Machine Learning for Particle Physics

Sean Gasiorowski (SLAC)
sgaz@slac.stanford.edu

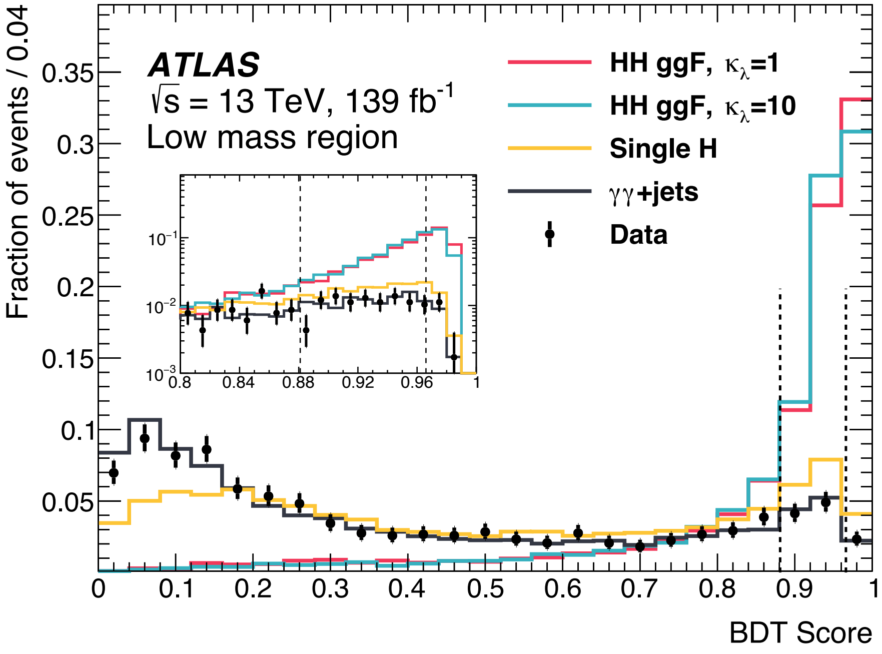
TRISEP

June 19th-20th, 2025

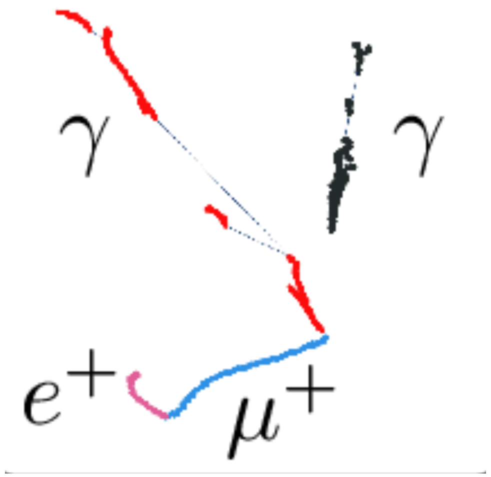
Machine Learning in Particle Physics

Machine Learning in Particle Physics

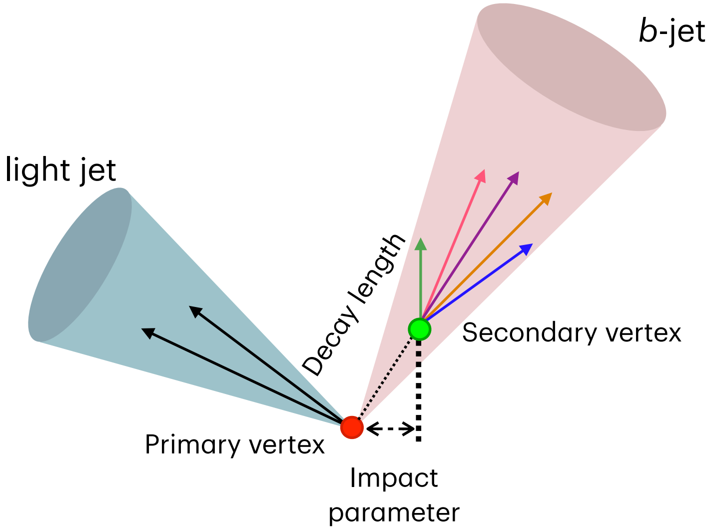
Classification: Make decisions about what group things belong to (e.g. signal vs background, particle identification/flavor tagging)



ATLAS bbyy



SPINE (LArTPC)

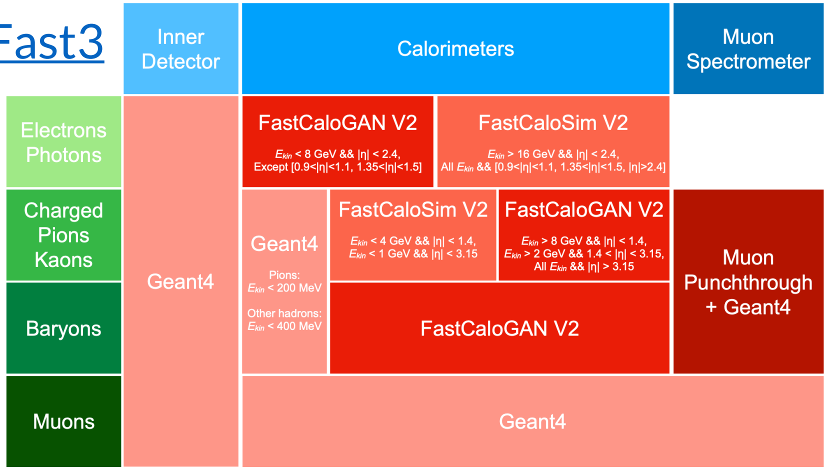


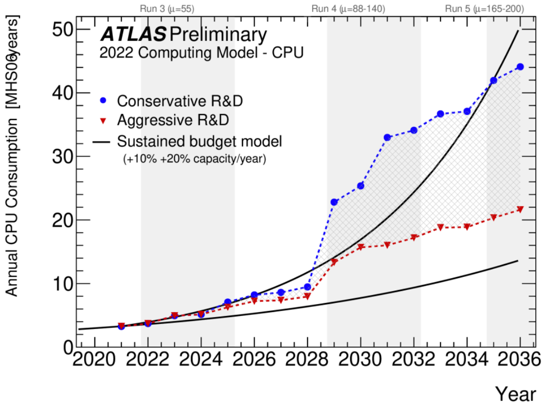
ATLAS GN2

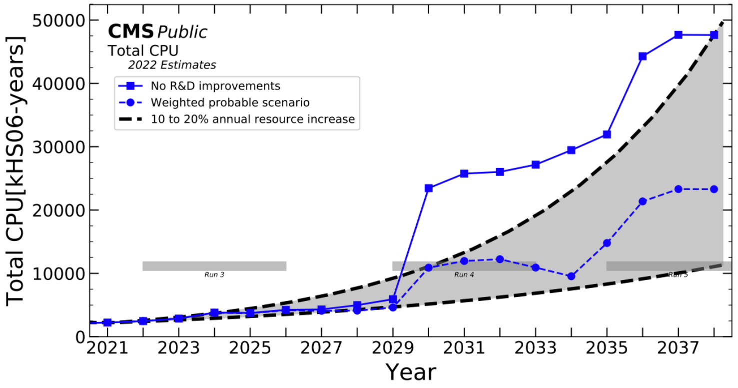
Machine Learning in Particle Physics

Simulation: Learn surrogate model (approximate, fast) to speed up simulation.

AtlFast3





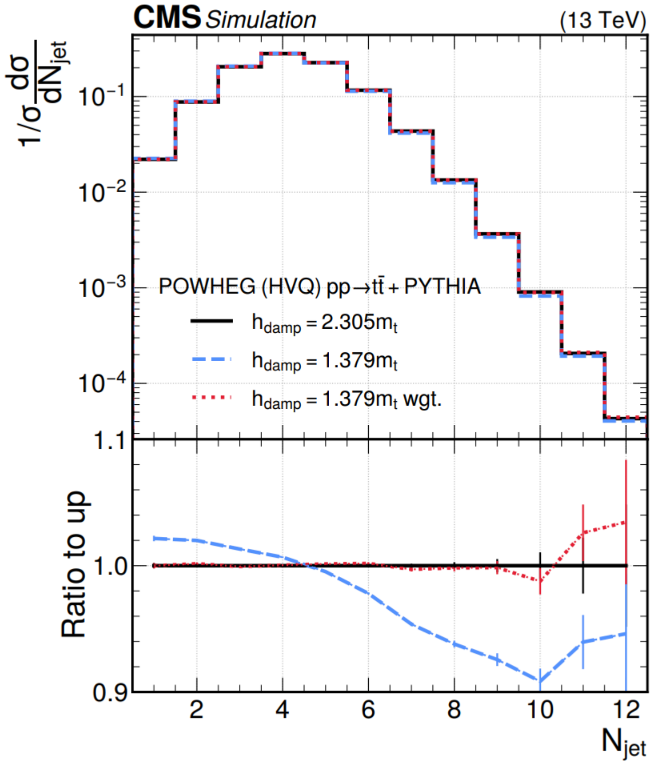


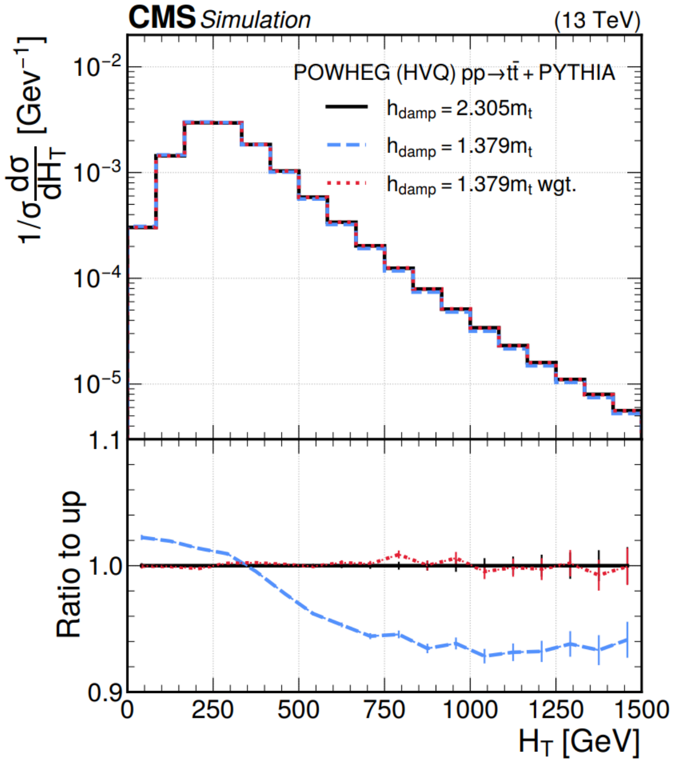
Approach	Model	Code	Dataset 1 − γ	1 − π	2	3	Section
GAN	CaloShowerGAN [21]	[22]	✓	✓			3.1
	MDMA [23, 24]	[25]			✓	✓	3.2
	BoloGAN [26]	[27]	✓	✓			3.3
	DeepTree [28, 29]	[30]			✓		3.4
NF	L2LFlows [31, 32]	[33]			✓	✓	4.1
	CaloFlow [34, 35]	[36, 37]	✓	✓	✓	✓	4.2
	CaloINN [38]	[39]	✓	✓	✓		4.3
	SuperCalo [40]	[41]			✓		4.4
	CaloPointFlow [42]	[43]			✓	✓	4.5
Diffusion	CaloDiffusion [44]	[45]	✓	✓	✓	✓	5.1
	CaloClouds [46, 47]	[48, 49]				✓	5.2
	CaloScore [50, 51]	[52, 53]	✓		✓	✓	5.3
	CaloGraph [54]	[55]	✓	✓			5.4
	CaloDiT [56]	[57]			✓		5.5
VAE	Calo-VQ [58]	[59]	✓	✓	✓	✓	6.1
	CaloMan [60]	[61]	✓	✓			6.2
	DNNCaloSim [62, 63]	[64]		✓			6.3
	Geant4-Transformer [65]	[66]				✓	6.4
	CaloVAE+INN [38]	[39]	✓	✓	✓	✓	6.5
	CaloLatent [67]	[68]			✓		6.6
CFM	CaloDREAM [69]	[70]			✓	✓	7.1
	CaloForest [71]	[72]	✓	✓			7.2

CaloChallenge

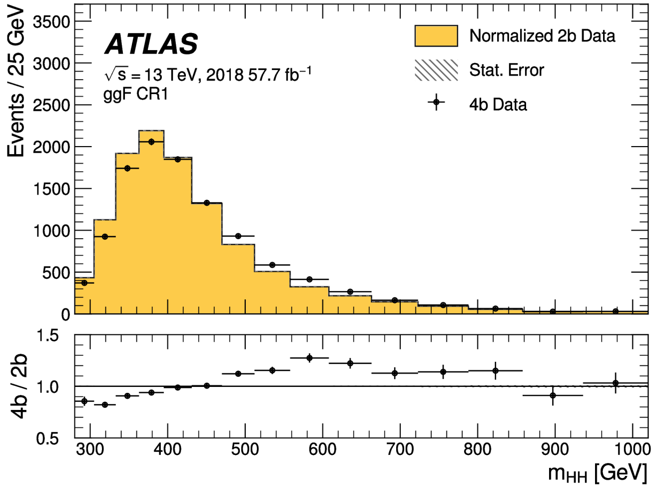
Machine Learning in Particle Physics

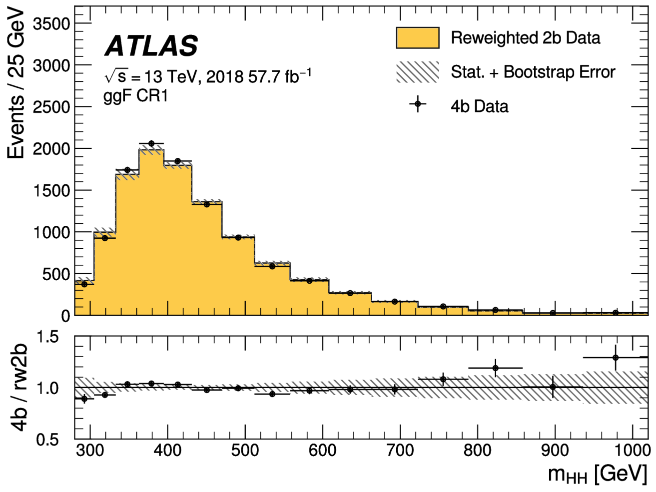
Reweighting: Learn transfer functions or density ratios to transform between distributions.





CMS Simulation Reweighting

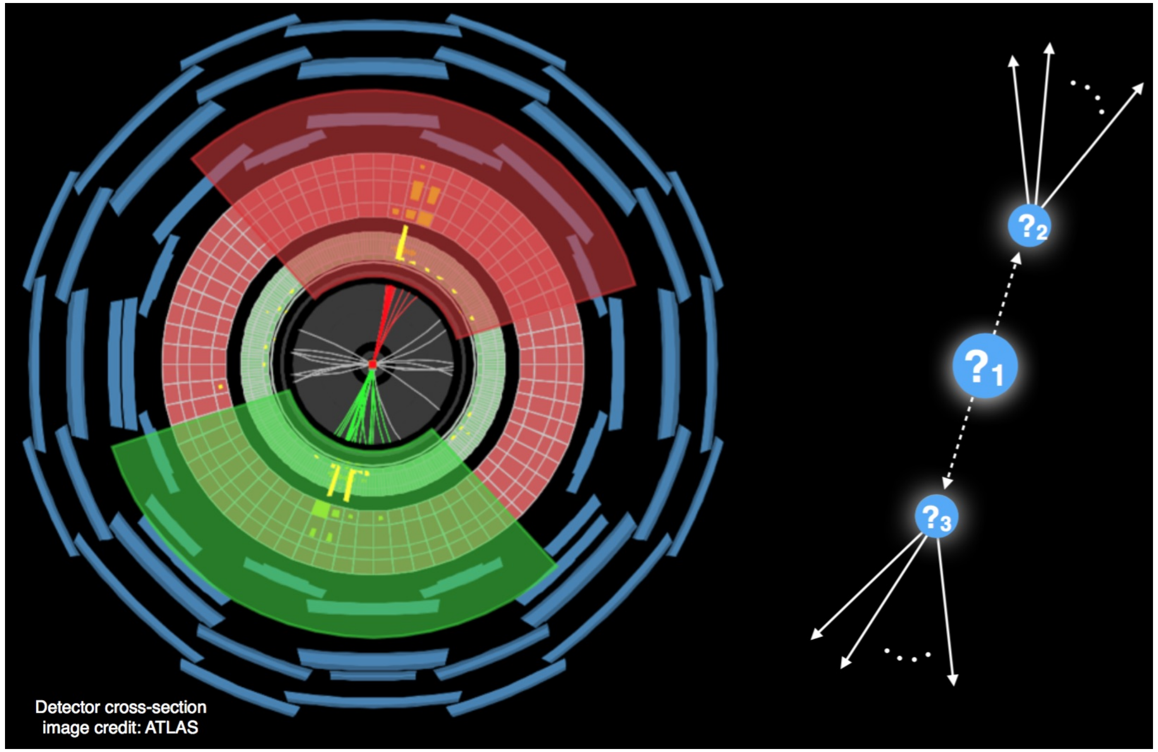




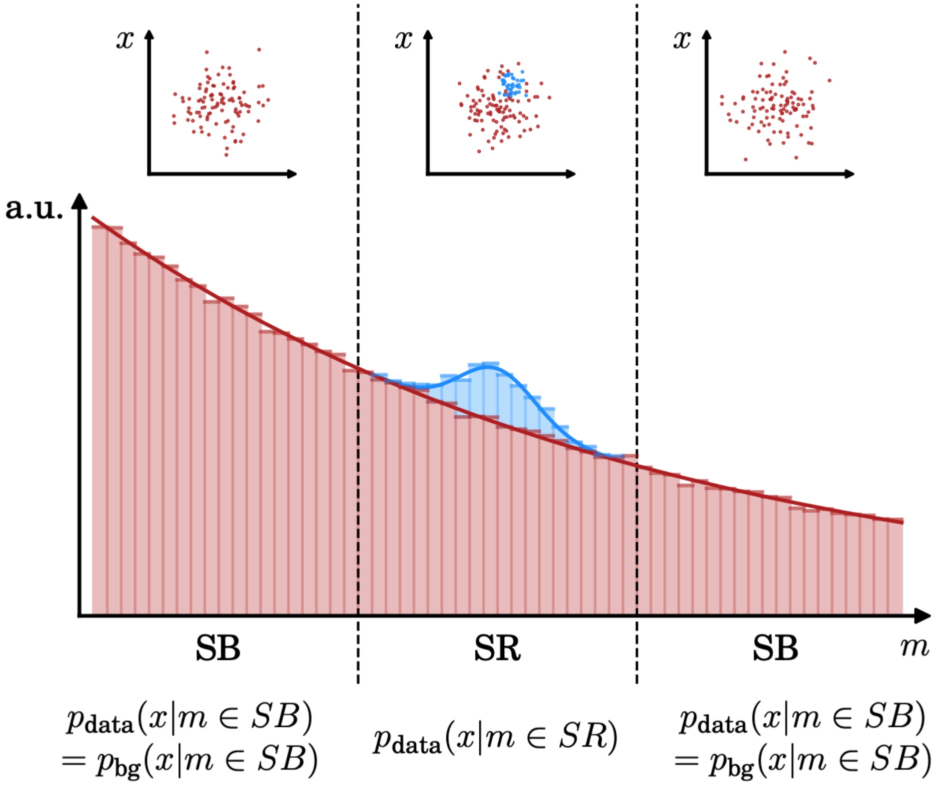
ATLAS non-resonant HH→4b

Machine Learning in Particle Physics

Anomaly Detection: Find events that are out of the ordinary.



[LHC Olympics]



[Review article]

Goals

Machine learning is increasingly a part of how people do science!

Aims for lectures:

- Broad overview of the field/relevant techniques
- Build some intuition for what machine learning is/does
- Give some detailed insight on what goes on “behind the scenes” for neural networks

Audience target:

- Fairly minimal assumptions, but lots of concepts!
- Please feel free to ask questions during lectures!

Outline

Part 1: Overview and Landscape

- What is machine learning?
- Broad ML paradigms
- Neural network introduction
- High level overview of common tools/architectures

Part 2: Nuts and bolts

- How do machines learn?
- How do we evaluate models?

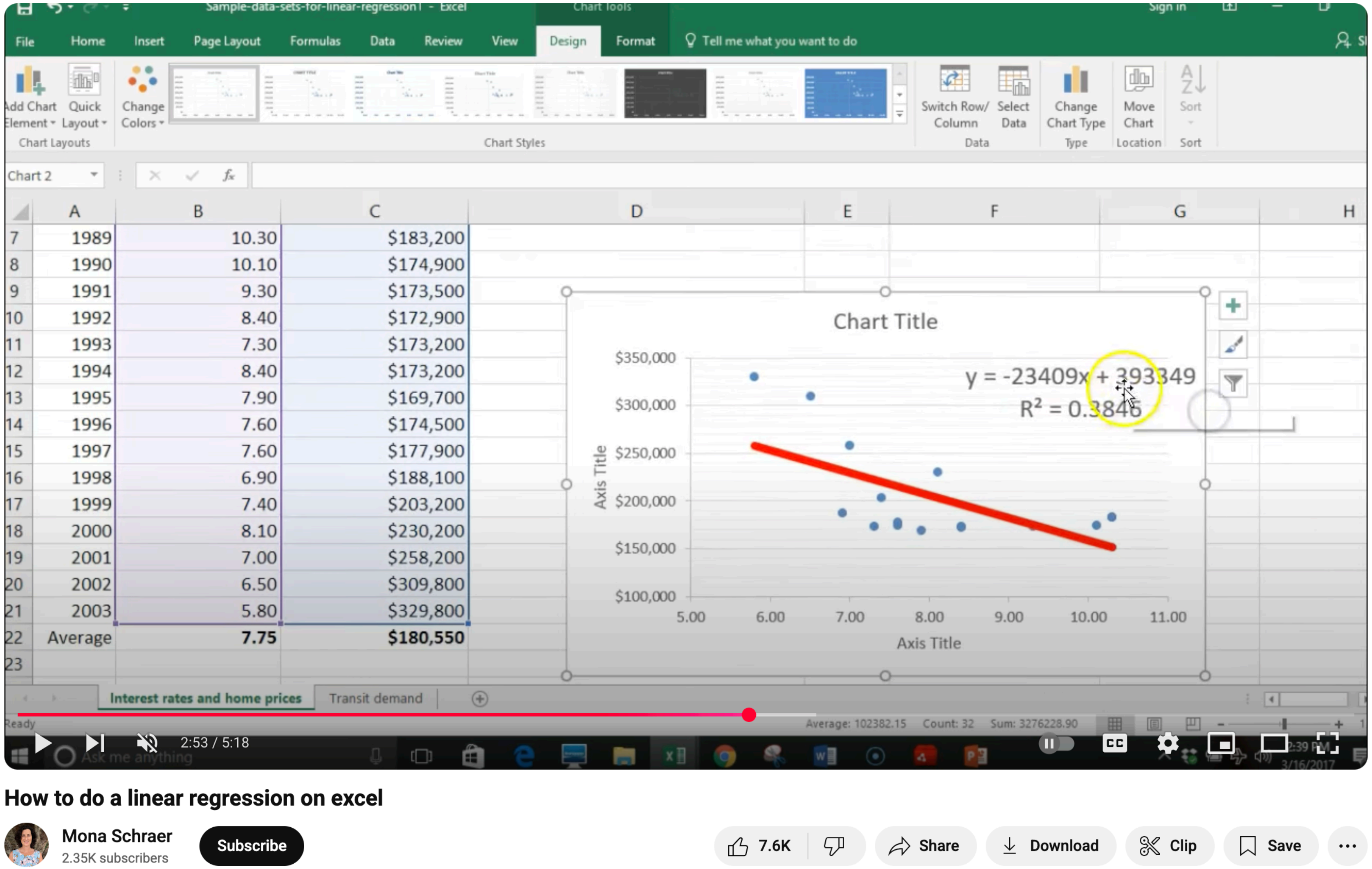
Resources

These lectures are intended as a “I’ve heard those words before” level of understanding!

- Highly recommend further study if things are of interest
  - Focused summer schools:
    - e.g. Machine Learning for Fundamental Physics @ Berkeley Lab
  - Machine learning books/courses:
    - Andrew Ng (free on YouTube)
    - An Introduction to Statistical Learning (pdfs for free)
    - Many resources online!

What is machine learning?

What is Machine Learning?



Source

What is Machine Learning?

ChatGPT ⌄ Share

Write me an ode to machine learning

Ode to Machine Learning

O mind of circuits, born from code and spark,
You sift the chaos, learn to see the dark;
From pixel'd eyes and tensor'd thought you rise,
To draw the truth where human vision lies.

You walk the lands of numbers, wide and deep,
Through loss and gradient, in restless sleep.
Where once we guessed, you measure, weigh, infer—
A silent oracle without a slur.

What is Machine Learning?

$$f$$

Model

What is Machine Learning?

$$f(x, \theta)$$

Model

What is Machine Learning?

Inputs

$$f(x, \theta)$$

Model

What is Machine Learning?

Parameters

Inputs

$$f(x, \theta)$$

Model

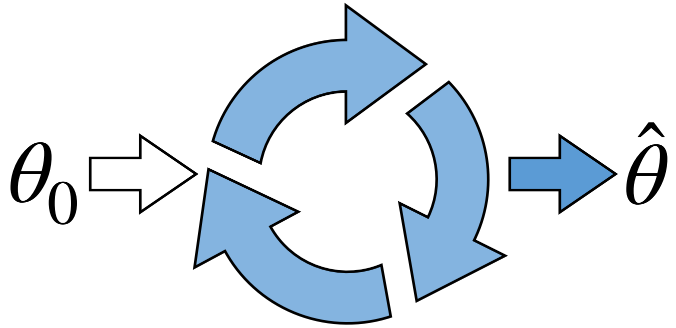
What is Machine Learning?

Inputs

Parameters

$$f(x, \theta)$$

Model



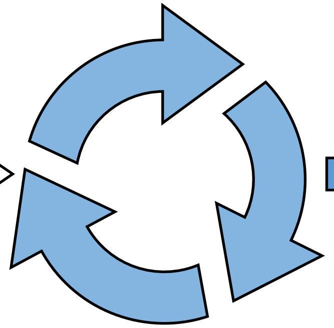
Training

What is Machine Learning?

Inputs → x, Parameters → θ

$$f(x, \theta)$$

Model

θ_0 ⇨  ➡ $\hat{\theta}$

Training

$$\mathscr{L}(f, \mathscr{D})$$

Objective

What is Machine Learning?

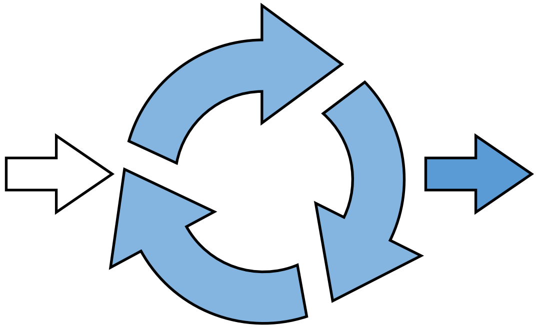
$$\mathscr{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

Inputs

Parameters

Dataset

$$f(x, \theta)$$

θ_0  $\hat{\theta}$

$$\mathscr{L}(f, \mathscr{D})$$

Model

Training

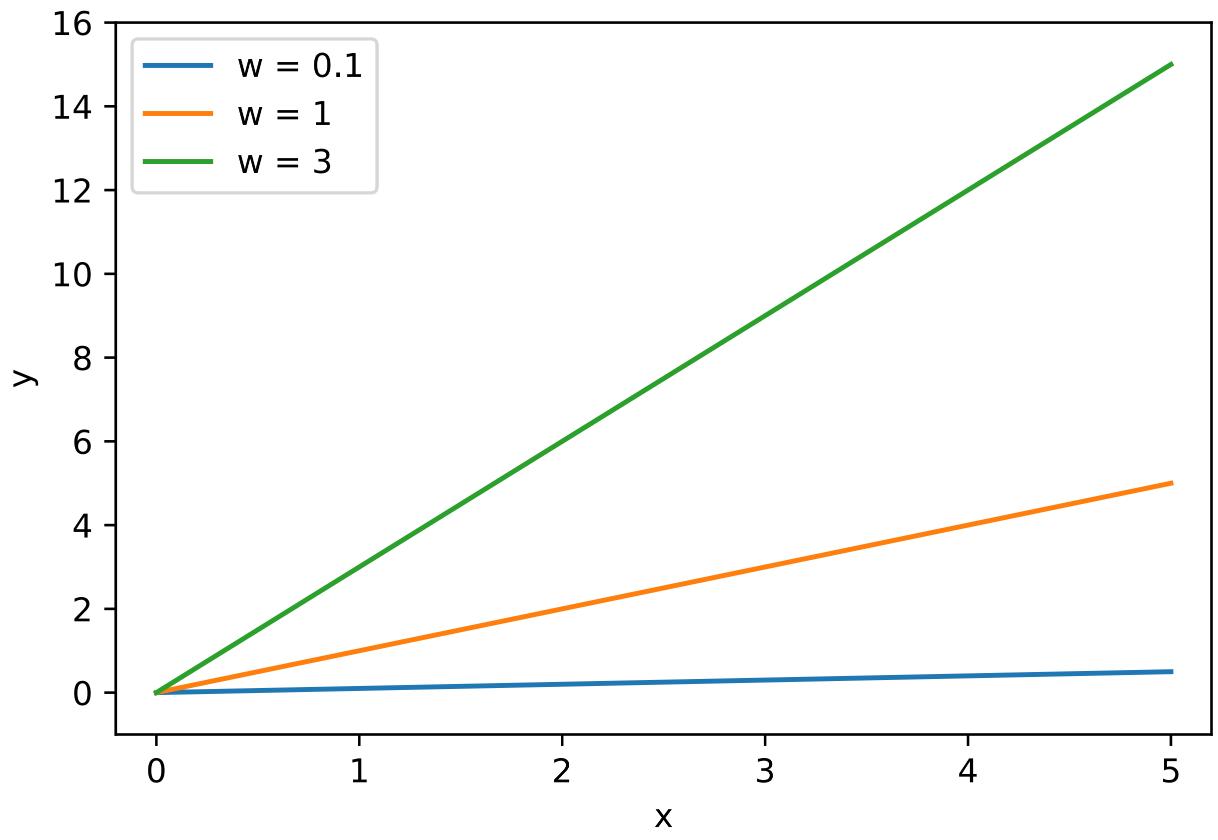
Objective

Training **updates model parameters** to minimize the objective (**“loss function”**) on the **dataset**

Example: Fitting a line

$$f(x, w) = w \cdot x$$

Model



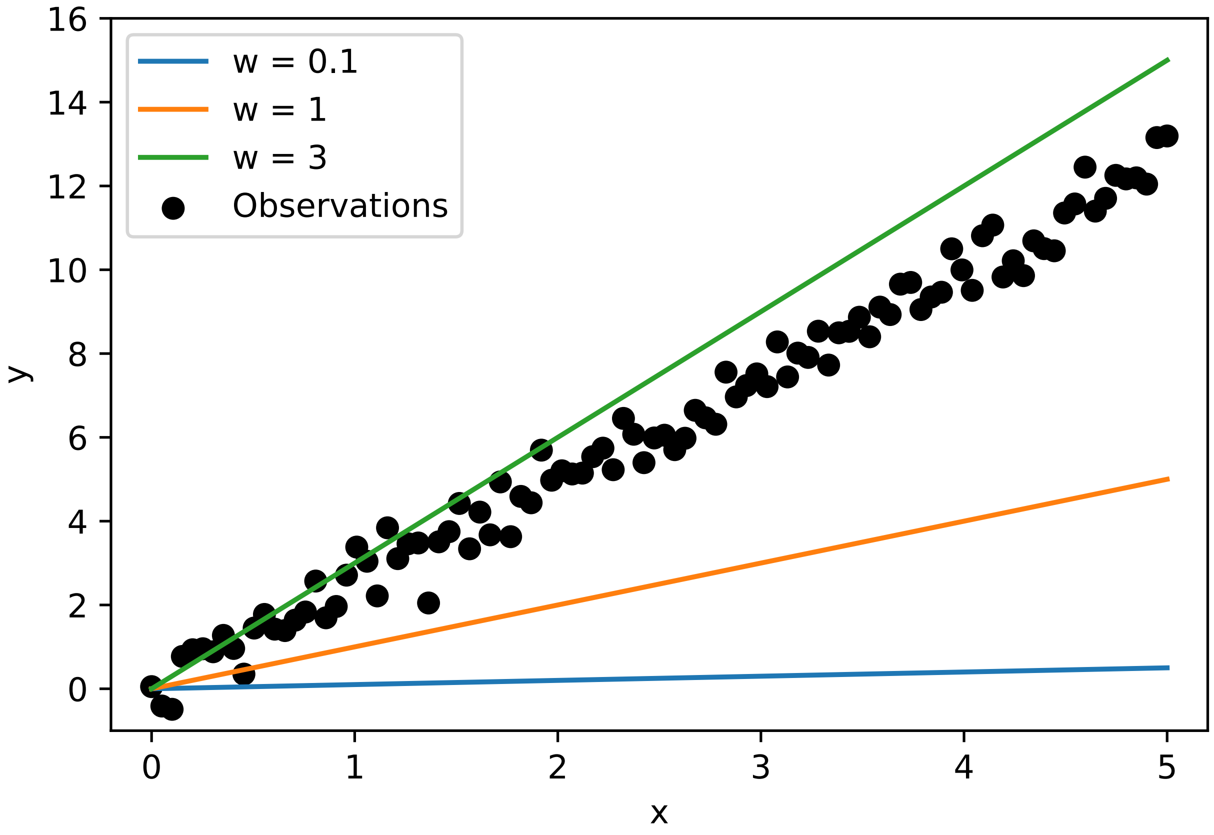
Example: Fitting a line

$$f(x, w) = w \cdot x$$

Model

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$$

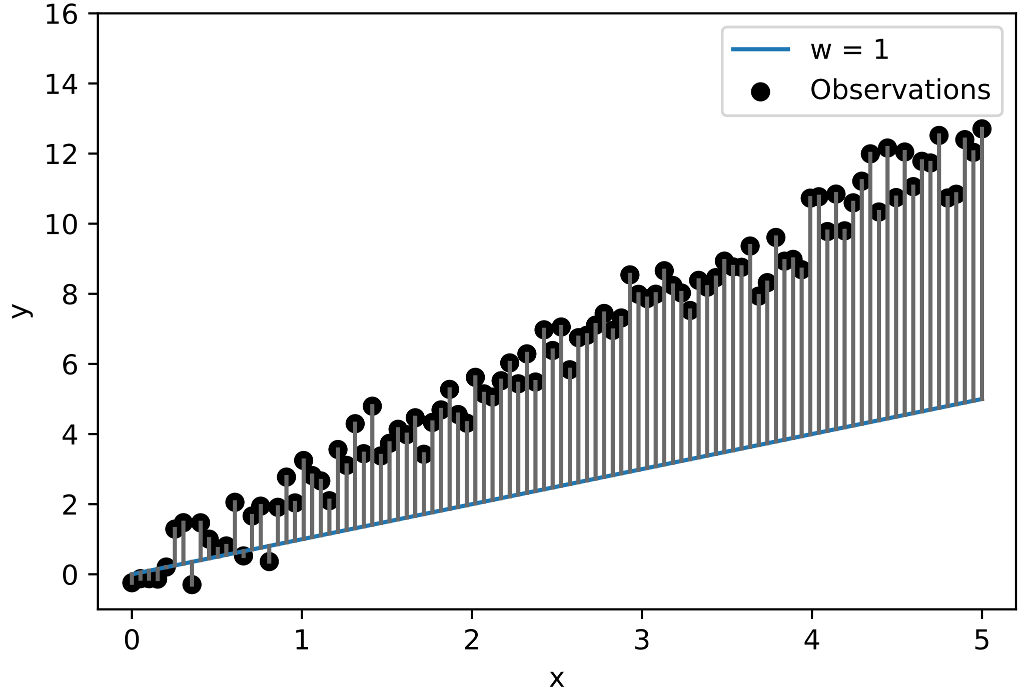
Dataset



Example: Fitting a line

$$f(x, w) = w \cdot x$$

$$\mathscr{D} = \{(x_i, y_i)\}_{i=1}^{n}$$



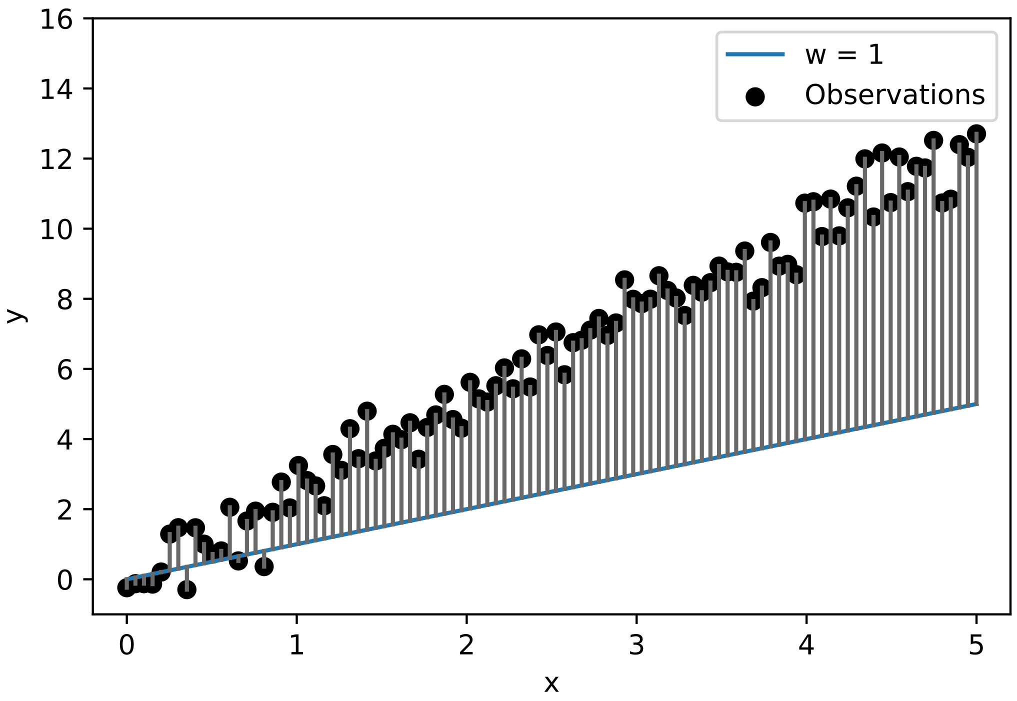
$$\mathscr{L}(f, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (f(x_i, w) - y_i)^2$$

Objective: Make $f(x_i)$ match y_i

("least squares" or "mean squared error")

Example: Fitting a line

$$f(x, w) = w \cdot x$$



$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$$

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i - y_i)^2$$

SLAC

Example: Fitting a line

Training procedure: minimize $\mathscr{L}$

$$\mathscr{L}(w, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i - y_i)^2$$

Example: Fitting a line

Training procedure: minimize $\mathscr{L}$

- Here, can do analytically (take derivative, set equal to 0)

$$\mathscr{L}(w, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i - y_i)^2$$

$$\frac{\partial \mathscr{L}}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} 2x_i \cdot (w \cdot x_i - y_i) = 0$$

Example: Fitting a line

Training procedure: minimize $\mathscr{L}$

- Here, can do analytically (take derivative, set equal to 0)

$$\mathscr{L}(w, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i - y_i)^2$$

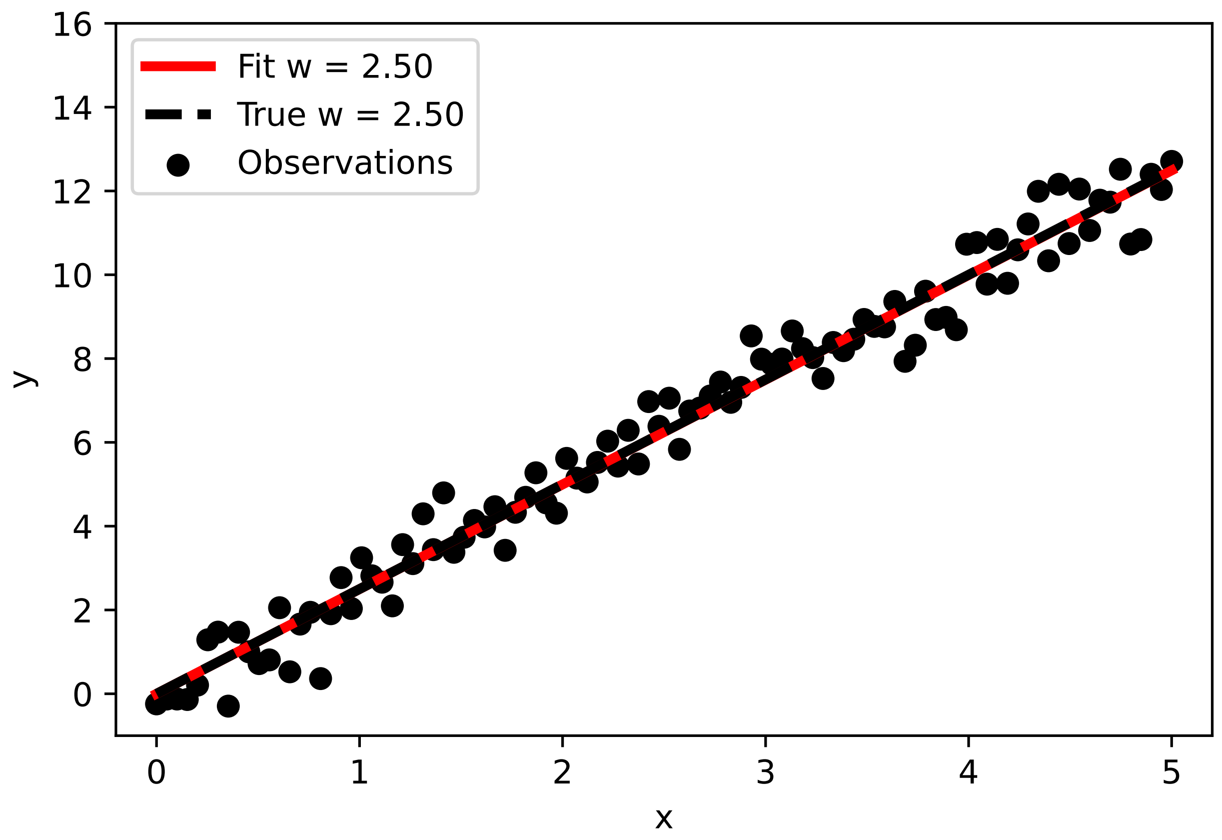
$$\frac{\partial \mathscr{L}}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} 2x_i \cdot (w \cdot x_i - y_i) = 0$$

$$\hat{w} = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sum_{i=1}^{n} x_i^2}$$

Example: Fitting a line

$$f(x, w) = w \cdot x$$

$$\hat{w} = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sum_{i=1}^{n} x_i^2}$$

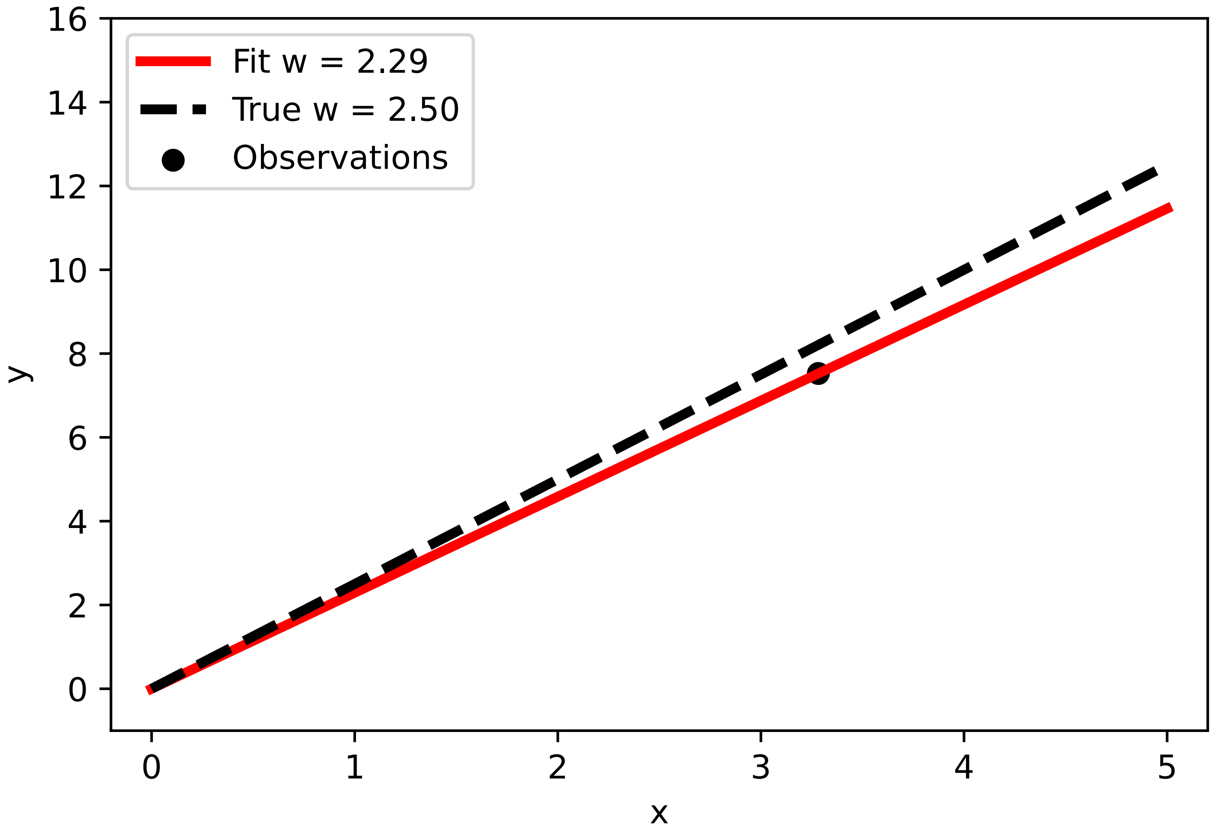


Example: Fitting a line

$$f(x, w) = w \cdot x$$

$$\hat{w} = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sum_{i=1}^{n} x_i^2}$$

Note: dependence on dataset! ("Two points make a line")

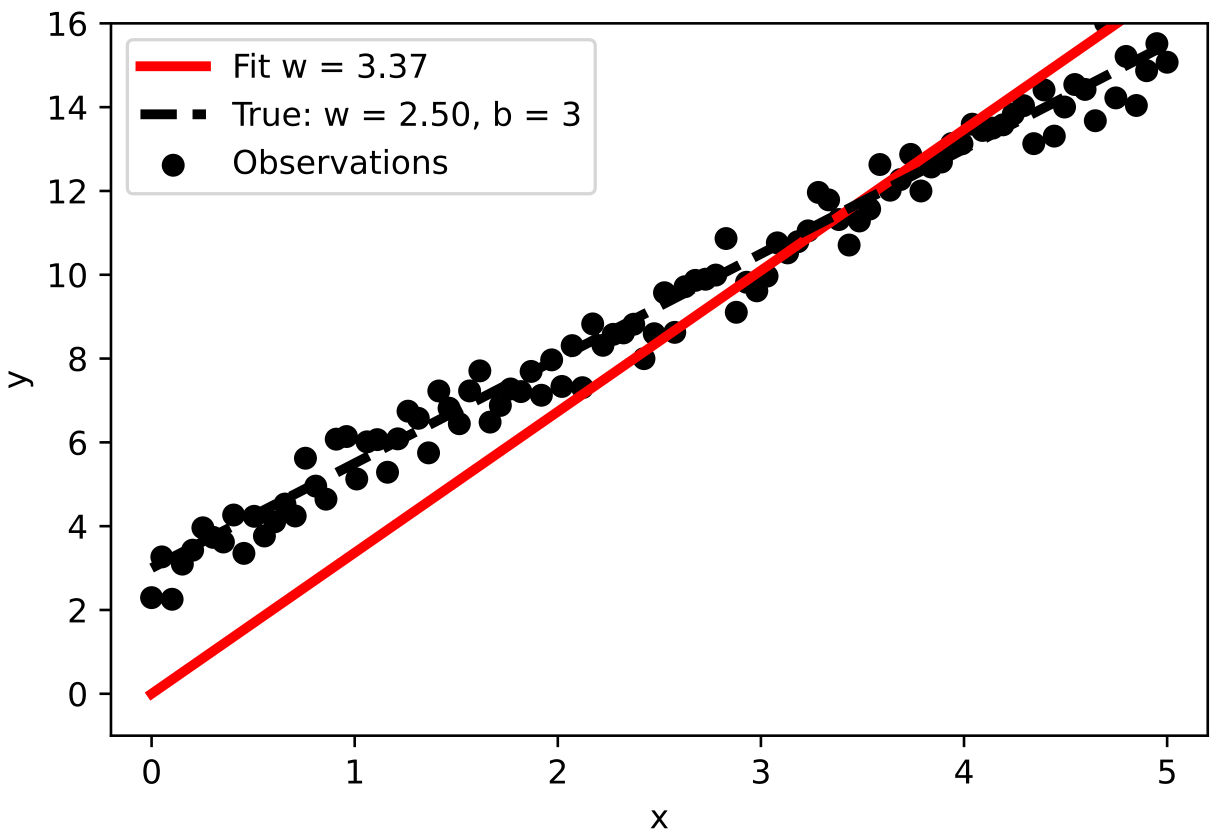


Example: Fitting a line

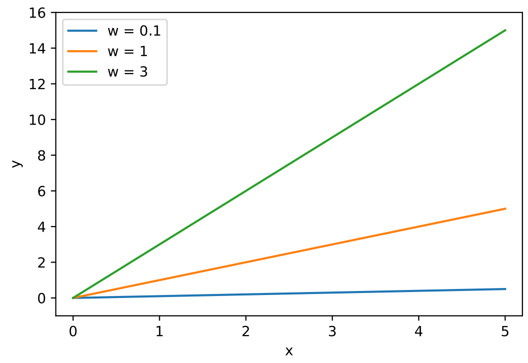
$$f(x, w) = w \cdot x$$

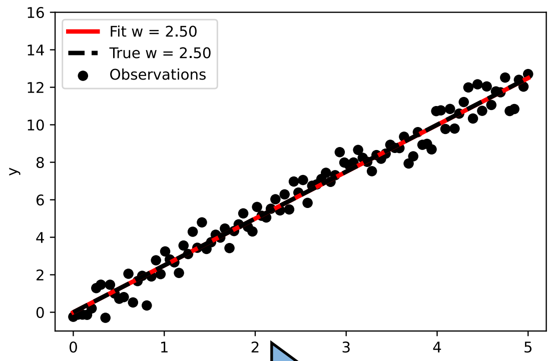
$$f_{true}(x, w) = w \cdot x + b$$

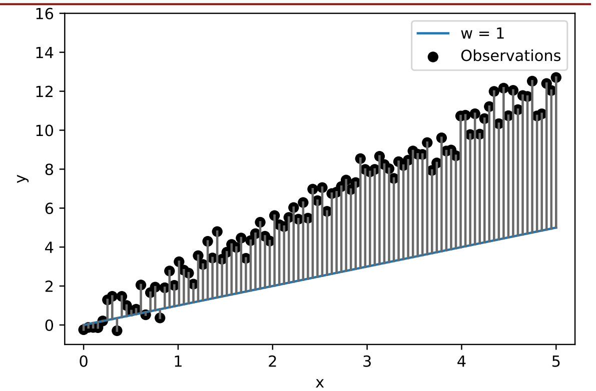
Model needs to match dataset! Need additional parameter b



What is Machine Learning?

