Regularization and Optimization in NN & Deep Learning

Dr. Santosh Kumar Vipparthi

Dept. of C.S.E

Website: https://visionintelligence.github.io/

Malaviya National Institute of Technology (MNIT), Jaipur

Regularization and Optimization in Deep Learning

Credits:

1. Deep Learning, Ian Goodfellow, Yoshua Bengio, Aaron Corville
2. An Introduction to Statistical Learning, Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani
3. http://cs231n.stanford.edu/
4. https://medium.com/@tm2761/regularization-hyperparameter-tuning-in-a-neural-network-f77c18c36cd3
5. https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0
6. https://srdas.github.io/DLBook/ImprovingModelGeneralization.html
7. http://laid.delanover.com/difference-between-l1-and-l2-regularization-implementation-and-visualization-in-tensorflow/
8. https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261
9. https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2
10. https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-ii-438fb4f6d135
11. https://medium.com/deeper-learning/glossary-of-deep-learning-batch-normalisation-8266dcd2fa82
12. https://medium.com/@krishna_84429/understanding-batch-normalization-1eaca8f2f63e

Some Basic Concepts

- Generalization
- Underfitting-Overfitting
- Bias-Variance

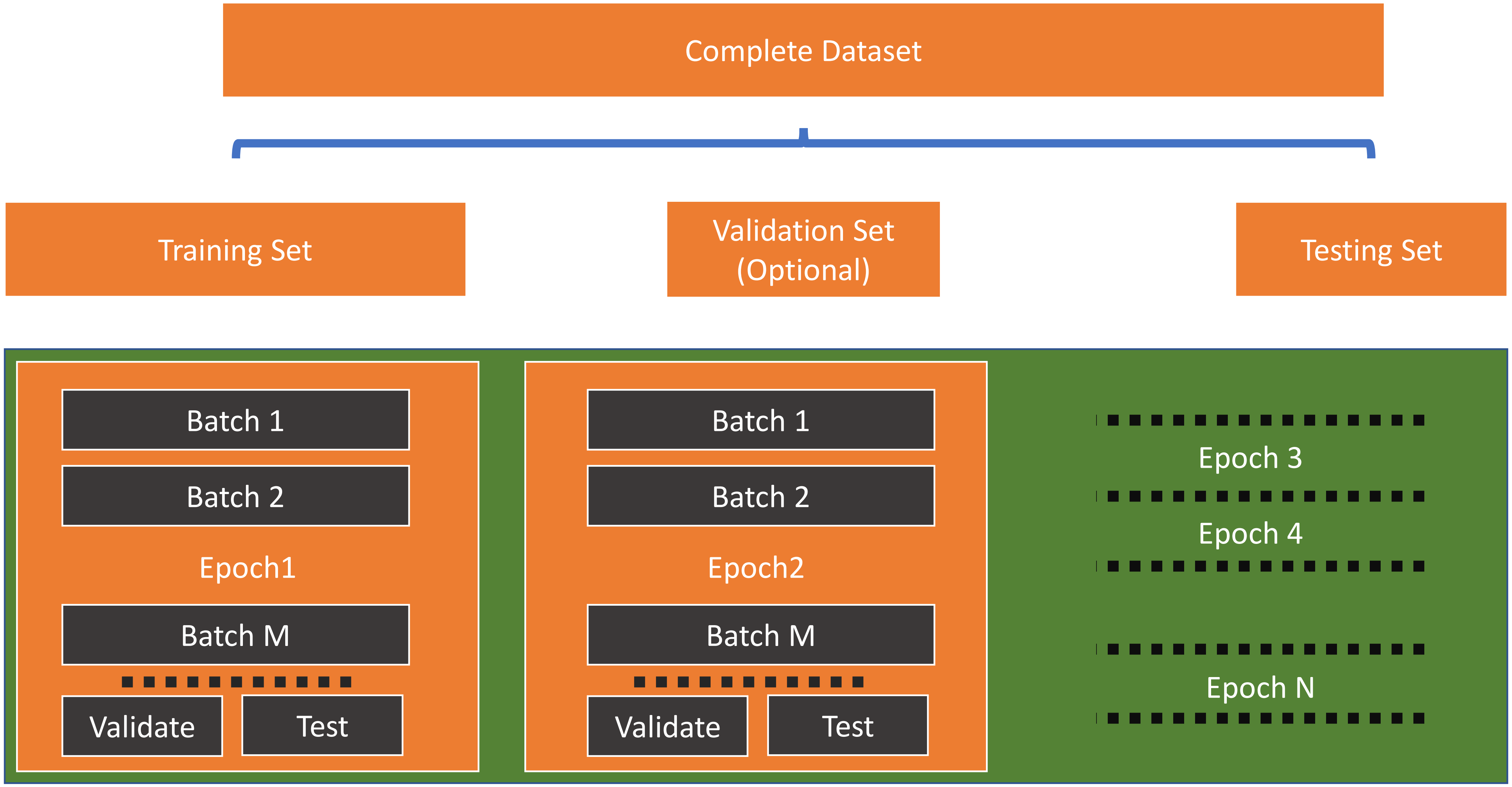
Regularization

- Parameter Norm-Penalties. (L1-norm and L2-norm)
- Dataset Augmentation
- Early Stopping
- Bagging and other ensemble methods
- Dropout

Optimization

- Stochastic Gradient Descent
- Parameter Initialization
- Adagrad
- RMSProp
- Batch Normalization

Data Management for Training and Evaluation

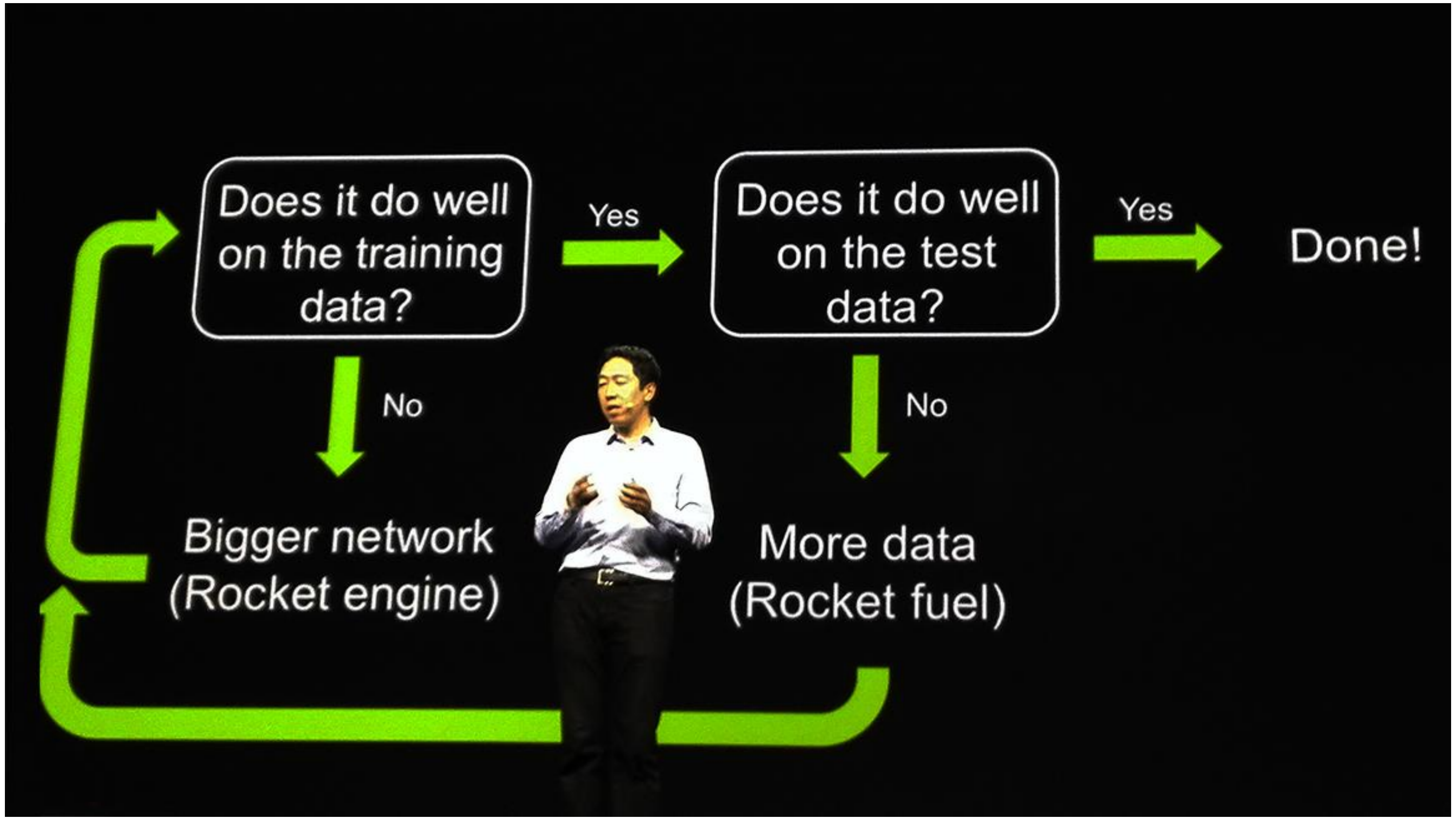


Generalization

- The ability of a trained model to perform well over the test data is known as its Generalization ability. There are two kinds of problems that can afflict machine learning models in general:

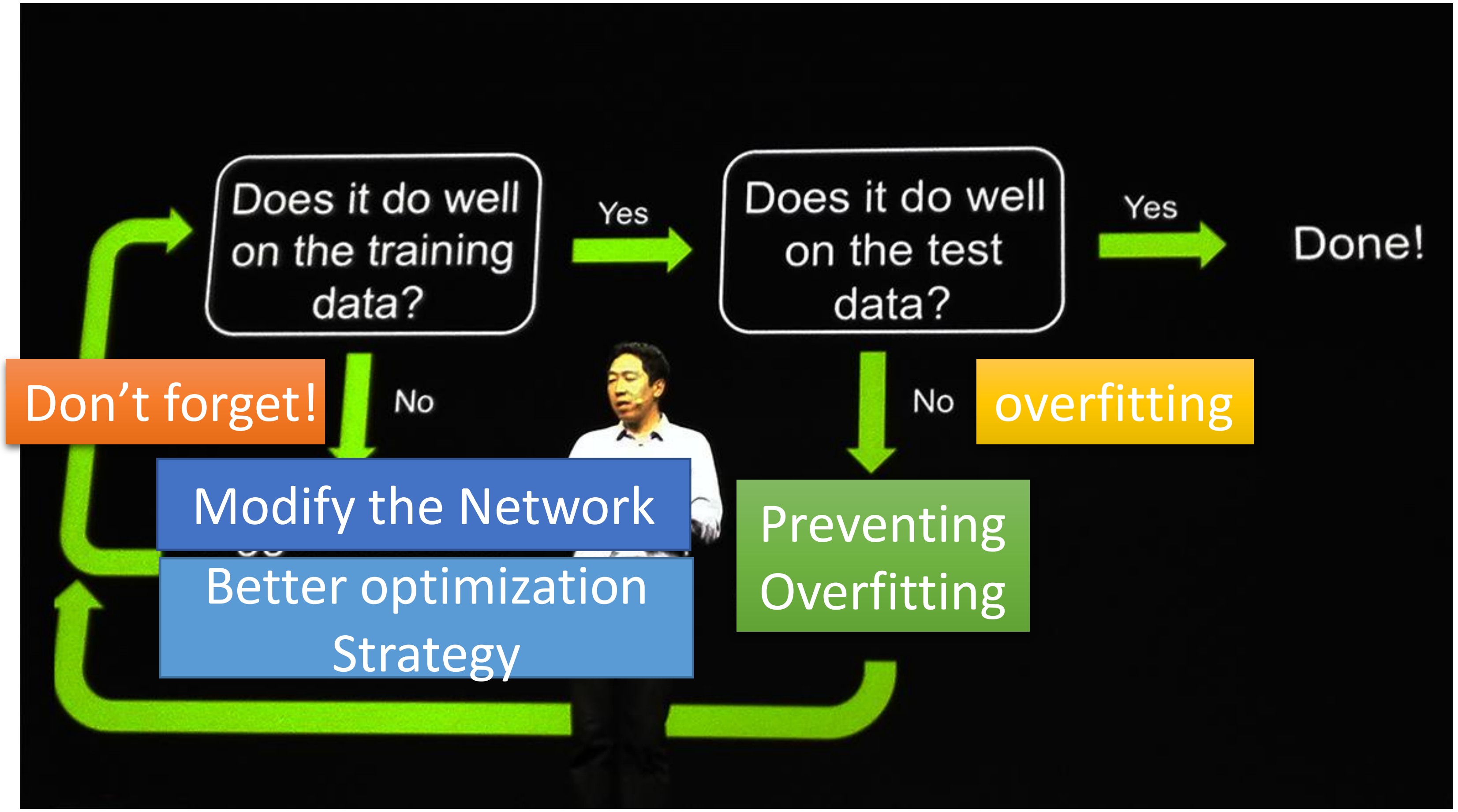
- Even after the model has been fully trained such that its training error is small, it exhibits a high test error rate. This is known as the problem of Overfitting.
- The training error fails to come down in-spite of several epochs of training. This is known as the problem of Underfitting

Recipe for Learning


Does it do well on the training data?
Yes
Does it do well on the test data?
Yes
Done!
No
No
Bigger network (Rocket engine)
More data (Rocket fuel)

http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Recipe for Learning


Does it do well on the training data?
Yes
Does it do well on the test data?
Yes
Done!
Don't forget!
No
No
overfitting
Modify the Network
Better optimization Strategy
Preventing Overfitting

http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

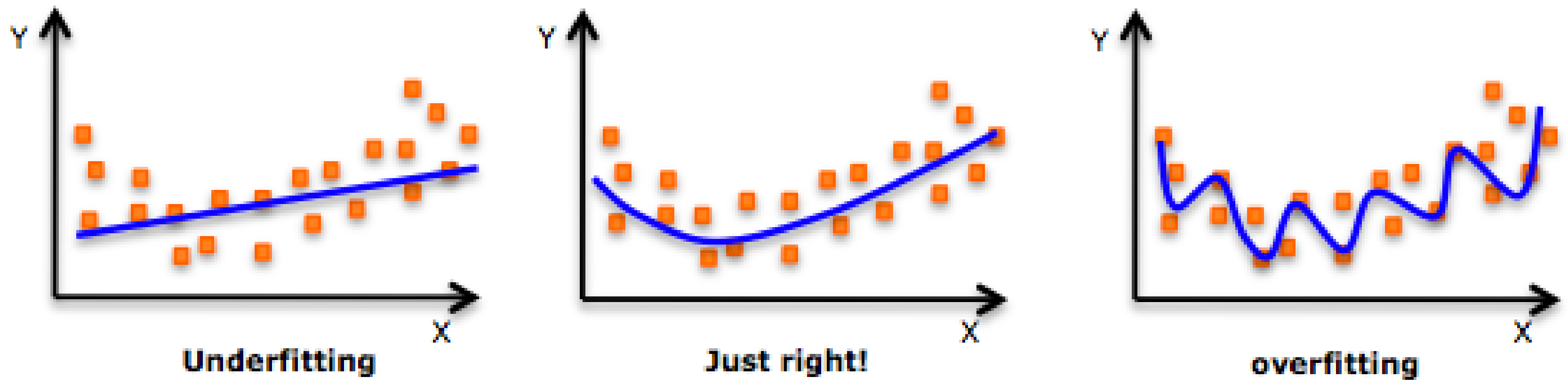
Underfitting-Overfitting

- This becomes especially problematic as you make your model increasingly complex.
- Underfitting is a related issue where your model is not complex enough to capture the underlying trend in the data.
- The problem of overfitting is not limited to computers, humans are often no better.

Underfitting-Overfitting

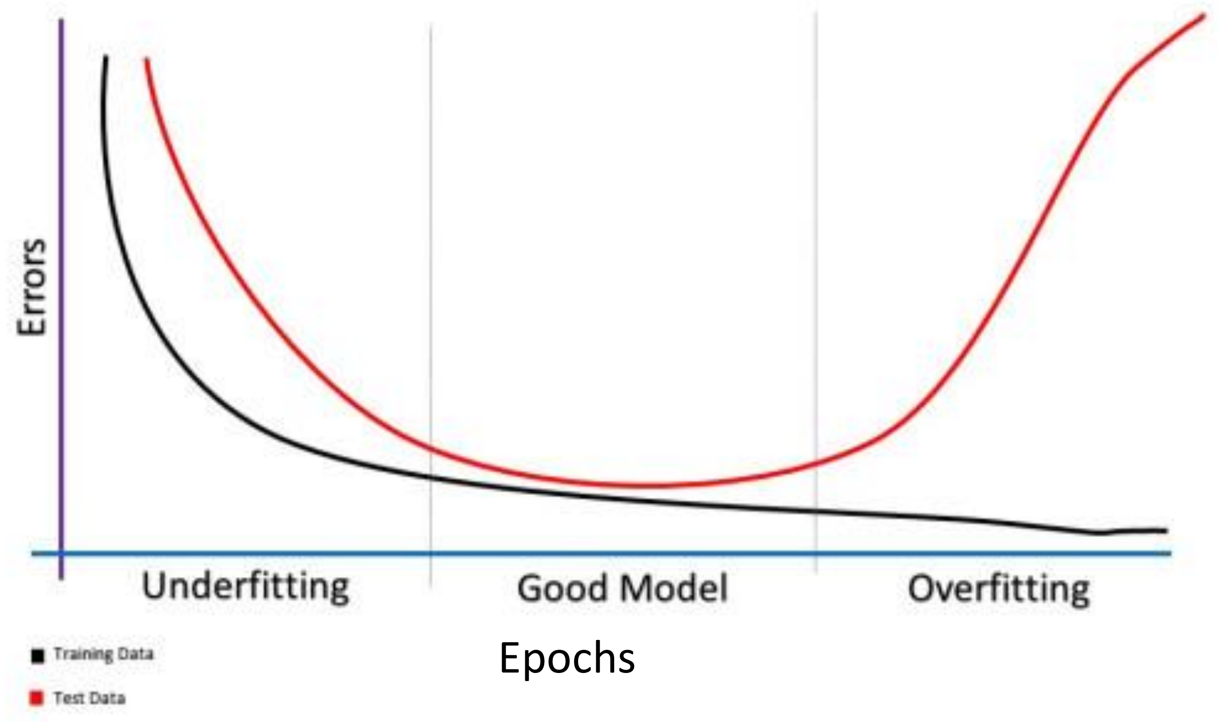
- For instance, say you had a bad experience with an XYZ Airline, maybe the service wasn’t good, or that the airline was riddled with delays.
- You might be tempted to say that all flights on XYZ airline are bad.
- This is called **overfitting** whereby we overgeneralize something, which otherwise, might have been us just having a bad day.

Underfitting-Overfitting



Source: Quora: Luis Argerich

Underfitting-Overfitting



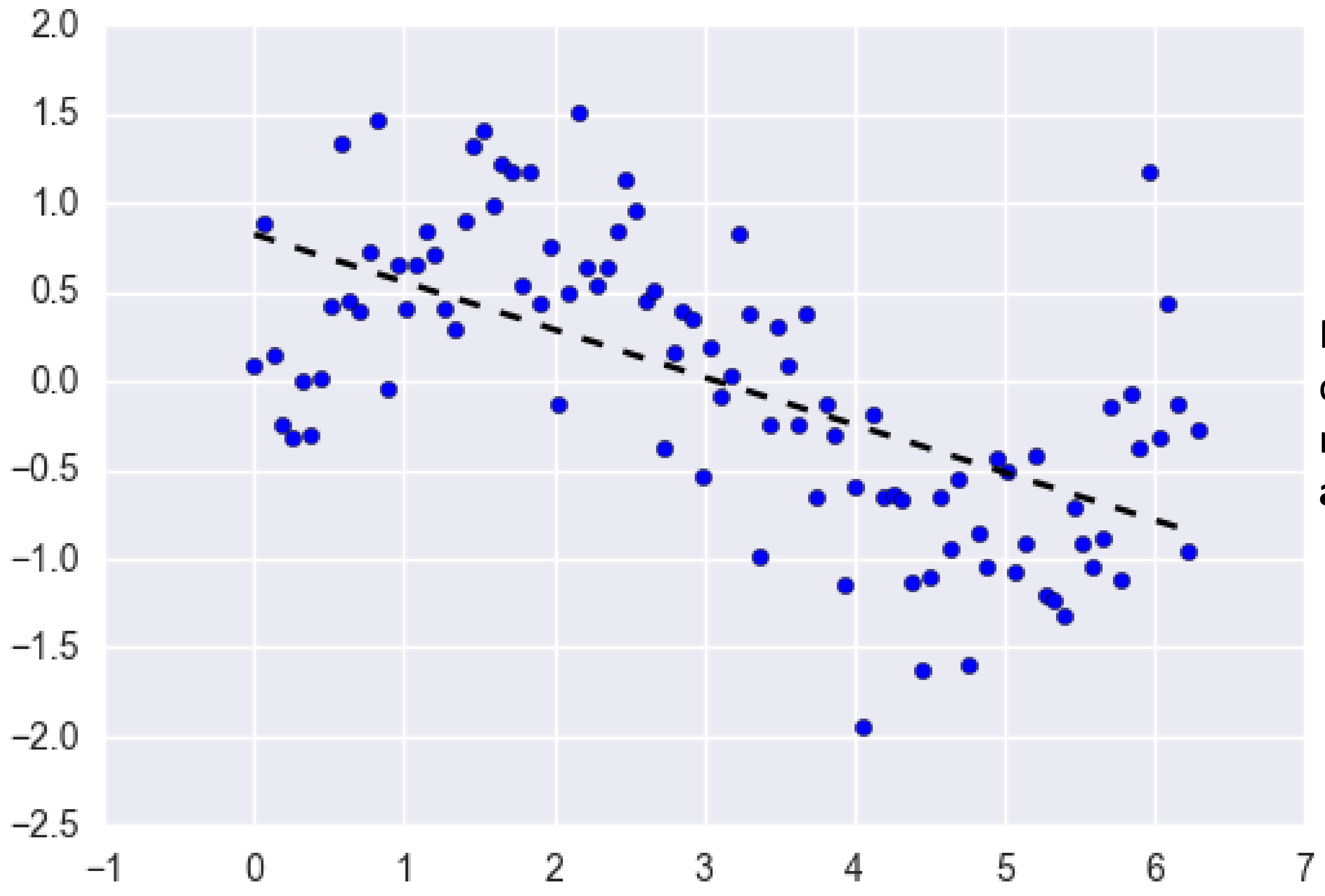
Source: https://meditationsonbianddatascience.com/2017/05/11/overfitting-underfitting-how-well-does-your-model-fit/

Bias-Variance

- **Bias** is the **difference** between your **model's expected predictions** and the **true values**.
- That might sound strange because shouldn't you "expect" your predictions to be close to the true values?
- Well, it's not always that easy because some algorithms are simply too rigid to learn complex signals from the dataset.

