Introduction to machine learning

Recitation 1

MIT - 6.802 / 6.874 / 20.390 / 20.490 / HST.506 - Spring 2021

Jackie Valeri

Slides adapted from Sachit Saksena and previous course materials

The BE Data & Coding Lab

https://bedatalab.github.io

A peer-to-peer educational community supporting computational novices, competent practitioners, and experts in their journey to learn new languages and use those languages to answer important world problems.

MIT BE
BIOLOGICAL ENGINEERING

We are a safe space where it is okay to ask for help

We are a confidential space

We will help you answer your questions, we will not solve your problems for you

We can help with:

…problem sets for classes

…brainstorming ways to incorporate computational modeling into your projects or preliminary project design

…brainstorming and implementing computation into your research projects

…code review, code efficiency, and reproducibility

…much more – just ask if you aren't sure!

BE Data and Coding Lab

Inaugural Cohort

Pablo Cardenas

Dan Anderson

Itai Levin

Maxine Jonas

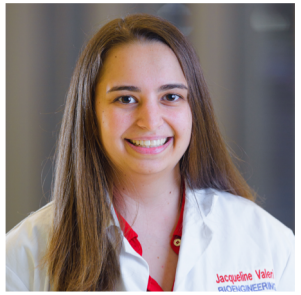
Patrick Holec

Divya Ramamoorthy

Meelim Lee

Krista Pullen

Miguel Alcantar



Jackie Valeri

Python, Matlab, R, COMSOL

MIT BE
BIOLOGICAL ENGINEERING

Onto recitation R01!

A. What can you do with ML?

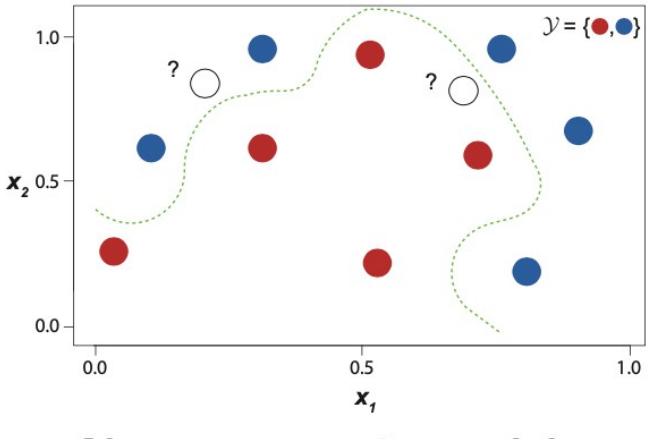
B. Basics of machine learning

C. Neural networks

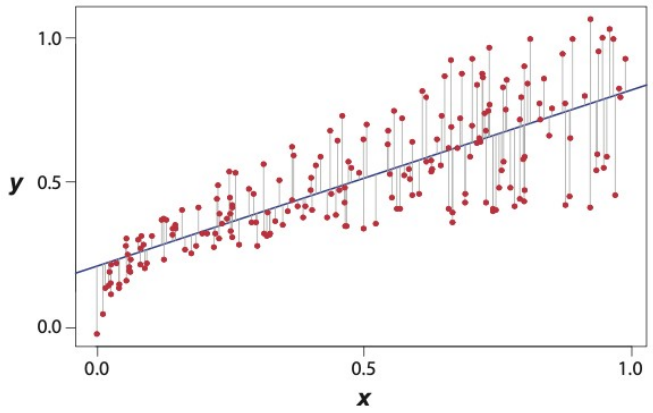
D. Brief preview of pset 1

What can you do with ML?

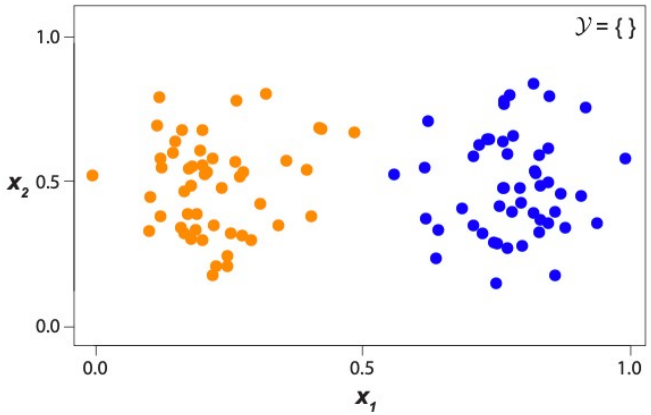
Classification



Regression

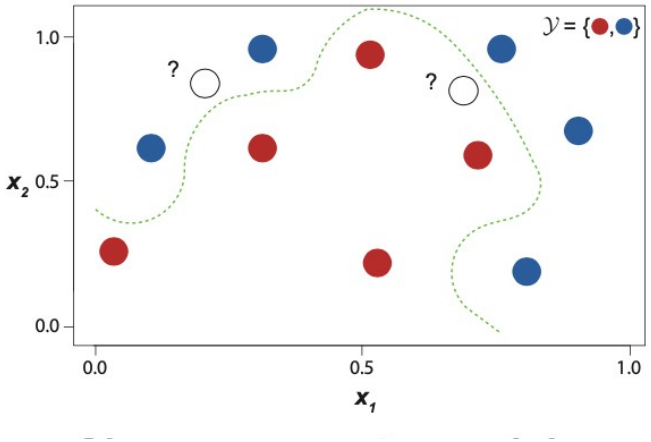


Unsupervised learning

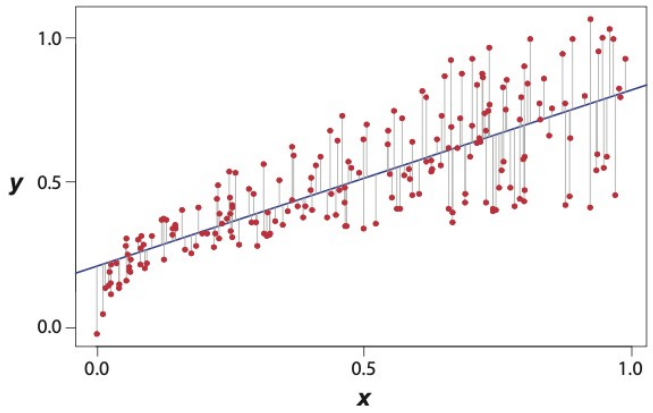


What can you do with ML?

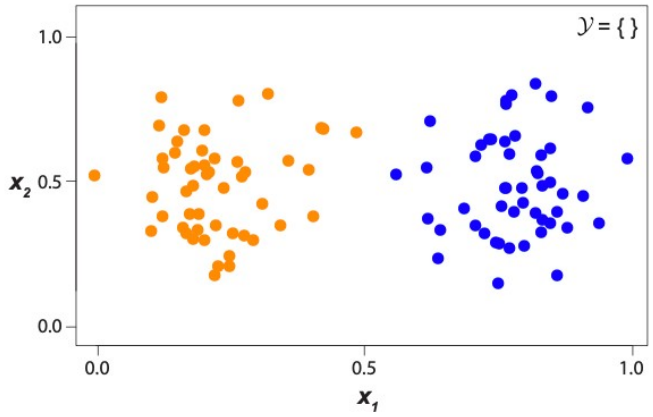
Classification



Regression

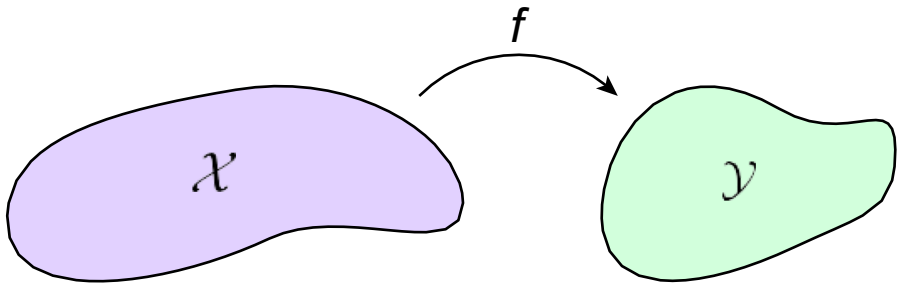


Unsupervised learning



$Y \neq \emptyset$	supervised or semi-supervised learning
$Y = R$	regression
$Y = R^K, K > 1$	multivariate regression
$Y = \{0, 1\}$	binary classification
$Y = \{1, \ldots, K\}$	multi-class classification (integer encoding)
$Y = \{0, 1\}^K, K > 1$	multi-label classification
$Y = \emptyset$	unsupervised learning

What can you do with ML? Terminology



Input $\boldsymbol{X} \in \mathrm{X}$:

- **features** (in machine learning)
- predictors (in statistics)
- independent variables (in statistics)
- regressors (in regression models)
- input variables
- covariates

Output $\boldsymbol{y} \in \mathrm{Y}$:

- **labels** (in machine learning)
- responses (in statistics)
- dependent variables (in statistics)
- regressand (in regression models)
- target variables

Training set $\mathrm{S}_{\text{training}} = \{(\boldsymbol{X}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N} \in \{\mathrm{X}, \mathrm{Y}\}^{N}$, where N is number of training examples

An example is a collection of features (and an associated label)

Training: use $\mathrm{S}_{\text{training}}$ to learn functional relationship $f : \mathrm{X} \rightarrow \mathrm{Y}$

What can you do with ML? Terminology

$$f : X \rightarrow Y$$

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \hat{\boldsymbol{y}}$$

$\boldsymbol{\theta}$:

- **weights** and **biases** (intercepts)
- coefficients $\boldsymbol{\beta}$
- parameters

f:

- model
- hypothesis h
- classifier
- predictor
- discriminative models: P(Y|X)
- generative models: P(X, Y)

Think of weights and biases like lots and lots of
y = mx + b

R01 Outline

A. What can you do with ML?

B. Basics of machine learning
   I. Get data
   II. Identify the space of possible solutions
   III. Formulate an objective
   IV. Choose algorithm
   V. Train (loss)
   VI. Validate results (metrics)

C. Neural networks

D. Brief preview of pset 1

Basics of machine learning: data

Training set ($\mathcal{S}_{\text{training}}$):

- set of examples used for learning
- usually 60 - 80 % of the data

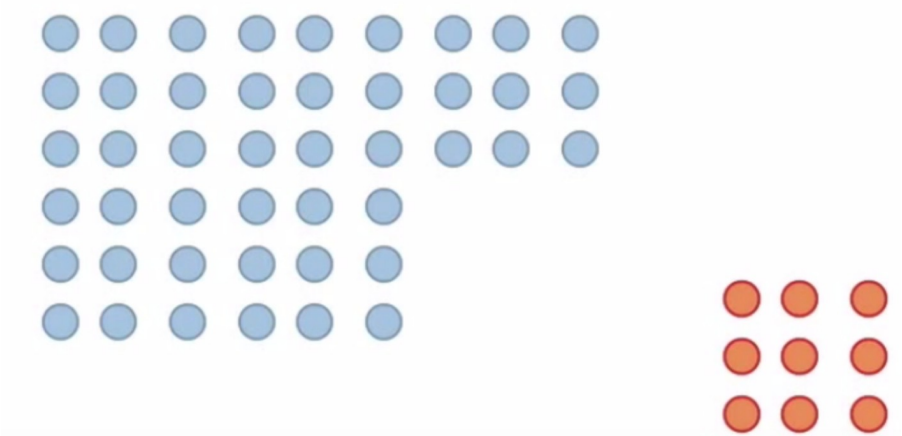
Validation set ($\mathcal{S}_{\text{validation}}$):

