UVA DATA SCIENCE

DS 6050 Deep Learning

5. How to Train Convolutional Neural Networks Part II

Sheng Li
Associate Professor
School of Data Science
University of Virginia

Regularization

Preventing weights going crazy…

- Too flexible weights → Too flexible models → Overfitting!

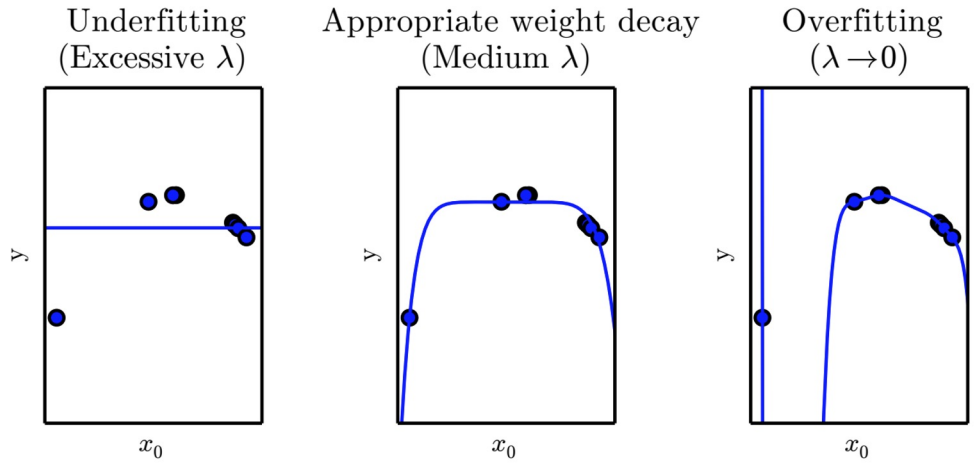
- You can *regularize* (meaning, prevent the size of weights going too big) by adding a regularization term in the loss function.

$$L(W,b) = \frac{1}{N}\sum_{i=1}^{N}\left\|y^{(i)} - f\left(x^{(i)} \mid W,b\right)\right\|^2 + \lambda R(W,b)$$

Regularization

- Regularization is the modification made to a learning algorithm that is intended to reduce its generalization error but not its training error.
- For example, we can modify the loss function for linear regression to include a preference for the weights to have smaller L2 norm as a regularizer:

$$J(\mathbf{w}) = \mathcal{L}_{\text{mse}}(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}}) + \lambda \mathbf{w}^\top \mathbf{w}$$



Parameter Norm Penalties

- Many regularization approaches are based on limiting the model capacity by adding a parameter norm penalty to the objective (loss) function:

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\mathbf{w})$$

Data Loss Norm Penalty

where $\boldsymbol{\lambda}$ is a hyperparameter that controls the contribution of the norm penalty term, $\boldsymbol{\Omega}$, relative to the standard data loss function L

- Larger value of $\boldsymbol{\lambda}$ correspond to more regularization
- Setting $\boldsymbol{\lambda}$ to 0 results in no regularization
- Norm penalty $\boldsymbol{\Omega}$ penalizes only the weights of the affine transformation
- Different choice of $\boldsymbol{\Omega}$ can result in different solutions being preferred.

L2 Parameter Regularization

- L2 regularization is aka Ridge Regression or Tikhonov regularization
- The L2 norm penalty commonly known as weight decay

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \frac{\lambda}{2}||\mathbf{w}||^2$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \mathbf{w}$$

To take a single **Gradient Descent** step to update the weights:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha(\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \mathbf{w})$$

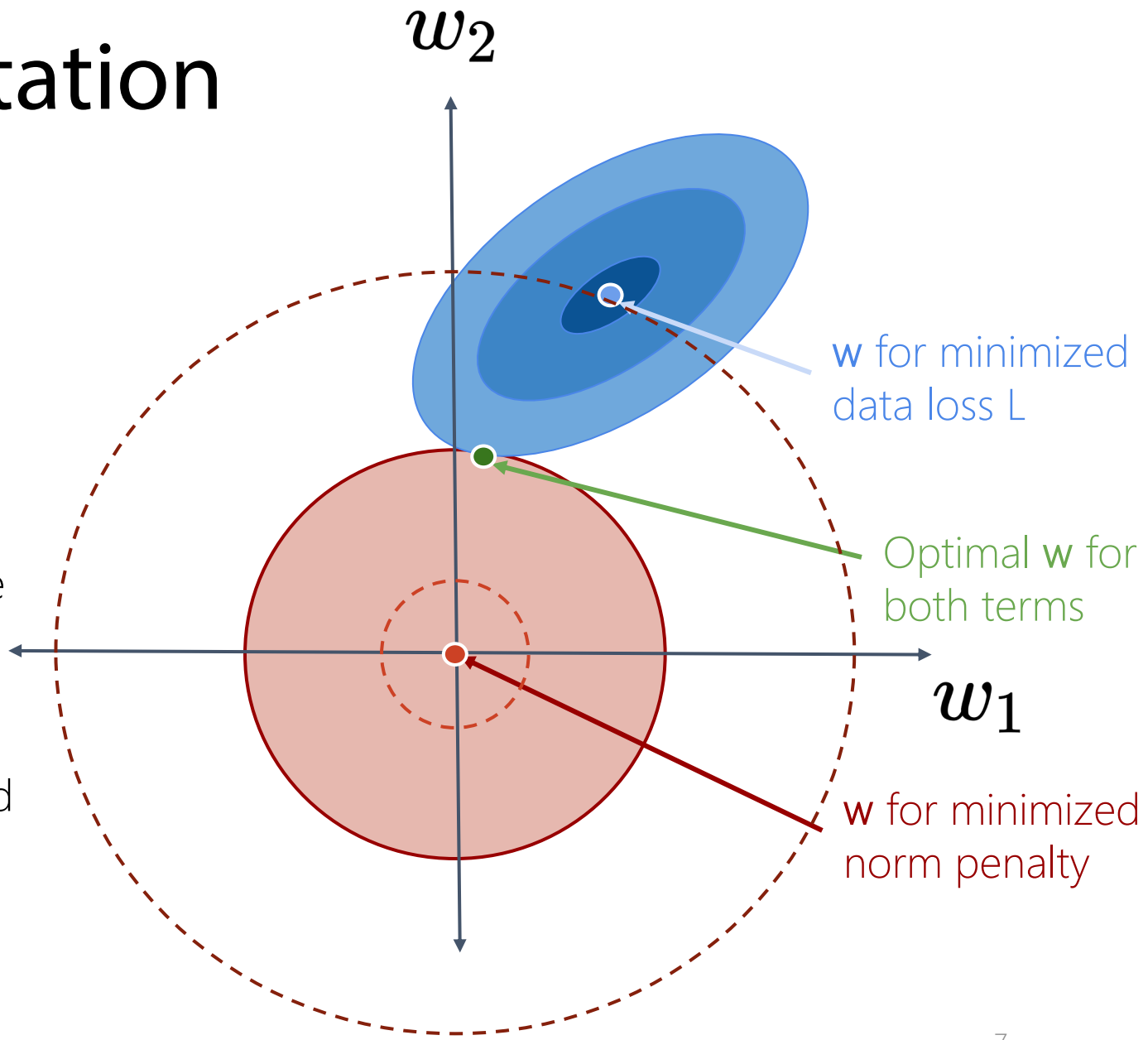
$$\mathbf{w} \leftarrow (1 - \alpha\lambda)\mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$$

< 1

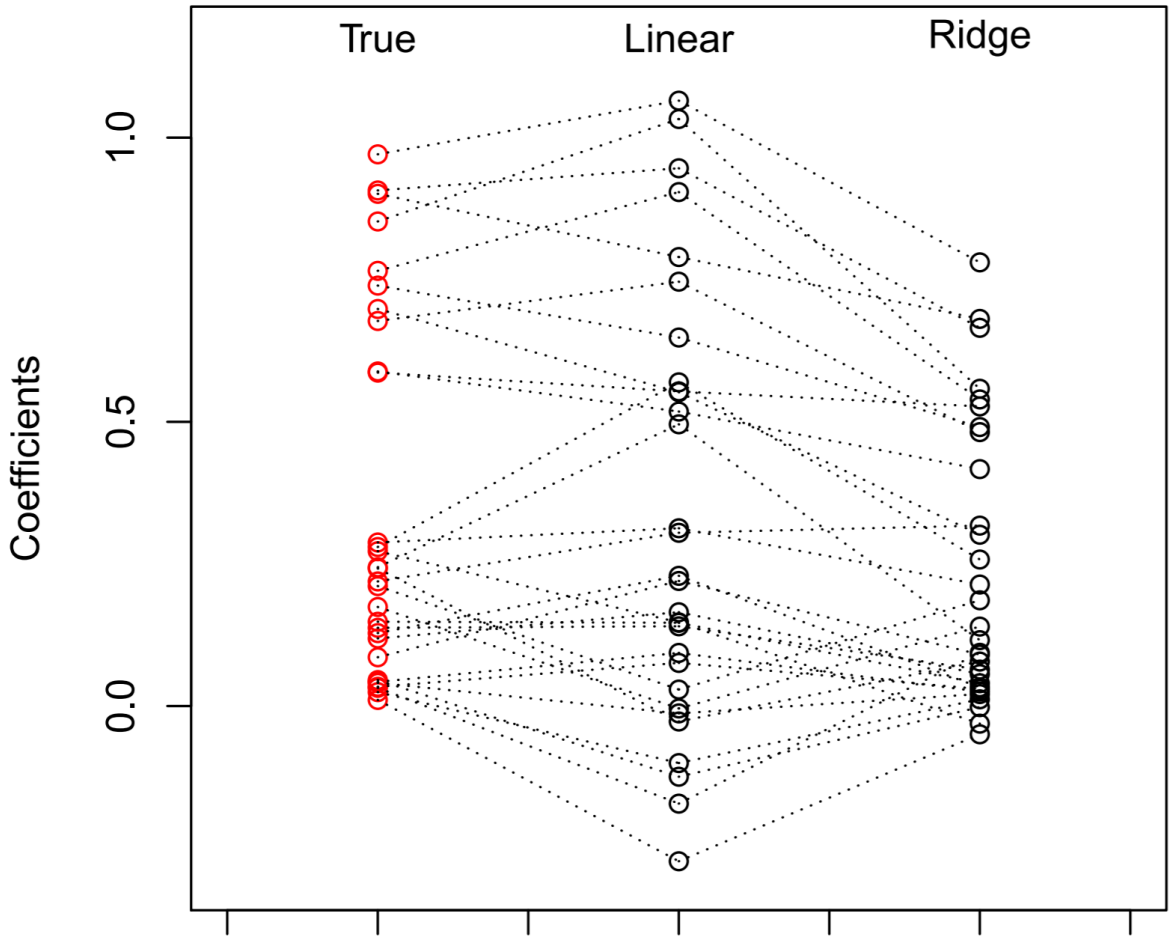
Shrink the weight vector before gradient update

Geometric Interpretation

- L2 regularizations $||w||^2$ can be geometrically represented as concentric (red) circles.
- When circle is too small, the params are not useful to the model.
- When the circle region (in red) grows, due to its shape the region intersects the data contour closer to the origin, L2 makes both parameters shrink and w1 near zero.
- When the circle grows too large, you end up with a similar params as data loss.



Visual Representation of Coefficients



L1 Parameter Regularization

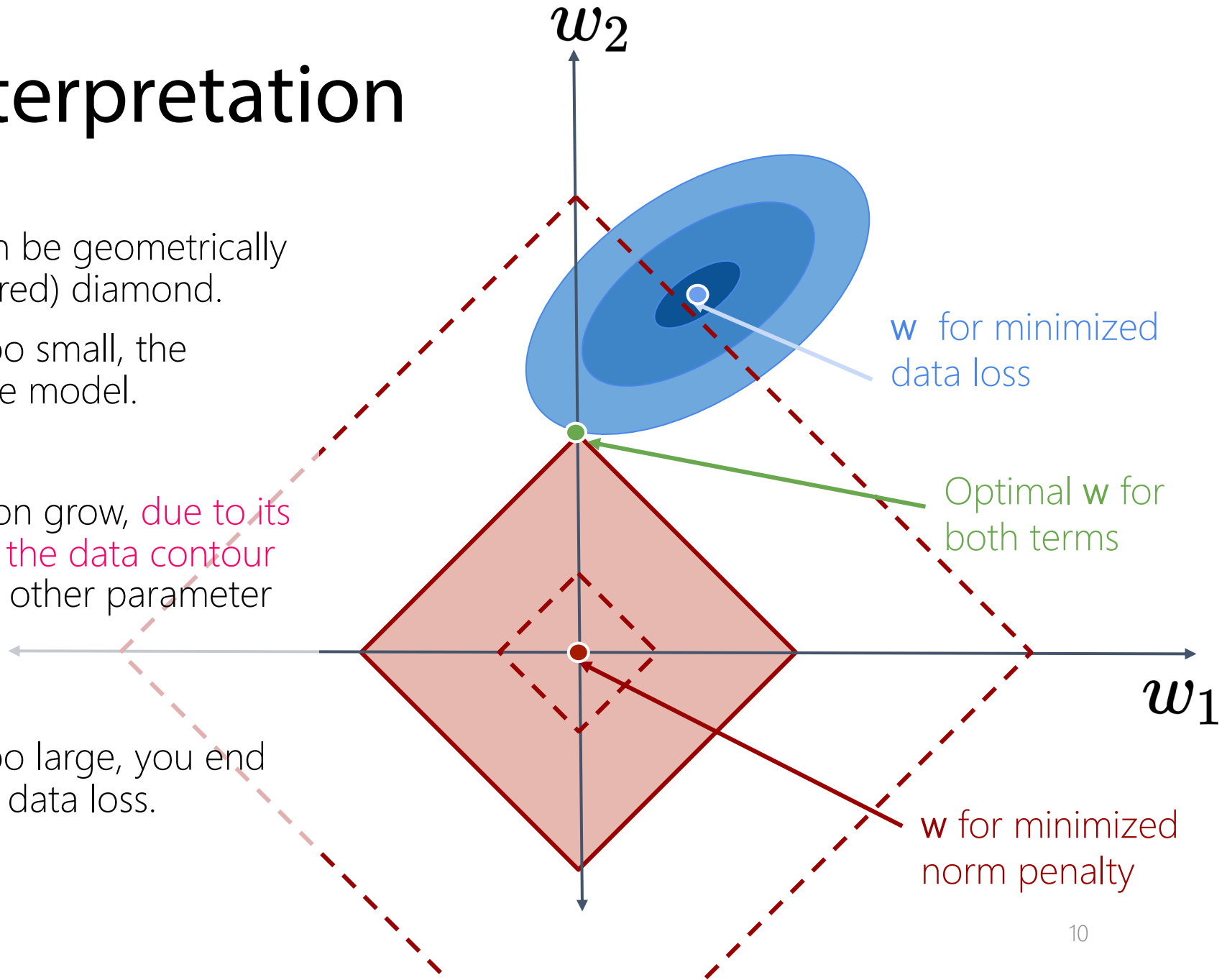
- L1 regularization is aka LASSO (least absolute shrinkage and selection operator)
- L1 norm commonly is known as the Manhattan Distance.

