Neural Network Basics

Chris Dyer

https://github.com/redpony/seaml_intro_nns

DeepMind

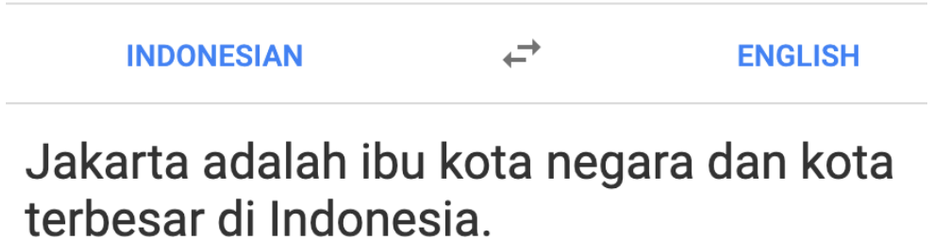
Carnegie Mellon University

July 9, 2019

Southeast Asia Machine Learning School

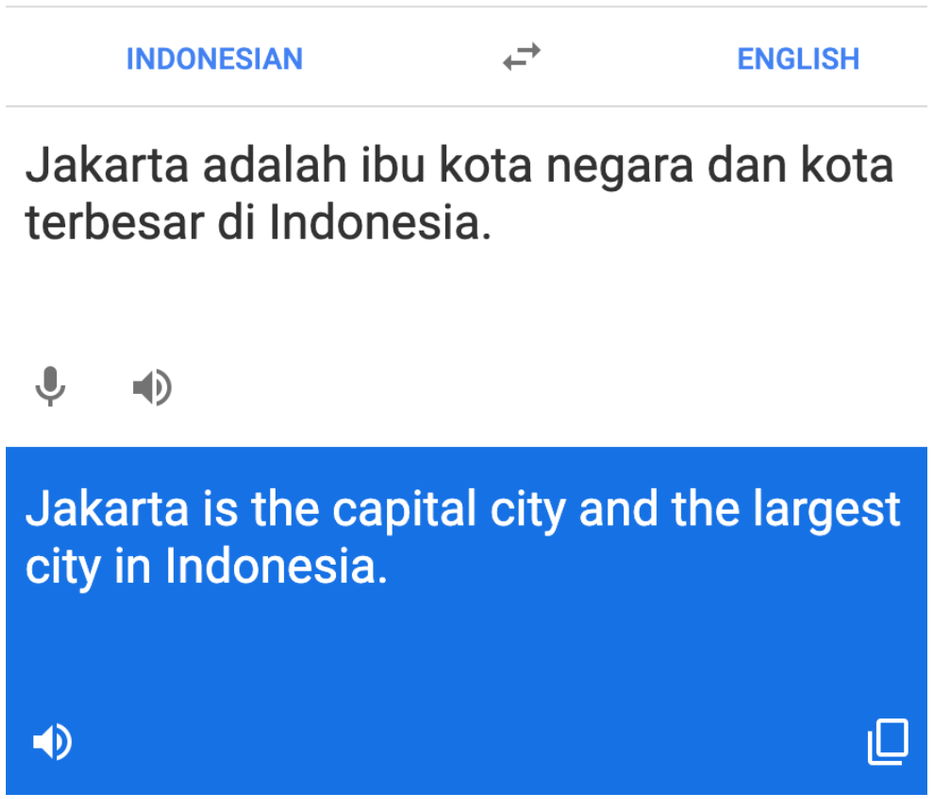
What can we do with neural networks?

What can we do with neural networks?

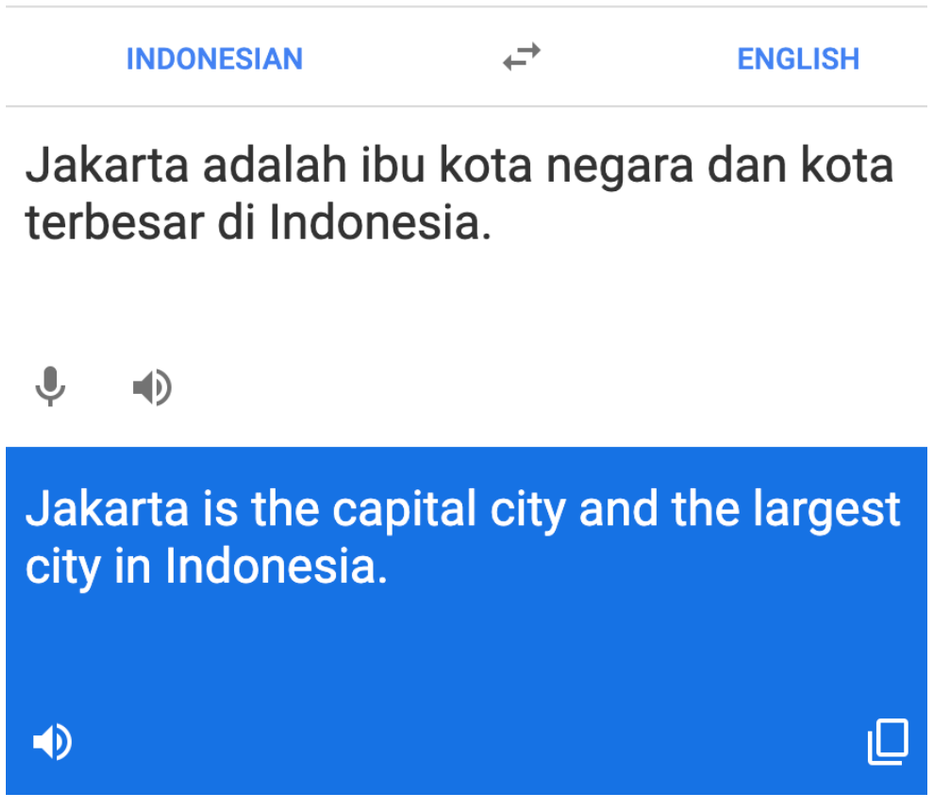


Google Translate

What can we do with neural networks?



What can we do with neural networks?

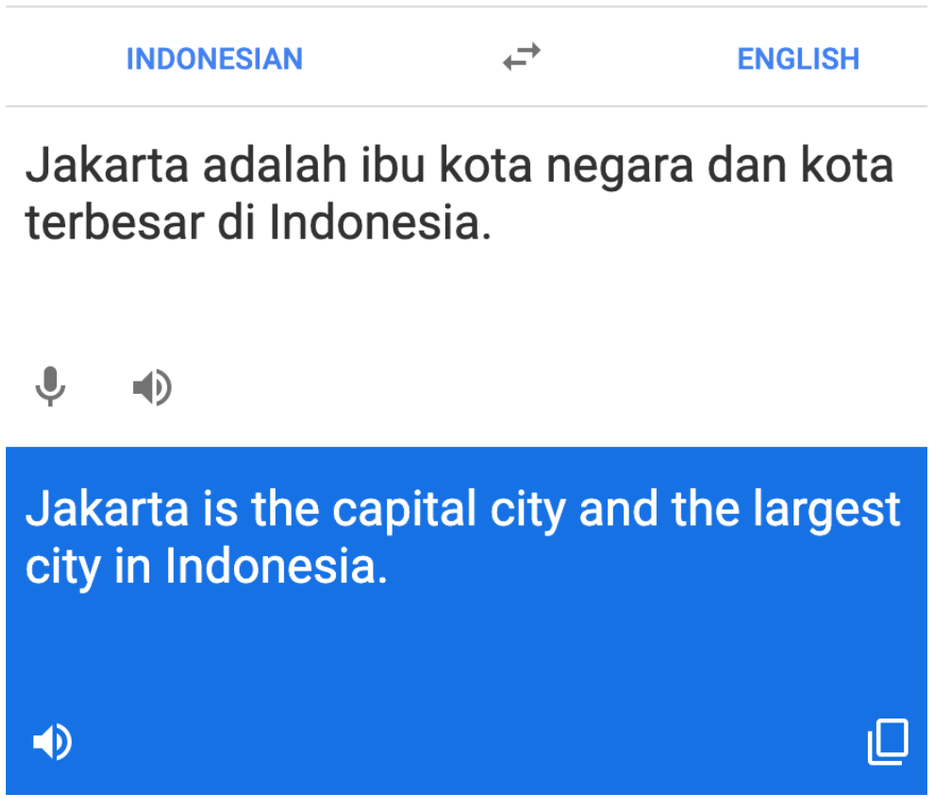


Google Translate

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

OpenAI GPT-2 Language Model

What can we do with neural networks?



Google Translate

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

OpenAI GPT-2 Language Model

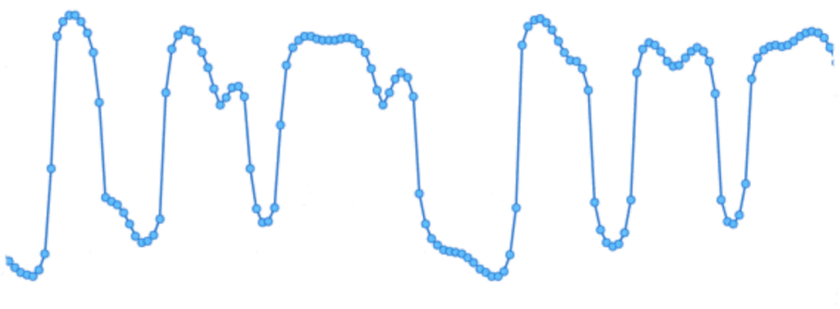
What can we do with neural networks?

Image generation:

What can we do with neural networks?

Image generation:

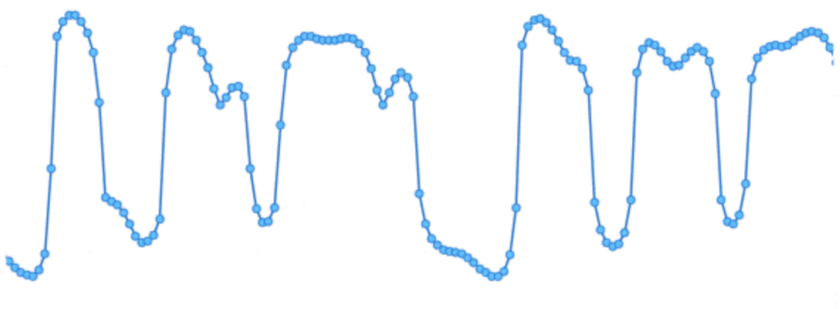
Sound generation:



What can we do with neural networks?

Image generation:

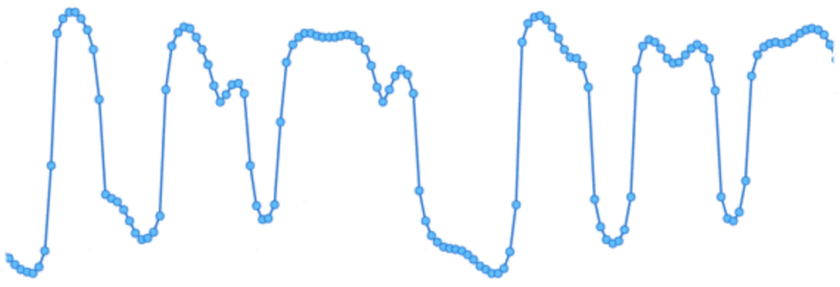
Sound generation:

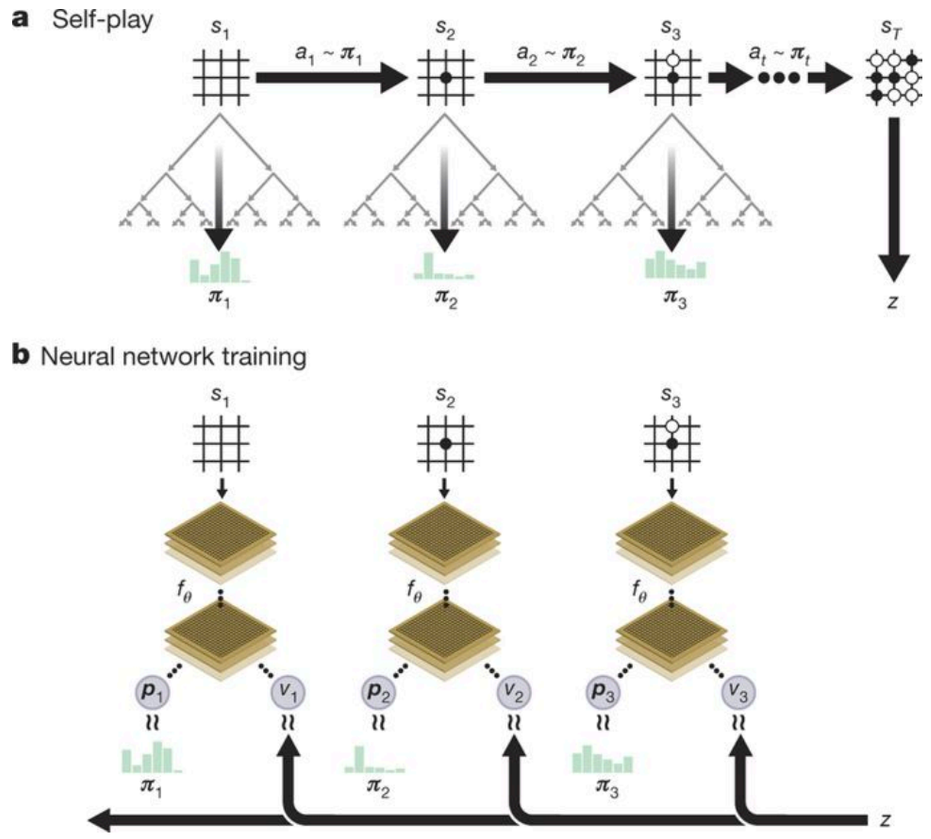


What can we do with neural networks?

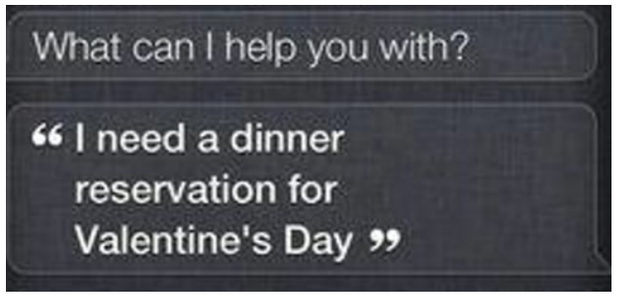
Image generation:

Sound generation:





What can we do with neural networks?



What can we do with neural networks?

What can we do with neural networks?

So we can’t do everything yet, but we can do a lot of amazing stuff, and more stuff is coming, which you’ll be a part of.

Goals of this lecture:

- How do the neural networks work?
- Why are they so powerful compared to other techniques?
- What are deep learning frameworks doing behind the scenes?
- When you’re working with neural networks, what can you do to improve performance?

Learning and Induction

[Induction is knowing that] instances of which we have had no experience resemble those of which we have had experience.

D. Hume (1739). Treatise of Human Nature.

Only if the [learner] has other sources of information, or biases for choosing one generalization over the other, can it non-arbitrarily classify instances beyond those in the training set.

T. Mitchell (1980). The need for biases in learning generalizations.

Outline

- Why neural networks?
- Error-driven learning
- Bias and variance
- Feed-forward neural networks
- Obtaining gradients: by hard work and by automatic differentiation
- Example: solving XOR

Why Neural Networks?

- **Error-driven learning**: pick the error function you want, neural networks learn by minimising errors
    - Often the real world tells us what the error functions are. E.g., is a false positive or a false negative more costly?
- **Trade between bias and variance**
    - Lots of data -> use bigger and more complicated (higher variance) models
    - Regularization, dropout, data augmentation
- **Modularity**
    - Everything is represented as a vector (or matrix or tensor), so different data types and sources can be made to “interact”

Error Driven Learning

- Feed-forward networks are an instance of **supervised learning**: take an input and make some prediction

- Learning works by initializing the network, running the training data through it, and adjusting the behavior of the network so that it makes “less serious errors” relative to the true answers present in the training data, and repeating

- Our learner consists of two distinct functions

    - A differentiable function that takes parameters and inputs and produces either an **output prediction** or a **distribution over outputs predictions**

    - A differentiable function that takes the prediction/distribution over predictions and **computes a loss**.

Examples: Continuous

Inputs		Target Output	
Temperature in C	36.2	necessary fuel in L	252
Wind speed in km/h	-14		
Distance in km	250		
Cargo Weight in kg	800		
Humidity	0.40		

Prediction function

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Loss function

$$\mathcal{L} = \frac{1}{2}(\mathbf{y}^* - \hat{\mathbf{y}})^2$$

Examples: Continuous

Inputs	
Temperature in C	36.2
Wind speed in km/h	-14
Distance in km	250
Cargo Weight in kg	800
Humidity	0.40

Target Output	
necessary fuel in L	252

Prediction function

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Loss function

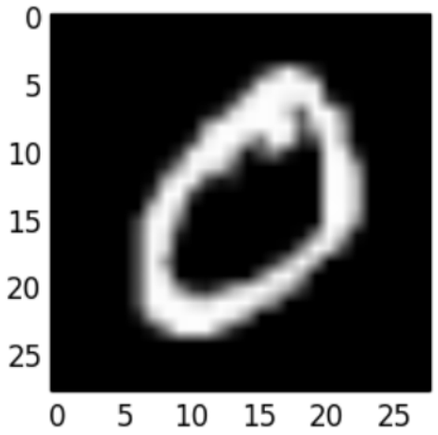
$$\mathcal{L} = \frac{1}{2}(\mathbf{y}^* - \hat{\mathbf{y}})^2$$

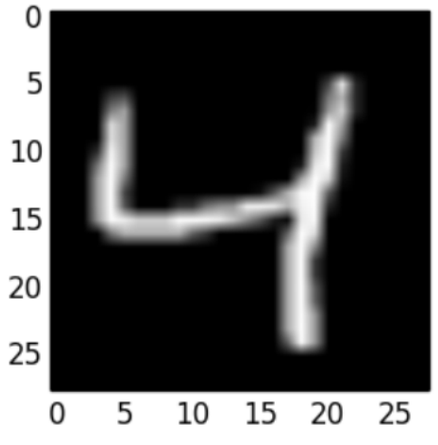
$$\mathcal{L} = \max\{0, |\mathbf{y}^* - \hat{\mathbf{y}}| - \epsilon\}$$

Examples: Discrete

Inputs

28x28 grid of grey-scale pixels





Target Output

$$y^* = 0$$

Prediction function

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\text{softmax}(\mathbf{u}) = \frac{\exp \mathbf{u}}{\sum_i (\exp \mathbf{u})_i}$$

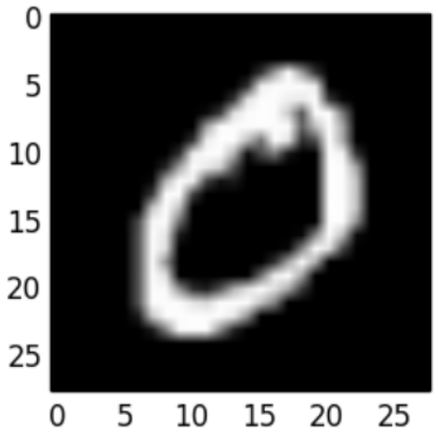
Loss function

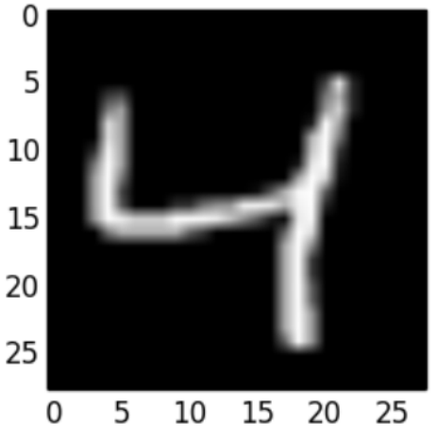
$$\mathcal{L} = -\log(\hat{\mathbf{y}})_{y^*}$$

Examples: Discrete

Inputs

28x28 grid of grey-scale pixels





Target Output

$$y^* = 0$$

Prediction function

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

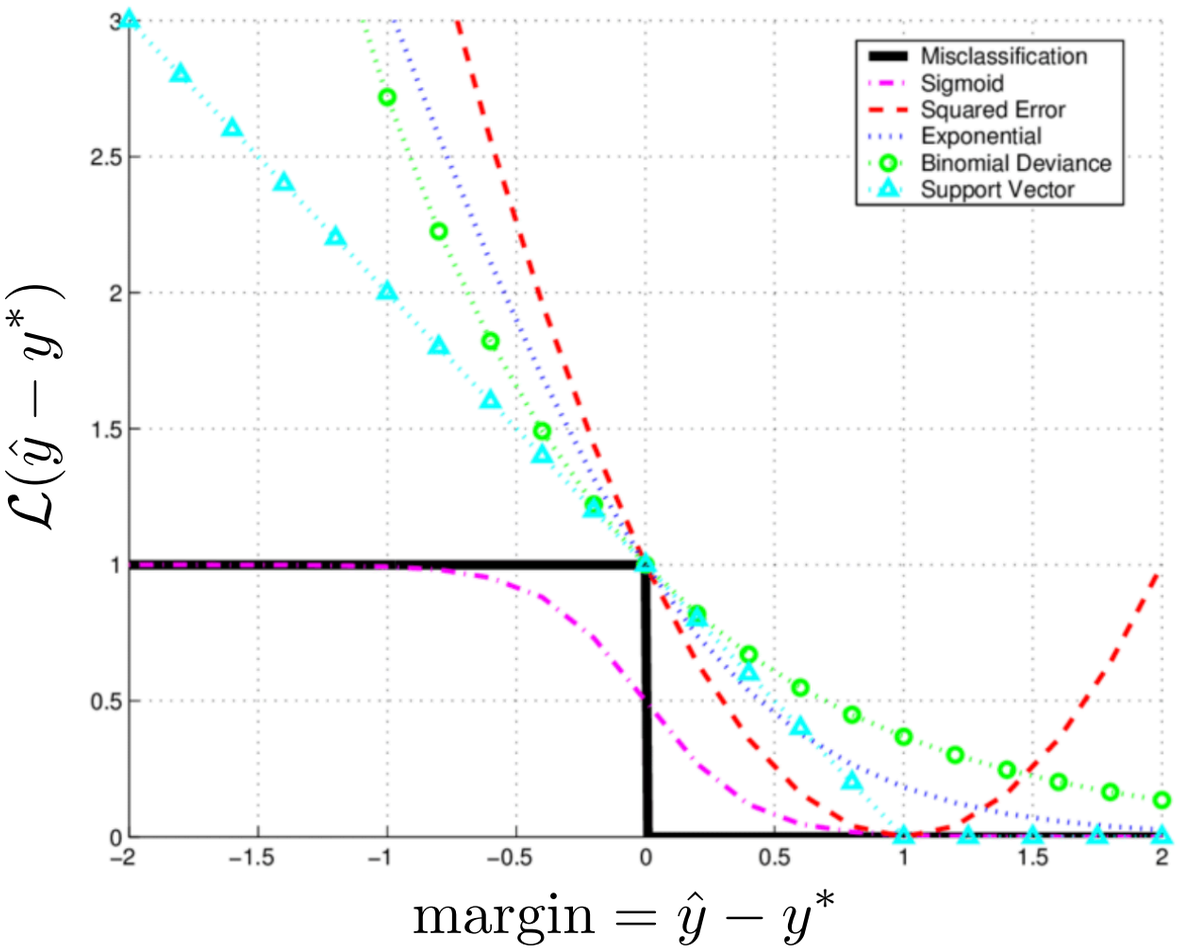
Loss function

$$\mathcal{L} = -\log(\hat{\mathbf{y}})_{y^*}$$

$$\mathcal{L} = \max\{0, 1 - \hat{y}_{y^*} + \max_{i \neq y^*} \hat{y}_i\}$$

Differentiable Losses

Binary classification losses



Multiclass classification losses

- **cross entropy** ("softmax regression")
- perceptron loss
- max-margin
- When you know the real costs
  - softmax margin
  - expected cost

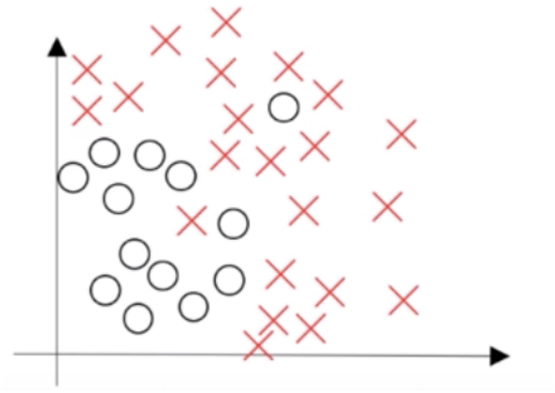
Continuous prediction

- **squared error**
- If outliers are a problem:
  - absolute error
  - Huber's loss

Other things to be aware of: Poisson regression, Dirichlet regression, ordinal regression. Pick a loss that suits your problem - it can have a big impact!

