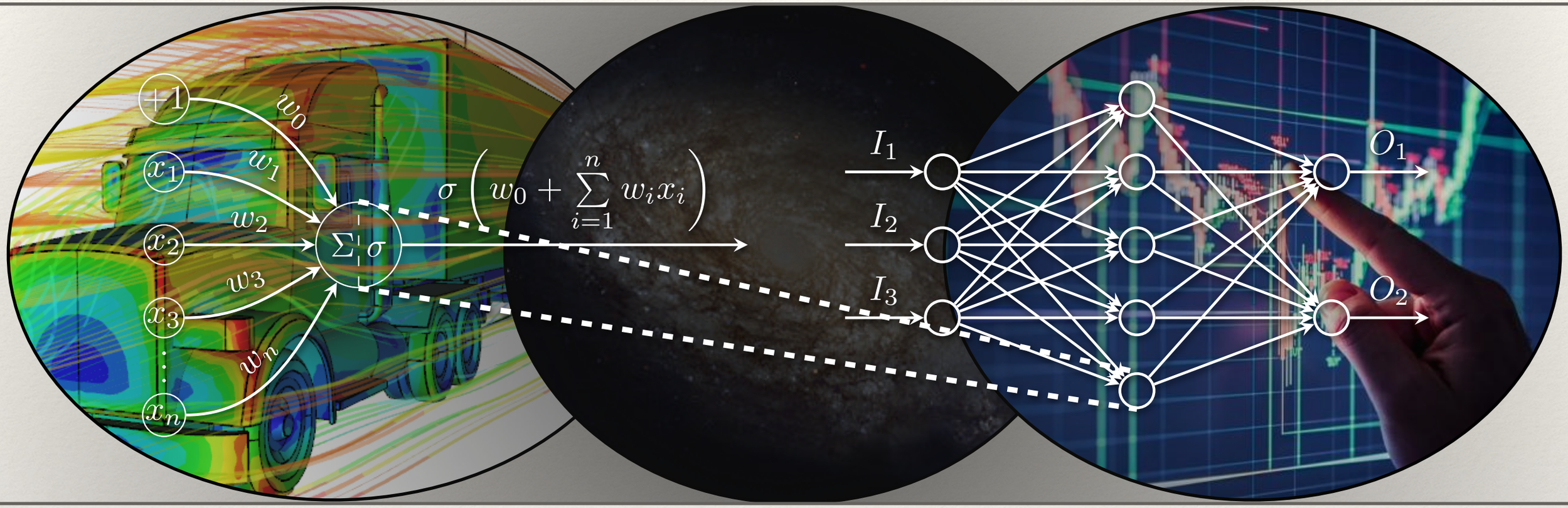


VKI Short Course

Introduction to Solving PDEs with Machine Learning

J.B. Scoggins
CMAP, Ecole Polytechnique
www.jbscoggins.com
@jb_scoggins

Before we begin

- **Tentative Course schedule**
  - 13:30-14:15 - Introduction to Machine Learning with Neural Networks
  - 14:20-15:00 - Solving PDEs with Neural Networks
  - 15:00-15:30 - Coffee Break
  - 15:30-16:15 - Introduction to Tensorflow
  - 16:20-17:00 - Solving PDEs with Tensorflow

- Access to this presentation and Jupiter notebooks: https://bit.ly/2uuRL85

- **Acknowledgments**
  - Thanks to Prof. Miguel Mendez for inviting me and organizing the short course
  - ML slides are based heavily on the course notes from *MAP 631: Deep Learning* at Ecole Polytechnique by Prof. Erwan Scornet

Outline

- **(Quick) Introduction to machine learning**
  - {supervised, unsupervised, reinforcement} learning

- **Artificial neural networks**
  - History
  - Hyperparameters
  - Regularization
  - Optimization
  - ML pipeline / best practices

- **Solving PDEs with neural networks**
  - Recall PDEs and solution techniques
  - Basic neural network approach
  - Advance methods (a survey)

- **Working with Tensorflow**

Outline

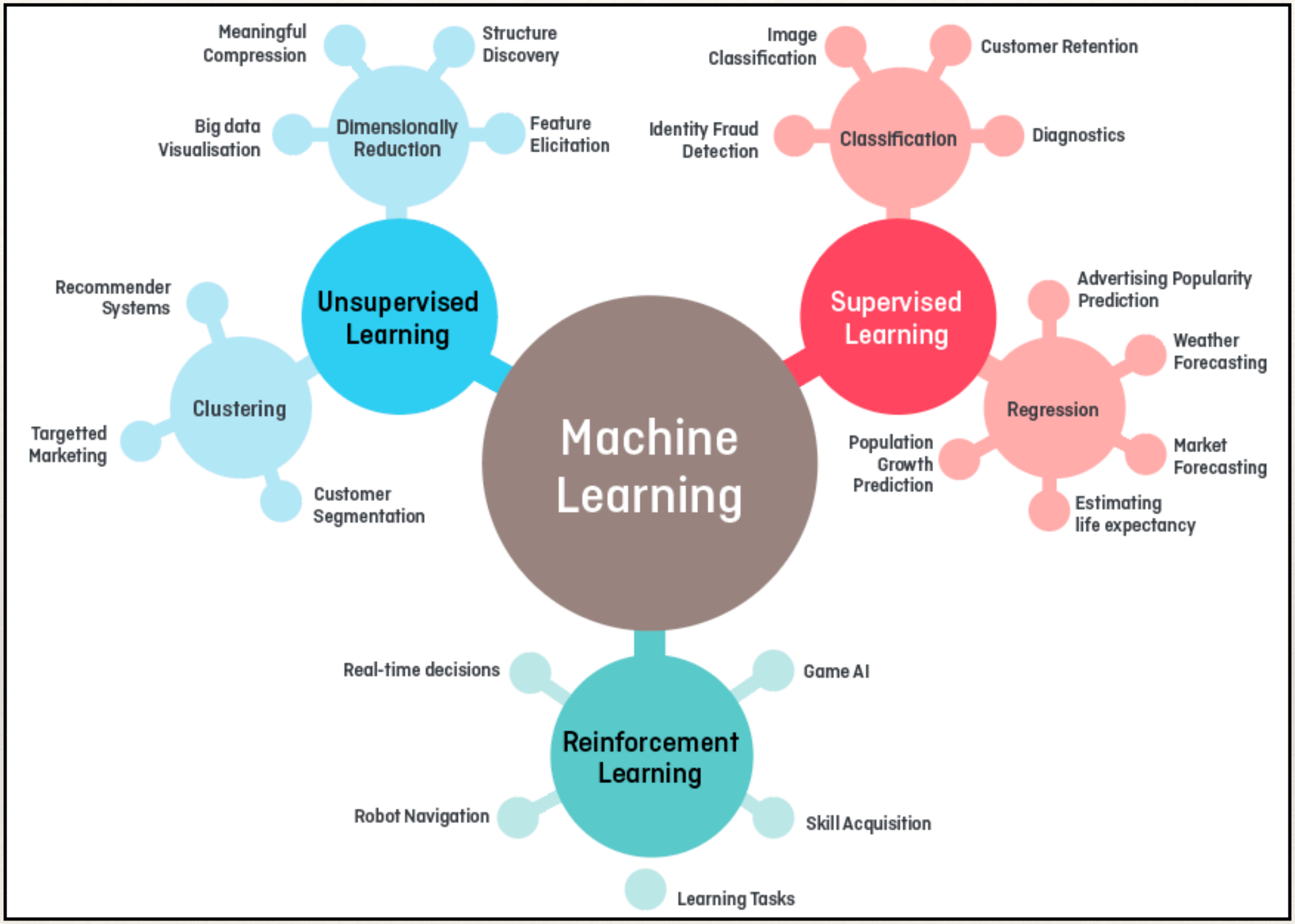
- **(Quick) Introduction to machine learning**
  - {supervised, unsupervised, reinforcement} learning

- **Artificial neural networks**
  - History
  - Hyperparameters
  - Regularization
  - Optimization
  - ML pipeline / best practices

- **Solving PDEs with neural networks**
  - Recall PDEs and solution techniques
  - Basic neural network approach
  - Advance methods (a survey)

- **Working with Tensorflow**

Machine Learning



"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." — Tom Mitchell (http://www.cs.cmu.edu/~tom/)

Supervised Learning

- **Problem setup**
  - Input measurement or data: $\mathbf{X} \in \mathcal{X}$
  - Output measurement, label, target: $\mathbf{Y} \in \mathcal{Y}$
  - $(\mathbf{X}, \mathbf{Y}) \sim \mathbb{P}$ with $\mathbb{P}$ unknown
  - Training data: $\mathcal{D}_n = \{(\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\} \quad (\text{i.i.d} \sim \mathbb{P})$

- **Goal**
  - Construct a good predictor $\hat{f}$ from $\mathcal{F} = \{f : \mathcal{X} \mapsto \mathcal{Y}\}$

- **Examples**
  - Classification: $\mathbf{X} \in \mathbb{R}^d, Y \in \{-1, 1\}$
  - Regression: $\mathbf{X} \in \mathbb{R}^d, Y \in \mathbb{R}$
  - Classification and regression are nearly the same problem!

As we will see later, we can formulate the PDE problem as a supervised learning task.

What do we mean with "good" predictor?

Loss and Probabilistic Framework

- **Loss (or cost) function for a generic predictor**
  - Loss function $l(Y, f(\mathbf{X}))$ measures the goodness of the prediction of Y by $f(\mathbf{X})$
  - Positive function, $l : (\mathcal{Y}, \mathcal{Y}) \mapsto \mathbb{R}_+$
  - Example: quadratic loss $l(Y, f(\mathbf{X})) = \|\mathbf{Y} - f(\mathbf{X})\|^2$

- **Expectation**
  - Expectation of a function of random variable is $\mathbb{E}_{x\sim p}[f(x)] \equiv \int f(x)p(x)dx$
  - Mean: $\mathbb{E}[X] = \bar{X}$
  - Variance: $\mathbb{V}[X] \equiv \mathbb{E}[(X - \bar{X})^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$

- **Risk function**
  - Risk measures the expected (average) loss for a new data pair
  - $R(f) = \mathbb{E}_{(\mathbf{X},Y)\sim\mathbb{P}}[l(Y, f(\mathbf{X}))]$
  - In the weeds: $\hat{f}$ depends on dataset $\mathcal{D}_n$ —> $R(\hat{f})$ is a random variable!

Supervised Learning

- **Problem setup**
  - Training data: $\mathscr{D}_n = \{(\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\} \quad (\text{i.i.d} \sim \mathbb{P})$
  - Predictor: $f : \mathscr{X} \mapsto \mathscr{Y}$ measurable
  - Cost/loss function: $l(Y, f(\mathbf{X}))$ measures how well $f(\mathbf{X})$ predicts Y
  - Risk: $R(f) = \mathbb{E}_{(\mathbf{X},Y)\sim\mathbb{P}}[l(Y, f(\mathbf{X}))] = \mathbb{E}_{\mathbf{X}}[\mathbb{E}_{Y|\mathbf{X}}[l(Y, f(\mathbf{X}))]$

- **Goal**
  - Construct a good predictor f from $\mathscr{F} = \{f : \mathscr{X} \mapsto \mathscr{Y}\}$

- **Precise goal**
  - Learn a rule to construct a predictor $\hat{f}$ from the training data $\mathscr{D}_n$ such that the risk is small on average (or with high probability w.r.t $\mathscr{D}_n$)

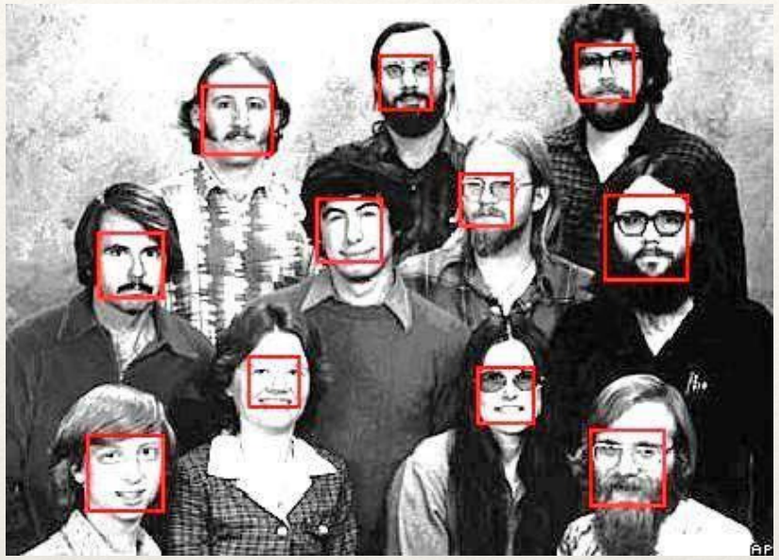
- **Optimal solution**
  - Best solution (independent of $\mathscr{D}_n$) satisfies
    $f^* = \arg\min_{f\in\mathscr{F}} R(f) = \arg\min_{f\in\mathscr{F}} \mathbb{E}[l(Y, f(\mathbf{X}))]$
  - Regression with quadratic loss: $f^* = \mathbb{E}[Y|\mathbf{X}]$
  - Requires knowledge of $\mathbb{E}[Y|\mathbf{X}]$ for all values of $\mathbf{X}$!