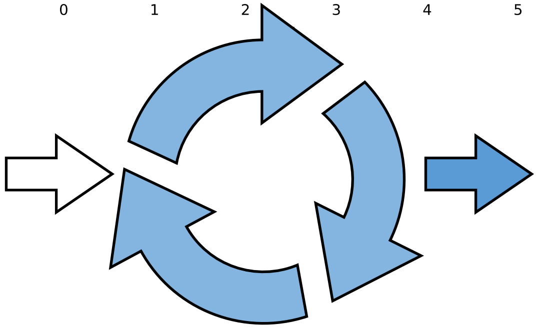






$$f(x, \theta)$$

$$\theta_0$$



$$\hat{\theta}$$

$$\mathscr{L}(f, \mathscr{D})$$

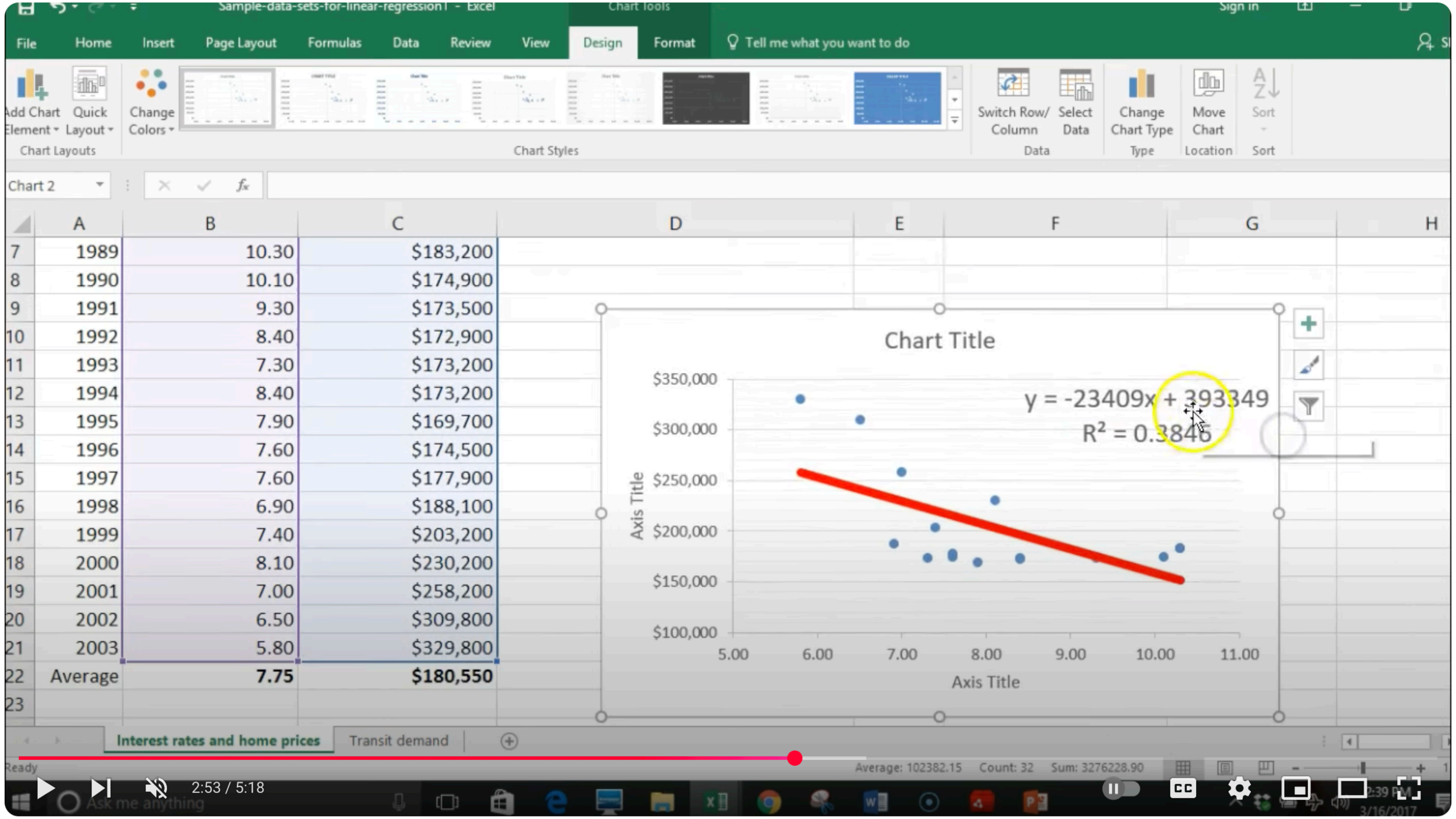
$$f(x, w) = w \cdot x$$

$$\hat{w} = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sum_{i=1}^{n} x_i^2}$$

$$\mathscr{L}(w, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i - y_i)^2$$

What is Machine Learning?

AI is my passion



How to do a linear regression on excel

Mona Schraer
2.35K subscribers
Subscribe

7.6K | Share | Download | Clip | Save

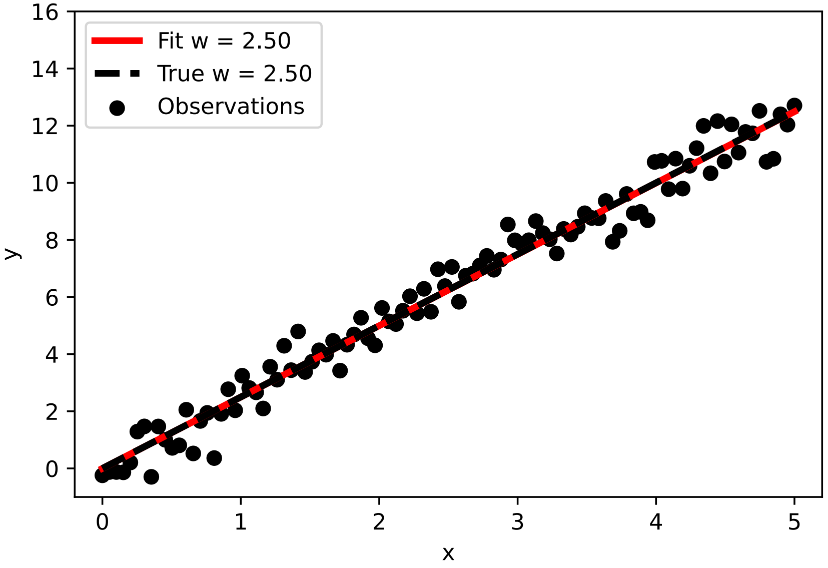
Source

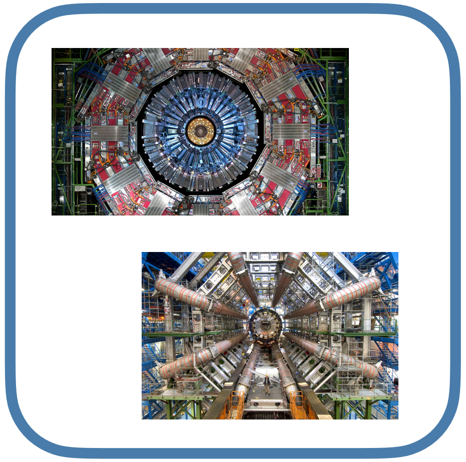
ML Paradigms

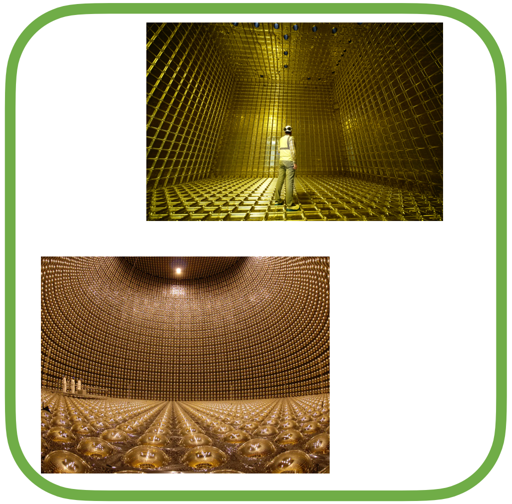
Machine Learning Paradigms

What data do we have?

- **Supervised learning:** map from input x to output y given data with known **labels** (x_i, y_i)
  - **Regression:** continuous valued y
  - **Classification:** discrete/categorical valued y
- **Unsupervised learning:** analyze **unlabeled** data to learn patterns or structure





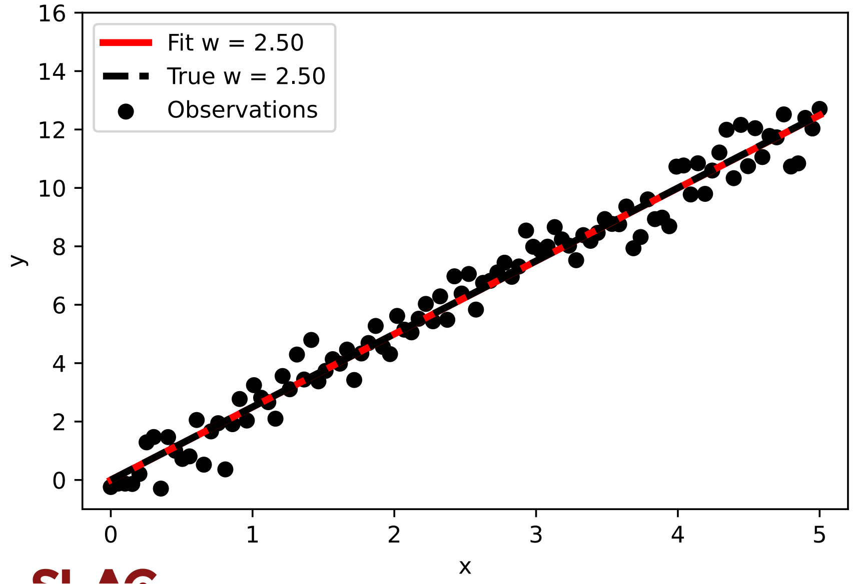


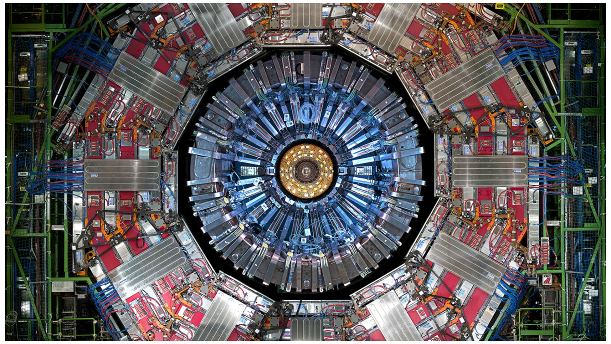
ML Paradigms: Supervised Learning

$$\mathcal{L}(f, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} (f(x_i) - y_i)^2 \qquad \mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$$

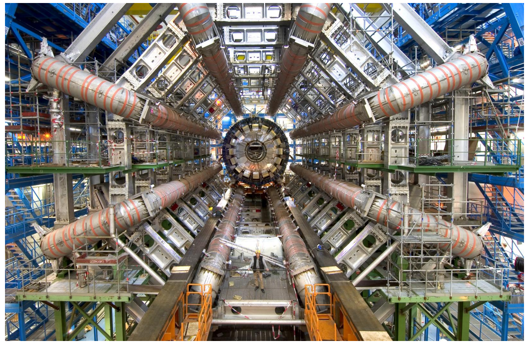
Model Prediction

Known y value ("**Label**")



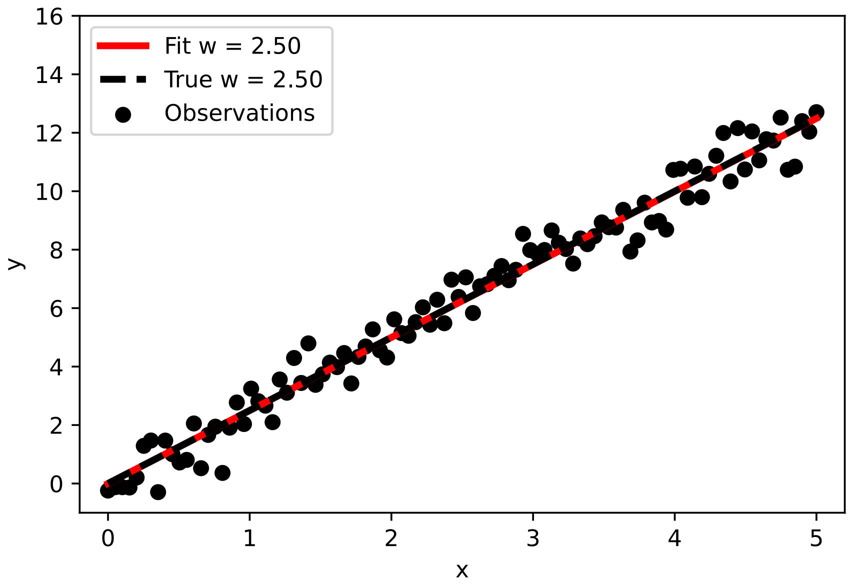


CMS



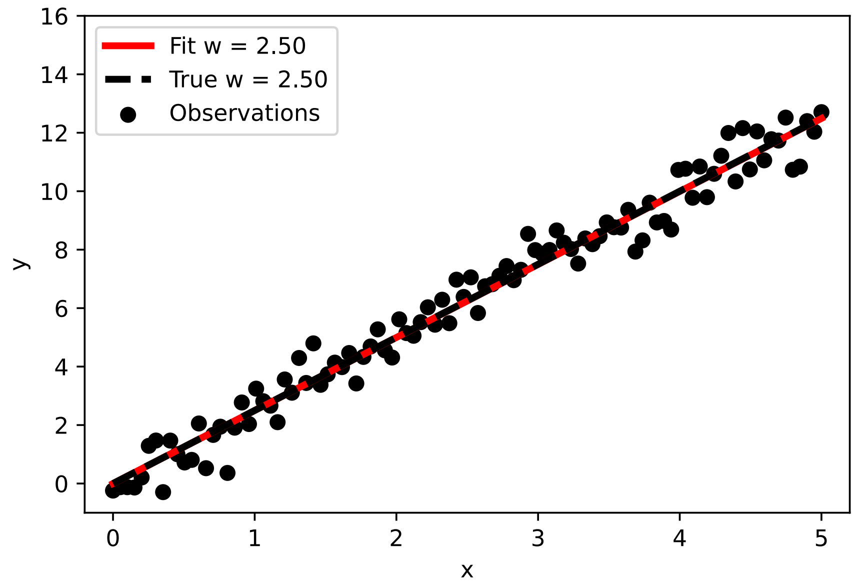
ATLAS

ML Paradigms: Regression



Regression: Predict continuous numerical values (e.g. fit a line)

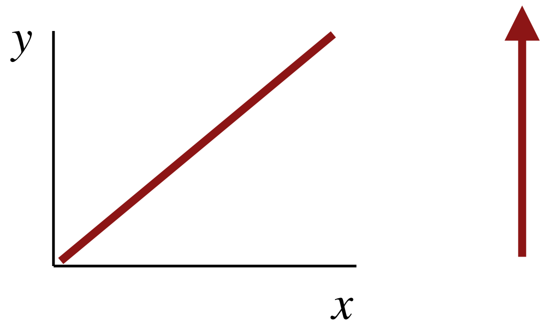
ML Paradigms: Regression



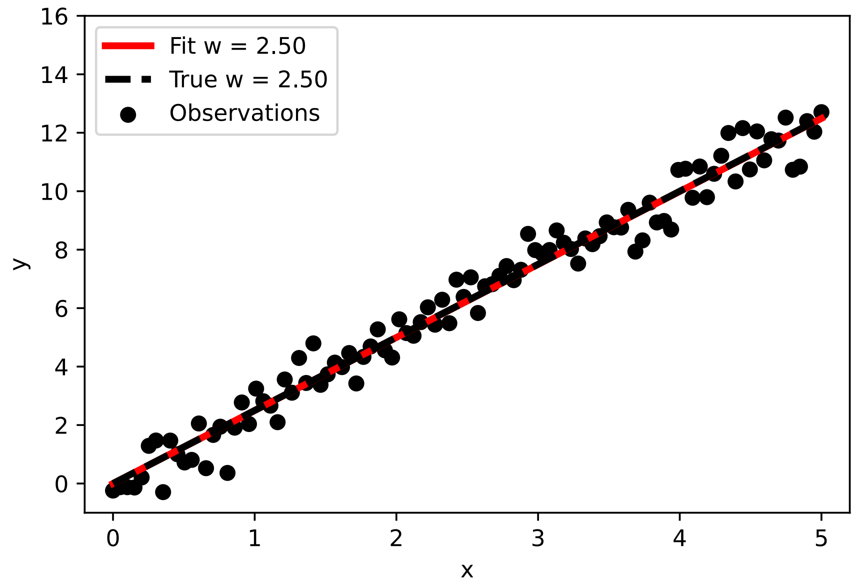
Regression: Predict continuous numerical values (e.g. fit a line)

Model: $f(x) =$ Continuous value

$$f(x) = w \cdot x$$



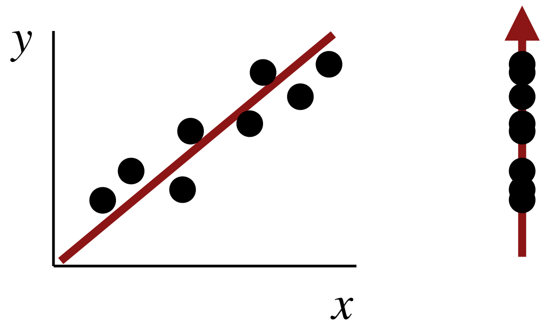
ML Paradigms: Regression



Regression: Predict continuous numerical values (e.g. fit a line)

Model: $f(x) =$ Continuous value

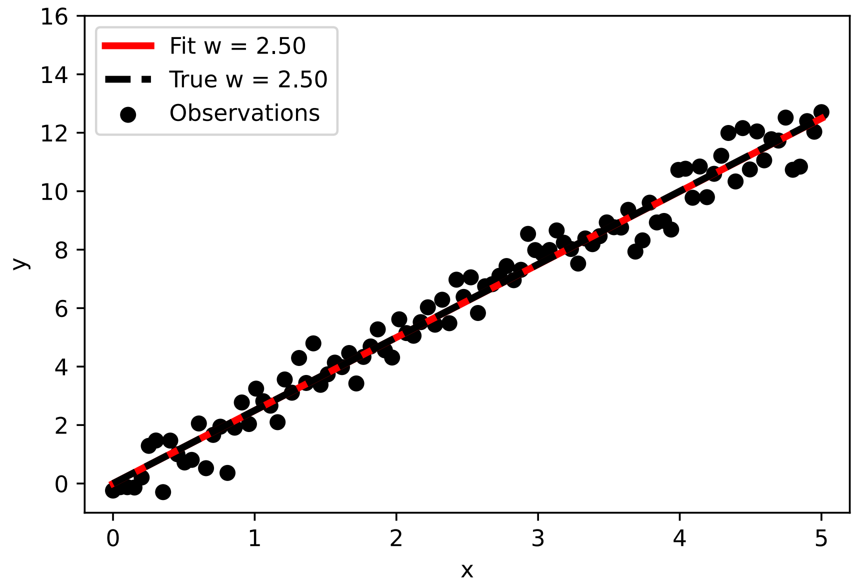
$$f(x) = w \cdot x$$

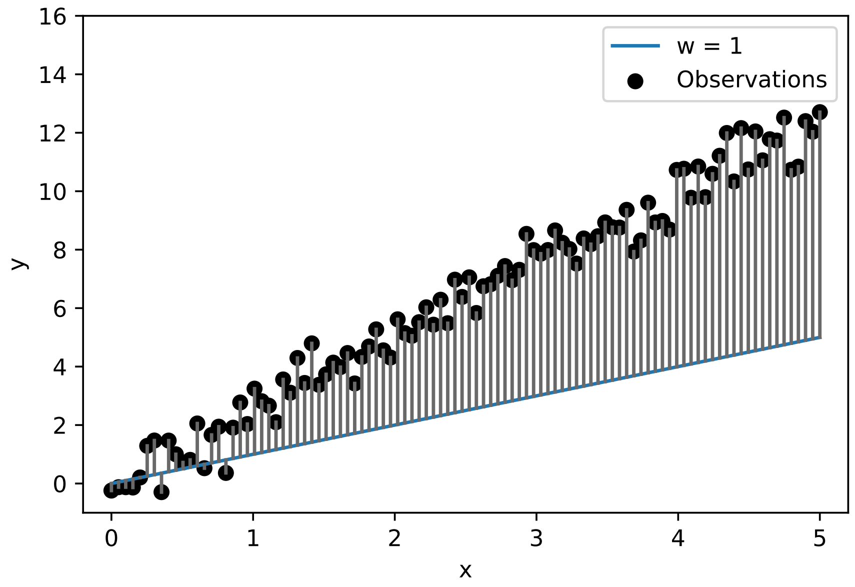


Data: Label $y =$ Continuous value

ML Paradigms: Regression

Training: Minimize distance between $f(x)$ and y



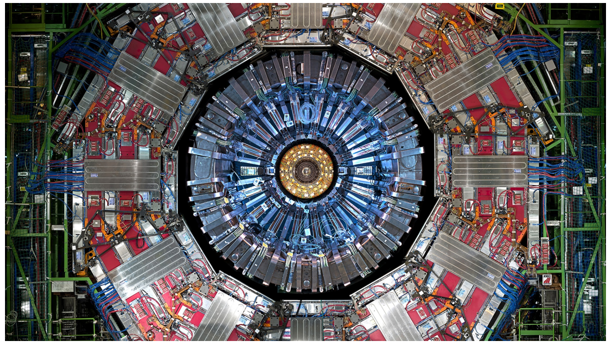


Regression: Predict continuous numerical values (e.g. fit a line)

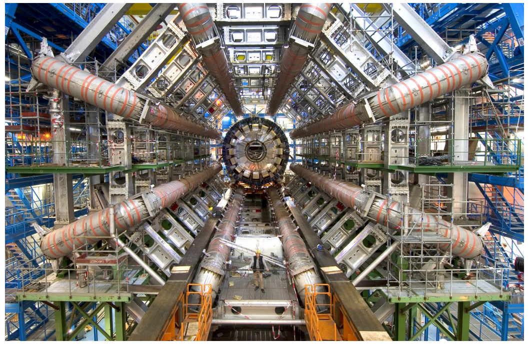
$$\mathscr{L}(f, \mathscr{D}) = \frac{1}{n} \sum_{i=1}^{n} (f(x_i) - y_i)^2$$

e.g. mean squared error, mean absolute error

ML Paradigms: Classification



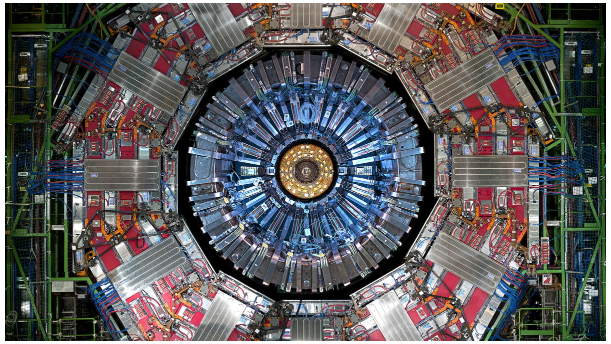
CMS



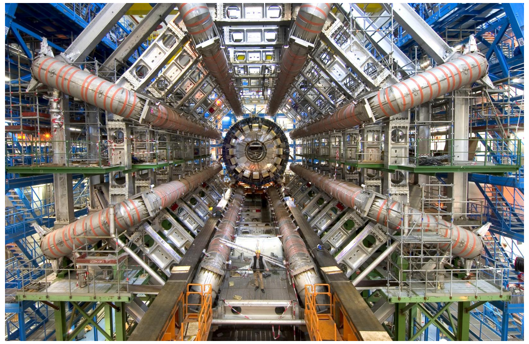
ATLAS

Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

ML Paradigms: Classification



CMS

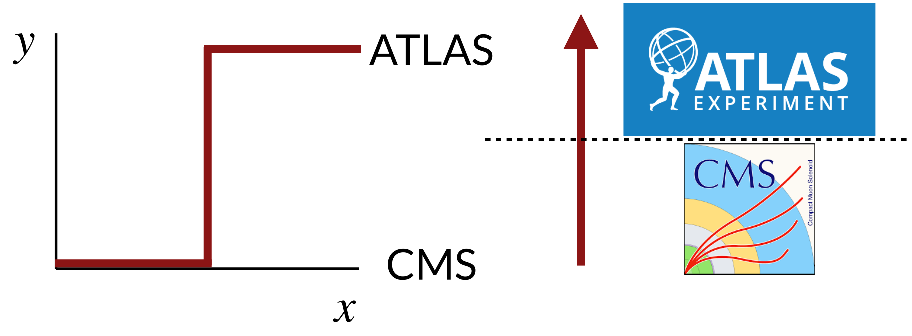


ATLAS

Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

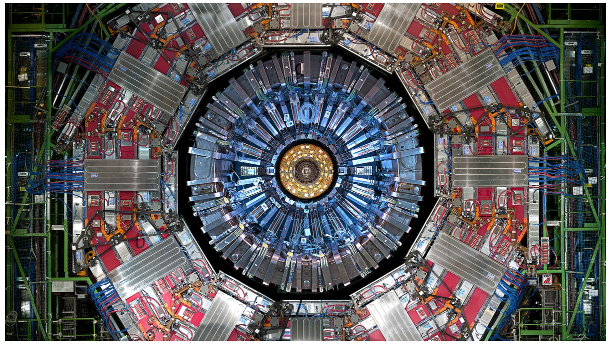
Model: $f(x) =$ Discrete value (class prediction)

$f(x) =$ ATLAS if $x > 0$, else CMS

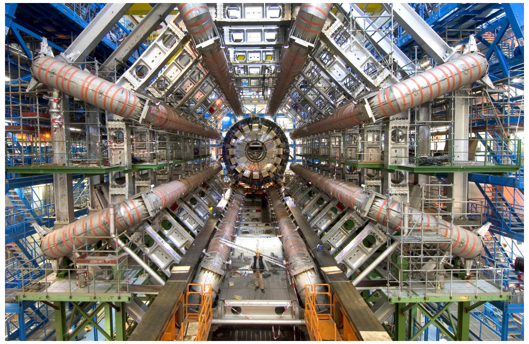


e.g. $x =$ number of toroid magnets

ML Paradigms: Classification



CMS

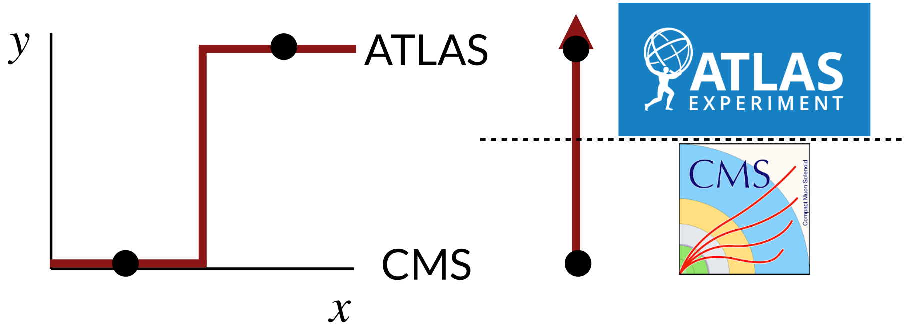


ATLAS

Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

Model: $f(x) =$ Discrete value (class prediction)

$f(x) =$ ATLAS if $x > 0$, else CMS

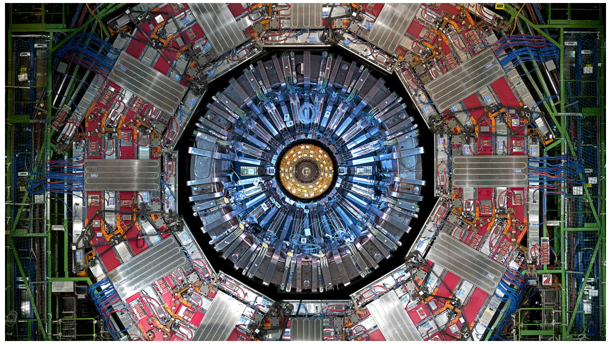


Data: Label $y =$ Discrete value (class prediction)

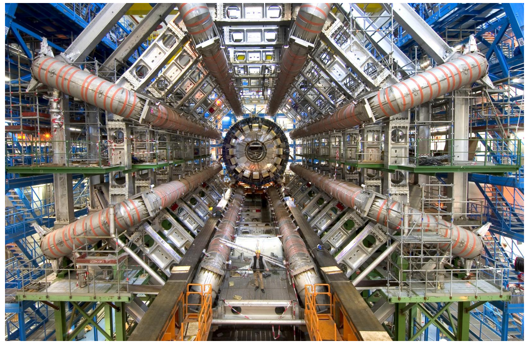
ML Paradigms: Classification

Model: $f(x) =$ ~~Discrete value~~
Class probability

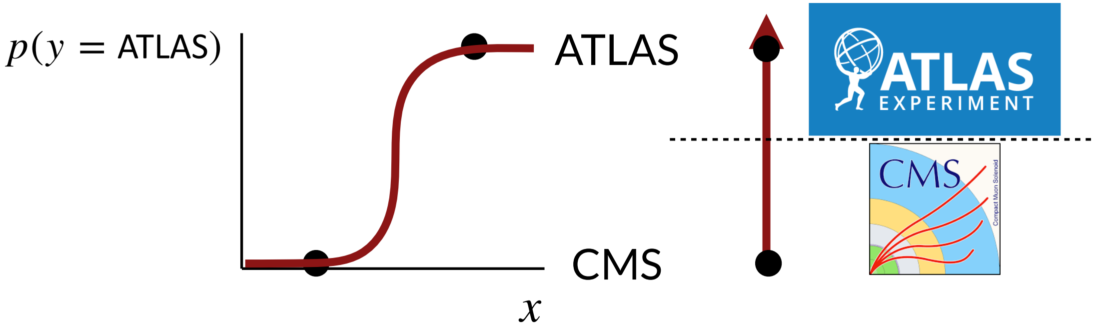
$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$



CMS



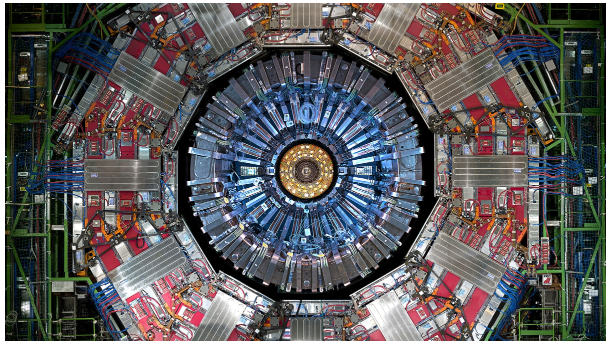
ATLAS



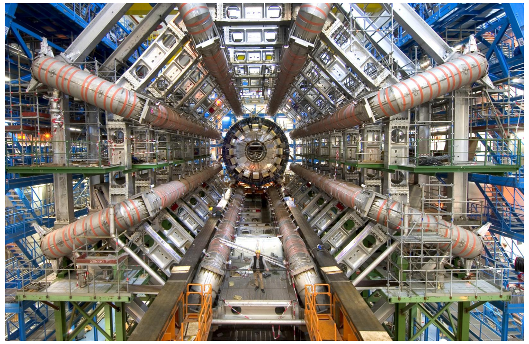
Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

Data: Label $y =$ Discrete value (class prediction)

ML Paradigms: Classification



CMS

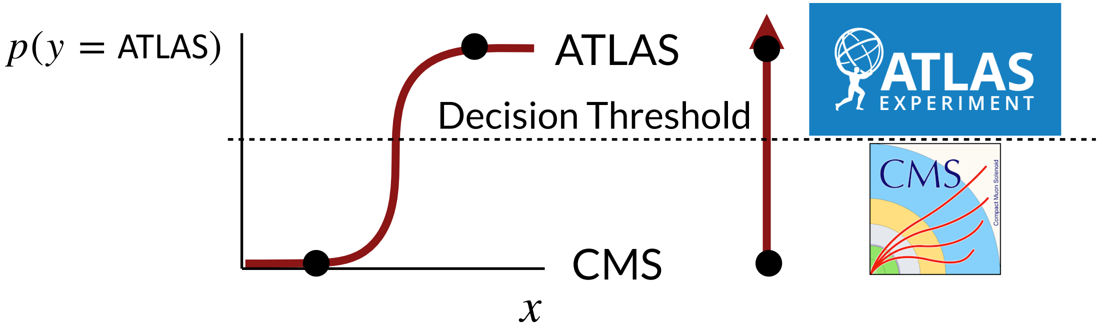


ATLAS

Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

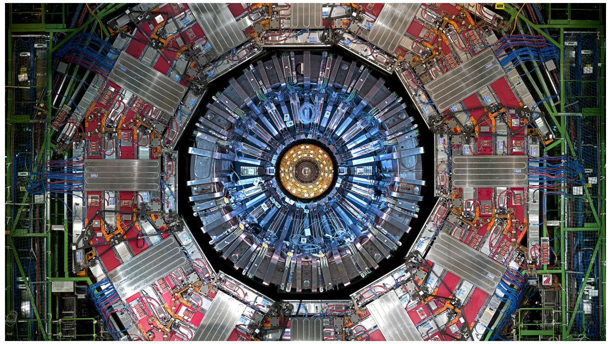
Model: $f(x) =$ ~~Discrete value~~
Class probability

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

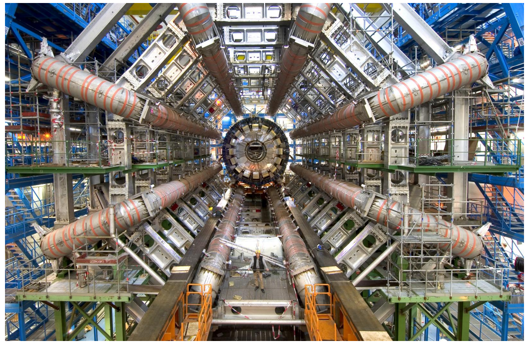


Data: Label $y =$ Discrete value (class prediction)

ML Paradigms: Classification



CMS

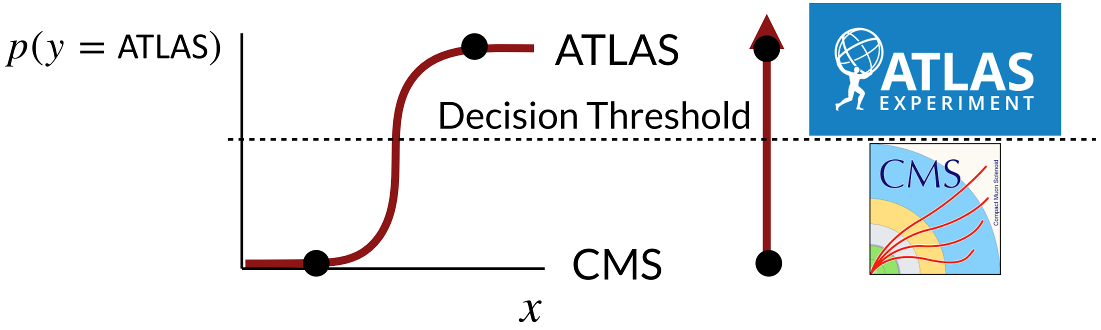


ATLAS

Classification: Predict discrete/ categorical variables (e.g. which experiment is this image)

Model: $f(x) =$ ~~Discrete value~~
Class probability

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$



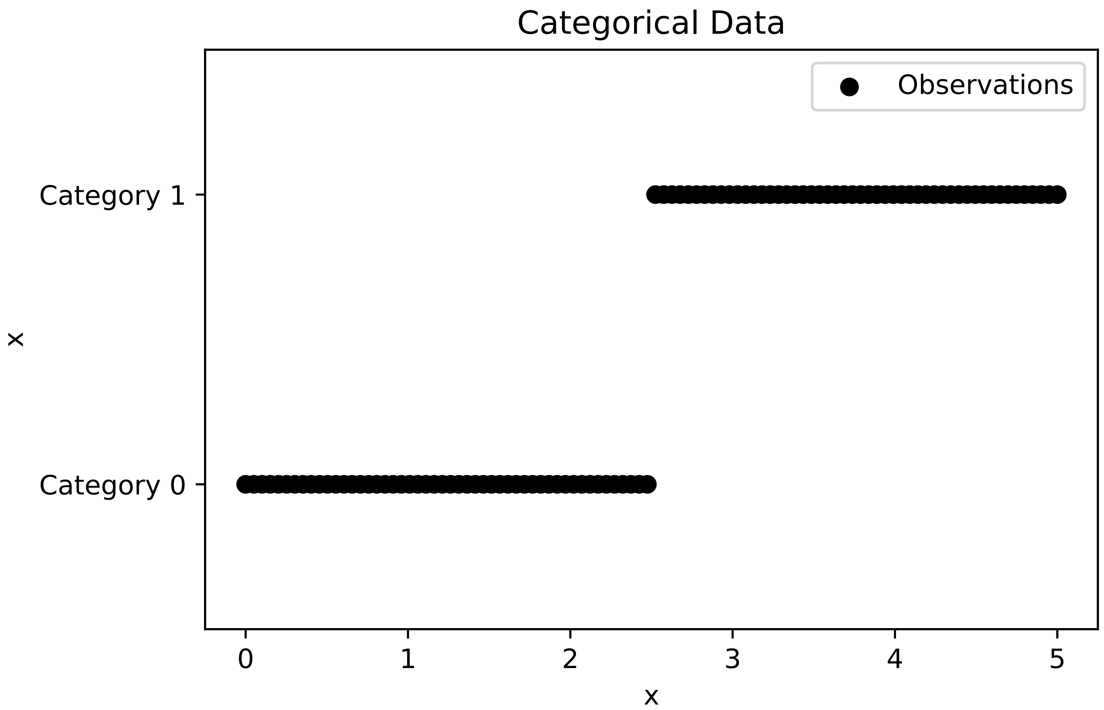
Data: Label $y =$ Discrete value (class prediction)

