Optimization, Regularization

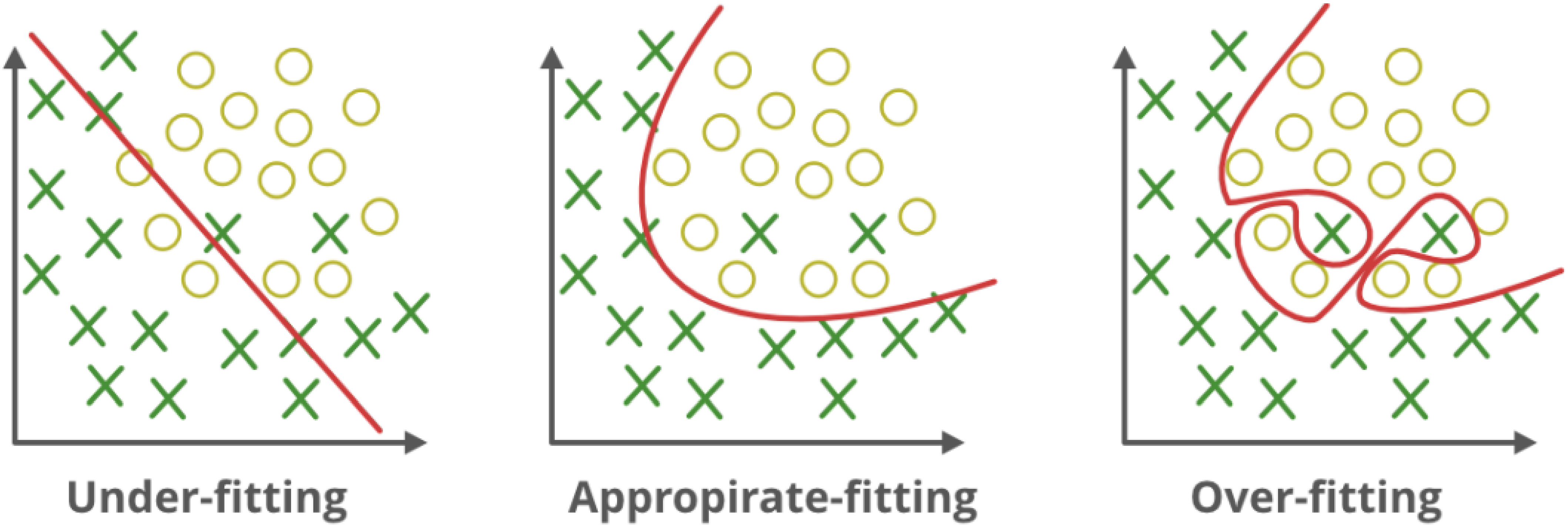
HESAM HOSSEINI

SUMMER 2024

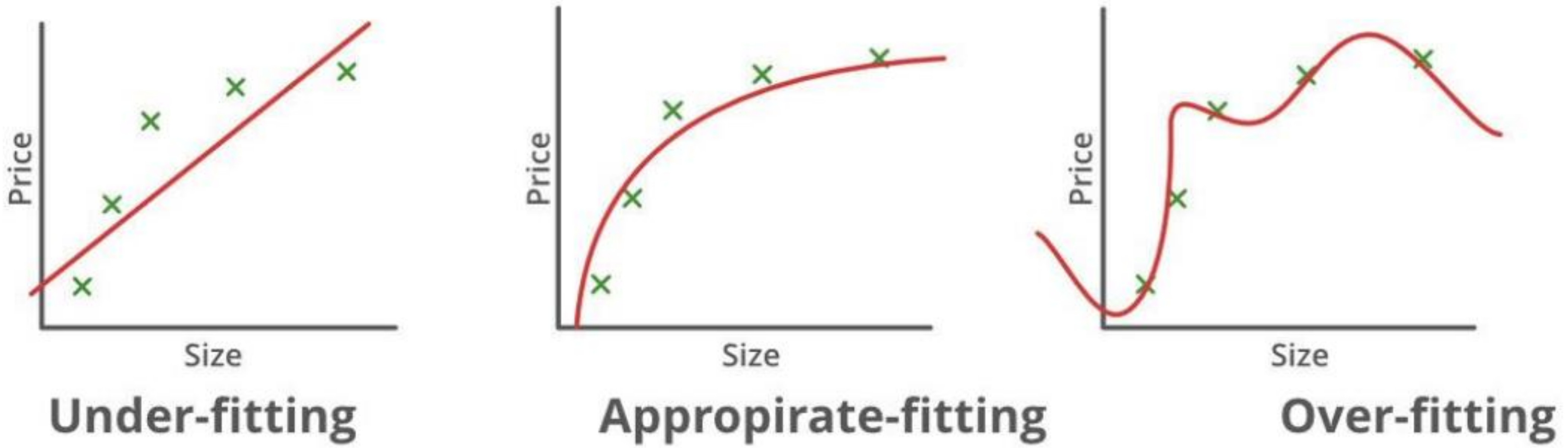
Model Fitness

- **Underfitted Model:** Model performs poorly on the training data:
  - May be too simple
- **Overfitted Model:** Model performs well on the training data but poorly on test data.
  - May be too complex
- **Proper Model:** Model performs well on both training and test data

Model Fitness



Model Fitness



Bias-Variance Dilemma

- **Bias Error:** The error due to bias is taken as the difference between the expected (or average) prediction of our model and the correct value which we are trying to predict. Bias measures how far off in general these models' predictions are from the correct value.
- **Variance Error:** The error due to variance is taken as the variability of a model prediction for a given data point. The variance is how much the predictions for a given point vary between different realizations of the model

$$E[\overbrace{(\hat{y} - y)^2}^{\text{How far is a model from the ground truth}}] = E[\overbrace{(\hat{y} - E[\hat{y}])^2}^{\text{How far is a model from the «average model»}}] + \overbrace{(E[\hat{y}] - y)^2}^{\text{How far is the «average model» from the ground truth}}$$

$$MSE = Variance + Bias^2$$

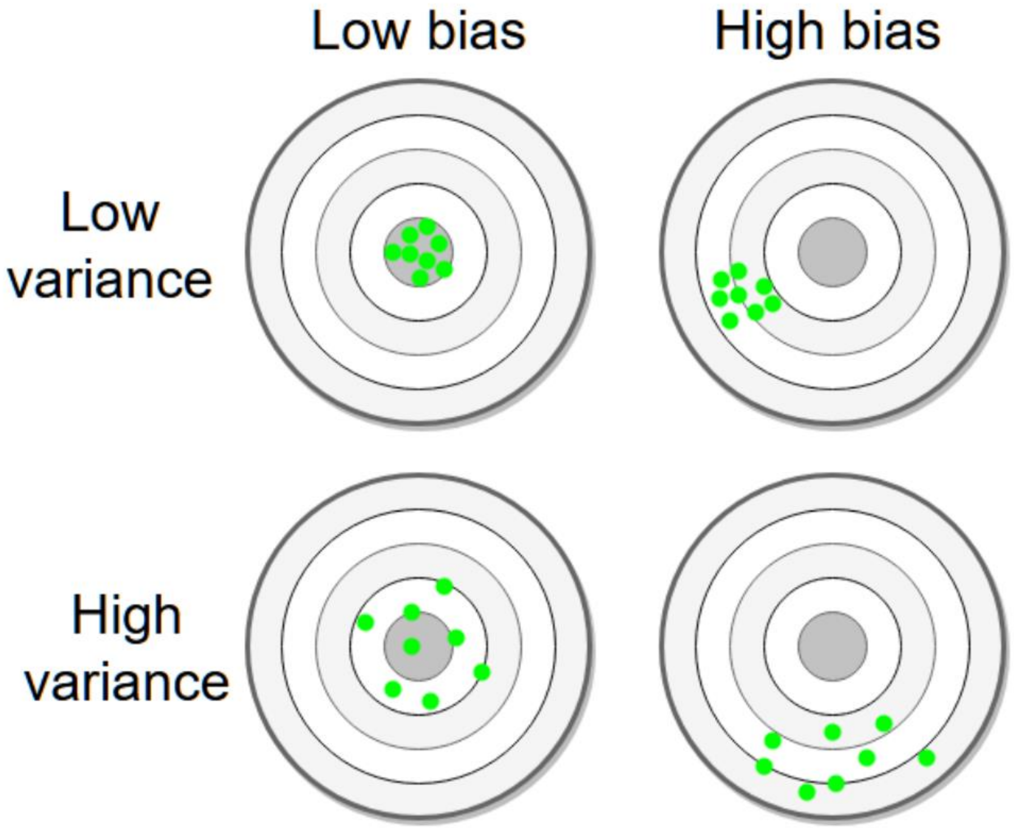
Bias-Variance Dilemma

Too Simple Model:

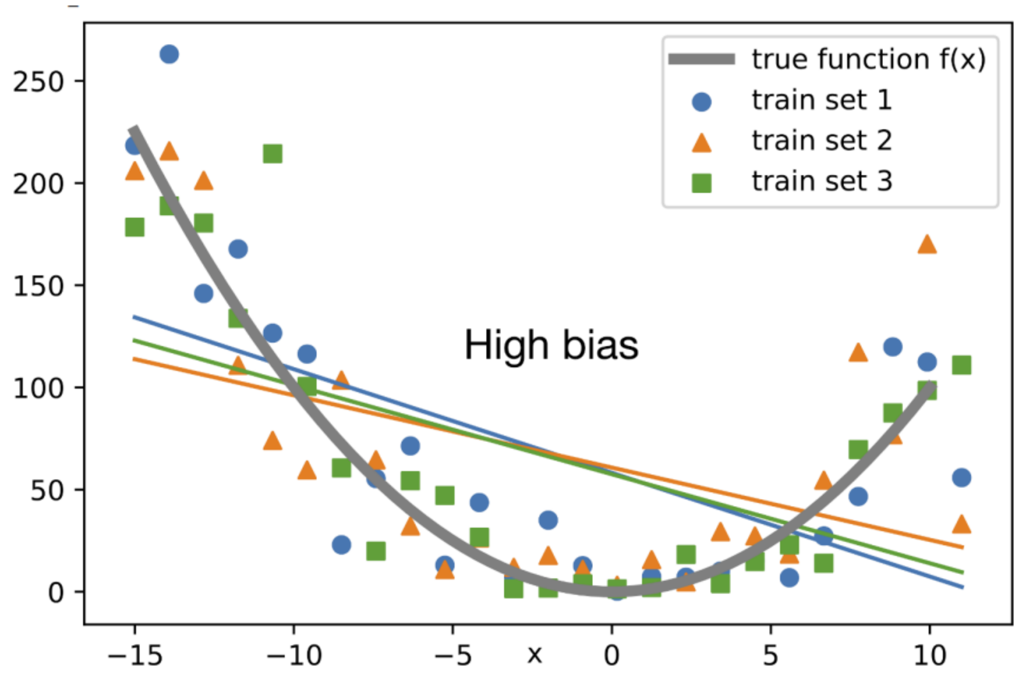
- High Bias-Low Variance

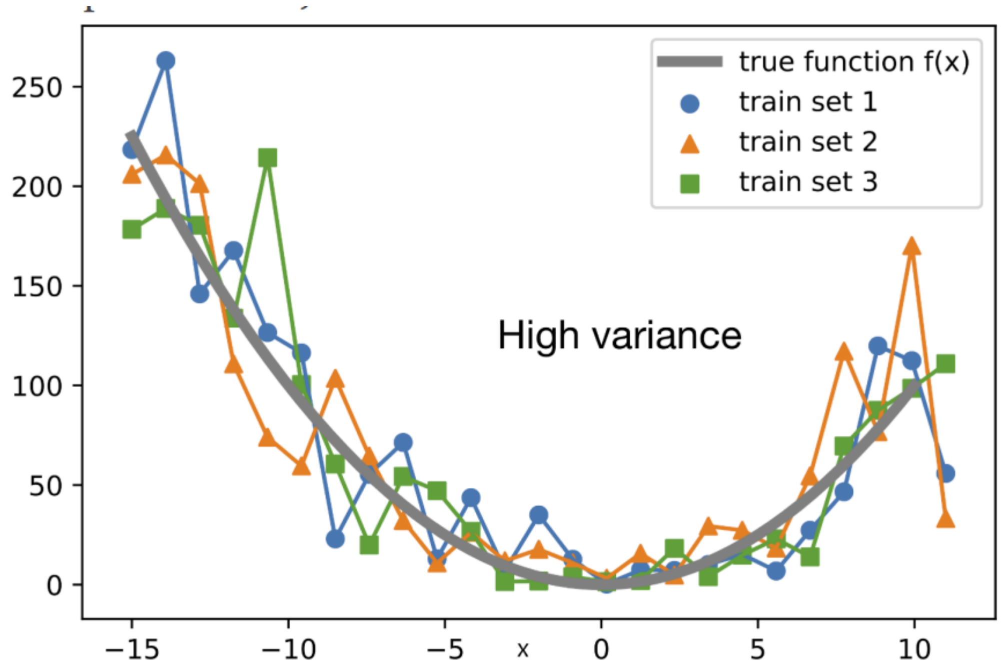
Too Complex Model:

- Low Bias-High Variance



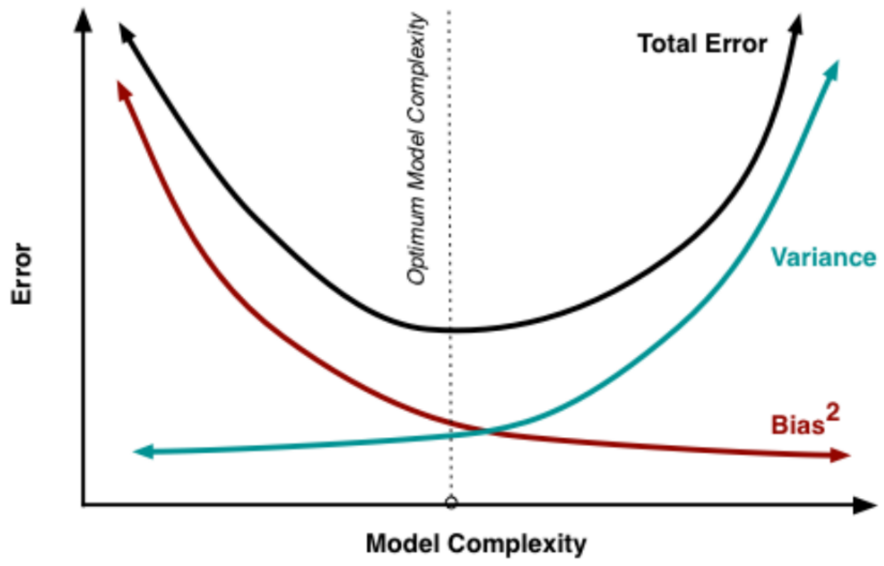
Bias-Variance Dilemma



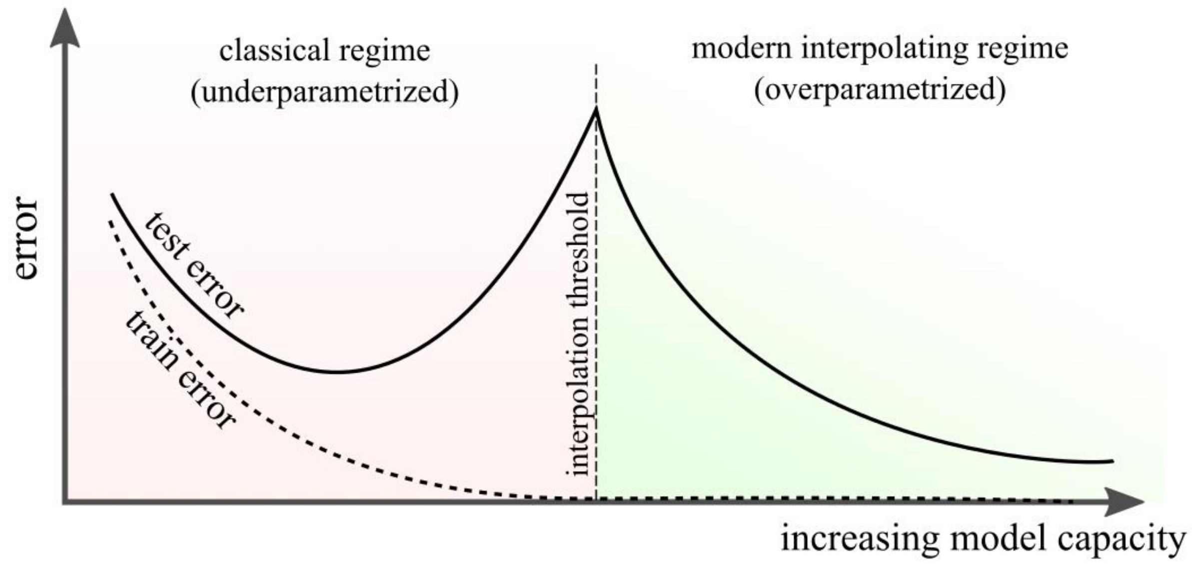


Bias-Variance Dilemma

Understanding Over- and Under-Fitting



Classical ML



Double descent curve

Regularization

Symptoms		Cause	Solution
Training Error	Validation Error		
High	High	High Bias	• Increase model complexity • Train for more epochs • Add Features • Boosting
Low	High	High Variance	• Add more training data (Augmentation) • Reduce model complexity • Regularization • Bagging
Low	Low	Perfecto	Good job!