Examples of supervised learning

Spam detection

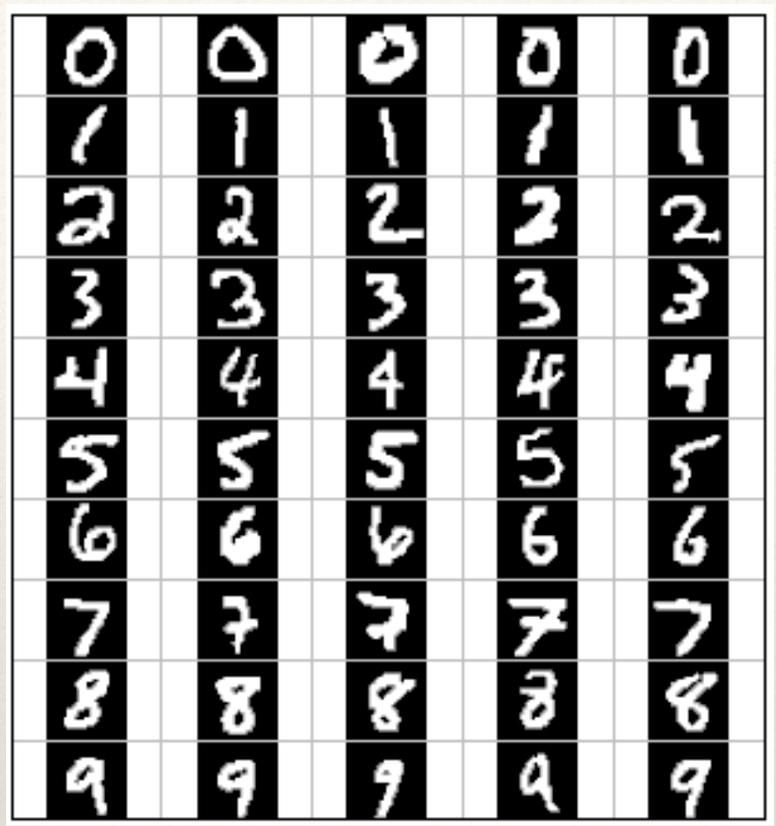
Data: Email collection labelled “spam” or “no spam”

Face detection



Data: Annotated database of images. Input is sub window with “face” or “no face label”

Digit recognition



Data: Annotated database of images, labelled with corresponding digit.

Unsupervised Learning

- **Problem setup**
  - Training data: $\mathscr{D}_n = \{\mathbf{X}_1, \ldots, \mathbf{X}_n\}$ (i.i.d. $\sim \mathbb{P}$)
  - Task: ?
  - Performance measure: ?

No obvious task definition!

- **Clustering**
  - Group data into "similar" classes ("unsupervised classification")
  - Knowledge discovery
  - Algorithms: k-means, mixture models, EM, spectral methods, …

- **Dimensionality Reduction**
  - Construct map of data to a low dimensional space without distortion
  - Feature selection/extraction
  - Data projection
  - Algorithms: PCA, POD, Auto-Encoders, …

Reinforcement Learning

- **Problem setup**
  - Agent acting in an environment with observable states (state space)
  - Observe the environment's state, take actions (action space), and earn rewards
  - Reward signal possibly delayed, sparse, or weak
  - State and action spaces can be continuous or discrete
  - Actions may have stochastic outcomes

- **Goal**
  - Learn mapping from observations to actions (policy) to maximize reward

- **Applications**
  - Games (GO, Atari, StarCraft, …)
  - Autopilot for robots, autonomous vehicles, etc.
  - Traffic control
  - Trading/manage investment portfolios

Outline

- **(Quick) Introduction to machine learning**
  - {supervised, unsupervised, reinforcement} learning

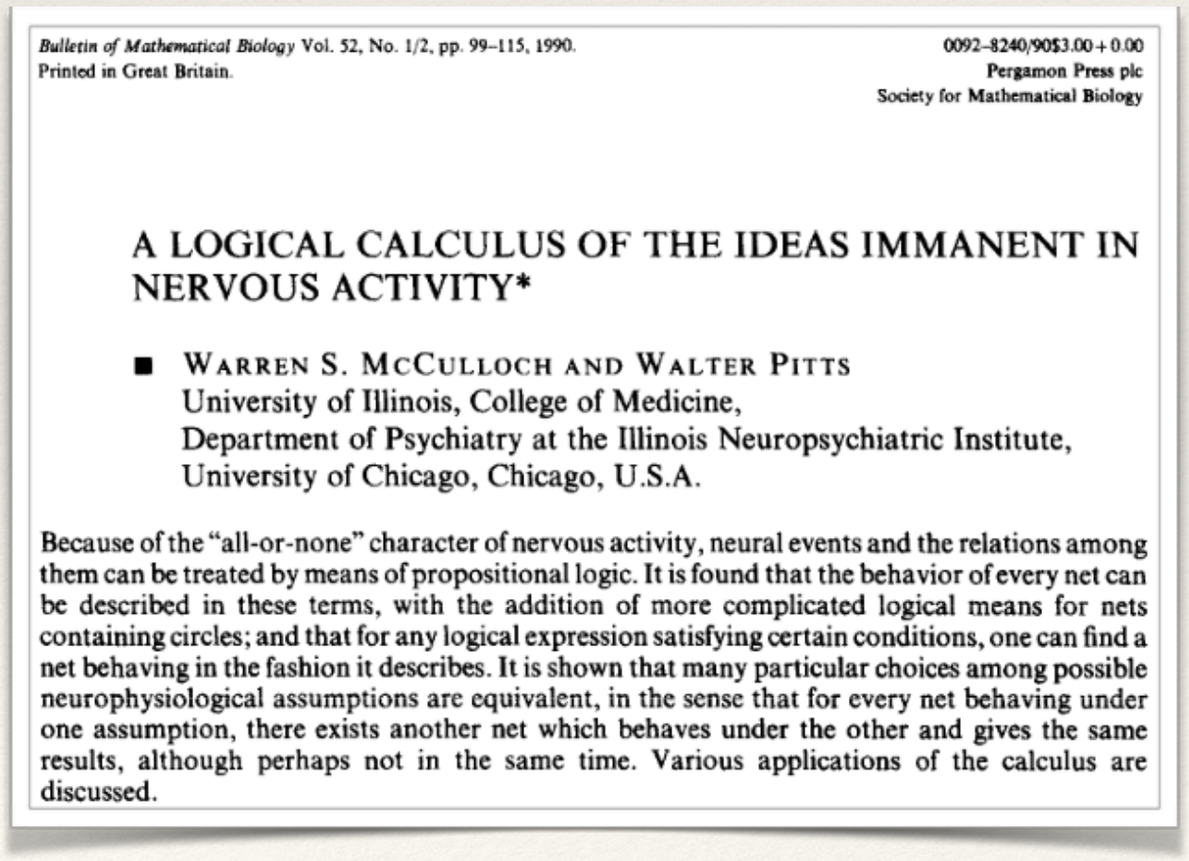
- **Artificial neural networks**
  - History
  - Hyperparameters
  - Regularization
  - Optimization
  - ML pipeline / best practices

- **Solving PDEs with neural networks**
  - Recall PDEs and solution techniques
  - Basic neural network approach
  - Advance methods (a survey)

- **Working with Tensorflow**

Modeling neurons

- First electrical circuits designed to model how neurons function developed by Warren McCulloch and Walter Pitts in 1943.
- Termed "connectionism" and used connected circuits to simulate intelligent behavior.
- Around 1950', Frank Rosenblatt developed the perceptron to model decision systems in a fly's eye based on the McCulloch-Pitts neuron.
- Mark 1 Perceptron implemented in 1958.



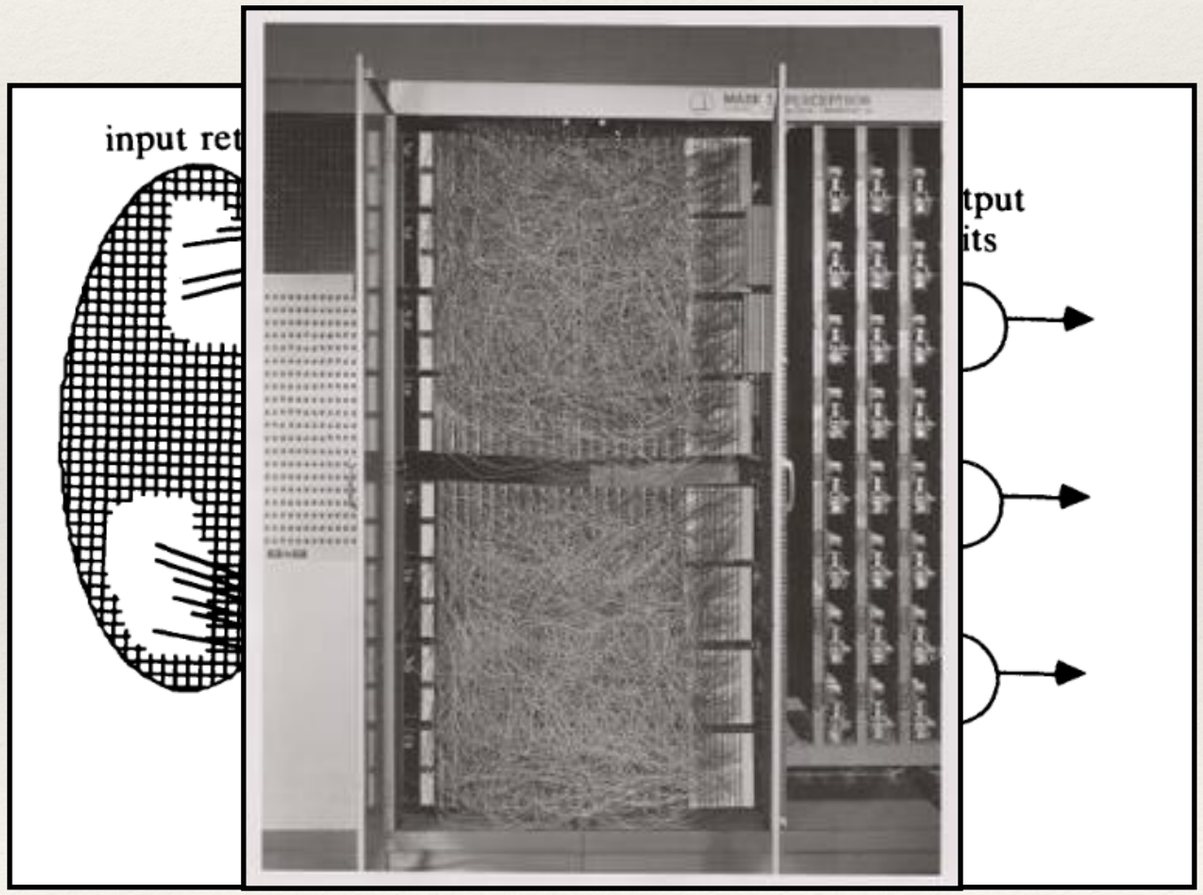
Bulletin of Mathematical Biology Vol. 52, No. 1/2, pp. 99–115, 1990.
Printed in Great Britain.

0092–8240/90$3.00 + 0.00
Pergamon Press plc
Society for Mathematical Biology

A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

WARREN S. MCCULLOCH AND WALTER PITTS
University of Illinois, College of Medicine,
Department of Psychiatry at the Illinois Neuropsychiatric Institute,
University of Chicago, Chicago, U.S.A.

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.



Mark 1 Perceptron.

Representation of a perceptron.

Early artificial neural networks

- In 1959, Bernard Widrow and Marcian Hoff developed the AdaLine (ADAptive LINear Elements) and MAdaLinE (Multiple AdaLinE) networks at Stanford.
- MAdaLinE was first neural network successfully applied to a real world problem.
  - Speech and pattern recognition
  - Weather forecasting
  - Adaptive filtering and signal processing

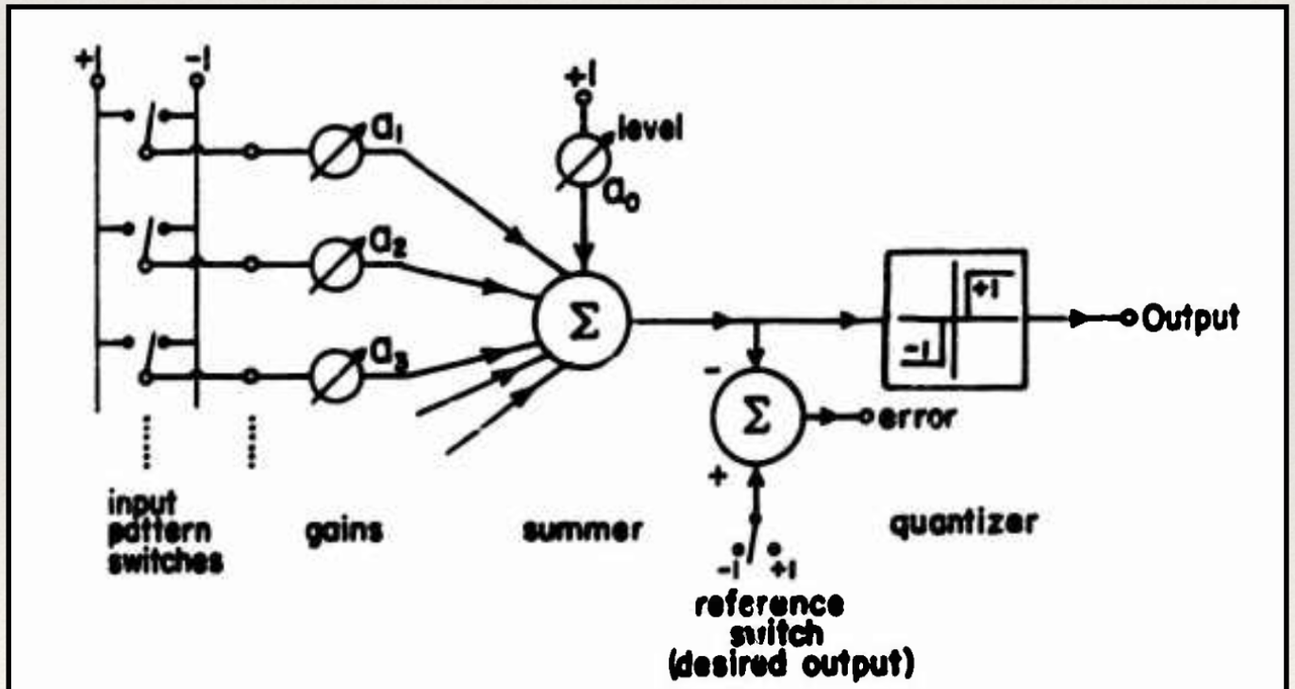


FIG. 3.--SCHEMATIC OF ADALINE.