Method: **logistic regression**

ML Paradigms: Classification

Model: $f(x) =$ **Class probability**

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$



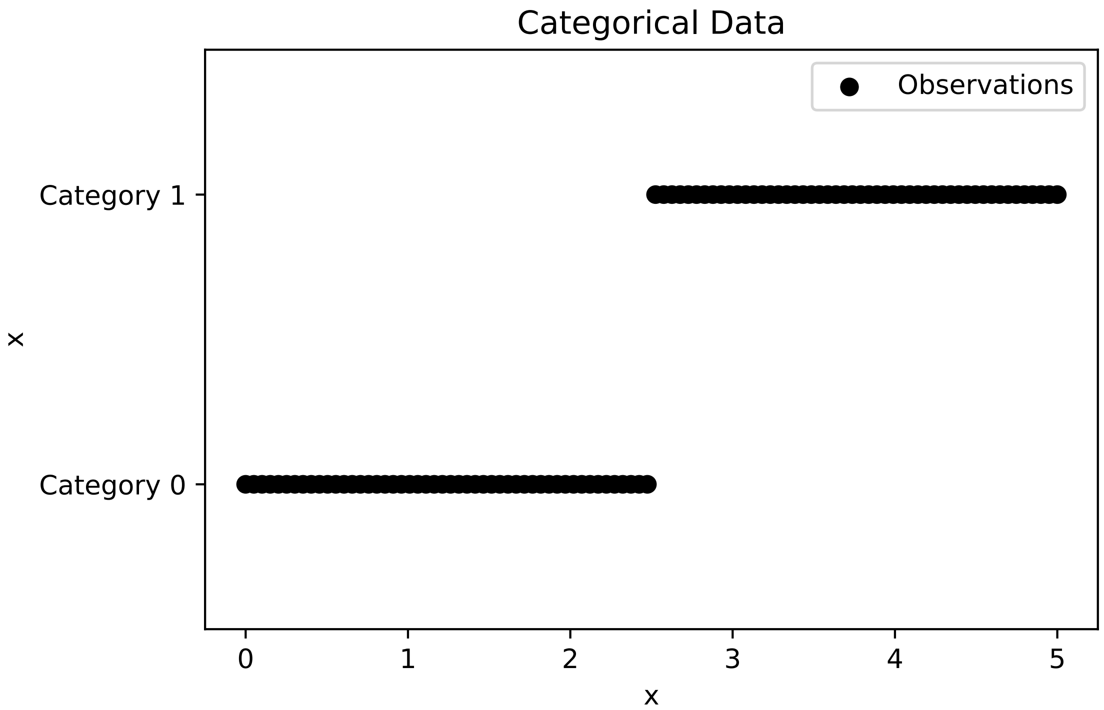
ML Paradigms: Classification

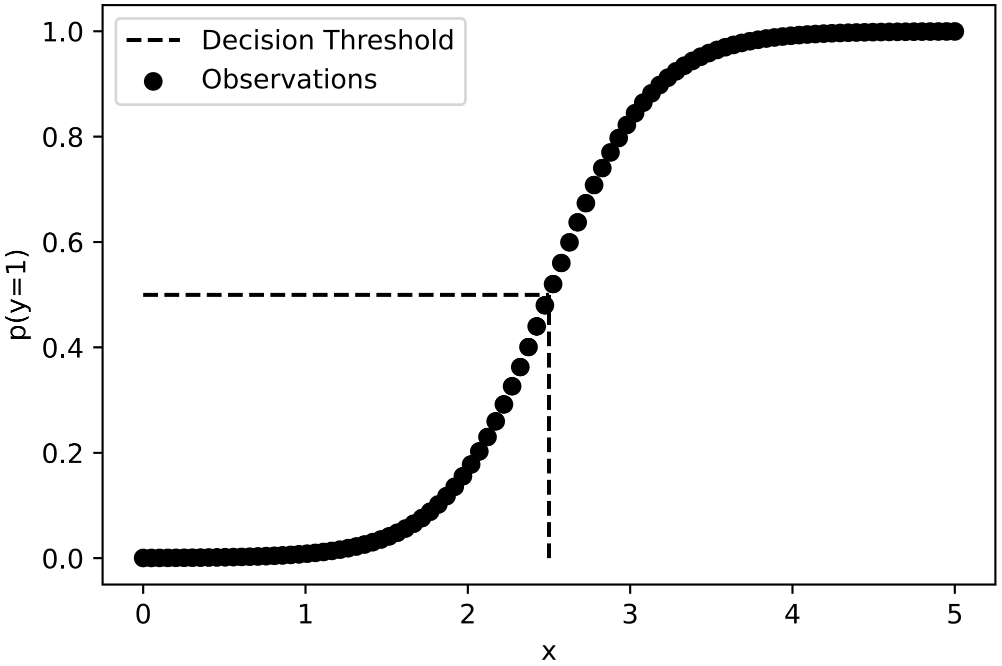
Model: $f(x) =$ **Class probability**

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

Fitted params:

$$\sigma = 3.2, b = -8.0$$

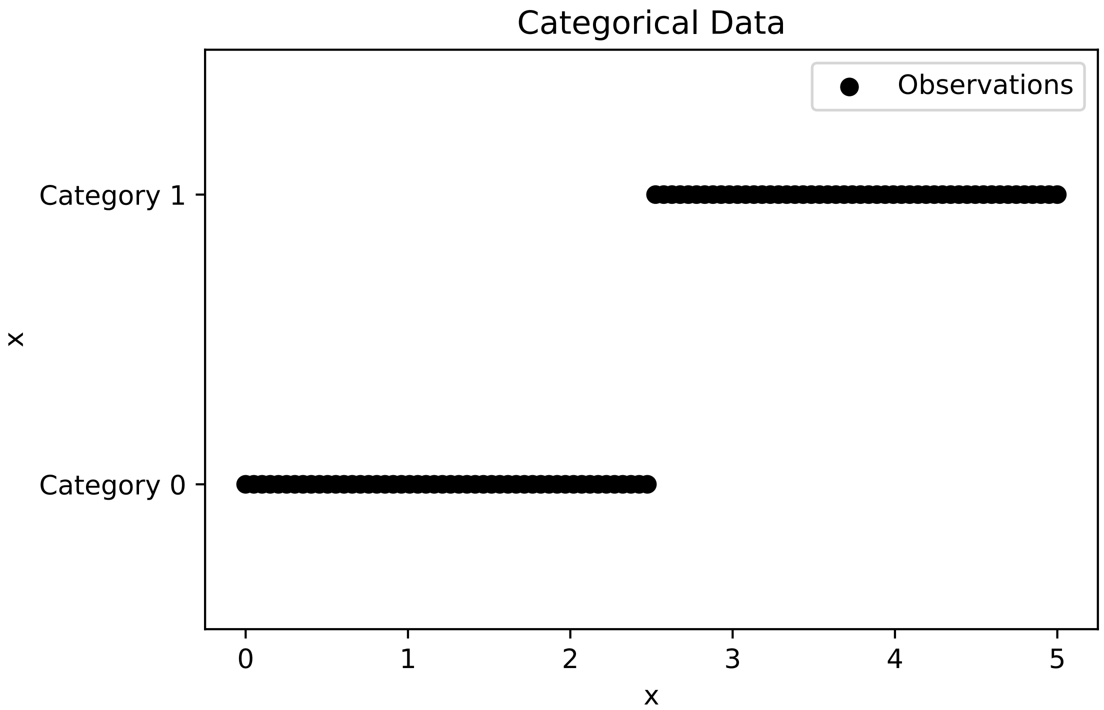


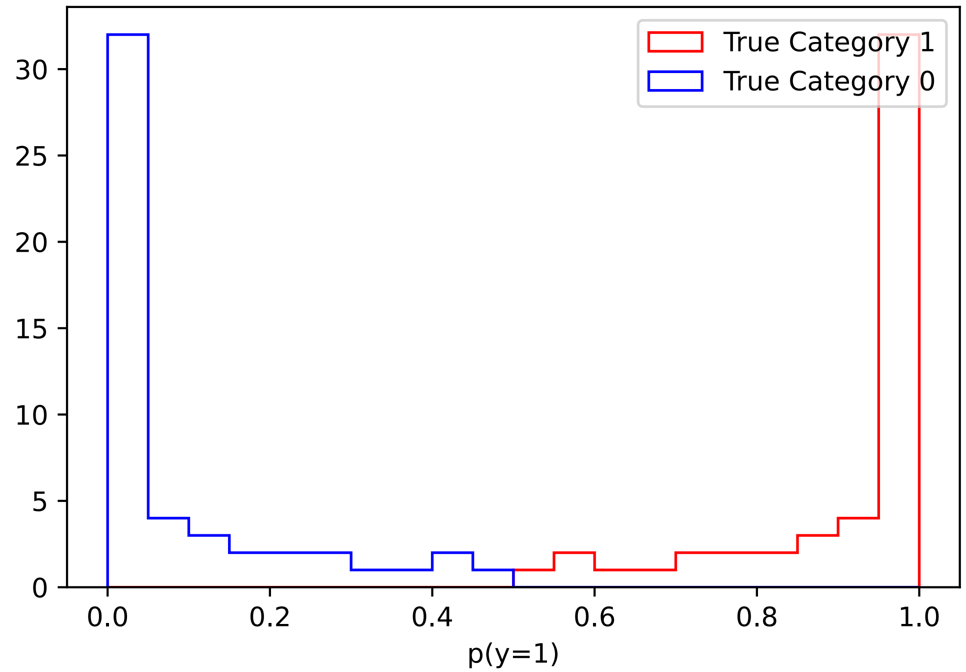


ML Paradigms: Classification

Model: $f(x) =$ **Class probability**

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

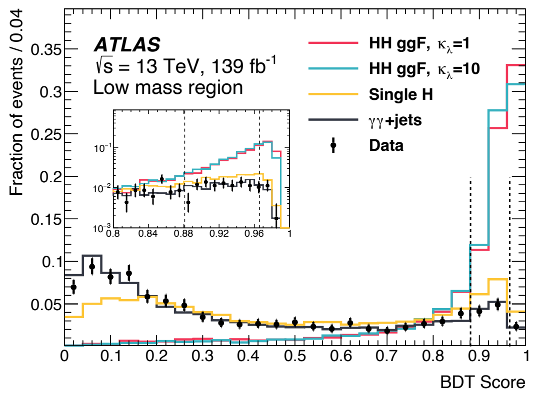




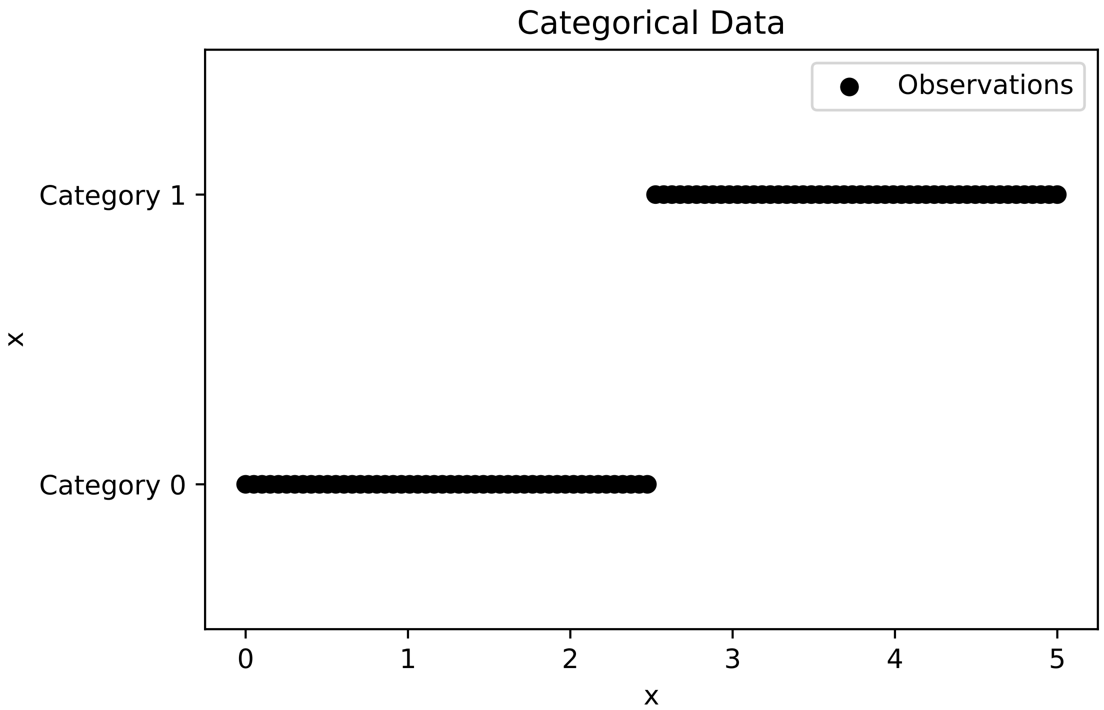
ML Paradigms: Classification

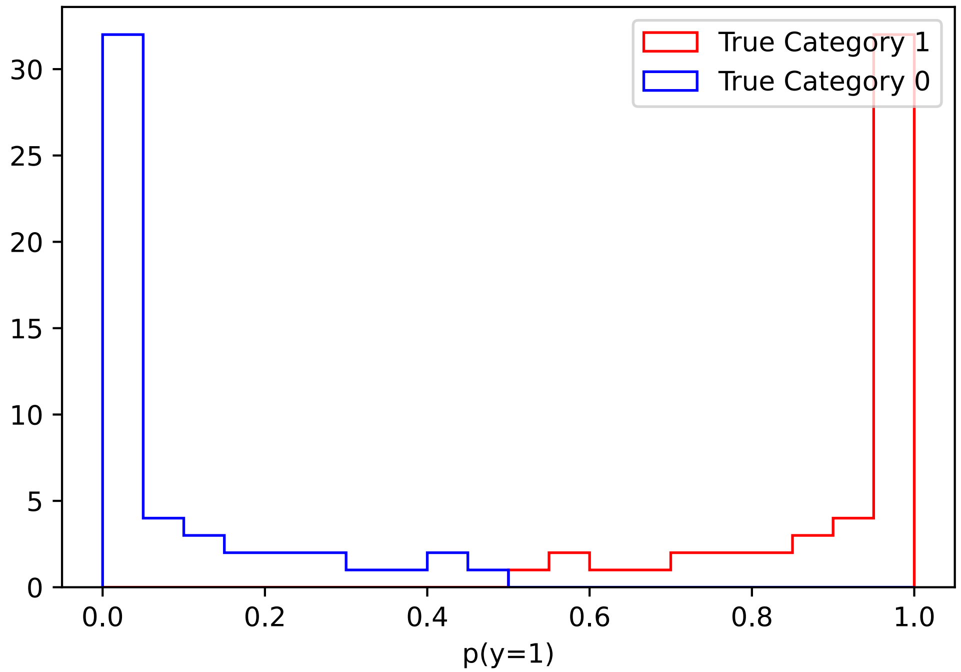
Model: $f(x) =$ **Class probability**

$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$



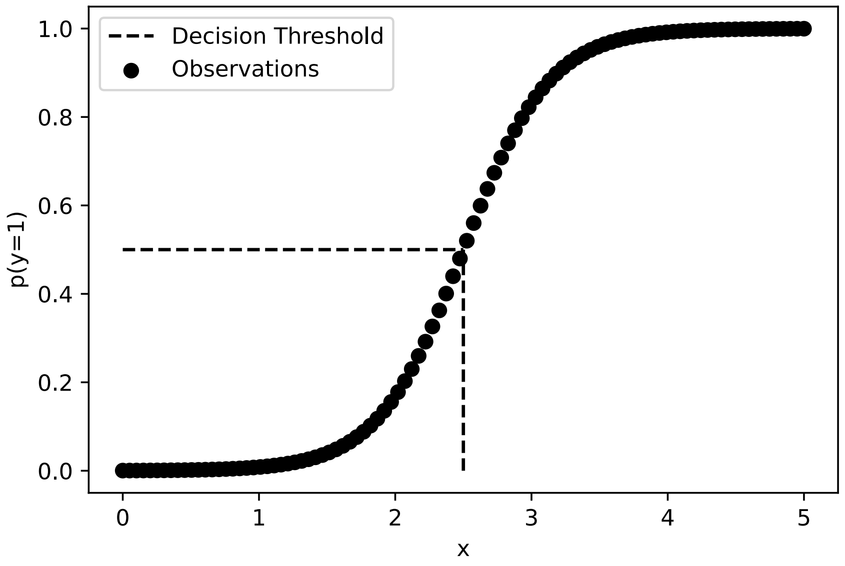
[ATLAS bbyy]





ML Paradigms: Classification

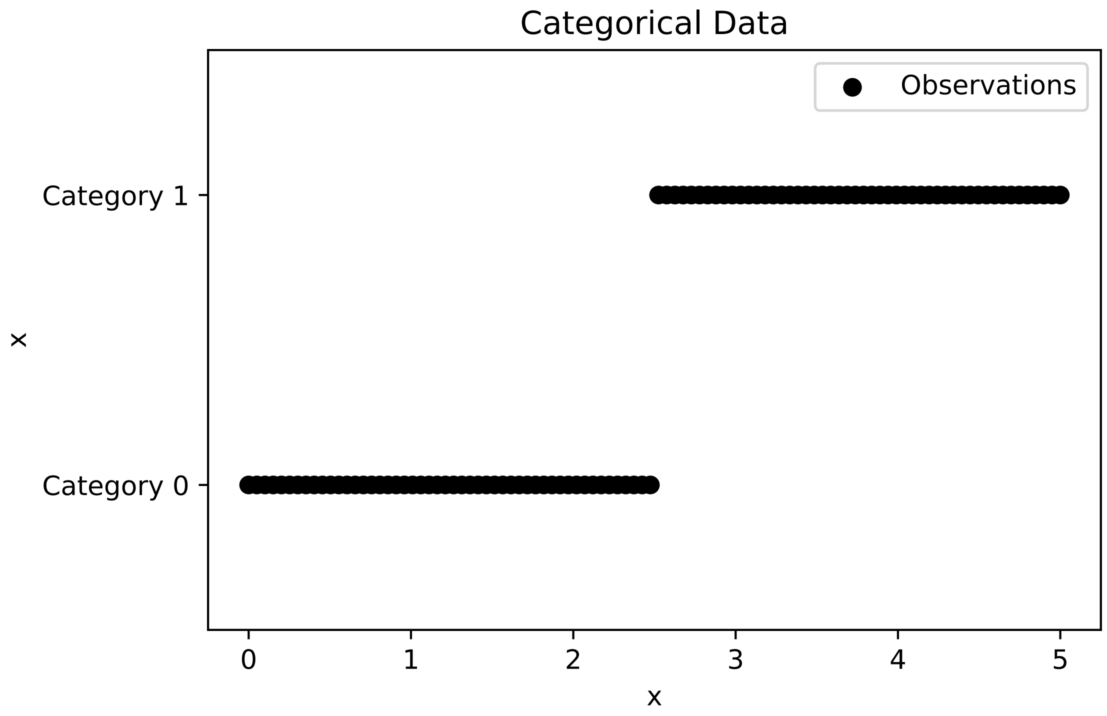
Training: Model gives a probability estimate $p(y | \theta, x)$ $(\theta = \{w, b\})$



For measurements $\{(x_i, y_i)\}_{i=1}^{n}$, we can then evaluate a **likelihood**

$$L(\theta) = \prod_{i=1}^{n} p(y_i | x_i, \theta)$$

ML Paradigms: Classification



For two categories ($y \in \{0,1\}$), think of this as an uneven coin-flip (**Bernoulli distribution**)

Model:

$$\hat{y}_i = p(y_i = 1 \,|\, x_i, \theta)$$

Two possible categories:

$$p(y_i = 0 \,|\, x_i, \theta) = 1 - \hat{y}_i$$

Combined:

$$p(y_i \,|\, x_i, \theta) = \hat{y}_i^{\,y_i} \cdot (1 - \hat{y}_i)^{1-y_i}$$

ML Paradigms: Classification

For n observations:

$$L(\theta) = \prod_{i=1}^{n} \hat{y}_i^{y_i} \cdot (1 - \hat{y}_i)^{1-y_i}$$

Maximizing likelihood <=> Minimizing negative log-likelihood (NLL):

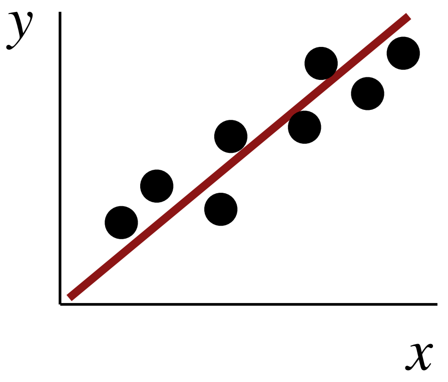
$$\mathscr{L}(\theta, \mathscr{D}) = -\sum_{i=1}^{n} \left(y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i) \right)$$

Binary cross-entropy loss

Supervised Learning: Summary

Supervised learning: map from input x to output y given data with known **labels** (x_i, y_i)

Regression (continuous, numeric)

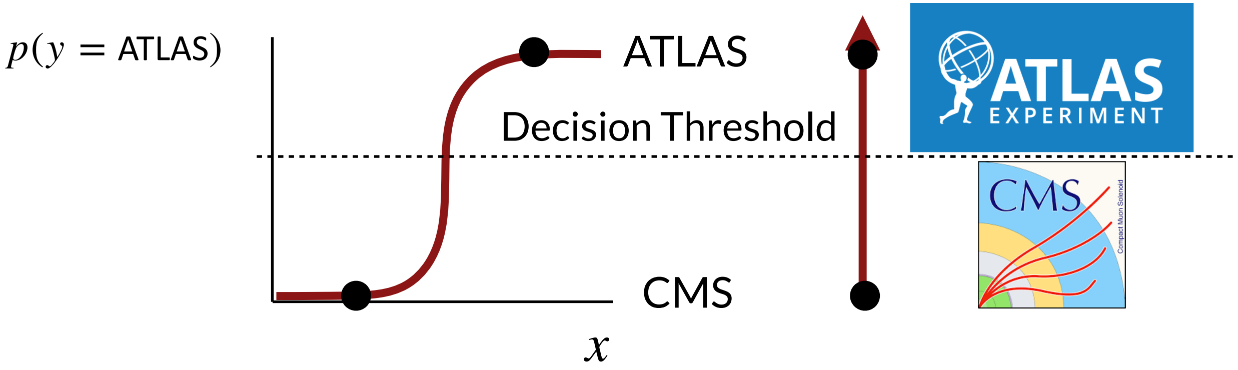


$$f(x) = w \cdot x + b$$

e.g. linear regression

Loss: Mean squared error

Classification (categorical, discrete)



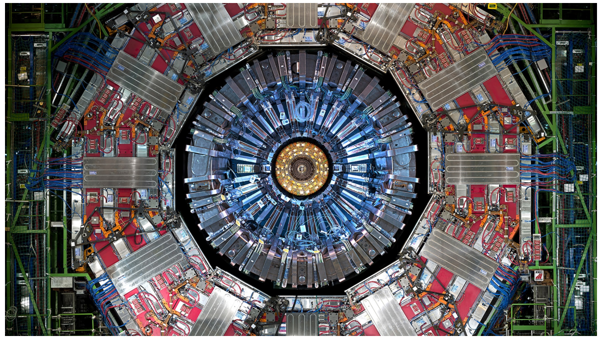
$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

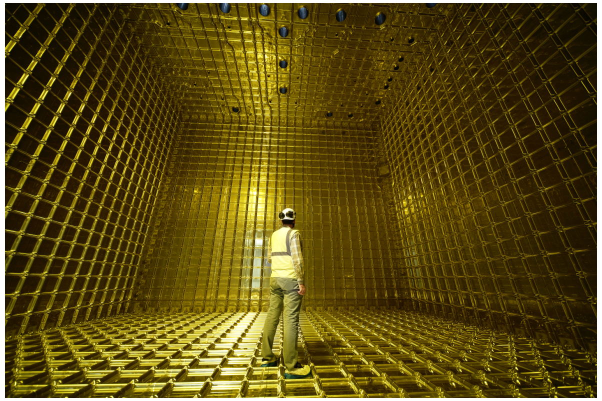
e.g. logistic regression

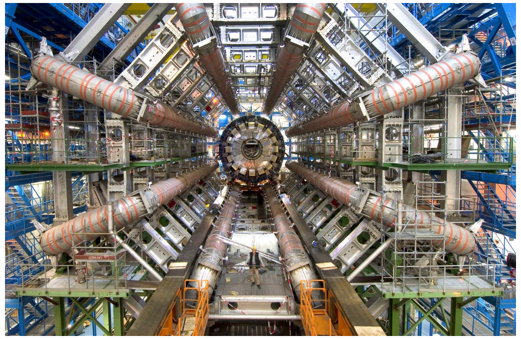
Loss: (Binary) cross-entropy

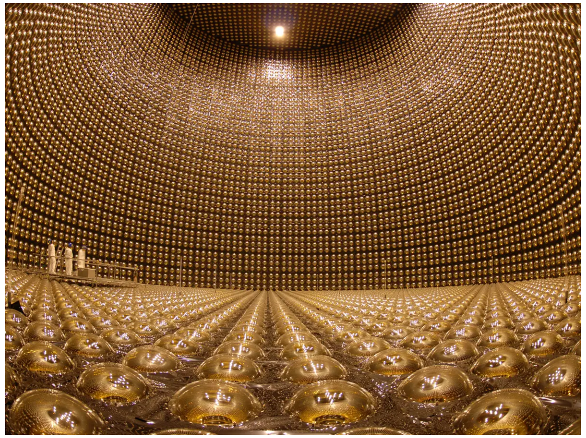
ML Paradigms: Unsupervised Learning

Unsupervised learning has no explicit training labels





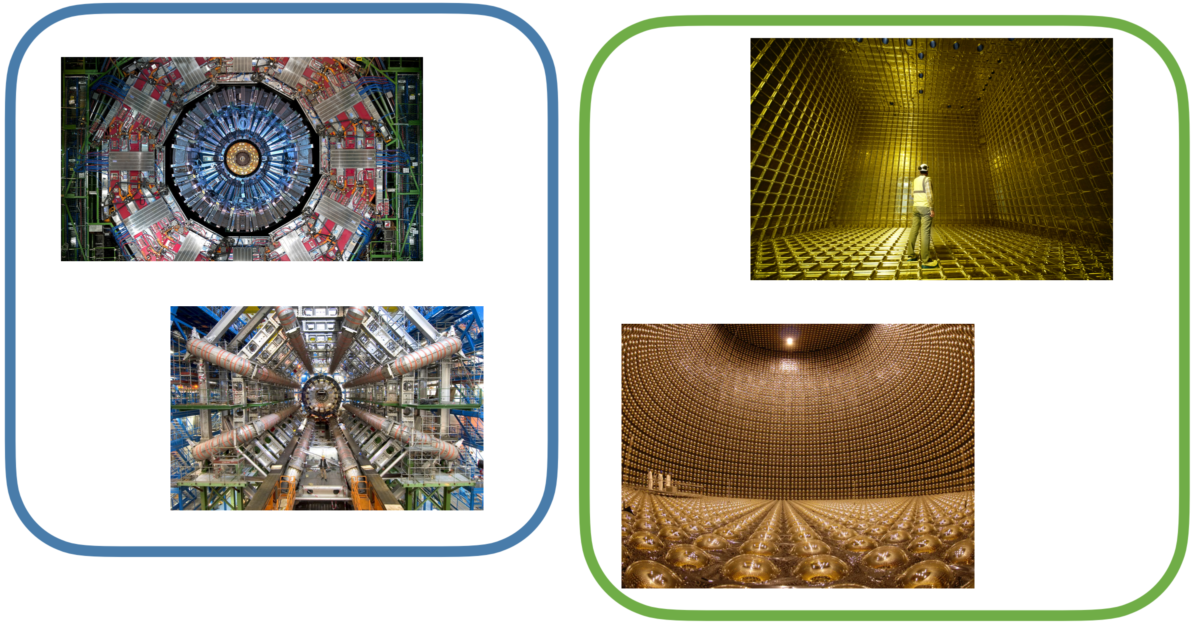


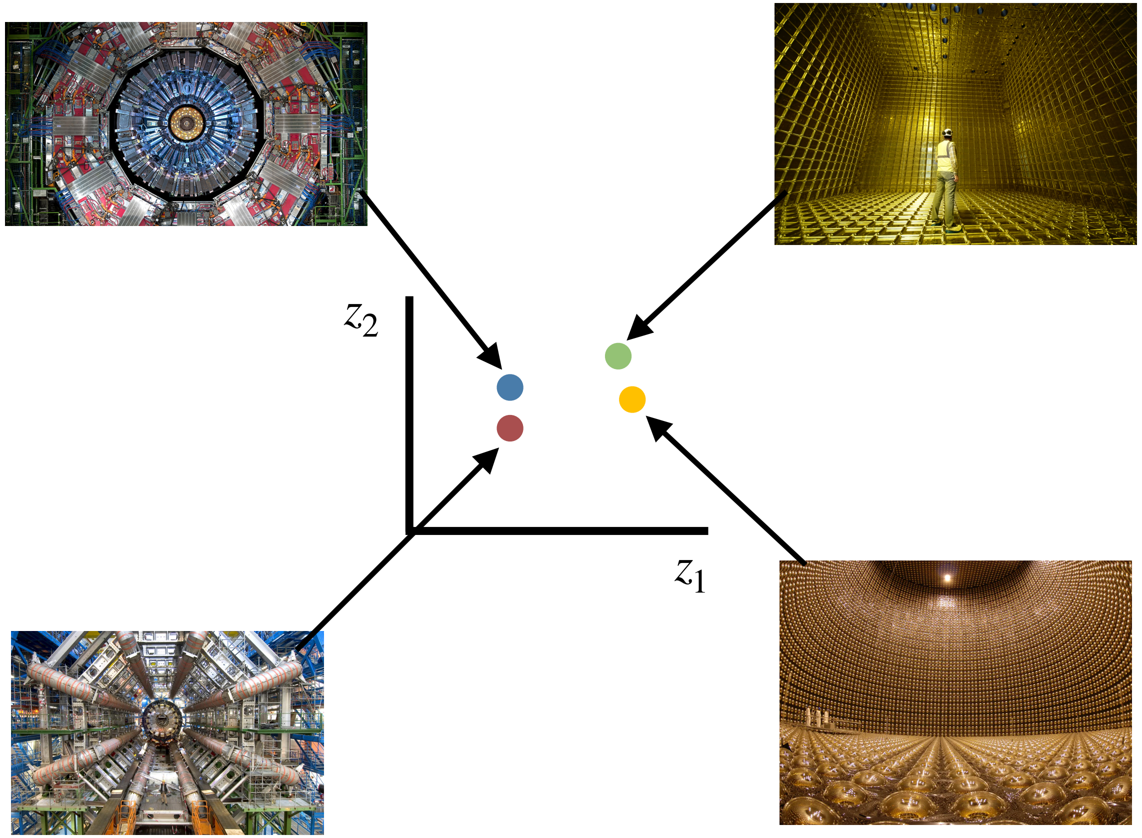


ML Paradigms: Unsupervised Learning

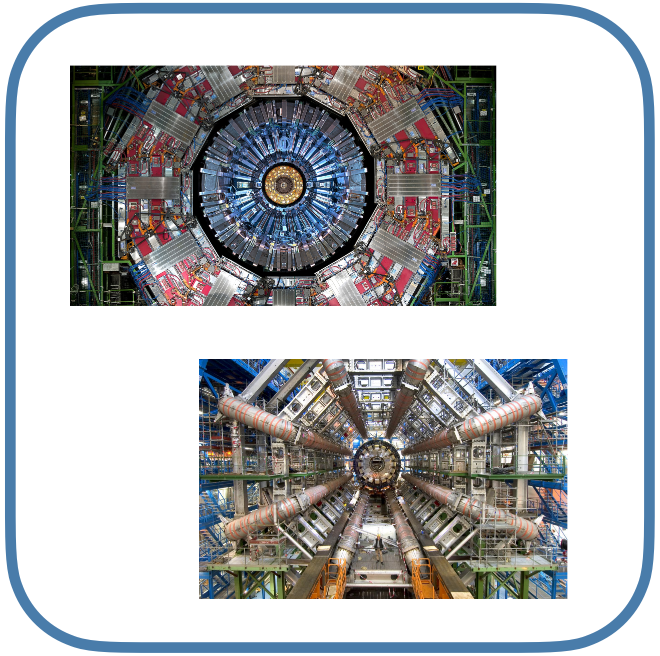
Unsupervised learning has no explicit training labels

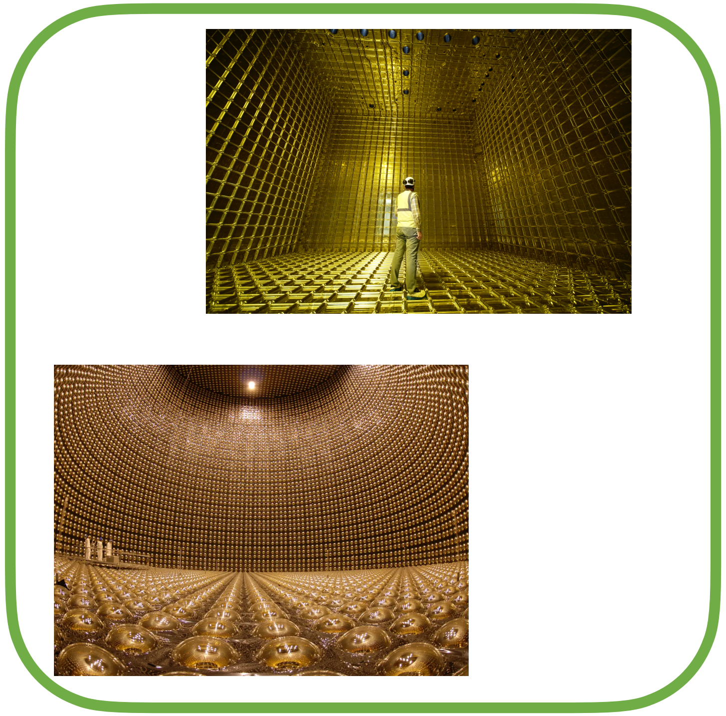
- Often the goal is to find some **clustering** or lower dimensional **representation** of the data





ML Paradigms: Unsupervised Learning





Clustering: e.g. k-means

$f(x) = c_i$ (cluster assignment)

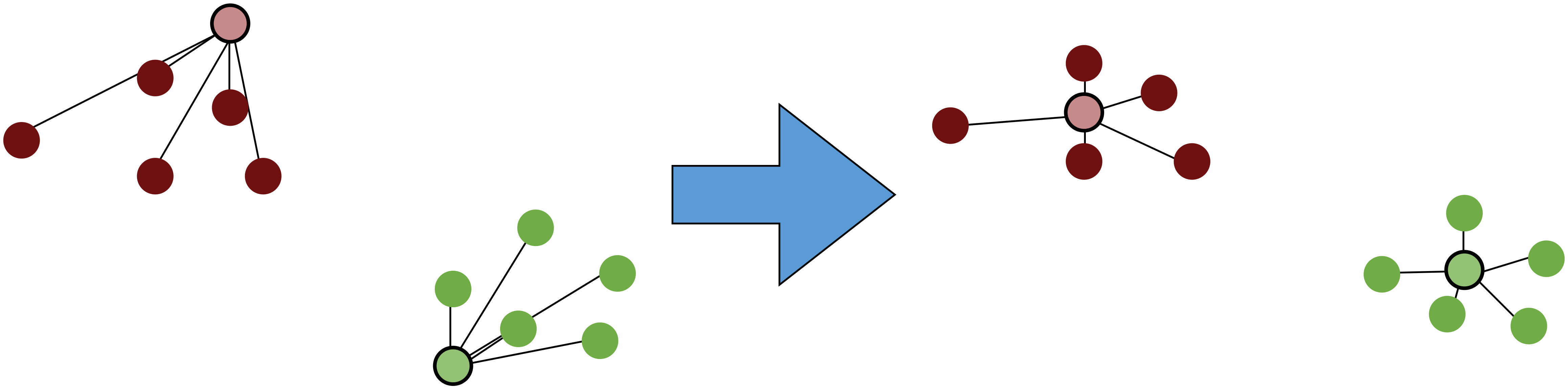
c_i based on closest match to means $\{\mu_i\}_{i=1}^{k}$

Training: adjust $\{\mu_i\}_{i=1}^{k}$ to minimize:

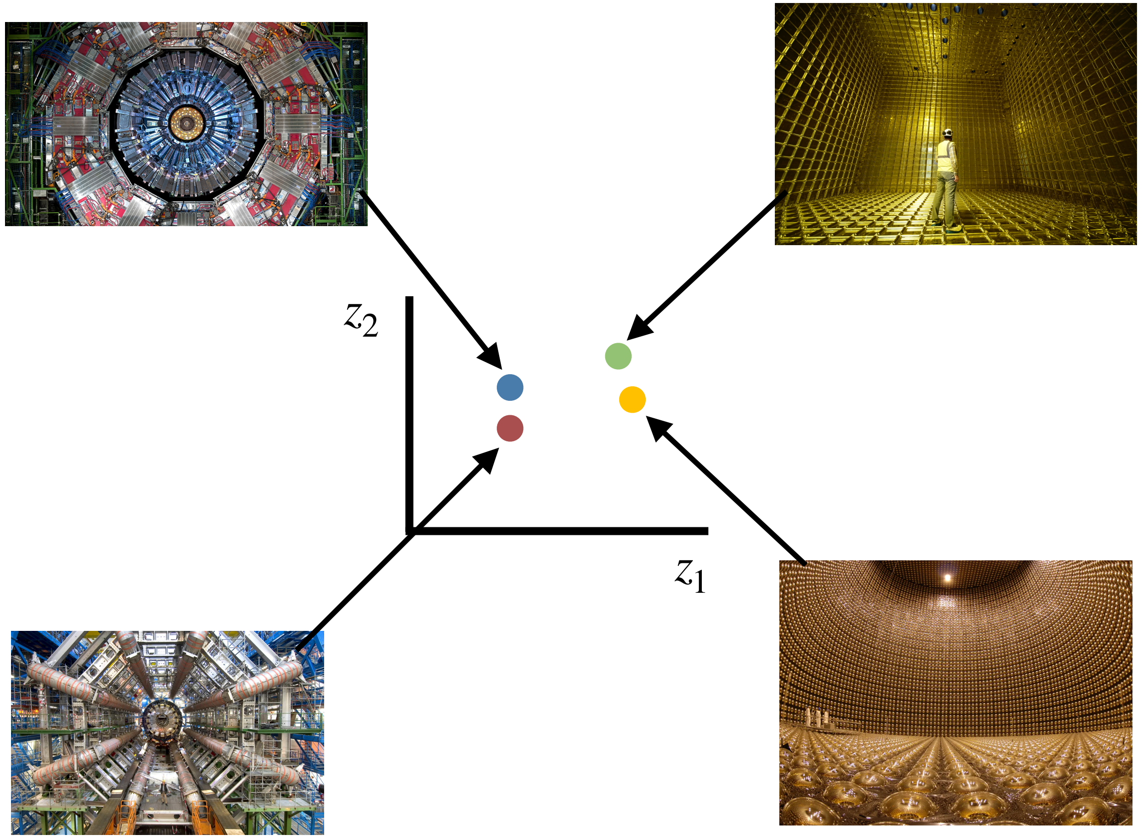
$$\mathscr{L}(\{\mu_i\}_{i=1}^{k}, \{x_i\}_{i=1}^{n}) = \sum_{j=1}^{n} \|x_j - \mu_{c(x_j)}\|^2$$

Closest cluster idx

ML Paradigms: Unsupervised Learning

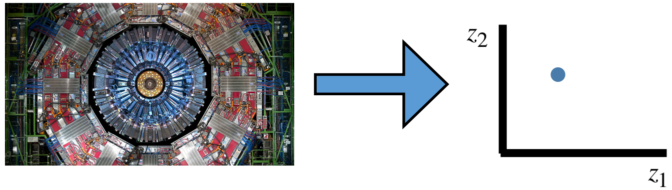


ML Paradigms: Unsupervised Learning

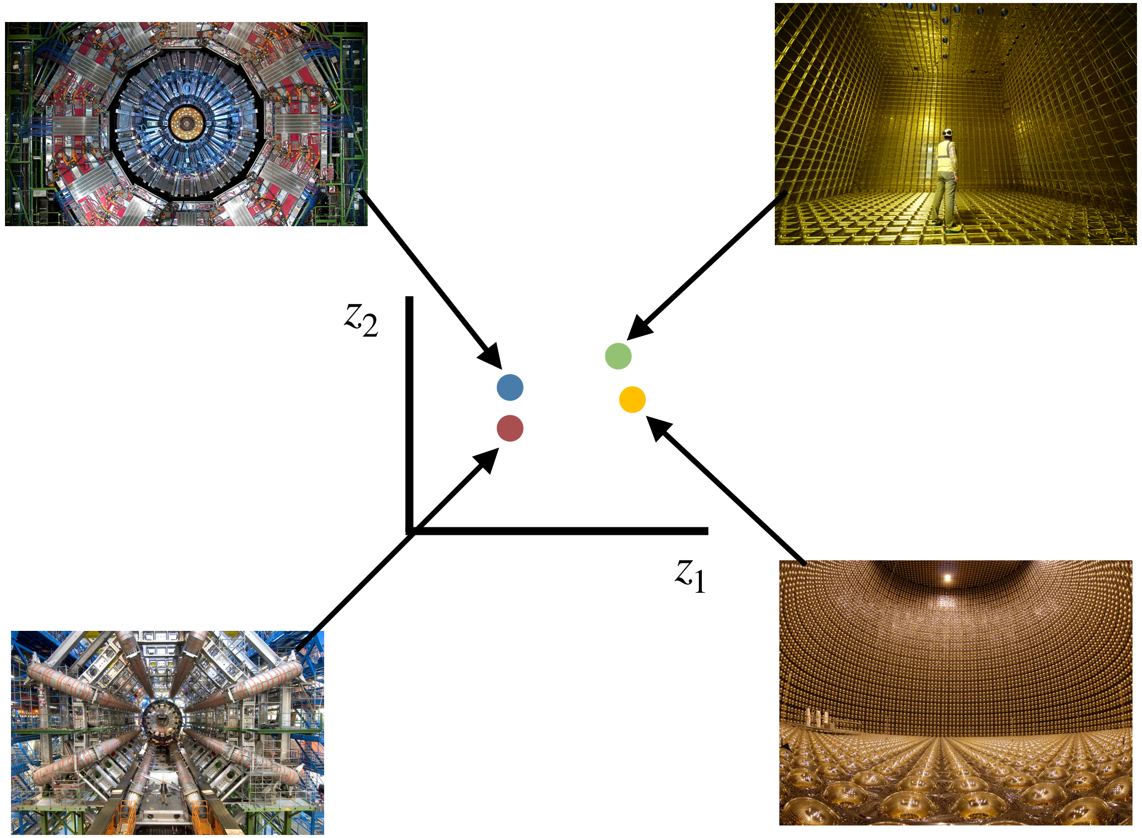


Representations, e.g. autoencoders

$$f_{enc}(x) = z \text{ (encoder)}$$

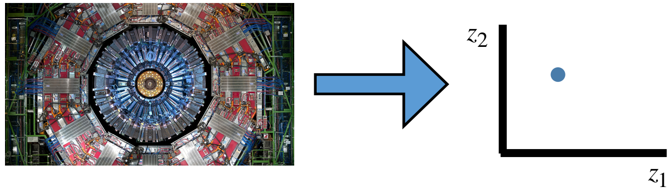


ML Paradigms: Unsupervised Learning



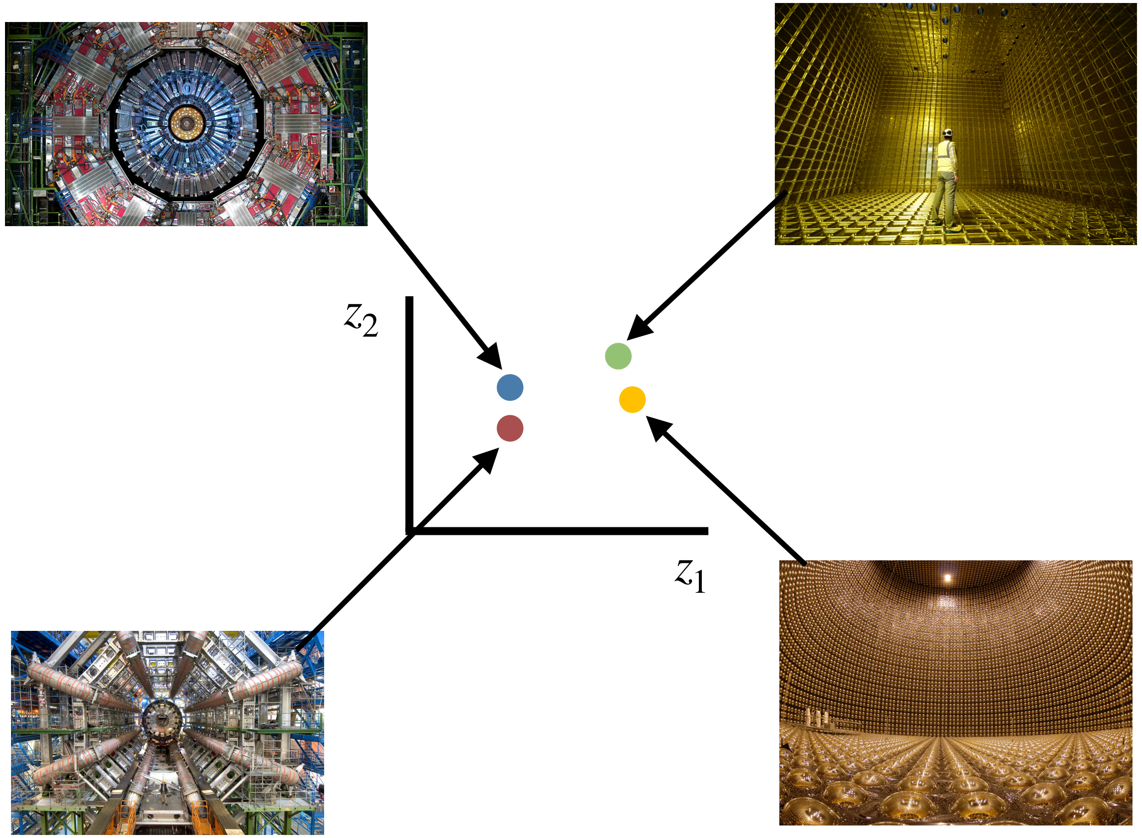
Representations, e.g. autoencoders

$$f_{enc}(x) = z \text{ (encoder)}$$



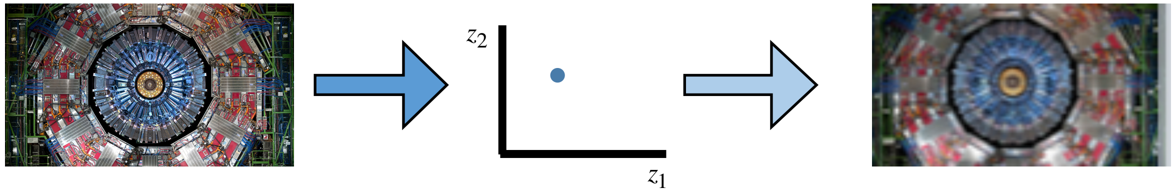
Latent space
(Learned representation)

ML Paradigms: Unsupervised Learning



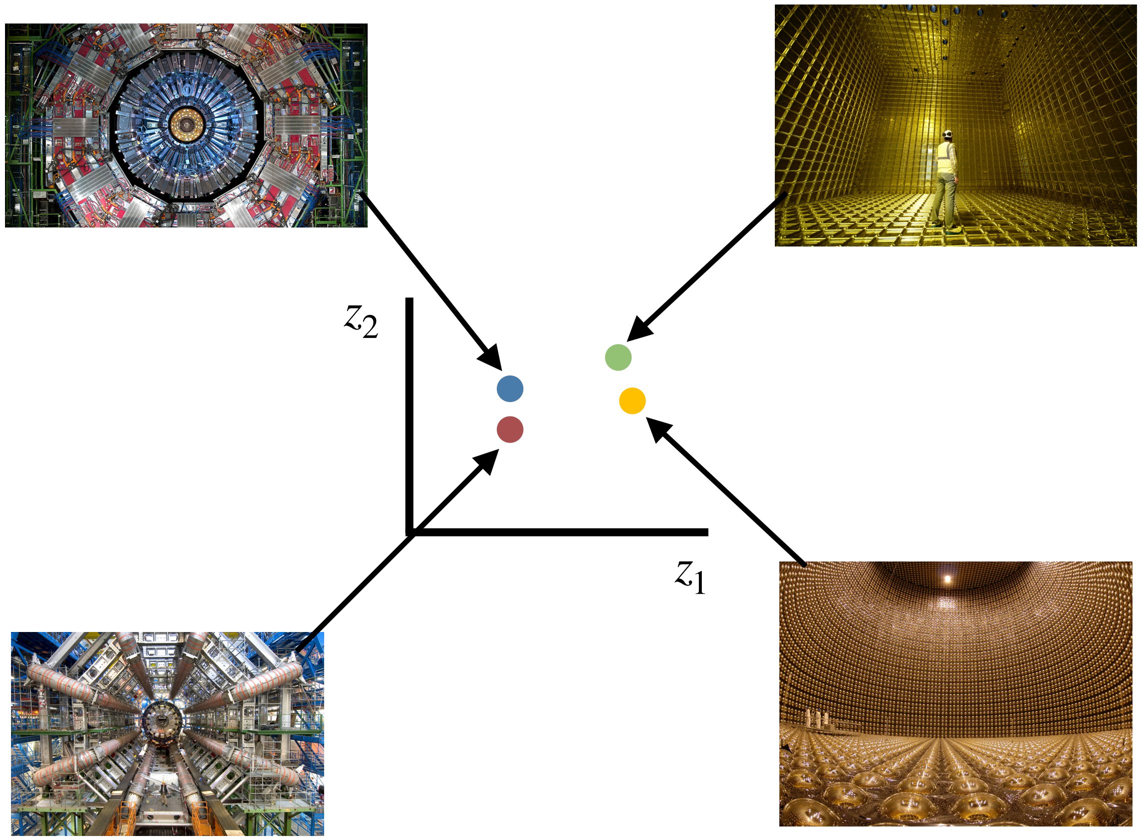
Representations, e.g. autoencoders

$$f_{enc}(x) = z \text{ (encoder)}$$
$$f_{dec}(z) = \tilde{x} \text{ (decoder)}$$



Latent space
(Learned representation)

ML Paradigms: Unsupervised Learning

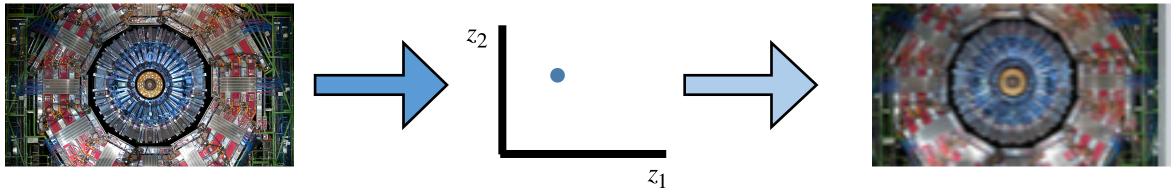


Representations, e.g. autoencoders

$$f_{enc}(x) = z \text{ (encoder)}$$
$$f_{dec}(z) = \tilde{x} \text{ (decoder)}$$

Learn parameters of f_{enc} and f_{dec}
(usually neural networks) to minimize

$$\mathcal{L}(\{f_{enc}, f_{dec}\}, \{x_i\}_{i=1}^{n}) = \sum_{j=1}^{n} \|x_j - \tilde{x}_j\|^2$$

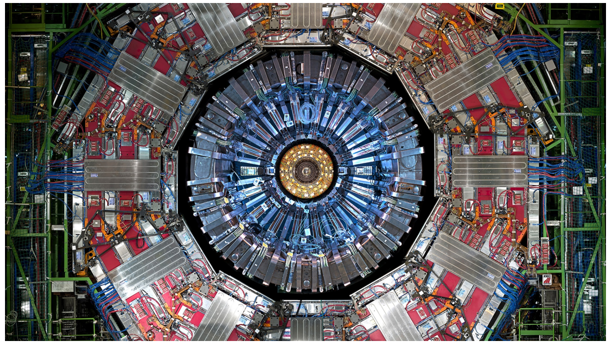


Latent space
(Learned representation)

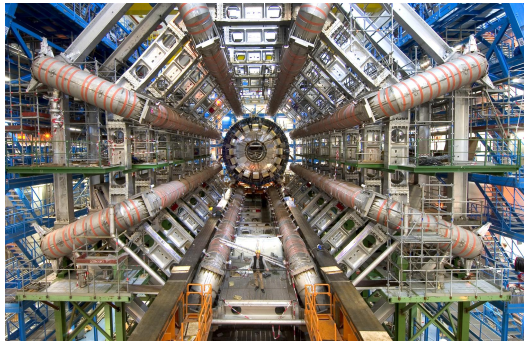
Machine Learning Paradigms

What do we want to do?