Bias-Variance

- **Imagine fitting a linear regression to a dataset that has a non-linear pattern:**



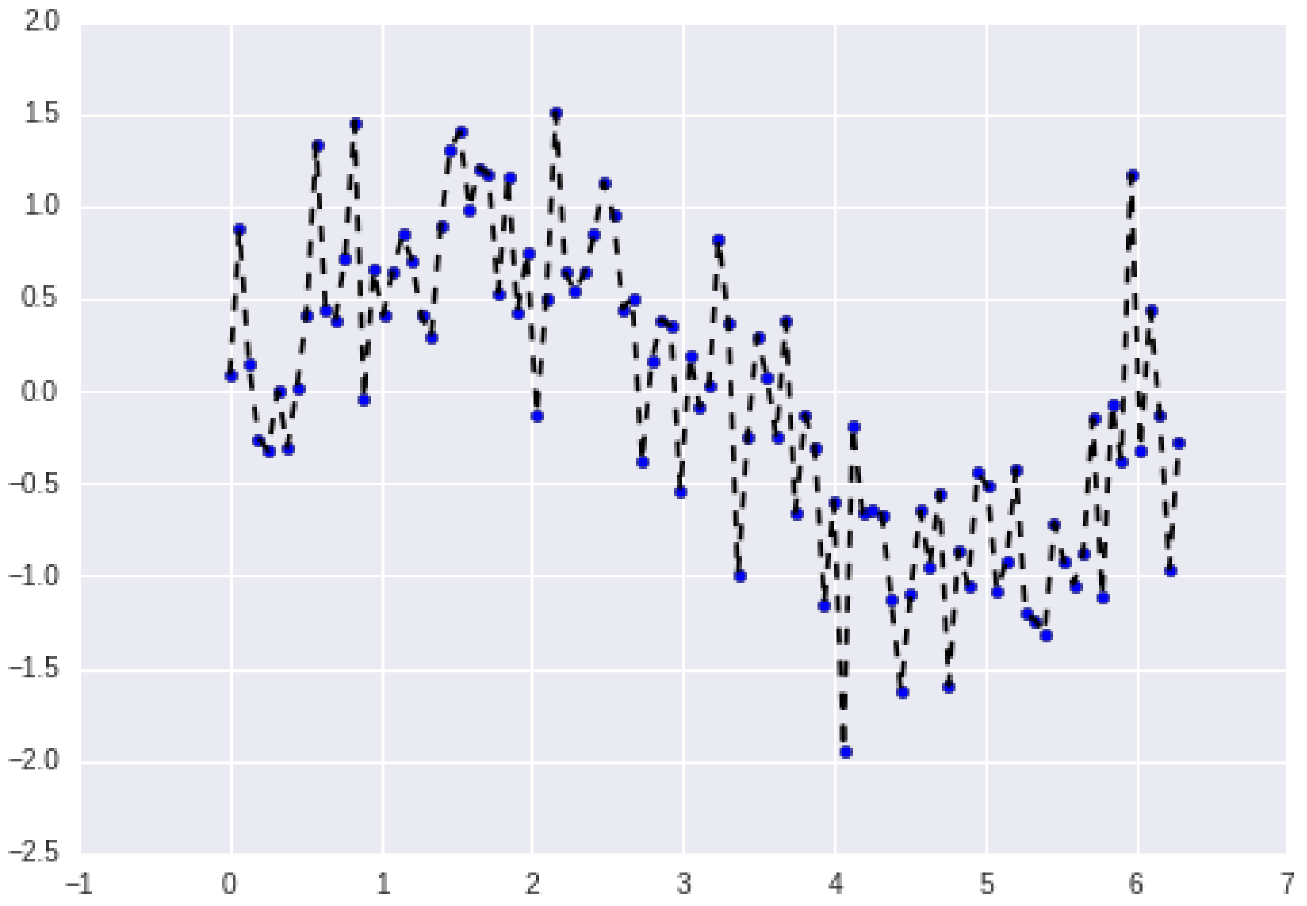
No matter how many more observations you collect, a linear regression won't be able to model the curves in that data! **This is known as under-fitting**

Bias-Variance

- Variance refers to your algorithm's **sensitivity to specific sets of training data**.
- High variance algorithms will **produce drastically different models** depending on the **training set**.

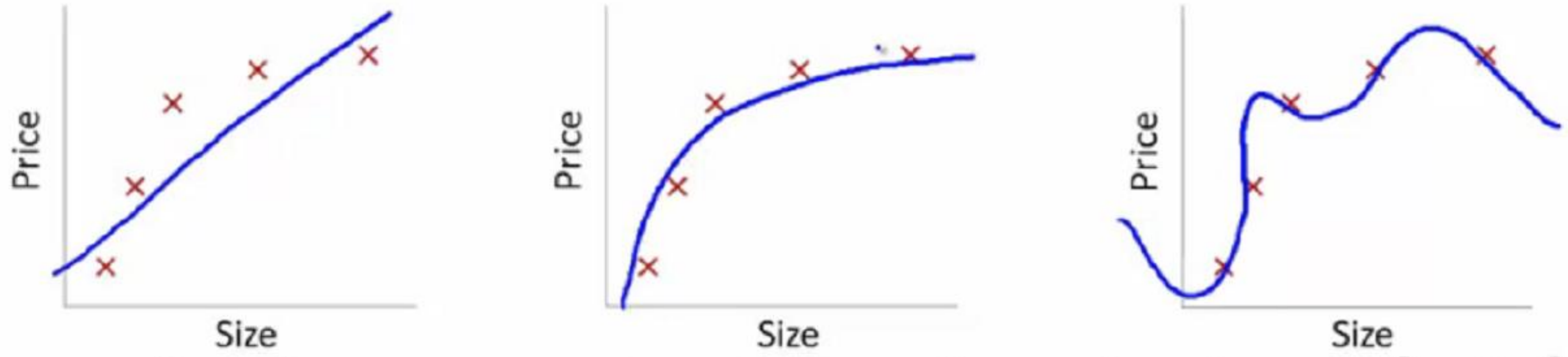
Bias-Variance

- For example, imagine an algorithm that **fits a completely unconstrained, super-flexible model** to the **same dataset** from above:



As you can see, this unconstrained model has basically memorized the training set, including all the noise. **This is known as over-fitting.**

Bias-Variance



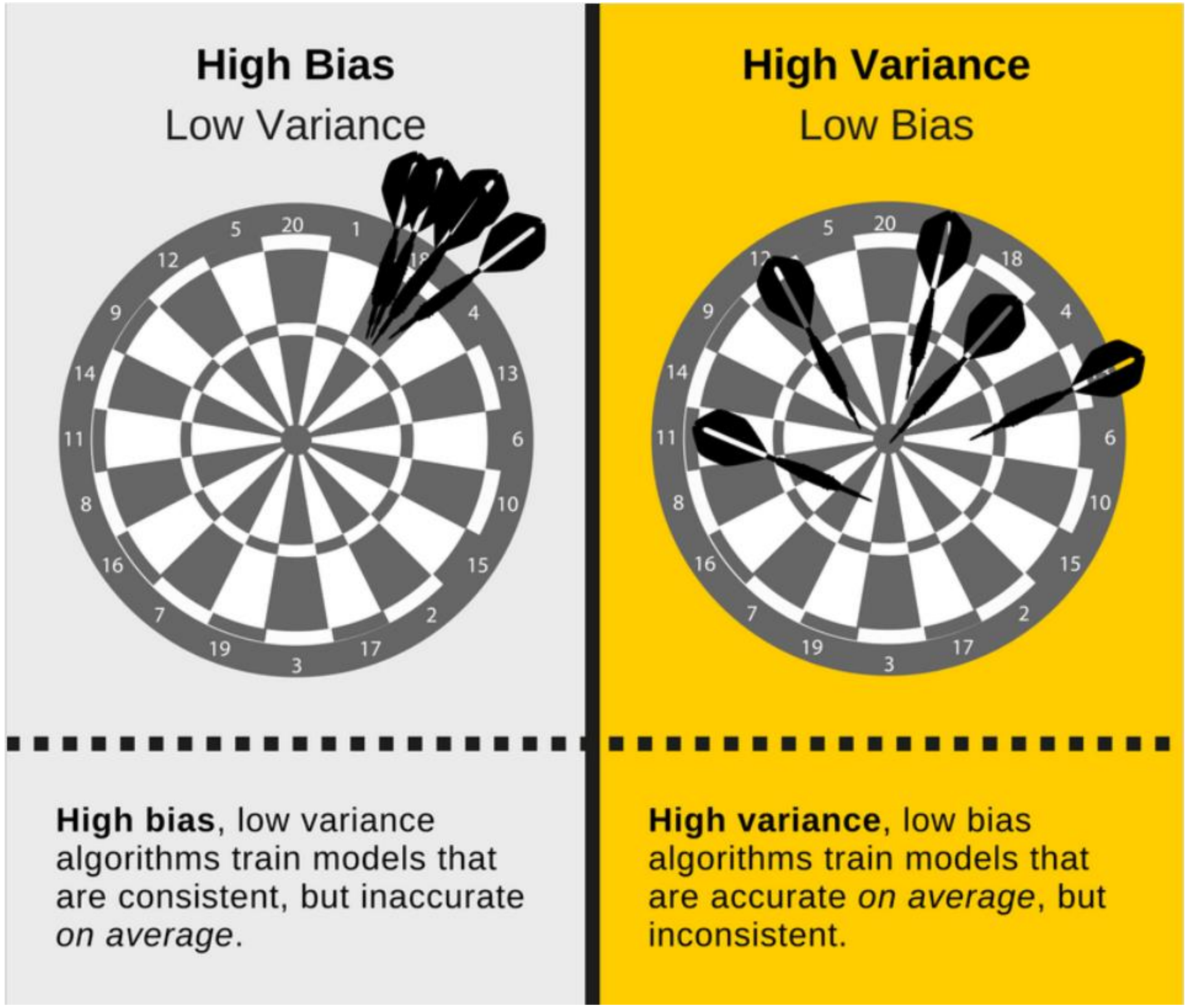
$\theta_0 + \theta_1 x$ | $\theta_0 + \theta_1 x + \theta_2 x^2$ | $\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

High bias (underfit) | "Just right" | High variance (overfit)

Source: Coursera's ML course, taught by Andrew Ng

Bias-Variance

Here’s what those 5 models tell you about the chosen Algorithms

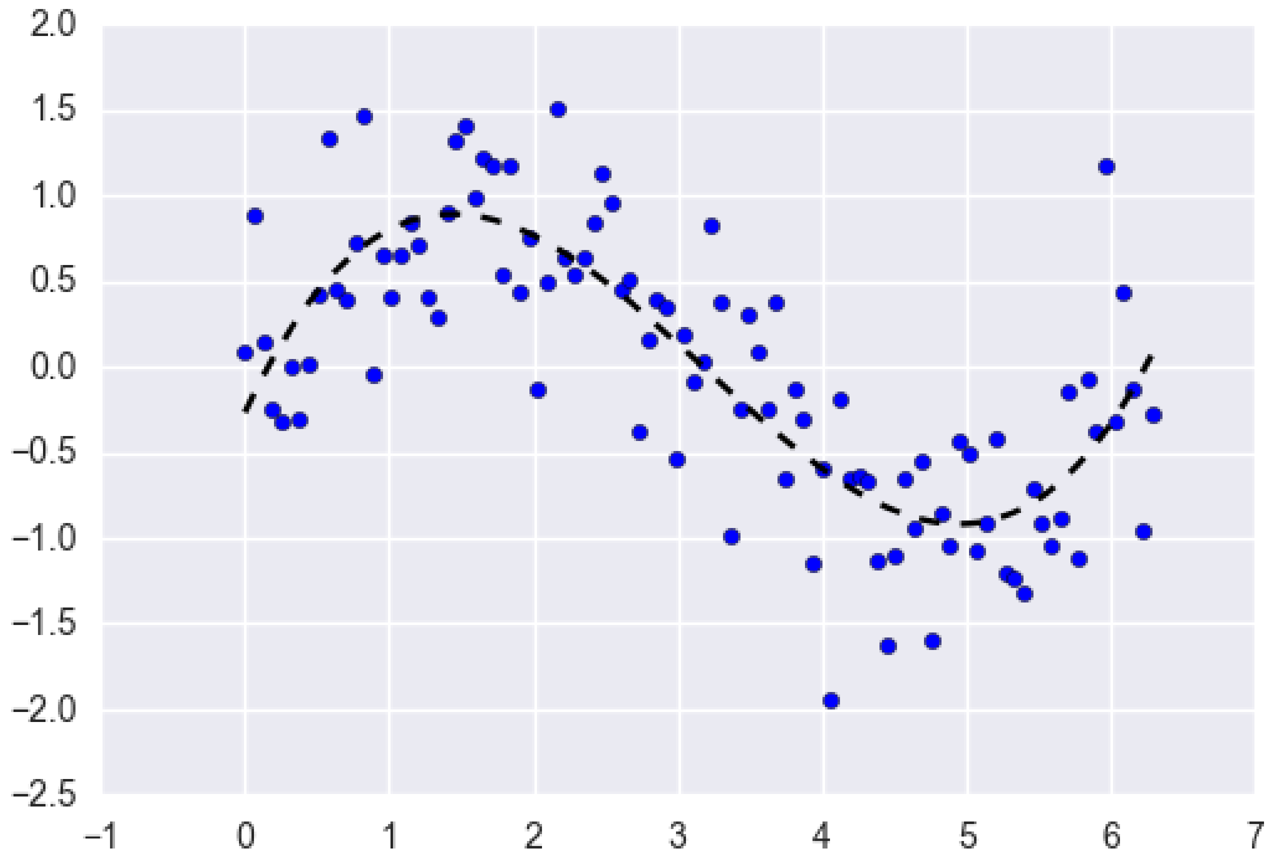


High bias, low variance algorithms train models that are consistent, but inaccurate *on average*.

High variance, low bias algorithms train models that are accurate *on average*, but inconsistent.

Bias-Variance

- Finally, **an optimal balance of bias and variance leads** to a model that is neither overfit nor underfit:



This is the ultimate goal of supervised machine learning –
To isolate the signal from the dataset while ignoring the noise!

How to Combat Overfitting?

- **Two ways to combat overfitting:**

1. Use more training data. The more you have, the harder it is to overfit the data by learning too much from any single training example.

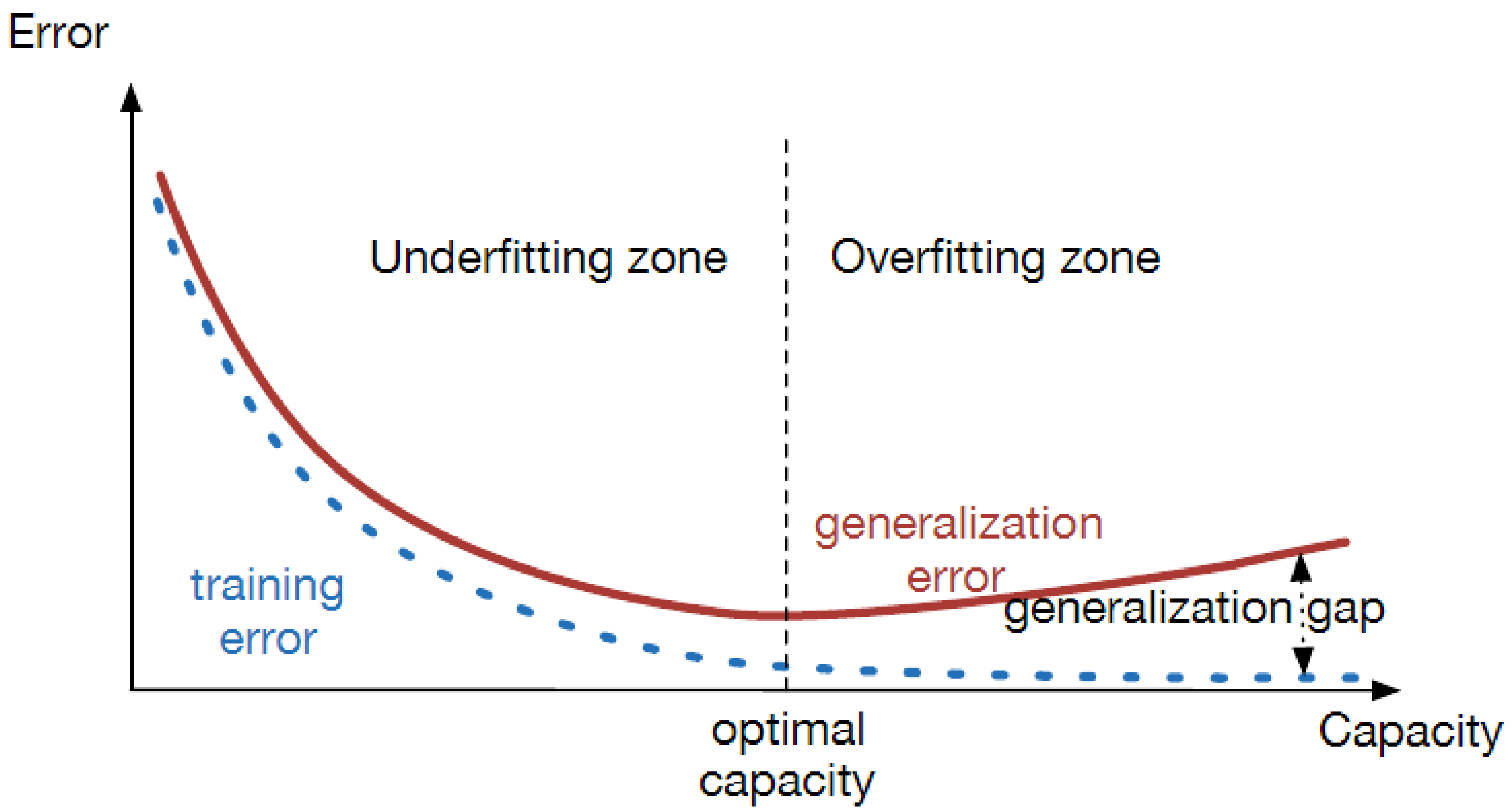
2. Use regularization. Add in a penalty in the loss function for building a model

How to Combat Overfitting?

$$Cost = \frac{\sum_1^n((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n} + \lambda \sum_{i=0}^{1} \beta_i^2$$

- The fist piece of the sum is our **normal cost function**.
- The second piece is a **regularization term** that adds a penalty for **large beta coefficients**
- With these two elements in place, the cost function now balances between two priorities: **explaining the training data** and **preventing that explanation from becoming overly specific.**

Regularization in Machine Learning



Illustrates the relationship between model capacity and the concepts of underfitting and overfitting by plotting the training and test errors as a function of model capacity. When the capacity is low, then both the training and test errors are high. As the capacity increases, the training error steadily decreases, but the test error initially decreases, but then starts to increases due to overfitting. Hence the optimal model capacity is the one at which the test error is at a minimum

Regularization in Machine Learning

- How to make an algorithm/model perform well not just on the training data, but also on new inputs?

- Many strategies are designed explicitly to **reduce the test error**, possibly at the expense of **increased training error**. These strategies are known collectively as **regularization**.

Regularization in Machine Learning

- The following factors determine how well a model is able to generalize from the training dataset to the test dataset:

- The **model capacity** and its relation to **data complexity**:
- In general if the model capacity is less than the data complexity then it leads to underfitting, while if the converse is true, then it can lead to overfitting. Hence, the objective is to choose a model whose capacity matches the complexity of the training and test datasets.
- Even if the model capacity and the data complexity are well matched, we can still encounter overfitting due to an insufficient amount of training samples.

How to Regularize?

- Put extra constraints on the model. Example: Add restrictions on the parameter values.
- Add extra terms (as penalties) in the *objective function.* Indirectly putting constraint on the parameter values.

Regularization Techniques

- Parameter Norm-Penalties. (L1-norm and L2-norm)
- Dataset Augmentation
- Early Stopping
- Bagging and other ensemble methods
- Dropout

Parameter Norm-Penalties

- Limit the capacity of models such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty **Ω(θ)** to the **objective function J**.

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J.

Setting α to zero results in no regularization. Larger values of α correspond to more regularization

We use the vector **w** to indicate all of the weights that should be affected by a norm penalty, while the vector **θ** denotes all of the parameters, including both **w** and the unregularized parameters.

L2-Regularization

- The L2 parameter norm penalty commonly known as **weight decay**.
- L2-regularization is also known as **ridge regression** or **Tikhonov regularization**
- L2 regularization drives the weights closer to origin by adding a regularization term $\Omega(\boldsymbol{\theta}) = 1/2||\mathbf{w}||^2_2$ to the objective function.

$$||\mathbf{w}||_2 = (w_1^2 + w_2^2 + ... + w_N^2)^{\frac{1}{2}}$$

2-norm (also known as L2 norm or Euclidean norm)

L2-Regularization

- Such a model has following total objective function:

$$J(w; X, y) = \alpha/2(w'w) + J(w; X, y)$$

where ' denotes transpose. To simplify the presentation, we assume no bias parameter, so **θ** is just **w.**

- The corresponding **parameter gradient**

$$\nabla_w J\hat{}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$$

L2-Regularization

- To take a single gradient step to update the weights, we perform this update:

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

- Written another way, the update is:

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

- We can see that the addition of the **weight decay term** has modified the learning rule to **multiplicatively shrink the weight vector** by a **constant factor on each step**, just before performing the usual gradient update.

L2-Regularization

- What are the effects over the entire course of training?

- The L2-regularization causes the learning algorithm to “perceive” the **input X** as having higher variance. Thus, it shrink the weights on features
- The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors.
- Due to **multiplicative interactions** between weights and inputs, this has the appealing property of encouraging the network to use **all of its inputs a little rather that some of its inputs a lot**.