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda ||\mathbf{w}||_1$$
$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \mathbf{sign}(\mathbf{w})$$

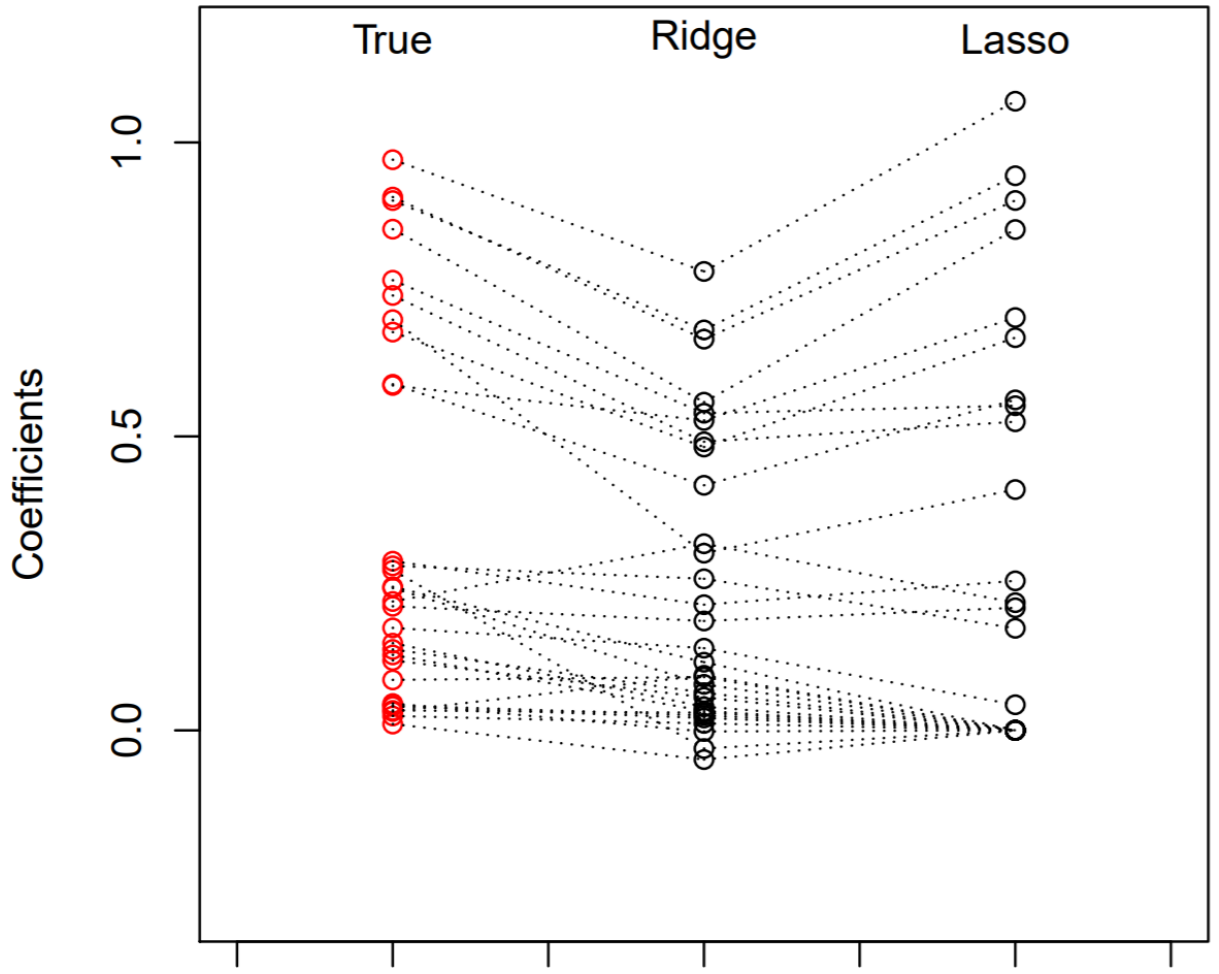
- The regularization to the gradient no longer scale linearly with each $\mathbf{w}$, instead it is a constant factor with a sign equal to $\mathbf{sign}(\mathbf{w})$. Thus, there is no clean algebraic solution to approximates J as we have just seen in L2 regularization
- L1 norm makes the parameters to become sparse (contains lots of zeros)
- L1 norm tends to cause a subset of the network weights to become zero, suggesting that the corresponding signal may safely be discarded (dead neuron).

Geometric Interpretation

- L1 regularizations $||w||^1$ can be geometrically represented as concentric (red) diamond.
- When diamond region is too small, the params are not useful to the model.
- When as the diamond region grow, due to its shape the region intersects the data contour on one of the axis, thus the other parameter is zero out.
- When diamond region is too large, you end up with a similar params as data loss.

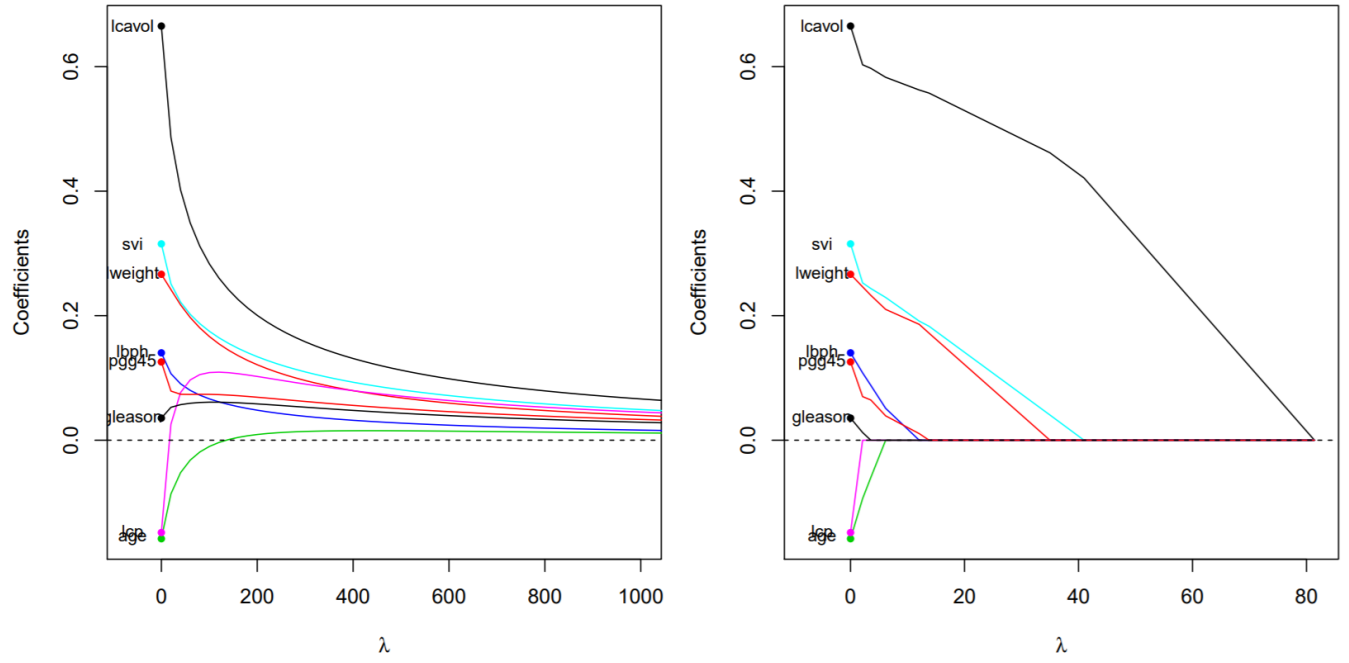


Visual Representation of Coefficients



Example of Variable Selection

- Example: suppose that we are studying the level of prostate-specific antigen (PSA), which is often elevated in men who have prostate cancer. We look at n = 97 men with prostate cancer, and p = 8 clinical measurements. We are interested in identifying a small number of predictors, say 2 or 3, that drive PSA



Ridge Regression

LASSO

Elastic Net

- Is a middle ground between Ridge (L2) and Lasso (L1) with a ratio r.

$$J(\mathbf{w};\mathbf{X},\mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}},\mathbf{y}) + r\lambda||\mathbf{w}||_1 + (1-r)\lambda||\mathbf{w}||_2^2$$

LASSO

RIDGE

Geometric Interpretation

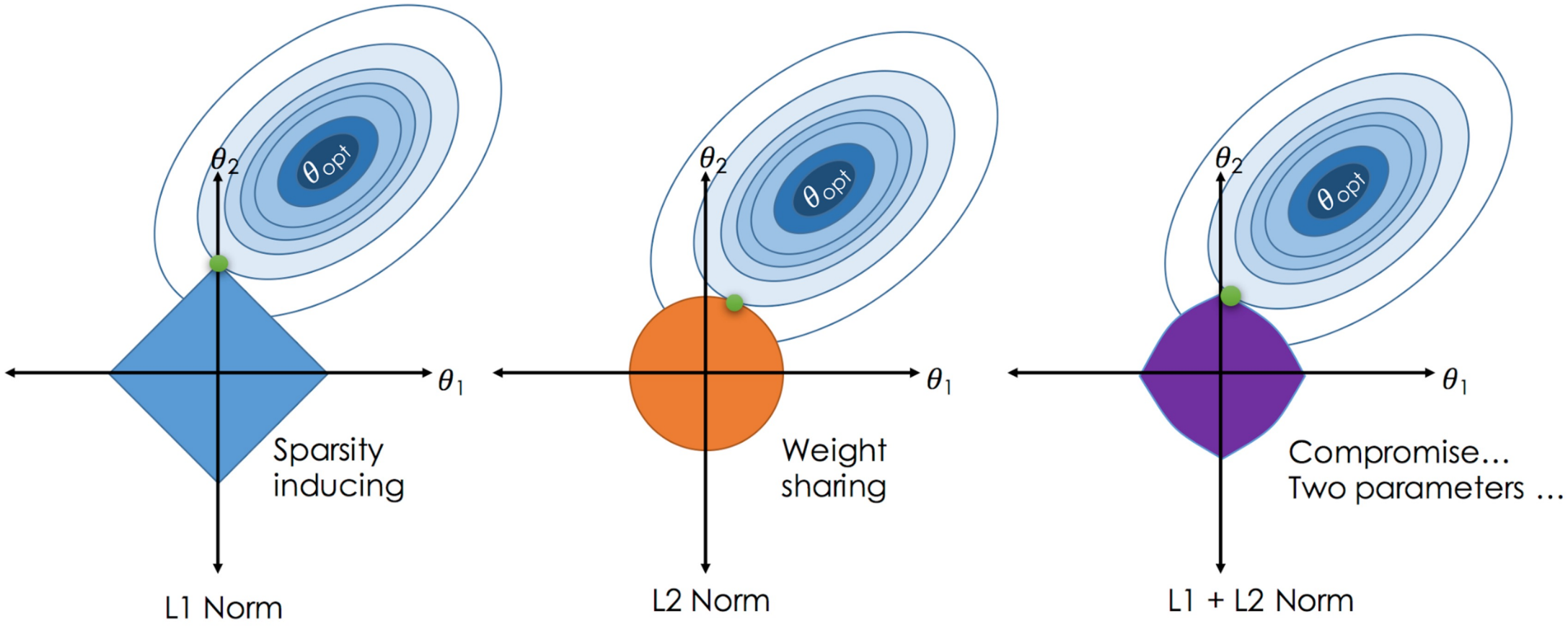
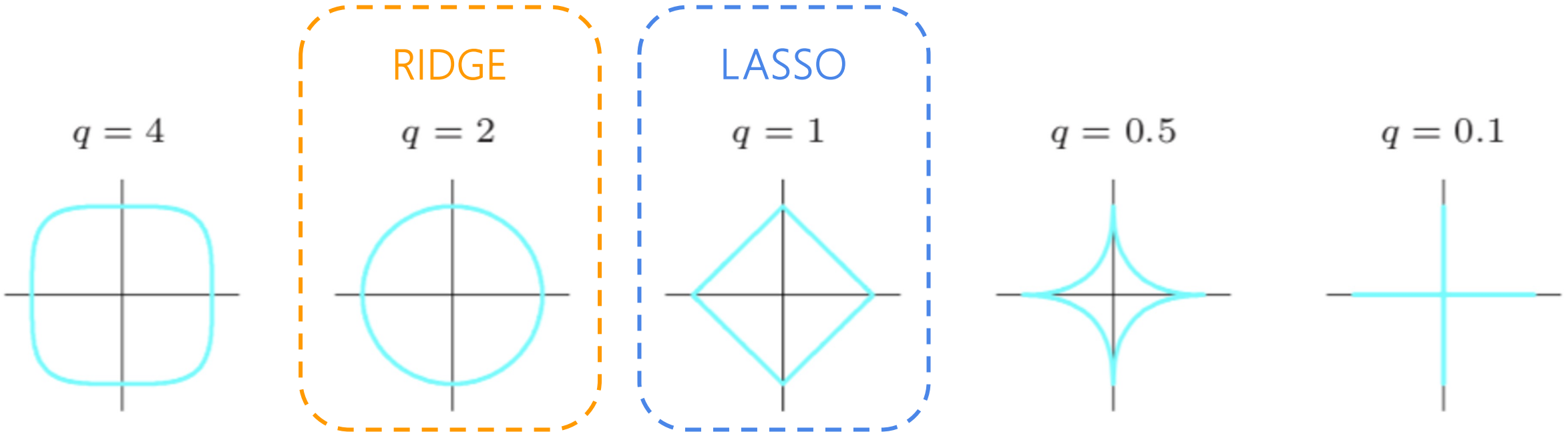