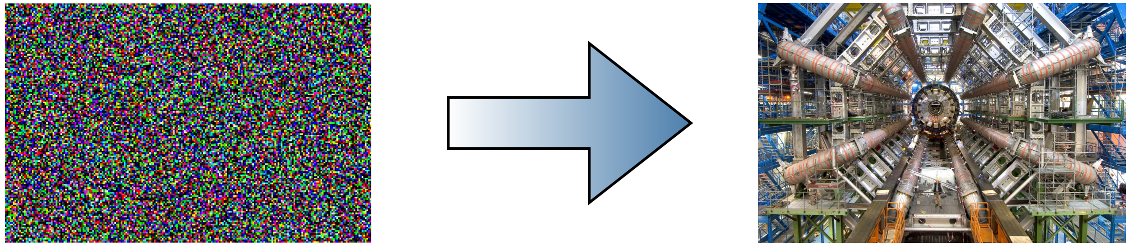
- **Discriminative modeling:** map x to label y (~ learns $p(y \mid x)$) (e.g. classification, logistic regression) — see above!
- **Generative modeling:** learn data distribution ($p(x)$ or $p(x, y)$), in order to generate new samples



CMS



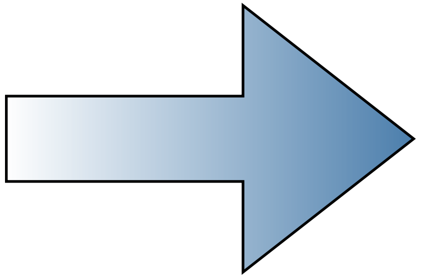
ATLAS

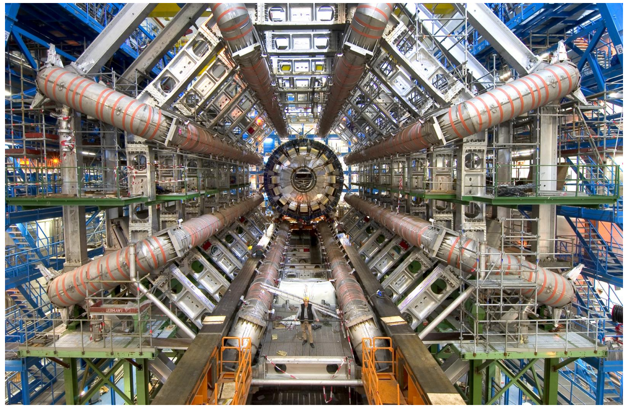


ML Paradigms: Generative Modeling

Generative models aim to learn the probabilistic distribution of a dataset

- Often the goal is to then **sample** from that dataset to generate realistic (data-like) outputs





Prior (latent) distribution

$$p(z)$$

Easy to sample from (e.g. standard normal)

Generative distribution

$$p(x \mid z)$$

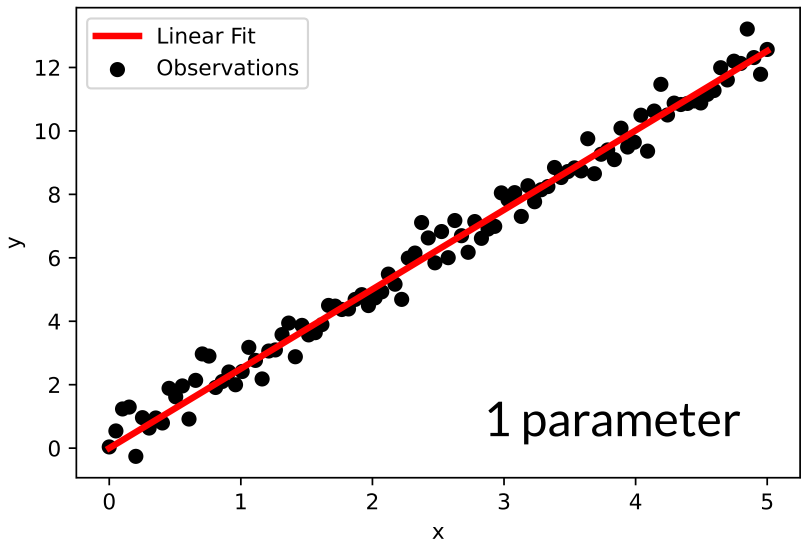
$p_{data}(x)$ hard to sample from. Model trained to match data distribution, given sampled z

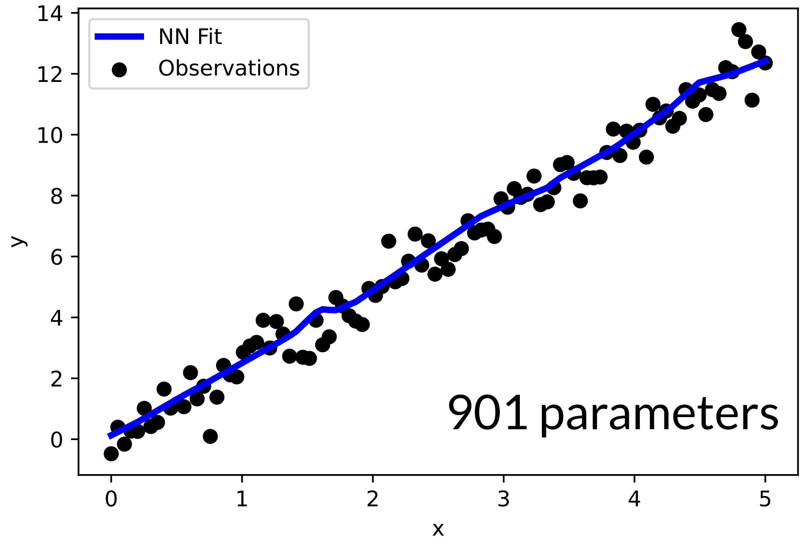
Classic Methodology

(Some) “Classic” Methodology You Should Know

Excitement these days is around neural networks

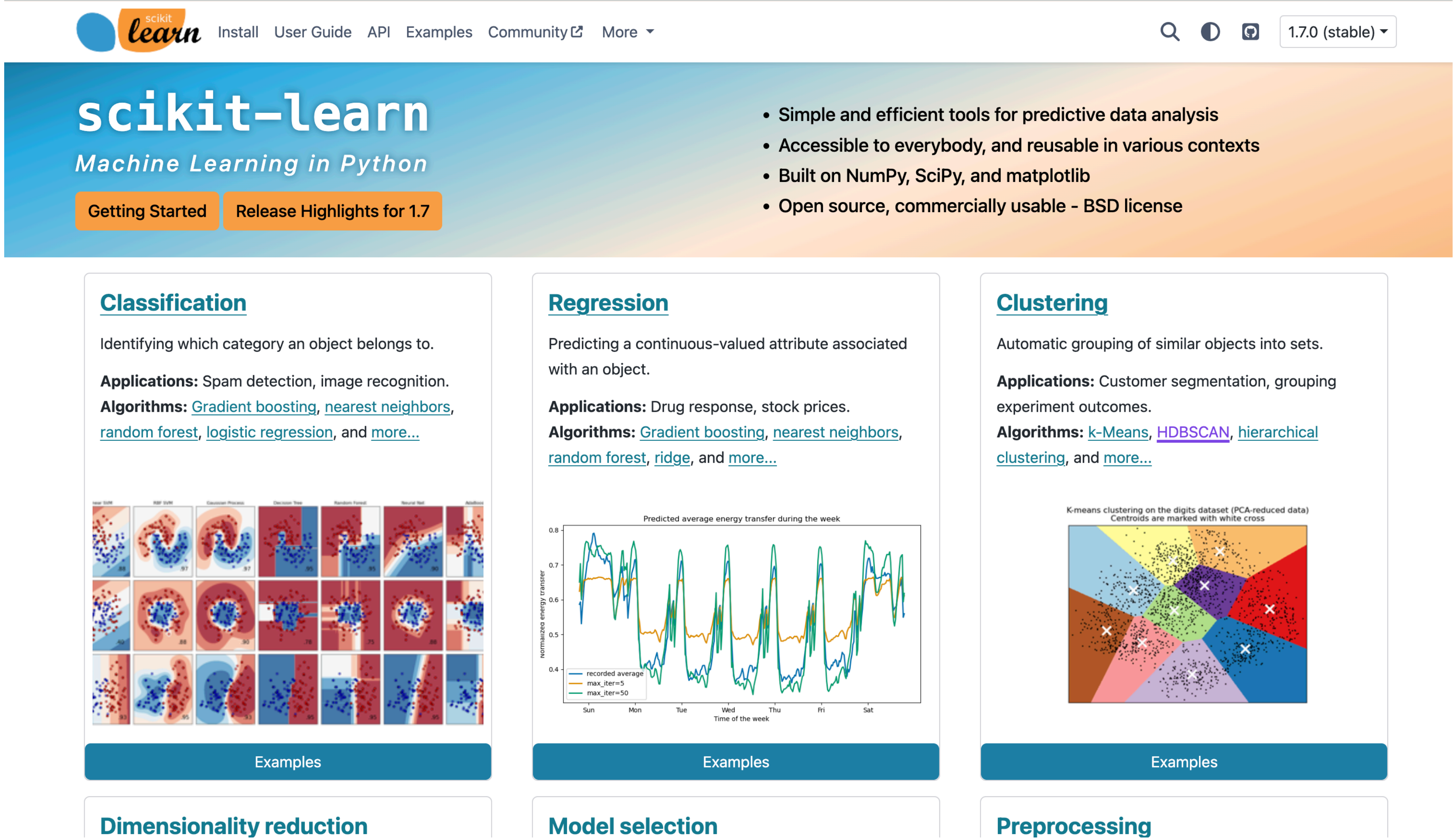
- But! Some “classic” methods can work as well (or better!), depending on context
- Advantages:
  - Simplicity
  - Interpretability
  - Better for small datasets
- **If linear regression will work, use linear regression.**



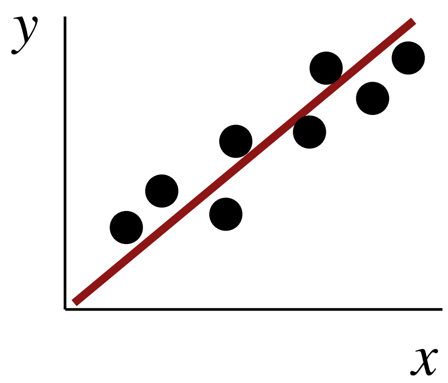


(Some) "Classic" Methodology You Should Know

Very easy to run many classic methods with packages like [scikit-learn]



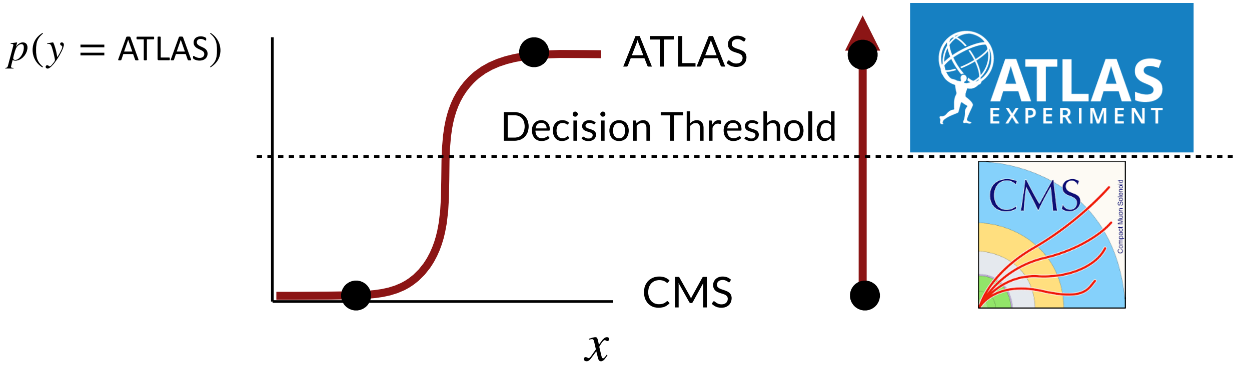
"Classic" Methodology: Linear and Logistic Regression



$$f(x) = w \cdot x + b$$

Linear regression

Loss: Mean squared error



$$f(x) = \sigma(w \cdot x + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

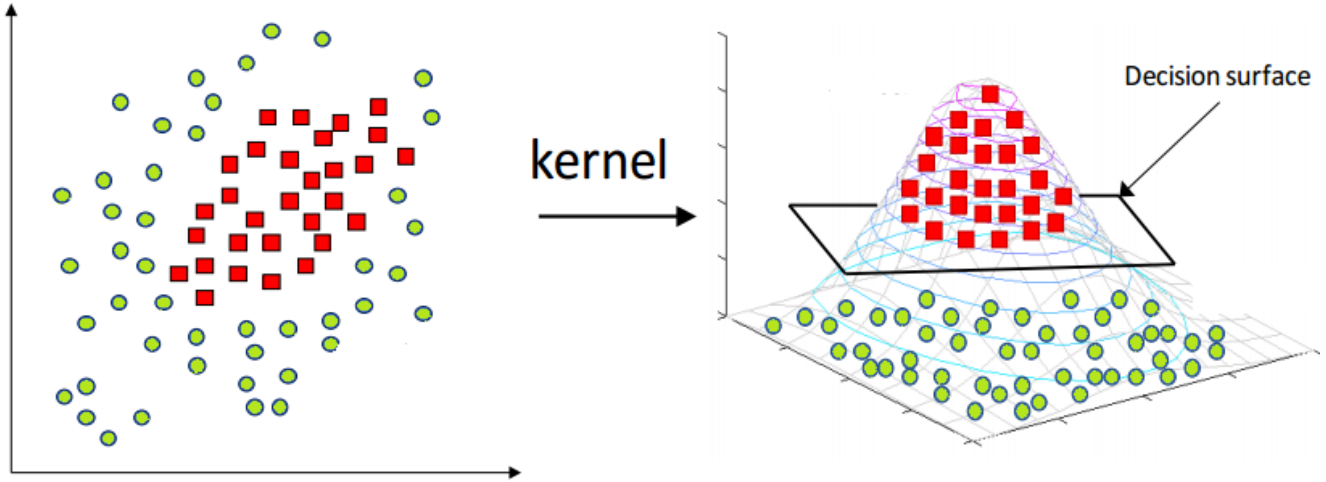
Logistic regression

Loss: (Binary) cross-entropy

"Classic" Methodology: Kernel Functions

A **kernel function** $K(x, x')$ describes the similarity between two data points

- Similarity calculated in a high dimensional feature space, but no explicit map to that feature space
- All we need is the **inner product** between high dimensional vectors: easily computable function of the original inputs



$$K(x, x') = \langle \phi(x), \phi(x') \rangle$$

$$= \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2} \right) \quad \text{RBF Kernel}$$

$$= (\gamma x^T x' + c)^d \quad \text{Polynomial Kernel}$$

Source

See also: Support vector machines (SVMs)

"Classic" Methodology: Gaussian Processes

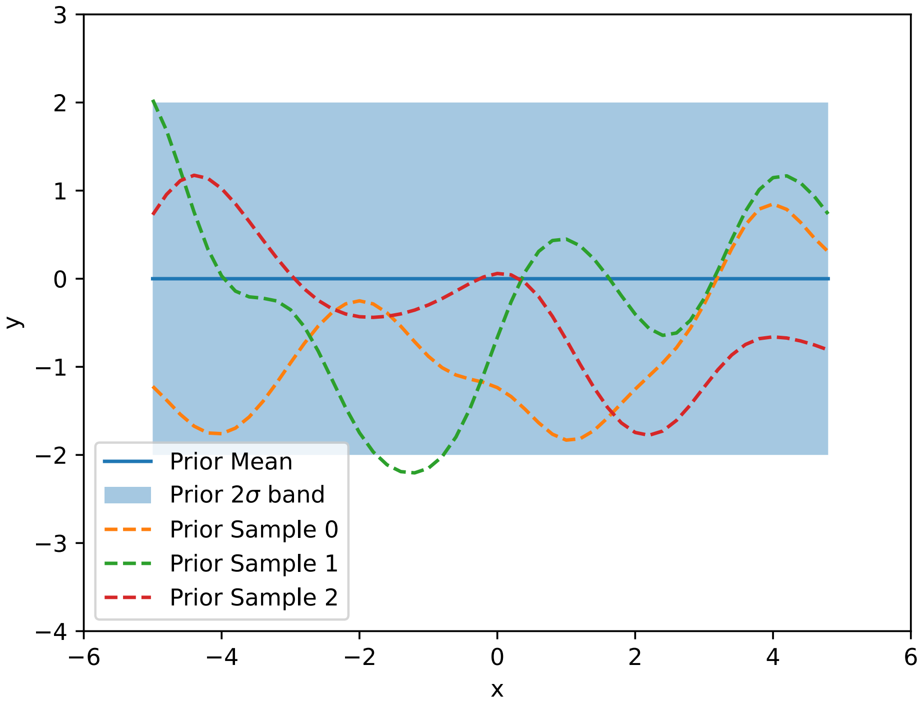
Gaussian processes are Bayesian models defined by a **mean function** $\mu(x)$ a **kernel function** $K(x, x')$. They define a **distribution over functions** (=> **uncertainty estimation**)

- For points $\{x_1, \ldots, x_n\}$, we have $f(x) \sim \mathcal{N}(\mu(x), K(x, x'))$, where $K(x, x')$ defines an $n \times n$ covariance matrix
- Given observations, we may use Bayes' rule to update our model

$$K(x, x') = \exp\left(- \frac{\|x - x'\|^2}{2\sigma^2} \right)$$

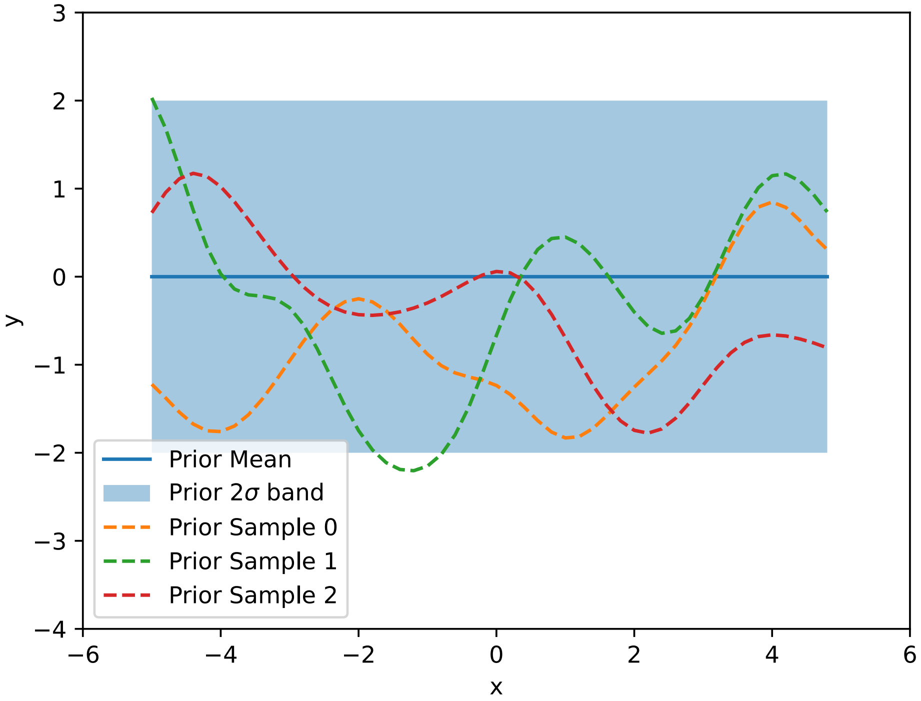
RBF Kernel

"Points near each other impact each other, points far away don't"

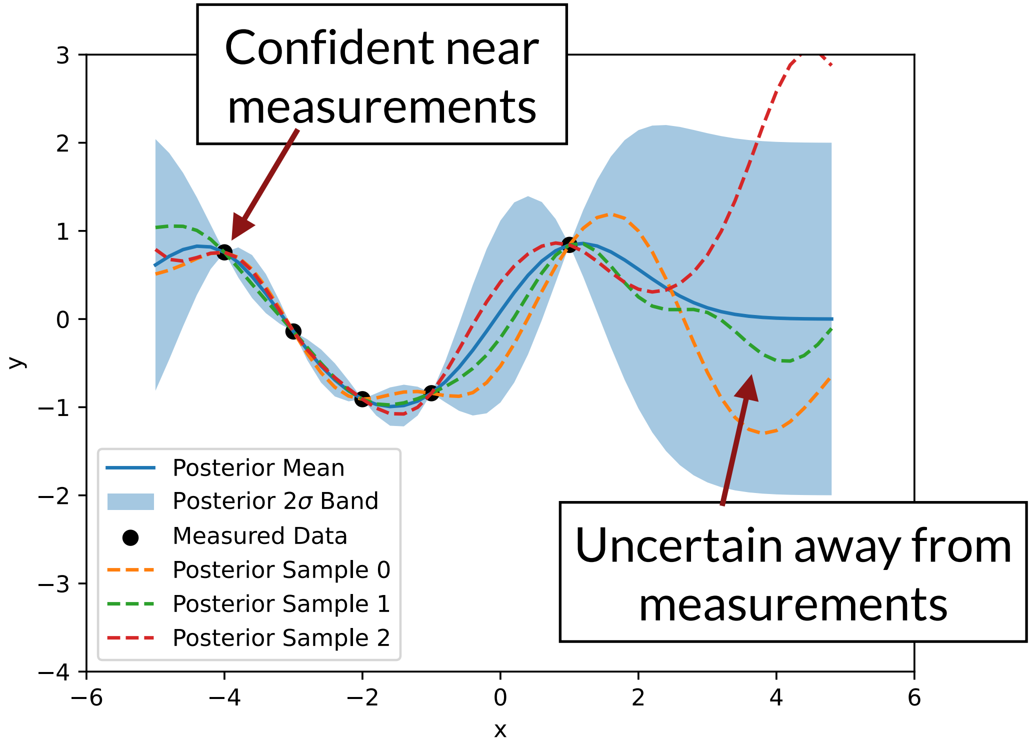


SLAC

“Classic” Methodology: Gaussian Processes



Measurements



Prior:

$$p(f)$$

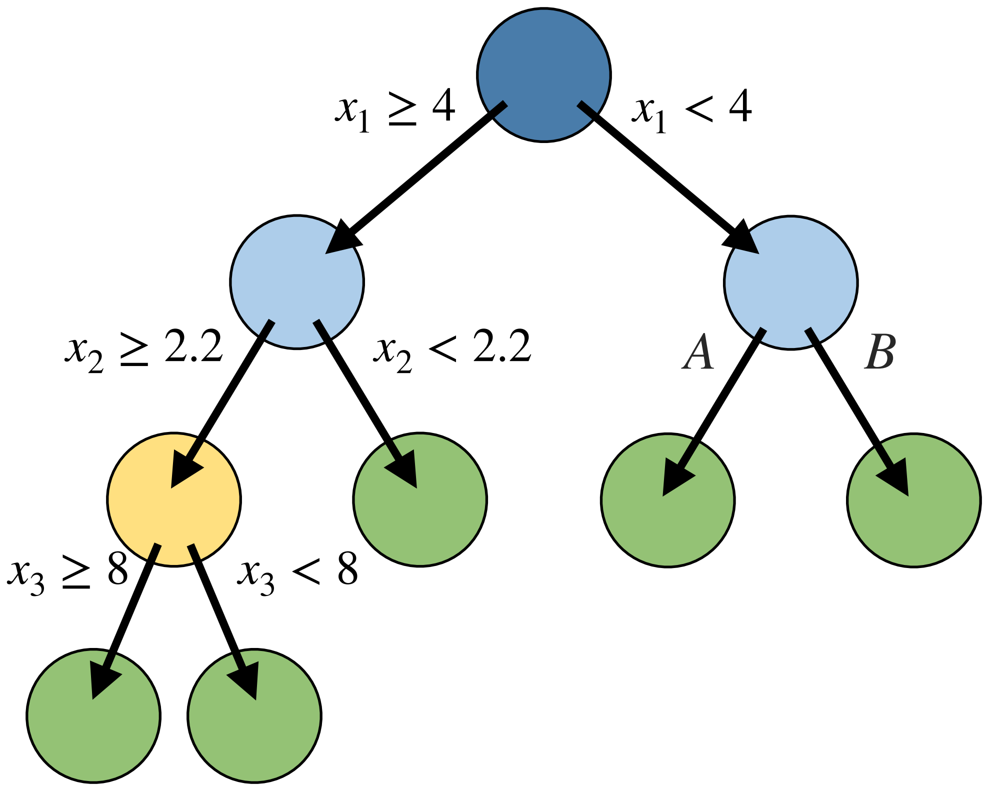
Model with no observations. Structure from kernel, mean function choice.

Posterior:

$$p(f \,|\, \mathscr{D})$$

Update of prior given observed data

“Classic” Methodology: Decision Trees

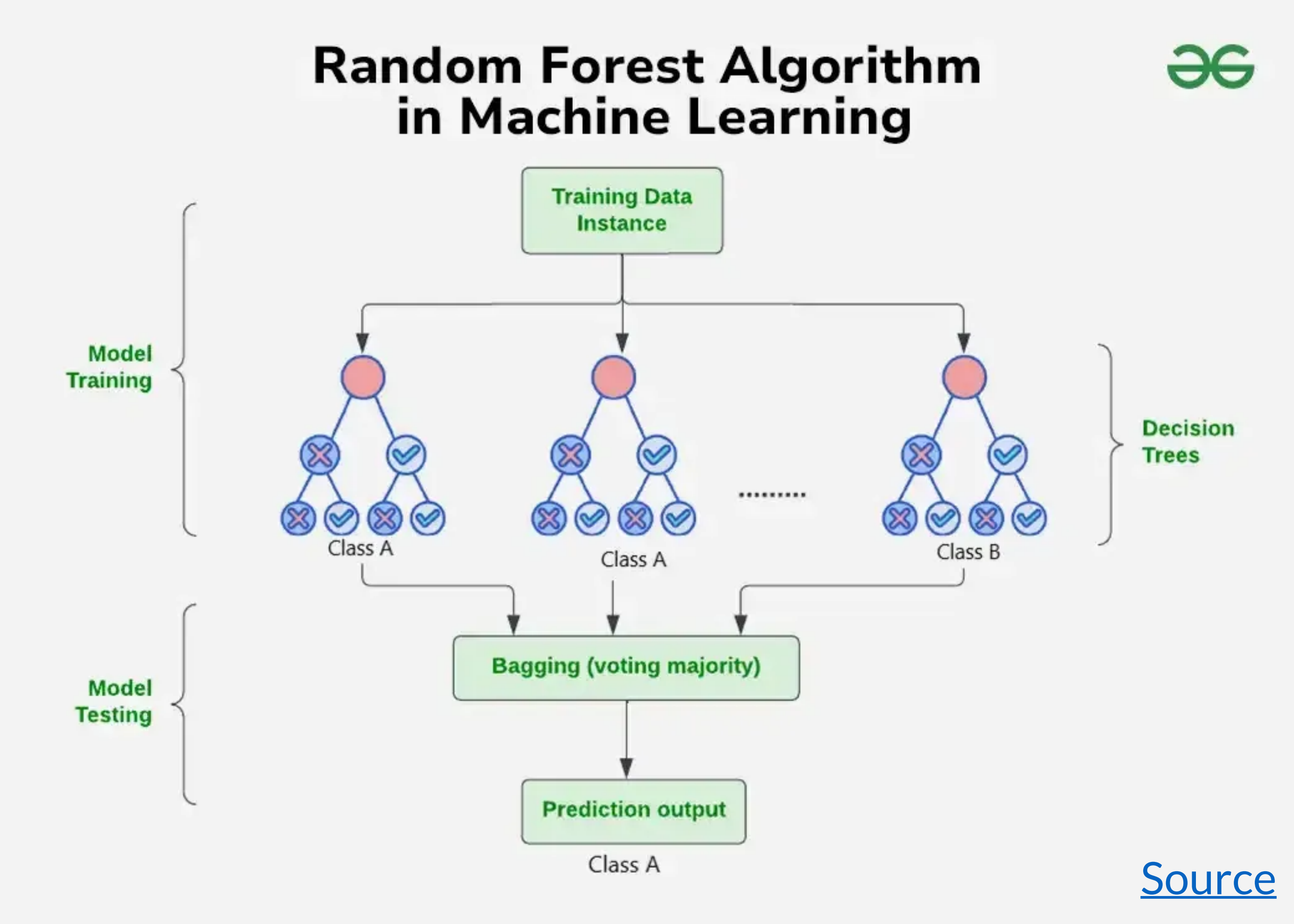


- Based on binary splits of input variables
  - thresholds on continuous variables (e.g. $x_1 \geq 4$)
  - categorical variable values (A or B)
- Tree is a sequence of decisions => multi-dimensional
- End “leaf” nodes contain predictions (regression prediction, classification label)
- **Training:** greedily choose splits starting from the base node, recursively move through

“Classic” Methodology: Random Forests

Random forests: train several trees (an **ensemble**) on

- Randomly **sampled subsets of input data** (bagging)
- With a **random selection of features** for each split
- Final result: Average (regression) or highest vote (classification) across trees

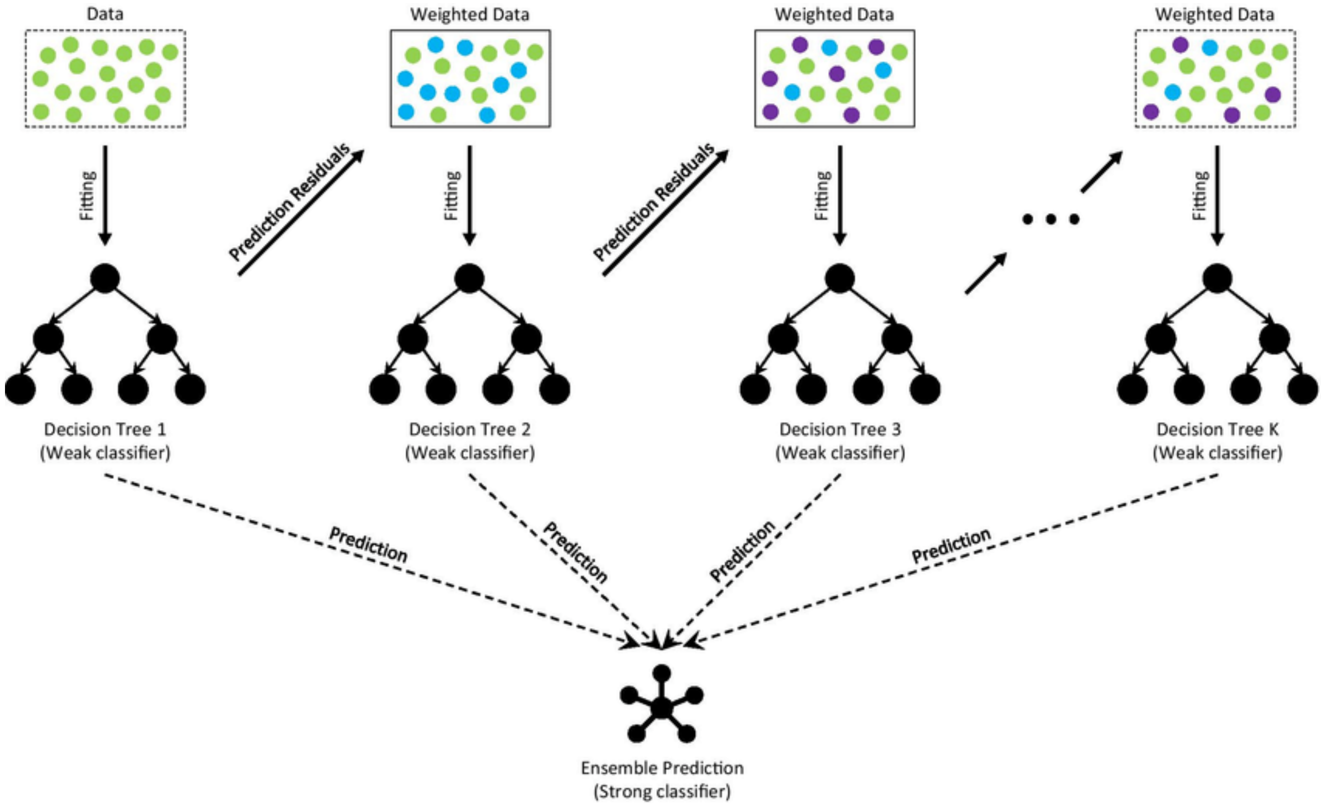


Source

“Classic” Methodology: Boosted Decision Trees (BDTs)

Very classic in HEP

- Train many shallow (only a few decisions) trees **sequentially**
- Each tree tries to correct errors of previous trees by
  - **Focusing on incorrectly predicted data** (AdaBoost)
  - **Predicting residuals** (gradient boosting)
- Final prediction is a (weighted) sum of trees



Source

See e.g. XGBoost

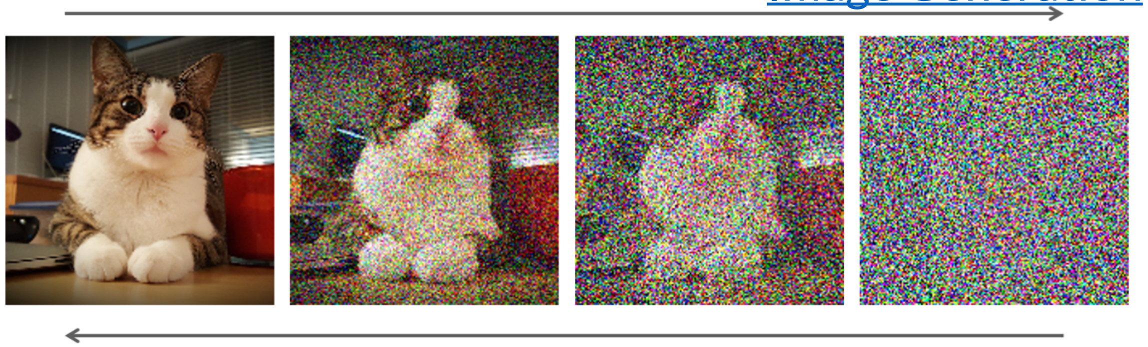
Intro to Neural Networks

Introduction to Neural Networks

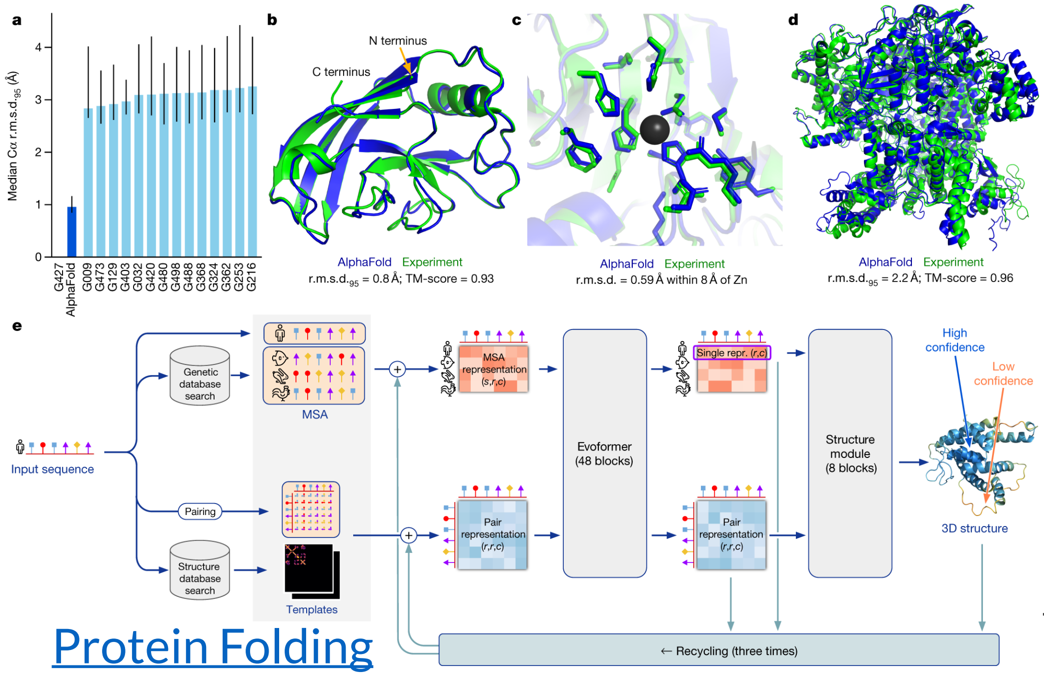
Neural networks are the backbone of modern machine learning

Large Language Models

Image Generation



Semantic Segmentation

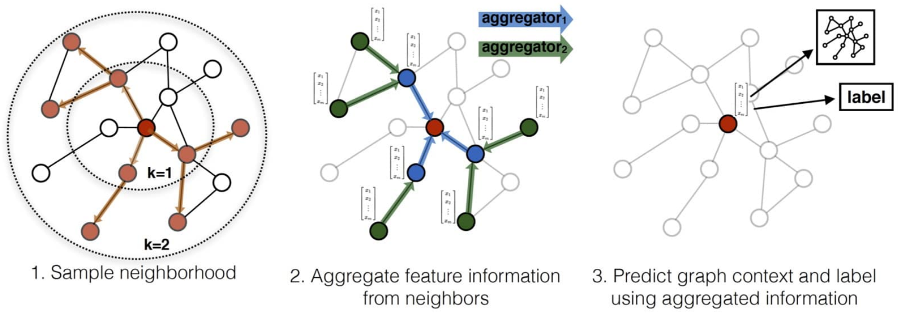


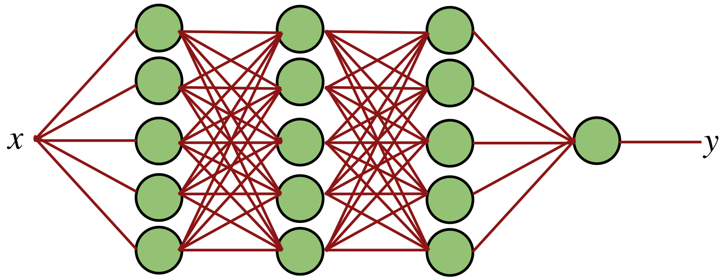
Protein Folding

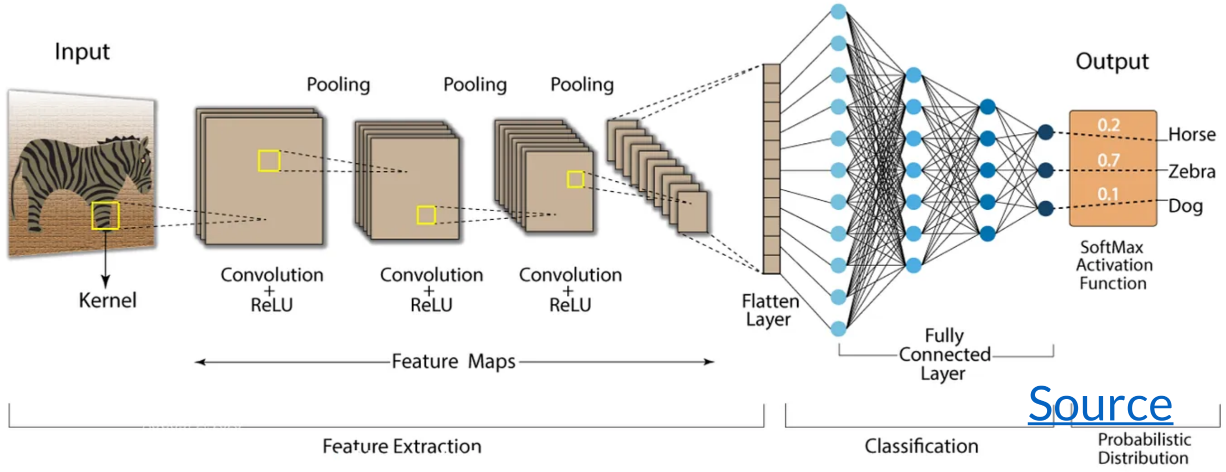
Introduction to Neural Networks

Our focus:

- Build up multi-layer perceptrons (fully connected networks) in detail
- Broadly highlight other network architectures/ why they're useful

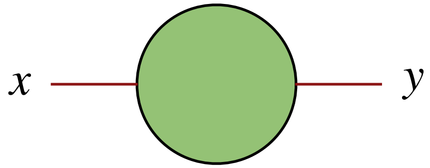






Source

Introduction to Neural Networks

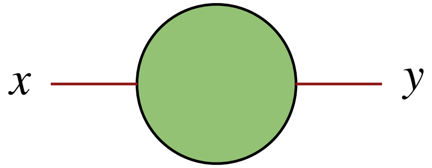


$$\mathrm{NN}(x) = Wx + b$$

Weights   Biases

"Fully connected neural network with linear activation function in 1d"

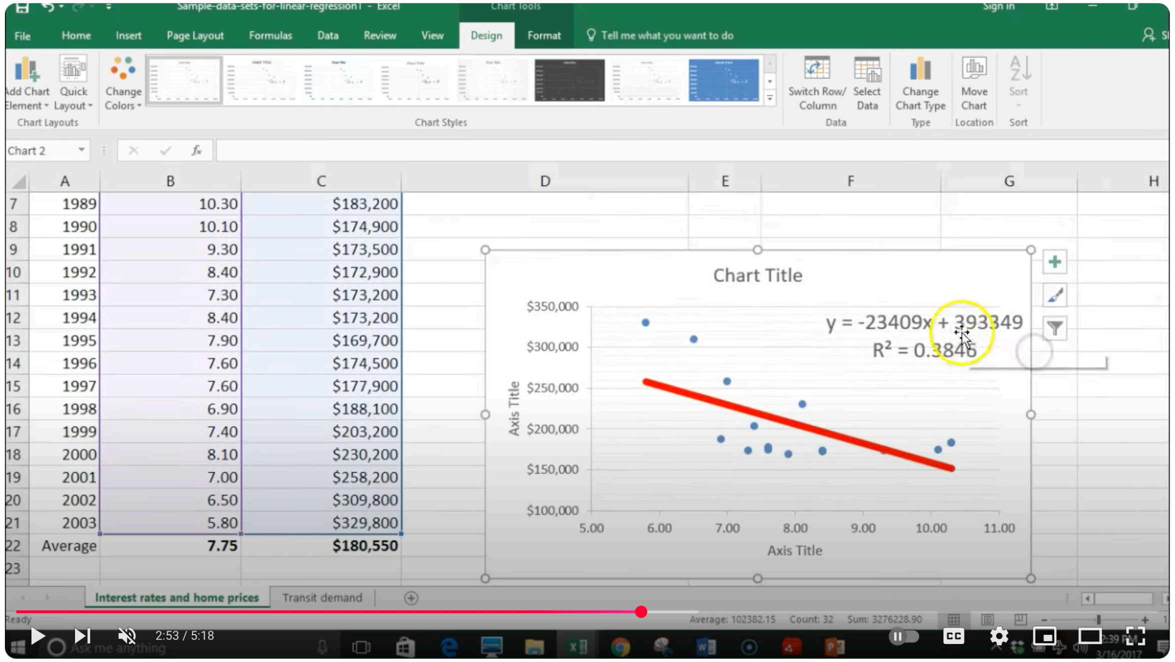
Introduction to Neural Networks



$$\mathrm{NN}(x) = Wx + b$$

Weights   Biases

“Fully connected neural network with linear activation function in 1d”



How to do a linear regression on excel

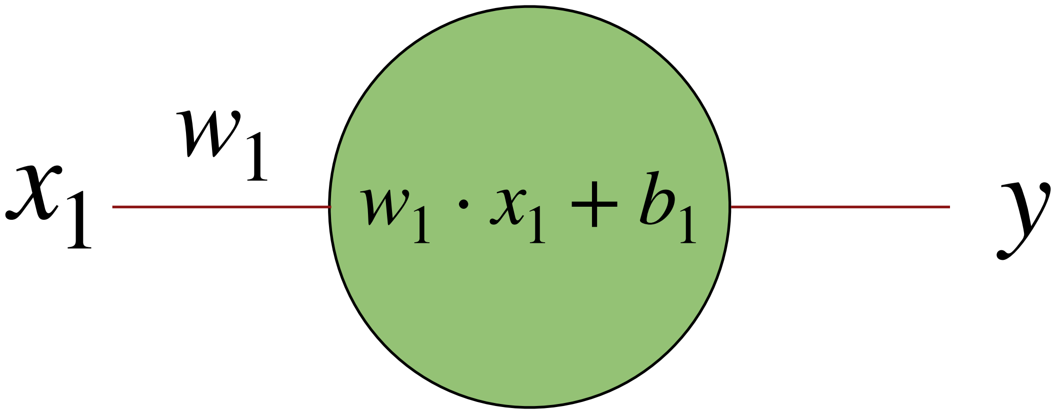
Mona Schraer
2.35K subscribers
Subscribe
7.6K Share Download Clip Save

Introduction to Neural Networks

$$W = (w_1) \qquad \text{NN}(x) = Wx + b$$

$$b = \left(b_1\right)$$

"Fully connected neural network with linear activation function in 1d"



Introduction to Neural Networks

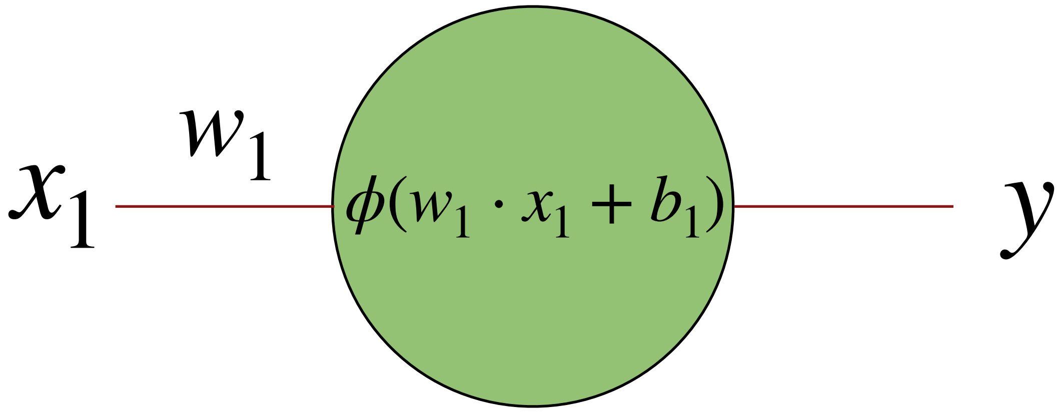
$$W = (w_1)$$

$$b = (b_1)$$

$$\mathrm{NN}(x) = \phi(Wx + b)$$

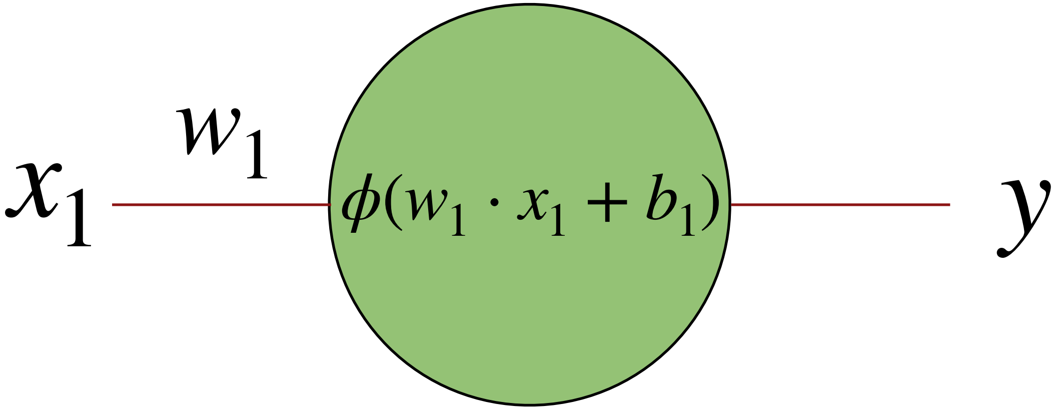
Activation Function

"Fully connected neural network with activation function ϕ in 1d"



Introduction to Neural Networks

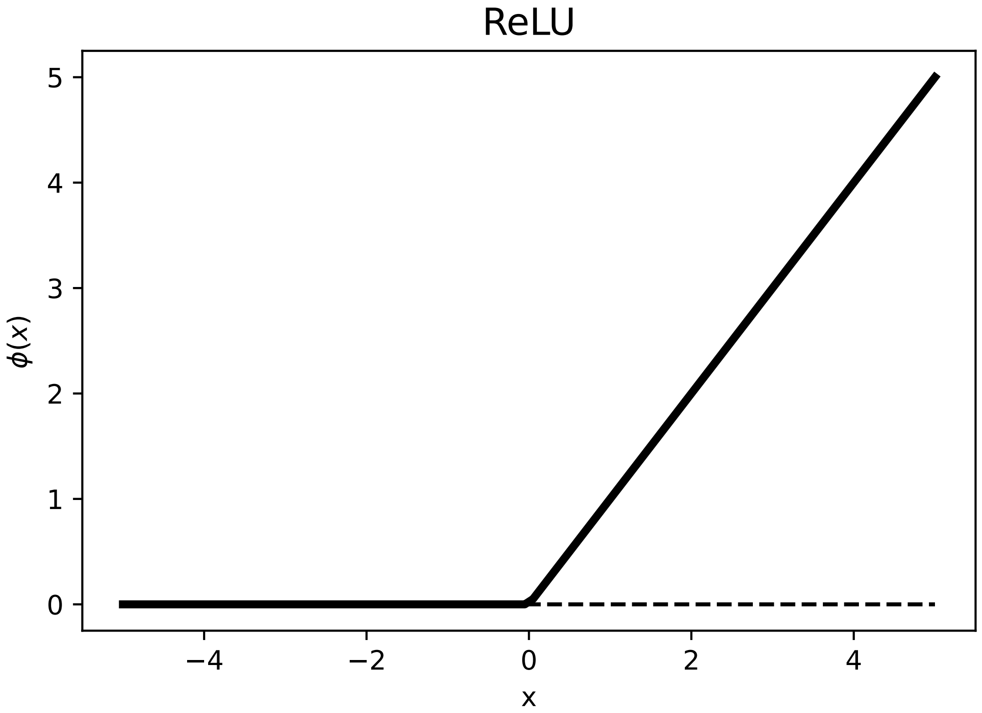
$$\mathrm{NN}(x) = \phi(Wx + b)$$



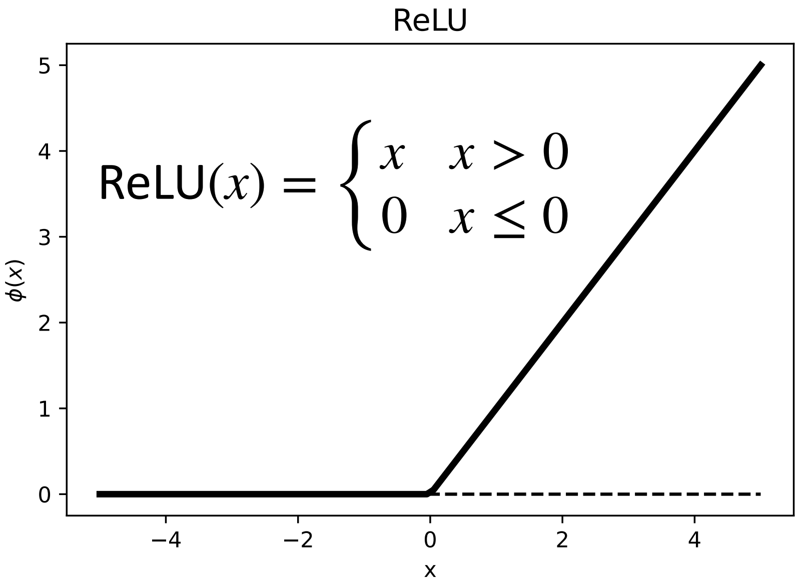
$$\phi(x) = \mathrm{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

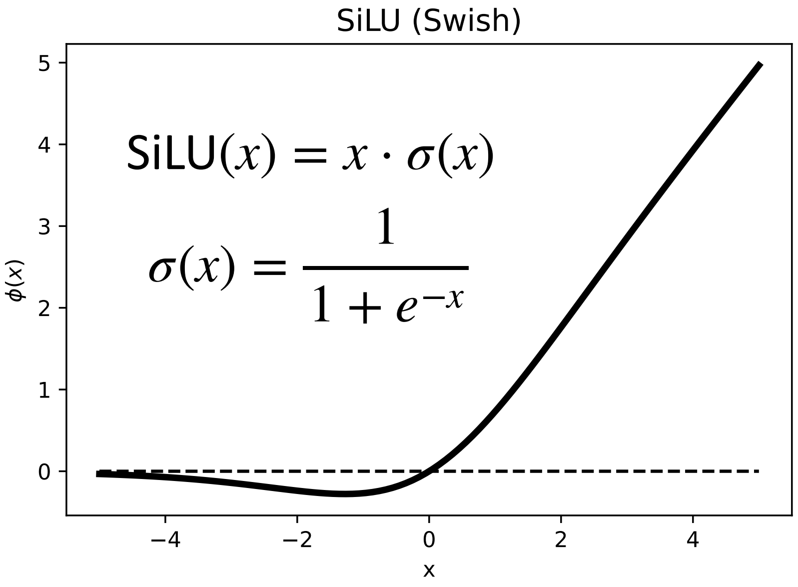
"Rectified Linear Unit"

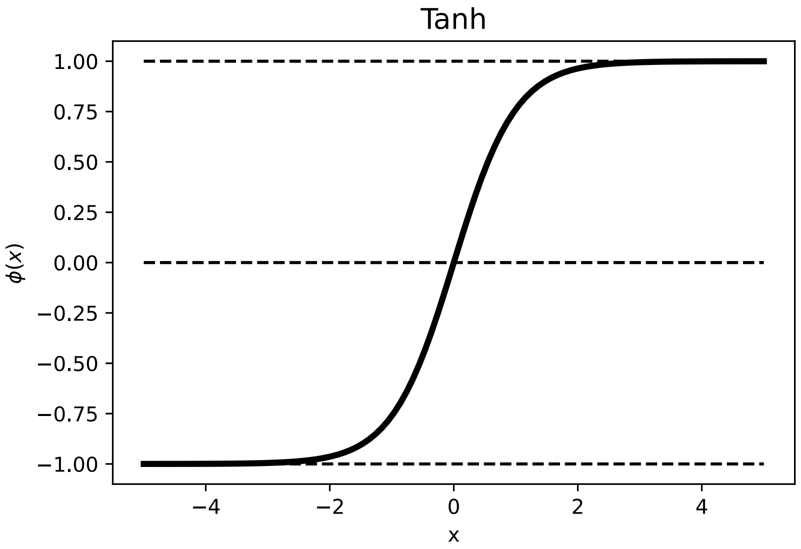
Activation functions introduce non-linearity — increases expressivity of neural networks



Introduction to Neural Networks

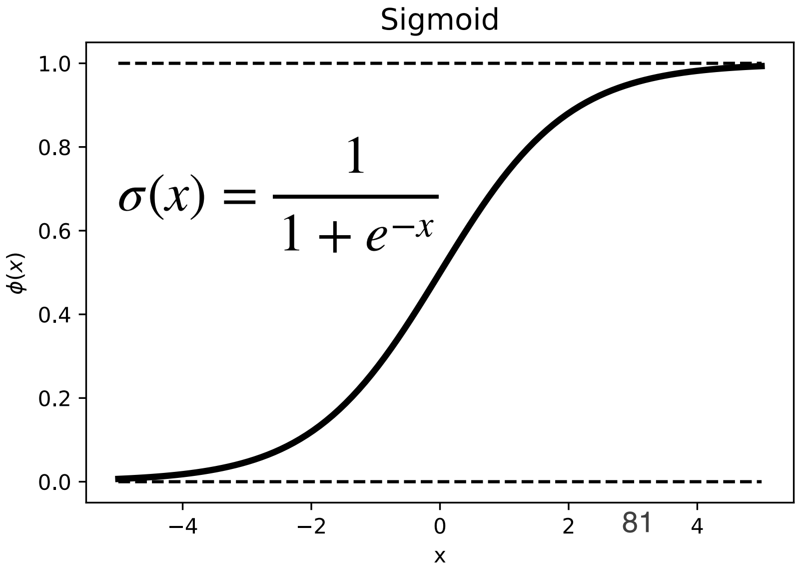






Choice of activation function has an impact on network output/structure

- ReLU most common (simple, sparse activation), SiLU smooths out ReLU
- Tanh is bounded/zero centered, sigmoid good for probabilistic interpretation

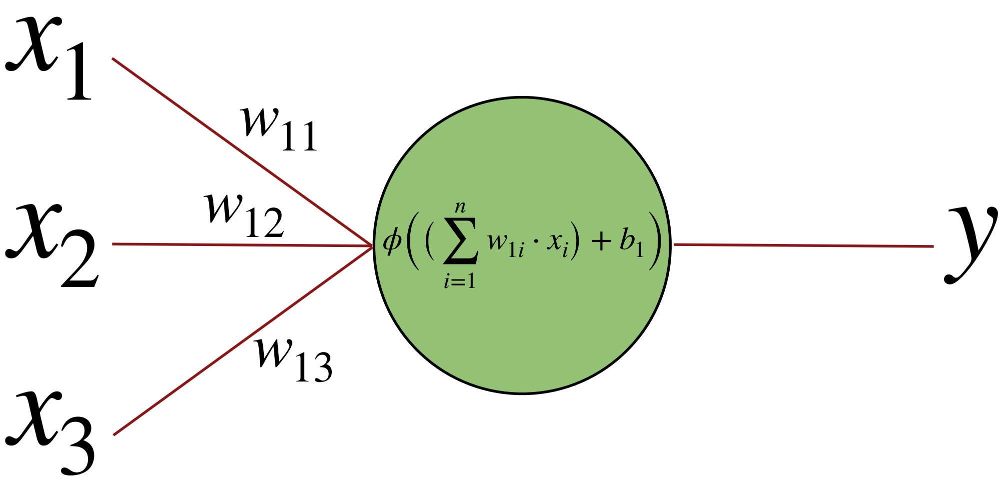


SLAC

Introduction to Neural Networks

$$\mathrm{NN}(x) = \phi(Wx + b)$$

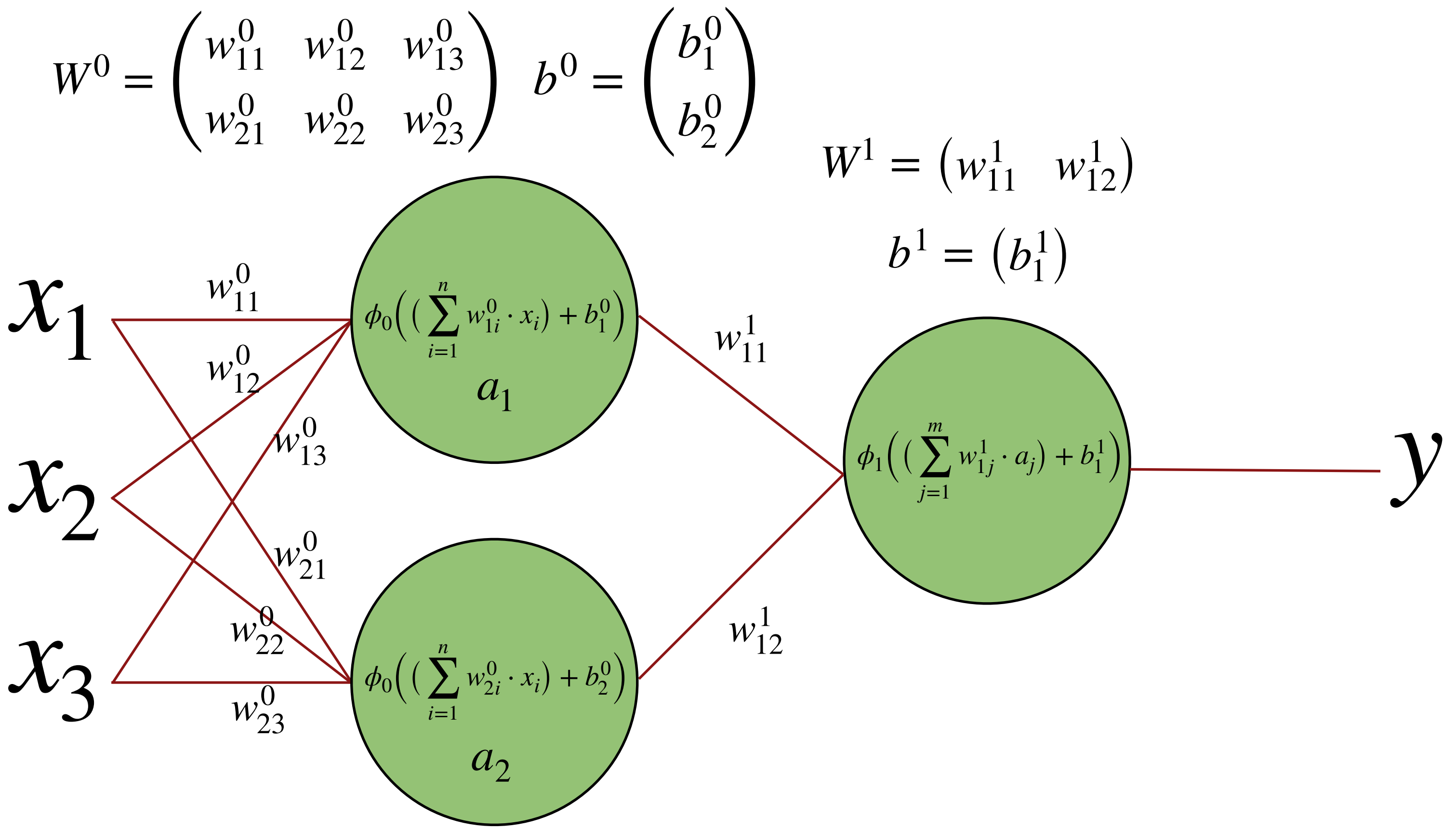
$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \end{pmatrix} \quad b = \begin{pmatrix} b_1 \end{pmatrix}$$



In practice, W is some $m \times n$ matrix

- Each node is then a weighted sum of its inputs, + bias, passed through an activation function
- Shape of weight matrix comes from input/output dimensions
- **Neural networks:** matrix multiplications, bias vectors, and non-linearities

Introduction to Neural Networks

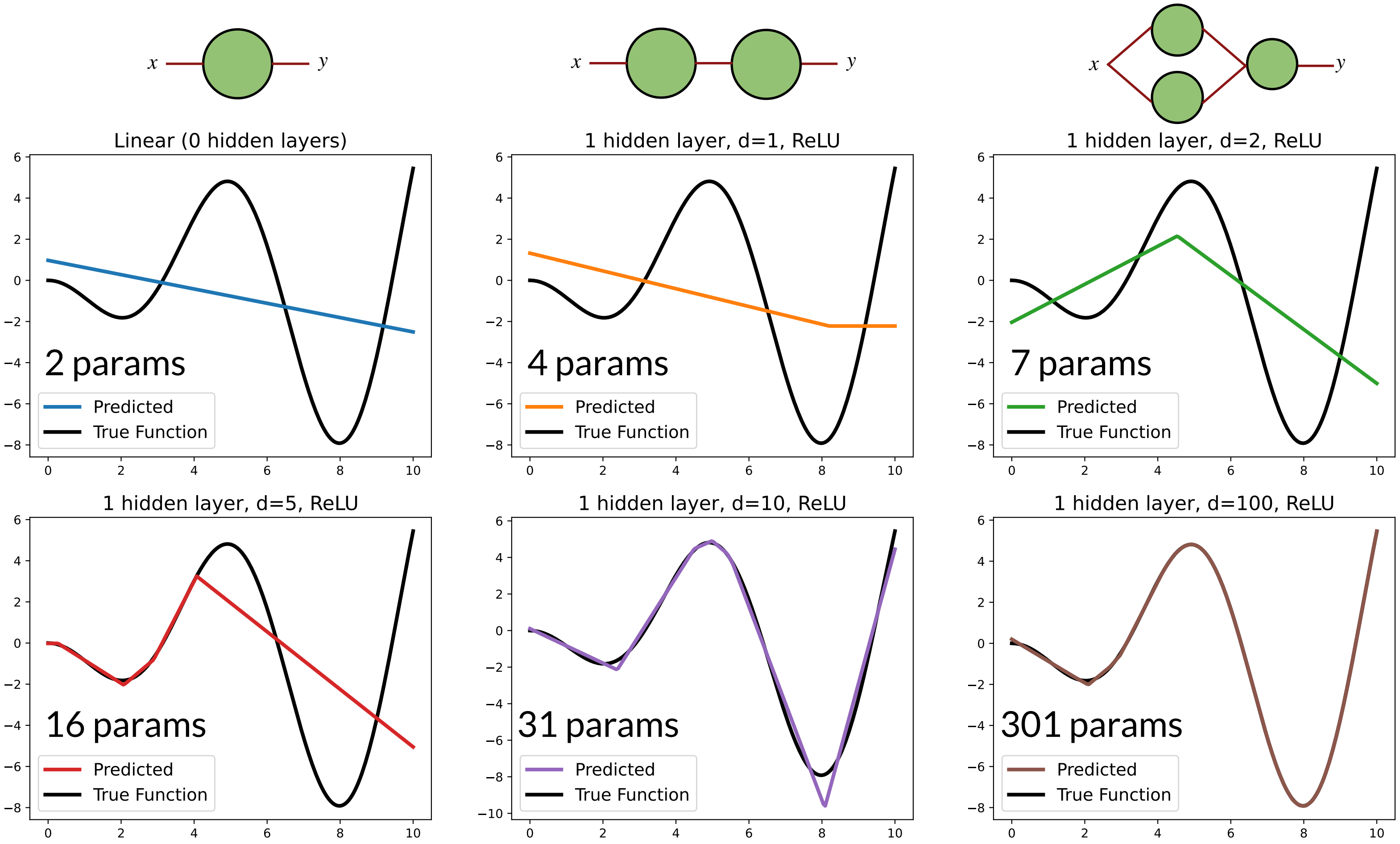


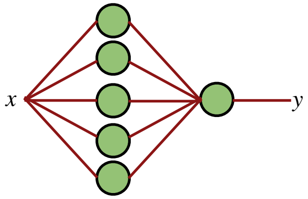
We can further add complexity by introducing **hidden layers**

- Intermediate computations between inputs and outputs
- **Neural networks: composition of** matrix multiplications, bias vectors, and non-linearities

$$\text{NN}(x) = \phi_1(W^1 \cdot \phi_0(W^0 x + b^0) + b^1)$$

Neural Networks: Width and Complexity





...

Wider neural networks => more hidden features => more complexity

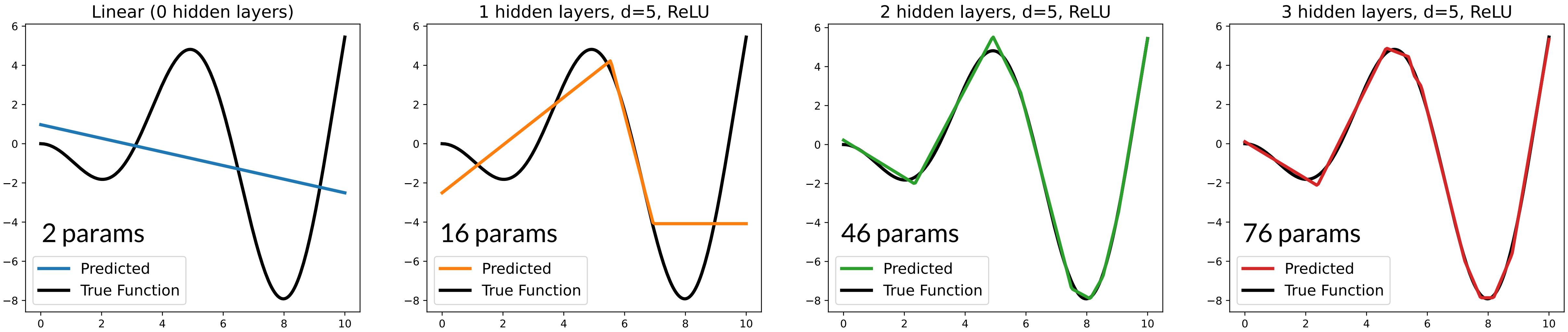
- More independent terms enter the final sum

$$\phi_1\left(\left(\sum_{j=1}^{m} w_{1j}^1 \cdot a_j\right) + b_1^1\right)$$

Neural Networks: Depth and Complexity

Deeper neural networks => composition of features => more complexity

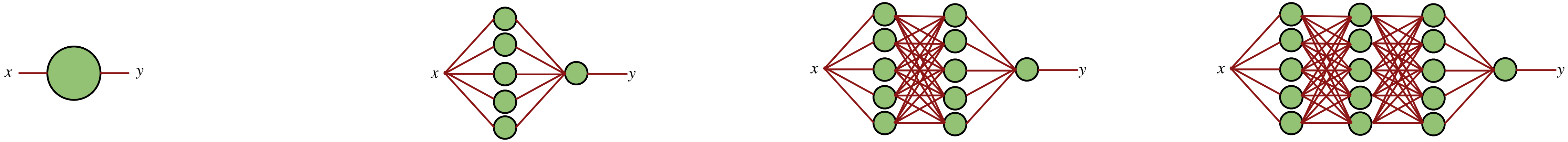
- E.g. internal coarse => fine featurization



$\mathrm{NN}(x) = \phi_0(W^0 x + b^0)$

$\mathrm{NN}(x) = \phi_1(W^1 \cdot \phi_0(W^0 x + b^0) + b^1)$

$\mathrm{NN}(x) = \phi_2(W^2 \cdot \phi_1(W^1 \cdot \phi_0(W^0 x + b^0) + b^1) + b^2)$

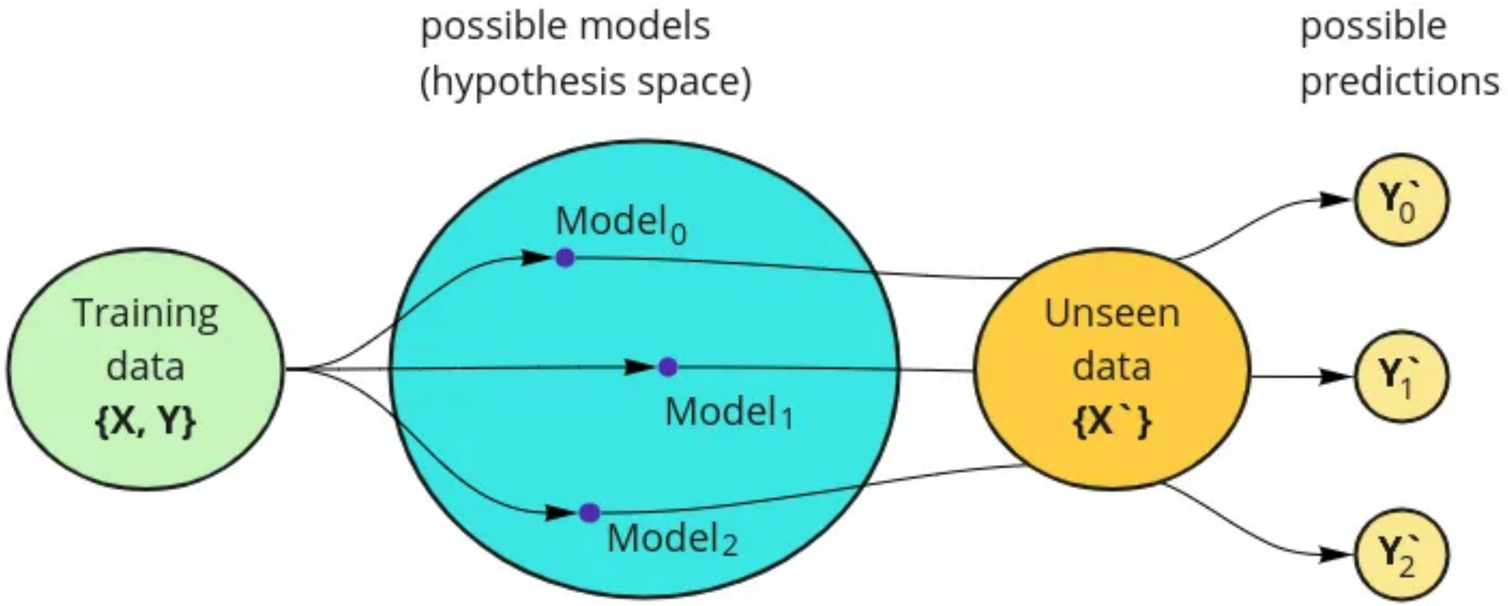


Neural Network Architectures

Structured NNs and Inductive Biases

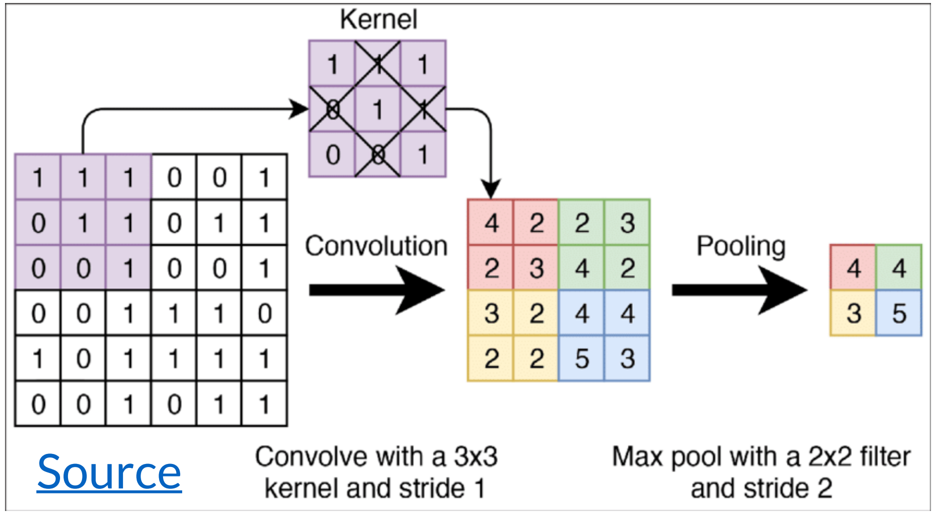
Multi-layer perceptrons (MLPs) or fully connected networks are great for vector inputs/outputs

- But, sometimes our data has structure we want to encode
- Putting assumptions into our model architectures and training can help with learning and generalization — this is called **inductive bias**



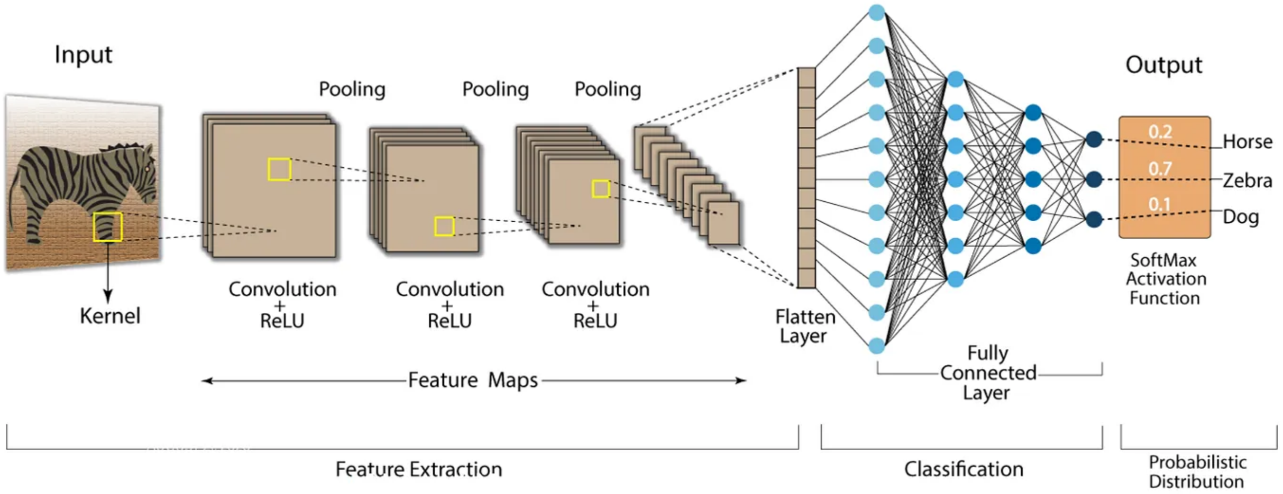
Source

Convolutional Neural Networks

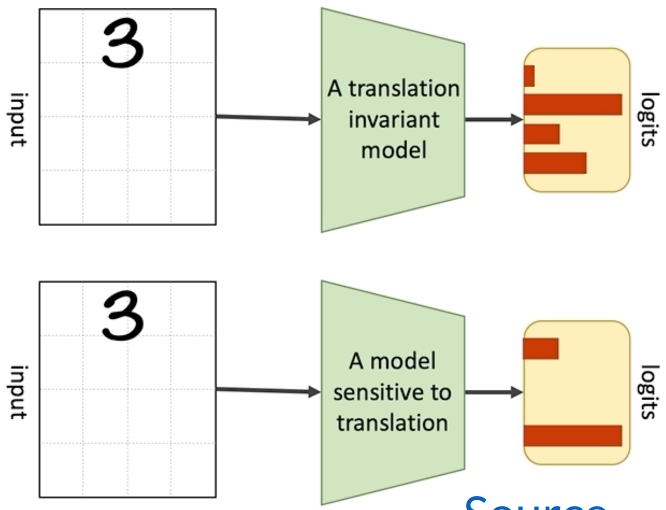


Convolutional Neural Networks (CNNs) are a classic example for images (e.g. in neutrino physics)

- Introduce learned **filter (kernel) matrices** and **convolution operations** that slide the filter over the input, + **pooling** to spatially downsample
- Encodes multi-scale, translationally invariant nature of images!



Source

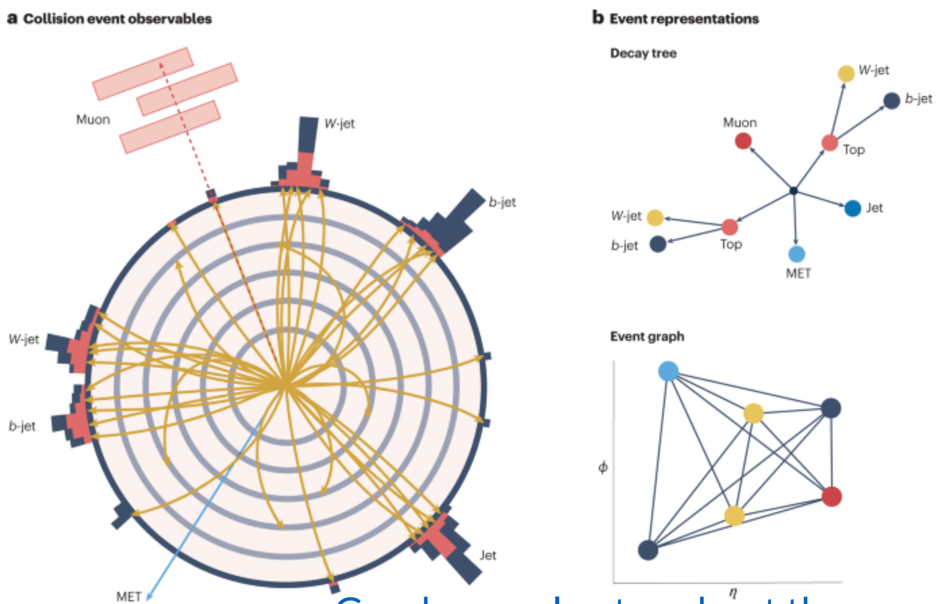


Source

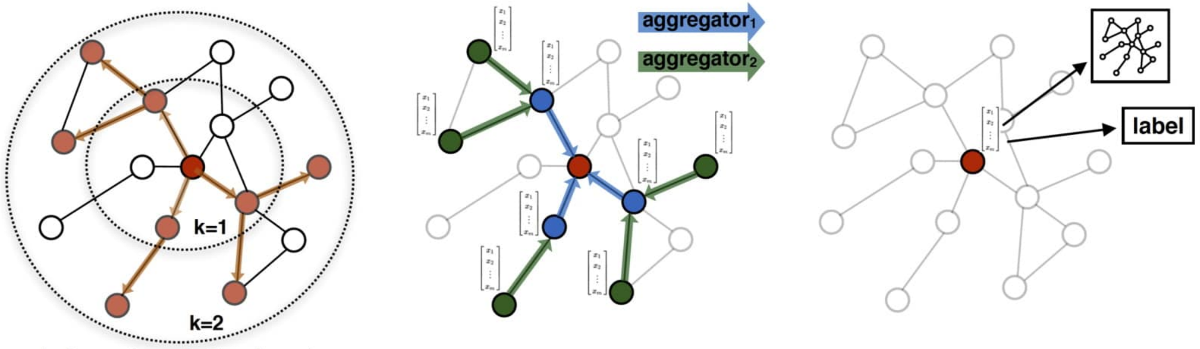
Graph Neural Networks

Graph neural networks (GNNs) encode structured data by representing information via **node** and **edge** features

- Learning respects **neighborhood structure** and **graph topology**
  - Information flows along graph edges
  - Features are aggregated from neighboring nodes



[Graph neural networks at the Large Hadron Collider]



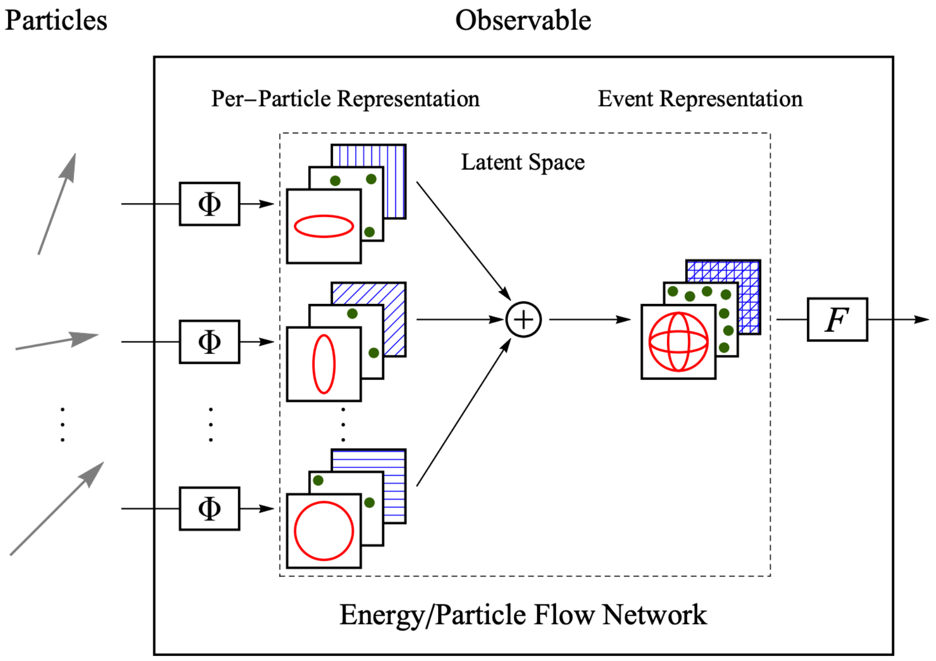
1. Sample neighborhood
2. Aggregate feature information from neighbors
3. Predict graph context and label using aggregated information

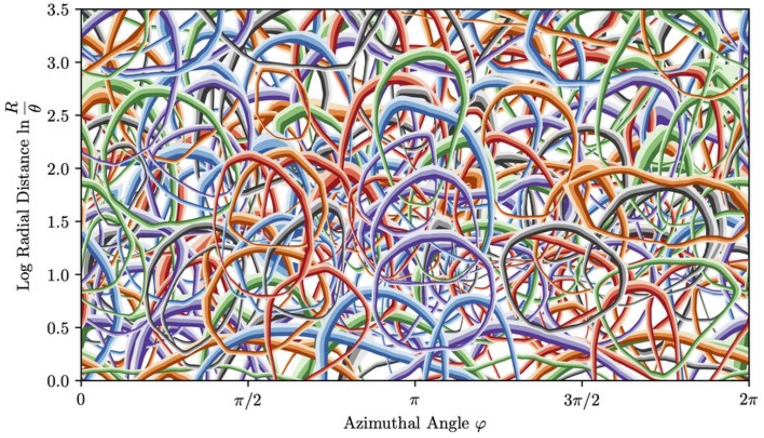
[Source]

Deep Sets

Particle physics problem: can have a **variable number** of particles in an event

- Historically: treat as a **sequence** and use a **recurrent neural network** (e.g. RNNIP)
    - Arbitrary length, but requires imposing an ordering
- Deep sets use a permutation/length invariant **summing** operation to overcome this problem (e.g. DIPS)
    - Operate on a variable length set of particles, not a sequence