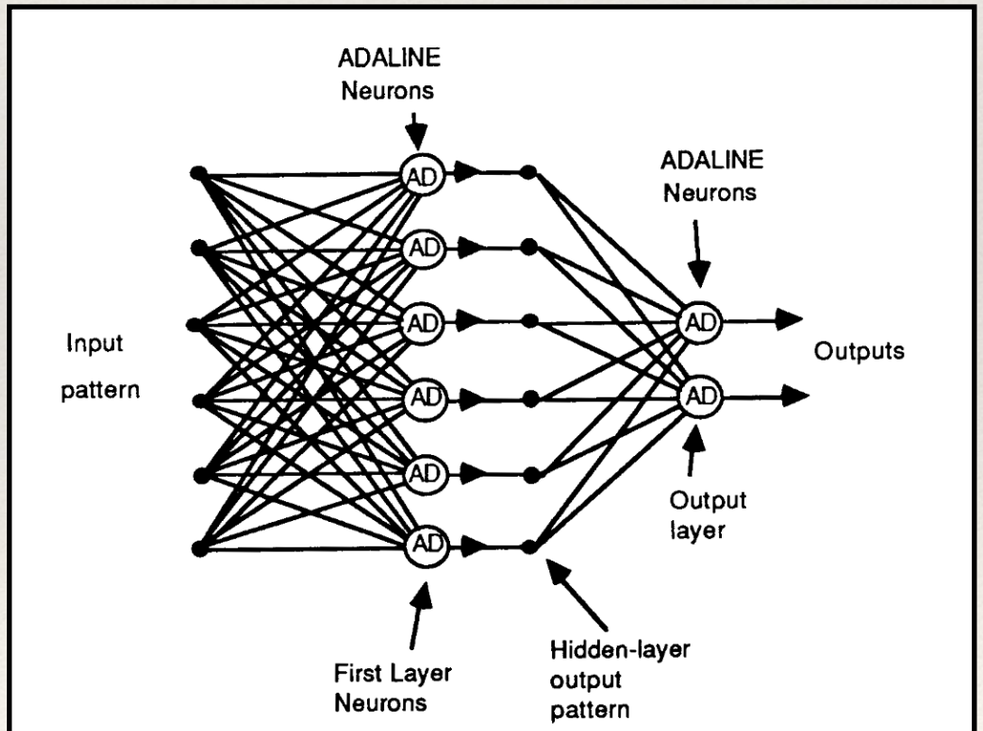


Figure 1: Layered feed-forward ADALINE network.

A promising future ahead!

"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language, it was predicted."

— Press conference, 7 July 1958, New York Times

AI Winter

- In 1969, Minsky and Papert publish a book showing that it was difficult for the perceptron to detect parity and connectedness, not possible to represent XOR.
- This book is the starting point of a period known as "AI winter", corresponding to a significant decline in funding of neural network research lasting about 20 years.
- Ignited a controversy between Rosenblatt and Minsky, Papert. (see "A sociological study of the official history of the perceptrons controversy" by Olazaran in 1996)

"There is no reason to suppose that any of [the virtue of perceptrons] carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting "learning theorem" for the multilayered machine will be found."

— Minsky and Papert, *Perceptrons: An introduction to computational geometry (1969)*

Exercise: logical functions

- The logical AND function is defined for two binary variables $x_1, x_2 \in \{0,1\}$ as

$$\text{AND}(x_1, x_2) = \begin{cases} 1 & x_1 = x_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

- The logical XOR function is defined for two binary variables $x_1, x_2 \in \{0,1\}$ as

$$\text{XOR}(x_1, x_2) = \begin{cases} 0 & x_1 = x_2 = 0 \text{ or } x_1 = x_2 = 1 \\ 1 & \text{otherwise} \end{cases}$$

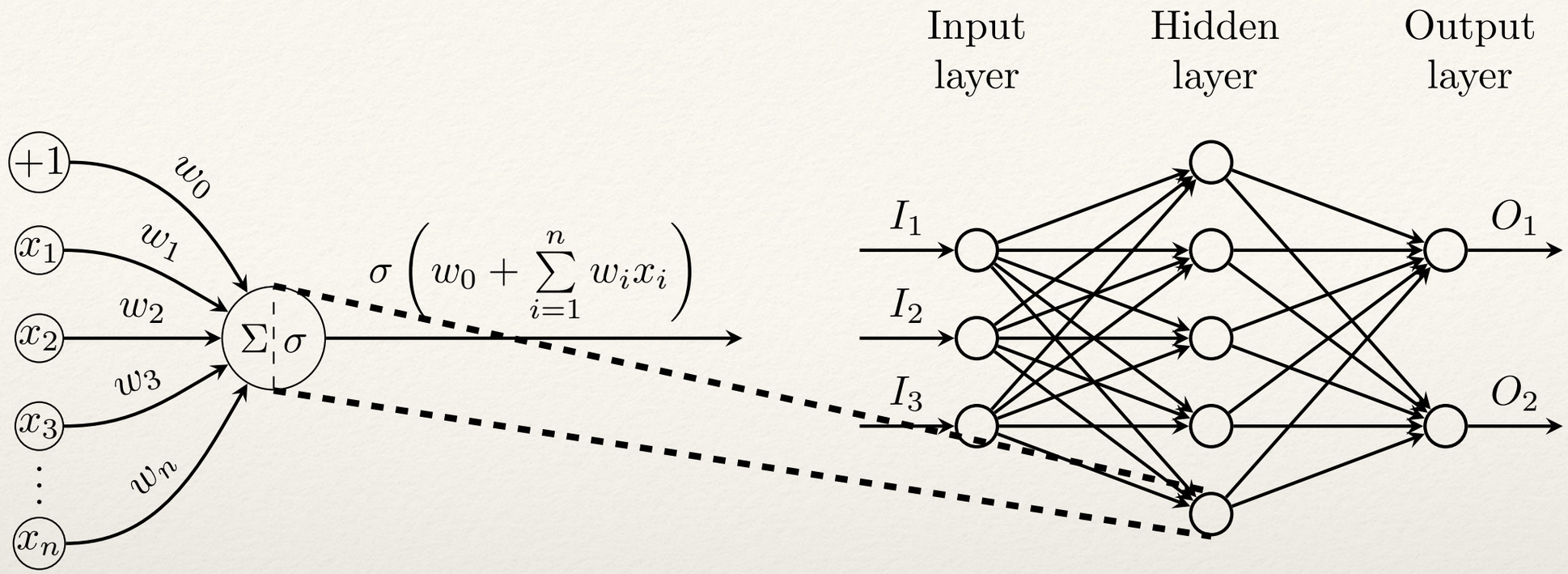
- Show how the logical AND function can be represented by a single layer perceptron.

- Prove that no single layer perceptron can implement the XOR function. Hint: think about the problem graphically.

- Find a neural network with one hidden layer that implements the XOR function.

- **Interesting fact**: Any logical function can be represented as a combination of NAND gates, which can be represented as a network. Thus, any logical function (and by extension, any mathematical function) can be represented by a neural network. This is one version of the Universal Approximation Theorem for ANNs.

Dense networks with multiple layers

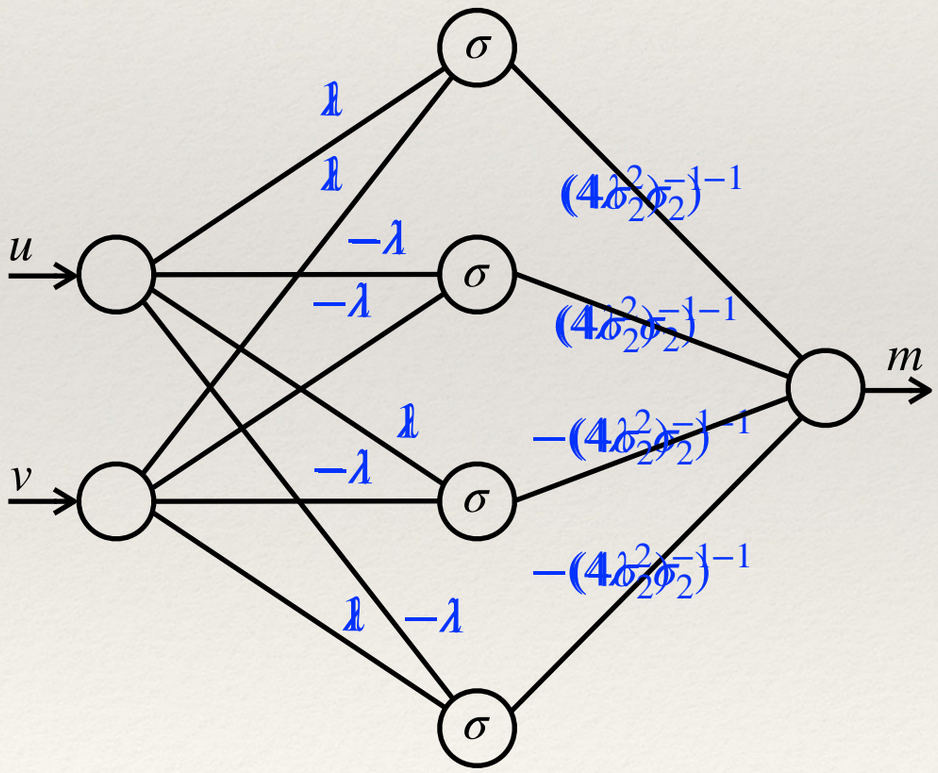


Multilayer Perceptron (MLP)

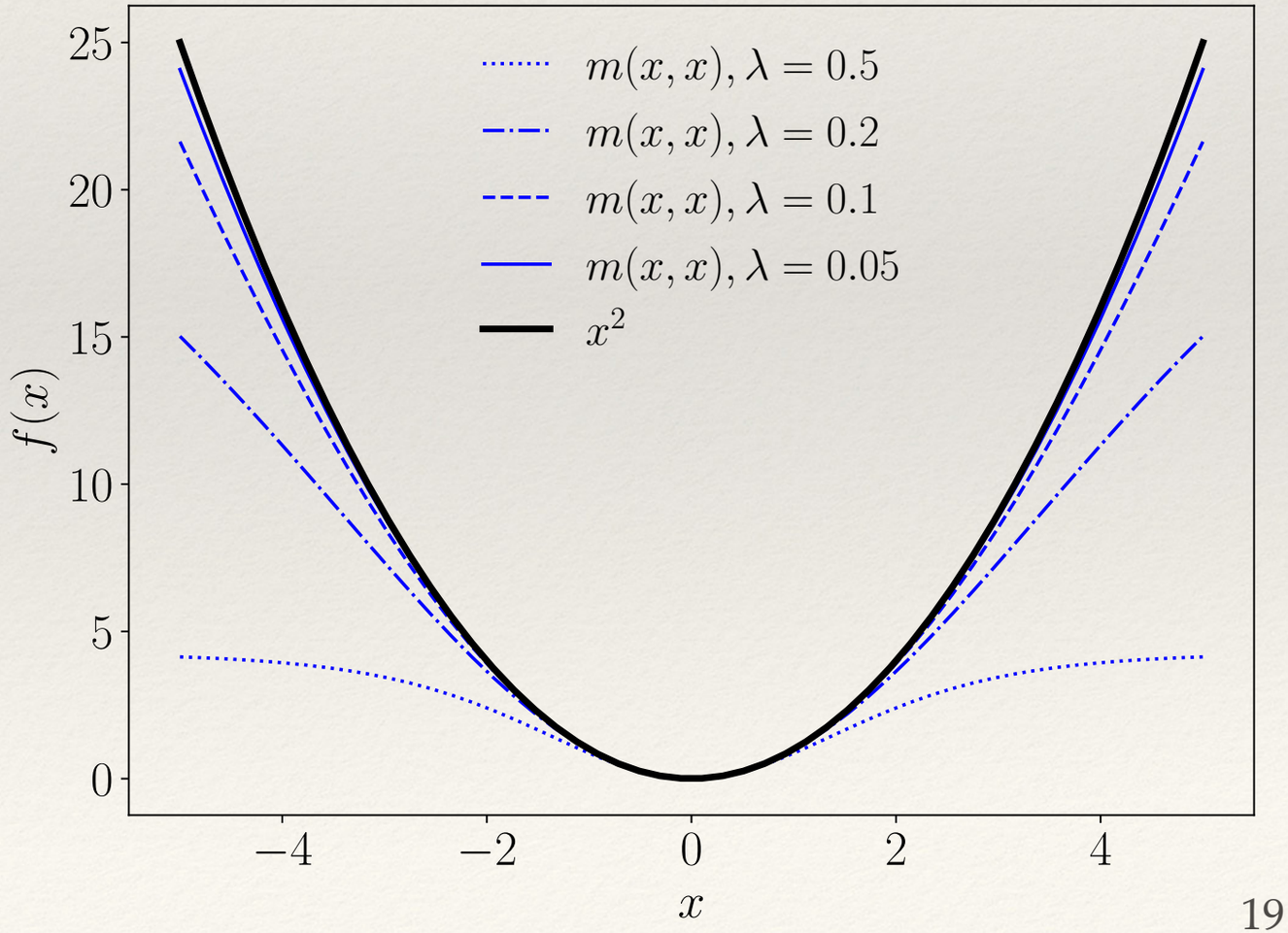
- **Generic notation**
  - $W_{ij}^{(l)}$: weights between neuron j in layer $l-1$ and neuron i if layer l
  - $b_j^{(l)}$: bias of neuron j in layer l
  - $z_j^{(l)}$: input of neuron j in layer l, such that $z_j^{(l)} = \sum_i W_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$
  - $a_j^{(l)}$: output of neuron j in layer l, such that $a_j^{(l)} = \sigma^{(l)}(z_j^{(l)})$, called activation
  - $\sigma^{(l)}$: activation function used for layer l, such that $\sigma : \mathbb{R} \mapsto \mathbb{R}$
  - x_j : input j to the network, such that $a_j^{(0)} \equiv x_j$
  - y_j : output j of the network, such that $y_j \equiv a_j^{(L)}$
  - L : depth of the network, there are $L-1$ hidden layers

