香港浸會大學
HONG KONG BAPTIST UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE
計算機科學系

COMP7015 Artificial Intelligence (S1, 2023-24)

Lecture 7: Basics of Deep Learning

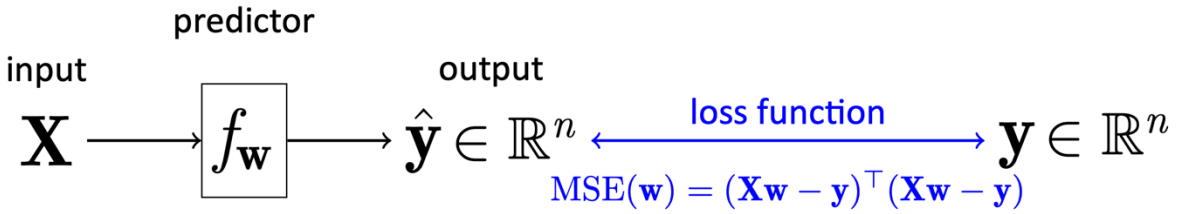
Instructor: Dr. Kejing Yin (cskjyin@hkbu.edu.hk)

Department of Computer Science
Hong Kong Baptist University

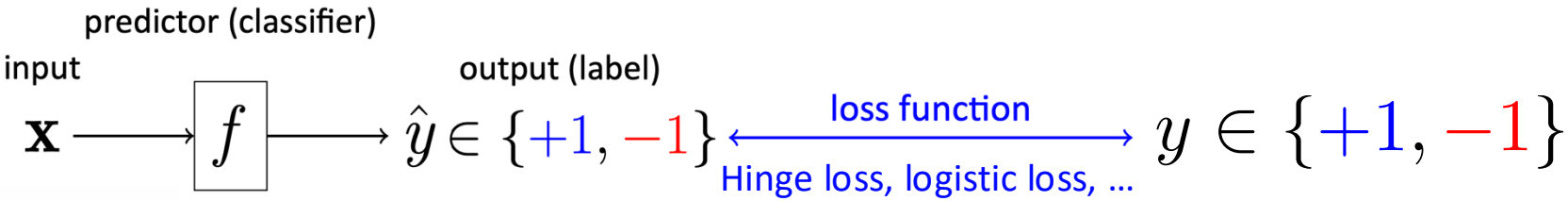
November 1, 2024

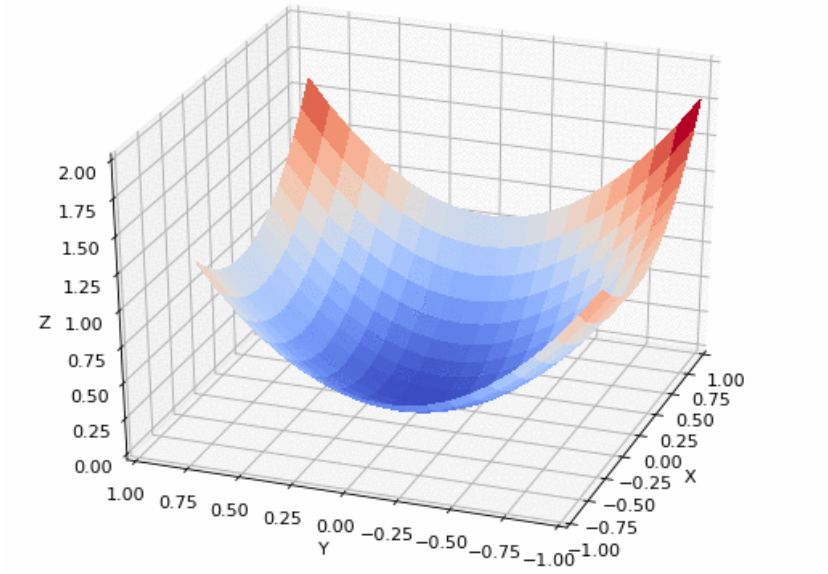
Recap: Loss Minimization Framework for Machine Learning

Regression:



Classification:





Gradient Descent Algorithm for Linear Regression

Initialize $w = 0$ and $b = 0$;

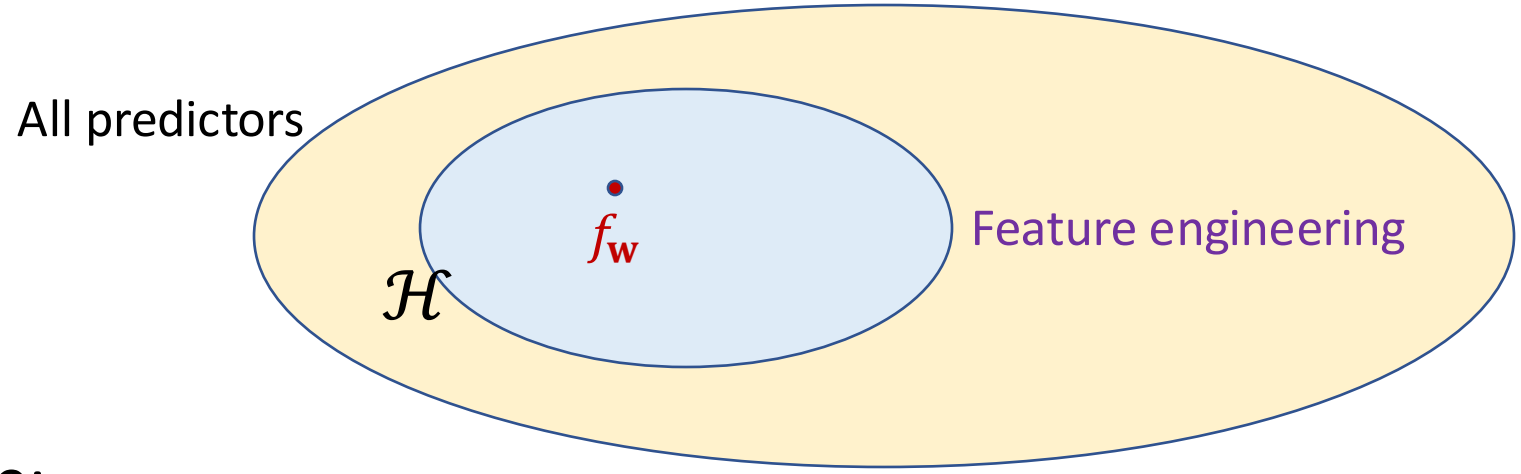
For $t = 1, \ldots, T$, do

$$w \leftarrow w - \eta \nabla_w \mathrm{MSE}(w, b)$$

$$b \leftarrow b - \eta \nabla_b \mathrm{MSE}(w, b)$$

Recap: Feature Engineering and Model Learning

$$\mathcal{H} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w}^{\mathrm{T}}\phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$



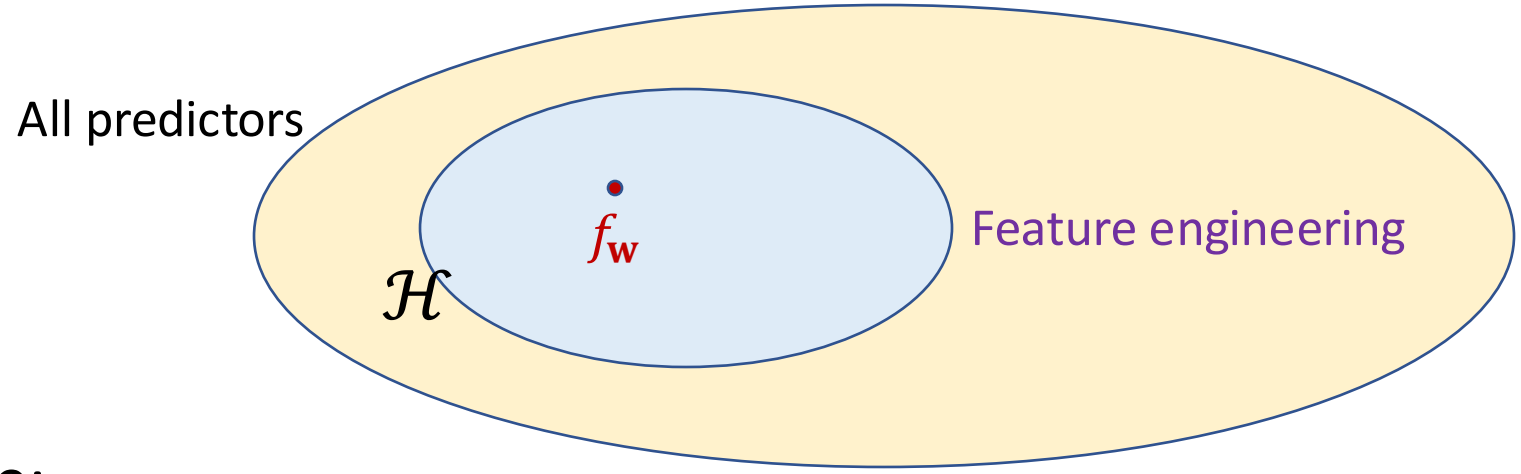
- Example:
  - Linear function: $\phi(x) = x$, the hypothesis class is $\mathcal{H}_1 = \{w_1 x : w_1 \in \mathbb{R}\}$
  - Quadratic function: $\phi(x) = [x, x^2]$, the hypothesis class is

$$\mathcal{H}_2 = \{w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 \in \mathbb{R}\}$$

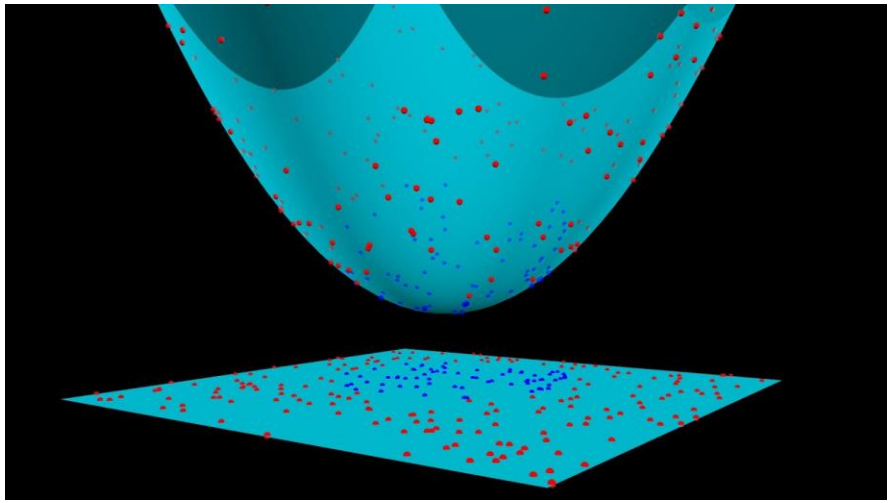
  - We can model nonlinearity via a complex feature mapping.

Recap: Feature Engineering and Model Learning

$$\mathcal{H} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w}^{\mathrm{T}}\phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$



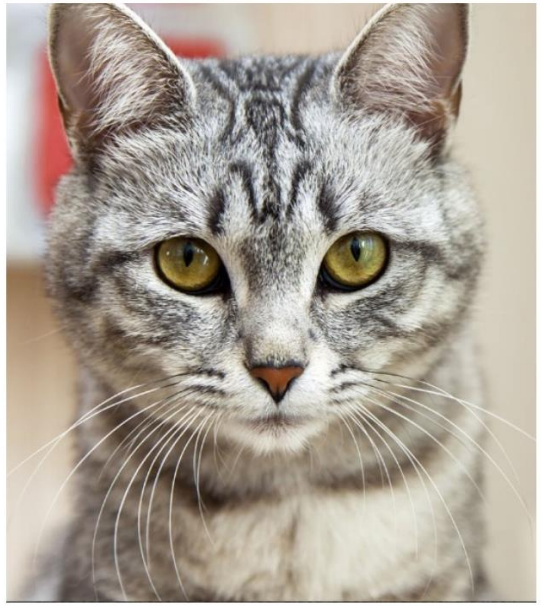
- Example:



Demo: polynomial kernel in SVM

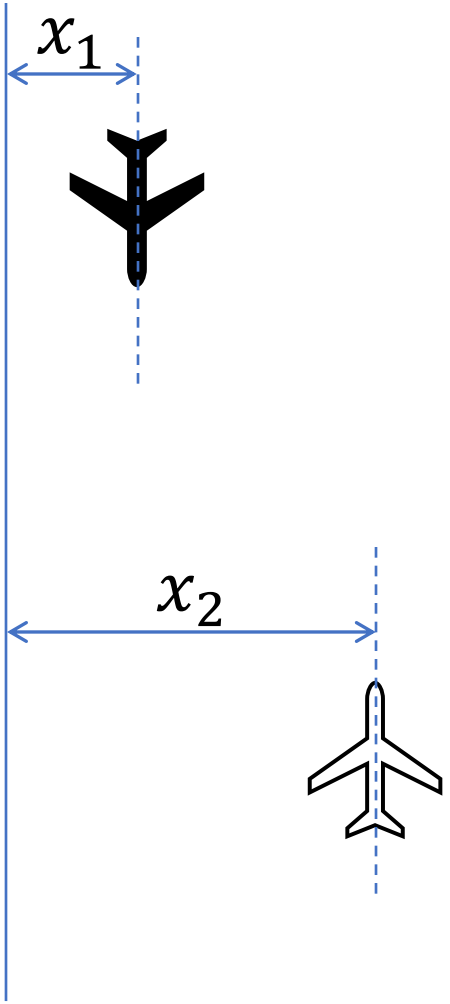
https://www.youtube.com/watch?v=3liCbRZPrZA

Motivations I: how to build a system to detect cats?



- What would be a good feature extractor $\phi(x)$ for this task?
- Can we automatically learn a feature extractor?

