Algorithms and Loss functions

Machine Learning Theory (MLT)

Edinburgh

Rik Sarkar

Algorithms and loss functions

- We saw how to think about the sample complexity
  - Why machine learning works with reasonably small amounts of data
  - Why we need to decide hypothesis classes for learning to work
- Next:
  - How to find good models within the classes
  - Common types of loss functions and their properties
  - Common algorithms
    - Linear and polynomial predictors
    - Loss functions – convex and non-convex

Learning algorithms

- Each hypothesis or model is described by vector $\boldsymbol{w}$ of weights
  - The length of $\boldsymbol{w}$ is the dimension of the space of models
- We write bold $\boldsymbol{w}$ or $\boldsymbol{x}$ to indicate vectors.
- When writing by hand, it is perhaps best to write with an arrow overhead: $\vec{w}$ since bold is tricky in handwriting

- The weights $\boldsymbol{w}$ are the parameters that determine the model
- So, an ML algorithm searches in the space of $\boldsymbol{w}$ trying to find the best one

Two spaces: Models and Data

- Eg. For classifiers given by $y \leq mx + c$, the space of models is all possible values of (m, c), so it is 2 dimensional
- A model that has k parameters will have a model space that is k-dim

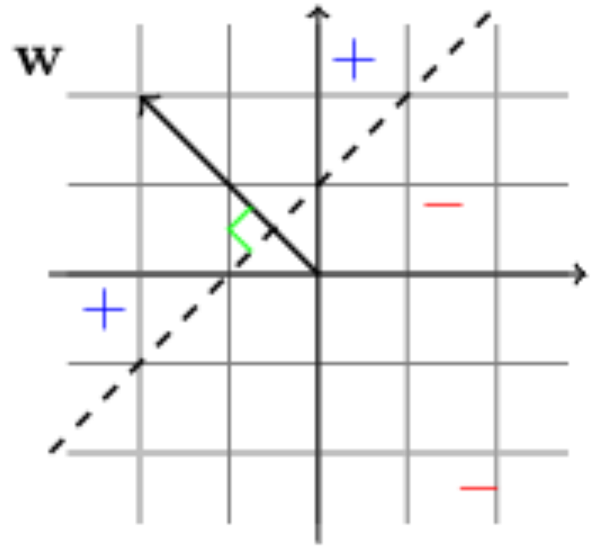
Models $\boldsymbol{w}$
Each point is a possible model

Data $\boldsymbol{x}$
Each point is a possible data point

Linear predictors

- Popular class of models
- Easy to train
- Easy to interpret

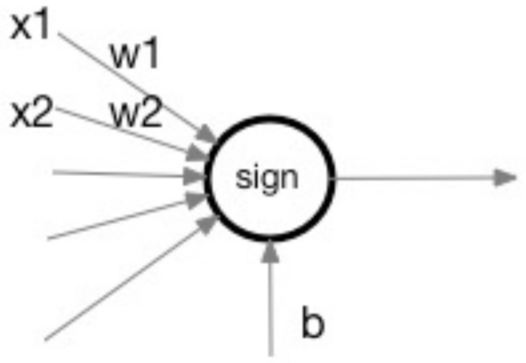
Halfspaces

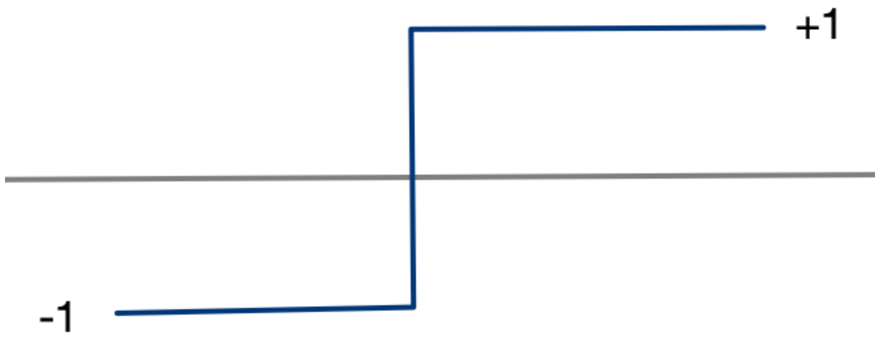


- All the elements on one side of a straight line
- Written as $sign(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b)$
  - Sign function returns +1 or -1 depending on sign
  - $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ is an inner product $\langle \boldsymbol{w}, \boldsymbol{x} \rangle = \sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{d}} \boldsymbol{w_i x_i}$
- VC dimension of class of halfspaces is $d + 1$
- Thus we should be able to learn the good halfspaces
- The realizable case for halfspaces is called separable
- LP can be used to solve the separable half space problem (omitted in class)

Perceptron

- A simple neuron denoting a half space classifier
- The activation function is a threshold function
- Challenge: learn the weights $\boldsymbol{w}$





Homogeneous coordinates

- Simplify $sign(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b)$
- We can extend
  - $\boldsymbol{w} = [b, w_1, w_2, \dots]$
  - $\boldsymbol{x} = [1, x_1, x_2, \dots]$
- Now we can write simply $sign(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$

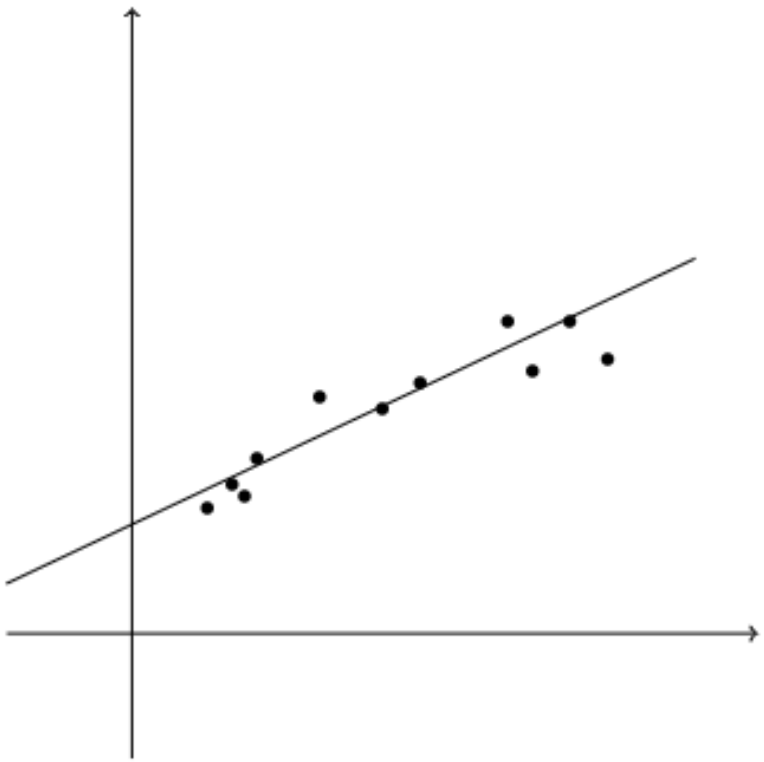
Perceptron algorithm

- Input: Training set $(x_1, y_1), (x_2, y_2), \ldots$
- Initialize $\boldsymbol{w}^1 = [0, \ldots, 0]$
- At each iteration $t = 1,2, \ldots$
  - If there is a sample $\boldsymbol{x_i}$ that is wrongly classified i.e. if $y_i \langle \boldsymbol{w}^t, \boldsymbol{x_i} \rangle \leq 0$
    - Update $w^{t+1} = w^t + y_i \boldsymbol{x}_i$
  - Else
    - Output $\boldsymbol{w}^t$