Bias and Variance

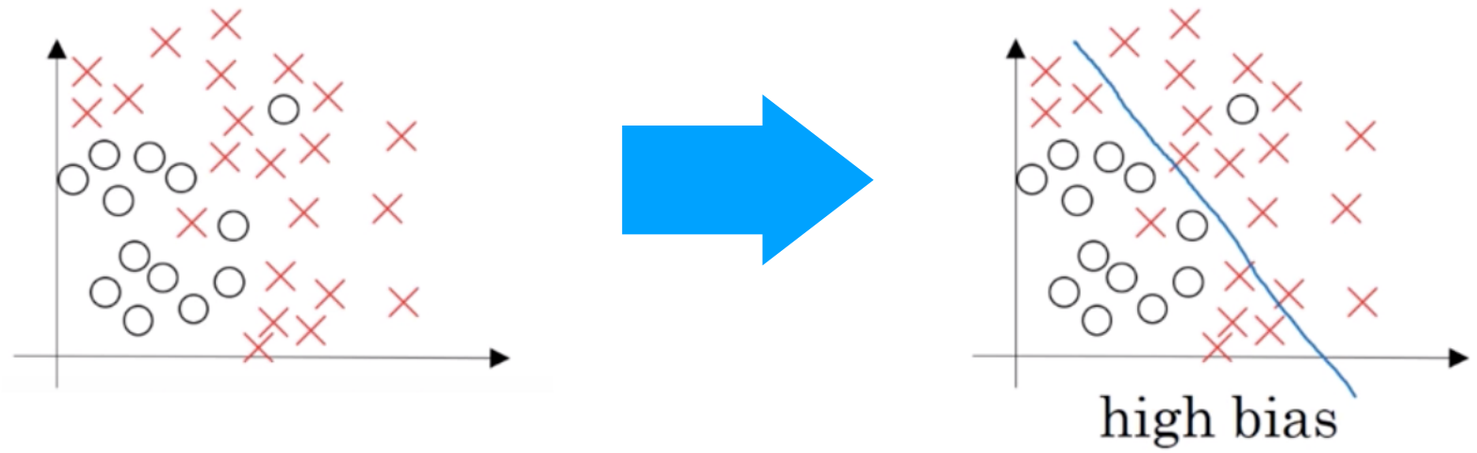
Linear Models



Example by Andrew Ng

Bias and Variance

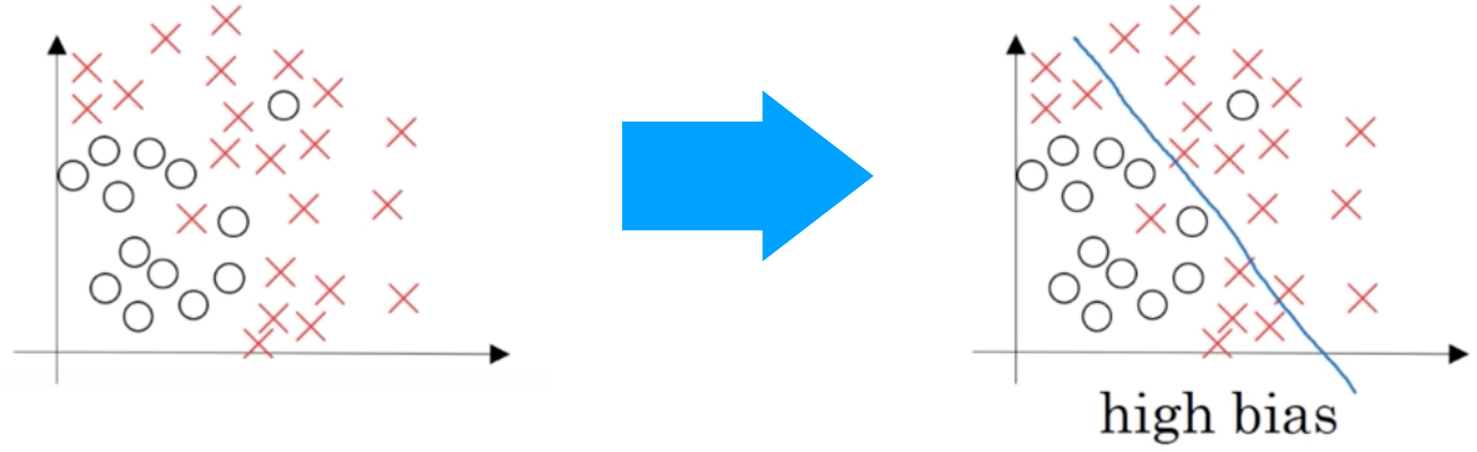
Linear Models



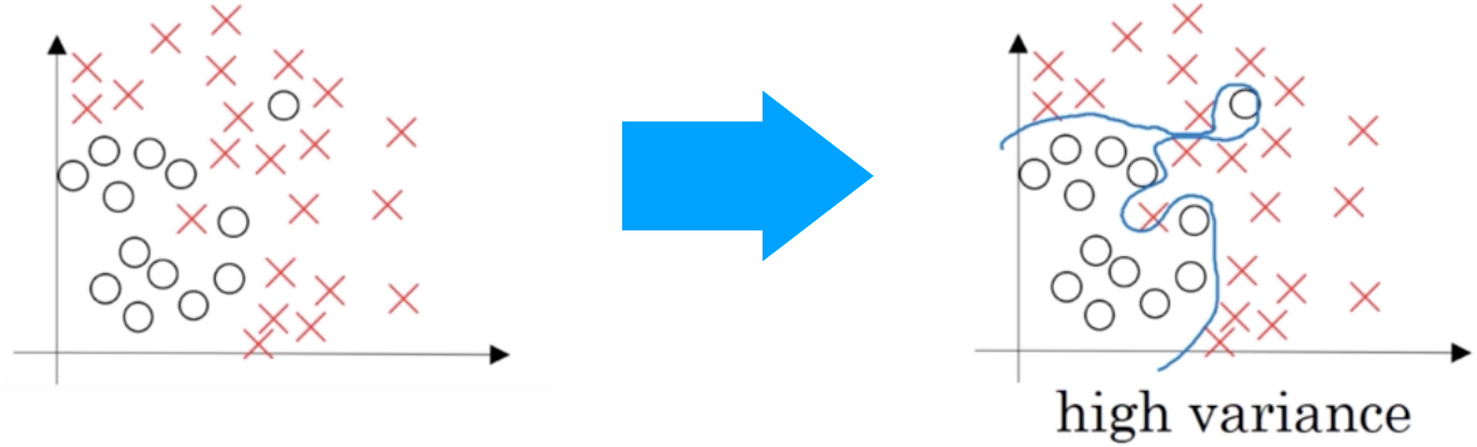
Example by Andrew Ng

Bias and Variance

Linear Models



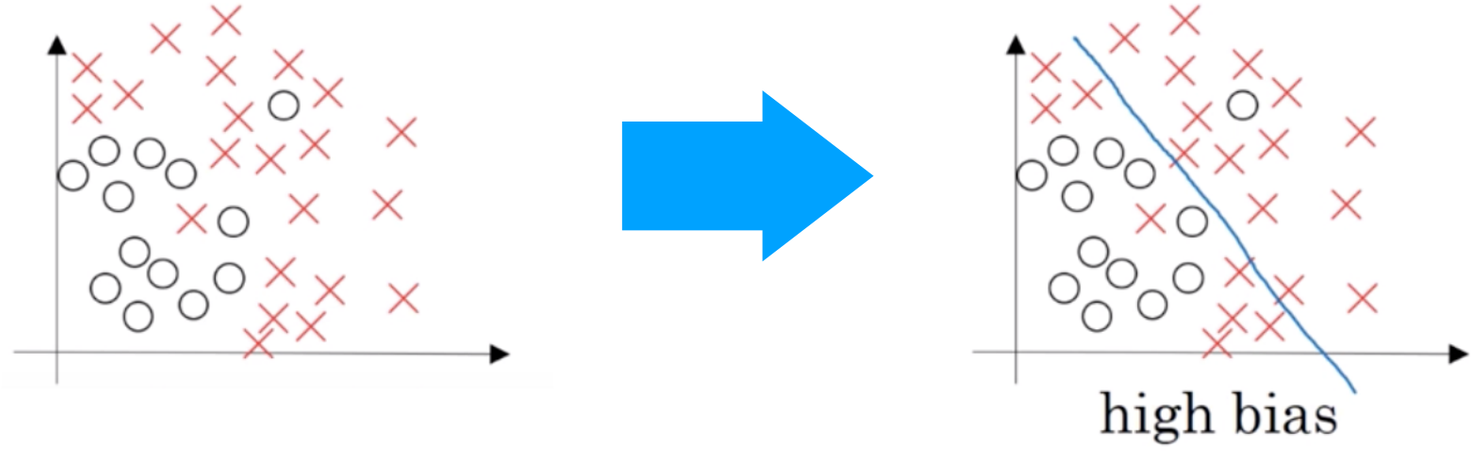
kNN Classifiers



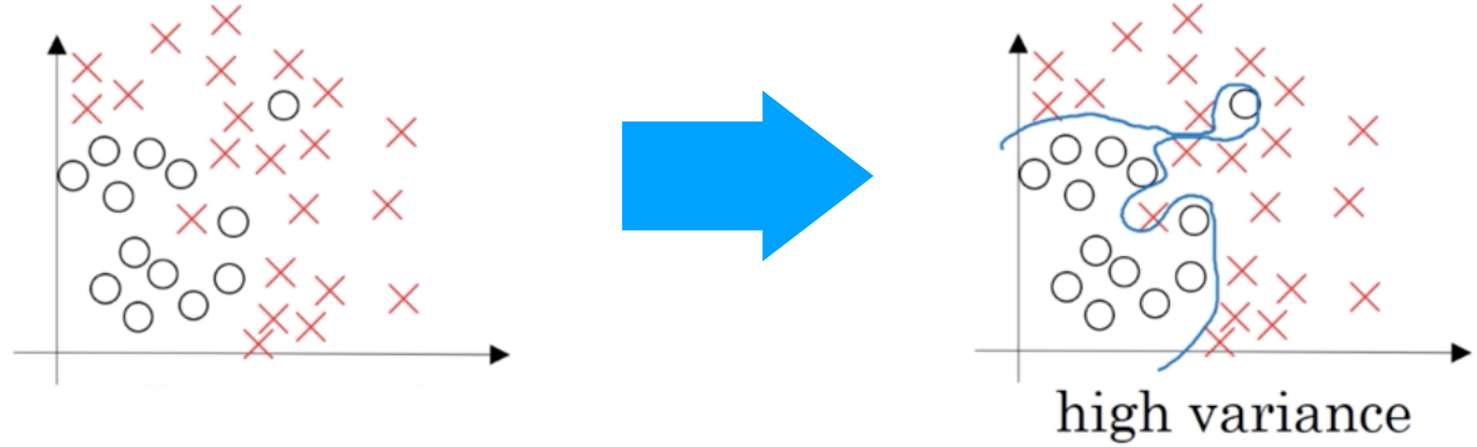
Example by Andrew Ng

Bias and Variance

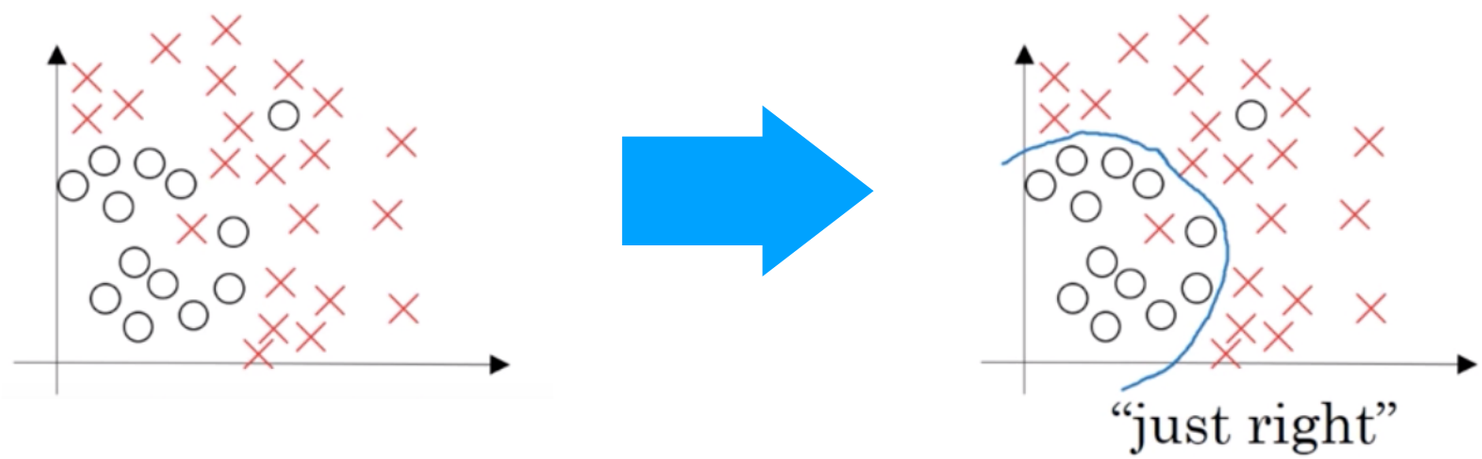
Linear Models



kNN Classifiers



Neural Networks (and a few others)

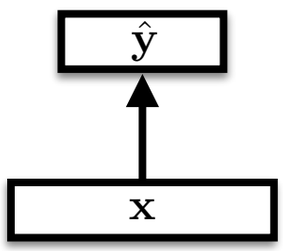


Example by Andrew Ng

Feed-forward Networks

Linear Model

$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$



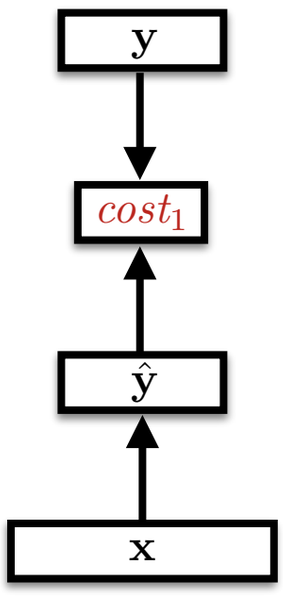
Feed-forward Networks

Linear Model

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Squared error:

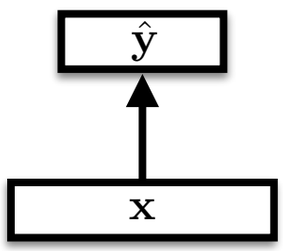
$$\mathcal{L} = \frac{1}{2}(\mathbf{y}^* - \hat{\mathbf{y}})^2$$



Feed-forward Networks

Linear Model

$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$



Feed-forward Networks

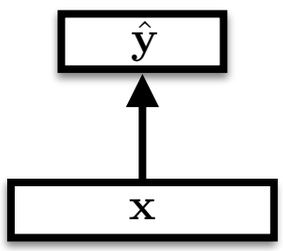
Linear Model

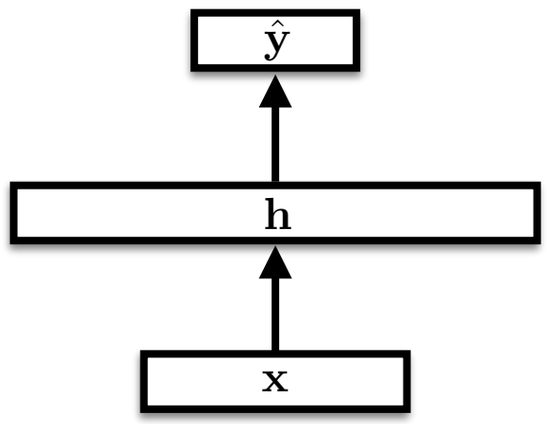
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$





Feed-forward Networks

Linear Model

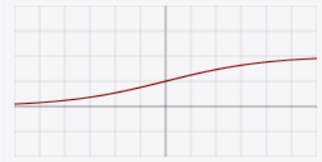
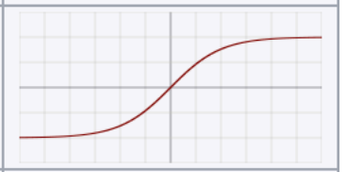
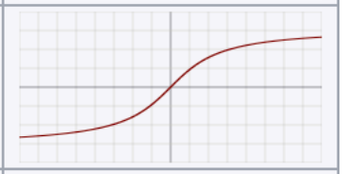
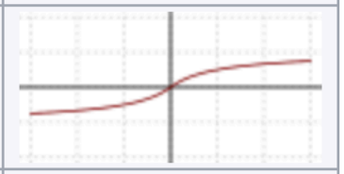
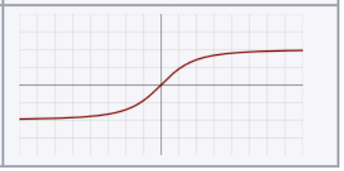
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

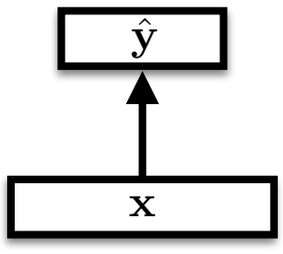
MLP

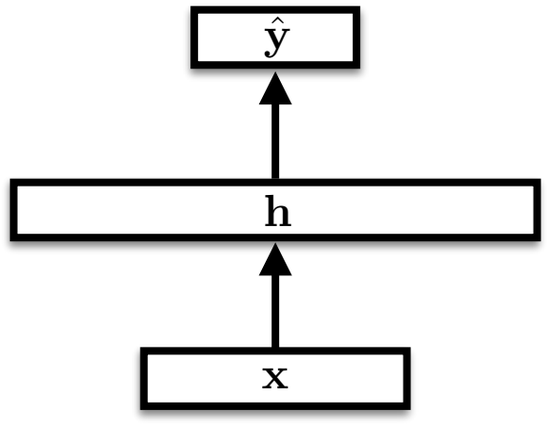
$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Nonlinearities

Logistic (a.k.a. Sigmoid or Soft step)	
TanH	
ArcTan	
ArSinH	
ElliotSig[8][9][10] Softsign[11][12]	
Inverse square root unit (ISRU)[13]	





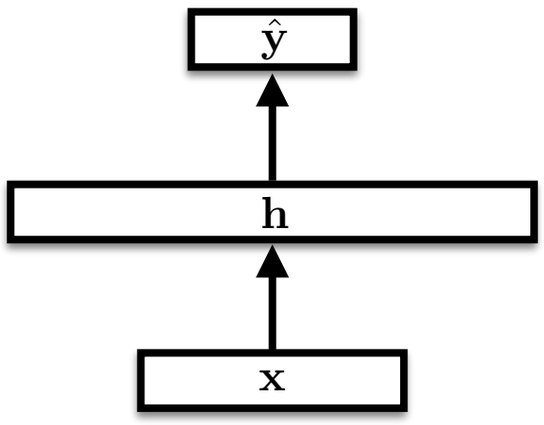
And many more…

Feed-forward Networks

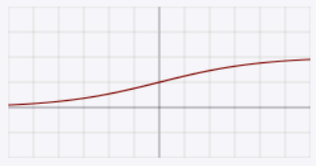
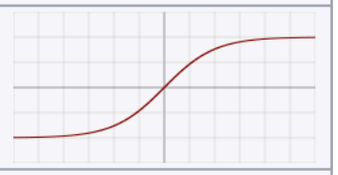
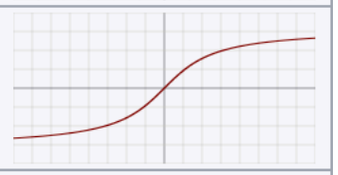
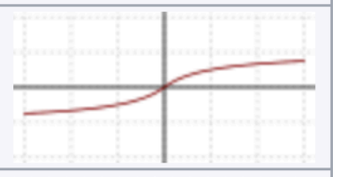
MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

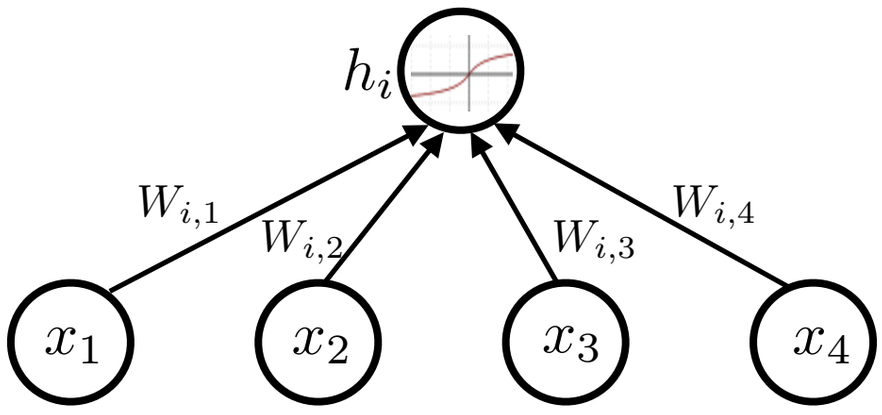


Nonlinearities

Logistic (a.k.a. Sigmoid or Soft step)	
TanH	
ArcTan	
ArSinH	
ElliotSig[8][9][10] Softsign[11][12]	
Inverse square root unit (ISRU)[13]	

And many more…

This formulation is where the term “neural network” comes from. Each element of the vector simulates the behavior of an idealized biological neuron, which integrates inputs and then fires as a result.