Motivations II: aircraft collision prediction



- Input: position of two aircrafts $x = [x_1, x_2]$
- Output: safe ($y = +1$) or collision ($y = -1$)

Suppose we know the true function:

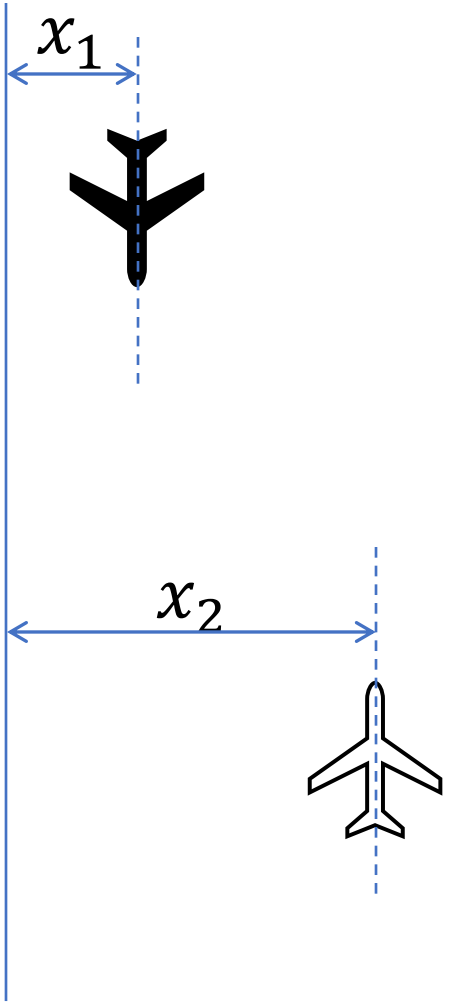
Safe if they are far enough: $y = \text{sign}(|x_1 - x_2| - 1)$

Examples:

$$x = [1, 3] \quad y = +1$$
$$x = [3, 1] \quad y = +1$$
$$x = [1, 1.5] \quad y = -1$$

Motivation II section adapted from https://stanford-cs221.github.io/spring2022-extra/modules/machine-learning/nonlinear-and-neural.pdf

Motivations II: aircraft collision prediction



Decomposing the problem into subproblems:

- Test if aircraft 2 is far right of aircraft 1:

$$h_1 = 1[x_2 - x_1 \geq 1]$$

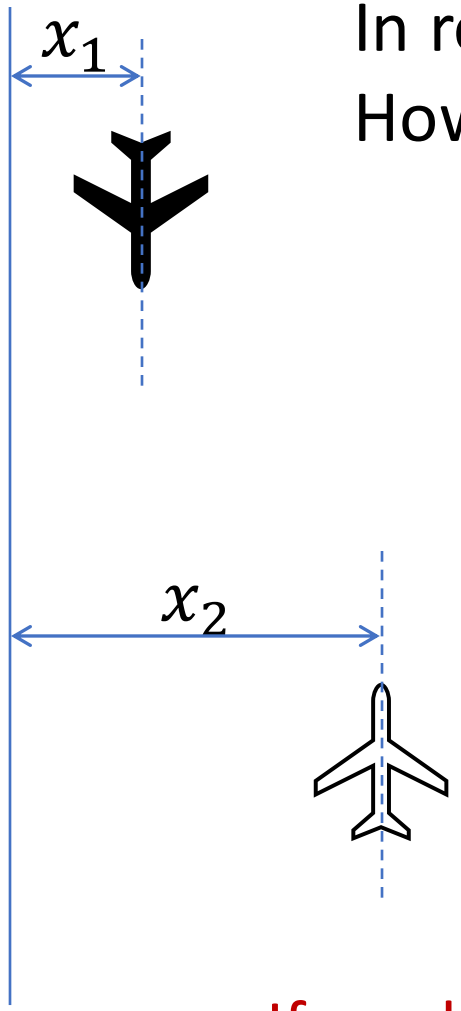
- Test if aircraft 1 is far right of aircraft 2:

$$h_2 = 1[x_1 - x_2 \geq 1]$$

- Safe if at least one is true.

$$y = \text{sign}(h_1 + h_2)$$

Motivations II: aircraft collision prediction



In reality, we do not know the true functions.
How do we learn them from data?

- Define $\phi(x) = [1, x_1, x_2]$
- Test if aircraft 2 is far right of aircraft 1:

  ~~$h_1 = 1[x_2 - x_1 \geq 1]$~~ → $h_1 = 1[\mathbf{v}_1^T \phi(x) \geq 0]$

  If $\mathbf{v_1} = [-1, +1, -1]$, they are equivalent.
- Test if aircraft 1 is far right of aircraft 2:

  ~~$h_2 = 1[x_1 - x_2 \geq 1]$~~ → $h_2 = 1[\mathbf{v}_2^T \phi(x) \geq 0]$

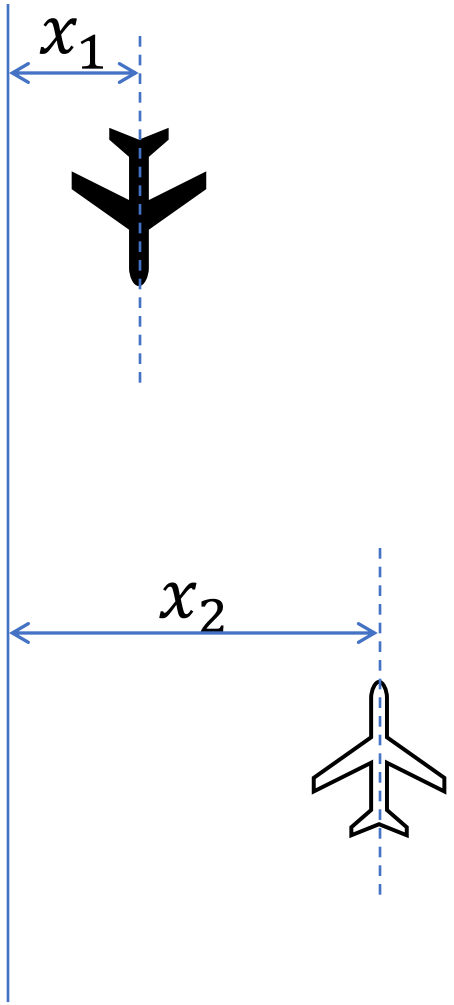
  If $\mathbf{v_2} = [-1, -1, +1]$, they are equivalent.
- Safe if at least one is true.

  ~~$y = \text{sign}(h_1 + h_2)$~~ → $f_{\mathbf{V},\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2)$

  If $\mathbf{w} = [1, 1]$, they are equivalent.

If we do not know the true functions, learn both $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2]$ and $\mathbf{w}$.

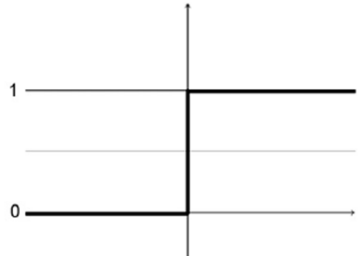
Motivations II: aircraft collision prediction



Learning via gradient descent

- $h_1 = 1[\mathbf{v}_1^T \phi(x) \geq 0]$ gradient with respect to $\mathbf{v}_1$ is 0

Solution: replace the indicator function with an approximation with non-zero gradient.



Indicator function (unit-step)

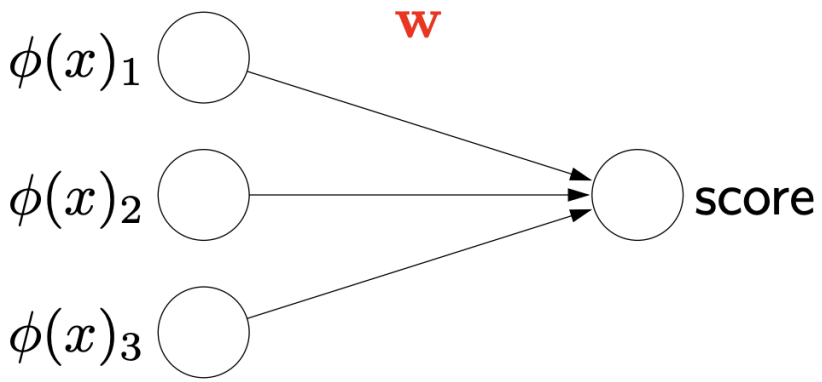
Sigmoid function

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

- $h_1 = \sigma(\mathbf{v}_1^T \phi(x))$

From Linear Functions to Neural Networks

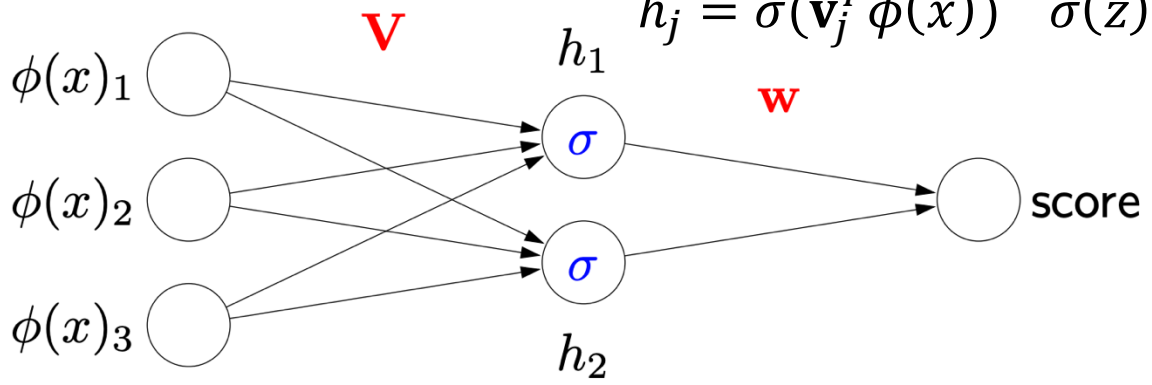
Linear functions:



$$\text{score} = \mathbf{w}_1^T \phi(x)$$

Neural network:
(with 1 hidden layer)

$$h_j = \sigma(\mathbf{v}_j^T \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$



$$\text{score} = \mathbf{w}^T \mathbf{h}$$

Intuition: neural networks try to break down the problem into a set of subproblems

Neural Networks: Feature (Representation) Learning

Interpret $h(x)$ as a learned feature representation

Key idea: feature learning

Before: apply linear predictor on manually specify features

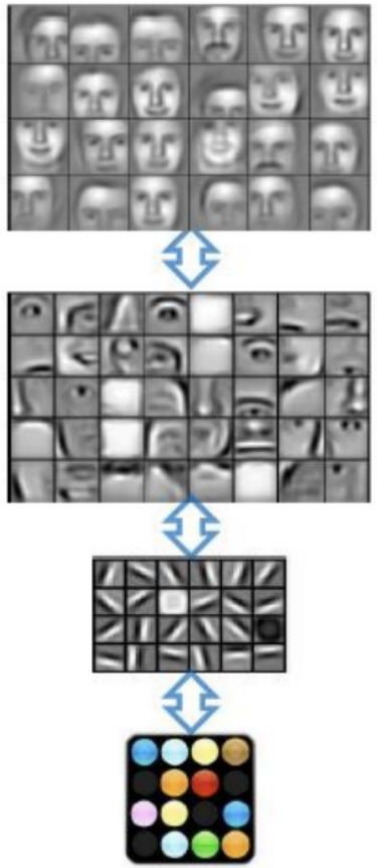
$$\phi(x)$$

Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

Neural Networks: Feature (Representation) Learning

multiple levels of abstractions



3rd layer
"Objects"

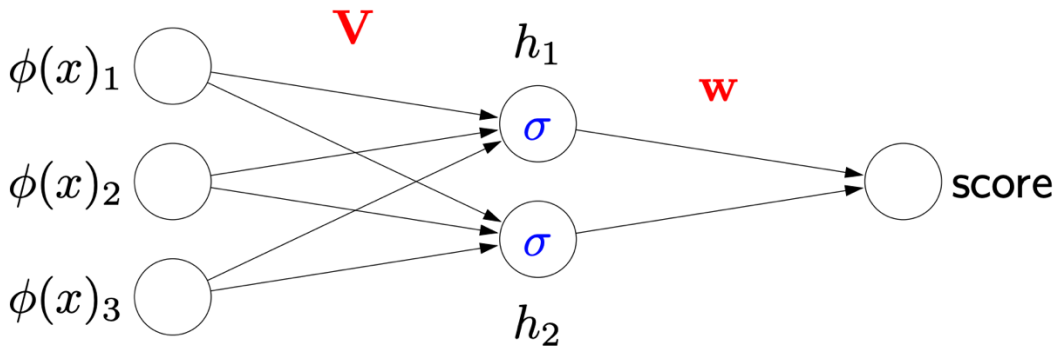
2nd layer
"Object parts"

1st layer
"Edges"

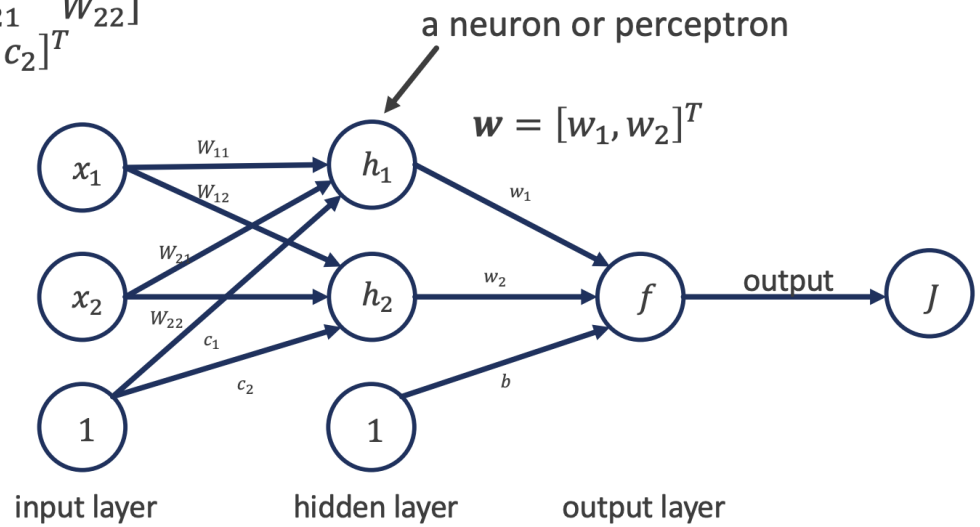
Pixels

Feedforward Neural Networks

- Feedforward: information only travels forward in the network (no loops)
- Neuron: nodes through which data and computations flow.
- Layer: a collection of neurons. (hidden size: number of neurons in hidden layers)
- Activation function: introduces nonlinearity.



$$\boldsymbol{W} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}$$
$$\boldsymbol{c} = [c_1, c_2]^T$$

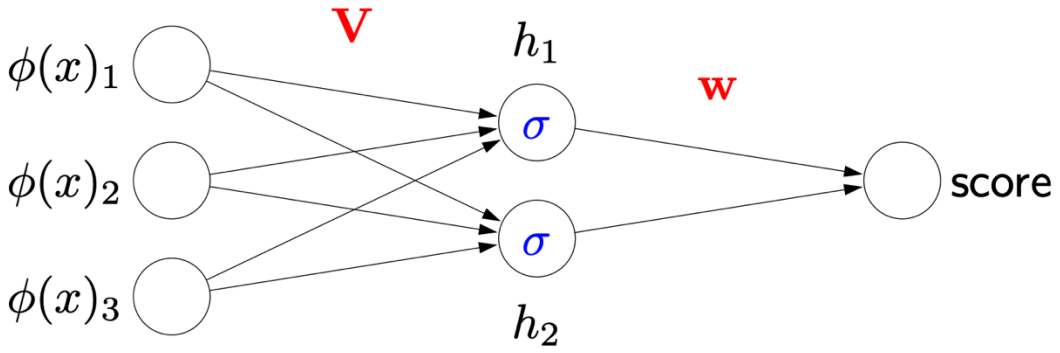


A multi-layer perceptron (MLP)

Feedforward Neural Networks

- What if we remove the activation function?

$$\sigma(z) = z\ ?$$



$$f(\mathbf{x}) = ?$$

$$\begin{aligned} f(\mathbf{x}) &= w^T\left(\mathbf{V}^T\phi(x)\right) \\ &= w_1 h_1 + w_2 h_2 + b \\ &= w_1(v_{11}\phi(x)_1 + v_{21}\phi(x)_2 + v_{31}\phi(x)_3) \\ &\quad + w_2(v_{12}\phi(x)_1 + v_{22}\phi(x)_2 + v_{32}\phi(x)_3) + b \\ &= (\ldots)\phi(x)_1 + (\ldots)\phi(x)_2 + (\ldots)\phi(x)_3 + b \end{aligned}$$

Equivalent to a linear model!

online demos: https://playground.tensorflow.org/

Role of activation functions: introduce nonlinearity (nonlinear in input $\phi(x)$)

Note: the input to the neural network is $\phi(x)$, not x.