- Perceptron algorithm produces a half space classifier. (Thm 9.1)
- In the separable case produces the correct solution/model

Linear regression

- $\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{R}$
- $h: \mathcal{X} \to \mathcal{Y}$ should be linear
- Loss $\ell\big(h, (\boldsymbol{x}, y)\big) = (h(\boldsymbol{x}) - y)^2$
- Empirical risk
  - $L_S(h) = \frac{1}{m}\sum(h(\boldsymbol{x}_i) - y_i)^2$



- Note that the definition applies to any dimensional data

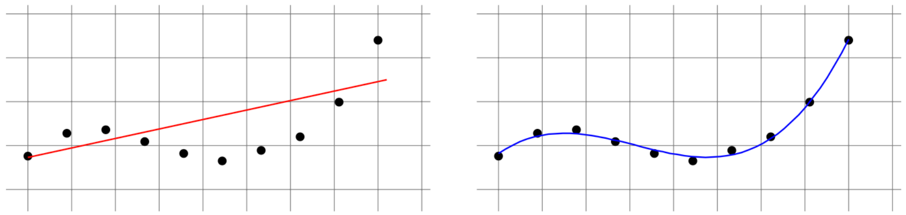
Least squares – solution to linear regression

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, L_S(h_{\mathbf{w}}) = \underset{\mathbf{w}}{\operatorname{argmin}}\ \frac{1}{m}\sum_{i=1}^{m}(\langle \mathbf{w}, \mathbf{x}_i\rangle - y_i)^2$$

- Idea: When the risk is at a minimum, its gradient is 0
- That is: $\frac{2}{m}\sum(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i)\boldsymbol{x}_i = 0$
- Solved using linear algebra (matrix) techniques

Polynomial regression

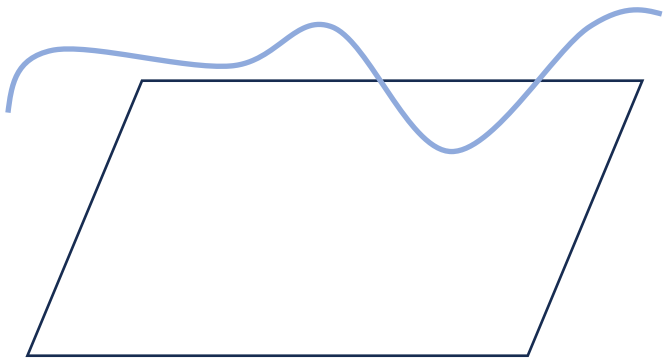
$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$



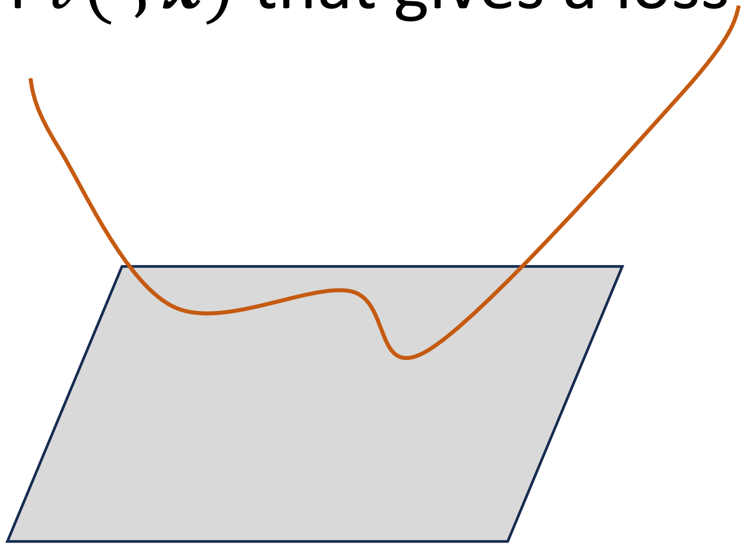
- Assume $\mathcal{X} = \mathbb{R}, \mathcal{Y} = \mathbb{R}$
  - I.e, 1-D, non-linear problems
- Define $\psi(x) = (1, x, x^2, \dots, x^n)$
  - And $$p(\psi(x)) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \langle \mathbf{a}, \psi(x) \rangle$$
- And apply linear regression
- That is, treat each degree term of x as a different dimension, and apply multi-dimensional linear regression.

Loss functions

- Loss $\ell(\boldsymbol{w}, \boldsymbol{x})$ is a function of both data and models
- For every model $\boldsymbol{w}$, there is a function $\ell(\boldsymbol{w}, \cdot)$ on data space that defines the loss at every point
- For every data point x there is a function $\ell(\cdot, \boldsymbol{x})$ that gives a loss for each model



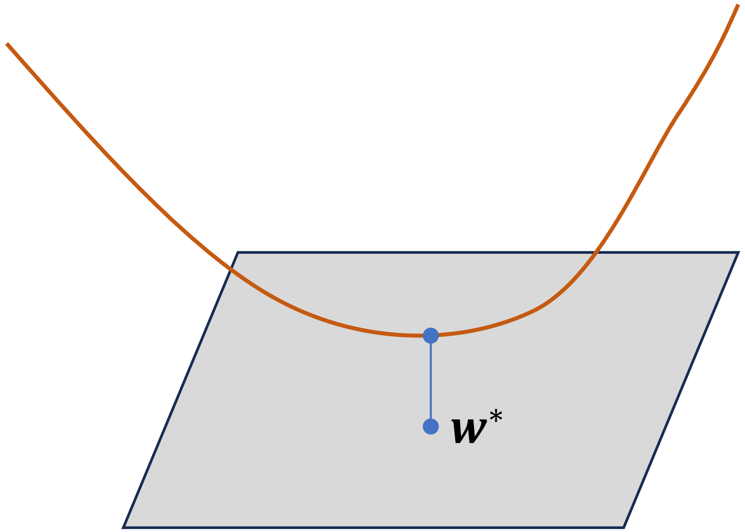
Data $\boldsymbol{x}$



Model $\boldsymbol{w}$

Loss functions

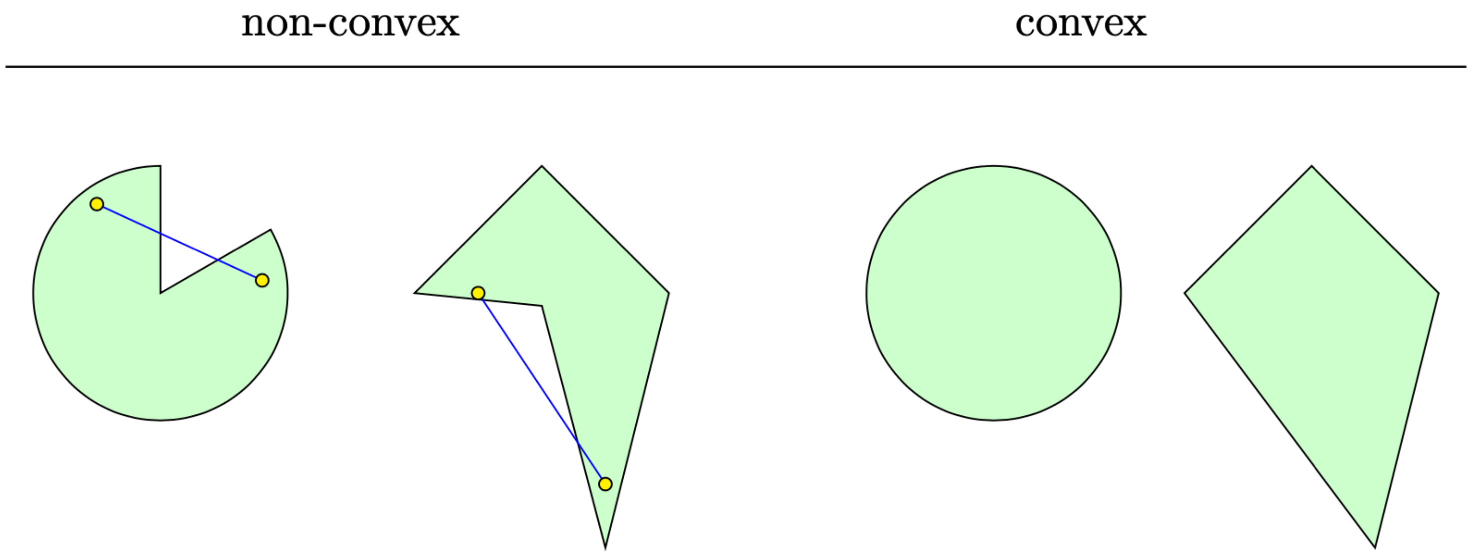
- We are usually interested in the average of $\ell(\cdot, \boldsymbol{x})$ over all data points
- And want to find $\boldsymbol{w}$ that minimizes the average $L(\boldsymbol{w}, \boldsymbol{x})$
  - Call it $\boldsymbol{w}^*$



Models $\boldsymbol{w}$

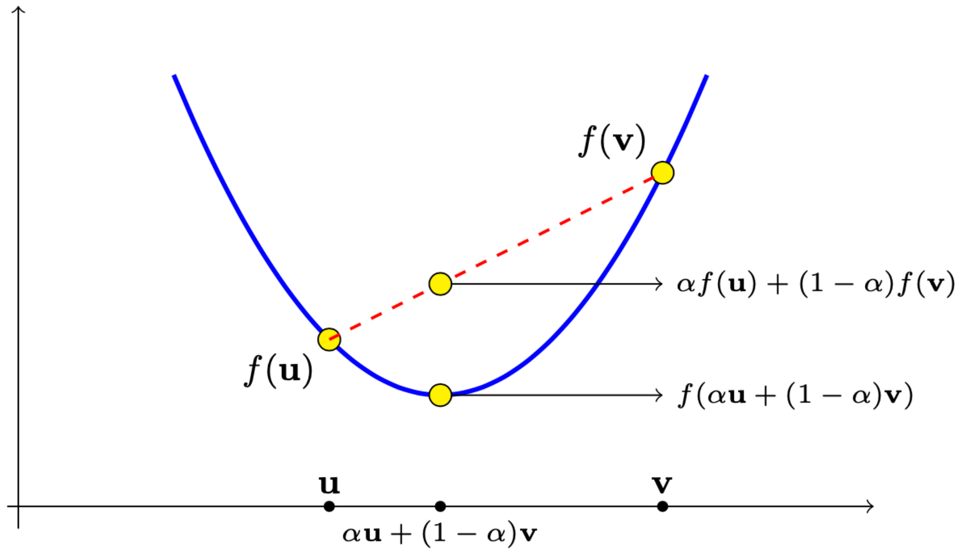
Convexity and convex learning

- A set C is convex if for any $u, v \in C$, the line segment connecting $\boldsymbol{u}, \boldsymbol{v}$ is in C. (Any intermediate point is in C)
  - Can be written formally as:
  - For any $\alpha \in [0,1]$, it is true that $\alpha \boldsymbol{u} + (1-\alpha)\boldsymbol{v} \in C$



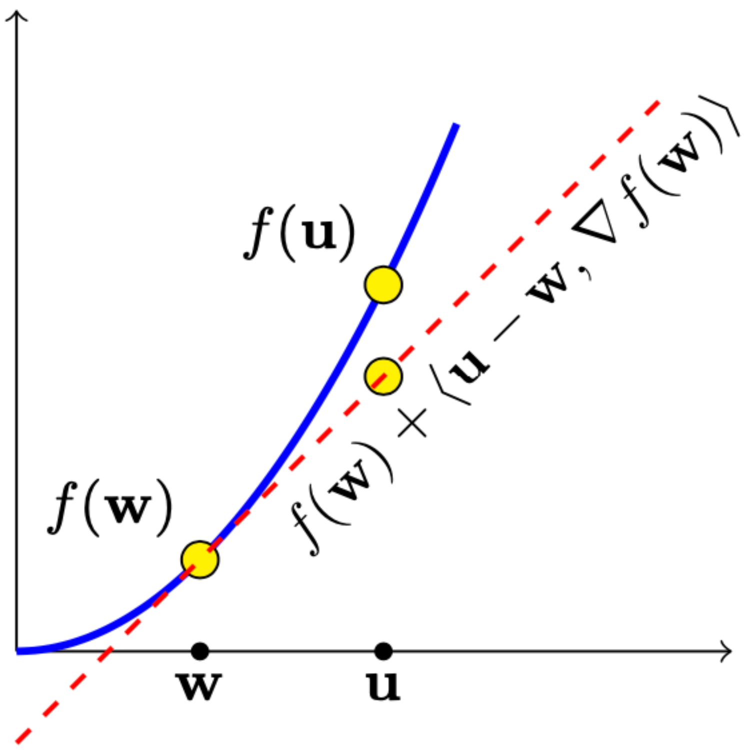
Convex function

- For a convex C, a function $f: C \to \mathbb{R}$ is convex if
- $f(\alpha \boldsymbol{u} + (1-\alpha)\boldsymbol{v}) \leq \alpha f(\boldsymbol{u}) + (1-\alpha) f(\boldsymbol{v})$
- The graph of f lies below the straight line connecting u and v



Properties of convex functions

- Every local minimum is also a global minimum
  - Question: is the global minimum unique?
- For every $\boldsymbol{w}$ the tangent at $\boldsymbol{w}$ lies below f:
  - $\forall \boldsymbol{u}, f(\boldsymbol{u}) \geq f(\boldsymbol{w}) + \langle \nabla f(\boldsymbol{w}), \boldsymbol{u} - \boldsymbol{w} \rangle$
- If $f: \mathbb{R} \to \mathbb{R}$ is twice differentiable, then
  - f is convex
  - f' is monotone nondecreasing
  - f'' is nonnegative
- Are equivalent



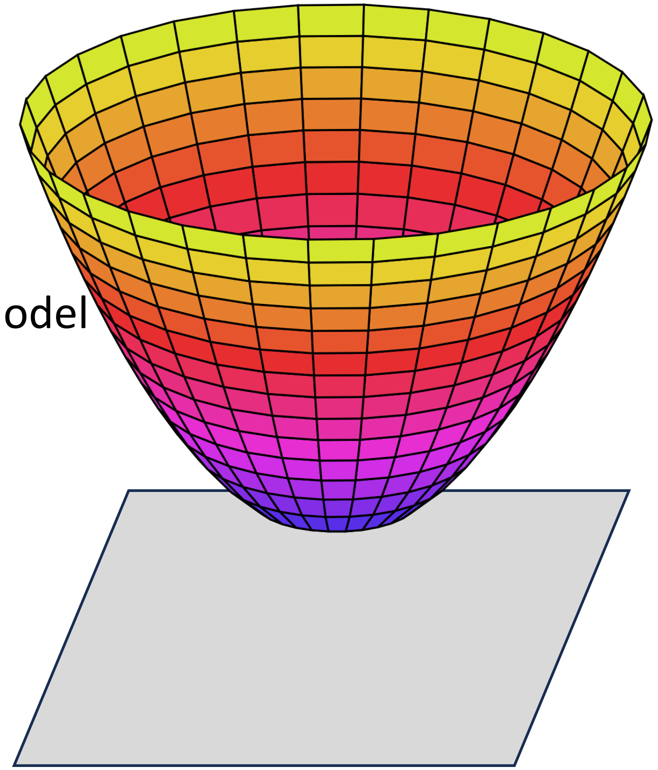
Examples

- $f(x) = x^2$

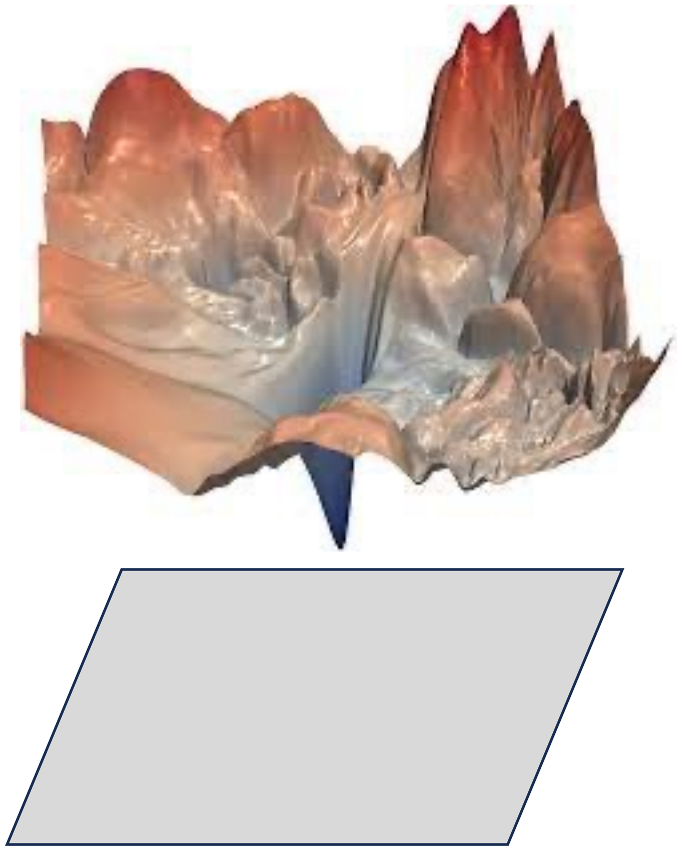
- $f(x) = \log(1 + e^x)$

Convex and non-convex loss: The loss landscape

Empirical Loss of the model
Over training data

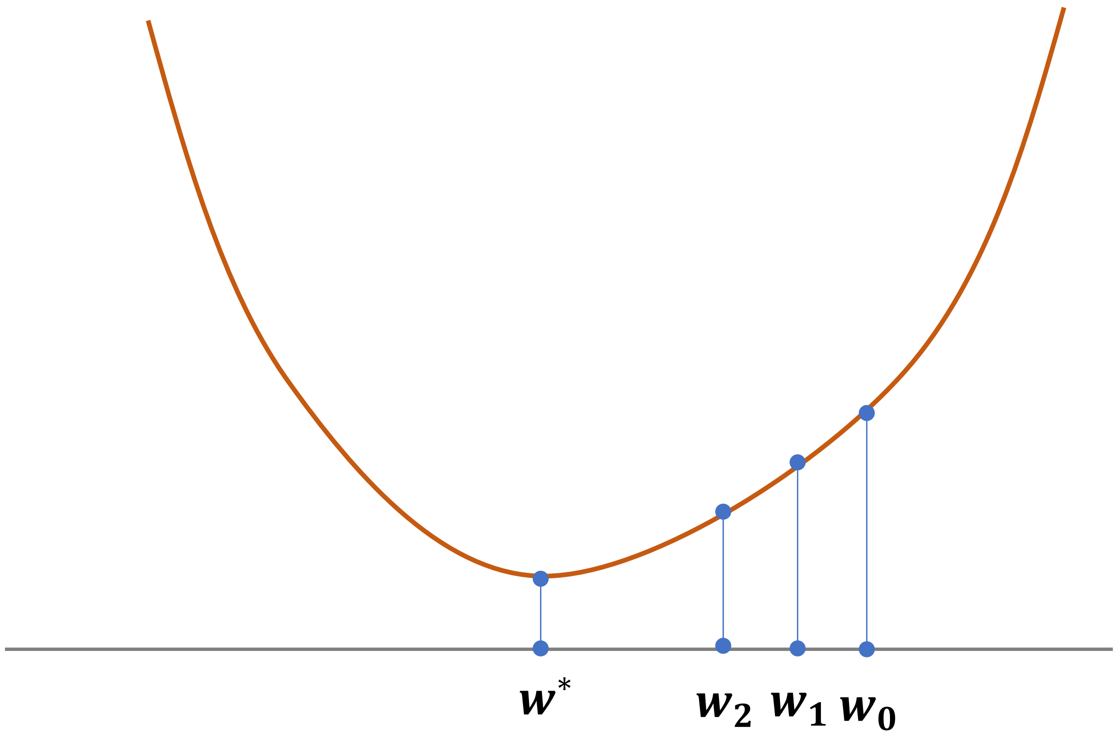


Space of models $\boldsymbol{w}$
Each point is a
model



Convex learning is easy!

- Start with any model $\boldsymbol{w}_0$
- Take a step in a direction that makes the loss smaller
- Repeat until we are at $\boldsymbol{w}^*$ with smallest loss
- Gradient descent
  - Compute the derivative at current $\boldsymbol{w}$, move a step in that direction



Gradient

- Gradient (a vector derivative in multiple dimensions)
  - The direction and speed of fastest increase

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)$$

  - (here a w_i is a parameter or dimension of the model)

- Partial derivatives
  - Compute the derivative along each dimension, put them in a vector

Convex learning problems

- A learning problem $(\mathcal{H}, \mathcal{Z}, \ell)$ is convex, if
  - $\mathcal{H}$ is a convex set
  - For all $z \in \mathcal{Z}$, the loss function $\ell(\cdot, z)$ is a convex function.

- E.g. linear regression with squared loss, logistic regression

Combining convex functions

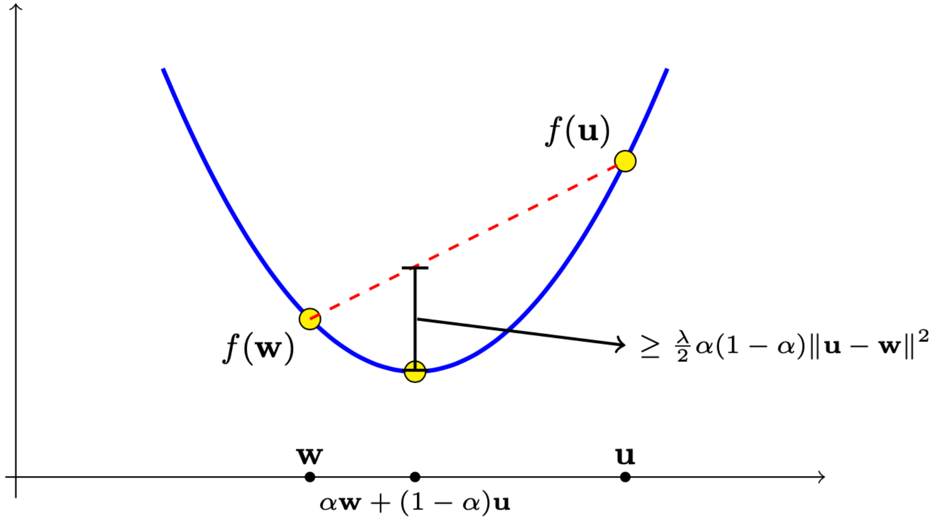
- If g is convex, then $f(\boldsymbol{w}) = g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + y)$ is convex
- If f_i are convex functions
- $g(x) = \max_i f_i(x)$ is convex
- $g(x) = \sum_i w_i f_i(x)$ is convex
  - What is the consequence for loss functions?

Other properties of loss functions

Strong Convexity

- Function f is λ-strongly convex if

$$f(\alpha \mathbf{w} + (1-\alpha)\mathbf{u}) \leq \alpha f(\mathbf{w}) + (1-\alpha) f(\mathbf{u}) - \frac{\lambda}{2}\alpha(1-\alpha)\|\mathbf{w}-\mathbf{u}\|^2$$



Lipschitzness

- A function f is ρ-Lipschitz if
  - $\left||f(\boldsymbol{w}_1) - f(\boldsymbol{w}_2)|\right| \leq \rho||\boldsymbol{w}_1 - \boldsymbol{w}_2||$

- A function that does not change too fast
  - If the derivative is bounded by ρ,
    - What can we say about its lipschitzness?
    - Then the function is also ρ-Lipschitz
    - But lipschitzness can be defined/computed even when the derivative does not exist

Smoothness

- Gradient (a vector derivative in multiple dimensions)
  - The direction and speed of fastest increase

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)$$

- f is β-smooth if ∇f is β-Lipschitz:
  - $\left||\nabla f(\boldsymbol{v}) - \nabla f(\boldsymbol{w})|\right| \leq \beta||\boldsymbol{v} - \boldsymbol{w}||$

Convex-Lipschitz-Bounded learning problems

- A learning problem $(\mathcal{H}, \mathcal{Z}, \ell)$ where:

- $\mathcal{H}$ is convex, $\forall \boldsymbol{w} \in \mathcal{H}, \left|\left|\boldsymbol{w}\right|\right| \leq B$
- $\forall z \in \mathcal{Z}$ the loss $\ell(\cdot, z)$ is convex and ρ-Lipschitz (for some ρ)

Convex-smooth-bounded learning

- A learning problem $(\mathcal{H}, \mathcal{Z}, \ell)$ where:

- $\mathcal{H}$ is convex, $\forall \boldsymbol{w} \in \mathcal{H}, \left|\left|\boldsymbol{w}\right|\right| \leq B$
- $\forall z \in \mathcal{Z}$ the loss $\ell(\cdot, z)$ is convex, nonnegative and β-smooth (for some β)

The why do we want convexity, smoothness, lipschitzness etc?

The why do we want convexity, smoothness, lipschitzness etc?

- Avoids sudden changes in function and its gradients
- Easier to compute and apply gradients as optimization steps

What is the problem of 0-1 empirical risk as loss function?

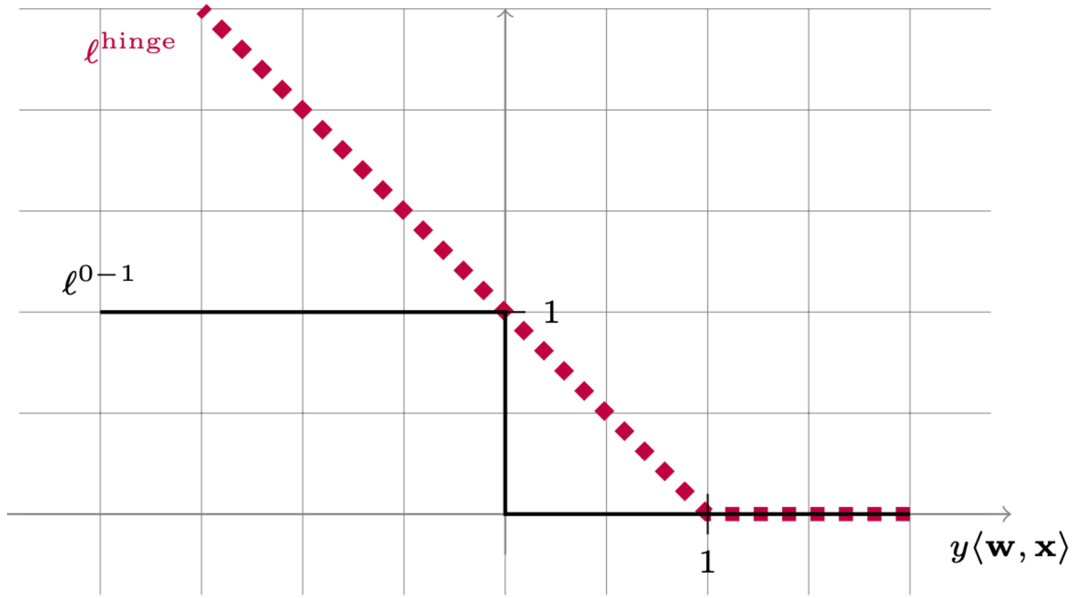
- Remember that we had defined the average empirical error as the loss.
  - Can we use that for gradient descent?

Surrogate loss functions

- Some loss functions are hard to work with. E.g.
  - They are not convex
  - They are hard to optimize for
  - E.g. 0-1 loss in halfspace-based classification
- Solution
  - Use a “surrogate” loss function
  - That is kind of similar, but easier to manage, e.g. convex
- Usual rule for surrogate loss
  - Should be convex
  - Should upper bound (be larger than original loss.)

Example: Hinge loss

$$\ell^{\text{hinge}}(\mathbf{w}, (\mathbf{x}, y)) \stackrel{\text{def}}{=} \max\{0, 1 - y\langle \mathbf{w}, \mathbf{x} \rangle\}$$



Regularization

- Instead of the pure loss, minimize loss with a regularization term:

$$\underset{\mathbf{w}}{\operatorname{argmin}}\left(L_S(\mathbf{w})+R(\mathbf{w})\right)$$

- Commonly used: $R(\boldsymbol{w}) = \lambda\left|\left|\boldsymbol{w}\right|\right|^2$
  - Called Tikhonov regularization

Try yourself:

Go to wolfram alpha and plot a polynomial: $y = a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$

- With numbers of your choice in place of coefficients a_i
- Now scale the coefficients: multiply all the coefficients with the same number (may be fractions too). What do you see?

Ridge regression

- Linear regression with Tikhonov regularization

$$\operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \left(\lambda \|\mathbf{w}\|_2^2 + \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i)^2 \right)$$

- $R(\boldsymbol{w}) = \lambda||\boldsymbol{w}||^2$ is 2λ-strongly convex
- If f is λ-strongly convex and g is convex, then $f + g$ is λ-strongly convex

- Thus, Ridge regression is strongly convex
- Strongly convex loss implies stability

Stability

- Intuitively: A learning algorithm is stable if
  - A small change to training set does not cause a big change to the output (model or hypothesis)

- This is a desirable property because…

Stability

- Intuitively: A learning algorithm is stable if
  - A small change to training set does not cause a big change to the output (model or hypothesis)

- This is a desirable property because
  - It implies that it is not too sensitive to specific S. does not overfit
  - If we continue to use it, it will not abruptly change behavior

- Suppose in S, we replace z_i with $z' \sim \mathcal{D}$
- Let us write this as S^i

- A good algorithm A should have small value for
  - $\ell\left(A\left(S^i\right), z_i\right) - \ell(A(S), z_i)$

- The loss on z_i does not depend too much on it being in the sample

Stability definition and result

- Algorithm A is on-average-replace-one-stable with rate $\epsilon(m)$
- If
  - $\mathbb{E}\left[\ell\left(A\left(S^i\right), z_i\right) - \ell(A(S), z_i)\right] \leq \epsilon(m)$

Stability definition and result

- Algorithm A is on-average-replace-one-stable with rate $\epsilon(m)$
- If
  - $\mathbb{E}\left[\ell\left(A\left(S^i\right), z_i\right) - \ell(A(S), z_i)\right] \leq \epsilon(m)$

- Theorem:
  - $\mathbb{E}\left[L_{\mathcal{D}}\left(A(S)\right) - L_S\left(A(S)\right)\right] = \mathbb{E}\left[\ell\left(A\left(S^i\right), z_i\right) - \ell(A(S), z_i)\right]$
  - The generalization gap is bounded by the stability

Generalisation Gap

- Empirical or training loss: $L_S(h)$
- Generalisation loss or true loss : $L_{\mathcal{D}}(h)$

- $L_{\mathcal{D}}(h) - L_S(h)$
- A measure of overfitting
  - (sometimes generalization gap is referred to as generalization loss)

Gradient descent

- Gradient is $$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w[1]}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w[d]} \right)$$

- Gradient represents the direction in which f increases fastest
- Gradient Descent: At every step t :
  - $w^{t+1} = w^t - \eta \nabla f(w^t)$
    - (Move in the direction that f decreases fastest With a step scale of η)

- After T steps, output the average vector $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^t$
- Other version: output final vector $\boldsymbol{w_T}$
- For us, f is the average loss L

Theorem (14.2 in book)

- For convex lipschitz bounded learning
- Setting $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$
- We can get $$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \ \leq \ \frac{B\,\rho}{\sqrt{T}}.$$
- Alternatively, to achieve $f(\overline{\boldsymbol{w}}) - f(\boldsymbol{w}^*) \leq \epsilon$ the number of rounds is:

$$T \geq \frac{B^2 \rho^2}{\epsilon^2}$$

Stochastic gradient descent

- Computing the gradient of empirical loss is expensive
  - Because empirical loss depends on all training data
- Idea: Instead of computing gradient on the entire dataset each time, compute them on small samples: like single data points.
  - (Each i.i.d data point is treated like a tiny sample of data)

Stochastic gradient descent (from book)

Stochastic Gradient Descent (SGD) for minimizing $f(\mathbf{w})$

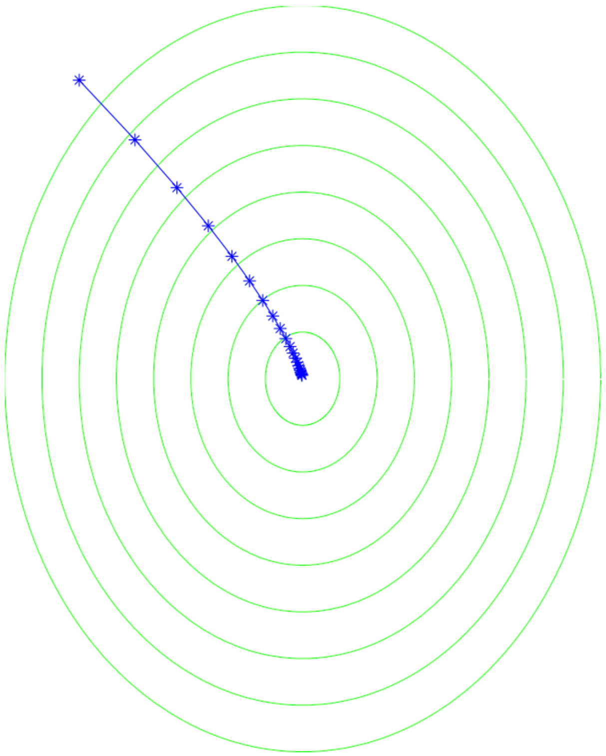
parameters: Scalar $\eta > 0$, integer $T > 0$
initialize: $\mathbf{w}^{(1)} = \mathbf{0}$
for $t = 1, 2, \ldots, T$
  choose $\mathbf{v}_t$ at random from a distribution such that $\mathbb{E}[\mathbf{v}_t \mid \mathbf{w}^{(t)}] \in \partial f(\mathbf{w}^{(t)})$
  update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{v}_t$
output $\bar{\mathbf{w}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}^{(t)}$

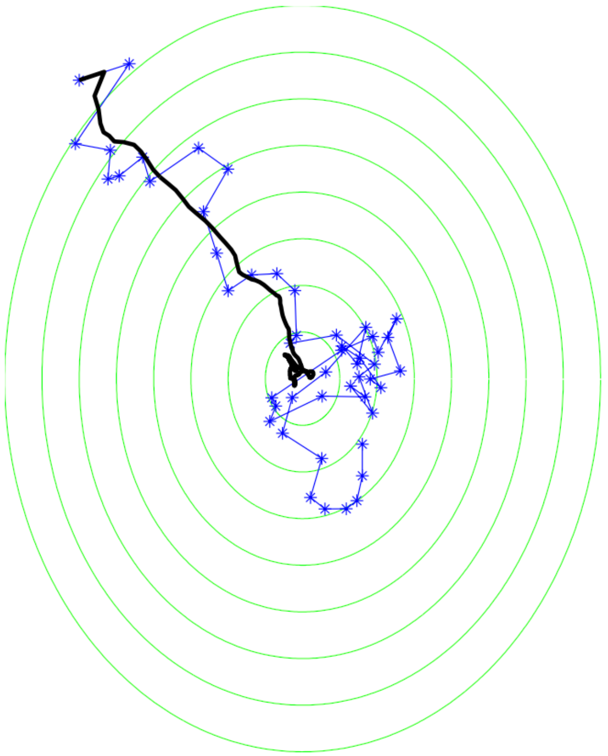
Stochastic gradient descent other version

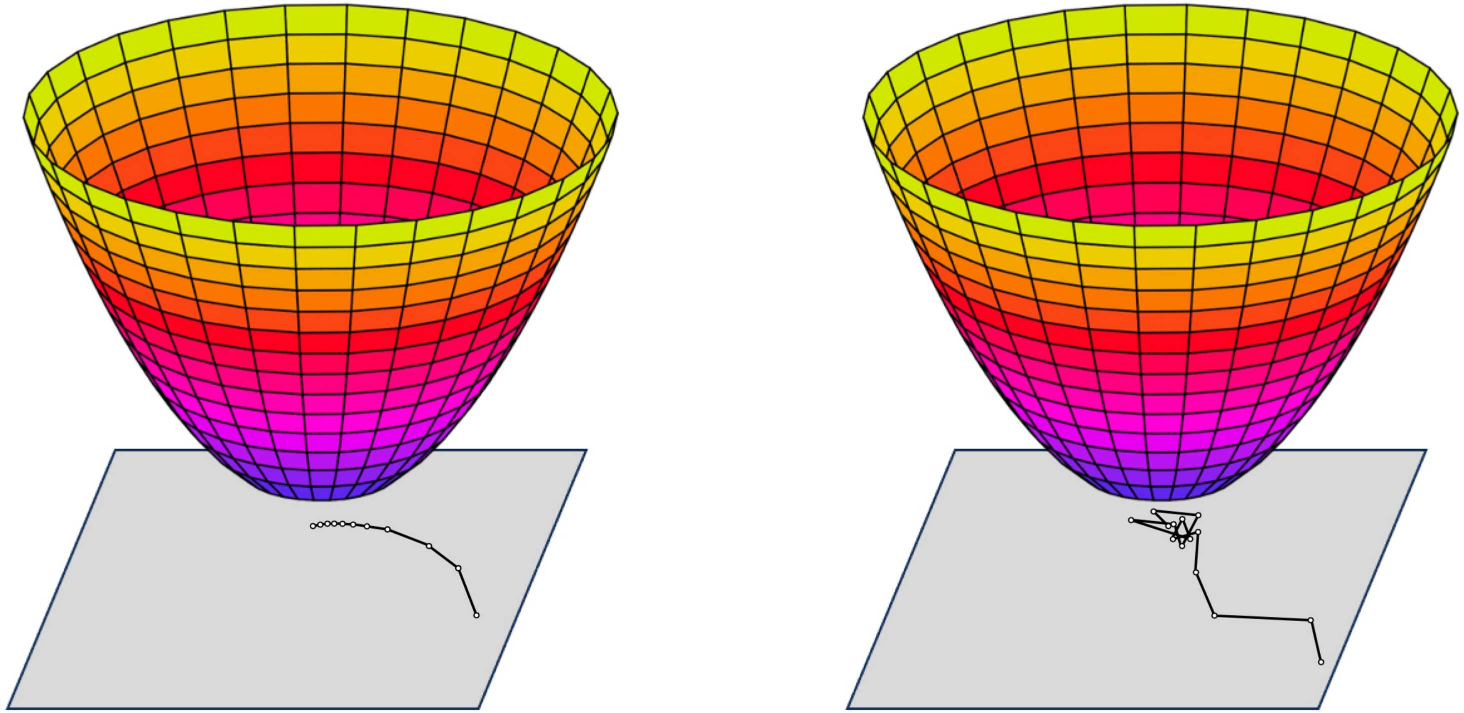
- Initialize $\boldsymbol{w}^1$ randomly (uniform or gaussian)

- For $t = 1 \ldots T$
  - Take a random small sample of data (mini batch)
  - Compute gradient $\boldsymbol{v}^t$ on this sample
  - Update $\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta \boldsymbol{v}^t$
- Output $\boldsymbol{w}^T$

GD vs SGD







Theorem (14.8)

- Similar result to deterministic GD:

$$\mathbb{E}\left[f(\bar{\mathbf{w}})\right] - f(\mathbf{w}^{\star}) \leq \frac{B\,\rho}{\sqrt{T}}$$

Practical modifications

- Mini batching:
  - Instead of one data item at time, take them in batches of a few at a time.
  - Faster, and fewer unhelpful moves
- Run in epochs. In each epoch
  - Order the data points in a random permutation
  - For each data point (or mini-batch)
    - Compute the gradient and move the model
- Other modifications:
  - Change learning rates
  - Add momentum, add dropout etc

Uniform Stability

- Suppose we get S^i by replacing one element z_i at position i of S with a new element z_i'
- And suppose that $z \in \mathcal{Z}$ is some possible input element
- As before $A(S)$ refers to the model that algorithm A computes using S
- We can write the loss on z as $\ell(A(S), z)$

- Algorithm A is ϵ-uniformly stable if
  - $\mathrm{Sup}_{z \in \mathcal{Z}}\left[E_A \ell\left(A(S^i), z\right) - E_A \ell(A(S), z)\right] \leq \epsilon$
- E_A means expectation taken over all possible random behaviour of A

Stability implies generalization

- Theorem:
- If Algorithm A is ϵ-uniformly stable then
  - $E_S E_A \ell(A(S), \mathcal{D}) \leq E_S E_A \ell(A(S), S) + \epsilon$
  - True loss $\leq$ Training loss + ϵ

Stability implies generalization

- Theorem:
- If Algorithm A is ϵ-uniformly stable then
  - $E_S E_A \ell(A(S), \mathcal{D}) \leq E_S E_A \ell(A(S), S) + \epsilon$
  - True loss $\leq$ Training loss + ϵ

- Proof:
  - Observe that $(\ell\left(A\left(S^i\right)z\right) \sim \ell\left(A\left(S^i\right)z_i \quad\right)$
    - Since z_i is just another random point outside S^i
  - Given S, Consider another random sample set $S' = \{z_1', z_2', \dots\}$

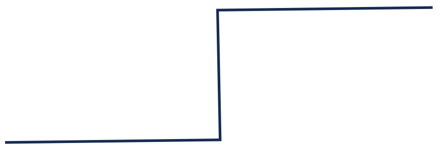
- $E_{S'}E_SE_A\ell(A(S),\mathcal{D}) - E_{S'}E_SE_A\ell(A(S),S)$

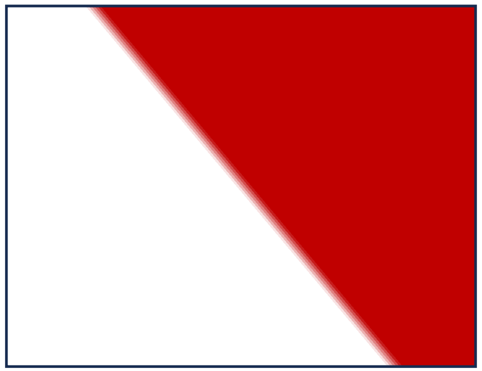
$= \frac{1}{m}\sum^m \quad E_{S'}E_SE_A\ell\left(A\left(S^i\right),z_i\right) - \frac{1}{m}\sum^m \quad E_{S'}E_SE_A\ell(A(S),z_i)$
$= \frac{1}{m}\sum^m \quad E_{S'}E_S\left[E_A\ell\left(A\left(S^i\right),z_i\right) \; - E_A\ell(A(S),z_i)\right] \leq \epsilon$

Thus, Uniform stability implies generalization.

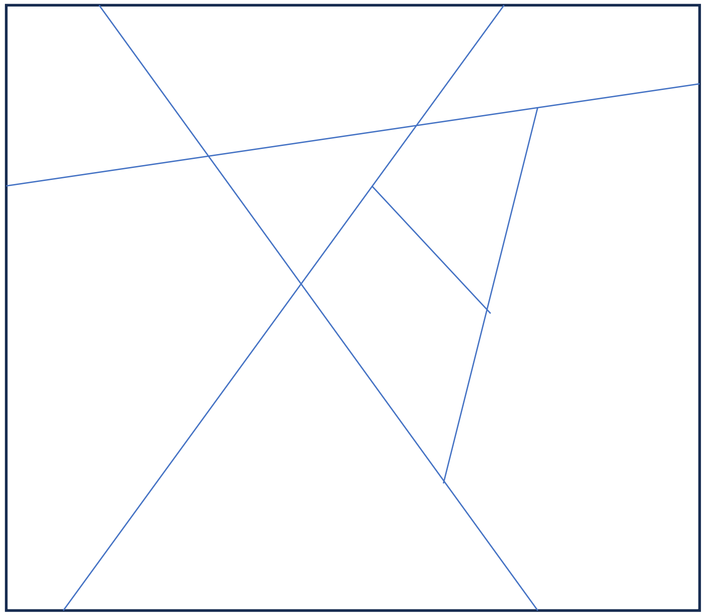
- Regularization creates strong convexity
- Strong convexity implies stability
- Stability implies generalization

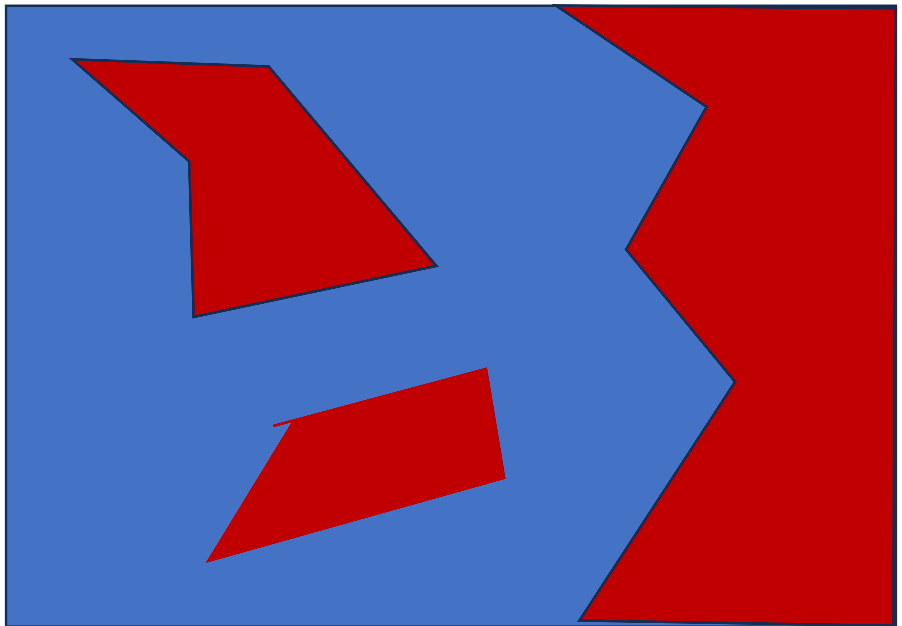
Neural networks

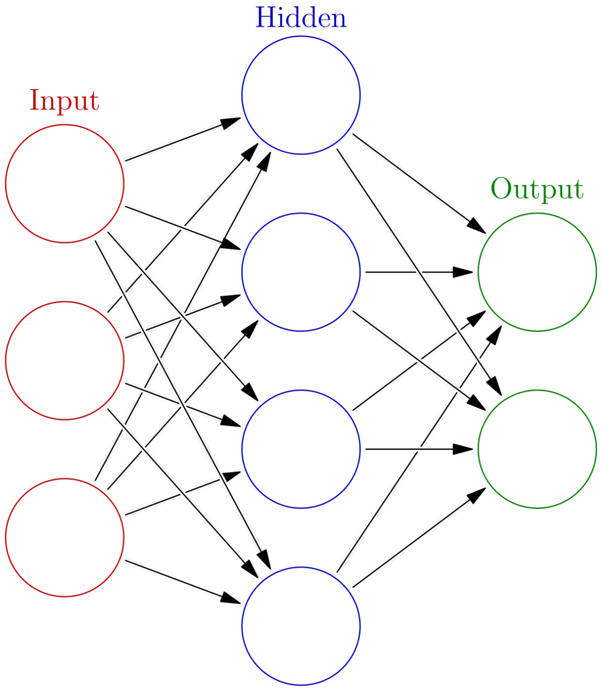




- Perceptron activation functions
- Each perceptron defines a half plane
- Together they can form complex boundaries
- More perceptrons, more options for regions available in the arrangement of lines



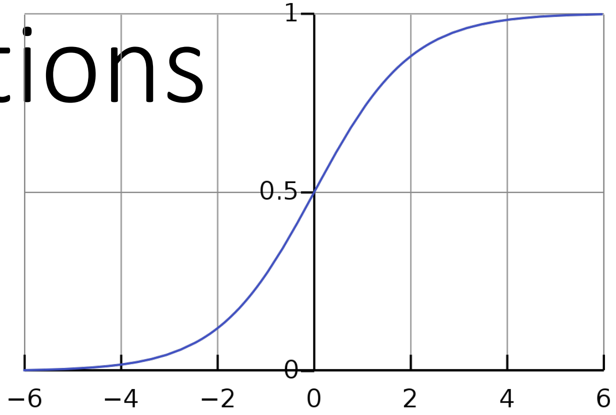


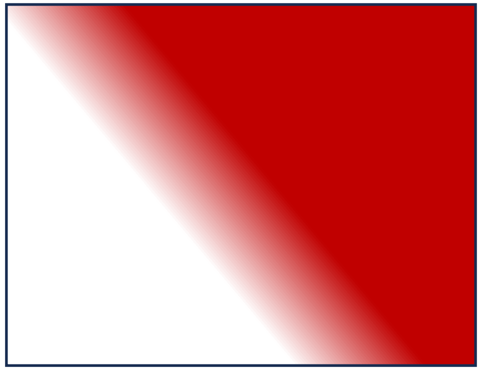


Challenges

- Gradients are not always useful
  - Eg. If a small change does not change the classification of any point
  - Hard to apply SGD type methods
- Sometimes it is useful to have real values

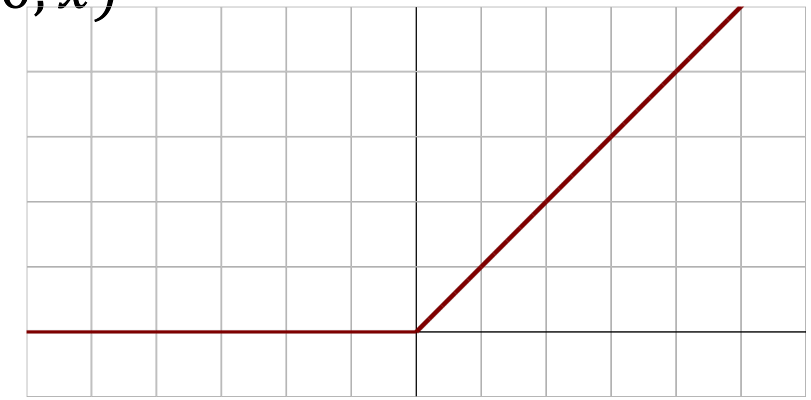
Other activations





- Sigmoid
  - $f(x) = \frac{1}{1+e^{x}}$

- ReLU
  - $f(x) = \max(0, x)$



Neural network structure

- Use ReLU or similar activation functions
  - More compatible with gradients
  - Easy to compute

- The middle layers produce a vector $\boldsymbol{y}$ of ”scores” for each class, called logit values
- Final layer: apply “softmax” to logits:
  - $softmax(\boldsymbol{y}_i) = \frac{e^{y_i}}{\sum e^{y_j}}$ (improved the notation from the lecture)

Question: Why softmax?

Hard max or exact max

- Take a vector of values eg. [2,3,5,2,6,4,9,2,2,4]
- Make one indicating the position of the max eg. [0,0,0,0,0,0,1,0,0,0]

Softmax

- Substitute for hard-max, but differentiable
- Normalized, can be treated as probability p_i for each class

Cross entropy loss

- Given:
  - Sample x
  - Probability estimate p_i
  - Truth label vector t: indicator vector or one-hot encoding where only the true class has value 1.

p=[0.1, 0.5, 0.2, 0.2]
t= [0.0, 1.0, 0.0, 0.0]

- Cross entropy loss: $\ell_{CE} = -\sum t_i \ln p_i$
  - Measures difference between the two functions