*Figure from Yin Hu, Medium

Family of Parameter Norm Models

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda ||\mathbf{w}||_q$$



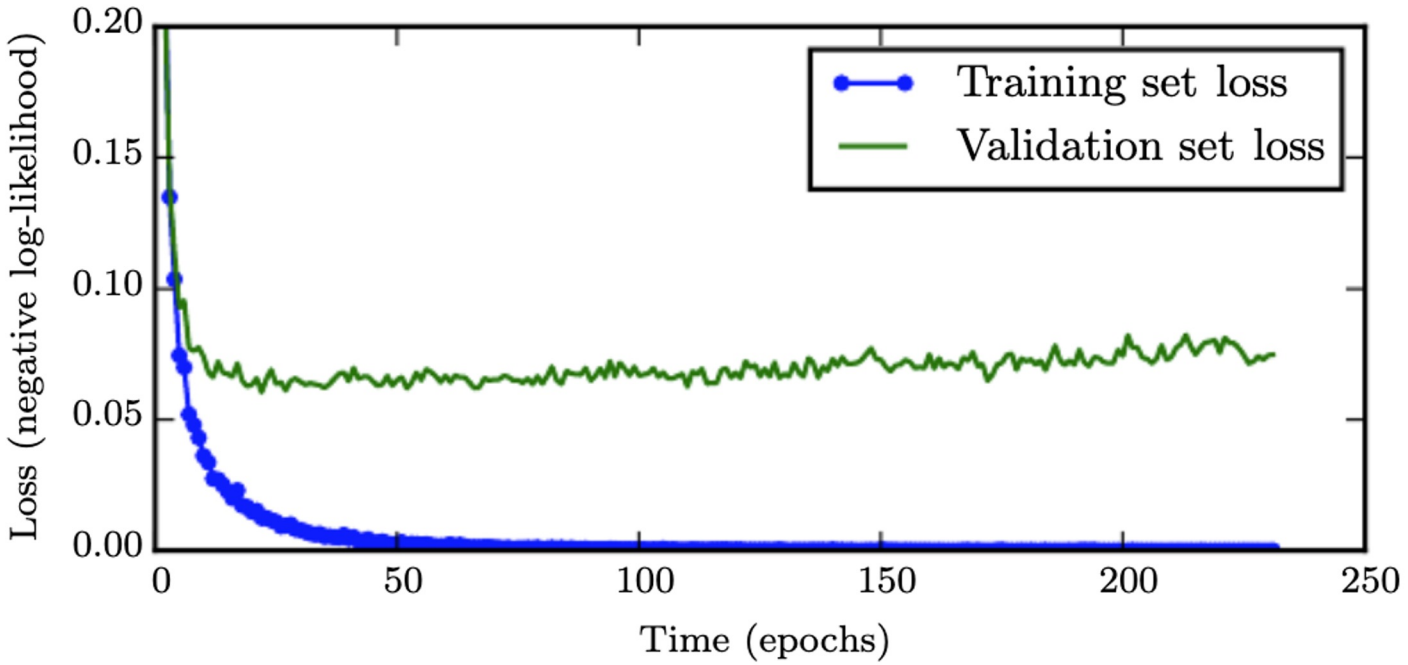
TL;DR

- It is preferable to have at least a little bit of regularization. Ridge (L2) is a good default.
- If you suspect that only a few features are actually useful, use Lasso (L1)
- Lasso may behave erratically when # features > # examples → use Elastic Net

Early Stopping

Early Stopping

- When training large models with sufficient capacity to overfit the task, we often observe reliably that training error decreases steadily over time, but validation error begins to rise again → we can stop at the lowest validation error!
- The most commonly used form of regularization in deep learning: simple yet effective!



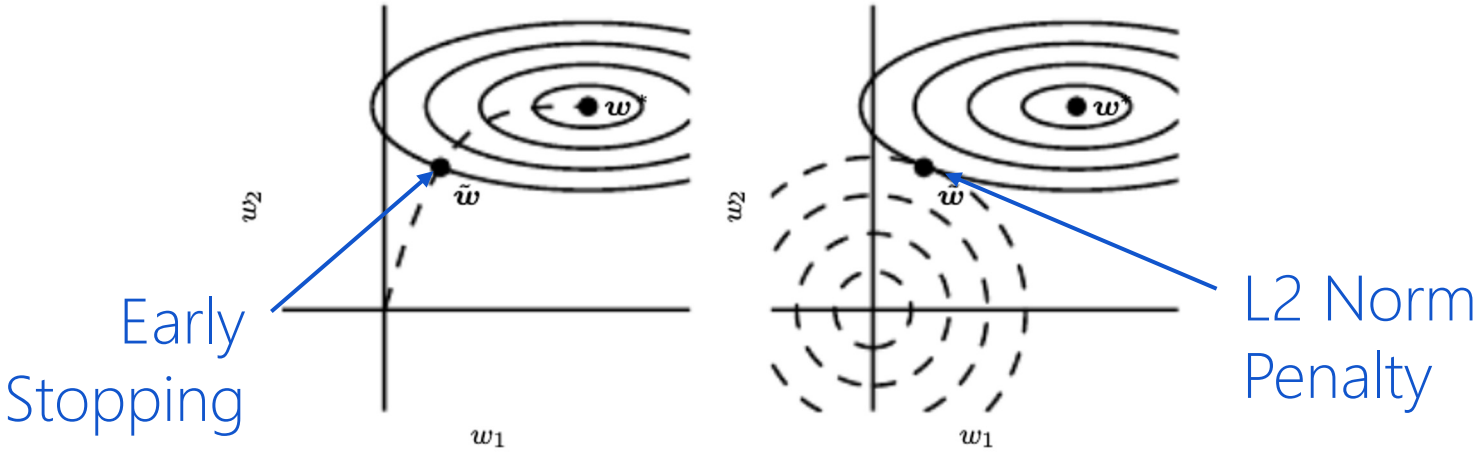
Early Stopping: Algorithm

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*.

Early Stopping: Properties

- Is a very efficient hyperparameter selection method (saves the best set).
- Controls the model capacity by determining how many steps to fit the training data (and reduce the computational cost of the training procedure)
- Unobtrusive form of regularization as almost no change in training procedure
- Can be used either alone or with other regularization strategies
- Is equivalent to L2 regularization, and better because it automatically determines the correct amount of regularization while L2 requires the tuning of hyperparameter $\boldsymbol{\lambda}$



Loss Functions

Lots of Puzzling Terms!

- L1 Loss: `torch.nn.L1Loss(y_true, y_pred)`
- Mean Squared Error (MSE): `torch.nn.MSELoss(y_true, y_pred)`
- Cross Entropy: `torch.nn.CrossEntropyLoss(y_true, y_pred)`
- Kullback-Leibler Divergence: `torch.nn.KLDivLoss(y_true, y_pred)`
- Hinge Embdding Loss: `torch.nn.HingeEmbeddingLoss(y_true, y_pred)`
- Huber Loss: `torch.nn.HuberLoss(y_true, y_pred)`
- Margin Ranking Loss: `torch.nn.MarginRankingLoss(y_true, y_pred)`
- Multi-label Margin Loss: `torch.nn.MultiLabelMarginLoss(y_true, y_pred)`
- Cosine Embedding Loss: `torch.nn.CosineEmbeddingLoss(y_true, y_pred)`
- Triplet Margin Loss: `torch.nn.TripletMarginLoss(y_true, y_pred)`
- Negative Log-likelihood Loss: `torch.nn.NLLLoss(y_true, y_pred)`
- Gaussian Negative Log-likelihood Loss: `torch.nn.GaussianNLLLoss(y_true, y_pred)`

Information Theory (Claude Shannon, 1948)

- Digital information: series of bits (either 0 or 1)
- Sending a single bit (useful) of information = reducing receiver's uncertainty into half
- For example, who would win the Virginia–Virginia Tech game, assuming the both team have the equal chance of winning (50%)?
  - With no information, the uncertainty is 50-50.
  - One bit of information (Cavaliers won!) → resolves the uncertainty.
- Another example, imagine a league of 8 teams. Who is the winner?
  - With no information, the uncertainty is 12.5% each.
  - How many bits of information do you need?
  - 3 bits! ($2^3 = 8$, or $\log_2(8)=3$)

Information Theory (Claude Shannon, 1948)

- What if the chance of winning is not equal?
  - Say, Cavaliers (75% of winning) and Hokies (25% of winning).
  - If a sender says, Hokies won the game, the uncertainty drops by the factor of 4.
    - Uncertainty reduction = $-\log_2(1/4) = 2$
  - If a sender says, Cavaliers won the game,
    - Uncertainty reduction = $-\log_2(3/4) = 0.42$
  - Therefore, the expected number of bits to resolve the uncertainty:
    - $-0.75*\log_2(3/4) - 0.25*\log_2(1/4) = 0.75*0.42 + 0.25*2 = 0.82$ bits

- Entropy: $H(p) = -\sum_i p_i \log_2(p_i)$
  - Average amount of information that can be derived from one sample drawn from a given probability distribution
  - Indicator of how unpredictable the probability distribution is.
  - More variation in the data → larger entropy.

Cross Entropy

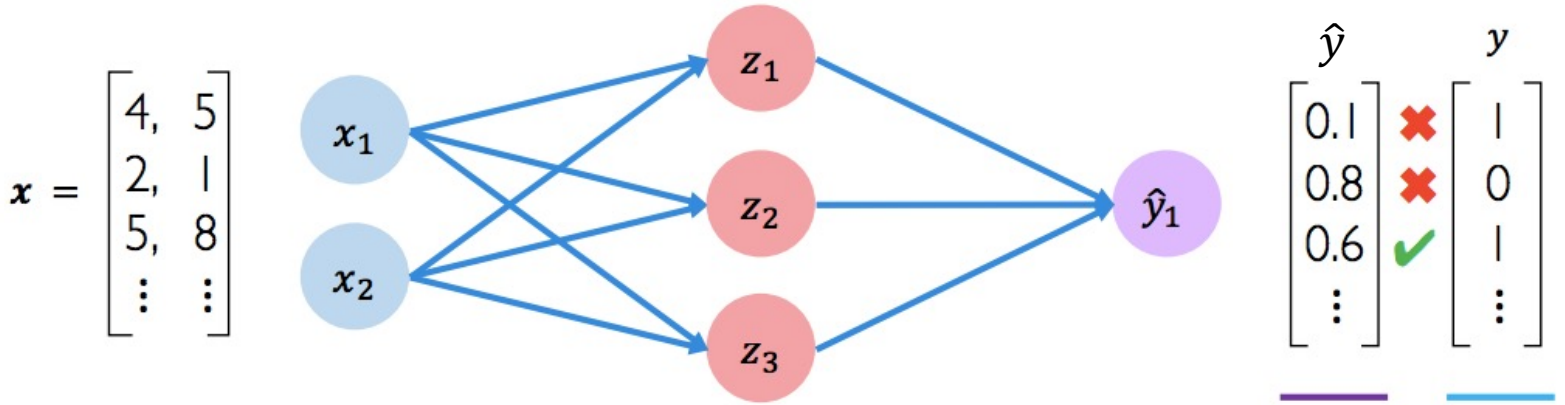
- Entropy: $H(p) = -\sum_i p_i \log_2(p_i)$
- Cross Entropy: $H(p, q) = -\sum_i p_i \log_2(q_i)$
  - Think of p as a true distribution and q as a predicted distribution.
  - If $q = p$ (correct prediction), cross entropy equals to entropy.
  - If $q \neq p$ (erroneous prediction), cross entropy gets greater (why?) than entropy by some number of bits

https://en.wikipedia.org/wiki/Gibbs'_inequality

- Kullback-Leibler (KL) Divergence: $D_{KL}(p, q) = H(p, q) - H(p)$
  - The amount of difference between cross entropy and entropy.
  - a.k.a, relative entropy

Cross Entropy Loss

- Binary Cross Entropy Loss:



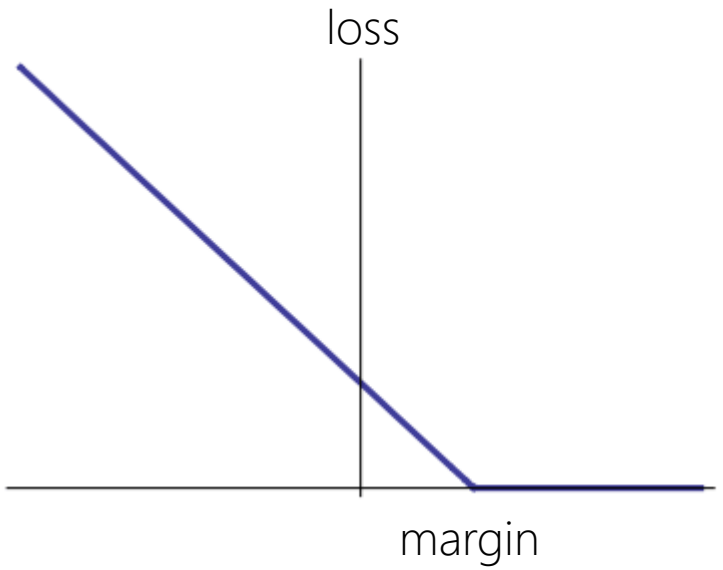
$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}(Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

- Multiclass Cross Entropy Loss:

$$L(\hat{y}, y) = -\sum_{k}^{K} y^{(k)} \log \hat{y}^{(k)}$$

Hinge Loss

- Penalizes incorrectly classified examples + correctly classified examples that lie within the margin
- Hinge loss is generally faster than cross entropy but less accurate.



