

# MSBA 434: Advanced Workshop on Machine Learning

## Lecture 1: Deep Neural Networks

# Danylo Vashchilenko

- Machine Learning Engineer at UJET
- Over the last 10 years:
  - Backend Software Engineer
  - Data Warehouse Engineer
  - Data Analyst Contractor
  - Web App Developer
  - Game Developer
- Stockholm School of Economics (Riga)
- LinkedIn: [linkedin.com/in/helldanylo](https://linkedin.com/in/helldanylo)
- Email: [hello+ucla@danylo.me](mailto:hello+ucla@danylo.me)

# Course Overview

Date	Topic	Assignment Due
10/7	Deep Neural Networks	EOD 10/13
10/14	Convolutional Neural Networks	EOD 10/20
10/21	Recurrent Neural Networks	EOD 10/27
10/28	Generative Adversarial Networks	EOD 11/3
11/4	Ensemble Methods	EOD 11/10

# Today's Agenda

- Review of NN Theory and Practice
  - Model Training
  - Network Architecture
  - Optimization Algorithms
  - Regularization Methods
  - Hyperparameter Optimization
- Autoencoders
  - Architectures: undercomplete, sparse
  - Training modes: reconstruction, denoising
- Case Study: Denoising Images

# Part 1

Review of NN Theory and Practice

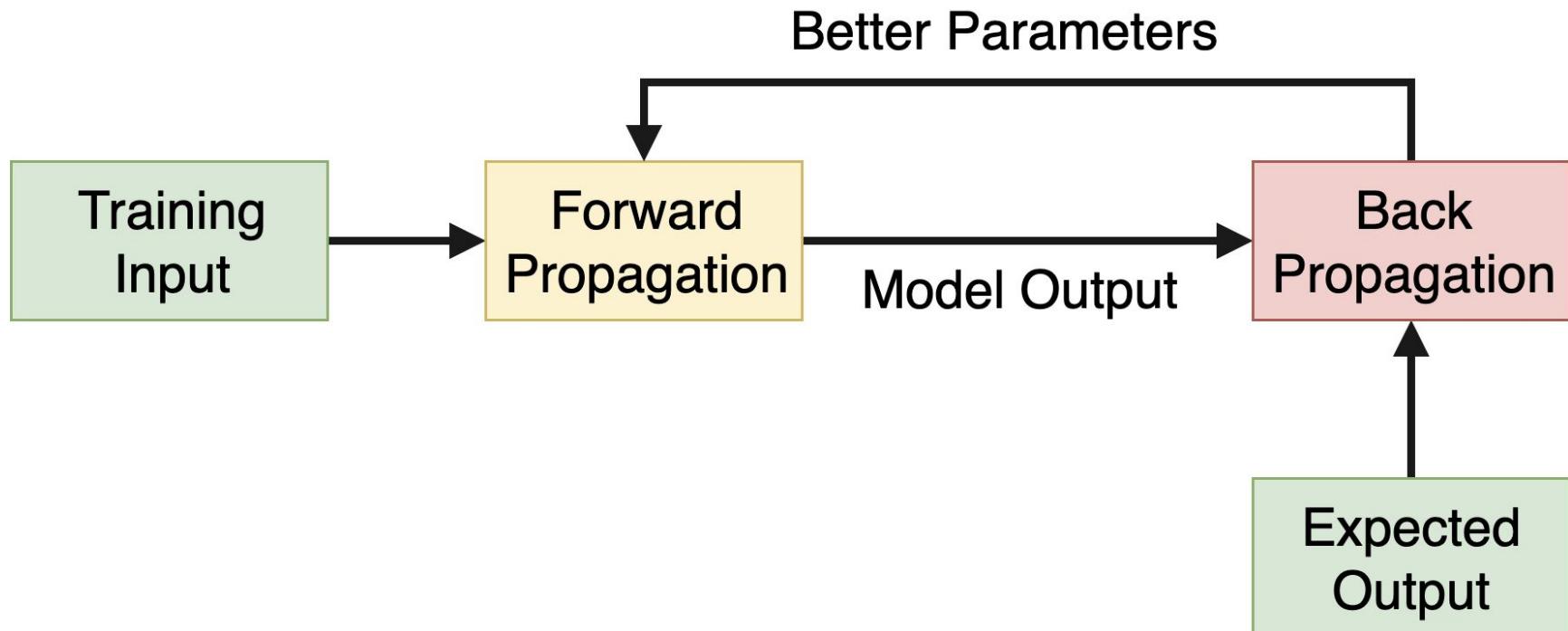
# Deep Learning Applications

- Photo/Video Analysis and Generation
  - Object detection
  - Style transfer
- Voice <-> Text Conversion
  - Voice assistants
- Natural Language Processing
  - Sentiment analysis
  - Text completion
- Games (Reinforcement Learning)
  - Chess
  - Go
  - Pacman

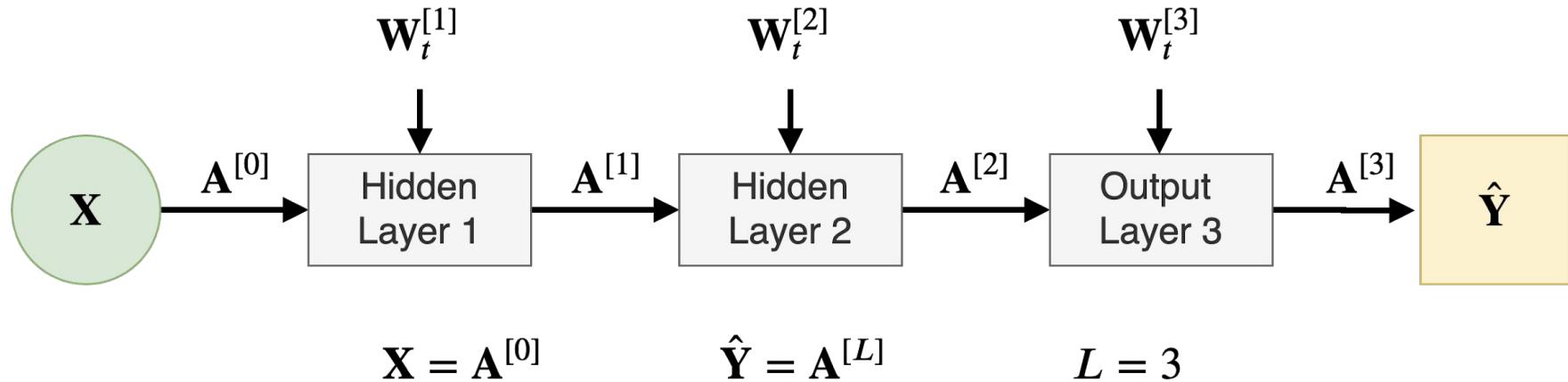
# Managerial Economics of Deep Learning

Project Goal	Project Duration	Compute Resource	Required Training
Design new architecture and train from scratch	Months	Expensive	Machine Learning Researcher
Use pre-existing architecture and/or transfer learning	Weeks	Cheaper	Machine Learning Engineer
Find a pre-trained model and/or buy Inference as a Service API	Days	Almost Free	Automation Software Engineer

# Training Iteration



# Feedforward Architecture



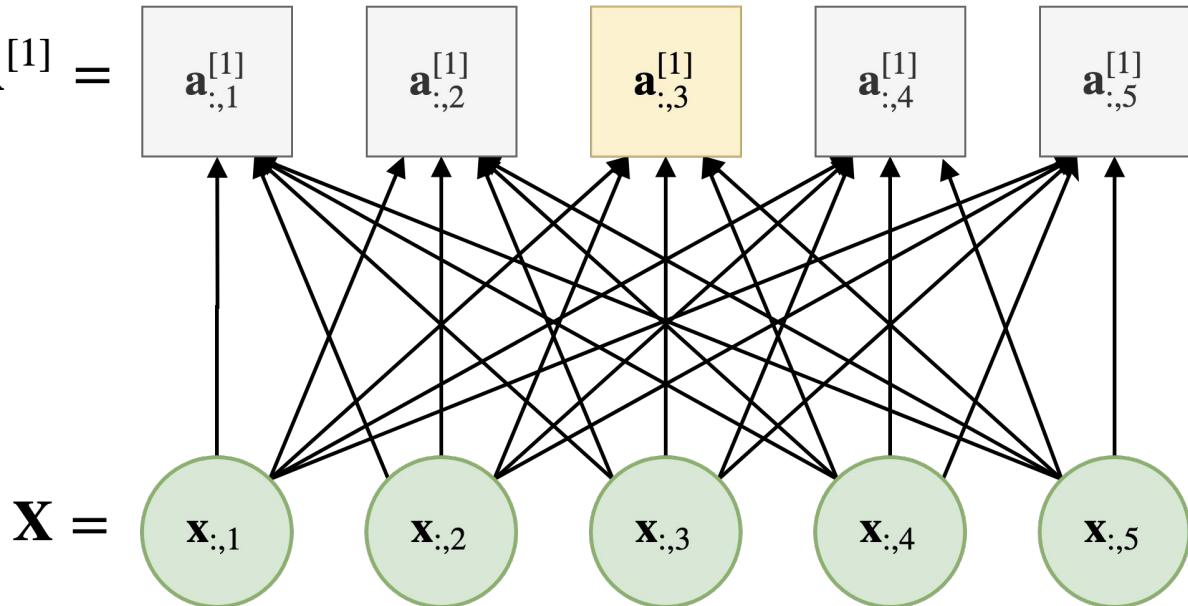
Forward Propagation in a 3-Layer **Feed-forward** Network

# FC: Single Neuron Activation

$$\mathbf{z}_{:,3}^{[1]} = \mathbf{X} \cdot \mathbf{w}_{3,:}^{[1]T} + b_3^{[1]}$$

$$\mathbf{a}_{:,3}^{[1]} = f(\mathbf{z}_{:,3}^{[1]})$$

$$\mathbf{A}^{[1]} =$$



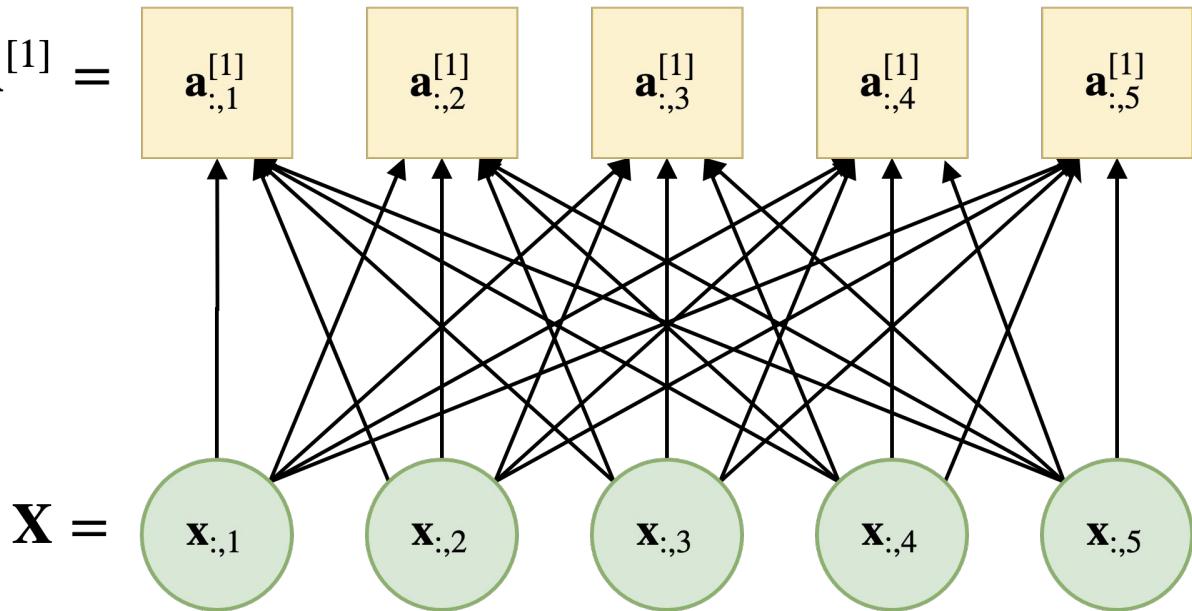
$$\mathbf{X} =$$

# FC: Layer Activation

$$\mathbf{Z}^{[1]} = \mathbf{X} \cdot \mathbf{W}^{[1]T} + \mathbf{b}^{[1]}$$

$$\mathbf{A}^{[1]} = f(\mathbf{Z}^{[1]})$$

$$\mathbf{X} =$$



# Fully Connected Layer

For input and activation matrices, **row vectors are individual examples** in the dataset.

For weight matrix, **row vectors are individual neurons** in the layer.

Activation function is applied **element-wise**.

$$\mathbf{Z}^{[l]} = \mathbf{A}^{[l-1]} \cdot \mathbf{W}^{[l]T} + \mathbf{b}^{[l]}$$

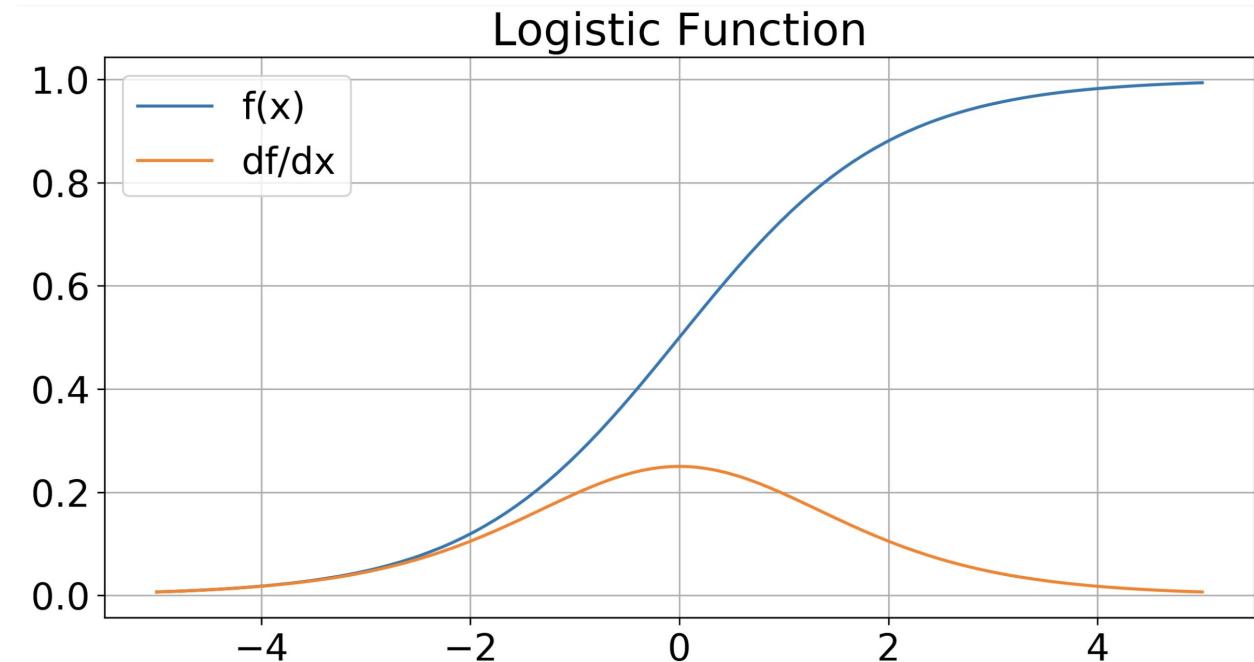
$$\mathbf{A}^{[l]} = f(\mathbf{Z}^{[l]})$$

See **FullyConnectedLayer.ipynb** for notation and linear algebra details.

# Logistic / Sigmoid

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

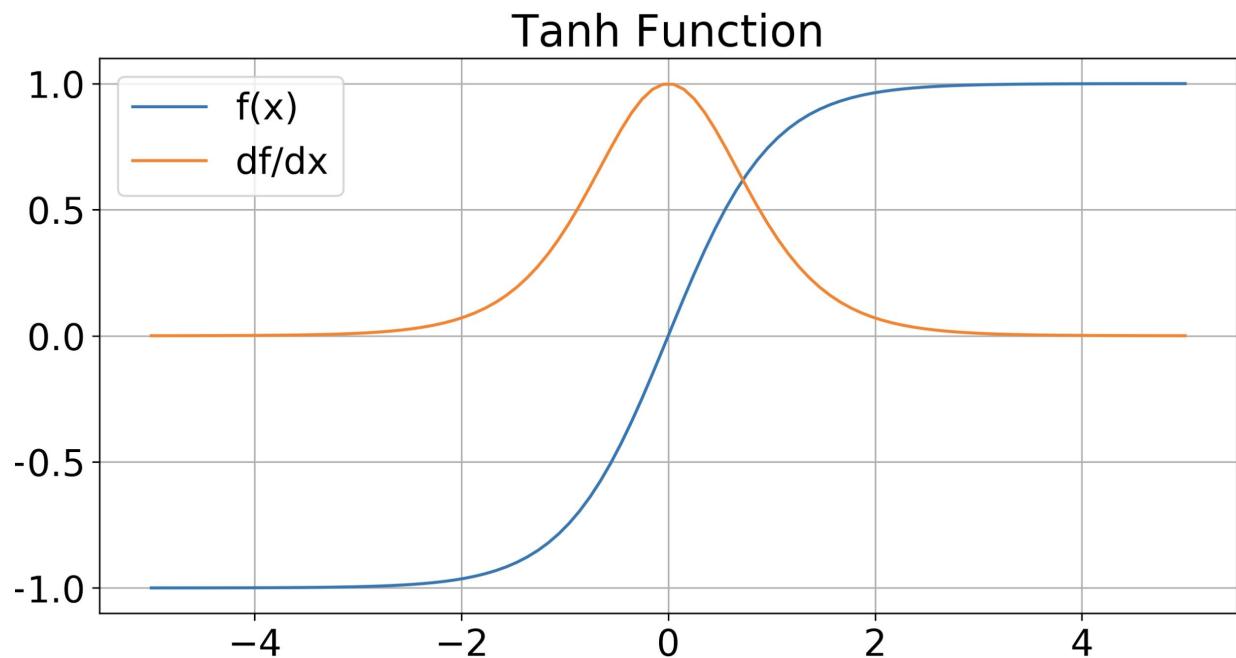
$$\frac{df}{dx} = \frac{e^x}{(e^x + 1)^2}$$



# Hyperbolic Tangent

$$f(x) = 2L(2x) - 1$$

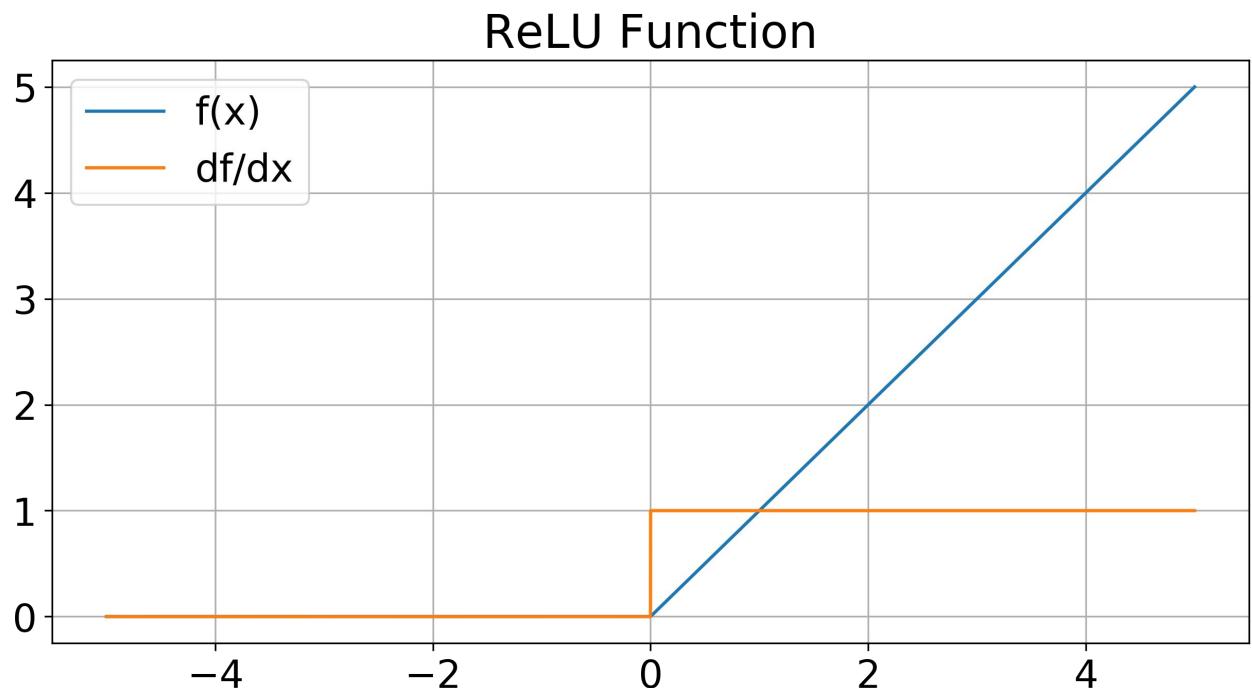
$$\frac{df}{dx} = 4L'(2x)$$



# Rectified Linear Unit

$$f(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$$

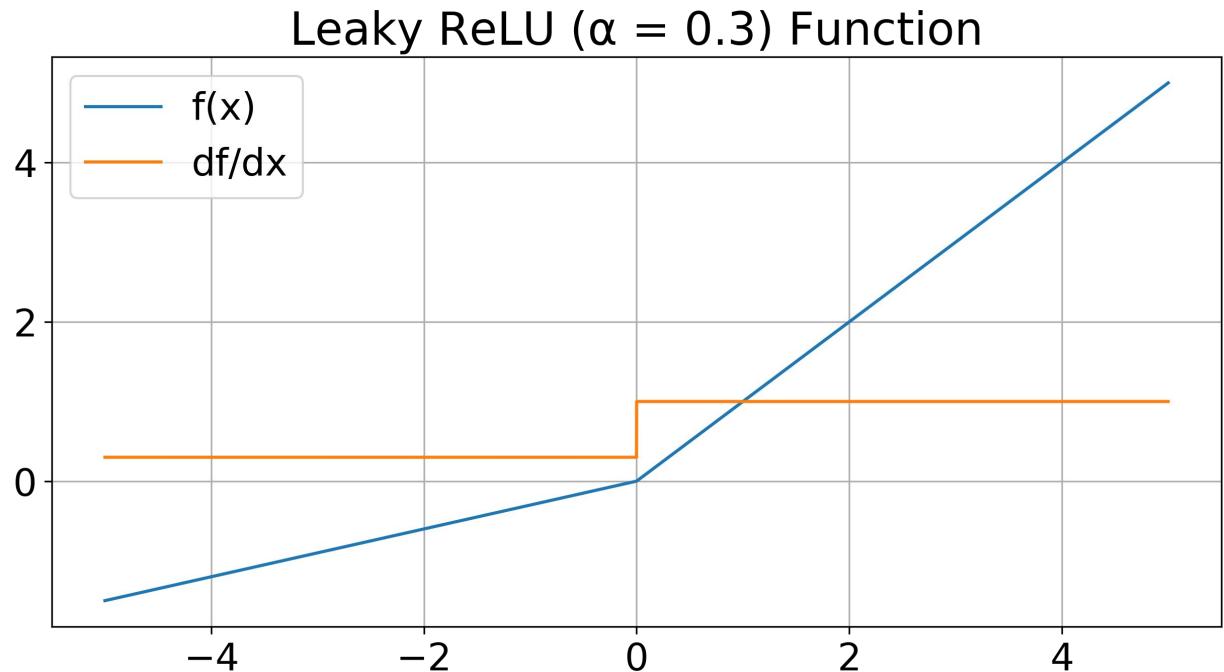
$$\frac{df}{dx} = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x < 0 \\ \text{undefined} & \text{for } x = 0 \end{cases}$$