- set of examples used to tune the model hyperparameters
- usually 10 - 20 % of the data

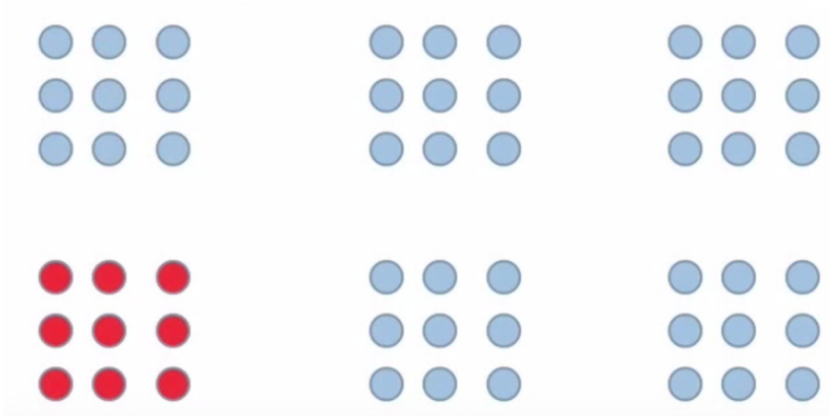
Test set ($\mathcal{S}_{\text{test}}$):

- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30 % of the data

Train—test split (1-fold)



Cross-validation (6-fold)



https://towardsdatascience.com/cross-validation-code-visualization-kind-of-fun-b9741baea1f8

Basics of machine learning: objective functions

An **objective function** J (Θ) is the function that you optimize when training machine learning models. It is usually in the form of (but not limited to) one or combinations of the following:

Loss / cost / error function $L(\hat{y}, y)$:

Likelihood function / posterior:

Regularizers and constraints

https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Basics of machine learning: objective functions

An **objective function** J (Θ) is the function that you optimize when training machine learning models. It is usually in the form of (but not limited to) one or combinations of the following:

Loss / cost / error function $L(\hat{y}, y)$:

Classification

- 0-1 loss
- cross-entropy loss
- hinge loss

Regression

- mean squared error (MSE, L_2norm)
- mean absolute error (MAE, L_1norm)
- Huber loss (hybrid between L_1 and L_2 norm)

Probabilistic inference

- Kullback-Leibler divergence (KL divergence)

Likelihood function / posterior:

- negative log-likelihood (NLL) in maximum likelihood estimation (MLE)
- posterior in maximum a posteriori estimation (MAP)

Regularizers and constraints

- L_1 regularization $||\Theta||_1 = \lambda \sum_{i=1}^{N} |\theta_i|$
- L_2 regularization $||\Theta||_2^2 = \lambda \sum_{i=1}^{N} \theta_i^2$
- max-norm $||\Theta||_2^2 \leq c$

https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Basics of machine learning: objective functions

An **objective function** J (Θ) is the function that you optimize when training machine learning models. It is usually in the form of (but not limited to) one or combinations of the following:

Loss / cost / error function $L(\hat{y}, y)$:

Classification

- 0-1 loss
- cross-entropy loss
- hinge loss

Regression

- mean squared error (MSE, L_2 norm)
- mean absolute error (MAE, L_1 norm)
- Huber loss (hybrid between L_1 and L_2 norm)

Probabilistic inference

- Kullback-Leibler divergence (KL divergence)

Likelihood function / posterior:

- negative log-likelihood (NLL) in maximum likelihood estimation (MLE)
- posterior in maximum a posteriori estimation (MAP)

Regularizers and constraints

- L_1 regularization $||\Theta||_1 = \lambda \sum_{i=1}^{N} |\theta_i|$
- L_2 regularization $||\Theta||_2^2 = \lambda \sum_{i=1}^{N} \theta_i^2$
- max-norm $||\Theta||_2^2 \leq c$

https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Basics of machine learning: loss

Task	Loss
Regression (penalize large errors)	$\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
Regression (penalize error linearly)	$\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}\left\lvert y^{(i)} - \hat{y}^{(i)}\right\rvert$

Basics of machine learning: objective functions

An **objective function** J (Θ) is the function that you optimize when training machine learning models. It is usually in the form of (but not limited to) one or combinations of the following:

Loss / cost / error function $L(\hat{y}, y)$:

Classification

- 0-1 loss
- cross-entropy loss
- hinge loss

Regression

- mean squared error (MSE, L_2norm)
- mean absolute error (MAE, L_1norm)
- Huber loss (hybrid between L_1 and L_2 norm)

Probabilistic inference

- Kullback-Leibler divergence (KL divergence)

Likelihood function / posterior:

- negative log-likelihood (NLL) in maximum likelihood estimation (MLE)
- posterior in maximum a posteriori estimation (MAP)

Regularizers and constraints

- L_1 regularization $||\Theta||_1 = \lambda \sum_{i=1}^{N} |\theta_i|$
- L_2 regularization $||\Theta||_2^2 = \lambda \sum_{i=1}^{N} \theta_i^2$
- max-norm $||\Theta||_2^2 \leq c$

https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Basics of machine learning: loss

0-1 loss:

$$\mathcal{L}_{0\text{-}1}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} \mathbb{1}([\hat{y}^{(i)}] \neq y^{(i)}) = \sum_{i=1}^{N} \begin{cases} 1, & \text{for } \hat{y}^{(i)} \neq y^{(i)} \\ 0, & \text{for } \hat{y}^{(i)} = y^{(i)} \end{cases}$$

where $[x]$ is the function that rounds x to the nearest integer.

Binary cross-entropy loss (for binary classification):

$$\begin{aligned} \mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) &= \sum_{i=1}^{N} -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ &= \sum_{i=1}^{N} \begin{cases} -\log(\hat{y}^{(i)}), & \text{for } y^{(i)} = 1 \\ -\log(1 - \hat{y}^{(i)}), & \text{for } y^{(i)} = 0 \end{cases} \end{aligned}$$

$\boldsymbol{y}$	$\hat{\boldsymbol{y}}$	$[\hat{\boldsymbol{y}}]$	$\mathcal{L}_{0\text{-}1}(\hat{\boldsymbol{y}}, \boldsymbol{y})$	$\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$[1, 0, 0]$	$[0.9, 0.2, 0.4]$	$[1, 0, 0]$	0	0.84
$[1, 1, 0]$	$[0.6, 0.4, 0.1]$	$[1, 0, 0]$	1	1.53
$[1, 0, 1]$	$[0.1, 0.7, 0.3]$	$[0, 1, 0]$	3	4.71

Basics of machine learning: metrics

		True condition		
	Total population	Condition positive	Condition negative	Accuracy = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$
	Predicted condition negative	**False negative**, Type II error	**True negative**	
		Recall, Sensitivity = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	Specificity = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	F_1 score = $\frac{1}{\frac{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}{2}}$

Basics of machine learning: metrics

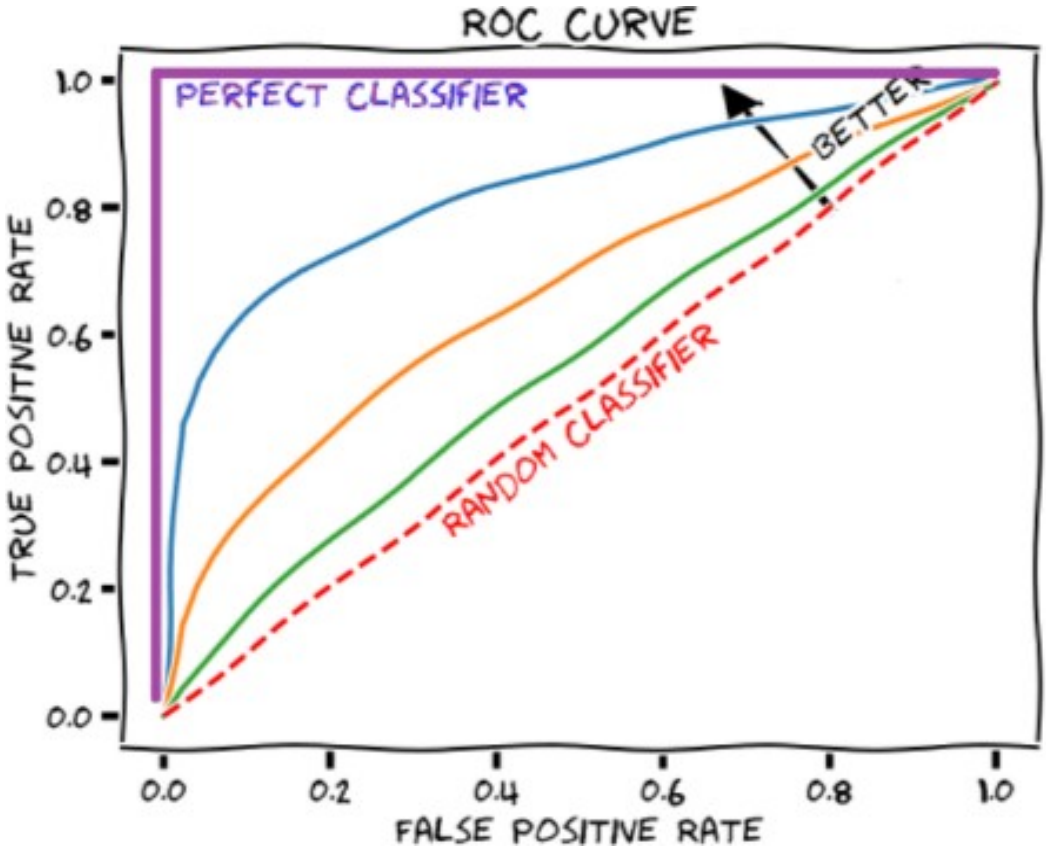
		True condition		
	Total population	Condition positive	Condition negative	Accuracy = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$
	Predicted condition negative	**False negative**, Type II error	**True negative**	
		Recall, Sensitivity = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	Specificity = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	F_1 score = $\frac{1}{\frac{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}{2}}$

