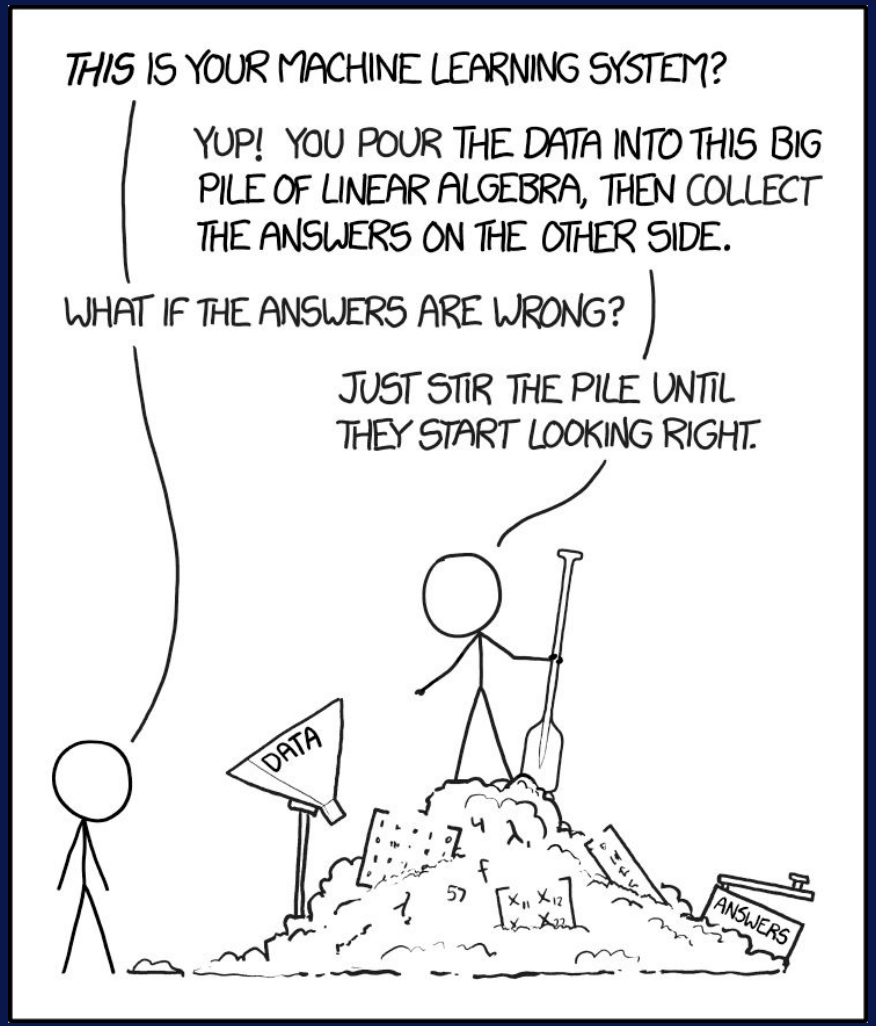


Source:
https://xkcd.com/1838/

When poll is active, respond at **PollEv.com/konradkordin059**

Text **KONRADKORDIN059** to **22333** once to join

Are you logging into poll everywhere so we can track attendance?

Yes

No

Well, maybe?

CIS 522: Lecture 2R

Introduction to neural nets
01/23/20

Penn Engineering

Course Announcements

- HW0 was due on 01/21 before the class, HW1 is due on 1/30 at midnight
- Final OH schedule is posted.
- Debugging Session is this Saturday from 2-4pm
- Lectures are recorded (under Canvas, Class Recordings)
- Exam is **April 2nd** during class time, location TBD
- Reminder to make a private Piazza post if you'd like Canvas access to submit the homework as an auditor.

Correction: Tensorflow 2.0

Now does have dynamic compute graphs

Different mechanism from Pytorch though

For the exam

You will not be tested on anecdotes

Or peripheral proofs

Or arcane aspects of intuition

You do need to understand the central concepts

The kinds of things that you clearly should know to be a credible DL engineer

Thank you for feedback

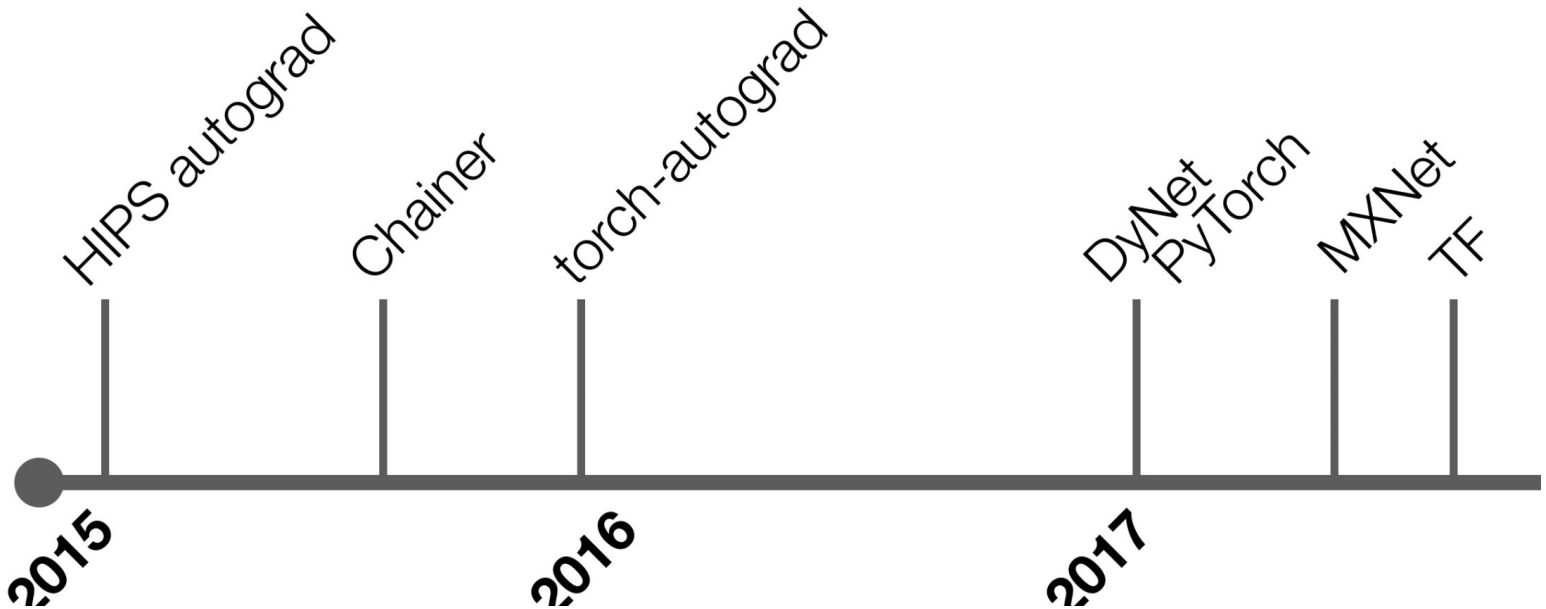
I should be slower

Things should be more interactive

More technical details

And, again, how does this compute graph thing work?

