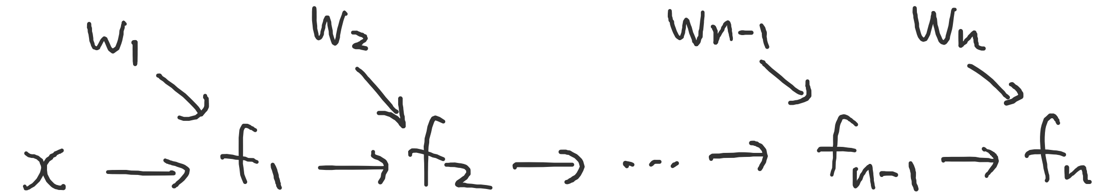
Chain rule in computation graphs: another example

- Assume all variables are scalars for the moment
- How to calculate $\frac{\partial f_n}{\partial w_1}$?
- We will calculate $\frac{\partial f_n}{\partial v}$ for all nodes v in the graph
- Calculation order:

  $\frac{\partial f_n}{\partial f_{n-1}}$ -> $\frac{\partial f_n}{\partial f_{n-2}}$ -> …. -> $\frac{\partial f_n}{\partial f_1}$ -> $\frac{\partial f_n}{\partial w_1}$

  - $\frac{\partial f_n}{\partial f_{n-2}} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}}$,
  - …
  - $\frac{\partial f_n}{\partial f_1} = \frac{\partial f_n}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1}$

  - Finally, $\frac{\partial f_n}{\partial w_1} = \frac{\partial f_n}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1}$

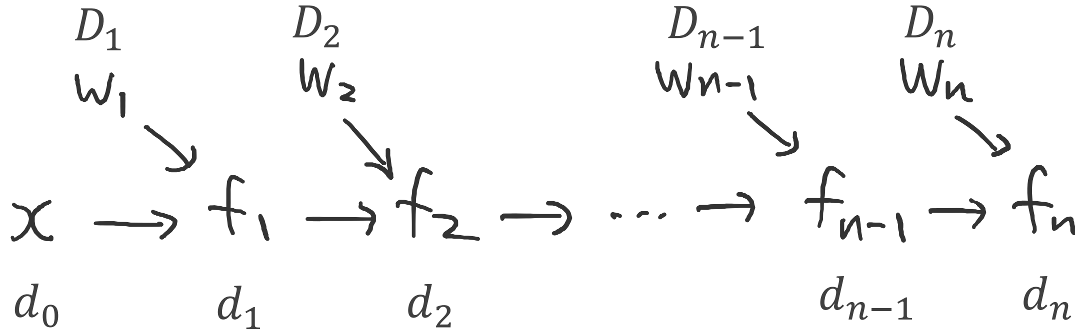


Chain rule in computation graphs: another example (cont'd)

- Moreover, the same calculation carries over when the variables are *vectors*
  - In this case, all partial derivatives should be interpreted as *Jacobians*
  - All multiplications are now *matrix multiplications*
- Intuition: $g(w) = Aw, f(v) = Bv, u = f(g(w)) = (B \cdot A) \cdot w$
- $\frac{\partial u}{\partial w} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} = B \cdot A$

- Previous example: $\underbrace{\frac{\partial f_n}{\partial f_{n-2}}}_{d_n \times d_{n-2}} = \underbrace{\frac{\partial f_n}{\partial f_{n-1}}}_{d_n \times d_{n-1}} \cdot \underbrace{\frac{\partial f_{n-1}}{\partial f_{n-2}}}_{d_{n-1} \times d_{n-2}}$

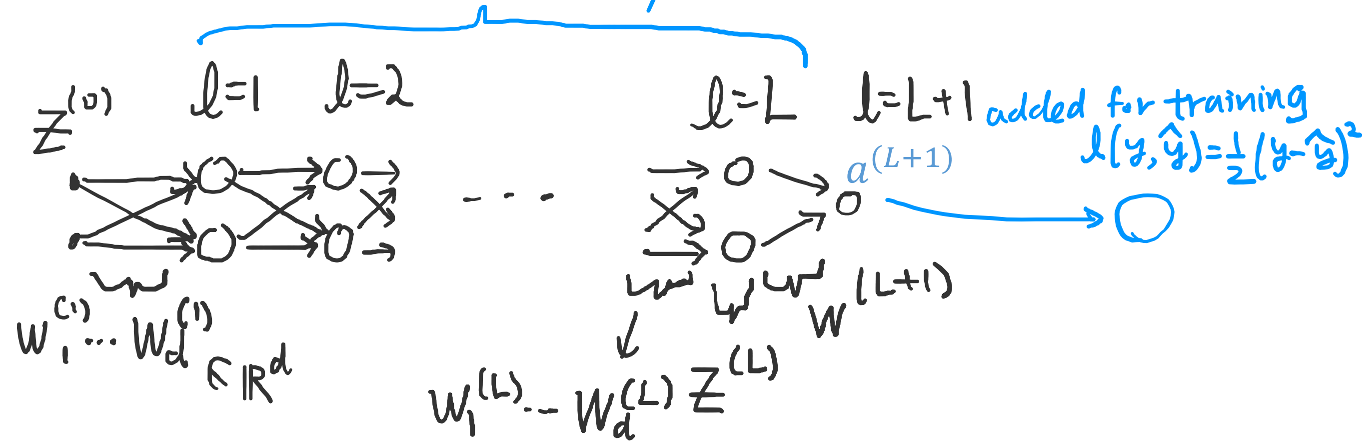
$$\underbrace{\frac{\partial f_n}{\partial w_1}}_{d_n \times D_1} = \underbrace{\frac{\partial f_n}{\partial f_1}}_{d_n \times d_1} \cdot \underbrace{\frac{\partial f_1}{\partial w_1}}_{d_1 \times D_1}$$



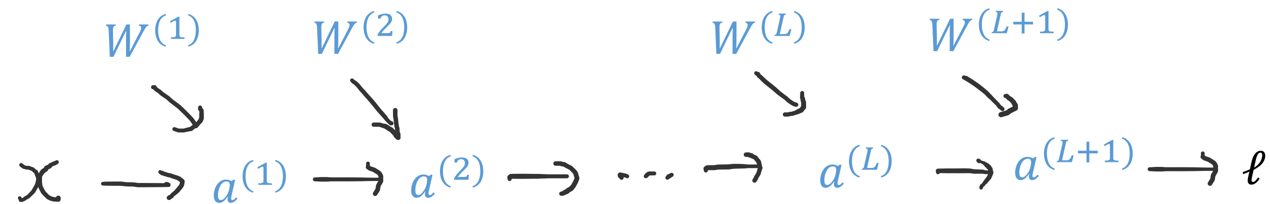
Important case study: loss gradients for MLP learning