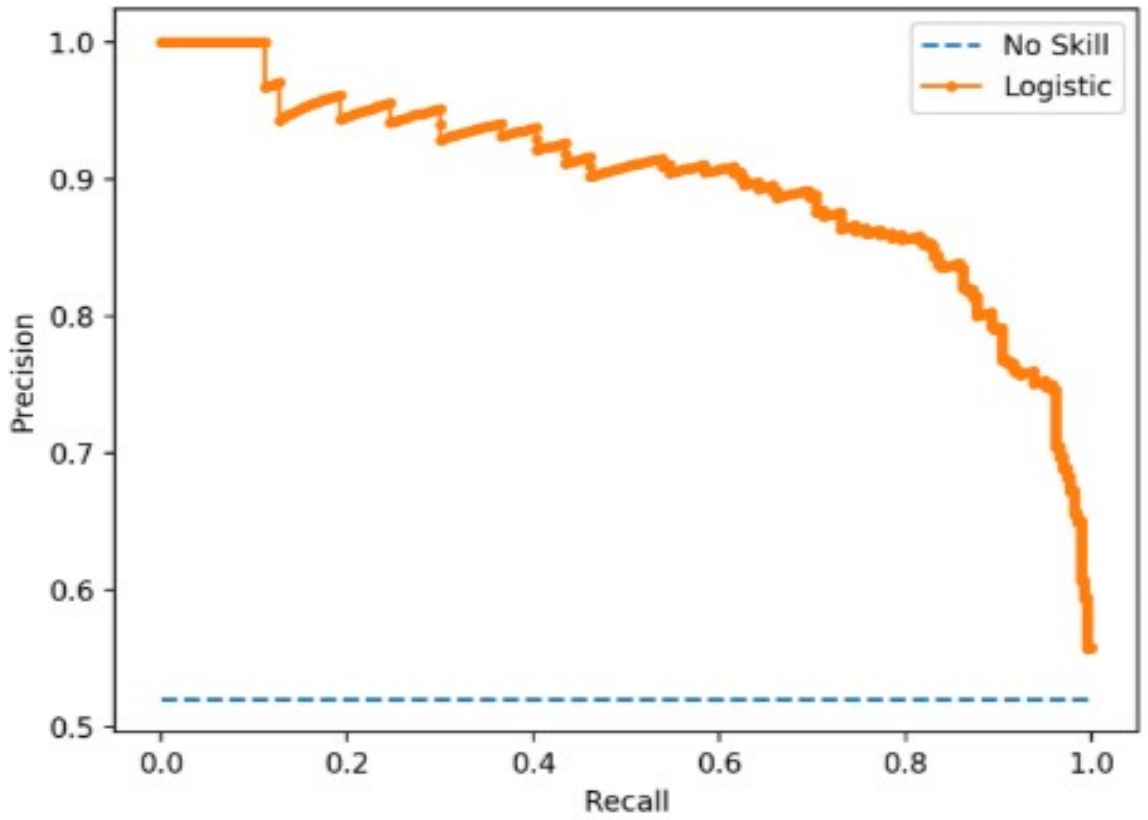
Precision = # true positive / # predicted positive

True positive rate = Sensitivity

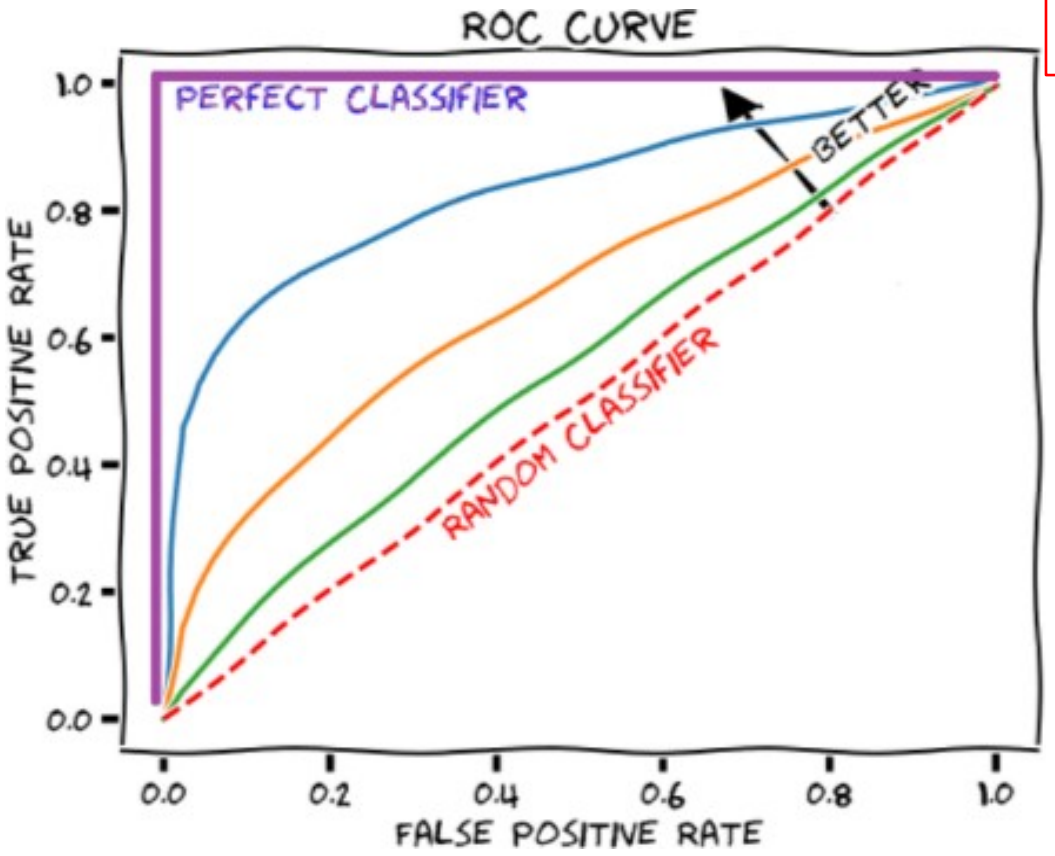
False positive rate = 1 – specificity
False positive rate = # false positives / all condition negatives

Basics of machine learning: metrics





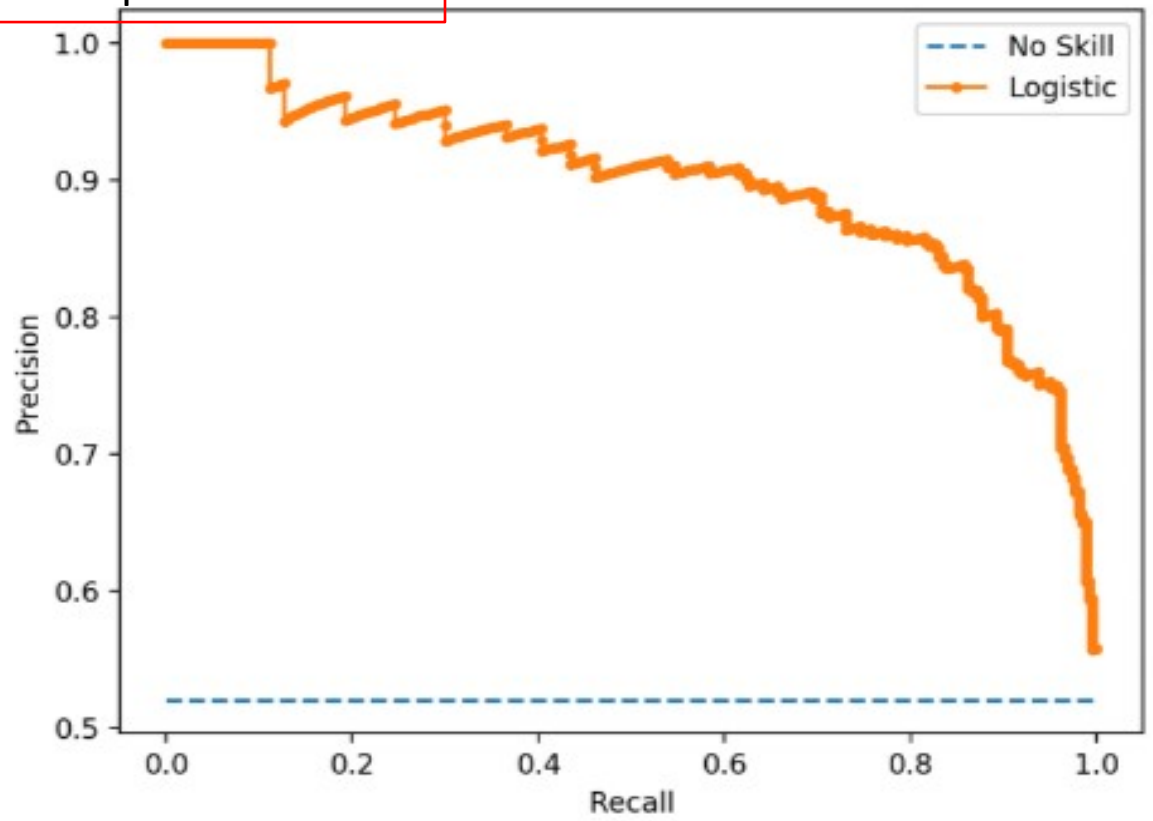
Basics of machine learning: metrics



True positive rate = # true positive / # actually positive

False positive rate = 1 – specificity
False positive rate = # false positives / all condition negatives

Precision = # true positive / # predicted positive



Recall is the same as true positive rate!

R01 Outline

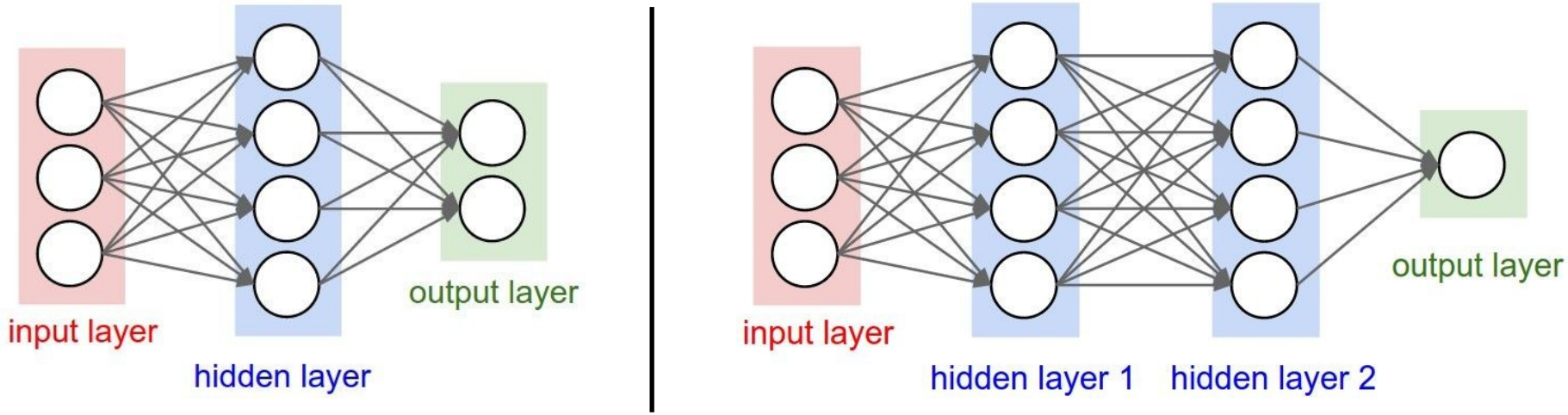
A. What can you do with ML?