Simple functions as neural networks

- **Multiplication operator**
  - Consider Taylor expansion of activation function:
    $\sigma(u) = \sigma_0 + \sigma_1 u + \sigma_2 u^2/2 + \mathcal{O}(u^3)$
  - With a bit of algebra, we can show that
    $$m(u,v) \equiv \frac{\sigma(u+v) + \sigma(-u-v) - \sigma(u-v) - \sigma(-u+v)}{4\sigma_2} = uv\left[1 + \mathcal{O}(u^2+v^2)\right]$$

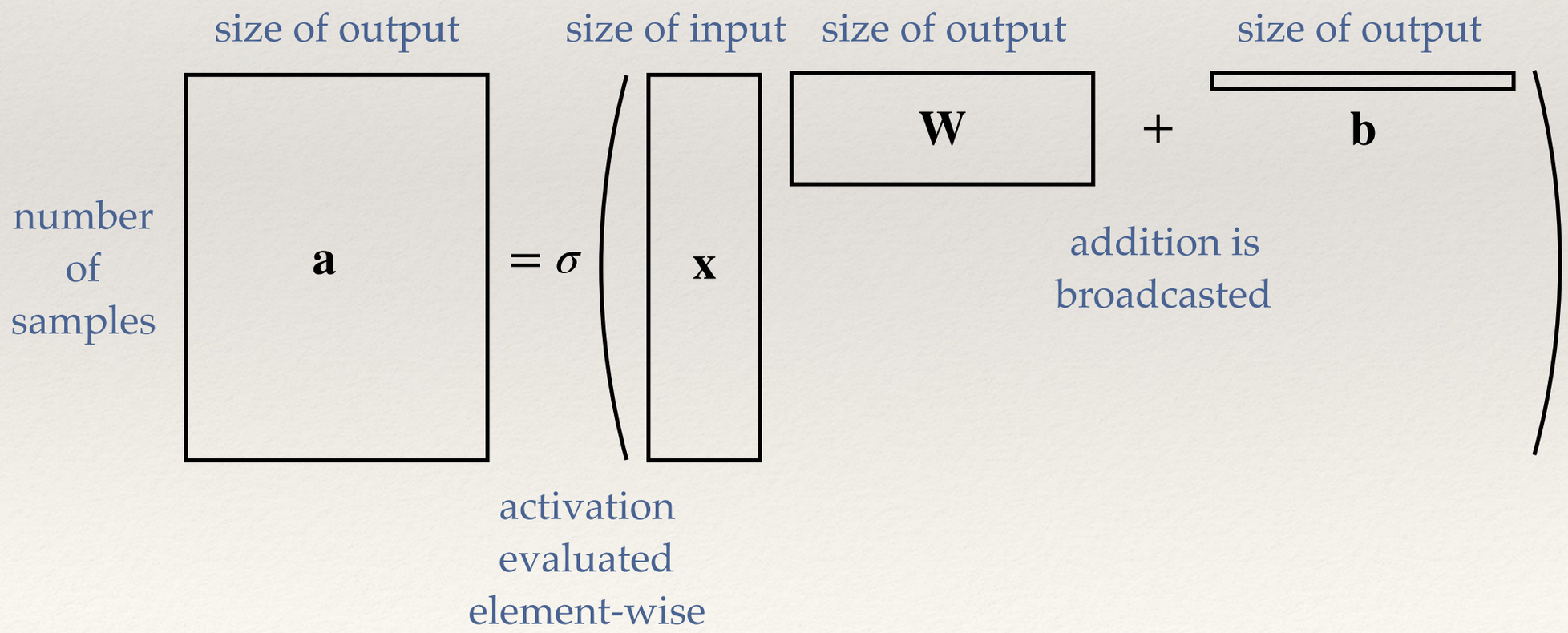


Multiplication network



Matrices and vectors

- Typically think of neural network as series of matrix/vector operations
- Consider dense feed forward network, single hidden layer, linear output layer
  $y_k = \sum_j W^{(2)}_{jk} \sigma(\sum_i W^{(1)}_{ij} x_i + b^{(1)}_j) + b^{(2)}_k$
- In matrix/vector notation, this is written as $\mathbf{y} = \mathbf{W}^{(2)} \sigma(\mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}$
- Think of rows as samples, columns as features/nodes/targets
- Far more efficient to evaluate many samples at once with matrix-vector operations



Gradient descent

- How do we learn the weights and biases from data (supervised learning)?
- **Gradient Descent**
  - Consider all trainable parameters in the network $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots\}$
  - Minimize the empirical risk:
    $\theta^* = \arg\min_\theta \mathscr{L} \equiv \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{y}_i, f(\mathbf{x}_i; \theta))$
  - Gradient of the loss w.r.t. parameters: $\nabla_\theta \mathscr{L} = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta l_i$
  - **Update rule**: $\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathscr{L} = \theta_t - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_\theta l_i$
  - Learning rate η controls speed of descent (stability)
- **(Batch) Stochastic Gradient Descent (SGD)**
  - Approximate loss gradient with fewer samples than the whole dataset
  - Sample points $I_t \subset \mathscr{D}_N$
  - **Update rule**: $\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathscr{L} = \theta_t - \frac{\eta}{|I_t|} \sum_{i \in I_t} \nabla_\theta l_i$
- Guaranteed to converge on convex problems, in practice works on non-convex too

Backpropagation (backprop)

- Need to compute the gradient of the loss w.r.t. trainable parameters
- Analytical derivatives are error-prone and expensive to implement/maintain
- Finite-differences require $|\theta| + 1$ evaluations
- **Backpropagation to the rescue!**
  - Introduced by Werbos in 1974
  - Popularized by Rumelhart, McClelland, and Hinton in 1986
  - Successive evaluations of the chain rule with touch of dynamic programming
  - Reignited neural network research
  - Still at the core of Deep Learning today

Backprop equations (dense networks)

- Consider dense network with L layers, quadratic cost, vectorial output
  $\mathscr{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{a}^{(L)}\|^2$
- Let's define the quantity
  $\delta_j^{(l)} \equiv \dfrac{\partial \mathscr{L}}{\partial z_j^{(l)}}$
- We can then derive the four fundamental equations of backpropagation
  - $\delta^{(L)} = \nabla_{\mathbf{a}} \mathscr{L} \odot \sigma'(\mathbf{z}^{(L)})$
  - $\delta^{(l)} = (\delta^{(l+1)}\mathbf{W}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$
  - $\nabla_{\mathbf{b}^{(l)}} \mathscr{L} = \delta^{(l)}$
  - $\nabla_{\mathbf{W}^{(l)}} \mathscr{L} = (\mathbf{a}^{(l-1)})^T \delta^{(l)}$

Backprop algorithm

- The backdrop equations lead to an efficient and elegant algorithm
- **Backpropagation algorithm**
  - Randomly initialize weights and biases in the network
  - While not done (stopping criteria):
    - Feedforward: evaluate network with input vector $\mathbf{a}^{(0)} = \mathbf{x}$ and store activations $\mathbf{a}^{(l)}$, pre-activations $\mathbf{z}^{(l)}$, and derivatives $\sigma'(\mathbf{z}^{(l)})$
    - Compute cost $\mathscr{L}$ and it's derivative $\nabla_{\mathbf{a}^{(L)}}\mathscr{L}$
    - Backpropgate: Recursively compute the vectors $\delta^{(l)}$ and update the weights and biases using the backprop equations
- Some terminology
  - An epoch is one forward pass and one backward pass over all training examples.
  - A (mini) batch is a set of training examples for one forward/backward pass. More samples in a batch means better accuracy for gradients, but more costly.
  - An iteration is a forward/backward pass over a single batch with update.
  - **Example**: dataset with 1000 samples, batch size of 200, needs 5 iterations per epoch

Hyperparameters

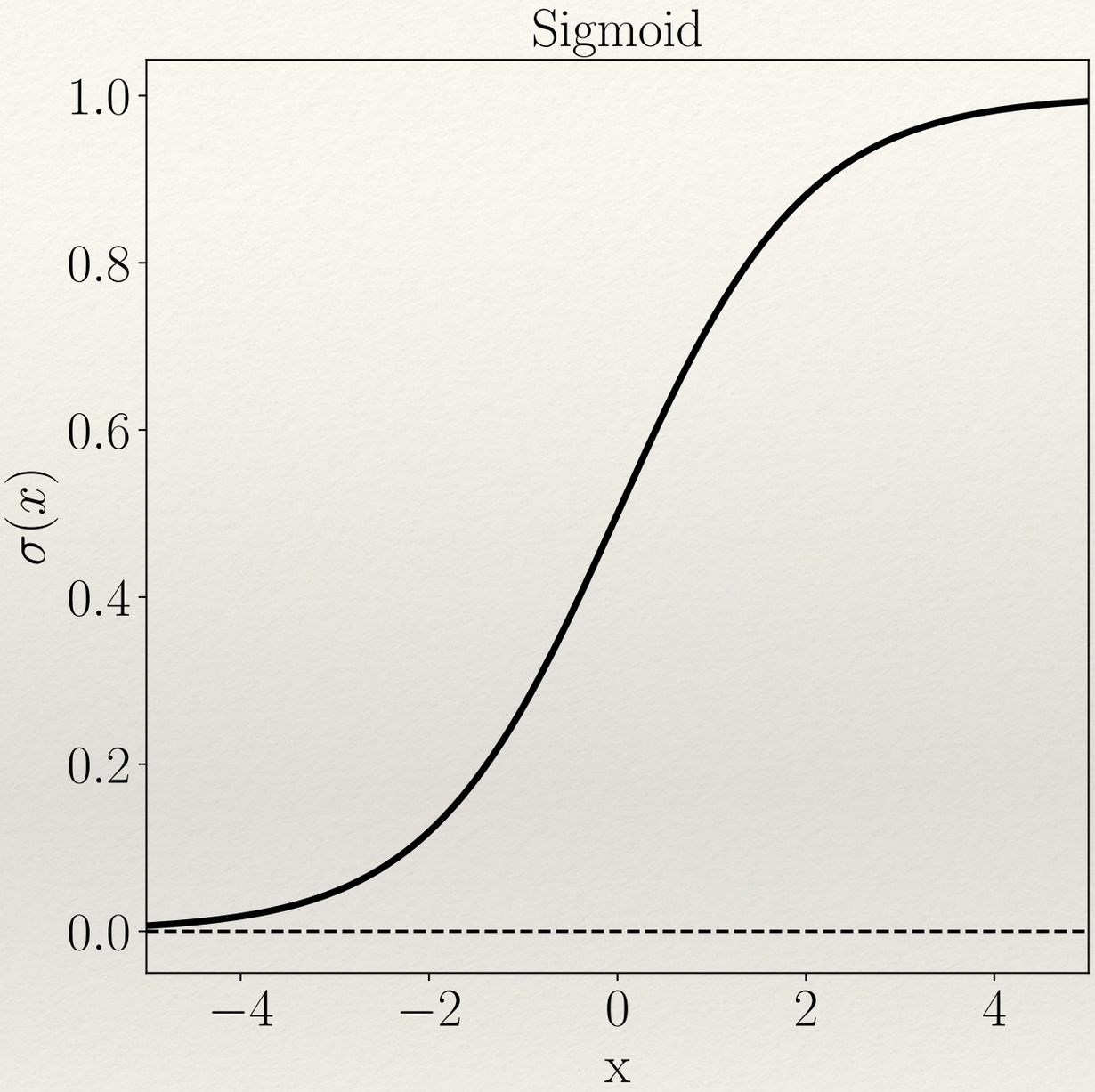
- A hyperparameter is a parameter or network design/optimization choice that is not directly learned during the optimization
- We have already seen many of these
  - activation functions (hidden layers)
  - output units (activation of the last layer)
  - loss functions
  - weight initialization
  - optimization parameters such as learning rate, batch size, etc.
  - stopping criteria
- Can have a huge impact on performance and training of the neural network model
- In practice, hyperparameters are also optimized! Priority on accuracy and efficiency of model, not on training time.