Operator overload - the pytorch way of AD



Recapitulation of compute graphs

Neural network (L=f(x,y,W)):

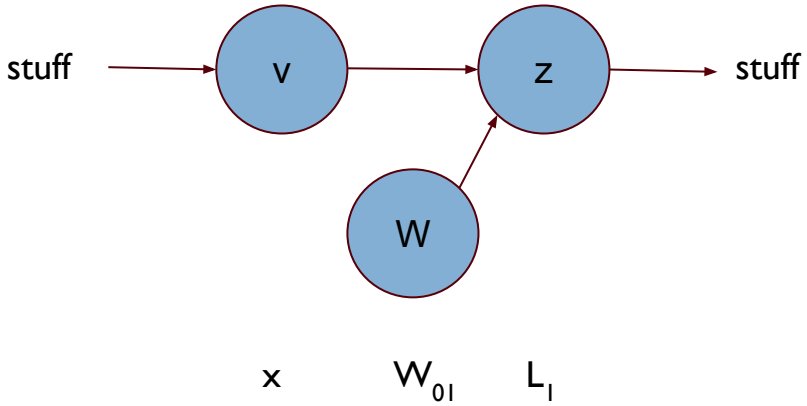
$$L_1 = W_{01}x$$

$$L_1 = ReLU(L_1)$$

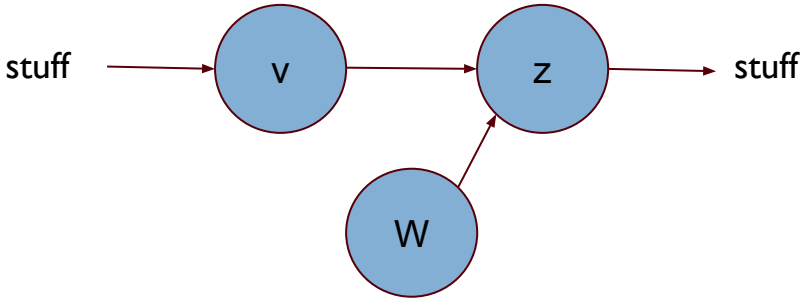
$$L_2 = W_{12}L_1$$

$$L = (L_2 - y)^T(L_2 - y)$$

Lets zoom into $L_1 = W_{01}x$



Chain rule in practice

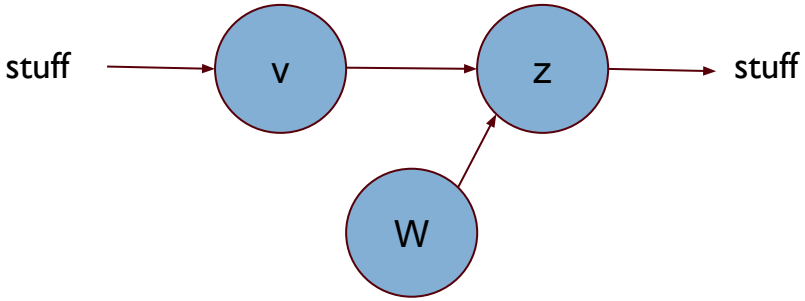


x W_{01} L_1

$$z = Wv$$

$$\frac{\partial L}{\partial v_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial v_i}$$

So what do we need to store?



$$z = Wv$$

$$\frac{\partial L}{\partial v_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial v_i}$$

$$\frac{\partial L}{\partial v_i} = \sum_j \frac{\partial L}{\partial z_j} W_{ij}$$

How operator overload solves such problems

```
class MyMM(torch.autograd.Function):
        def forward(ctx, v, W):
            ctx.save_for_backward(W)
            return W*v
        def backward(ctx, grad_output):
            W=ctx.saved_tensors
            grad=W.T*grad_output
            return grad
```

How operator overload solves such problems

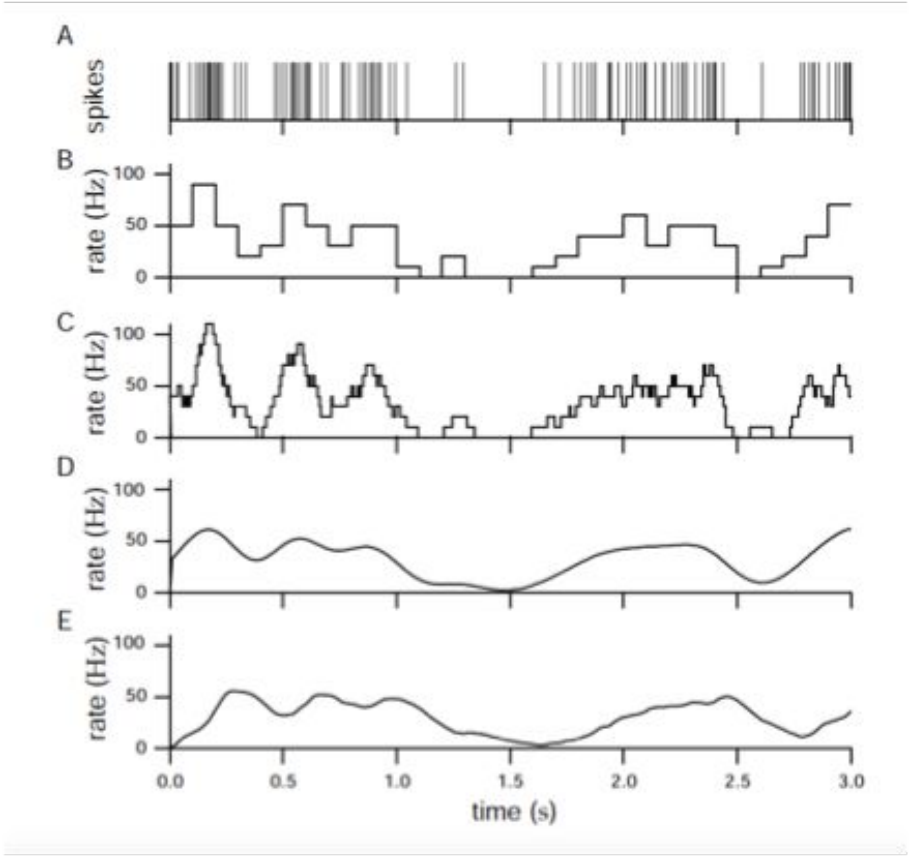
```
class ReLU(torch.autograd.Function):
        def forward(ctx, input):
            ctx.save_for_backward(input)
            return input.clamp(min=0)
        def backward(ctx, grad_output):
            input=ctx.saved_tensors
            grad_input=grad_output.clone()
            grad_input[input<0]=0
            return grad_input
```