$L = \max(0, 1 - ty)$

Regression: L_p Distances

- L_p norm or Minkowski distance:

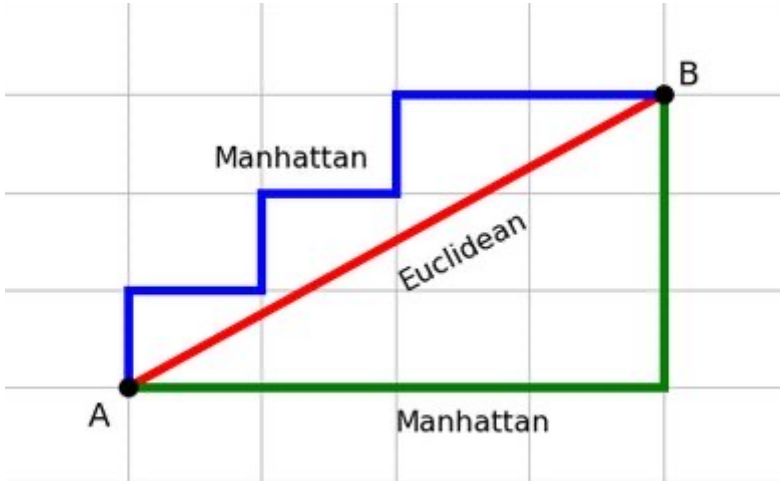
$$L_p(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

- For $p = 1$, Manhattan (or city-block) distance:

$$L_1(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

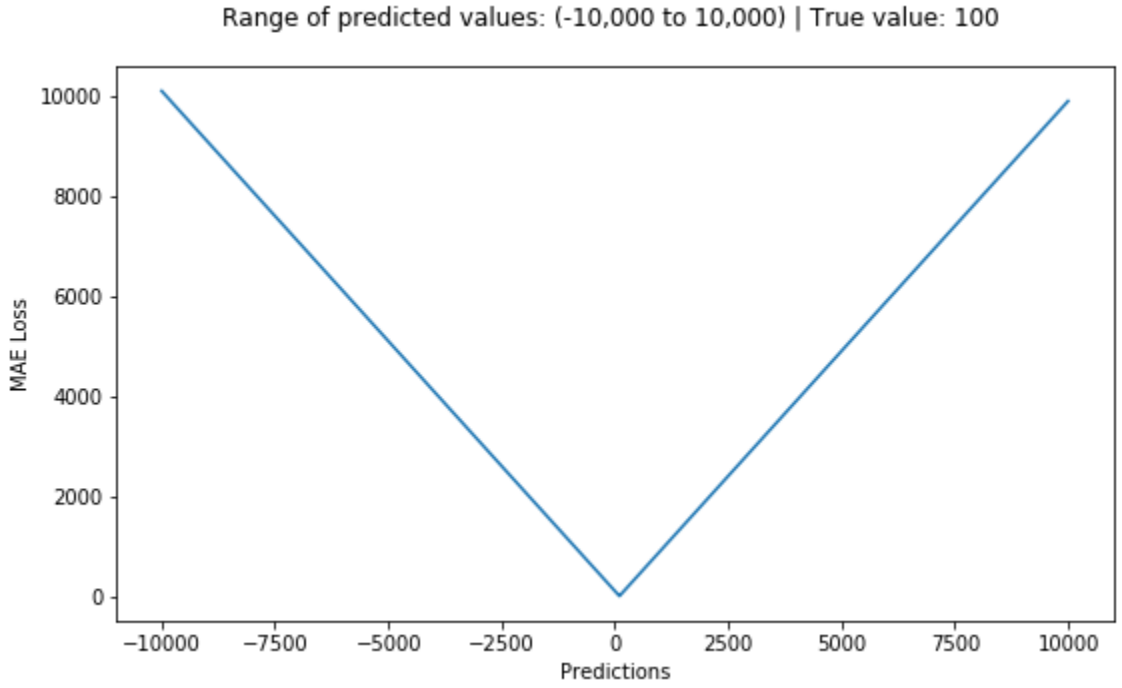
- For $p = 2$, Euclidean distance:

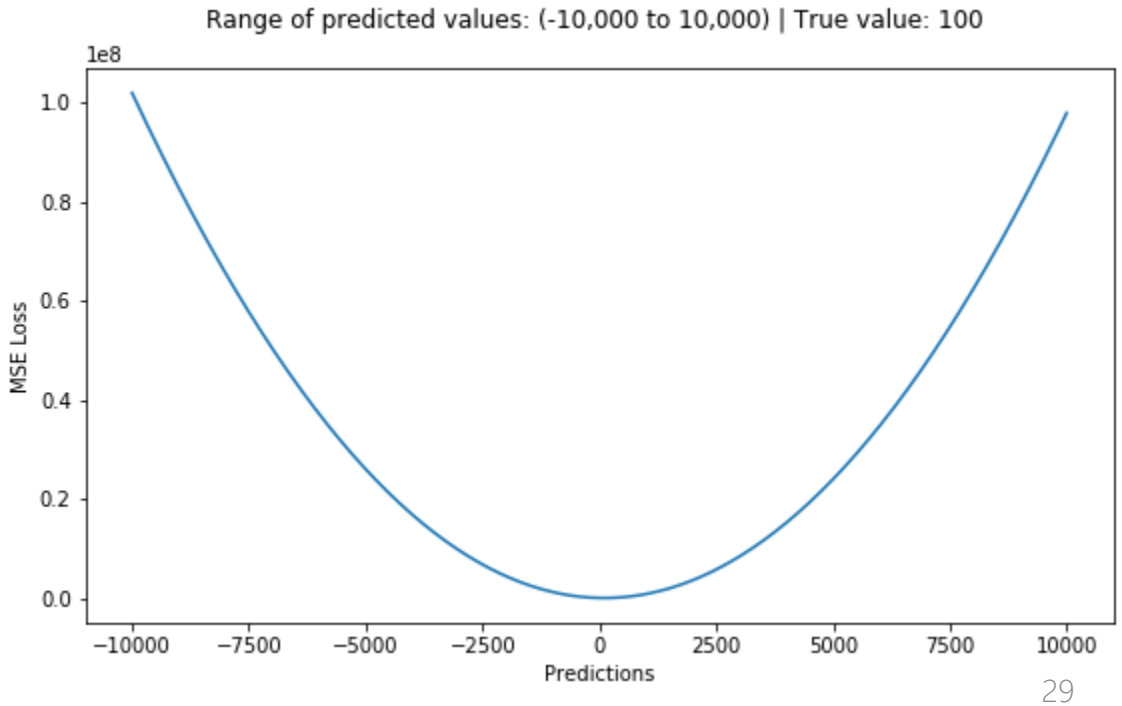
$$L_2(x, y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$



L1 and L2 Loss

- Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Mean Squared Error (MSE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2$

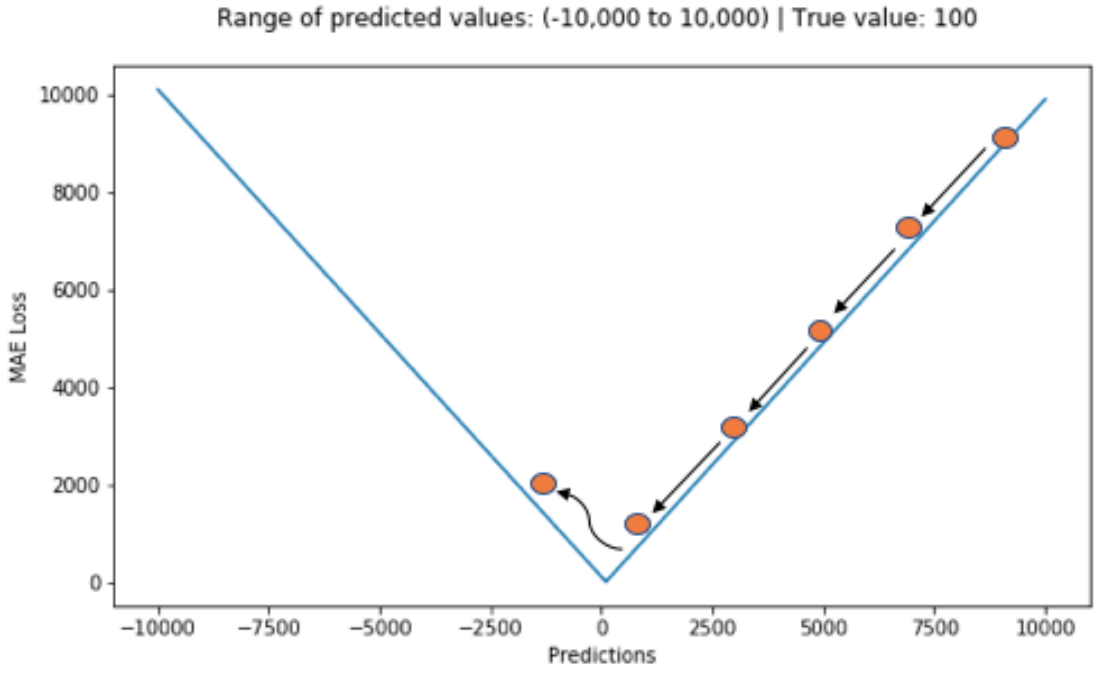


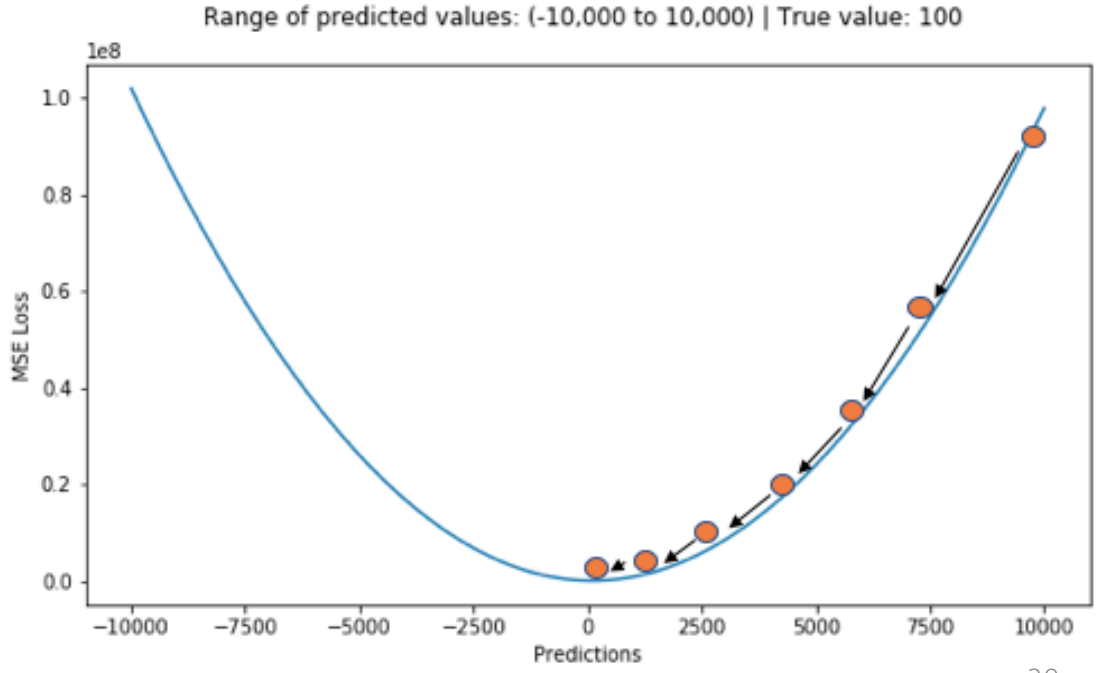


https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

L1 and L2 Loss

- Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Mean Squared Error (MSE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2$

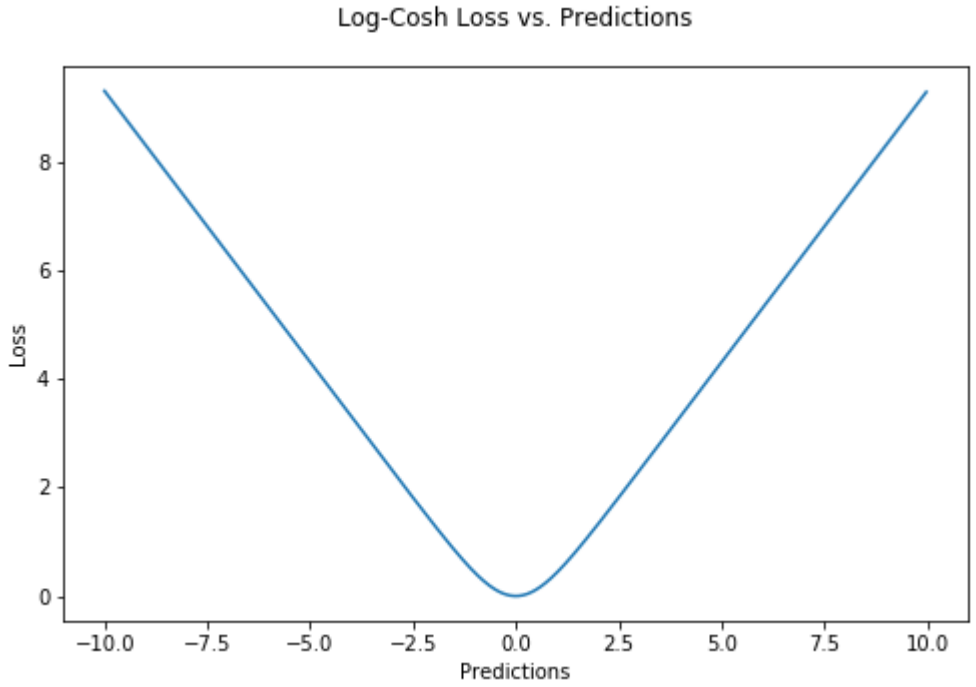




https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

Logarithm of Hyperbolic Cosine (logcosh)

- Logcosh: $\frac{1}{n}\sum_{i=1}^{n} \log\cos(\hat{y}_i - y_i)$



https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

Beyond SGD:
Fancier Optimization Methods

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

- A. The function value $f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right)$ need to be evaluated for all $\mathbf{x}^{(i)}$ in the dataset.

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

- A. The function value $f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right)$ need to be evaluated for all $\mathbf{x}^{(i)}$ in the dataset.

- Q. What about the gradient?

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

- A. The function value $f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right)$ need to be evaluated for all $\mathbf{x}^{(i)}$ in the dataset.

- Q. What about the gradient?
- A. Same! Backpropagation needs to happen for all and each $\mathbf{x}^{(i)}$

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

- A. The function value $f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right)$ need to be evaluated for all $\mathbf{x}^{(i)}$ in the dataset.

- Q. What about the gradient?
- A. Same! Backpropagation needs to happen for all and each $\mathbf{x}^{(i)}$

- Q. Then what happens to the gradient descent algorithm?

Thought Experiment on Gradient Descent

- Q. What does it take to evaluate loss?

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right) \right\|^2$$

- A. The function value $f\left(\mathbf{x}^{(i)} | \, \boldsymbol{\theta}\right)$ need to be evaluated for all $\mathbf{x}^{(i)}$ in the dataset.

