By Zied Ben Chaouch

6.036 Machine Learning

Last Updated April 16, 2019

Introduction

Machine Learning

Goal: make decisions or predictions based on data.
Problems to solve: Estimation + Generalization

Problem Characterization:

- Problem Class: Nature of training data (Supervised?)
- Assumptions: Source of data? Form of solution?
- Evaluation Criteria: Prediction goal? Performance measure?

Solution Characterization:

- Model Type: Intermediate model needed? Model used?
- Model Class: Parametric class of model needed?
- Algorithm: Computational process for making predictions

Problem Class

Superv. Learning: Given inputs $x^{(i)} \in \mathbb{R}^d$ or discrete + outputs $y^{(i)}$

- **Classification:** $y^{(i)}$ takes a finite set of values
- **Regression:** $y^{(i)} \in \mathbb{R}^k$

Unsuperv. Learning: Given inputs $x^{(i)} \in \mathbb{R}^d$ only

- **Density Estimation:** $\{x^{(i)}\}_{i=1}^n \overset{\text{iid}}{\sim} \mathbb{P}(X)$, find $\mathbb{P}$ for predictions
- **Clustering:** Partition samples into similar groups
- **Dimensionality Reduction:** e.g., Princip Component Analysis

Reinforc. Learning: Learn policy $\pi : x \to y$ maximizing reward

- Agent observes current state $x^{(0)}$
- Selects action $y^{(0)}$ & gets reward $r^{(0)}\left(x^{(0)}, y^{(0)}\right)$
- Environment generates new state $x^{(1)}$ under $\mathbb{P}\left(X | x^{(0)}, y^{(0)}\right)$

Sequential Learning: (Supervised) Learn mapping from input seq $x_0, \ldots, x_n$ to output seq $y_0, \ldots, y_m$.
Represent Map as **state machine** with functions f & g:
f: compute next hidden internal state given input
g: compute output given current hidden state

Other Settings:

- **Semi-Supervised Learn:** For some $x^{(i)}$, missing $y^{(i)}$
- **Active Learn.:** Minimize cost of obtaining labels $y^{(i)}$
- **Transfer/Meta Learn:** Multip tasks, data∈related distribs

Assumptions

Data: IID or Markov Chain or adversarial

True Model: Can be described by a set of hypotheses

Evaluation Criteria

Loss Function: $L(g, a)$ between guess g & actual a

0-1 Loss: $L(g, a) = \mathbb{I}\{g = a\}$

Squared Loss: $L(g, a) = (g - a)^2$

Linear Loss: $L(g, a) = |g - a|$

Asymetric Loss: $L(g,a) = \begin{cases} 1 & g = 1 \,\&\, a = 0 \\ 10 & g = 0 \,\&\, a = 1 \\ 0 & g = a = 0,\ g = a = 1 \end{cases}$

Model Type

No Model: Predict directly from training data without construction of any intermediate model. ex: k-Nearest Neighbor method

With Model: Fit model to training data (get prediction rule)+ use model to make predictions
Prediction Rule: Hypothesis $y = h(x; \theta)$
Training Error: $\mathcal{E}_n(h) = \frac{1}{n}\sum_{i=1}^n L\left(h(x^{(i)}; \theta), y^{(i)}\right)$
Testing Error: $\mathcal{E}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} L\left(h(x^{(i)}; \theta), y^{(i)}\right)$

Model Class $\mathcal{M}$/Parameter Fitting

Model Class: $\mathcal{M}$, set of possible models typically parametrized by vector of parameters $\Theta = (\theta, \theta_0)$
ex: Linear Function: $h(x; \theta, \theta_0) = \theta^T x + \theta_0$

Algorithm

What sequence of computational instructions should we run in order to find a good model from our class?
ex: Least-Squares Minimization Algorithm: Minimize training error $\mathcal{E}_n(\theta)$ to determine Θ for $h(x; \theta, \theta_0)$

Linear Classifiers

Classification

Def. (Binary Classifier) Map $x \in \mathbb{R}^d$ to $y \in \{-1, +1\}$

Def. (Feature) $\phi : \mathbb{R}^d \mapsto \mathbb{R}^{d'}$: work with $\phi(x)$ instead of x.

Def. (Training Data) $\mathcal{D}_n := \left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\right\}$
Input $x^{(i)} \in \mathbb{R}^{d \times 1}$; Output $y \in \{-1, +1\}$.

Def. (Hypothesis Class) $\mathcal{H} := \{\text{classifiers } h : \mathbb{R}^d \to \{-1, +1\}\}$

Def. (Hypothesis) $h : x \mapsto y$

Def. (Learning Algorithm) Procedure mapping $\mathcal{D}_n \mapsto h \in \mathcal{H}$

Def. (Training Error) Given training dataset $\mathcal{D}_n$:
$\mathcal{E}_n(h) = \frac{1}{n}\sum_{i=1}^n \mathbb{I}\{h(x^{(i)}) \neq y^{(i)}\}$

Def. (Testing Error) Given testing dataset $\mathcal{D}_{n'}$:
$\mathcal{E}(h) = \frac{1}{n'}\sum_{i=1}^{n'} \mathbb{I}\{h(x^{(i)}) \neq y^{(i)}\}$

Linear Classifier

Def. (Sign Function) $\text{sgn}(x) \in \{+1, 0, -1\}, \forall x \in \mathbb{R}$

Def. (Linear Classifier) Hypothesis class given by:
$\mathcal{H} := \left\{h(x; \theta, \theta_0) = \text{sgn}\left(\theta^T x + \theta_0\right) : \theta \in \mathbb{R}^{n \times 1}, \theta_0 \in \mathbb{R}\right\}$

Def. (Hyperplane P_Θ induced by Θ) $\theta^T x + \theta_0 = 0 \implies \theta \perp P_\Theta$
Side of $\theta \to \oplus$; Side of $(-\theta) \to \ominus$; On $P_\Theta \to \circledcirc$.
x_i intercept: $x_i = -\theta_0/\theta_i$

Def. (Linear Separable Data) Training dataset $\mathcal{D}_n$ is lin. separable
$\iff \exists(\theta, \theta_0)$ s.t. $y^{(i)}(\theta^T x^{(i)} + \theta_0) > 0 \, \forall i = 1 \ldots n$
$\iff h(x^{(i)}; \theta, \theta_0) = y^{(i)} \iff \mathcal{E}_n(h) = 0$

Learning Alg. for the Linear Classifier

RANDOM-LINEAR-CLASSIFIER $(\mathcal{D}_n, k, d)$:

```
for j = 1 to k do
    Randomly sample (θ^(j), θ_0^(j)) from (R^d, R)
j* = argmin_{j∈{1,...,k}} E_n (θ^(j), θ_0^(j))
return (θ^(j*), θ_0^(j*))
```

Note: $k \nearrow \implies \mathcal{E}_n \searrow$

Evaluating a Learning Algorithm

Idea: (To Evaluate the Performance of a)

Classifier $h \in \mathcal{H}$: Measure **test** error $\mathcal{E}_n(h)$

Learning Algorithm: Hard! Ex: try **Cross-Validation**

- Train on other datasets: get $h_1, \ldots, h_k$
- Compare the h_k's performance on a new testing set

Def. (Cross-Validation) k-fold Cross Validation for evaluation
CROSS-VALIDATE $(\mathcal{D}, k)$:

```
Divide D into k chunks: D_1, ..., D_k of similar size
for i = 1 to k do
    Train h_i on D \ D_i
    Compute test error E_i(h_i) on non-D_i data
return (1/k) Σ_{i=1}^k E_i(h_i)
```

Note: Cross-Validation evaluates the **algorithm** that produces the hypotheses h, but does NOT evaluate the hypotheses h produced.

The Perceptron

Algorithm

Def. (Perceptron — Rosenblatt (1962))

Training Dataset: $\mathcal{D}_n = \left\{(x^{(i)}, y^{(i)}) : x^{(i)} \in \mathbb{R}^{d \times 1}, y^{(i)} \in \{\pm 1\}\right\}_{i=1}^n$

Binary Classifier: $h(x; \theta, \theta_0)$; **Parameters:** $\theta \in \mathbb{R}^{d \times 1}$, $\theta_0 \in \mathbb{R}$

Iterations: τ steps.

PERCEPTRON $(\tau, \mathcal{D}_n)$:

```
θ_0 = 0, and θ = [0  0  ...  0]^T
for t = 1 to τ do
    for i = 1 to n do
        if y^(i)(θ^T x^(i) + θ_0) ≤ 0 then
            θ = θ + y^(i) x^(i)
            θ_0 = θ_0 + y^(i)
return (θ, θ_0)
```

Note: If alg does not enter IF loop for n iterations: $\mathcal{E}_n(h) = 0$!
Prop: If data is linearly separable, Perceptron will find it.

Offset

Thm. (Dim. Increase)
Given θ_0, $\theta = \begin{bmatrix}\theta_1 & \ldots & \theta_d\end{bmatrix}^T$, and $x = \begin{bmatrix}x_1 & \ldots & x_d\end{bmatrix}^T$:
Let $\theta_{\text{new}} = \begin{bmatrix}\theta_1 & \ldots & \theta_d & \theta_0\end{bmatrix}^T$, and $x_{\text{new}} = \begin{bmatrix}x_1 & \ldots & x_d & 1\end{bmatrix}^T$
$\implies \theta_{\text{new}}^T \cdot x_{\text{new}} = \theta^T \cdot x + \theta_0$
Note: Perceptron with offset $\Leftrightarrow$ Perceptron though origin in dim $d+1$
PERCEPTRON-THROUGH-ORIGIN $(\tau, \mathcal{D}_n)$:

```
θ = [0  0  ...  0]^T
for t = 1 to τ do
    for i = 1 to n do
        if y^(i)(θ^T x^(i)) ≤ 0 then
            θ = θ + y^(i) x^(i)
return θ
```

Theory of the Perceptron

Proposition (Distance of a Point x) to the hyperplane (θ, θ_0):
$\text{Dist}_{(\theta,\theta_0)}(x) = \frac{1}{\|\theta\|}\left(\theta^T x + \theta_0\right)$

Def. (Margin of a Labeled Point (x, y)) w.r.t hyperplane (θ, θ_0):
$\gamma_{(\theta,\theta_0)}(x, y) = y \cdot \frac{1}{\|\theta\|}\left(\theta^T x + \theta_0\right)$ **Prop:** $\gamma(x, y) > 0 \iff x$ is classified correctly as y by the linear classifier (hyperplane)

Def. (Margin of a Dataset $\mathcal{D}_n$) w.r.t hyperplane (θ, θ_0):
$\gamma_{(\theta,\theta_0)}(\mathcal{D}_n) = \min_i y^{(i)} \cdot \frac{1}{\|\theta\|}\left(\theta^T x^{(i)} + \theta_0\right)$ **Prop:** $\gamma(\mathcal{D}_n) > 0 \iff$ all pts in $\mathcal{D}_n$ are classified correctly by the linear classifier (hyperplane)
Note: If $\gamma(\mathcal{D}_n) > 0$, γ represents the dist from hyperpln to closest pt.

Thm. (Perceptron-Through-Origin Convergence Thm) If:

(a) $\exists \theta^* \in \mathbb{R}^d, \exists \gamma > 0$ s.t. $y^{(i)} \cdot \frac{1}{\|\theta^*\|}\left(\theta^{*T} x^{(i)} + \theta_0\right) \geq \gamma \ \forall i = 1 \ldots n$

(b) $\|x^{(i)}\| \leq R \ \forall i = 1 \ldots n$ (all pts have bdd size)

Then: Perceptron-Through-Origin makes at most $(R/\gamma)^2$ mistakes.

Feature Representation

Feature Transformation

Def. (X-OR Dataset) $D = \begin{bmatrix} - & + \\ + & - \end{bmatrix}$

Proposition (Transform X-OR) $\phi(x) = \begin{bmatrix} x & x^2 \end{bmatrix}^T$
$\implies$ X-OR is now linearly separable in 2D.

Note: This is the basis for **Kernel Methods**

Polynomial Basis

Method: get ϕ systematically (domain independent)
Idea: Use k^{th}-order basis:
$\phi : [1, x, x^2, x^3] \to [1, x_1, \ldots, x_1^2, x_1x_2, \ldots, x_1x_2x_3, \ldots]$
General:
$\phi : [1, x, \ldots, x^k] \to \left[x_1^{k_1} x_2^{k_2} \ldots x_k^{k_d}\right]_{\{k_1+k_2+\ldots+k_d=k | 0 \leq n_i \leq k\}}$
In k^{th}-order basis, we have $\binom{k'+d-1}{d-1}$ terms of order $k' \leq k$.
$\implies$ We have a basis of size: $\sum_{k'=0}^{k} \binom{k'+d-1}{d-1}$

Discrete Features

Method: get ϕ deliberately with our domain in mind (can be related to semantics)
Idea: (Encoding Strategy) Assume data takes one of k discr values:

Numeric: Standardize them! (speeds up learning algo)
$\phi(x) = (x - \bar{x})/\sigma$ with $\bar{x} = \text{avg}(x^{(i)})$ & $\sigma = \text{std}(x^{(i)})$

Numeric with Breakpoints: Break into bins & use one-hot
ex: Age $\lesseqgtr 21 \implies$ (Age < 21)&(Age ≥ 21)

Thermometer Code: Number but no natural ordering:
$0 < j \leq k \implies [1, 1, \ldots, 1^{(j)}, 0, \ldots, 0]$ (vector of length k)

Factored Code: Value can be split into two factors: Treat factors separately ex: Car $\to$ (Brand; Model)

One-Hot Code: No natural numeric/ordering/factor structure:
$0 < j \leq k \implies [0, 0, \ldots, 0, 1^{(j)}, 0, \ldots, 0]$ (vector of length k)

Binary Code: Bad idea! need to teach your algo to decode input...

Text: Bag-Of-Words (BOW) model:
$d = \#$ words in our vocab: vector $\in \{0, 1\}^d$ with $1^{(j)} \Leftrightarrow$ word j occurs

Margin Maximization

ML as Optimization

Idea: Frame ML problem into optimization problem
+ use standard algorithms/implementations to get hypothesis $h(x; \Theta)$

Def. (Objective Function) $J(\Theta; \mathcal{D}_n)$: Params $\Theta = (\theta, \theta_0)$, Data $\mathcal{D}_n$
Typically: $J(\Theta; \mathcal{D}_n) = \frac{1}{n}\sum_{i=1}^{n} L\left(h(x^{(i)}; \Theta), y^{(i)}\right) + \lambda \cdot R(\theta)$
$\implies$ Choose $\Theta^* := \text{argmin}_\Theta J(\Theta; \mathcal{D}_n)$

Regularization

Idea: Want to perform well on un-seen data (generalization): avoid overfitting!

Def. (Regularizer) Typically: $R(\Theta) = \|\Theta - \Theta_{\text{prior}}\|^2$
ex: No prior knowledge? $\implies$ Regularize towards zero $R(\Theta) = \|\Theta\|^2$

Maximize the Margin

Assumptions Classification setting, 0-1 loss

Idea: Want to maximize the margin of dataset (regularization!)
Recall: $\gamma_{(\theta,\theta_0)}(\mathcal{D}_n) = \min_i y^{(i)} \cdot \frac{1}{\|\theta\|}\left(\theta^T x^{(i)} + \theta_0\right)$

Def. (Margin Maximization) $J(\Theta, \mathcal{D}_n) = -\min_i \gamma(x^{(i)}, y^{(i)}, \Theta)$
$\implies \Theta^* = \text{argmin}_\Theta J(\Theta, \mathcal{D}_n)$
$\implies \Theta^* = \text{argmax}_\Theta \min_i \gamma(x^{(i)}, y^{(i)}, \Theta)$

Warning: This form of the objective can be tricky to optimize as it is only sensitive to a single data point at a time
$\implies$ gradient methods won't work very well

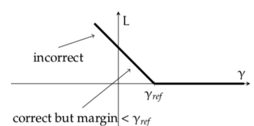
Idea: Use a target γ_{ref} & try to find separator s.t.

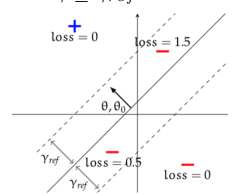
(a) $\gamma(x^{(i)}, y^{(i)}) > \gamma_{ref}$

(b) γ_{ref} is big

Def. (Hinge Loss) $L_h(v) = \max(1 - v, 0) = \begin{cases} 1 - v, & \text{if } v < 1 \\ 0, & \text{if } v \geq 1 \end{cases}$

Example: $L_h\left(\frac{\gamma}{\gamma_{ref}}\right) = \begin{cases} 1 - \frac{\gamma}{\gamma_{ref}}, & \text{if } \gamma < \gamma_{ref} \\ 0, & \text{if } \gamma \geq \gamma_{ref} \end{cases}$





Def. (Objective Function) To maximize the margin, minimize:
$J(\Theta, \gamma_{ref}) = \frac{1}{n}\sum_{i=1}^{n} L_h\left(\frac{\gamma(x^{(i)}, y^{(i)}, \Theta)}{\gamma_{ref}}\right) + \lambda \cdot \left(\frac{1}{\lambda_{ref}}\right)^2$
$= \frac{1}{n}\sum_{i=1}^{n} L_h\left(\frac{1}{\gamma_{ref} \cdot \|\theta\|} \cdot y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right)\right) + \lambda \cdot \left(\frac{1}{\lambda_{ref}}\right)^2$

Support Vector Machines (SVM)

Idea: The scale of θ does not affect the classifier (separator) obtained:
$\implies$ Pick $\|\theta\| \stackrel{!}{=} \frac{1}{\gamma_{ref}}$ **Note:** large magin $\Leftrightarrow$ small θ

Def. (SVM Objective Function) Want to minimize:
$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_h\left(y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right)\right) + \lambda \cdot \|\theta\|^2$

Note: $\lambda \to 0 \implies$ No regularization ; $\lambda \to \infty \implies \theta \to 0$;
Lin. Sep. Data: very small $\lambda \implies \exists$ at least 1 pt on the margin
$\gamma(x^{(i)}, y^{(i)}) = \gamma_{ref} = \frac{1}{\|\theta\|} \implies y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right) = 1$

Gradient Descent

Idea: Goal: minimize the $f(\Theta) = J(\mathcal{D}_n; \Theta)$ surface

- Start at an arbitrary point;
- Take a small step in direction of steepest descent;
- Take more small step in the new directions of steepest descent

One Dimension

Parameter: $\Theta \in \mathbb{R}$ (1-dimensional!) $-$ Initial Value: $\Theta_{\text{init}} \in \mathbb{R}$

Step Size: $\eta \in \mathbb{R}$

Function to Minimize: $f(\mathcal{D}_n; \Theta)$ (so $f : \mathbb{R} \to \mathbb{R}$)

Derivative of the Function: $f'(\mathcal{D}_n; \Theta)$

Accuracy Parameter: $\varepsilon \in \mathbb{R}_+$

1D-GRADIENT-DESCENT $(\Theta_{\text{init}}, \eta, f, f', \varepsilon)$:
$\Theta^{(0)} = \Theta_{\text{init}}$; $t = 0$
while $|f'(\Theta^{(t)})| \geq \varepsilon$ **do**
$t = t + 1$
$\Theta^{(t)} = \Theta^{(t-1)} - \eta f'\left(\mathcal{D}_n; \Theta^{(t-1)}\right)$
return $\Theta^{(t)}$

Note: Other ways to terminate: cap # of iterations t, or stop when $|\Theta^{(t)} - \Theta^{(t-1)}| < \varepsilon$, or when $|f(\Theta^{(t)}) - f(\Theta^{(t-1)})| < \varepsilon$.
Note: Small $\eta \Rightarrow$ slow convergence; Big $\eta \Rightarrow$ oscillations or divergence
Thm. (Convex Optimization) If $J(\Theta)$ is convex:
$\forall \varepsilon > 0, \exists \eta$ s.t. 1D-GD converges within ε of the optimal Θ.
Note: Non-convex J: may $\exists$ local minima!

Multiple Dimensions

Parameter: $\Theta \in \mathbb{R}^{d+1}$ $-$ Initial Value: $\Theta_{\text{init}} \in \mathbb{R}^{d+1}$

Step Size: $\eta \in \mathbb{R}$

Function to Minimize: $f(\mathcal{D}_n; \Theta)$ (so $f : \mathbb{R}^{d+1} \to \mathbb{R}$)

Gradient of the Function: $\nabla_\Theta f(\mathcal{D}_n; \Theta) = \begin{bmatrix} \frac{\partial f}{\partial \Theta_1} & \cdots & \frac{\partial f}{\partial \Theta_{d+1}} \end{bmatrix}^T$

Accuracy Parameter: $\varepsilon \in \mathbb{R}_+$

BATCH-GRADIENT-DESCENT $(\Theta_{\text{init}}, \eta, f, \nabla_\Theta f, \varepsilon)$:
$\Theta^{(0)} = \Theta_{\text{init}}$; $t = 0$
while $|f(\Theta^{(t)}) - f(\Theta^{(t-1)})| \geq \varepsilon$ **do**
$t = t + 1$
$\Theta^{(t)} = \Theta^{(t-1)} - \eta \nabla_\Theta f(\mathcal{D}_n; \Theta^{(t-1)})$
return $\Theta^{(t)}$

Application to SVM Objective

Hinge Loss: $L_h(v) = \max(1 - v, 0) = 1 - v \cdot \mathbb{I}\{v < 1\}$

Derivative of Hinge Loss: $L_h'(v) = -1 \cdot \mathbb{I}\{v < 1\}$

Objective: $J(\mathcal{D}_n; \theta, \theta_0) \in \mathbb{R}$

$$J(\mathcal{D}_n; \theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_h\left(y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0)\right) + \frac{1}{2}\lambda\|\theta\|^2$$

θ-Gradient: $\nabla_\theta J(\mathcal{D}_n; \theta, \theta_0) \in \mathbb{R}^{d+1}$

$$\nabla_\theta J(\mathcal{D}_n; \theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_h'\left(y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0)\right) y^{(i)} x^{(i)} + \lambda\theta$$

θ_0-Gradient: $\nabla_{\theta_0} J(\mathcal{D}_n; \theta, \theta_0) = \frac{\partial}{\partial \theta_0} J(\mathcal{D}_n; \theta, \theta_0) \in \mathbb{R}$

$$\nabla_{\theta_0} J(\mathcal{D}_n; \theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_h'\left(y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0)\right) y^{(i)}$$

SVM-GRADIENT-DESCENT $(\theta_{\text{init}}, \theta_{0,\text{init}}, \eta, J, \varepsilon)$:

$\theta^{(0)} = \theta_{\text{init}}$; $\theta_0^{(0)} = \theta_{0,\text{init}}$; $t = 0$

while $|J(\theta^{(t)}, \theta_0^{(t)}) - J(\theta^{(t-1)}, \theta_0^{(t-1)})| \geq \varepsilon$ **do**

$t = t + 1$

$$\theta_0^{(t)} = \theta_0^{(t-1)} + \eta \cdot \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}\left\{y^{(i)} \cdot (\theta^{(t-1)T} x^{(i)} + \theta_0^{(t-1)}) < 1\right\} y^{(i)}$$

$$\theta^{(t)} = \theta^{(t-1)} + \eta \cdot \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}\left\{y^{(i)} \cdot (\theta^{(t-1)T} x^{(i)} + \theta_0^{(t-1)}) < 1\right\} y^{(i)} x^{(i)} + \lambda\theta^{(t-1)}$$

return $(\theta^{(t)}, \theta_0^{(t)})$

Note: λ does not appear in θ_0 updates: don't regularize the offset! only the slope needs to be regularized (made simpler). Offset $\approx$ scaling

Stochastic Gradient Descent

Idea: If gradient is in form of a sum: $f(\mathcal{D}_n; \Theta) = \sum_{i=1}^n f_i(\mathcal{D}_n^{(i)}; \Theta)$
Don't take 1 small step in the direction of the gradient
$\rightarrow$ randomly select 1 term in sum and take tiny step in that direction.
You will move in the direction of the gradient on average.

STOCHASTIC-GRAD-DESCENT $(\Theta_{\text{init}}, \eta, f, \nabla_\Theta f_1, \ldots, \nabla_\Theta f_n, T)$:

$\Theta^{(0)} = \Theta_{\text{init}}$

for $t = 1$ to T **do**

Randomly get $i \in \{1, \ldots, n\} \implies$ Focus on $(x^{(i)}, y^{(i)}) \in \mathcal{D}_n^{(i)}$

$\Theta^{(t)} = \Theta^{(t-1)} - \eta(t) \cdot \nabla_\Theta f_i(\mathcal{D}_n^{(i)}; \Theta^{(t-1)})$

return $\Theta^{(t)}$

Thm. (Convex Optimization) If $J(\Theta)$ is convex:

$$\sum_{t=1}^{\infty} \eta(t) = \infty \;\&\; \sum_{t=1}^{\infty} \eta(t)^2 < \infty \implies \text{SGD converges a.s. to optimal } \Theta$$

Note: For SGD, η must decrease! Ex: $\eta \sim 1/t$

Note: • If f non-convex with many local optima: BGD gets trapped! $\implies$ taking samples from the gradient at some point Θ can make you bounce off of local optima.
• May not want to optimize f perfectly (overfitting of training set) $\implies$ SGD can get lower test error (but probably not lower training error) than BGD.

Regression

Data: $\mathcal{D}_n = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, with $x^{(i)} \in \mathbb{R}^{d\times 1}$, $y^{(i)} \in \mathbb{R}$.

Hypothesis: $h : \mathbb{R}^d \rightarrow \mathbb{R}$; Linear: $h(x; \theta, \theta_0) = \theta^T x + \theta_0$

Non-Linear Feature Transformation ϕ: $h(x; \theta, \theta_0) = \theta^T\phi(x) + \theta_0$

Loss Function: Squared-Error $L(\text{guess} - \text{actual})^2$

Objective: Mean SE $J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^n \left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)^2$

Solution: $(\theta^*, \theta_0^*) = \text{argmin}_{\theta,\theta_0} J(\theta, \theta_0)$

OLS Analytical Solution

Def. (Ordinary Least Squares) Linear hypothesis + MSE

Assumptions $x^{(i)}$ augmented with row of 1's $\Rightarrow$ can ignore θ_0.

$X \in \mathbb{R}^{d\times n}$: $X = \left[x^{(1)} | \cdots | x^{(n)}\right]$, $x^{(i)} = \left[x_1^{(i)} \cdots x_d^{(i)}\right]^T \in \mathbb{R}^{d\times 1}$

$Y = \left[y^{(1)} \cdots y^{(n)}\right] \in \mathbb{R}^{1\times n}$

$W = X^T \in \mathbb{R}^{n\times d}$ and $T = Y^T \in \mathbb{R}^{n\times 1}$

Thm. (OLS Solution)

• **Objective:** $J_{\text{OLS}}(\theta) = \frac{1}{n}(W\theta - T)^T(W\theta - T)$

• **Gradient:** $\nabla_\theta J_{\text{OLS}} = \frac{2}{n}W^T(W\theta - T) \stackrel{!}{=} 0$

• **Solution:** $\theta_{\text{OLS}}^* = (W^TW)^{-1}W^TT = (XX^T)^{-1}XY^T$

Regularization

Def. (Ridge Regression)

• **Objective:** $J_{\text{Ridge}}(\theta) = \frac{1}{n}\sum_{i=1}^n \left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)^2 + \lambda\|\theta\|^2$

Warning: In what follows: θ_0 included in θ!

• **Gradient:** $\nabla_\theta J_{\text{Ridge}} = \frac{2}{n}W^T(W\theta - T) + 2\lambda\theta \stackrel{!}{=} 0$

• **Solution:** $\theta_{\text{Ridge}}^* = (W^TW + n\lambda\mathbf{1}_{d\times d})^{-1}W^TT$

Note: $(W^TW + n\lambda\mathbf{1}_{n\times n})$ invertible when $\lambda > 0$

Def. (Bias-Variance Tradeoff) Hypoth $h \in \mathcal{H}$ contributes to errors on test data by:

• **Structural Err:** (Bias) $\nexists h \in \mathcal{H}$ describing data well ($\mathcal{H}$ too simple)

• **Estimation Err:** (Variance) Not enough data to pick good $h \in \mathcal{H}$

Note: Regularization: $\lambda \nearrow \Rightarrow$ Bias $\nearrow$ & Variance $\searrow$

Optimize via Gradient Descent

Idea: Closed form solution $\sim O(d^3)$ to invert W^TW: too long!

Def. (Ridge Gradient Descent/SGD)

• **Objective:** $J_{\text{Ridge}}(\theta) = \frac{1}{n}\sum_{i=1}^n \left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)^2 + \lambda\|\theta\|^2$

• **Gradients:** $\nabla_\theta J_{\text{Ridge}} = \frac{2}{n}\sum_{i=1}^n \left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)x^{(i)} + 2\lambda\theta$

$\nabla_{\theta_0} J_{\text{Ridge}} = \frac{\partial}{\partial\theta_0}J_{\text{Ridge}} = \frac{2}{n}\sum_{i=1}^n \left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)$

Thm. (Convex Optimization) OLS & Ridge are convex objectives! $\implies$ unique minimum & guaranteed BGD convergence to optimum for small enough step size η

Neural Networks I

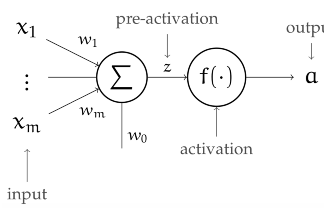
View 1: NN = Application of SGD for classification/regression with a potentially very rich hypothesis class $\mathcal{H}$

View 2: NN = Brain-inspired network of neuron-like computing elements that learn distributed representations

View 3: NN = Method to build applications that make predictions with huge data in very complex domains

Basic Element

Def. (Neuron/Unit/Node:)



Input: $x \in \mathbb{R}^m$ **Output:** $a = f(z) \in \mathbb{R}$

Weights: $w \in \mathbb{R}^m$ **Offset:** $w_0 \in \mathbb{R}$

Pre-Activation: $z = w^T x^{(i)} + w_0 = \sum_{j=1}^m w_j x_j^{(i)} + w_0$

Activation Function: $a = f(z) = f(w^T x^{(i)} + w_0)$

Def. (Objective Function) Note: Use in BGD/SGD!

$J(\mathcal{D}_n; w, w_0) = \sum_{i=1}^n L\left(NN(x^{(i)}; w, w_0), y^{(i)}\right)$

$NN(\cdot)$ = NN output ; $L(\text{guess}, \text{actual})$ = Loss Function

Note: Linear Classifiers with Hinge Loss + Linear Regressions with Quadratic loss $\implies$ 1 neuron with $f(x) = x$

Example: 1 Neuron, $f(z) = e^z$ & $L(g, a) = (g - a)^2$:

$J(w, w_0) = \sum_{i=1}^n \left(\exp\left(\sum_{j=1}^m w_j x_j^{(i)} + w_0\right) - y^{(i)}\right)^2$

$\nabla_w J = 2\sum_{i=1}^n x^{(i)} \exp\left(w^T x^{(i)} + w_0\right)\left(\exp\left(w^T x^{(i)} + w_0\right) - y^{(i)}\right)$

$\nabla_{w_0} J = 2\sum_{i=1}^n \exp\left(w^T x^{(i)} + w_0\right)\left(\exp\left(w^T x^{(i)} + w_0\right) - y^{(i)}\right)$

Networks

Def. (NN) Input $= x \in \mathbb{R}^m$; Output $= a \in \mathbb{R}^n$ (n **Output Units**)

Def. (Feed-Forward NN) Acyclic (neuron input $\perp\!\!\!\perp$ of own output)
+ Data flows one way: inputs $\rightarrow$ outputs
+ $NN(\cdot)$ = composition of each neuron's function

Single Layer: Linear Hypothesis

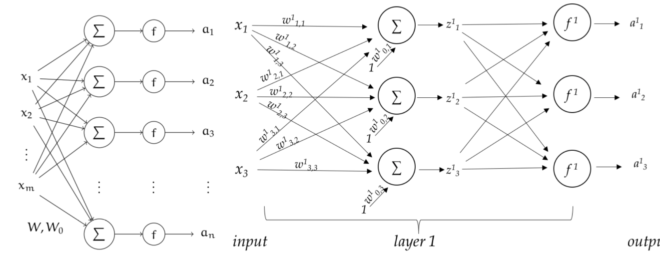
Def. (Layer) Set of non-connected units with:
Input: $x \in \mathbb{R}^m$; **Output/Activation:** $a \in \mathbb{R}^n$
Fully Connected: Same inputs to each layer $x_1^{(i)}, \ldots, x_m^{(i)}$

Layer's Weight Matrix: $W^l \in \mathbb{R}^{m\times n}$ **Offset Vect:** $W_0^l \in \mathbb{R}^{n\times 1}$

Layer Inputs: $X \in \mathbb{R}^{m\times 1}$ **Pre-Activat°:** $Z = W^TX + W_0 \in \mathbb{R}^{n\times 1}$

Activation: $A = f(Z) = f(W^TX + W_0) \in \mathbb{R}^{n\times 1}$ applied element-wise

Note: Single Layer $\iff$ Linear Hypotheses!



Multiple Layers

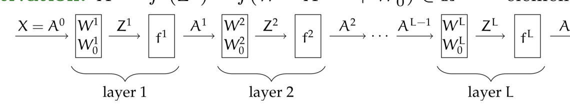
Def. (Layers) Set of non-connected units with:

Layer's Weight Matrix: $W^l \in \mathbb{R}^{m^l\times n^l}$ **Offset:** $W_0^l \in \mathbb{R}^{n^l\times 1}$

Layer Inputs: $A \in \mathbb{R}^{m^l\times 1}$; m^l inputs & $n^l = m^{l+1}$ outputs

Pre-Activat°: $Z^l = W^{l^T}A^{l-1} + W_0^l \in \mathbb{R}^{n^l\times 1}$

Activation: $A^l = f^l(Z^l) = f(W^{l^T}A^{l-1} + W_0^l) \in \mathbb{R}^{n^l\times 1}$ element-wise



Activation Functions

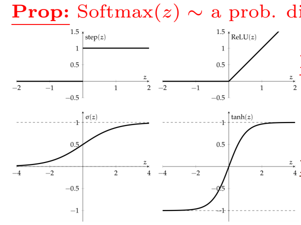
Thm. (No Activation) If $f^l(Z) = Z\ \forall l$ (so activation = identity)
$\implies A^L = W^{L^T}W^{(L-1)^T}\cdots W^{1^T}X = W^{\text{Total}^T}X$
$\implies A^L$ = a linear function of X! One layer is enough

Example: (Activation Functions)
Step Function: $\text{step}(z) = \mathbb{I}\{z \geq 0\}$ (discontinuity $\Rightarrow$ hard for BGD)
Rectified Linear Unit: $\text{ReLU}(z) = \max(0, z)$
Sigmoid/Logistic Function: $\sigma(z) = \frac{1}{1+e^{-z}} \in [0, 1] \sim$ probability

Hyperbolic Tangent: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \in [-1, 1]$

Softmax Funct°: $\text{softmax}(z) = \begin{bmatrix} e^{z_1} / \sum_{i=1}^n e^{z_i} \\ \cdots \\ e^{z_n} / \sum_{i=1}^n e^{z_i} \end{bmatrix} \in [0, 1]^n, \forall Z \in \mathbb{R}^n$

Prop: Softmax$(z) \sim$ a prob. distribution ($(\because)$ $\sum$components = 1)



Prop: $\frac{d}{dz}\text{ReLU}(z) = \text{step}(z)$
$\frac{d}{dz}\sigma(z) = \sigma(z)\cdot(1 - \sigma(z))$
$\frac{d}{dz}\tanh(z) = \frac{4}{(e^z + e^{-z})^2}$

Note: ReLU: use in hidden layers
Sigmoid: binary classification output
Softmax: multi-class classification output

Error Back-Propagation

Note: We will frame it for SGD; For BGD do $\sum_i \nabla_W L^{(i)}$

Idea: (Goal) Compute $\nabla_W L(NN(x;W),y)$, $W := \{W^{(l)}, W_0^{(l)}\}_{l=1}^L$

Proposition (Final Layer) loss $= L(NN(x;W),y) = L(A^L, y)$

$$\implies \underbrace{\frac{\partial \text{loss}}{\partial W^L}}_{m^L \times n^L} = \frac{\partial Z^L}{\partial W^L}\frac{\partial A^L}{\partial Z^L}\frac{\partial \text{loss}}{\partial A^L} = \underbrace{A^{L-1}}_{m^L \times 1}\underbrace{\left(\frac{\partial \text{loss}}{\partial Z^L}\right)^T}_{1 \times n^L}$$

$(\because)$ $A^L = f^L(Z^L)$; $Z^L = {W^L}^T A^{L-1} + W_0^L$

Proposition (Any Layer)
$$\underbrace{\frac{\partial \text{loss}}{\partial W^l}}_{m^l \times n^l} = \underbrace{A^{l-1}}_{m^l \times 1}\underbrace{\left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T}_{1 \times n^l}$$

Proposition (First Layer) Note: $m^{l+1} = n^l$

$$\underbrace{\frac{\partial \text{loss}}{\partial Z^1}}_{n^1 \times 1} = \underbrace{\frac{\partial A^1}{\partial Z^1}}_{n^1 \times n^1}\underbrace{\overbrace{\frac{\partial Z^2}{\partial A^1}}_{m^2 \times n^2}\underbrace{\underbrace{\frac{\partial A^2}{\partial Z^2}}_{n^2 \times n^2} \cdots \underbrace{\frac{\partial A^{L-1}}{\partial Z^{L-1}}}_{n^{L-1} \times n^{L-1}}\underbrace{\frac{\partial Z^L}{\partial A^{L-1}}}_{m^L \times n^L}\underbrace{\frac{\partial A^L}{\partial Z^L}}_{n^L \times n^L}\underbrace{\frac{\partial \text{loss}}{\partial A^L}}_{n^L \times 1}}_{\partial \text{loss}/\partial Z^2}}_{\partial \text{loss}/\partial A^1}$$

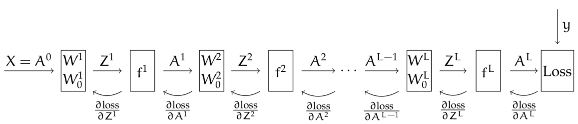
Note: (Dimensions) Recall that:
- $\frac{\partial \text{loss}}{\partial A^L} = n^L \times 1$
- $\frac{\partial Z^l}{\partial A^{l-1}} = W^l = m^l \times n^l$ and $\frac{\partial Z^l}{\partial W^l} = A^{l-1}$, $\frac{\partial Z^l}{\partial W_0^l} = \mathbb{I}_{n^l \times n^l}$
- $\frac{\partial A^l}{\partial Z^l} = W^l = n^l \times n^l$

Thm. (First Layer) We finally get:

$$\underbrace{\frac{\partial \text{loss}}{\partial Z^1}}_{n^1 \times 1} = \underbrace{\frac{\partial A^1}{\partial Z^1}}_{n^1 \times n^1}\underbrace{W^2}_{m^2 \times n^2}\underbrace{\frac{\partial A^2}{\partial Z^2}}_{n^2 \times n^2} \cdots \underbrace{\frac{\partial A^{L-1}}{\partial Z^{L-1}}}_{n^{L-1} \times n^{L-1}}\underbrace{W^L}_{m^L \times n^L}\underbrace{\frac{\partial A^L}{\partial Z^L}}_{n^L \times n^L}\underbrace{\frac{\partial \text{loss}}{\partial A^L}}_{n^L \times 1}$$

Thm. (Any Layer: Error Back-Propagation) We finally get:

$$\underbrace{\frac{\partial \text{loss}}{\partial Z^l}}_{n^l \times 1} = \underbrace{\frac{\partial A^l}{\partial Z^l}}_{n^l \times n^l}\underbrace{W^{l+1}}_{m^{l+1} \times n^{l+1}}\underbrace{\frac{\partial A^{l+1}}{\partial Z^{l+1}}}_{n^{l+1} \times n^{l+1}} \cdots \underbrace{W^L}_{m^L \times n^L}\underbrace{\frac{\partial A^L}{\partial Z^L}}_{n^L \times n^L}\underbrace{\frac{\partial \text{loss}}{\partial A^L}}_{n^L \times 1}$$



Note: Like a "blame propagat°": how much is each module to blame?
- **Forward:** input → output
- **Backward:** input, output, ∂loss/∂output → ∂loss/∂input
- **Weight Grad:** input, ∂loss/∂output → ∂loss/∂W

Training

Idea: (Goal) SGD for a Feed-Forward NN

SGD-NEURAL-NETS $(\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L))$:

```
for l = 1 to L do                                   ▷ Initialize W^l matrix and W_0^l vector
    W_ij^l ~ N(0, 1/m^l)                            ▷ m^l = # of input units
    W_0j^l ~ N(0, 1)
for t = 1 to T do
    i = Unif{1, ..., n} ; A^0 = x^(i)
    for l = 1 to L do                               ▷ Forward pass: get A^L
        Z^l = W^l^T A^(l-1) + W_0^l
        A^l = f^l(Z^l)
    loss = L(A^L, y^(i))
    for l = L to 1 do
        if l = L then                               ▷ Error Back-Propagation
            ∂loss/∂A^l = ∂loss/∂A^L
        else
            ∂loss/∂A^l = ∂loss/∂Z^(l+1) · ∂Z^(l+1)/∂A^l
        ∂loss/∂Z^l = ∂loss/∂A^l · ∂A^l/∂Z^l
        ∂loss/∂W^l = ∂loss/∂Z^l · ∂Z^l/∂W^l = ∂loss/∂Z^l A^(l-1)     ▷ Get Gradient
        ∂loss/∂W_0^l = ∂loss/∂Z^l · ∂Z^l/∂W_0^l = ∂loss/∂Z^l            w.r.t Weights
        W^l = W^l − η(t) · ∂loss/∂W^l                                ▷ SGD Update
        W_0^l = W^l − η(t) · ∂loss/∂W_0^l
return {(W^1, W_0^1), ..., (W^L, W_0^L)}
```

Loss Functions & Activation Functions

Loss	f^L
Squared	Linear
Hinge	Linear
NLL	Sigmoid
NLLM	Softmax

Idea: (Loss Function Matching) Match Loss Function with Activation Function of **last** layer!

Two-Class Classification & Log Likelihood

Idea: For Classification:
✓ Hinge loss: smoother than 0-1 loss (which is not good for BGD)
✓ NLL Loss: nice probabilistic interpretation + extends to multi-class

Assumptions Output layer: Activation function f^l = sigmoid.
Note: So output $a^L \in [0,1] \sim \mathbb{P}(y(x)=1)$, (where labels $y \in \{0,1\}$)

Idea: (Goal) We want to **maximize:**
$\mathbb{P}$(NN assigns correct labels to all inputs)
$= \prod_{i=1}^n (a^{(i)})^{y^{(i)}} \cdot (1-a^{(i)})^{1-y^{(i)}}$ **Note:** Maximize the log!

Def. (Negative Log Likelihood — NLL) Log Loss/Cross Entropy:
$L_{NLL}(a^{(i)}, y^{(i)}) := -\left(y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})\right)$
Objective: $J(\mathcal{D}_n; W) := \sum_{i=1}^n L_{NLL}(a^{(i)}, y^{(i)})$

Multi-Class Classification & Log Likelihood

Assumptions $K = \#$ Classes
Labels: 1-hot vector $y = [y_1, \dots, y_K]^T$ with $y_k = 1$ if $x_k \in$ Class k
Output layer: Activation function f^l = Softmax.
So Output $a^L = [a_1, \dots, a_K]^T \in [0,1]^K \sim$ Prob Dist over all K classes

Idea: (Goal) We want to **maximize:**

$\mathbb{P}$(NN assigns correct class to all inputs) $= \prod_{i=1}^n \prod_{k=1}^K (a_k^{(i)})^{y_k^{(i)}}$
Note: Maximize the log!

Def. (Negative Log Likelihood Multiclass — NLLM)
$L_{NLLM}(a^{(i)}, y^{(i)}) := -\sum_{k=1}^K y_k^{(i)} \log a_k^{(i)}$
Objective: $J(\mathcal{D}_n; W) := \sum_{i=1}^n L_{NLLM}(a^{(i)}, y^{(i)})$
Note: If $k=2$: $y_2 = 1 - y_1$ and $a_2 = 1 - a_1$.

Neural Networks II

Parameter Optimization

Idea: Take advantage of structure of Loss Function + Hypothesis Class to improve optimization of weights.

Batches

Assumptions Objective: $J(\mathcal{D}_n; W) = \sum_{i=1}^n L\left(h(x^{(i)}; W), y^{(i)}\right)$

Idea: Update Rules:
- **BGD:** $W = W - \eta \sum_{i=1}^n \nabla_W L\left(h(x^{(i)}; W), y^{(i)}\right)$
- **SGD:** $W = W - \eta(t) \nabla_W L\left(h(x^{(i)}; W), y^{(i)}\right)$, $i \sim Unif\{1, \dots, n\}$
- **Size k Mini-Batch:** $W = W - \eta \sum_{i=1}^k \nabla_W L\left(h(x^{(i)}; W), y^{(i)}\right)$

Note: BGD = too much computation if n large
SGD = if data has a lot of variability, η must be very small to average moving over competing directions ⇒ slow

Def. (Mini-Batch of size k) Between BGD and SGD!
- Select k datapoints uniformly at random from data $\mathcal{D}_n$
- Update over the batch: $W = W - \eta \sum_{i=1}^k \nabla_W L\left(h(x^{(i)}; W), y^{(i)}\right)$

Note: $k = n \implies$ BGD ; $k = 1 \implies$ SGD

Idea: Randomly shuffle data $\mathcal{D}_n$, and cut into $\sim n/k$ batches of size k.

MINI-BATCH-SGD $(NN, \mathcal{D}_n, k)$:

```
n = length(D_n)
while not terminated do
    Run RANDOM-SHUFFLE (D_n)
    for i = 1 to n/k do
        Run BATCH-GRADIENT-UPDATE (NN, D_n[(i − 1)k : ik])
```

Adaptive Step-Size

Goal: Choose step size η to avoid exploding/vanishing gradients in back-propagation due to multiplication
$\implies$ Use independent step-size parameter for each weight + update based on local view of how the gradients updates have performed
$\implies$ Need different η in each layer & for each weight

Running Averages:
Idea: Estimate a weighted avg of a sequence of data
Input: Sequence $a_1, a_2, \dots, a_T$
Output: Sequence of running avg values $A_0, A_1, \dots, A_T$

Def. (Running Avg) $\begin{cases} A_0 = 0 \\ A_t = \gamma_t A_{t-1} + (1-\gamma_t) a_t \end{cases}$ with $\gamma_t \in (0,1)$

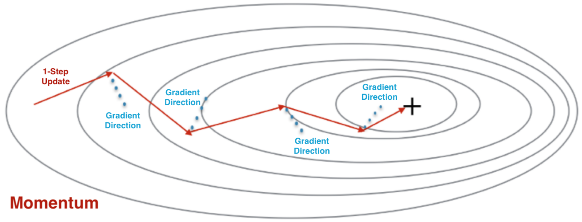
Example: (Moving Avg) Cst $\gamma_t = \gamma$: $A_T = \sum_{t=0}^T \gamma^{T-t} \cdot (1-\gamma) a_t$
Also called **Decaying Average**!
Note: Later inputs a_t have more effect on A_t than early inputs

Example: (Equal-Weighted Avg) $\gamma_t = \frac{t-1}{t}$: $A_T = \frac{1}{T+1} \sum_{t=0}^T a_t$

Momentum:
Idea: Special case of running avg to describe strategies to compute η
Momentum = "avg" recent grad updates to avoid bounce back & forth
$\implies$ Smoothening of trajectory

Def. (Momentum) Start with $V_0 = 0$:
- Def 1: $\begin{cases} V_t = \gamma \cdot V_{t-1} + \eta \cdot \nabla_W J(W_{t-1}) \\ W_t = W_{t-1} - V_t \end{cases}$ Def 1 $\iff$ Def 2: use $\eta := \tilde{\eta} \cdot (1-\gamma)$
- Def 2: $\begin{cases} M_t = \gamma \cdot M_{t-1} + (1-\gamma) \cdot \nabla_W J(W_{t-1}) \\ W_t = W_{t-1} - \tilde{\eta} \cdot M_t \end{cases}$

Note: Def 2: gradient update with step size $\tilde{\eta}$ on Moving Avg of gradients with param γ
Prop: V_t bigger in dims of ∇_θ that consistently have same sign
Note: Need to set 2 extra params: η & $\gamma \longrightarrow$ usually $\gamma \sim 0.9$
Prop: γ small $\implies$ no averaging/usual no-momentum method



Adadelta:
Idea: BGD/SGD can be slow if $J(W)$ has a plateau (flat region)

Goal: Pick large η in flat parts; small η in steep parts
$\Longrightarrow$ Care about **magnitude** of gradient

Def. (Adadelta) In each layer of the NN:

$$\begin{cases} g_{t,j} = \nabla_W J(W_{t-1})_j \\ G_{t,j} = \gamma \cdot G_{t-1,j} + (1-\gamma)\cdot g_{t,j}^2 & \Longrightarrow \text{ large when steep/small when flat} \\ W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j}+\varepsilon}}\cdot g_{t,j} & \longrightarrow \text{ use } \varepsilon \text{ to avoid blow-ups} \end{cases}$$

$G_{t,j}$ = Moving Avg of square (ignore sign) of grad's j^{th} component

Adam:
Idea: Today's default method to manage step sizes η in NN
$\Longrightarrow$ Combine momentum + Adadelta ideas!

Warning: Adam might actually violate SGD convergence conditions!
Paper: `arxiv.org/abs/1705.08292`

Def. (Adam)
Step 1: Moving Avg of Grad & (Grad)^2 $\sim$ mean/var of weight j's grad

$$\begin{cases} m_0 = v_0 = 0 \\ g_{t,j} = \nabla_W J(W_{t-1})_j \\ m_{t,j} = B_1\cdot m_{t-1,j} + (1-B_1)\cdot g_{t,j} \\ v_{t,j} = B_2\cdot v_{t-1,j} + (1-B_2)\cdot g_{t,j}^2 \end{cases}$$

Step 2: Bias-Correction for initializing $m_0 = v_0 = 0$

$$\begin{cases} \hat{m}_{t,j} = \frac{1}{1-B_1^t}\cdot m_{t,j} \\ \hat{v}_{t,j} = \frac{1}{1-B_2^t}\cdot v_{t,j} \end{cases}$$

Step 3: Gradient update $W_{t,j} = W_{t-1,j} - \eta\cdot\frac{1}{\sqrt{\hat{v}_{t,j}+\varepsilon}}\hat{m}_{t,j}$

Suggestion: Use $B_1 = 0.9$, $B_2 = 0.999$, and $\varepsilon = 10^{-8}$
Note: Adam is not very sensitive to (B_1, B_2, ε) parameters
Implement: Store matrix for $(m_t^l, v_t^l, g_t^l, (g_t^l)^2)$ in each layer of NN.

Regularization

Recap: Optimize loss on training data $\Longrightarrow$ overfitting possible
Large Deep NN: a lot of data & params $\sim$ actually not major issue
Still want to make sure that minimizing training loss generalizes well

Methods For Ridge Regression

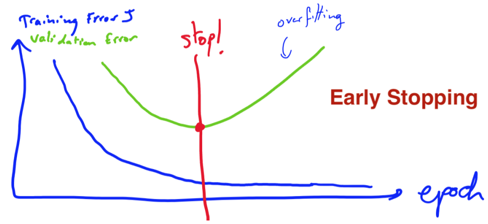
Weight Decay:
Goal: Penalize the norm of all the weights $\sim$ Ridge Regression

Def. (Weight Decay) Objective:
$J(W) = \sum_{i=1}^n L\left(NN(x^{(i)};W)\ ,\ y^{(i)}\right) + \frac{1}{2}\lambda\|W\|^2\ ,\ \ \lambda\in(0,1)$

Proposition (Weight Updates) Using weight decay: $(\eta\in(0,1))$
$W_t = W_{t-1}(1-\lambda\eta) - \eta\cdot\nabla_W L\left(NN(x^{(i)};W_{t-1})\ ,\ y^{(i)}\right)$
$(\because)\ \ W_t = W_{t-1} - \eta\cdot\left[\nabla_W L\left(NN(x^{(i)};W_{t-1})\ ,\ y^{(i)}\right) + \lambda W_{t-1}\right]$
Note: "Decay" W_{t-1} by a factor of $(1-\lambda\eta)$ + take a gradient step

Early Stopping: (Equivalent to Weight Decay)
Def. (Epoch) One pass through training (or could be more)

Def. (Early Stopping) At each epoch: evaluate loss of current W on a validation set. $\Longrightarrow$ Stop when error starts to increase systematically



Noise Addition [Bishop]:(Equivalent to Weight Decay)

Def. (Noise Addition) Perturb the $x^{(i)}$ values of training data:
Add small amount of $N(0,\sigma_{\text{err}}^2)$ noise before each gradient computation
Note: Overfitting $\searrow$ as training data perturbed on each training step

Dropout

Idea: Instead of perturbing data each time: perturb the **network**!
Note: Good for Deep Learning + robust to data perturbation

Def. (Dropout) During training phase, for each training example:
For each unit $\longrightarrow$ randomly pick $a_j^l \sim Ber(1-p) \implies a_j^i \in \{0,1\}$
With prob p: $a_j^l = 0 \Rightarrow$ no contrib to output & no grad update for unit
After training: $\times$ all weights by $p \Rightarrow$ achieve same avg activation levels

Proposition (Dropout Implementation) During Training,
- On each Forward Pass: $a^l = f(z^l) * d^l$, with $d^l \in \{Ber(1-p)\}^{n^l}$
- Backwards Pass: no further changes (depends on a^l anyway)

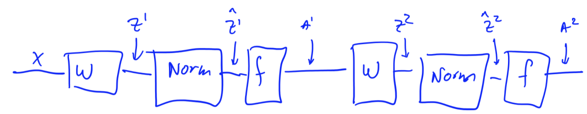
$\longrightarrow$ Common to set $p = 0.5$ **Note:** $*$ = **componentwise** $\times$

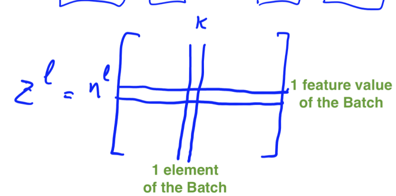
Batch Normalization

Ref. `arxiv.org/abs/1502.03167`

Idea: (Covariate Shift) Input: $X\sim\mathbb{P}_X \implies$ Output $A\sim\mathbb{P}_{X,W}$
But A = input to 2^{nd} layer of NN
$\Longrightarrow$ Distrib of input changes each time we update weights W
$\Longrightarrow$ Standardize input values for each mini-batch!

Note: Batch Normalization has regularization effect!
Each mini-batch of data is mildly perturbed: overfitting $\searrow$





Def. (Batch Norm) Add Batch-Norm Unit before activation module
$Z^l\in\mathbb{R}^{n^l\times K} \longrightarrow \hat{Z}^l\in\mathbb{R}^{n^l\times K} \longrightarrow A^l\in\mathbb{R}^{m^l\times K}$ $\quad (K = \text{batch size})$
Forward Pass: For each feature value $i\in\{1,\ldots,n^l\}$,

$$\begin{cases} \mu_i^l = \frac{1}{K}\sum_{j=1}^K Z_{ij}^l & \Longrightarrow \mu^l\in\mathbb{R}^{n^l\times 1} \\ \sigma_i^l = \sqrt{\frac{1}{K}\sum_{j=1}^K\left(Z_{ij}^l-\mu_i\right)^2} & \Longrightarrow \sigma^l\in\mathbb{R}^{n^l\times 1} \end{cases}$$

$$\Longrightarrow \hat{Z}_{ij}^l := G_i^l\cdot\frac{Z_{ij}^l-\mu_i^l}{\sqrt{(\sigma_i^l)^2+\varepsilon}} + B_i^l \qquad (G_i^l\ \&\ B_i^l \text{ allows for flexibility})$$

Backwards Pass: Given $\frac{\partial L}{\partial\hat{Z}^l}$, want $\begin{cases}\frac{\partial L}{\partial Z^l} & \text{(back-propagation)} \\ \frac{\partial L}{\partial G^l}\ \&\ \frac{\partial L}{\partial B^l} & (W^l \text{ grad updates})\end{cases}$

$$\begin{cases} \frac{\partial L}{\partial G_i^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l}\cdot\frac{\partial\hat{Z}_{ik}^l}{\partial G_i^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l}\cdot\frac{Z_{ik}^l-\mu_i^l}{\sqrt{(\sigma_i^l)^2+\varepsilon}} \\ \frac{\partial L}{\partial B_i^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l}\cdot\frac{\partial\hat{Z}_{ik}^l}{\partial B_i^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l} \end{cases}$$

Thm. (Back-Propagation) Given $\frac{\partial L}{\partial\hat{Z}^l}$: (using $\delta_{ij} = \mathbb{I}\{i=j\}$)

$$\frac{\partial L}{\partial Z_{ij}^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l}\cdot G_i^l\cdot\frac{1}{K\cdot\sigma_i^l}\cdot\left(\delta_{jk}\cdot K - 1 - \frac{(Z_{ik}^l-\mu_i^l)(Z_{ij}^l-\mu_i^l)}{(\sigma_i^l)^2}\right)$$

$(\because)$ $\exists$ dependencies across the batch, not across the unit outputs:

$$\frac{\partial L}{\partial Z_{ij}^l} = \sum_{k=1}^K\frac{\partial L}{\partial\hat{Z}_{ik}^l}\cdot\frac{\partial\hat{Z}_{ik}^l}{\partial Z_{ij}^l}$$

Convolutional Neural Networks

Filters
Max Pooling
Typical Architecture

Sequential Models

State Machines
Markov Decision Processes (MDP)
MDP: Finite-Horizon Solutions
Evaluating a Given Policy
Finding an Optimal Policy
MDP: Infinite-Horizon Solutions
Evaluating an Optimal Policy
Finding an Optimal Policy
Theory

Reinforcement Learning

Bandit Problems
Sequential Problems (SQP)
Model-Based RL
Policy Search
SQP: Value Function Learning & Q-Learning
Q-Learning
Function Approximation
Fitted Q-Learning

Recurrent Neural Networks

RNN Model
Sequence-to-Sequence RNN