# Leaky ReLU

$$f(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha x & \text{for } x < 0 \end{cases}$$

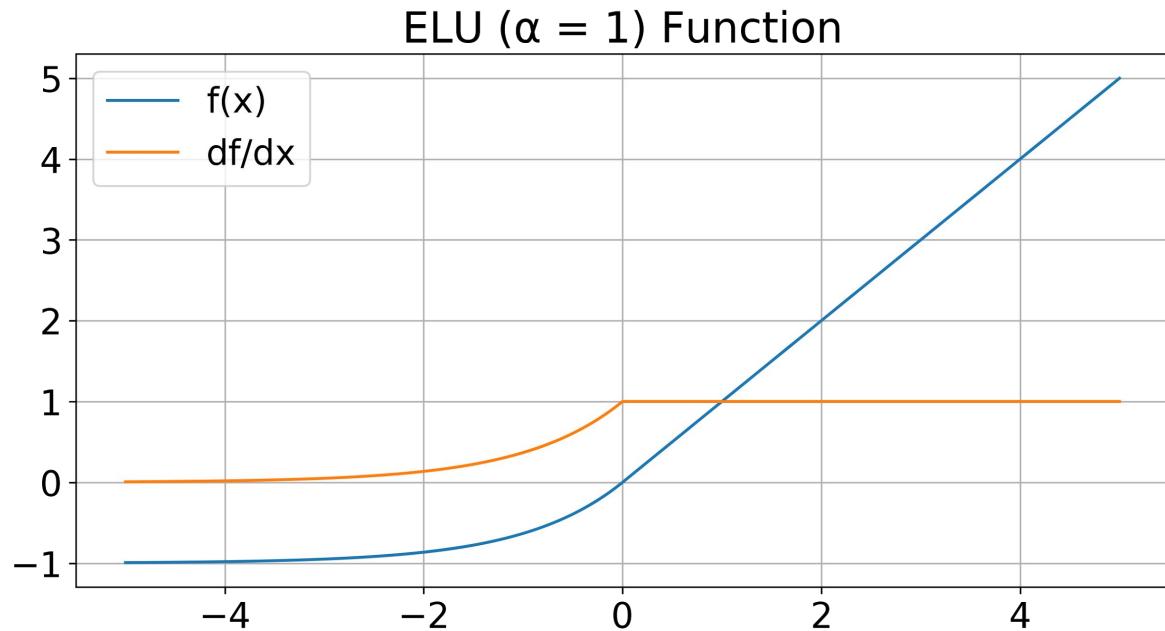
$$\frac{df}{dx} = \begin{cases} 1 & \text{for } x > 0 \\ \alpha & \text{for } x < 0 \\ \text{undefined} & \text{for } x = 0 \end{cases}$$



# Exponential Linear Unit

$$f(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$$

$$\frac{df}{dx} = \begin{cases} 1 & \text{for } x > 0 \\ \alpha e^x & \text{for } x < 0 \\ \text{undefined} & \text{for } x = 0 \end{cases}$$



# Activation Functions

Function	Hidden Layer	Output Layer
Logistic	Mostly RNNs	Yes
Tanh	Mostly RNNs	Yes
ReLU	Yes	Yes
Leaky ReLU	Yes	No, why?
ELU	Yes	No, why?

Choice of activation functions in Hidden Layers is a hyper-parameter.

See [ActivationFunctions.ipynb](#) notebook for vectorized NumPy implementations.

# Softmax Layer

**Problem:** multi-class classifier can't use independent logistic activation in output layer, because their sum won't add up to 1.

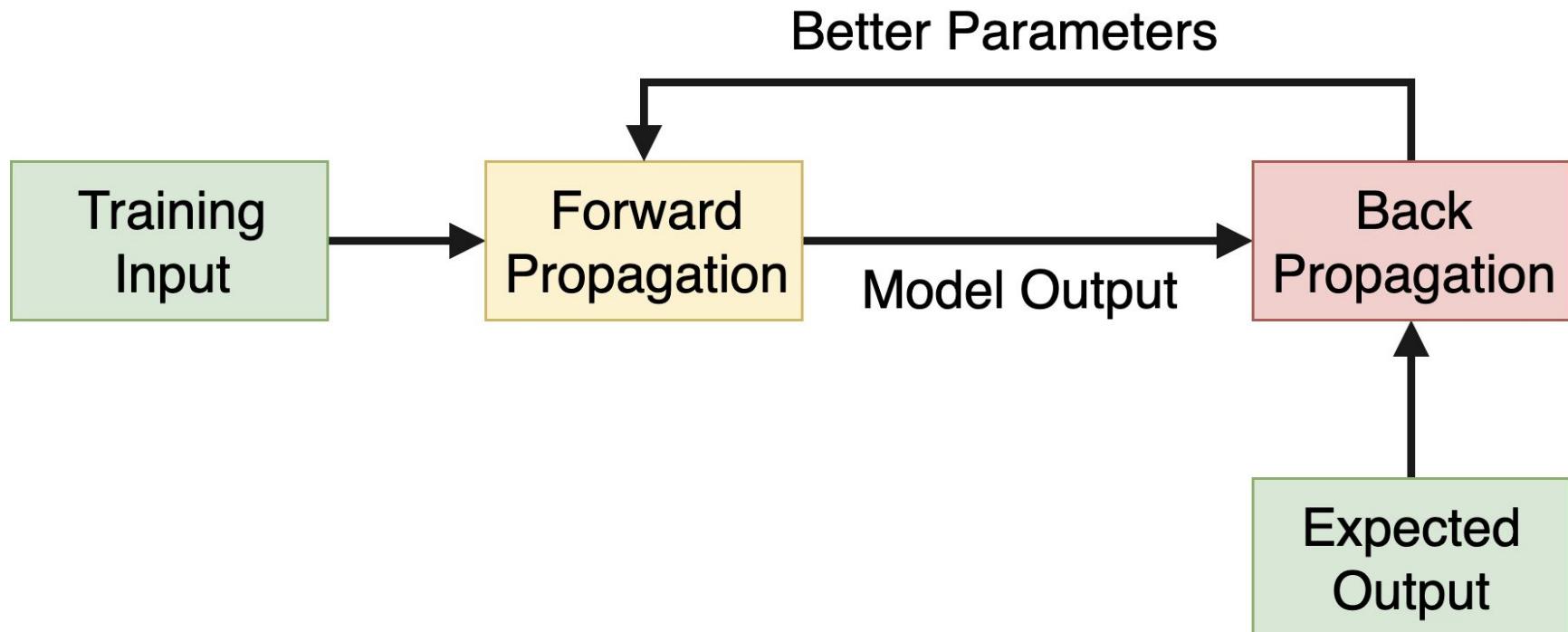
**Solution:**

$$\mathbf{z}^{[l]} = \exp(\mathbf{a}^{[l-1]})$$

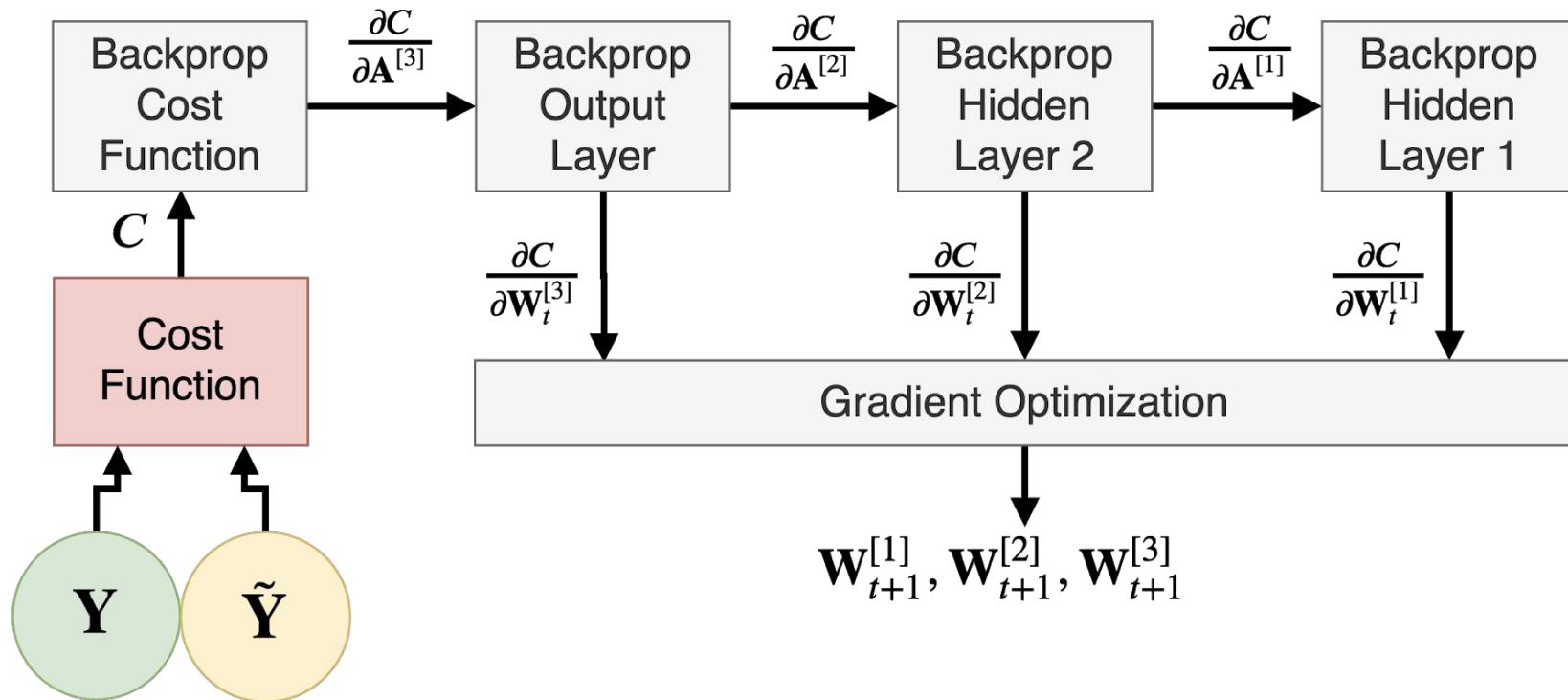
$$\mathbf{a}_{:,m}^{[l]} = \frac{\mathbf{z}_{:,m}^{[l]}}{\sum_{k=1}^M \mathbf{z}_{:,k}^{[l]}}$$

	$\mathbf{a}^{[l-1]}$	$\mathbf{z}^{[l]}$	$\mathbf{a}^{[l]}$
$m$			
0	-2	0.14	0.01
1	-1	0.37	0.03
2	0	1.00	0.09
3	1	2.72	0.23
4	2	7.39	0.64

# Training Iteration



# Back Propagation in Feedforward Architecture



# Regression Loss

Regression loss is used when the output is **real-valued**.

Note that MAE is not differentiable at zero.

Total cost is just an **average loss across samples and features**.

Cost Function  $\in (\mathbb{R}^{S \times M}, \mathbb{R}^{S \times M}) \rightarrow \mathbb{R}$

$$\text{Mean Absolute Error} = \frac{1}{SM} \sum_s^S \sum_m^M |y_{s,m} - \hat{y}_{s,m}|$$

$$\text{Mean Squared Error} = \frac{1}{SM} \sum_s^S \sum_m^M (y_{s,m} - \hat{y}_{s,m})^2$$

# Classification Loss

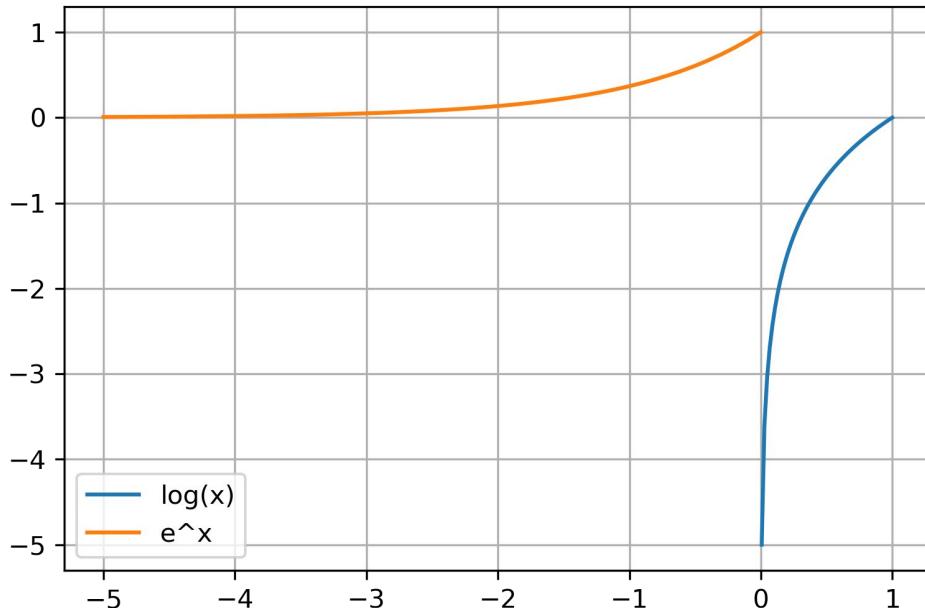
Classification loss is used when the **output is in  $[0, 1]$**  range.

Cross entropy measures the difference between two probability distributions in an **asymmetric way!**

Total cost is just an **average loss across samples and classes.**

Cost Function  $\in ([0, 1]^{S \times M}, [0, 1]^{S \times M}) \rightarrow \mathbb{R}$

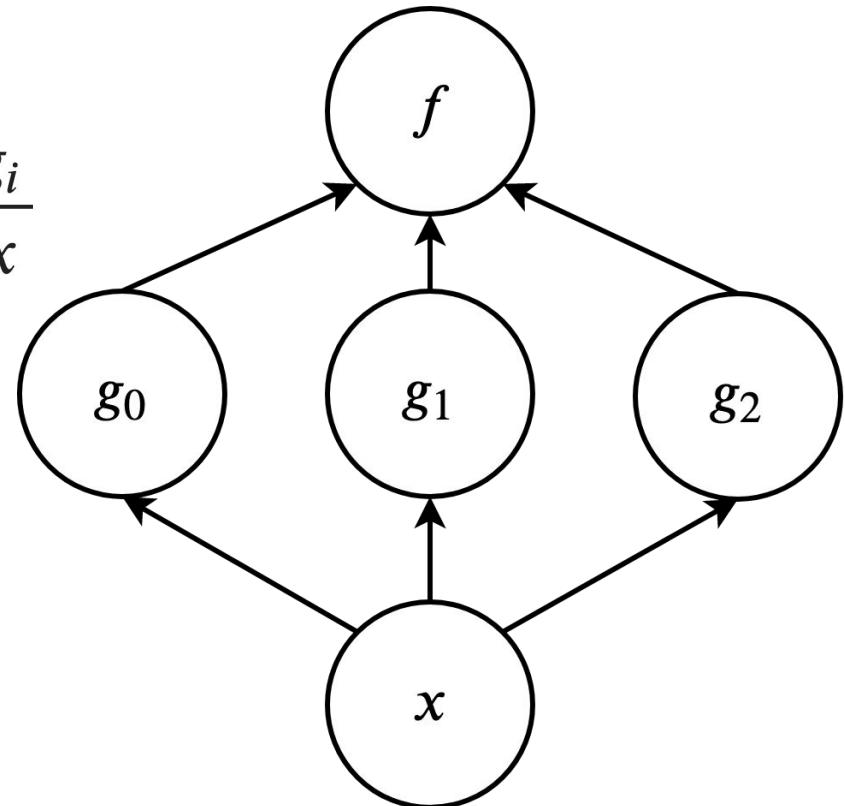
$$\text{Cross Entropy} = -\frac{1}{SM} \sum_s^S \sum_m^M y_{s,m} \log(\hat{y}_{s,m})$$



# Multivariate Chain Rule

$$\frac{\partial f(g_0(x), \dots, g_n(x))}{\partial x} = \sum_i^n \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

Since a **neural network is a deep composition of simple differentiable functions**, we can use the chain rule to calculate the derivative of the cost function with respect to every parameter.

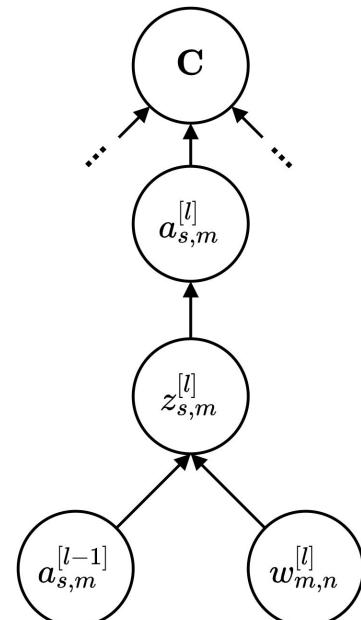


# Backpropagation in Fully Connected Layer

$$\frac{\partial a_{s,m}^{[l]}}{\partial z_{s,m}^{[l]}} = f'(z_{s,m}^{[l]}) \quad \frac{\partial z_{s,m}^{[l]}}{\partial w_{m,n}^{[l]}} = a_{s,n}^{[l-1]}$$

$$\frac{\partial C}{\partial w_{m,n}^{[l]}} = \sum_{s=1}^S \left[ \frac{\partial C_s}{\partial a_{s,m}^{[l]}} * \frac{\partial a_{s,m}^{[l]}}{\partial z_{s,m}^{[l]}} * \frac{\partial z_{s,m}^{[l]}}{\partial w_{m,n}^{[l]}} \right]$$

$$\frac{\partial C}{\partial w_{m,n}^{[l]}} = \sum_{s=1}^S \left[ \frac{\partial C_s}{\partial a_{s,m}^{[l]}} * f'(z_{s,m}^{[l]}) * a_{s,n}^{[l-1]} \right]$$

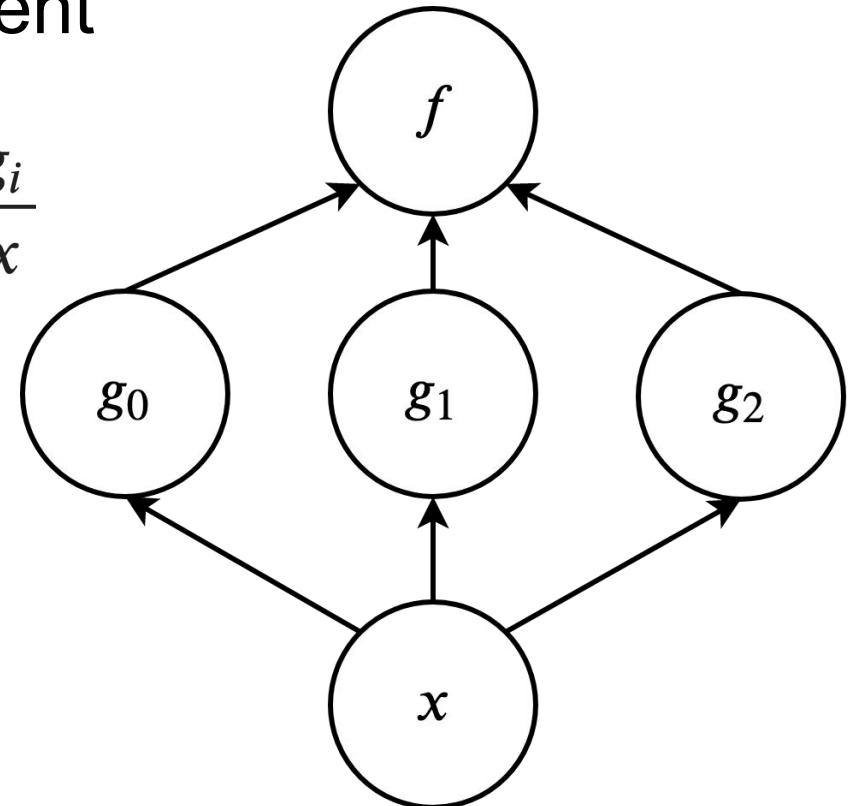


# Exploding / Vanishing Gradient

$$\frac{\partial f(g_0(x), \dots, g_n(x))}{\partial x} = \sum_i^n \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

**Problem:** Repeated multiplication makes gradients at the lower layers very large or very small.

**Solution:** Gradient clipping helps with the exploding gradient. Batch normalization helps with the vanishing gradient.