- Input $x = (x_1, \ldots, x_d)$, label y
- Assume L hidden layers, layer l dimension n_l
- Assume regression task

(illustrating n_l =2 for all $l \leq L$)

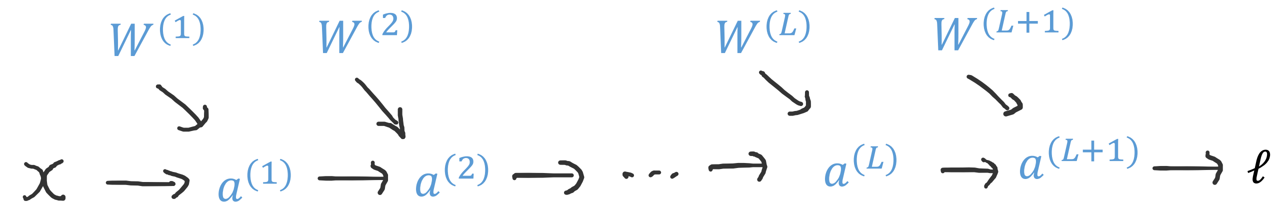


- Goal: compute the gradients of $\ell(y, \hat{y})$ w.r.t. weights
- Treating layers / weights as variables, getting equivalent computation graph:



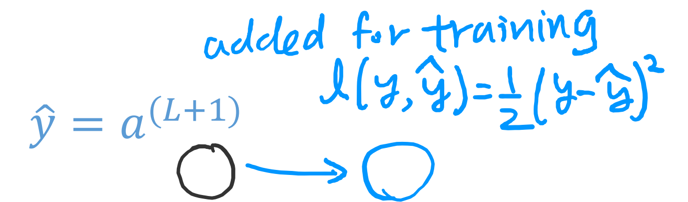
see also: https://courses.cs.washington.edu/courses/cse446/19wi/notes/backprop.pdf

Loss gradients for MLP learning (cont'd)

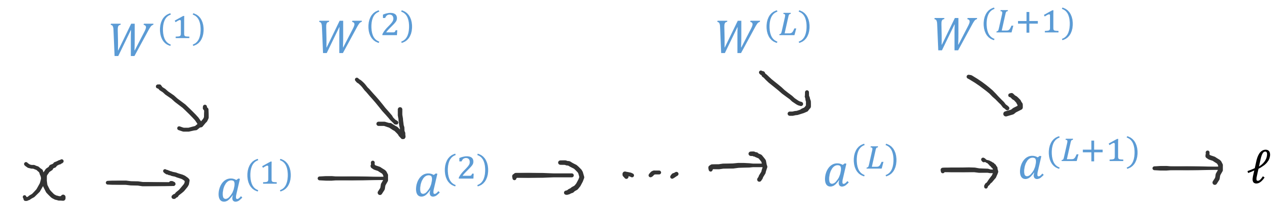


- Calculating the gradient of ℓ wrt variables in reverse topological order

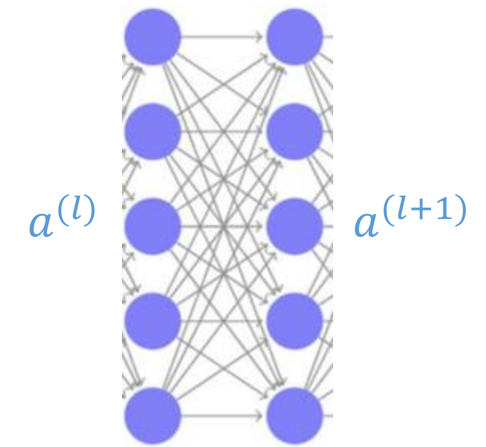
- First variable: $a^{(L+1)}$
  - $\frac{\partial l}{\partial a^{(L+1)}} = \frac{\partial l}{\partial \hat{y}} = \frac{\partial\left(\frac{1}{2}(\hat{y}-y)^2\right)}{\partial \hat{y}} = (\hat{y} - y) \in \mathbb{R}$



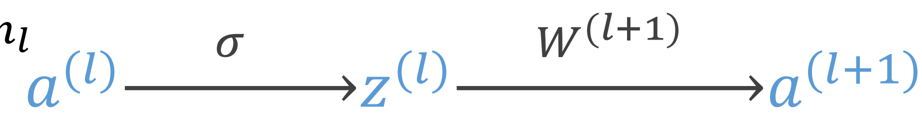
Loss gradients for MLP learning (cont'd)



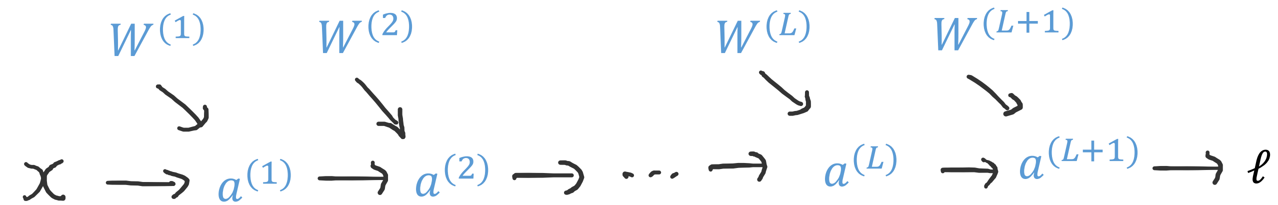
- Calculating the gradient of ℓ wrt variables in reverse topological order



- For every layer $l \geq 1$, deriving $\frac{\partial \ell}{\partial a^{(l)}}$ from $\frac{\partial \ell}{\partial a^{(l+1)}}$: $\quad \frac{\partial \ell}{\partial a^{(l)}} = \frac{\partial \ell}{\partial a^{(l+1)}} \cdot \frac{\partial a^{(l+1)}}{\partial a^{(l)}}$
  - $\frac{\partial \ell}{\partial a^{(l+1)}}$ is obtained from previous step
  - $\frac{\partial a^{(l+1)}}{\partial a^{(l)}} = \frac{\partial a^{(l+1)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial a^{(l)}} = W^{(l+1)} \cdot \text{diag}\left(\sigma'\left(a^{(l)}\right)\right) \in \mathbb{R}^{n_{l+1} \times n_l}$