B. Basics of machine learning

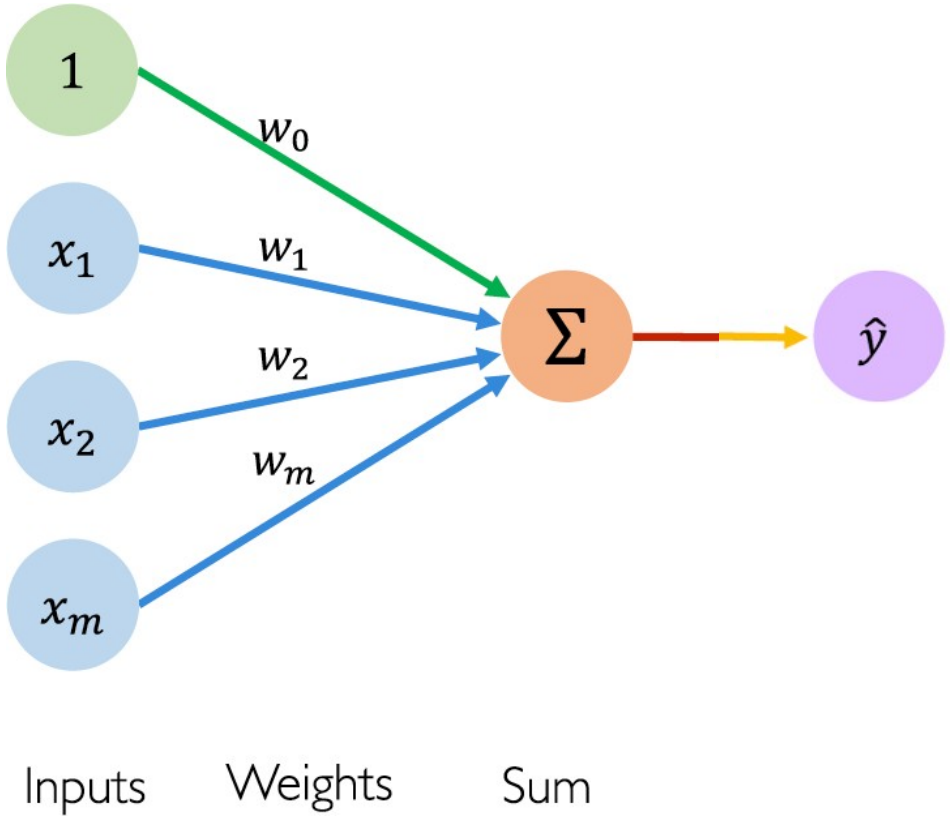
C. Neural networks
   I. Perceptrons to neurons
   II. Activation functions
   III. Training with backpropagation
   IV. Gradient descent
   V. Regularization

D. Brief preview of pset 1

Neural networks: perceptrons to neurons

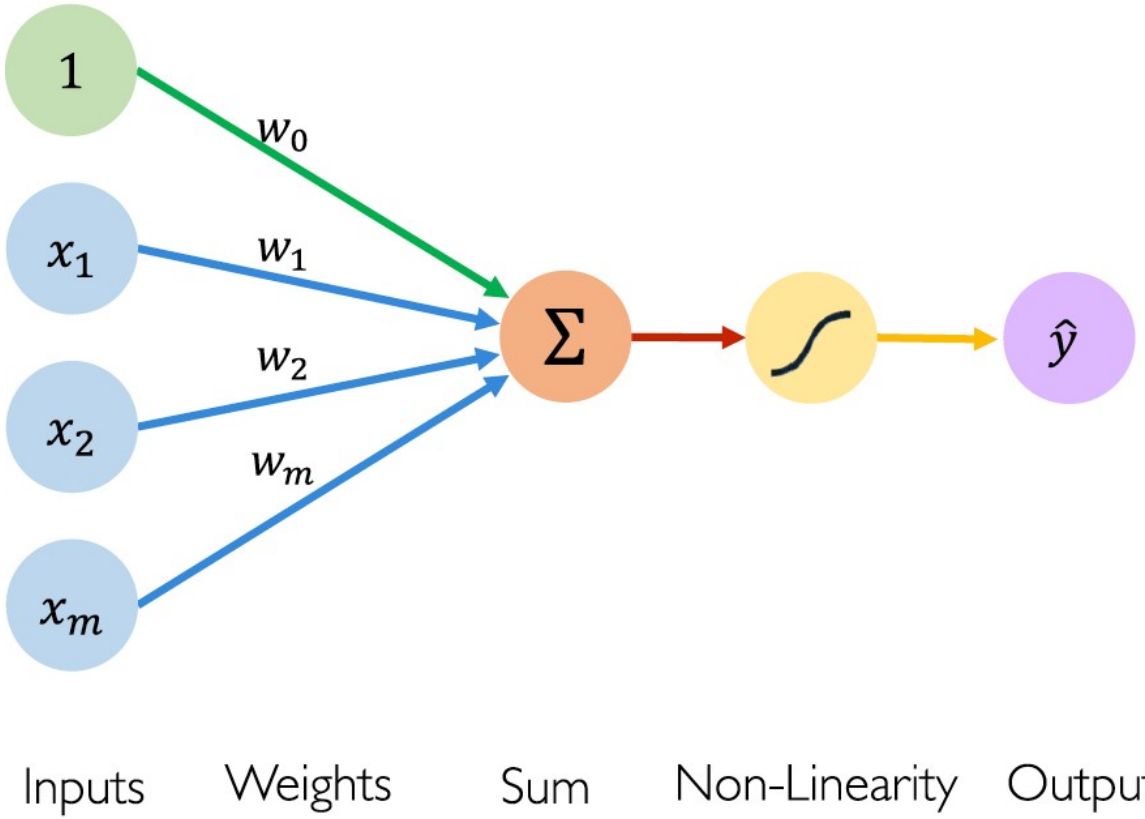


Neural networks: perceptrons to neurons



$$\hat{y} = w_0 + \sum_{i=1}^{m} x_i w_i$$

Neural networks: perceptrons to neurons



Output

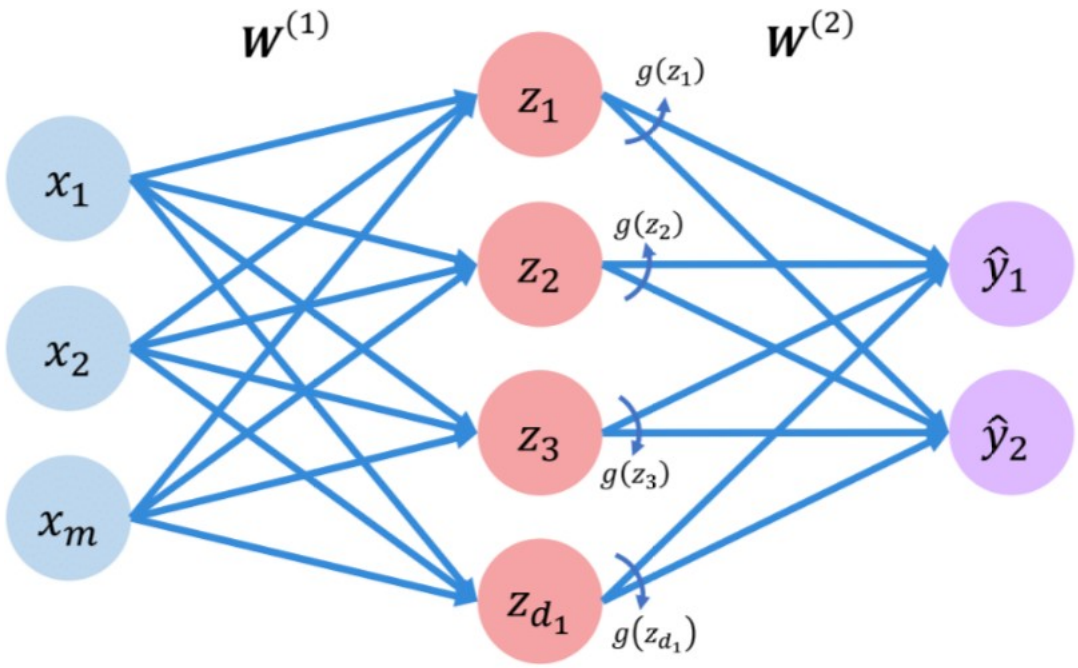
Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Non-linear activation function

Bias

Neural networks: single layer feed-forward NN

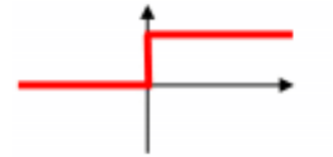


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)} \right)$$

Neural networks: activation functions

Step function

$$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

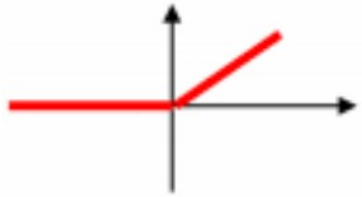


Sigmoid

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

Rectified linear unit

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} = \max(0, z)$$

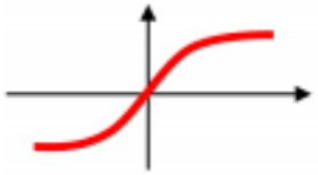


Softmax

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1) / \sum_i \exp(z_i) \\ \vdots \\ \exp(z_n) / \sum_i \exp(z_i) \end{bmatrix}$$

Hyperbolic tangent

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



Neural networks: activation functions

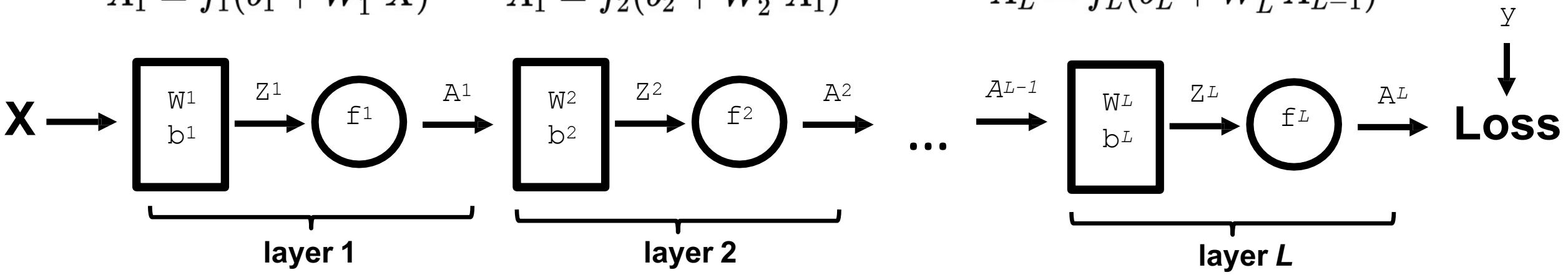
Task	Activation	Loss
Regression (penalize large errors)	**Linear (ReLU, Leaky ReLU, etc)**	$\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
Regression (penalize error linearly)	**Linear (ReLU, Leaky ReLU, etc)**	$\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}\left\lvert y^{(i)} - \hat{y}^{(i)}\right\rvert$
Classification (binary)	**Sigmoid, tanh**	$\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} -y^{(i)}\log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)$
Classification (multi-class)	**Softmax**	$\mathcal{L}_{\text{CCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N}\sum_{j=1}^{K} y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right)$
Generative	**Linear (ReLU, Leaky ReLU, etc)**	$\mathcal{L}_{\text{minimax}}(\boldsymbol{G}, \boldsymbol{D}) = E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$