# Batch Normalization

## Problem

Make distribution of inputs to each layer have mean = 0 and standard deviation = 1.

## Solution

1. During training, **standardize hidden pre-activations** within a batch.
2. Add **trainable parameters** to undo standardization via optimization.
3. During inference, use the **training dataset statistics** and learned parameters.

$$\mathbf{u}^{[l]} = \frac{1}{S} \sum_{s=1}^S \mathbf{z}_{s,:}^{[l]}$$

$$\mathbf{v}^{[l]} = \frac{1}{S} \sum_{s=1}^S (\mathbf{z}_{s,:}^{[l]} - \mathbf{u}^{[l]})^2$$

$$\mathbf{z}^{[l]} \leftarrow \frac{\mathbf{z}^{[l]} - \mathbf{u}}{\sqrt{\mathbf{v}^{[l]} + \epsilon}}$$

$$\mathbf{z}^{[l]} \leftarrow \mathbf{z}^{[l]} \odot \tilde{\mathbf{v}}^{[l]} + \tilde{\mathbf{u}}$$

Ioffe S., Szegedy C., (2015)

# Batch Normalization of ... ?

If the activation function is not zero-centered, prefer:

1. Dot product + bias
2. Activation function
3. Batch normalization

If the activation function is zero-centered, also try:

1. Dot product + bias
2. Batch normalization
3. Activation function

See [BatchNormalization.ipynb](#) for discussion, NumPy, and Keras implementations.

# Parameter Initialization

## Problem

Initializing every weight identically will prevent any learning due to gradient symmetry.

$\forall g, h :$

$$\mathbf{w}_{g,:}^{[l]} = \mathbf{w}_{h,:}^{[l]}$$

$$b_g^{[l]} = b_h^{[l]}$$

$$z_{s,g}^{[l]} = \mathbf{a}_{s,:}^{[l-1]} \cdot \mathbf{w}_{g,:}^{[l]T} + b_g^{[l]}$$

$$z_{s,h}^{[l]} = \mathbf{a}_{s,:}^{[l-1]} \cdot \mathbf{w}_{h,:}^{[l]T} + b_h^{[l]}$$

$$z_{s,g}^{[l]} = z_{s,h}^{[l]}$$

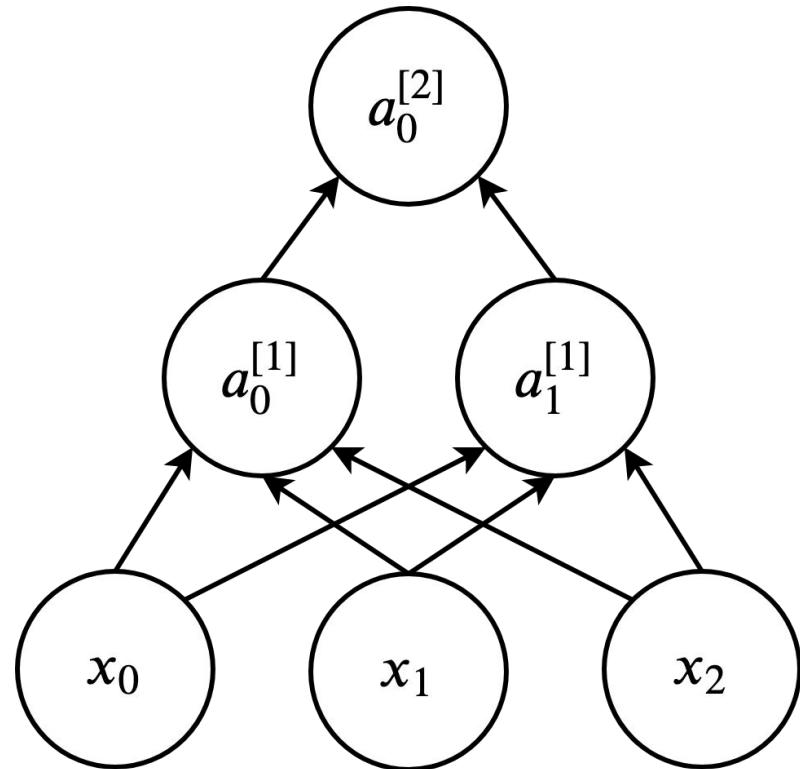
$$a_{s,g}^{[l]} = f(z_{s,g}^{[l]}) = f(z_{s,h}^{[l]}) = a_{s,h}^{[l]}$$

# Parameter Initialization

$$\frac{\partial C_s}{\partial w_{g,n}^{[l]}} = \frac{\partial C_s}{\partial a_{s,g}^{[l]}} * f'(z_{s,g}^{[l]}) * a_{s,n}^{[l-1]}$$

$$\frac{\partial C_s}{\partial w_{h,n}^{[l]}} = \frac{\partial C_s}{\partial a_{s,h}^{[l]}} * f'(z_{s,h}^{[l]}) * a_{s,n}^{[l-1]}$$

$$\frac{\partial C_s}{\partial w_{g,n}^{[l]}} = \frac{\partial C_s}{\partial w_{h,n}^{[l]}}$$



# Parameter Initialization

- Xavier Glorot, and Yoshua Bengio (2010)
  - Makes variance of activation and gradient between layers similar
  - Derivation constraints the variance of the distribution, not its shape

$$W_i \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

$n_{in}(n_{out})$  = count of input (output) connections

See [WeightInitialization.ipynb](#) for discussion, and Keras implementations.

# Parameter Initialization

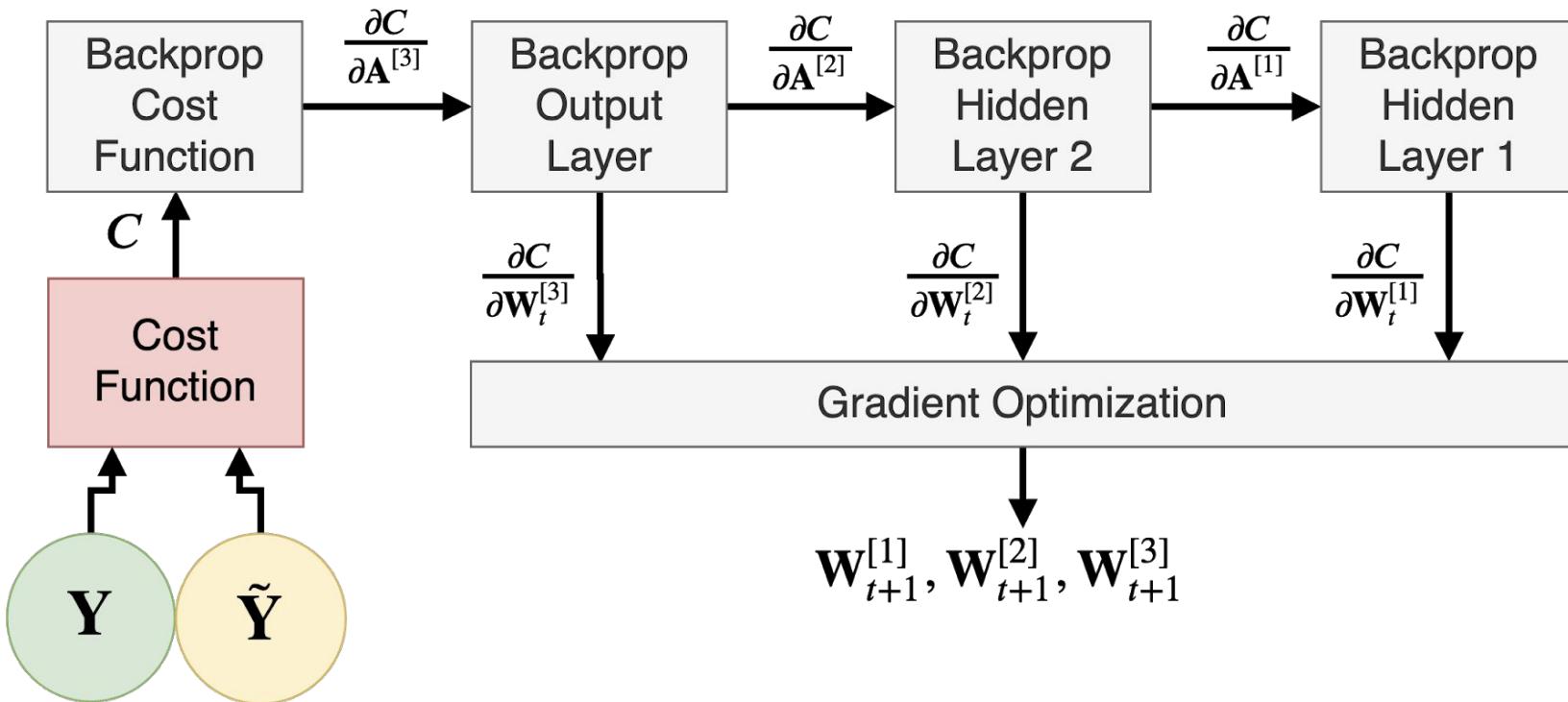
- He, K., Zhang, X., Ren, S., and Sun, J., (2015)
  - Makes variance of activation and gradient between layers similar
  - Derivation constraints the variance of the distribution, not its shape

$$W_i \sim N\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

$n_{in}$  = count of input connections

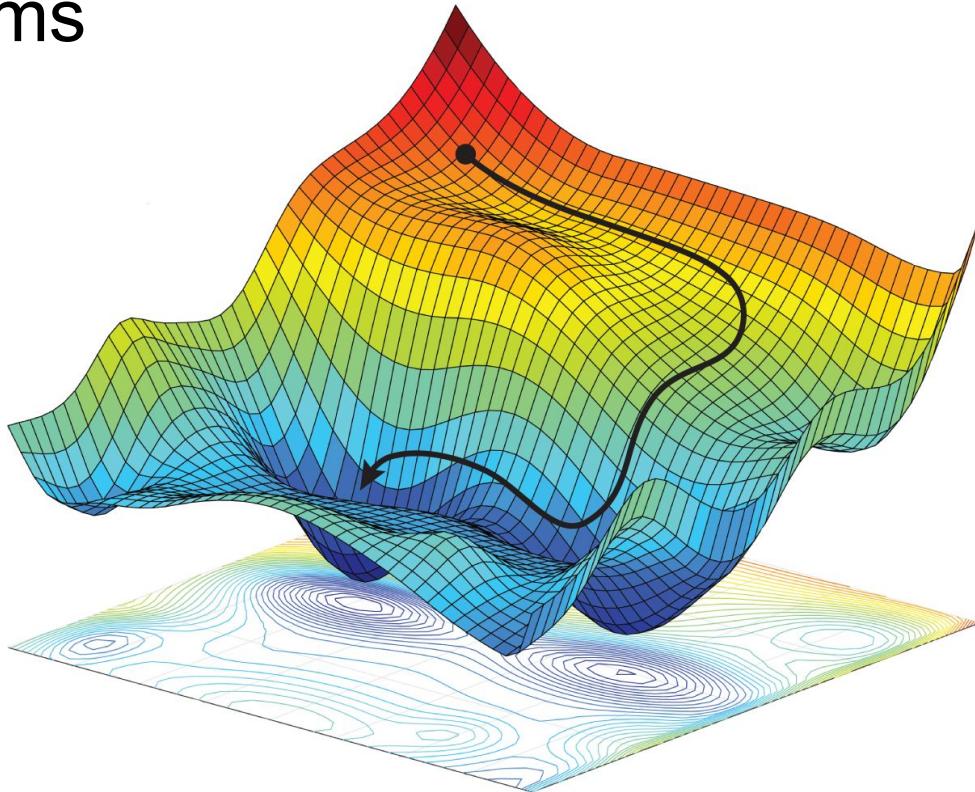
See [WeightInitialization.ipynb](#) for discussion, and Keras implementations.

# Back Propagation in Feedforward Architecture



# Optimization Algorithms

- Gradient Descent
  - Batch
  - Stochastic
  - Mini-batch
- Gradient Descent with Improvements:
  - Momentum
  - RMSprop
  - Adam



Source: Stochastic Gradient/Mirror Descent: Minimax Optimality and Implicit Regularization  
(Azizan, N. & Hassibi, B., 2018)

# Gradient Descent

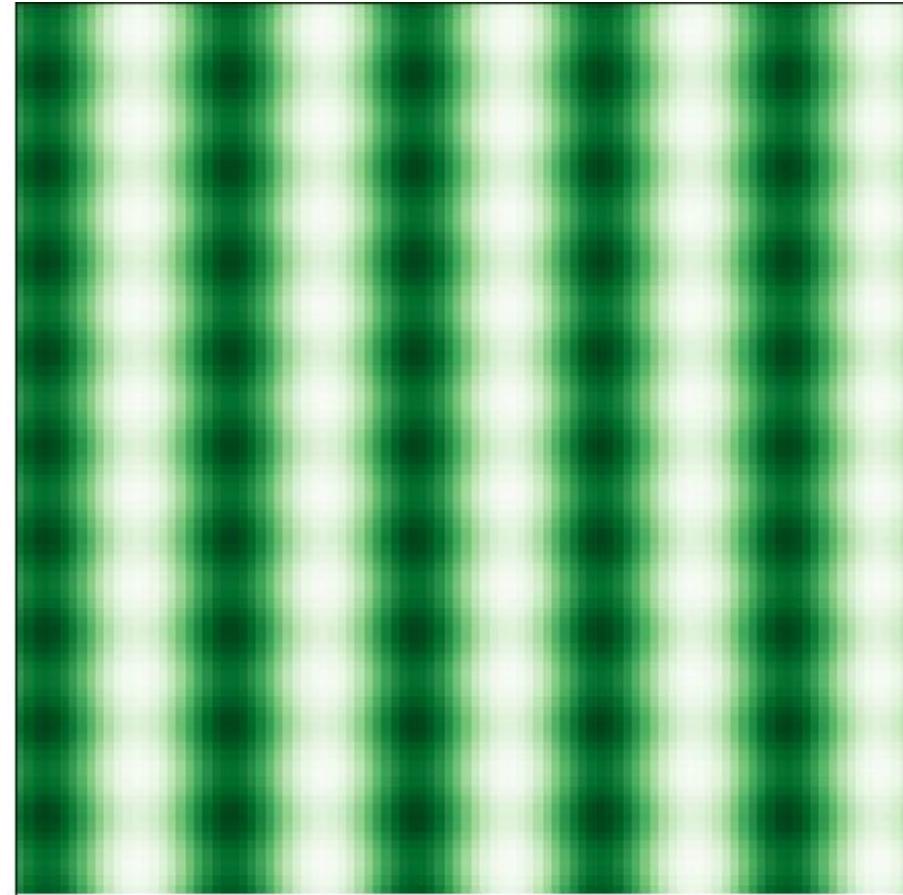
$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \nabla_{\mathbf{W}_t}$$

$$\nabla_{\mathbf{W}_t} = \frac{\partial C(\mathbf{W}_t)}{\partial \mathbf{W}_t}$$

Three flavors:

- Stochastic ( $n = 1$ )
- Batch ( $n = S$ )
- Mini-batch ( $1 < n < S$ )

See [GradientDescent.ipynb](#) for discussion,  
NumPy, Tensorflow, and Keras  
implementations.

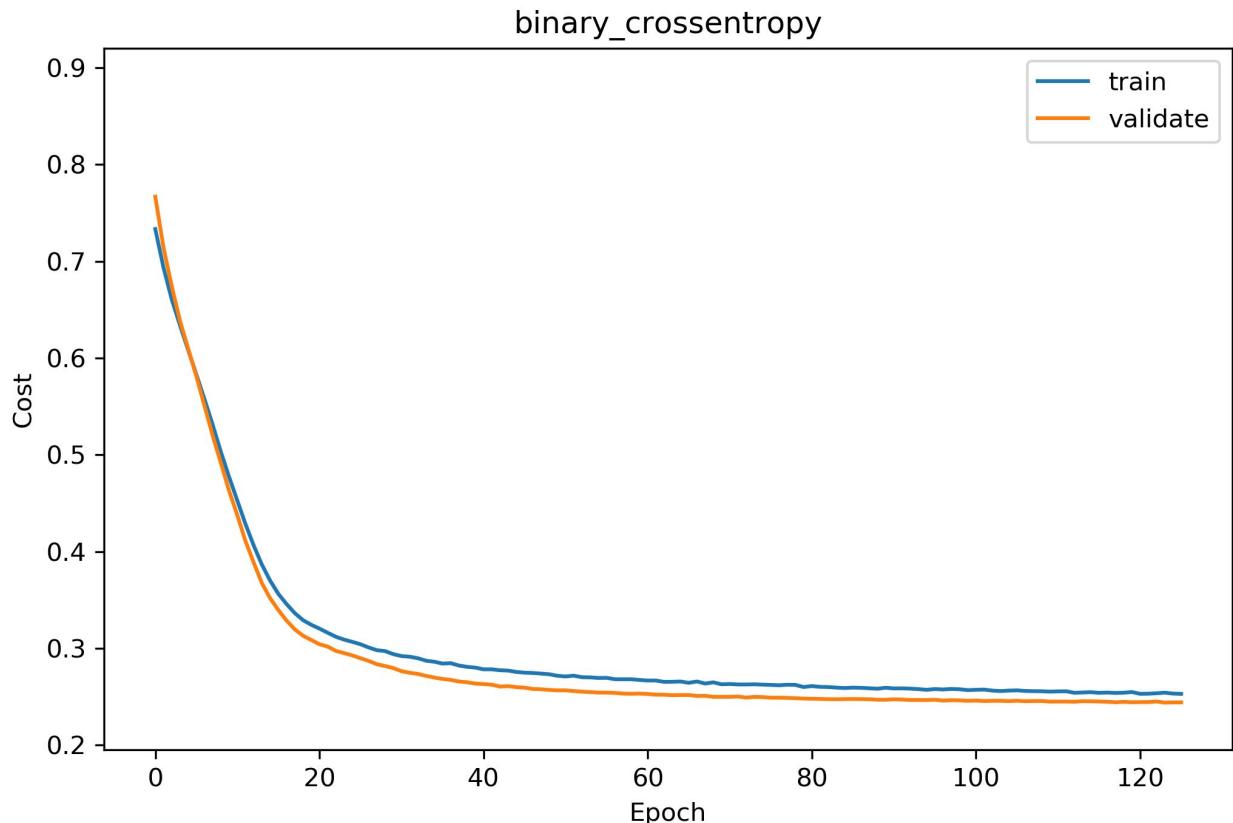


$$S(\mathbf{x}) = 3\sin(x_0) + \sin(x_1)^2$$

# Convergence of Learning Curve

Typically, you want to continue training until it the **training cost as a function of time converges**.