Activation functions

- **Sigmoid function**

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

- Pros
  - Smooth squashing function
- Cons
  - Not zero centered
  - Saturated, gradient killer
  - Exponential is expensive

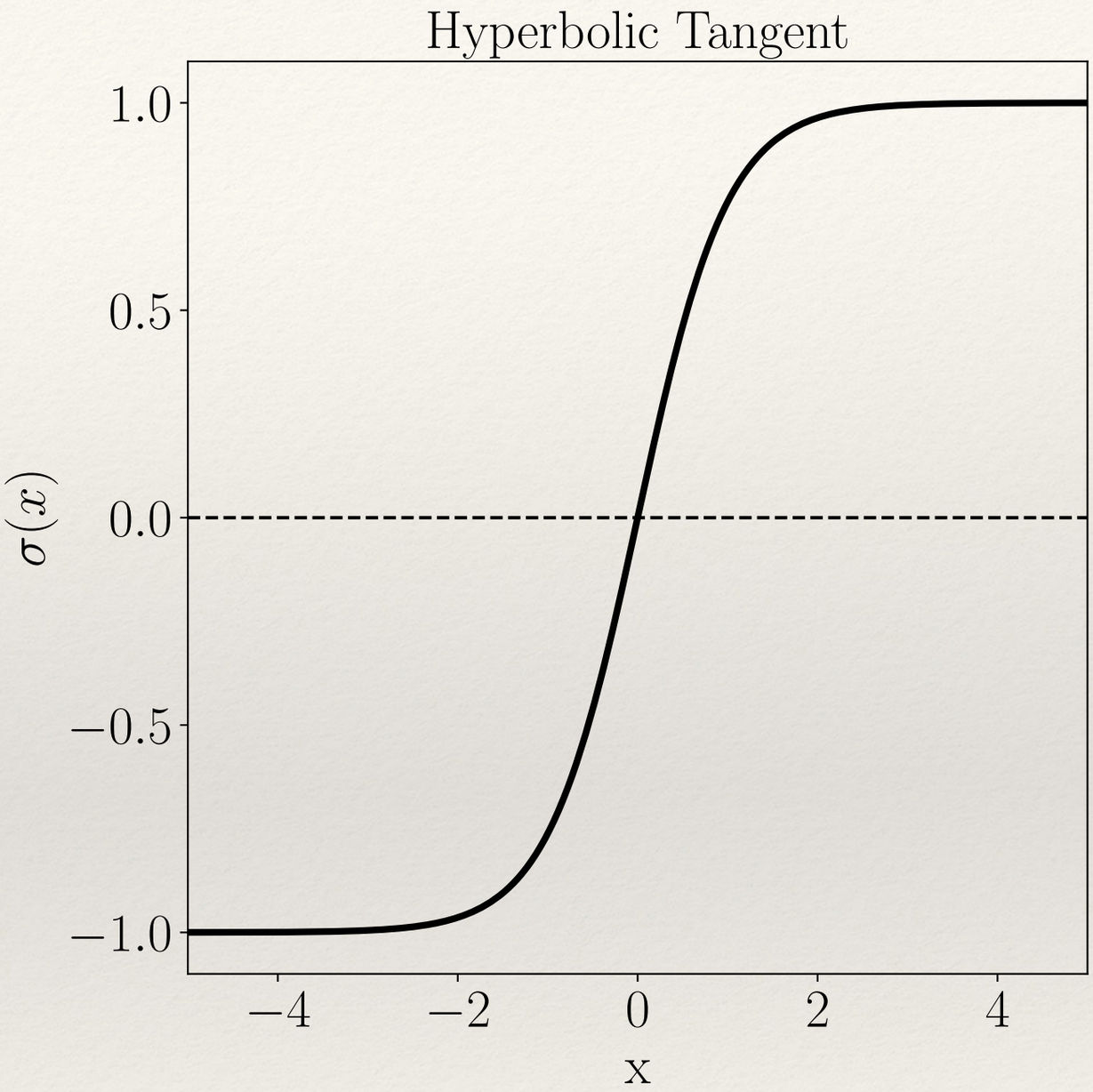


Activation functions

- **Hyperbolic tangent function**

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

- Pros
  - Smooth squashing function
  - Zero-centered
- Cons
  - Saturated, gradient killer
  - Exponential is expensive
- Note: $\tanh(x) = 2\sigma(2x) - 1$

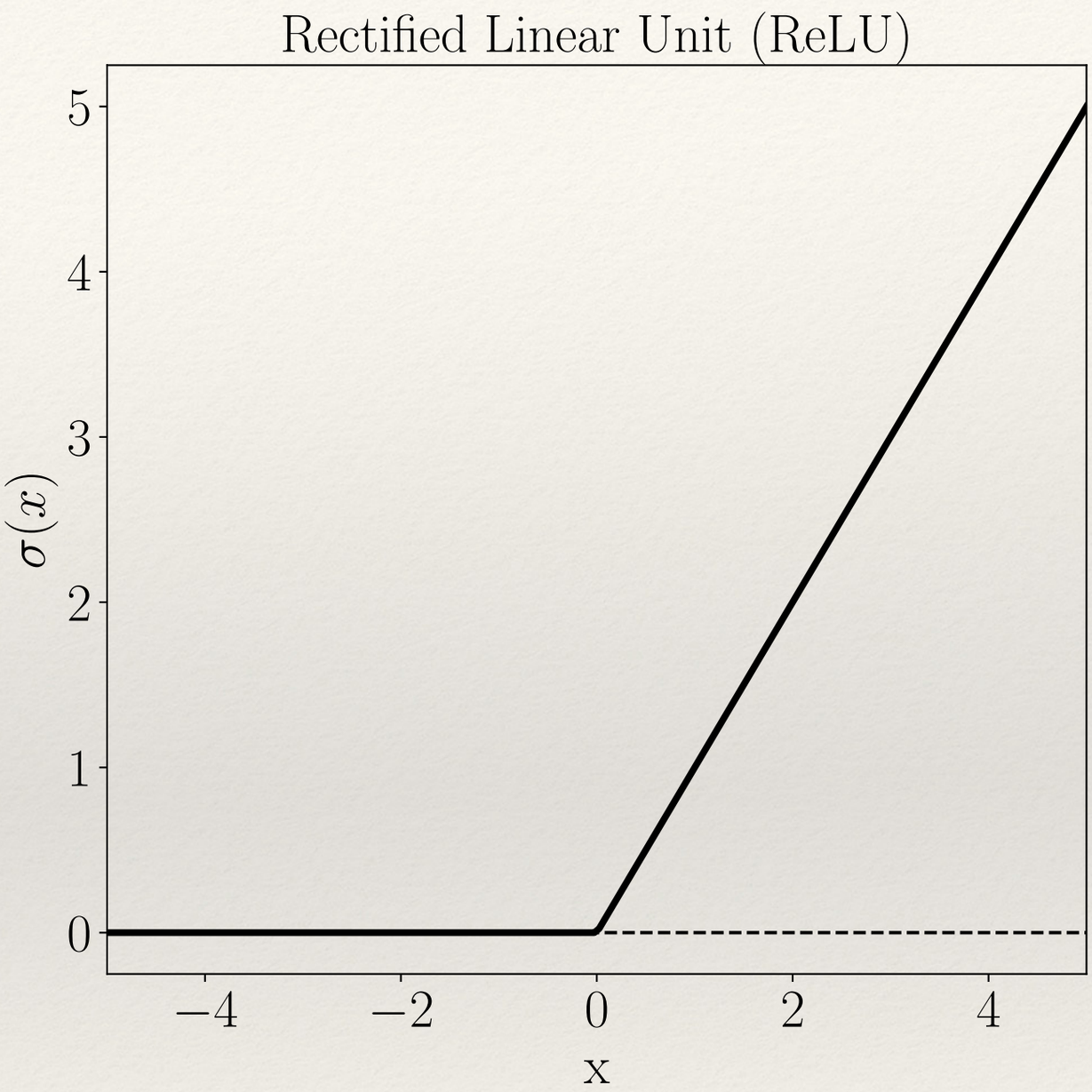


Activation functions

- **Rectified linear unit (ReLU)**

$$\text{relu}(x) = \max(0, x)$$

- Pros
  - Not saturated
  - Efficient
  - Faster empirical convergence
  - Biologically plausible
- Cons
  - Not zero-centered
  - Requires proper initialization
  - Zero gradient for negative inputs
  - Zero higher-order derivatives (important for PDEs…)

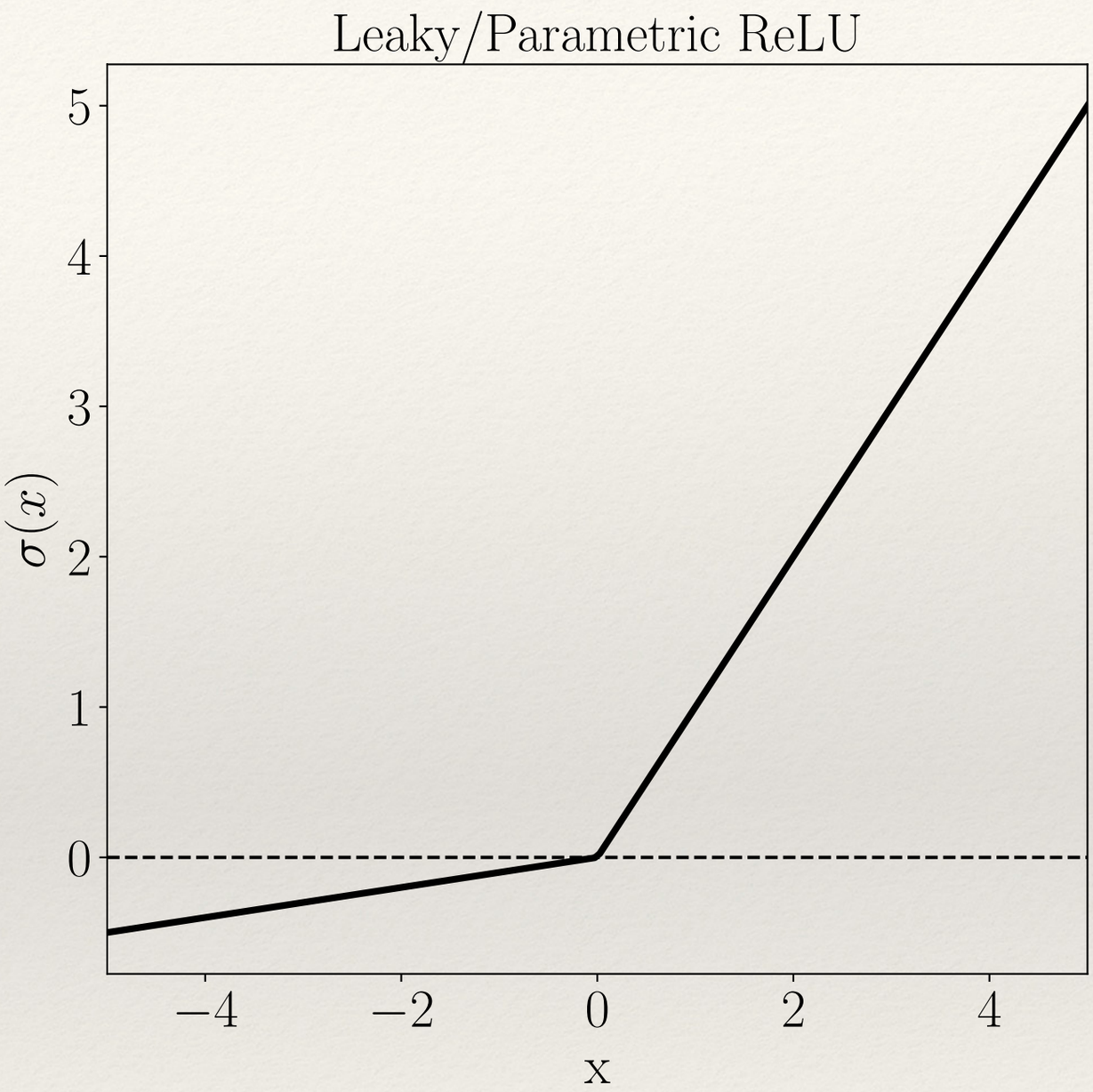


Activation functions

- **General ReLU**

  $\text{relu}(x; \alpha) = \max(\alpha x, x)$

- **Leaky ReLU**
  $0 < \alpha < 1$
- **Absolute value rectification**
  $\alpha = -1$
- **Parametric ReLU**: α optimized
- Pros
  - ReLU pros +
  - Less sensitive to initialization
  - Non-zero gradients
- Cons
  - Remaining ReLU cons

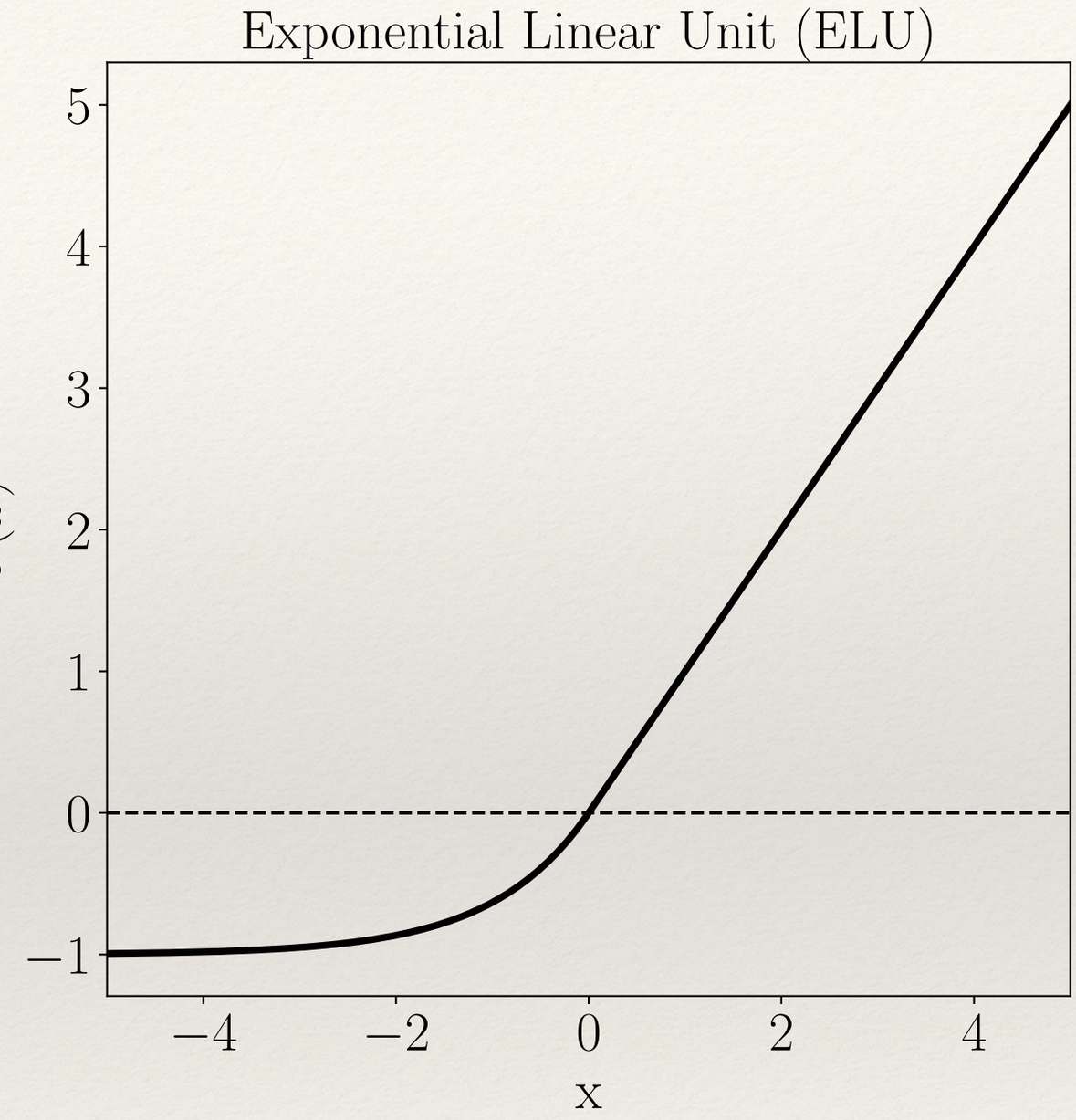


Activation functions

- **Exponential linear unit (ELU)**

$$\mathrm{elu}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \exp(x) - 1 & \text{otherwise} \end{cases}$$

- Pros
  - Not saturated
  - In saturation regime, closer to zero mean output
  - More robust to noise
- Cons
  - Not zero-centered
  - Zero gradient for negative inputs
  - Zero higher-order derivatives (important for PDEs…)



Activation functions

- Conclusions for typical ML problems
  - Use ReLU
  - Test Leaky ReLU, ELU, maxout
  - Try tanh but don't expect too much
  - Don't use sigmoid
- Conclusions for solving PDEs
  - Same logic doesn't necessarily apply
  - High-order derivatives are much more important
  - More on this later

Output units

- **Linear output units**:

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

  —> linear regression based on new variables $\mathbf{h}$

- **Sigmoid output unit:**

  $p(\mathbf{y} = 1 \mid \mathbf{h}) = \sigma(\mathbf{h}\mathbf{W} + b)$

  —> used to predict $\{0, 1\}$outputs, logistic regression based on new variables $\mathbf{h}$

- **Softmax output unit:**

  $$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{k=1}^{K} \exp(z_k)}$$

  where $\mathbf{z}$ is the pre-activation based on the previous layer, $\mathbf{z} = \mathbf{h}\mathbf{W} + \mathbf{b}$

  —> multinomial logistic regression based on new variables $\mathbf{h}$

- We are only concerned with linear outputs in this course…

Loss functions

- Recall the cost function

$$\mathscr{L} = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}_i, f(\mathbf{x}_i))$$

- **Mean squared error (MSE)**

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2$$

- **Mean absolute error (MAE)**

$$l(\mathbf{y}, \hat{\mathbf{y}}) = |\mathbf{y} - \hat{\mathbf{y}}|$$

Backprop Equations

- $\delta^{(L)} = \nabla_{\mathbf{a}} \mathscr{L} \odot \sigma'(\mathbf{z}^{(L)})$
- $\delta^{(l)} = (\delta^{(l+1)}\mathbf{W}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$
- $\nabla_{\mathbf{b}^{(l)}} \mathscr{L} = \delta^{(l)}$
- $\nabla_{\mathbf{W}^{(l)}} \mathscr{L} = (\mathbf{a}^{(l-1)})^T \delta^{(l)}$

- **Cross-entropy** (negative log-likelihood)

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \ln(\hat{y}_k) + (1 - y_k)\ln(1 - \hat{y}_k), \text{ where } y_k \in \{0,1\}$$

- How do we decide which loss to use for our problem?
  - ➡ Avoid vanishing gradients!

	MSE	Cross-entropy
Linear	$\delta^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}$	~~$\delta^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y})[\mathbf{a}^{(L)}(1 - \mathbf{a}^{(L)})]^{-1}$~~
Sigmoid	~~$\delta^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y})\sigma'(\mathbf{z}^{(L)})[1 - \sigma'(\mathbf{z}^{(L)})]$~~	$\delta^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}$

Weight initialization

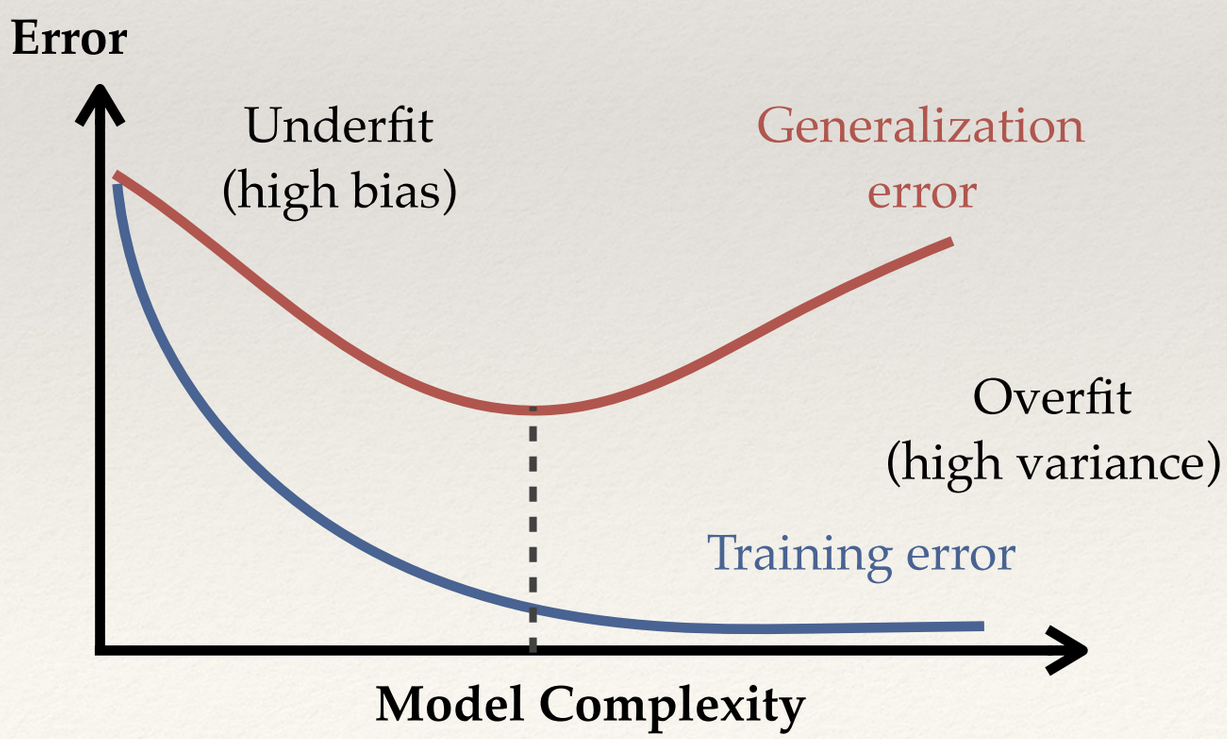
- Set all weights to same value or zero? Bad idea!
- **Better idea**: use random weights initially. What distribution?
  - Small random values, sampled from $0.01\mathcal{N}(0,10^{-4})$? Nope, becomes a Dirac function at zero after a few layers.
  - Large random values, for example $\mathcal{N}(0,10^{-4})$? Also bad, quickly saturates.
  - At any rate, we can safely set biases to zero.
- **Even better idea**: make variance of the output match variance of the input.
- Xavier initialization: bias to zero with weights sampled from
$$\mathcal{U}\left(-\sqrt{\frac{6}{n_j+n_{j+1}}},+\sqrt{\frac{6}{n_j+n_{j+1}}}\right)$$
- He et al. initialization: bias to zero with weights sampled from
$$\mathcal{N}\left(0,\sqrt{\frac{2}{n_j}}\right)$$

Exercise: weight initialization

- Consider a neural network with two hidden layers containing n_1 and n_2 neurons, respectively, with linear hidden units. Find a simple sufficient condition on weights so that the variance of the hidden units stay constant across the layers.
- Using the same network, find a simple sufficient condition on the weights so that the gradient stay constant across layers when applying backpropagation.
- Based on previous questions, propose a simple way to initialize weights.

Regularization

- In general all ML models have to deal with the bias-variance tradeoff.
- Avoid overfitting with regularization by imposing constraints on parameter space.
  - Early stopping: stop training when validation error increases
  - Penalization (L1 or L2): replace cost function with
    $\tilde{\mathscr{L}}(\mathbf{X}, \mathbf{Y}, \theta) = \mathscr{L}(\mathbf{X}, \mathbf{Y}, \theta) + \lambda \, \mathrm{pen}(\theta)$
  - Dropout: randomly kill some neurons during optimization, predict with full NN
  - Soft weight sharing: reduce parameter space using explicit constraints (i.e. CNN)
- Less of a problem with PDEs (unlimited "data")…



Pipeline for NN learning

1. **Preprocess data**
   - Subtract mean and divide by standard deviation
   - Construct feature and target tensors, divide into training, test, validation sets
2. **Choose an architecture for the network**
   - Layers, nodes per layer, activations, output, loss
3. **Train**
   3.1. Optimize network on training data. Check if loss is reasonable compared to dumb model: uniform for classification, mean for regression.
   - Try Adam, SGD, GD. GD may work best on late stage optimization.
   - Use test set for early stopping

   3.2. Check that error on training set increases with regularization.

   3.3. On small dataset, check that you can overfit by decreasing regularization.

   3.4. Find best learning rate.
   - Lower bound: error does not change much
   - Upper bound: error explodes or NaNs
   - Use rough range to search for optimal

   3.5. Use validation set for overall picture of network performance

Outline

- **(Quick) Introduction to machine learning**
  - {supervised, unsupervised, reinforcement} learning

- **Artificial neural networks**
  - History
  - Hyperparameters
  - Regularization
  - Optimization
  - ML pipeline / best practices

- **Solving PDEs with neural networks**
  - Recall PDEs and solution techniques
  - Basic neural network approach
  - Advance methods (a survey)

- **Working with Tensorflow**

PDEs permeate our world!

They lay at the heart of predictive modeling

$$\frac{\partial \mathbf{u}}{\partial t} = \mathfrak{F}[t, \mathbf{x}, \mathbf{u}], \quad \mathbf{u} = \mathbf{u}(t, \mathbf{x}), \quad t, \mathbf{x} \in [0, T] \times \Omega$$

Physical Law
The *rate of change* of a quantity over time is related to
the local value of that quantity and how it changes in space.

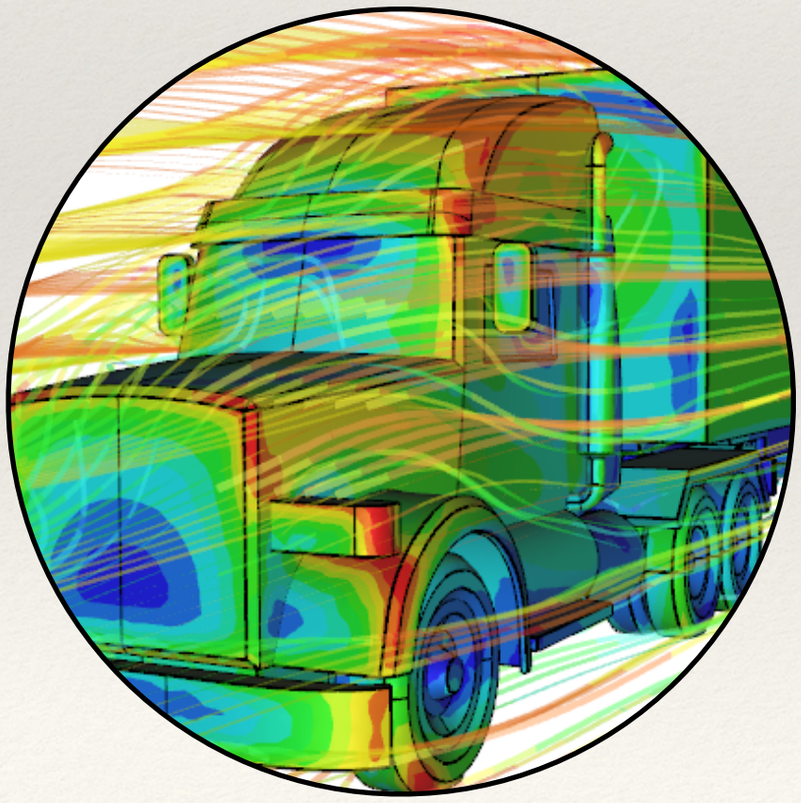
Goal
Solve for the hidden solution, given initial and/or boundary conditions.

PDEs permeate our world!