L1-Regularization

- The regularized objective function J(w; X, y) is given by:

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

- With the corresponding gradient (actually, sub-gradient):

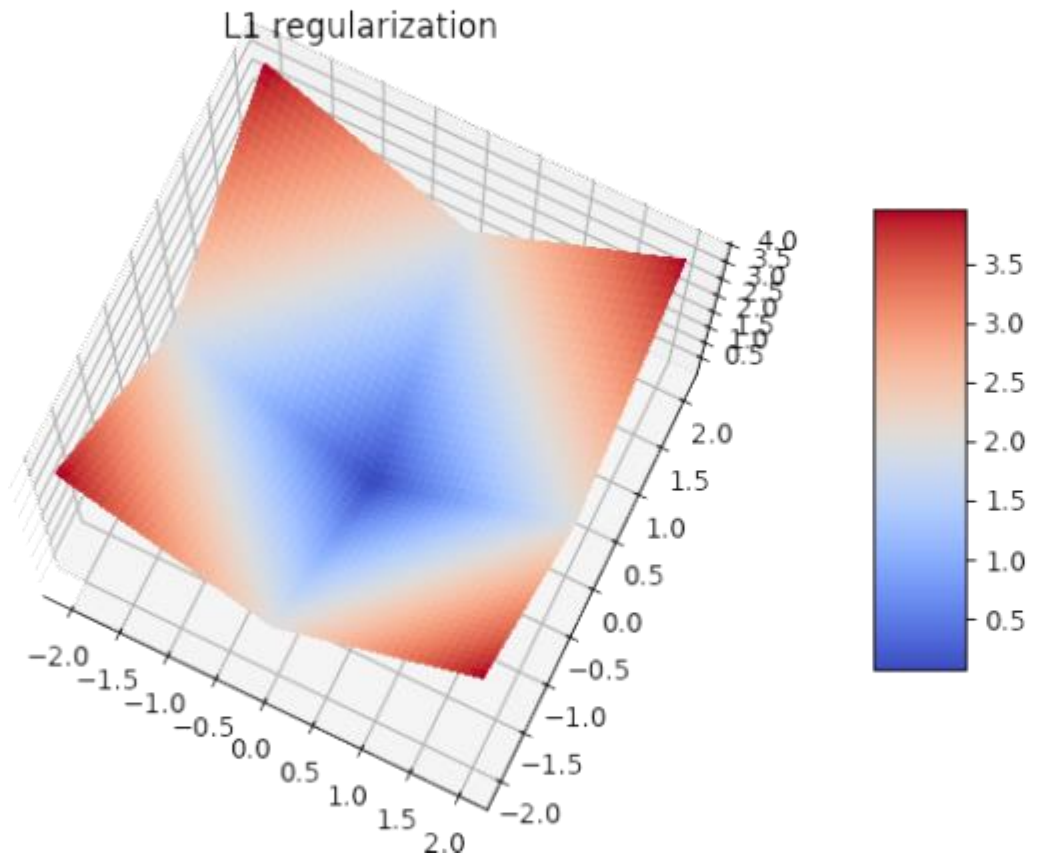
$$\nabla_w J\hat{}(w; X, y) = \alpha.\text{sign}(w) + \nabla_w J(w; X, y)$$

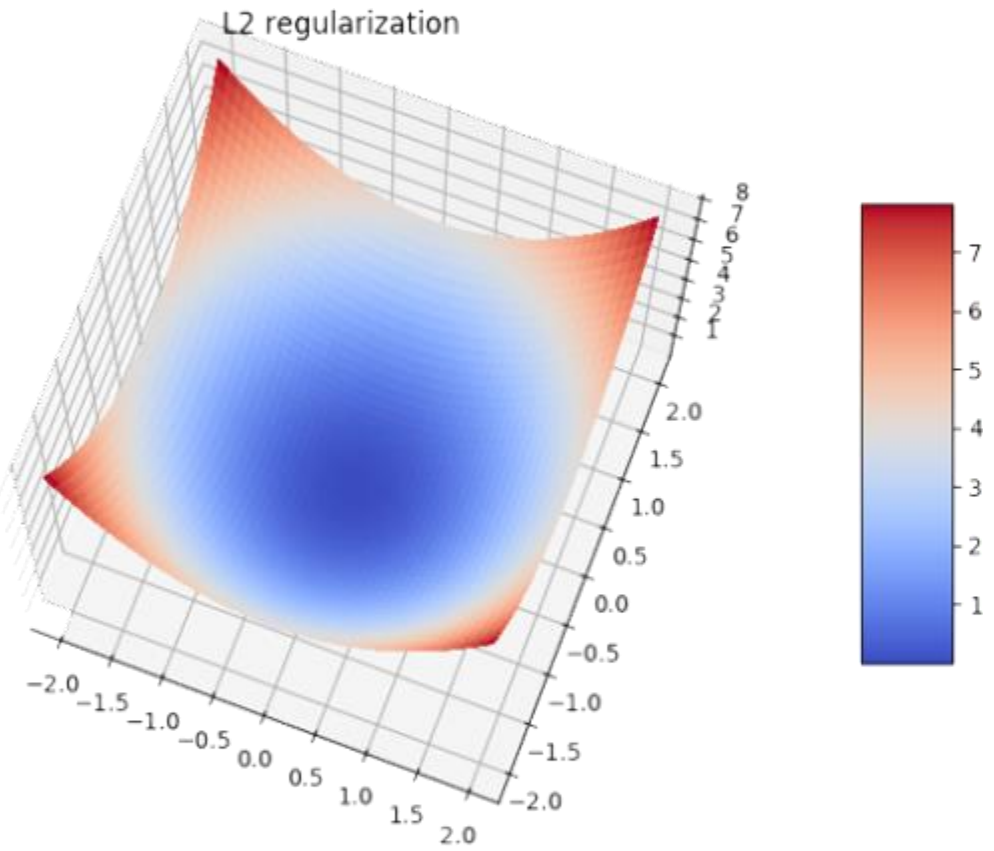
where **sign(w)** is simply the **sign** of **w** applied element-wise.

$$||\mathbf{w}||_1 = |w_1| + |w_2| + \ldots + |w_N|$$

1-norm (also known as L1 norm)

L1 and L2-Regularization





The graphs above show how the functions used in L1 and L2 regularization look like. The penalty in both cases is **zero** in the **center** of the plot, but this also implies that the weights are zero and the model will not work. The **values of the weights try to be as low as possible to minimize this function**, but inevitably they will leave the center and will head outside.

L1 and L2-Regularization

- In case of L2 regularization, going towards any direction (from the center) is okay because, as we can see in the plot, the function increases equally in all directions. Thus, L2 regularization mainly focuses on keeping the weights as low as possible.
- In contrast, L1 regularization’s shape is diamond-like and the weights are lower in the corners of the diamond. These corners show where one of the axis/feature is zero thus leading to sparse matrices.
- Note how the shapes of the functions shows their differentiability: L2 is smooth and differentiable and L1 is sharp and non-differentiable.

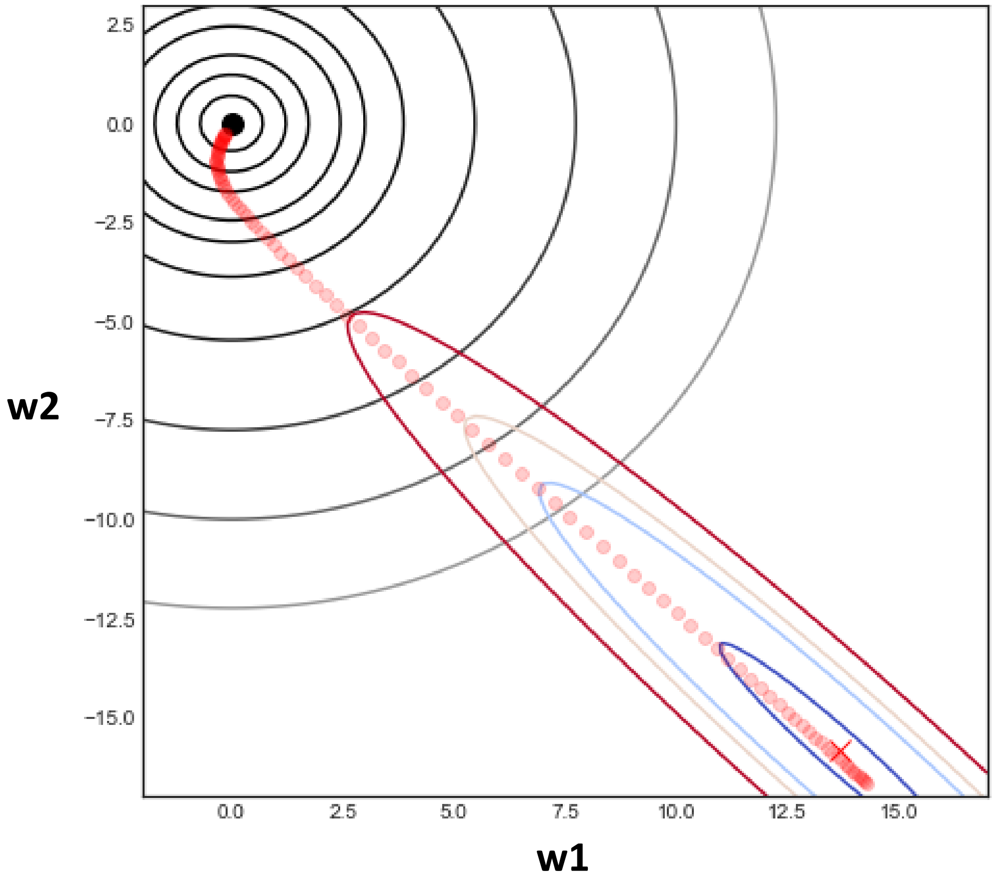
L1 and L2-Regularization

- In few words, L2 will aim to find small weight values whereas L1 could put all the values in a single feature.
- L1 and L2 regularization methods are also combined in what is called **elastic net regularization**.

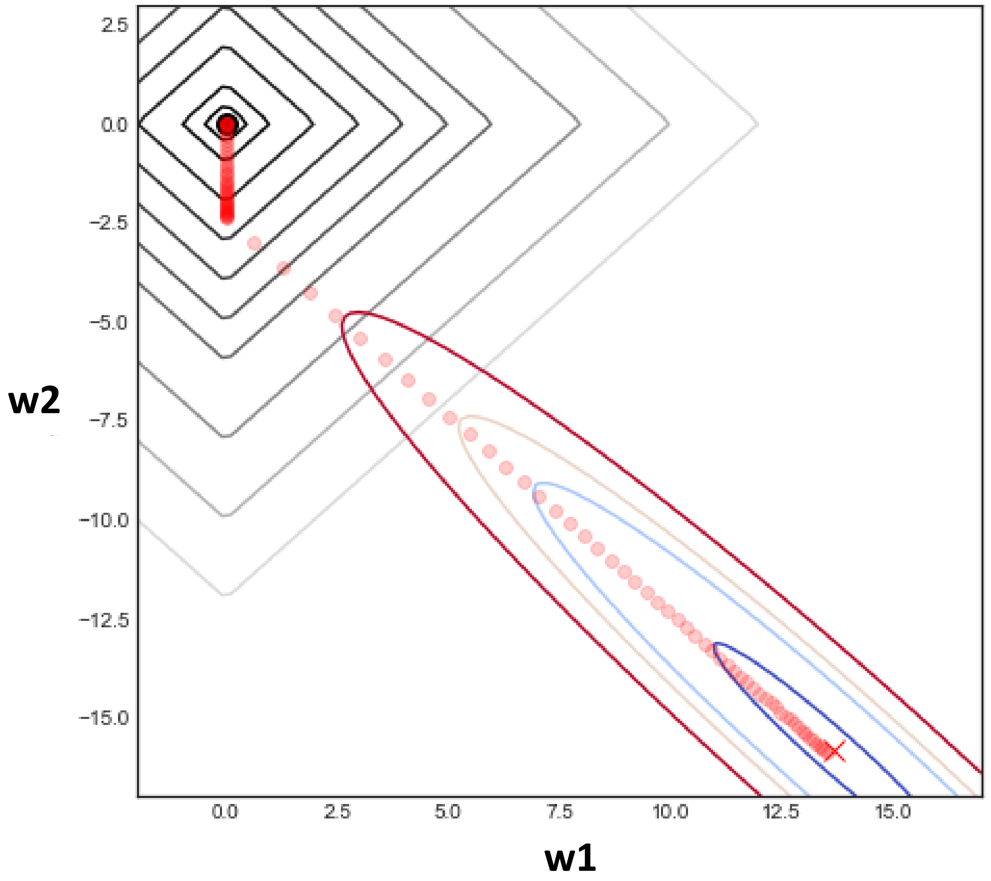
L1 Vs L2-Regularization

For a Linear Regression problem with only 2 parameters **w1** and **w2**

L2/Ridge solution as a function of **α, L2 and J(w,X,y)**



L1/Lasso solution as a function of **α, L1 and J(w,X,y)**

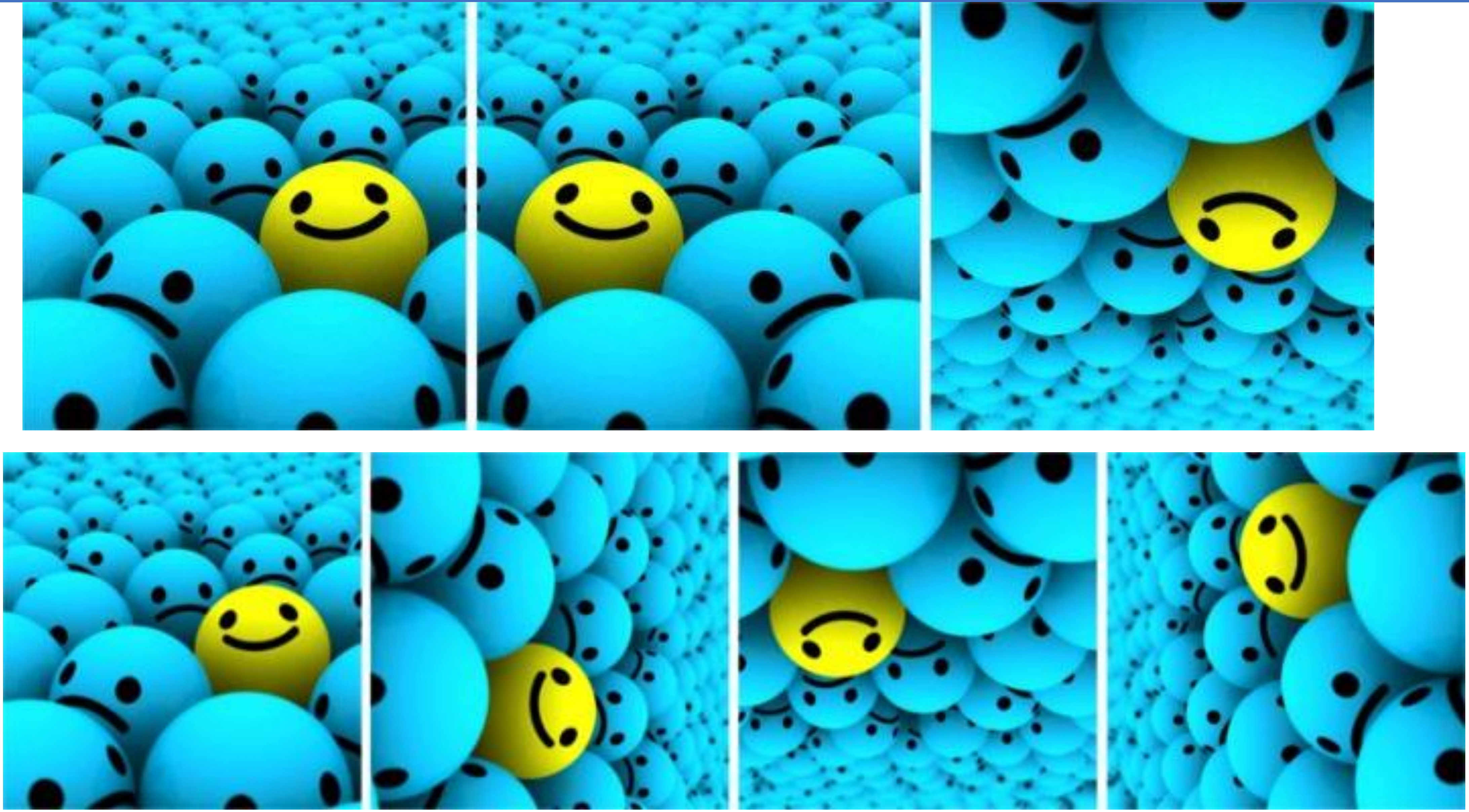


Dataset Augmentation

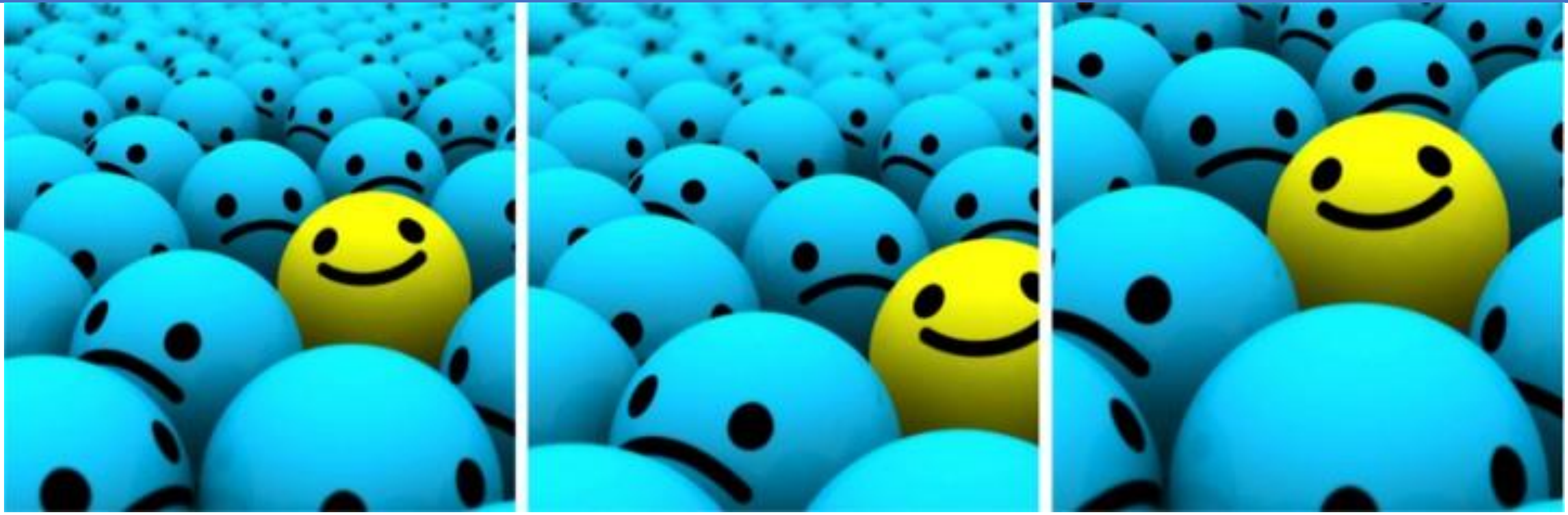
- The best way to make a machine learning model generalize better is to train it on more data.
- How to generate more data?

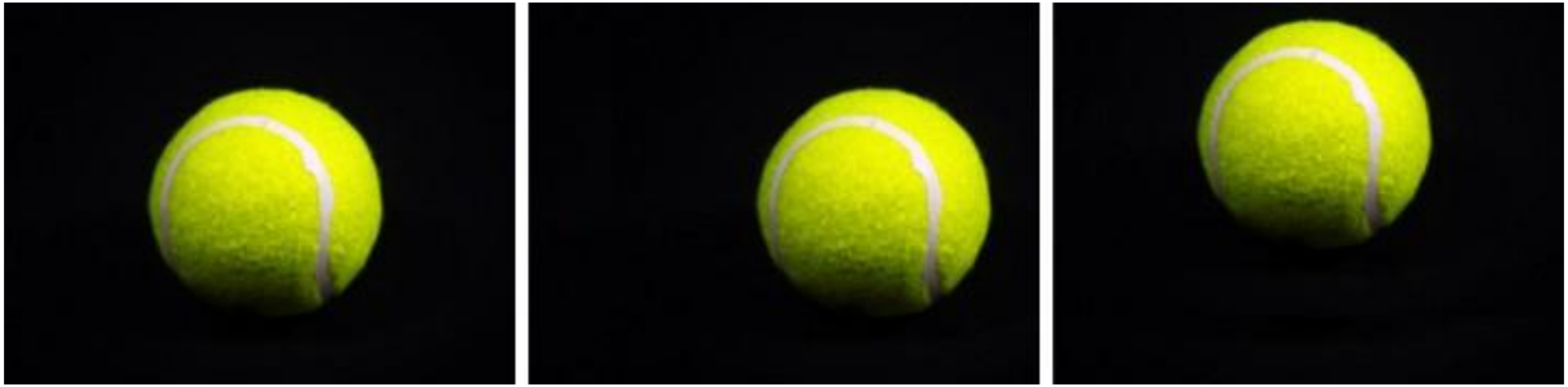
- Label more data.
- Create fake data.
- Injecting noise in the data
- Inject noise to the model parameters
- Inject noise to the output
- For image data: rotation, translation and other transformation, inject noise