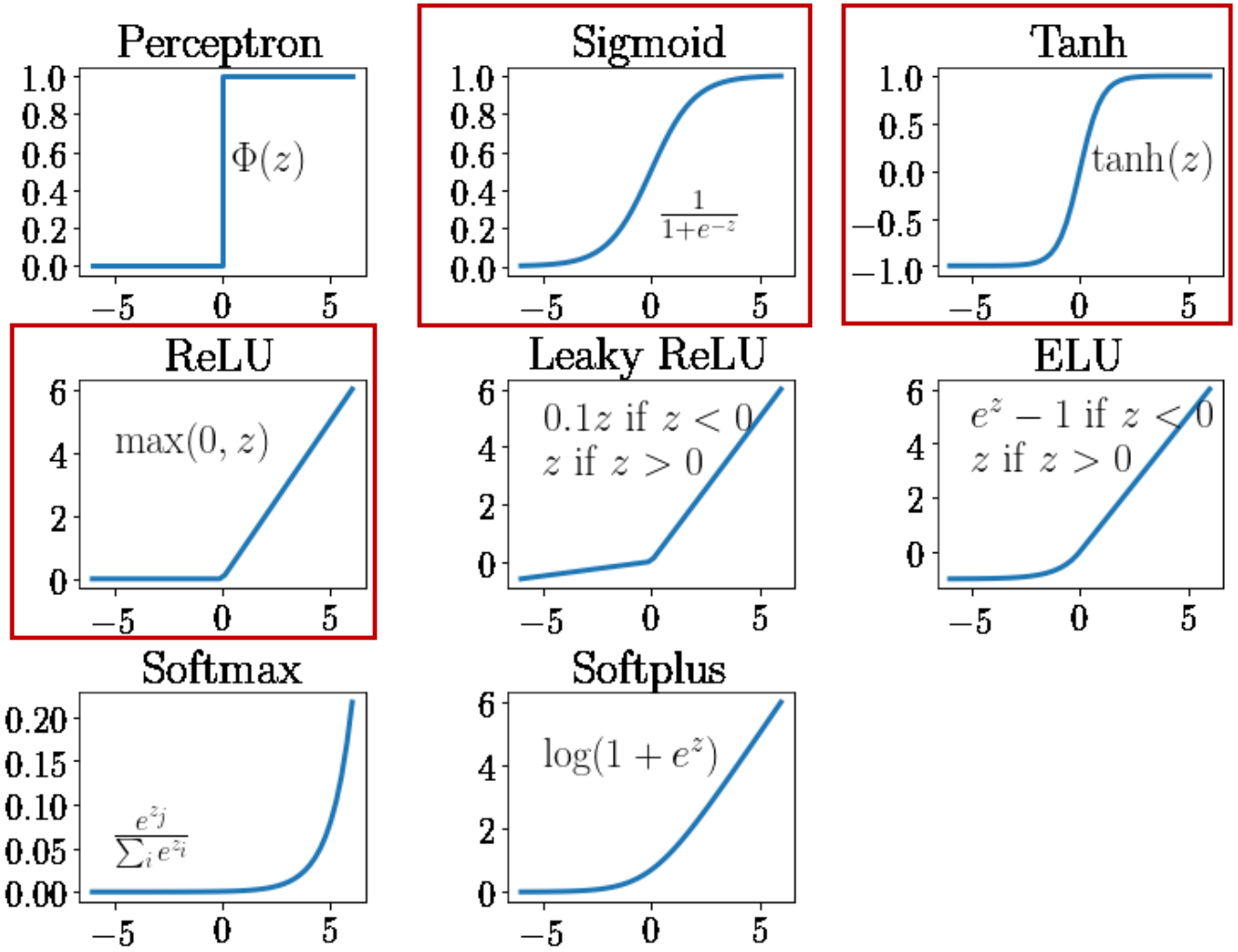
Feedforward Neural Networks

- Commonly used activation functions
  - Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Rectified Linear Unit (ReLU):

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

ReLU is recommended for most FNNs

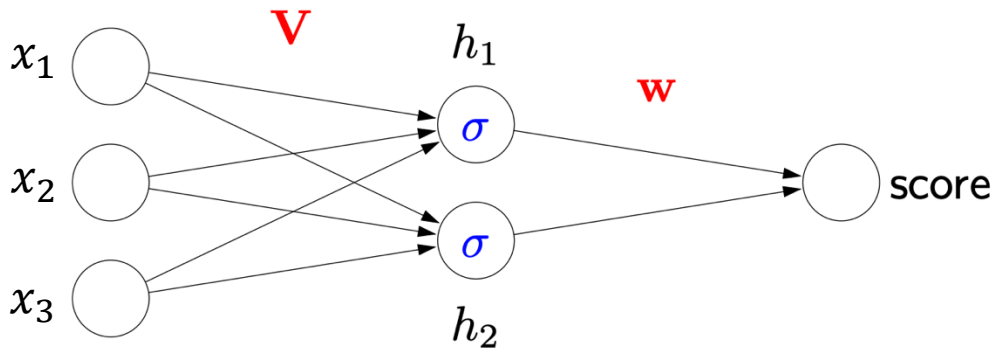
  - Tanh: $\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



Feedforward Neural Networks

- What if we we use ReLU activation function?

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



$$\begin{aligned} f(\mathbf{x}) &= w^T(\mathbf{V}^T\mathbf{x}) \\ &= w_1h_1 + w_2h_2 + b \\ &= w_1(v_{11}x_1 + v_{12}x_2 + v_{13}x_3) \\ &\quad + w_2(v_{21}x_1 + v_{22}x_2 + v_{23}x_3) + b \\ &= (\dots)x_1 + (\dots)x_2 + (\dots)x_3 + b \end{aligned}$$

No activation function: Linear in $\mathbf{x}$

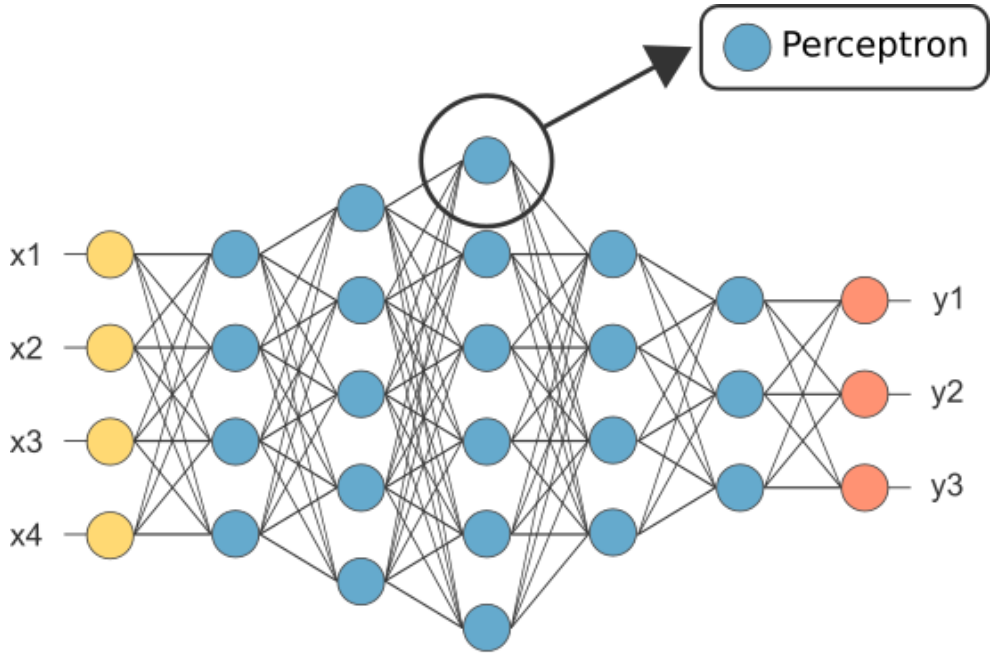
$$f(\mathbf{x}) = w^T \max\{0, \mathbf{V}^T\mathbf{x}\}$$

No longer linear in $\mathbf{x}$

Role of activation functions: introduce nonlinearity (nonlinear in input $\mathbf{x}$)

Summary of Feedforward Neural Network

- Feedforward: Information only travels forward in the network
- Use fully connected layers
- Also known as Multilayer Perceptron (MLP)



Training Neural Network Models

- Consider a regression task
- What loss function should we use?

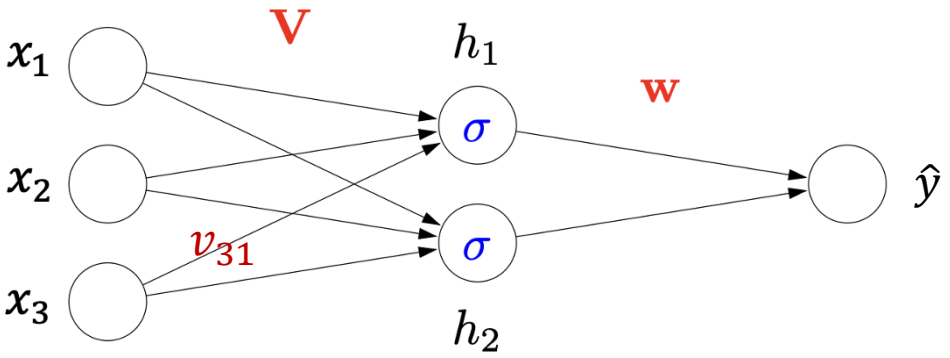
$$\ell = \frac{1}{N}\sum_{n=1}^{N}(y-\hat{y})^2$$

- What are the parameters to be learned?

$$\mathbf{V}, \mathbf{w}$$

- How can we learn them?

Gradient descent



What is the gradient with respect to v_{31}?

$$\nabla_{v_{31}} = \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial h_1}\frac{\partial h_1}{\partial v_{31}} \quad \textit{(chain rule)}$$

$$= \frac{1}{N}\sum_{n=1}^{N} -2(y-\hat{y})\, w_1 \frac{\partial h_1}{\partial v_{31}}$$

$$\frac{\partial h_1}{\partial v_{31}} = \frac{\partial \sigma(\mathbf{v}_1^T\mathbf{x})}{\partial v_{31}} = \frac{\partial \sigma(\mathbf{v}_1^T\mathbf{x})}{\partial(\mathbf{v}_1^T\mathbf{x})}\frac{\partial(\mathbf{v}_1^T\mathbf{x})}{\partial v_{31}} = 1[\mathbf{v}_1^T\mathbf{x} > 0]x_3$$

Training Neural Network Models

- Consider a regression task
- What loss function should we use?

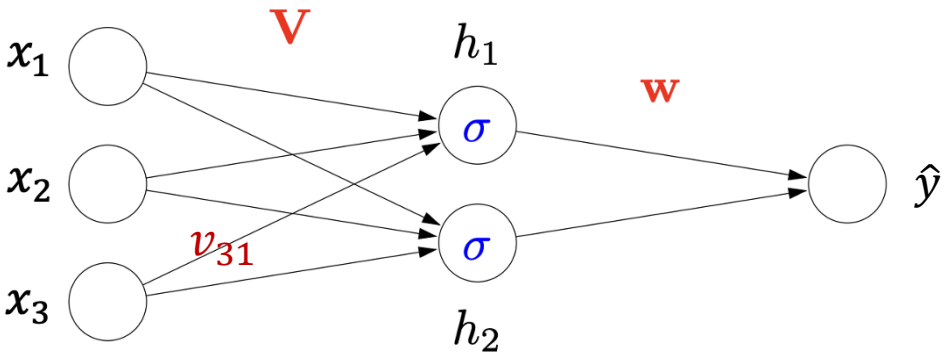
$$\ell = \frac{1}{N}\sum_{n=1}^{N}(y - \hat{y})^2$$

- What are the parameters to be learned?

$$\mathbf{V}, \mathbf{w}$$

- How can we learn them?

Gradient descent



What is the gradient with respect to v_{31}?

What is the gradient with respect to v_{32}?

What is the gradient with respect to v_{21}?

……

Would you like to derive them all?

What if we have 30 layers?

Computational Graphs and Backpropagation

- Mathematically derive gradients w.r.t. all parameters

  Doable by just grind through the chain rules.

- Automate this process?

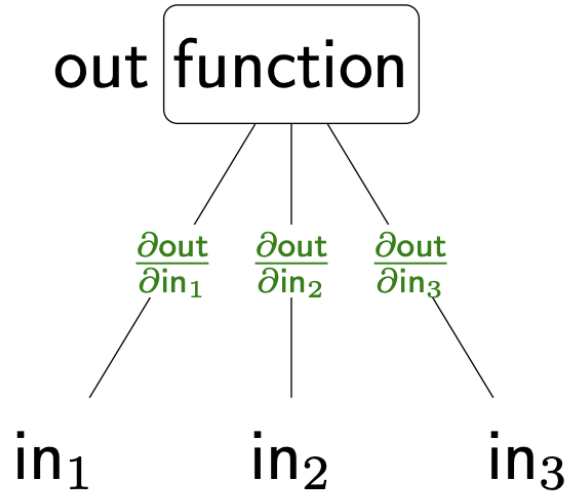
  visualize the computation using a computation graph

- Advantages:
  - Avoid long equations
  - Reveal structure of computations (modularity, efficiency, dependencies)
    — TensorFlow/PyTorch are built on this

This section is partially adapted from https://stanford-cs221.github.io/spring2022-extra/modules/machine-learning/nonlinear-and-neural.pdf

Computational Graphs and Backpropagation

- Think of functions as boxes



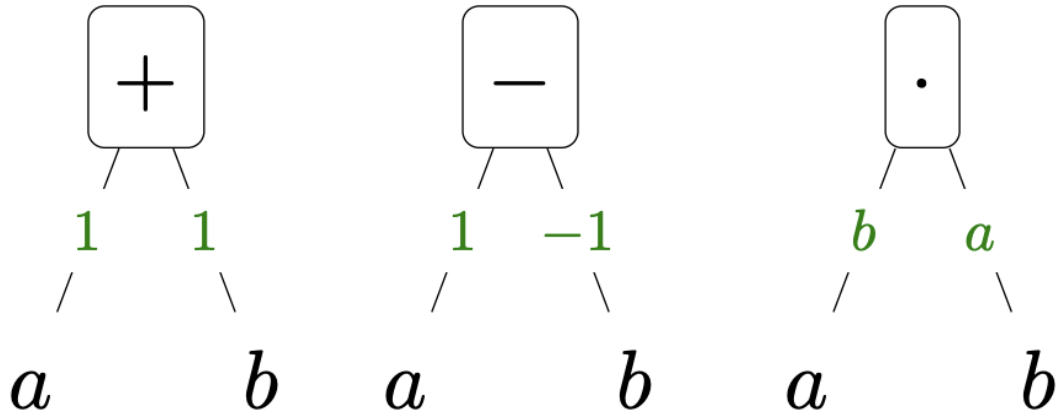
- Partial derivative (gradients):
  how much does the output change if an input slightly changes?