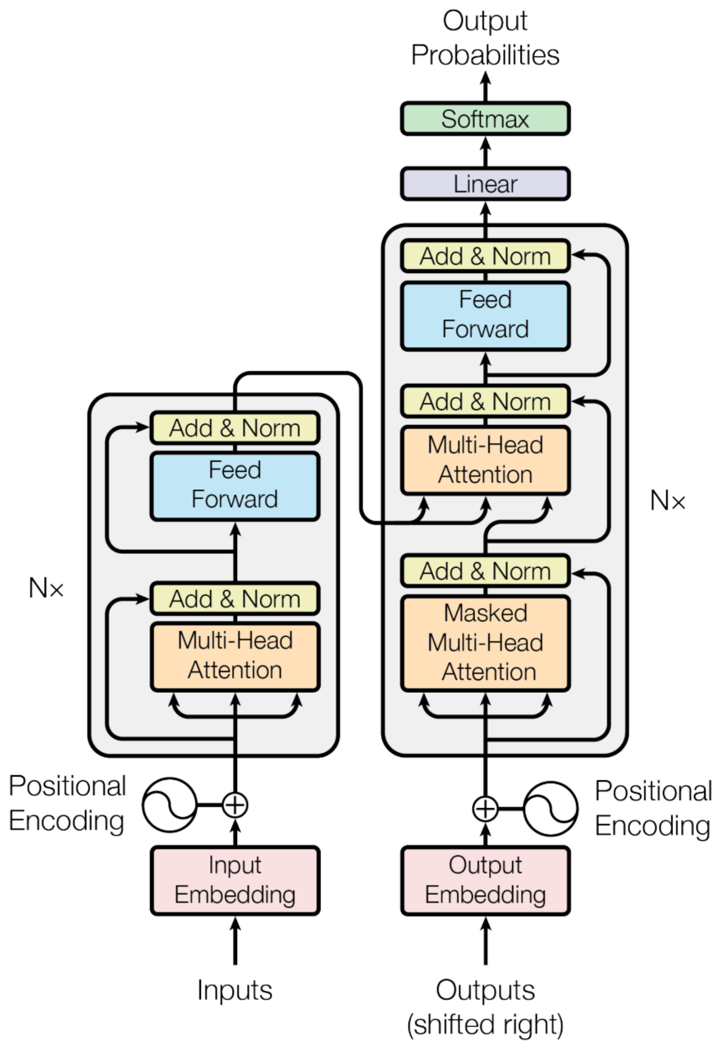


Deep Sets for Particle Jets

Transformers

Transformers are the architecture behind a lot of the current AI hype (e.g. most large language models)

- Excellent at sequence modeling (e.g. language)…and more (including flavor tagging)!
  - Have (mostly) replaced recurrent neural networks
- NB:
  - Deep sets still useful: permutation invariance
  - GNNs still useful: explicit graph structure



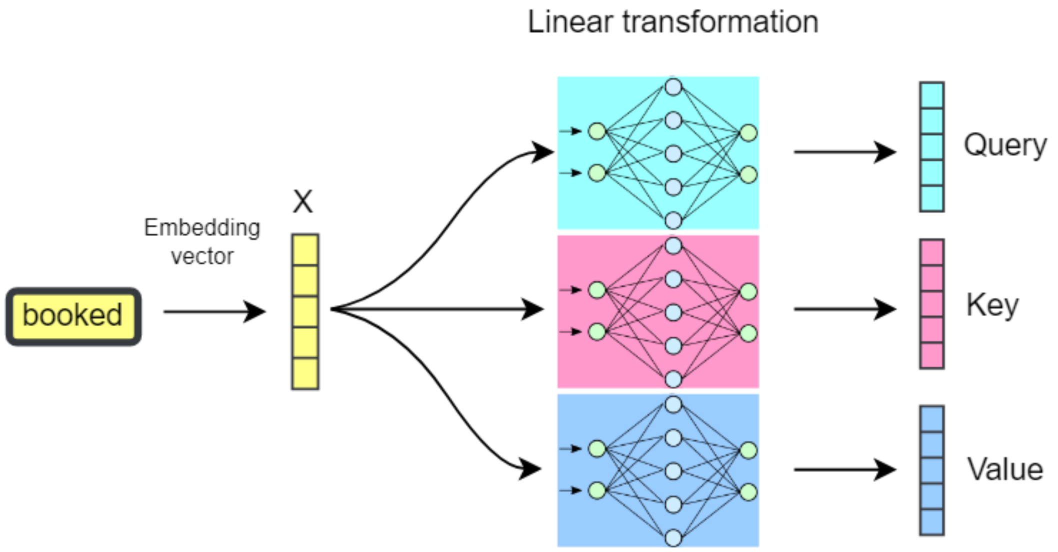
Attention Is All You Need

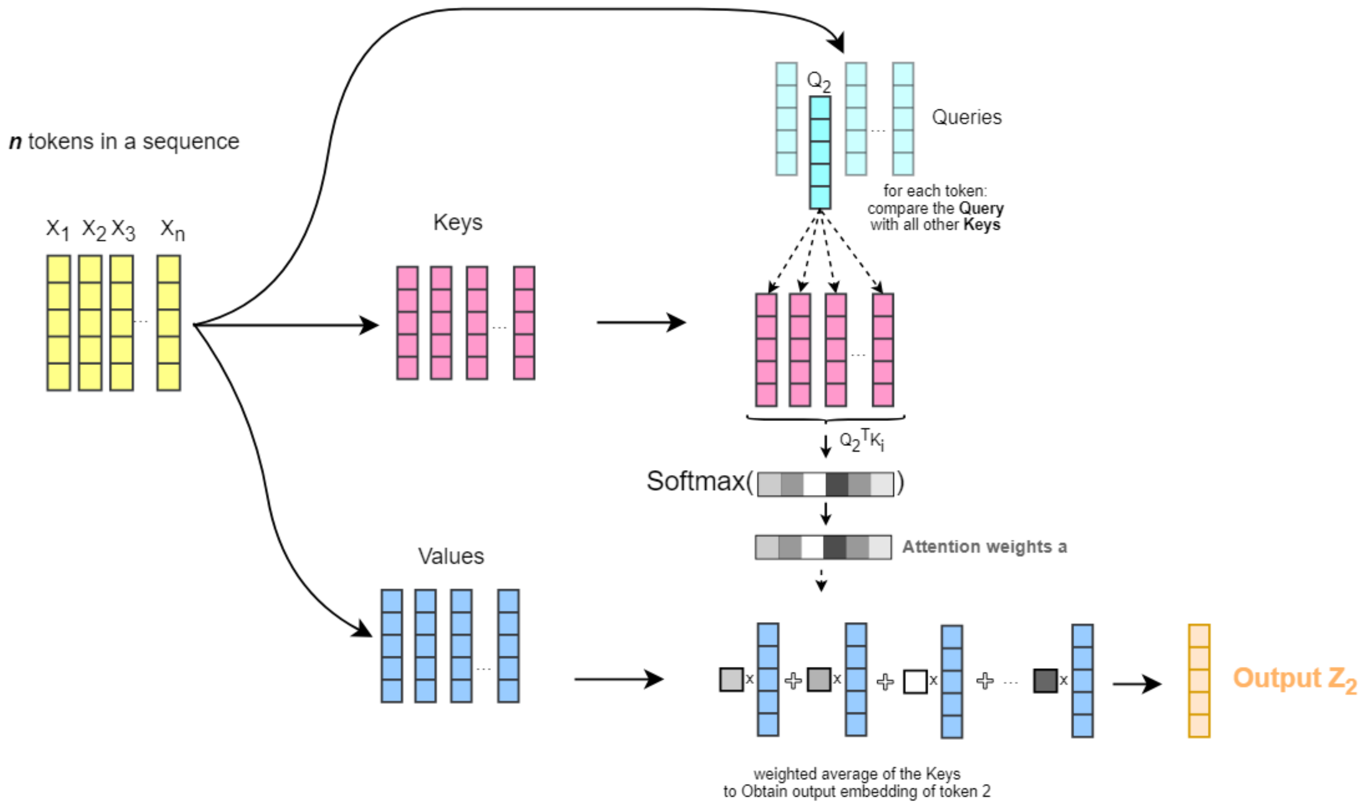
Transformers

Transformers take in full sequence of input **tokens** at once

- Key piece: **attention mechanism** that learns relationships between tokens

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$



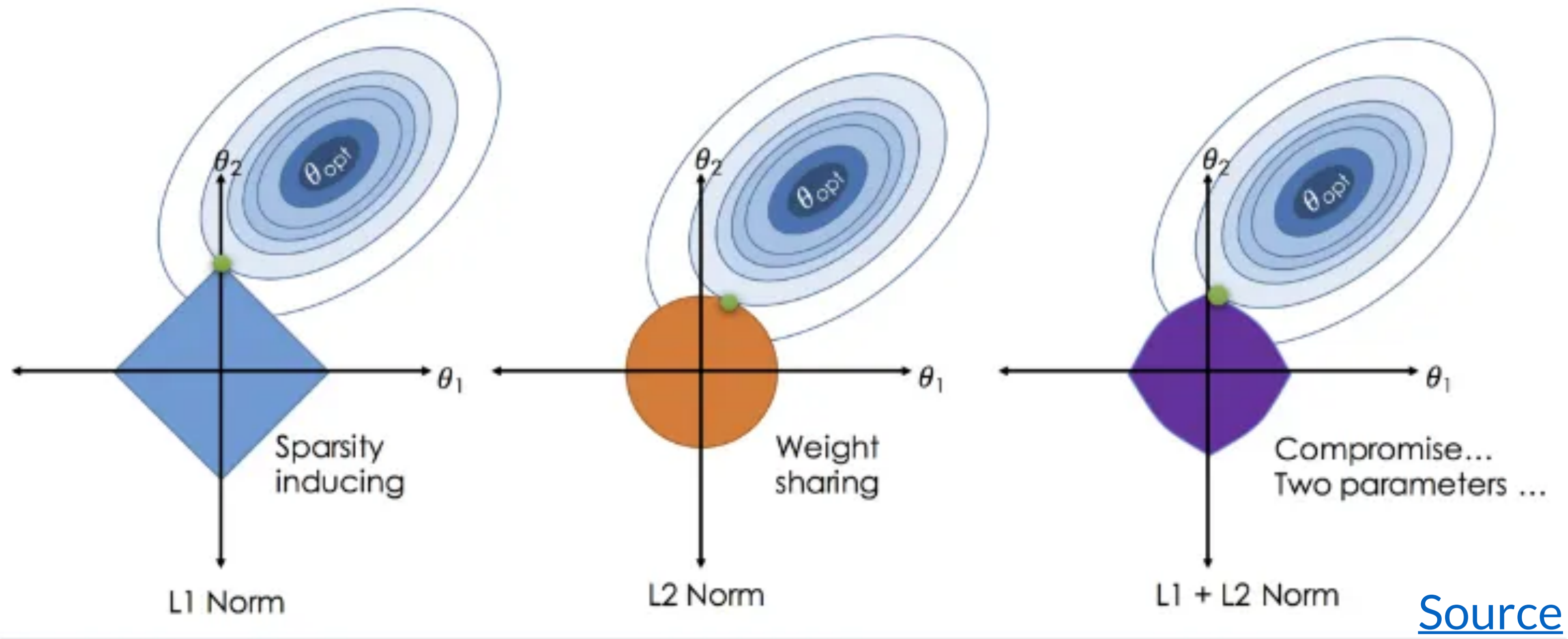


Source

Inductive Biases and Regularization

Network architectures aren't the only way to encode assumptions

- We can add **regularization** terms to our loss functions
- Pushes training towards e.g. sparsity (L1/Lasso), stability (L2/Ridge), smoothness (e.g. total variation/gradient)



Source

$$\mathscr{L}(w, \mathscr{D}) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i, w) - y_i)^2 + \lambda_1 \cdot \sum_j |w_j| + \lambda_2 \cdot \sum_j w_j^2$$

λ = Regularization strength

Generative Model Architectures

Reminder: Generative Modeling

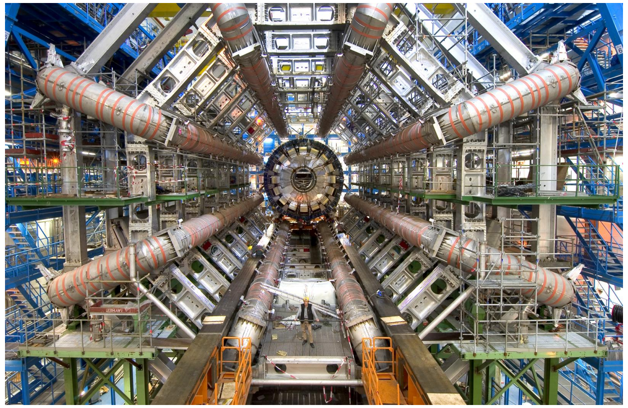
Generative models aim to learn the probabilistic distribution of a dataset

- Often the goal is to then **sample** from that dataset to generate realistic (data-like) outputs

Prior (latent) distribution

$$p(z)$$

Easy to sample from (e.g. standard normal)



Generative distribution

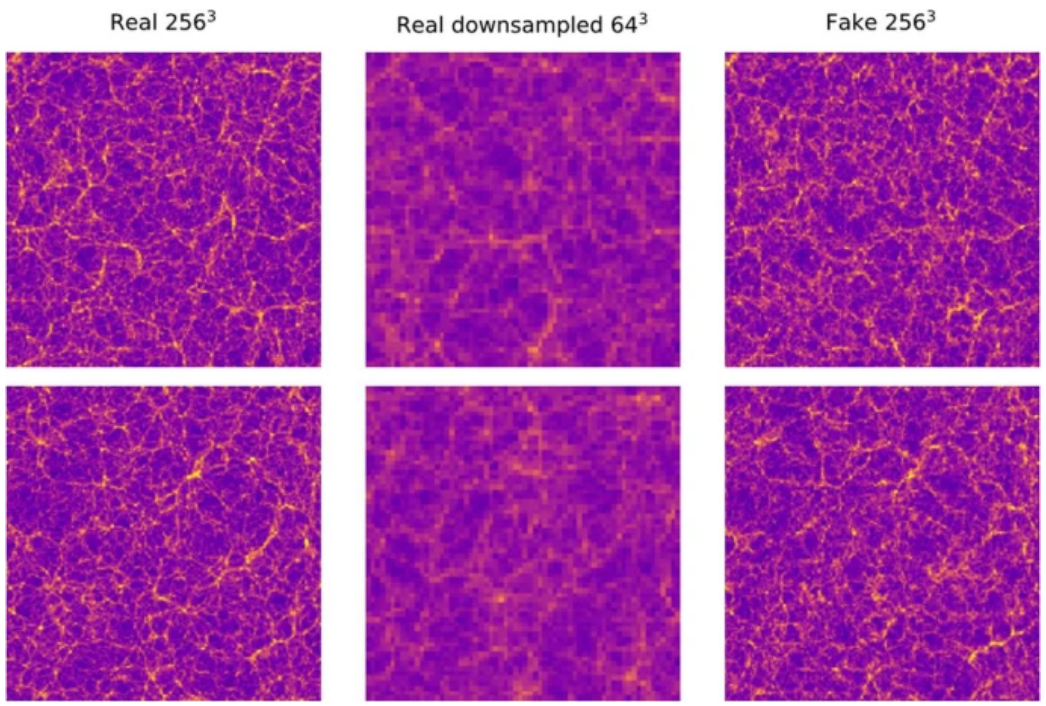
$$p(x \mid z)$$

$p_{data}(x)$ hard to sample from. Model trained to match data distribution, given sampled z

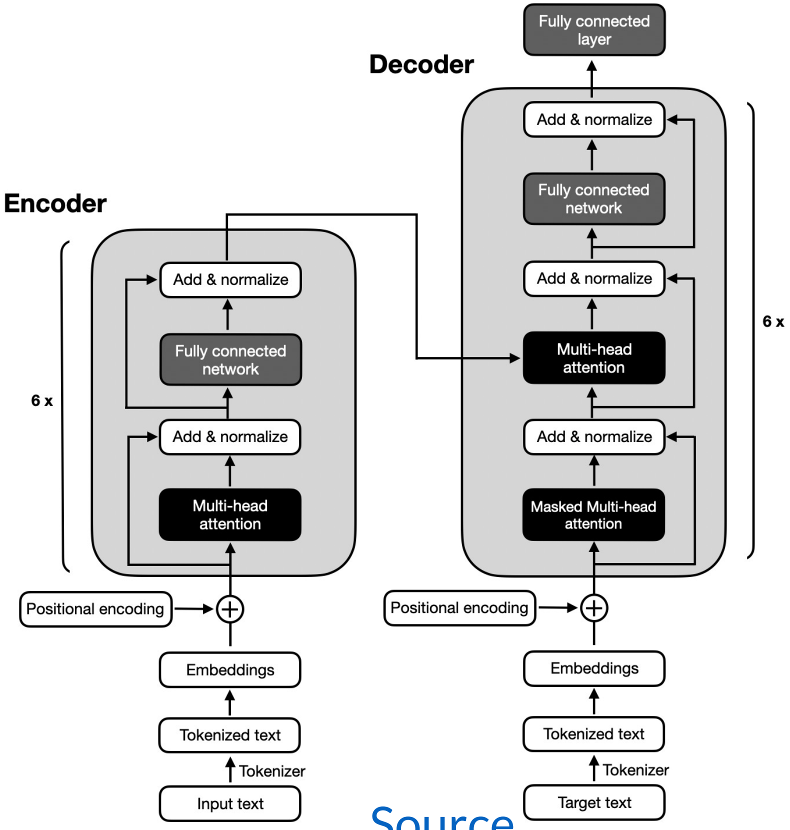
Generative Modeling Comments

Generative models can make use of above architectures!

- e.g. transformers used for GPT models (next token prediction)
- Much development here for **image data** but applications in e.g music, physics simulation



N-body simulations

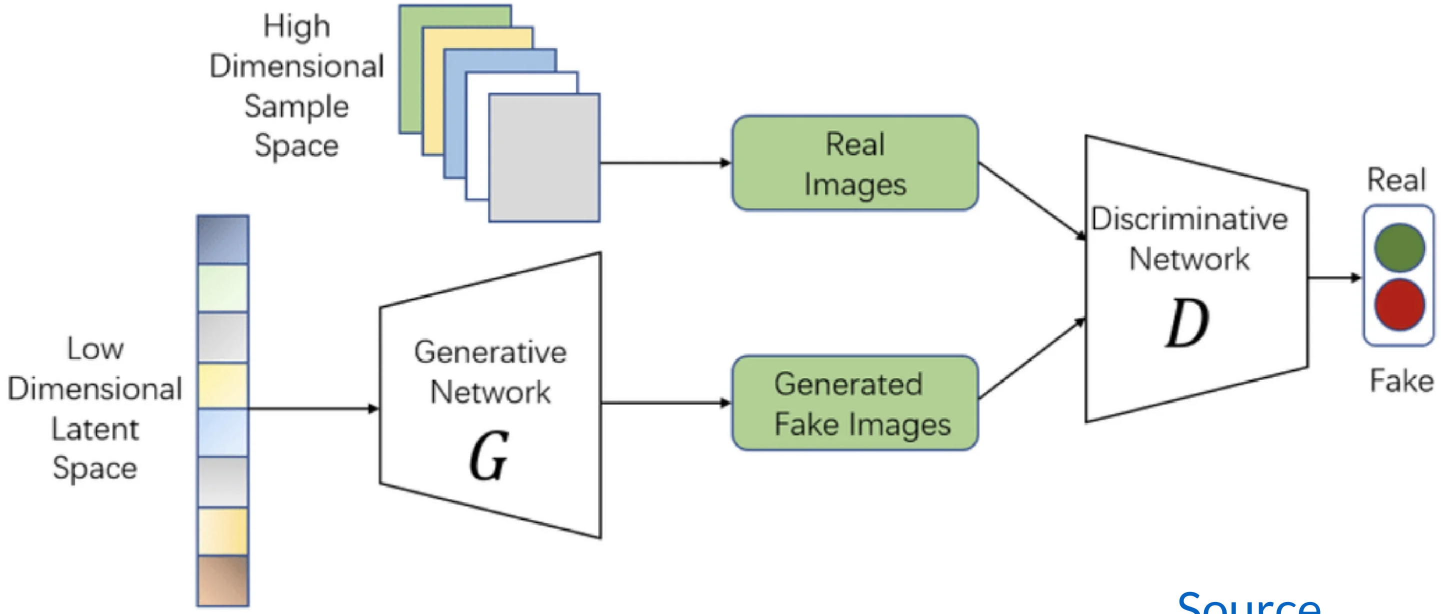


Source

Generative Models: GANs

Generative adversarial networks learn by "fighting" two neural networks

- **Generator** produces fake data from latent samples
- **Discriminator** tries to distinguish real data from fake data
- Trained generator then used to produce high quality data
- NB: High quality samples, but harder to train than some others



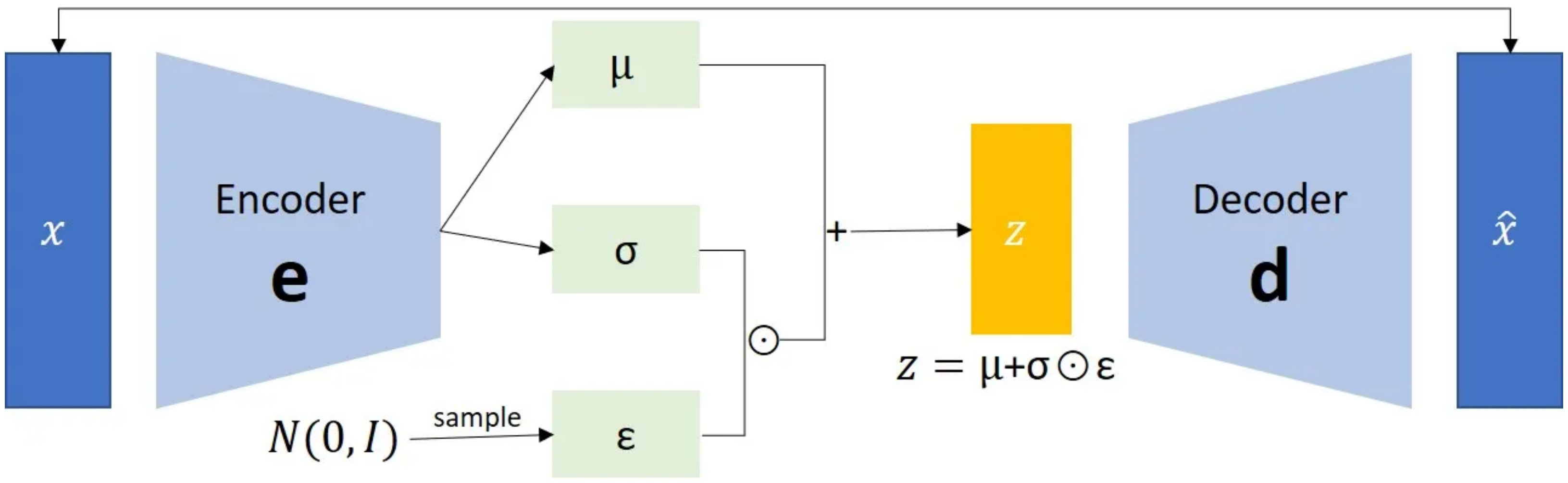
Source

Generative Models: Variational Autoencoders

Variational autoencoders are a probabilistic version of autoencoders

- Trained in a similar way (reproduce inputs), but (1) introduce stochastic latent variables (sampling) (2) add an additional **KL divergence term** to the loss to make the latent distribution close to a standard prior (see variational inference, evidence lower bound)
- Latent space => interpretability

Minimize 1: $(x - \hat{x})^2$



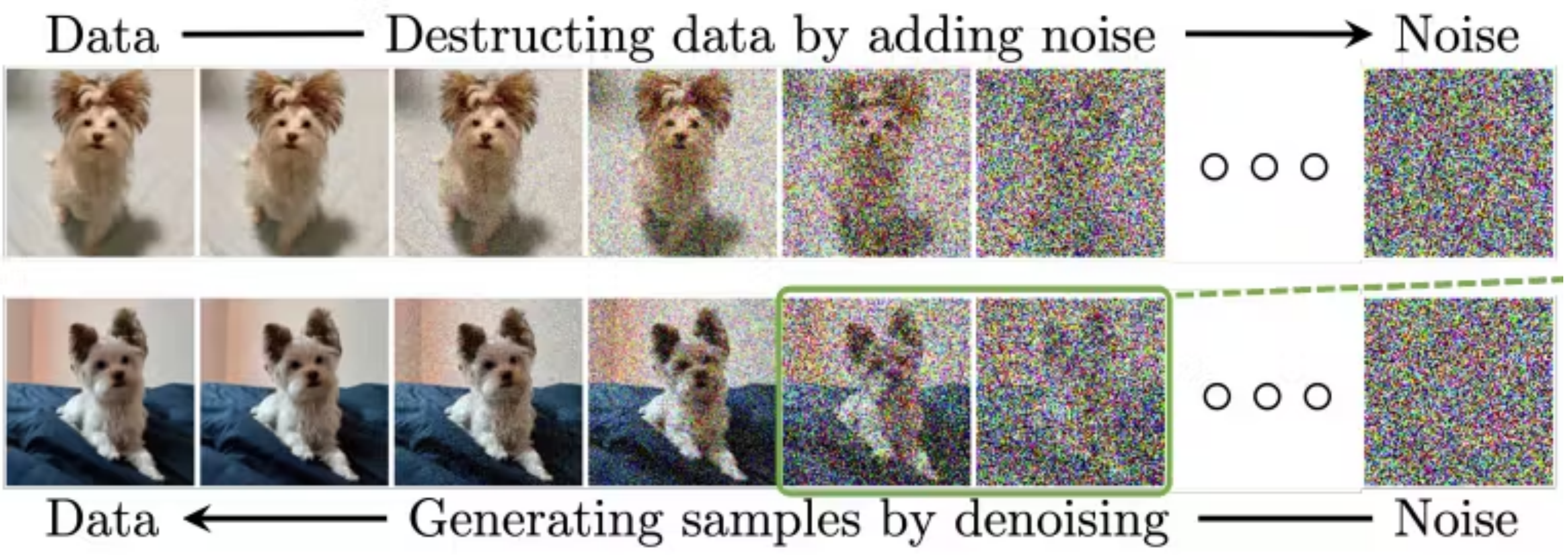
Minimize 2: $\frac{1}{2}\sum_{i=1}^{N}(\exp(\sigma_i) - (1+\sigma_i) + {\mu_i}^2)$

Source

Generative Models: Diffusion

Diffusion models: based on forward/reverse noising process. E.g. denoising diffusion probabilistic models (DDPM)

- **During training:** run forward diffusion with a random time step t to get a noisy image (known noise). Model predicts noise added to the noisy image at t
- **During inference:** sample noise and run reverse diffusion
- Also **score-function** based approaches with slightly different procedure.



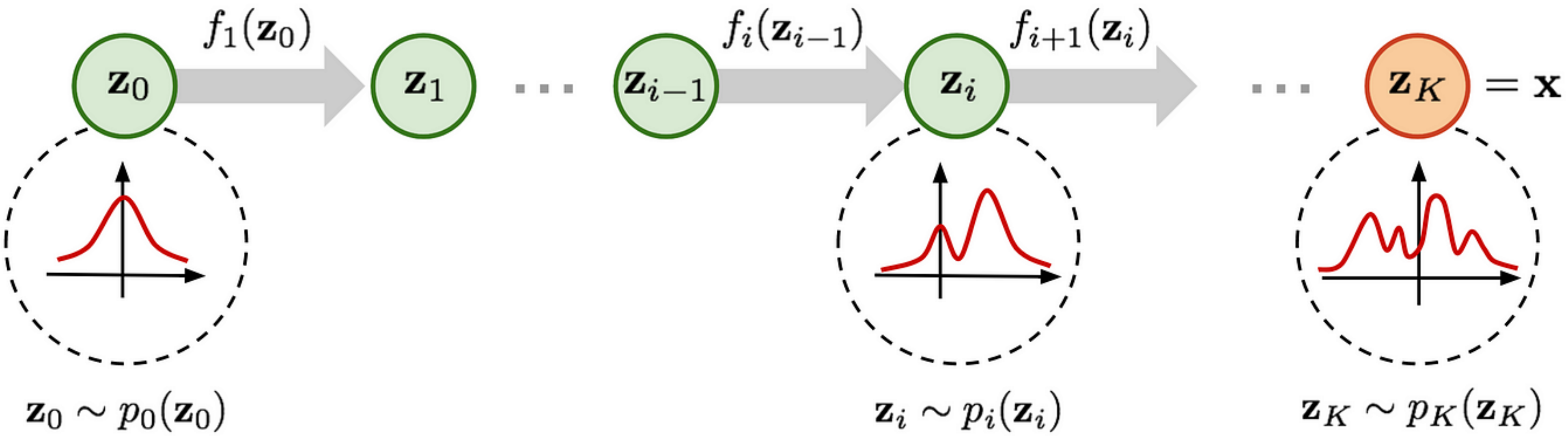
NB: State-of-the-art, but slow to sample.

Source

Generative Models: Normalizing Flows

Normalizing flows: learn invertible sequence of transformations between simple (easy to sample) and complex distribution

- e.g. parameterized by "shifts and scales" (affine transformations) as in RealNVP
- Training: transform data to latent distribution, minimize NLL
- Exact likelihoods, and fast! But less expressive than, e.g. diffusion. Check out conditional flow matching for a method somewhere between the two!

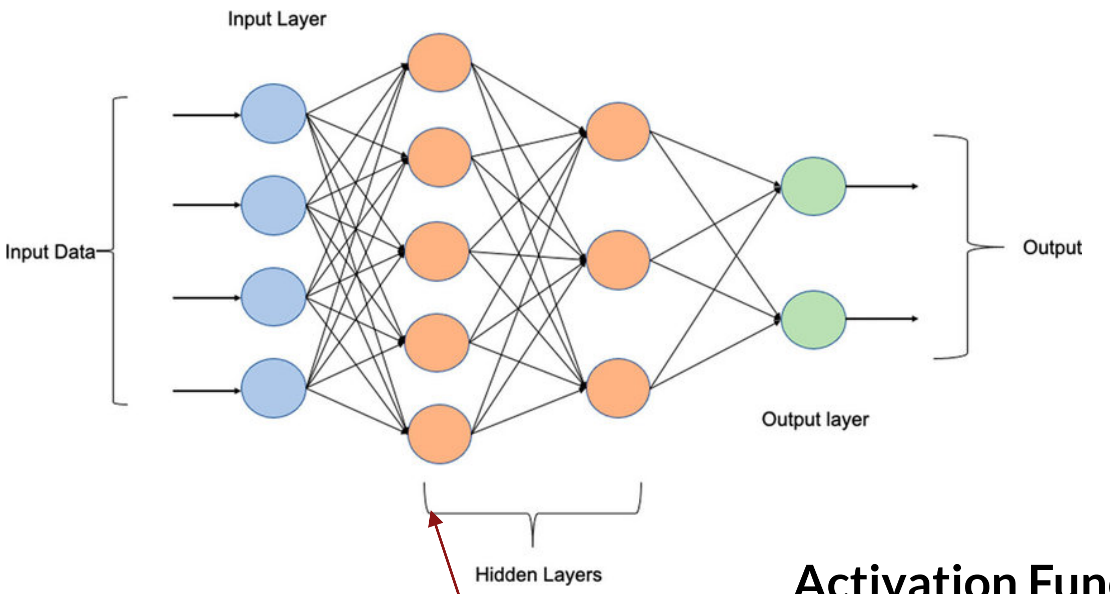


Source

How do machines learn?

How do machines learn?

When we train a neural network, what's happening?



Activation Function

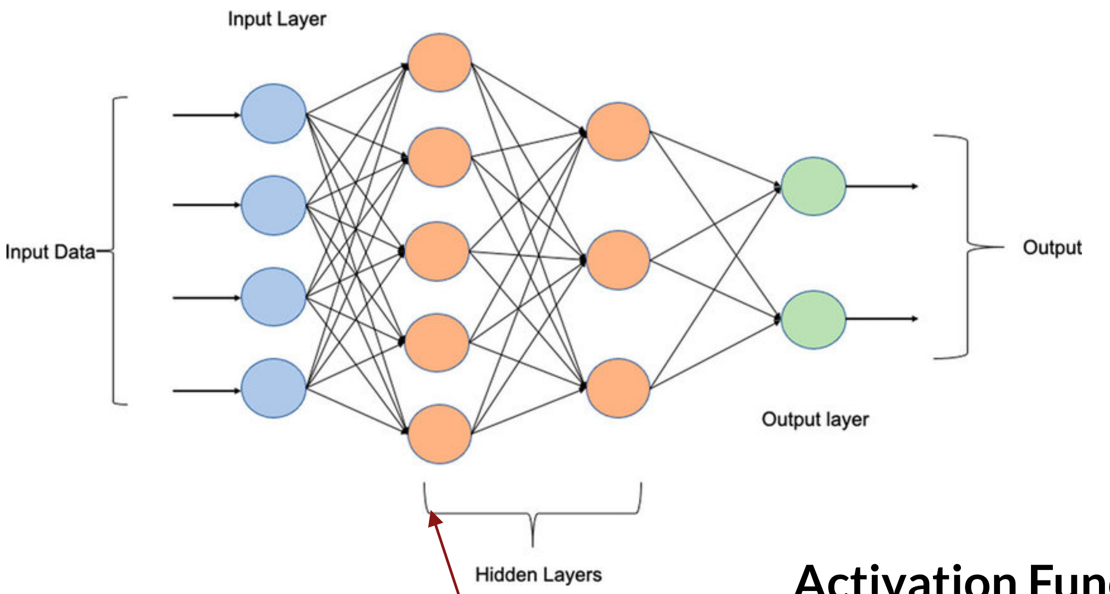
$$h^{(i)}(\mathbf{x}) = \phi^{(i)}(\mathbf{w}^T\mathbf{x} + b)$$

Weights

Biases

How do machines learn?

When we train a neural network, what's happening?



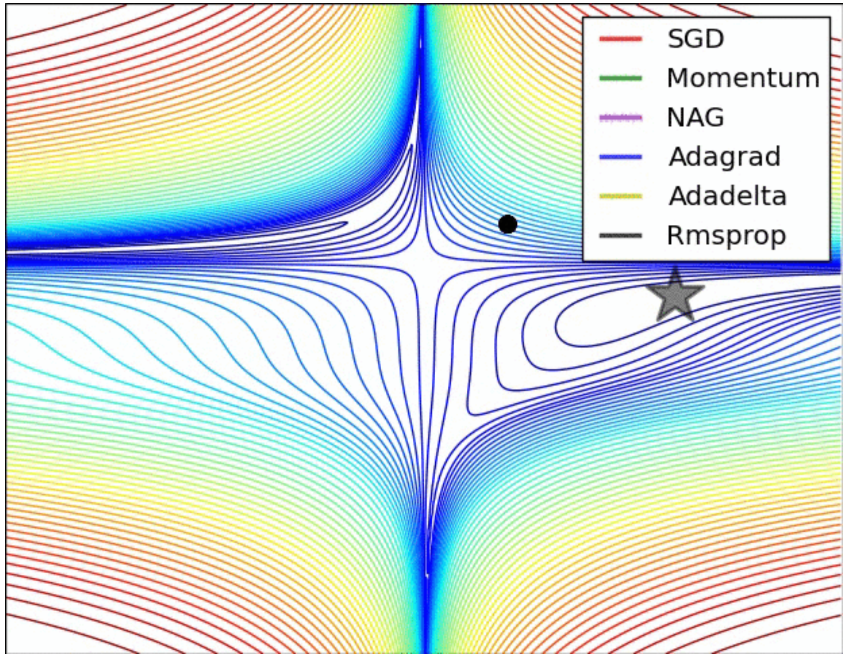
Activation Function

$$h^{(i)}(\mathbf{x}) = \phi^{(i)}(\mathbf{w}^T\mathbf{x} + b)$$

Weights

Biases

Ruder, link



NN weights and biases are adjusted to **minimize a loss function** using an **optimizer**

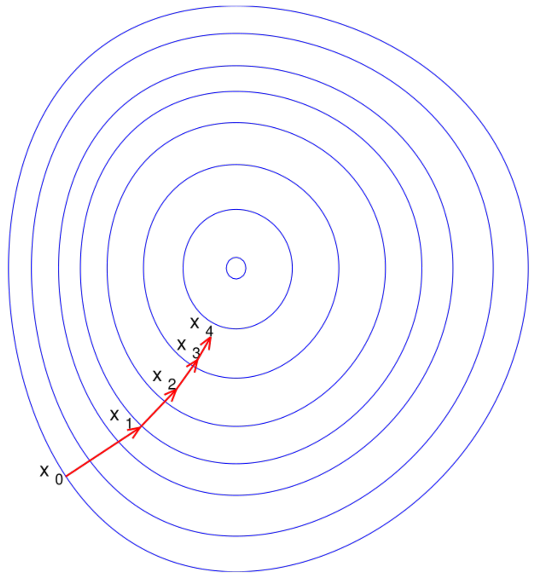
Breaking down an optimizer

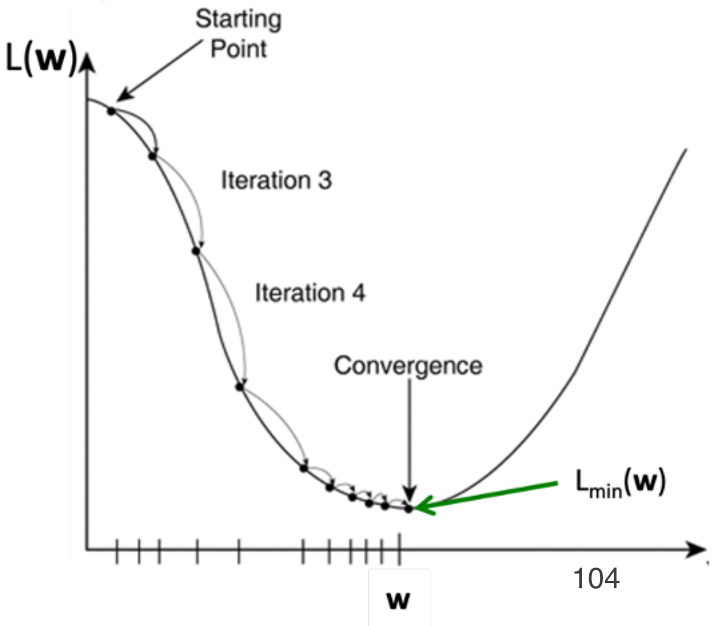
E.g. supervised learning:

- **Data** with labels: $\{(x_i, y_i)\}_{i=1}^{N}$
- **Model:** $h(x_i; \mathbf{w})$ (parameters $\mathbf{w}$)
- Element-wise **loss** (e.g. squared error, cross-entropy):
  $\mathscr{L}_i(\mathbf{w}) \equiv \mathscr{L}(y_i, h(x_i; \mathbf{w}))$

Gradient descent: Minimize total loss $\mathscr{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \mathscr{L}_i(\mathbf{w})$. At iteration t:

- Compute gradient $\nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}^{(t)})$
- Update model weights as: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}^{(t)})$, where η is a learning rate controlling the size of the gradient step.
- Negative gradient gives (local) direction of steepest descent





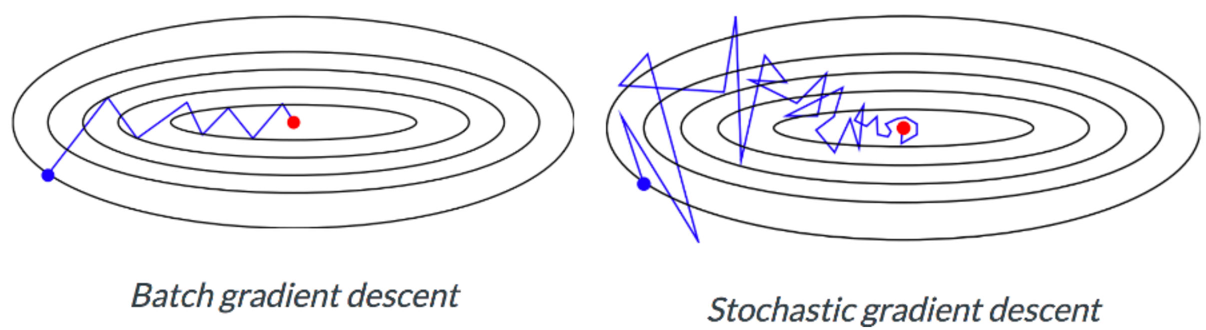
Breaking down an optimizer

Gradient descent is the foundation of most common optimizers

- **In practice:** stochastic/mini-batch gradient descent is used
  - Cost of full gradient descent scales with the number of samples:

$$\nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{w}} \mathscr{L}_i(\mathbf{w})$$

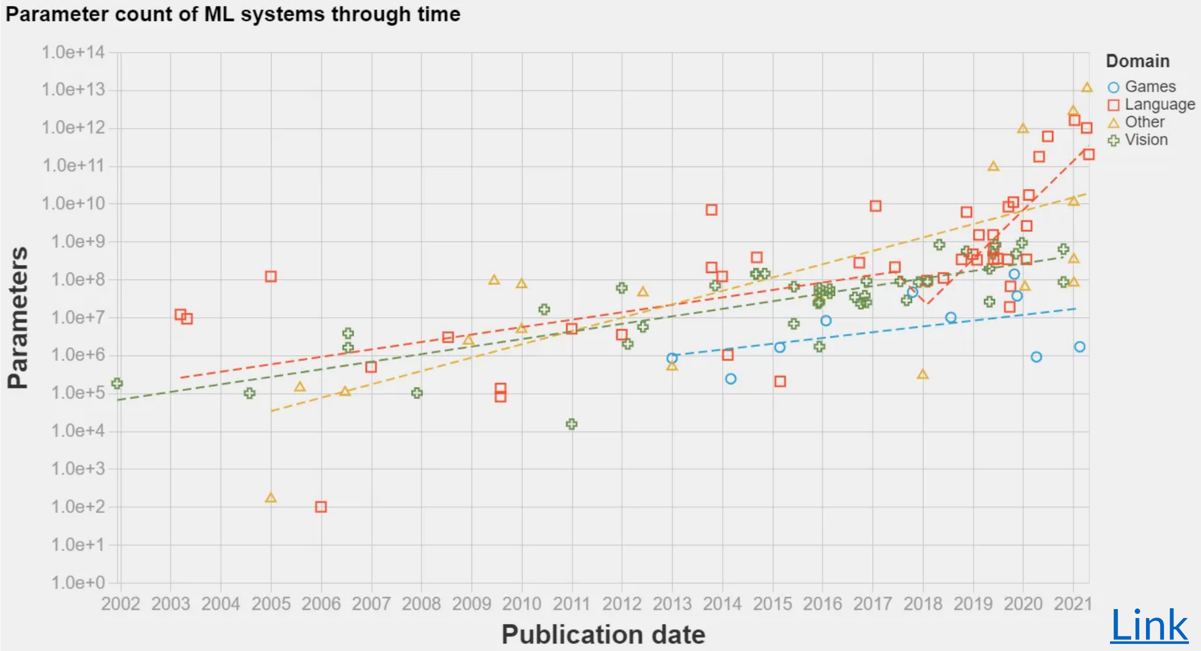
  - Instead, compute each update over a randomly sampled data point/batch of points
    - Unbiased estimator of full gradient: on average moves in the right direction
- **Benefits:** less costly to compute/faster, randomness may help break out of local minima
- Common extensions: momentum, Adam, RMSProp, …



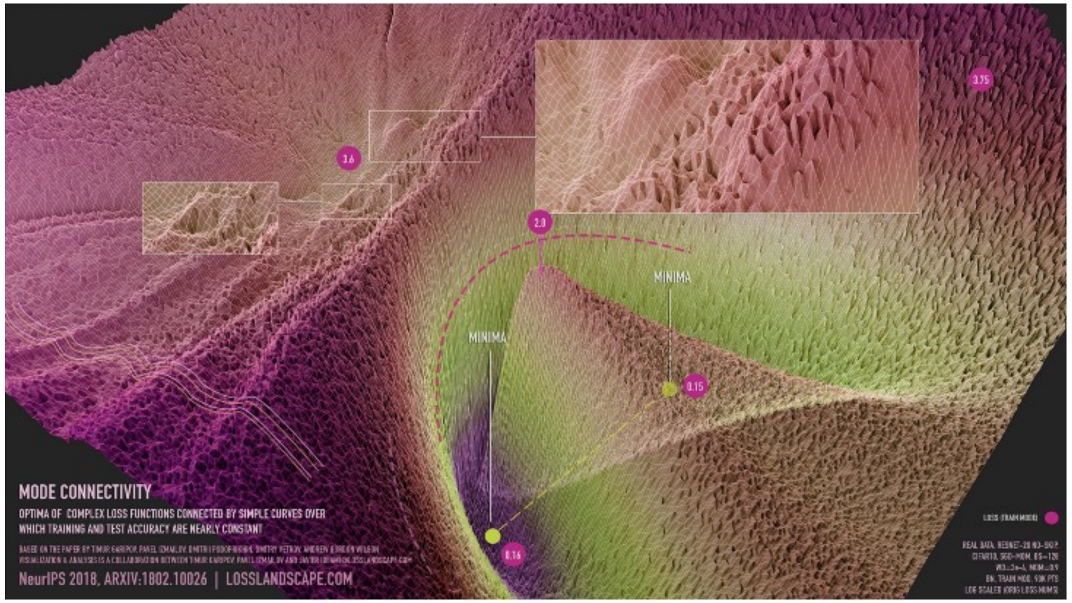
Why gradients?

Gradient-based optimizers have been used to train models with (at least) $O(10^{12})$ parameters

- => works well for high dimensional optimization
- Batch methods/SGD => scalable with dataset size
- Gradients are **easy to compute**



Link



https://arxiv.org/abs/1802.10026

B Details of Model Training

To train all versions of GPT-3, we use Adam with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$

https://arxiv.org/abs/2005.14165

How to Compute Gradients

Popularity of gradient-based methods => good toolkits for computing gradients!