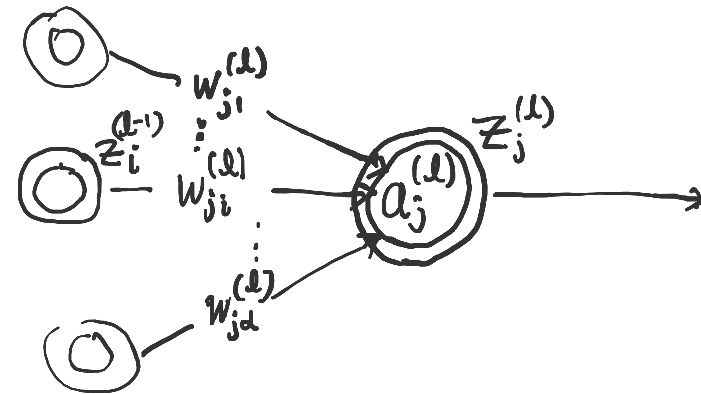


Loss gradients for MLP learning (cont'd)



- Calculating the gradient of ℓ wrt variables in reverse topological order

- Finally, deriving $\frac{\partial \ell}{\partial W^{(l)}}$ from $\frac{\partial \ell}{\partial a^{(l)}}$:
  - Observe that $w_{ji}^{(l)}$ only directly influences $a_j^{(l)}$ and no other nodes
  - $\frac{\partial \ell}{\partial w_{ji}^{(l)}} = \frac{\partial \ell}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = \frac{\partial \ell}{\partial a_j^{(l)}} \cdot z_i^{(l-1)}$
  - In matrix form: $\frac{\partial \ell}{\partial W^{(l)}} = \left(z^{(l-1)} \frac{\partial \ell}{\partial a^{(l)}}\right)^{\top} \in \mathbb{R}^{n_l \times n_{l-1}}$



Loss gradients for MLP learning: Summary

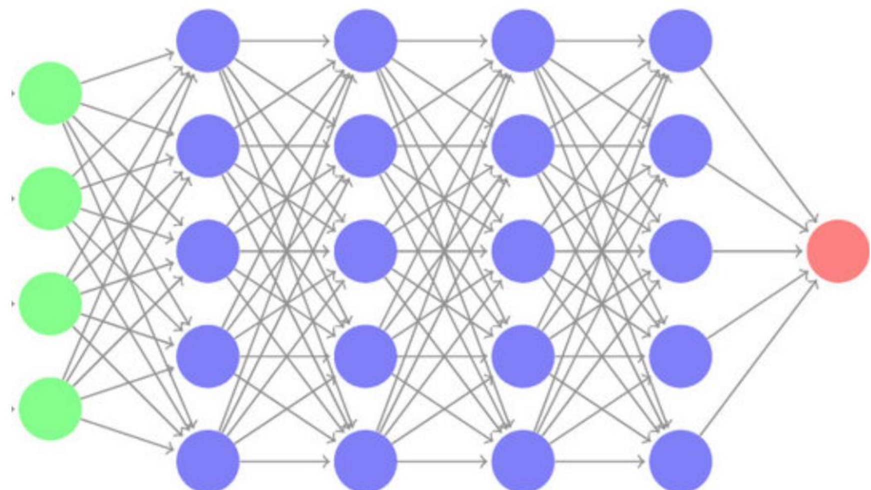
To calculate $\frac{\partial \ell}{\partial W}$ given example (x, y)

(Forward propagation)

- For $l = 1, .., L$:
  - Compute $a^{(l+1)}$ based on $a^{(l)}$
- Compute loss $\ell = \frac{1}{2}(\hat{y} - y)^2$, where $\hat{y} = a^{(L+1)}$

(Backward propagation)

- Last layer: $\frac{\partial l}{\partial a^{(L+1)}} = \frac{\partial l}{\partial \hat{y}} = (\hat{y} - y)$
- For $l = L, .., 1$:
  - $\frac{\partial \ell}{\partial a^{(l)}} = \frac{\partial \ell}{\partial a^{(l+1)}} \cdot W^{(l+1)} \cdot \mathrm{diag}\left(\sigma'\left(a^{(l)}\right)\right) \in \mathbb{R}^{1 \times n_l}$
  - $\frac{\partial \ell}{\partial W^{(l)}} = \left(z^{(l-1)} \frac{\partial \ell}{\partial a^{(l)}}\right)^{\mathsf{T}} \in \mathbb{R}^{n_l \times n_{l-1}}$

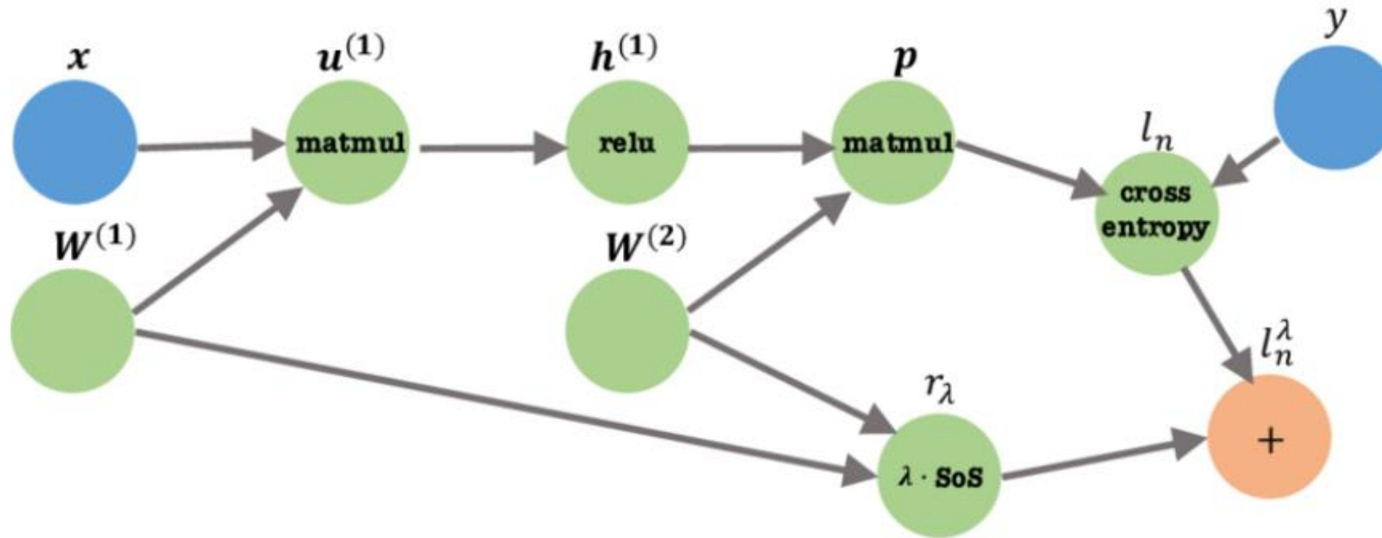


$W^{(1)}$ $W^{(2)}$ $W^{(L)}$ $W^{(L+1)}$

$x \rightarrow a^{(1)} \rightarrow a^{(2)} \rightarrow \cdots \rightarrow a^{(L)} \rightarrow a^{(L+1)} \rightarrow \ell$

Gradient calculations on general computation graphs

- So far, we have seen gradient calculations on special computation graphs, e.g. chains; trees
- How to perform gradient calculations for general computation graphs?