- Example:

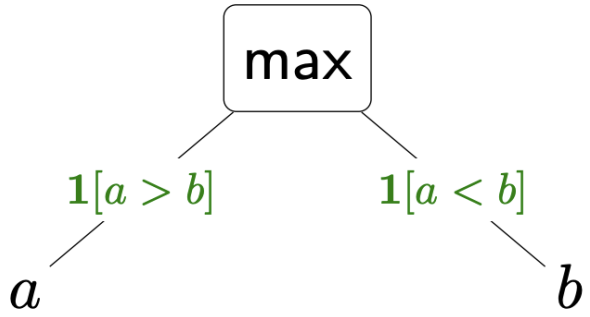
	$2\text{in}_1 + \text{in}_2\text{in}_3 = \text{out}$
(change in_1 slightly)	$2(\text{in}_1 + \epsilon) + \text{in}_2\text{in}_3 = \text{out} + 2\epsilon$
(change in_2 slightly)	$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \text{in}_3\epsilon$

Computational Graphs and Backpropagation

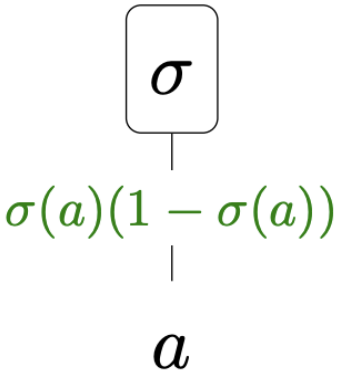
- Building blocks (Five examples)



$a + b$ $a - b$ $a \cdot b$



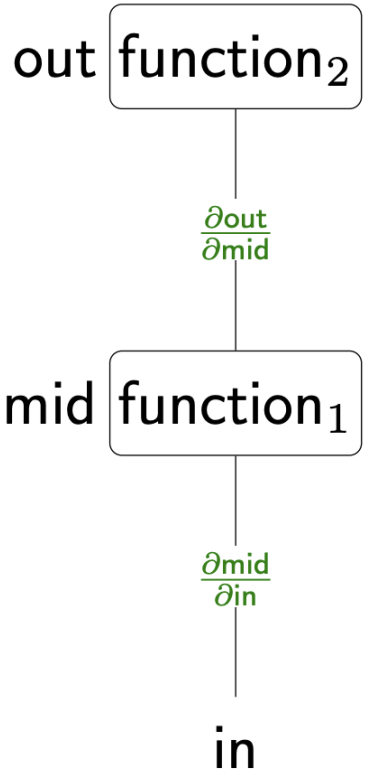
$\max\{a, b\}$



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

Computational Graphs and Backpropagation

- Composing functions



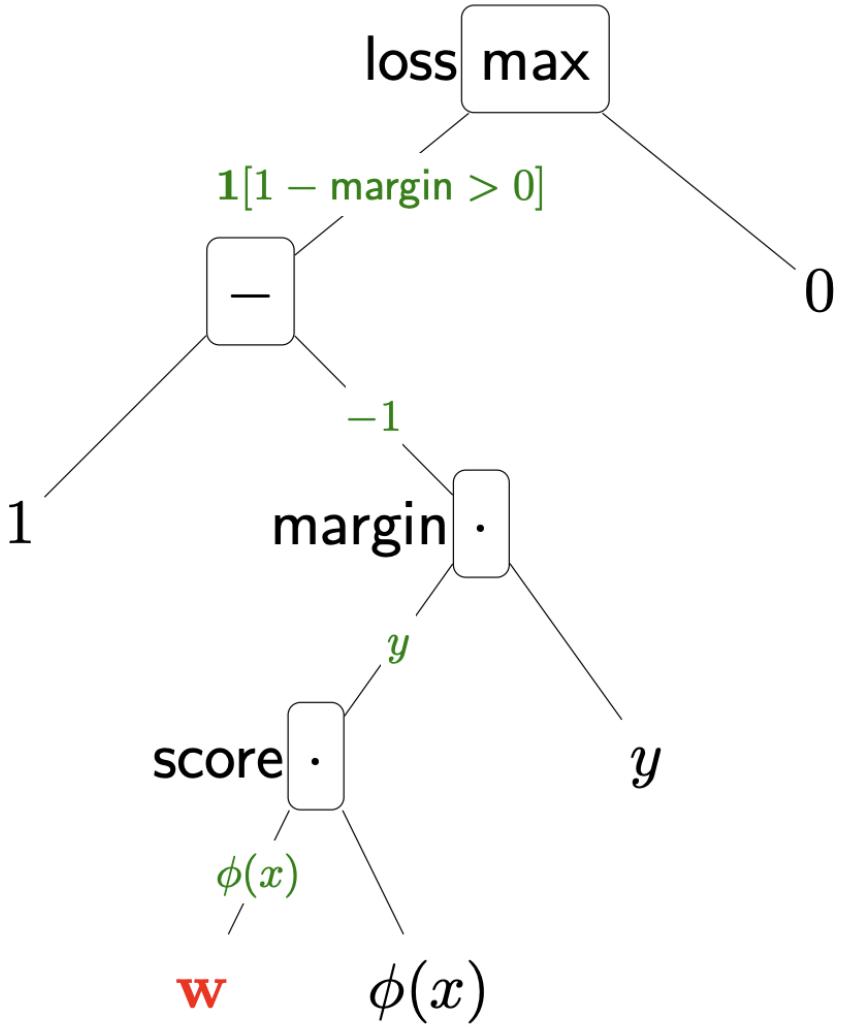
Chain rule: $\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$

Computational Graphs and Backpropagation

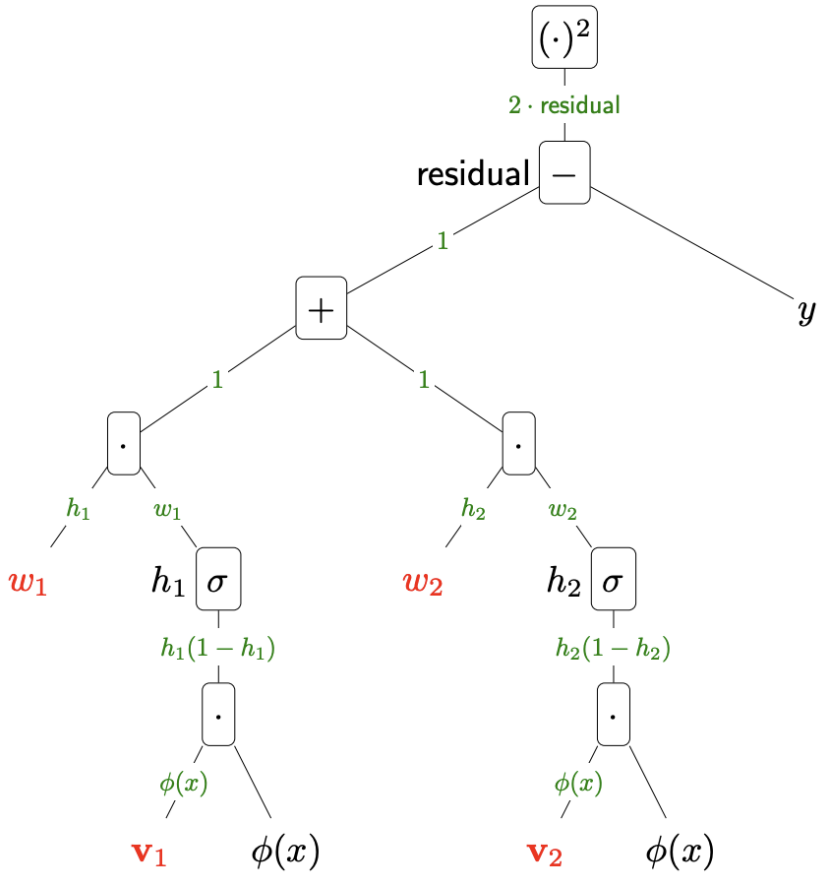
- Example of constructing a computational graph
  - Hinge loss: $\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$
  - Compute:

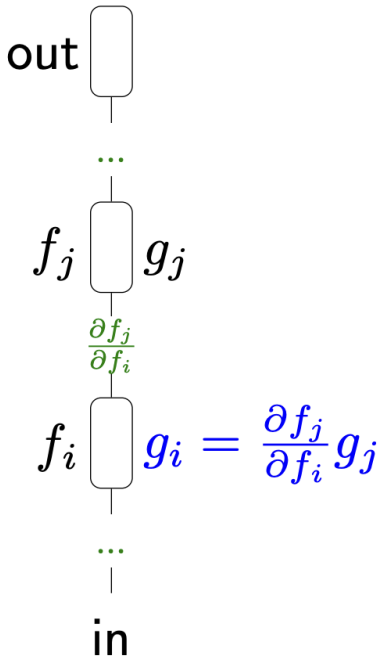
$$\frac{\partial \text{Loss}(x, y, \mathbf{w})}{\partial \mathbf{w}}$$

  - Gradient:
    multiplying the edges: $-\mathbf{1}[\text{margin} < 1]\phi(x)y$



Computational Graphs and Backpropagation





Backpropagation

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{out}}{\partial f_i}$ is how f_i influences output

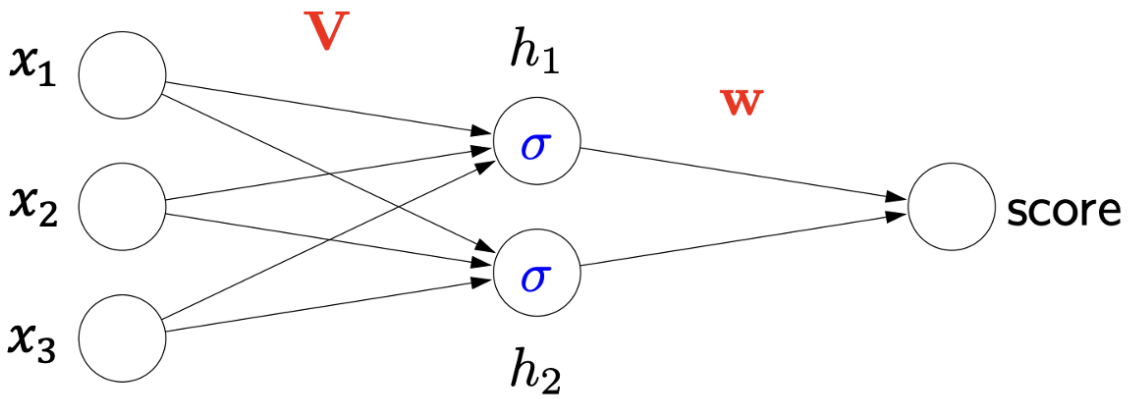
Algorithm: backpropagation

Forward pass: compute each f_i (from leaves to root)

Backward pass: compute each g_i (from root to leaves)

n 2019 / Liang & Sadigh

Computational Graphs and Backpropagation

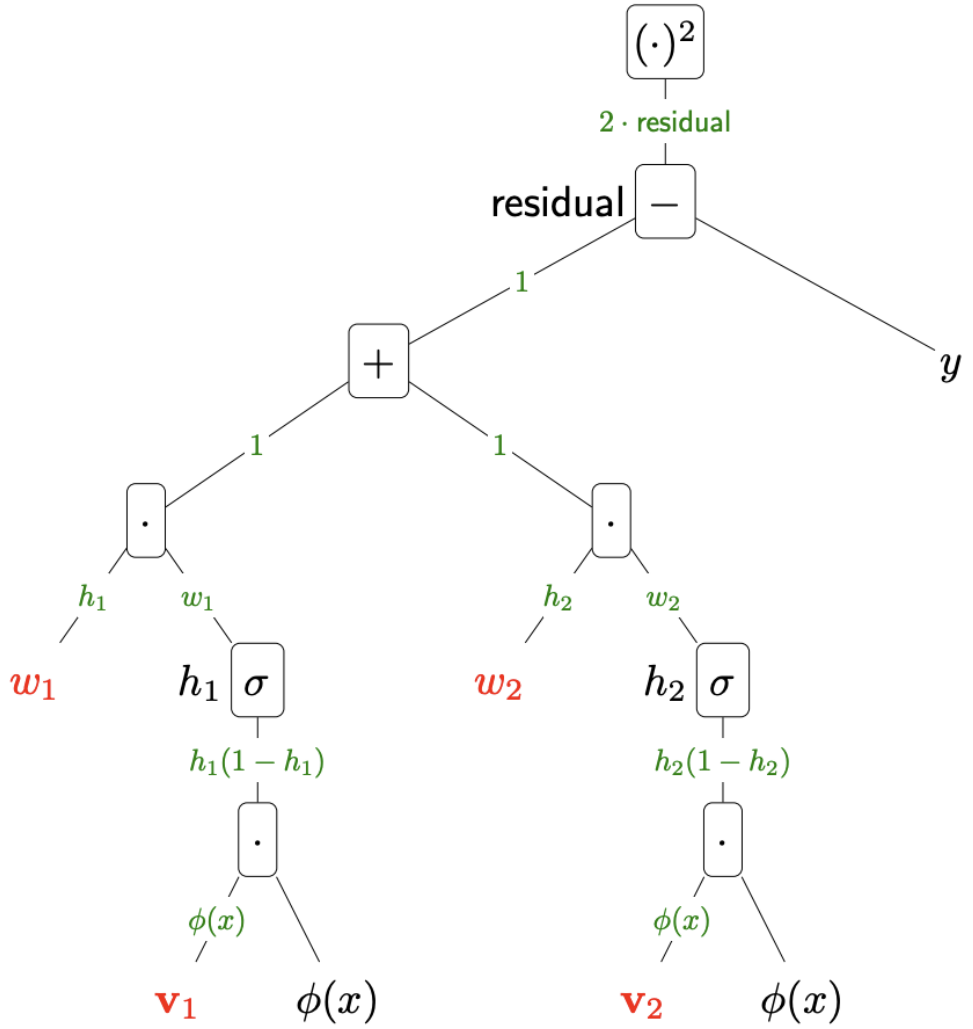


- Exercise:

$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

Construct a computational graph for this neural network

Computational Graphs and Backpropagation



$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

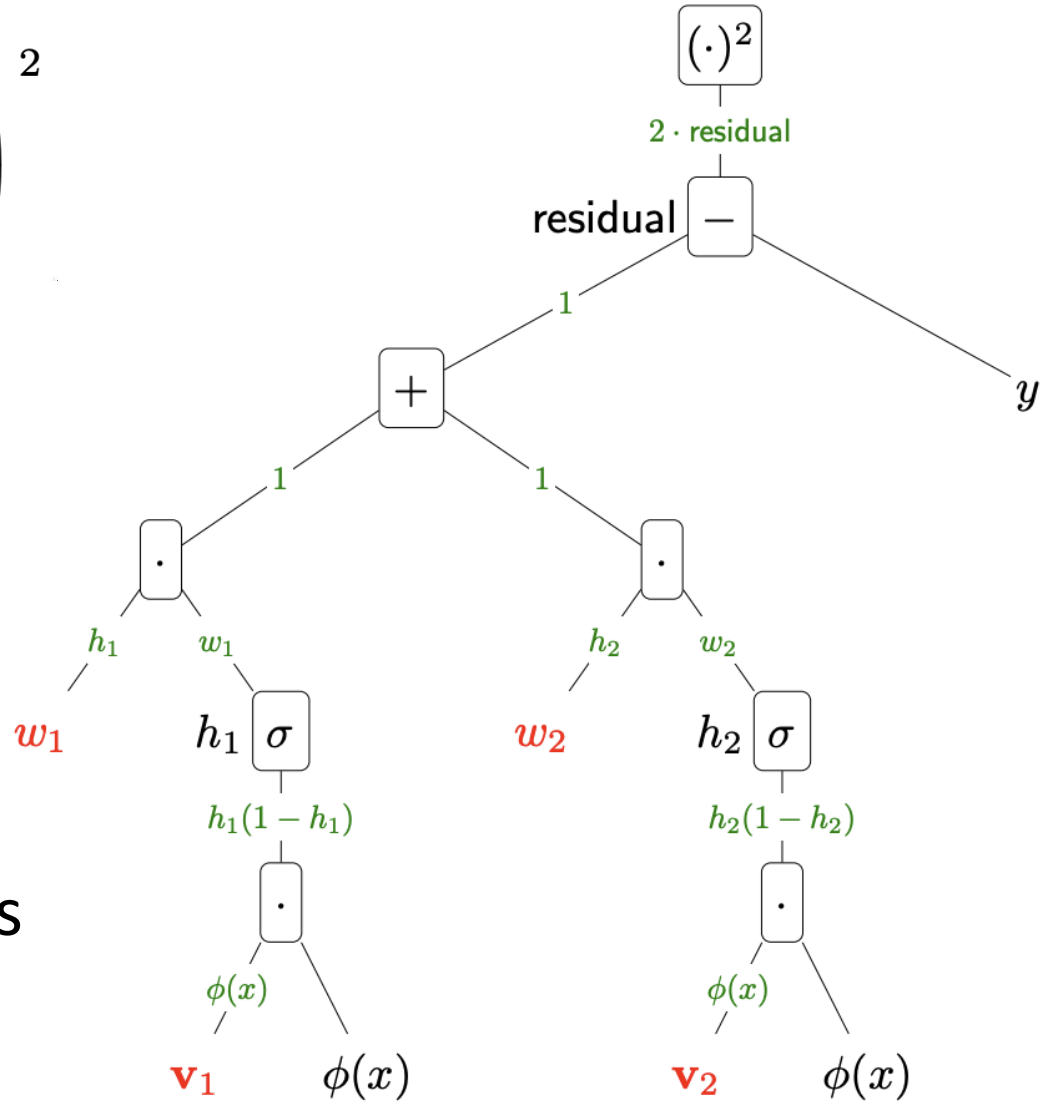
$$\frac{\partial \text{Loss}(x, y, \mathbf{w})}{\partial \mathbf{v}_1} = ?$$

$$2 \cdot \text{residual} \cdot 1 \cdot 1 \cdot w_1 \cdot h_1(1 - h_1) \cdot \phi(x)$$

Computational Graphs and Backpropagation

- Exercise: $\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$

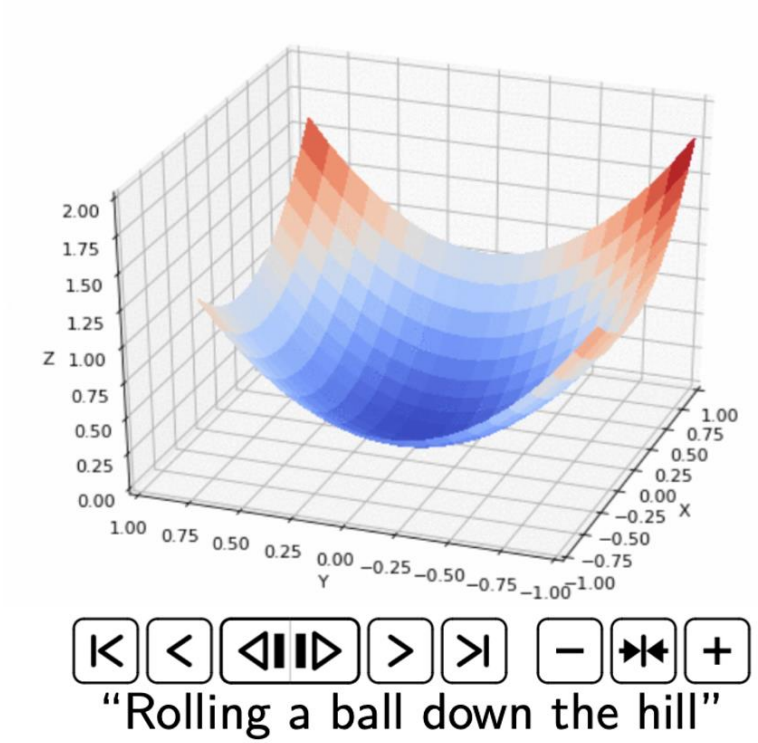
  - Given a data point:
    $\phi(x) = [1.5, -0.7, 0.6], \quad y = 1.8$
  - Model parameters:
    $\mathbf{v}_1 = [1.1, -1, -0.5], \ \mathbf{v}_2 = [2, 1, 1.2]$
    $\mathbf{w} = [2, 0.6]$
  - Compute the loss function in forward pass and gradients in the backward pass



Stochastic Gradient Descent

Recall the gradient descent algorithm

- **Graident**: the gradient $\nabla_w f(w)$ is the direction of the greatest increase of $f(w)$.
- Start from an initial location, move a bit along the opposite direction of the gradient.



"Rolling a ball down the hill"

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

Stochastic Gradient Descent

Algorithm: gradient descent

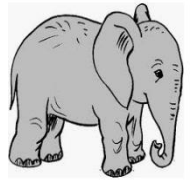
Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

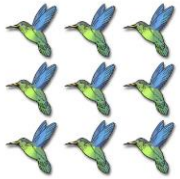
$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

requires iterating through all data points!



Can be expensive



In between: minibatch gradient descent (sample B data points each time)

Most recommended for deep learning B*: batch size*

Algorithm: stochastic gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

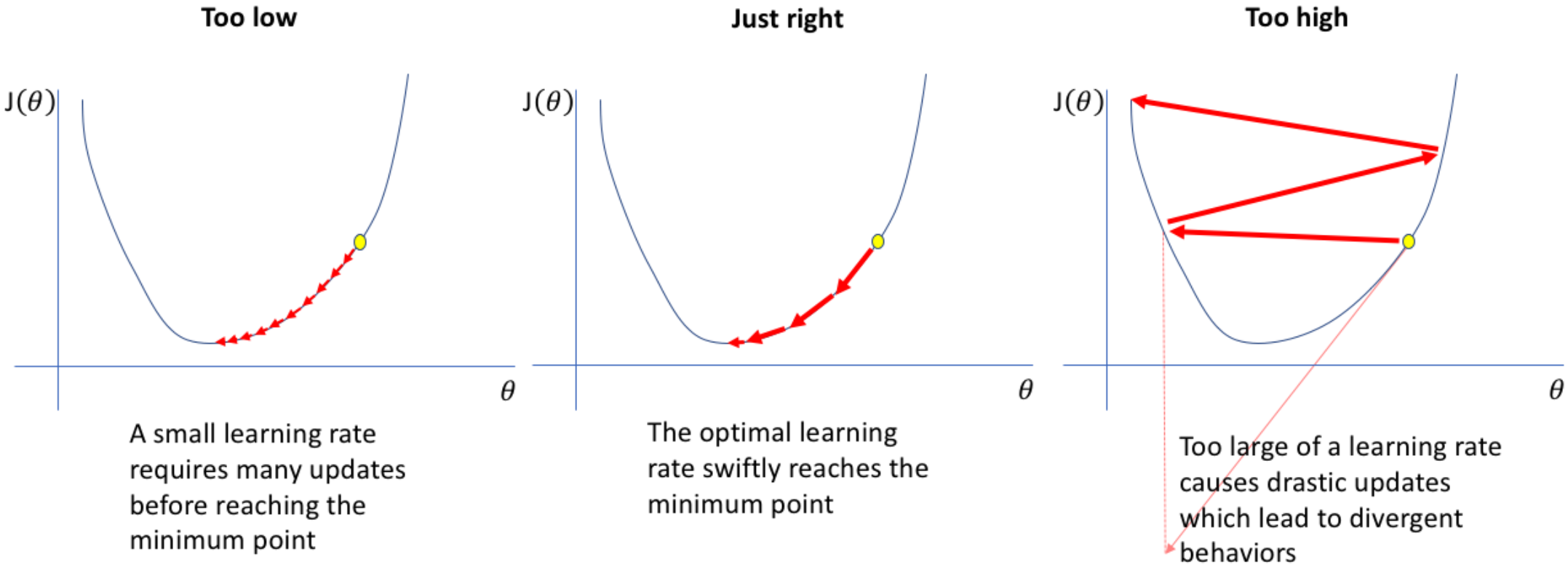
For $(x, y) \in \mathcal{D}_{\text{train}}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

SGD iterates through data samples and update parameters based on each sample

Each update is not as good as GD but we make many more updates.

Impact of Step Size (Learning Rate)

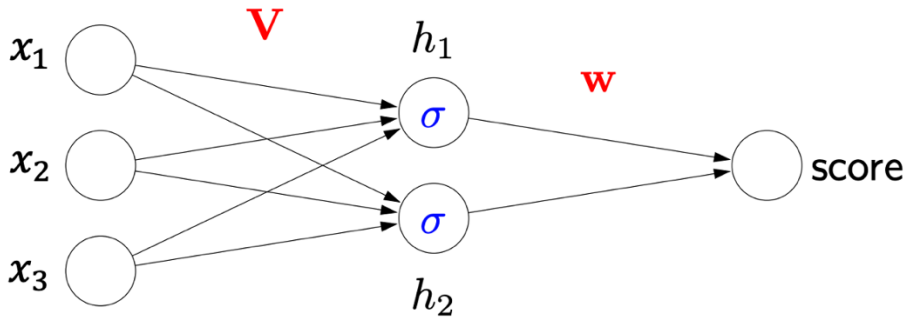


Try this with different learning rates: https://uclaacm.github.io/gradient-descent-visualiser/#playground

Rule of thumb: turn it as a hyperparameter

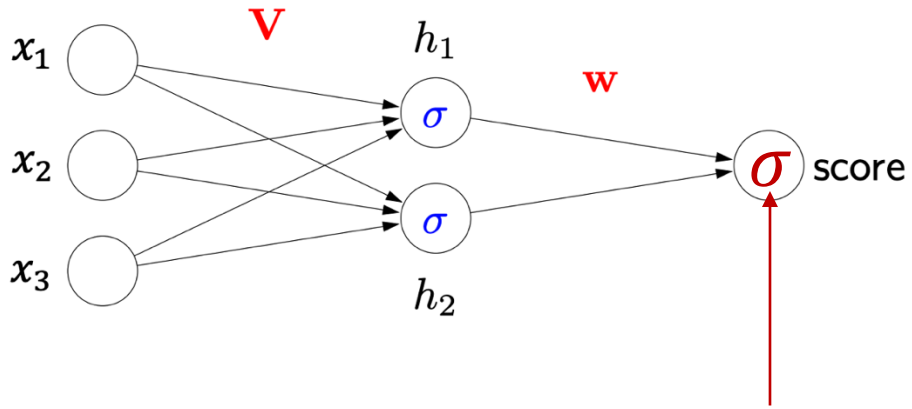
Feedforward Neural Network for Classification

For regression:



$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

For binary classification:



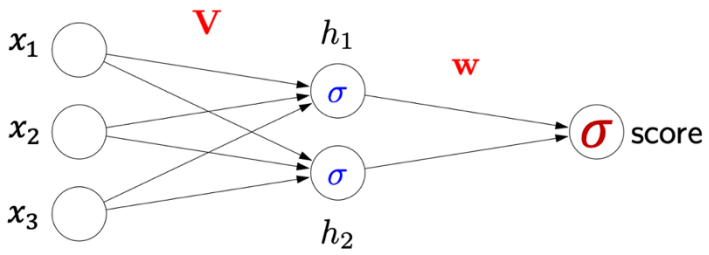
Use a sigmoid function to map the real-valued score to (0, 1)

~~$\text{Loss}(\mathbf{x}, y, \mathbf{w}) = \log(1 + \exp(-(\mathbf{w}^\top \mathbf{x})y))$~~

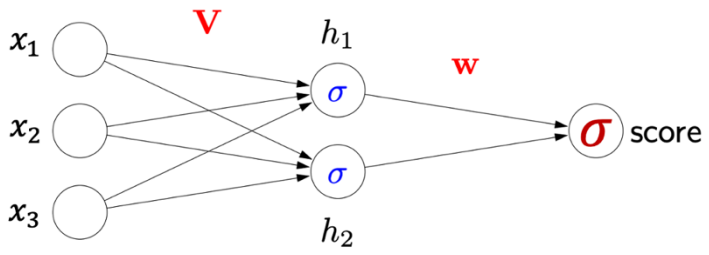
$$\text{Loss}(\mathbf{x}, y) = \log\left(1 + \exp\left(-\left(\mathbf{w}^\top \mathbf{h}\right) y\right)\right)$$

Feedforward Neural Network for Classification

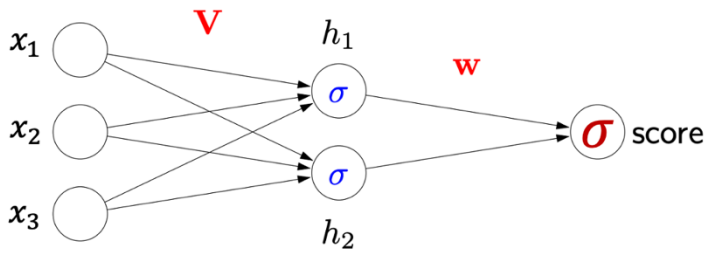
For multiclass classification (OvR):



class 1 vs *not class 1*



class 2 vs *not class 2*

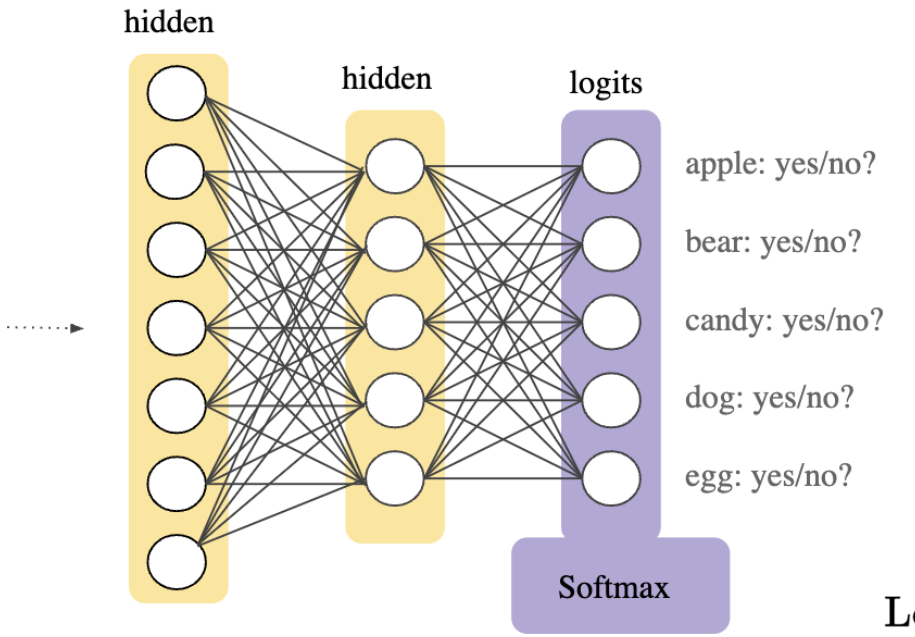


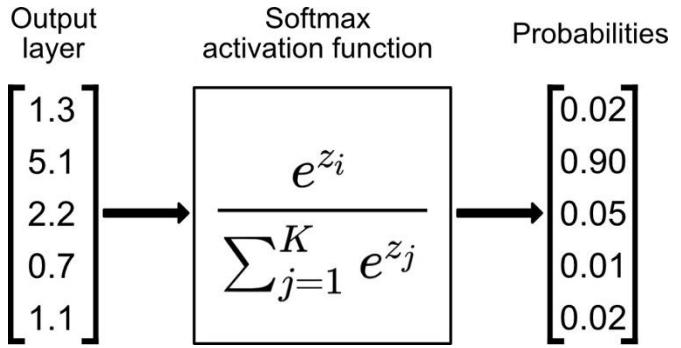
class 3 vs *not class 3*

Not efficient!