Dataset Augmentation

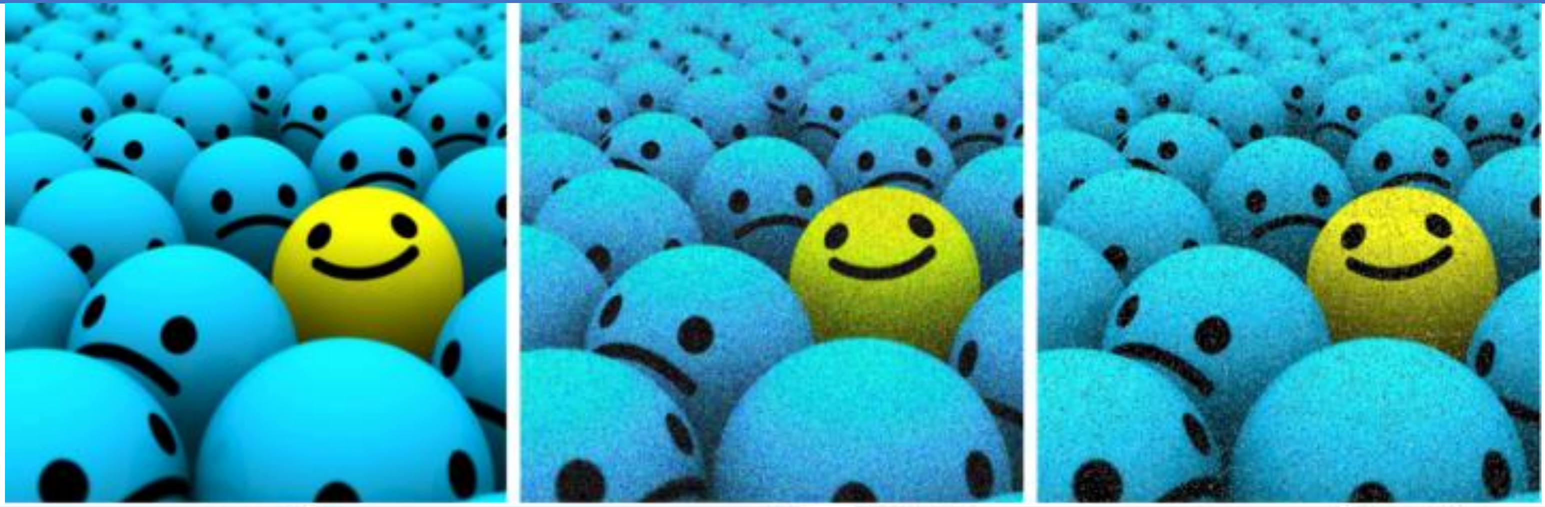


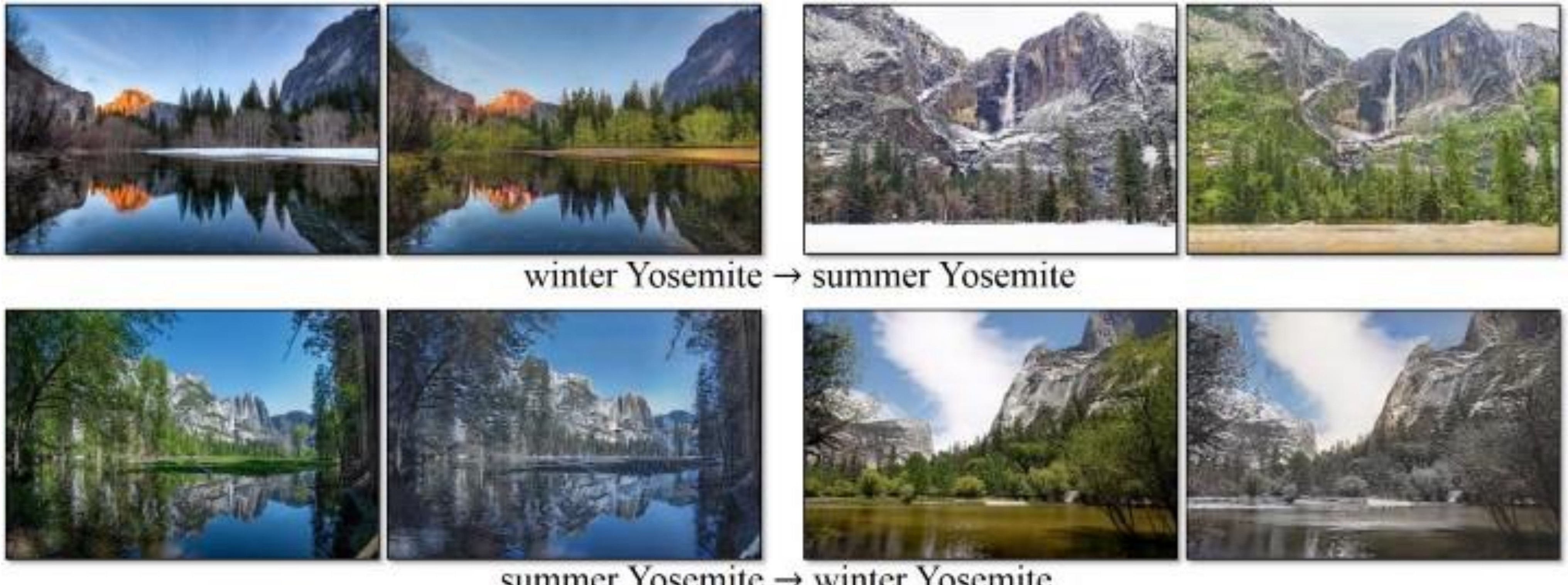
Dataset Augmentation



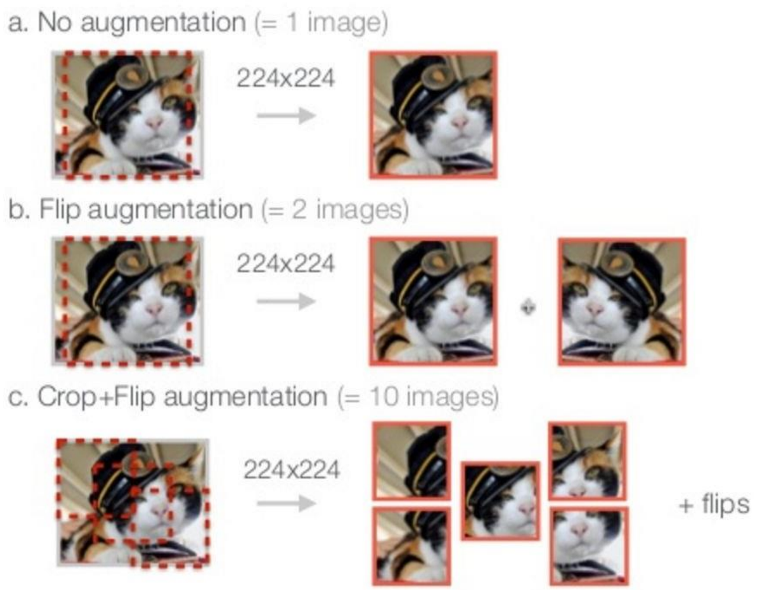


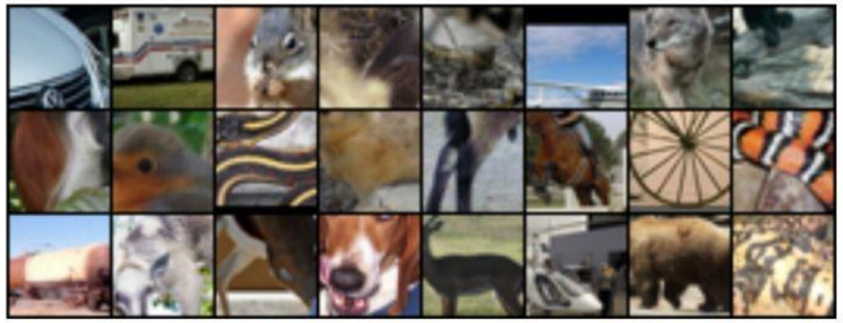
Dataset Augmentation

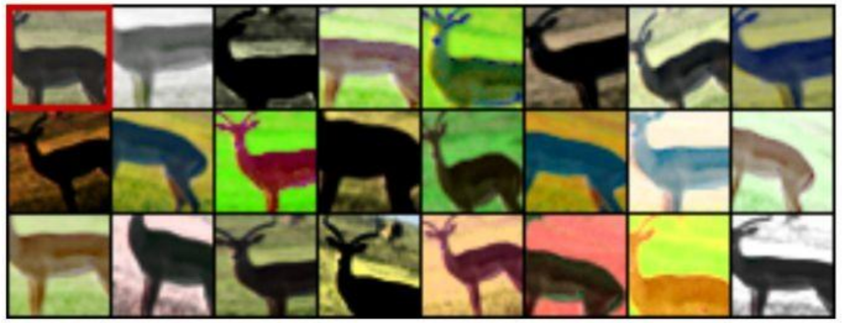



winter Yosemite → summer Yosemite
summer Yosemite → winter Yosemite

Dataset Augmentation


a. No augmentation (= 1 image)
224x224
b. Flip augmentation (= 2 images)
224x224
+
c. Crop+Flip augmentation (= 10 images)
224x224
+ flips


Example of classes

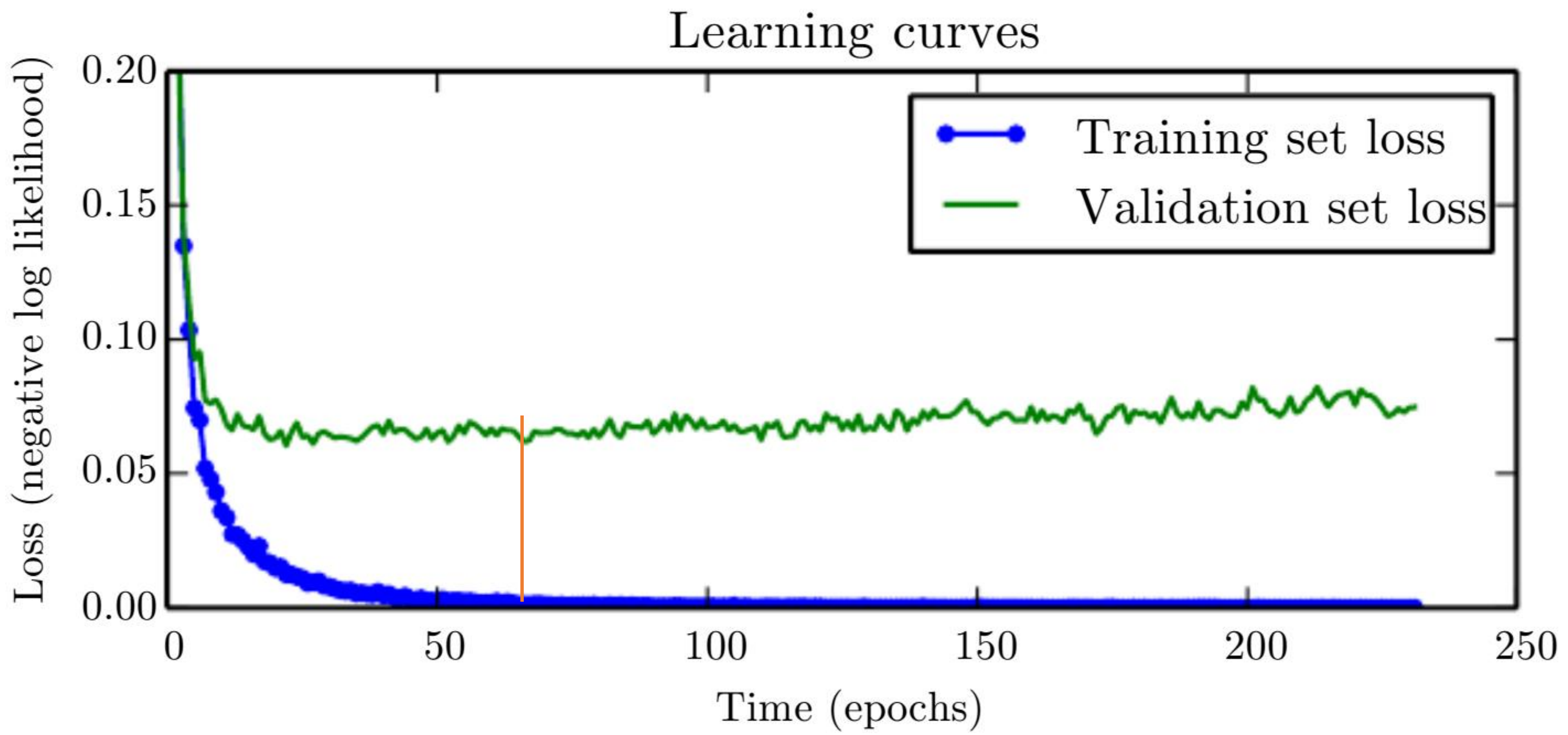

Example of examples for one class

Early Stopping

- When training large models with sufficient representational capacity to overfit the task, it is often observed that training error decreases steadily over time, but validation set error begins to rise again.

- This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.

Early Stopping

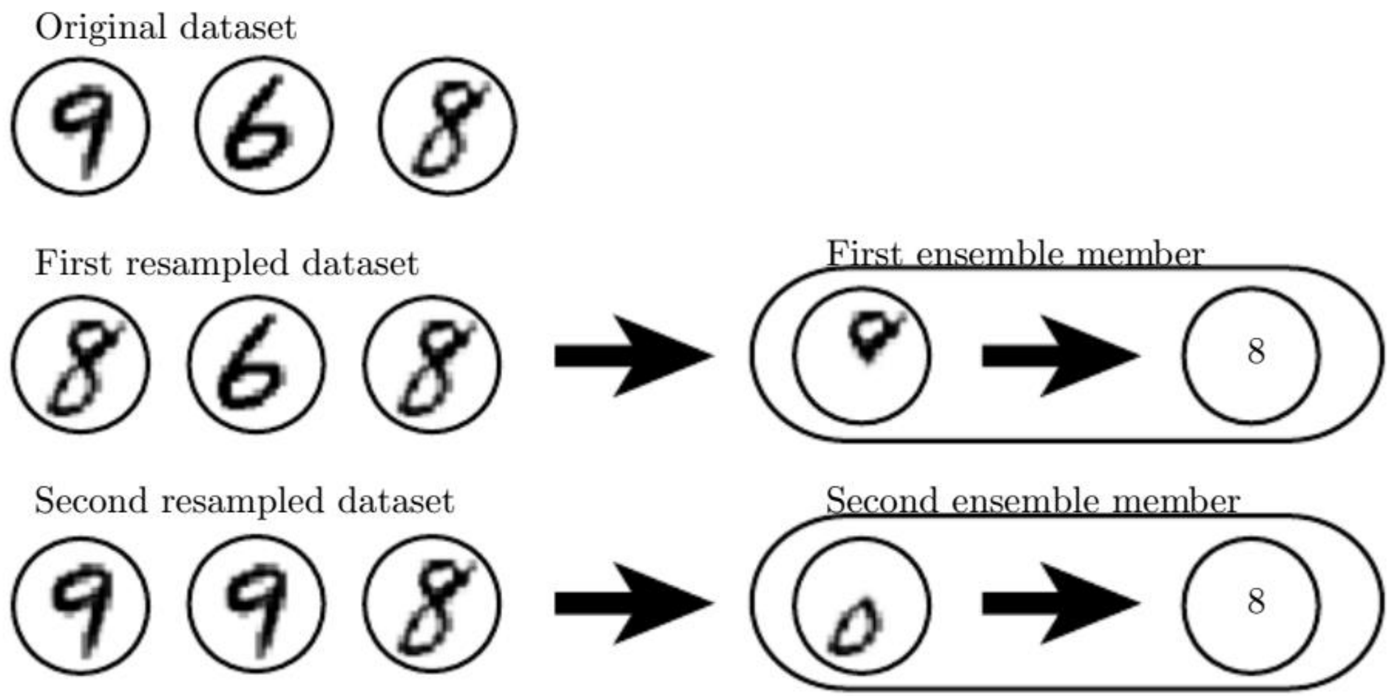


Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or epochs). In this example, a network is trained on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve

Bagging and Other Ensemble Methods

- Bagging is a technique for reducing generalization error by combining several models.
- The idea is to train several different models separately, then have all of the models vote on the output for test examples.
- This is an example of a general strategy in machine learning called model averaging. Techniques employing this strategy are known as ensemble methods

Bagging and Other Ensemble Methods



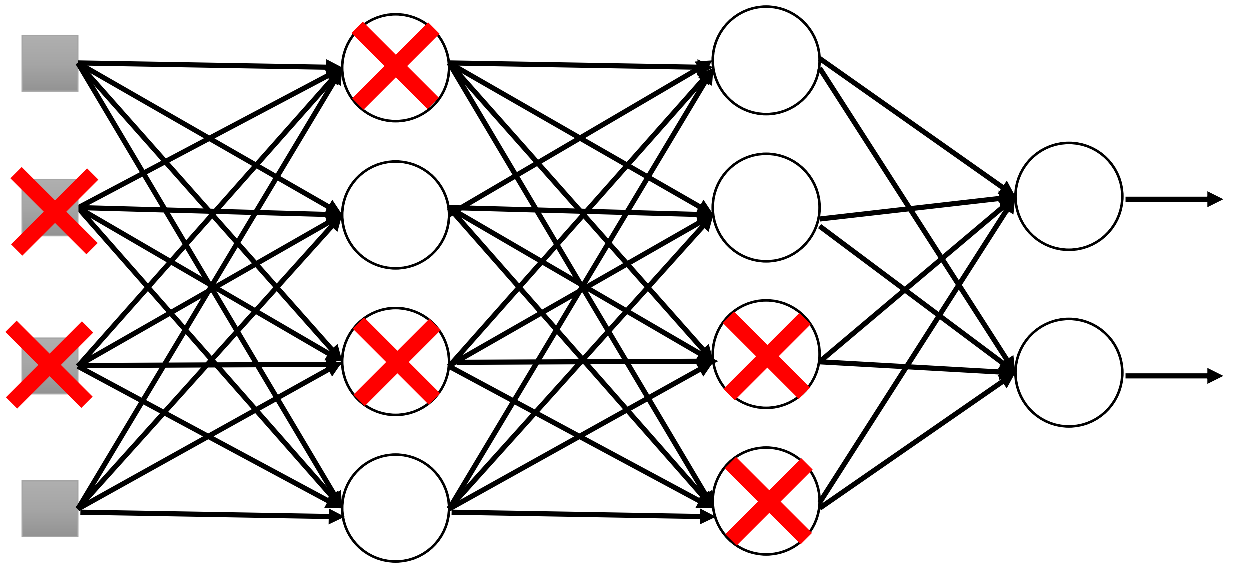
A cartoon depiction of how bagging works. Suppose we train an ‘8’ detector on the dataset depicted above, containing an ‘8’, a ‘6’ and a ‘9’. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the ‘9’ and repeats the ‘8’. On this dataset, the detector learns that a loop on top of the digit corresponds to an ‘8’. On the second dataset, we repeat the ‘9’ and omit the ‘6’. In this case, the detector learns that a loop on the bottom of the digit corresponds to an ‘8’. Each of these individual classification rules is brittle, but if we average their output then the detector is robust, achieving maximal confidence only when both loops of the ‘8’ are present.

Dropout

Pick a mini-batch

$$\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

Training:



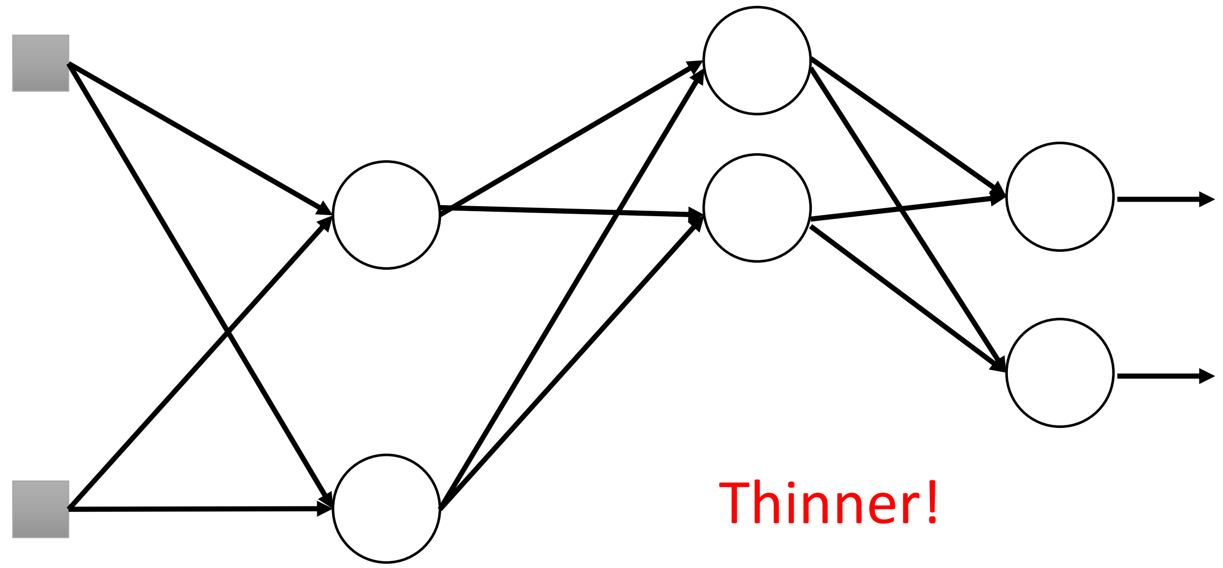
- **Each time before computing the gradients**
  - Each neuron has p% to dropout

Dropout

Pick a mini-batch

$$\theta^{t} \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

Training:



Thinner!

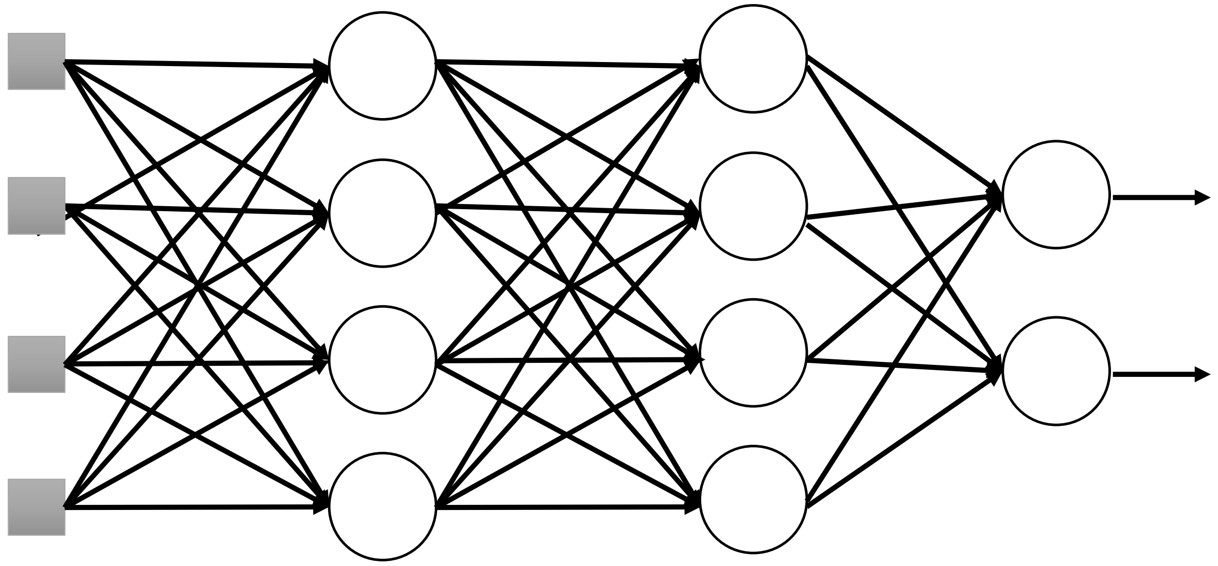
- **Each time before computing the gradients**
  - Each neuron has p% to dropout

    ⇨ **The structure of the network is changed.**
  - Using the new network for training

For each mini-batch, we resample the dropout neurons

Dropout

Testing:

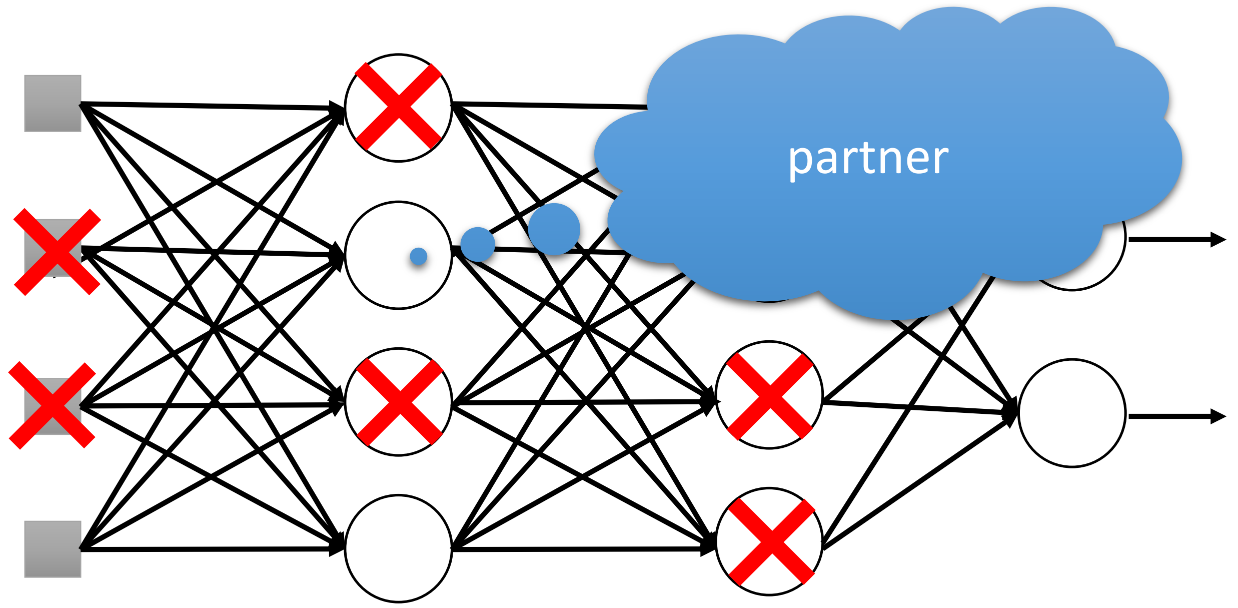


- **No dropout**

- If the dropout rate at training is p%, all the weights times (1-p)%

- Assume that the dropout rate is 50%. If a weight $\mathrm{w} = 1$ by training, set $w = 0.5$ for testing.