Regularization Definition

Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

Examples:

- Parameter (weights) Penalties
- Dataset Augmentation
- Noise Robustness (input/output)
- Early Stopping
- Bagging and Other Ensemble Methods
- Dropout

Parameter Penalties

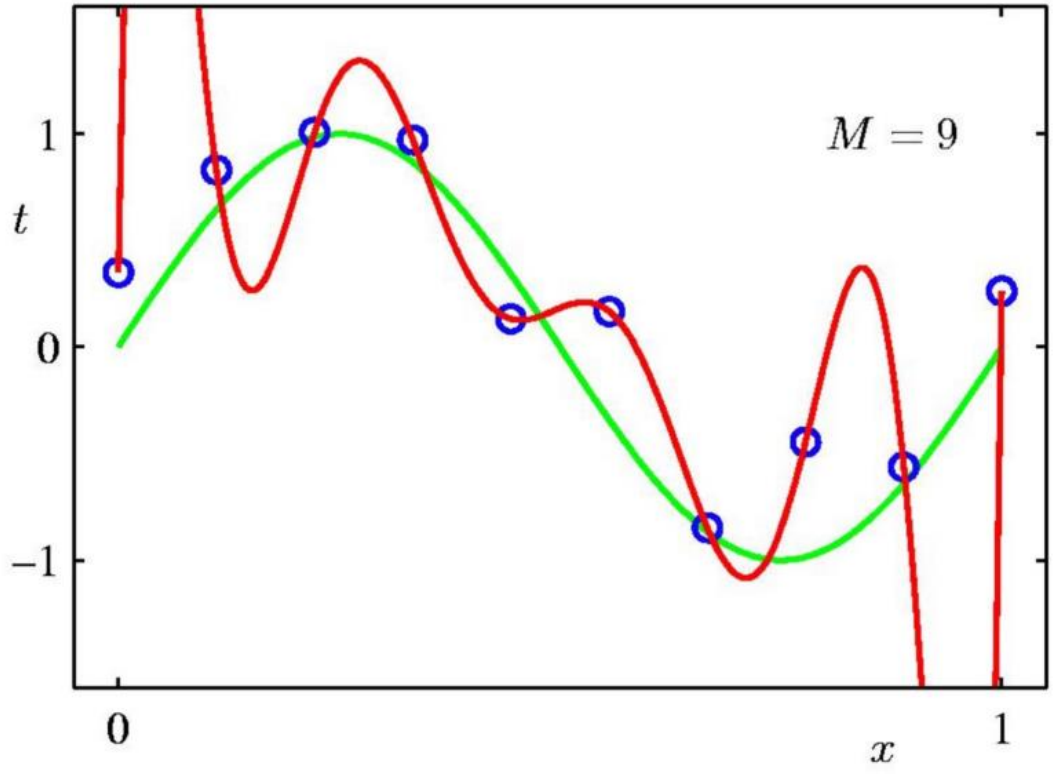
Instead empirical risk minimization:

- $minimize\ Loss(Data|Model)$

Try structural risk minimization:

- $minimize\ Loss(Data|Model) + \lambda Complexity(Mode)$

How can we measure complexity?



	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

L2 Regularization (Ridge)

- Formulation:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2} \|\boldsymbol{w}\|_2^2 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- It can be shown:

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

- or:

$$\boldsymbol{w} \leftarrow (1 - \alpha\epsilon)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

$$\text{L1 norm: } \|\mathbf{w}\|_1 = \sum_i^n |w_i|$$

$$\text{Squared L2 norm: } \|\mathbf{w}\|_2^2 = \sum_i^n w_i^2$$

L1 Regularization (Lasso)

- Formulation

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})$$

- Then

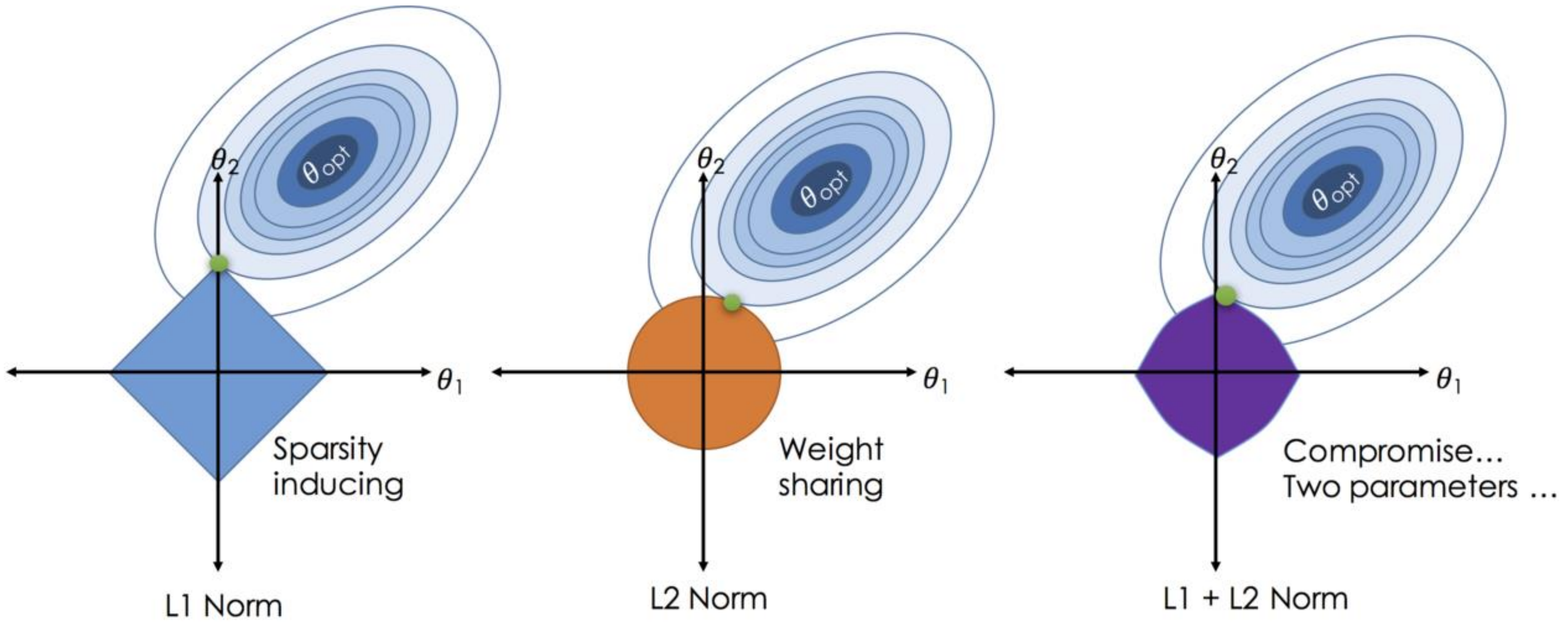
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha sign(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}))$$

- Exact solution is

$$w_i = sign(w_i^*) max\left\{|w_i^*| - \frac{\alpha}{\mathbf{H}_{i,i}}, 0\right\}$$

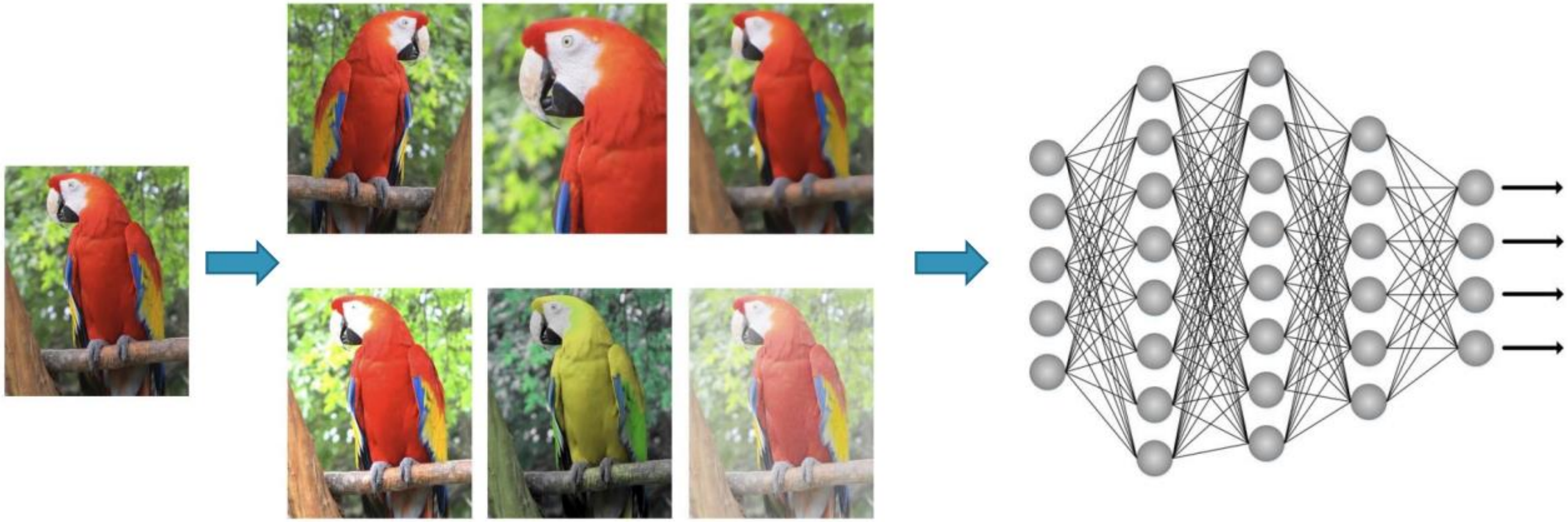
- This is weight sparsifier

Geometric intuition

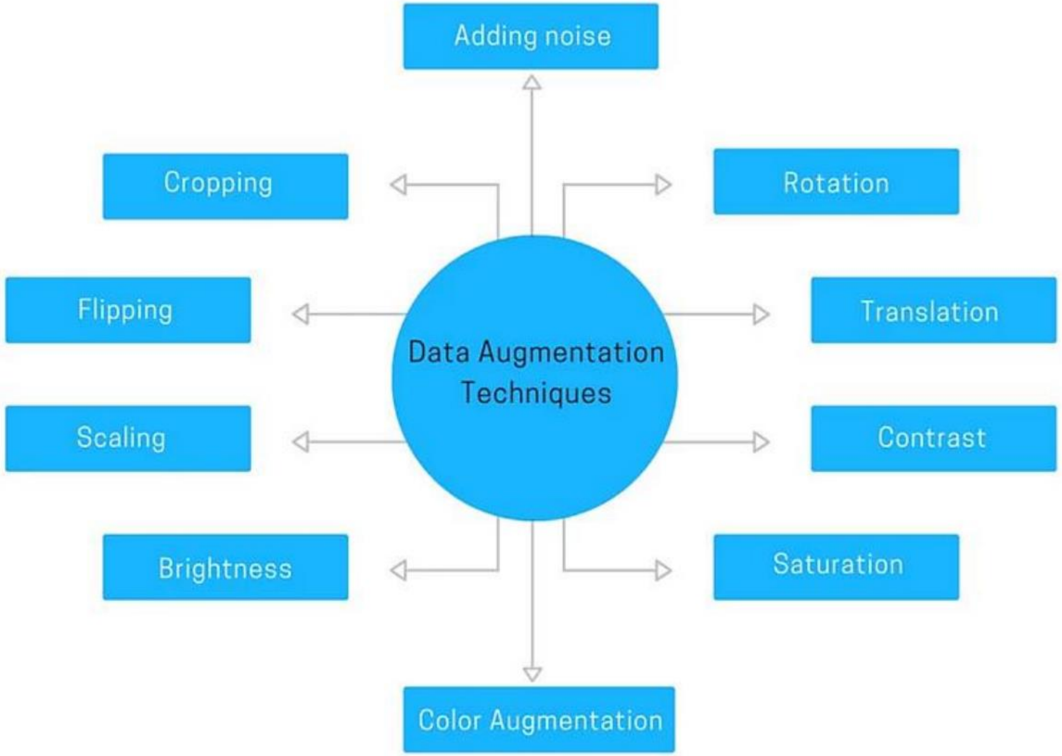


Data Augmentation

Invariance (Rotation, Scaling, Translation, Shearing, Mirroring, …)

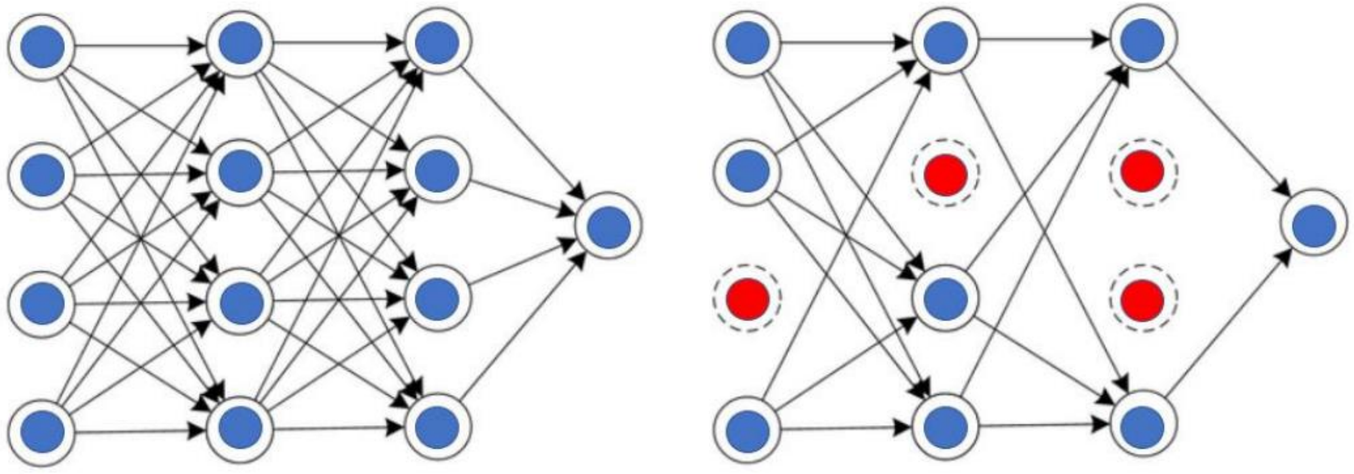


Data Augmentation



Dropout

- **Co-Adaptation:** If two or more neurons extract the same feature repeatedly or highly correlated behavior (coadaptation), the network isn't reaching its full capability
- **Solution:** Update (EBP) a fraction (Randomly Chosen) of all weights in each iteration!



Dropout

- The key idea is to randomly drop units from the neural network during training.

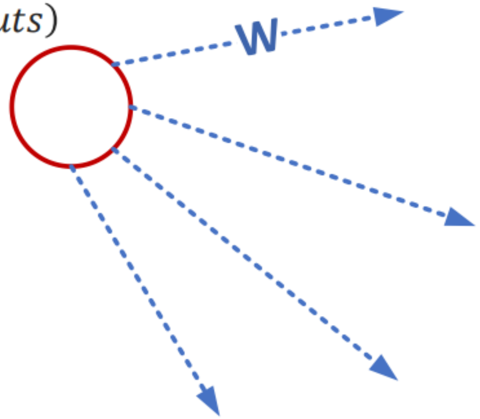
- **During training:** dropout samples from number of different "thinned" network. For each hidden layer, for each training sample, for each iteration, dropout (zero out) a random fraction, $(1 - p)$, of neuron (and corresponding weight).
  - Input Layer: $P > 0.8$ or $P = 1.0$
  - Hidden Layer: $P = 0.5$
  - Output Layer: $P = 1.0$

- **At test time:** Use all activations, but multiplied neuron output by p. to account for the missing activations during training

Present with probability p

$O_i = X_i f(inputs)$

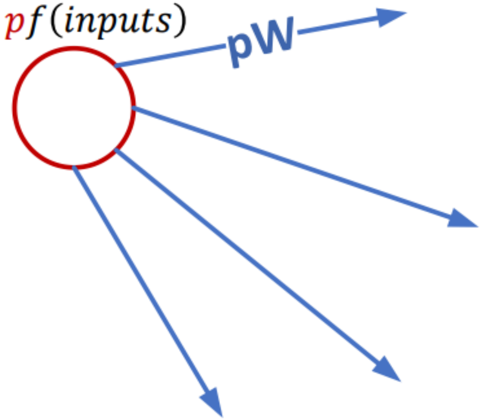
$X_i \in \{0.1\}$



Training Phase

Always Present

$O_i = p f(inputs)$

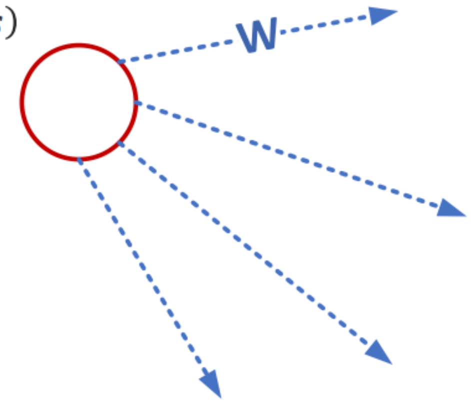


Test Phase

Present with probability p

$O_i = \frac{1}{p} X_i f(inputs)$

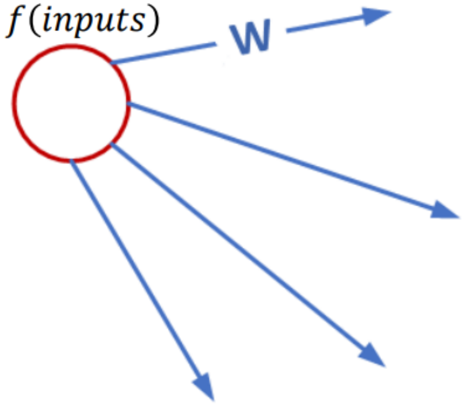
$X_i \in \{0.1\}$



Training Phase

Always Present

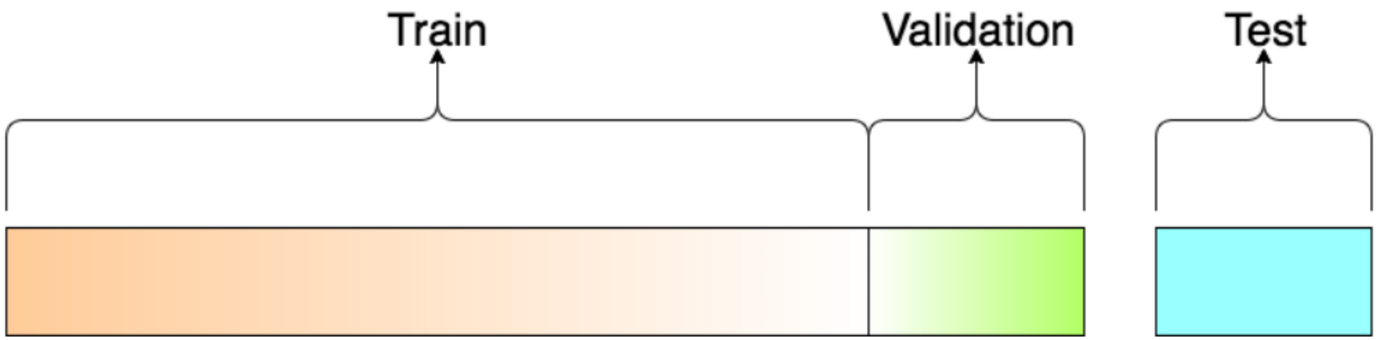
$O_i = f(inputs)$



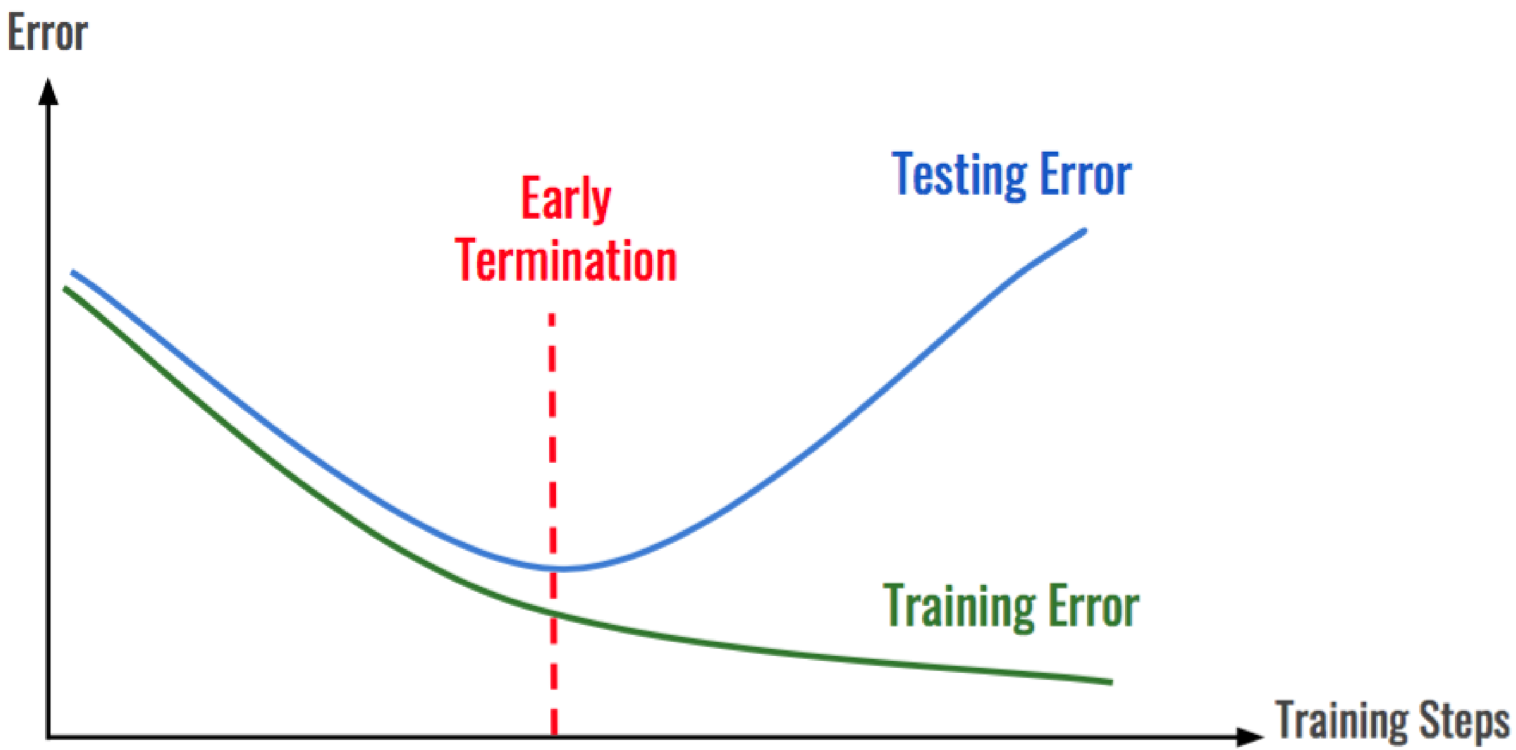
Test Phase

Review: How to Use Labeled Data

- **Training Set:** The actual dataset that we use to train the model, find parameters. The model sees and learns from this data.
- **Validation/Development Set:** The sample of data used to provide an unbiased and fair evaluation of a model fit on the training dataset while tuning model hyperparameters. • The model occasionally sees this data, but never does it “Learn” from this.
- **Test Set:** The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.



Early Stopping



Optimization

Deep learning deals with non-convex optimization a real challenge!

Local minimum and saddle points

Vanishing/Exploding Gradient

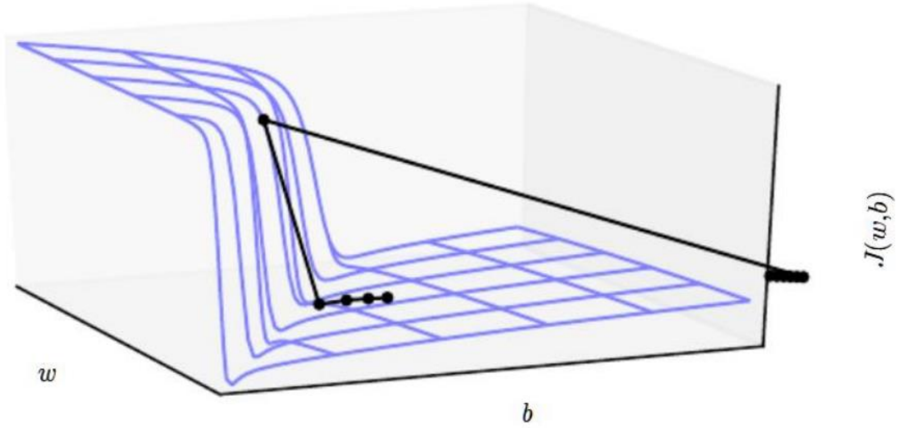
- Suppose a simple computational graph (Back Propagation) in which we repeatedly (t times) multiplying by a matrix

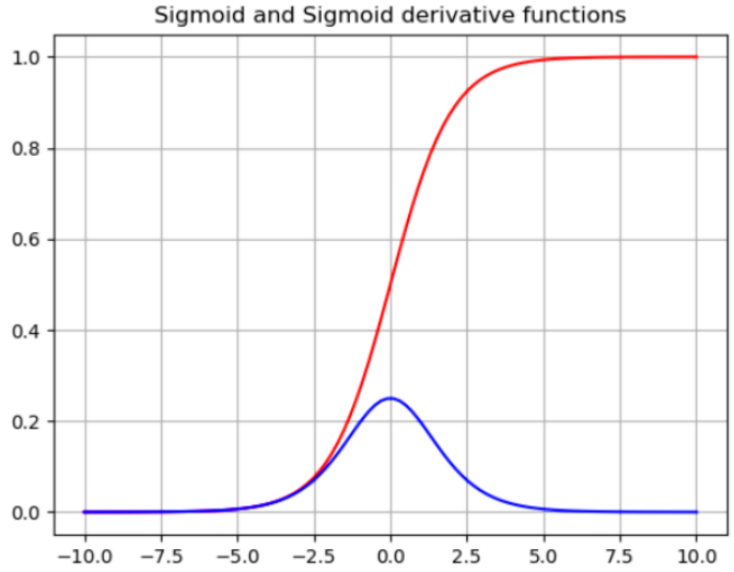
$$\boldsymbol{W} = \boldsymbol{V} diag(\boldsymbol{\lambda}) \boldsymbol{V}^{-1}$$

$$\boldsymbol{W}^{t} = \left(\boldsymbol{V} diag(\boldsymbol{\lambda}) \boldsymbol{V}^{-1}\right)^{t} = \boldsymbol{W} = \boldsymbol{V} diag(\boldsymbol{\lambda})^{t} \boldsymbol{V}^{-1}$$

Vanishing and Exploding gradient

Vanishing ($\lambda i < 1$) or Exploding ($\lambda i > 1$)





ReLU is a good choice!

Gradient Clipping ($\alpha \frac{g}{\|g\|}$)

Stochastic Gradient Descent

Problems with vanilla gradient descent

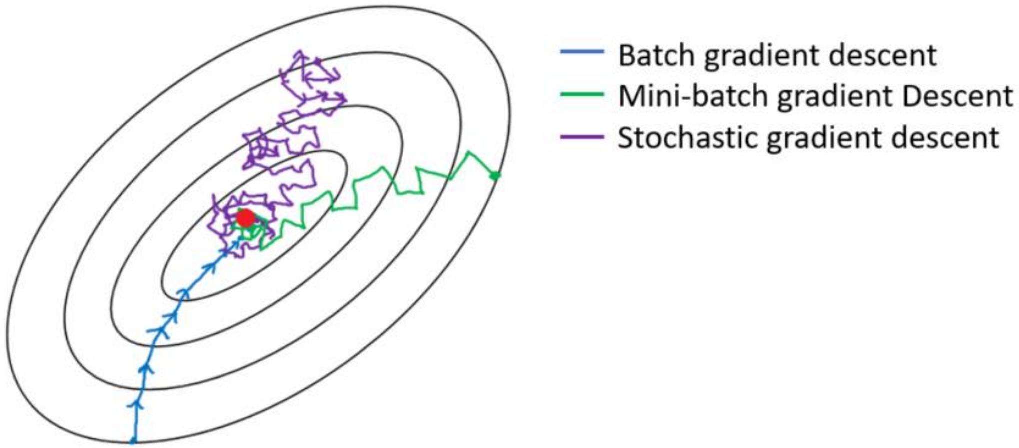
- The computational cost is very high when N increases
- The power of exploring and getting out of local minima is low

$$w^{t+1} = w^t - \alpha \frac{1}{N} \sum_{i=1}^{N} \frac{\partial loss(y_i, \tilde{y}_i)}{\partial w}$$

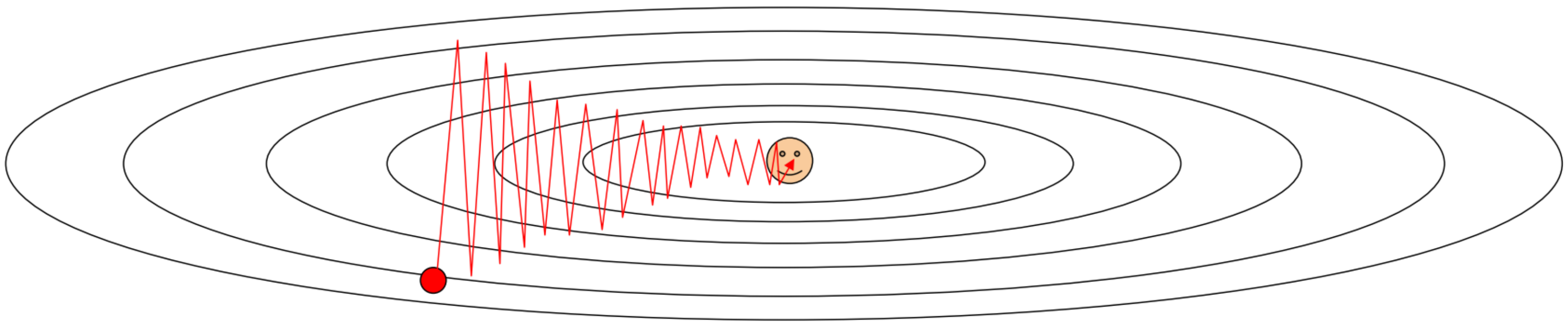
Solution: Using smaller parts or mini batch of data (32/64/128):

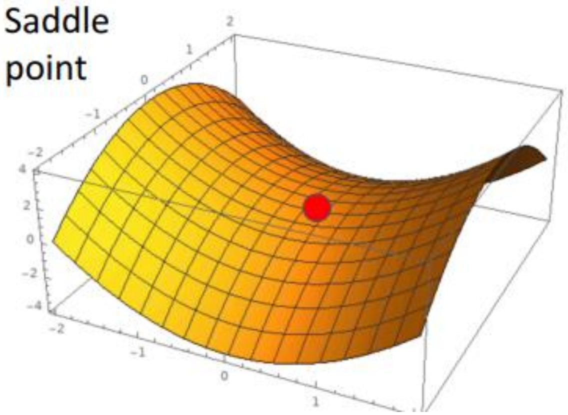
$$w^{t+1} = w^t - \alpha \frac{1}{M} \sum_{i=1}^{M} \frac{\partial loss(y_i, \tilde{y}_i)}{\partial w}$$

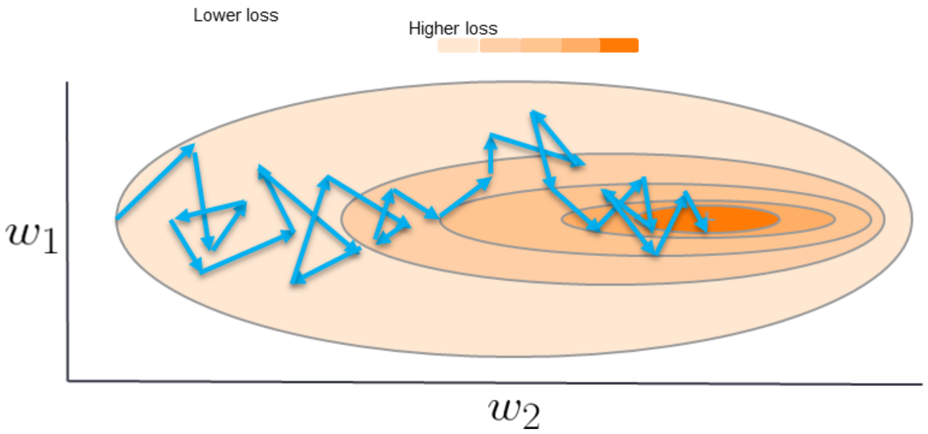
$$M = mini \;\; batch \;\; size$$



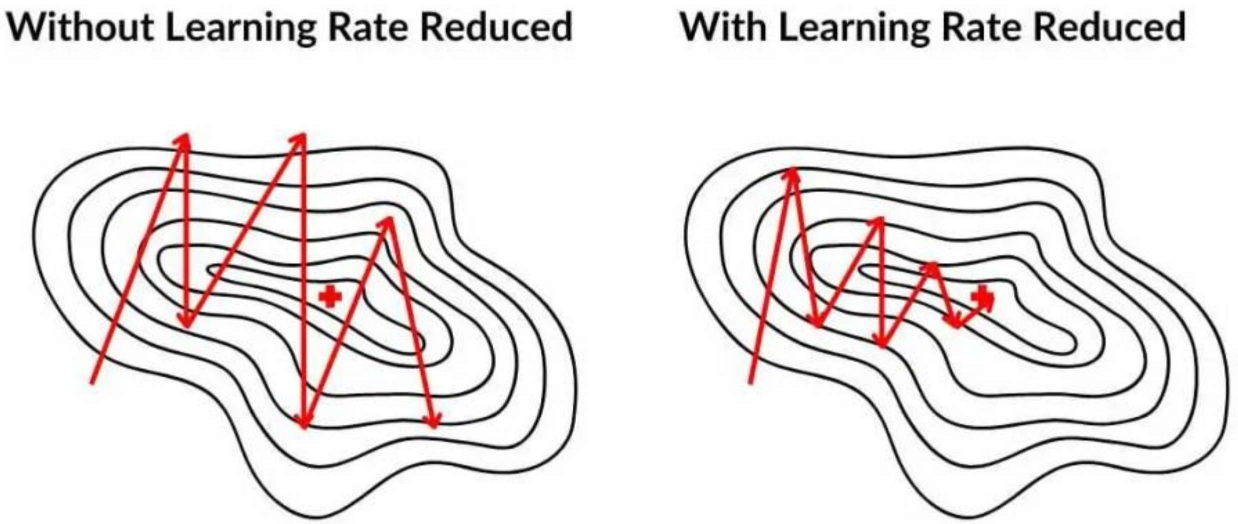
SGD problems





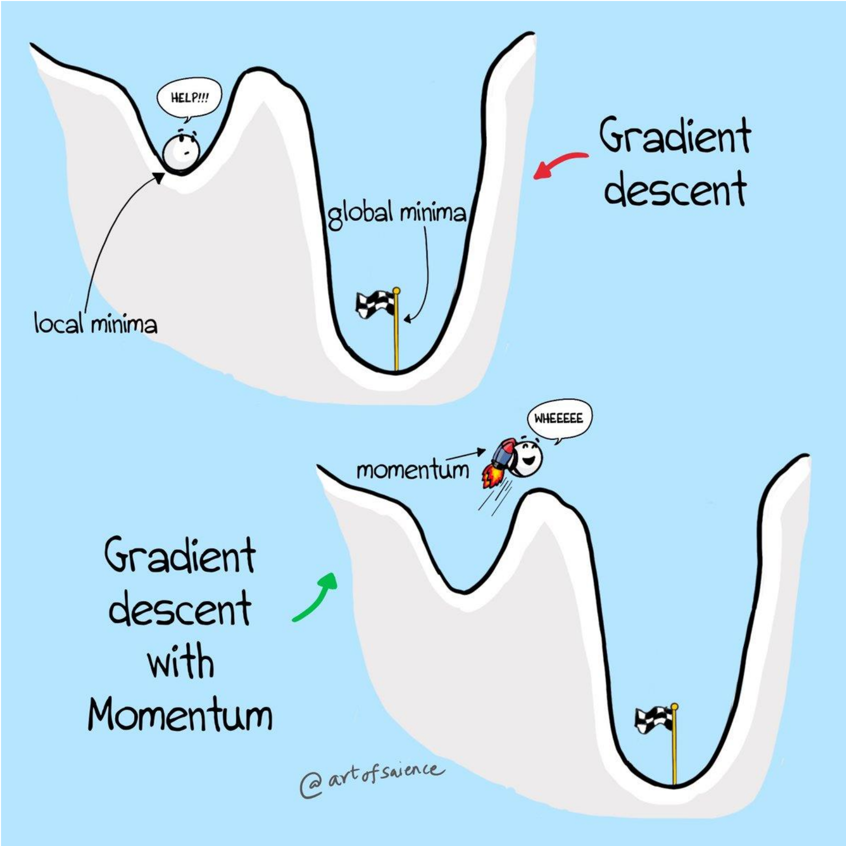


Annealing the learning rate



SGD + Momentum

Since the force on the particle is related to the gradient of potential energy (i.e. $F=-\nabla U$), the force felt by the particle is precisely the (negative) gradient of the loss function. Moreover, $F=ma$ so the (negative) gradient is in this view proportional to the acceleration of the particle.



SGD + Momentum

SGD

$$w^{t+1} = w^t - \alpha \nabla loss(mini\ batch)$$

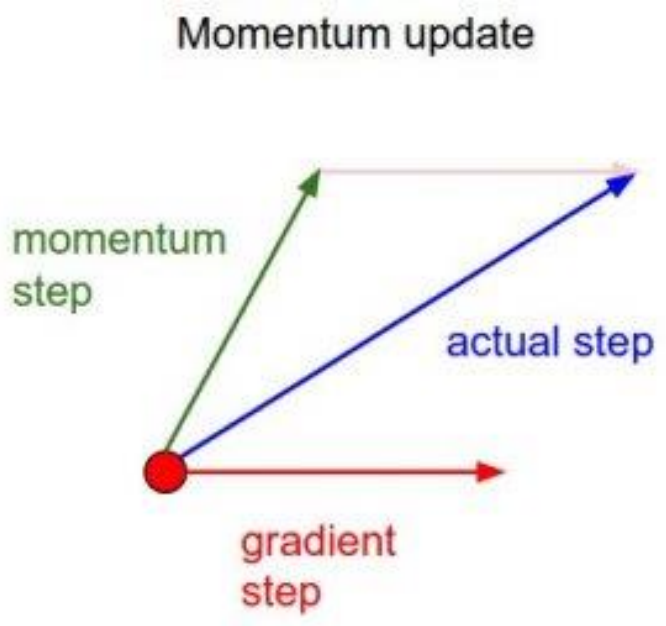
```
while True:
    dw = compute_gradient(x)
    w = w - alpha * dw
```

SGD + Momentum

$$v^{t+1} = \rho v^t + (1 - \rho) \nabla loss(mini\ batch)$$
$$w^{t+1} = w^t - \alpha v^{t+1}$$

```
v = 0
while True:
    dw =
compute_gradient(x)
    v = rho * v + dw
    w = w - alpha * v
```

Nesterov Momentum

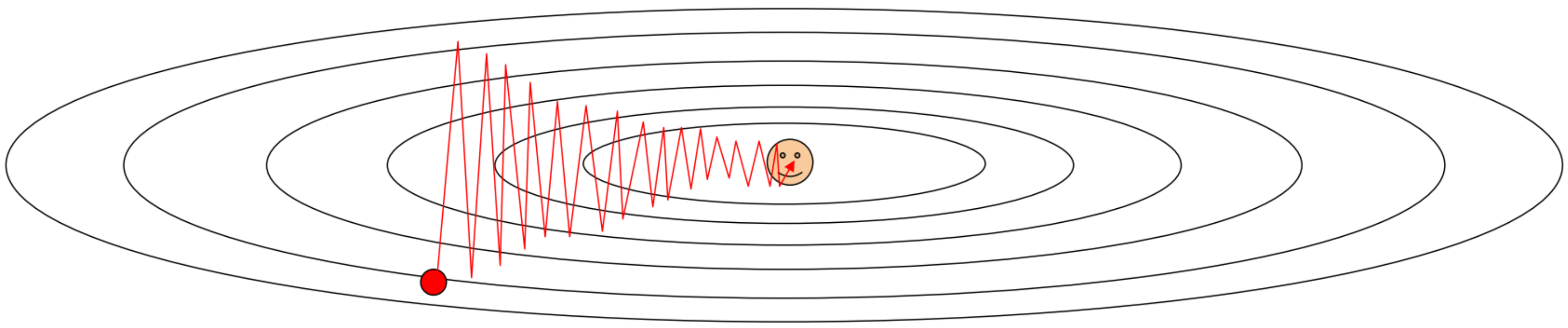


Combine gradient at current point with velocity to get step used to update weights

"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

RMSProb

- Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large
- What can be done about the large difference in gradient in some directions?
- Consider an adaptive learning rate for each dimension.
- Progress slows down in directions where the gradient is high
- Progress accelerates in directions where the gradient is low



RMSProb

A kind of averaging

Storing sum-of-squares history in each dimension for gradient scaling

$$\Delta^{t+1} = \gamma \Delta^t + (1-\gamma)(\nabla loss(mini\ batch))^2$$

$$w^{t+1} = w^t - \alpha \frac{\nabla loss(mini\ batch)}{\sqrt{\Delta^{t+1}} + \epsilon}$$

Scale each parameter according to the amount of changes (adaptive learning rate)

Avoid division by zero

Adam (almost): RMSProp + Momentum

$$\Delta^{t+1} = \gamma_1 \Delta^t + (1 - \gamma_1)(\nabla loss(mini\ batch))^2$$

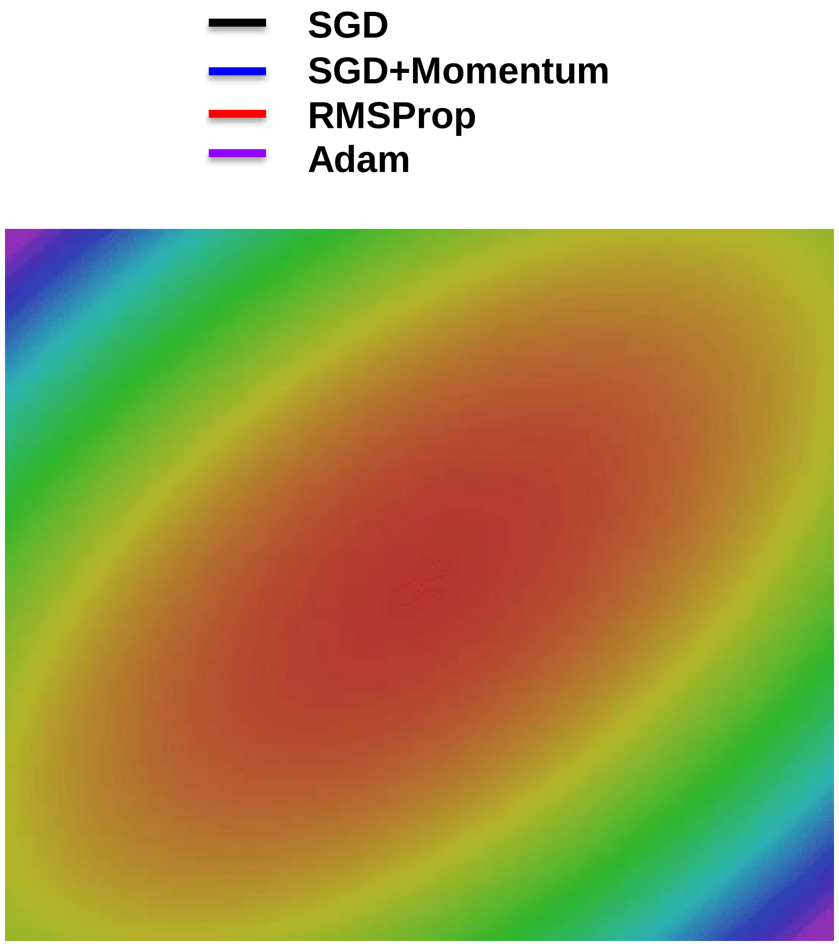
Momentum

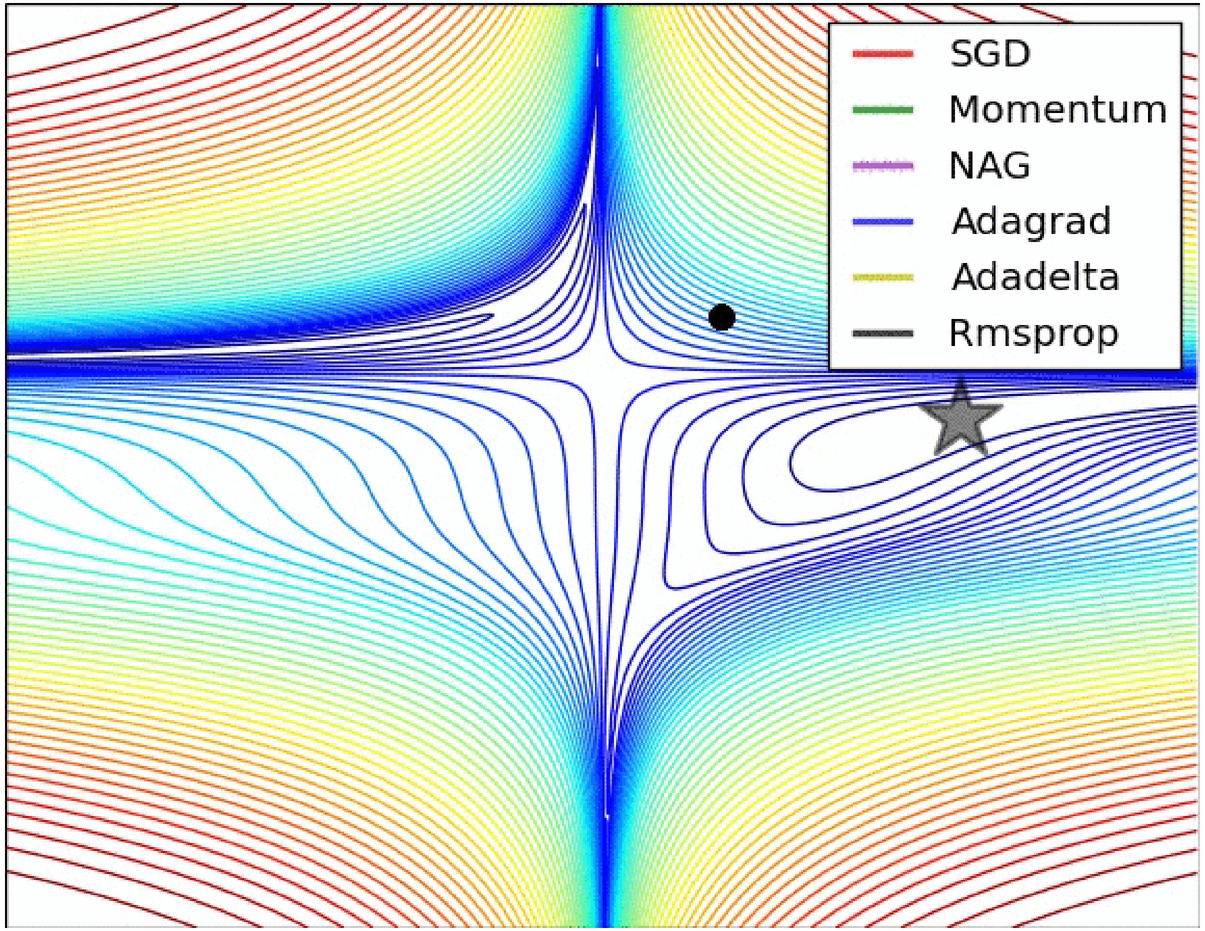
$$v^{t+1} = \gamma_2 v^t + (1 - \gamma_2)\nabla loss(mini\ batch)$$

RMSProp

$$w^{t+1} = w^t - \alpha \frac{v^{t+1}}{\sqrt{\Delta^{t+1}} + \epsilon}$$

RMSProp + Momentum


SGD
SGD+Momentum
RMSProp
Adam


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

Normalization

Imagine that we have two features and a simple neural network. One is **age** with a range between 0 and 65, and another is **salary** ranging from 0 to 10 000.

Different scales of inputs cause different weights updates and optimizers steps towards the minimum. It also makes the shape of the loss function disproportional. In that case, we need to use **a lower learning rate** to not overshoot, which means a slower learning process.

$$\mu_n = \frac{1}{m}\sum_{i=0}^{m} x_{ni} \quad - \quad mean$$

$$\sigma_n = \sqrt{\frac{1}{m}\sum_{i=0}^{m} (x_{ni} - \mu_n)^2} \quad - \quad std$$

$$\hat{x}_n = \frac{x_n - \mu_n}{\sigma_n} \quad - \quad normalized\ input$$

Batch Normalization

In 2015 it was found that the input layer distribution is **constantly changing** due to weight update. In this case, the following layer always needs to adapt to the new distribution. It causes slower convergence and unstable training.

Batch Normalization presents a way to **control** and **optimize** the distribution after each layer. The process is identical to the input normalization, but we add two **learnable** parameters, **γ**, and **β**.

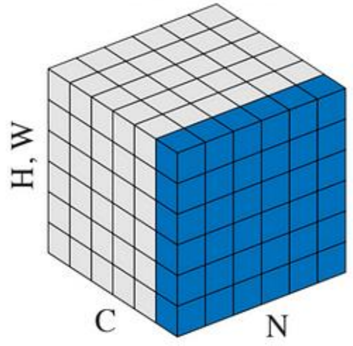
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i, \text{ minibatch means}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2, \text{ minibatch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \text{ normalization}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i), \text{ scale and shift}$$

Batch Norm



N — batch, C — channels, H,W— spatial width and height

Other types of noramlization