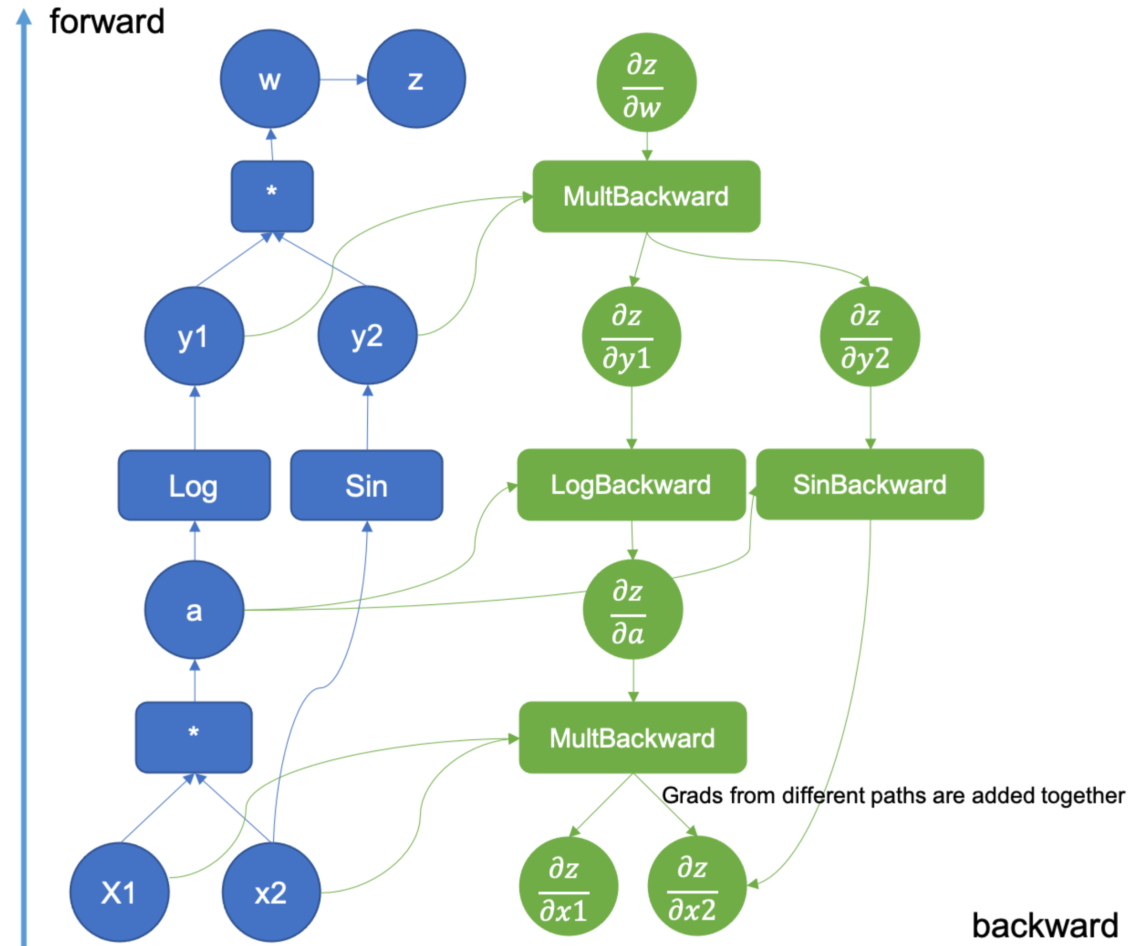


- Chain rule still applies (although the reason is much more subtle)
- For each node v (in reverse topological order):
  - $\frac{\partial \ell}{\partial v} = \sum_{u: v \text{ is a parent of } u} \frac{\partial \ell}{\partial u} \cdot \frac{\partial f_u}{\partial v}$ -- f_u: the *operation* at node u
  - The subtlety: for general graphs $\frac{\partial f_u}{\partial v} \neq \frac{\partial u}{\partial v}$ (although it is true for the examples we went over)

https://www.researchgate.net/figure/The-computational-graph-illustrates-the-loss-9-For-simplicity-we-omit-the-bias-terms_fig2_332368735

Backward gradient calculation is also a computation graph

- Taken from https://pytorch.org/blog/overview-of-pytorch-autograd-engine/



Automatic Differentiation (Autodiff)

- In fact, implementations of complicated functions (sin, cos, etc.) in computers are approximation computed by basic arithmetic operations.
- Reverse-mode auto-differentiation
  - input: a function f in a programming language (e.g., python) and its input x
  - output: the derivative of f at x
  - Draw the computation graph, evaluate values at nodes, then compute the gradients backwards.
- E.g., pytorch

Autodiff example

```
### following https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

import torch

#- turn on the gradient tracking
x = torch.tensor([[1.0,2],[3,4]],requires_grad=True)
# requires_grad     is False by default
print(x)

#-
y = x + 2
print(y) #- because we did `requires_grad`, it tracks who created it.

z = y * y * 3
out = z.mean()
print(z)
print(out)

z.retain_grad()
out.backward() # execute backward pass

print("#- grads")
print(y.grad)
print(z.grad)
print(x.grad)
```