They lay at the heart of predictive modeling

$$\frac{\partial \mathbf{u}}{\partial t} = \mathfrak{F}[t, \mathbf{x}, \mathbf{u}], \quad \mathbf{u} = \mathbf{u}(t, \mathbf{x}), \quad t, \mathbf{x} \in [0, T] \times \Omega$$

Engineering



Physics

Finance

Ordinary differential equations (ODEs)

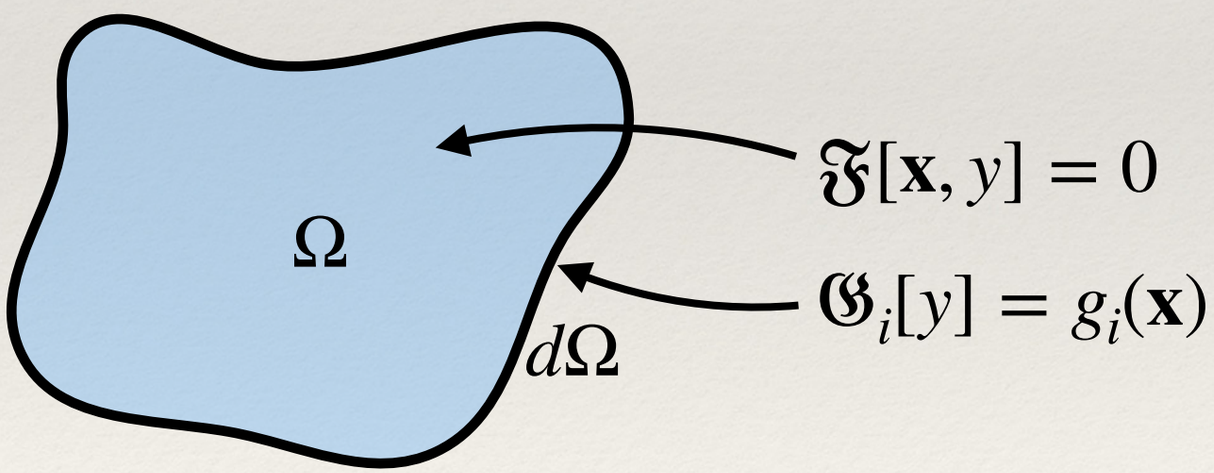
$$\frac{\partial^n y}{\partial t^n} = f\left(t, y, \frac{\partial y}{\partial t}, \ldots, \frac{\partial^{n-1} y}{\partial t^{n-1}}\right), \quad y = y(t), \quad \frac{\partial^k y}{\partial t^k}(t_0) = y_k \; \forall \; 0 \leq k < n$$

- Equation has order n
- Function of a single independent variable t
- Requires $n - 1$ initial conditions
- Linear if f is a linear polynomial of y and its derivatives, called nonlinear otherwise
- Homogeneous when f has no term in t only, non-homogeneous otherwise
- Order n ODE can be converted to a system of first order ODEs by change of variables
- Numerical integration of system of first-order ODEs can done efficiently using explicit or implicit methods, depending on the stiffness of the equations

Partial differential equations (PDEs)

$$\mathfrak{F}[\mathbf{x}, y] = 0, \quad y(\mathbf{x}) : \mathbb{R}^d \mapsto \mathbb{R}, \quad \mathbf{x} \in \Omega$$
$$\mathfrak{G}_i[y] = g_i(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_i, \quad i \in \mathscr{B}$$

- $\mathfrak{F}$ is a differential operator acting on a function y and its partial derivatives of $d > 1$ independent variables $\mathbf{x}$, on the domain Ω.
- Initial/boundary conditions are expressed by the differential operators $\mathfrak{G}_i$ and functions g_i on each boundary of the domain $\partial\Omega_i$.
- Linearity and homogeneity are analogous to ODEs



Classification of PDEs

- **Physical classification**
  - Equilibrium problems have solutions in a closed domain subject to BCs.
    - Steady-state, boundary value problems (BVP)
  - Marching problems have solutions on open domains with ICs and BCs.
    - Transient, initial value problems (IVP), initial boundary value problems (IBVP)

- **Mathematical classification**
  - Characteristic curves (or surfaces) on which "information" can be transmitted
  - Hyperbolic: admit wave-like solutions with a domain of influence
  - Parabolic: admit damped wave solutions depending on all points in the past
  - Elliptic: admit non wave-like solutions depending on all boundary points
  - Character can depend on location in the domain and can be mixed

- Well-posed problems problems have solutions that
  - exist
  - are unique
  - depend continuously on the initial or boundary data

- Not nearly enough time to talk about all this, but important to keep it in mind!

Boundary conditions

- Dirichlet boundary conditions
  - Solution imposed along the boundary
    $y(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega$
- Neumann boundary conditions
  - Solution gradients imposed along the boundary
    $\hat{\mathbf{n}} \cdot \nabla y = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega$
- Robin (Mixed) boundary conditions
  - Linear combination of Dirichlet and Neumann
    $a_1(\mathbf{x})\, y(\mathbf{x}) + a_2(\mathbf{x})\, \hat{\mathbf{n}} \cdot \nabla y = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega$
- Names refer to specific problems associated with the Laplace equation ($\nabla^2 y = 0$)
  - Accepted terminology for any PDE
  - "Dirichlet boundary data is used" means that solution is prescribed on boundary

Some PDEs of interest

Equations	Physical and mathematical classification	
• **Linear wave (advection) equation** $\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0$	Marching wave	Hyperbolic
• **Burgers equation (inviscid when $\nu = 0$)** $\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$	Damped wave	Parabolic (Hyperbolic)
• **Poisson equation (Laplace when $f = 0$)** $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$	Equilibrium	Elliptic
• **Advection-diffusion equation** $\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$	Damped wave	Parabolic
• **Heat equation** $\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2}$	Diffusion	Parabolic

Numerical discretization of PDEs

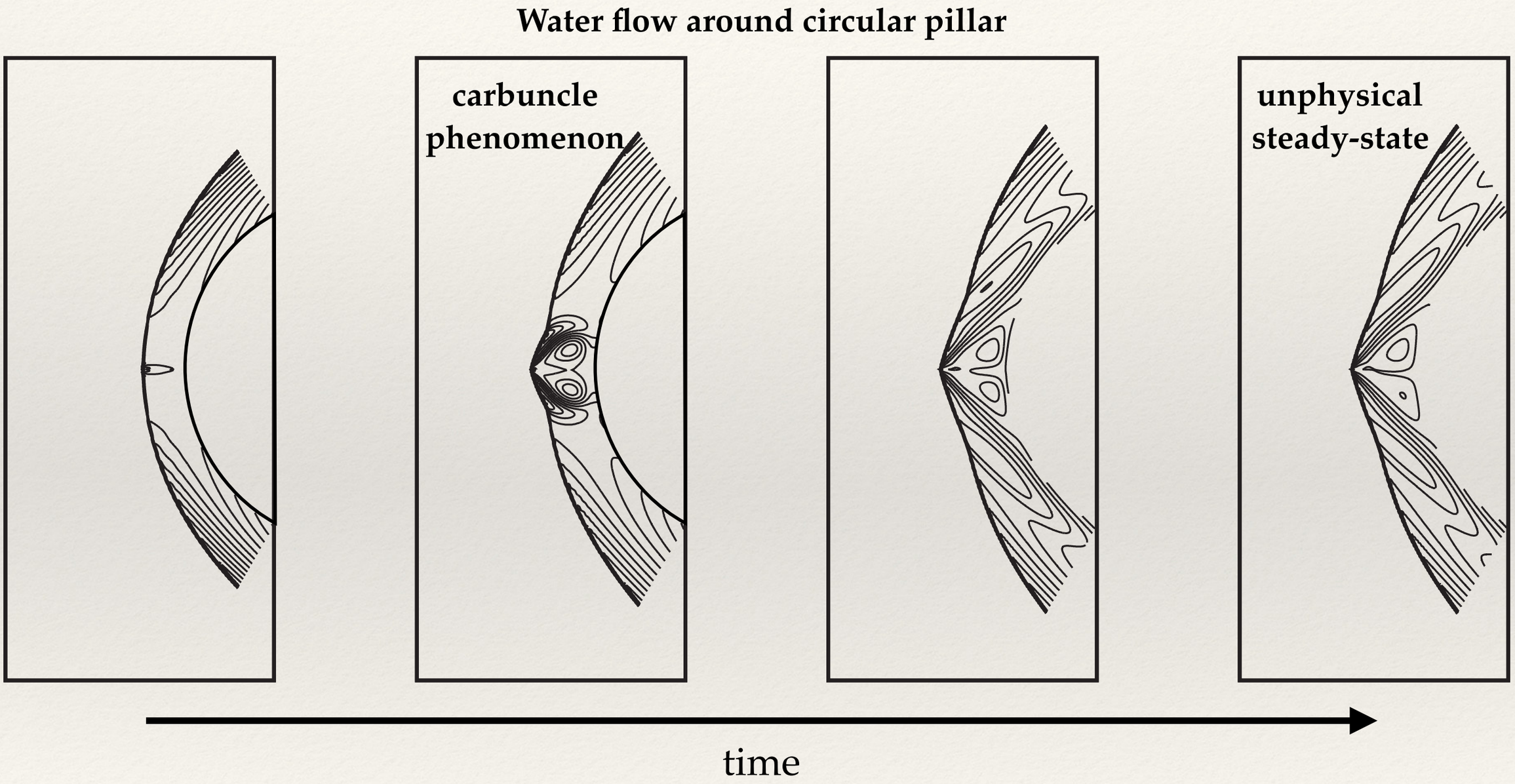
- "Classical" discretization techniques such as finite-{difference, volume, element}
  - Convert a PDE into an algebraic system or system of ODEs
  - Represent solution on discrete cells/points in spatial domain.

- Must construct a mesh taking into account:
  - Alignment with key features of the solution
  - Mesh quality, skewness
  - Mesh size (curse of dimensionality)

- Discretization of the equations requires careful consideration of
  - Consistency
  - Stability
  - Convergence
  - Speed

Mesh alignment and resolution

- Mesh must be adapted to align with critical flow structures to maintain accuracy.

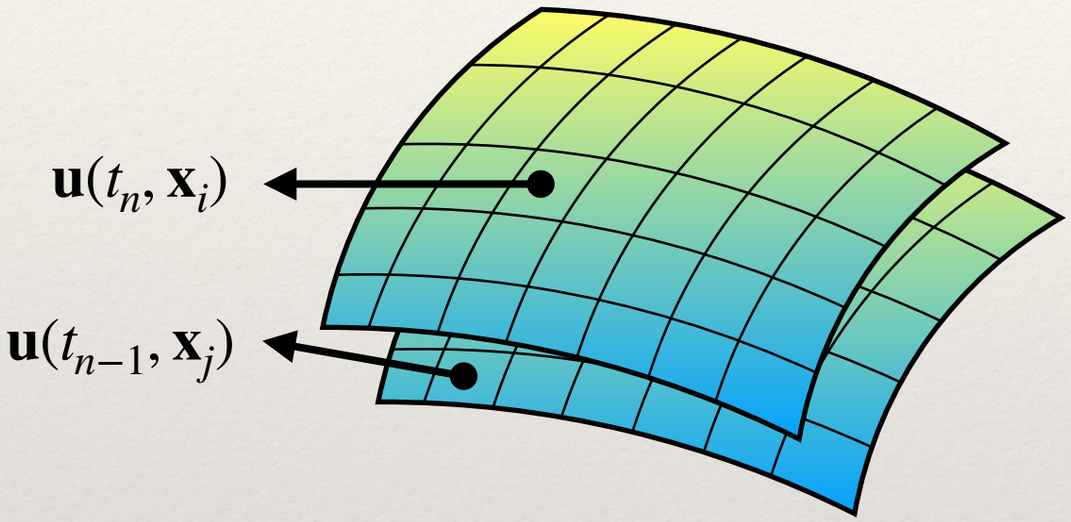


Friedemann Kemm, App. Math. and Comp. 320:596-613, 2018

Can we remove the mesh completely?

- **Idea**: convert PDE into an optimization problem using combination of collocation and machine learning with an appropriate loss function.

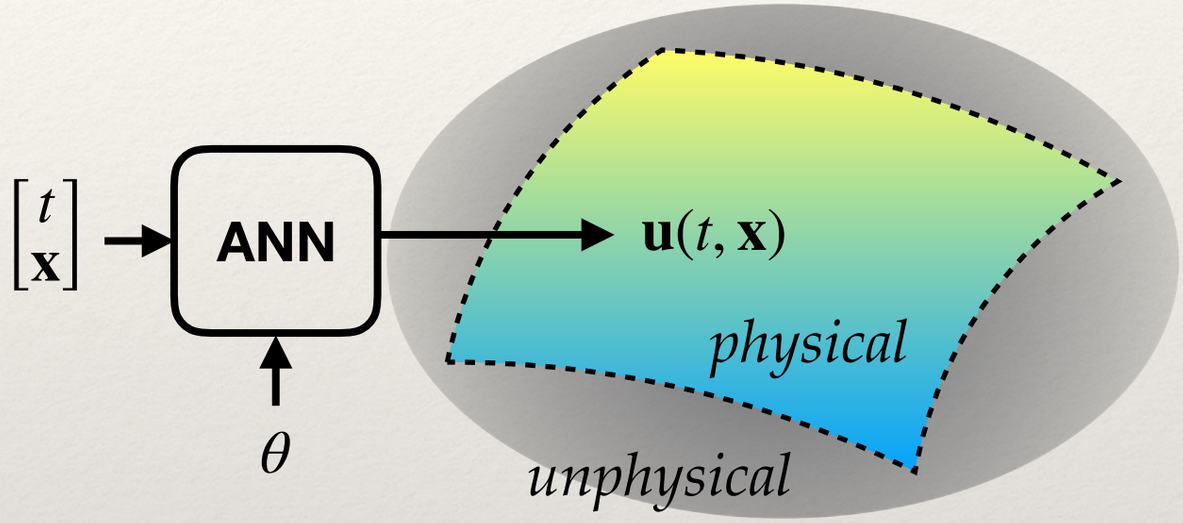
Conventional Discretization Methods



Problem converted to large system of ordinary differential equations

$$\frac{\partial \mathbf{u}_i}{\partial t} = F(\mathbf{u}_1, \ldots, \mathbf{u}_N)$$

Machine Learning Approach



Problem converted to optimization of neural network parameters.

$$\min_{\theta} \sum_{(t,x)_i} \left| \frac{\partial \mathbf{u}(\theta)}{\partial t} - \mathscr{F}[\mathbf{u}(\theta)] \right|$$

General approach

- Recall our general PDE
  $\mathfrak{F}[\mathbf{x}, u] = 0, \quad u(\mathbf{x}) : \mathbb{R}^d \mapsto \mathbb{R}, \quad \mathbf{x} \in \Omega$
  $\mathfrak{G}_i[u] = g_i(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_i$

- Assume the hidden solution u can be represented by a NN
  $u(\mathbf{x}) = \hat{u}(\mathbf{x}; \theta)$,
  where $\hat{u}$ is a NN with parameters θ, taking $\mathbf{x}$ as input and outputs approximation of u

- Construct a loss function on the average residual error in the PDE for a given θ
  $\mathscr{L}(\theta) = \frac{1}{\Delta\Omega} \int_\Omega (\mathfrak{F}[\mathbf{x}, \hat{u}(\mathbf{x}; \theta)])^2 d\mathbf{x}$

- Approximate the loss using collocation by selecting points in the domain
  $\mathscr{L}(\theta) \approx \frac{1}{|\hat{\Omega}|} \sum_{\mathbf{x} \in \hat{\Omega}} (\mathfrak{F}[\mathbf{x}, \hat{u}(\mathbf{x}; \theta)])^2$

- Minimize loss with BC constraints to find optimal θ^* for the network solution
  $\theta^* = \arg\min_\theta \mathscr{L}(\theta)$,
  $\mathfrak{G}_i[\hat{u}(\mathbf{x}; \theta)] - g_i(\mathbf{x}) = 0, \forall x \in \partial\Omega_i$

Advantages of NN solutions to PDEs

- Solution in a closed analytic form which is differentiable
  - Can evaluate solution and its derivatives at any point on the domain, regardless of the points used to train the network
  - The derivatives are exact and do not require further processing
  - No mesh!