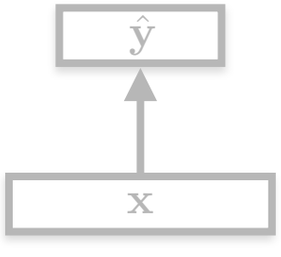


$$h_i = g\left(\sum_j x_j \times W_{i,j}\right)$$

Feed-forward Networks

Linear Model

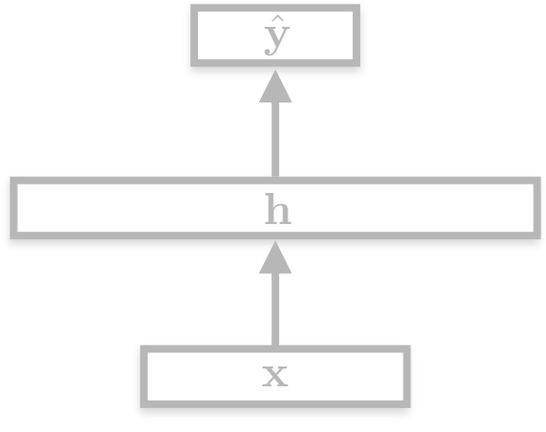
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

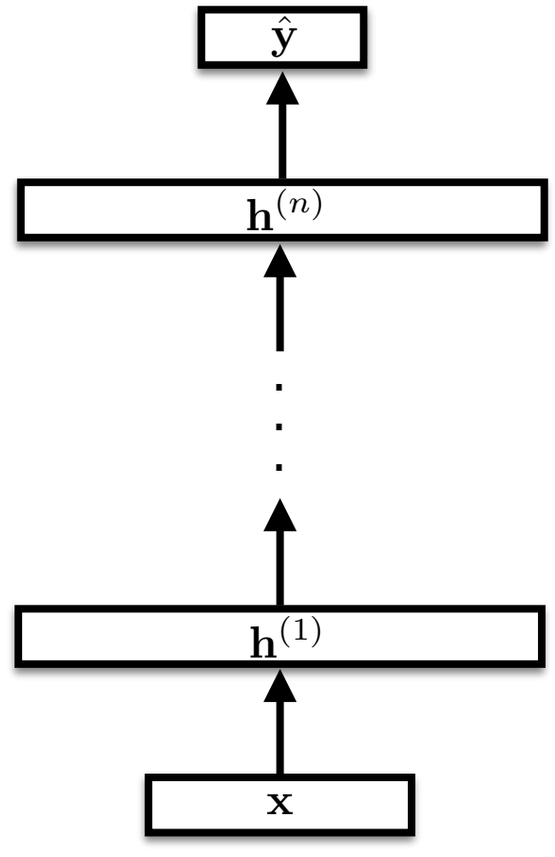


n-Layer MLP

$$\mathbf{h}^{(0)} = \mathbf{x}$$

$$\mathbf{h}^{(i)} = g(\mathbf{V}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{c}^{(i)}$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}^{n} + \mathbf{b}$$



Feed-forward Networks

Linear Model

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

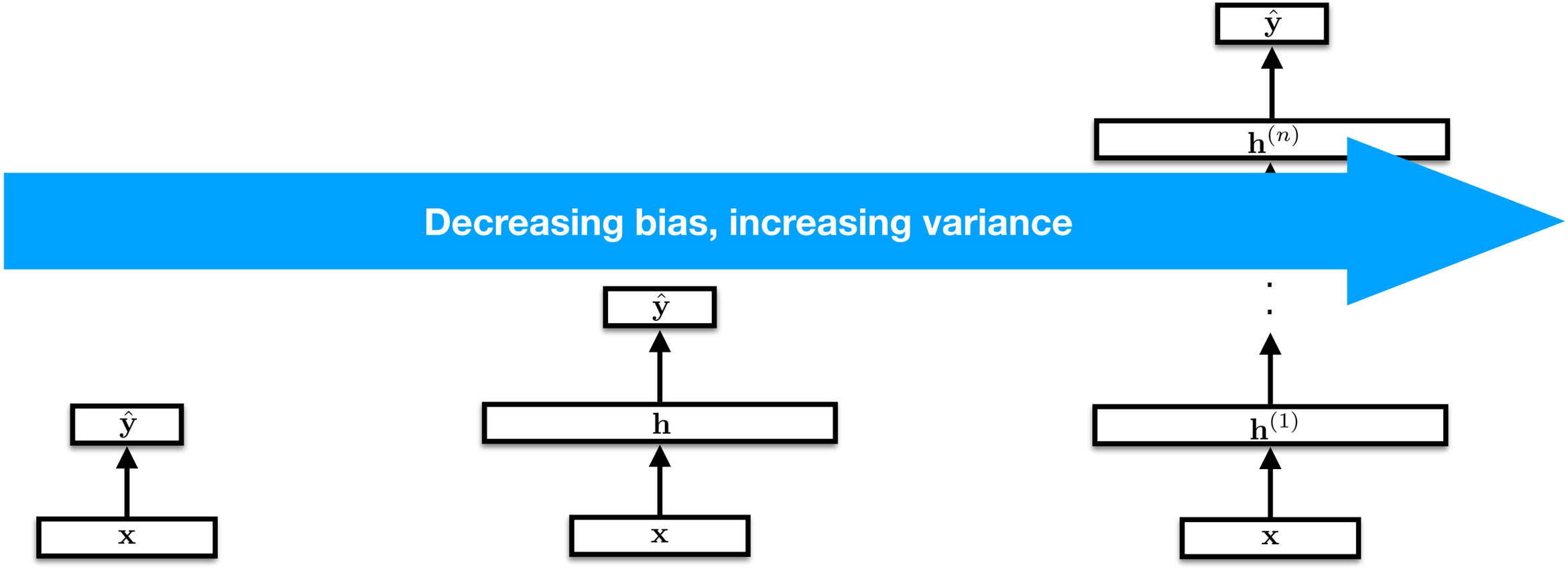
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

n-Layer MLP

$$\mathbf{h}^{(0)} = \mathbf{x}$$

$$\mathbf{h}^{(i)} = g(\mathbf{V}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{c}^{(i)}$$

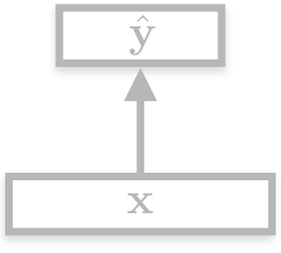
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}^{n} + \mathbf{b}$$



Feed-forward Networks

Linear Model

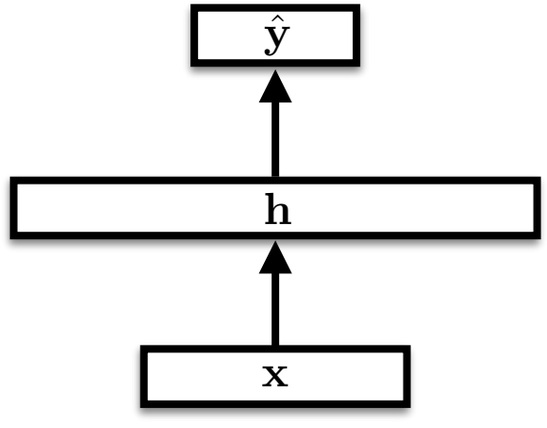
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

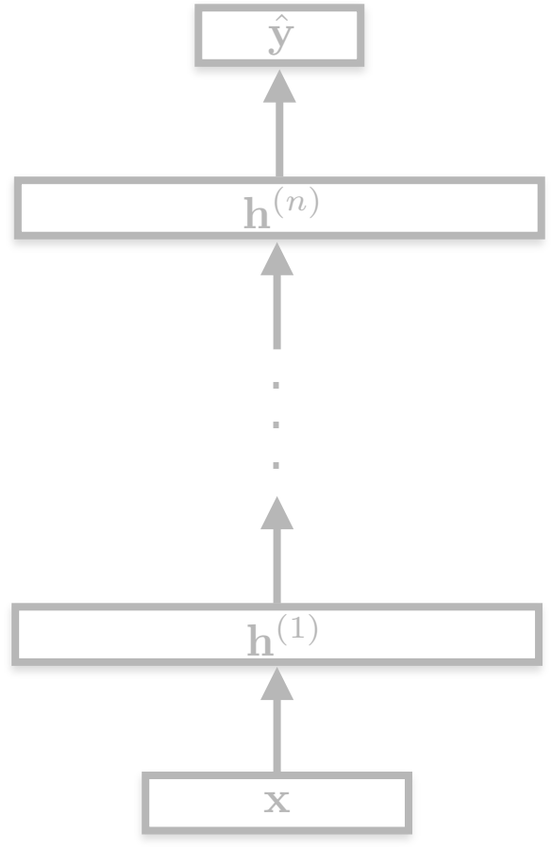


n-Layer MLP

$$\mathbf{h}^{(0)} = \mathbf{x}$$

$$\mathbf{h}^{(i)} = g(\mathbf{V}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{c}^{(i)}$$

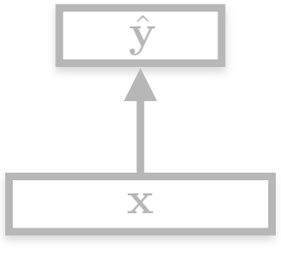
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}^{n} + \mathbf{b}$$



Feed-forward Networks

Linear Model

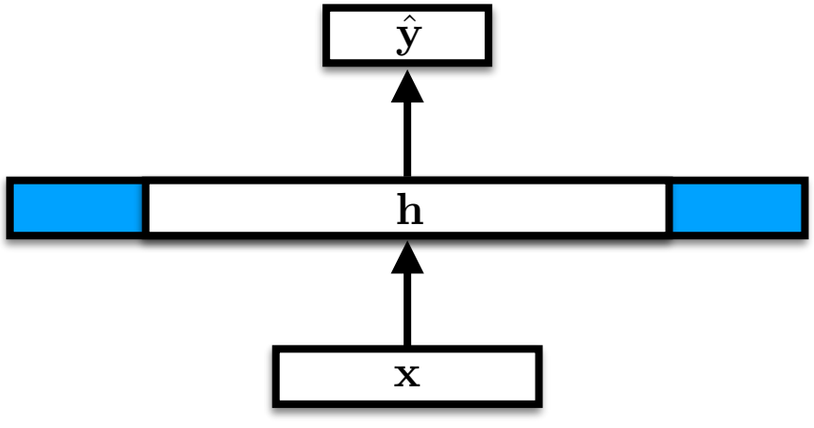
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

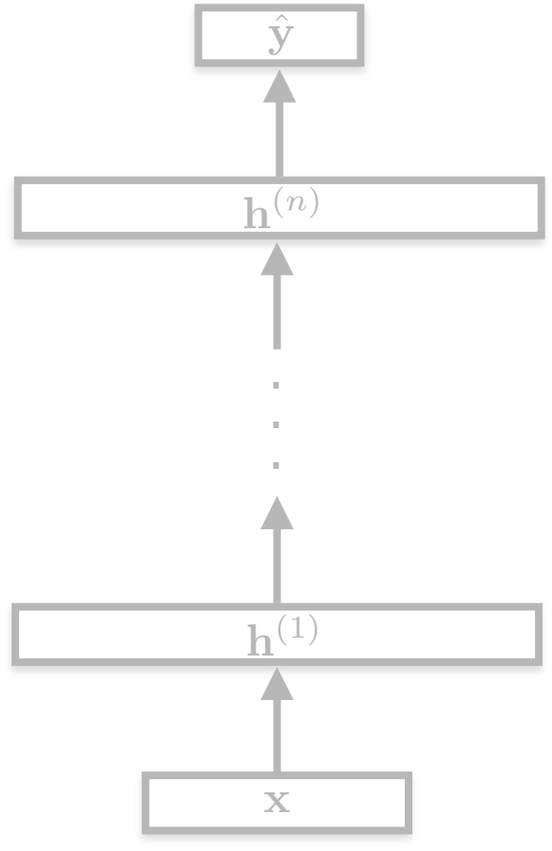


n-Layer MLP

$$\mathbf{h}^{(0)} = \mathbf{x}$$

$$\mathbf{h}^{(i)} = g(\mathbf{V}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{c}^{(i)}$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}^{n} + \mathbf{b}$$



MLPs: What's so great?

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathcal{F} = \frac{1}{M}\sum_{i=1}^{M} ||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2$$

In linear regression, the goal is to learn **W** and **b** such that *F* is minimized for a dataset *D* consisting of *M* training instances. An engineer must select/design **x** *carefully*.

MLPs: What's so great?

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathcal{F} = \frac{1}{M}\sum_{i=1}^{M} ||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2$$

In linear regression, the goal is to learn **W** and **b** such that *F* is minimized for a dataset *D* consisting of *M* training instances. An engineer must select/design **x** *carefully*.

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

"nonlinear regression"

Use "naive features" **x** and *learn* their transformations (conjunctions, nonlinear transformation, etc.) into **h**.

Feature Induction

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

- What functions can this parametric form compute?
  - If **h** is big enough (i.e., enough dimensions), it can represent any vector-valued function to any degree of precision

- This is a much more powerful regression model!

Feature Induction

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

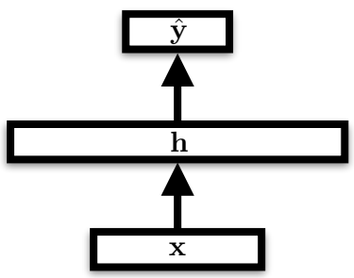
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

- What functions can this parametric form compute?
  - If **h** is big enough (i.e., enough dimensions), it can represent any vector-valued function to any degree of precision
- This is a much more powerful regression model!
- You can think of **h** as “induced features” in a linear classifier
  - The network did the job of a feature engineer

Deep Neural Networks

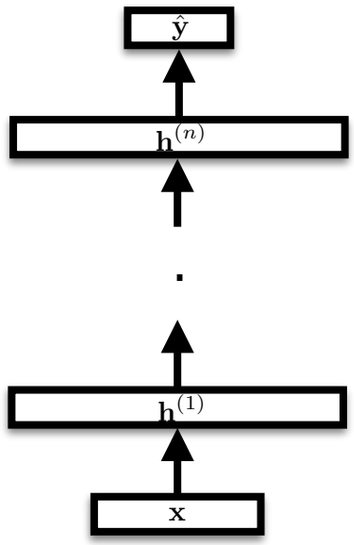
MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$



n-Layer MLP

$$\mathbf{h}^{(0)} = \mathbf{x}$$
$$\mathbf{h}^{(i)} = g(\mathbf{V}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{c}^{(i)})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}^{n} + \mathbf{b}$$



As we saw earlier, we can increase capacity by either making networks wider or deeper. What is better?

Deeper models can learn more expressive functions with fewer parameters; however, the learning problem can become quite difficult and the computational costs increase. (Although we will see some strategies for making it easier to learn deeper networks.)

Intuitively, a hidden layer with nonlinear activations can learn conjunctions of features from the lower layer. A one-layer network has to learn complex conjunctions of basic features, whereas a deeper network can learn conjunctions of conjunctions which can represent a combinatorial space of pattern detectors.

Learning: Optimization

- We have written down the **loss** as a function of the **data** (constants) as a function of some **parameters** (variables).

- **Differentiate this function with respect to your parameters**

- **Write code to compute the first derivatives of the loss wrt the parameters**

- Execute an optimisation algorithm: **SGD**, LBFGS, CG, AdaGrad, AdaDelta, Adam, …

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$$

Obtaining Derivatives

- **An MLP is simple, but powerful enough to solve any problem in theory** (given enough data and a good learner)

- Challenge: how do we get derivatives of the loss of the MLP with respect to parameters?

  - Calculus!

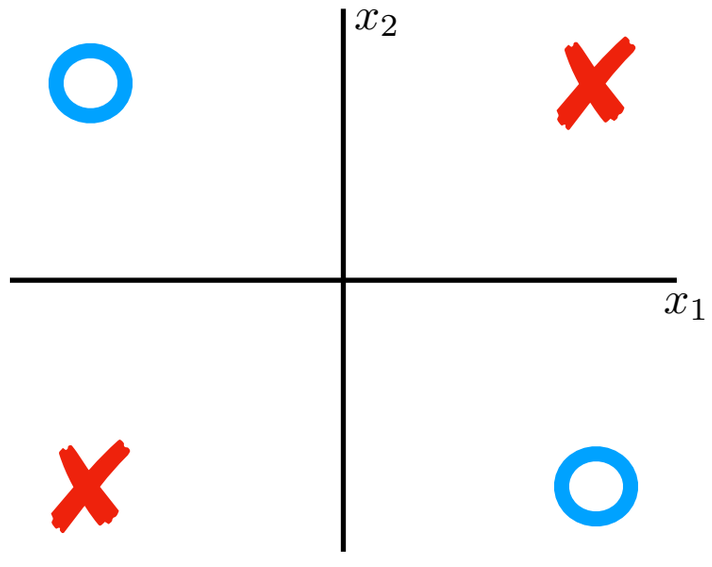
  - In particular, we can use the chain rule for derivatives.

$$\frac{dw}{dz} = \frac{dw}{dx} \cdot \frac{dx}{dy} \cdot \frac{dy}{dz}$$

The XOR Problem

- The XOR function plays an important role in the history of machine learning

- The first ML/AI Winter came about when it was shown that the original perceptron learn just learned linear functions, and XOR - as simple as it is - is nonlinear



- We will use an MLP to learn to solve this problem in just a moment.

Implementation Details: Minibatching

- GPU hardware is
  - pretty fast for elementwise operations (IO bound- can't get enough data through the GPU)
  - very fast for matrix-matrix multiplication (usually compute bound - the GPU will work at 100% capacity, and GPU cores are **fast**)
- RNNs, LSTMs, GRUs all consist of
  - lots of elementwise operations (addition, multiplication, nonlinearities, …)
  - lots of matrix-vector products
- Minibatching: convert many matrix-vector products into a single matrix-matrix multiplication (or matrix-matrix products into batched matrix-matrix products, and so on)

Minibatching

Single-instance MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Minibatching

Single-instance MLP

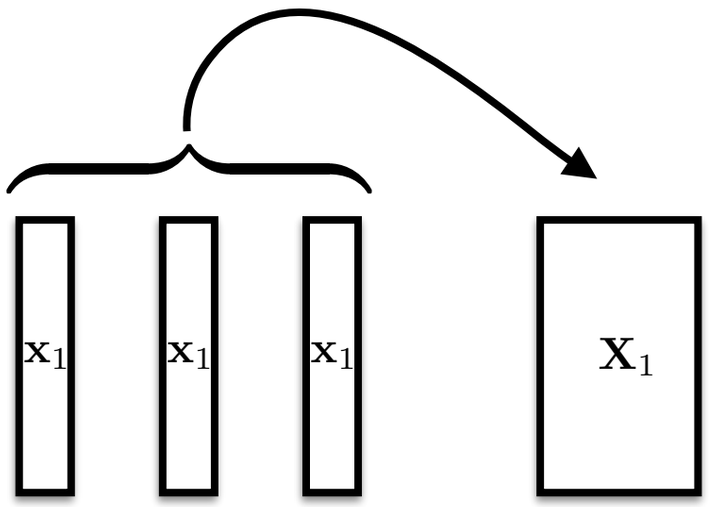
$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Minibatch MLP

$$\mathbf{H} = g(\mathbf{V}\mathbf{X} + \mathbf{c})$$

$$\hat{\mathbf{Y}} = \mathbf{W}\mathbf{H} + \mathbf{b}$$



Minibatching

Single-instance MLP

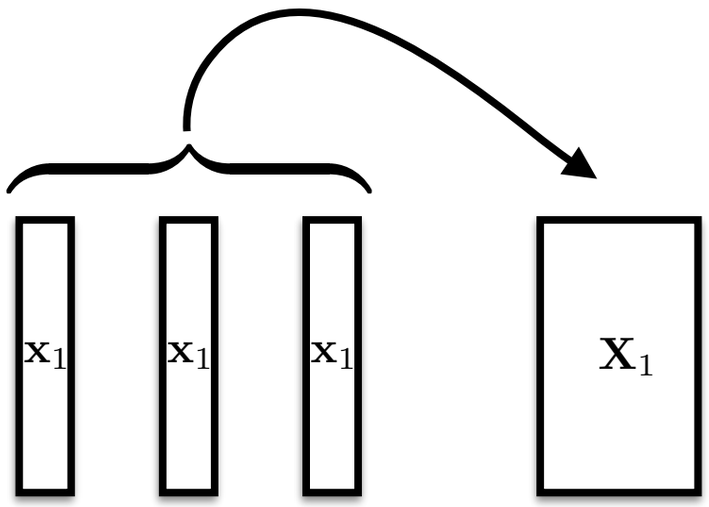
$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Minibatch MLP

$$\mathbf{H} = g(\mathbf{V}\mathbf{X} + \mathbf{c})$$

$$\hat{\mathbf{Y}} = \mathbf{W}\mathbf{H} + \mathbf{b}$$



Minibatching

Single-instance MLP

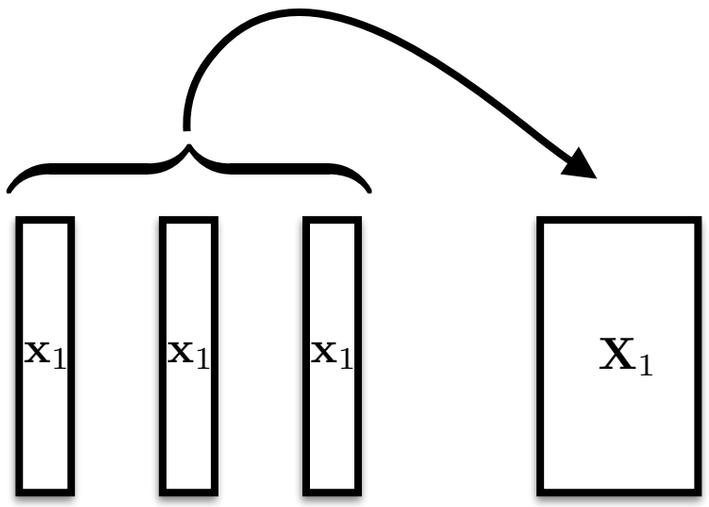
$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Minibatch MLP

$$\mathbf{H} = g(\mathbf{V}\mathbf{X} + \mathbf{c})$$

$$\hat{\mathbf{Y}} = \mathbf{W}\mathbf{H} + \mathbf{b}$$



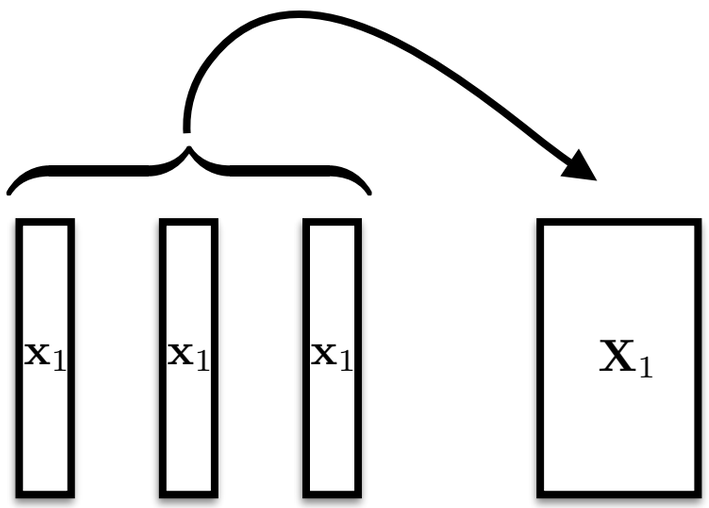
Minibatching

Single-instance MLP

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Minibatch MLP



$$\mathbf{H} = g(\mathbf{V}\mathbf{X} + \mathbf{c})$$

$$\hat{\mathbf{Y}} = \mathbf{W}\mathbf{H} + \mathbf{b}$$

anything wrong here?