Forming the computational graph

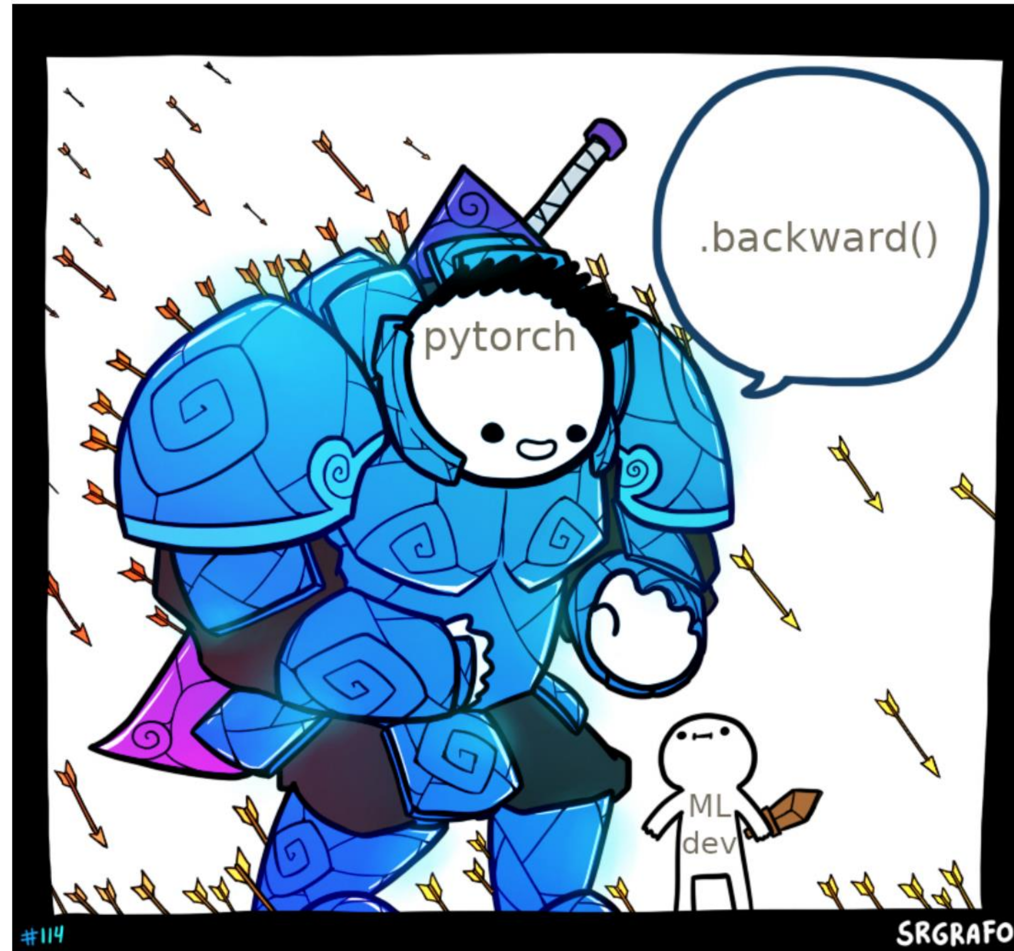
Computing $\frac{\partial \text{out}}{\partial v}$ for all v by backward calculation

```
tensor([[1., 2.],
        [3., 4.]], requires_grad=True)
tensor([[3., 4.],
        [5., 6.]], grad_fn=<AddBackward0>)
tensor([[ 27.,  48.],
        [ 75., 108.]], grad_fn=<MulBackward0>)
tensor(64.5000, grad_fn=<MeanBackward0>)
#- grads
None
tensor([[0.2500, 0.2500],
        [0.2500, 0.2500]])
tensor([[4.5000, 6.0000],
        [7.5000, 9.0000]])
<ipython-input-40-a3156942a32d>:22: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the gradient for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations.
  print(y.grad)
```

pytorch is optimized for obtaining the gradient of f w.r.t. the input only. If you want to obtain intermediate gradients, you need to use retain_grad().

Autodiff

(Taken from Matus Telgarsky's deep learning lecture: https://mjt.cs.illinois.edu/ml/lec8.pdf)



"Did you compute your gradients correctly?" — Leon Bottou.

When autodiff is not available..

- The standard answer to "my stochastic gradient descent is not converging":
  - Are you sure your derivatives are computed correctly? Debug it with **finite difference method**.

- Finite difference method for $f(x)$ where $x = (x_1, \dots, x_d) \in \mathbb{R}^d$.

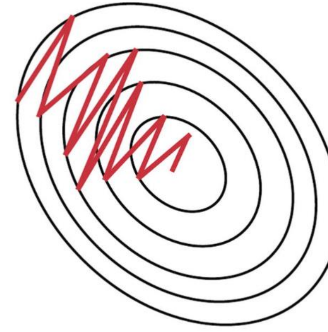
$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i}$$

for small enough h_i

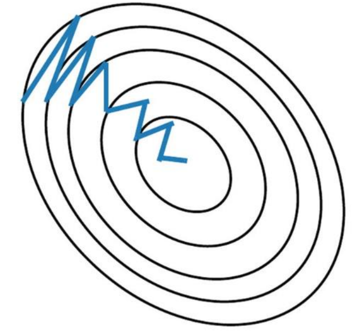
  - e_i is the indicator vector (equivalently, one-hot vector for the i-th coordinate)

Better optimization methods

- Issues with SGD: $w_{t+1} = w_t - \eta \nabla f(w_t)$
  - Oscillating iterates



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Solutions

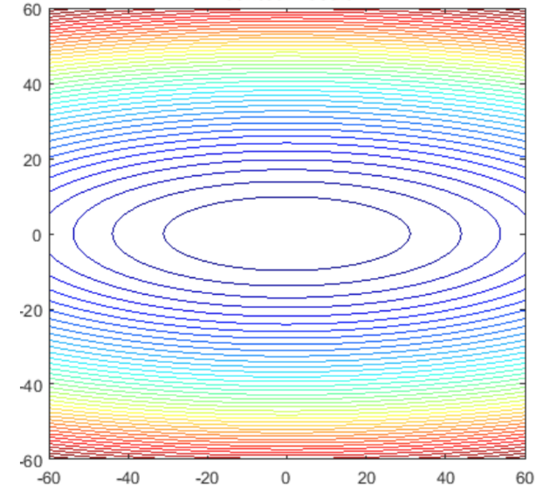
- Momentum: $w_{t+1} = w_t + \delta_t$ where $\delta_t = \mu \cdot \delta_{t-1} - \eta \nabla f(w_t)$ (e.g., $\mu \approx 0.99$)

- Nesterov Momentum: provably better (but for convex, nonstochastic (=batch) gradient descent)