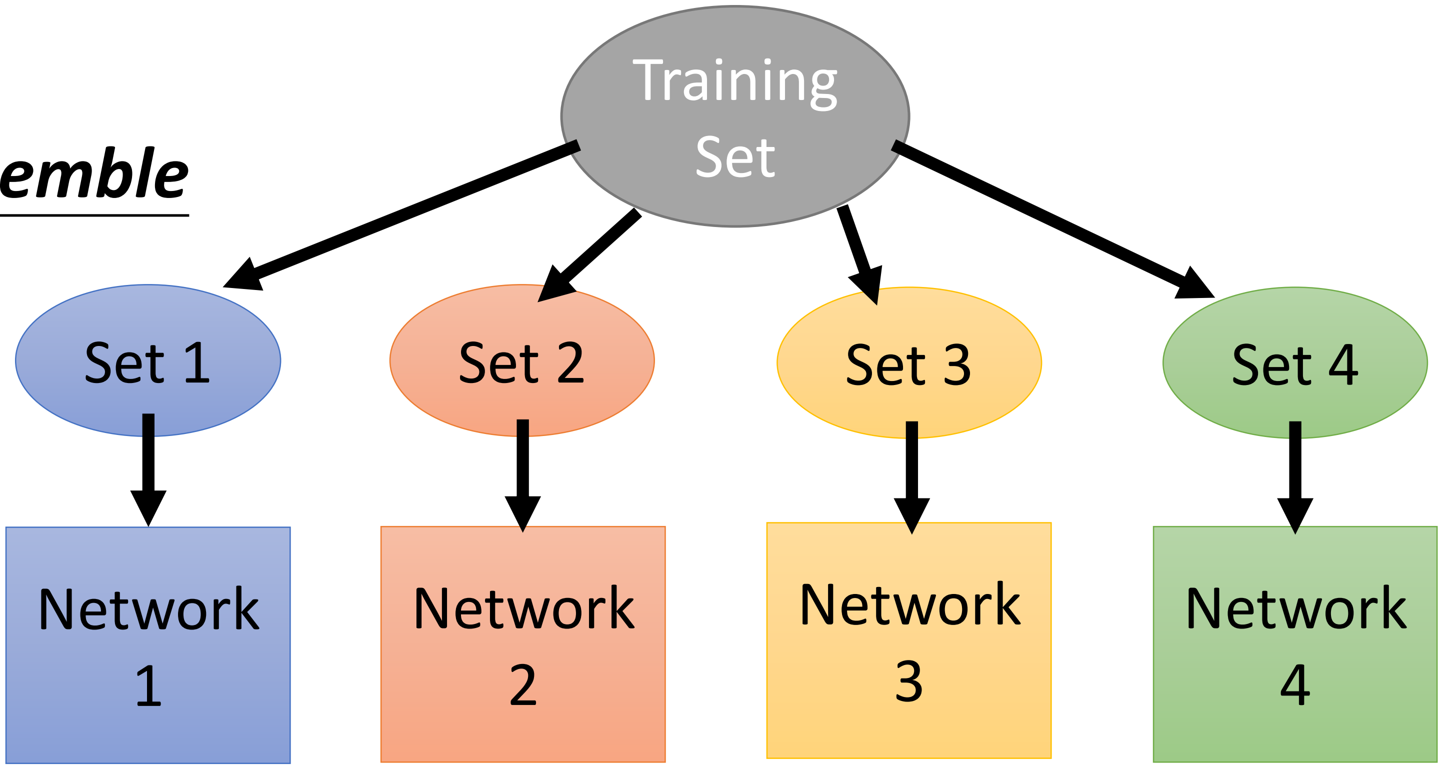
Dropout - Intuition



- When teams up, if everyone expect the partner will do the work, nothing will be done finally.
- However, if you know your partner will dropout, you will do better.
- When testing, no one dropout actually, so obtaining good results eventually.

Dropout is a kind of ensemble.

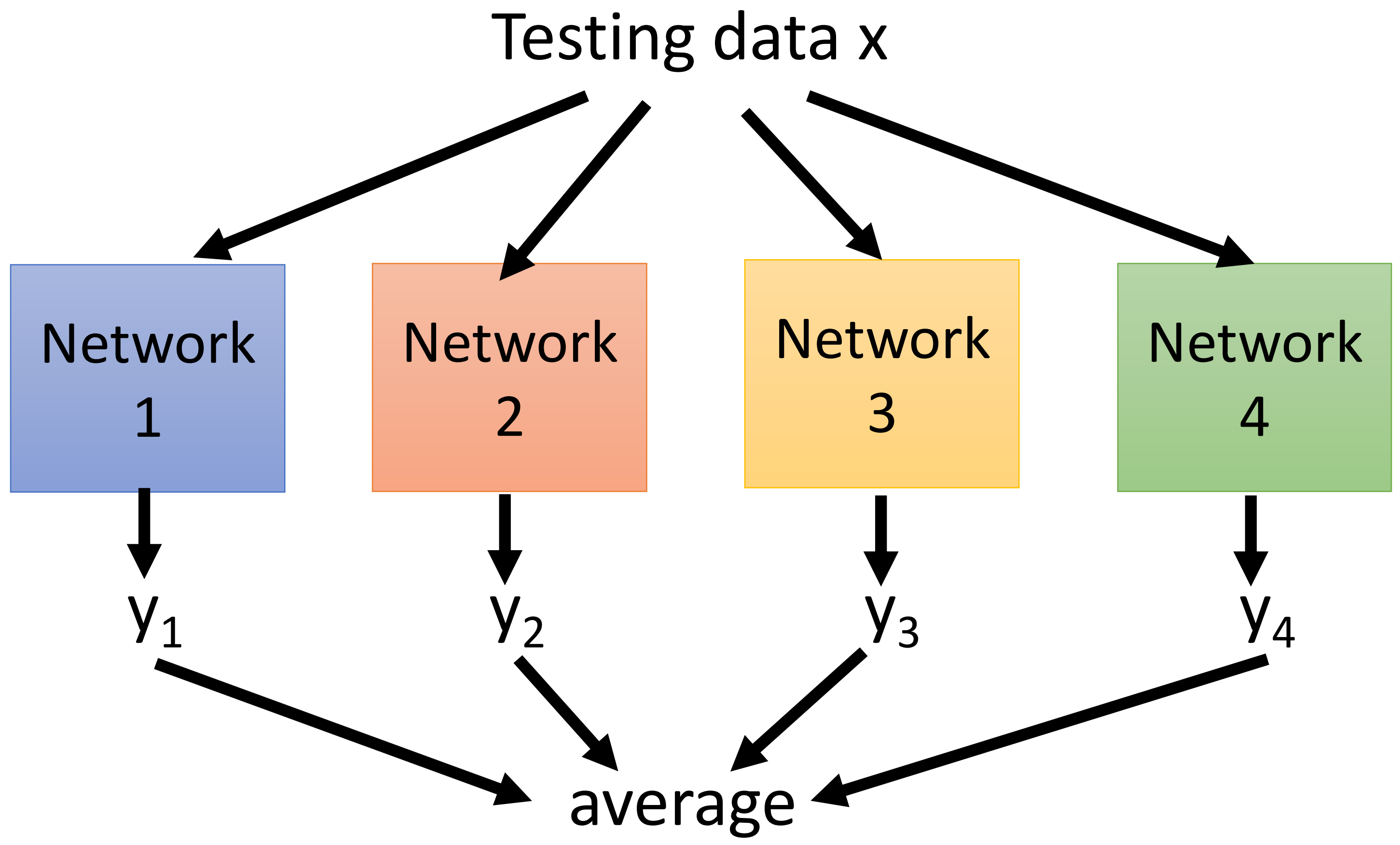
Ensemble



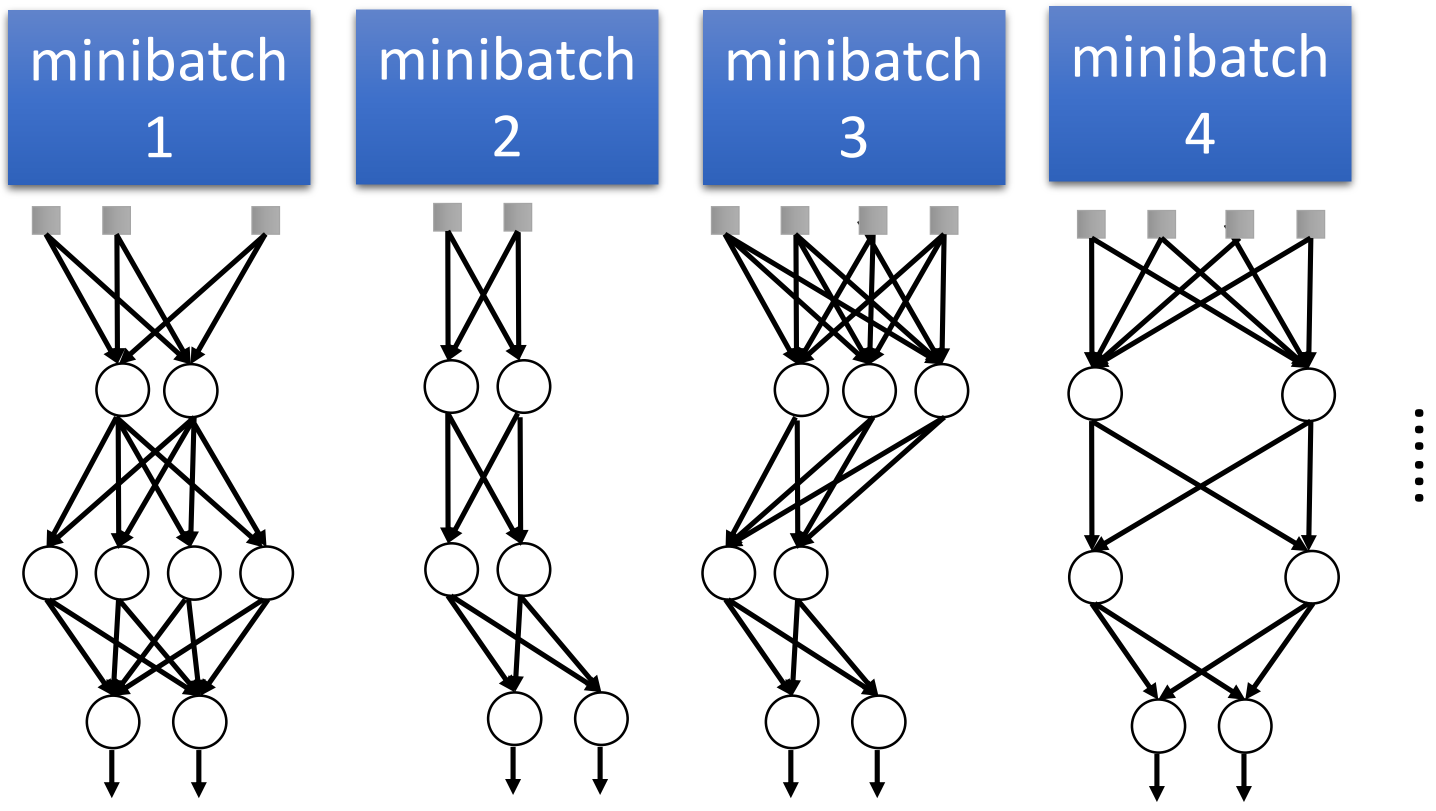
Train a bunch of networks with different structures

Dropout is a kind of ensemble.

Ensemble



Dropout is a kind of ensemble.



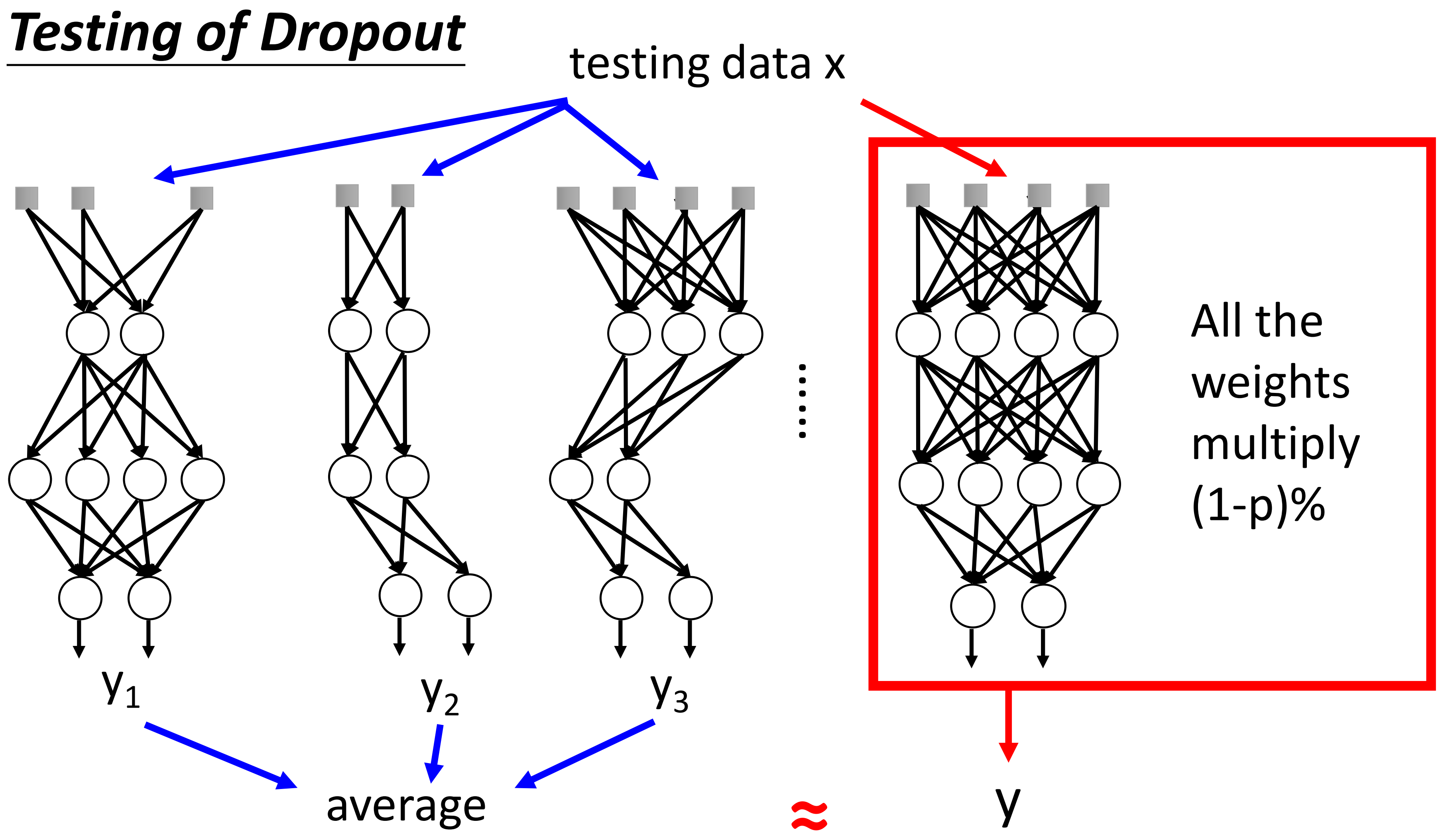
<u>Training of Dropout</u>

M neurons

2^M possible networks

- ➢Using one mini-batch to train one network
- ➢Some parameters in the network are shared

Dropout is a kind of ensemble.

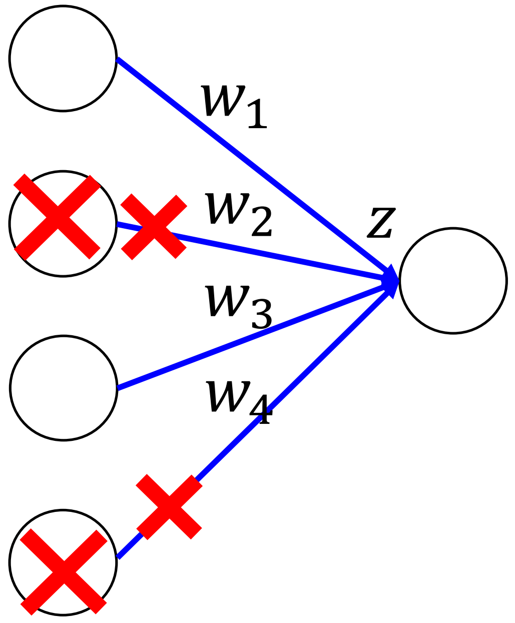


Dropout - Intuitive Reason

- Why the weights should multiply (1-p)% (dropout rate) when testing?

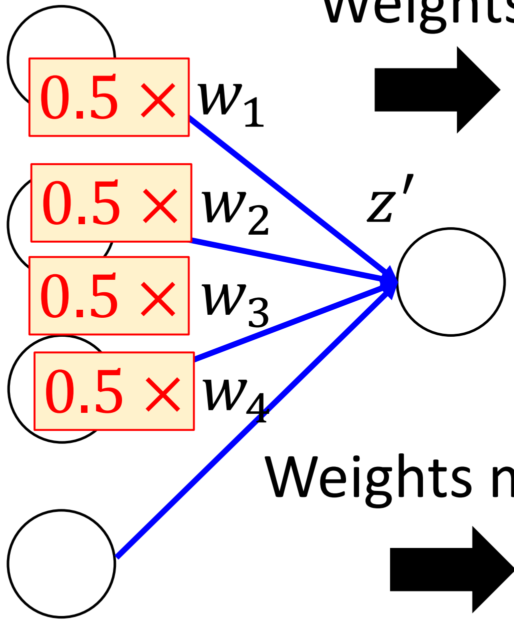
Training of Dropout

Assume dropout rate is 50%



Testing of Dropout

No dropout



Weights from training

$\Rightarrow z' \approx 2z$

Weights multiply (1-p)%

$\Rightarrow z' \approx z$

Usage of initializers

Initializations define the way to set the initial random weights of layers.

Available initializers

1. **Zeros**
2. **Ones**
3. **Constant**
4. **Random Normal**
5. **Random Uniform**
6. **Truncated Normal**
7. **Variance scaling**
8. **Orthogonal**
9. **Identity**
10. **Lecun_uniform**
11. **Glorat_normal**
12. **Glorat_uniform**
13. **He_normal**
14. **Lecun_normal**
15. **Custum initializaion**

Optimization

- Optimization is the most essential ingredient in the recipe of machine learning algorithms. It starts with defining some kind of loss function/cost function and ends with minimizing it using one or the other optimization routine.
- The choice of optimization algorithm can make a difference between getting a good accuracy in hours or days.

How Learning Differs from Pure Optimization

- Typically, the cost function with respect to the training set can be written as:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- where **L** is the per-example loss function, **f(x; θ)** is the predicted output when the input is **x**, **p'$_{\text{data}}$** is the empirical distribution. In the supervised learning case, **y** is the target output.

- The objective is to minimize the corresponding objective function

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Gradient descent

Gradient descent is a way to minimize an objective function J(θ)

  θ ∈ Rd : model parameters

  η: learning rate

  ∇θ J(θ): gradient of the objective function with regard to the parameters

Update equation: θ = θ − η · ∇θ J(θ)

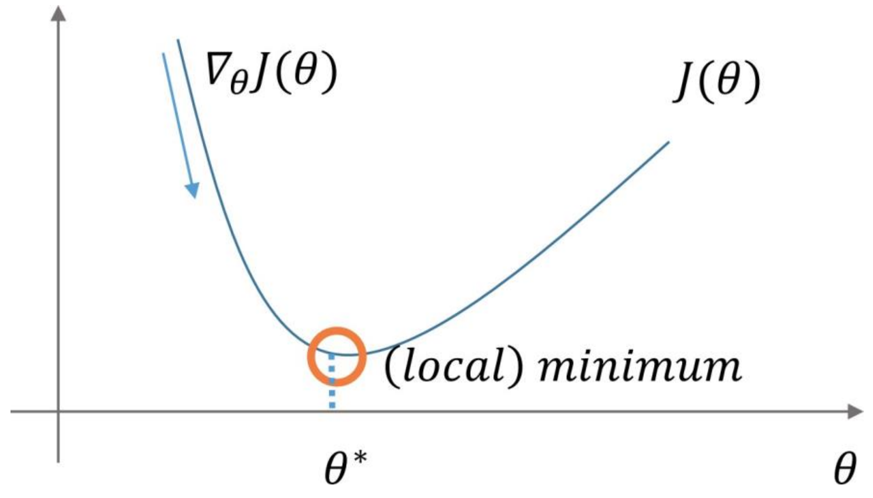


Figure: Optimization with gradient descent

Gradient descent variants

Batch gradient descent

Stochastic gradient descent

Mini-batch gradient descent

Difference: Amount of data used per update

Batch gradient descent

Computes gradient with the entire dataset.

Update equation: $\theta = \theta - \eta \cdot \nabla\theta \, J(\theta)$

```
 for i in range( nb_epochs ):  params_ grad = evaluate_ gradient (
          loss_function , data , params)
     params = params - learning_ rate * params_ grad
```

Listing : Code for batch gradient descent update

Batch gradient descent

Pros:

Guaranteed to converge to **global** minimum for **convex** error surfaces and to a **local** minimum for **non-convex** surfaces.

Cons:

Very slow.
Intractable for datasets that **do not fit in memory**.
No online learning.

Stochastic gradient descent

Computes update for **each** example $x^{(i)}y^{(i)}$.

Update equation: $\vartheta = \vartheta - \eta \cdot \nabla_{\vartheta} J(\vartheta; x^{(i)}; y^{(i)})$

```
for i in range( nb_epochs ):  np.random.
  shuffle(data)  for example in data:
    params_ grad =evaluate_ gradient (  loss_function ,
      example , params)
    params  = params  - learning_ rate  *params_ grad
```

Listing: Code for stochastic gradient descent update

Stochastic gradient descent

Pros

- **Much faster** than batch gradient descent.
- Allows **online learning**.

Cons

- **High variance** updates.

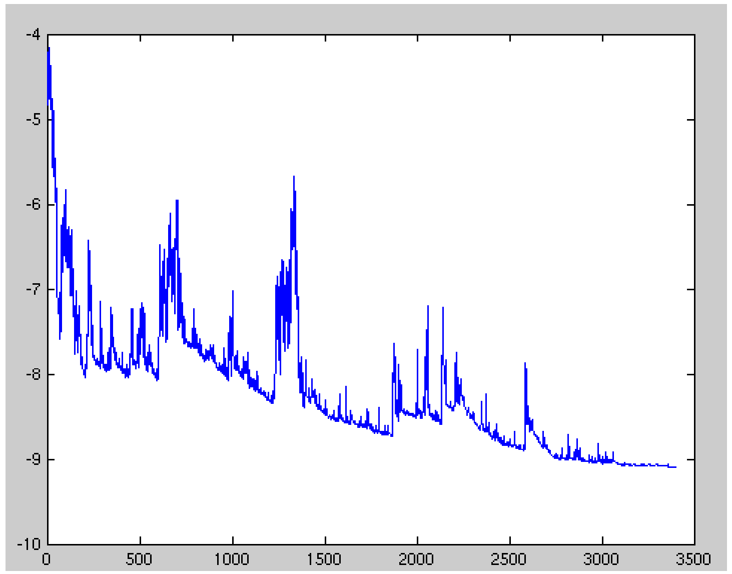


Figure: SGD fluctuation (Source: Wikipedia)

Batch gradient descent vs. SGD fluctuation

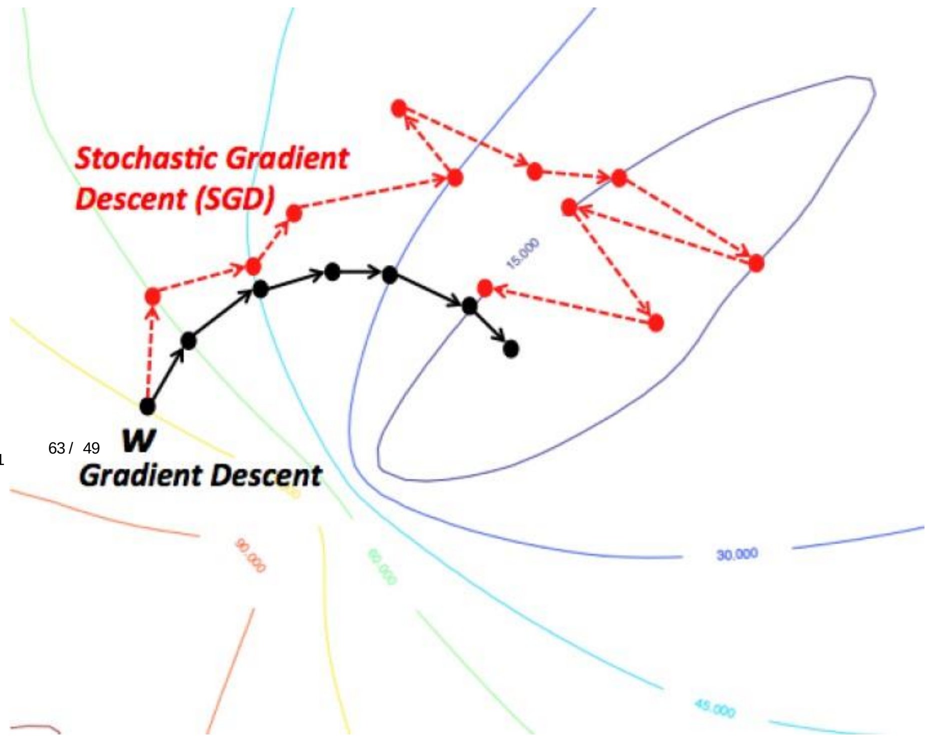


Figure: Batch gradient descent vs. SGD fluctuation (Source: [wikidocs.net)](wikidocs.net)

SGD shows same convergence behaviour as batch gradient descent if learning rate is **slowly decreased (annealed)** over time.