Other considerations: gradient intensity, computational activation cost, exploding/vanishing gradients, depth of network (linear is useless)

Neural networks: training with backpropagation

$$W_l = m^l \times n^l$$
$$b = n^l \times 1$$

$$A_1 = f_1(b_1 + W_1^T X) \qquad A_1 = f_2(b_2 + W_2^T A_1) \qquad A_L = f_L(b_L + W_L^T A_{L-1})$$

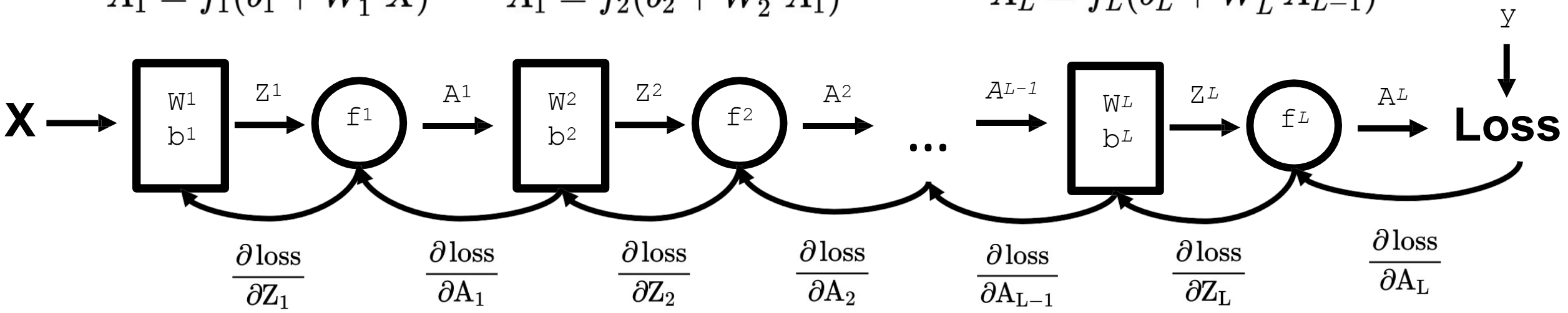


Inspired by 6.036 lecture notes (Leslie Kaebling)

Neural networks: training with backpropagation

$$W_l = m^l \times n^l$$
$$b = n^l \times 1$$

$$A_1 = f_1(b_1 + W_1^T X) \qquad A_1 = f_2(b_2 + W_2^T A_1) \qquad A_L = f_L(b_L + W_L^T A_{L-1})$$



Inspired by 6.036 lecture notes (Leslie Kaebling)

Neural networks: training with backpropagation

So let's use the following shorthand from the previous figure,

$$NN(x; W) = A_L$$

First, let's break down how the loss depends on the final layer,

$$\frac{\partial \text{loss}}{\partial \mathrm{W_L}} = \frac{\partial \text{loss}}{\partial \mathrm{A_L}} \cdot \frac{\partial \mathrm{A_L}}{\partial \mathrm{Z_L}} \cdot \frac{\partial \mathrm{Z_L}}{\partial \mathrm{W_L}}$$

Since,

$$\frac{\partial \mathrm{Z_L}}{\partial \mathrm{W_L}} = \frac{\partial}{\partial \mathrm{W_L}}(W^T A_{L-1}) = A_{L-1}$$

We can re-write the equation as,

$$\frac{\partial \text{loss}}{\partial \mathrm{W_L}} = A_{L-1} \frac{\partial \text{loss}}{\partial \mathrm{Z_L}}$$

Since we have the outputs of every layer, all we need to compute for the gradient of the last layer with respect to the weights is the gradient of the loss with respect to the pre-activation output.

Now, to propagate through the whole network, we can keep applying the chain rule until the first layer of the network,

$$\frac{\partial \text{loss}}{\partial \mathrm{Z_1}} = \frac{\partial \text{loss}}{\partial \mathrm{A_L}} \cdot \frac{\partial \mathrm{A_L}}{\partial \mathrm{Z_L}} \cdot \frac{\partial \mathrm{Z_L}}{\partial \mathrm{A_{L-1}}} \cdot \frac{\partial \mathrm{A_{L-1}}}{\partial \mathrm{Z_{L-1}}} \cdots \frac{\partial \mathrm{A_2}}{\partial \mathrm{Z_2}} \cdot \frac{\partial \mathrm{Z_2}}{\partial \mathrm{A_1}} \cdot \frac{\partial \mathrm{A_1}}{\partial \mathrm{Z_1}}$$

If you spend a few minutes looking at matrix dimensions, it becomes clear that this is an informal derivation. Here are the dimensions to think about:

$$\frac{\partial \text{loss}}{\partial A^L} \text{ is } n^L \times 1 \qquad W_{L-1} = \frac{\partial \mathrm{Z_L}}{\partial A_{L-1}} \text{ is } m^L \times n^L \qquad \frac{\partial \mathrm{A_L}}{\partial Z_L} \text{ is } n^L \times n^L$$

The equation with the correct dimensions for matrix multiplication,

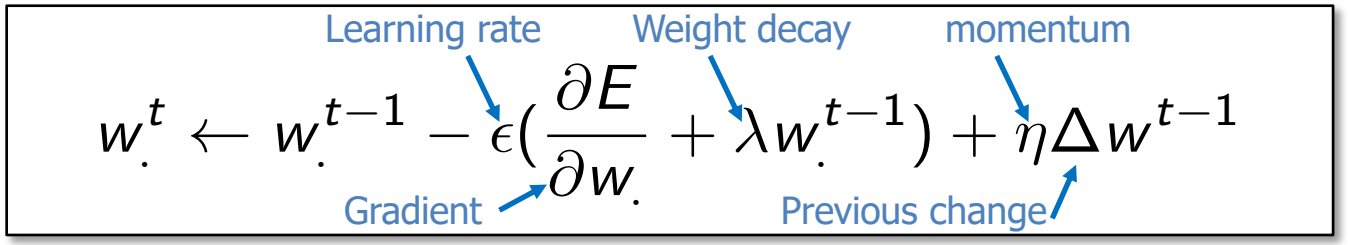
$$\frac{\partial \text{loss}}{\partial Z_l} = \frac{\partial A_l}{\partial Z_l} \cdot W_{l+1} \cdot \frac{\partial A_{l+1}}{\partial Z_{l+1}} \cdot \ldots W_{L-1} \cdot \frac{\partial A_{L-1}}{\partial Z_{L-1}} \cdot W_L \cdot \frac{\partial A_L}{\partial Z_L} \cdot \frac{\partial loss}{A_L}$$

On your own time!

Inspired by 6.036 lecture notes (Leslie Kaebling)

Neural networks: gradient descent

Gradient-based learning: use derivative to update weights



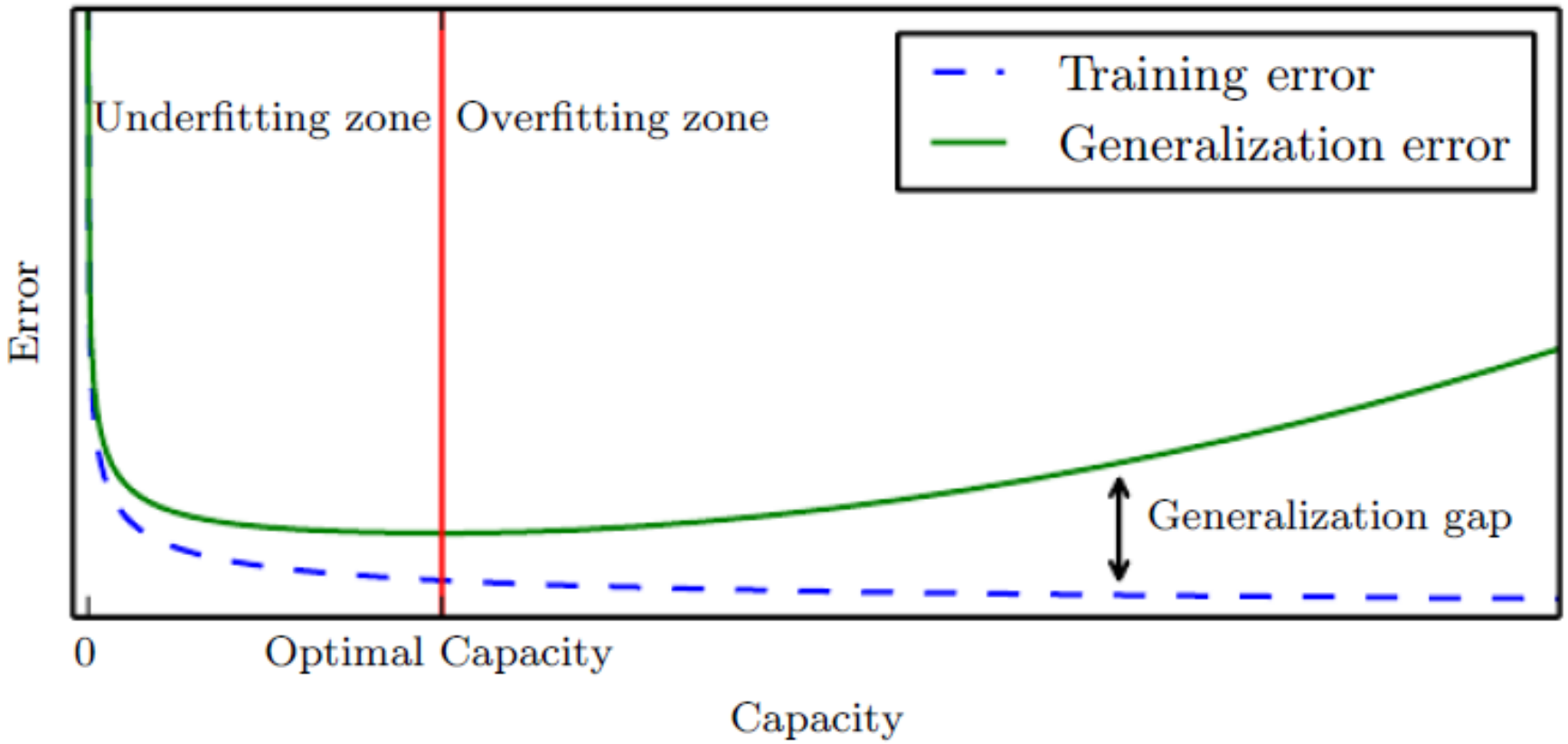
where

- Gradient descent: **∂E/∂w** = partial derivative of error E wrt w
- ϵ = **learning rate** (e.g. <0.1), needed to not overshoot the optimal solution
- λ = **weight decay**, penalizes large weights to prevent overfitting
- η = **momentum**, based on magnitude+sign of previous update (Δw^{t-1}); when direction of update is consistent ➔ faster convergence