Towards nonlinear neural networks

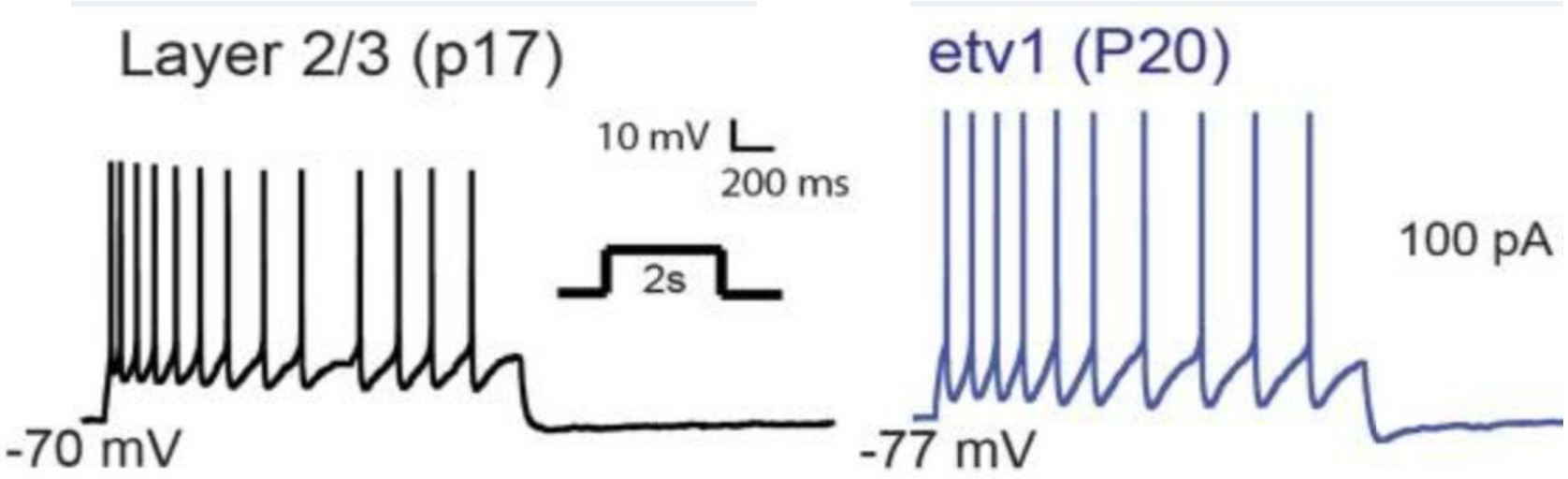
Neuroscience of simple computations

Penn Engineering

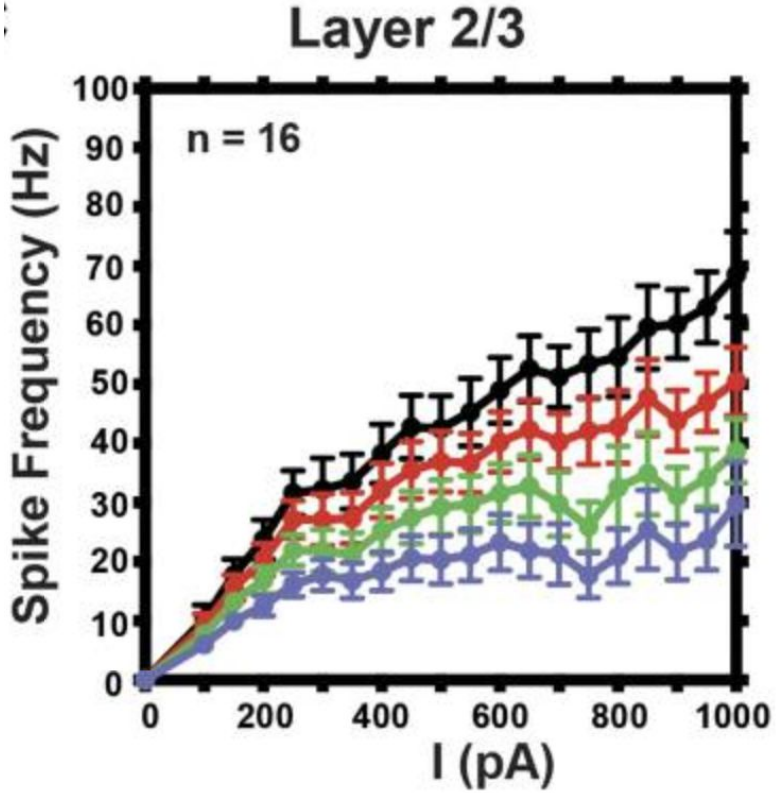
Neural codes

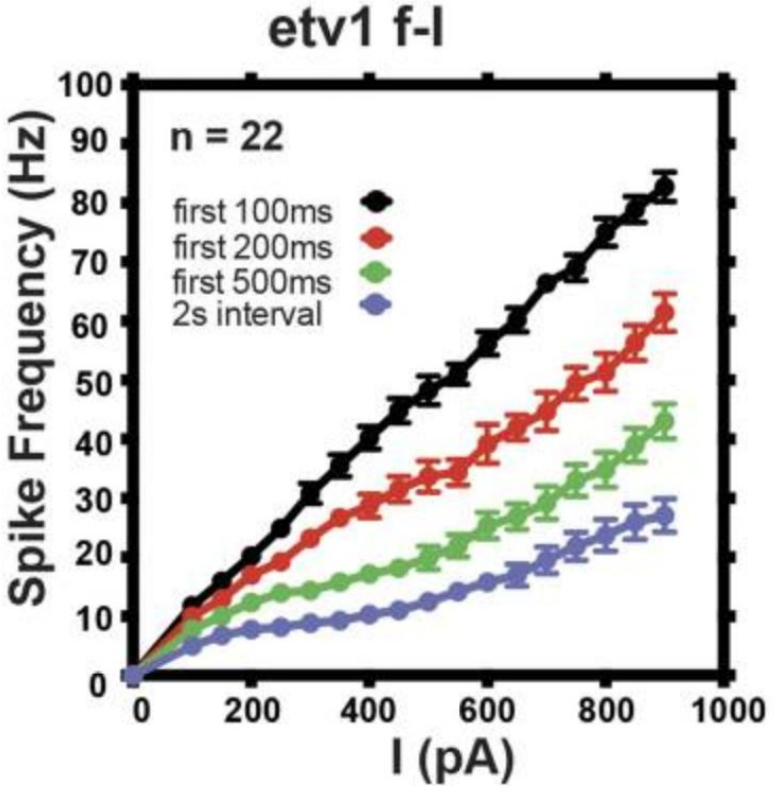


Neural responses to inputs

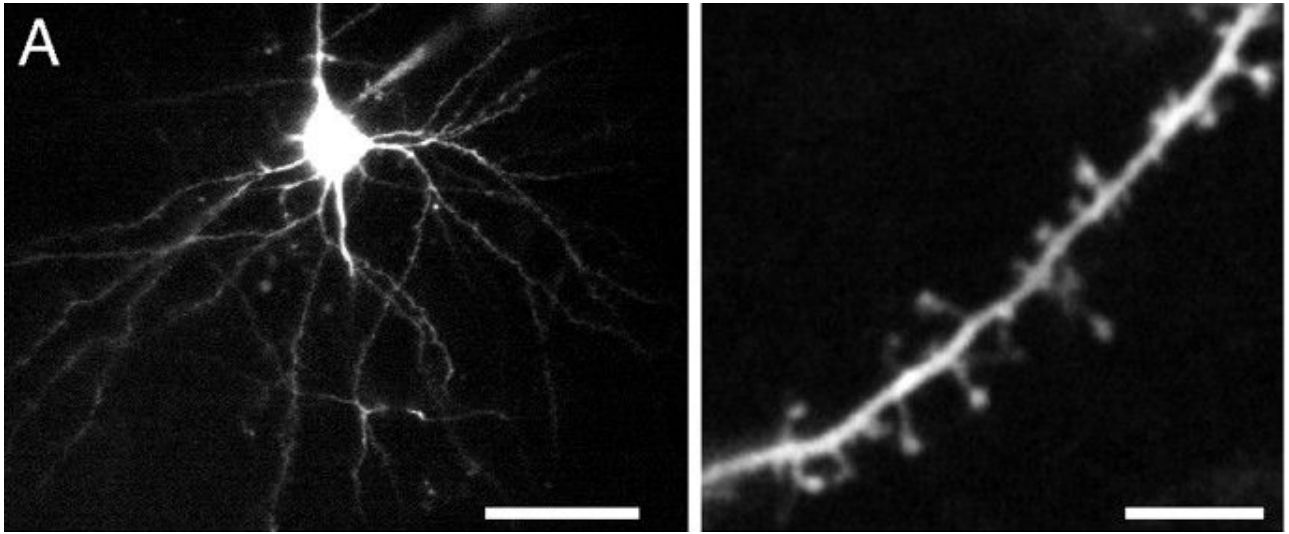


Nonlinear (a bit)

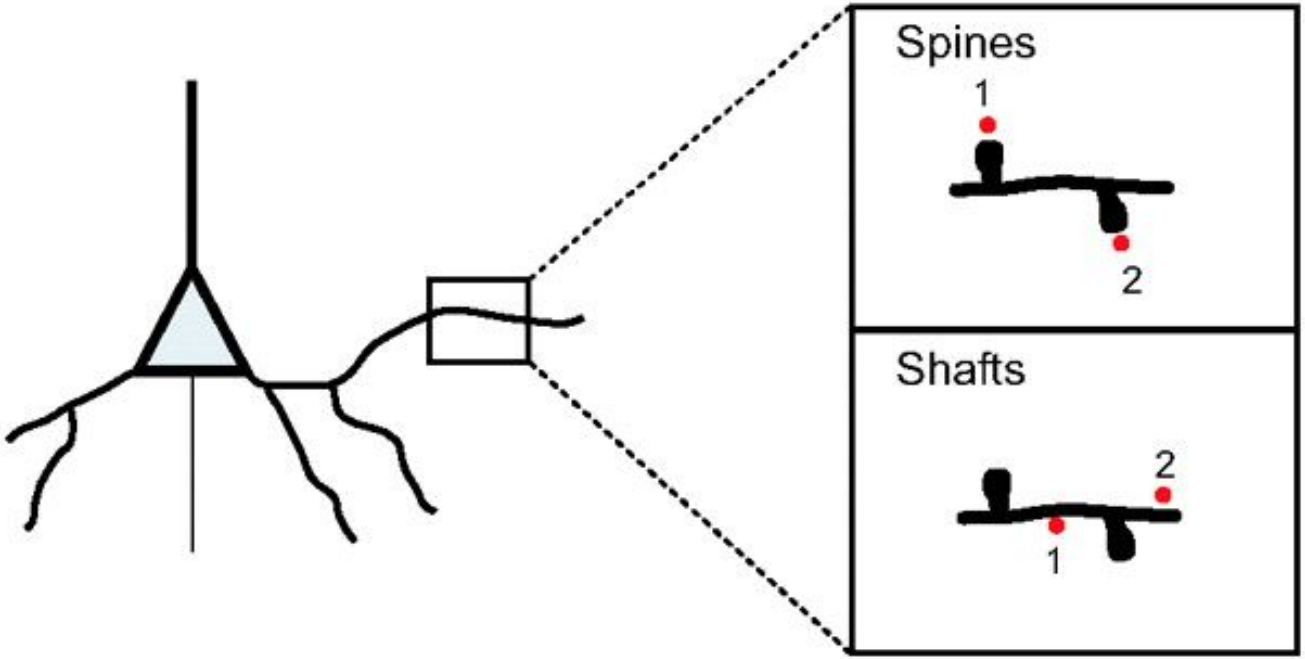


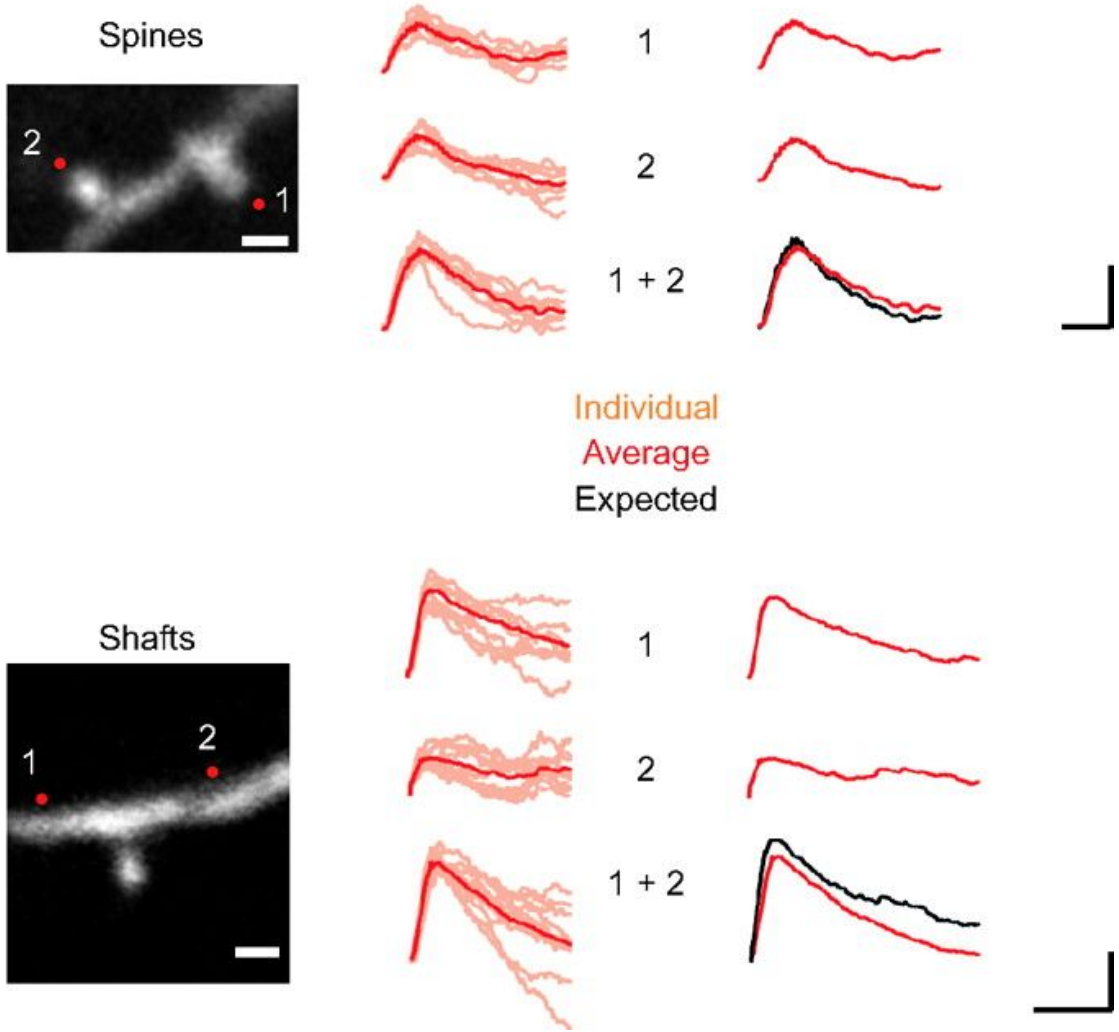


Linear integration within dendritic trees

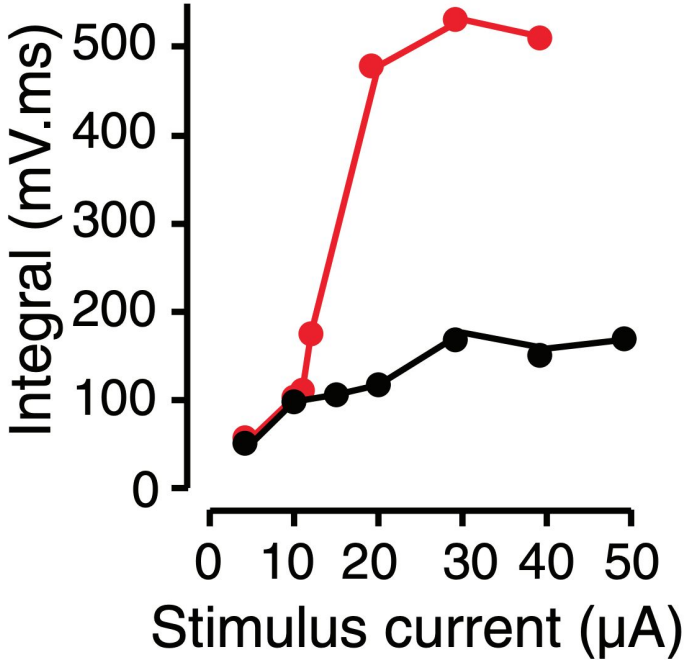


Linear integration within dendritic trees

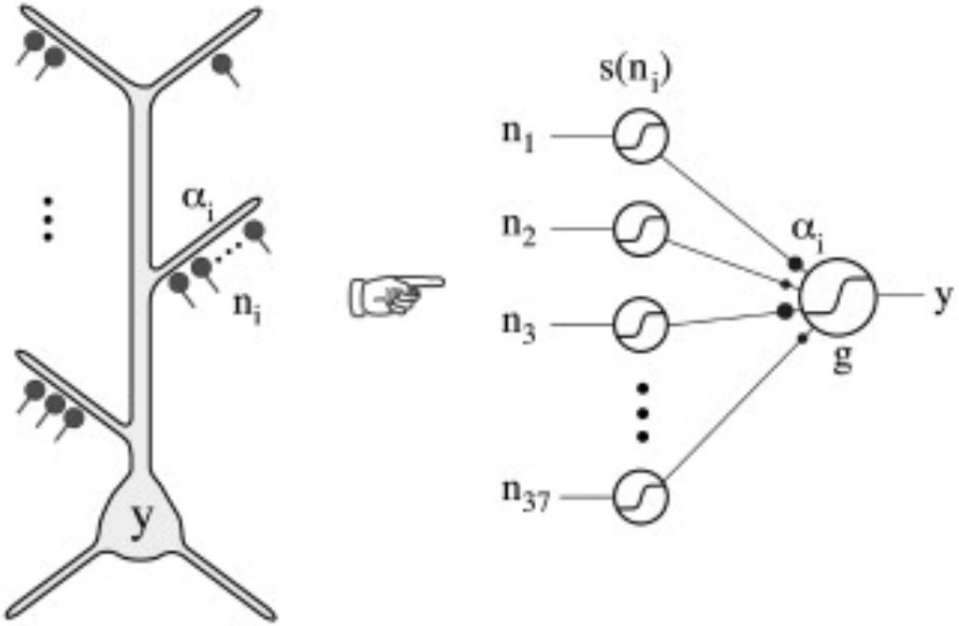



Spines
2
1
1
2
1 + 2
Individual
Average
Expected
Shafts
1
2
1
2
1 + 2

Or maybe not - NMDA spikes



So then what does a neuron do?



How does neuroscience help for DL in your opinion.

Main source of inspiration

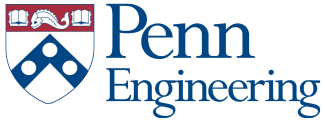
Source of crazy people

Source of loss functions

Inspiration. For some. I guess

Distraction

From linear to nonlinear neural networks



What is supervised learning?

- Suppose we have data points (x_1, y_1), (x_2, y_2), …, (x_s, y_s)

  (both x_i, y_i may be multivariate)

- If we see a new x, we want to predict the y.
- Implicitly, we are assuming there is an unknown function $y(x) = y$.
- Examples of useful functions y:

y(image) = cat or dog, y(audio signal) = sentence, y(chess position) = good move to make

What is (supervised) learning?

- It's very hard to approximate *y(x)* since we can only work with limited data(training data). Hence, we approximate the intractable function *y(x)* by fitting it with a family of easier-to-express functions.

A first cut at learning - linear functions

- The easiest way to fit data: $y(x) \approx Wx$
- Mean squared error (MSE) - a natural loss function:

$$\text{argmax}_W \frac{1}{s} \sum_{i=1}^{s} ||y_i - Wx_i||_2^2$$

A first cut at learning - linear functions

- The easiest way to fit data: $y(x) \approx Wx$
- Mean squared error (MSE) - a natural loss function:

$$\text{argmax}_W \frac{1}{s} \sum_{i=1}^{s} ||y_i - Wx_i||_2^2$$

- This is just linear regression, and it can be solved exactly:

$$W_{\text{opt}} = (X^T X)^{-1} X^T Y$$

Perceptrons

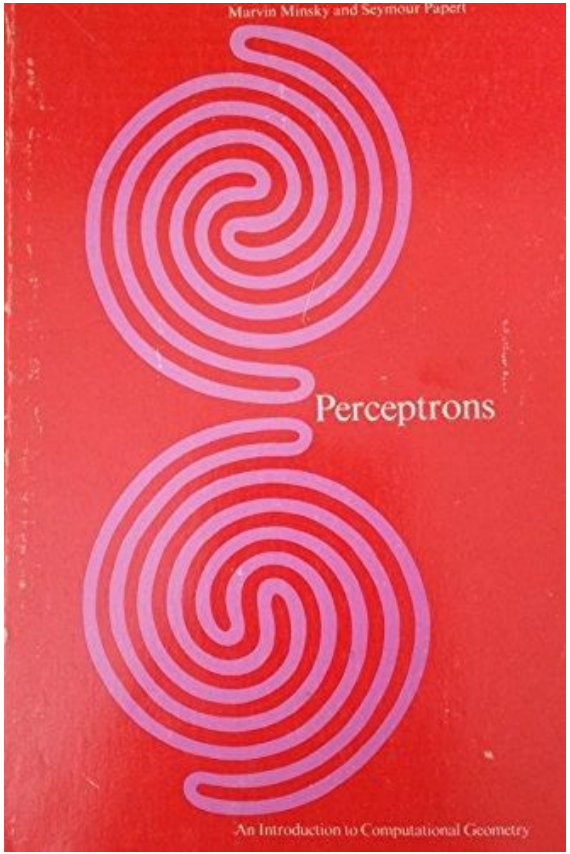
- What about binary classification? I.e., if $y(x) = \pm 1$.
- We can fit a function of the form:

$$y(x) \approx \text{sign}(Wx)$$

- Perceptron Learning Algorithm .
  - Always works if the data is actually linearly separable.
  - Otherwise may not work at all.
  - Best solution depends on a loss function.

Do perceptrons actually work?

- Not really, but sometimes surprisingly well.
- It's a good first cut.
- Very rarely are data actually linearly separable.



Do perceptrons actually work?

- MNIST image dataset for classification of digits
- Essentially the simplest deep learning test dataset.
- Random performance (10 classes) is 10%
- Perceptrons (trained by backpropagation) get ~93%.
- How is that possible? High-dimensional input (*d=784* here) makes it a bit easier.

Multilayer perceptrons

Penn Engineering

What is the simplest thing that isn't linear?

- We have tried fitting linear functions.
- What about iterating linear functions?

$$y(x) \approx W_k \cdots W_2 W_1 x$$

What is the simplest thing that isn't linear?

- We have tried fitting linear functions.
- What about iterating linear functions?

$$y(x) \approx W_k \cdots W_2 W_1 x$$

- Unfortunately, that's still a linear function.
- If we want to iterate, we can alternate a simple nonlinear function σ :

$$y(x) \approx W_k \sigma(\cdots \sigma(W_2 \sigma(W_1 x)) \cdots)$$

(the nonlinear function is usually univariate and applied component-wise to vectors)

What is the simplest thing that isn't linear?

- More generally, we can add *bias* terms:

$$y(x) \approx N(x) = W_k \sigma(\cdots \sigma(W_2 \sigma(W_1 x + b_1) + b_2) \cdots + b_{k-1}) + b_k$$

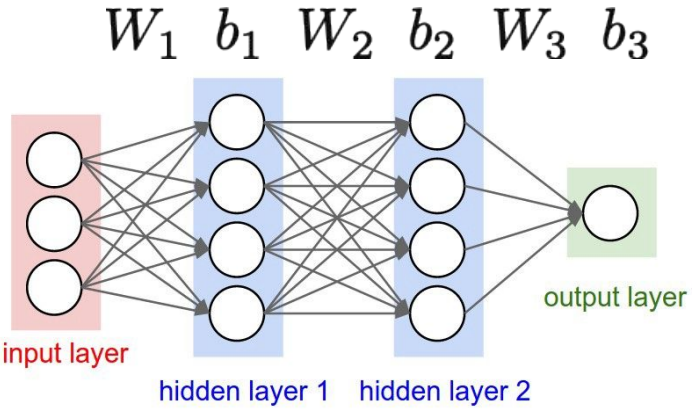
- The function $N(x)$ is called a *multilayer perceptron* (MLP) or *fully-connected network*.

- These are the most basic general-purpose "deep learning"-style neural networks.

A bit like a biological neuron

- Neurons have one output
- They linearly integrate on their dendrites
- They have a ReLU-like nonlinearity
- And they are hierarchically organized

Architecture of a neural net



- A ***layer*** is one of the intermediate vectors.
- A ***neuron*** is one of the entries of a layer vect
- The ***depth*** is the number of layers.
- The ***width*** of a layer is the layer's dimension.
- The ***weights*** are the coefficients of the matrices W_k.
- The ***biases*** are the terms b_k.
- The ***activation function*** or ***non-linearity*** is the function σ .

$$y(x) \approx N(x) = W_k\sigma(\cdots\sigma(W_2\sigma(W_1x + b_1) + b_2)\cdots + b_{k-1}) + b_k$$

How to deal with discrete output?

- Often (e.g. in classification problems), the output will take on discrete values
  - Does an image show the digit 0, 1, etc? (10 possibilities)
  - Which word comes next in a sentence? (finite number of possible words)
- The simplest way to deal with this is *one-hot encoding*.

$$\text{label } L \qquad \underbrace{(0, \ldots, 0, 1}_{L\text{th entry}}, 0 \ldots, 0)$$

- Then, the output of the network is a length-C vector if there are C possible classes.

- This has the added advantage of essentially encoding *confidence* (or, alternatively, a *probability distribution*): $(p_1, p_2, \ldots, p_L, \ldots, p_C)$

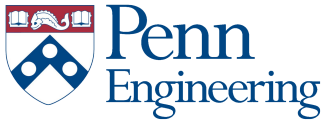
How to deal with discrete output?

- A disadvantage of one-hot encoding is if there may be a *lot* of possibilities.
- It also doesn't capture information about structure in the outputs.
- In natural language processing (NLP), often Word2Vec representations are used instead (will be covered more later).