Mini-batch gradient descent

Performs update for every **mini-batch** of *n* examples. Update equation: $\vartheta = \vartheta - \eta \cdot \nabla_{\vartheta} J(\vartheta; x^{(i:i+n)}; y^{(i:i+n)})$

```
for i in range( nb_epochs ):
  np.random.shuffle(data)
  for batch in get_ batches( data , batch_size =50):
    params_grad  =evaluate_gradient(
      loss_function , batch , params)
    params =params - learning_ rate * params_ grad
```

Listing : Code for mini-batch gradient descent update

Mini-batch gradient descent

Pros

- **Reduces variance** of updates.
- Can exploit **matrix multiplication** primitives.

Cons

- **Mini-batch size** is a hyperparameter. Common sizes are 50-256.

Typically the algorithm of choice.

Usually referred to as SGD even when mini-batches are used.

Comparison of trade-offs of gradient descent variants

Method	**Accuracy**	**Update Speed**	**Memory Usage**	**Online Learning**
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Table: Comparison of trade-offs of gradient descent variants

Regression with GD

Regression

Y = f(X) + ε, where X = (x1, x2…xn)

Training: machine learns **f** from labeled training data

Test: machine predicts **Y** from unlabeled testing data

Note: **X** can be a **tensor** with an any number of dimensions. A 1D tensor is a vector (1 row, many columns), a 2D tensor is a matrix (many rows, many columns), and then you can have tensors with 3, 4, 5 or more dimensions (e.g. a 3D tensor with rows, columns, and depth).

Linear Regression (LR)

- Linear regression is a **parametric method**, which means it makes an assumption about the form of the function relating **X** and **Y**.
- Our model will be a function that predicts **ŷ** given a specific **x**:

$$\hat{y} = \beta_0 + \beta_1 * x + \varepsilon$$

- In this case, we make the explicit assumption that there is a **linear relationship** between **X** and **Y**—that is, for each one-unit increase in **X**, we see a constant increase (or decrease) in **Y**.

Linear Regression with GD

- Our goal is to learn the **model parameters** (**in this case, β0 and β1**) that minimize error in the model's predictions.
- To find the best parameters:
- Define a **cost function, error function** or **loss function**, that measures how inaccurate our model's predictions are.
- Find the parameters that **minimize loss**, i.e. make our model as accurate as possible.

Linear Regression with GD

A note on dimensionality:

- Our example is 2-dimensional for simplicity, but you’ll typically have more features (x’s) and coefficients (betas) in your model.
- **For example:** When adding more relevant variables to improve the accuracy of your model predictions. The same principles generalize to higher dimensions, though things get much harder to visualize beyond 3 dimensions.

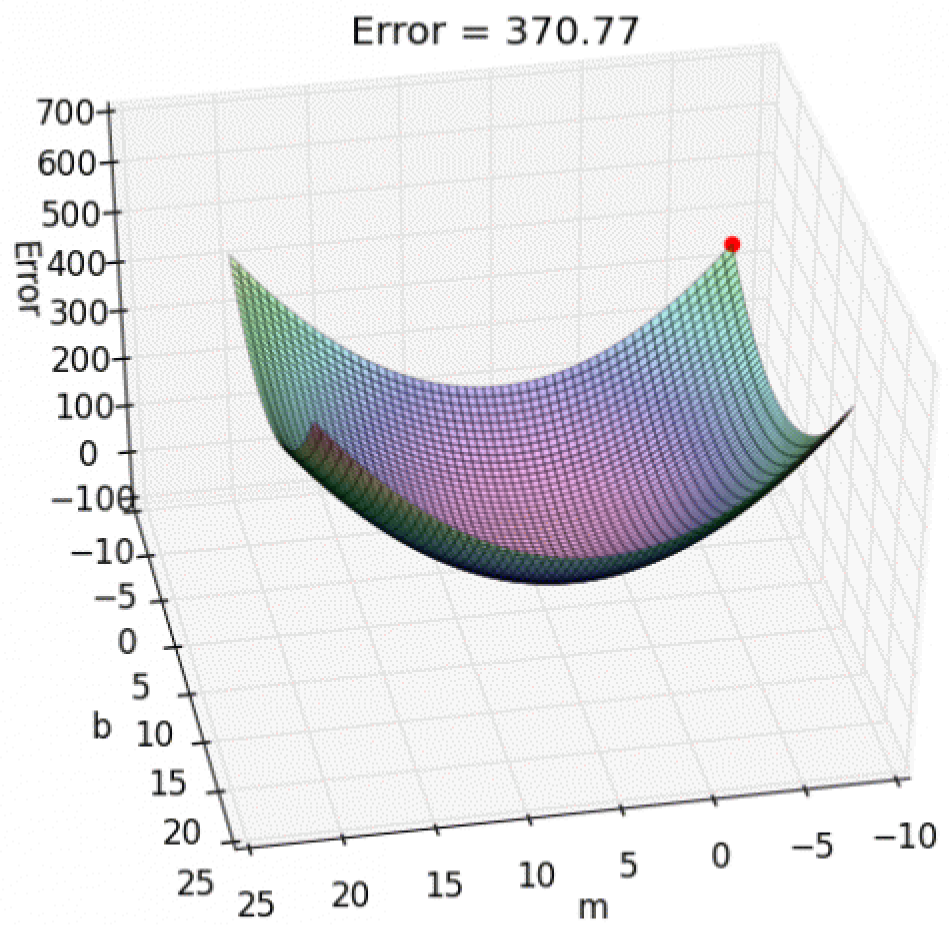
Linear Regression with GD

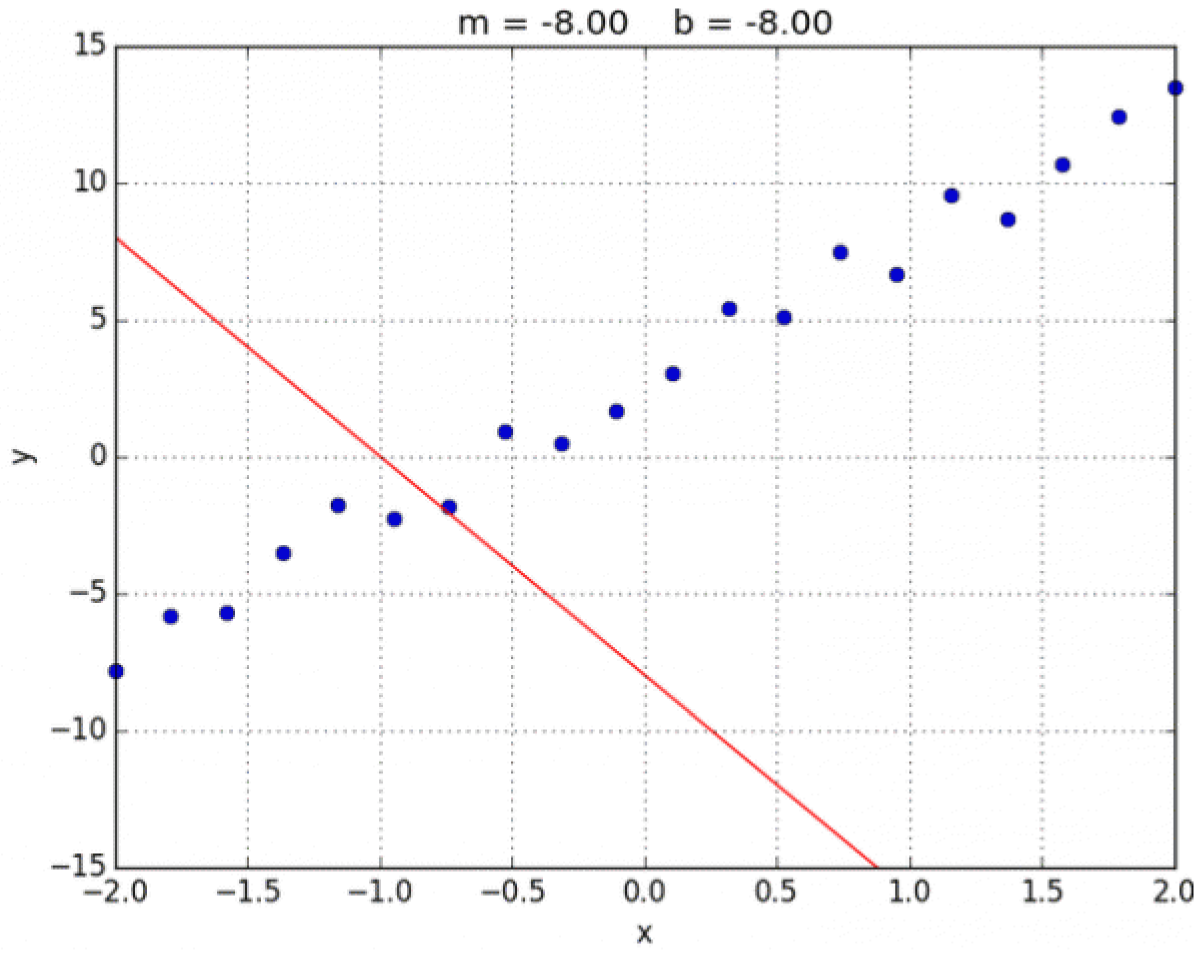
- Mathematically, we look at the difference between each real data point (y) and our model's prediction (ŷ).
- This is a measure of how well our data fits the line.

$$Cost = \frac{\sum_i ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n}$$

n = no. of observations.

Linear Regression with GD

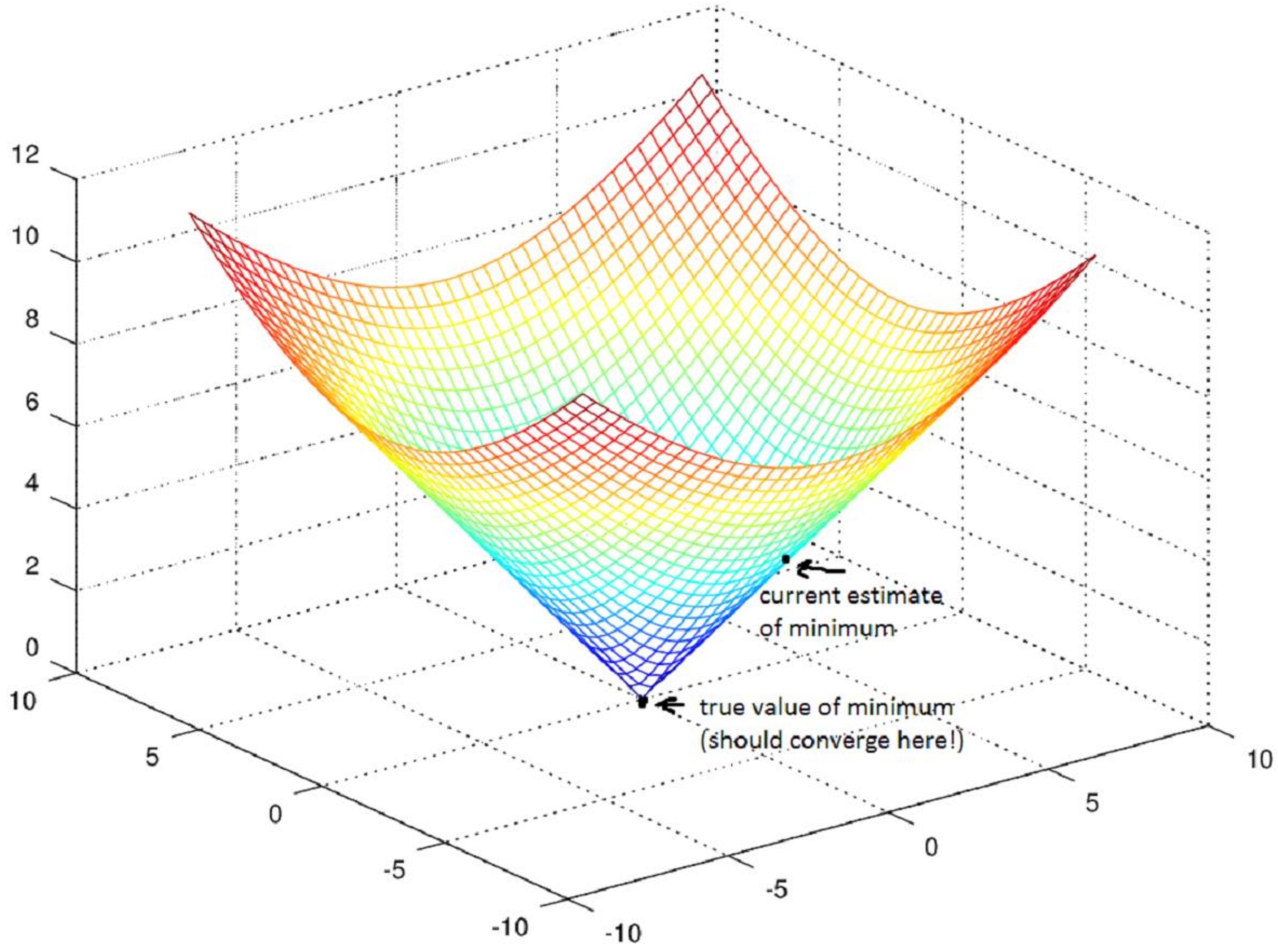




Source: [Github Alykhan Tejani]()

Linear Regression with GD

- We see that the **Cost function** is really a function of two variables: **β0 and β1.**
- All the rest of the variables are determined, since X, Y, and n are given during training.
- **We want to try to minimize this function.**



Linear Regression with GD

Idea: Choose **β0 and β1,** so that $\hat{y}$ is **close** to **y** for our training examples (x, y).

Remember: $\hat{y} = \beta_0 + \beta_1 * x + \varepsilon$

Goal: $\underset{\beta_0,\beta_1}{minimize}\ Cost(\beta_{0,}\beta_1)$

- $\hat{y}$: (for fixed **β0 and β1,** this is a function of **x**)

- **Cost :** (function of the parameter fixed **β0 and β1**)

Linear Regression with GD

For simplicity, lets assume $\hat{y}$ = **β1.x**

Then Our Goal becomes $\underset{\beta_1}{minimize} \quad Cost(\beta_1)$

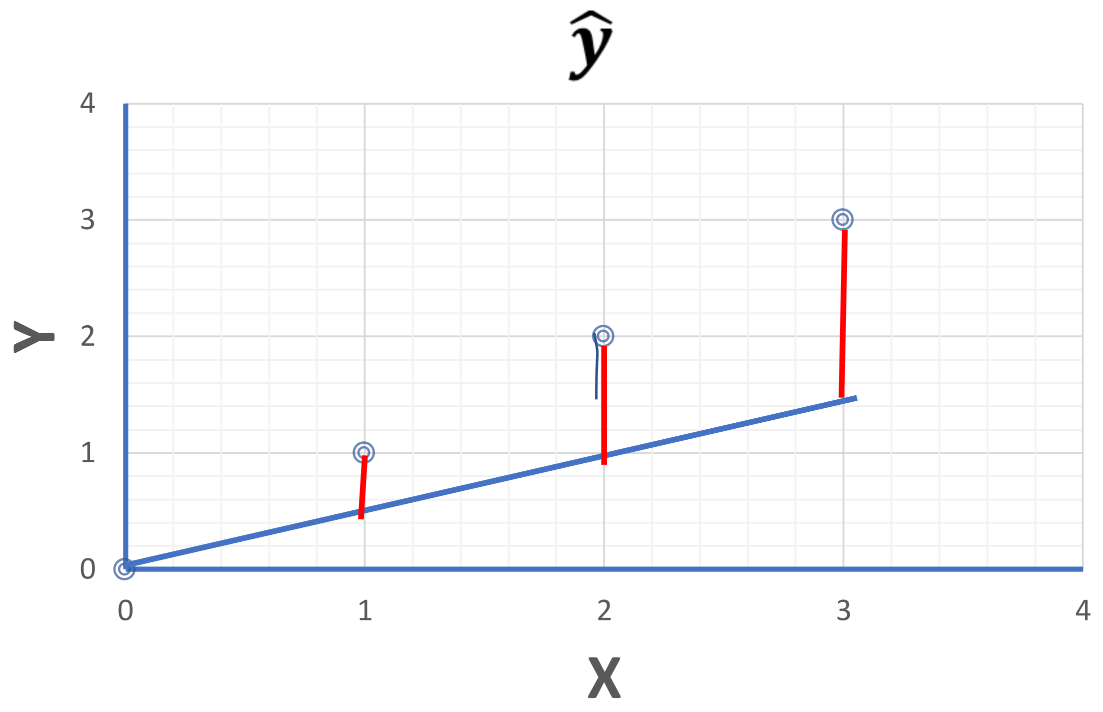
Now, if **β1=1,** then lets compute what will be the value of **Cost function?**

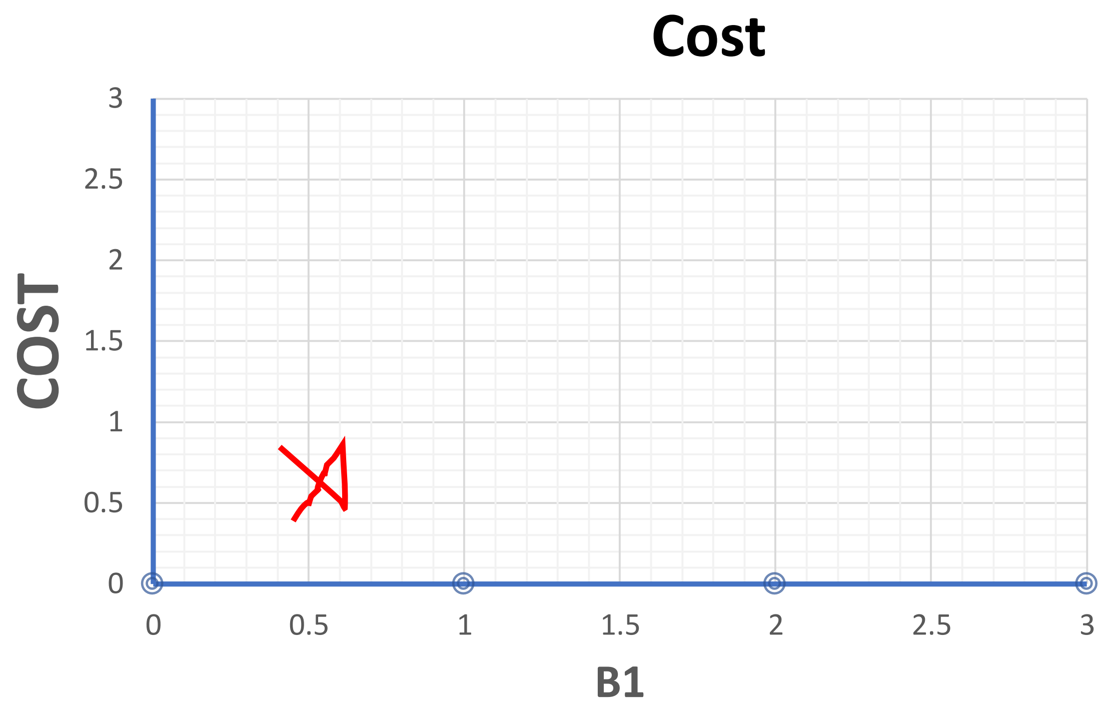
$$Cost(\beta_1) = \frac{\sum_i^n (\hat{y} - y_i)^2}{2 * n}$$

Linear Regression with GD

x	1	2	3
y	1	2	3

x	1	2	3
$\hat{y}$	0.5	1	1.5



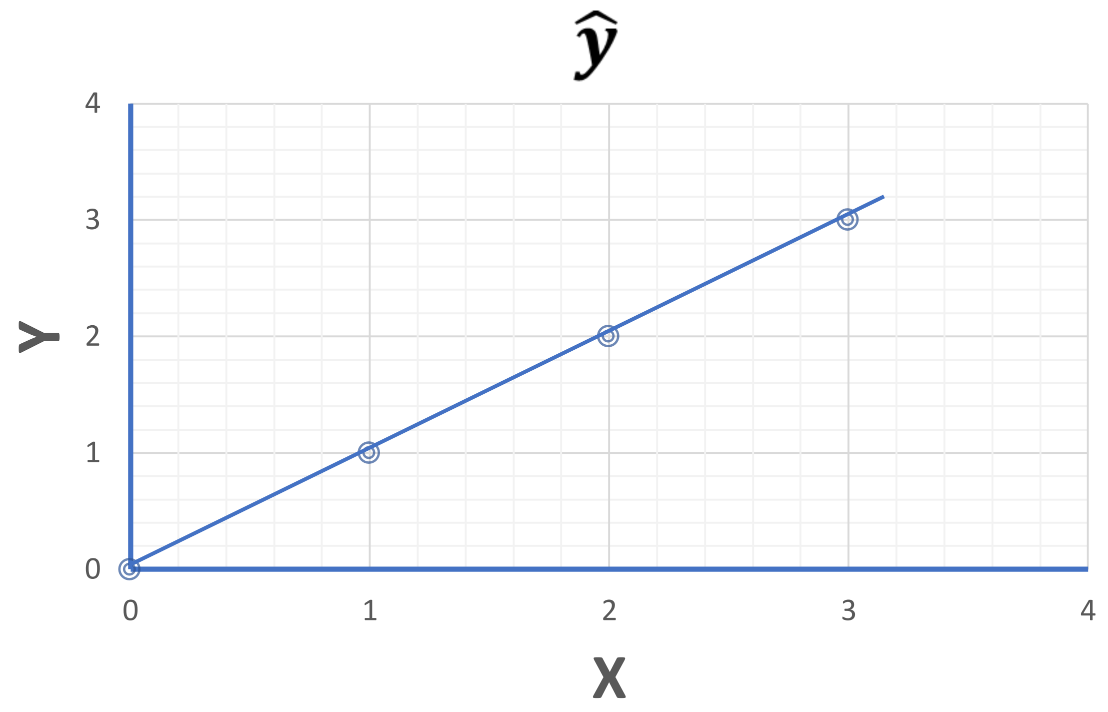


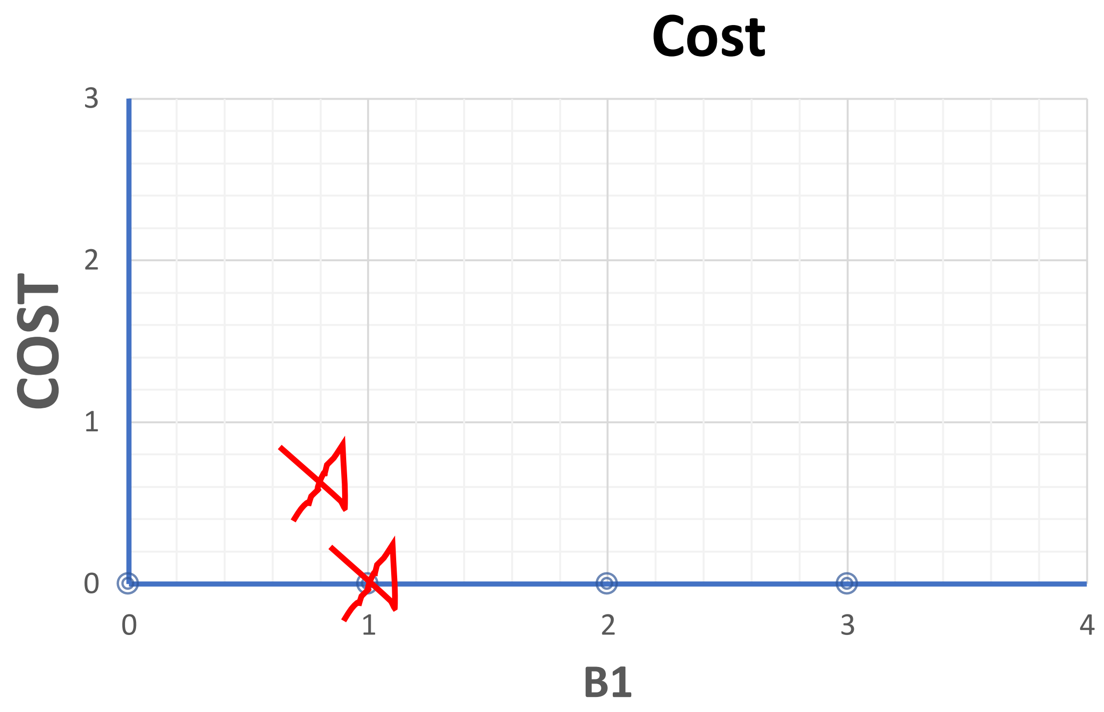
if **β1=0.5,** then **Cost(β1=0.5) = 0.58**