- Neural networks have excellent generalization properties
  - Can often extrapolate beyond the training domain with good accuracy

- Relatively few parameters may be necessary to model complex solution, compared to large storage requirements (memory trace) for discretized approaches

- Computational complexity is linear in the number of points sampled
  - Avoids the curse of dimensionality for higher dimensional problems

- Temporal and spatial derivatives may be treated in the same way

- Straightforward to implement using existing deep learning libraries

- Easily parallelizable on CPUs and GPUs

Disadvantages of the approach

- Still not clear how to choose a “good” architecture for a given problem
  - Finding the right network may require some experience and trial-and-error
  - Analogous to building a “good” mesh

- No guarantee of physically allowed solutions during early stages of training
  - Initial solution will be random (though we can enforce some structure)
  - May require special treatment

- Difficult to formulate convergence, stability, uniqueness arguments

- Still not clear how to best find entropic solutions for hyperbolic problems
  - Entropic constraints analogous to flux splitting schemes require special attention

- In general there are not a lot of resources you can turn to for advice or best practices
  - Classical methods have decades of experience, literature, courses based on fundamental research
  - Deep learning remains a new field with lots of fundamental work left to be done

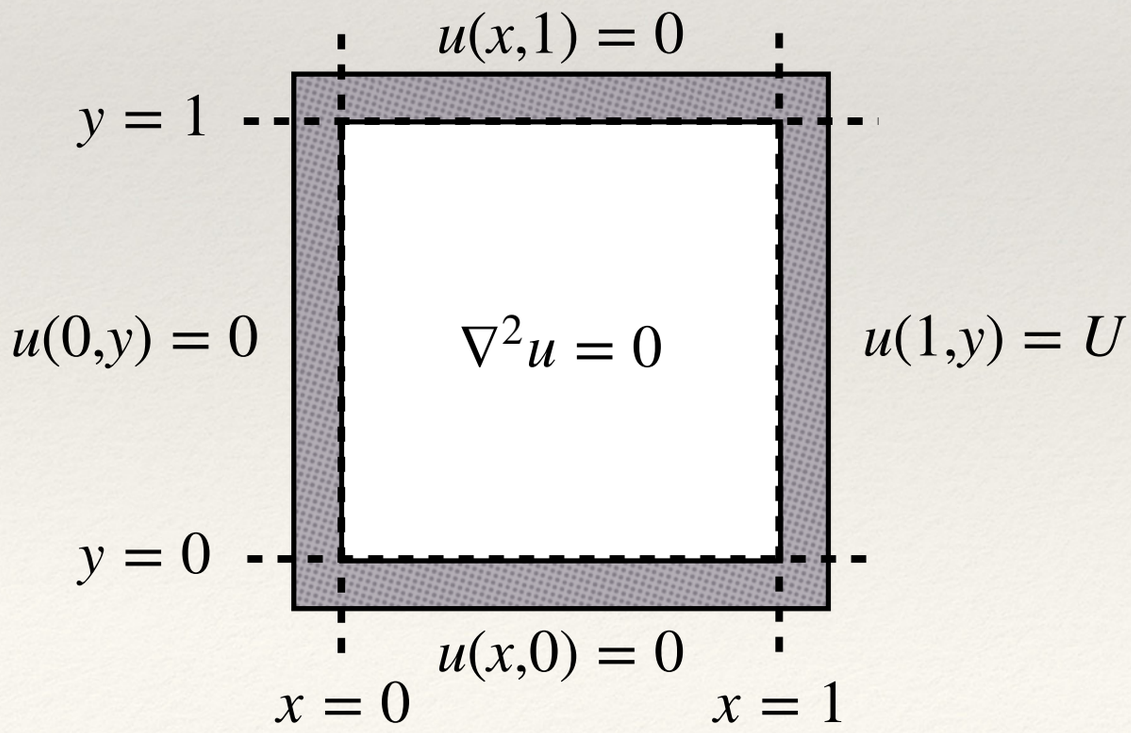
Treatment of boundary conditions

- Constrained optimization problem in very high dimensional space!
- In general, we have two approaches
  - Satisfy BCs by construction of $\hat{u}$ (unconstrained optimization)
  - Solve constrained minimization problem
- Choice depends on
  - Simplicity of the BC formulation
  - Boundary regularity
  - Problem dimension
  - Easier to use construction method on low dimensional problems with simple geometries, high dimensional geometries may be easier with constrained minimization

Unconstrained optimization

- Boundary conditions are satisfied explicitly by construction of solution network
- Consider the Dirichlet condition
  $u(\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega$
- Write the NN solution as
  $\hat{u}(\mathbf{x};\theta) = A(\mathbf{x}) + B(\mathbf{x})\ \tilde{u}(\mathbf{x};\theta)$
  - $A(\mathbf{x})$ satisfies the boundary condition when $\mathbf{x} \in \partial\Omega$
  - $B(\mathbf{x}) = 0$ when $\mathbf{x} \in \partial\Omega$ and $B(\mathbf{x}) \neq 0$ when $\mathbf{x} \notin \partial\Omega$

- **Example**:



$$\hat{u}(x,y;\theta) = A(x,y) + B(x,y)\ \tilde{u}(x,y;\theta)$$

$$A(x,y) = x[y(y-1)]^{1-x}U$$

$$B(x,y) = x(x-1)y(y-1)$$

Unconstrained optimization

- Boundary conditions are satisfied explicitly by construction of solution network
- Consider the Dirichlet condition
  $u(\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega$
- Write the NN solution as
  $\hat{u}(\mathbf{x};\theta) = A(\mathbf{x}) + B(\mathbf{x})\ \tilde{u}(\mathbf{x};\theta)$
  - $A(\mathbf{x})$ satisfies the boundary condition when $\mathbf{x} \in \partial\Omega$
  - $B(\mathbf{x}) = 0$ when $\mathbf{x} \in \partial\Omega$ and $B(\mathbf{x}) \neq 0$ when $\mathbf{x} \notin \partial\Omega$
- In general, we can always write
  $B(\mathbf{x}) = \prod_{i \in \mathcal{B}} d_i(\mathbf{x})$
  where d_i is a distance measure for boundary i.
- $A(\mathbf{x})$ may be tricker for complex boundaries
  - Option: let $A(\mathbf{x})$ be a neural network and train on boundary data! —> regression problem.
  - At least this provides an error estimate for the boundary conditions…

Constrained optimization

- Update the loss function to incorporate boundary constraints and initial conditions
  $\hat{\mathscr{L}}(\theta) = \mathscr{L}(\theta) + \sum_{i \in \mathscr{B}} \frac{\lambda_i}{|\partial \hat{\Omega}_i|} \sum_{x \in \partial \hat{\Omega}_i} \left(\mathfrak{G}_i[\hat{u}(\mathbf{x}; \theta)] - g_i(\mathbf{x}) \right)^2$

- Minimize the new loss function
  $\theta^* = \arg\min_\theta \hat{\mathscr{L}}(\theta)$

- Note the similarity with regularization! —> In fact both are essentially using the Lagrange multiplier method for constrained optimization.

- Requires the use of additional point sets for each boundary.
  - Potentially more complicated book keeping in the code
  - No guarantee about the accuracy of the solution on the boundaries
  - More general framework compared to the construction method

Computing derivatives

- Need to be able to compute the derivatives of the neural network with respect to its inputs (independent variables)
- Backpropagation to the rescue again!

Backprop Equations

- $\delta^{(l)} \equiv \nabla_{\mathbf{z}^{(l)}} \mathscr{L}$
- $\delta^{(L)} = \nabla_{\mathbf{a}} \mathscr{L} \odot \sigma'(\mathbf{z}^{(L)})$
- $\delta^{(l)} = (\delta^{(l+1)} \mathbf{W}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$
- $\nabla_{\mathbf{b}^{(l)}} \mathscr{L} = \delta^{(l)}$
- $\nabla_{\mathbf{W}^{(l)}} \mathscr{L} = (\mathbf{a}^{(l-1)})^T \delta^{(l)}$

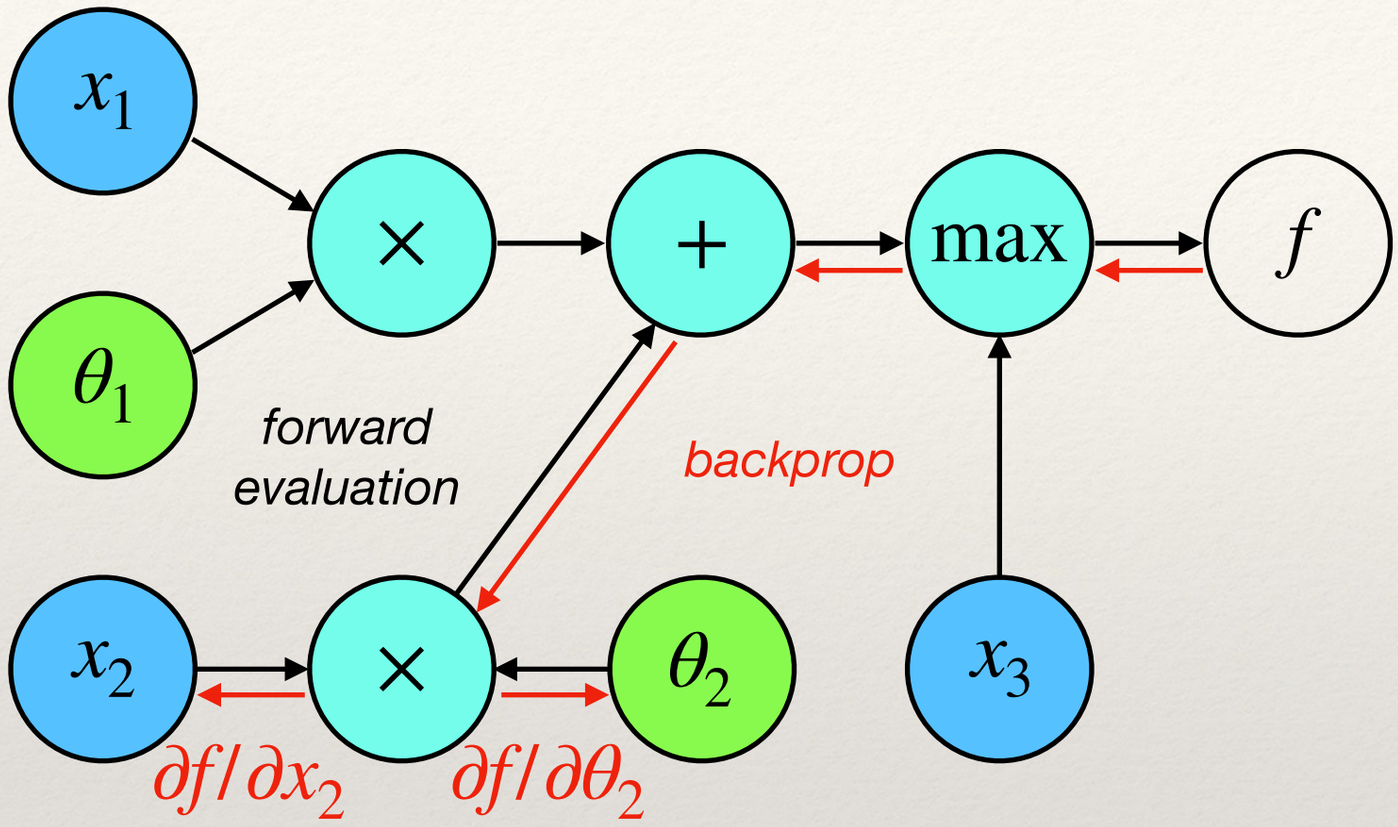
Gradient Equations

- $\xi^{(l)} \equiv \nabla_{\mathbf{z}^{(l)}} \hat{u}$
- $\xi^{(L)} = \sigma'(\mathbf{z}^{(L)})$
- $\xi^{(l)} = (\xi^{(l+1)} \mathbf{W}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$
- $\nabla_{\mathbf{x}} \hat{u} = \frac{\partial \hat{u}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{x}} = \xi^{(1)} \mathbf{W}^{(1)}$

- **Key point**: the same framework which provides gradients of the loss w.r.t. training parameters can also provide solution gradients w.r.t. independent variables.
- Can extend to higher order derivatives as well

Computational graphs

$$f = \max(\theta_1 x_1 + \theta_2 x_2, x_3)$$



Gradient calculation through recursive uses of the chain rule.

- Backpropagation is usually implemented using computational graphs (CGs).
- Modern ML libraries like Tensorflow and Pytorch rely heavily on CGs to represent all computations
- Automatic differentiation makes computing gradients for parameters or inputs easy

Going further

- Discrete time methods
- More advanced network architectures
  - Highway networks
  - Convolutional networks
  - Recurrent networks
- Beyond neural networks
  - Support vector machines
- High dimensional problems
  - Solving SDEs
- Acceleration of conventional methods with ML
  - Approximate matrix inversion
  - ML discretization methods
  - ML moment methods

Outline

- **(Quick) Introduction to machine learning**
  - {supervised, unsupervised, reinforcement} learning

- **Artificial neural networks**
  - History
  - Hyperparameters
  - Regularization
  - Optimization
  - ML pipeline / best practices

- **Solving PDEs with neural networks**
  - Recall PDEs and solution techniques
  - Basic neural network approach
  - Advance methods (a survey)

- **Working with Tensorflow**