Training a neural network

- *Training* a network means changing the weights and biases to match the data.
- It's generally done by some form of backpropagation - calculating the gradient of the loss with respect to the weights and biases.
- The weights and biases change along the gradient to decrease the loss.
- Ingredients (more on these later):
  a. **Activation function / nonlinearity**
  b. **Loss function (e.g. mean squared error)**
  c. Network architecture
  d. Initialization of weights and biases
  e. Optimizer (e.g. SGD)

Choices for the non-linearity/ activation function



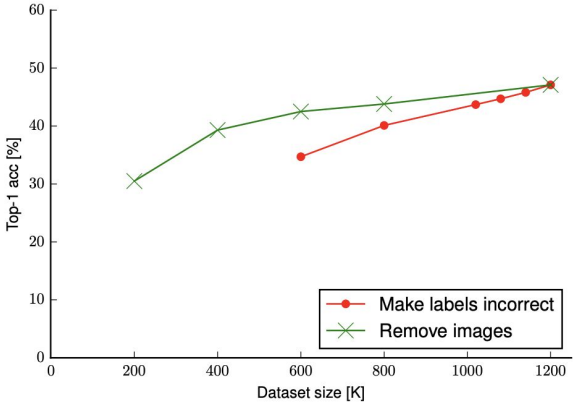
A history of activation functions

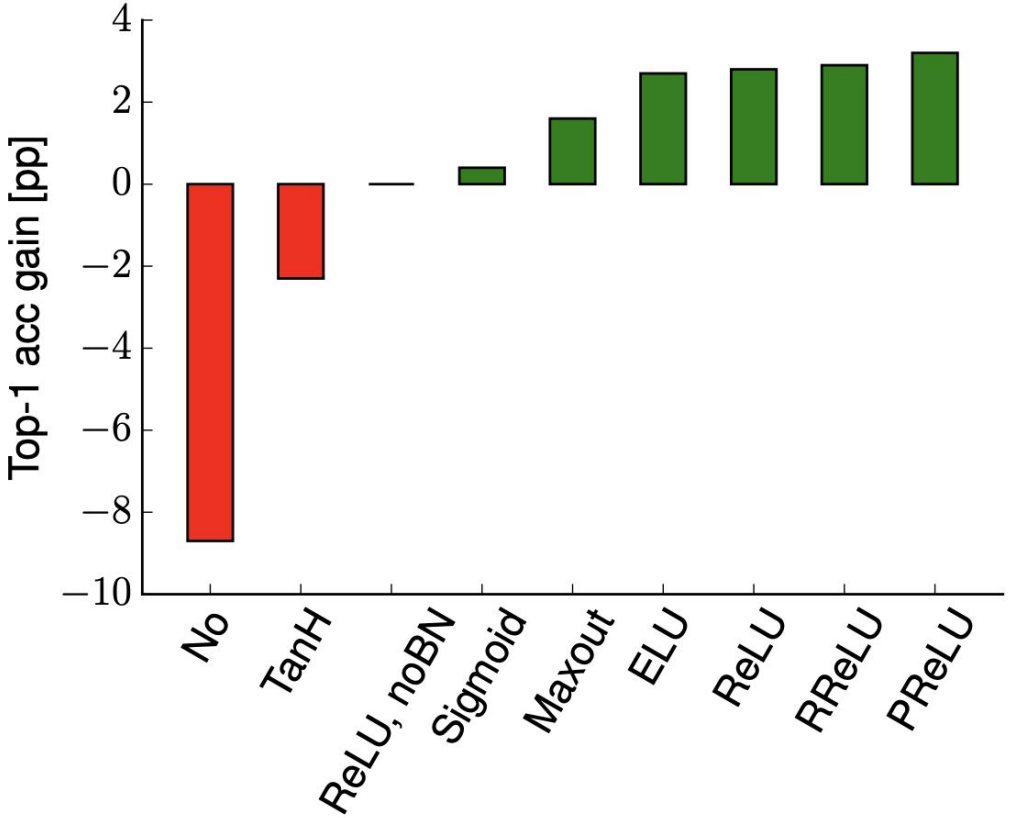
Table 3: Non-linearities tested.

Name	**Formula**	**Year**
none	$y = x$	-
sigmoid	$y = \frac{1}{1+e^{-x}}$	1986
tanh	$y = \frac{e^{2x}-1}{e^{2x}+1}$	1986
ReLU	$y = \max(x, 0)$	2010
(centered) SoftPlus	$y = \ln(e^x + 1) - \ln 2$	2011
LReLU	$y = \max(x, \alpha x), \alpha \approx 0.01$	2011
maxout	$y = \max(W_1 x + b_1, W_2 x + b_2)$	2013
APL	$y = \max(x,0) + \sum_{s=1}^{S} a_i^s \max(0, -x + b_i^s)$	2014
VLReLU	$y = \max(x, \alpha x), \alpha \in 0.1, 0.5$	2014
RReLU	$y = \max(x, \alpha x), \alpha = \text{random}(0.1, 0.5)$	2015
PReLU	$y = \max(x, \alpha x), \alpha$ is learnable	2015
ELU	$y = x$, if $x \geq 0$, else $\alpha(e^x - 1)$	2015

Systematic evaluation of CNN advances on the ImageNet, Mishkin et al

Performance

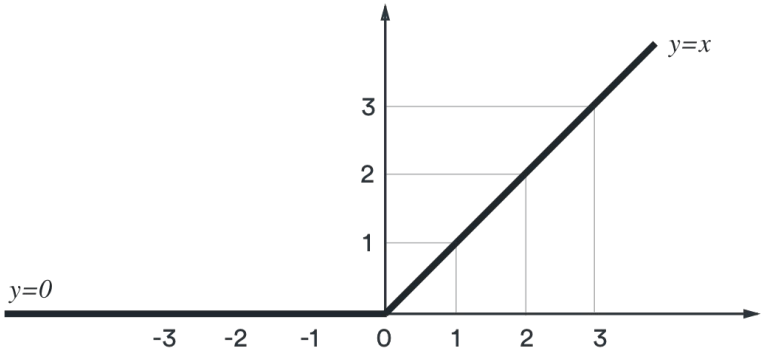




Systematic evaluation of CNN advances on the ImageNet, Mishkin et al

ReLU

$$\sigma(z) = \max(0, z)$$



- Very fast computation.
- Most common activation function.
- The resulting functions *N(x)* are *piecewise linear.*
- Not actually differentiable everywhere, but this is often okay.
- More of a problem is when the derivative is zero, “dead ReLU” problem
  - But very unlikely for a ReLU to be “dead” for all inputs.

A hierarchical network (MLP) of ReLUs

Implements higher order polynomials

Implements locally linear functions

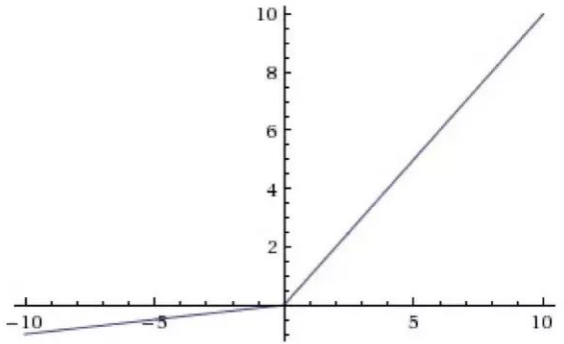
Implements any possible function

None of the above

Leaky ReLU

$$\sigma(z) = \max(\epsilon z, z), \text{ for small } \epsilon > 0$$

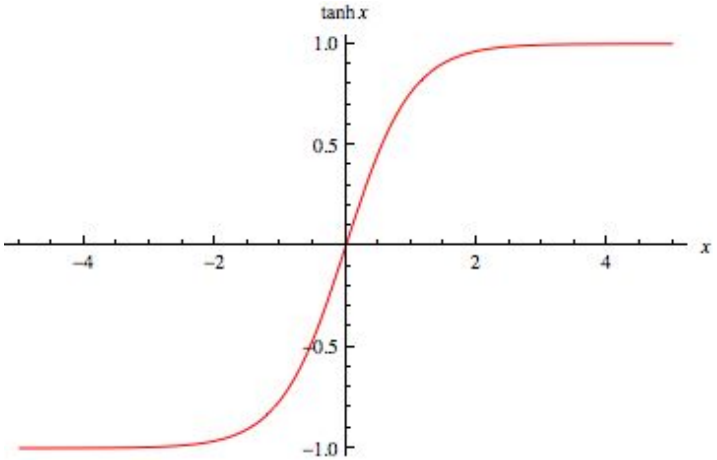
- Fixes the problem of "dead ReLUs."
- Vanishing/exploding gradients
- Not used very much in practice compared to ReLU.