- Q. What about the gradient?
- A. Same! Backpropagation needs to happen for all and each $\mathbf{x}^{(i)}$

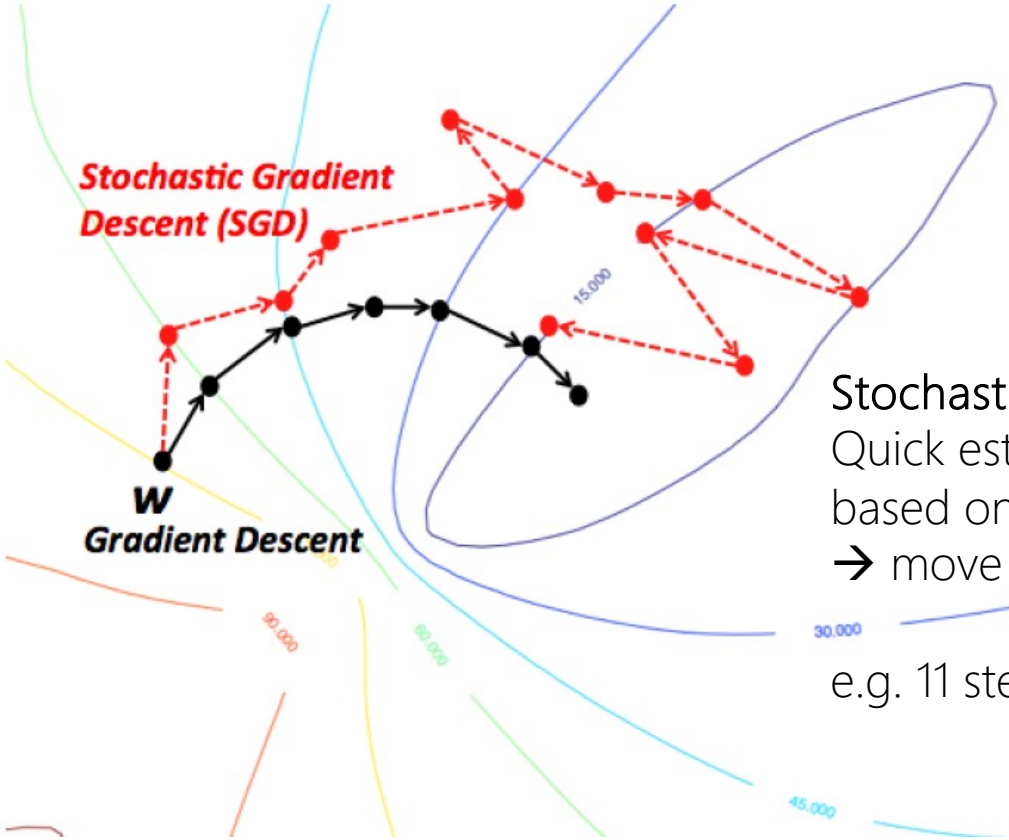
- Q. Then what happens to the gradient descent algorithm?
- A. Computational time increases exponentially as N goes up.

Stochastic Gradient Descent (SGD)

- Idea:

Gradient Descent
Compute everything
→ make the optimal one step

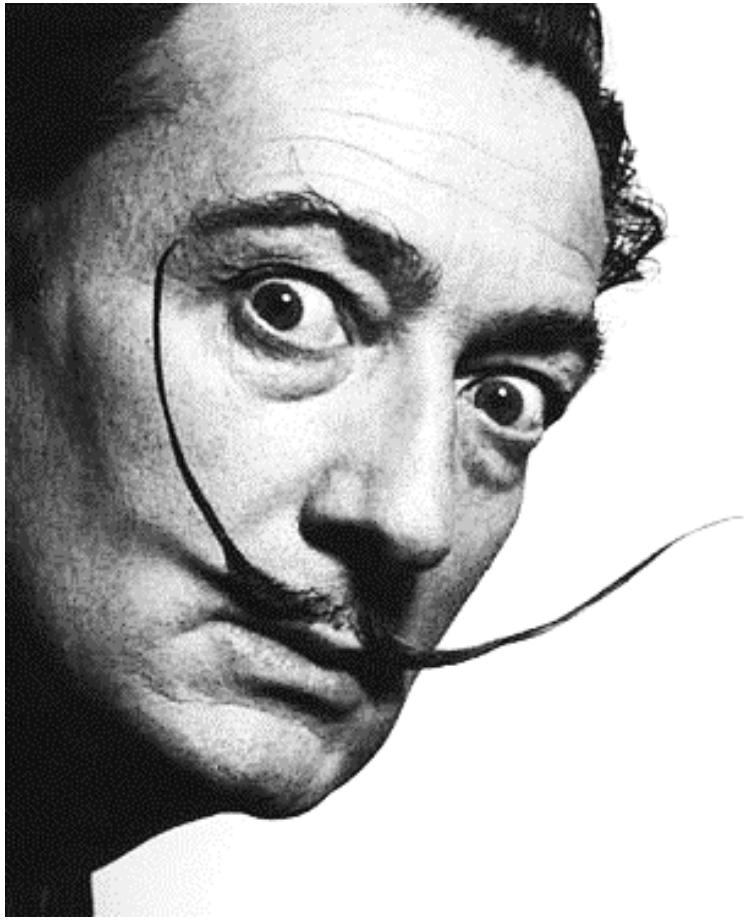
e.g. 6 steps * 1 hr/step = 6 hrs



Stochastic Gradient Descent
Quick estimation of the gradient
based on a sample (or a small batch)
→ move quickly even if it's not the optimal

e.g. 11 steps * 5 min/step = 55 min

Stochastic Gradient Descent (SGD)



"Have no fear of perfection,
you'll never reach it"

- Salvador Dali

TwistedSifter.com

Stochastic Gradient Descent (SGD)

1. Randomly shuffle dataset
2. Repeat until converge {
   randomly pick one sample
   compute gradient with the selected sample
   update weights
   }

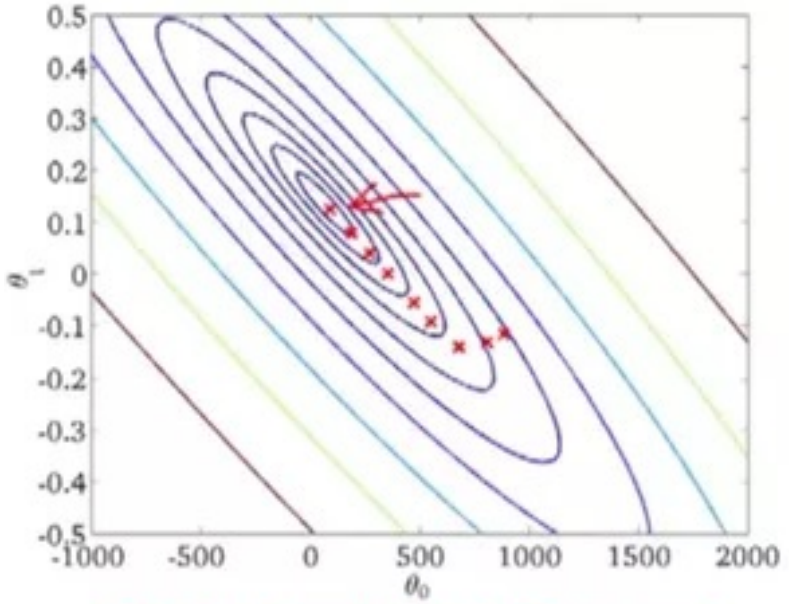
Mini-Batch SGD

1. Randomly shuffle dataset
2. Repeat until converge {
    for a mini-batch {
        compute gradient only with the mini-batch
        update weights
    }
}

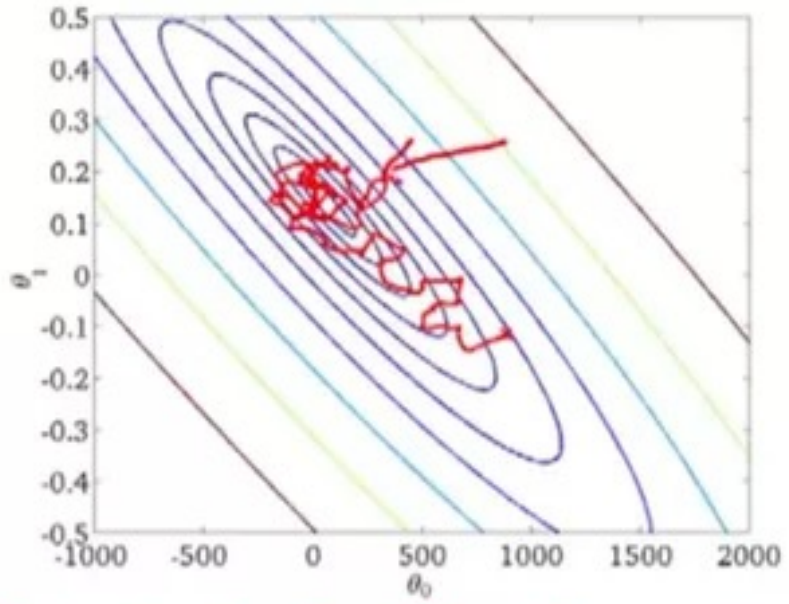
Note: In deep learning literature, we often use **mini-batch SGD** and **SGD** interchangeably.

Stochastic Gradient Descent (SGD)

- Gradients come from single samples, so they can be noisy and inaccurate!



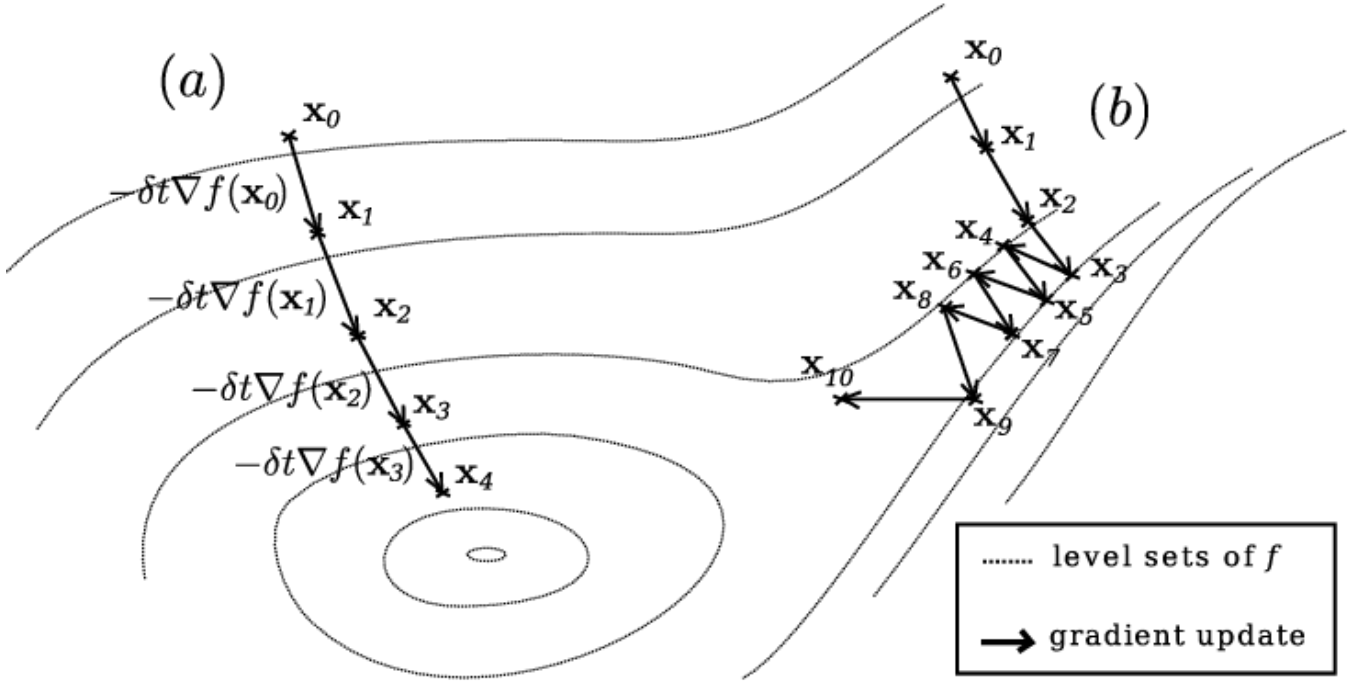
Batch Gradient Descent



Stochastic Gradient Descent

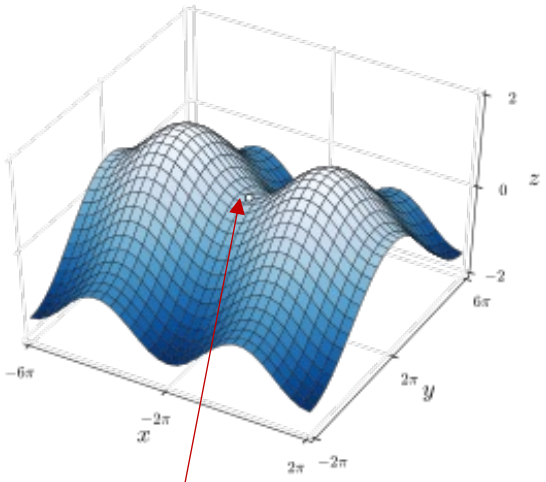
Problems of SGD

- "long-narrow valley"
  - What if your loss function happens to be steep in one direction but "flat" in another…
  - Condition number: ratio of largest to smallest singular value of the Hessian.
  - Large condition number → long-narrow valley

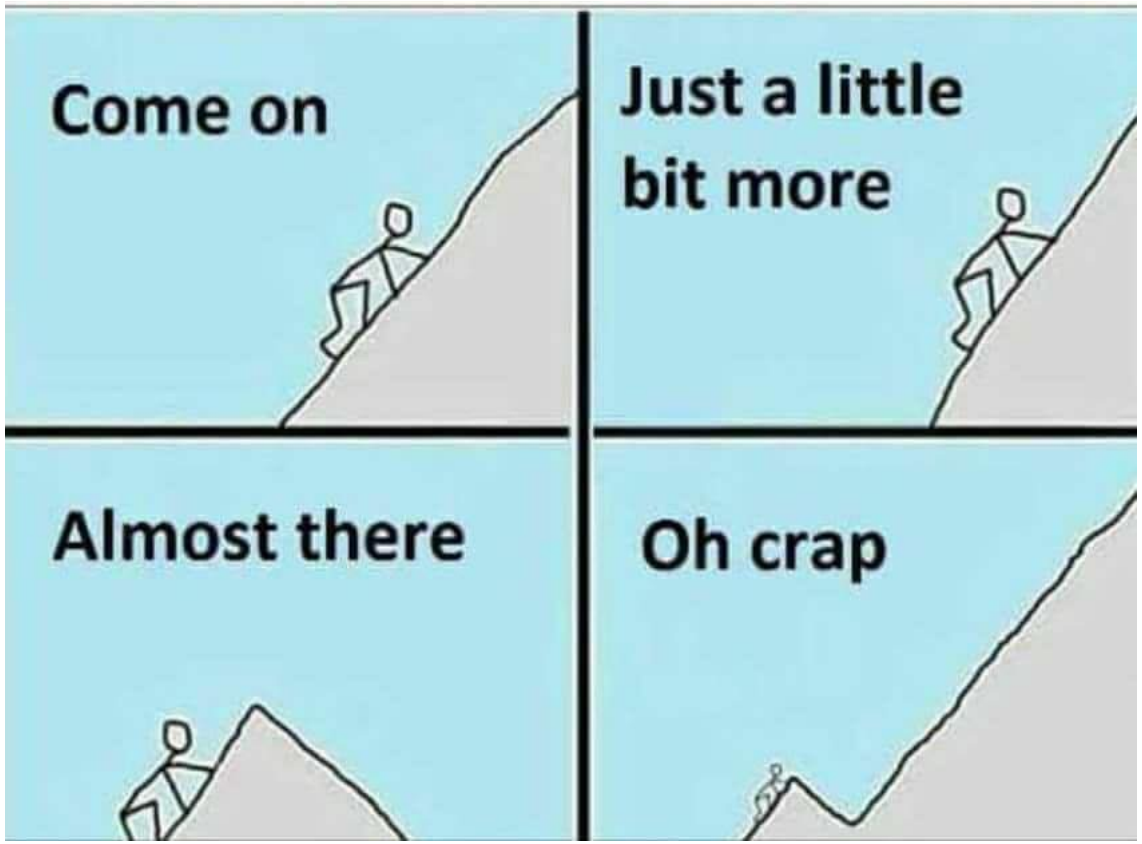


Problems of SGD

- Local minima or Saddle points → zero gradient! → No update (gets stuck)