- Fundamental component of common ML libraries
- All use a common technique: **automatic differentiation**
  - a.k.a. **backpropagation** (for neural networks), autodiff, autograd, AD

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts

Nature **323**, 533-536 (1986)

PyTorch

TensorFlow

How does it work: Automatic Differentiation

Neural Networks are Code

Example of a neural network in PyTorch

```
# Multi-layer perceptron
mlp = MLP(n_hidden=n_hidden, hidden_dim=hidden_dim)

# Optimizer
optimizer = torch.optim.Adam(mlp.parameters(), lr=lr)

# Mean squared error
loss_fn = torch.nn.MSELoss()

losses = []
for _ in range(n_epochs):

    # Shuffle data
    idxs = np.random.permutation(len(norm_x))

    # Make predictions
    out = mlp(norm_x[idxs])

    # Calculate loss
    loss = loss_fn(out, norm_y[idxs])

    # Zero out gradients
    optimizer.zero_grad()

    # Compute gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

```
mlp(norm_x[0:1])
```

```
tensor([[-1.6607]], grad_fn=<AddmmBackward0>)
```

```
list(mlp.parameters())
```

```
[Parameter containing:
 tensor([[ 0.5748],
         [ 0.4250],
         [ 0.6559],
         [ 0.0270],
         [ 0.2925],
         [ 0.0268],
         [-0.9413],
         [-0.8923],
         [-0.6783],
         [ 0.3521],
         [-0.3701],
         [ 0.7082],
         [ 0.8256],
         [ 0.1497],
         [-0.7315],
         [-1.0628],
         [ 0.3396],
         [ 0.8656],
         [ 0.0636],
         [-0.6879]], requires_grad=True),
 Parameter containing:
 tensor([-0.3643,  0.5639,  1.0288, -0.5873,  0.5042, -0.5881,  0.3506,  0.5615,
          0.6442, -0.9721,  0.0830,  0.3054, -0.7737, -0.5471,  0.8950, -0.8749,
         -0.2152,  0.6729, -0.1408,  0.9366], requires_grad=True),
```

Neural Networks are Code

Example of a neural network in PyTorch

```python
# Multi-layer perceptron
mlp = MLP(n_hidden=n_hidden, hidden_dim=hidden_dim)

# Optimizer
optimizer = torch.optim.Adam(mlp.parameters(), lr=lr)

# Mean squared error
loss_fn = torch.nn.MSELoss()

losses = []
for _ in range(n_epochs):

    # Shuffle data
    idxs = np.random.permutation(len(norm_x))

    # Make predictions
    out = mlp(norm_x[idxs])

    # Calculate loss
    loss = loss_fn(out, norm_y[idxs])

    # Zero out gradients
    optimizer.zero_grad()

    # Compute gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

What's happening when we call `loss.backward()`?

```python
mlp(norm_x[0:1])
```

```
tensor([[-1.6607]], grad_fn=<AddmmBackward0>)
```

```python
list(mlp.parameters())
```

```
[Parameter containing:
 tensor([[ 0.5748],
         [ 0.4250],
         [ 0.6559],
         [ 0.0270],
         [ 0.2925],
         [ 0.0268],
         [-0.9413],
         [-0.8923],
         [-0.6783],
         [ 0.3521],
         [-0.3701],
         [ 0.7082],
         [ 0.8256],
         [ 0.1497],
         [-0.7315],
         [-1.0628],
         [ 0.3396],
         [ 0.8656],
         [ 0.0636],
         [-0.6879]], requires_grad=True),
 Parameter containing:
 tensor([-0.3643,  0.5639,  1.0288, -0.5873,  0.5042, -0.5881,  0.3506,  0.5615,
          0.6442, -0.9721,  0.0830,  0.3054, -0.7737, -0.5471,  0.8950, -0.8749,
         -0.2152,  0.6729, -0.1408,  0.9366], requires_grad=True),
```

Neural Networks are Code

Example of a neural network in PyTorch

```python
# Multi-layer perceptron
mlp = MLP(n_hidden=n_hidden, hidden_dim=hidden_dim)

# Optimizer
optimizer = torch.optim.Adam(mlp.parameters(), lr=lr)

# Mean squared error
loss_fn = torch.nn.MSELoss()

losses = []
for _ in range(n_epochs):

    # Shuffle data
    idxs = np.random.permutation(len(norm_x))

    # Make predictions
    out = mlp(norm_x[idxs])

    # Calculate loss
    loss = loss_fn(out, norm_y[idxs])

    # Zero out gradients
    optimizer.zero_grad()

    # Compute gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

What's happening when we call `loss.backward()`?

What is this `grad_fn`?

```python
mlp(norm_x[0:1])
```

```
tensor([[-1.6607]], grad_fn=<AddmmBackward0>)
```

```python
list(mlp.parameters())
```

```
[Parameter containing:
 tensor([[ 0.5748],
         [ 0.4250],
         [ 0.6559],
         [ 0.0270],
         [ 0.2925],
         [ 0.0268],
         [-0.9413],
         [-0.8923],
         [-0.6783],
         [ 0.3521],
         [-0.3701],
         [ 0.7082],
         [ 0.8256],
         [ 0.1497],
         [-0.7315],
         [-1.0628],
         [ 0.3396],
         [ 0.8656],
         [ 0.0636],
         [-0.6879]], requires_grad=True),
 Parameter containing:
 tensor([-0.3643,  0.5639,  1.0288, -0.5873,  0.5042, -0.5881,  0.3506,  0.5615,
          0.6442, -0.9721,  0.0830,  0.3054, -0.7737, -0.5471,  0.8950, -0.8749,
         -0.2152,  0.6729, -0.1408,  0.9366], requires_grad=True),
```

Neural Networks are Code

Example of a neural network in PyTorch

```python
# Multi-layer perceptron
mlp = MLP(n_hidden=n_hidden, hidden_dim=hidden_dim)

# Optimizer
optimizer = torch.optim.Adam(mlp.parameters(), lr=lr)

# Mean squared error
loss_fn = torch.nn.MSELoss()

losses = []
for _ in range(n_epochs):

    # Shuffle data
    idxs = np.random.permutation(len(norm_x))

    # Make predictions
    out = mlp(norm_x[idxs])

    # Calculate loss
    loss = loss_fn(out, norm_y[idxs])

    # Zero out gradients
    optimizer.zero_grad()

    # Compute gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

What's happening when we call `loss.backward()`?

What is this `grad_fn`?

```python
mlp(norm_x[0:1])
```

```
tensor([[-1.6607]], grad_fn=<AddmmBackward0>)
```

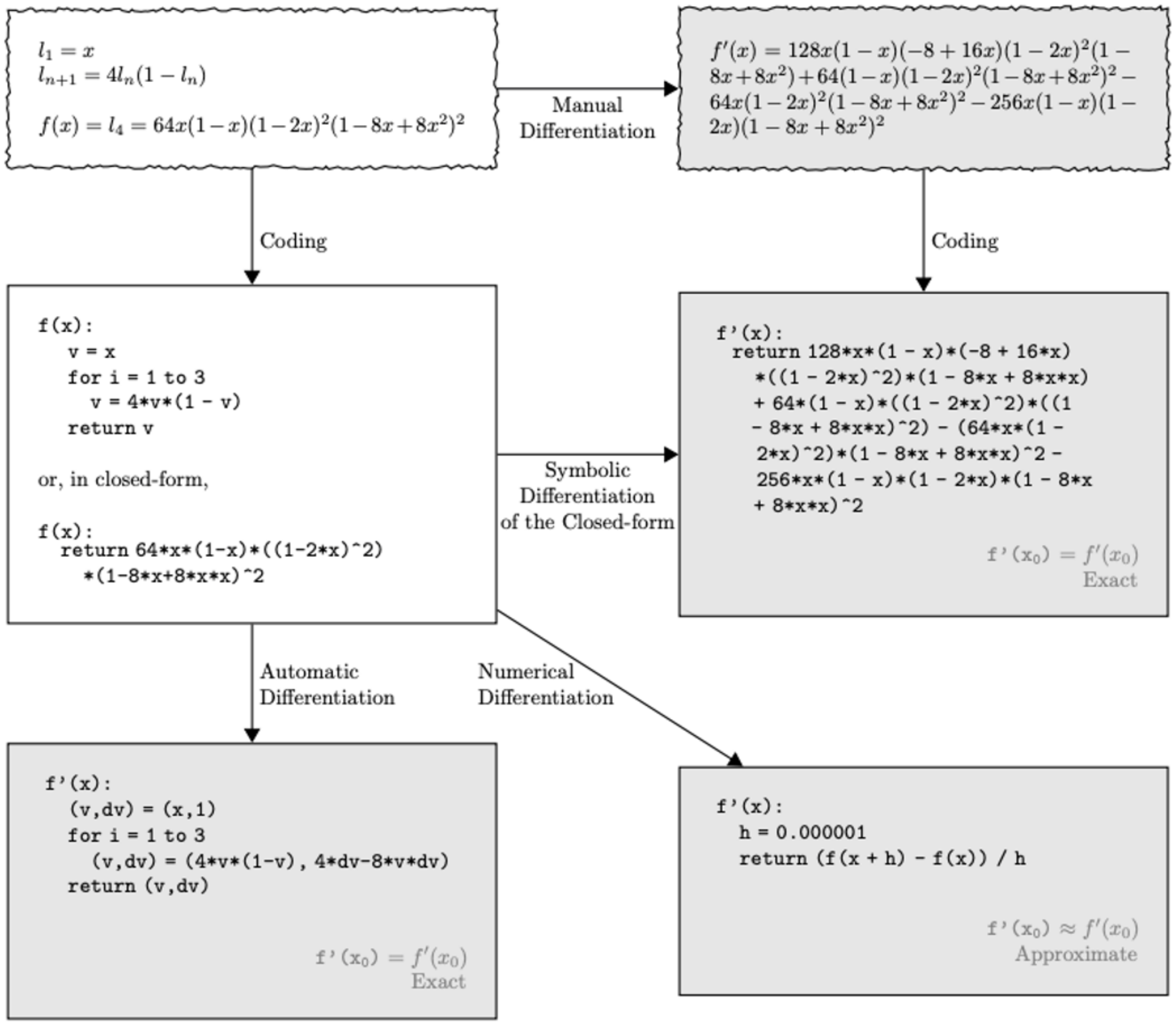
```python
list(mlp.parameters())
```

```
[Parameter containing:
 tensor([[ 0.5748],
         [ 0.4250],
         [ 0.6559],
         [ 0.0270],
         [ 0.2925],
         [ 0.0268],
         [-0.9413],
         [-0.8923],
         [-0.6783],
         [ 0.3521],
         [-0.3701],
         [ 0.7082],
         [ 0.8256],
         [ 0.1497],
         [-0.7315],
         [-1.0628],
         [ 0.3396],
         [ 0.8656],
         [ 0.0636],
         [-0.6879]], requires_grad=True),
 Parameter containing:
 tensor([-0.3643,  0.5639,  1.0288, -0.5873,  0.5042, -0.5881,  0.3506,  0.5615,
          0.6442, -0.9721,  0.0830,  0.3054, -0.7737, -0.5471,  0.8950, -0.8749,
         -0.2152,  0.6729, -0.1408,  0.9366], requires_grad=True),
```

And `requires_grad=True`?

Ways to Compute Derivatives of Code

Section modified from
M. Kagan

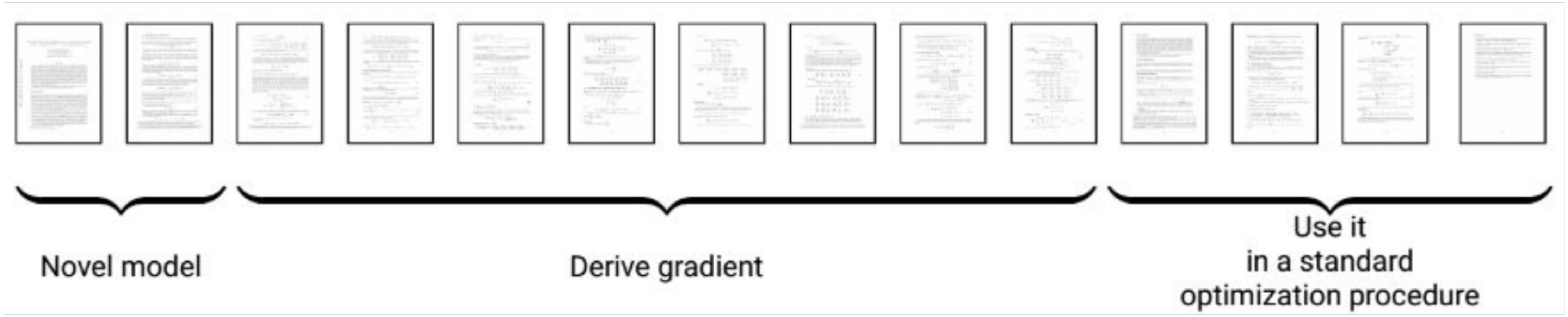


Baydin, Pearlmutter, Radul, Siskind. 2018. "Automatic Differentiation in Machine Learning: a Survey." Journal of Machine Learning Research (**JMLR**)

Ways to Compute Derivatives of Code

Manual differentiation:

- Derive expression by hand, then code it up
- Can be useful, but also labor intensive, case-by-case



Ways to Compute Derivatives of Code

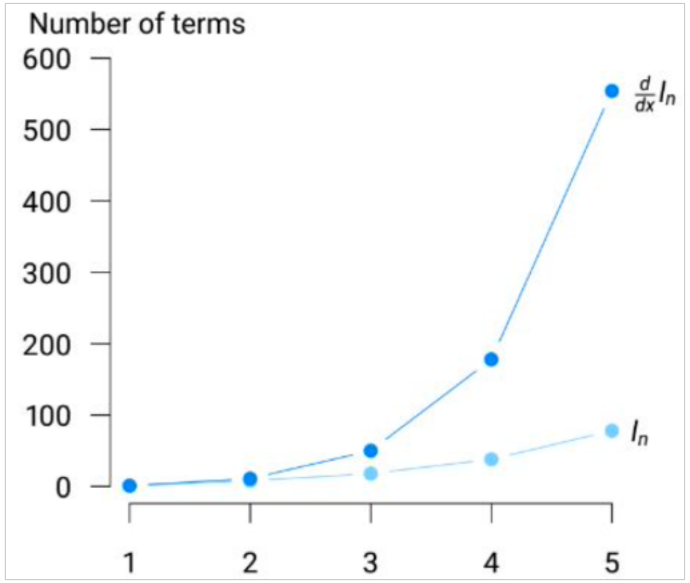
Symbolic differentiation:

- e.g. Mathematica, SymPy
- Gets messy/costly with number of terms
- Only applicable to closed form expressions (no control flow)

```
D[x^2, x]

2 x
```

Logistic map $l_{n+1} = 4l_n(1-l_n), l_1 = x$

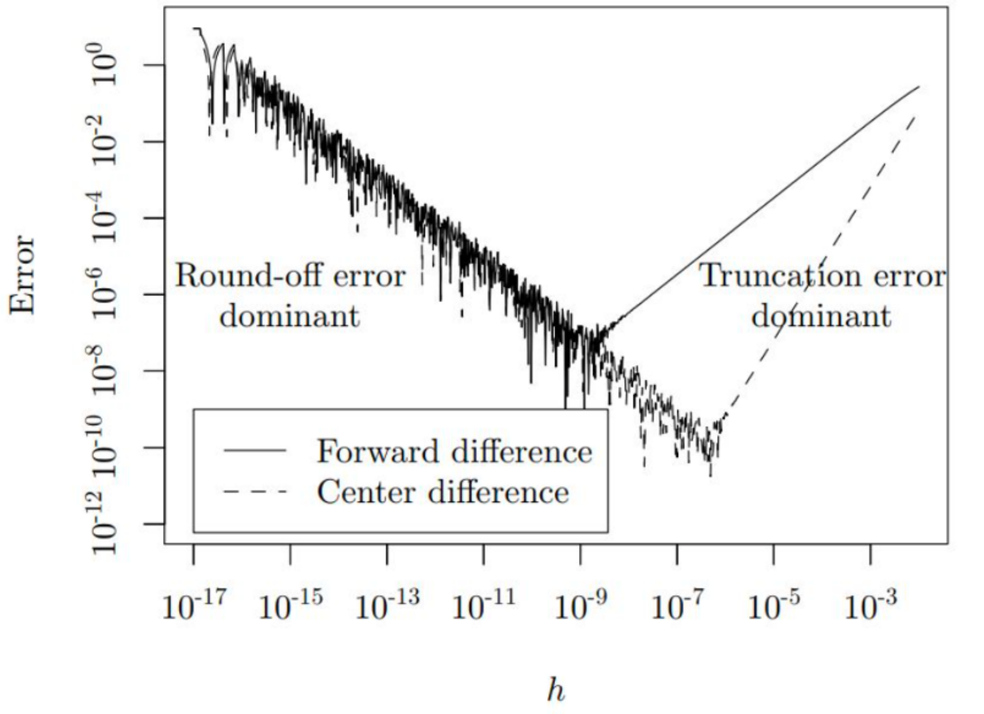
n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$



Ways to Compute Derivatives of Code

Numerical differentiation (finite differences):

- $\dfrac{\partial f(\mathbf{x})}{\partial x_i} \approx \dfrac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, 0 < h \ll 1$
- Blows up with input dimensionality (one function eval per basis vector $\mathbf{e}_i$)
- Approximation errors from choices of h

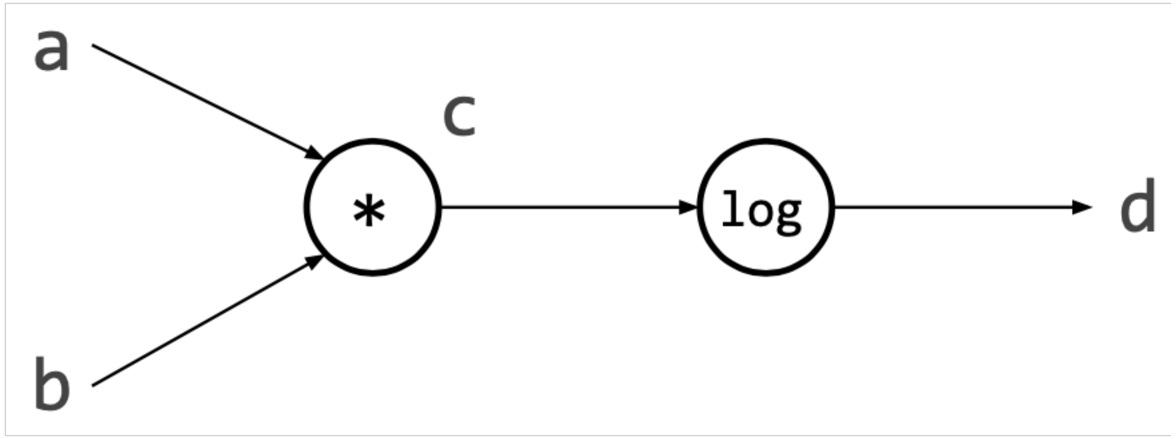


Ways to Compute Derivatives of Code

Automatic differentiation:

- Principle: break down arbitrary computer program into a graph of fundamental operations with known derivatives
- **Exact** gradient calculation, broadly applicable
- Scales well! Gradient cost ~ original code cost
  - e.g. neural networks ($f : \mathbb{R}^n \rightarrow \mathbb{R}$), forward + backward pass (gradients) ~2x cost of just forward (no gradients)

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```



Automatic Differentiation: The Chain Rule in Disguise

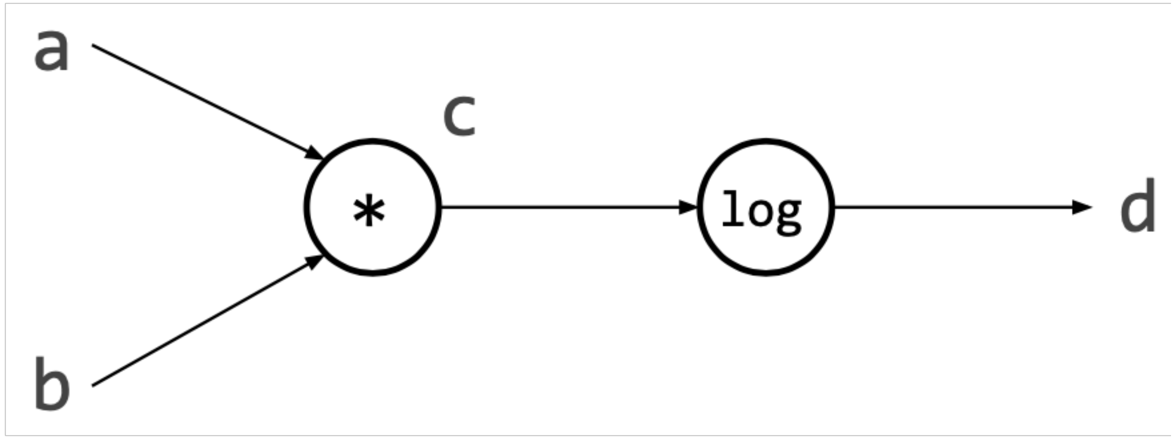
$$f(a,b) = \log(a \cdot b)$$

$$\nabla f(a,b) = \left(\frac{1}{a}, \frac{1}{b}\right)$$

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

Example: $\log(a \cdot b)$

- Represent as a **computational graph** showing all operations, dependencies

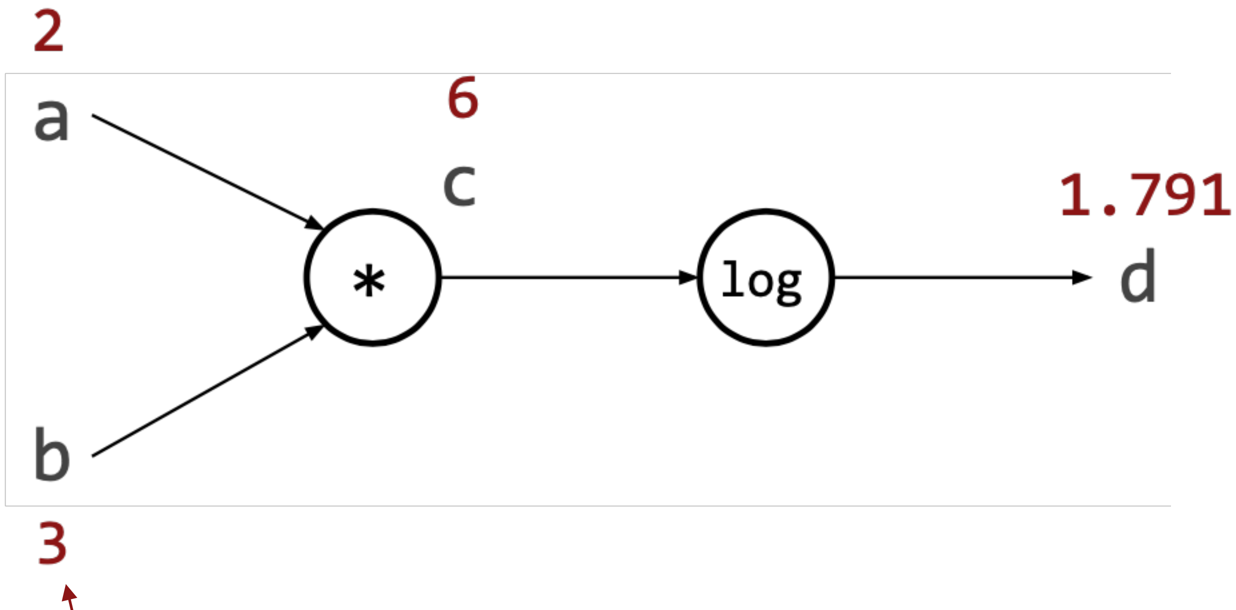


Automatic Differentiation: The Chain Rule in Disguise

Normal (forward) evaluation of the code for values of a, b results in a set of intermediate values (primals) at each stage of the computation

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

```
f(2, 3) = 1.791
```



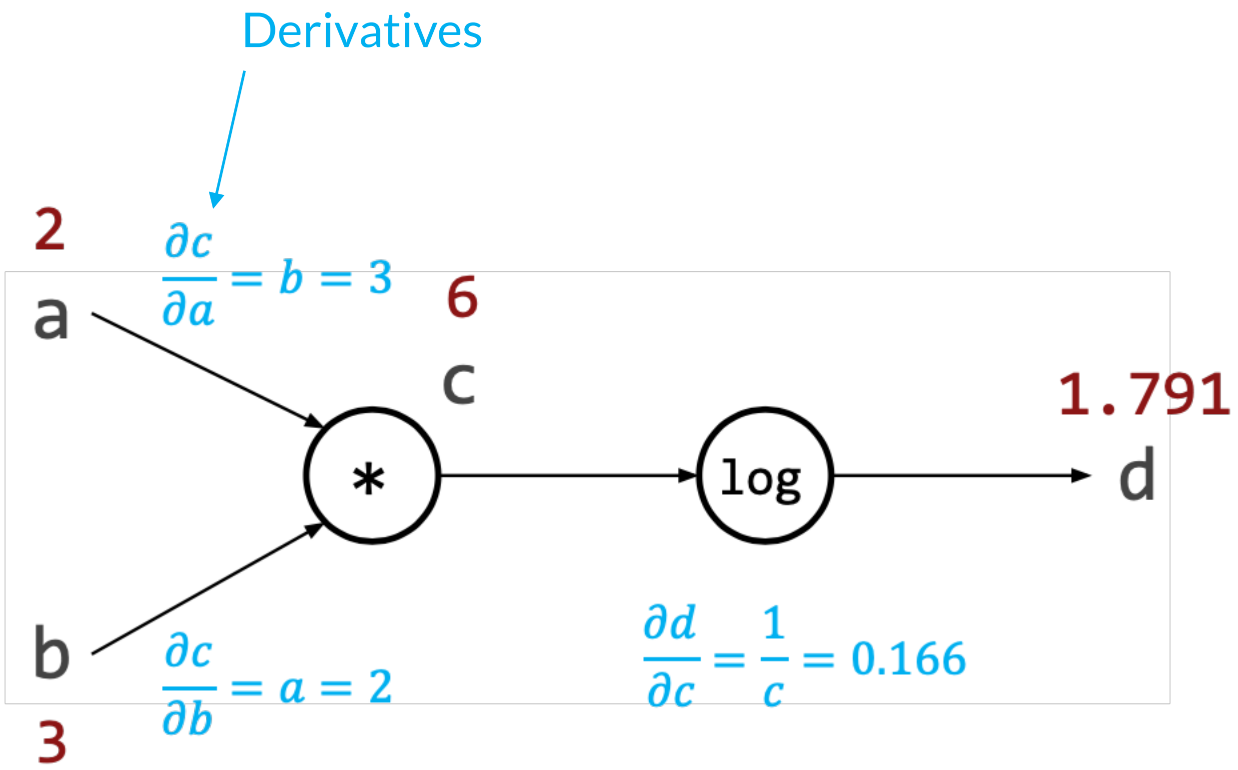
"Primals": intermediate function values

Automatic Differentiation: The Chain Rule in Disguise

The final result is a composition of the primal operations. The derivative of the final result is a product of the derivatives of each operation (via the chain rule).

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

```
f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```



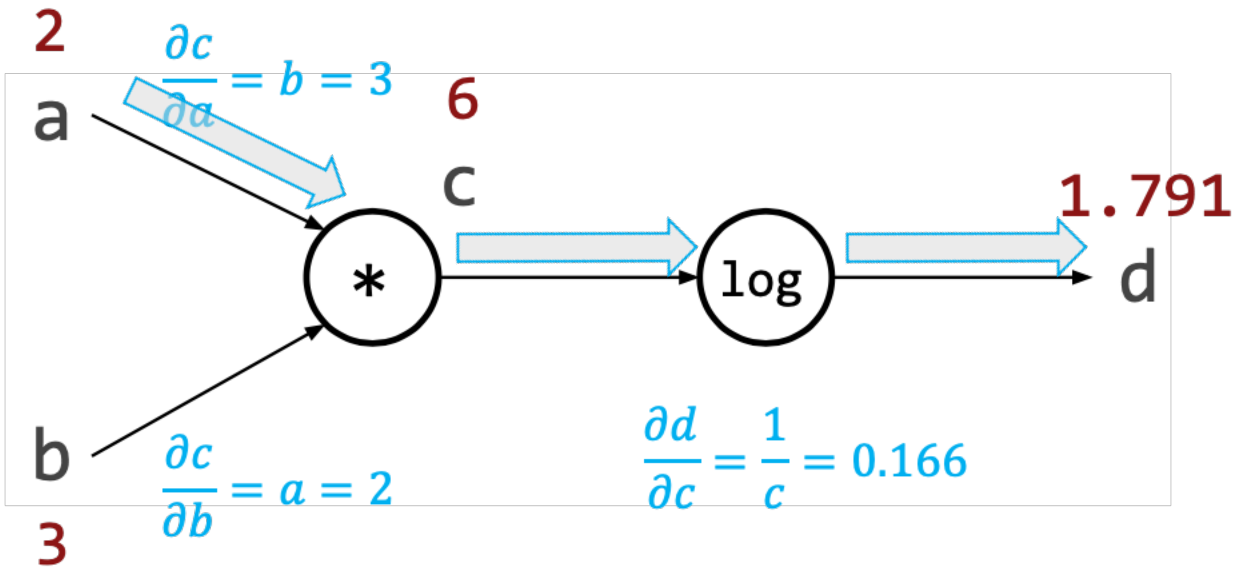
Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

Automatic Differentiation: The Chain Rule in Disguise

Different modes of automatic differentiation <=> different order of evaluation of terms in the chain rule

- **Forward mode AD:** Inner (inputs) to outer (end result)

```
f(a, b):
    c = a * b
    d = log(c)
    return d

f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```



Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

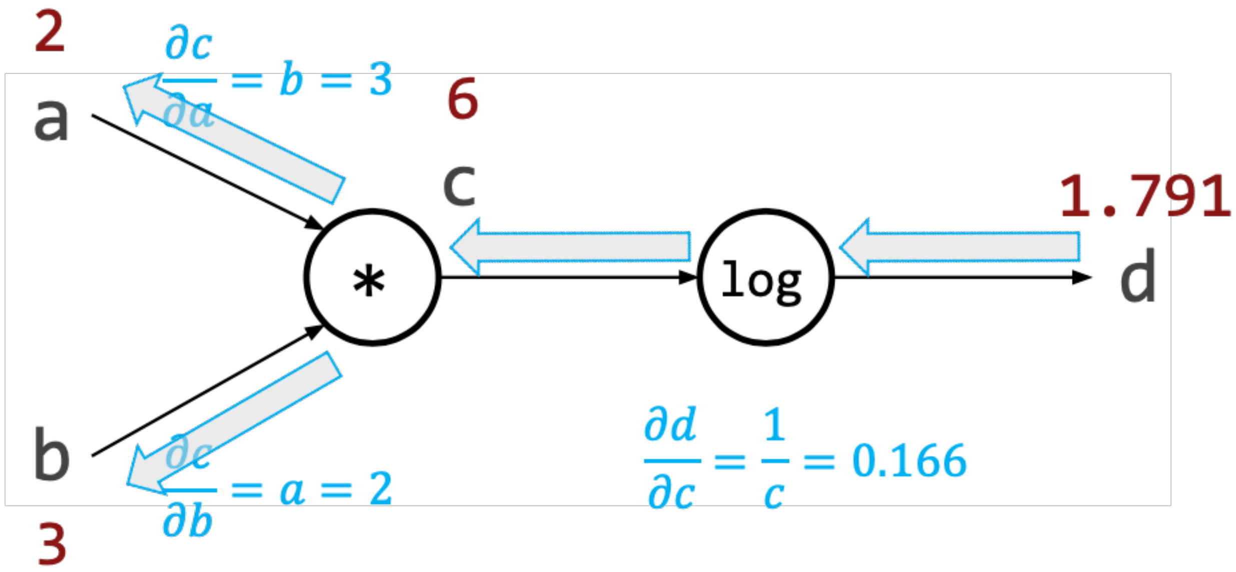
Outer ⟸ Inner

Automatic Differentiation: The Chain Rule in Disguise

Different modes of automatic differentiation <=> different order of evaluation of terms in the chain rule

- **Reverse mode AD (cf. backprop):** Outer (end result) to inner (inputs)

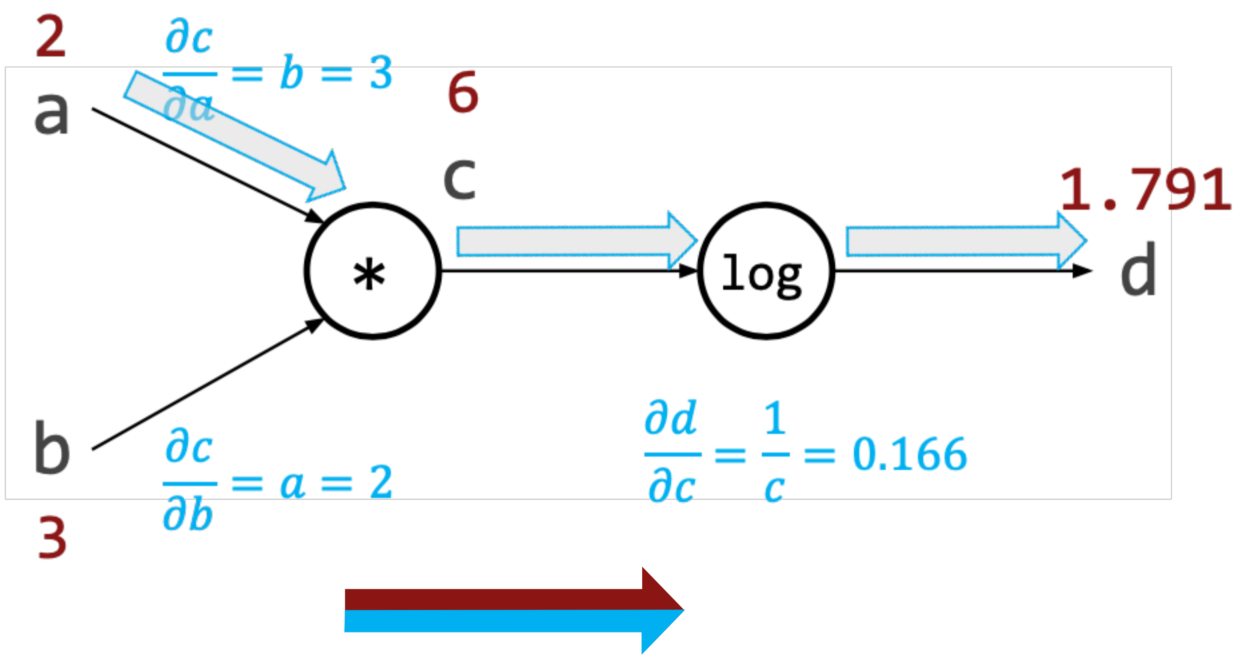
```
f(a, b):
    c = a * b
    d = log(c)
    return d

f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```



Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

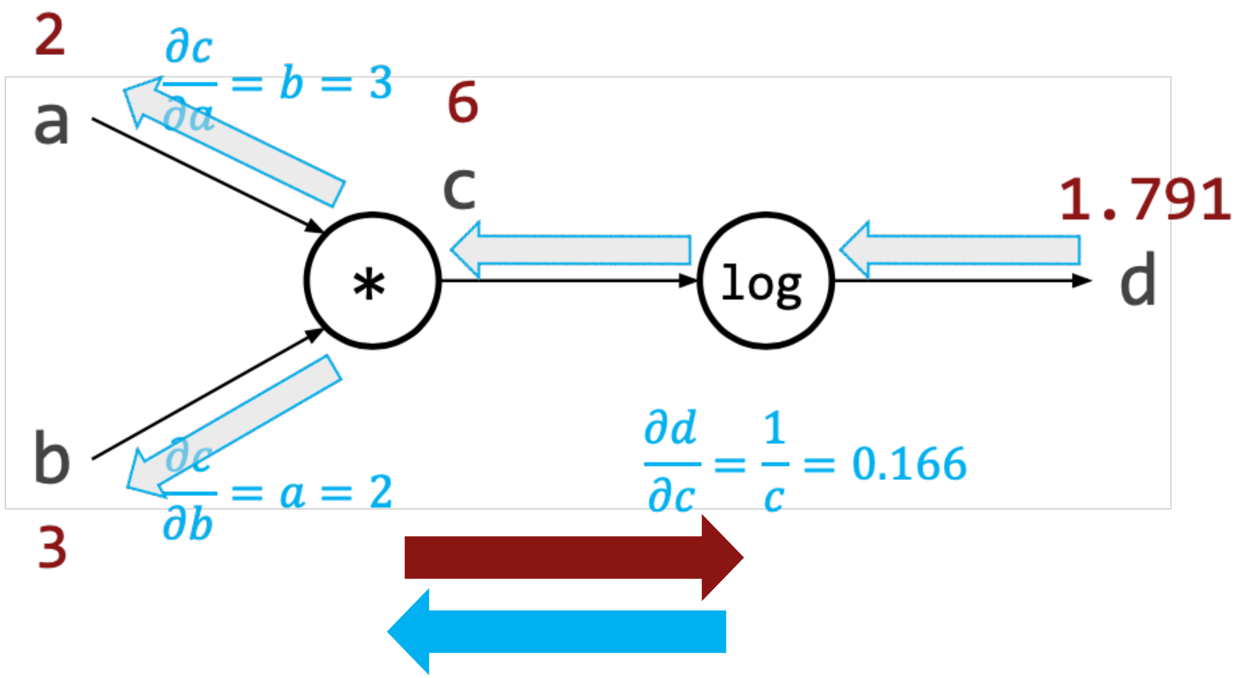
Outer ⟹ Inner

Automatic Differentiation: Forward vs Reverse Mode



Forward mode:
Compute primals and derivatives on single forward pass: follow the evaluation flow.

Additional sweep needed for each independent variable (e.g. b vs a)



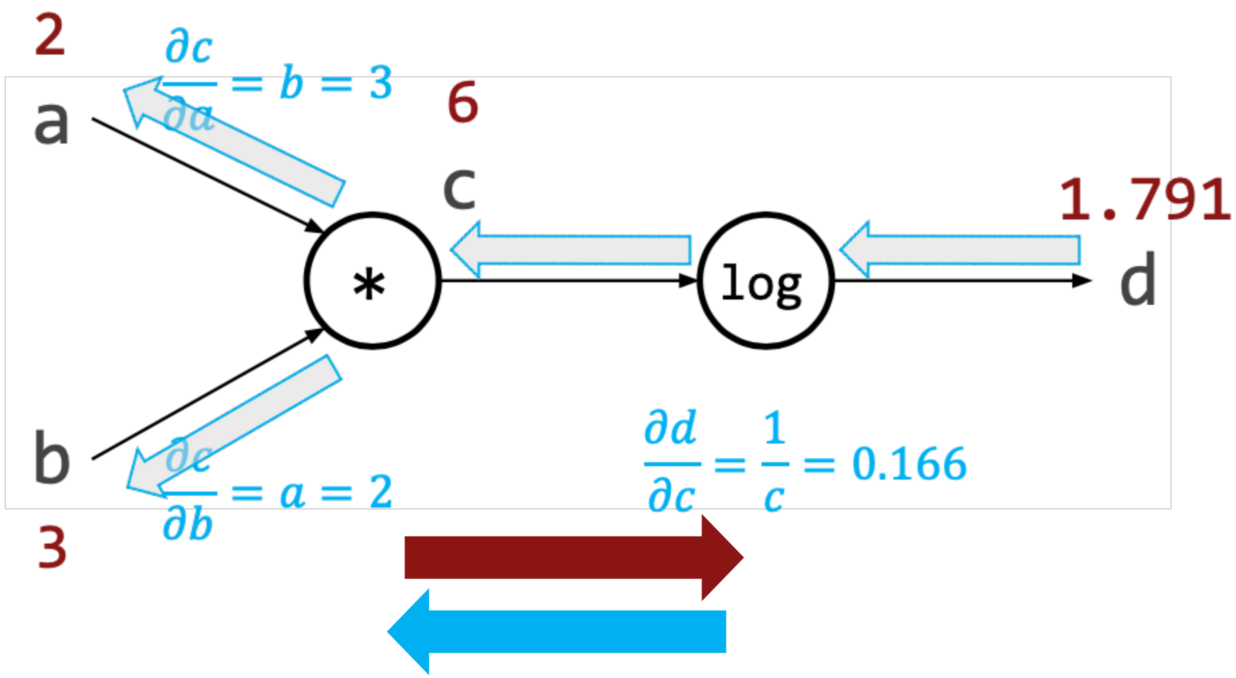
Reverse mode:
Compute and store primals on forward pass, compute and accumulate derivatives on backward pass

Additional sweep for needed for each dependent variable (e.g. multiple outputs)

Automatic Differentiation: Forward vs Reverse Mode

Neural networks usually have large number of inputs, small number of outputs (e.g. scalar loss function)

- => backpropagation <=> reverse mode AD more efficient



Reverse mode:
Compute and store primals on forward pass, compute and accumulate derivatives on backward pass

Additional sweep for needed for each dependent variable (e.g. multiple outputs)

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \rightarrow \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Forward mode (single evaluation):

Derivatives of all M outputs w.r.t. one input => column of Jacobian matrix

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Reverse mode (single evaluation):

Derivatives of one output w.r.t. N inputs => row of Jacobian matrix

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \to \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Forward mode (single evaluation):
Derivatives of all M outputs w.r.t. one input => column of Jacobian matrix

Relevant column can be extracted by multiplying by an appropriate basis vector:

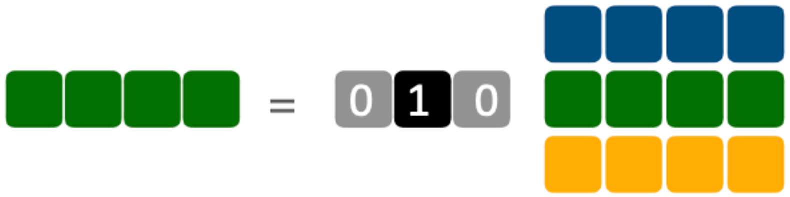
Forward mode AD <=> **Jacobian-vector product (JVP)**

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \to \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Reverse mode (single evaluation):
Derivatives of one output w.r.t. N inputs => row of Jacobian matrix



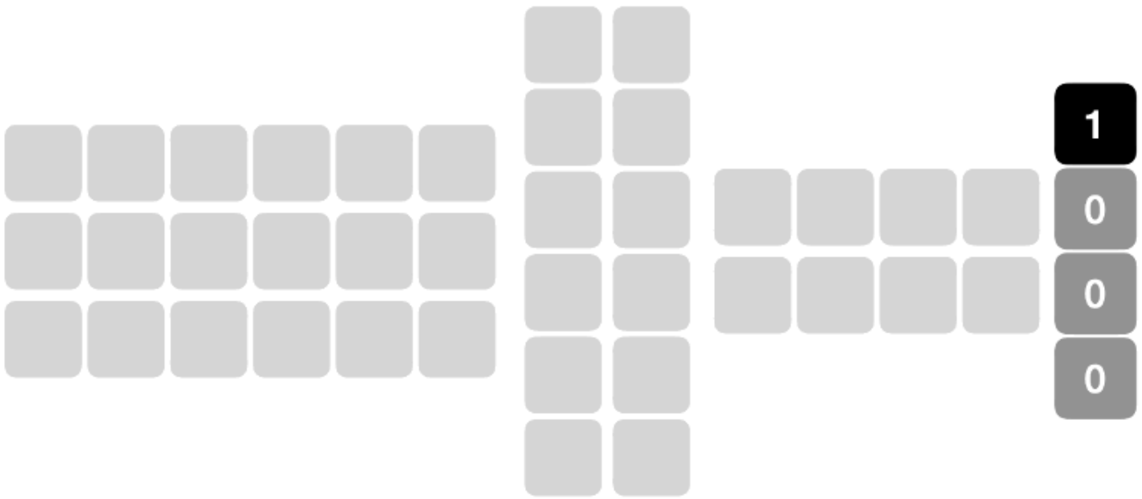
Relevant row can be extracted by multiplying by an appropriate basis vector:

Reverse mode AD <=> **vector-Jacobian product (VJP)**

How to compute efficiently?

Chain Rule: Jacobian matrix of function composition is product of Jacobian matrices of constituent functions

- e.g.: $J_{\mathbf{f}\circ\mathbf{g}(\mathbf{x})} = J_{\mathbf{f}}(\mathbf{g}(\mathbf{x})) \cdot J_{\mathbf{g}}(\mathbf{x})$
- Vector-Jacobian/Jacobian-vector product for **elementary operations** + composition => gradient computation
- See e.g. https://theoryandpractice.org/stats-ds-book/autodiff-tutorial.html for explicit examples



$$c_i = Me_i = M_3M_2M_1e_i$$

Video: L. Heinrich

Neural Networks are Code

Example of a neural network in PyTorch

```
# Multi-layer perceptron
mlp = MLP(n_hidden=n_hidden, hidden_dim=hidden_dim)

# Optimizer
optimizer = torch.optim.Adam(mlp.parameters(), lr=lr)

# Mean squared error
loss_fn = torch.nn.MSELoss()

losses = []
for _ in range(n_epochs):

    # Shuffle data
    idxs = np.random.permutation(len(norm_x))

    # Make predictions
    out = mlp(norm_x[idxs])

    # Calculate loss
    loss = loss_fn(out, norm_y[idxs])

    # Zero out gradients
    optimizer.zero_grad()

    # Compute gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

What's happening when we call `loss.backward()`?

Backpropagation (reverse mode AD)

What is this `grad_fn`?