Neural networks: gradient descent

Neural networks: regularization

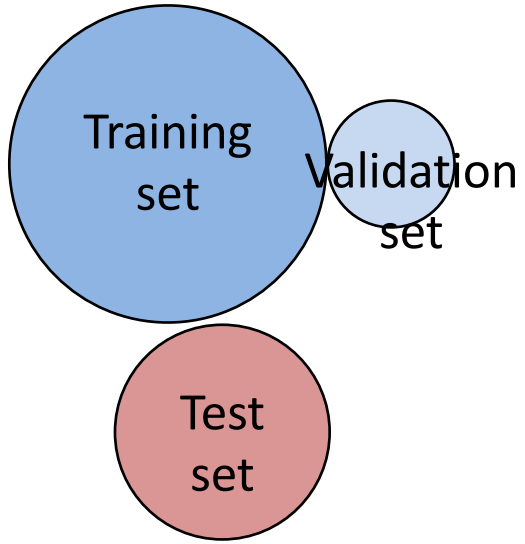
We need to control model complexity for good generalization

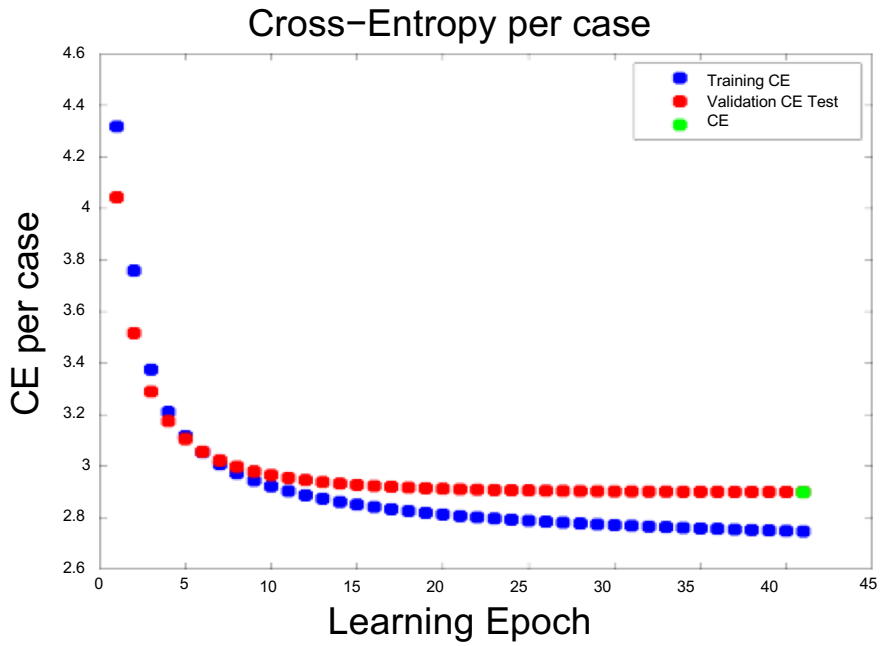


Neural networks: regularization via validation set, early stopping

Leave out small “validation set”

- not used to train the model
- used to evaluate model at each epoch/iteration (VCE, Validation cross-entropy)
- Stop when VCE increases, prevent overfitting





Neural networks: regularization

Objective function

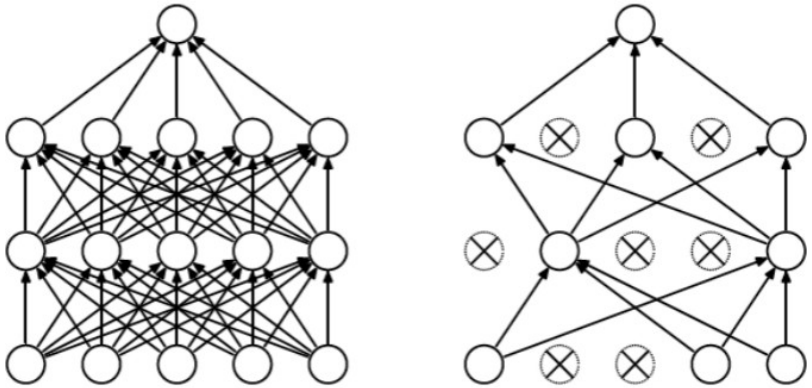
$$\mathrm{J}(W,b) = \frac{1}{\mathrm{n}} \sum_{\mathrm{i}=1}^{\mathrm{n}} \left(W^{\mathrm{T}} x^{(i)} + b - \mathrm{y}^{(i)} \right)^2$$

Objective function with ridge regularization

Penalize

$$\mathrm{J}(W,b) = \frac{1}{\mathrm{n}} \sum_{\mathrm{i}=1}^{\mathrm{n}} \left(W^{\mathrm{T}} x^{(i)} + b - \mathrm{y}^{(i)} \right)^2 + \lambda \parallel W \parallel^2$$

Dropout



(a) Standard Neural Net (b) After applying dropout.

$$\mathbf{a}^{\ell} = \mathbf{f}\left(z^{\ell}\right) * \mathbf{d}^{\ell}$$

R01 Outline

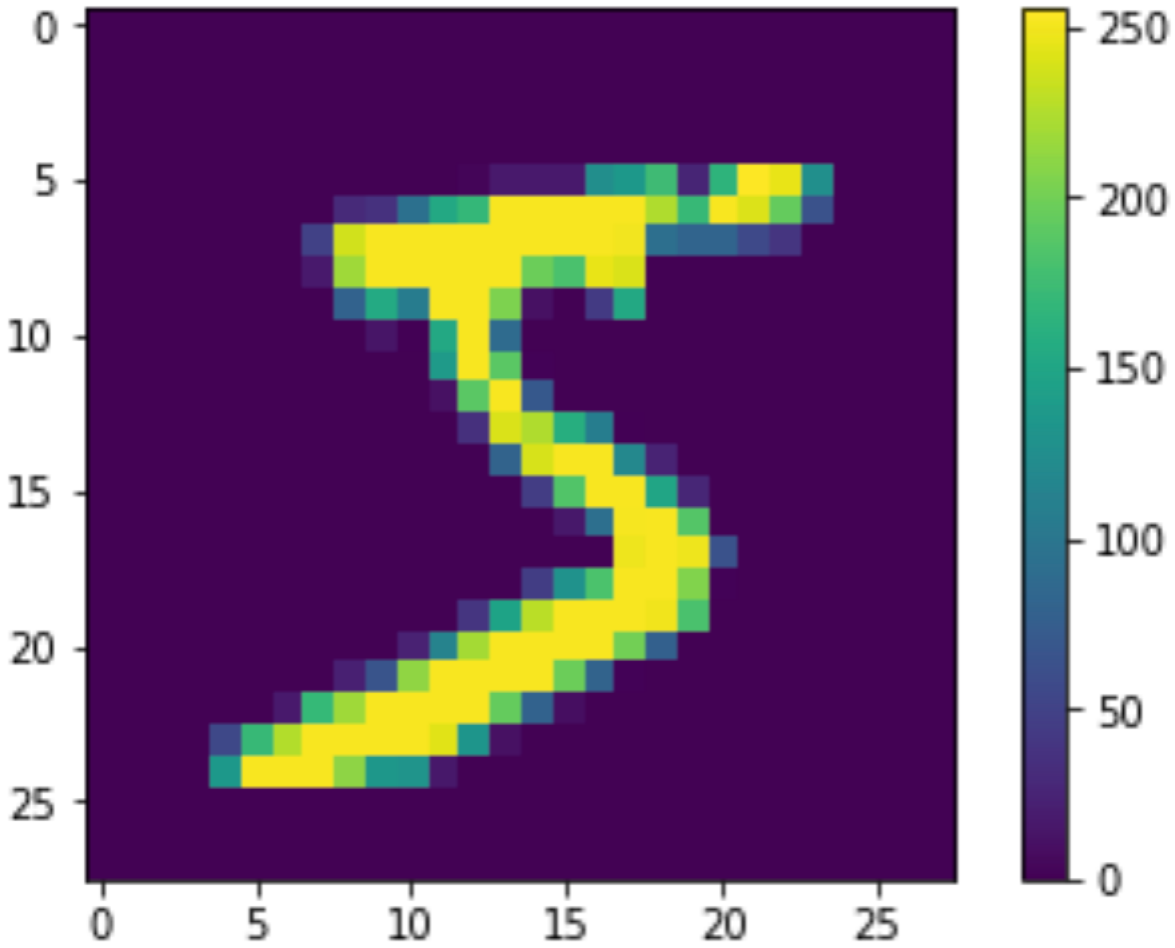
A. What can you do with ML?

B. Basics of machine learning

C. Neural networks

D. Brief preview of pset 1

PS1: TensorFlow Warm Up




```
ground truth: 5
```

PS1: Data

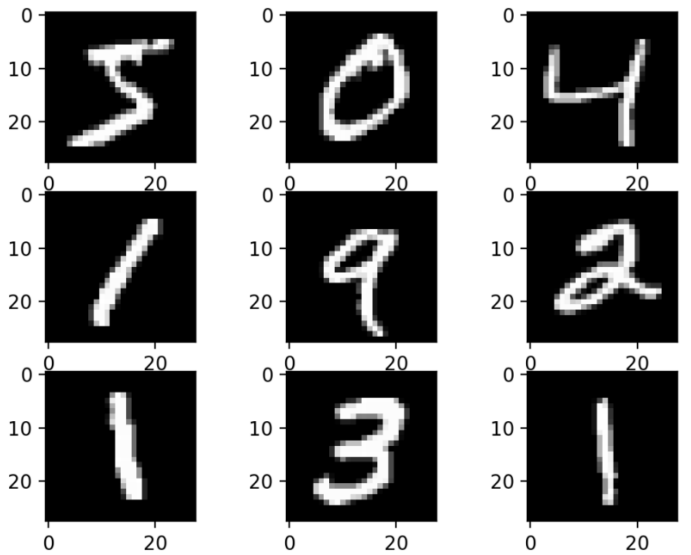
Problem Set 1

input space:
$X = \{0, 1, \ldots, 255\}^{28\times28}$

after rescaling:
$X^{I} = [0, 1]^{28\times28}$

after flattening:
$X^{II} = [0, 1]^{784}$

Classification



PS1: Data

Problem Set 1

input space:
$X = \{0, 1, \ldots, 255\}^{28\times 28}$

after rescaling:
$X^{I} = [0, 1]^{28\times 28}$

after flattening:
$X^{II} = [0, 1]^{784}$

integer-encoded label space:
$Y_i = \{0, 1, \ldots, 9\}$

one-hot-encoded label space:
$Y_h = [0, 1]^{10}$

$$\overbrace{\boldsymbol{x}^{(i)} \in \mathcal{X}}$$

$$\begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ 28 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & 28 \end{array} \\ \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,28} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,28} \\ \vdots & \vdots & \ddots & \vdots \\ x_{28,1} & x_{28,2} & \cdots & x_{28,28} \end{bmatrix} \end{array}$$