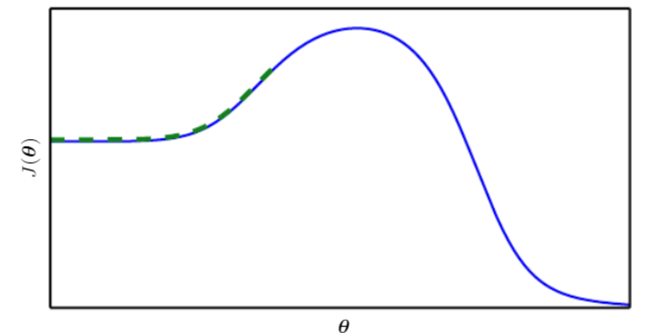
https://eloquentarduino.github.io/2020/04/stochastic-gradient-descent-on-your-microcontroller/

Better optimization methods

- Adagrad: $w_{t+1} = w_t + \delta_t,$ where $\delta_t = -\eta \cdot \left[\frac{\nabla_i f(w_t)}{\sqrt{A_i + \epsilon}}\right]_i$ and $A_i = \sum_{s=1}^{t} \left(\nabla_i f(w_t)\right)^2$, $\epsilon \approx 1e^{-7}$
  - Intuition: use a separate learning rate for each coordinate
  - $\epsilon > 0$ guards against the case where gradients are 0 at the beginning.
  - gradients get really small in the end!



- RMSProp: exponential moving averages of squared past gradients, $A_i \leftarrow \rho A_i + (1-\rho)\left(\nabla_i f(w_t)\right)^2$
  - adjustment so that A_i do not grow too large
  - Useful for nonconvex objectives
    - Optimizers may want to "forget" about historical info



Controversies on the optimization methods

- **Adam**: loosely speaking: combining RMSProp with momentum
  - The default setting seems to work very well.
  - Error in the convergence proof, and then a paper trying to fix the error again has an error..
  - The counterexample provided before is prominent in the **batch** gradient descent only.
  - Not clear how relevant it is in SGDs; there is no theoretical evidence that momentum gives you a better convergence rate.

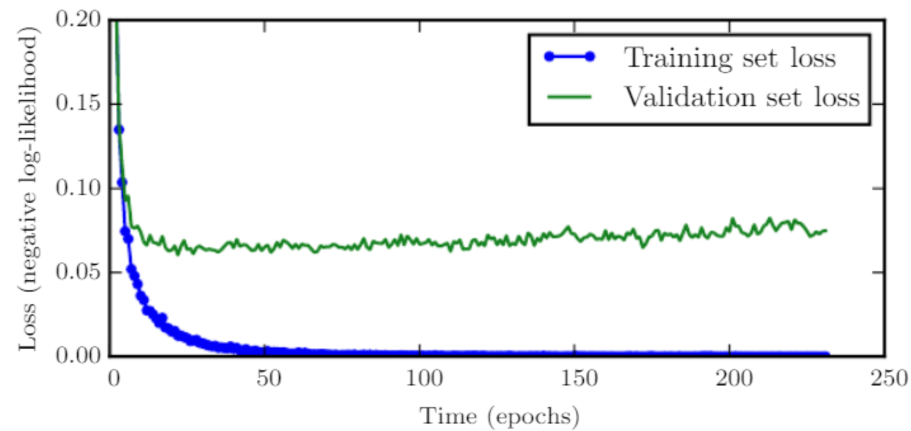
- NeurIPS'17
  - Tuning the learning rate correctly, SGD is no worse than adaptive gradient methods.
  - Others say that being less sensitive to tuning stepsize is exactly why we use adaptive gradient methods.

The Marginal Value of Adaptive Gradient Methods in Machine Learning

Ashia C. Wilson[♯], Rebecca Roelofs[♯], Mitchell Stern[♯], Nathan Srebro[†], and Benjamin Recht[♯]
{ashia,roelofs,mitchell}@berkeley.edu, nati@ttic.edu, brecht@berkeley.edu

Optimization tips

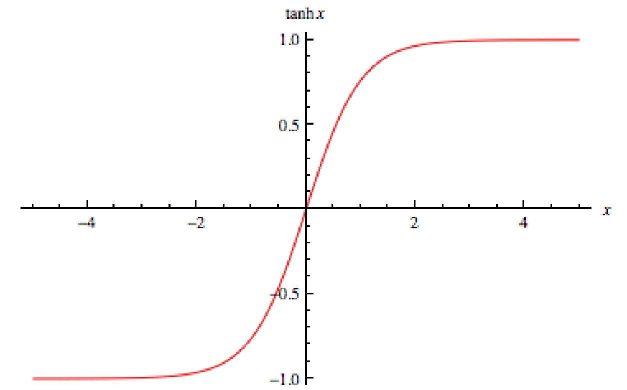
- Learning rate schedule
  - Simple way: halve it after each epoch
  - Let t be the iteration (=step) index (could also use epoch index)
  - $\eta = a \exp(-kt)$) for some a and k
  - $\eta = a/(1 + kt)$
  - Often, people monitor validation set error, and when it starts to saturate, divide the stepsize by 10
- Convergence
  - Monitor train set acc, **validation set accuracy** (or train/validation loss function value).



The vanishing / exploding gradient problem

- $\frac{\partial \ell}{\partial a^{(1)}} = (\hat{y} - y) \cdot W^{(L+1)} \cdot \operatorname{diag}\left(\sigma'\left(a^{(L)}\right)\right) \cdot \ldots \cdot W^{(2)} \cdot \operatorname{diag}\left(\sigma'\left(a^{(1)}\right)\right)$

- With L layers => $\frac{\partial \ell}{\partial a^{(1)}}$ can be exponentially small => vanishing gradient
  - Moreover: some $a^{(L)}$'s are "saturated", making $\sigma'\left(a^{(L)}\right)$ close to 0



- Likewise, $\frac{\partial \ell}{\partial a^{(1)}}$ can be exponentially large => exploding gradient

- As we will see, there are methods that alleviate this issue (e.g. good initialization, batch normalization, skip connection, …)

Tips and tricks

Weight initialization

- Never do symmetric weight initialization! => Otherwise, results in identical nodes
- Naïve: 0.01 * unit gaussian random number
- Xavier initialization (2010): (unit gaussian) / sqrt(fan_in)
  - fan_in: how many nodes in the previous layer
  - works for tanh() but not for ReLU
  - Intuition: for every node i, keeps input $\sum_{j=1}^{n_l} w_{ij}\, h_j = \Theta(1)$
- He et al. (2015): For ReLU, do (unit gaussian) / sqrt(fan_in/2)