Feedforward Neural Network for Classification

Recall: interpret h as learned features for x





Use softmax activation function for the output layer

Maps real-valued scores to a distribution over the classes

$$\text{Loss}(\mathbf{x}, \mathbf{y}, \Theta) = -\sum_{k=1}^{K} \mathbf{1}[y_k = +1] \log p(y_k = +1 \mid \mathbf{x}) + \mathbf{1}[y_k = -1] \log p(y_k = -1 \mid \mathbf{x})$$

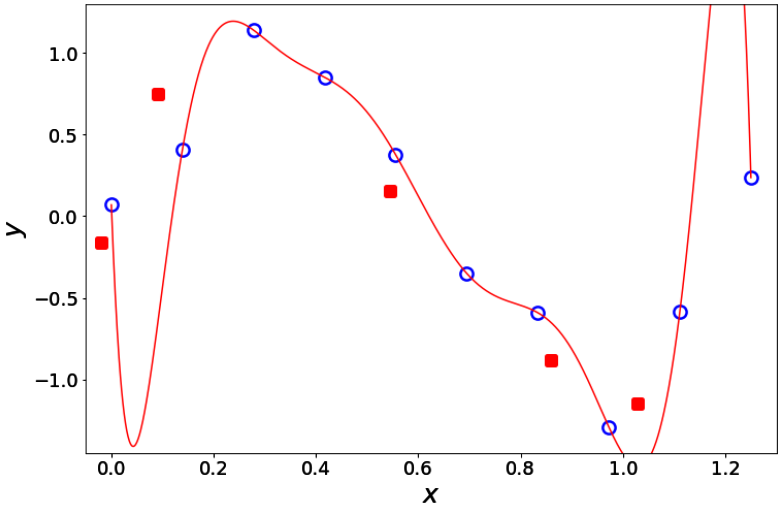
Θ: all model parameters, y_k: the k-th label

Regularization Techniques for Deep Learning

Regularization Alleviates Overfitting

- Recall: Overfitting happens if the model is too complex or training data is not enough.
- Solution: Use regularization techniques
- Occam's Razar: Among multiple competing hypotheses, the simplest is the best.

(William of Ockham 1285-1347)



MSE (original data) = 0

MSE (new data) = 136.76

Parameter Norm Penalties

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

- W: model parameters, λ: regularization strength (hyperparameter)
- Popular Choices:

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

- Benefits:
  - Express preferences over weights
  - Make the model simple to avoid overfitting

Parameter Norm Penalties

- Example:
  - $x = [1,\ 1,\ 1,\ 1]$
  - $w_1 = [1, 0, 0, 0]$
  - $w_2 = [0.4, 0.2, 0.25, 0.15]$

- $w_1^T x = w_2^T x = 1$

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

Which of w_1 or w_2 will the L2 regularization prefer?

L2 regularization tends to "spread out" the weights
→ More robust to error in individual features.

Parameter Norm Penalties

- Example:
  - $x = [1,\ 1,\ 1,\ 1]$
  - $w_1 = [1, 0, 0, 0]$
  - $w_2 = [0.4, 0.2, 0.25, 0.15]$

- $w_1^T x = w_2^T x = 1$

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Which of w_1 or w_2 will the L1 regularization prefer?

L1 regularization tends to encourage sparsity.
→ Useful for feature selection.

Parameter Norm Penalties

- They are often applied to linear models as well.

sklearn.linear_model.LogisticRegression

class `sklearn.linear_model.`**`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) [source]

penalty : *{'l1', 'l2', 'elasticnet', None}, default='l2'*

Specify the norm of the penalty:

- `None`: no penalty is added;
- `'l2'`: add a L2 penalty term and it is the default choice;
- `'l1'`: add a L1 penalty term;
- `'elasticnet'`: both L1 and L2 penalty terms are added.

sklearn.linear_model.Ridge ¶

class `sklearn.linear_model.`**`Ridge`**(*alpha=1.0, *, fit_intercept=True, copy_X=True, max_iter=None, tol=0.0001, solver='auto', positive=False, random_state=None*) [source]

Linear least squares with l2 regularization.

sklearn.linear_model.Lasso

class `sklearn.linear_model.`**`Lasso`**(*alpha=1.0, *, fit_intercept=True, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) ¶ [source]

Linear Model trained with L1 prior as regularizer (aka the Lasso).

Parameter Norm Penalties

- L2 regularization is equivalent to *weight decay* in deep learning.

- Update in stochastic gradient descent with L2 regularization:

$$\mathbf{w} \leftarrow (1 - 2\eta\lambda)\,\mathbf{w} - \eta\frac{\partial \ell}{\partial \mathbf{w}}$$

Equivalent to *weight decay*: decaying the weight before update it.

CLASS torch.optim.SGD(*params*, *lr=<required parameter>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, ***, *maximize=False*, *foreach=None*, *differentiable=False*) [SOURCE]

- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

Dataset Augmentation

- Idea: Train the model on more datasets.
- Dataset Augmentation: Create fake data by some transformations.
- Commonly used in training CNN models.
- Some commonly used transformation:

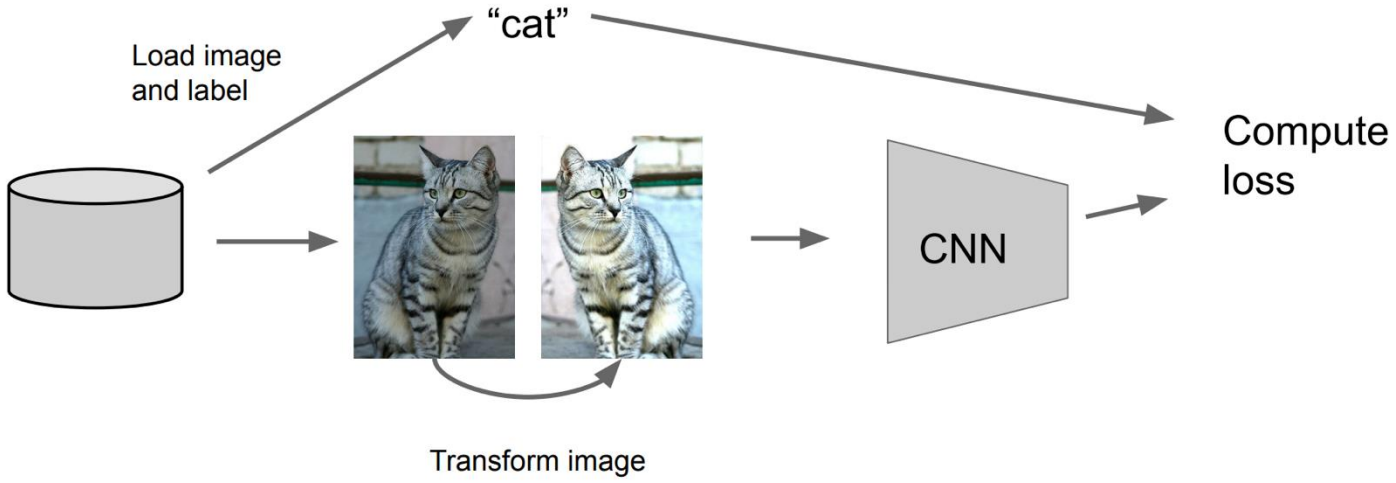


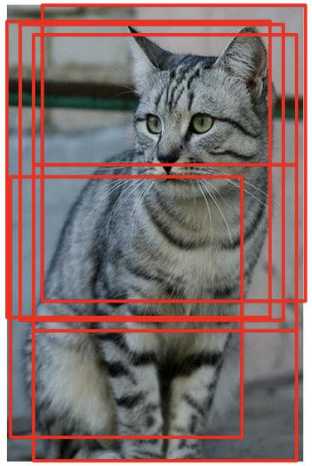
Image from Lecture 7 of Stanford cs231n

Dataset Augmentation

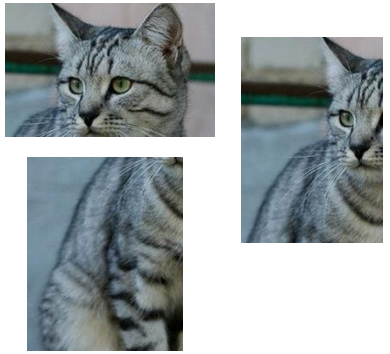
- Some commonly used transformations:

original data

Horizontal flips



Random crops and scales



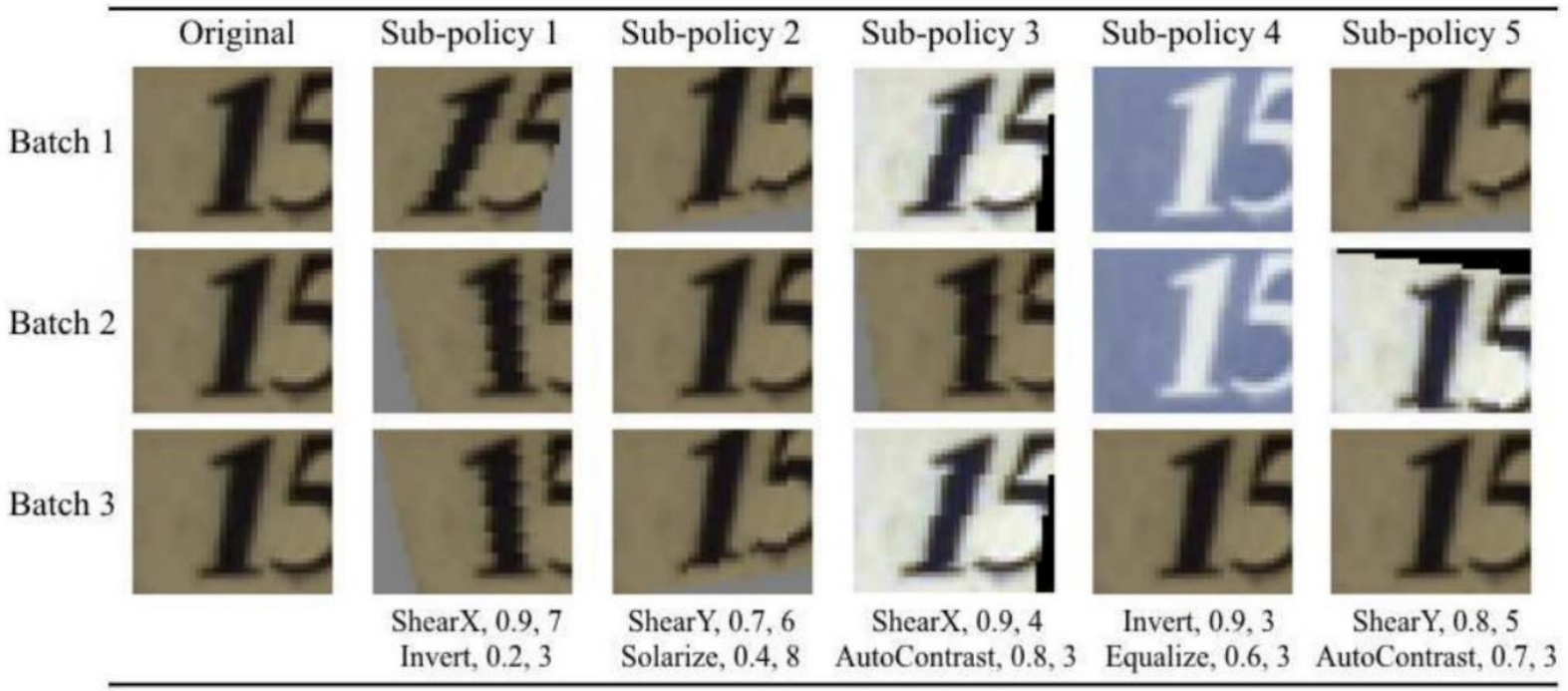
Color jitter (contrast & brightness)

- Translation
- Rotation
- Stretching
- …

Examples from Lecture 7 of Stanford cs231n

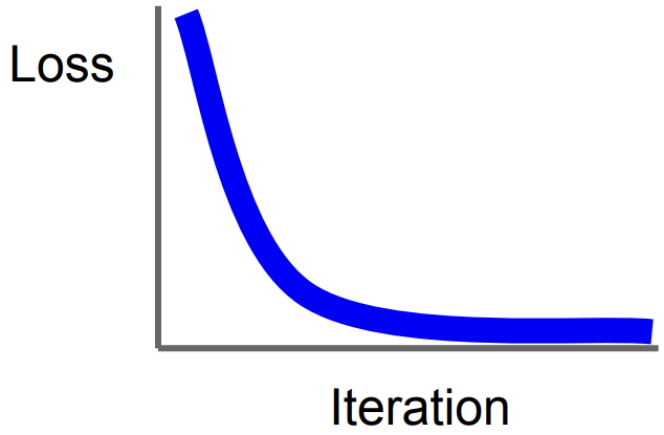
Dataset Augmentation

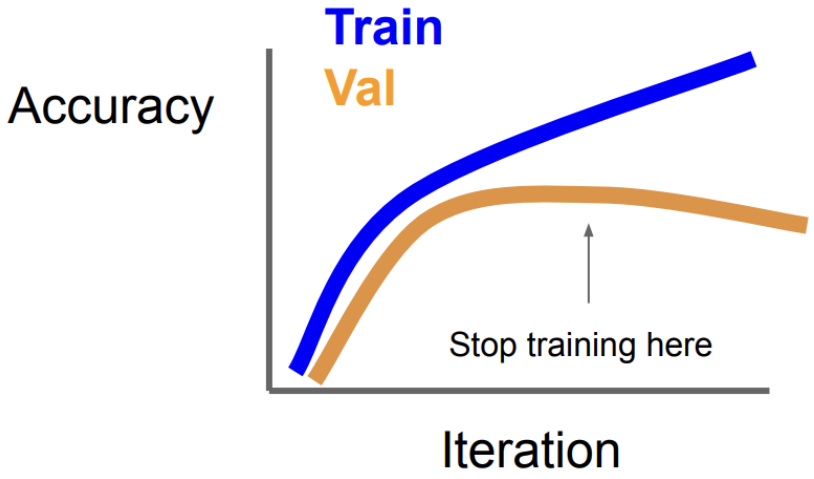
- Another example: Automatic Data Augmentation



Examples from Lecture 7 of Stanford cs231n
Original source: Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

Early Stopping





- When training the model, we usually hold out a part of the training data as validation set. *(Different from the test set, which is used for performance evaluation)*
- During model training, we monitor the accuracy (or other metrics) in training and validation sets.
- When accuracy on the validation set decreases, stop the training.
- Rule of thumb: always do this.

Rethinking Generalization: Bias-Variance Tradeoff

- Generalization error can be decomposed as:

$$\underbrace{E_{\mathbf{x},y,D}\left[(h_D(\mathbf{x})-y)^2\right]}_{\text{Expected Test Error}} = \underbrace{E_{\mathbf{x},D}\left[\left(h_D(\mathbf{x})-\bar{h}(\mathbf{x})\right)^2\right]}_{\text{Variance}} + \underbrace{E_{\mathbf{x},y}\left[(\bar{y}(\mathbf{x})-y)^2\right]}_{\text{Noise}} + \underbrace{E_{\mathbf{x}}\left[\left(\bar{h}(\mathbf{x})-\bar{y}(\mathbf{x})\right)^2\right]}_{\text{Bias}^2}$$

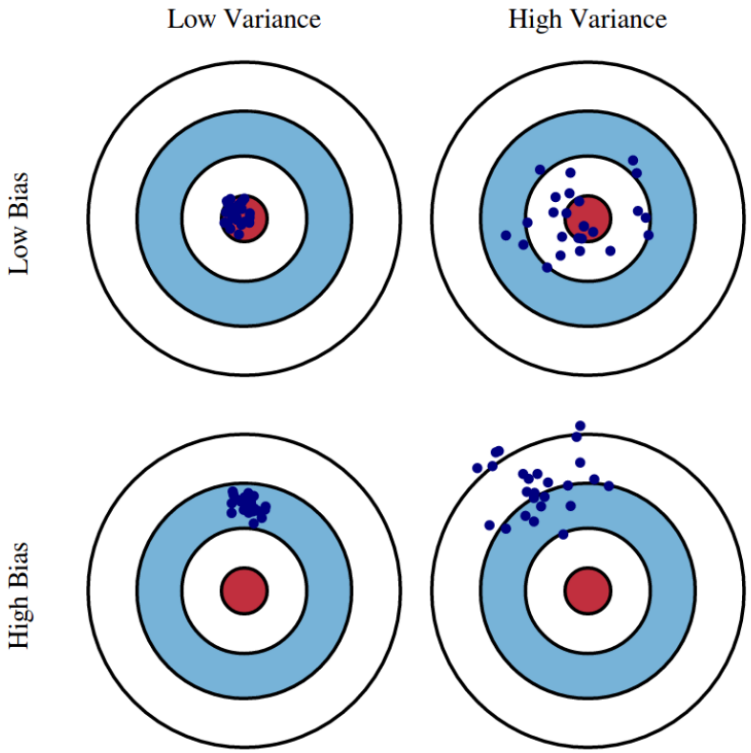
Don't be scared, you don't have to memorize this complex formula.

But you are encouraged to look at how it is derived, if interested:
https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html

Rethinking Generalization: Bias-Variance Tradeoff

- Generalization error can be decomposed as: (a simpler version)

$$\text{Expected test error} = \text{Variance} + \text{Bias}^2 + \text{Noise}$$



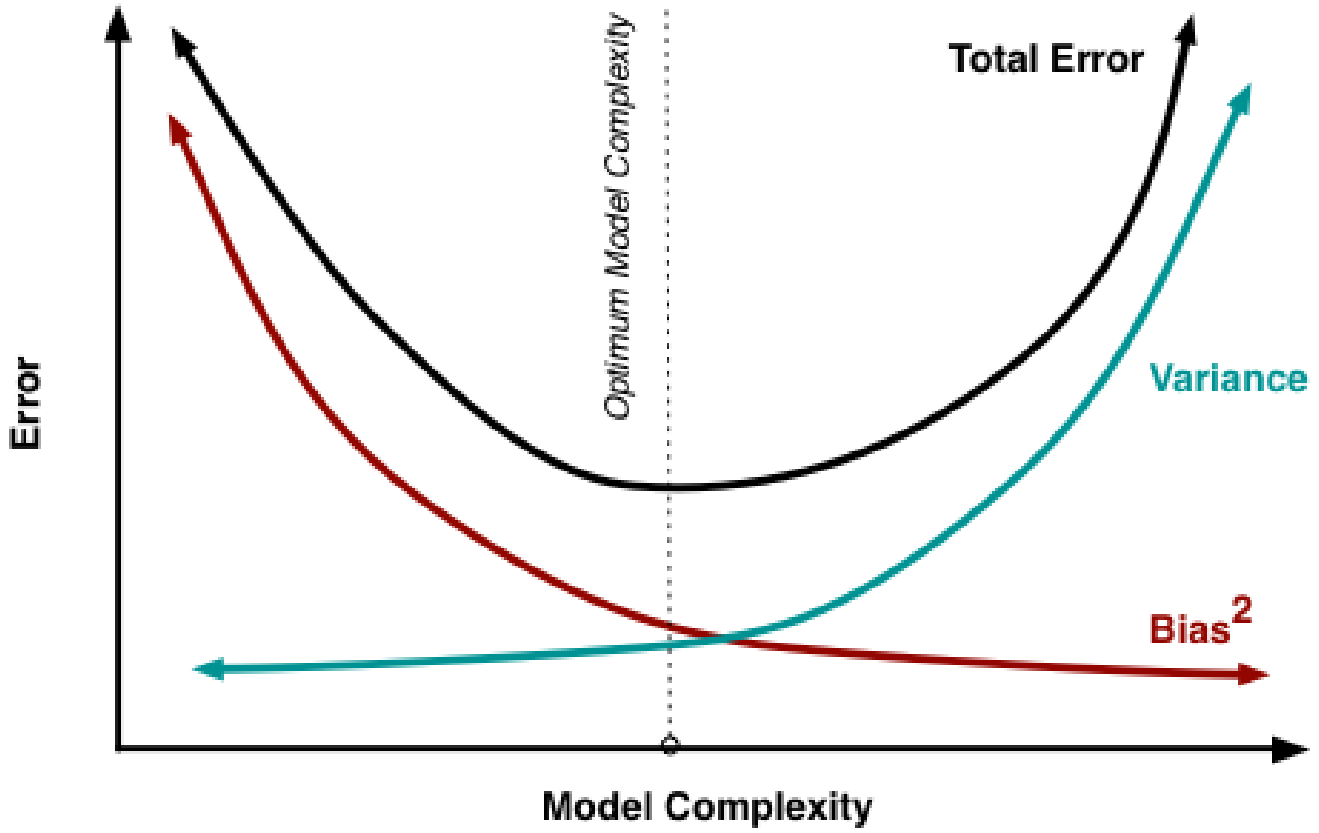
- Variance: how much the classifier changes if trained on a different training set.
- Bias: what is the inherent error obtained from the classifier even with infinite training data.

- Linear models tend to have high bias and low variance.
- Deep learning tends to have low bias but very high variance.
- Deep learning tends to explore interactions between features.

Rethinking Generalization: Bias-Variance Tradeoff

- Generalization error can be decomposed as: (a simpler version)

Expected test error = Variance + Bias^2 + Noise



Dropout

- Norm penalization promotes “simplicity”: use norm as a measure of simplicity.

- Another measure of simplicity: smoothness, i.e., the function should not be sensitive to small changes to its inputs.

- Srivastava et al. (2014): propose to inject noise in *every* layer of neural network. (adding noise could enhance smoothness in the input-output mapping)

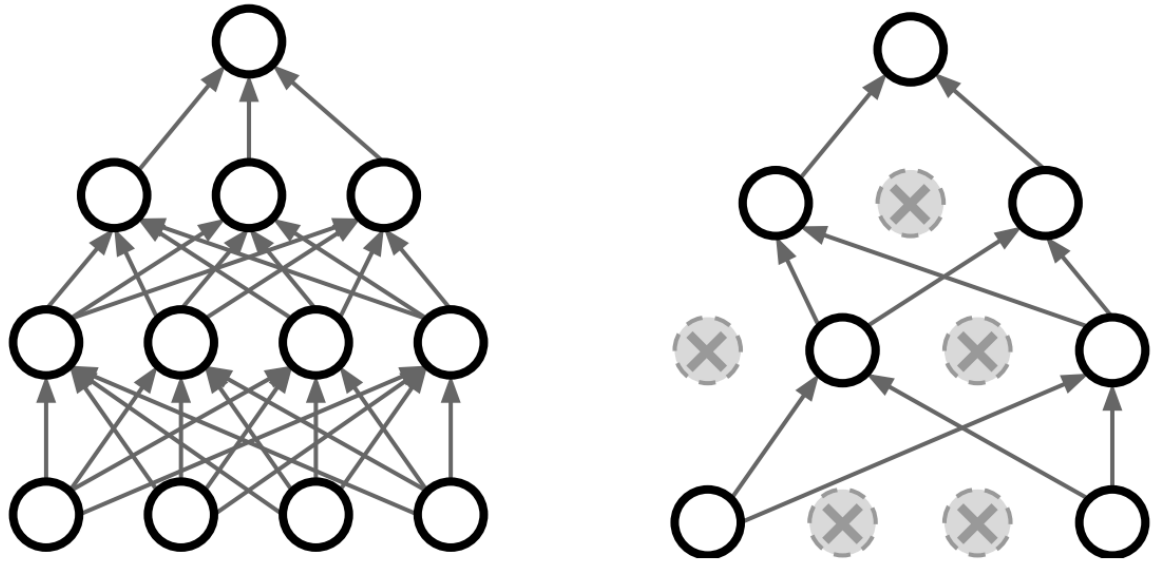
Dropout

- In each forward pass during model training, randomly set some neurons to zero according to some probability ("drop out").
- The probability of dropping: dropout rate p (a hyperparameter; common value: 0.1 - 0.5)

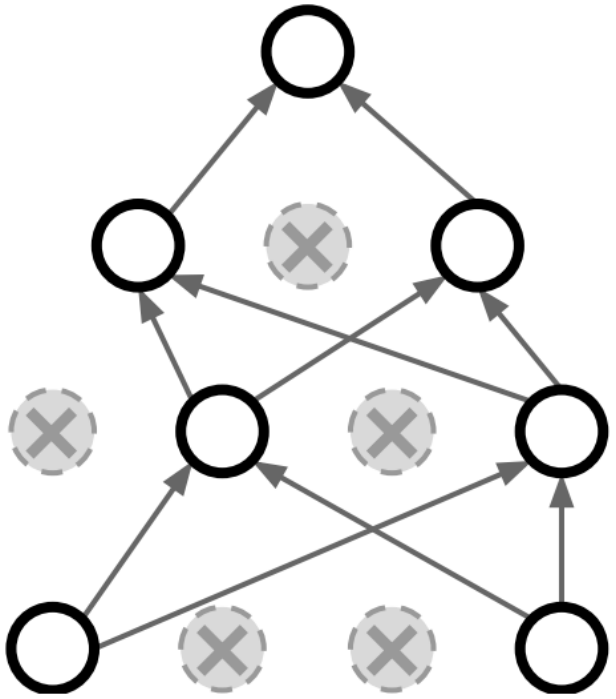
Replace the intermediate h by:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

The expectation is unchanged: $E[h'] = h$

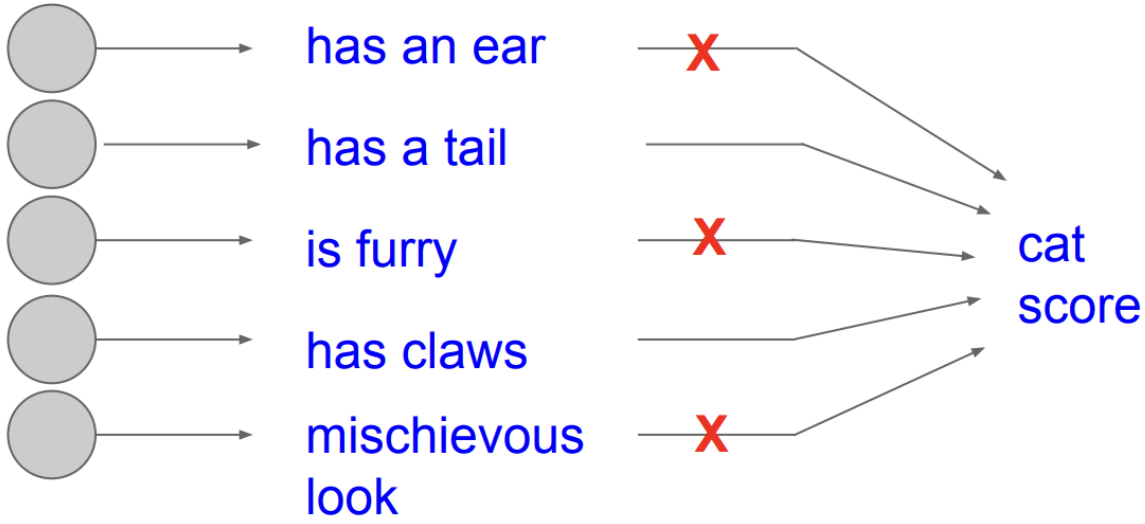


Dropout



- Why it works?

  Forces the network to have a redundant representation;
  Prevents co-adaptation of features



- Note: the testing phase of models with dropout differs from that without using dropout: need weight scaling.

  See page 253-257 of Deep Learning for derivations.
  DL frameworks will automatically handle this in practice.

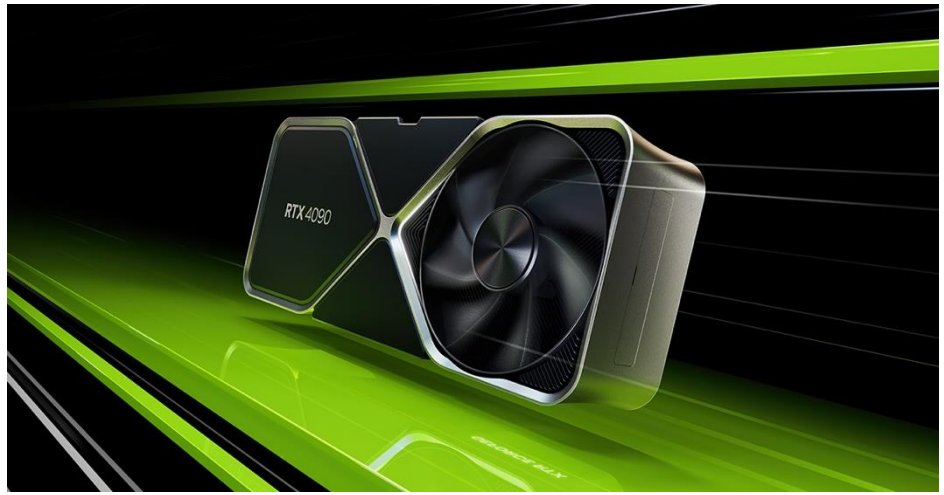
Deep Learning in Practice: Hardware and Software

Hardware: CPUs and GPUs

- Central processing unit (CPU): Fewer cores but each core is very fast; good at sequential tasks.
- Graphics processing unit (GPU): More cores but each core is much slower and "dumber"; good at parallel tasks: e.g., matrix operations.

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	$1199	~13.4 TFLOPs FP32

FLOPs: Floating-point operations per second



Example from https://jasonyanglu.github.io/files/lecture_notes/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0_2021/Lecture%204%20Hardware%20and%20Software.pdf

Hardware: CPUs vs GPUs

A GPU card released 9 years ago

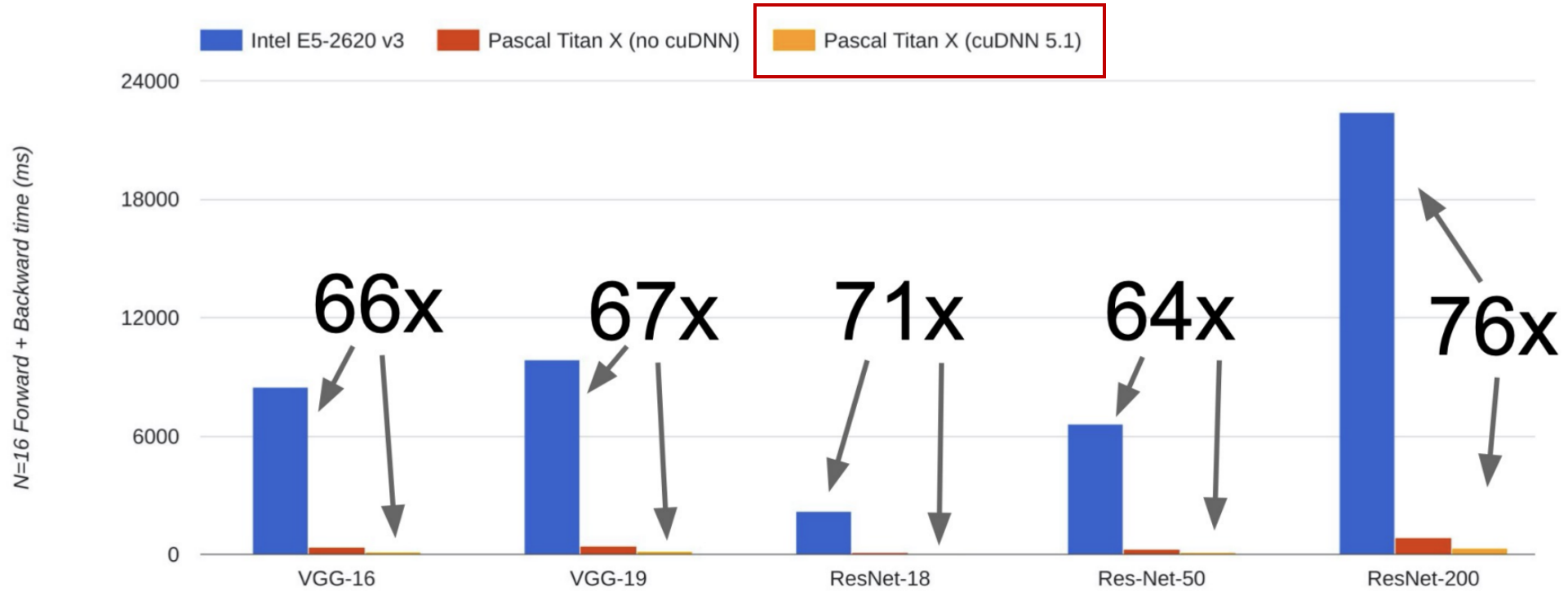


Image from Lecture 6 of Stanford cs231n

Unit Price of Computation Power

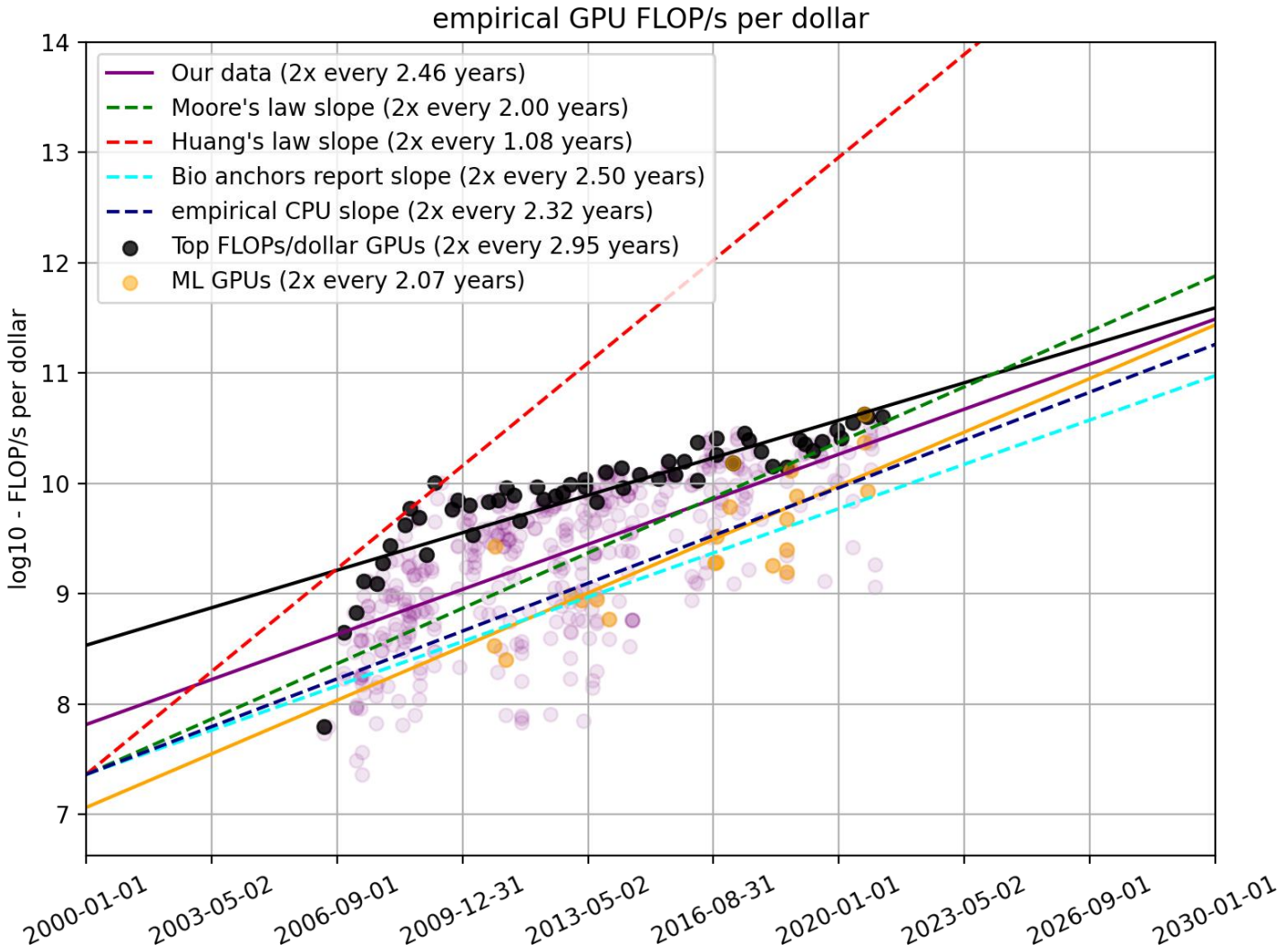
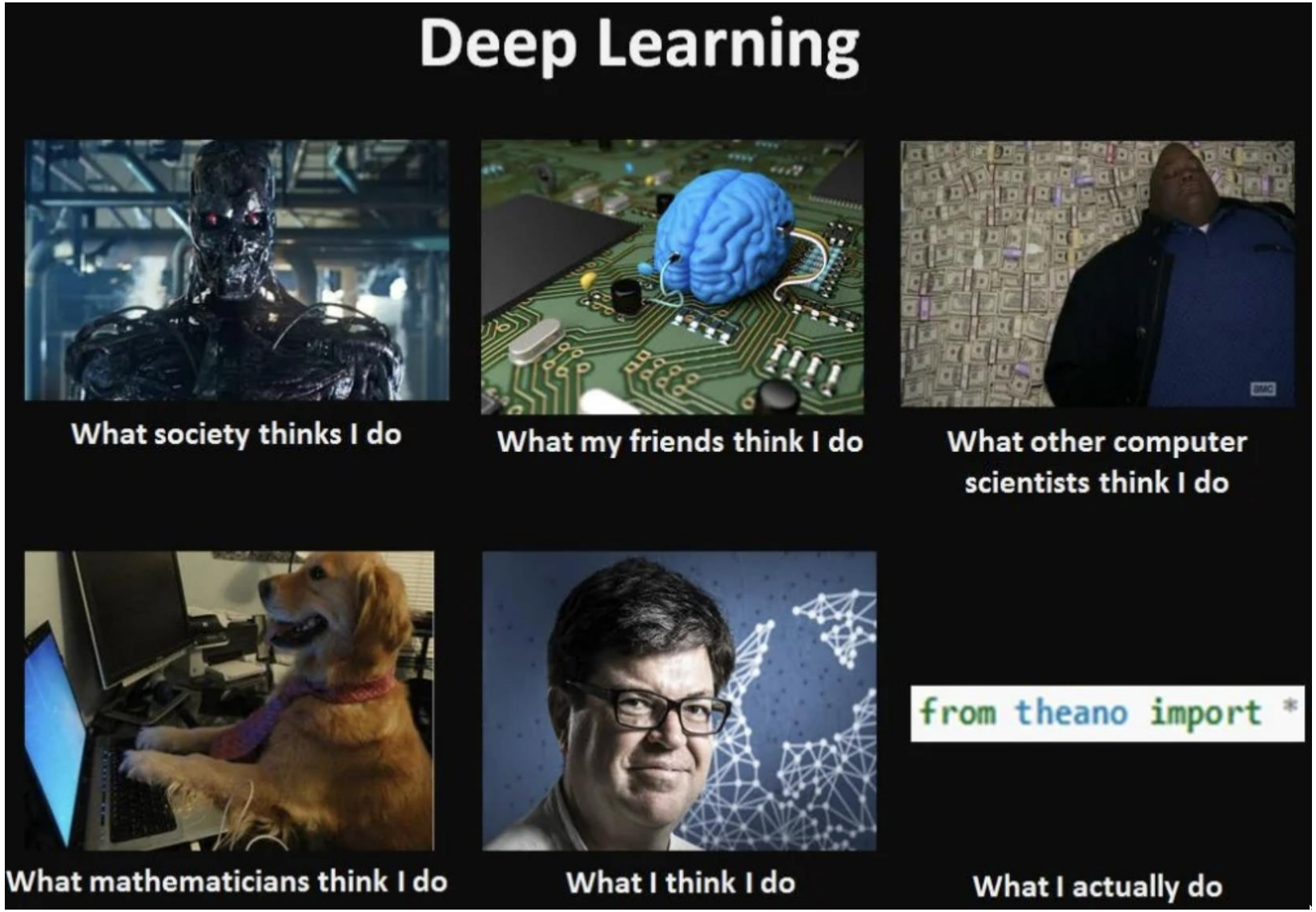
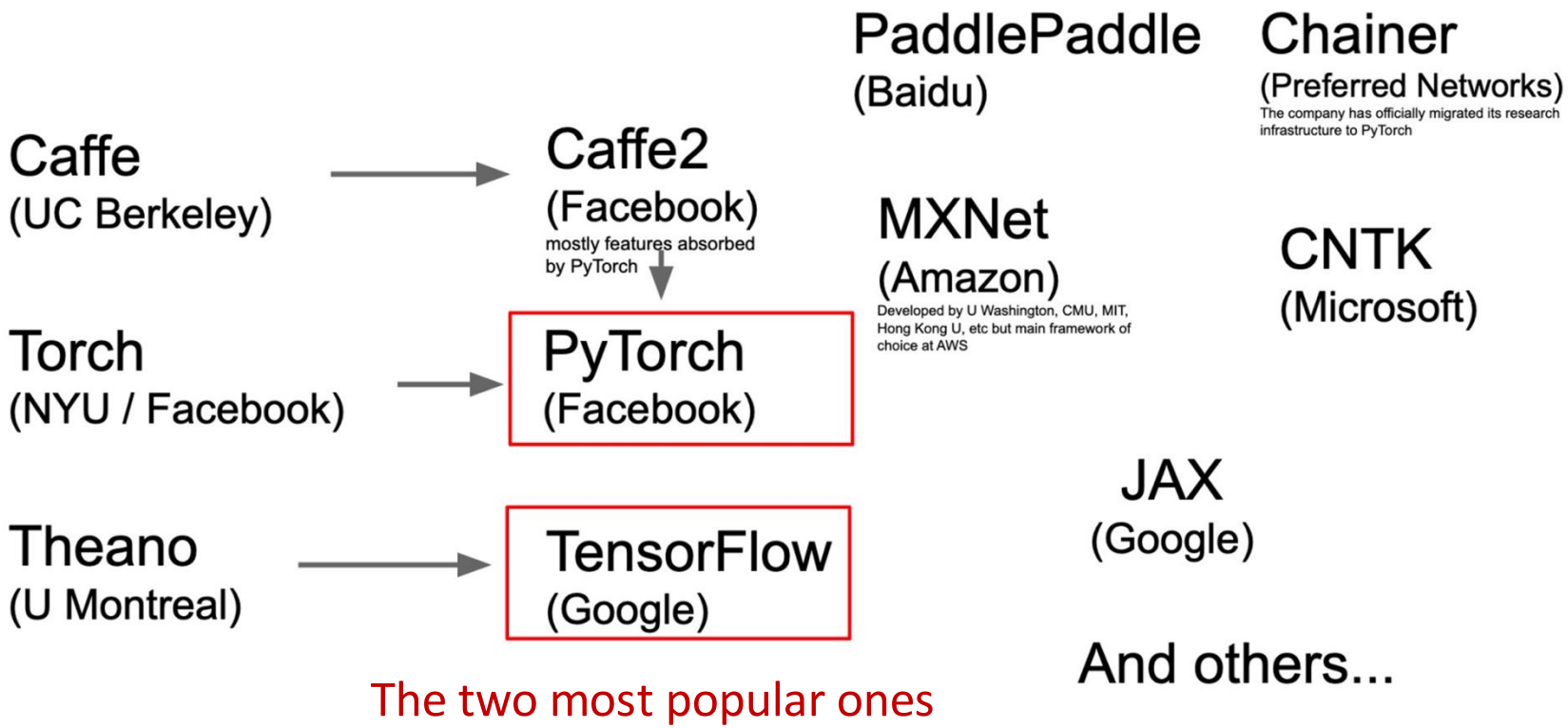


Image from: https://epochai.org/blog/trends-in-gpu-price-performance

Software: Deep Learning Frameworks



Software: Deep Learning Frameworks



We will see details about the basic usage of PyTorch on the lab session

Image from Lecture 6 of Stanford cs231n

Let your voice be heard!

Thank you for your feedback! 🙌