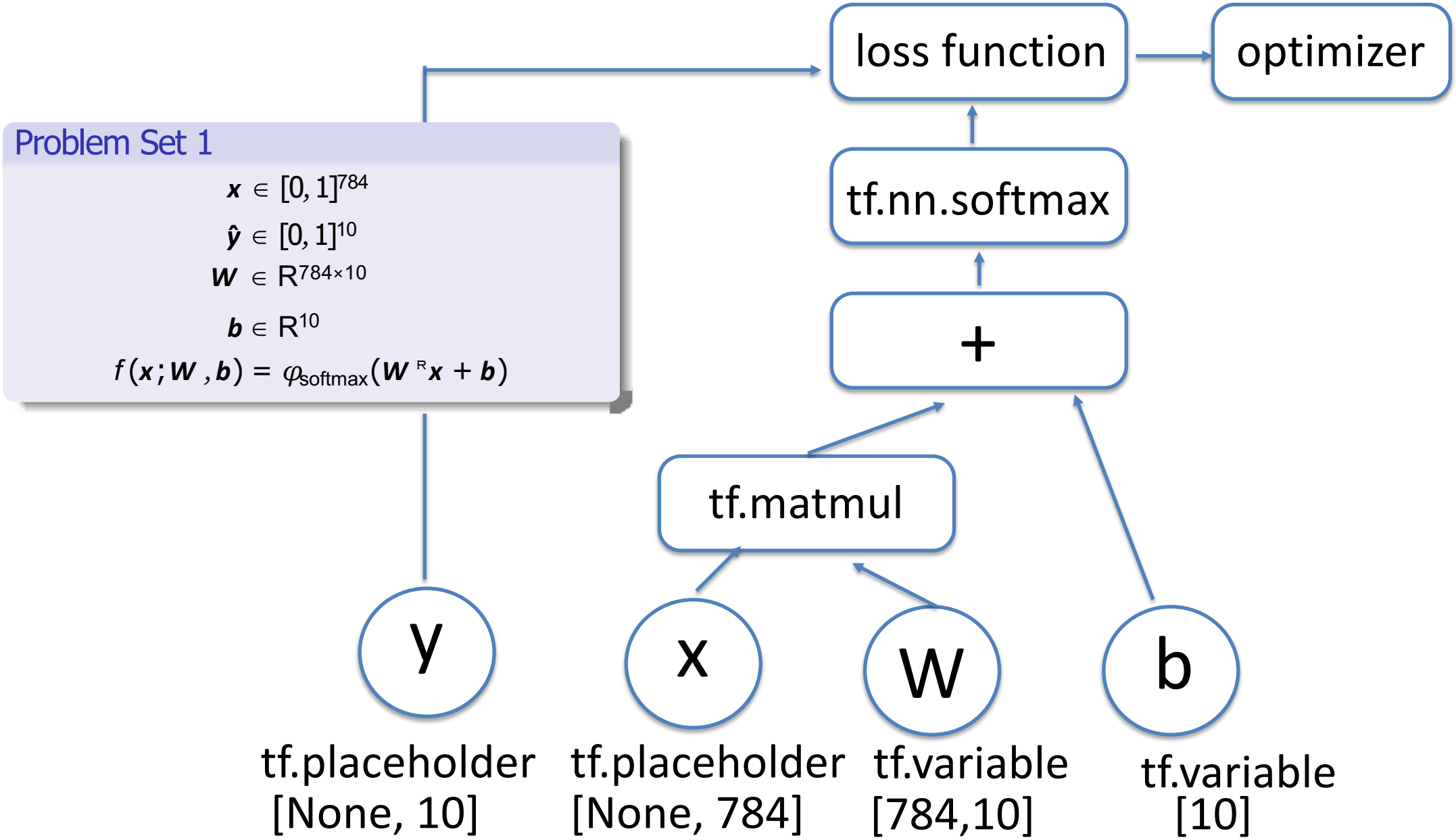
$$\overbrace{\boldsymbol{y}^{(i)} \in \mathcal{Y}_h}$$

$$\begin{array}{cccc} 1 & 2 & \cdots & 10 \end{array}$$
$$\begin{bmatrix} y_1 & y_2 & \cdots & y_{10} \end{bmatrix}$$

One hot encoding turns 1, 2, 3 into

1	0	0
0	1	0
0	0	1

PS1: Structure



PS1: Gradient of loss with respect to logits

i goes from 0 to # of training examples

z is one of 10 digits

$$z^i = W^T x^i + b$$

$$L_i = -\log(p_k^i) \, where \, k = y^i, \quad p_k^i = \frac{e^{z_k^i}}{\sum_j e^{z_j^i}}$$

$$\frac{\partial L_i}{\partial z_j} = p_j^i - \mathbf{1}(\mathbf{y_i} = \mathbf{j}) \; Update \; weights \; for \; j$$

$$p^i = [0.6,0.3,0.1] \, then \, gradient \rightarrow \quad [0.6, -0.4,0.1]$$

Correct Label

http://cs231n.github.io/neural-networks-case-study/

PS1: Gradient of loss with respect to logits

i goes from 0 to # of training examples

z is one of 10 digits

$$z^i = W^T x^i + b$$

L is loss

$$L_i = -\log(p_k^i) \; where \; k = y^i, \quad p_k^i = \frac{e^{z_k^i}}{\sum_j e^{z_j^i}}$$

y is actual labels

j goes from 0 to 9

$$\frac{\partial L_i}{\partial z_j} = p_j^i - \mathbf{1}(\mathbf{y_i} = \mathbf{j}) \; \textit{Update weights for j}$$

$$p^i = [0.6, 0.3, 0.1] \; then \; gradient \rightarrow \quad [0.6, -0.4, 0.1]$$

Correct Label

http://cs231n.github.io/neural-networks-case-study/

PS1: Gradient of loss with respect to logits

i goes from 0 to # of training examples

z is one of 10 digits

$$z^i = W^T x^i + b$$

L is loss

$$L_i = -\log(p_k^i) \; where \; k = y^i, \quad p_k^i = \frac{e^{z_k^i}}{\sum_j e^{z_j^i}}$$

y is actual labels

j goes from 0 to 9

$$\frac{\partial L_i}{\partial z_j} = p_j^i - \mathbf{1}(\mathbf{y_i} = \mathbf{j}) \; \textit{Update weights for j}$$

if y_i (the actual label) is j (a digit), then $dL_i/z_j = p^i_j$

$$p^i = [0.6, 0.3, 0.1] \; then \; gradient \rightarrow \quad [0.6, -0.4, 0.1]$$

Correct Label

http://cs231n.github.io/neural-networks-case-study/

PS1: Implementation

```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.mul(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)

# Gradient descent
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Next week

More neural network review

Convolutional neural networks

Recurrent neural networks

R01 Outline

A. What can you do with ML?

B. Basics of machine learning

C. Neural networks

D. Brief preview of pset 1

E. BONUS CONTENT if time!

BONUS CONTENT!

Non-parametric models

Gradient descent – batch vs stochastic

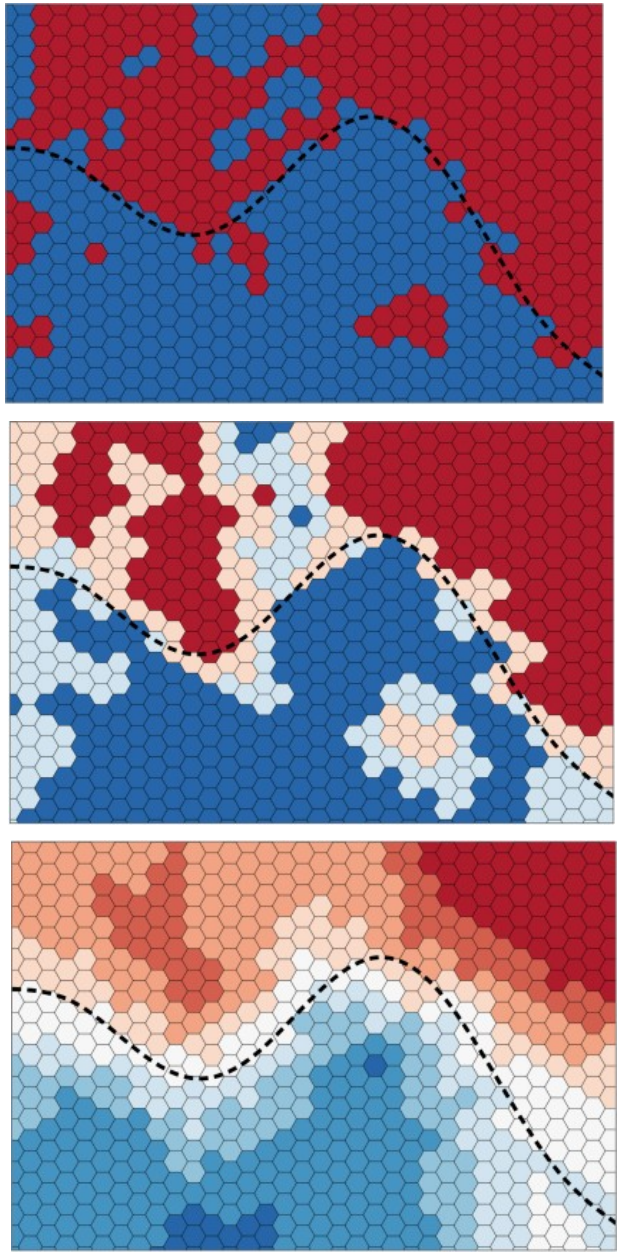
Momentum

Adam

Lecture question: “how do we actually calculate these derivatives?”
Answer: automatic differentiation

Basics of machine learning: non-parametric models

```
1 D = {(X$, Y$), (X(, Y(), ..., (X*, Y*)}
2 function knn(k, dist, D{tra/*}, D{t01t}):
3 votes = []
4 for i = 1 to len(D{t01t})
5     d = []
6     for j = 1 to len(D{tra/*})
7         d[j] = dist(x/, x4)
8     d = argsort(d)
9     votes[i] = most_common(set(labels[d]))
```



Neural networks: gradient descent - batch gradient update

Gradient of objective *J* with respect to parameter vector W

$$\nabla_W \mathrm{J} = \begin{bmatrix} \partial \mathrm{J}/\partial W_1 \\ \vdots \\ \partial \mathrm{J}/\partial W_\mathrm{m} \end{bmatrix}$$

Batch gradient update

$$W := W - \eta \sum_{i=1}^{n} \nabla_W \mathrm{J}\left(h\left(x^{(i)}; W\right), y^{(i)}\right)$$

Pseudocode for gradient update algorithm