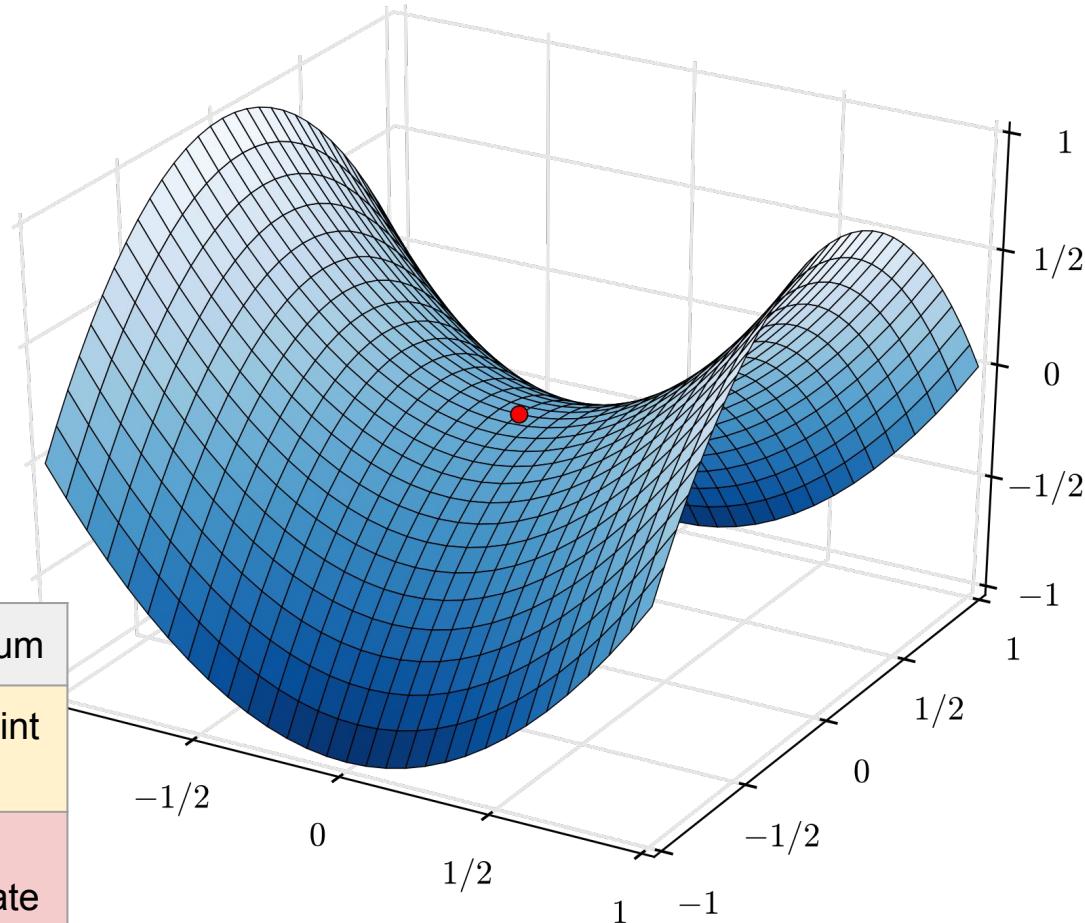
But if the learning curve converges, **did the algorithm find the function's global minimum?**



# Saddle Point

Gradient vector is zero  
but the point is not a  
local extremum.

Dimension	X: minimum	X: maximum
Y: minimum	Local minimum	Saddle point
Y: maximum	Saddle point	Too high learning rate

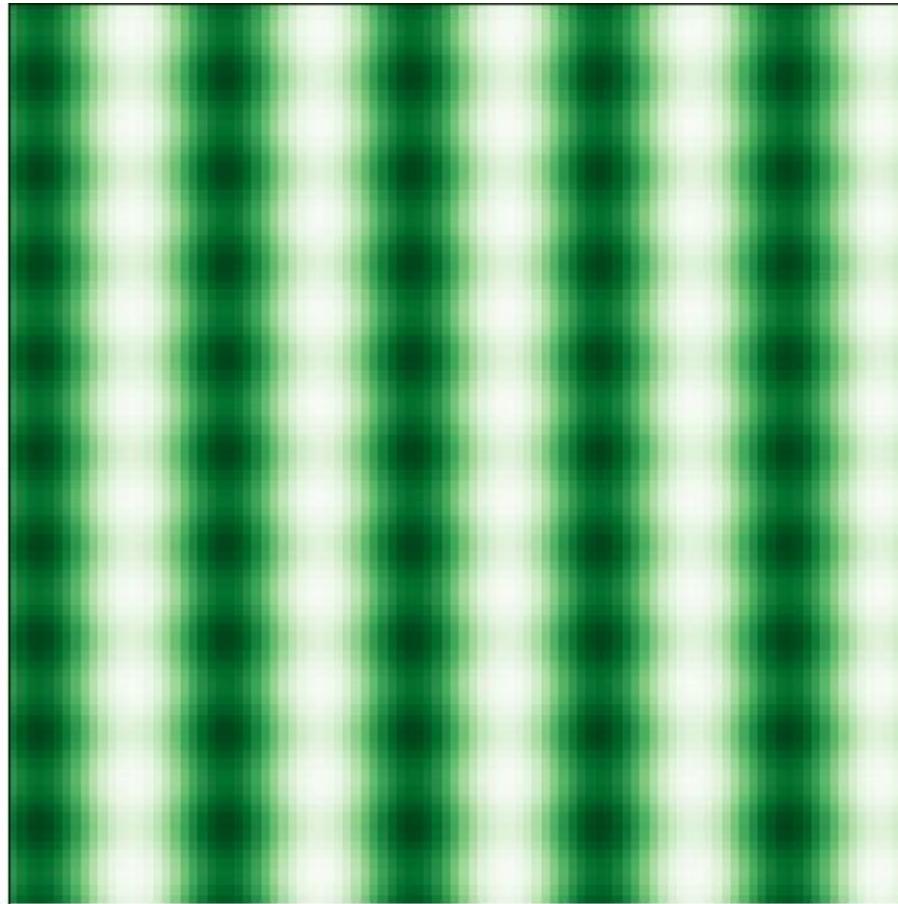


# Momentum

$$\mathbf{V}_{t+1} = \beta \mathbf{V}_t + (1 - \beta) \nabla_{\mathbf{W}_t}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \mathbf{V}_{t+1}$$

Constant  $\beta$  is typically = 0.9,  
but should be optimized as  
a hyperparameter.



Qian, G. (1999)

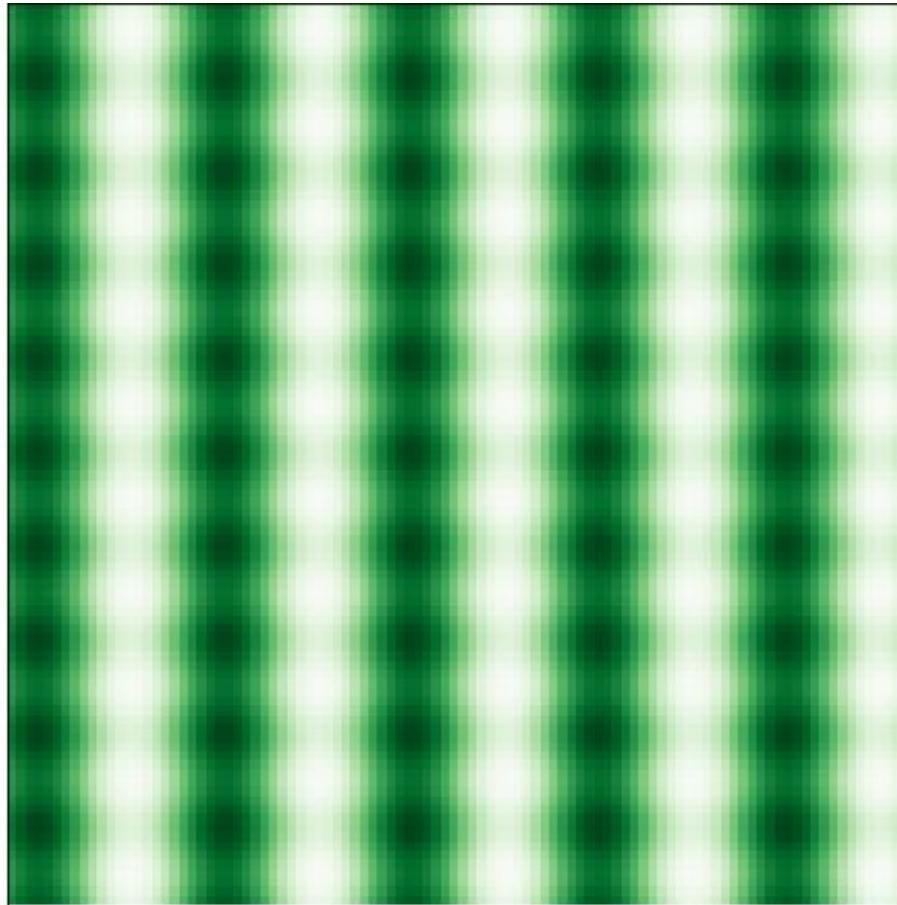
$$S(\mathbf{x}) = 3\sin(x_0) + \sin(x_1)^2$$

# RMSprop

$$\mathbf{S}_{t+1} = \gamma \mathbf{S}_t + (1 - \gamma) (\nabla_{\mathbf{W}_t})^2$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \frac{\nabla_{\mathbf{W}_t}}{\sqrt{\mathbf{S}_{t+1}}}$$

Constant  $\gamma$  is typically = 0.9,  
but should be optimized as a  
hyperparameter.



Unpublished work by Geoffrey Hinton.

$$S(\mathbf{x}) = 3\sin(x_0) + \sin(x_1)^2$$

# Adam

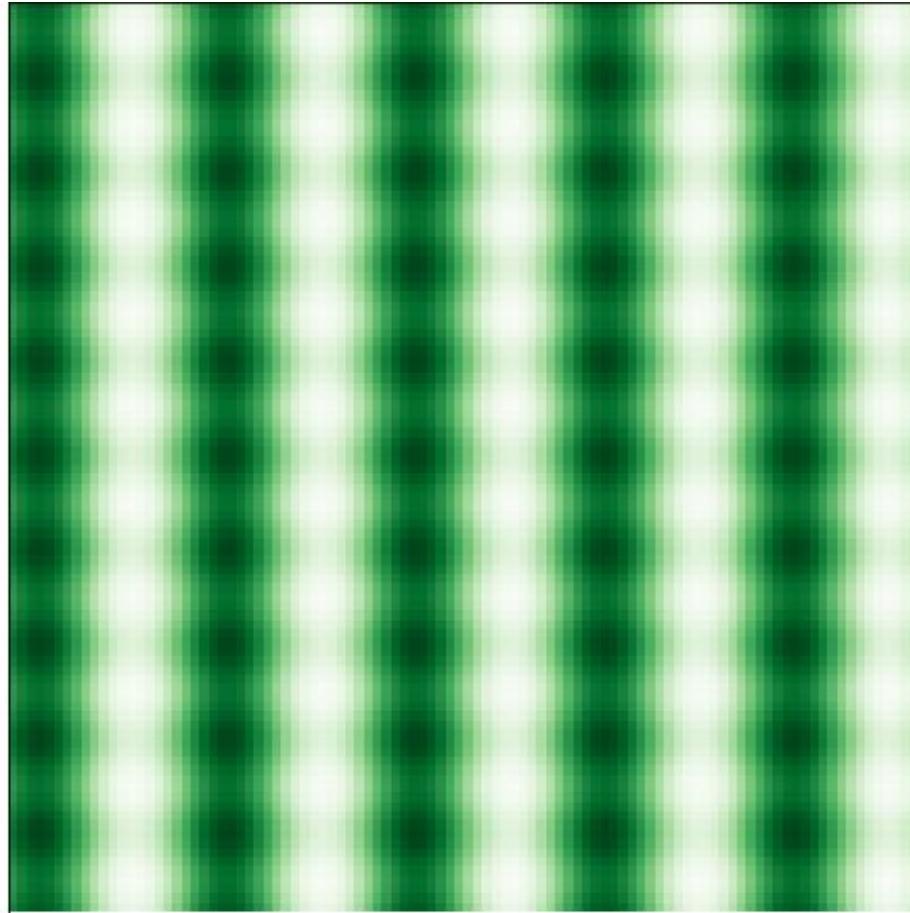
$$\mathbf{V}_{t+1} = \beta \mathbf{V}_t + (1 - \beta) \nabla_{\mathbf{W}_t}$$

$$\mathbf{S}_{t+1} = \gamma \mathbf{S}_t + (1 - \gamma) (\nabla_{\mathbf{W}_t})^2$$

$$\tilde{\mathbf{V}}_{t+1} = \frac{\mathbf{V}_{t+1}}{(1 - \beta^t)}$$

$$\tilde{\mathbf{S}}_{t+1} = \frac{\mathbf{S}_{t+1}}{(1 - \gamma^t)}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \frac{\tilde{\mathbf{V}}_{t+1}}{\sqrt{\tilde{\mathbf{S}}_{t+1}}}$$



# Learning is not just Memorization

- Verify any learning on data that was not used during parameter optimization.
- Any learning requires a prior on how to generalize from training instances to unseen instances.
- No Free Lunch Theorem

# Model Evaluation

Before you train anything, the dataset is split into:

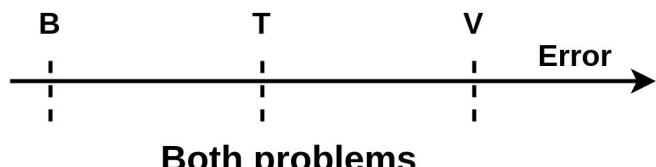
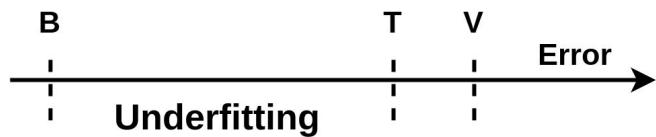
- Training segment (~80%)
- Validation segment (~10%)
- Testing segment (~10%)

Why do we need three segments?

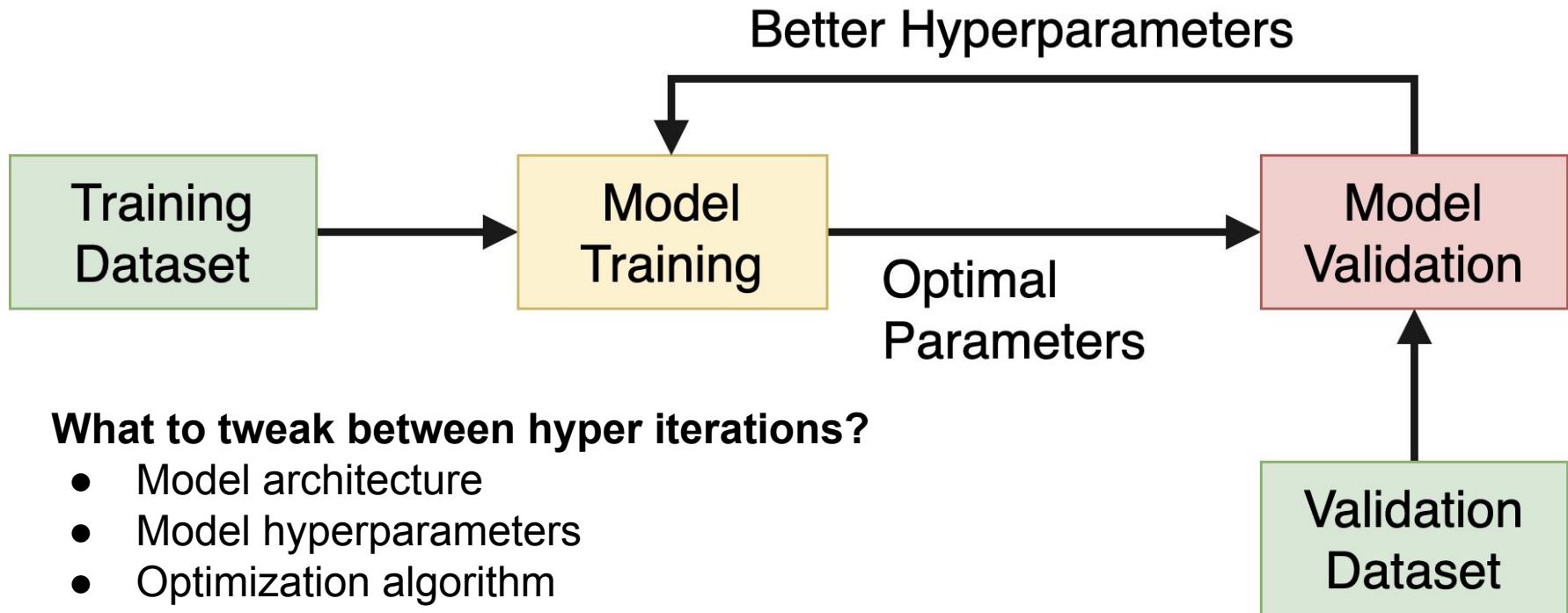
	Learn from...	Verify on...
Parameters	Training segment	Validation segment
Hyperparameters	Validation segment	Testing segment

# Generalization Inference

Scenario	Suggestions
Optimal	Write a research paper
Overfitting	<ul style="list-style-type: none"><li>• Collect more data</li><li>• Augment existing data</li><li>• Early stopping</li><li>• Decrease model capacity</li><li>• Increase regularization</li></ul>
Underfitting	<ul style="list-style-type: none"><li>• Change architecture</li><li>• Increase model capacity</li><li>• Reduce regularization</li></ul>
Both problems	All of the above (except for the paper)



# Hyper Training Iteration

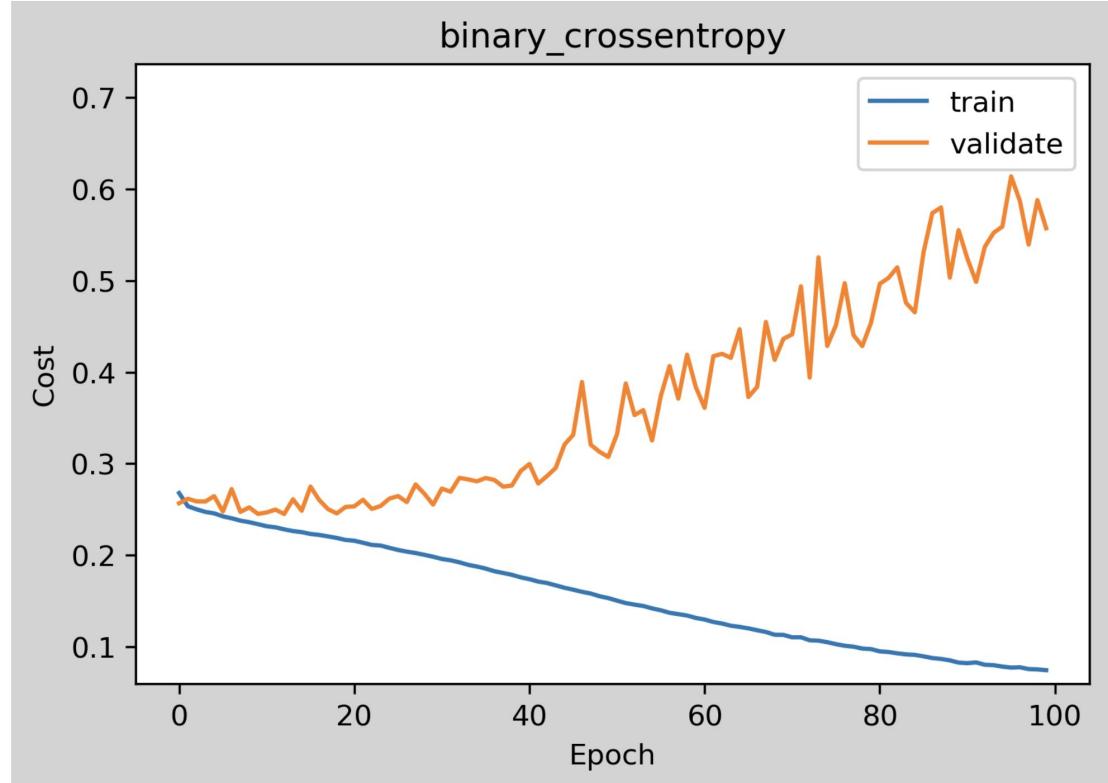


**What to tweak between hyper iterations?**

- Model architecture
- Model hyperparameters
- Optimization algorithm
- Optimization hyperparameters

# Maximum Overfit

1. Train a model that **overfits the training dataset** as much as possible.
2. Confirm the **training metrics** are satisfactory.
3. Introduce **regularization** to improve the validation metrics.