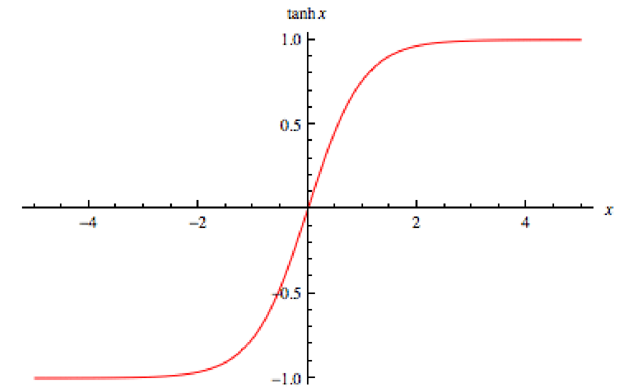


Image preprocessing

- Two popular: (i) subtract average image or (ii) subtract per-channel (∈{R,G,B}) average.

see more tricks: https://www.youtube.com/watch?v=gYpoJMlgyXA&list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC&index=5&ab_channel=AndrejKarpathy

Training loss

- Minimize: $\frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$ // f is the entire network
- It's common to add L2 regularizer: $\frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i) + \frac{\lambda}{2}\sum_{l,j,k}\left(w_{k,j}^{(l)}\right)^2$
- For regression: $\ell(f(x), y) = \frac{1}{2}(f(x) - y)^2$
- For binary classification: $\ell(f(x), y) = \log\big(1 + \exp(-y \cdot f(x))\big)$
- For K-class classification: logistic loss = cross entropy loss + softmax layer
  - Have K output nodes
  - $\ell\left(\vec{f}, y\right) = -\log\left(\sigma\left(\vec{f}\right)_y\right)$ (cross-entropy loss, aka negative log likelihood)

    with $\sigma\left(\vec{f}\right)_c = \frac{e^{f_c}}{\sum_{j=1}^{C} e^{f_j}}$, $c = 1, \ldots, K$ (softmax transformation)
  - Reduces to binary logistic loss if $K = 2$, $\vec{f}_2 = 0$
- Quick question: can you derive the formulae for $\frac{\partial \ell(\vec{f}, y)}{\partial W}$ for these new ℓ's?

l: layer index
j: input node index
k: output node index

$\hat{y} \in \{-1,1\}$

$\vec{f} \in \mathbb{R}^K$ is the output of NN

Other regularizations

- Data augmentation
  - Flip
  - Rotation
  - Crop
  - Scale
  - Translation
  - Adding Gaussian noise
  - More modern: do style transfer with generative models

Horizontal Flip

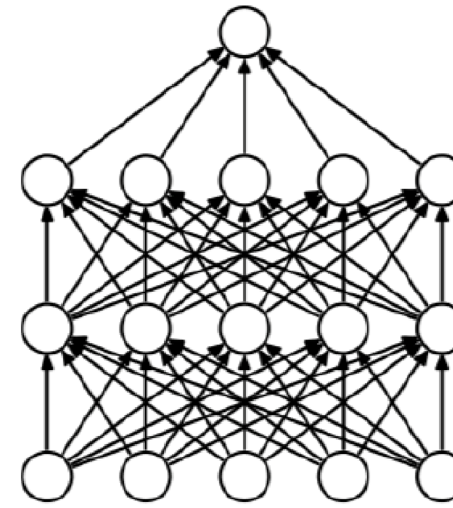
Crop

Rotate

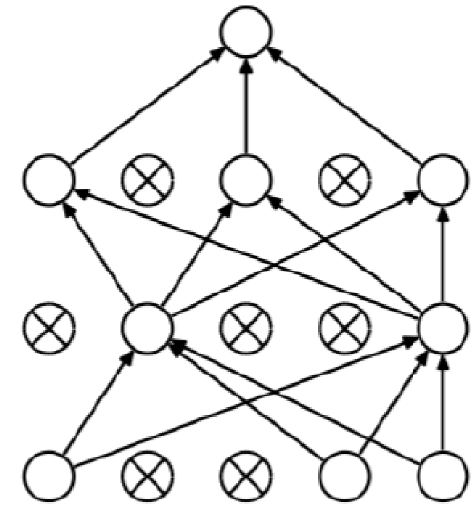
Figure from Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7, by Kevin Wang

Other regularizations

- Dropout
  - *For each example*, only train with a random subset of the nodes
  - Can be viewed as training an ensemble of models



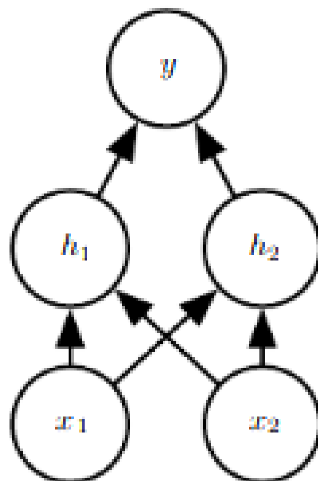
(a) Standard Neural Net



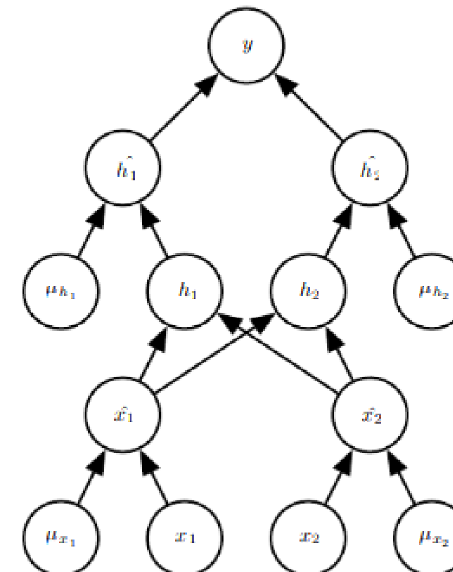
(b) After applying dropout.