```
1 function gradient_update(W/*/t, η, J, ϵ):
2 W^0 = W_init
3 t = 0
4 while |J(W^t) − J(W^t<$)| > ϵ
5     t = t+1
6     W^t = W^t<$ − η ∇@(J)
```

This is an arbitrary update criteria

Neural networks: gradient descent - stochastic gradient descent

Stochastic gradient update (per randomly sampled training example):

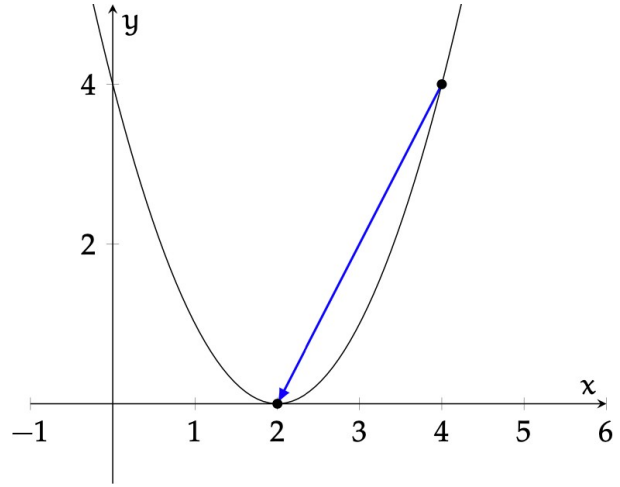
$$W := W - \eta \nabla_W \mathrm{J}\left(h\left(x^{(i)}; W\right), y^{(i)}\right)$$

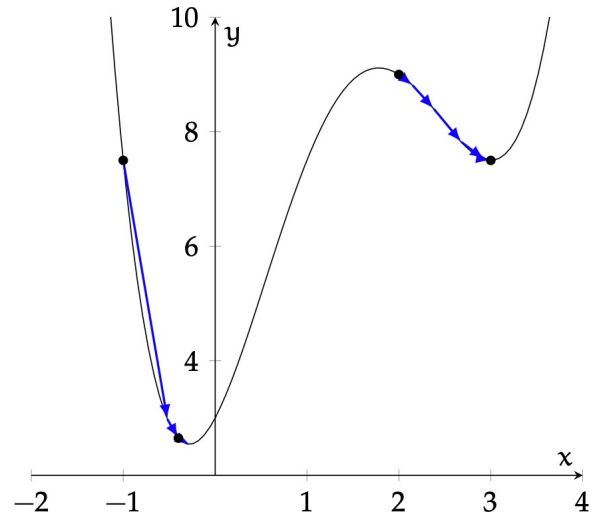
Pseudocode for stochastic gradient update algorithm

```
function sgd(W/*/t, η, J, T, ϵ):
W^0 = W_init
for t = 1 to T
    randomly select i ∈ {1,2,...,n}
    W^t = W^t<$ − η(t) ∇@(J)
```

Mini-batch gradient descent

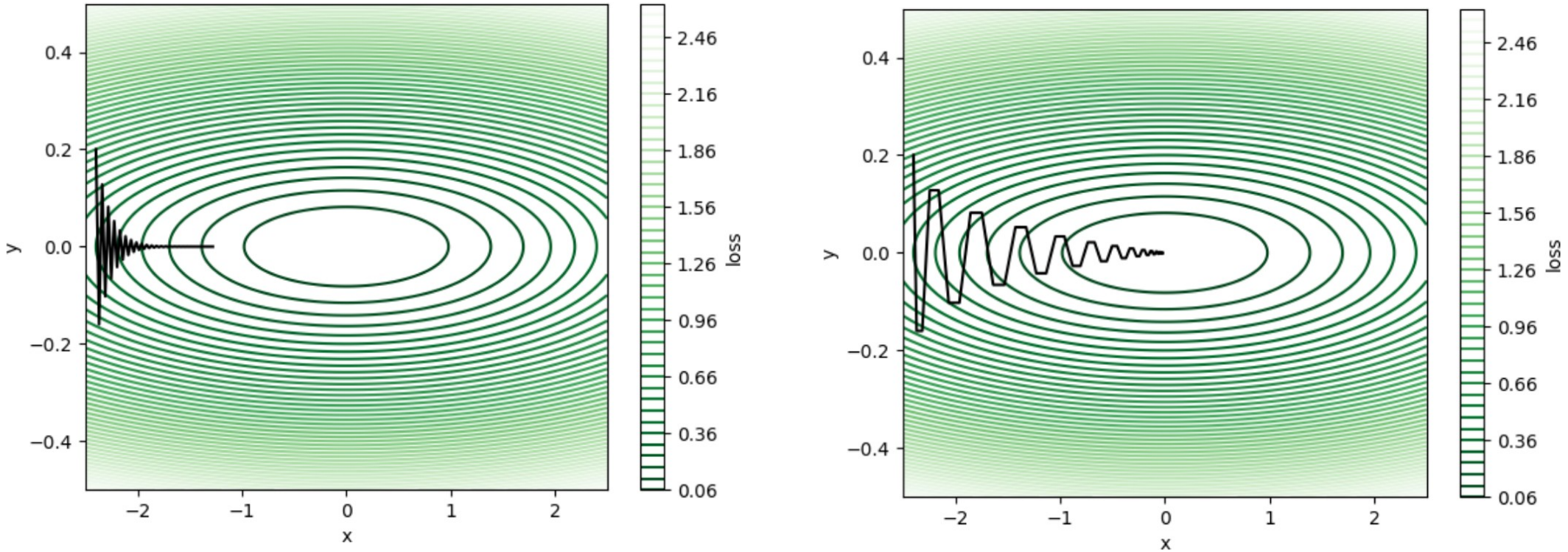
$$W := W - \eta \sum_{i=1}^{k} \nabla_W \mathcal{L}\left(h\left(x^{(i)}; W\right), y^{(i)}\right)$$





Optimization: momentum

$$v_{k+1} = \beta v_k + \nabla f(w_k)$$
$$w_{k+1} = w_k - \eta v_{k+1}$$



Nesterov inequality? How can we make momentum better?

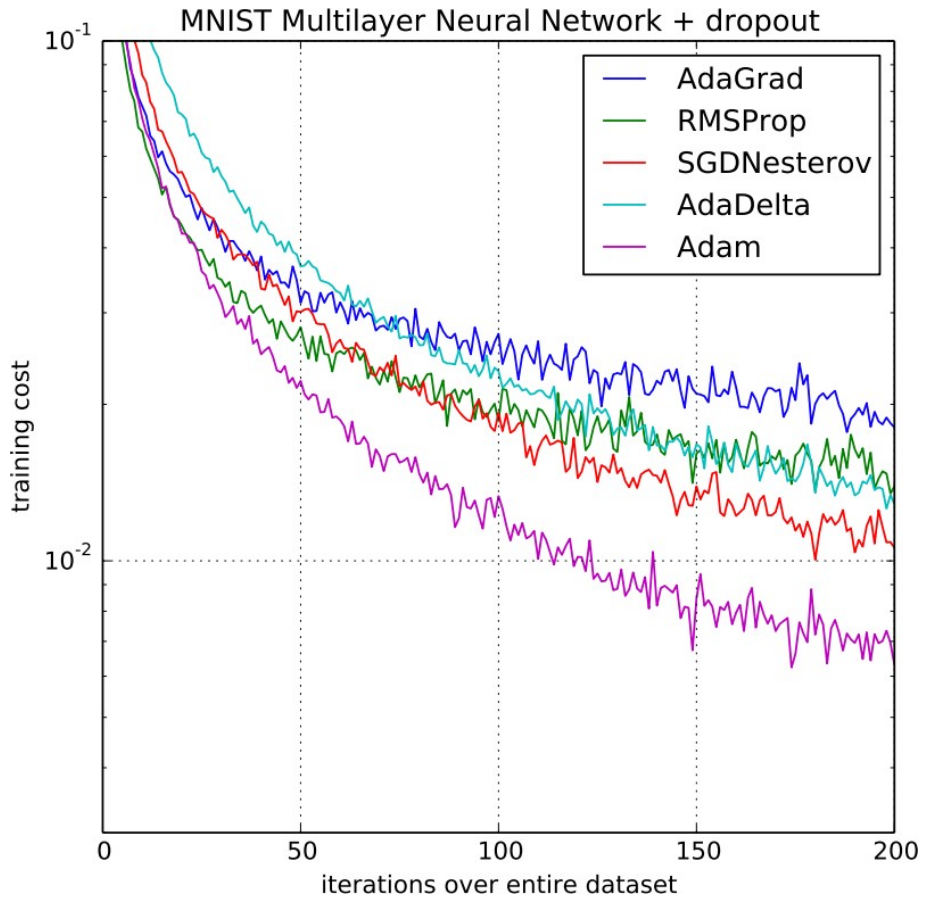
$$v_{k+1} = \beta v_k + \nabla f(w_k + \beta v_k)$$
$$w_{k+1} = w_k - \eta v_{k+1}$$

Goh, "Why Momentum Really Works", Distill, 2017. http://doi.org/10.23915/distill.00006

Optimization: adam

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)



https://arxiv.org/pdf/1412.6980.pdf

Neural networks: automatic differentiation

Dual numbers

Augment a standard Taylor series (numerical differentiation), with a “dual number”,

$$f(a+\epsilon) = f(a) + \frac{f'(a)}{1!}\epsilon + \frac{f''(a)}{2!}\epsilon^2 + \cdots + \frac{f^n}{n!}\epsilon^n$$

Because ”dual numbers” have the (manufactured) property,

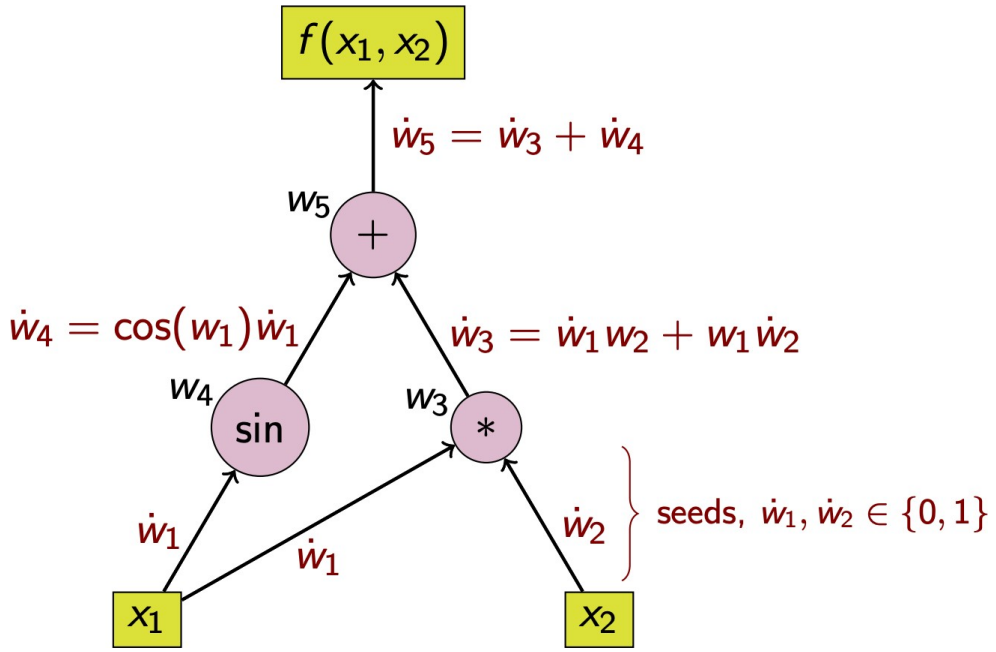
$$\epsilon^2 = 0$$

The Taylor Series simplifies to,

$$f(a+\epsilon) = f(a) + f'(a)\epsilon$$

Which recovers the function output as well as the first derivative.

Forward automatic differentiation



End result

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

https://www.robots.ox.ac.uk/~tvg/publications/talks/autodiff.pdf