Minibatching

- Manual minibatching is hard, especially when inputs are different sizes and shapes (as real data tends to be).

- New language support (Swift for TensorFlow, Julia, DyNet for C++) is making some headway, but **this is an opportunity for programming language innovation.**

- For this lecture, I'll assume inputs and outputs are always the same size and shape.

Let’s switch to code

The Challenge

- In the old days: we spent a lot of time differentiating and writing code to compute derivatives
    - It is very easy to get this wrong
        - (Yes, they look sort of fancy in papers; yes, they can provide intuitions you might miss; yes, they sometimes simplify in ways that save computation)
- Insight of the last few years: let's stop doing this and let computers differentiate for us
    - We have better things to do with our time- experimentation, new models, going to Jakarta

Alternatives

- **Numeric differentiation** (ND)

  - Implement the loss function, for every parameter, change it a tiny bit and recompute to estimate the derivative.
    Problems: runs in time O(*n*) where *n* is the # of params, rounding errors

- **Symbolic differentiation** (SD)

- **Automatic differentiation** (AD)

Automatic Differentiation

- Write code to compute the loss function as you normally would
    - For neural nets, this will be forward propagation
    - For structured models, this will be things like the “inside algorithm” or “forward algorithm”
- Augment the code with AD data types that store extra information that is then used to compute derivatives using the rules of calculus
- This works for any function

Any function?

- Any complex function is the composition of simpler functions

- Compilers/interpreters turn complex functions into a sequence of simpler functions (ultimately individual CPU operations!)

- The **chain rule for derivatives** says how you differentiate compositions of simple functions

- AD automatically tracks the extra information you need to compute derivatives

Any function?

- Most libraries handle all the standard mathematical operations
  - max, min, abs, compute sub derivatives
  - Linear algebraic operations (SVD, matrix inverse, etc) available, usually in optimised form
  - Control flow is easy, provided the control flow doesn't depend on a value that you're differentiating
- If your function isn't differentiable, AD won't make it so - e.g., you can't get derivatives with respect to integers!

Let's start with a really simple example.

$y = \log \sin^2 x$ What is the derivative at x_0 ?

components	***range***	***differential***	***d-range***
$y = f(u) = \log u$	$\mathbb{R}$	$\dfrac{dy}{du} = \dfrac{1}{u}$	$\mathbb{R}$
$u = g(v) = v^2$	$\mathbb{R}$	$\dfrac{du}{dv} = 2v$	$\mathbb{R}$
$v = h(x) = \sin x$	$\mathbb{R}$	$\dfrac{dv}{dx} = \cos x$	$\mathbb{R}$

$$\left.\frac{dy}{dx}\right|_{x=x_0} = \left.\frac{dy}{du}\right|_{u=g(h(x_0))} \cdot \left.\frac{du}{dv}\right|_{v=h(x_0)} \cdot \left.\frac{dv}{dx}\right|_{x=x_0}$$

In general, for our applications $\mathbf{x}$ in $f(\mathbf{x})$ will be a *vector*.

$$y = \sum_{i=1}^{n} (\mathbf{W} \exp \mathbf{x})_i \quad \text{where} \quad \mathbf{x} \in \mathbb{R}^d \quad \text{and} \quad \mathbf{W} \in \mathbb{R}^{n \times d}$$

components	***range***	***differential***	***d-range***
$y = f(\mathbf{u}) = \sum_{i=1}^{n} u_i$	$\mathbb{R}$	$\frac{\partial y}{\partial \mathbf{u}} = \mathbf{1}$	$\mathbb{R}^{1 \times n}$
$\mathbf{u} = g(\mathbf{v}) = \mathbf{W}\mathbf{v}$	$\mathbb{R}^n$	$\frac{\partial \mathbf{u}}{\partial \mathbf{v}} = \mathbf{W}$	$\mathbb{R}^{n \times d}$
$\mathbf{v} = h(\mathbf{x}) = \exp \mathbf{x}$	$\mathbb{R}^d$	$\frac{\partial \mathbf{v}}{\partial \mathbf{x}} = \operatorname{diag}(\exp \mathbf{x})$	$\mathbb{R}^{d \times d}$

$$\left.\frac{dy}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0} = \underbrace{\left.\frac{dy}{d\mathbf{u}}\right|_{\mathbf{u}=g(h(\mathbf{x}_0))}}_{\mathbb{R}^{1 \times n},\ \in \mathbb{R}^n} \cdot \underbrace{\left.\frac{d\mathbf{u}}{d\mathbf{v}}\right|_{\mathbf{v}=h(\mathbf{x}_0)}}_{\mathbb{R}^{n \times d},\ \in \mathbb{R}^d} \cdot \underbrace{\left.\frac{d\mathbf{v}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0}}_{\mathbb{R}^{d \times d},\ \in \mathbb{R}^d}$$

Two Evaluation Strategies

"Forward"

$$\left.\frac{dy}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0} = \left.\frac{dy}{d\mathbf{u}}\right|_{\mathbf{u}=g(h(\mathbf{x}_0))} \cdot \left.\frac{d\mathbf{u}}{d\mathbf{v}}\right|_{\mathbf{v}=h(\mathbf{x}_0)} \cdot \left.\frac{d\mathbf{v}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0}$$

$\mathbb{R}^{1\times n}$ $\mathbb{R}^{n\times d}$ $\mathbb{R}^{d\times d}$

$\in \mathbb{R}^n$ $\in \mathbb{R}^d$ $\in \mathbb{R}^d$

"Backward" or "Adjoint"

Two Evaluation Strategies

"Forward"

$$\left.\frac{dy}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0} = \underbrace{\left.\frac{dy}{d\mathbf{u}}\right|_{\mathbf{u}=g(h(\mathbf{x}_0))}}_{\in \mathbb{R}^n} \cdot \underbrace{\left.\frac{d\mathbf{u}}{d\mathbf{v}}\right|_{\mathbf{v}=h(\mathbf{x}_0)}}_{\in \mathbb{R}^d} \cdot \underbrace{\left.\frac{d\mathbf{v}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0}}_{\in \mathbb{R}^d}$$

$\mathbb{R}^{1\times n}$ $\mathbb{R}^{n\times d}$ $\mathbb{R}^{d\times d}$

"Backward" or "Adjoint"

Which is better?

Which is better?

- When you have vector inputs, **backward (adjoint) mode is usually more efficient** (often by a lot)
    - Rather than **intermediate matrices**, you just have **vectors**, which can save a lot of computation and usually memory
        - But you do have to keep around f(x)'s computed during function evaluation
- Depending on the problem (number of inputs, outputs, layers), one or the other may be better
- Forward mode is easier to implement from scratch, but **many tools exist that implement backward mode**

Implementation

- Operator/function overloading
    - Rather than use **`np.array`** use **`torch.Tensor`** or **`tf.Tensor`** (or in C++, rather than **`double`** use **`adouble`**)
    - It secretly keeps track of the operation stack (for backward mode) or gradients of the intermediate values with respect to inputs for forward mode
- Source code transformation (similar to symbolic differentiation)
    - The compiler transforms program (or representation of program, in TensorFlow “graph mode”) into code that computes derivatives. Still relies on chain rule; still uses either forward or reverse evaluation (or a mix)

expression:

$$\mathbf{x}$$

expression:

$$\mathbf{x}$$

graph:

expression:

$$\mathbf{x}$$

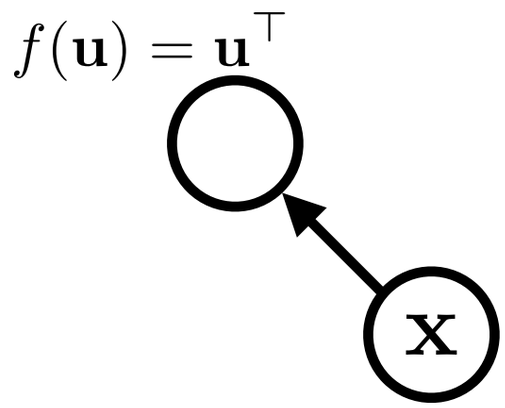
graph:

A **node** is a {tensor, matrix, vector, scalar} value

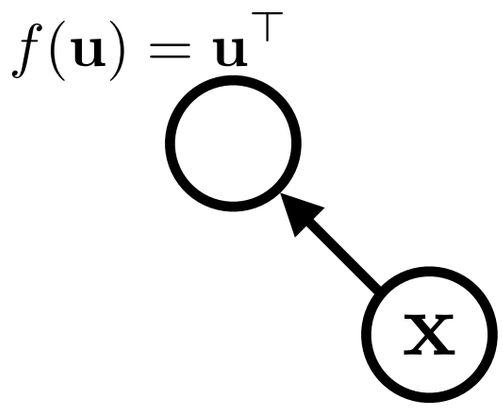
expression:

$$\mathbf{x}^\top$$

graph:

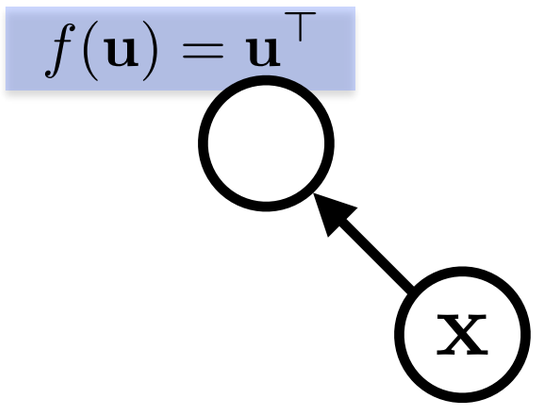


An **edge** represents a function argument (and also an data dependency). They are just pointers to nodes.



An **edge** represents a function argument (and also an data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

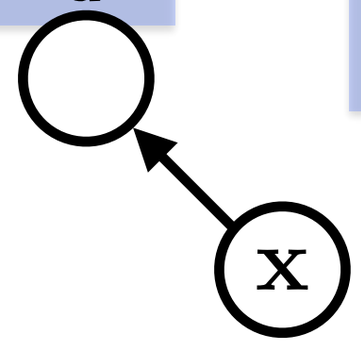


An **edge** represents a function argument (and also an data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

A **node** knows how to compute its value and the *value of its derivative w.r.t each argument (edge) times a derivative of an arbitrary input* $\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$.

$$f(\mathbf{u}) = \mathbf{u}^\top$$



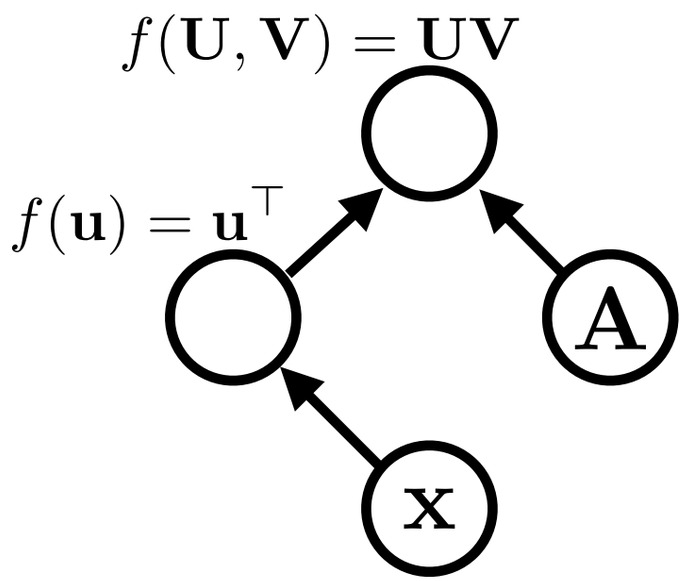
$$\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} = \left(\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} \right)^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A}$$

graph:

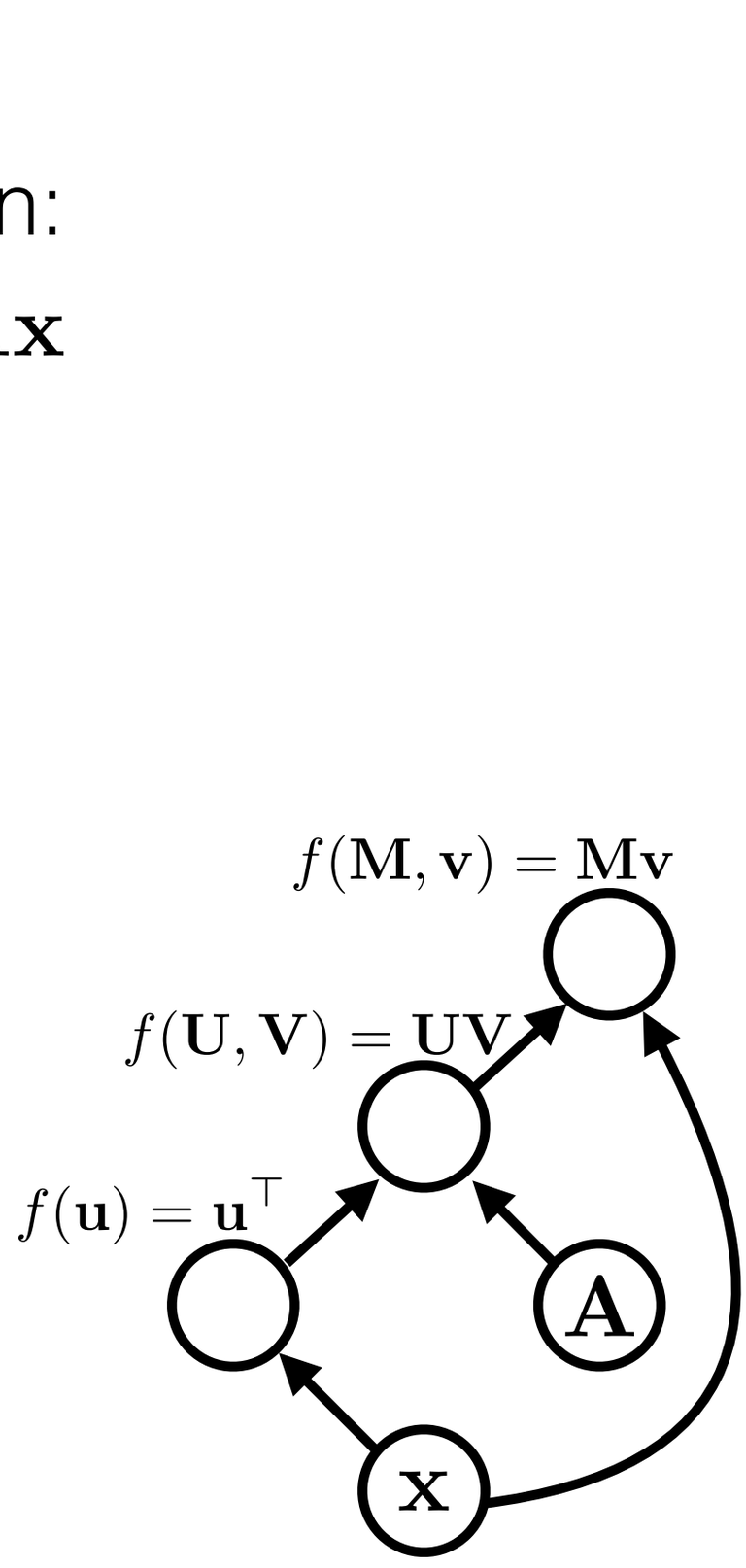
Functions can be nullary, unary,
binary, … *n*-ary. Often they are unary or binary.



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:

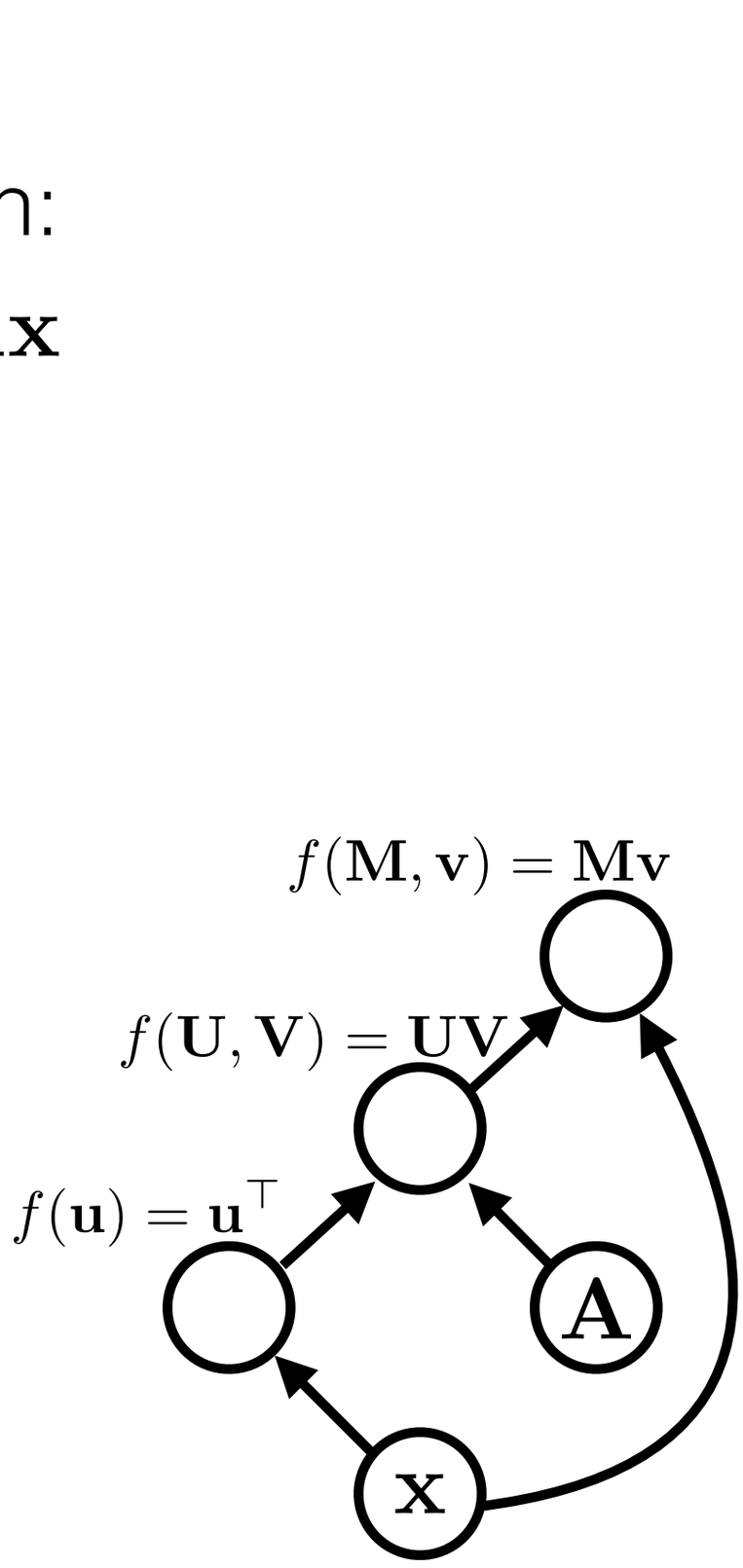


Computation graphs are directed and acyclic

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:



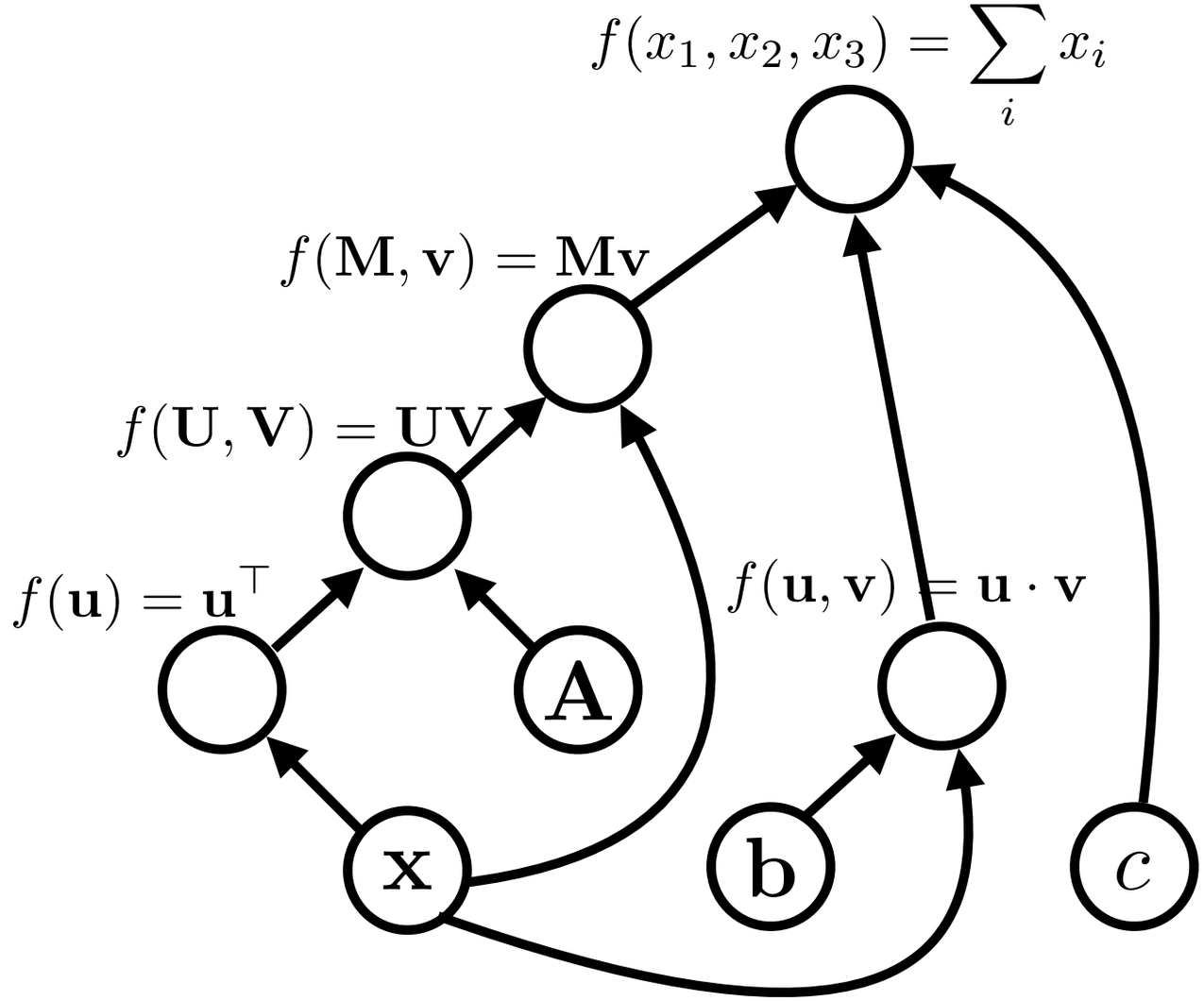
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{A}} = \mathbf{x}\mathbf{x}^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

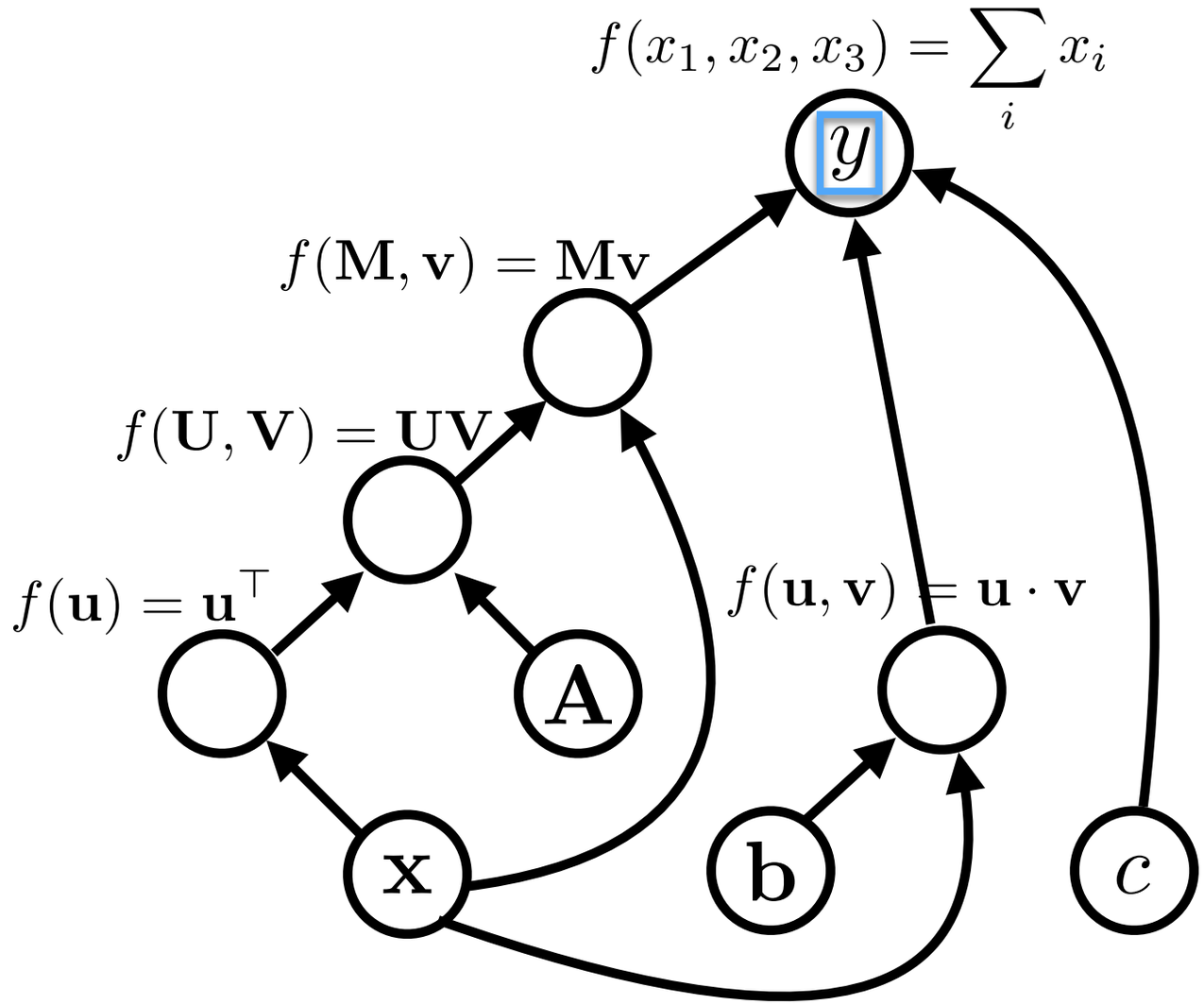
graph:



expression:

$$y = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

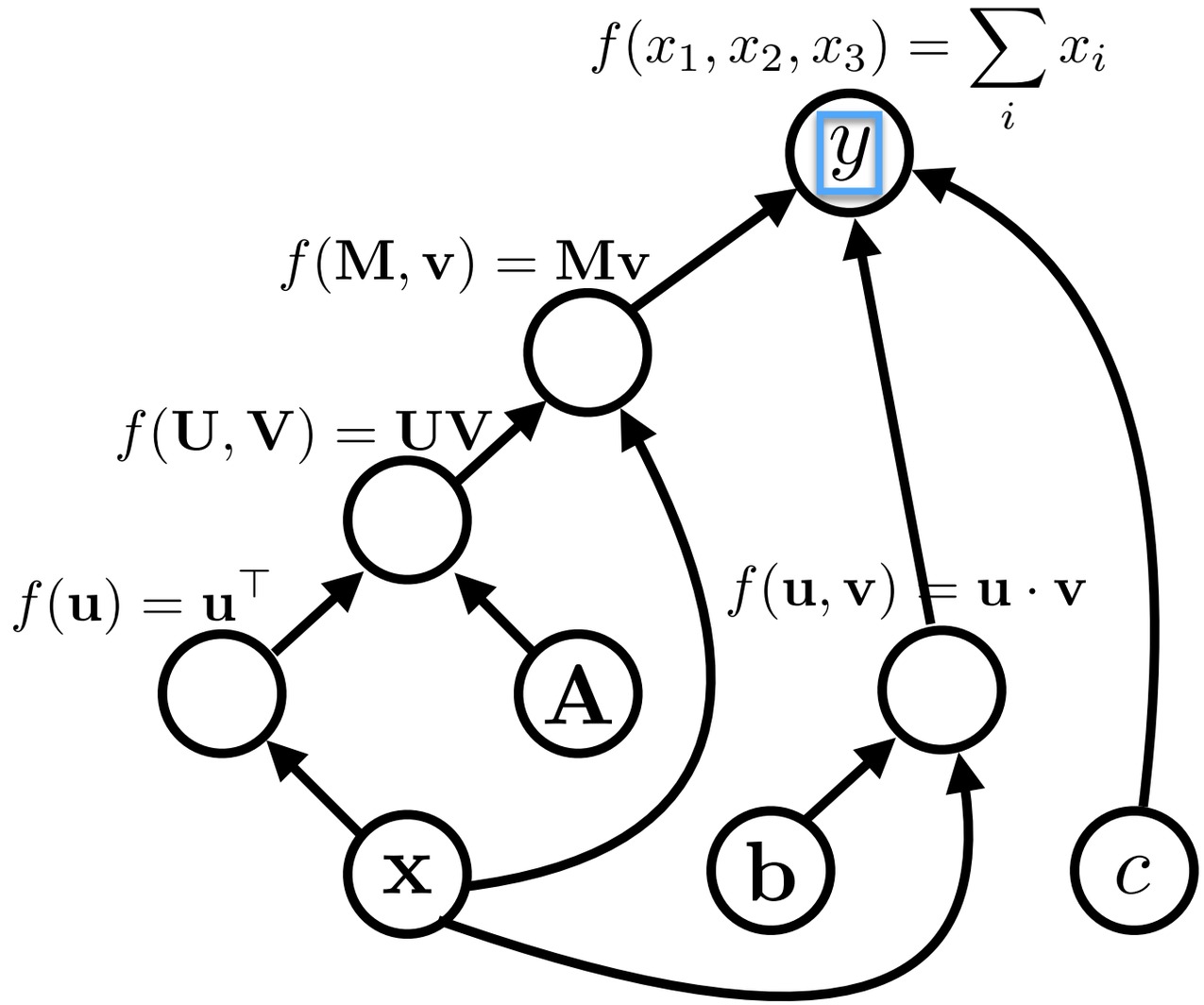
graph:



expression:

$$\boxed{y =} \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:



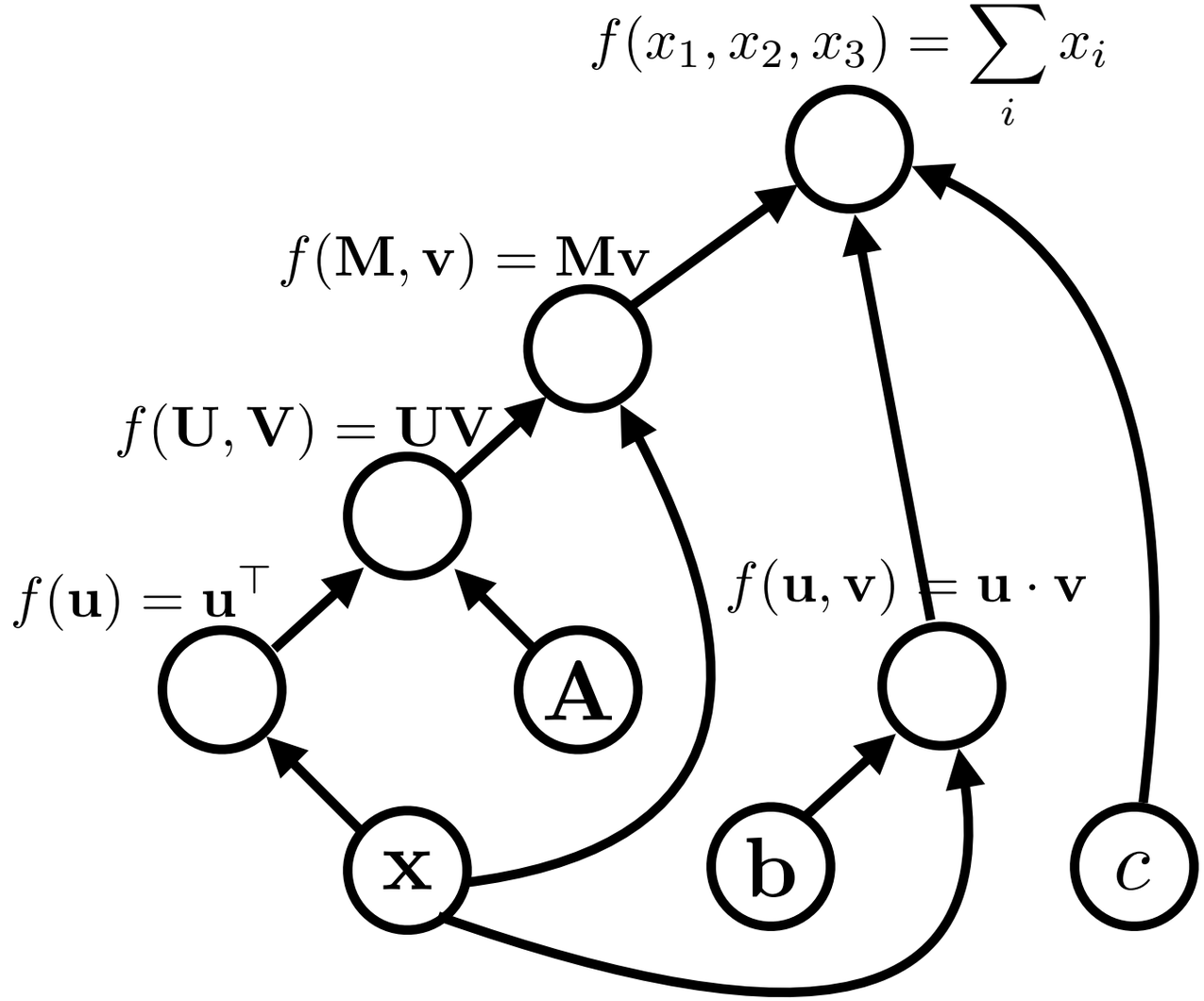
variable names are just labelings of nodes.

Algorithms

- **Graph construction**
- **Forward propagation**
    - Loop over nodes in topological order
        - Compute the value of the node given its inputs
        - *Given my inputs, make a prediction (or compute an "error" with respect to a "target output")*
- **Backward propagation**
    - Loop over the nodes in reverse topological order starting with a final goal node
        - Compute derivatives of final goal node value with respect to each edge's tail node
    - *How does the output change if I make a small change to the inputs?*

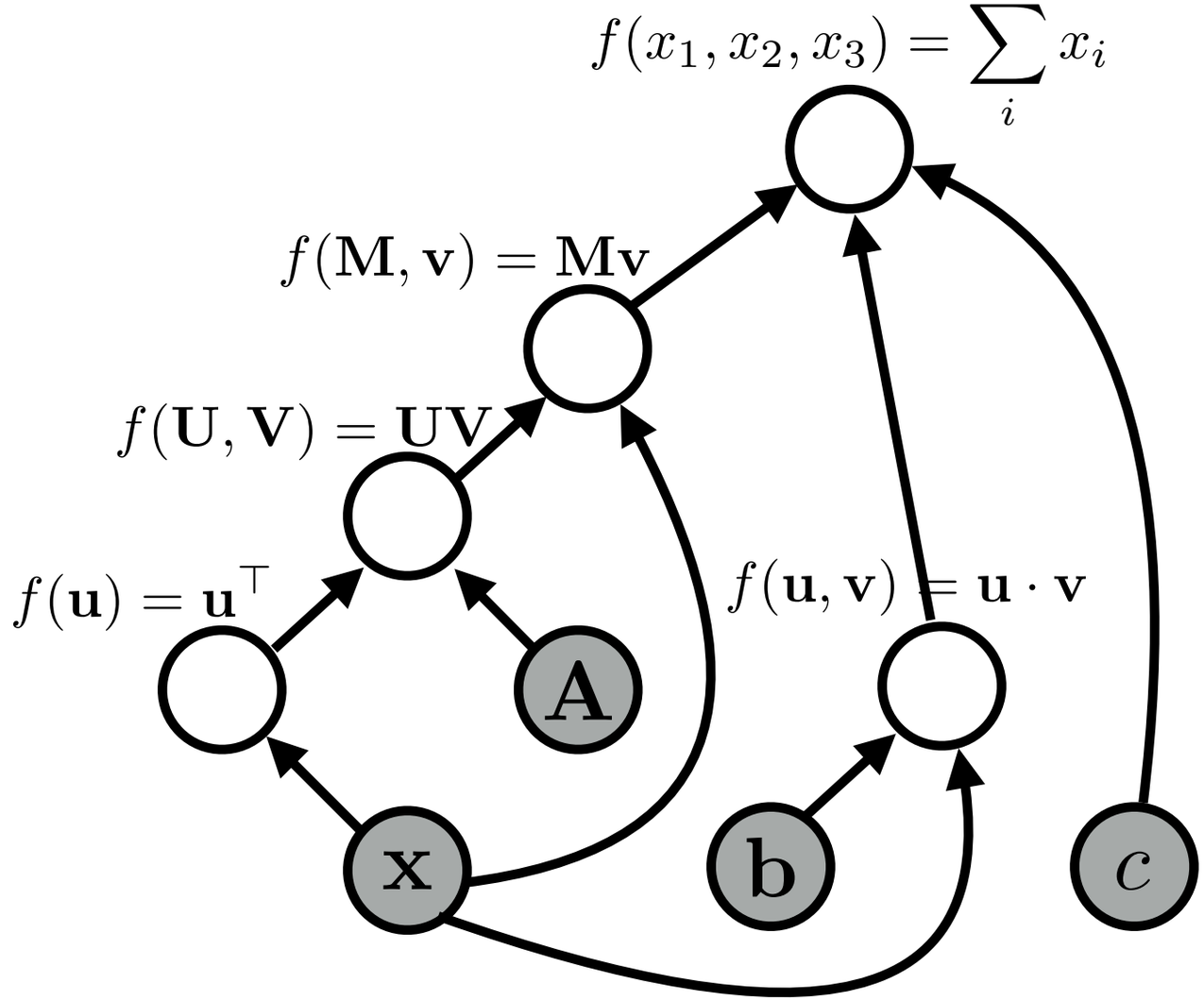
Forward Propagation

graph:



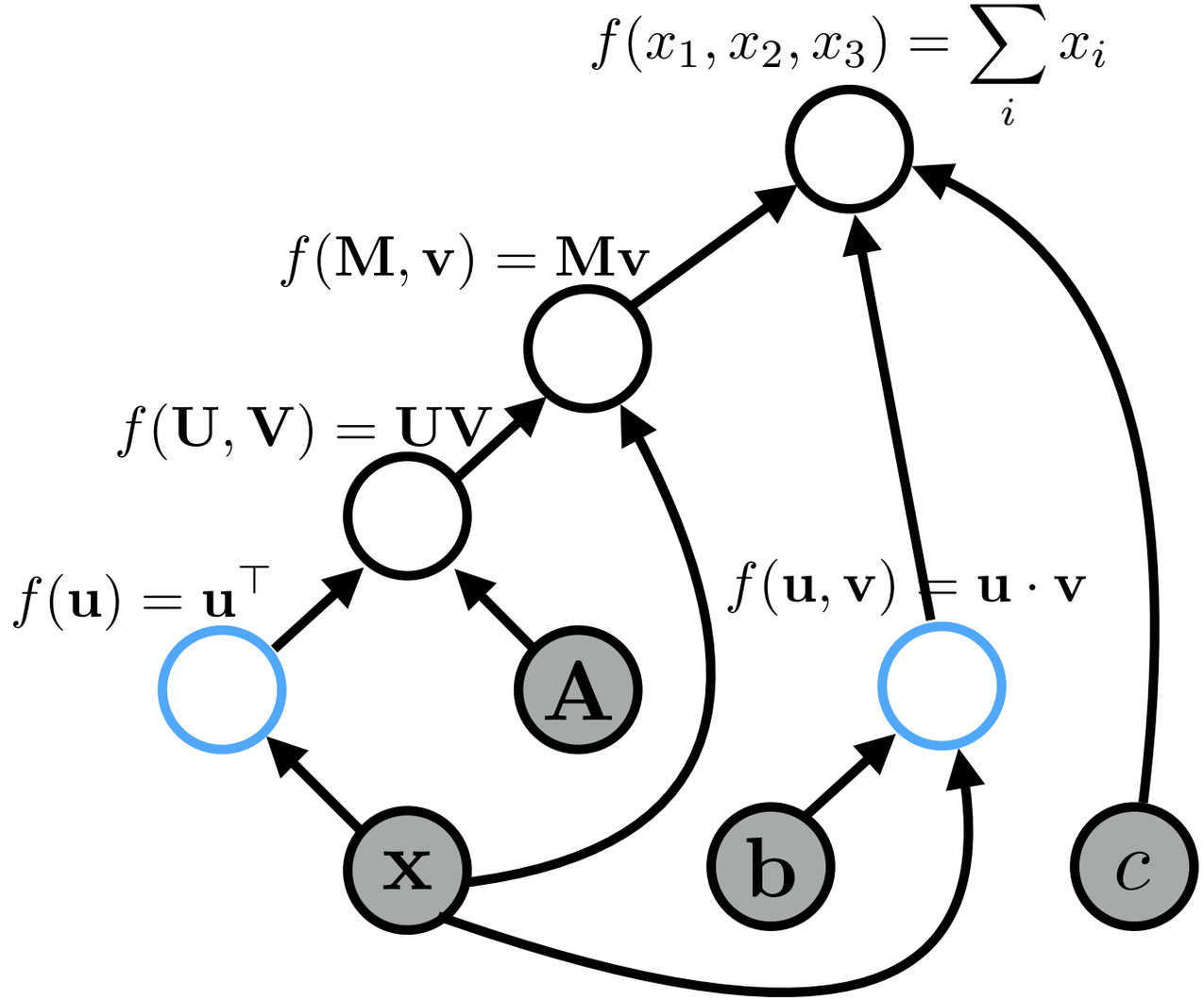
Forward Propagation

graph:



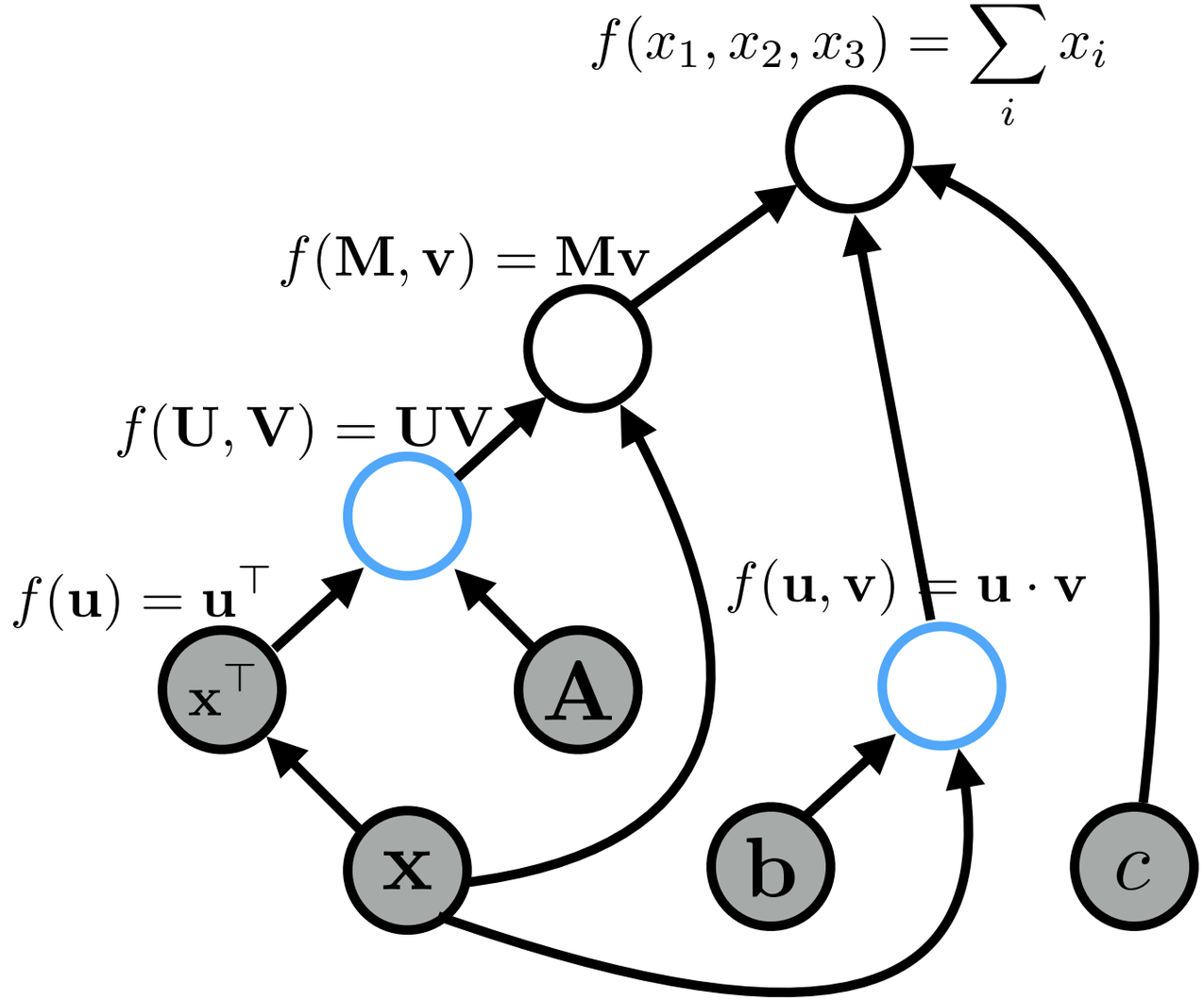
Forward Propagation

graph:



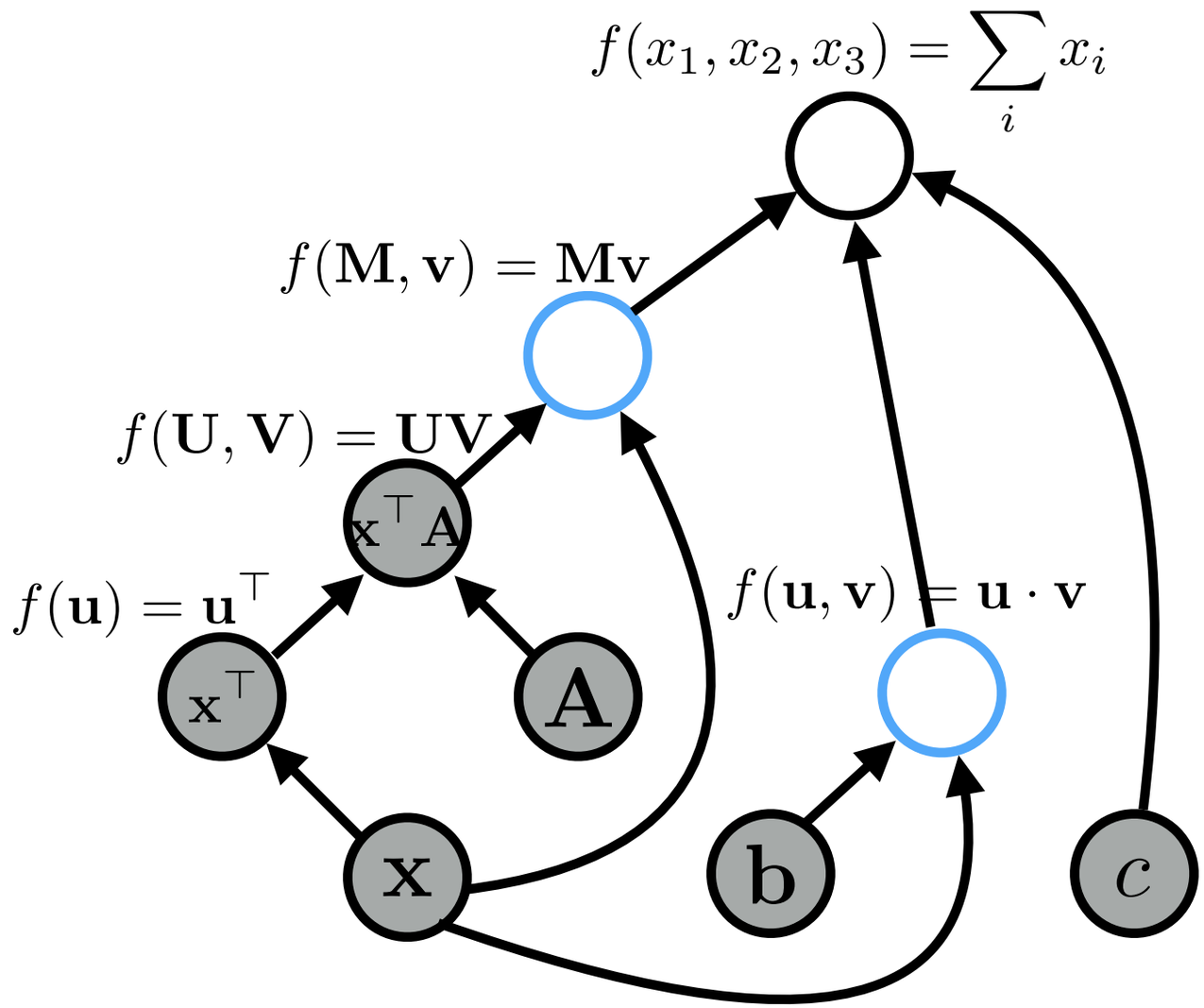
Forward Propagation

graph:



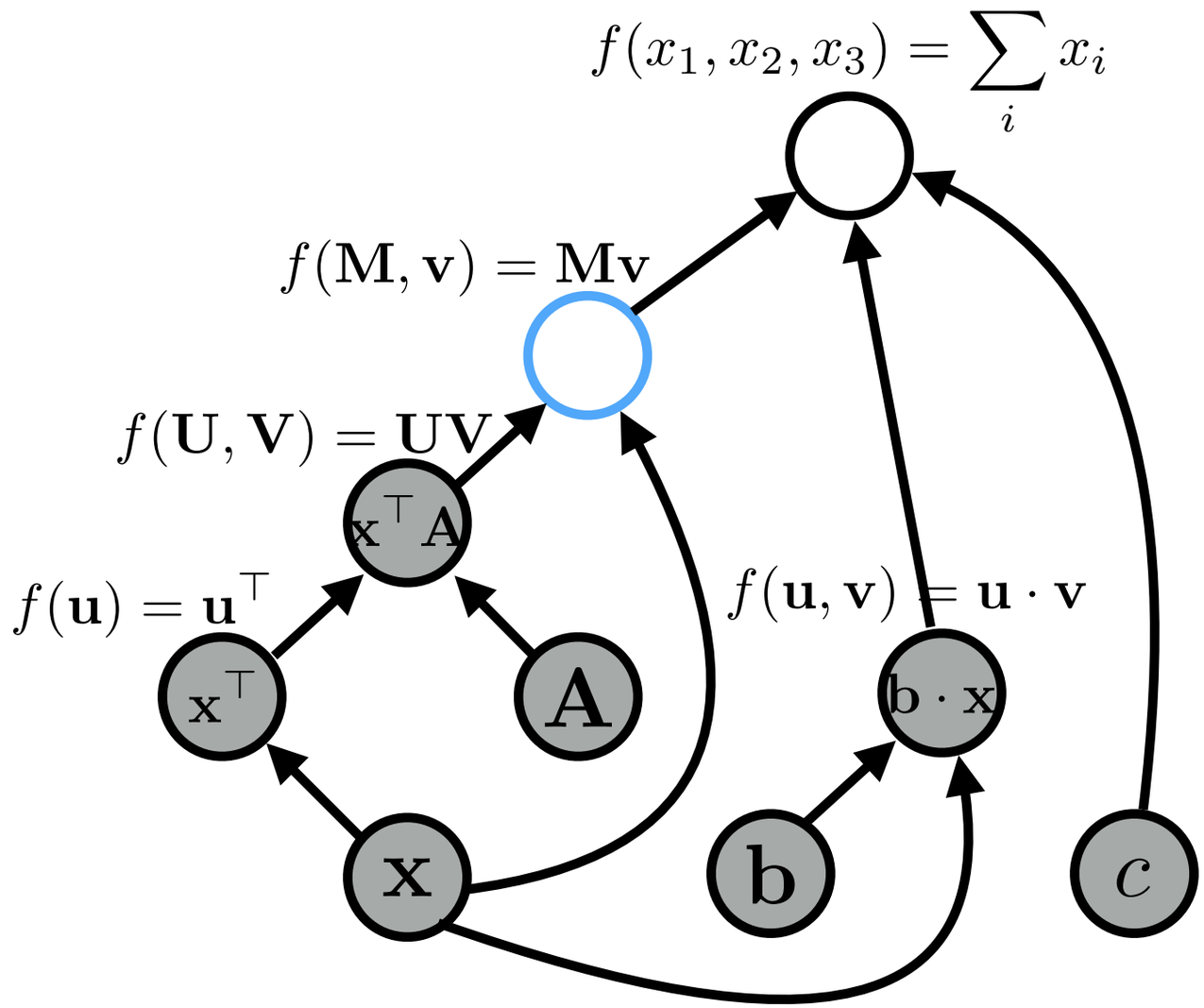
Forward Propagation

graph:



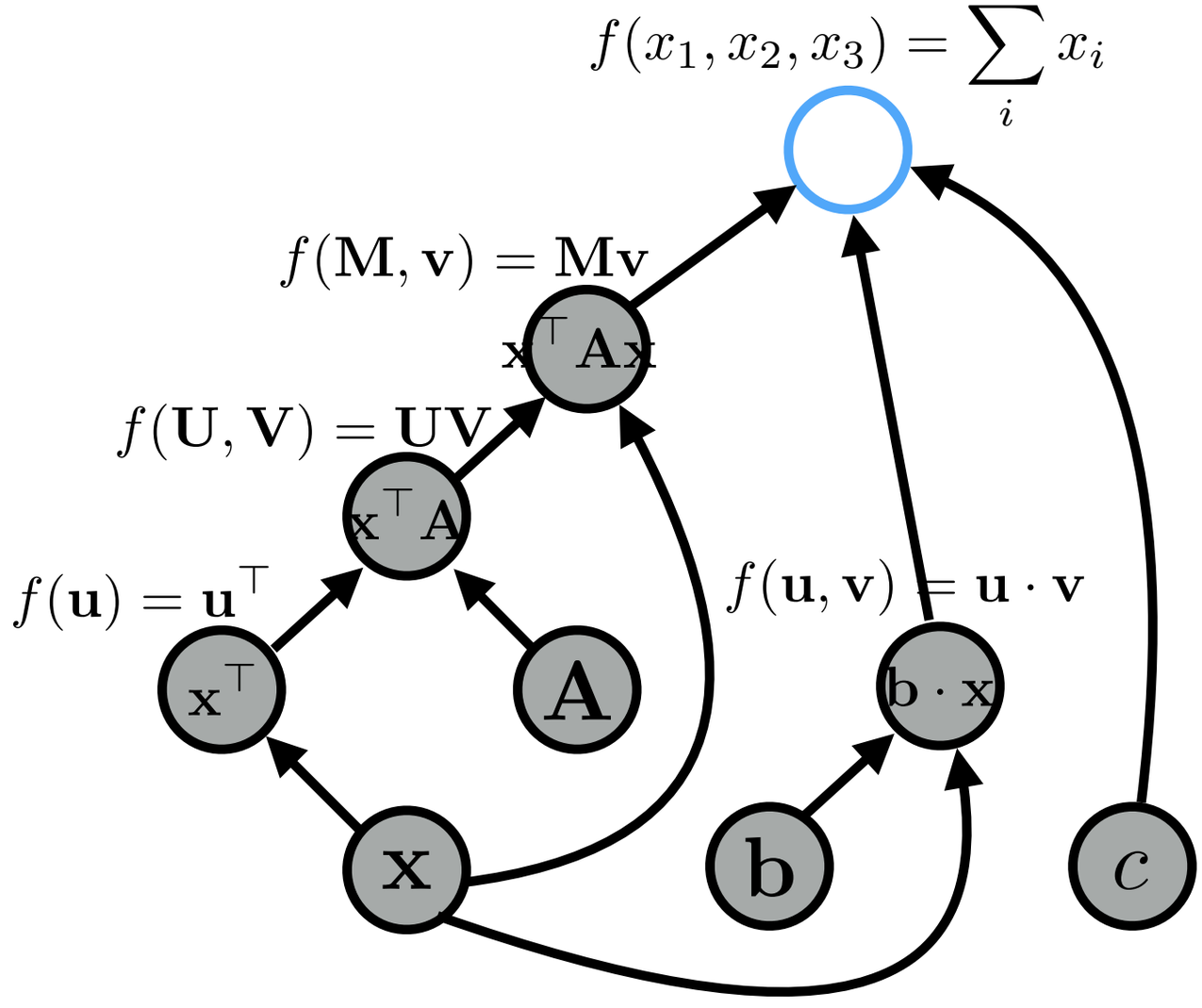
Forward Propagation

graph:



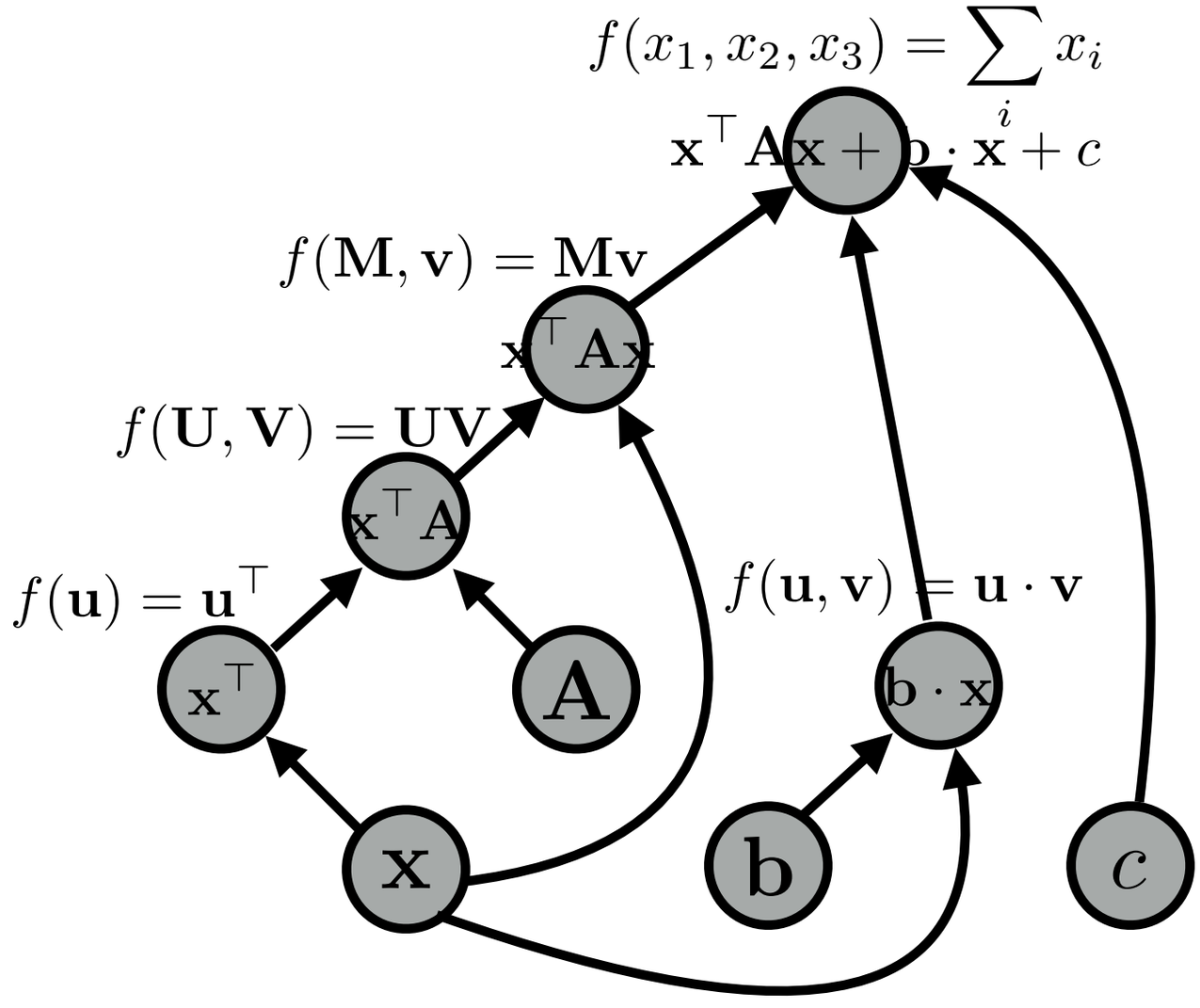
Forward Propagation

graph:



Forward Propagation

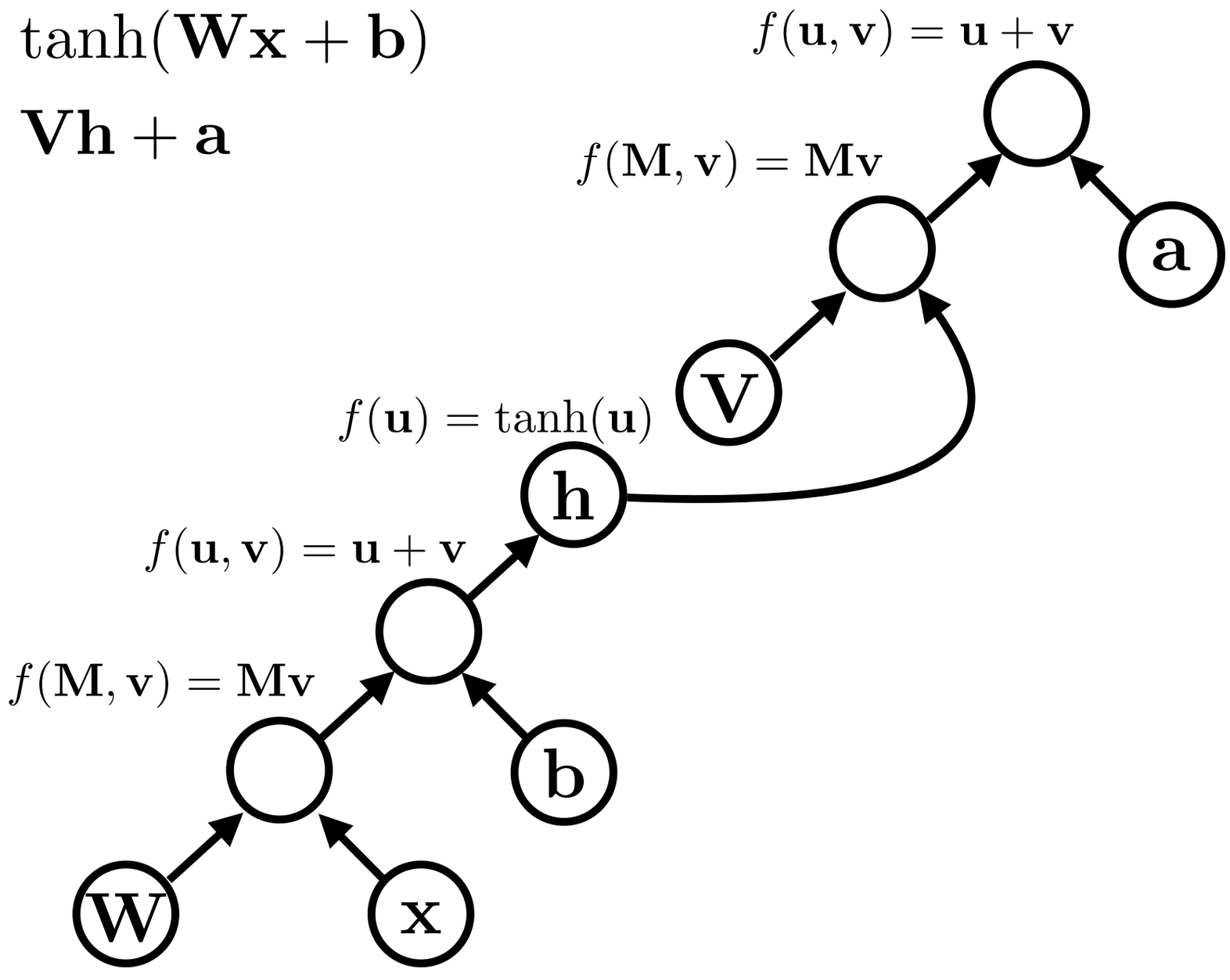
graph:



The MLP

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



Solving XOR with AD

(switch to code)

Tricks of the Trade

Working with NNs

- NNs are powerful because they have a lot of things that can be tuned
- Exhaustive exploration is difficult, but there are some useful tricks
- In general
  - Make sure you can overfit on a few examples
  - Then try to get a model that overfits on the training data (i.e., get rid of bias)
  - Then try to improve the overfit model to deal with high variance problems
  - Then you will have solved your problem

Reduce bias

Reduce bias

Tricks of the Trade

- **Better initialization and optim.**
- Get more training data
- **Standardize your features**
- Use engineered features
- Use pre-trained features
- Data augmentation
- Weight regularization
- Early stopping
- **Drop Out**
- **Batch & Layer Norm**

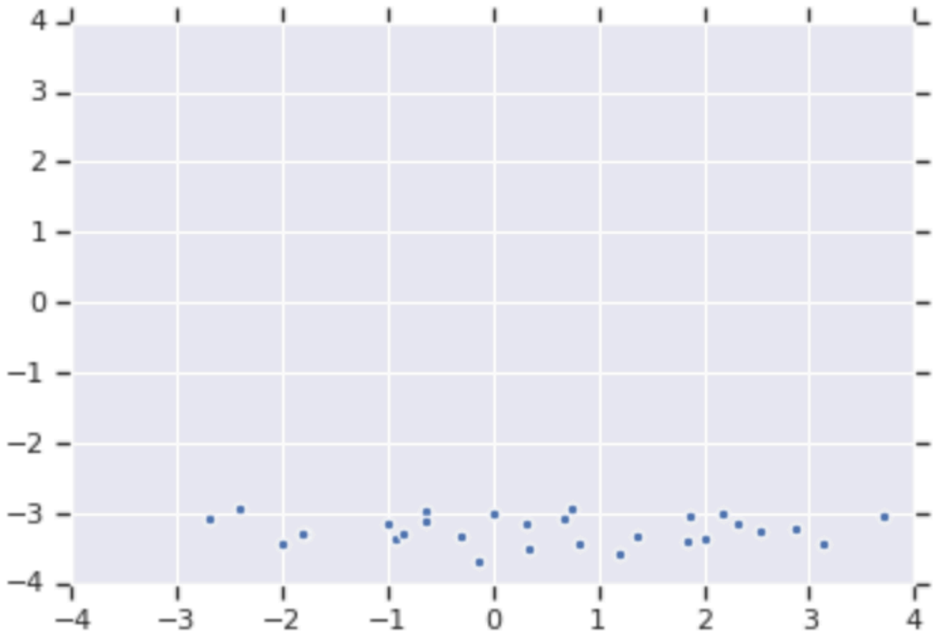
Optimization is Hard

- We have no guarantees about convergence in neural networks

- We have a few tricks
  - Make sure inputs and weights are statistically well behaved (i.e., are described by a well-behaved distribution during training)
  - Use large numbers of parameters
  - Use better optimizers

Input Normalization

If these are your raw features, learning may be slow:

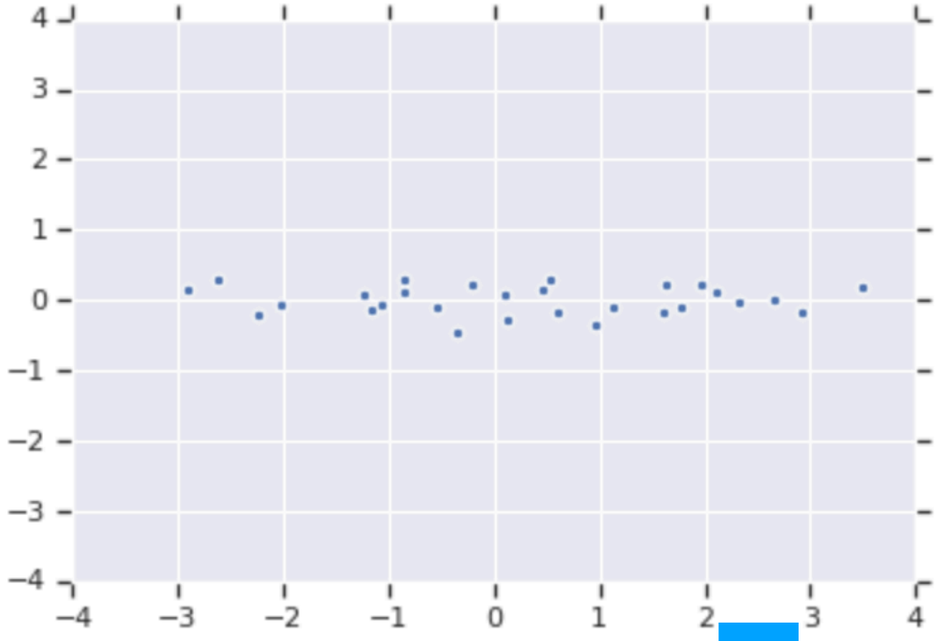


$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_{i,:}$$

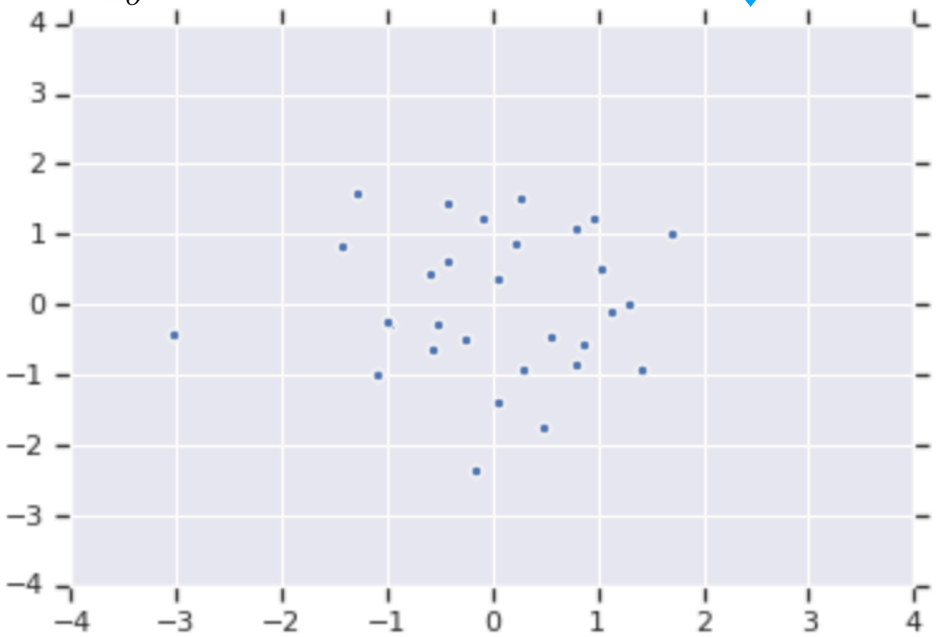
$$\mathbf{x}_\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}||\mathbf{x}_{i,:} - \overline{\mathbf{x}}||^2}$$

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \overline{\mathbf{x}}}{\mathbf{x}_\sigma}$$

$\mathbf{x} - \overline{\mathbf{x}}$ **mean is now 0.**

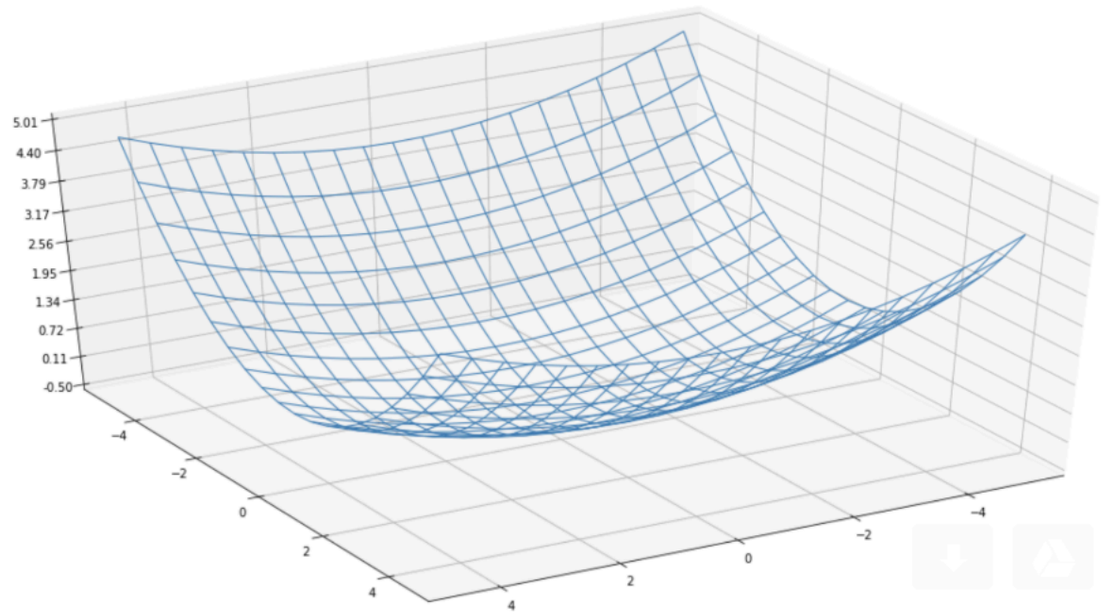


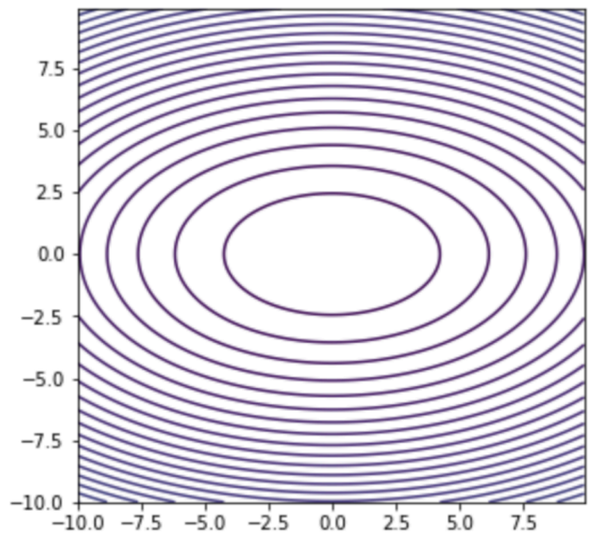
$\frac{\mathbf{x} - \overline{\mathbf{x}}}{\mathbf{x}_\sigma}$ **mean=0, var=1.**



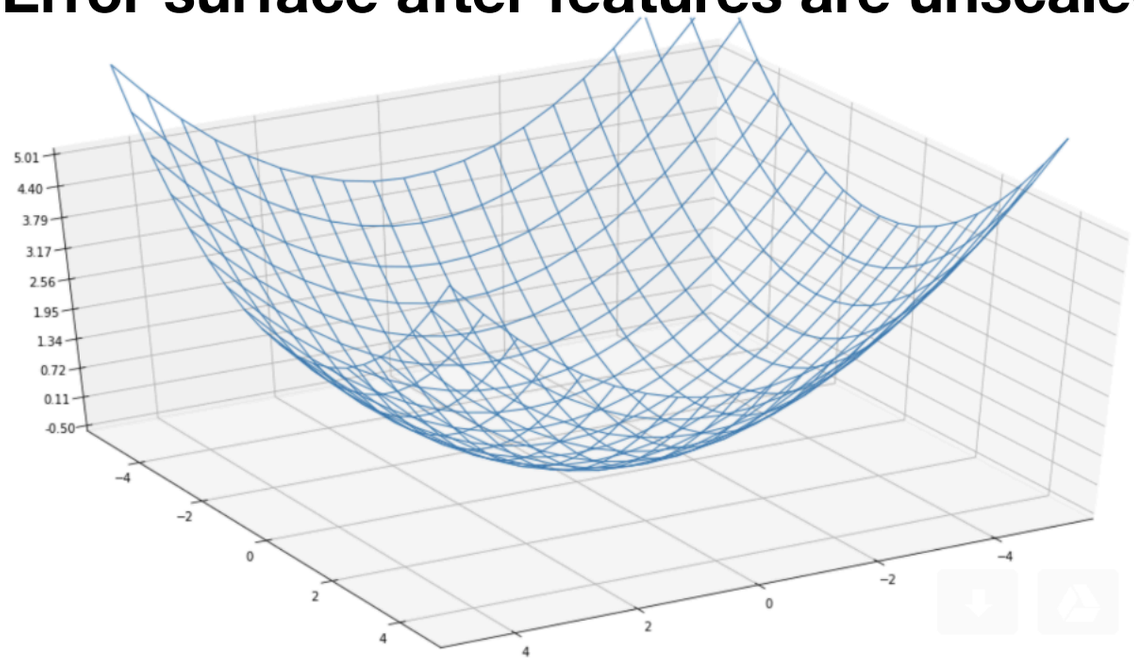
Input Normalization

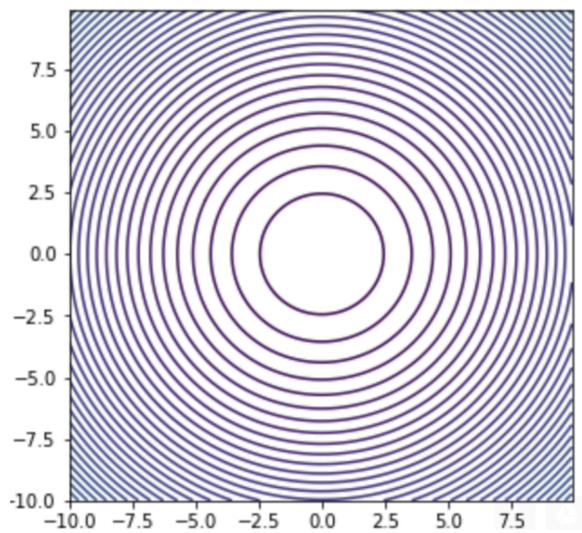
Error surface when features are unscaled





Error surface after features are unscaled



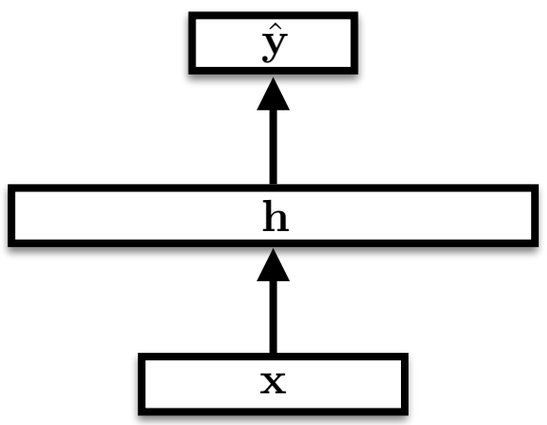


Input Normalization

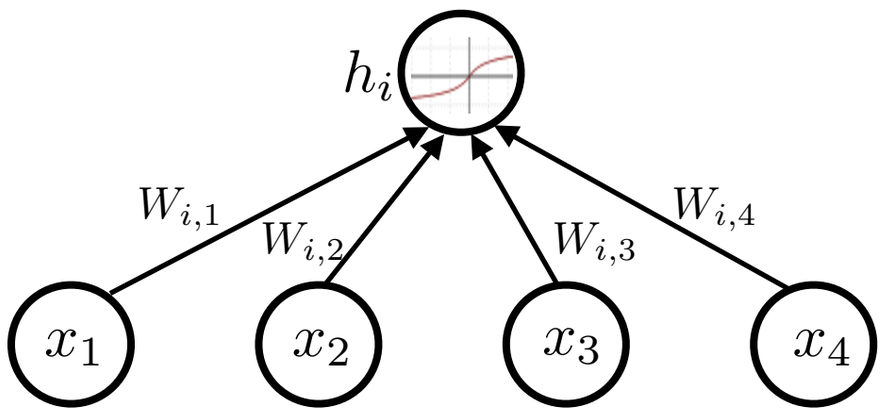
- Result of input normalization: you can use **larger learning rates**, and you will be less sensitive to getting lucky with respect to initial parameters

- Note: Make sure to use the same mean and stddev from training time at test time!

- Open questions:
  - How should we initialize weights?
  - What about standardizing “internal inputs”?
  - Can we do something smarter with our update rules to deal with badly shaped surfaces?



Weight Initialization



$$h_i = g\left(\sum_j x_j \times W_{i,j}\right)$$

Recall the activation rule for an individual neuron.

If the x_j's have unit variance, what would it take for z (the pre-activation sum) to also have unit variance?

Recall if X and Y are random variables with variances σ_X^2 and σ_Y^2 then X+Y has variance $\sigma_X^2 + \sigma_Y^2$, thus we want to scale the variance by 1/N where N is the number of inputs.

$$W_{i,j} \sim \mathcal{N}(0,1)/\sqrt{N}$$

There are a number of different variants based on what g is used, but setting the variance carefully can significantly help with learning!

Smarter Update Rules

- Stochastic gradient descent takes a fixed step size in the direction of steepest descent. Since each update is (usually) only computed from a very small number of training instances, they will generally be noisy, albeit correlated in expectation.

- Momentum:
  - Consider the what the direction of the steepest ascent is on a river bank: it will point you toward the water. However, if you want to get to a global minimum, it would be wise for you to follow to the river rather than to rush into the river (and, if your step-size is too large) up the other side, and then back and forth.

  - Momentum keeps track of changes in gradients and allows you to build up speed in a particular direction if the gradients point reliably in that direction; it also allows you to disregard abrupt changes since these are likely unreliable noise, not a meaningful learning signal.

Momentum

Classic (stochastic) gradient descent update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$$

SGD with momentum update:

How quickly do we adapt to the new gradient information?

$$\mathbf{p}_t = \beta \mathbf{p}_{t-1} + (1-\beta)\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \mathbf{p}_t$$

Note that this requires keeping around a second set of "parameters" to store the momentum (which is just an exponentially weighted average of the historical gradients)! We also have an additional hyperparameter.

RMSprop

Intuition: *normalize the gradient by the historical magnitude of the gradient*. Larger gradient dimensions should be dampened more; smaller ones should be magnified. But what is the magnitude of the gradient dimension? It's just the square root of the historical average of the squared gradient (i.e., the root-mean-squared, RMS, gradient).

$$\mathbb{E}[g^2]_t = \beta \mathbb{E}[g^2]_{t-1} + (1-\beta)\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}\right)^2$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t}} \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$$

Recall SGD with momentum update:

$$\mathbf{p}_t = \beta \mathbf{p}_{t-1} + (1-\beta)\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \mathbf{p}_t$$

Like with momentum, RMSProp also requires keeping around a second set of "parameters".

Adam

Intuition: Let's combine RMSProp (standardize gradient magnitudes) and Momentum (keep track of the direction we're moving in).

"Momentum"

$$\mathbf{p}_t = \beta_1 \mathbf{p}_{t-1} + (1 - \beta_1) \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$$

"RMSprop"

$$\mathbb{E}[g^2]_t = \beta_2 \mathbb{E}[g^2]_{t-1} + (1 - \beta_2) \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \right)^2$$

Perform update

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t}} \cdot \mathbf{p}_t$$

As a hybrid of momentum and RMSProp, Adam requires keeping around two extra sets of "parameters", so memory consumption can be an issue.

Additionally, some problems are sensitive to settings of the betas and learning rate.

Let’s switch to code

Better Internal Representations

- But the learned “internal features” are what makes deep learning powerful. Several techniques are designed to improve the quality and learnability of internal representations.

- **Drop out**: avoid feature coadaptation

- Batch and Layer normalization: Normalization of input features improves learning (use larger learning rates, find better solutions). We can initialize them so they start off well behaved, but **shouldn’t we worry about how they are normalized as learning continues?**

DropOut

- US Army wanted to be able to classify pictures of tanks
    - trained a neural network
    - and it did really well

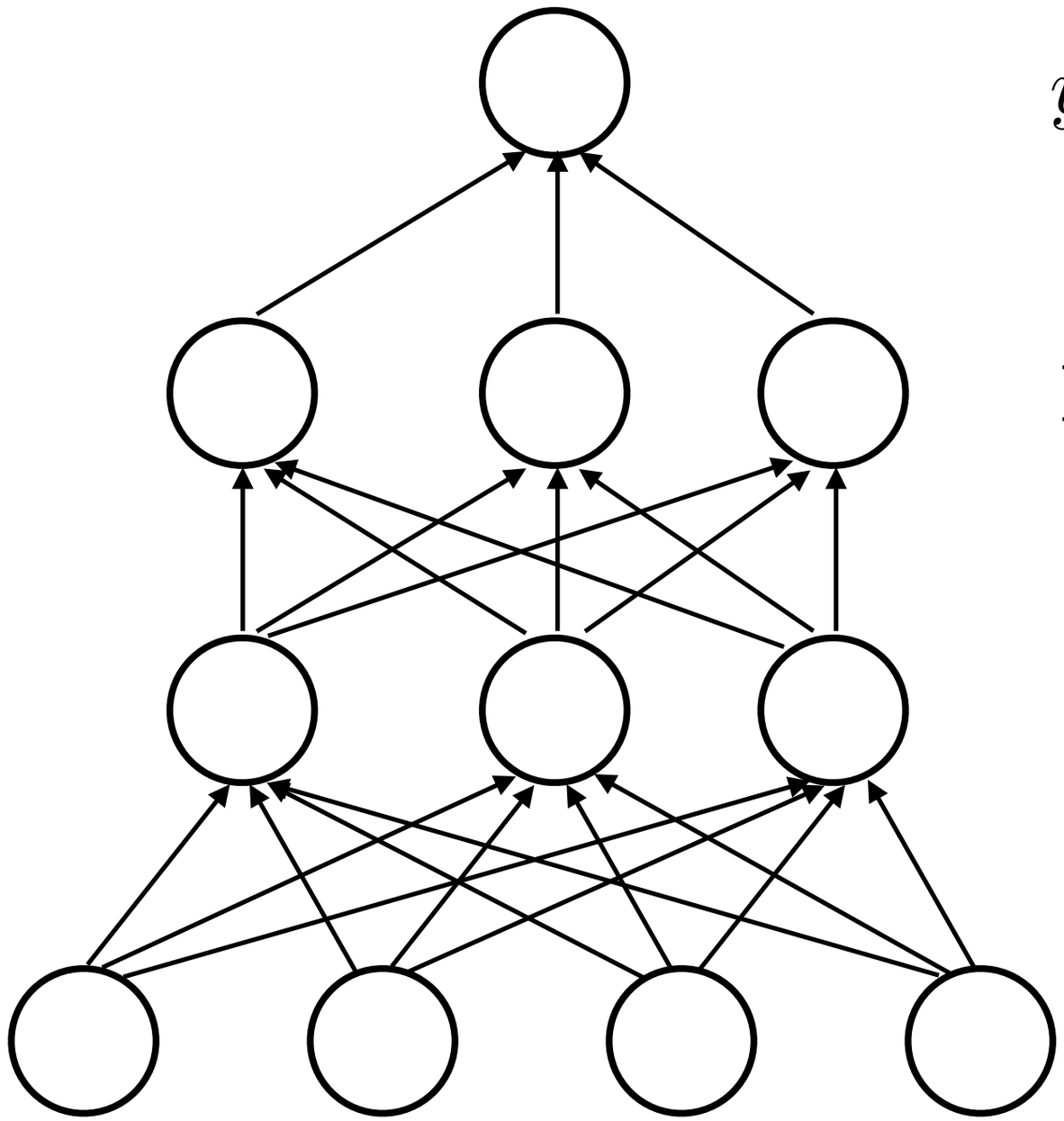
Thomas Poggio

- … in fact, it did too well
- Turns out there was an extremely informative feature that could be extracted from the image, but was not reliable; **the classifier came to depend exclusively on that feature**

DropOut

- Intuition behind DropOut
    - make sure that the classifier can’t depend on any single feature to the exclusion of all others
    - make learned features become generally useful and robustly encode information
- Strategy:
    - Each hidden unit is set to 0 with some probability (pairs particularly nicely with ReLUs which also zero out units)
    - Tune the dropout rate on held-out data

DropOut



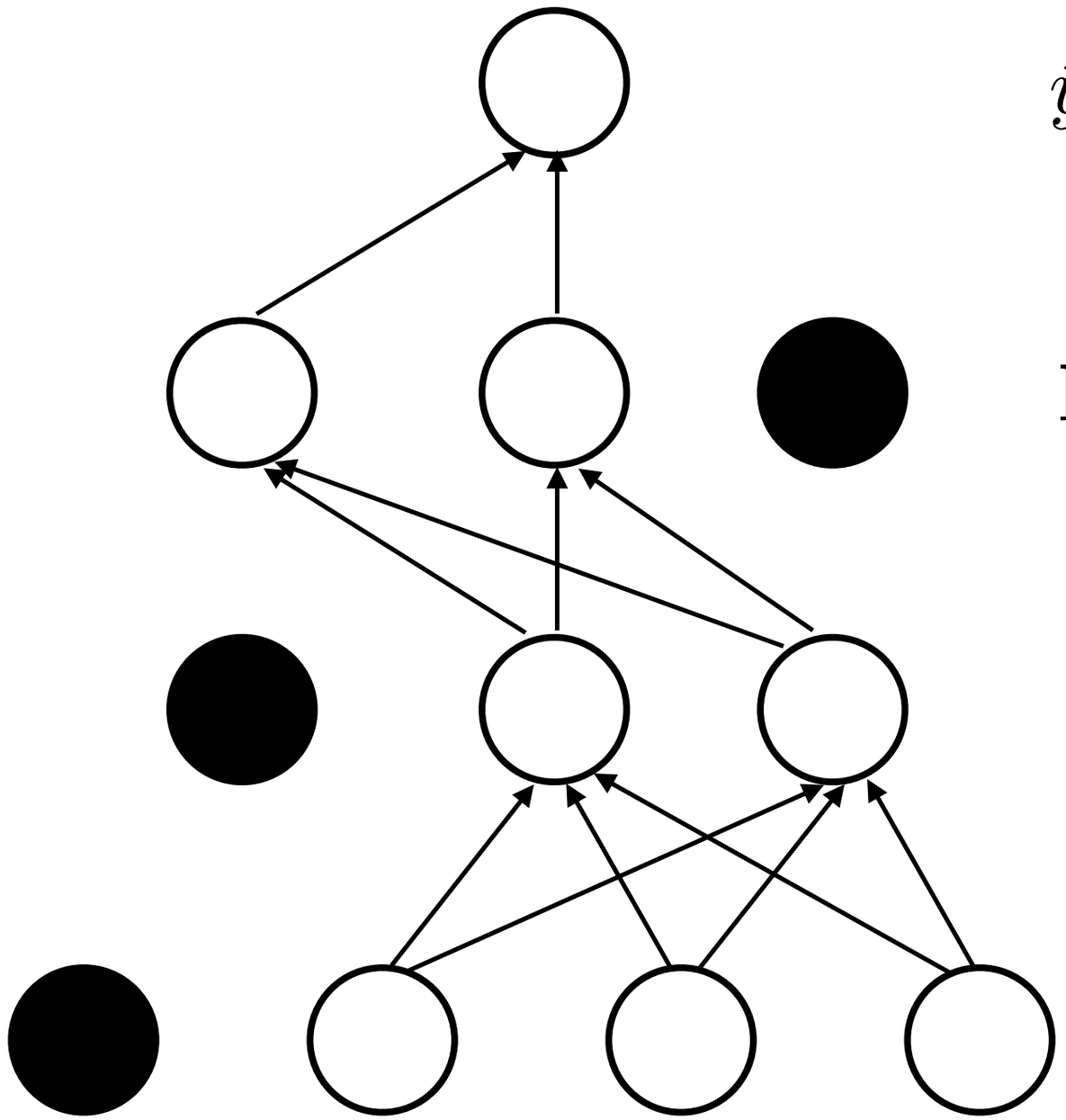
$$\hat{y} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

$$\mathbf{h}^{(2)} = \text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

$$\mathbf{h}^{(1)} = \text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$\mathbf{x}$

DropOut



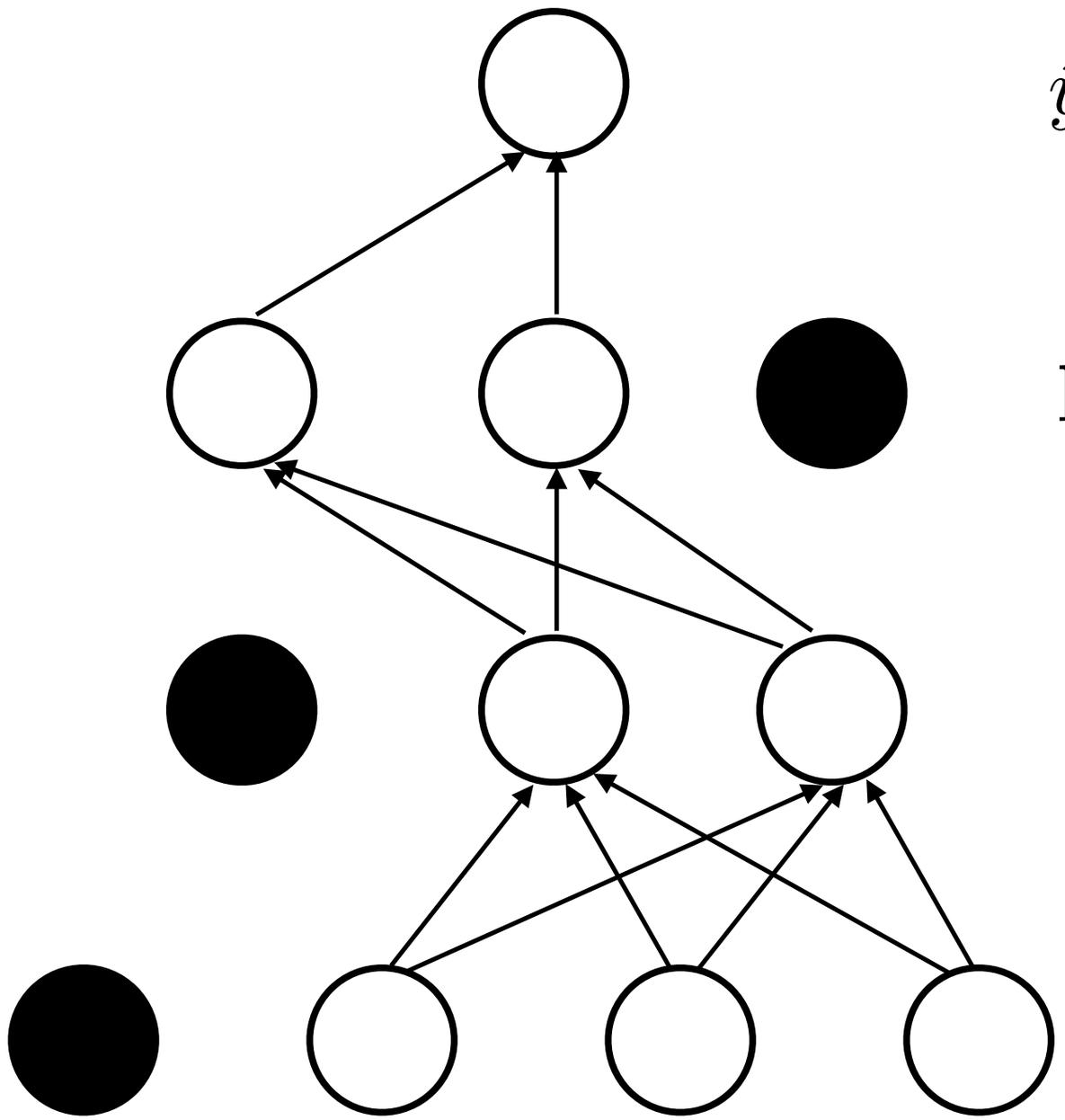
$$\hat{y} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

$$\mathbf{h}^{(2)} = \frac{\text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \odot \mathbf{m}^{(2)}}{p}$$

$$\mathbf{h}^{(1)} = \frac{\text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \odot \mathbf{m}^{(1)}}{p}$$

$$\frac{\mathbf{x} \odot \mathbf{m}^{(0)}}{p}$$

DropOut



$$\hat{y} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

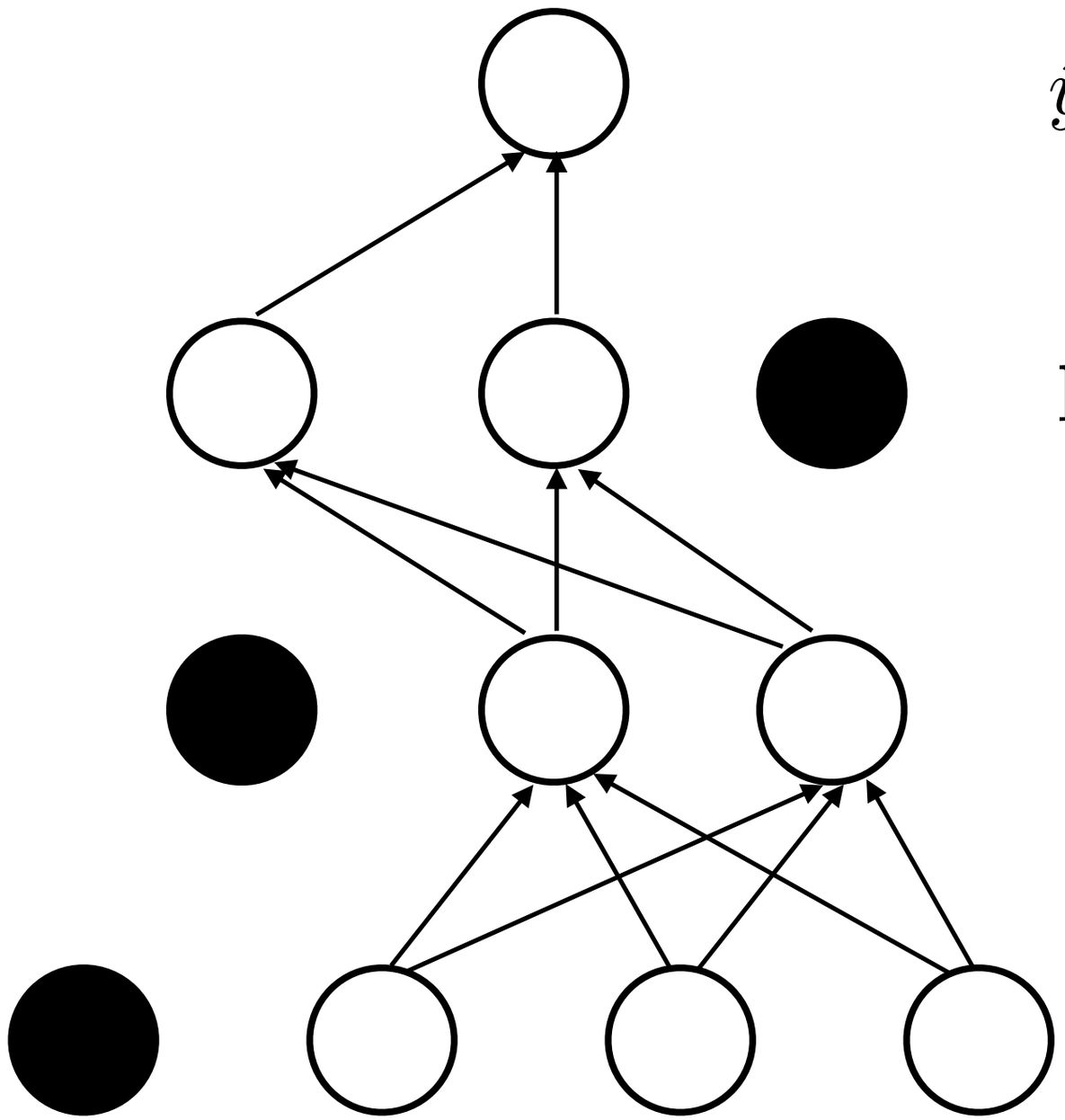
Random mask of 0's and 1's

$$\mathbf{h}^{(2)} = \frac{\text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \odot \mathbf{m}^{(2)}}{p}$$

$$\mathbf{h}^{(1)} = \frac{\text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \odot \mathbf{m}^{(1)}}{p}$$

$$\frac{\mathbf{x} \odot \mathbf{m}^{(0)}}{p}$$

DropOut



$$\hat{y} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

Random mask of 0's and 1's

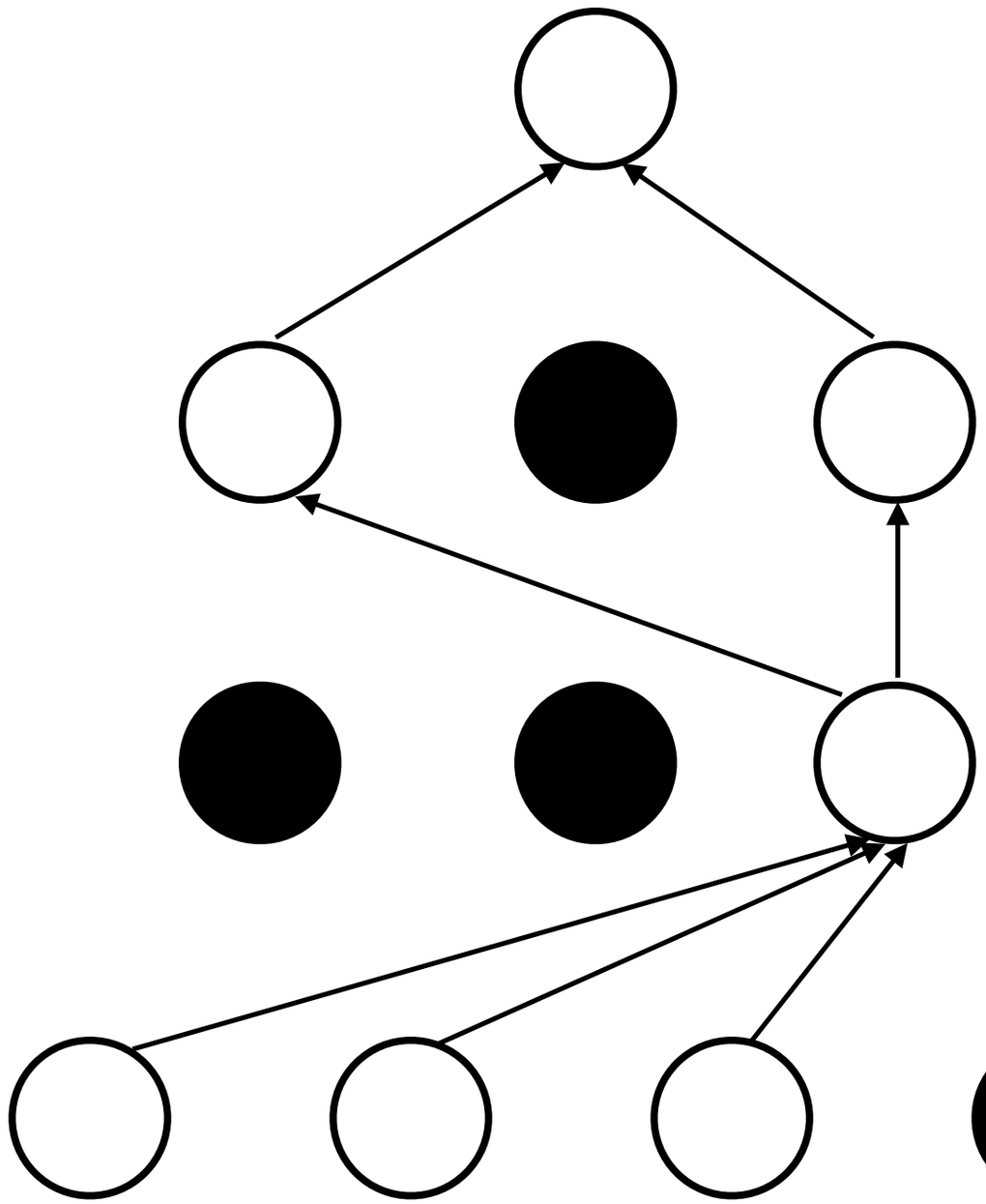
$$\mathbf{h}^{(2)} = \frac{\text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \odot \mathbf{m}^{(2)}}{p}$$

$$\mathbf{h}^{(1)} = \frac{\text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \odot \mathbf{m}^{(1)}}{p}$$

Dropout probability

$$\frac{\mathbf{x} \odot \mathbf{m}^{(0)}}{p}$$

DropOut



$$\hat{y} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

$$\mathbf{h}^{(2)} = \frac{\text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \odot \mathbf{m}^{(2)}}{p}$$

$$\mathbf{h}^{(1)} = \frac{\text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \odot \mathbf{m}^{(1)}}{p}$$

$$\frac{\mathbf{x} \odot \mathbf{m}^{(0)}}{p}$$

DropOut

- Each time you see a training instance, you will sample a different set of dropout masks

- At test time, you don’t add any noise -> give the classifiers the best chance to make a classification

- Intuitively: the model is encouraged to learn to make decisions that integrate a lot of weak, possibly unreliable signals. **What do you think this does to interpretability?**

Normalize Internal Inputs

- Very useful trick: train deeper networks, use larger learning rates, be more robust to hyperparameters

- Goal: in a deep network, we would like to be able to normalize $\mathbf{h}^{(\ell)}$.
  (Actually, we will normalize the pre-activation values, $\mathbf{z}^{(\ell)}$.)

Model

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\mathbf{h}^{(\ell)} = g(\mathbf{z}^{(\ell)})$$

Normalize Internal Inputs

- Very useful trick: train deeper networks, use larger learning rates, be more robust to hyperparameters

- Goal: in a deep network, we would like to be able to normalize $\mathbf{h}^{(\ell)}$.
  (Actually, we will normalize the pre-activation values, $\mathbf{z}^{(\ell)}$.)

Model

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\mathbf{h}^{(\ell)} = g(\mathbf{z}^{(\ell)})$$

Minibatched version

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{H}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\mathbf{H}^{(\ell)} = g(\mathbf{Z}^{(\ell)})$$

where

$$\mathbf{Z}^{(\ell)} = (\mathbf{z}_1^{(\ell)}, \mathbf{z}_2^{(\ell)}, \ldots, \mathbf{z}_m^{(\ell)})$$

Normalize Internal Inputs

- Very useful trick: train deeper networks, use larger learning rates, be more robust to hyperparameters

- Goal: in a deep network, we would like to be able to normalize $\mathbf{h}^{(\ell)}$.
  (Actually, we will normalize the pre-activation values, $\mathbf{z}^{(\ell)}$.)

Minibatched version

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{H}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\mathbf{H}^{(\ell)} = g(\mathbf{Z}^{(\ell)})$$

where

$$\mathbf{Z}^{(\ell)} = (\mathbf{z}_1^{(\ell)}, \mathbf{z}_2^{(\ell)}, \ldots, \mathbf{z}_m^{(\ell)})$$

Batch-normed version

$$\overline{\mathbf{z}}^{(\ell)} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{z}_m^{(\ell)}$$

$$\overline{\mathbf{z}}_\sigma^{(\ell)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\overline{\mathbf{z}}^{(\ell)} - \mathbf{z}_m^{(\ell)}\right)^2}$$

$$\tilde{\mathbf{Z}}^{(\ell)} = \gamma \cdot \left(\frac{\overline{\mathbf{Z}}^{(\ell)} - \overline{\mathbf{z}}^{(\ell)}}{\overline{\mathbf{z}}_\sigma^{(\ell)}}\right) + \beta$$

$$\mathbf{H}^{(\ell)} = g(\tilde{\mathbf{Z}}^{(\ell)})$$

Normalize Internal Inputs

- Very useful trick: train deeper networks, use larger learning rates, be more robust to hyperparameters

- Goal: in a deep network, we would like to be able to normalize $\mathbf{h}^{(\ell)}$.
  (Actually, we will normalize the pre-activation values, $\mathbf{z}^{(\ell)}$.)

Minibatched version

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{H}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

Will be absorbed by the mean!

$$\mathbf{H}^{(\ell)} = g(\mathbf{Z}^{(\ell)})$$

where

$$\mathbf{Z}^{(\ell)} = (\mathbf{z}_1^{(\ell)}, \mathbf{z}_2^{(\ell)}, \ldots, \mathbf{z}_m^{(\ell)})$$

Batch-normed version

$$\overline{\mathbf{z}}^{(\ell)} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{z}_m^{(\ell)}$$

$$\overline{\mathbf{z}}_\sigma^{(\ell)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\overline{\mathbf{z}}^{(\ell)} - \mathbf{z}_m^{(\ell)}\right)^2}$$

$$\tilde{\mathbf{Z}}^{(\ell)} = \gamma \cdot \left(\frac{\overline{\mathbf{Z}}^{(\ell)} - \overline{\mathbf{z}}^{(\ell)}}{\overline{\mathbf{z}}_\sigma^{(\ell)}}\right) + \beta$$

$$\mathbf{H}^{(\ell)} = g(\tilde{\mathbf{Z}}^{(\ell)})$$

Normalize Internal Inputs

- Very useful trick: train deeper networks, use larger learning rates, be more robust to hyperparameters

- Goal: in a deep network, we would like to be able to normalize $\mathbf{h}^{(\ell)}$.
  (Actually, we will normalize the pre-activation values, $\mathbf{z}^{(\ell)}$.)

Minibatched version

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{H}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

Will be absorbed by the mean!

$$\mathbf{H}^{(\ell)} = g(\mathbf{Z}^{(\ell)})$$

where

$$\mathbf{Z}^{(\ell)} = (\mathbf{z}_1^{(\ell)}, \mathbf{z}_2^{(\ell)}, \ldots, \mathbf{z}_m^{(\ell)})$$

Batch-normed version

$$\overline{\mathbf{z}}^{(\ell)} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{z}_m^{(\ell)}$$

$$\overline{\mathbf{z}}_\sigma^{(\ell)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\overline{\mathbf{z}}^{(\ell)} - \mathbf{z}_m^{(\ell)}\right)^2}$$

$$\tilde{\mathbf{Z}}^{(\ell)} = \gamma \cdot \left(\frac{\overline{\mathbf{Z}}^{(\ell)} - \overline{\mathbf{z}}^{(\ell)}}{\overline{\mathbf{z}}_\sigma^{(\ell)}}\right) + \beta$$

Two new parameters

$$\mathbf{H}^{(\ell)} = g(\tilde{\mathbf{Z}}^{(\ell)})$$

Effects of Batch Norm

- Larger learning rates can be used since weight scale is factored out by BN layers, i.e., BN(Z) = BN(aZ)

- Batch norm has a normalizing effect (noise is added since each example will cooccur with different examples in a single batch, which will modulate the representations)

- At test time, it is necessary to use an estimate of the batch mean and variances. Often, an exponentially weighted moving average suffices.

- Alternatively, I encourage you to go read the “Layer Normalization Paper” which I don’t have time to cover. :)

Thanks!

Terima kasih!