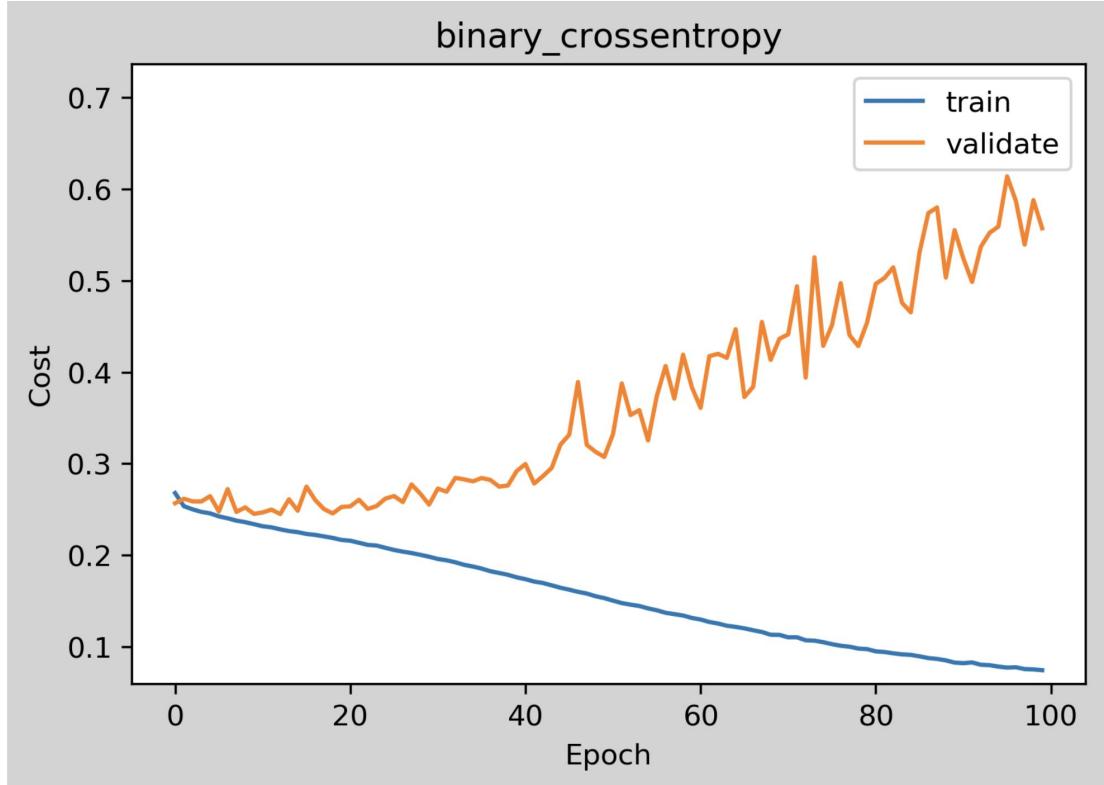
# Regularization

## Problem

Reduce overfitting via  
preference for simpler  
models.

## Solutions

- L1 Norm
- L2 Norm



# Regularization via Vector Norms

$$\text{Cost}(\mathbf{Y}, \hat{\mathbf{Y}}, \mathbf{W}) = \text{Supervised Cost}(\mathbf{Y}, \hat{\mathbf{Y}}) + \alpha * \text{Regularization Cost}(\mathbf{W})$$

$$L_1(\mathbf{W}) = \sum_i^n |w_i|$$

$$\frac{dL_1}{dw_i} = \begin{cases} 1 & \text{if } w_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

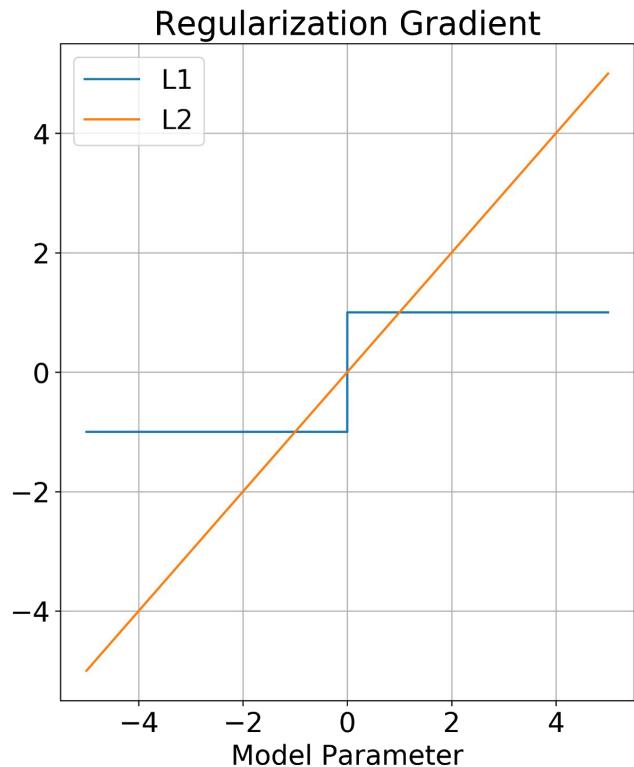
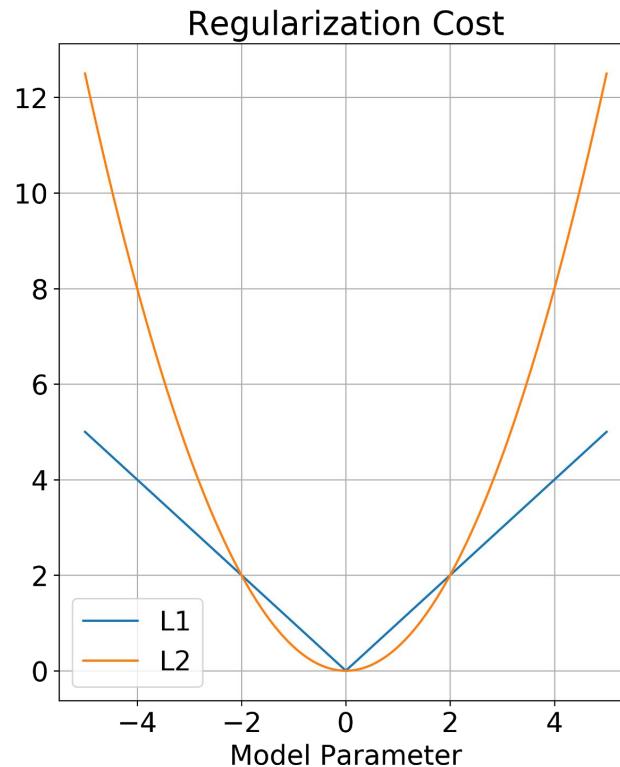
$$L_2(\mathbf{W}) = \frac{1}{2} \sum_i^n w_i^2$$

$$\frac{dL_2}{dw_i} = w_i$$

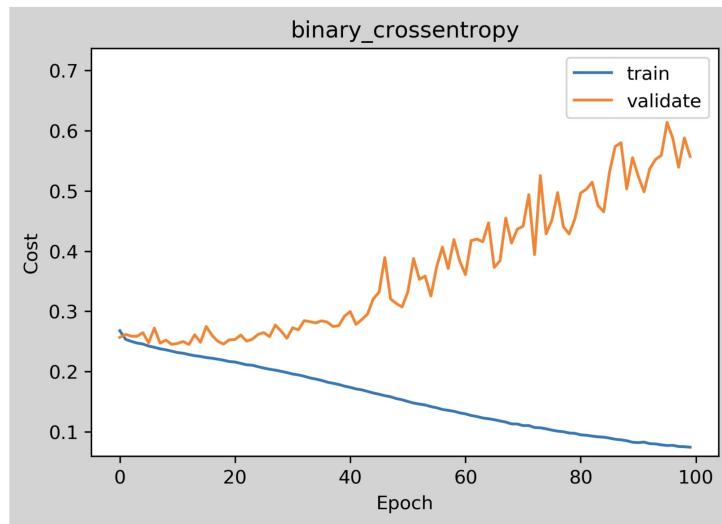
# Comparison of L1 and L2 Norms

L1 is more likely to prefer a sparse parameter matrix.

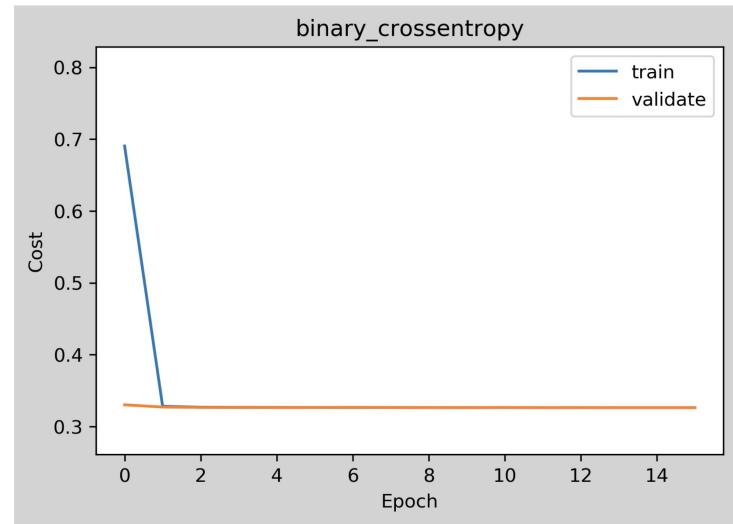
L2 is more likely to remove extremely large parameters.



# Effect of Regularization Cost

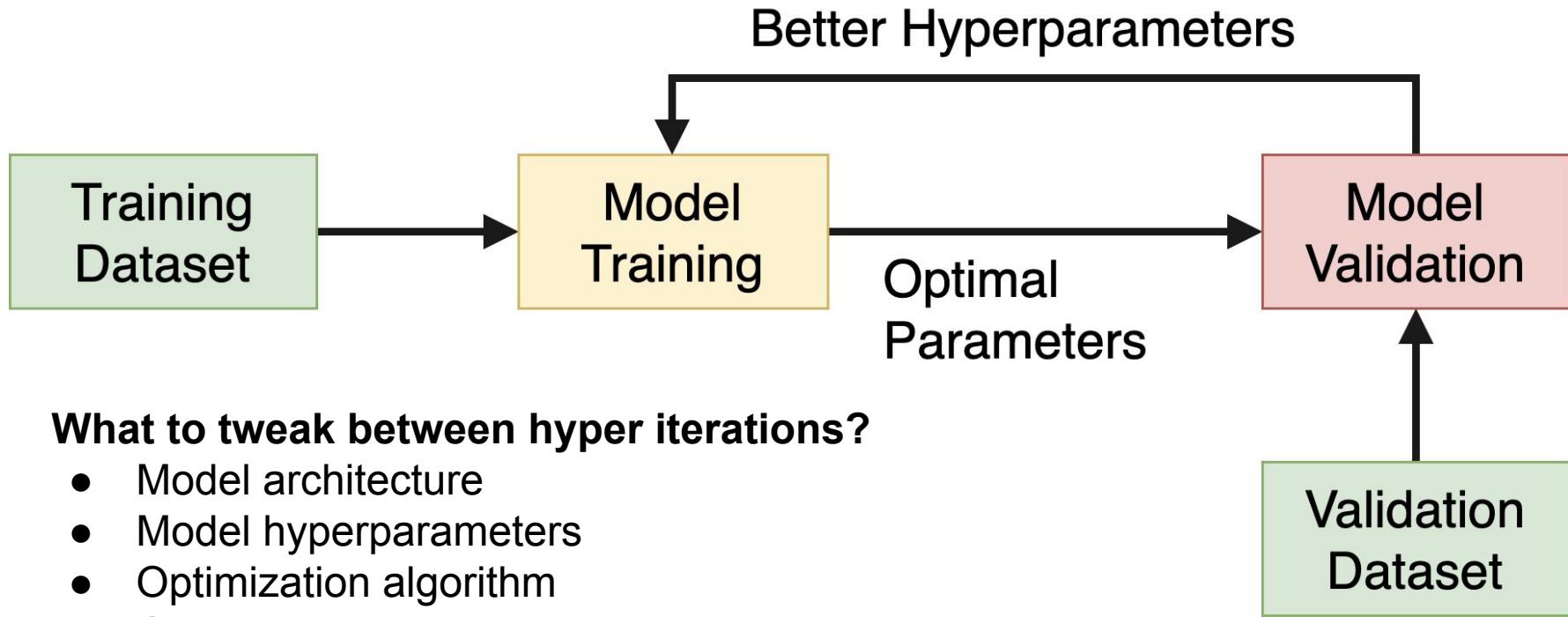


L2 Regularization Factor = 0.00



L2 Regularization Factor = 0.05

# Hyper Training Iteration

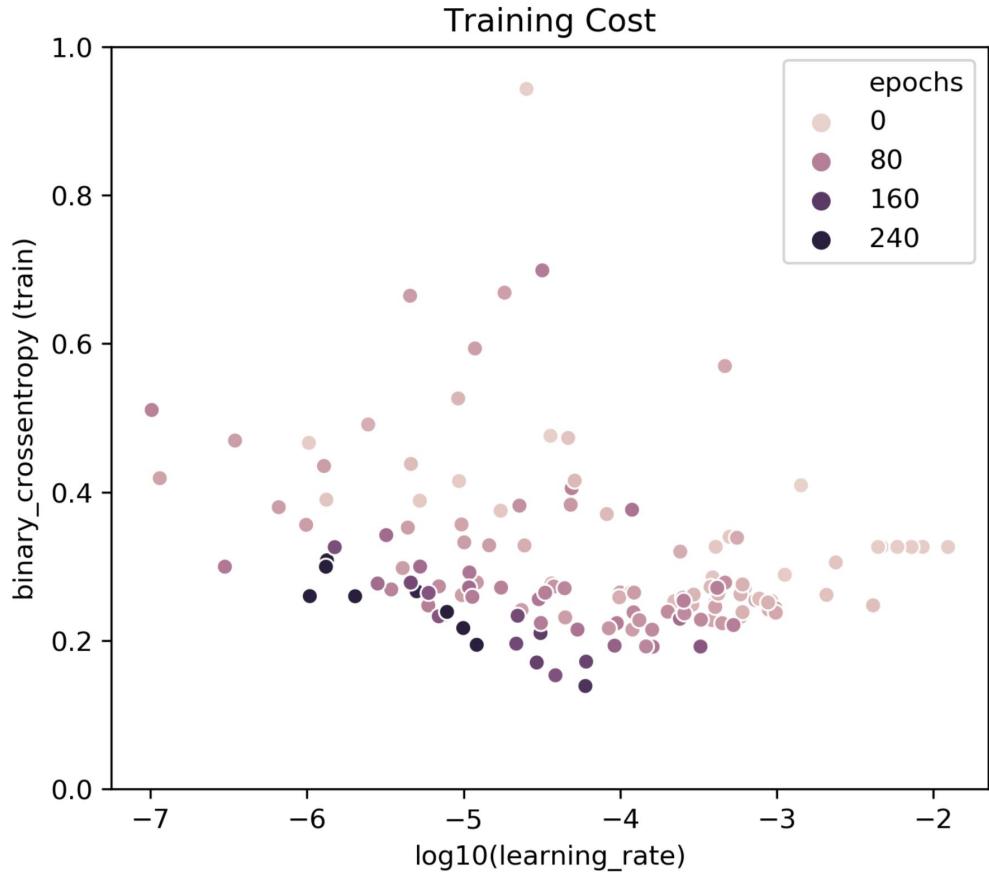


# Tuning Learning Rate

Sample from **uniform distribution**, but use exponential scale for ratios.

Narrow down after reviewing enough training jobs.

Some hyperparameters can be judged on **partial dataset before learning curve convergences**.

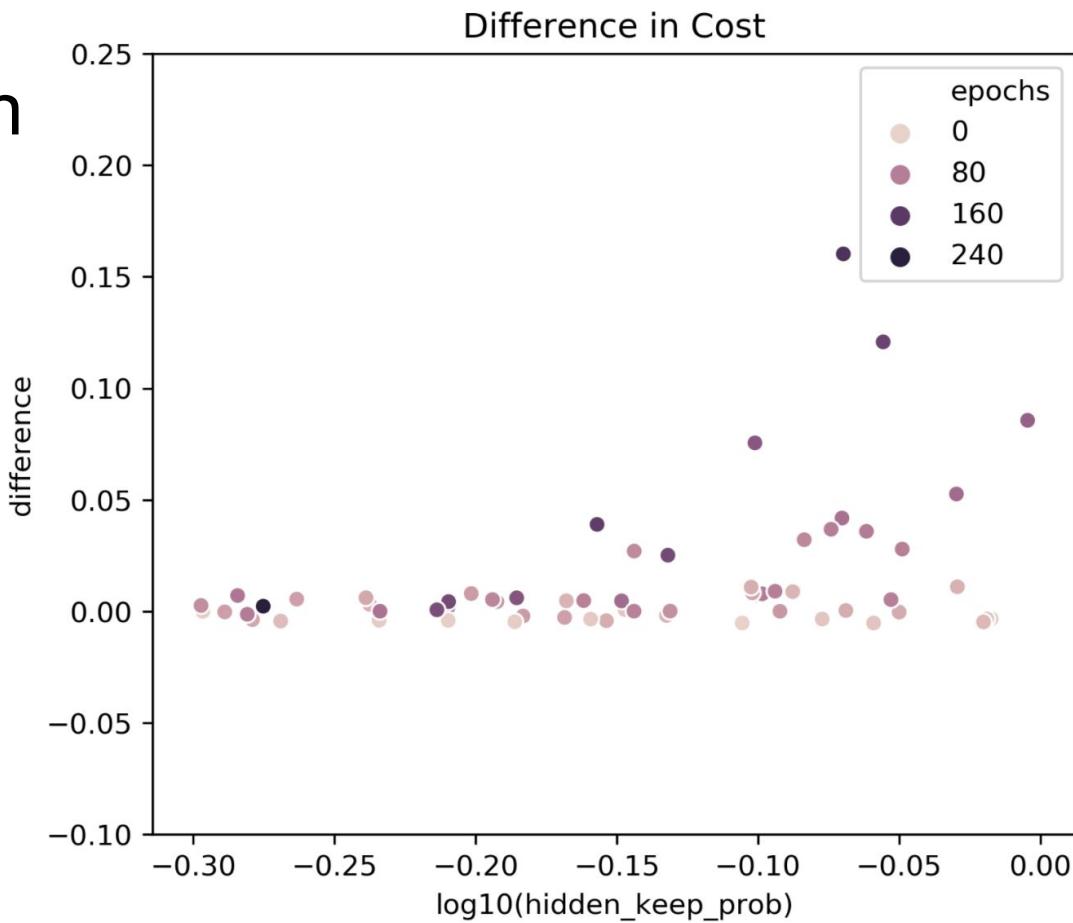


# Tuning Regularization

Regularization hyperparams need to be judged on the **validation/training cost gap**.

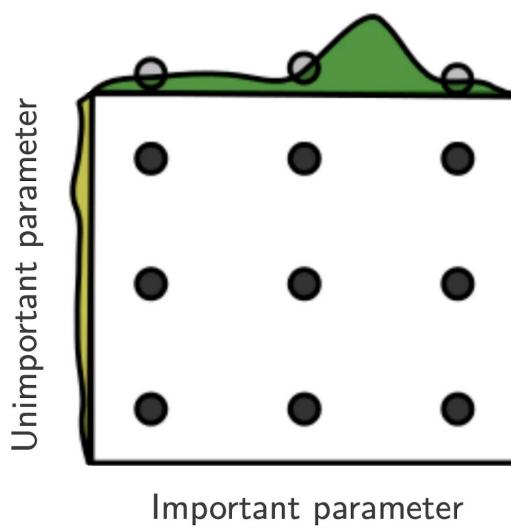
Plot the hyperparam vs the validation/training cost gap.

Identify maximum level of regularization that keeps the gap from exploding.

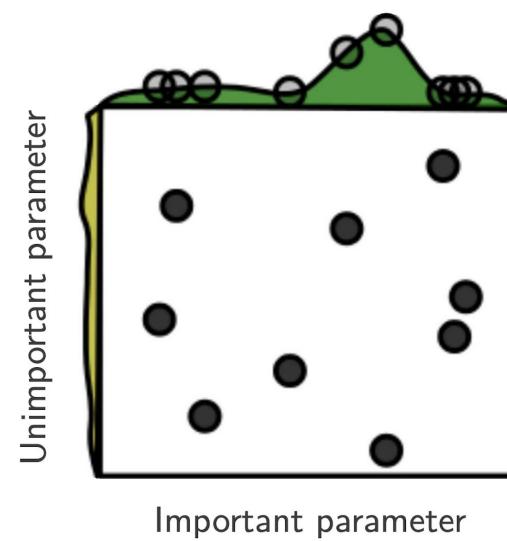


# Grid Search vs Random Search

Grid Layout



Random Layout



Source: Random Search for Hyper-Parameter Optimization (Bergstra, J. & Bengio, Y., 2012)

# TensorFlow / Keras

- Why graph-based computation?
  - Distributed computation
    - Between CPUs/GPUs
    - Between servers
  - Easy backpropagation
- Pre-trained models
  - <https://www.tensorflow.org/hub>
  - <https://keras.io/applications>
- Excellent community
  - <http://playground.tensorflow.org>



# Demo

Local Jupyter and AWS SageMaker

# Part 2

## Autoencoders

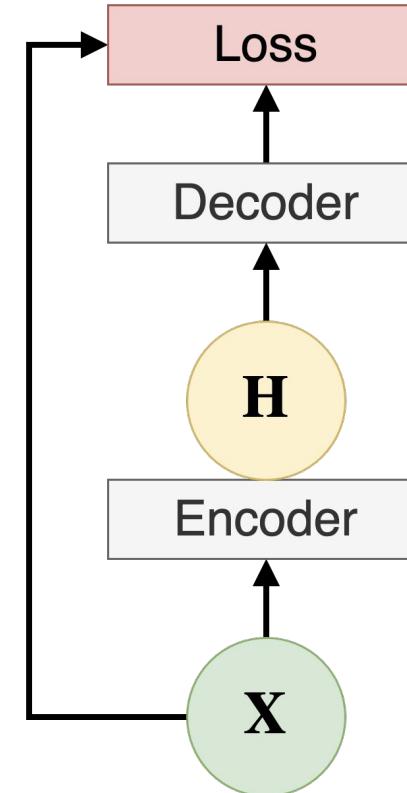
# Autoencoders

## Problem

Extract **useful representation** of the data without explicitly specifying how to do it.

## Solution

Train a network to reconstruct the input under **various restrictions**.



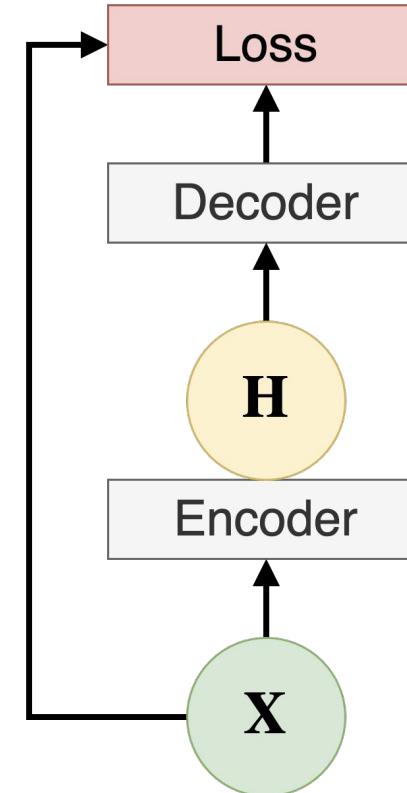
# Undercomplete Autoencoders

## Problem

Prevent the network from **copying the input** to the latent representation.

## Solution

Reduce **dimensionality** of the latent representation.



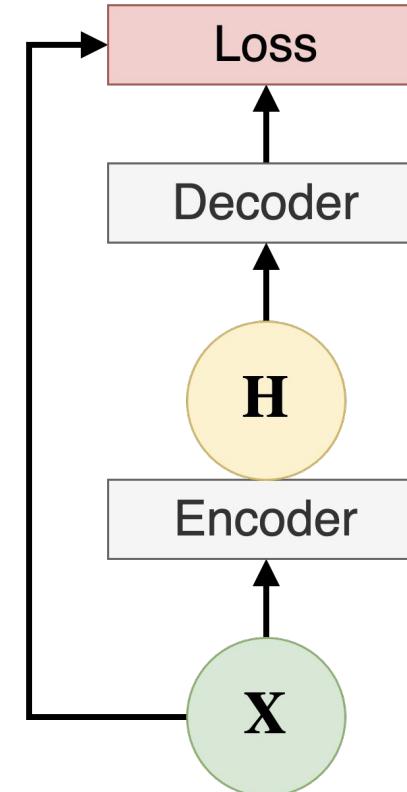
# Use Case: Dimensionality Reduction

## Problem

Reduce the **compute and storage** requirements without losing important information.

## Solution

Use the **latent representation** of the data.



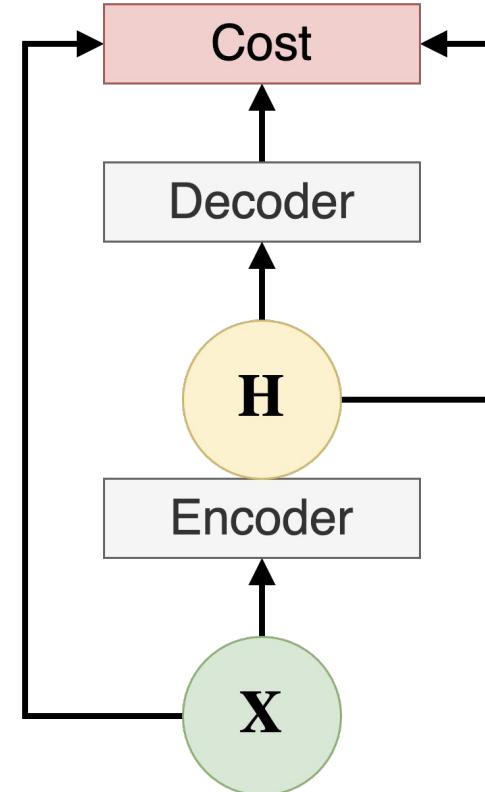
# Sparse Autoencoders

## Problem

Prevent the network from  
**copying the input** to the  
latent representation.

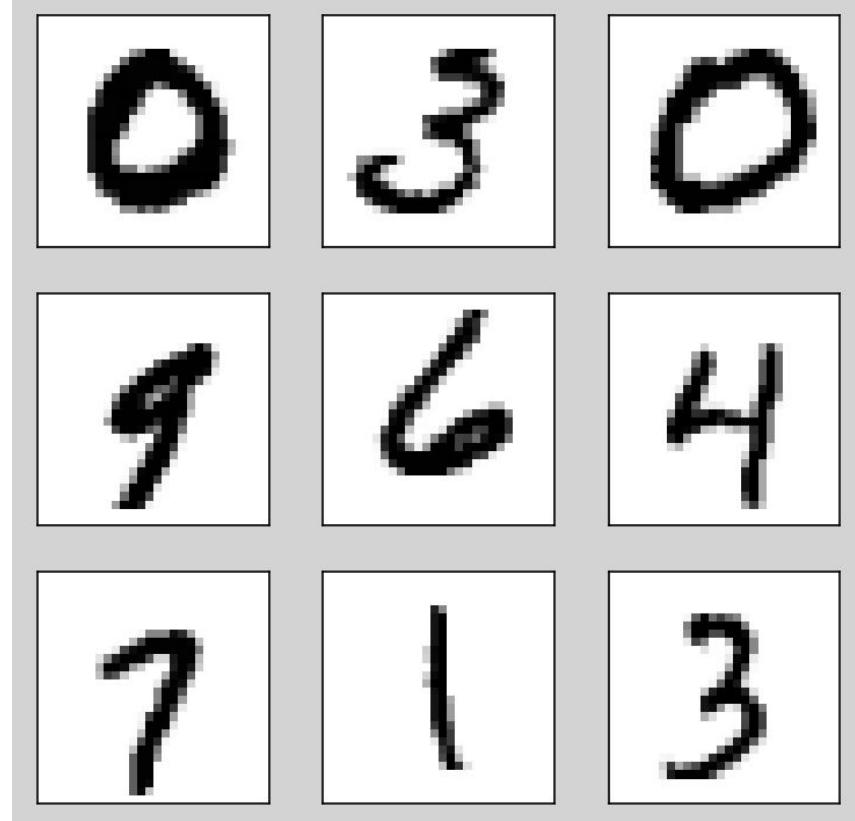
## Solution

Add **L1 penalty** on the latent  
representation (not the  
weights) to the total cost.



# Case Study: MNIST

- Dataset
  - 70K images of handwritten digits
  - Each image is  $28 \times 28$  in  $[0, 1]$  (grayscale)
- Dimensionality Reduction Task
  - Reduce the dimensions from 784 to ???
  - Scored with MSE of the reconstructions against the original images



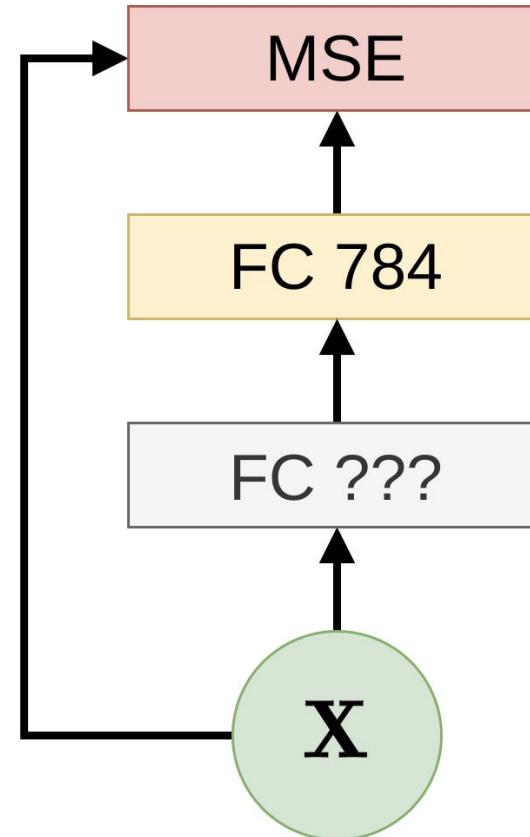
# Model: Training

**Undercomplete  
Autoencoder**

**Encoding Width**  
???

**Encoder Depth**  
1 layer

**Activation**  
ELU



# Hyperparam Search: Greedy Strategy

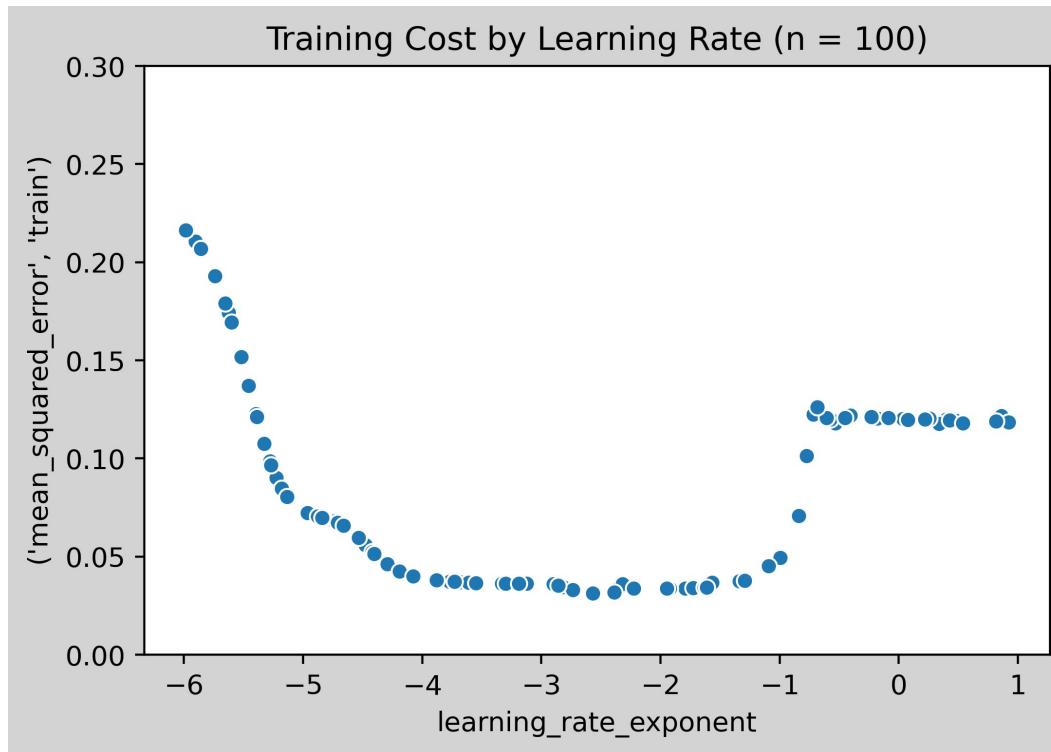
Sequence of hyperparam search:

1. Learning rate
2. Encoder width
3. Initialization seed
4. Final training

Pros: easy to implement

Cons: sequence needs to be carefully ordered

# Hyperparam Search: Learning Rate



## Iteration

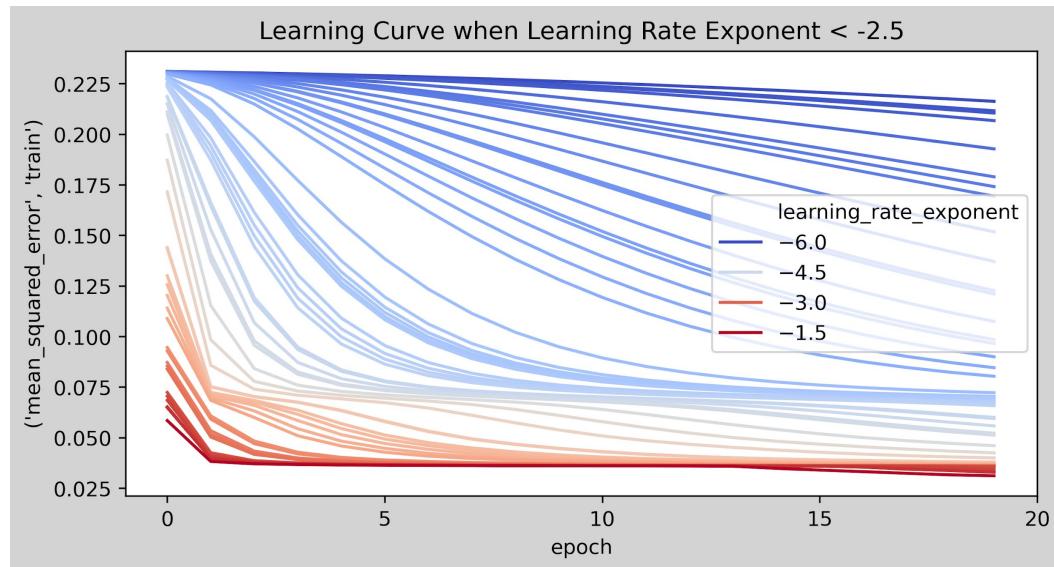
1.  $t \sim \text{Uniform}(-6, 1)$
2. Learning Rate =  $10^{** t}$
3. Train for 20 epochs

Repeat to train 100 models

## Conclusion

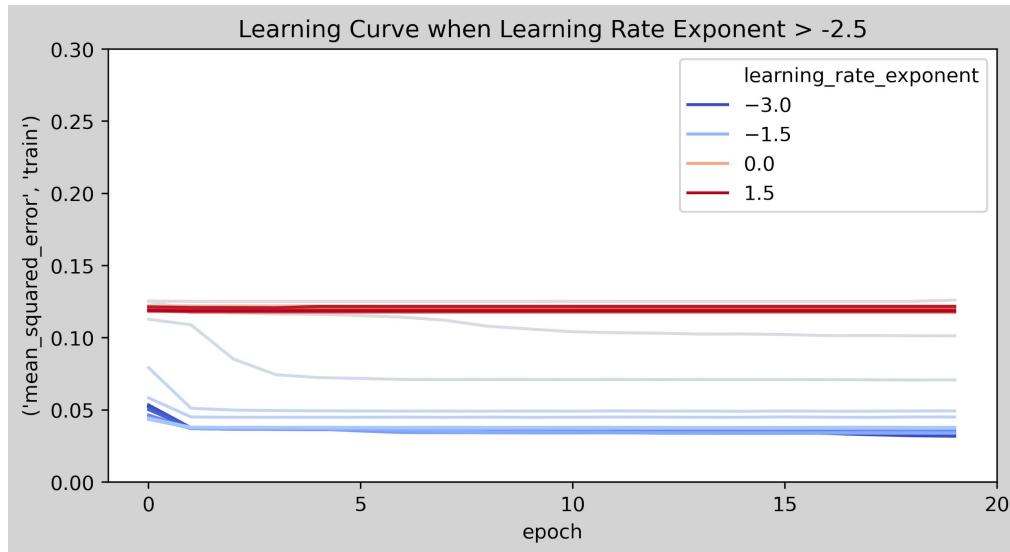
Optimal initial learning rate =  
 $10^{** -2.5}$

# Hyperparam Search: Learning Rate



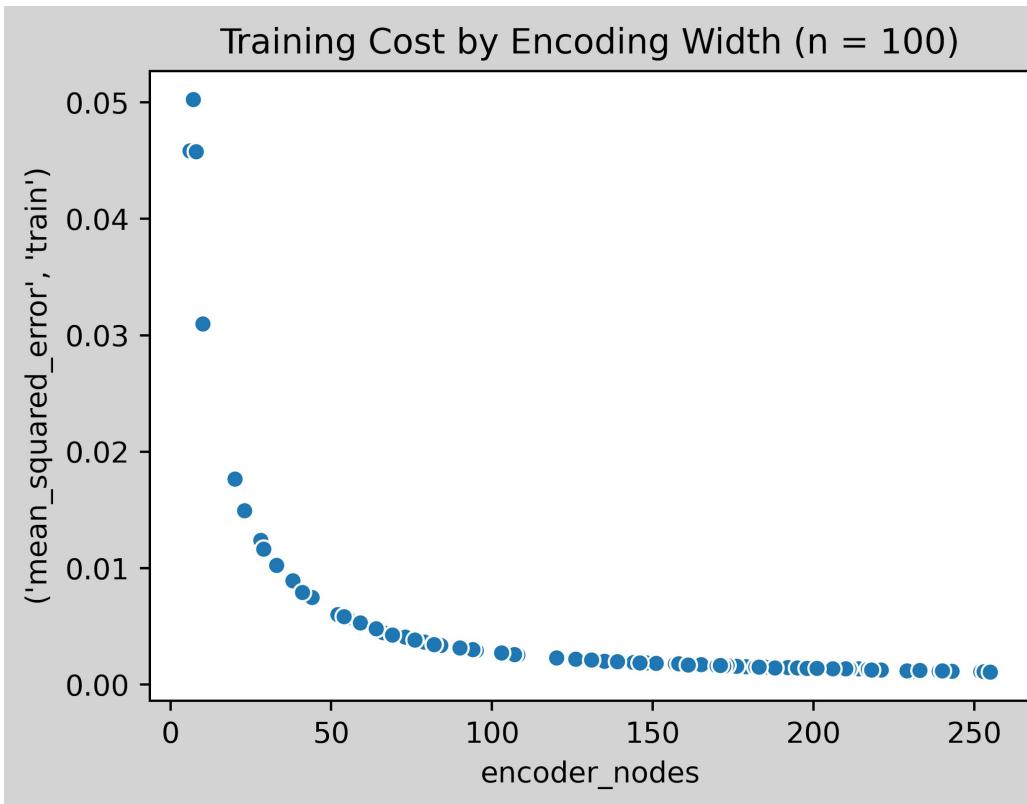
Learning curves with learning rate **below optimal** are more flat.

# Hyperparam Search: Learning Rate



Learning curves with learning rate **above optimal often appear very flat.**

# Hyperparam Search: Encoding Width



## Iteration

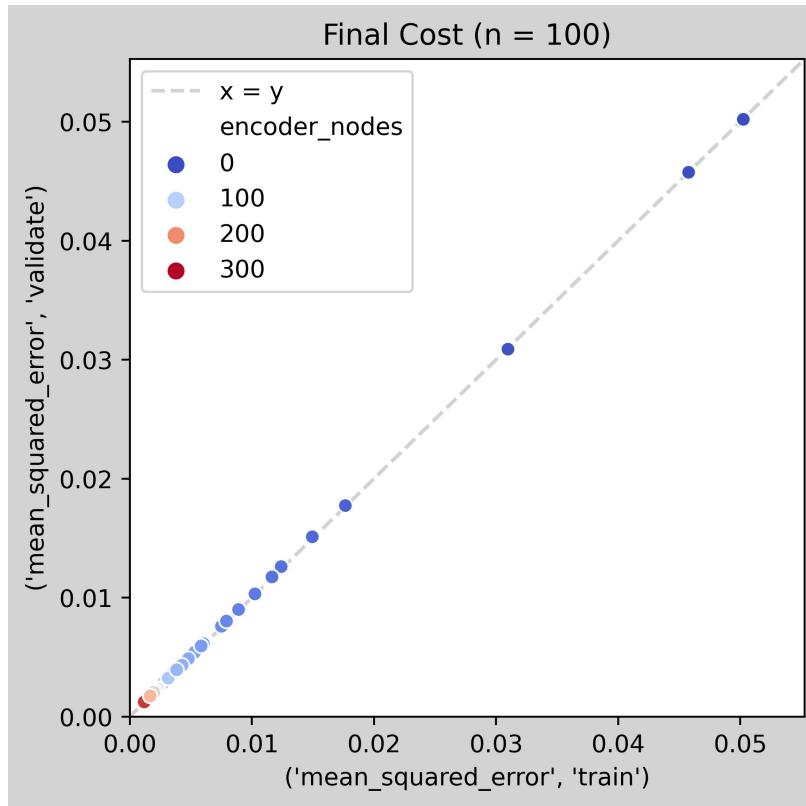
1. Width ~ Uniform(4, 256)
2. Train for 20 epochs

Repeat to train 100 models

## Conclusion

Denoising error levels out at ~100 nodes.

# Hyperparam Search: Encoding Width

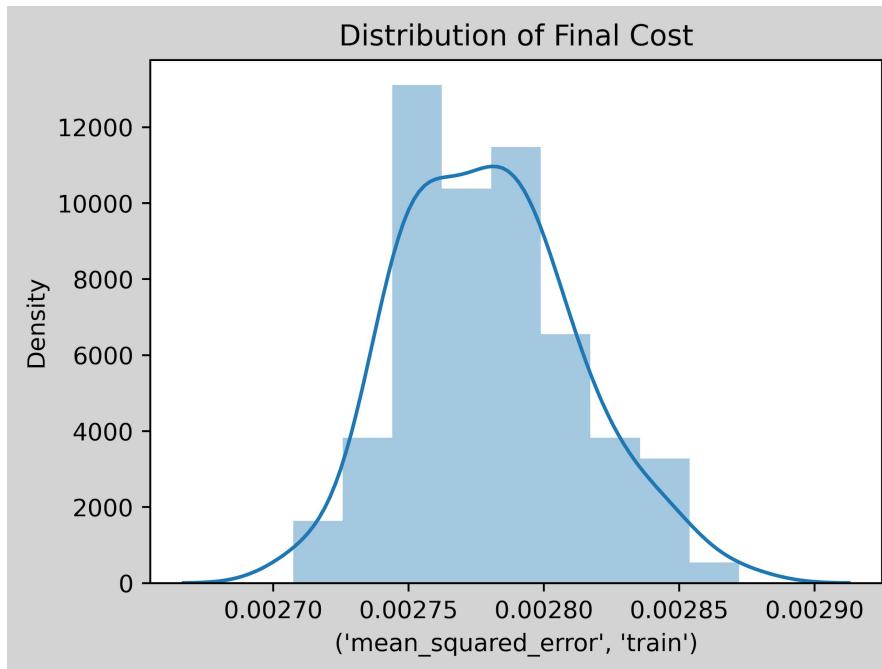


Autoencoder **can overfit if the decoder is reproducing training examples** without using the encoded representation.

## Conclusion

No overfitting under 256 nodes.

# Hyperparam Search: Weight Initialization



## Iteration

1. Seed ~ Uniform(0,  $2^{32} - 1$ )
1. Weights ~ Glorot Uniform
2. Train for 20 epochs

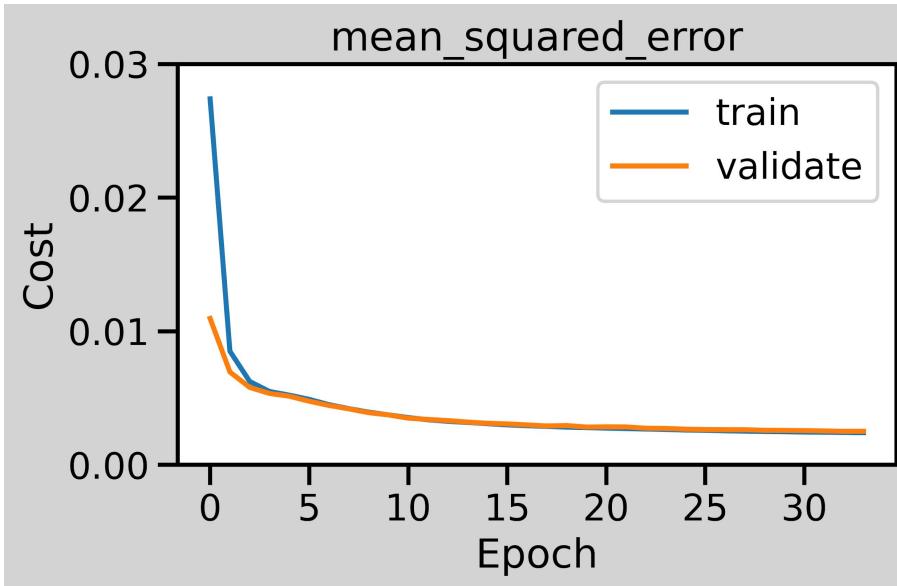
Repeat for 100 models

## Inference

Choice of initial weights does not seem to matter much in this case.

Use seed with the lowest log loss.

# Final Training



## Hyperparams

Optimizer = Adam

Batch Size = 128

Learning Rate =  $10^{-2.5}$

Encoder width = 100

Initialization seed = 2366951942

Epochs = Early stopping  
triggered after 34 epochs

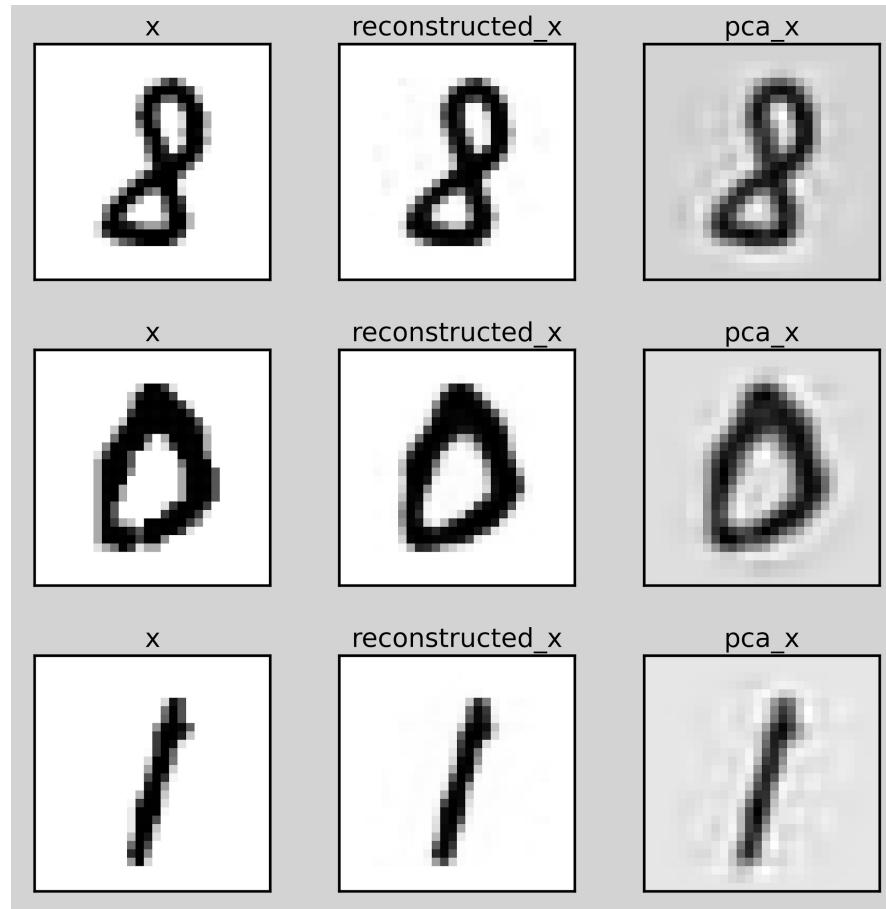
## Conclusion

Learning curve converged  
towards MSE = 0.0024

# Image Examples

Images reconstructed by Autoencoder appear to be almost identical to the original images.

Model	Task	MSE
Undercomplete AE	Reconstruction	0.0024
PCA	Reconstruction	0.0057



# Denoising Autoencoders: Training

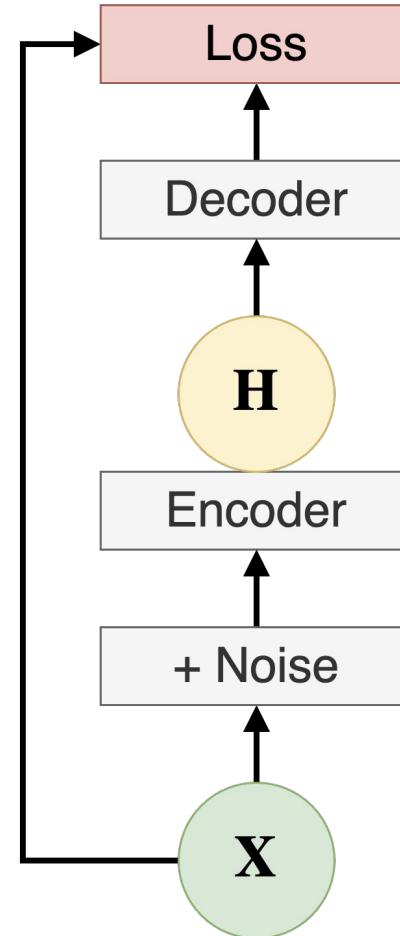
## Problem

Remove noise from the input  
given its distribution.

## Solution

Add random noise to the input,  
while minimizing difference with  
the clean input.

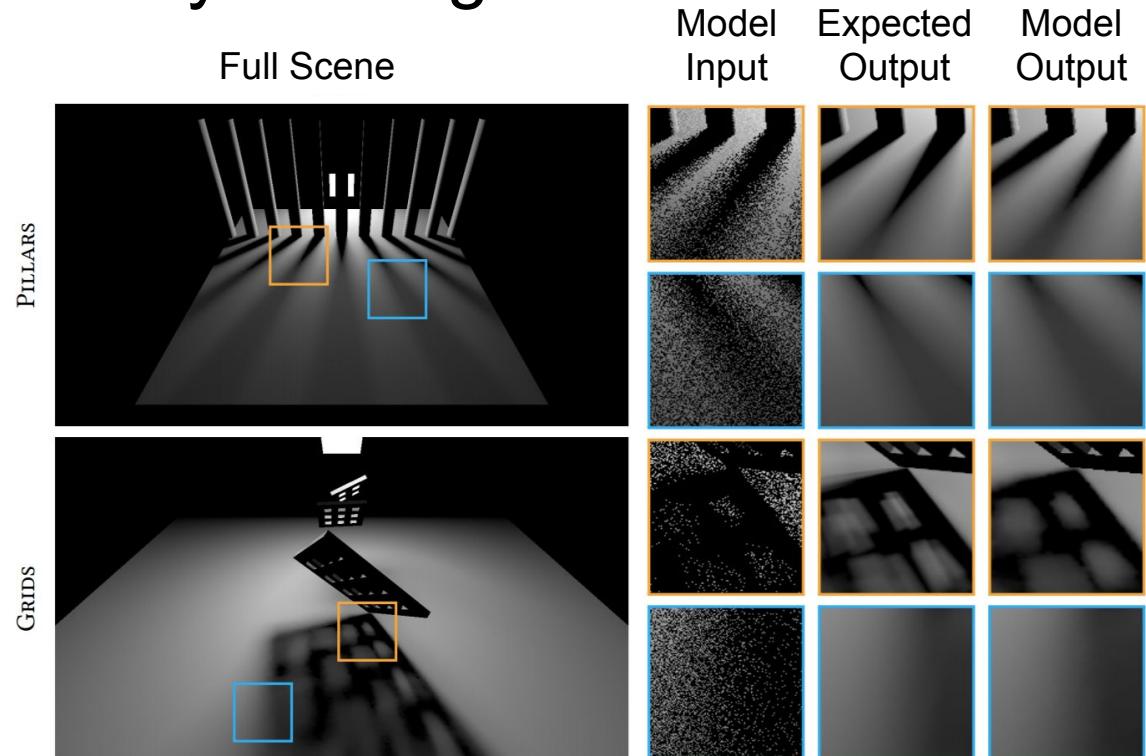
Vincent, P., et al. (2008)



# Case Study: NVIDIA Ray Tracing

Ray Tracing engine produces noisy images if not given enough time to run.

Denoising autoencoder **removes noise faster than the Ray Tracing engine** would take to achieve the same quality.

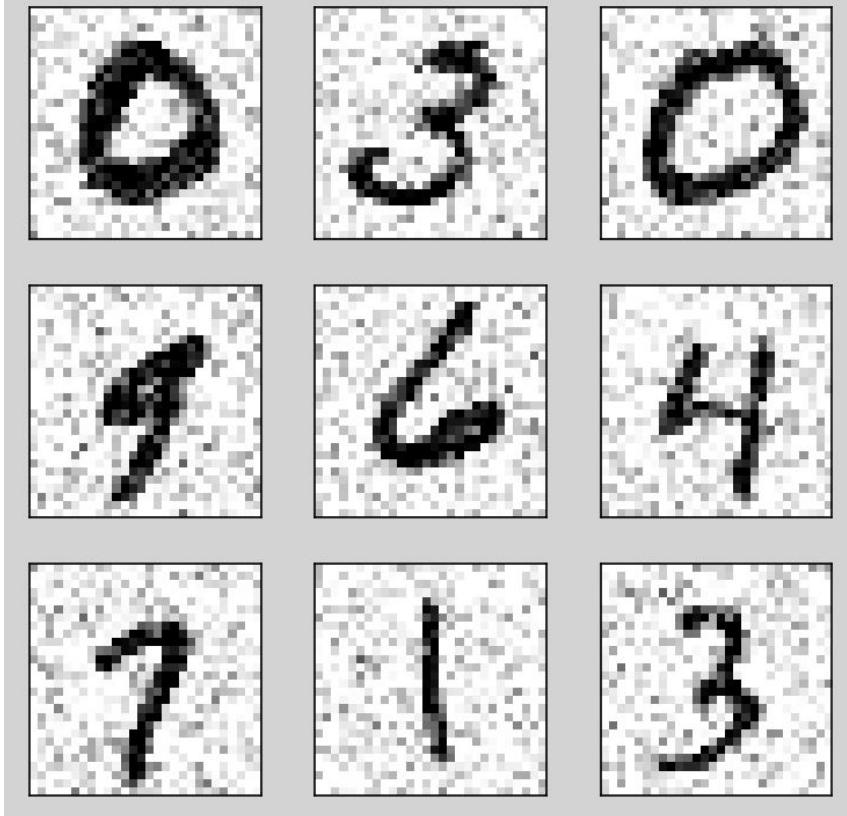


# Case Study: NVIDIA Ray Tracing



# Case Study: MNIST

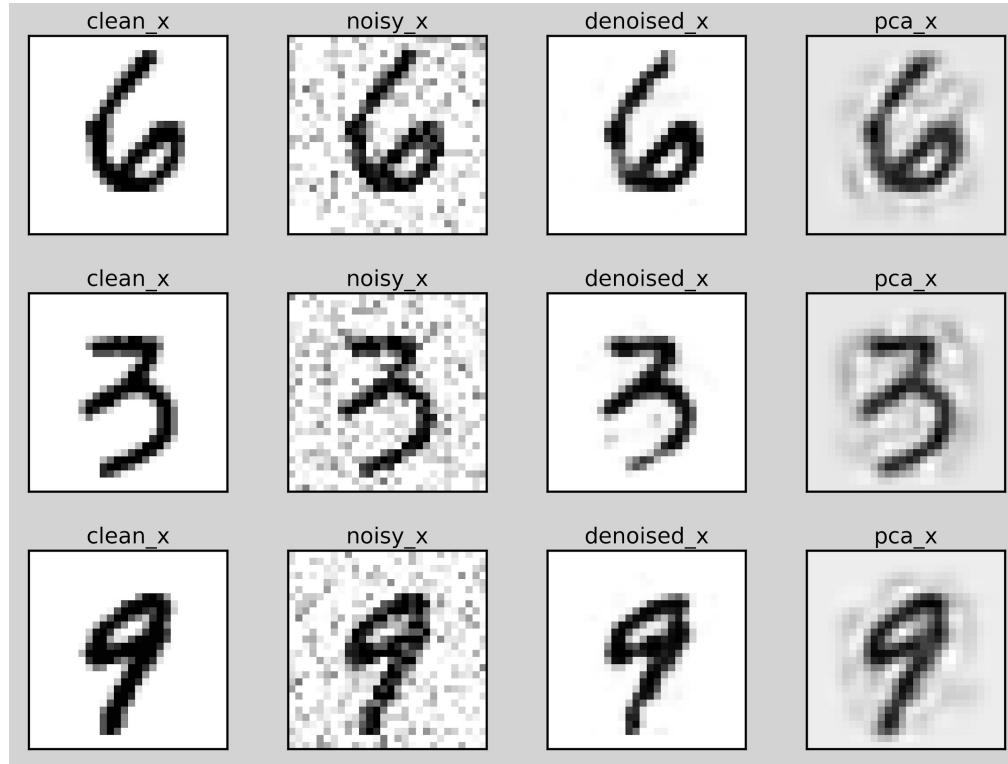
- Dataset
  - 70K images of handwritten digits
  - Each image is  $28 \times 28$  in  $[0, 1]$  (grayscale)
  - Augmented with  $\text{Gaussian}(0, 0.2)$
- Denoising Task
  - Remove noise from the images
  - Scored with MSE against original images



# Undercomplete Autoencoder?

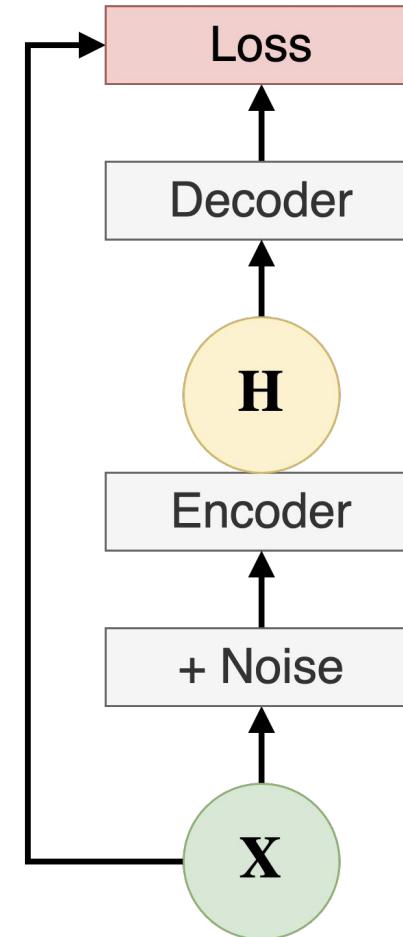
Undercomplete autoencoder can also be used for denoising even though **the noise distribution was not specified** either implicitly or explicitly during training.

Model	Task	MSE
Undercomplete AE	Reconstruction	0.0024
Undercomplete AE	Denoising	0.0123
PCA	Denoising	0.0107

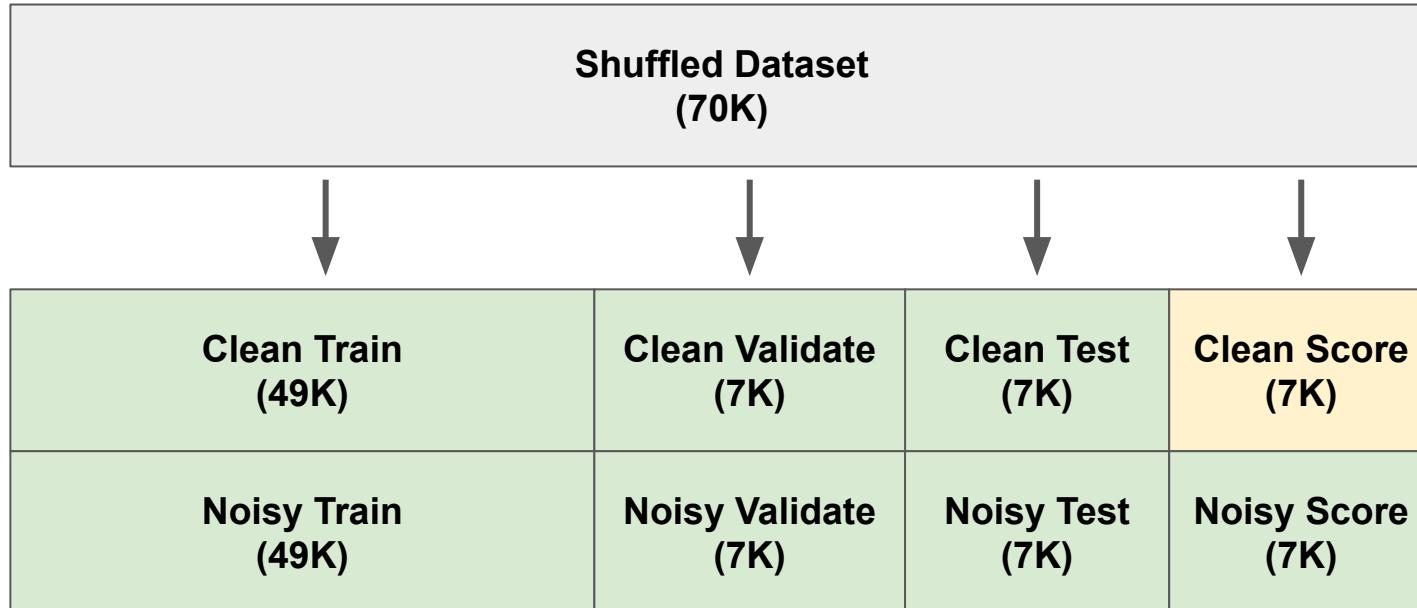


# Assignment #1

- Train an Autoencoder
  - Architecture: undercomplete and/or sparse
  - **Training mode: denoising**
- Scored on Denoising Task
  - Methodology on the next slides
- Augmented MNIST Dataset
  - Noise  $\sim$  Normal(0, 0.2)
  - Shape = 70K x 28 x 28
  - Data Type = float32 in [0, 1]
  - File Type = Parquet



# Assignment Dataset



# Solution Methodology

- Read the Jupyter notebooks
- Must implement denoising training mode
- Try different Autoencoder architectures:
  - undercomplete
  - sparse
  - or both
- Try different encoder width / depth
- Try different optimizers
- Try different learning rates
- Try learning rate decay

# Demo

UndercompleteAutoencoder.ipynb

# Assignment Deliverables

1. Jupyter notebook and/or Python script
  - a. Data preprocessing
  - b. Construction and training of the final model
  - c. Inference on the Noisy Score dataset
2. Final model definition
  - a. Keras serialization in JSON format
3. Final model parameters
  - a. Keras serialization in H5 format
4. Denoised images from the Noisy Score dataset
  - a. Shape = 7K x 28 x 28
  - b. Data Type = float64 in [0, 1]
  - c. File Type = Parquet

# Assignment Grading

- Due EOD 10/13
  - Zero points for late submissions
  - Zero points for incomplete submissions
  - Zero points for non-denoising training mode
- Maximum points = 20
  - 10 points if MSE (Denoised vs Clean) on Score Dataset < 0.02
  - 10 points proportional to percentile of MSE on Score Dataset relative to the class
  - In 2019, the average MSE was 0.0120
  - My best MSE was 0.004, but I know it can be even lower
- Compute Environments
  - Your laptop (Anaconda + Jupyter)
  - AWS EC2

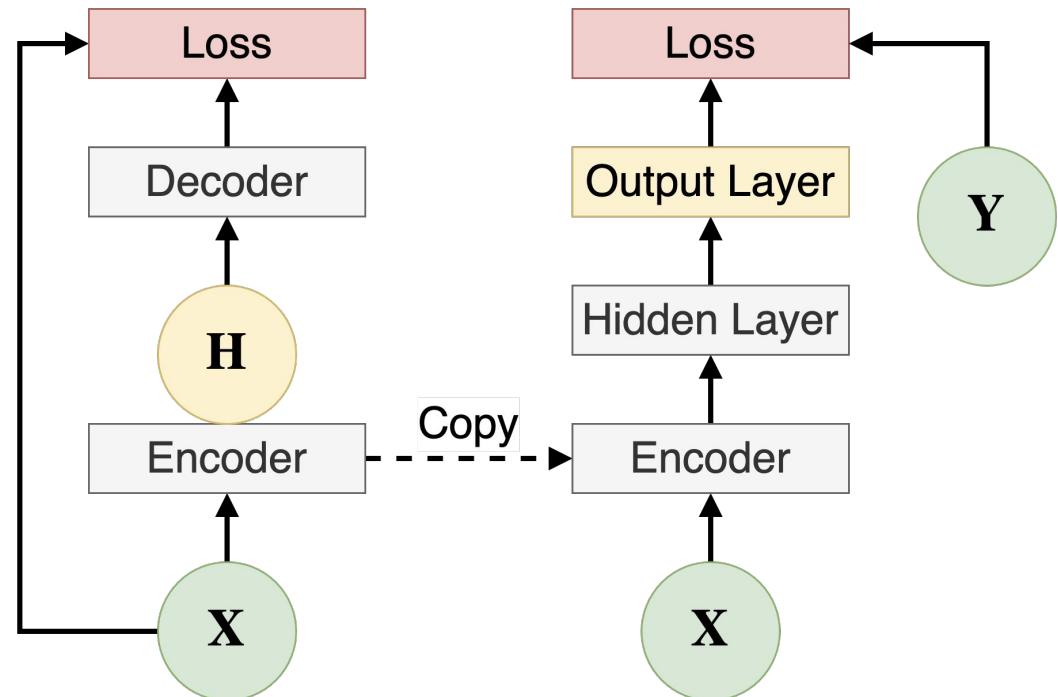
# Unsupervised Pre-training

## Problem

Initialize a network with parameters that are **better than random**.