Figures from Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al. JMLR 2014

- Computation graph view (W omitted):
  - Original



With Dropout

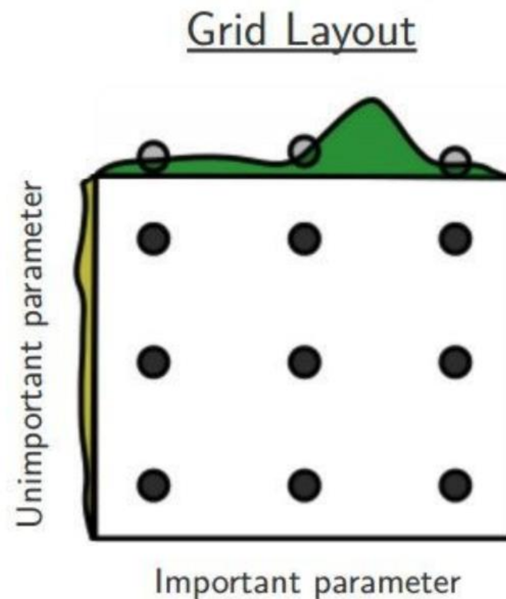


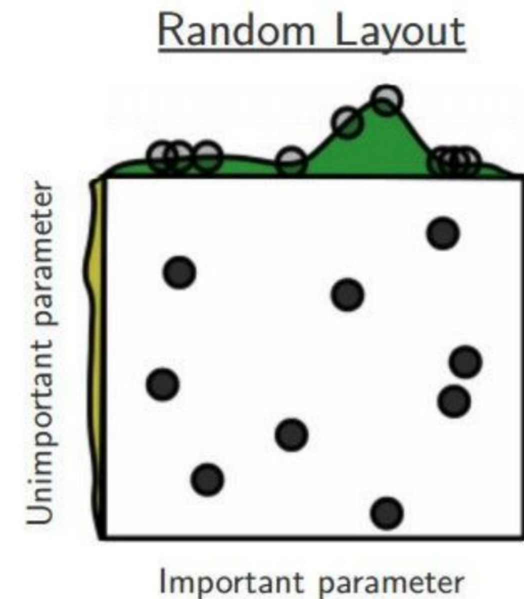
Hyperparameter search

- Those from architectural parameter (# of layers / # of hidden units)
- Those from optimization (stepsize, momentum discount factor)
- Those from regularization (L2 reg. strength)
- Use: Random Layout.

Ex: optimizing $f(x, y) = g(x) + h(y)$

x, y are hyperparameters
g has large variation
h has little variation

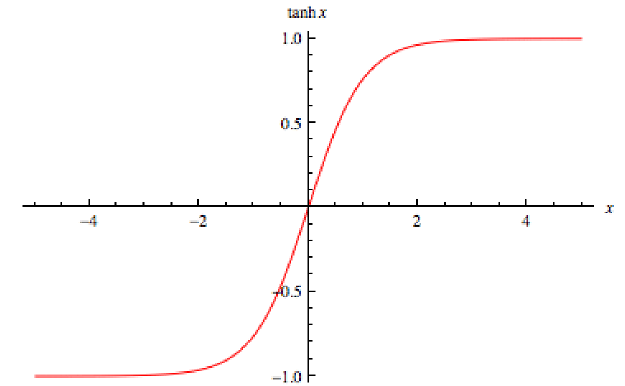




Random search for hyper-parameter optimization, Bergstra and Bengio, 2012

Batch normalization

- Recall: optimization is easier when the inputs of each layer is within constant interval, say [-2,2]
- Can we ``enforce'' this by modifying the NN's design?



- Key idea: Let's add a layer that normalizes the inputs!

- Recall minibatch SGD
  - Computes gradients for m data points, then updates the weights.
    - => gradients are more stable
    - e.g., ResNet (2015): m=256.

  - Can we ensure that within a batch, for a layer, most of the inputs are "standardized"?

Batch normalization layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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

All calculations are in the computational graph, (and are subject to autodiff / backprop)

Common heuristic: add BN layers *before* activation functions instead of after

Why is 'scale and shift' $\gamma x + \beta$ helpful?
This new parameterization may allow easier optimization

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Ioffe, Szegedy. ICML 2015

Batch normalization

- Improves 'gradient flow' => reduce dead ReLUs / vanishing gradients in tanh.
- Less sensitive to the initialization => allows higher learning rates.

- One twist for the test time
  - Once converged, gather statistics throughout the dataset to compute the final μ and σ^2 ($\exists$ variations of this)
  - On any test example x: use the μ and σ^2 computed above computed in training time

Why does it work?

(2015)

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

(2018)

How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu

Dimitris Tsipras* MIT tsipras@mit.edu

Andrew Ilyas* MIT ailyas@mit.edu

Aleksander Mądry MIT madry@mit.edu

Abstract

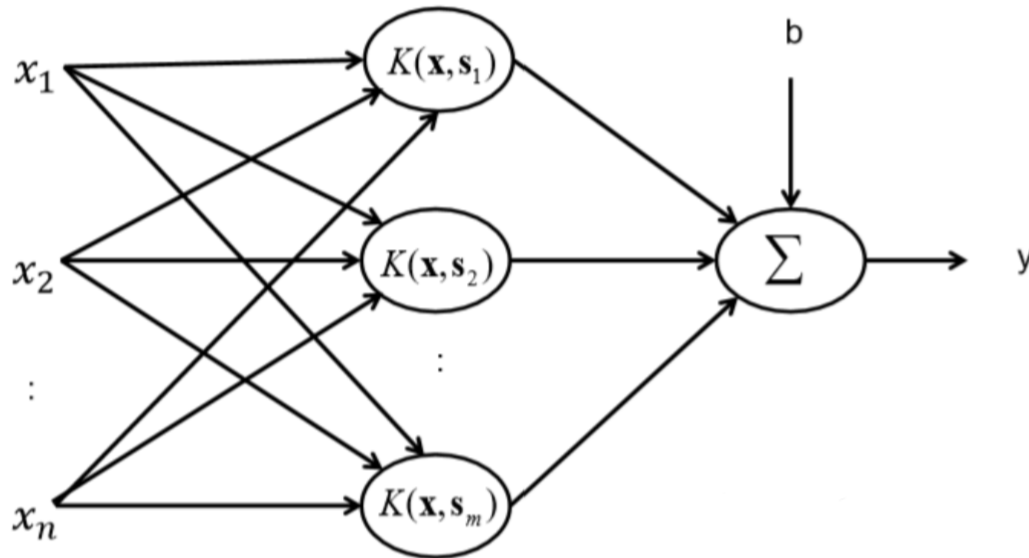
Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

"it makes the optimization landscape significantly smoother"

Kernel methods can be viewed a special case of NNs.

- Recall: kernels compute the feature map, followed by linear operations.

- $h(x) = \text{sign}(\langle w^*, \phi(x) \rangle + b^*) = \text{sign}(\sum_i \alpha_i^* y_i K(x_i, x) + b^*)$



Q: how is this different from NNs?

Next lecture (11/14)

- More on neural networks: convolutional neural networks (CNNs)

- Assigned reading: PyTorch tutorial from Berkeley CS 285: https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-3.pdf (go over the jupyter notebook at the end)
  - Can help your HW4

- Optional reading:
  - Matus Telgarsky's PyTorch tutorial: https://mjt.cs.illinois.edu/ml/ (the jupyter notebook)
  - Contains some more advanced materials, e.g. batchnorm