Saddle Point



Momentum

- Idea: let's build up a velocity (momentum)!
- SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

- SGD + Momentum

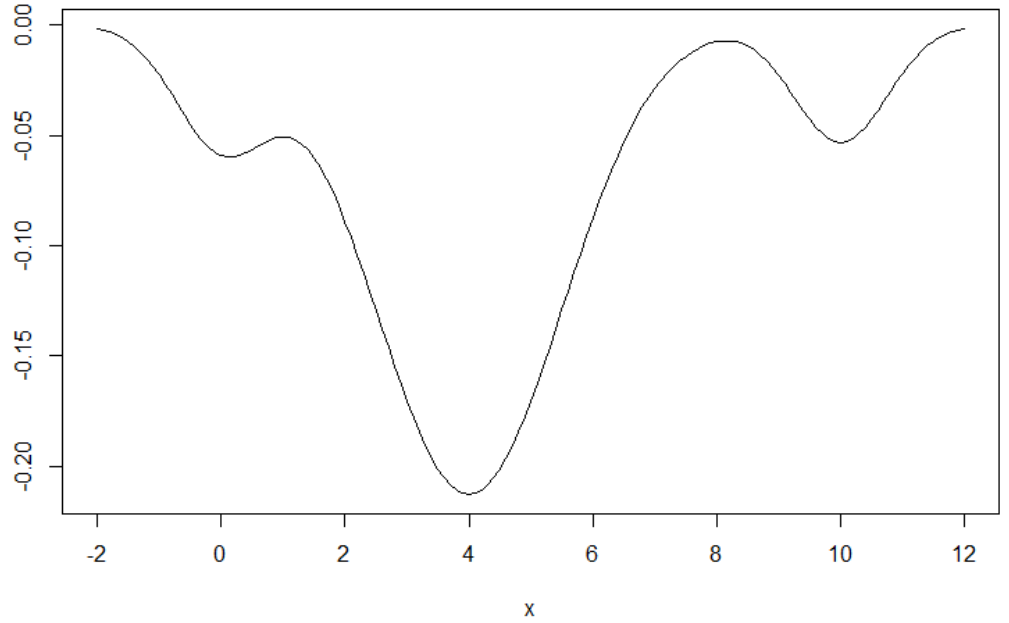
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

  - ρ: "friction" or "drag". Causes decrease of velocity. Typically, 0.9 or 0.99

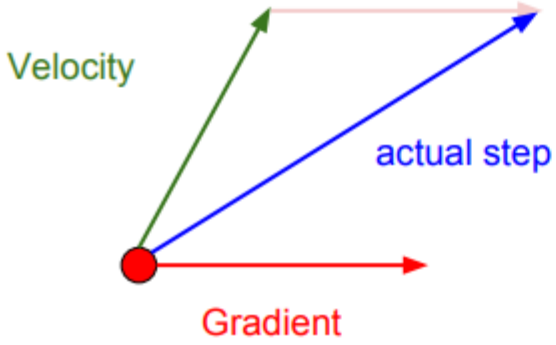
SGD + Momentum

- Discuss:
  - High condition number (long-narrow valley)
  - Local minima and saddle points
  - Noisy gradient



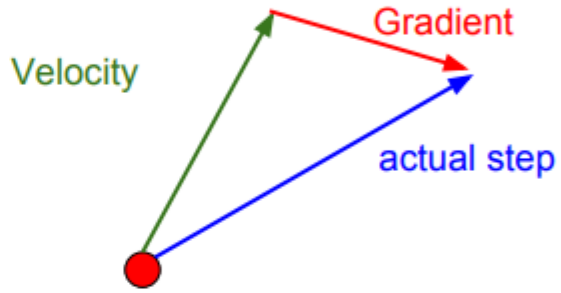
Nesterov Momentum

- Vanilla momentum method: Current gradient + Current velocity.



$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

- Nesterov Version: Look ahead to the point where the current velocity would take us. Take the gradient there and perform the update.



☹

$$v_{t+1} = \rho v_t + \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t - v_{t+1}$$

AdaGrad

$$G(t) = \sum_{t=1}^{t} g(t)g^T(t)$$

$$\Delta W(t) = -\frac{\eta}{\sqrt{diag(G(t)) + \epsilon I}} * g(t)$$

- Perform element-wise scaling of the gradient
  - Scale factors determined based on the historical sum of squares…

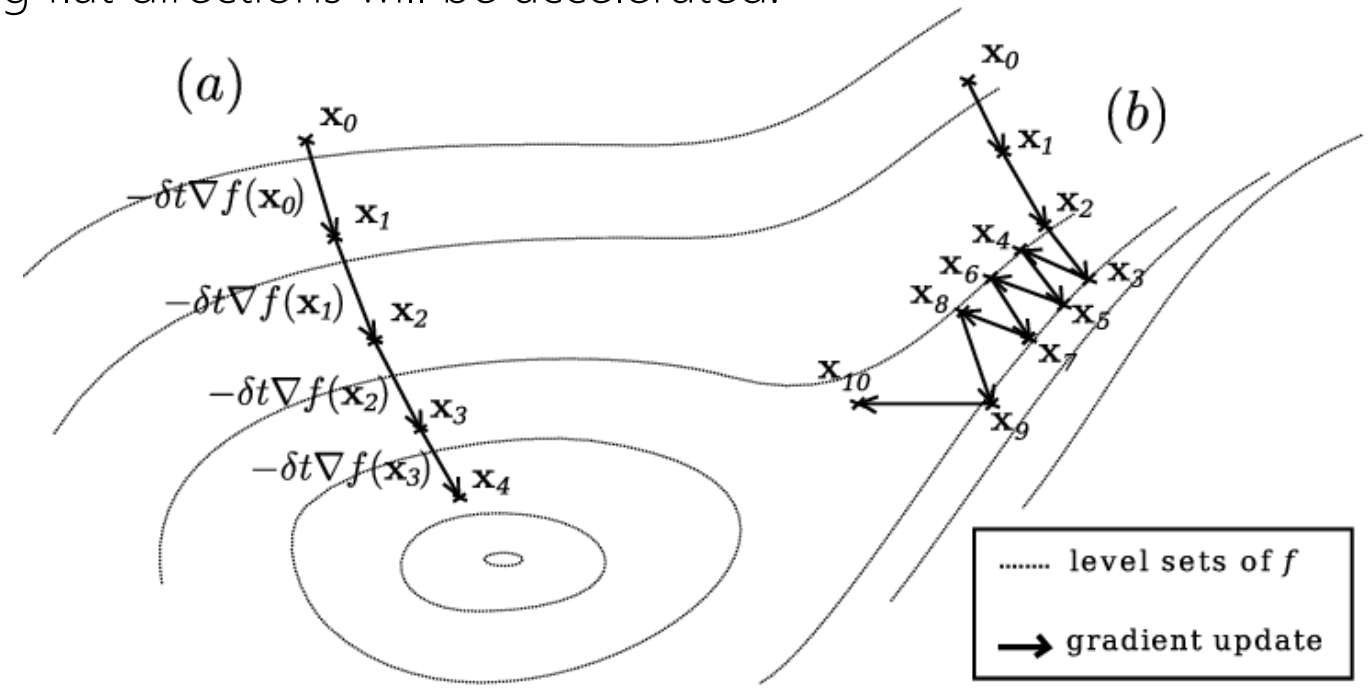
```
scale_factor = 0

for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    scale_factor += dx*dx
    x -= learning_rate * dx / (np.sqrt(scale_factor) + epsilon)
```

- The element-wise scaling has an effect of "per-parameter learning rates" or "adaptive learning rates," thus, the name Adaptive Gradient.

AdaGrad

- Long narrow valley: what happens with AdaGrad?
  - Step size along steep directions will be damped.
  - Step size along flat directions will be accelerated.



AdaGrad

- Historical sum: what happens with AdaGrad after many iterations?
  - Step size decays to zero… ☹

```
scale_factor = 0

for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    scale_factor += dx*dx
    x -= learning_rate * dx / (np.sqrt(scale_factor) + epsilon)
```

RMSProp

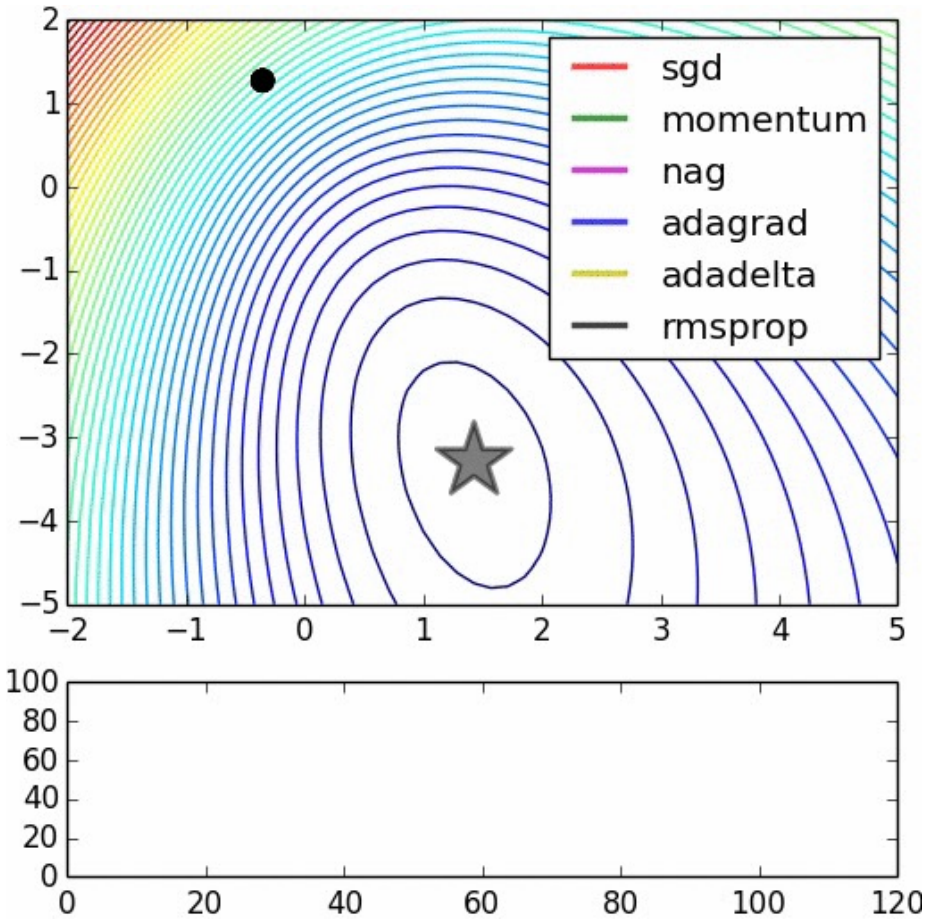
- AdaGrad (step size decays to zero):

```
scale_factor = 0
for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    scale_factor += dx*dx
    x -= learning_rate * dx / (np.sqrt(scale_factor) + epsilon)
```

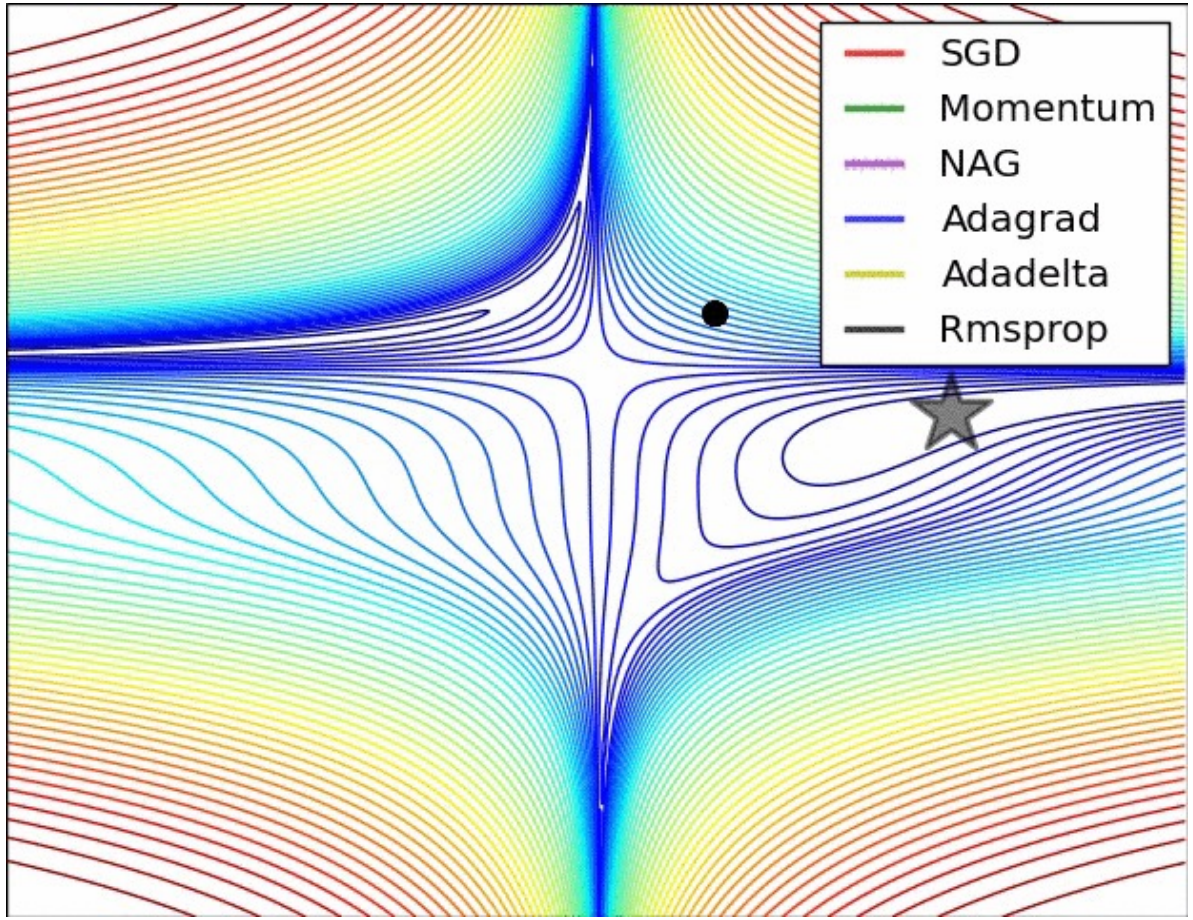
- RMSProp (problem solved ☺)

```
scale_factor = 0
for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    scale_factor = decay_rate*scale_factor + (1-decay_rate)*dx*dx
    x -= learning_rate * dx / (np.sqrt(scale_factor) + epsilon)
```

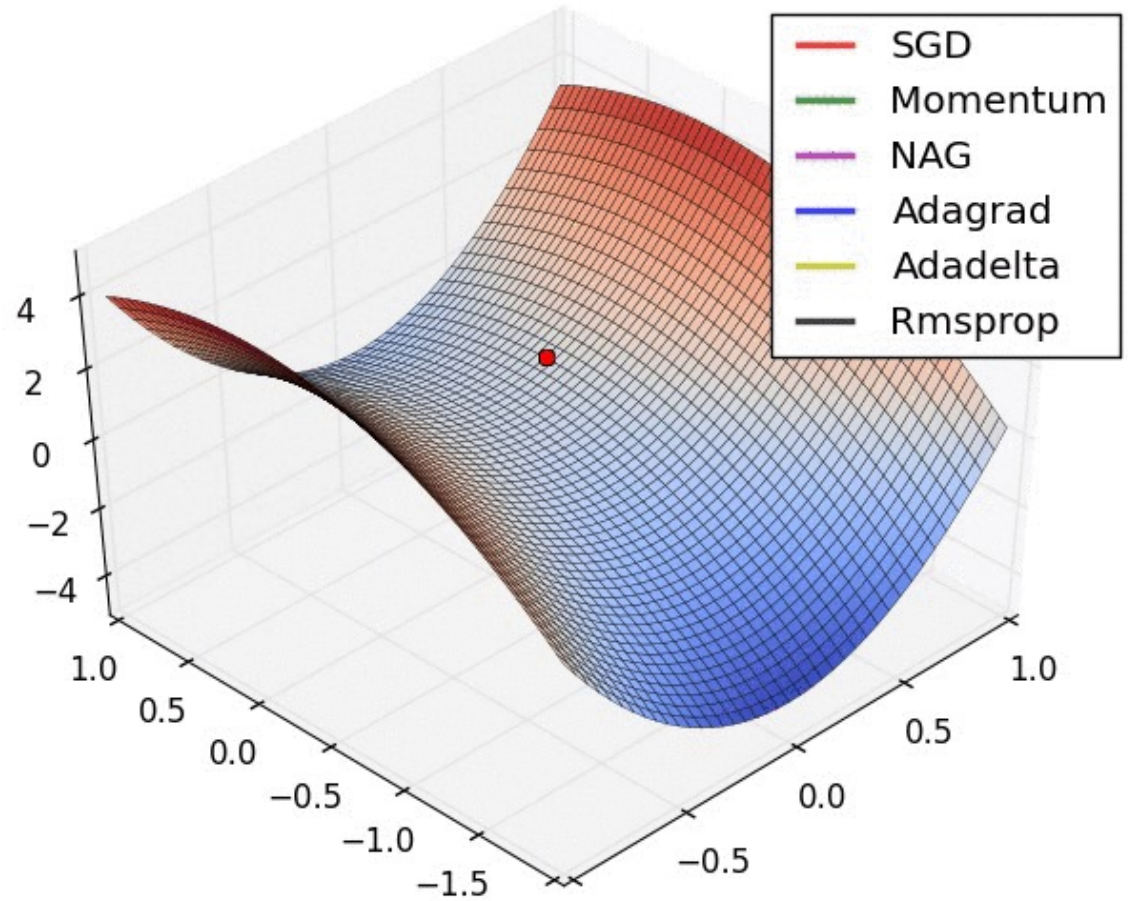
Comparison of Optimization Methods



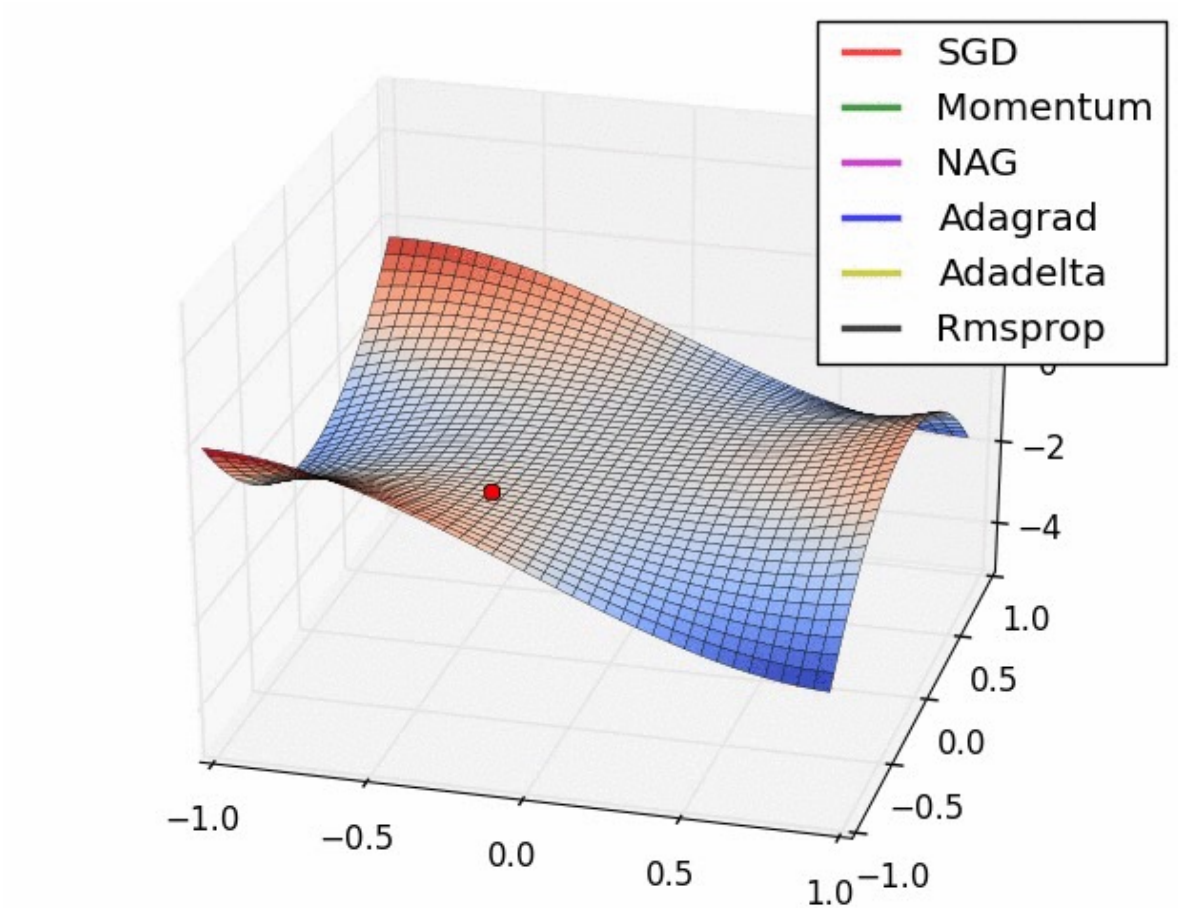
Comparison of Optimization Methods



Comparison of Optimization Methods



Comparison of Optimization Methods



Adam (All of the above!)

- Why not take the advantage of both momentum and adaptive gradient methods?

```
moment = [0, 0]
for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    moment[0] = beta[0]*moment[0] + (1-beta[0])*dx                    # momentum
    moment[1] = beta[1]*moment[1] + (1-beta[1])*dx*dx                 # RMSProp
    x -= learning_rate * moment[0] / (np.sqrt(moment[1]) + epsilon)
```

- Problem with the idea: what happens when iter = 0?
  - moments = 0 → bias!

Adam (All of the above!)

```
moment = [0, 0]
for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    moment[0] = beta[0]*moment[0] + (1-beta[0])*dx                  # momentum
    moment[1] = beta[1]*moment[1] + (1-beta[1])*dx*dx               # RMSProp
    x -= learning_rate * moment[0] / (np.sqrt(moment[1]) + epsilon)
```

- Modified version:

In typical scenarios, a good starting point:

- beta = some big number close to 1 (e.g. 0.9, 0.99)
- learning_rate = 1e-3

```
moment = [0, 0]
for iter in range(0, MAX_ITER):
    dx = backpropagate(x)   # compute gradient
    moment[0] = beta[0]*moment[0] + (1-beta[0])*dx                  # momentum
    moment[1] = beta[1]*moment[1] + (1-beta[1])*dx*dx               # RMSProp
    unbiased[0] = moment[0] / (1 - beta[0]**iter)                   # bias correction
    unbiased[1] = moment[1] / (1 - beta[1]**iter)
    x -= learning_rate * unbiased[0] / (np.sqrt(unbiased[1]) + epsilon)
```

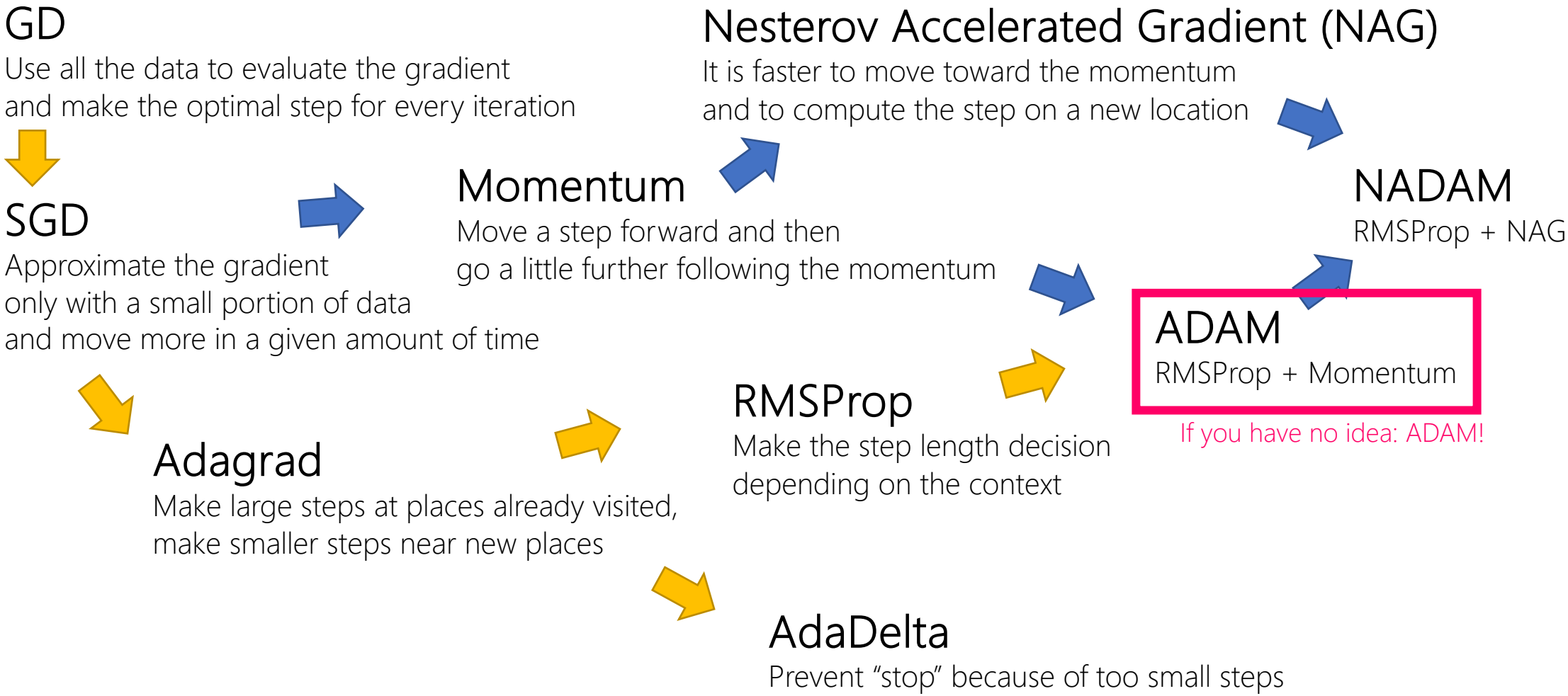
Adam (All of the above)

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

History of NN Optimizers



Tips on Learning Process & Hyperparameter Tuning

Training Process, Step-by-step

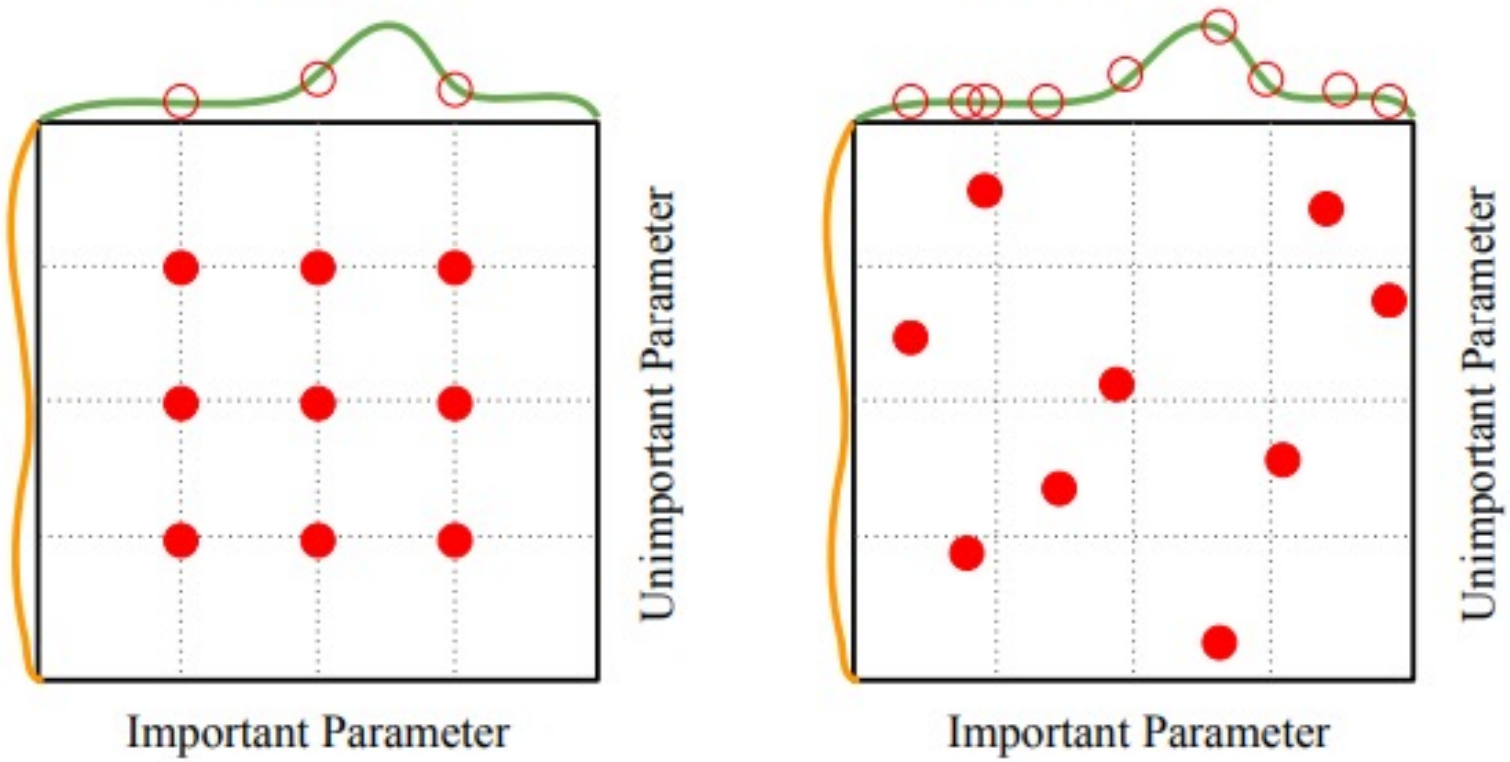
1. Preprocess the data (normalize/whiten, or at least zero-center them)
2. Design the network architecture (start with a simple one if possible)
3. Train with a small amount of data (a subset of your data). Make sure the model can overfit (i.e., loss go down to 0 and accuracy goes 1).
4. Now, the actual training, find the learning rate
   - LR too low: loss won't go down, LR too high: loss will explode.
   - Start with a small number and find the learning rate that makes some observable change.
5. Hyperparameter Tuning!

Hyperparameters Tuning

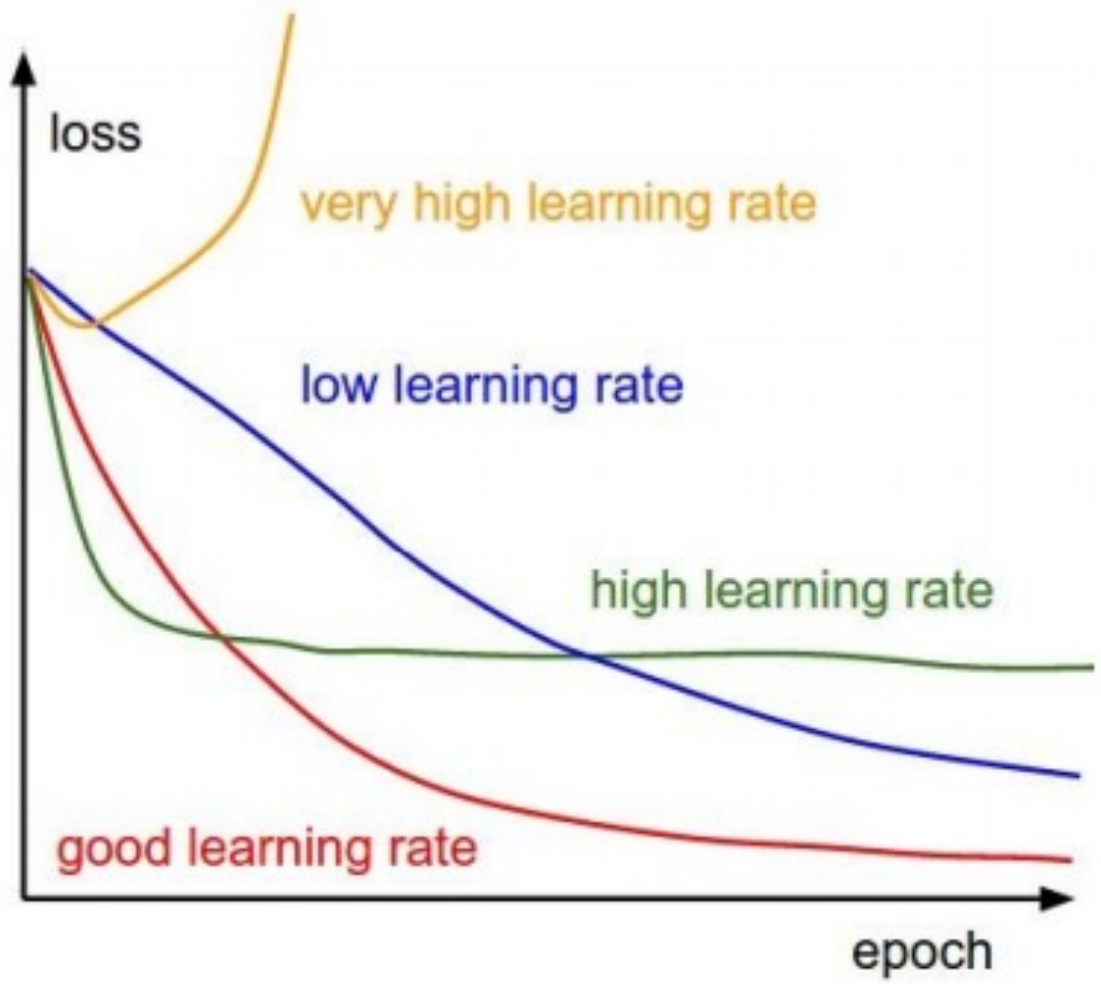
- Major hyperparameters: depth and breadth (or network architecture)
- Others: regularization terms, learning rate and its decay schedule, etc.

- Coarse-to-fine Strategy:
  - First, only a few epochs to get a rough idea of how parameters play
  - Second, longer epochs, do finer search
  - Repeat this as architecture changes.

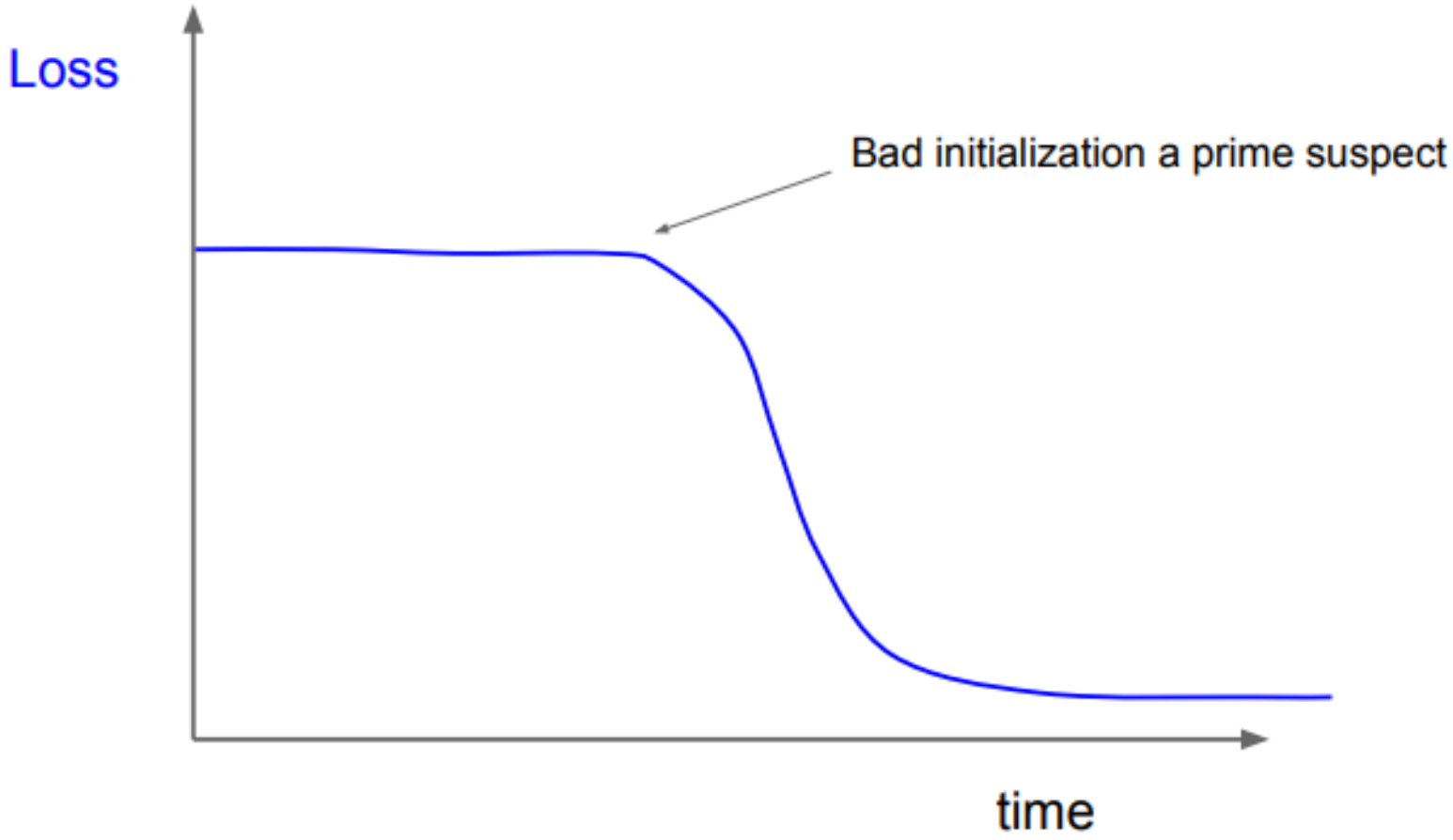
Grid Search vs. Random Search



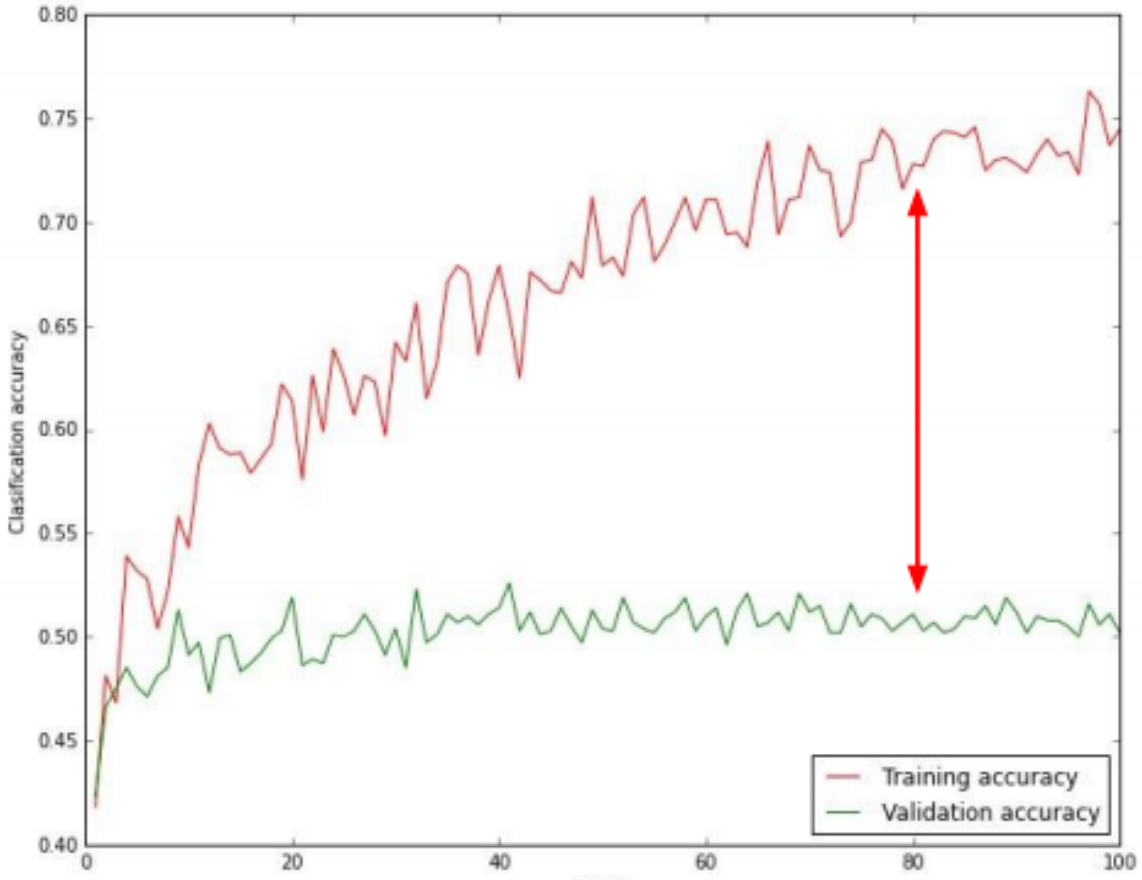
Always monitor and visualize the loss/accuracy!



Always monitor and visualize the loss/accuracy!



Always monitor and visualize the loss/accuracy!



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Stanford cs231n

Tracking the weights also helps

- Percentage of weight update over weight magnitudes
  - Around 0.001 is ideal
  - Around 0.01 is about okay

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Stanford cs231n

Model Ensemble

- Train multiple models independently.
- Throw those models to the test data points and take the average.
- Enjoy some extra accuracy (1~3%)
- (kind of regularization in a funny way)