Tanh

$$\sigma(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

- “hyperbolic tangent”, often pronounced like “tanch”.
- Smoothes out discontinuities.
- Used to be more popular (bad neuro)
- replaced by ReLU.
- Intuitions may relate to probabilities
- Works well on some meta learning problems



Cases where twice differentiable is needed

Metalearning:

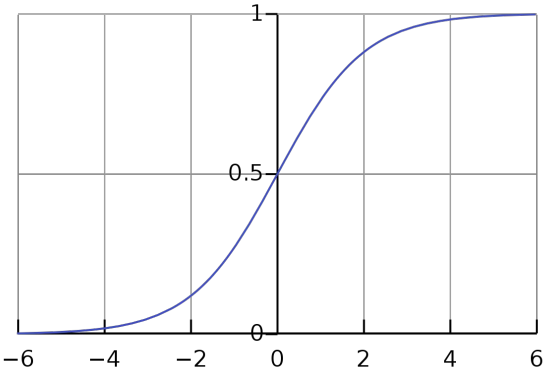
Learning how to learn (derivatives of gradients after parameters of learning process

Or simply wanting to use a method that utilizes the Hessian

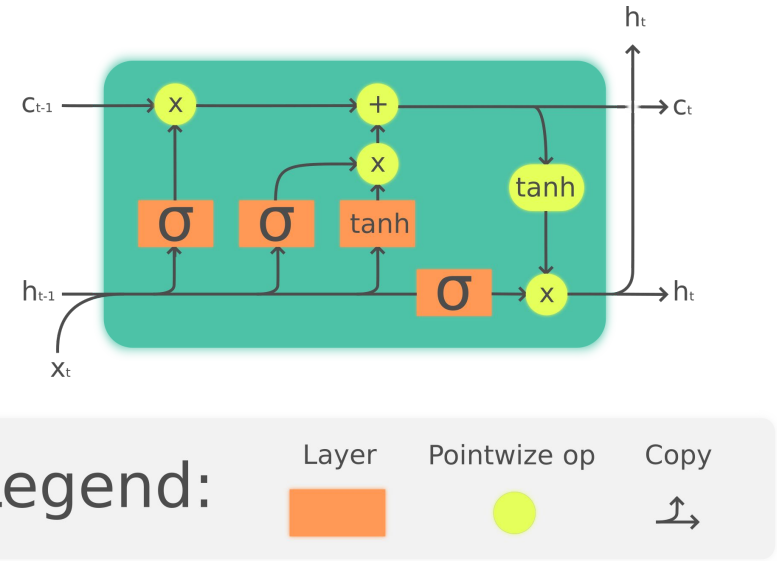
Logistic Sigmoid

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

- Similar to tanh, but asymptotes to 0 and 1, instead of -1 and 1.
- Like tanh, used in special cases nowadays.



As part of problems where 0 and 1 are special (e.g. LSTM,



Softmax

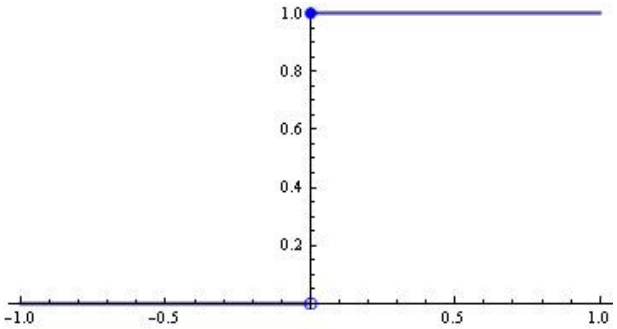
$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- Applied to an entire vector, not component-wise.
- Entries of the output sum to 1.
- The most common way to predict a probability distribution.
- Typically used in the final layer of networks for classification tasks.

Step function?

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

- What is the problem here? (if using backpropagation)
- The gradient is zero everywhere - gradient updates don't do anything.



What if a deep network were actually linear?

- What if the activation function is linear?
- The overall network would be a linear transformation.
- But any learning algorithms might give different results.
- Also, the resulting weight matrices would be factored into different layers.
- Used sometimes as a simple case to prove results, and in some neuro models.
- Various authors, esp. Andrew Saxe, have considered these networks, but not really used in practice.

What if a deep network were actually linear?

We've shown that deep linear networks — as implemented using floating-point arithmetic — are not actually linear and can perform nonlinear computation. We used evolution strategies to find parameters in linear networks that exploit this trait, letting us solve non-trivial problems.

99% on MNIST

Loss functions

Penn Engineering

Mean squared error

$$\mathcal{L}_N(X, Y) = \frac{1}{s} \sum_{i=1}^{s} ||y_i - N(x_i)||_2^2$$

- Often used if the network is predicting a scalar.

Cross-entropy

$$\mathcal{L}_N(X,Y) = \frac{1}{s}\sum_{i=1}^{s}\left(-\sum_j y_{ij}\log N(x_i)_j\right)$$

- Most common loss function for classification tasks.
- It is a measure of distance between two probability distributions.
- For classification, generally the target vector is one-hot encoded, which means that y_{ij} is 1 where x_i belongs to class *j*, and is otherwise 0.
- **Be careful** to apply cross-entropy loss only to probabilities! (e.g. after softmax)
- In PyTorch, softmax and cross-entropy can be applied in a single step.
- **Be careful** in that case not to apply it twice :)

Cosine similarity

$$\mathcal{L}_N(X,Y) = \frac{1}{s}\sum_{i=1}^{s}\frac{y_i \cdot N(x_i)}{||y_i||_2||N(x_i)||_2}$$

- Computes similarity between directions of two vectors.
- Often used in NLP contexts. For example, the two vectors might be the frequencies of different words.

How could lecture 3 have been better?

Re-asking the question question

Bad:

NLP

Vision

Good:

Quantify how much taking a deep learning course increases lifetime earnings.

Build an NLP system that runs through a deck of google slides, crosschecks them with r/sarcasm and then makes them funny

Describe a question you would like to answer