Node in computational graph

```
mlp(norm_x[0:1])
```

```
tensor([[-1.6607]], grad_fn=<AddmmBackward0>)
```

```
list(mlp.parameters())
```

```
[Parameter containing:
 tensor([[ 0.5748],
         [ 0.4250],
         [ 0.6559],
         [ 0.0270],
         [ 0.2925],
         [ 0.0268],
         [-0.9413],
         [-0.8923],
         [-0.6783],
         [ 0.3521],
         [-0.3701],
         [ 0.7082],
         [ 0.8256],
         [ 0.1497],
         [-0.7315],
         [-1.0628],
         [ 0.3396],
         [ 0.8656],
         [ 0.0636],
         [-0.6879]], requires_grad=True),
 Parameter containing:
 tensor([-0.3643,  0.5639,  1.0288, -0.5873,  0.5042, -0.5881,  0.3506,  0.5615,
          0.6442, -0.9721,  0.0830,  0.3054, -0.7737, -0.5471,  0.8950, -0.8749,
         -0.2152,  0.6729, -0.1408,  0.9366], requires_grad=True),
```

And `requires_grad=True`?

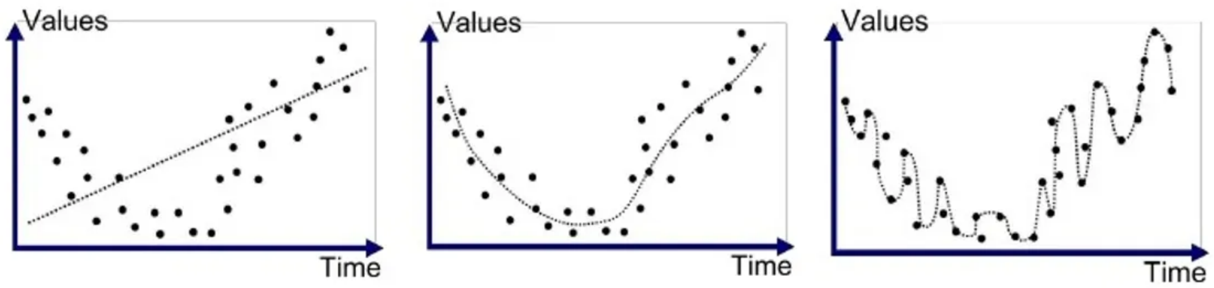
Tells PyTorch that we want a gradient with respect to this tensor out of `loss.backward()`

Practical Tips

Training, Validation, and Test Datasets

In the real world, we have limited data

- **Training set:** Dataset used for model training
- Should always keep two other datasets separate
  - **Validation set:** Use to check overfitting, tune model **hyperparameters** (e.g. number of layers, etc). Some methods here (e.g. **cross-validation**)
  - **Test set:** Only touch this at the very very end — this is what you use to report (unbiased) results

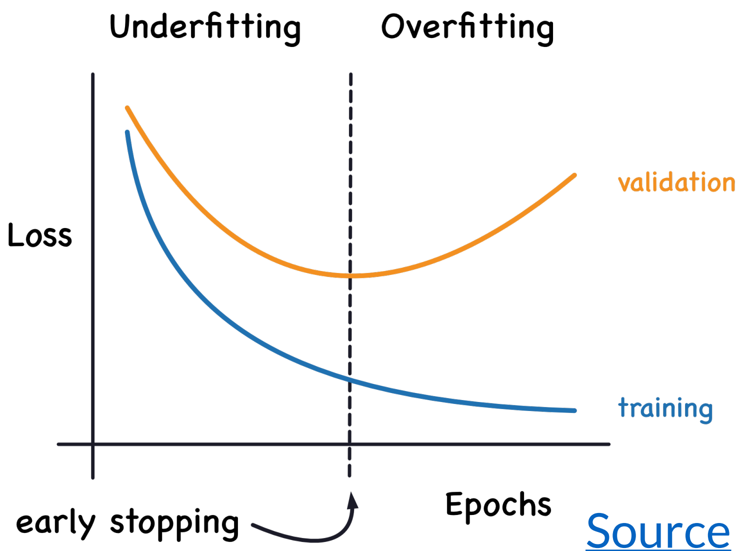


Source

Underfitted

Good Fit/Robust

Overfitted



Source

Normalization

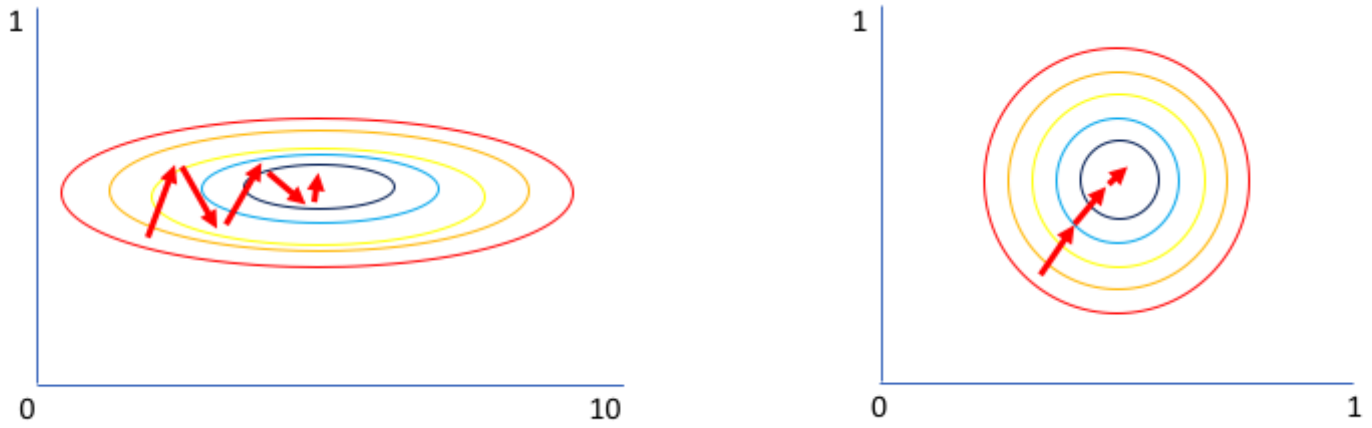
Generally a good idea to **normalize** your data

- Mismatch in feature sizes => model training will pay more attention to larger features, less to small
  - True for both inputs and outputs

$$\{(x_i^1, \ldots, x_i^n)\}_{i=1}^m$$

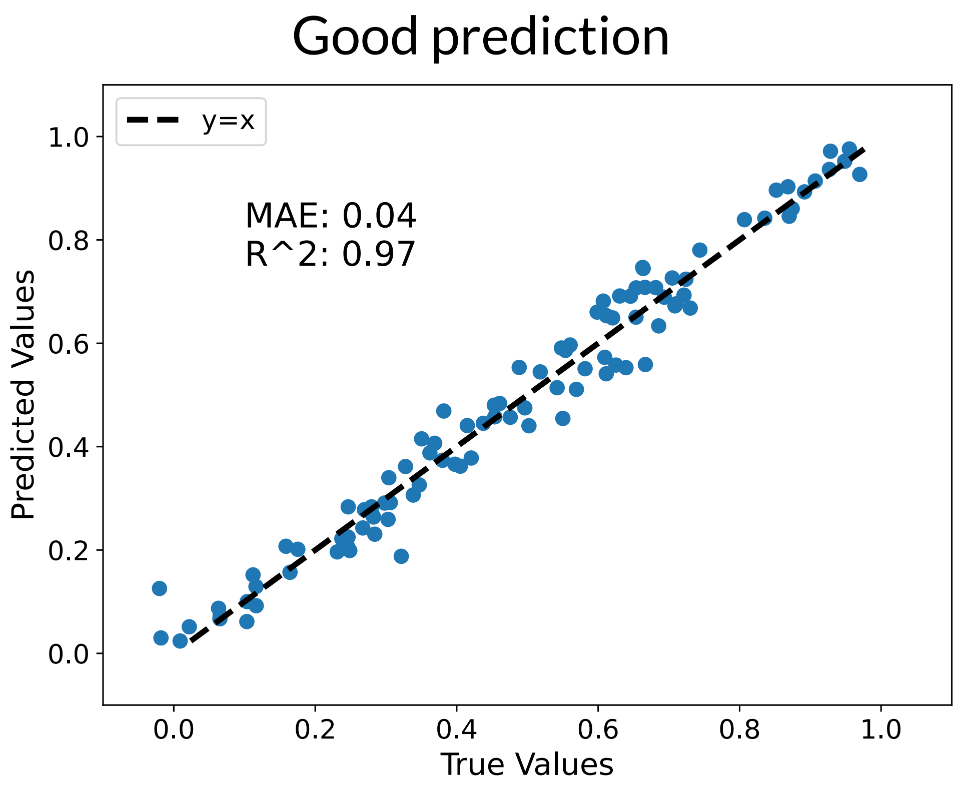
$$x_{i,norm}^j = \frac{x_i^j - \mu^j}{\sigma^j}$$

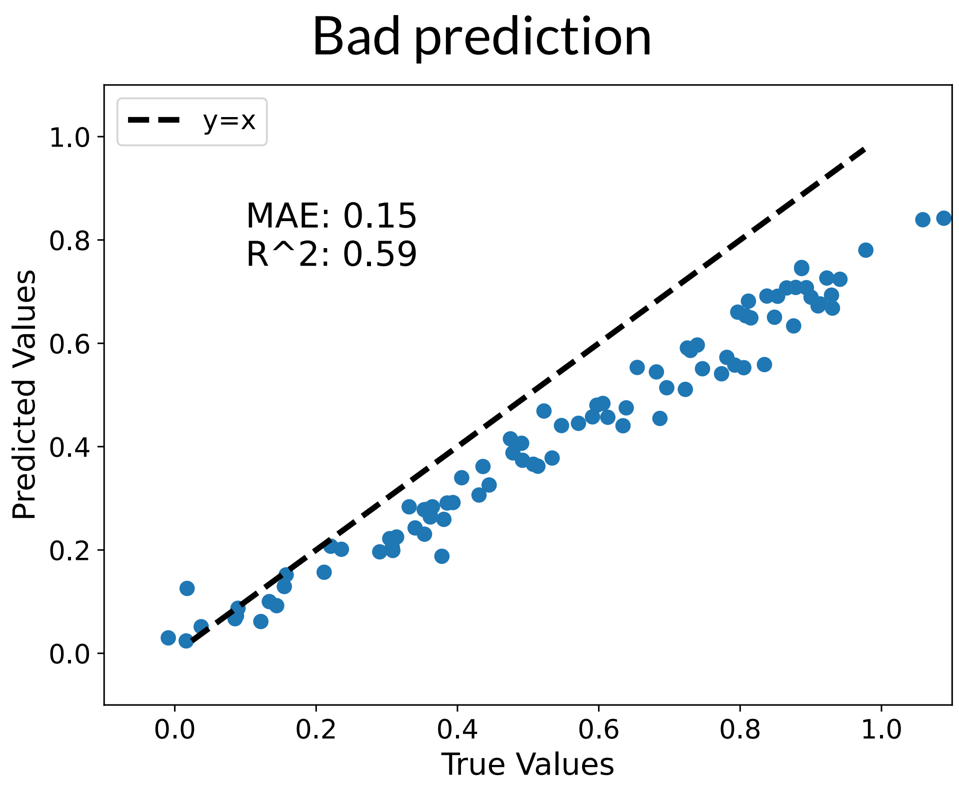
"Standard" Normalization:
Normalize each **feature** by feature mean and standard deviation across training dataset



Source

Regression Metrics





$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i^{pred} - y_i^{true}|$$

$$R^2 = 1 - \frac{\sum_i (y_i^{pred} - y_i^{true})^2}{\sum_i (y_i^{true} - \bar{y})^2} \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i^{true}$$

For regression tasks, often useful to include **parity plots** with mean absolute error (smaller is better) and R^2 (close to 1 is better)

Classification Metrics: Binary Classification

		Predicted	
		0	1
Actual	0	True Negative	False Positive
	1	False Negative	True Positive

In classification, either the model gets a prediction right (given x_i) or gets it wrong

- We can represent this with a **confusion matrix**
- For binary classification one class is negative, one is positive
- If we get it right, it's true (e.g. true negative) if not it's false (e.g. false positive)

Classification Metrics: Binary Classification

		Predicted	
		0	1
Actual	0	True Negative	False Positive
	1	False Negative	True Positive

$$\text{Precision} = \frac{TP}{TP + FP}$$

Out of all positive predictions, how many are actually positive? Higher is better (fewer false positives)

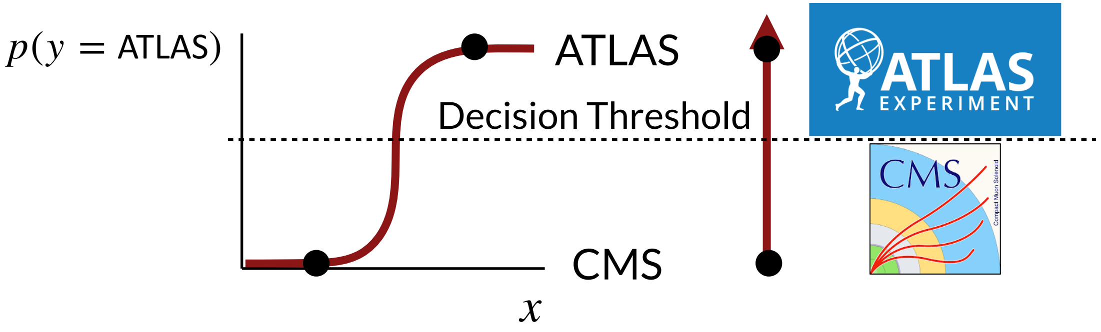
$$\text{Recall} = \frac{TP}{TP + FN}$$

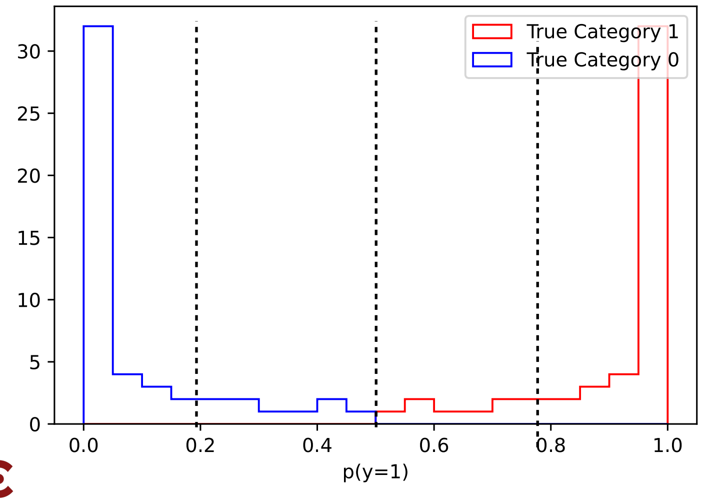
How many actual positives did we get right? Higher is better (more actual positives recovered)

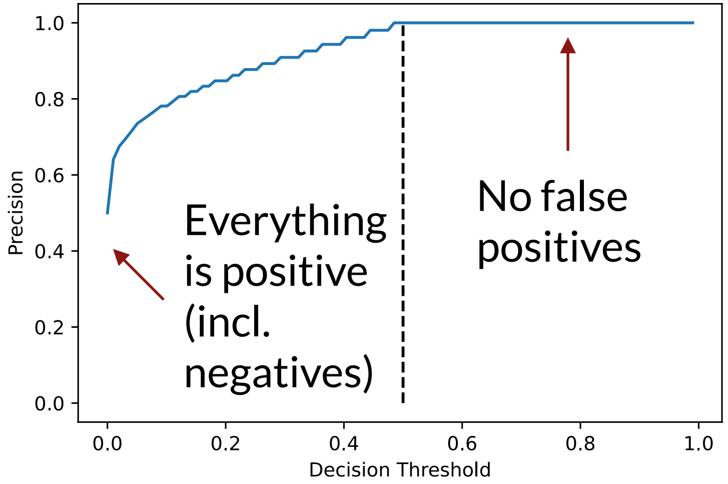
Classification Metrics: Binary Classification

Recall: we need to define a **decision threshold** to assign events to a category

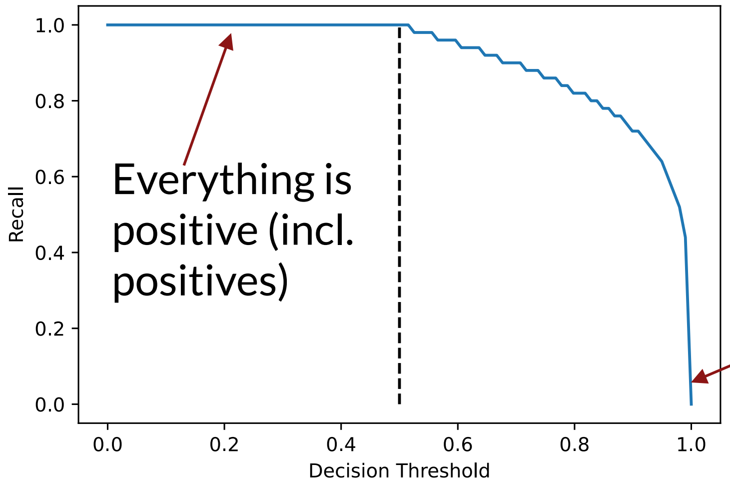
- The choice of this threshold will impact our precision and recall!







$$\text{Precision} = \frac{TP}{TP + FP}$$



$$\text{Recall} = \frac{TP}{TP + FN}$$

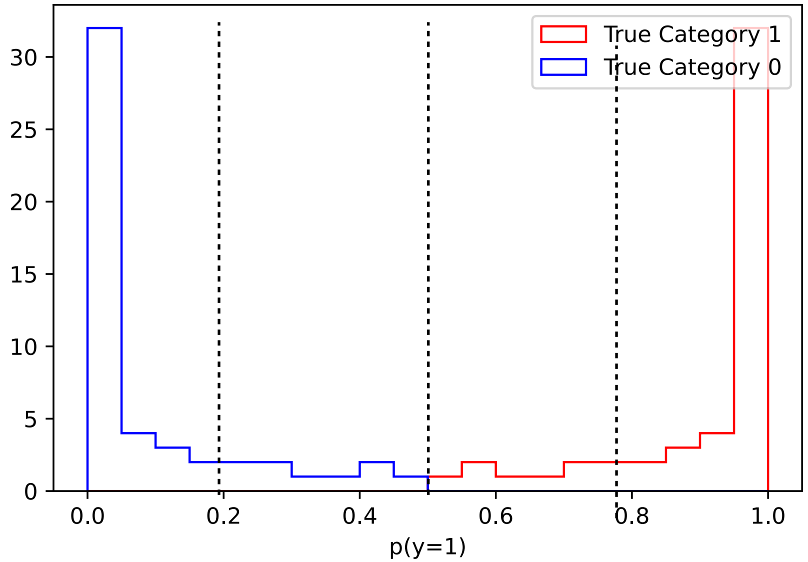
Nothing is positive (incl. positives)

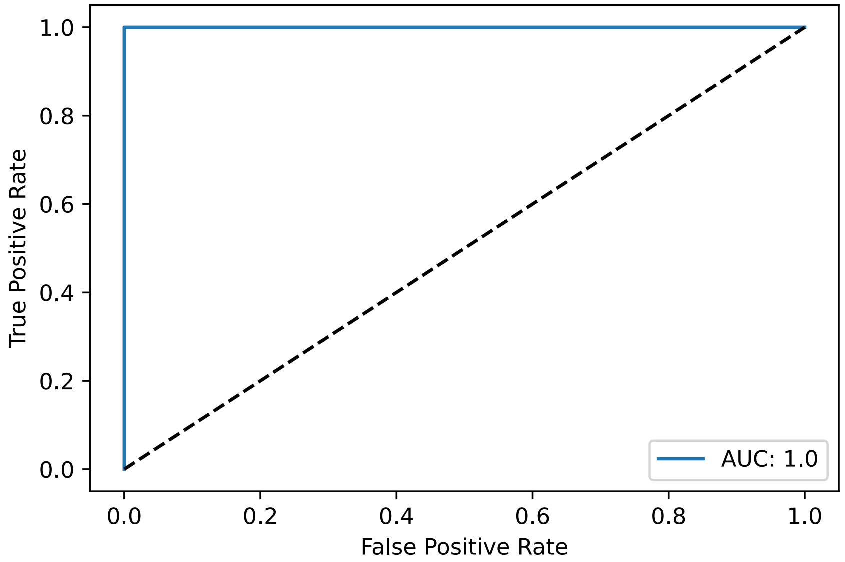
Classification Metrics: Binary Classification

In particle physics, we often look more at **ROC curves** (receiver operating characteristic)

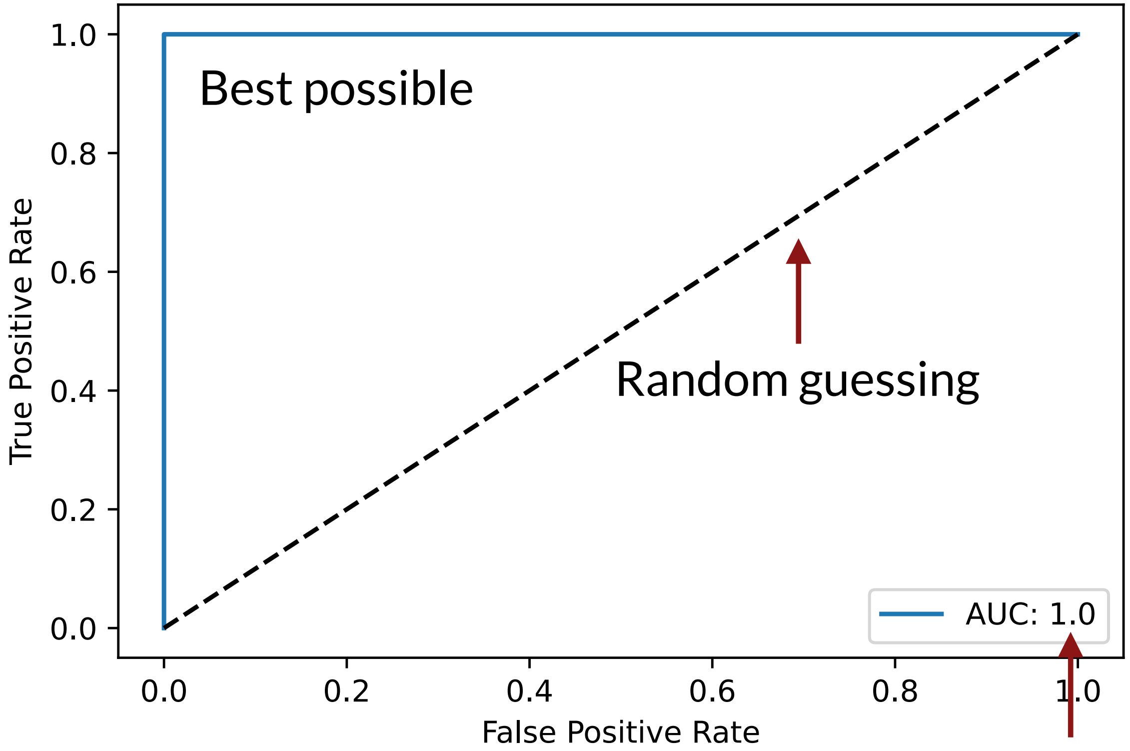
$$\text{True positive rate (=Recall)} = \frac{TP}{TP + FN} \qquad \frac{\text{Positives we got right}}{\text{All actual positives}}$$

$$\text{False positive rate} = \frac{FP}{FP + TN} \qquad \frac{\text{Negatives we got wrong}}{\text{All actual negatives}}$$

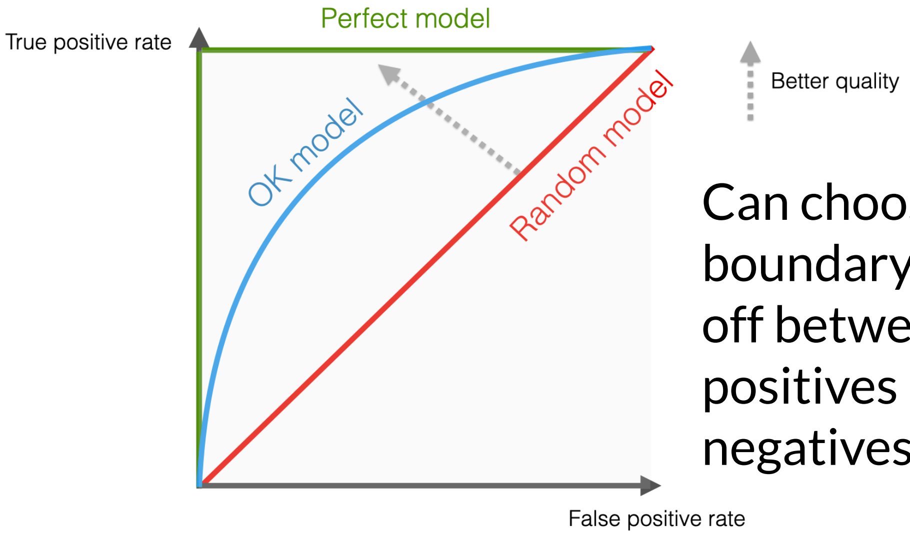




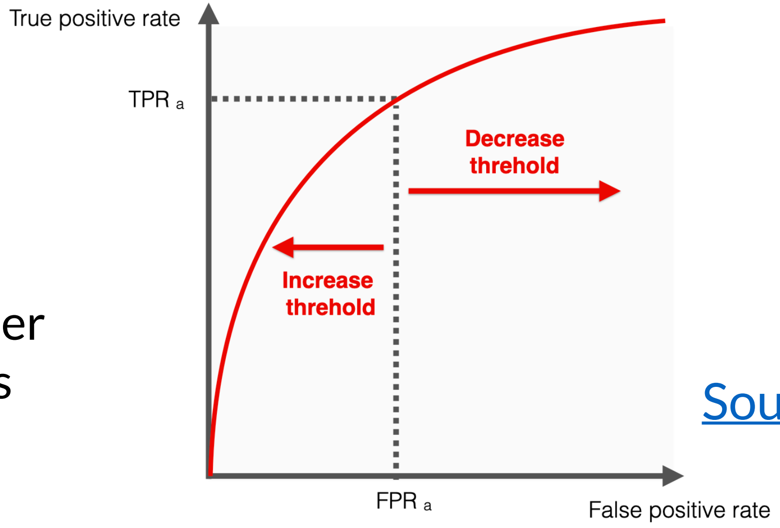
Classification Metrics: Binary Classification



Numerical metric:
Area under curve: higher is better (1 is max, 0.5 is random)

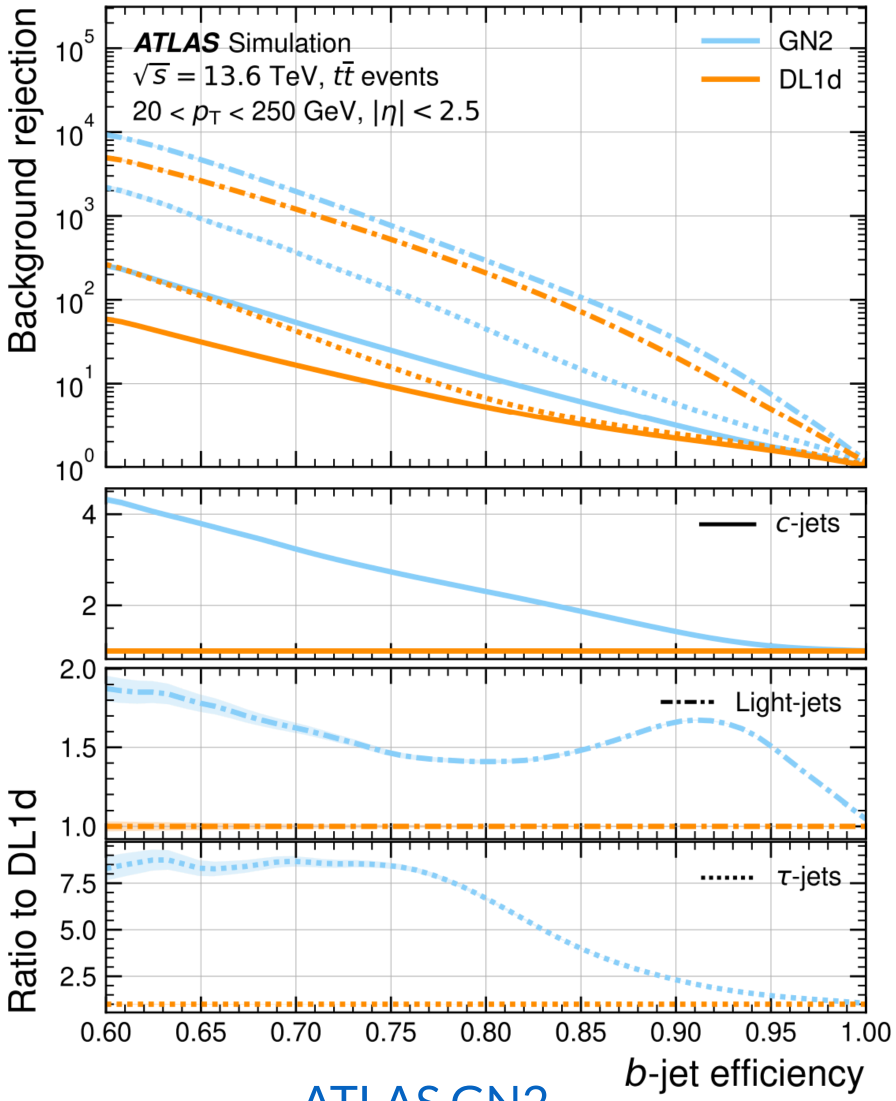


Can choose boundary to trade off between true positives and false negatives



[Source]

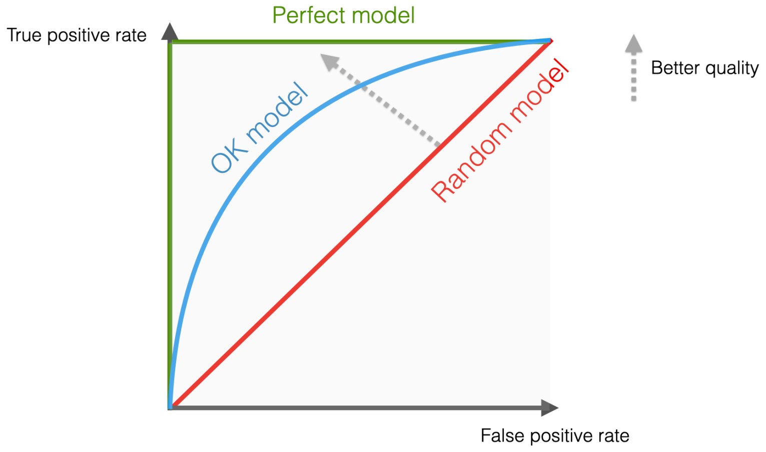
Classification Metrics: Binary Classification



[ATLAS GN2](#)

This is also a ROC curve!

- b-jet efficiency = true positive rate (number of b-jets we got right out of total b-jets)
- Background rejection = 1/false positive rate
  - FPR: fraction of background jets we incorrectly classify as b-jets (accept)
  - 1/FPR: how many background jets we correctly reject for each one we incorrectly accept



Summary

Machine learning is an expansive field, and is part of the way we do science!

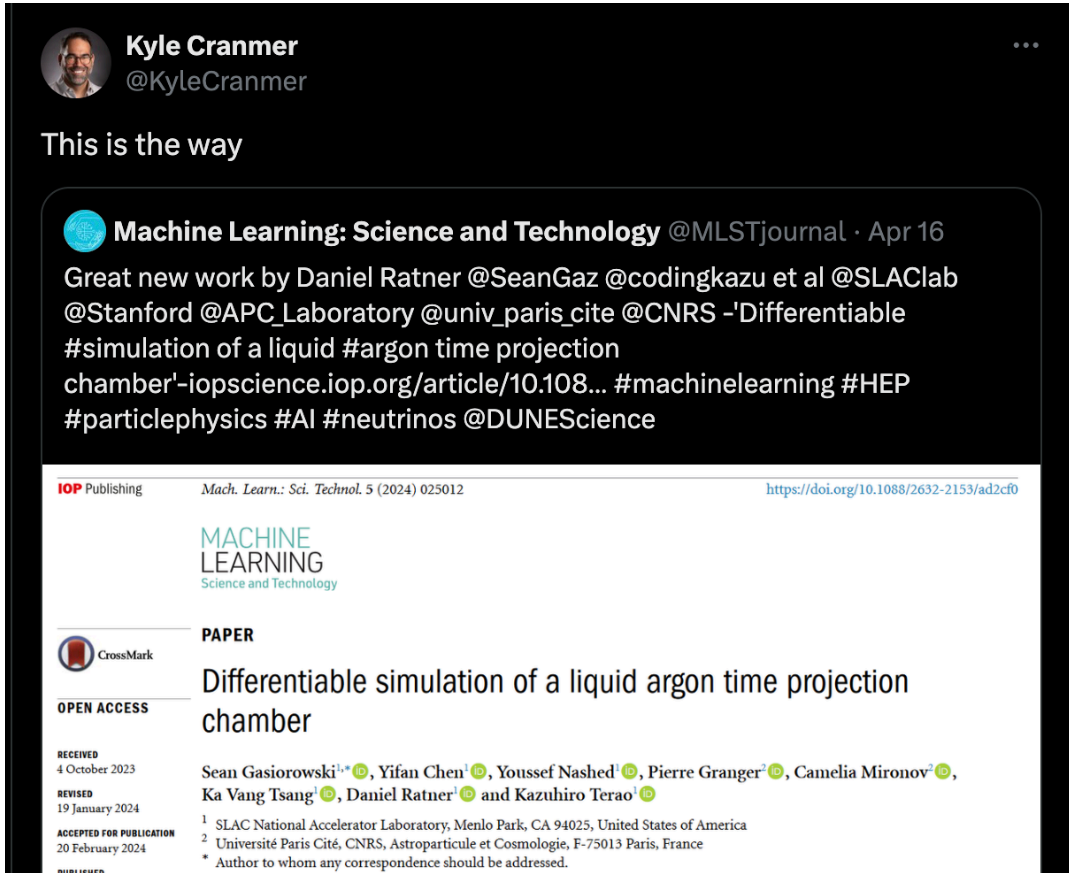
- Can be as simple as linear regression in Excel
- Many different types of machine learning models, suitable for different types of data and different goals
  - Each encode some inductive bias that guides their predictions
- Gradient-based optimization underpins much of machine learning, and we have efficient tools to compute gradients
  - Parameters change, but all of them need optimizing!
- If something interested you, **give it a try!** Easy to find examples online
  - + tutorials next week!

Bonus

Neural networks are just code

Machine learning libraries are able to efficiently calculate gradients with respect to neural network parameters

- Neural networks are just differentiable functions
- Why stop at neural networks?
- **Differentiable programming:** use ML libraries to write code (neural networks, but also e.g. exact physics simulators)
    - The **same techniques** that enable neural network training can be used to calculate gradients with respect to code parameters



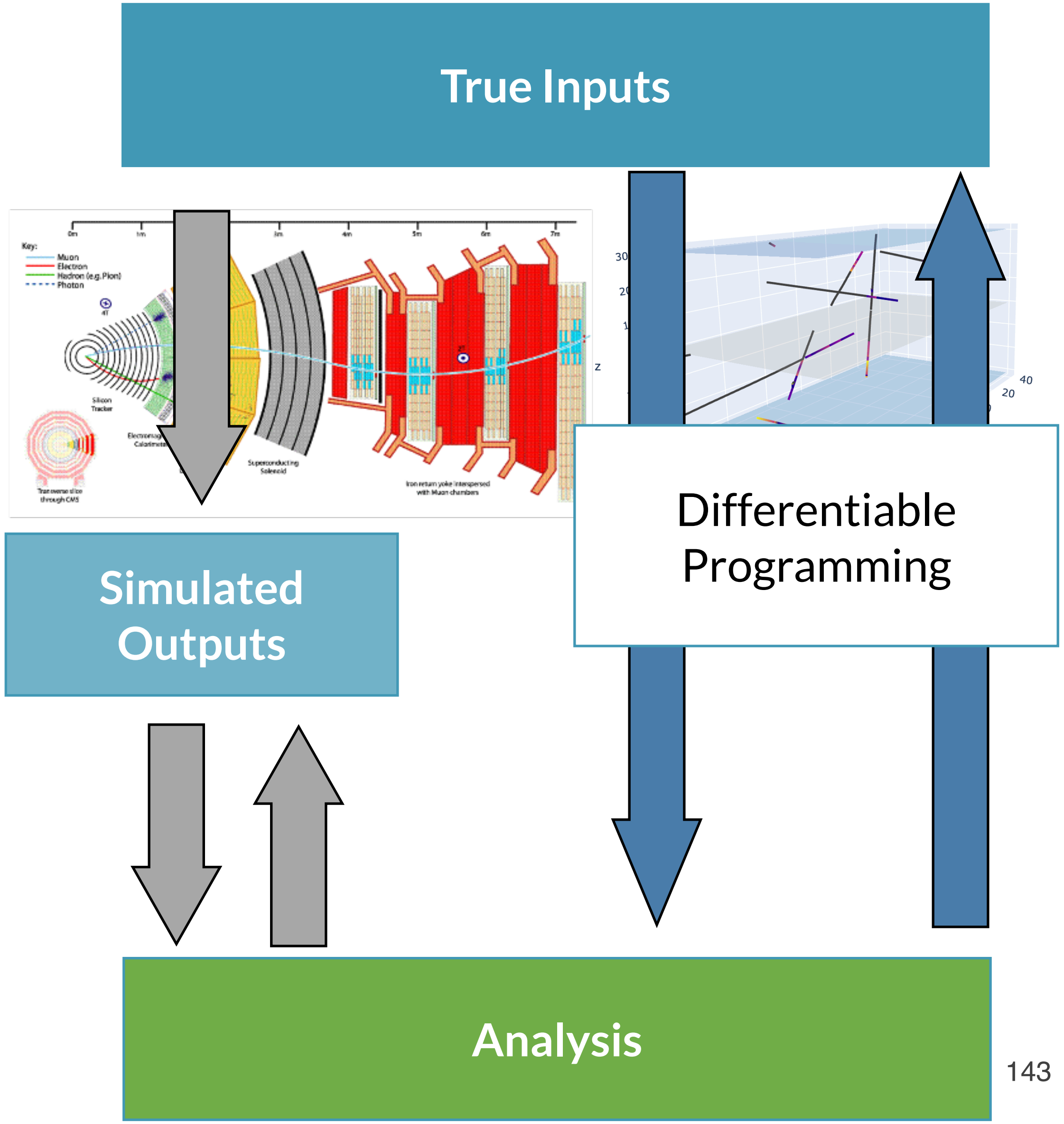
Why do we care?

Simulators are very important to HEP, but we often only use inputs and outputs

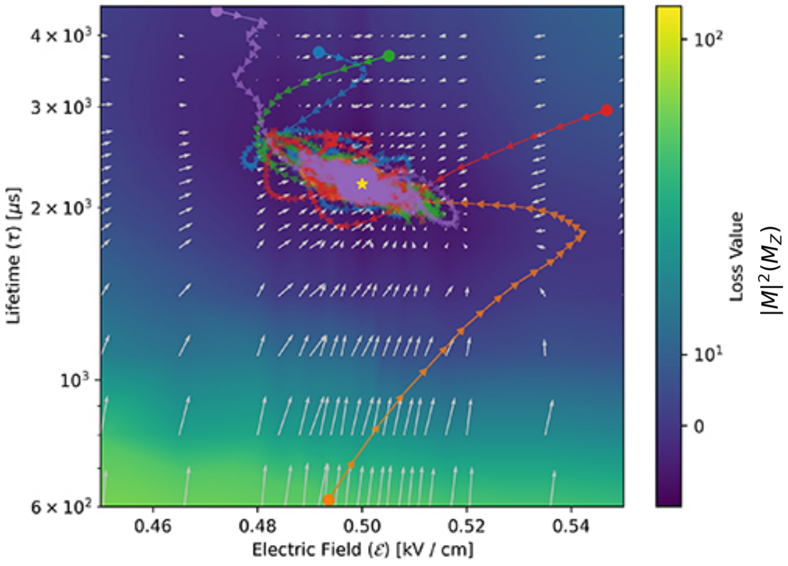
- Differentiable simulators can be directly used in ML pipelines — **explicitly use physics**, rather than relying on examples!
- Gradient information can be used to augment simulator output
- Fits of simulation to data can be used to understand and adjust underlying processes (e.g. **detector conditions/calibration**)

Analysis workflows feature many parameters (cuts, binning) that are often painstakingly tuned

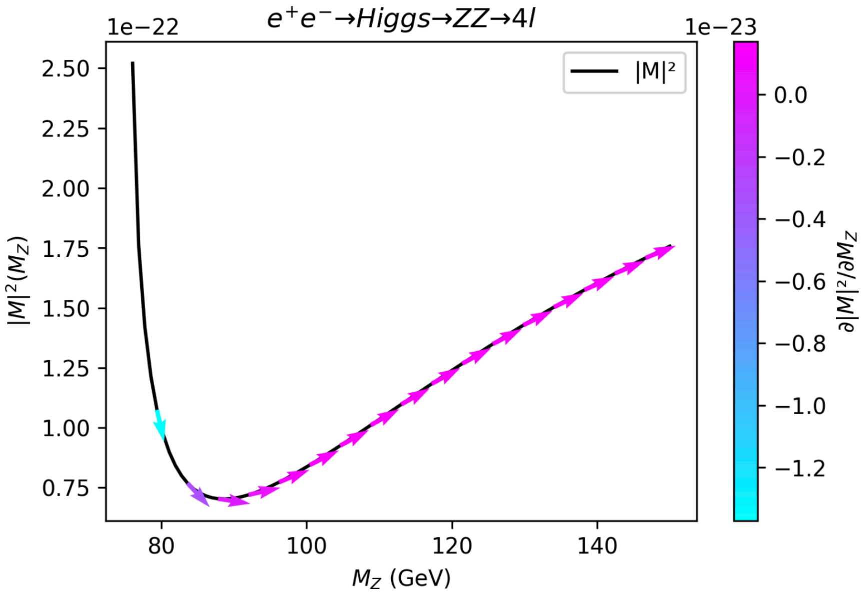
- Differentiable programming can make **optimizing** these **many parameters** possible



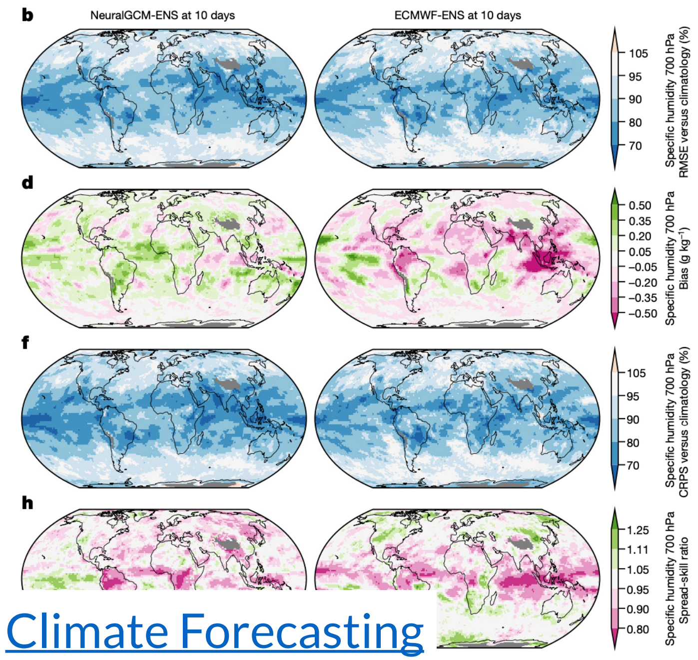
Differentiable Programming: Applications



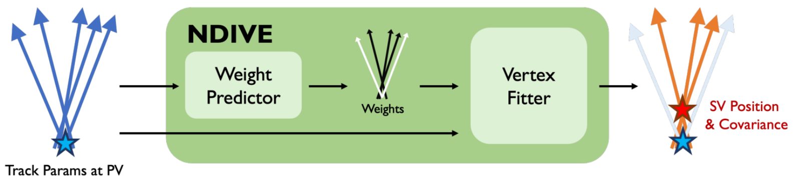
Neutrino Simulation



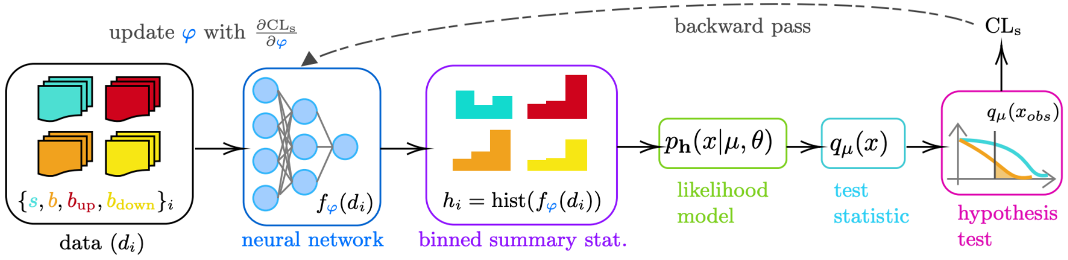
MadJAX



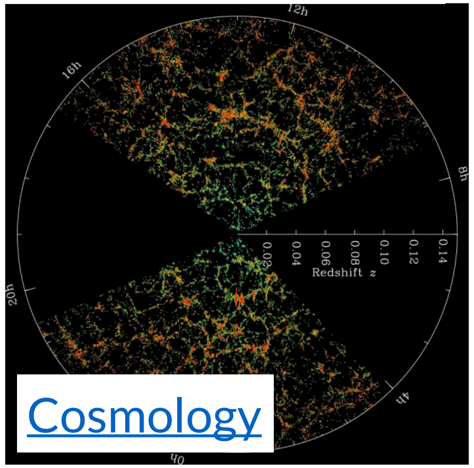
Climate Forecasting



Flavor Tagging



HEP Analysis



Cosmology