## Solution

Use **pre-trained encoder** layer as the bottom of the network.

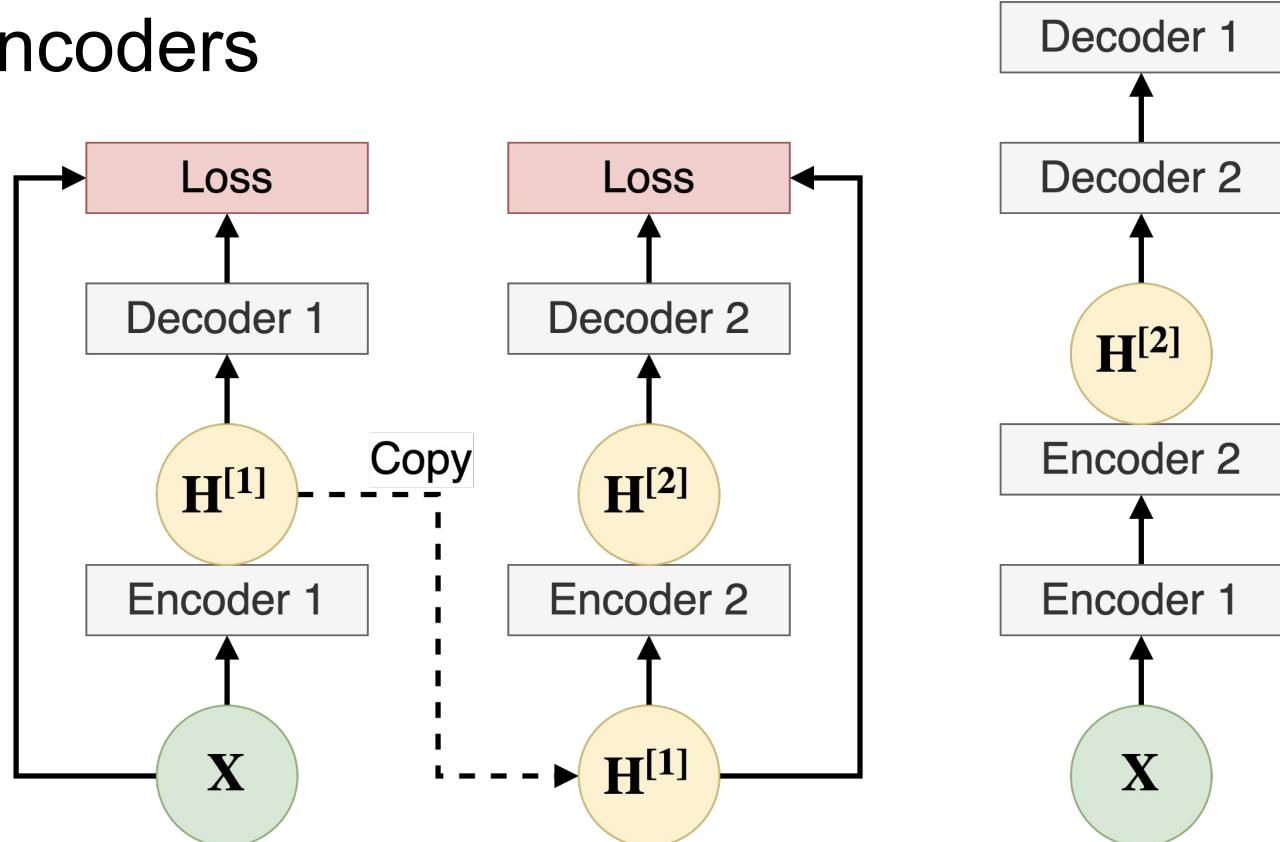


# Stacked Autoencoders

Deep autoencoders are typically trained in **greedy layer-wise** procedure.

Layers trained one at a time, starting with the outermost.

Finally, they are stacked into a single autoencoder.



Bengio, Y., et al. (2007)

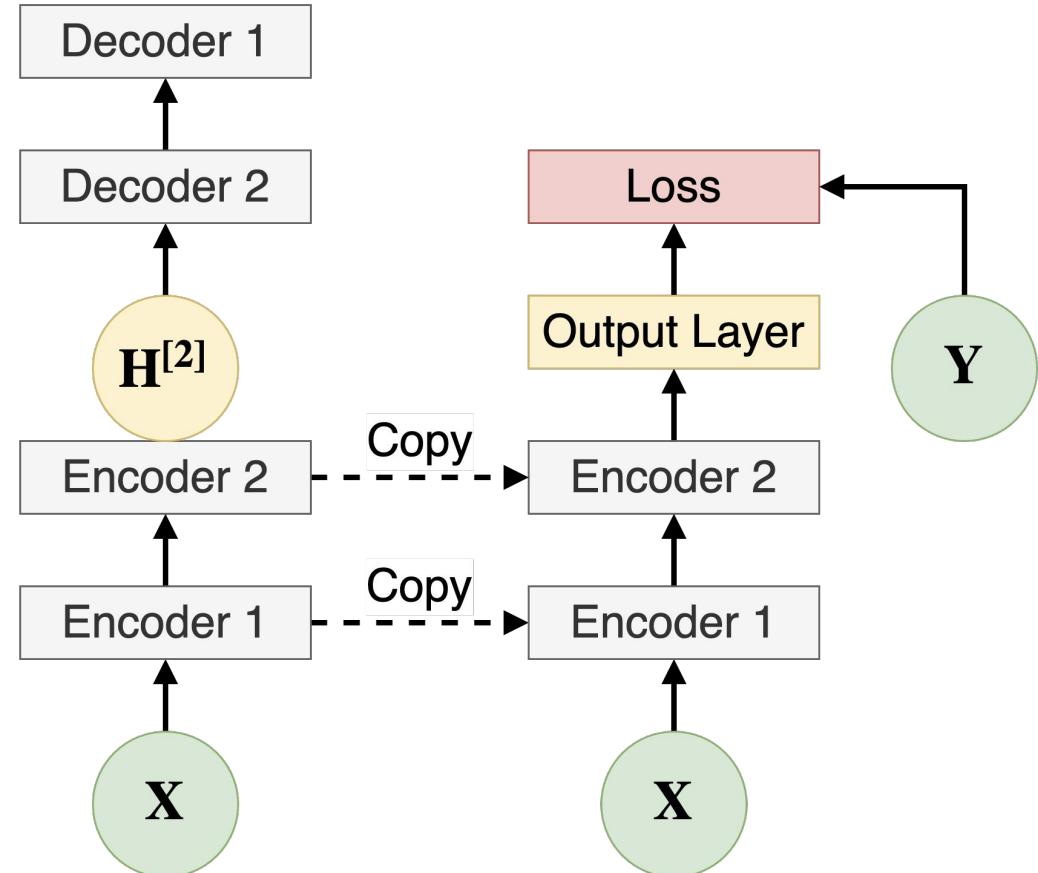
# Deep Unsupervised Pre-training

## Problem

Initialize a network with parameters that are **better than random**.

## Solution

Use **pre-trained stacked encoder** layers as the bottom of the network.



## Step 1: Store Records

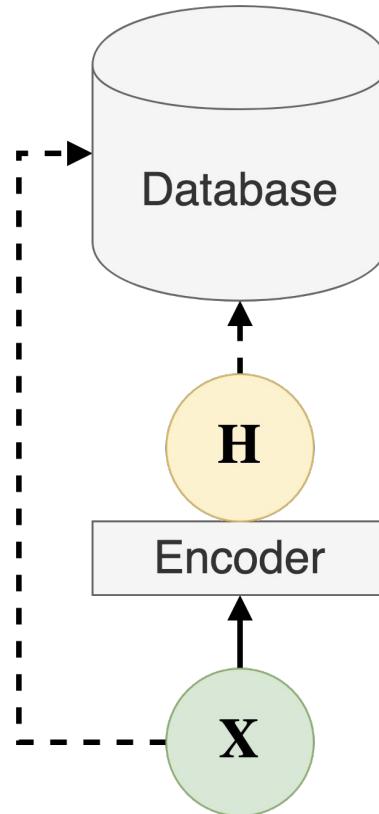
# Use Case: Semantic Hashing

### Problem

Find entries **similar to the query input** from a very large dataset.

### Solution

Use the latent representation as the key in a database.



Salakhutdinov, R., and Hinton, G. (2007)

# Use Case: Semantic Hashing

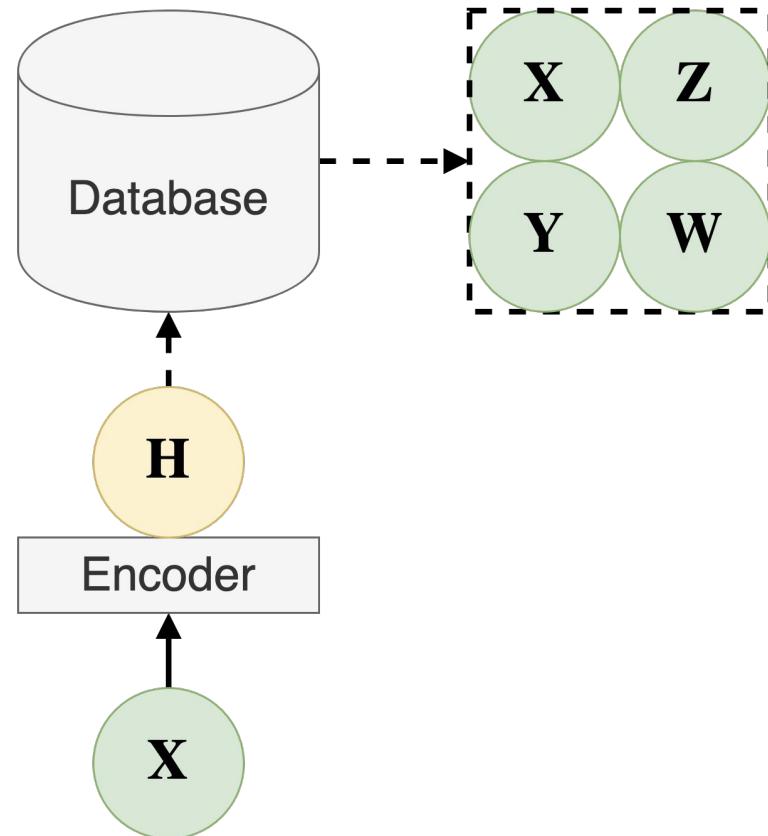
## Problem

Find entries **similar to the query input** from a very large dataset.

## Solution

Use the latent representation as the key in a database.

## Step 2: Retrieve Records



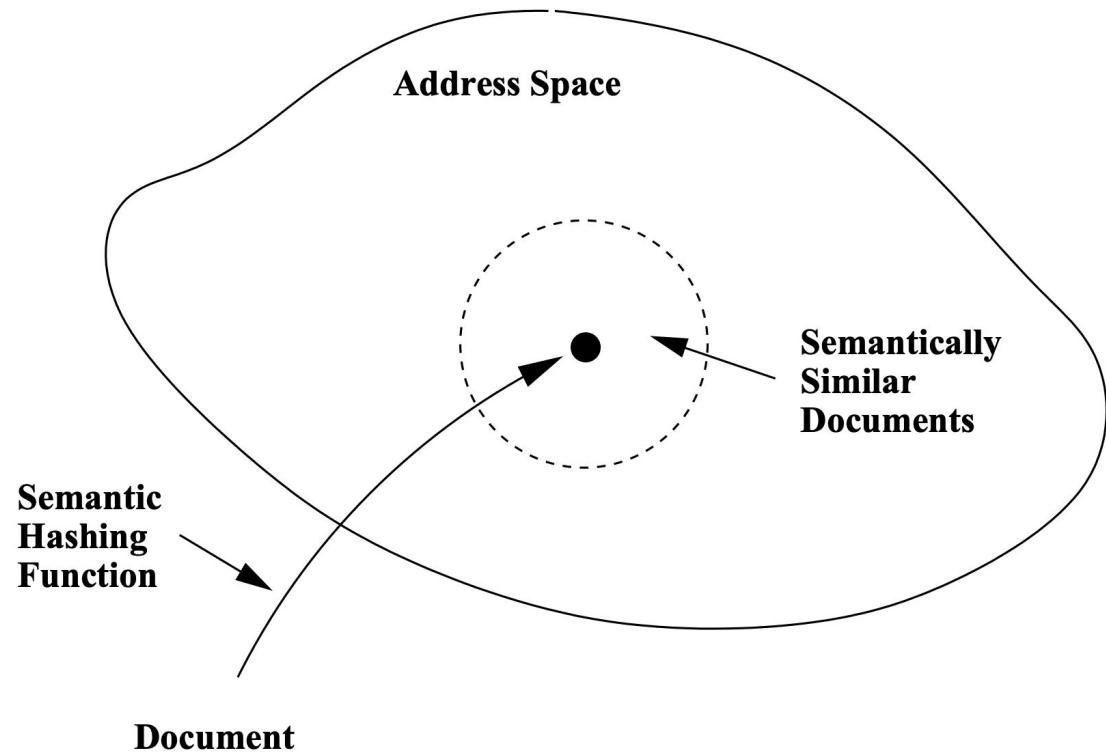
# Use Case: Semantic Hashing

## Problem

Find entries **similar to the query input** from a very large dataset.

## Solution

Use the latent representation as the key in a database.



Salakhutdinov, R., and Hinton, G. (2007)

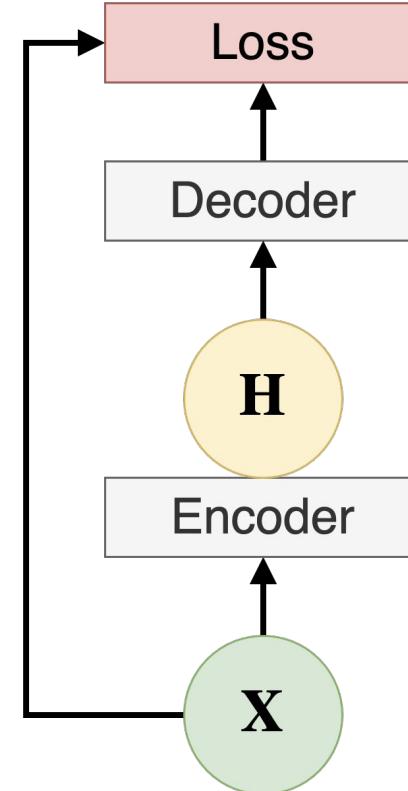
# Use Case: Anomaly Detection

## Problem

Detect inputs that are  
**very different from normal**  
inputs (e.g. anomalies).

## Solution

Use the reconstruction  
loss as the **estimate of the**  
**anomaly score.**



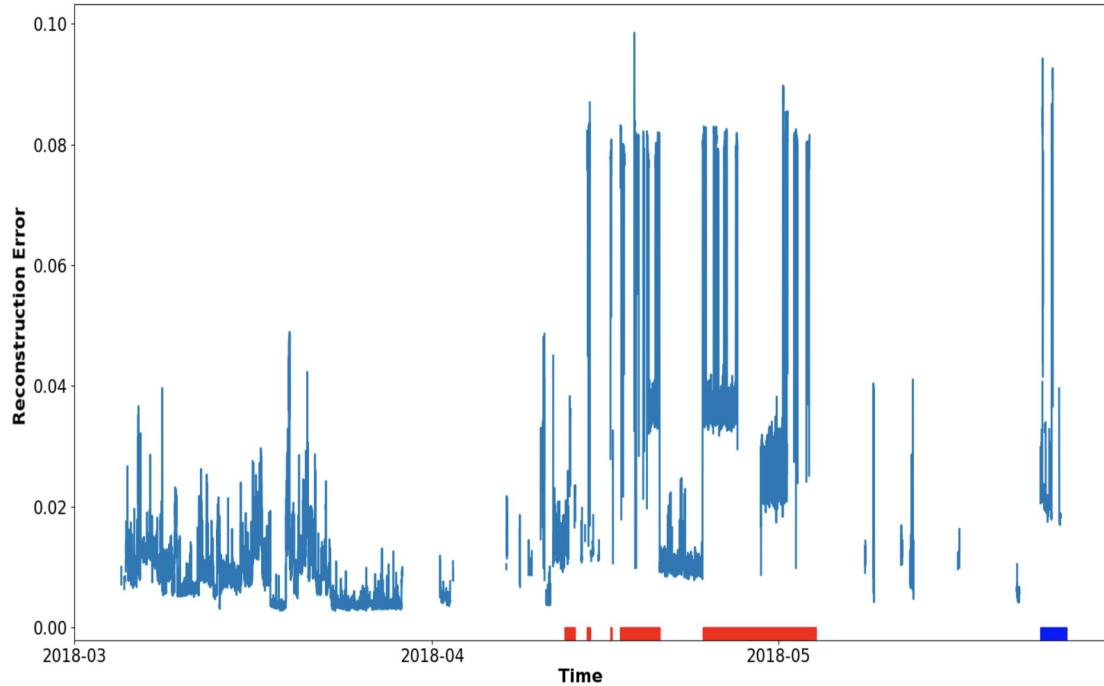
# Use Case: Anomaly Detection

## Problem

Detect inputs that are  
**very different from normal**  
inputs (e.g. anomalies).

## Solution

Use the reconstruction  
loss as the **estimate of the**  
**anomaly score.**



# References

1. Adam: A Method for Stochastic Optimization (Kingma, D., and Ba, J., 2017)
2. Anomaly Detection using Autoencoders in High Performance Computing Systems (Borghesi, A., and Bartolini, A., 2018)
3. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (Ioffe S., Szegedy C., 2015).
4. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (He, K., et al., 2015)
5. **Deep Learning (Goodfellow, I., Bengio, Y, and Courville, A., 2016)**
6. Extracting and Composing Robust Features with Denoising Autoencoders (Vincent, P., et al., 2008)
7. Greedy Layer-Wise Training of Deep Networks (Bengio, Y., et al., 2007)
8. Machine Learning (Mitchell, T., and Hill, M., 1997)
9. On the Momentum Term in Gradient Descent Learning Algorithms (Qian, N., 1999)
10. Random Search for Hyper-Parameter Optimization (Bergstra, J. & Bengio, Y., 2012)
11. Semantic Hashing (Salakhutdinov, R., and Hinton, G., et al., 2007)
12. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination (Schied, C., et al., 2017)
13. Stochastic Gradient/Mirror Descent: Minimax Optimality and Implicit Regularization (Azizan, N. & Hassibi, B., 2018)
14. Understanding the difficulty of training deep feedforward neural networks (Glorot, X., and Bengio Y., 2010)

# Thanks

+ Q&A Time