Linear Regression with GD

x	1	2	3
y	1	2	3

x	1	2	3
$\hat{y}$	1	2	3



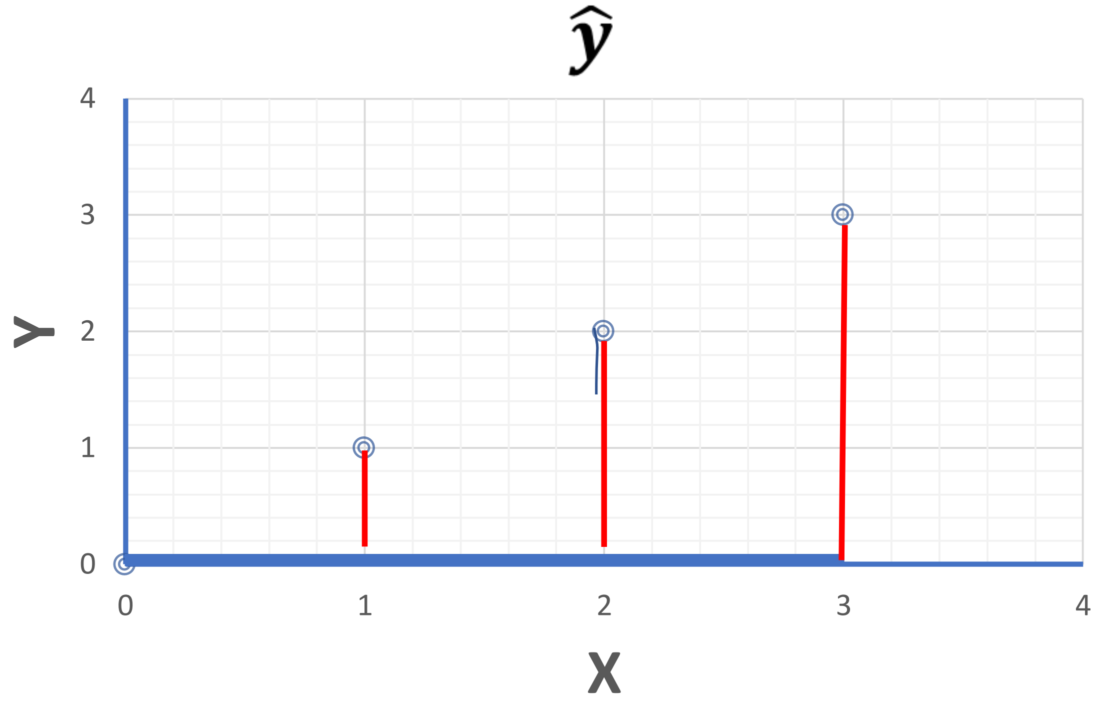


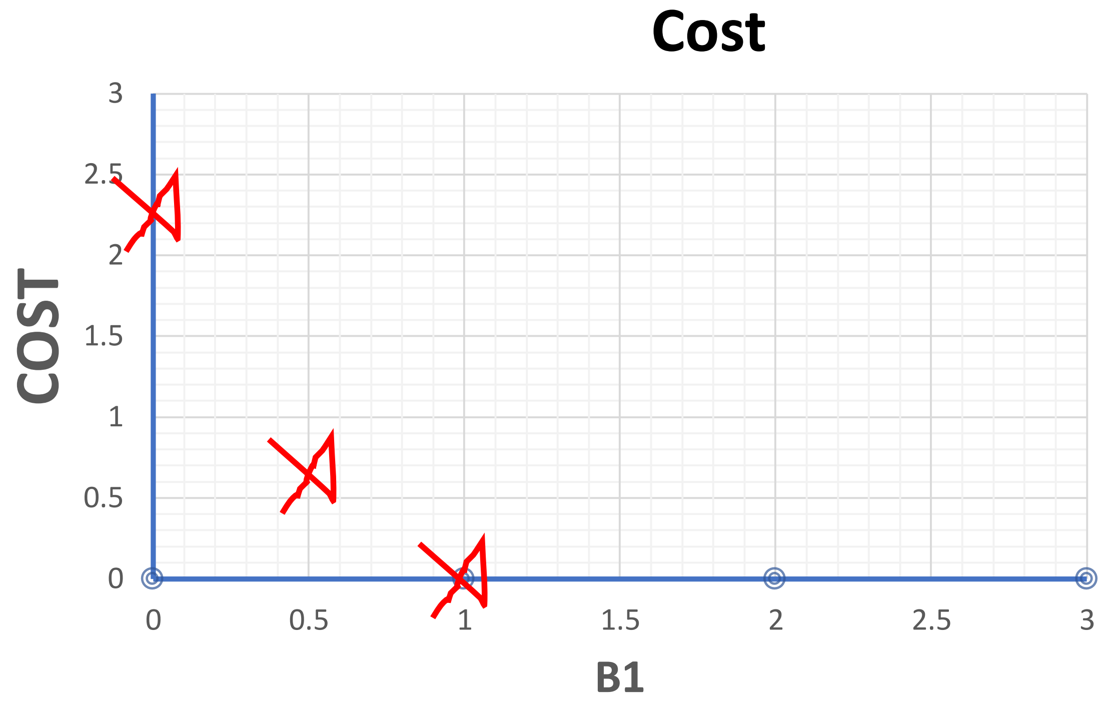
if **β1=1,** then **Cost(β1=1) = 0**

Linear Regression with GD

x	1	2	3
y	1	2	3

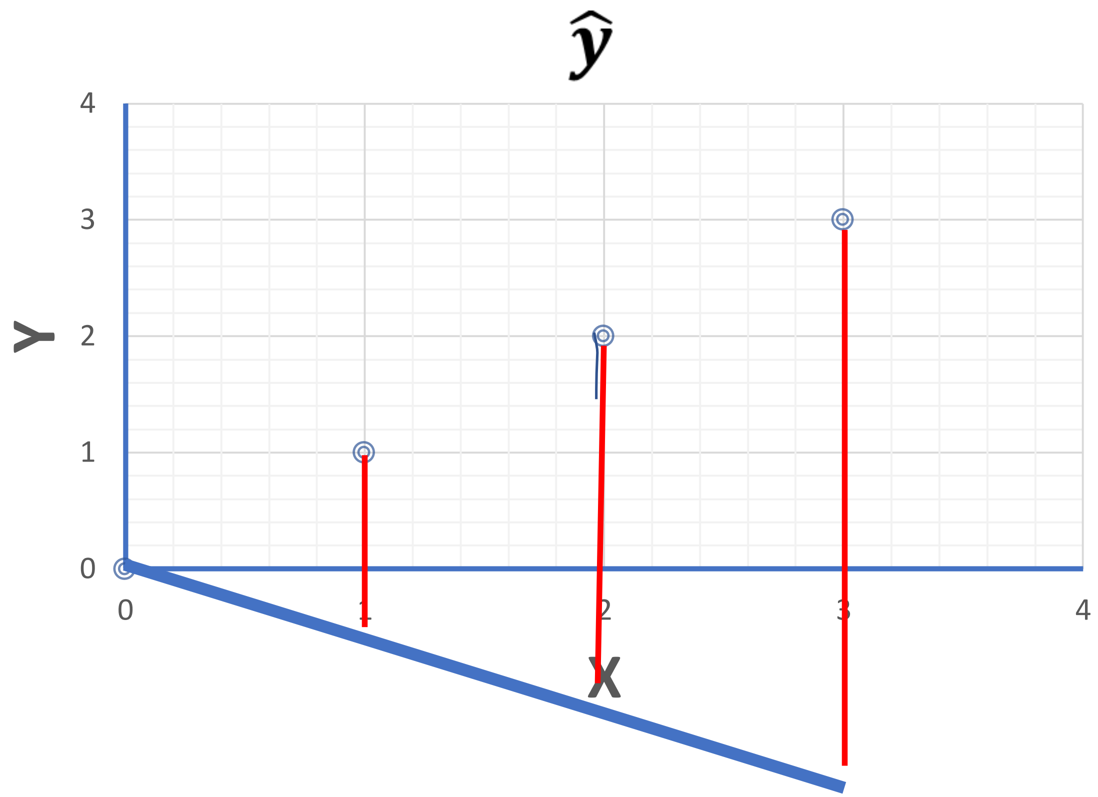
x	1	2	3
$\hat{y}$	0	0	0

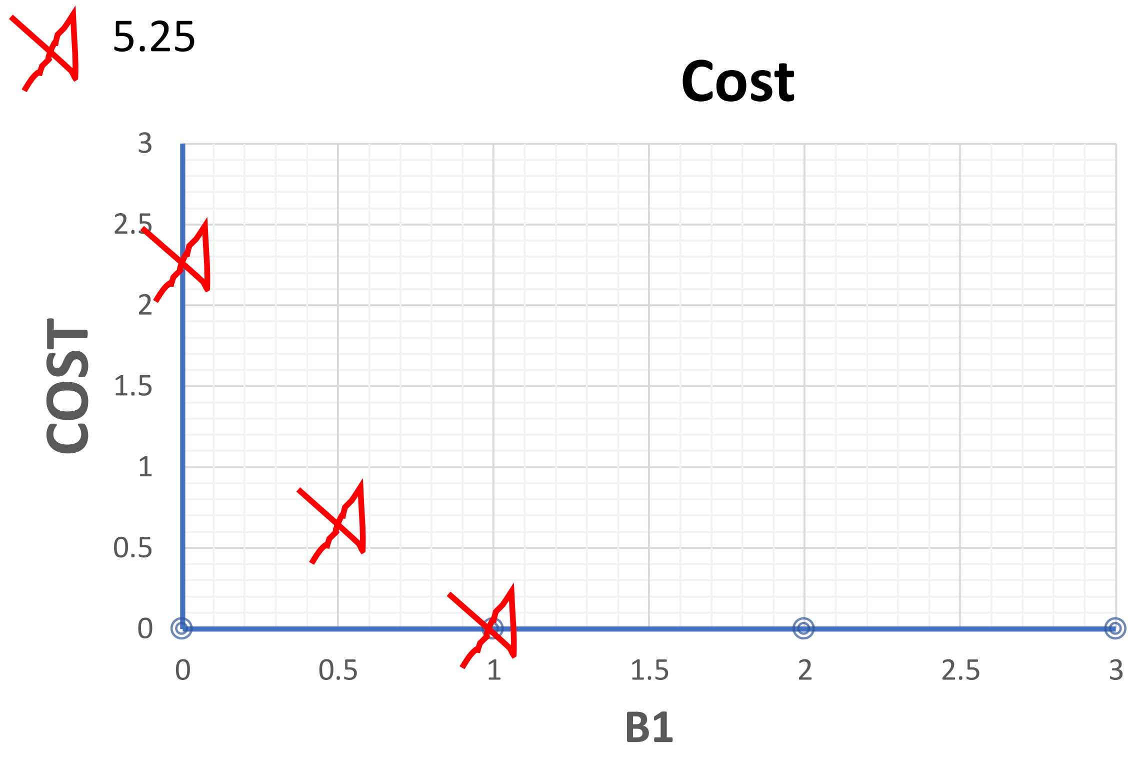




if **β1=0,** then **Cost(β1=0) = 2.3**

Linear Regression with GD

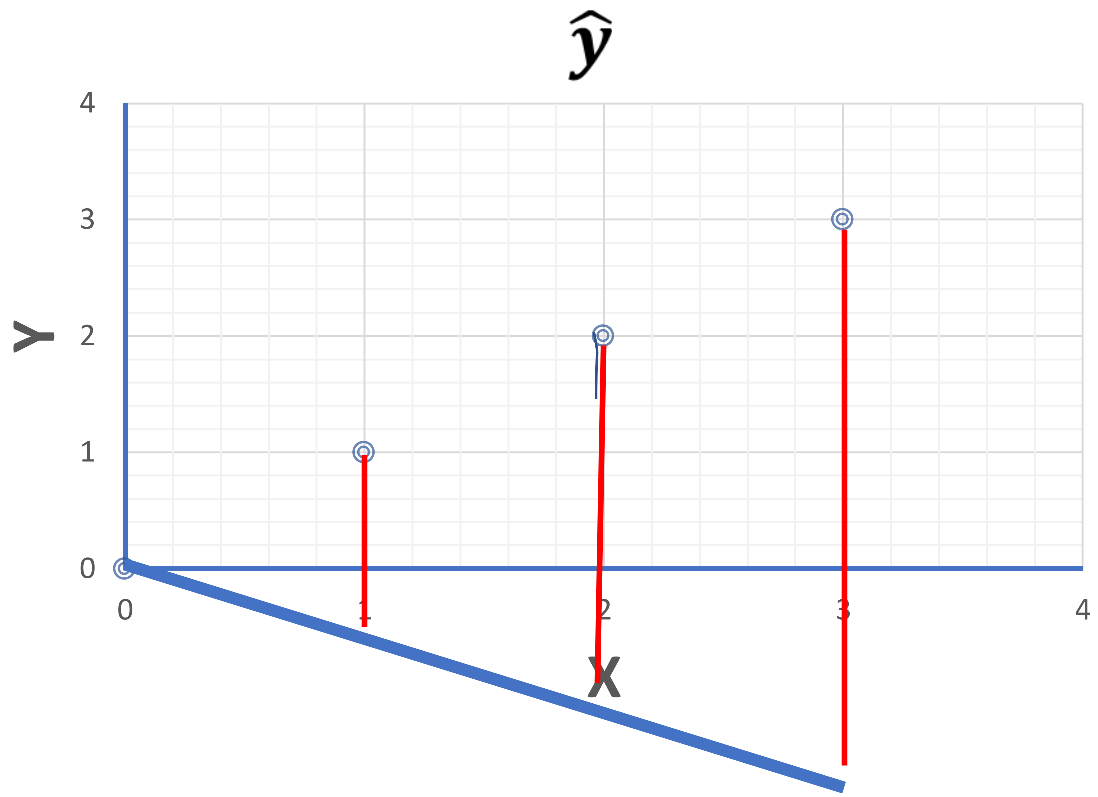


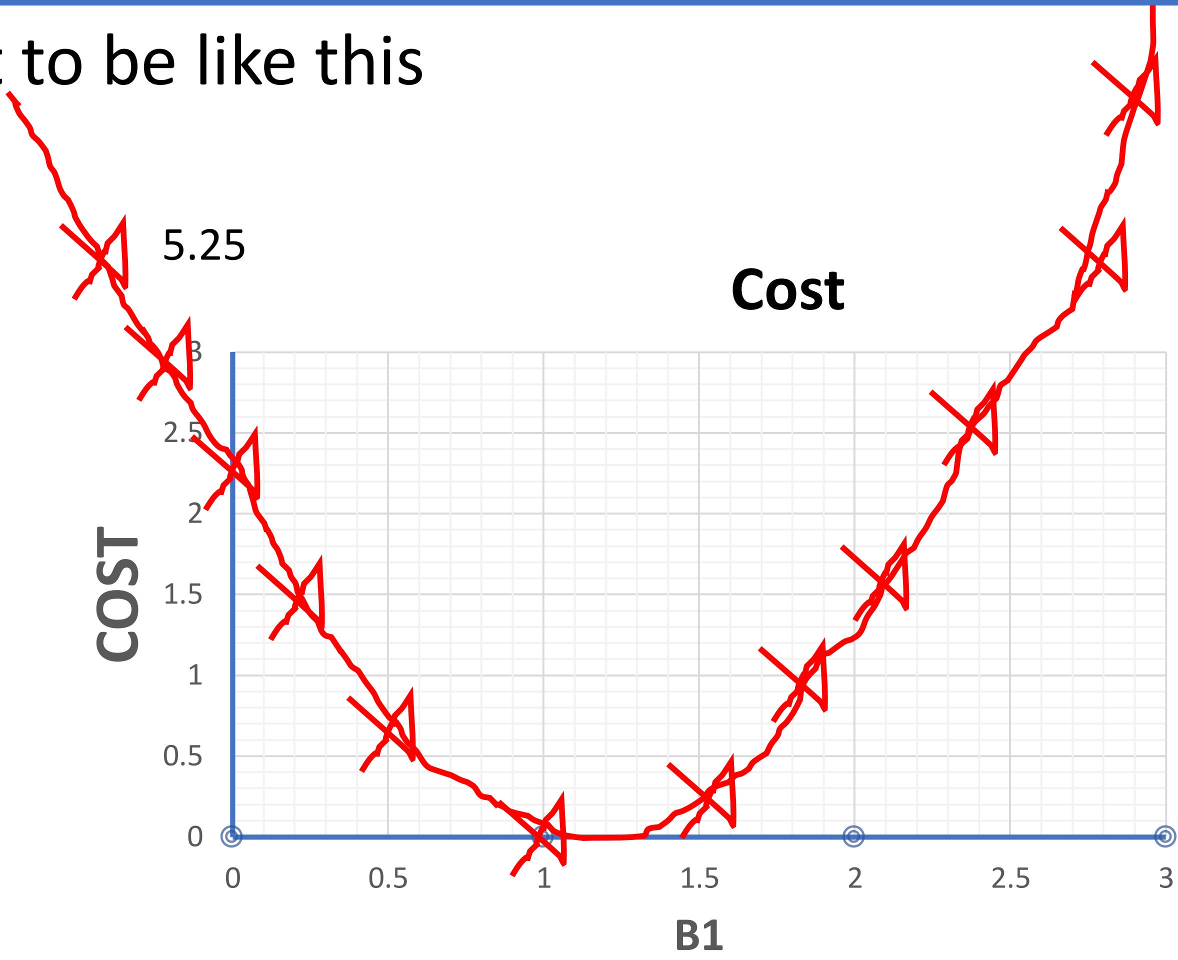


if **β1= - 0.5,** then **Cost(β1=-0.5) = 5.25**

Linear Regression with GD

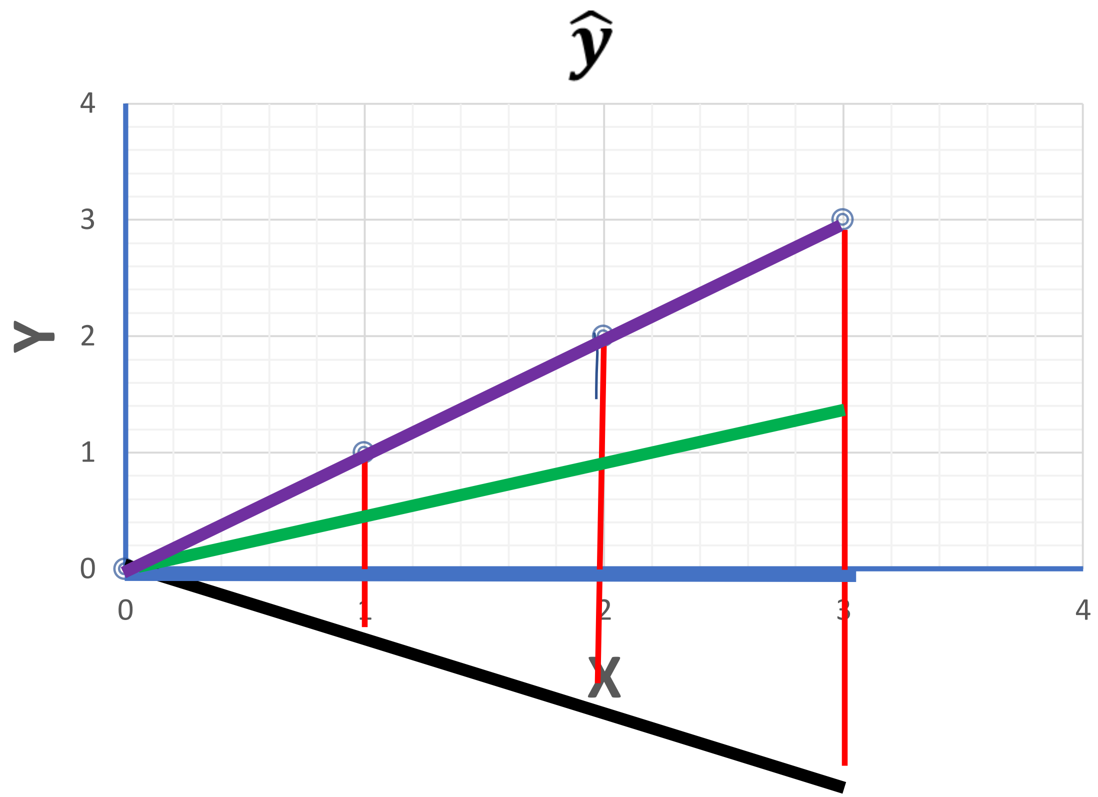
For Different Values of **β1,** it turns out to be like this

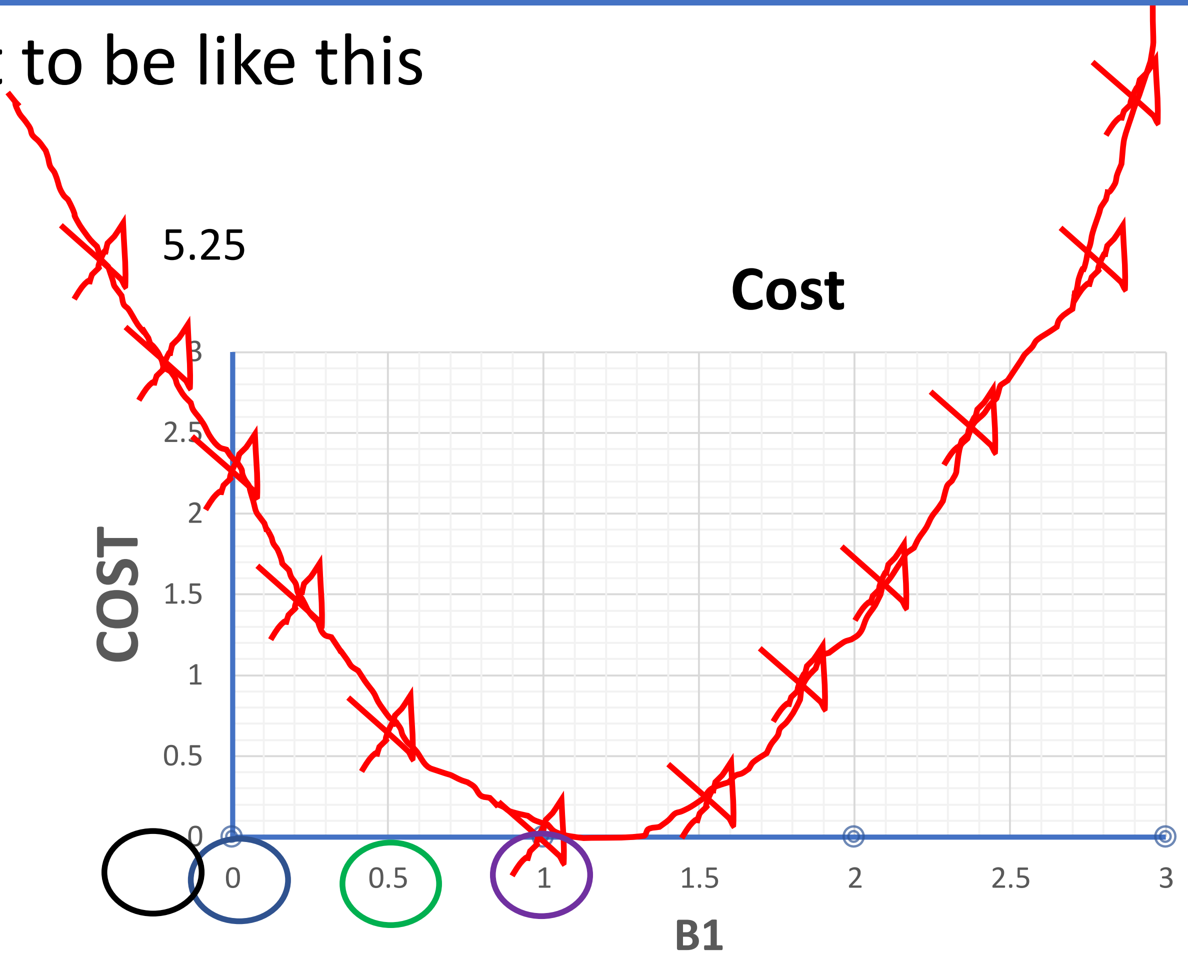




Linear Regression with GD

For Different Values of **β1,** it turns out to be like this





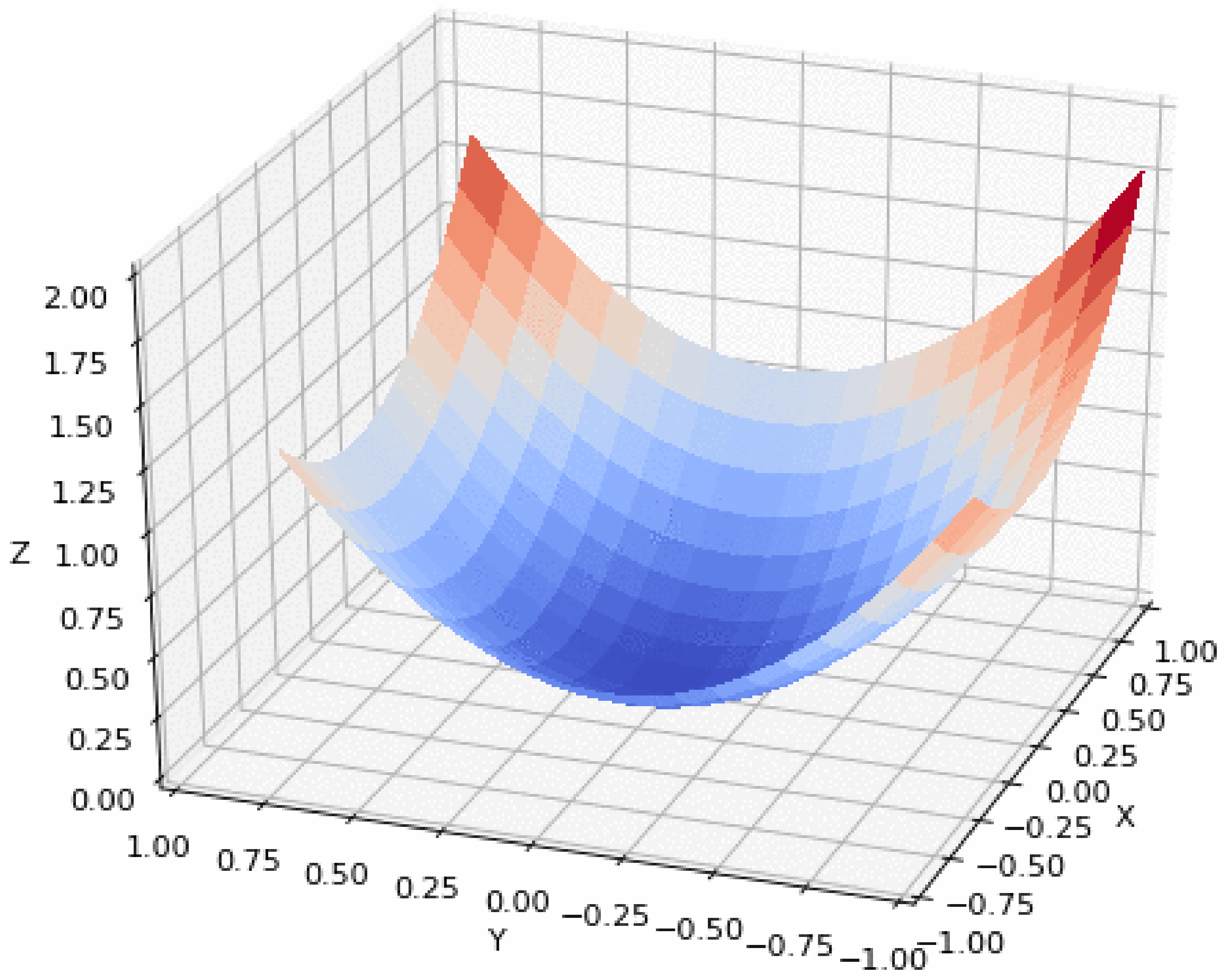
if **β1= - 0.5,** then **Cost(β1=-0.5) = 5.25**

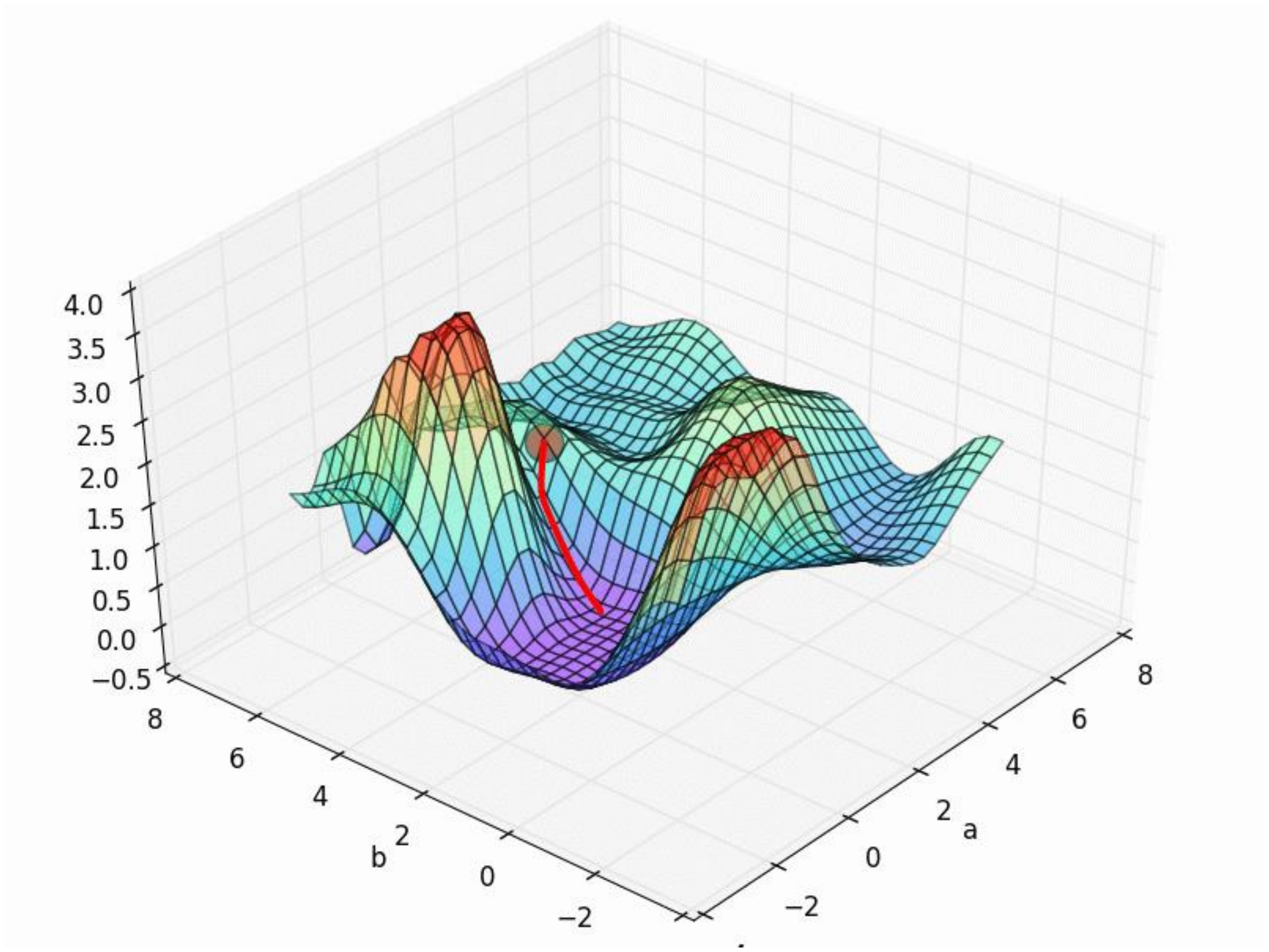
Linear Regression with GD

- **Remember Our Objective/Goal? :** **Did we achieve that? - yes**

$$\underset{\beta_1}{minimize} \quad Cost(\beta_1)$$

Linear Regression with GD




4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
8
6
4
b 2
0
−2
−2
0
2 a
4
6
8

Linear Regression with GD

- Repeat until convergence

$$\beta_j = \beta_j - \alpha \frac{\partial}{\partial \beta_j} Cost(\beta_0, \beta_1)$$

$\alpha = Learning\ Rate$

- Simultaneous Update

$$\beta_0 = \beta_0 - \alpha \frac{\partial}{\partial \beta_0} Cost(\beta_0, \beta_1)$$

$$\beta_1 = \beta_1 - \alpha \frac{\partial}{\partial \beta_1} Cost(\beta_0, \beta_1)$$

Linear Regression with GD

$$Cost = \frac{\sum_i ((\beta_1 x_i + \beta_0) - y_i)^2}{2*n}$$

$$\hat{y} = \beta_0 + \beta_1 * x + \varepsilon$$

Basically we need to find out

$$\frac{\partial}{\partial \beta_j} Cost(\beta_0, \beta_1)$$

$$j = 0: \frac{\partial}{\partial \beta_0} Cost(\beta_0, \beta_1) = \frac{\partial}{\partial \beta_0}\left(\frac{\sum_i^n (\beta_1 x_i + \beta_0 - y_i)^2}{2*n}\right) = \frac{\sum_i^n (\beta_1 x_i + \beta_0 - y_i)}{n}$$

$$j = 1: \frac{\partial}{\partial \beta_1} Cost(\beta_0, \beta_1) = \frac{\partial}{\partial \beta_1}\left(\frac{\sum_i^n (\beta_1 x_i + \beta_0 - y_i)^2}{2*n}\right) = \frac{\sum_i^n (\beta_1 x_i + \beta_0 - y_i) * x_i}{n}$$

Gradient Descent for Linear Regression

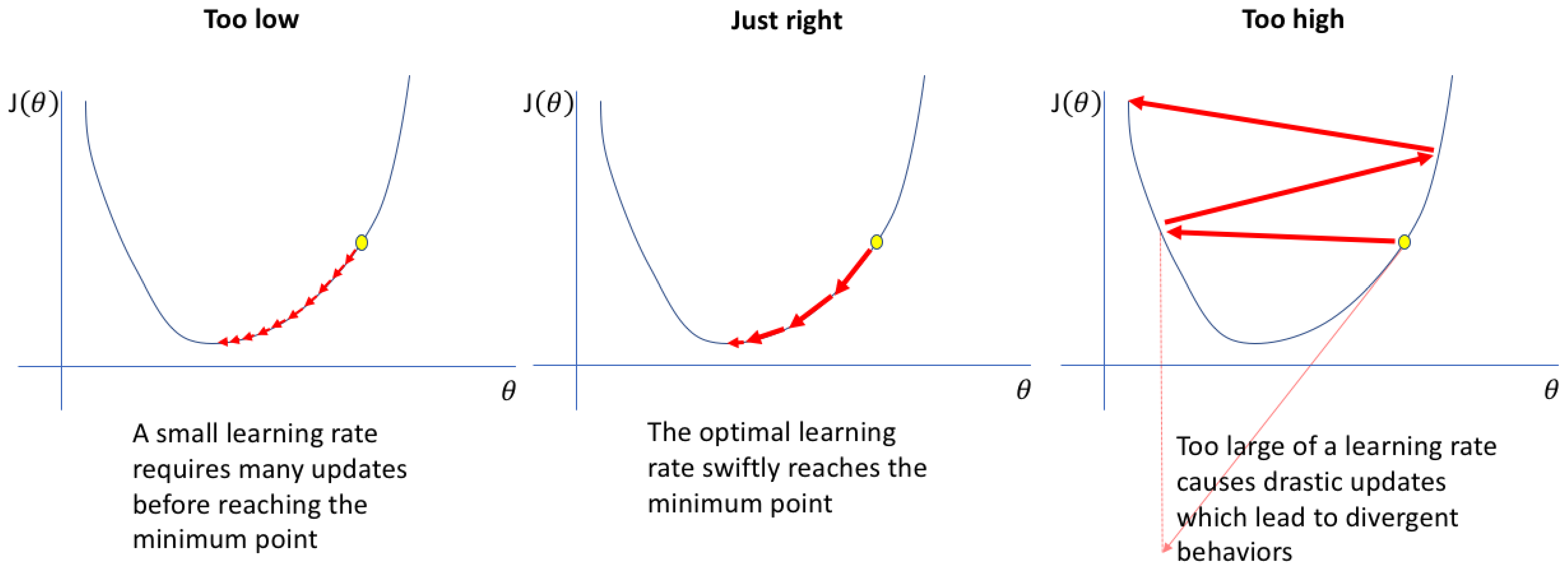
- **Gradient Descent Algorithm:**

Repeat until convergence:

$$\begin{cases} \beta_0 = \beta_0 - \alpha \dfrac{\partial}{\partial \beta_0} Cost(\beta_0, \beta_1) \\ \beta_1 = \beta_1 - \alpha \dfrac{\partial}{\partial \beta_1} Cost(\beta_0, \beta_1) \end{cases}$$

Update β0 and β1 simultaneously.

Optimization Algorithms: Stochastic Gradient Descent



A small learning rate requires many updates before reaching the minimum point

The optimal learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Learning rate selection. Source: https://www.jeremyjordan.me/nn-learning-rate/

Some other variants of SGD: Momentum

- SGD has trouble navigating **ravines.**
- Momentum [Qian, 1999] helps SGD **accelerate.**
- Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t . Momentum term γ is usually set to 0.9.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

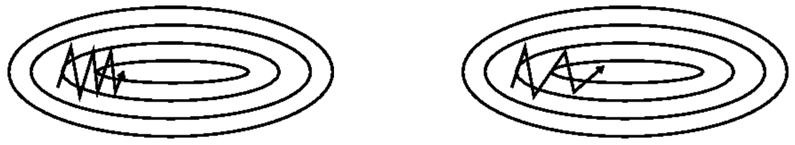

(a) SGD without momentum (b) SGD with momentum

Figure: Source: Genevieve B. Orr

Momentum

Reduces updates for dimensions whose gradients **change directions**.

Increases updates for dimensions whose gradients **point in the same directions**.

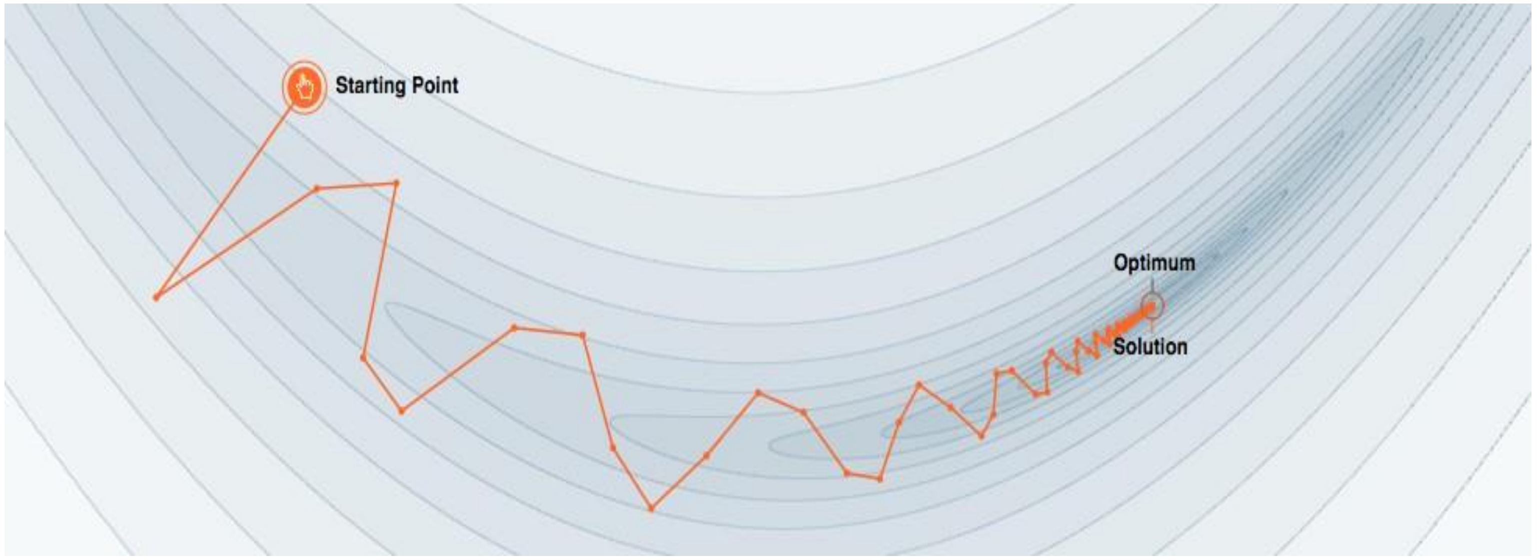


Figure: Optimization with momentum (Source: distill.pub)

Nesterov accelerated gradient

- Momentum **blindly accelerates** down slopes: First computes gradient, then makes a big jump.
- Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient
- $\vartheta - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

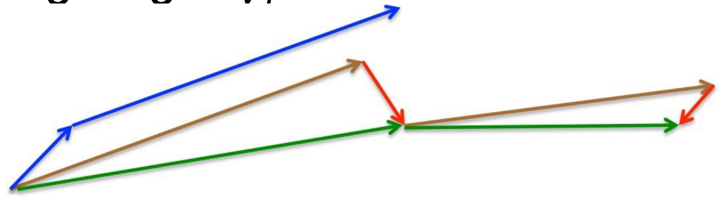


Figure: Nesterov update (Source: G. Hinton's lecture 6c)

Adagrad

Previous methods: **Same learning rate** η for all parameters ϑ. Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).

$$\text{SGD update: } \theta_{t+1} = \theta_t - \eta \cdot g_t$$

$$g_t = \nabla_{\vartheta_t} J(\vartheta_t)$$

Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.

Adagrad update:

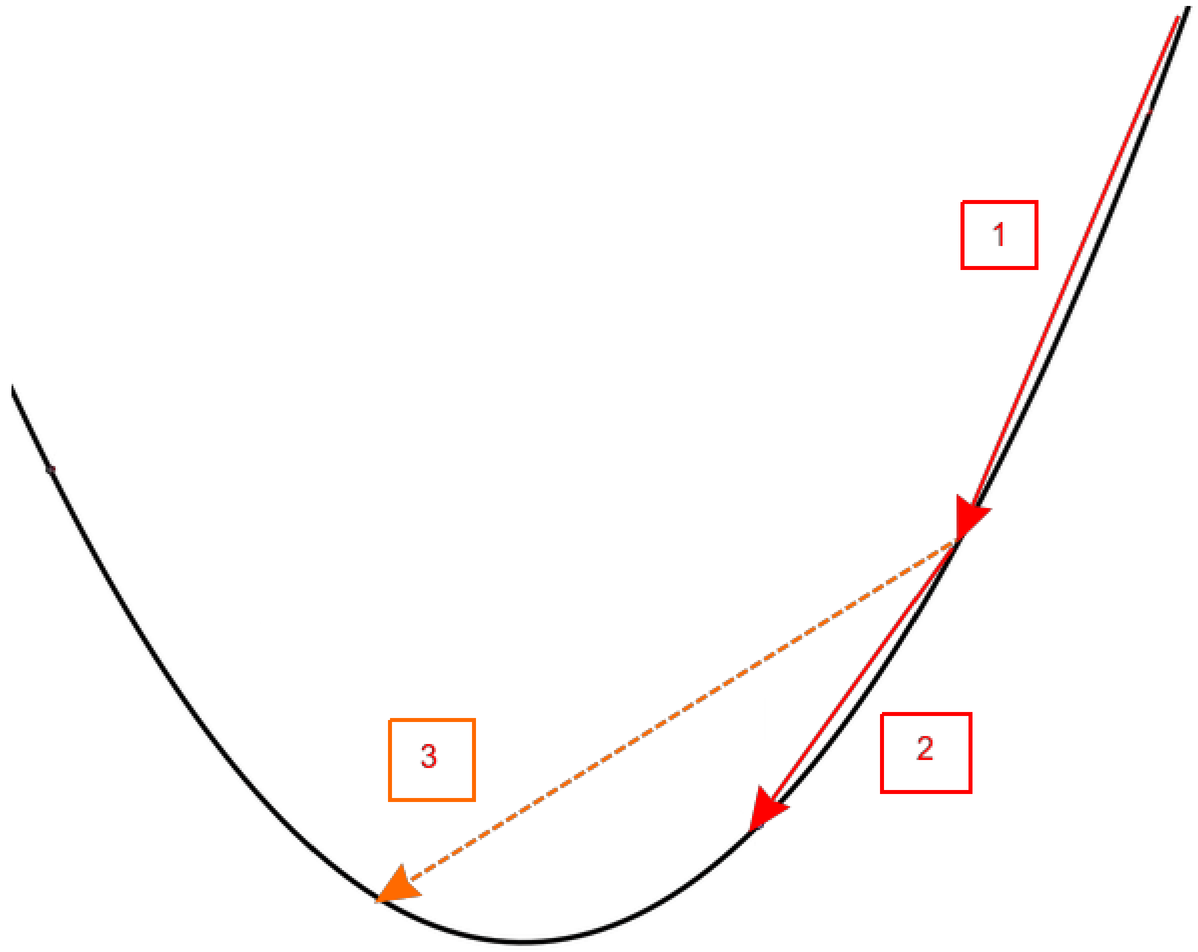
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + s}} \circledS g_t$$

$G_t \in R^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. ϑ_i up to time step t

s: smoothing term to avoid division by zero

$\circledS$: element-wise multiplication

AdaGrad



This figure illustrates the need to reduce the learning rate if gradient is large in case of a single parameter. 1) One step of gradient descent representing a large gradient value. 2) Result of reducing the learning rate — moves towards the minima 3) