Introduction to machine learning

Recitation 2

MIT - 6.802 / 6.874 / 20.390 / 20.490 / HST.506 - Spring 2019

Sachit Saksena

Recap of recitation 1

Introduction to **Tensorflow**

Local conda and **jupyter notebook** setup

Google Colabratory setup

Setting up **Google Cloud** with **Datalab**

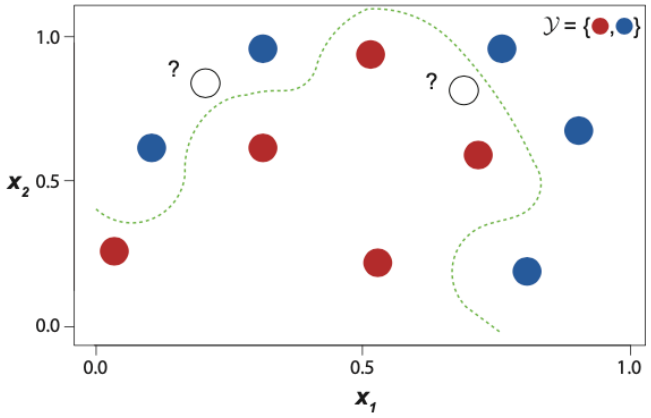
Everyone set up for PS1?

Basics of machine learning: steps

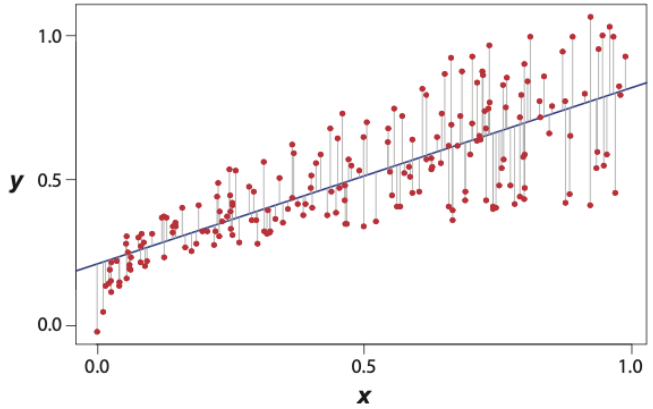
I. Get data
II. Identify the space of possible solutions
III. Formulate an objective
IV. Choose algorithm
V. Train (loss)
VI. Validate results (metrics)

Basics of machine learning: tasks

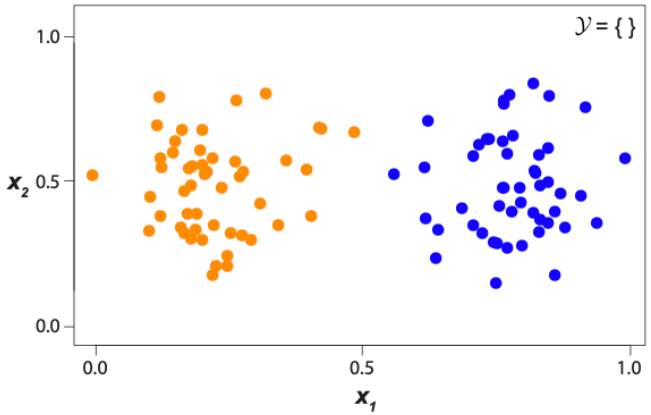
Classification



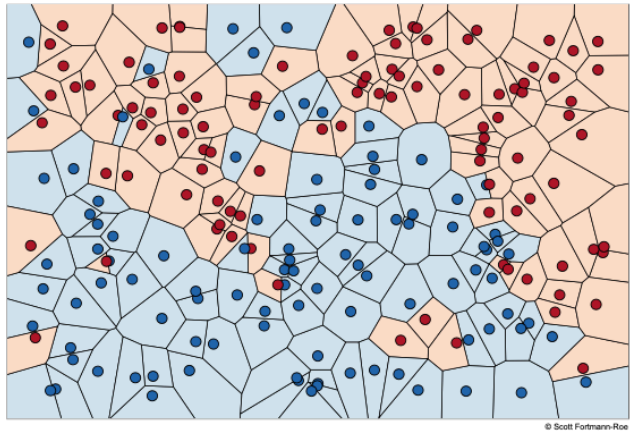
Regression



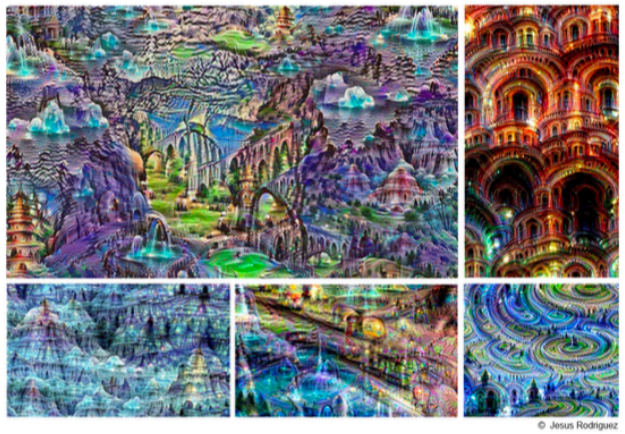
Unsupervised learning



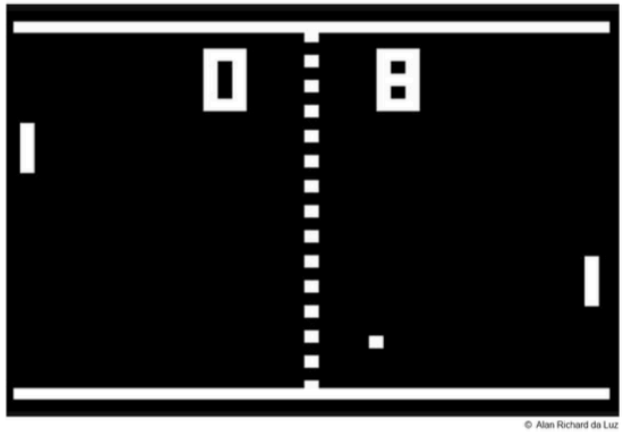
Non-parametric models



Generative models



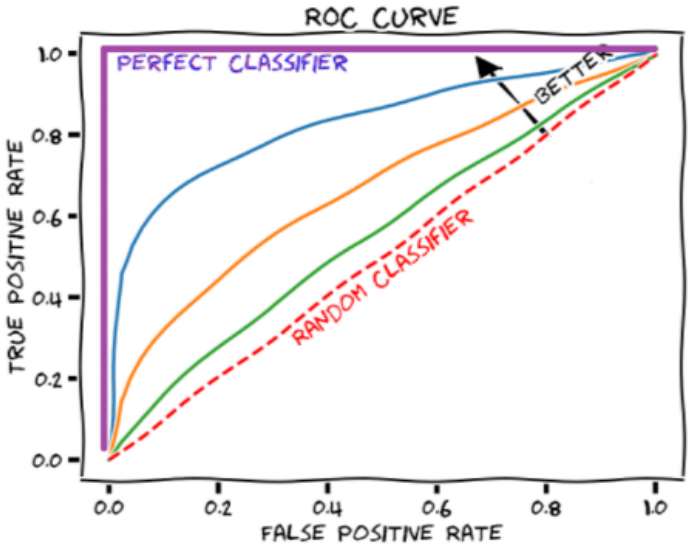
Reinforcement learning

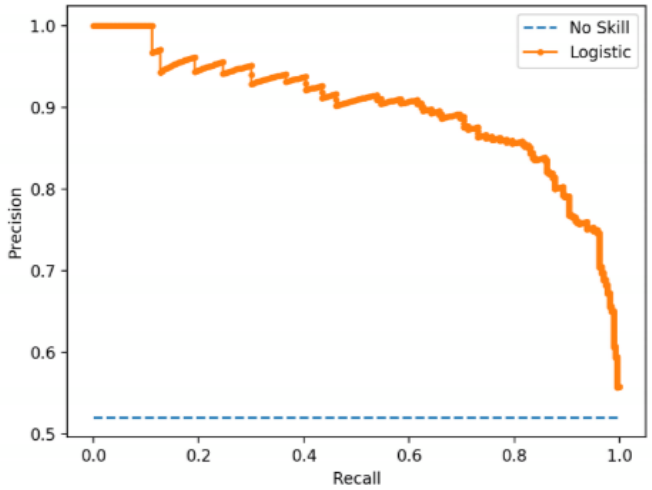


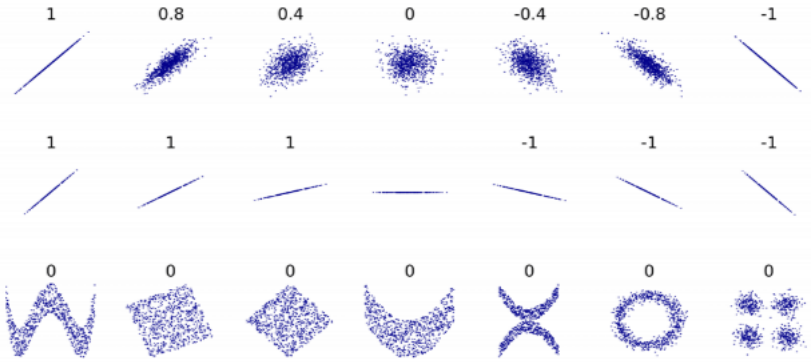
Basics of machine learning: loss

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)	$\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}\left\lvert y^{(i)} - \hat{y}^{(i)}\right\rvert$
Classification (binary)	$\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} -y^{(i)}\log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)$
Classification (multi-class)	$\mathcal{L}_{\text{CCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N}\sum_{j=1}^{K} y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right)$
Generative	$\mathcal{L}_{\text{minimax}}(\boldsymbol{G}, \boldsymbol{D}) = E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$

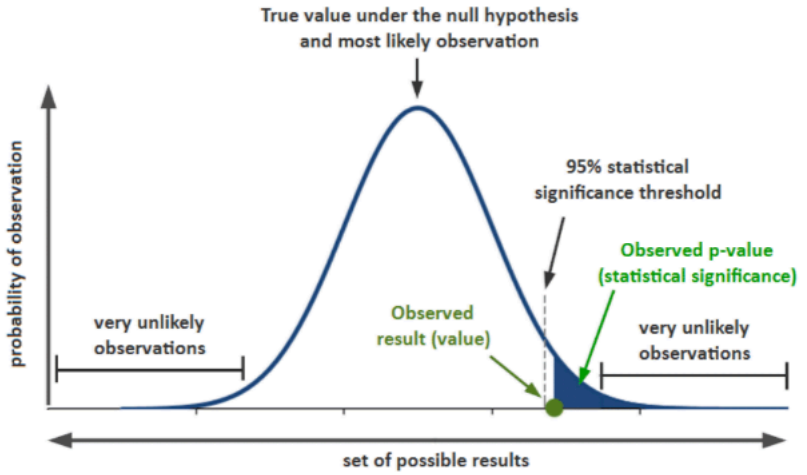
Basics of machine learning: metrics



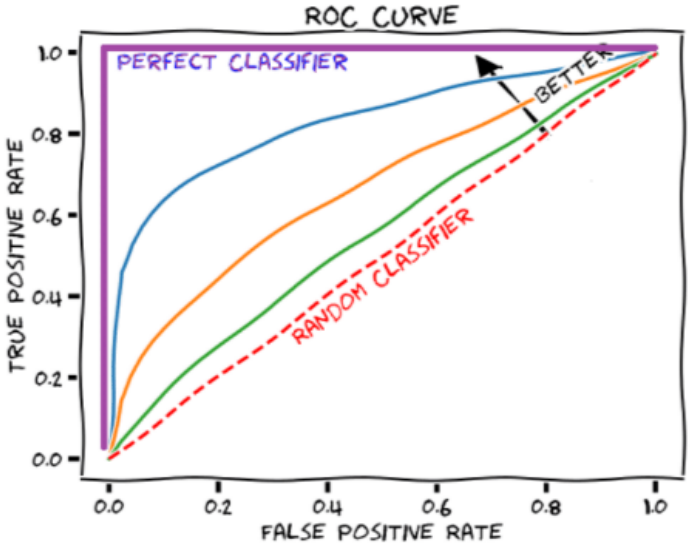


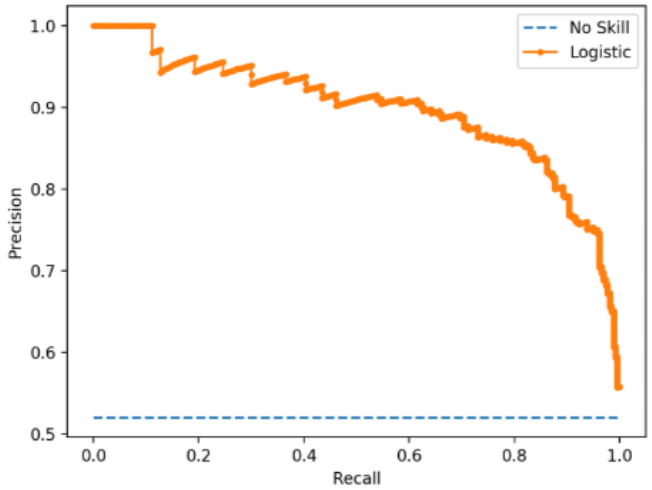


		True condition		
	Total population	Condition positive	Condition negative	Accuracy = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Precision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



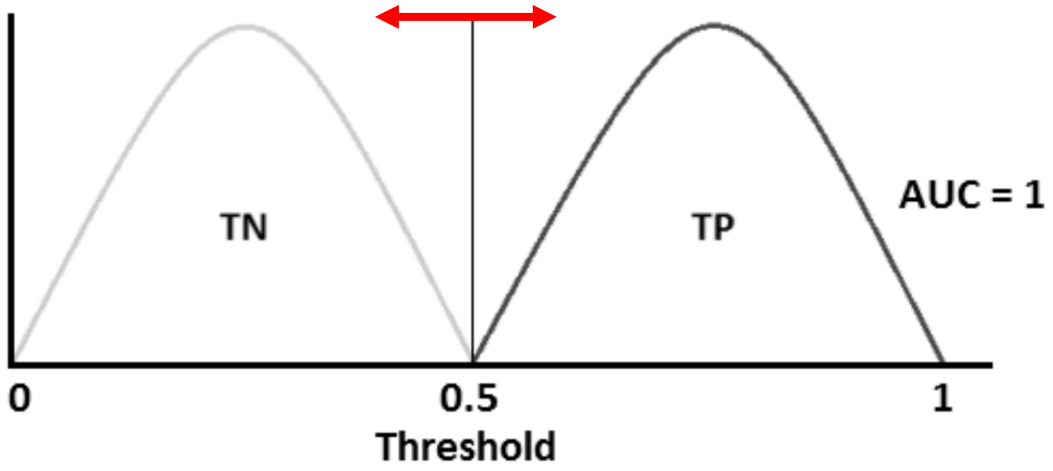
Basics of machine learning: metrics

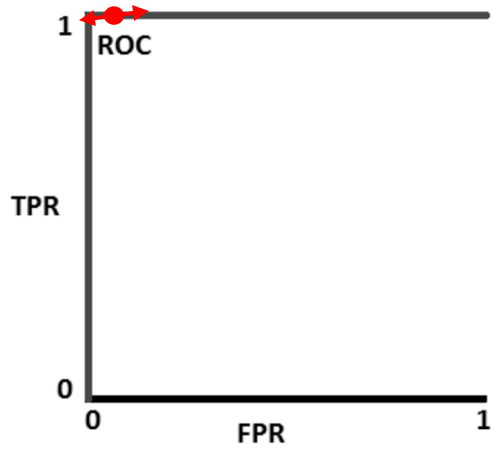


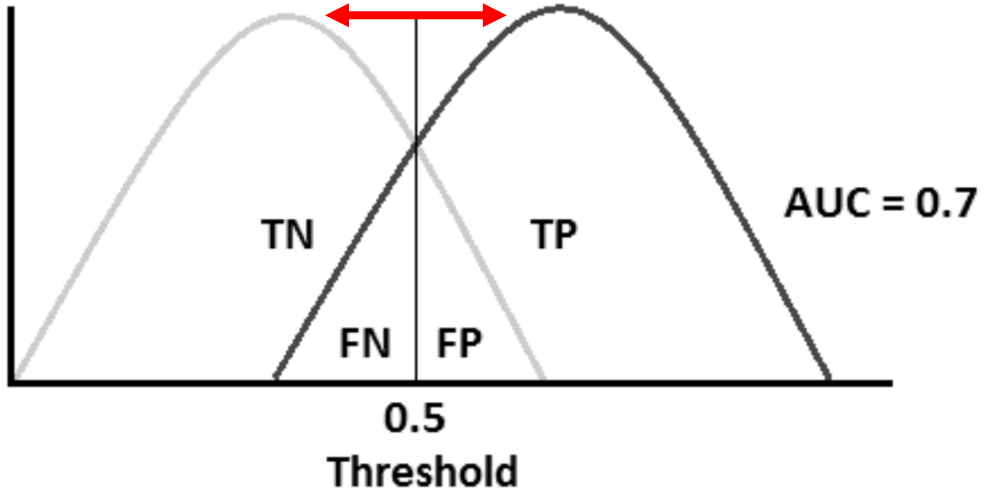


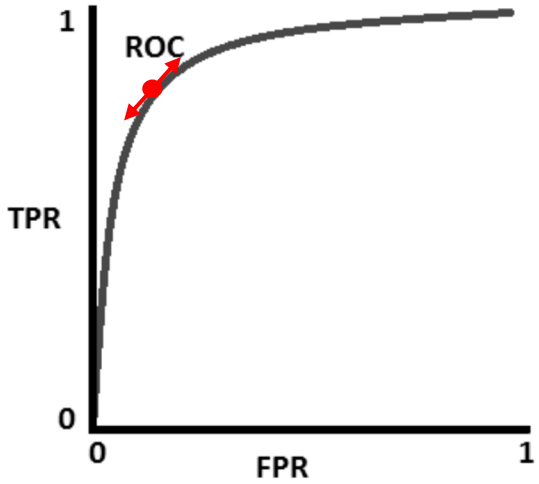
		True condition		
	Total population	Condition positive	Condition negative	Accuracy = Σ True positive + Σ True negative / Σ Total population
Predicted condition	Predicted condition positive	True positive, Power	False positive, Type I error	Precision = Σ True positive / Σ Predicted condition positive
	Predicted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

Basics of machine learning: ROC









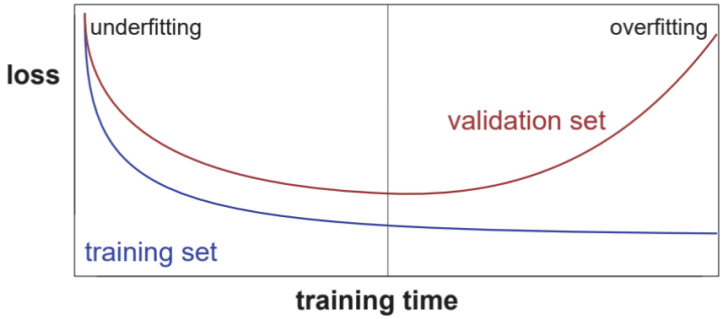
https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

Basics of machine learning: data

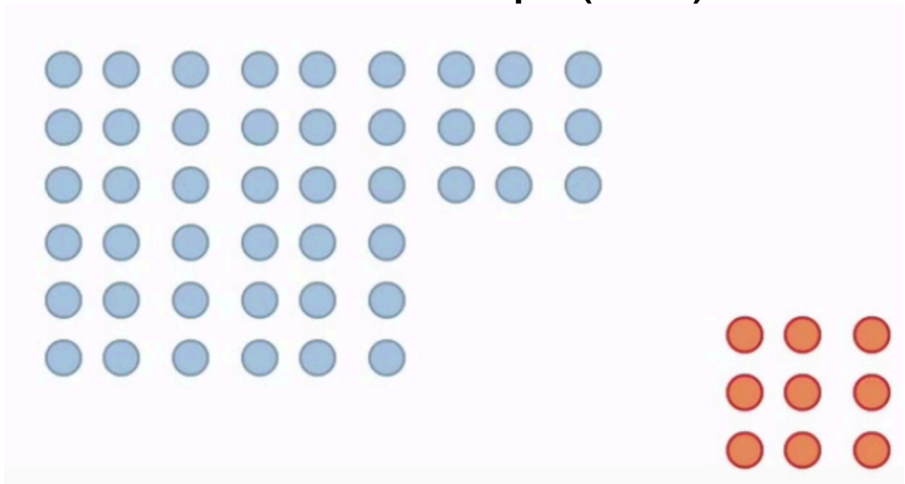
Training set ($\mathcal{S}_{\text{training}}$):
- set of examples used for learning
- usually 60 - 80 % of the data

Validation set ($\mathcal{S}_{\text{validation}}$):
- set of examples used to tune the model hyperparameters
- usually 10 - 20 % of the data

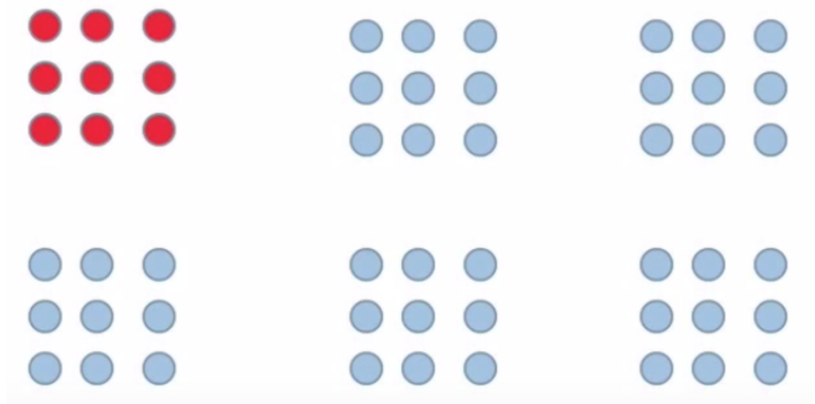
Test set ($\mathcal{S}_{\text{test}}$):
- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30 % of the data



Train—test split (1-fold)



Cross-validation (6-fold)



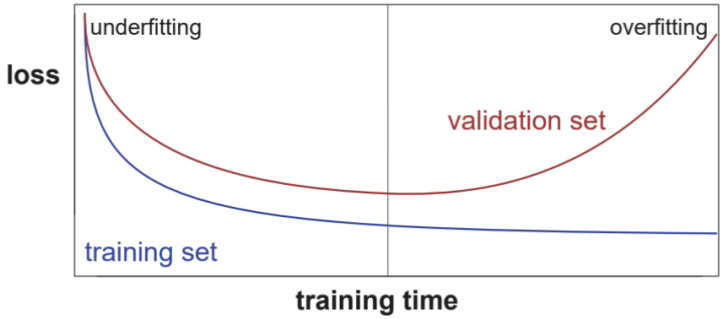
https://towardsdatascience.com/cross-validation-code-visualization-kind-of-fun-b9741baea1f8

Basics of machine learning: data

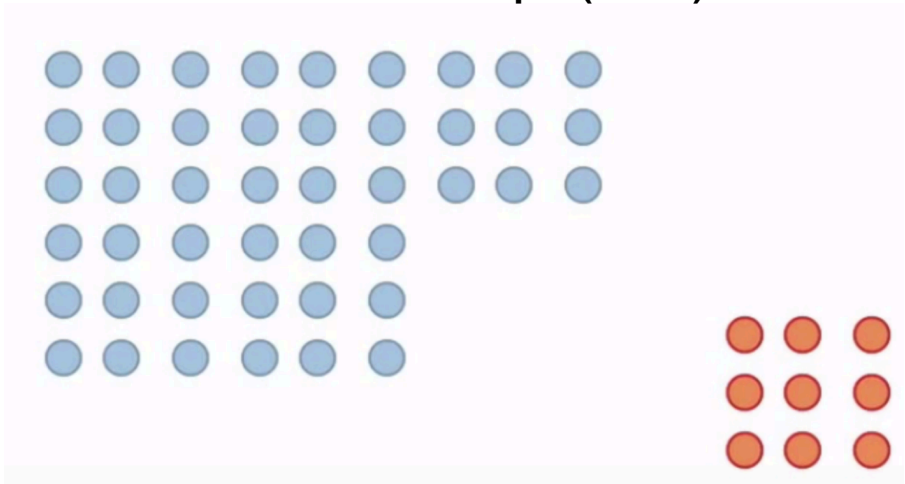
Training set ($\mathcal{S}_{\text{training}}$):
- set of examples used for learning
- usually 60 - 80 % of the data

Validation set ($\mathcal{S}_{\text{validation}}$):
- set of examples used to tune the model hyperparameters
- usually 10 - 20 % of the data

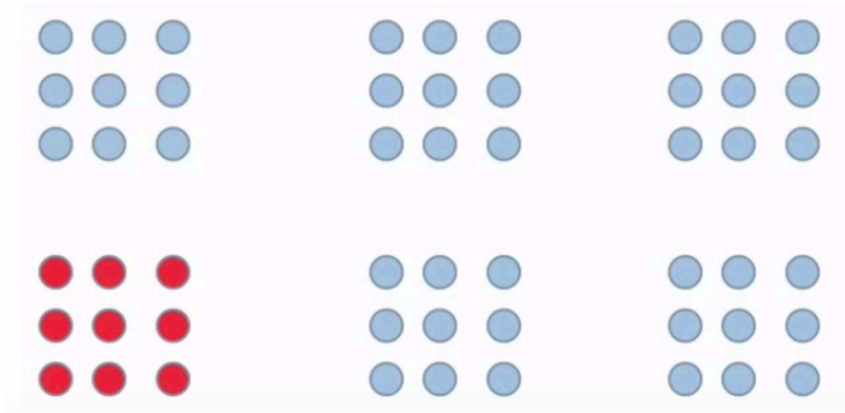
Test set ($\mathcal{S}_{\text{test}}$):
- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30 % of the data



Train—test split (1-fold)



Cross-validation (6-fold)



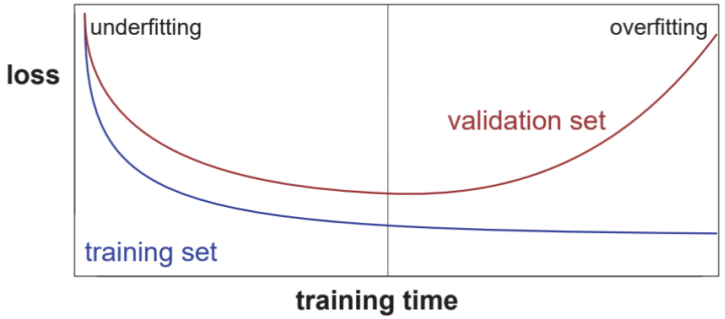
https://towardsdatascience.com/cross-validation-code-visualization-kind-of-fun-b9741baea1f8

Basics of machine learning: data

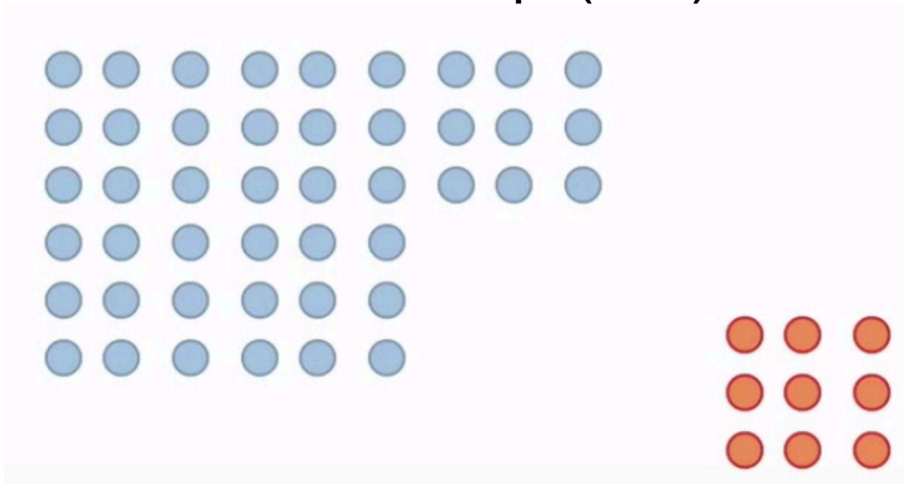
Training set ($\mathcal{S}_{\text{training}}$):
- set of examples used for learning
- usually 60 - 80 % of the data

Validation set ($\mathcal{S}_{\text{validation}}$):
- set of examples used to tune the model hyperparameters
- usually 10 - 20 % of the data

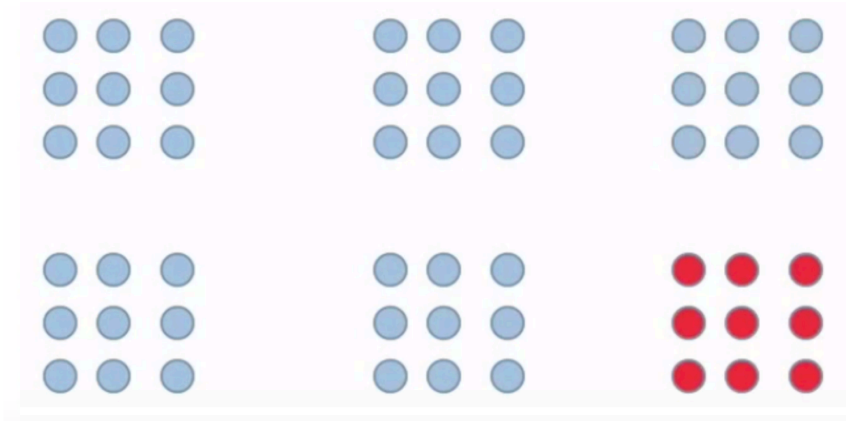
Test set ($\mathcal{S}_{\text{test}}$):
- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30 % of the data



Train—test split (1-fold)



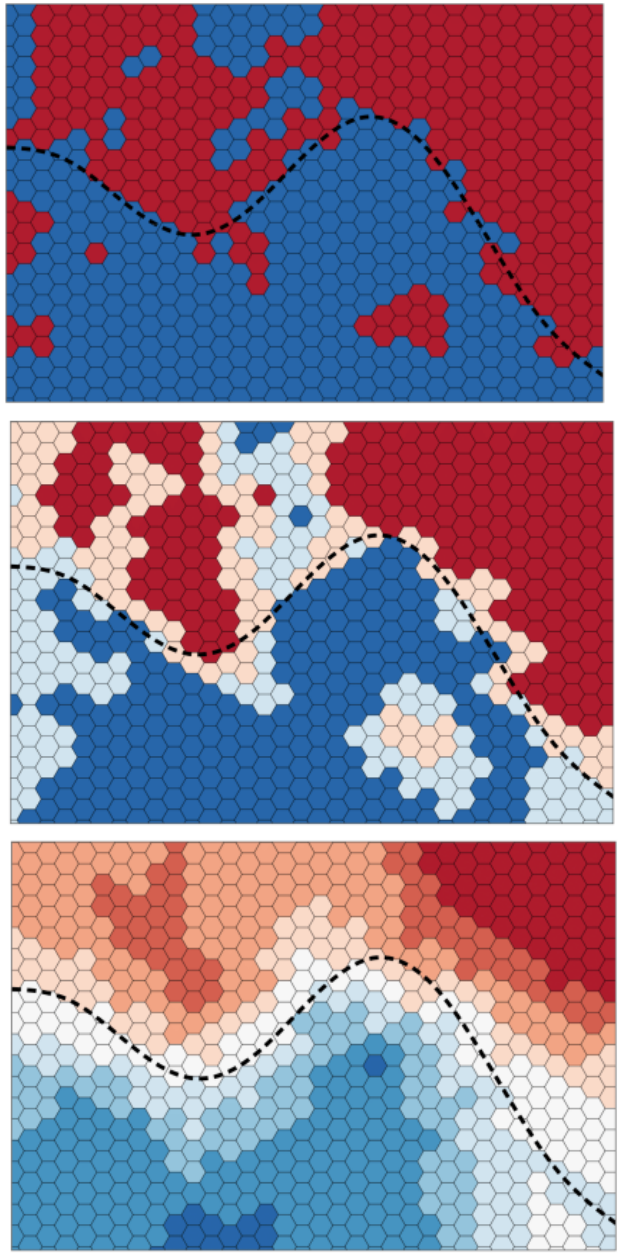
Cross-validation (6-fold)



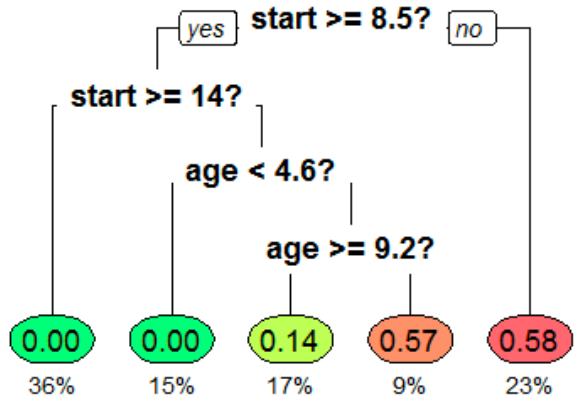
https://towardsdatascience.com/cross-validation-code-visualization-kind-of-fun-b9741baea1f8

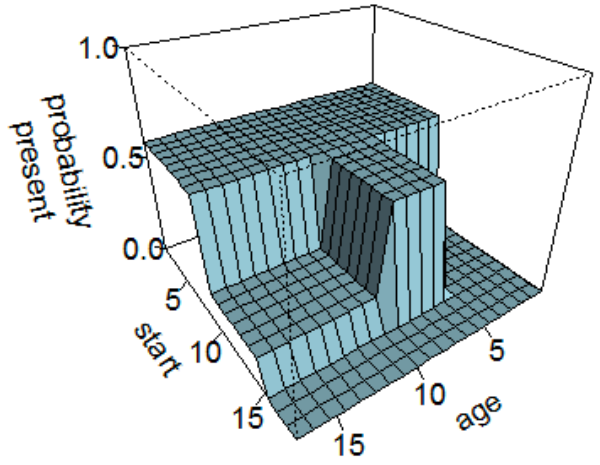
Basics of machine learning: non-parametric models

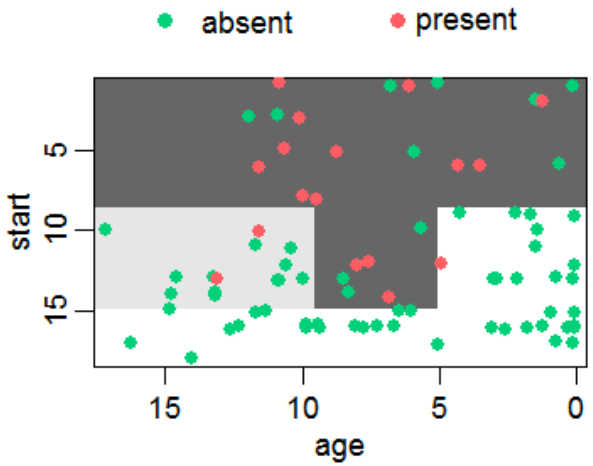
```
1 D = {(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)}
2 function knn(k,dist, D_{train}, D_{test}):
3 votes = []
4 for i = 1 to len(D_{test})
5     d = []
6     for j = 1 to len(D_{train})
7           d[j] = dist(x_i, x_j)
8     d = argsort(d)
9     votes[i] = most_common(set(labels[d]))
```



Basics of machine learning: semi-parametric models







The idea here is that we would like to find a partition of the input space and then fit very simple models to predict the output in each piece. The partition is described using a (typically binary) “decision tree,” which recursively splits the space.

Basics of machine learning: implementations

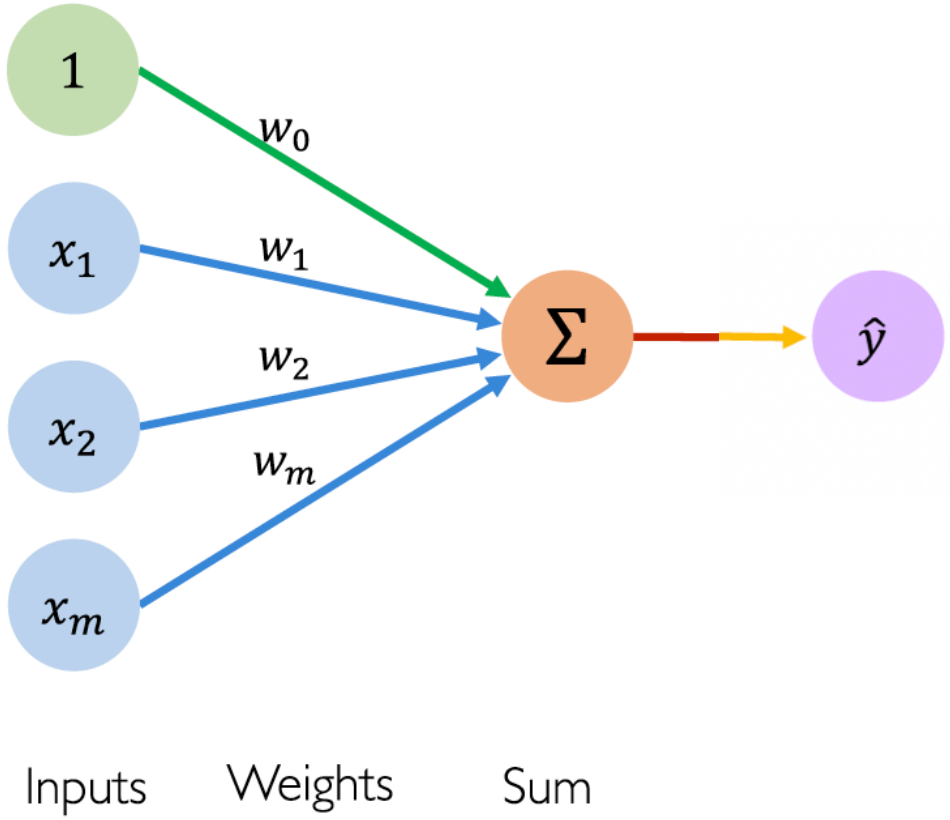
k-Nearest Neighbors (kNN)

```
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X, y)
neigh.predict([[1.5]])
```

Classification and regression decision trees (CART)

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
clf = DecisionTreeClassifier(random_state=0)
iris = load_iris()
clf.fit(iris.data, iris.target)
plt.figure()
plot_tree(clf, filled=True)
plt.show()
```

Neural networks: perceptrons to neurons



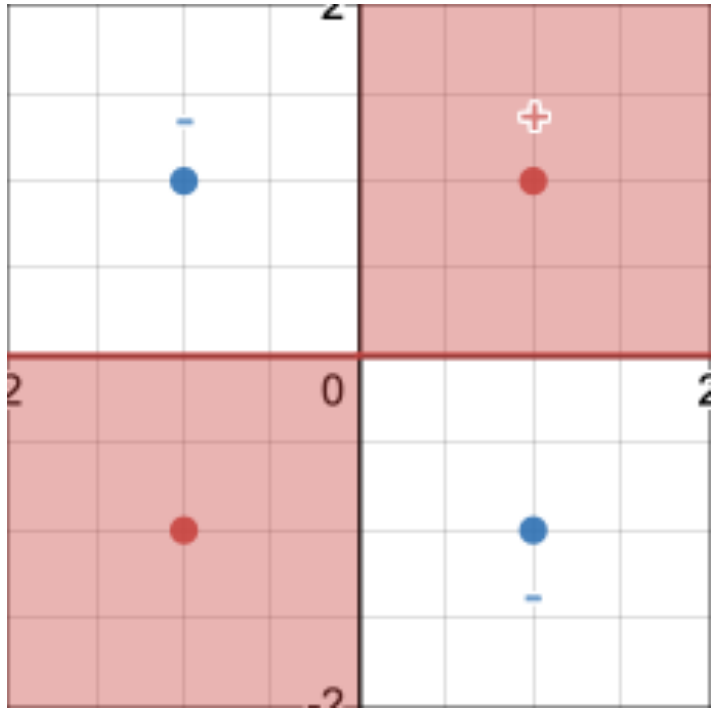
$$\hat{y} = w_0 + \sum_{i=1}^{m} x_i w_i$$

Basics of machine learning: feature representations

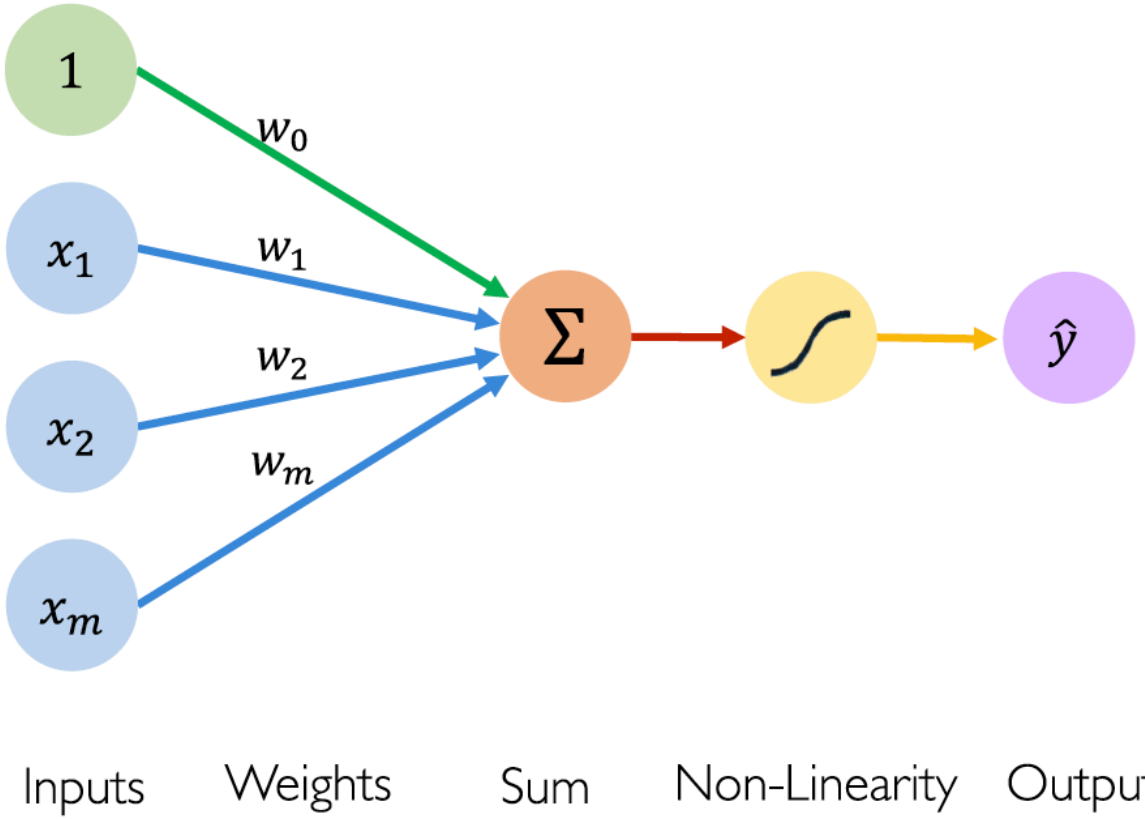
$$\phi((x_1, x_2)) = \left(1, x_1, x_2, x_1^2, x_1 x_2, x_2^2\right)$$

Polynomial basis

Order	d = 1	in general
0	$[1]$	$[1]$
1	$[1, x]$	$[1, x_1, \ldots, x_\mathrm{d}]$
2	$[1, x, x^2]$	$[1, x_1, \ldots, x_\mathrm{d}, x_1^2, x_1 x_2, \ldots]$
3	$[1, x, x^2, x^3]$	$[1, x_1, \ldots, x_1^2, x_1 x_2, \ldots, x_1 x_2 x_3, \ldots]$
$\vdots$	$\vdots$	$\vdots$



Neural networks: perceptrons to neurons



Output

Linear combination of inputs

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

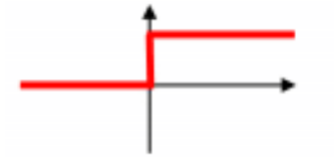
Non-linear activation function

Bias

Neural networks: activation functions

Step function

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

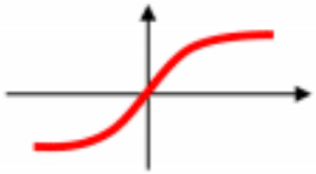


Rectified linear unit

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

Hyperbolic tangent

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



Sigmoid

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

Softmax

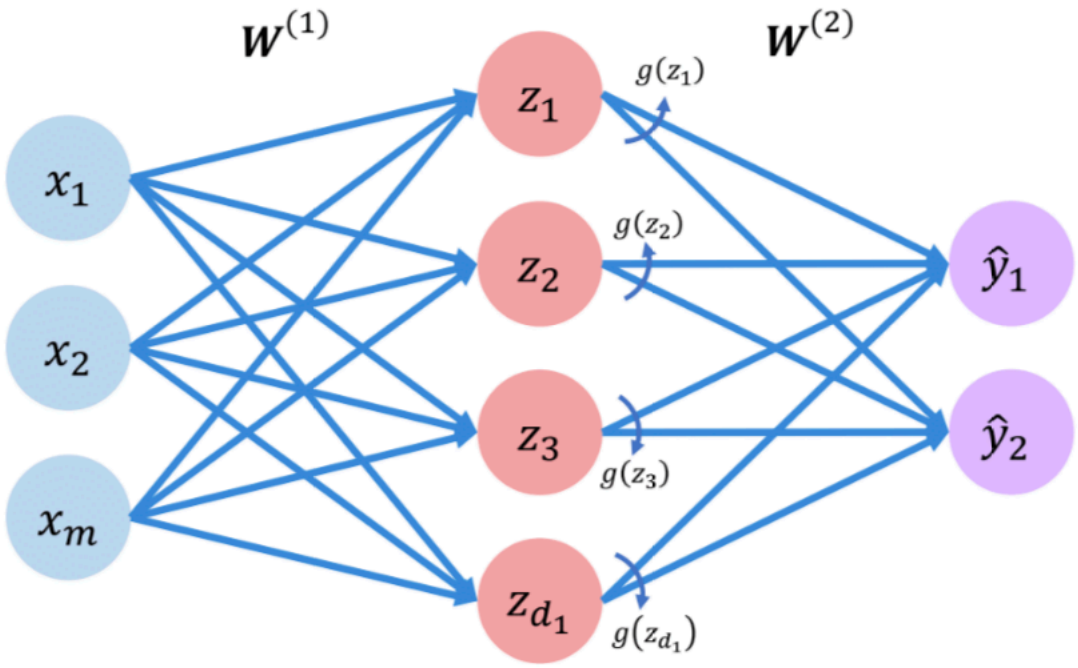
$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1) / \sum_i \exp(z_i) \\ \vdots \\ \exp(z_n) / \sum_i \exp(z_i) \end{bmatrix}$$

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: single layer feed-forward NN



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

Optimization: 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

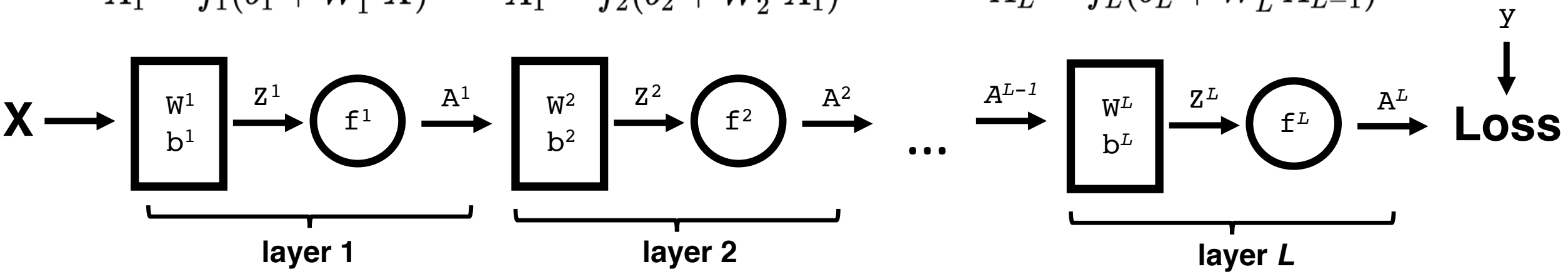
```
1 function gradient_update(W_init, η, J, ϵ):
2 W^0 = W_init
3 while |J(W^t) − J(W^(t−1))| > ϵ
4       W^t = W^(t−1) − η ∇_W(J)
```

This is an arbitrary update criteria

Neural networks: multi-layer feed-forward NN

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

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

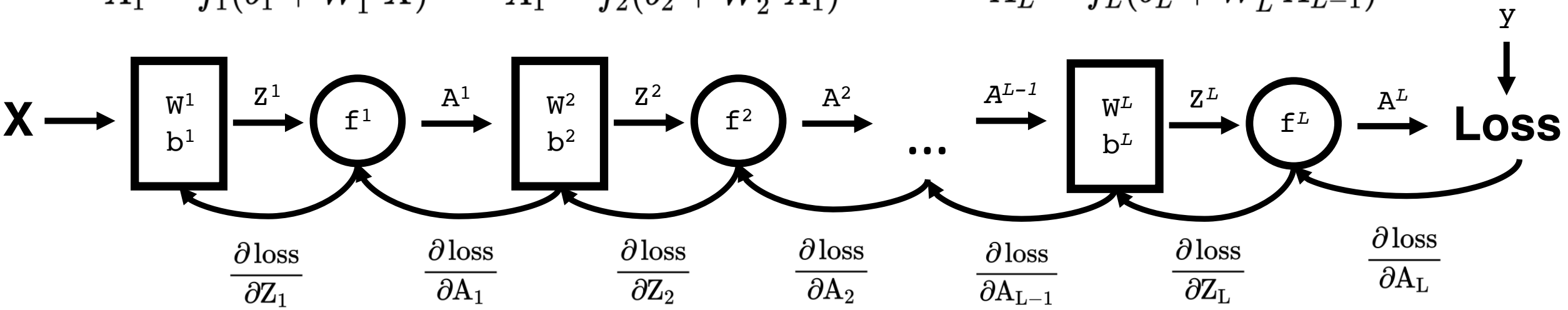


Inspired by 6.036 lecture notes (Leslie Kaebling)

Neural networks: error backpropogation

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

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



Let's work out error backprop on the board!

Inspired by 6.036 lecture notes (Leslie Kaebling)

Neural networks: error backpropogation

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,

$$\frac{\partial \text{loss}}{\partial \text{W}_\text{L}} = \frac{\partial \text{loss}}{\partial \text{A}_\text{L}} \cdot \frac{\partial \text{A}_\text{L}}{\partial \text{Z}_\text{L}} \cdot \frac{\partial \text{Z}_\text{L}}{\partial \text{W}_\text{L}}$$

Since,

$$\frac{\partial \text{Z}_\text{L}}{\partial \text{W}_\text{L}} = \frac{\partial}{\partial \text{W}_\text{L}}(W^T A_{L-1}) = A_{L-1}$$

We can re-write the equation as,

$$\frac{\partial \text{loss}}{\partial \text{W}_\text{L}} = A_{L-1} \frac{\partial \text{loss}}{\partial \text{Z}_\text{L}}$$

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,

$$\frac{\partial \text{loss}}{\partial \text{Z}_1} = \frac{\partial \text{loss}}{\partial \text{A}_\text{L}} \cdot \frac{\partial \text{A}_\text{L}}{\partial \text{Z}_\text{L}} \cdot \frac{\partial \text{Z}_\text{L}}{\partial \text{A}_\text{L-1}} \cdot \frac{\partial \text{A}_\text{L-1}}{\partial \text{Z}_\text{L-1}} \cdots \frac{\partial \text{A}_2}{\partial \text{Z}_2} \cdot \frac{\partial \text{Z}_2}{\partial \text{A}_1} \cdot \frac{\partial \text{A}_1}{\partial \text{Z}_1}$$

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:

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

The equation with the correct dimensions for matrix multiplication,

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

Inspired by 6.036 lecture notes (Leslie Kaebling)

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,

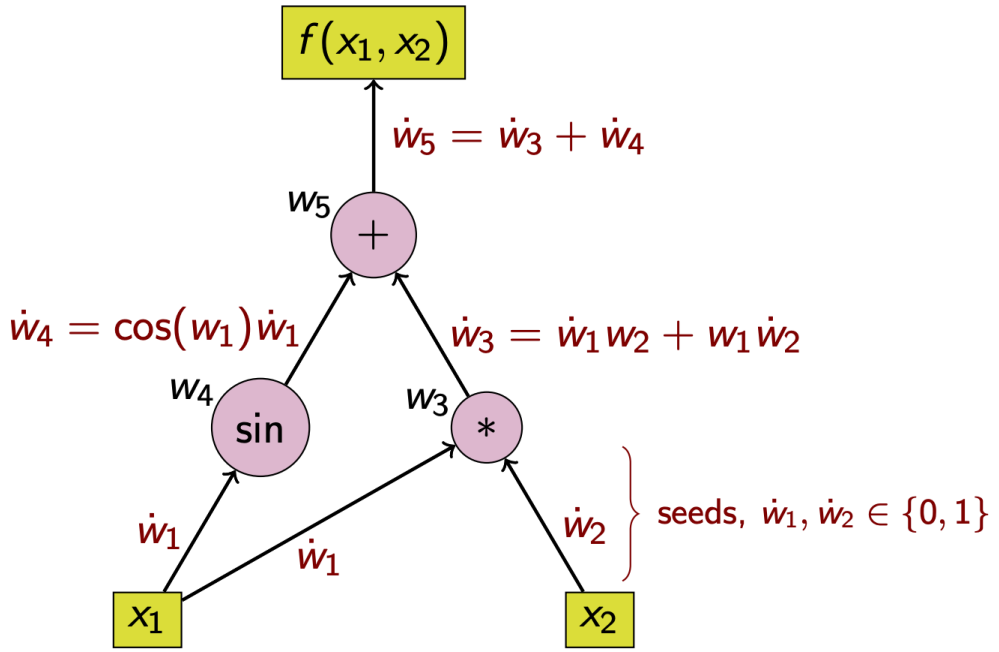
$$\epsilon^2 = 0$$

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

Neural networks: regularization

Objective function

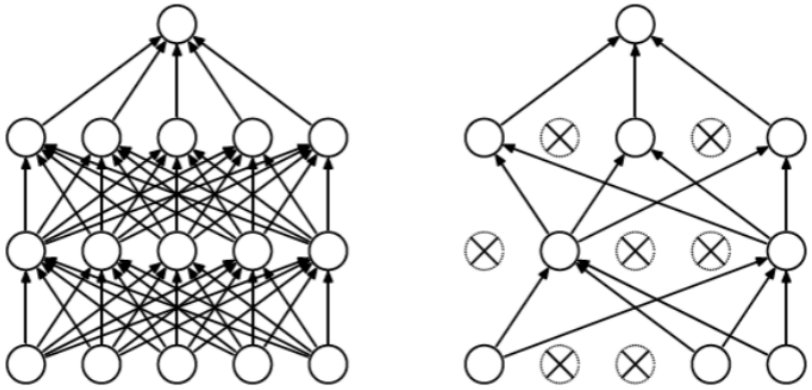
$$\mathrm{J}(W,b) = \frac{1}{\mathrm{n}} \sum_{\mathrm{i}=1}^{\mathrm{n}} \left(W^{\mathrm{T}} x^{(i)} + b - \mathrm{y}^{(i)} \right)^2$$

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.

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

Popular new approach: Batch normalization

Optimization: overview

Optimization: stochastic gradient descent

Stochastic gradient update (per randomly sampled training example):

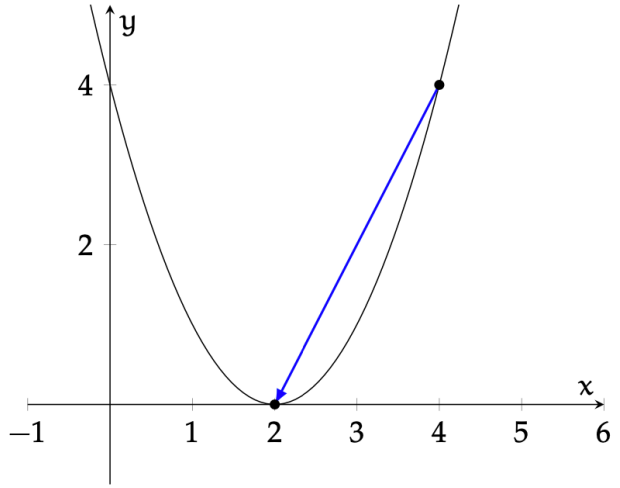
$$W := W - \eta \nabla_W \mathrm{J}\left(h\left(x^{(i)}; W\right), y^{(i)}\right)$$

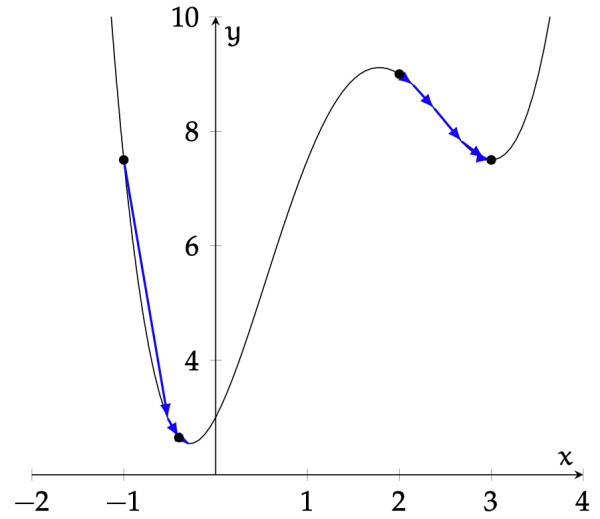
Pseudocode for stochastic gradient update algorithm

```
1 function sgd(W_init, η, J, T, ϵ):
2 W^0 = W_init
3 for t = 1 to T
4     randomly select i ∈ {1,2,...,n}
5     W^t = W^(t−1) − η(t) ∇_W(J)
```

Mini-batch gradient descent

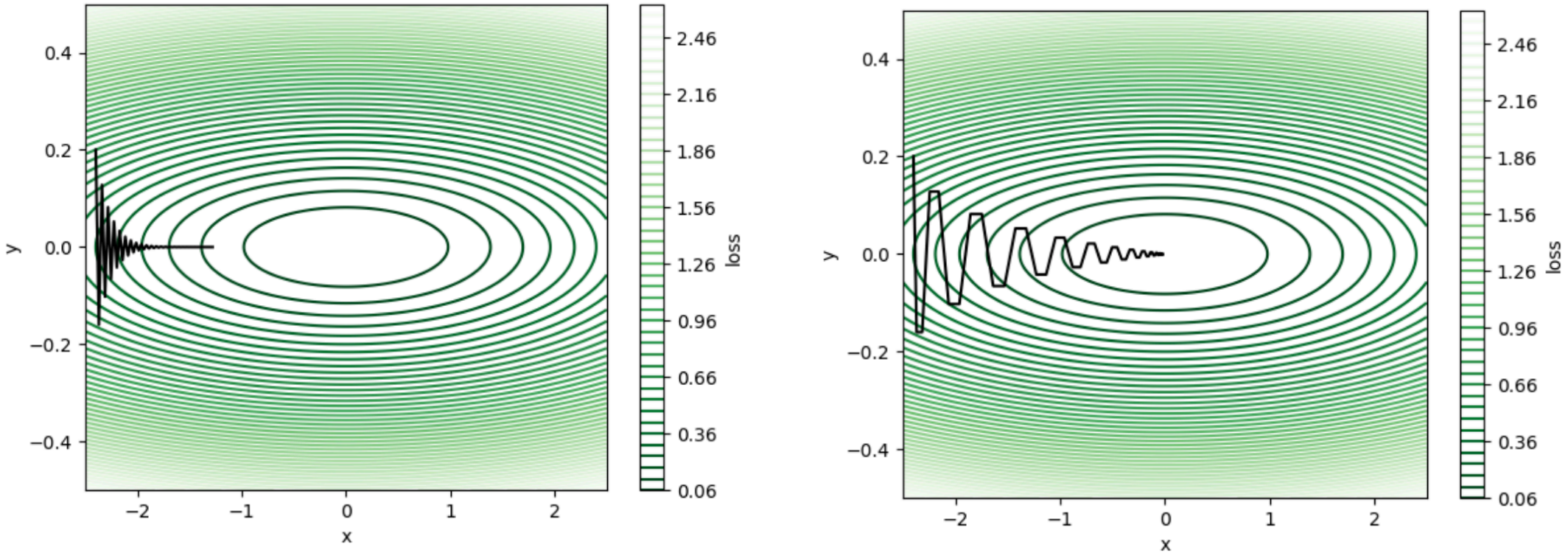
$$W := W - \eta \sum_{i=1}^{k} \nabla_W \mathcal{L}\left(h\left(x^{(i)}; W\right), y^{(i)}\right)$$





Optimization: momentum

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



Nesterov inequality? How can we make momentum better?

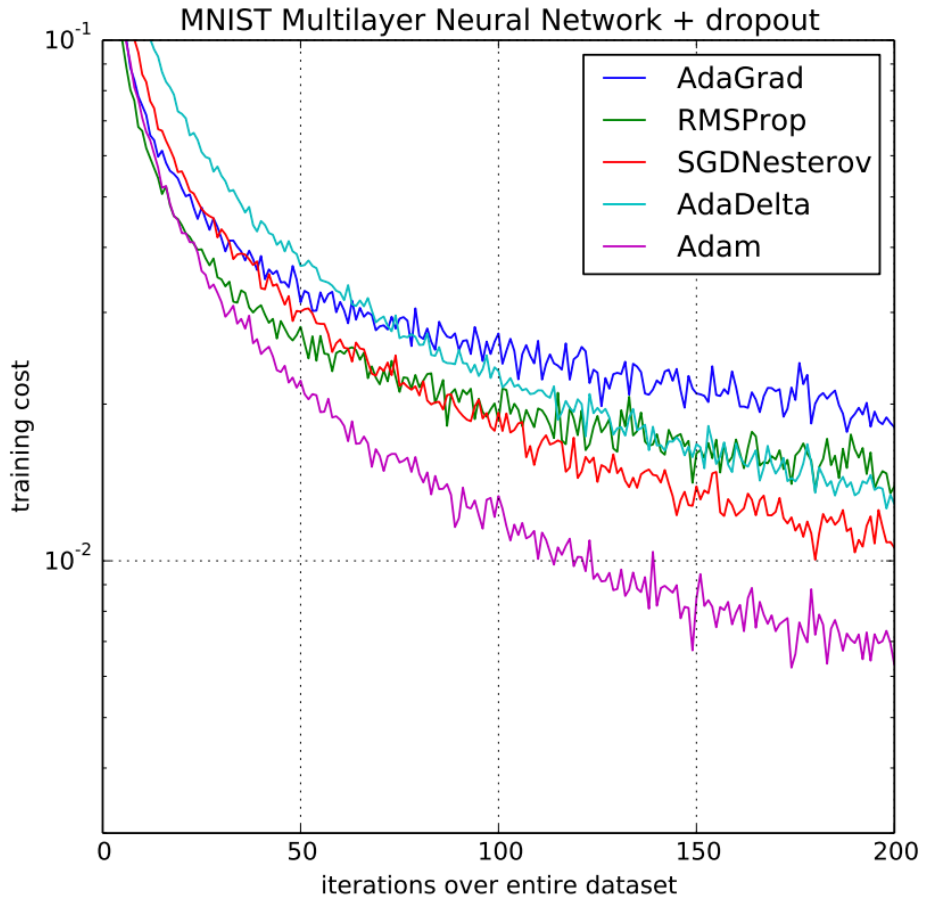
$$v_{k+1} = \beta v_k + \nabla f(w_k + \beta v_k)$$
$$w_{k+1} = w_k - \eta v_{k+1}$$

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

Next week

Batch normalization

Convolutional neural networks

Recurrent neural networks

Neural network interpretability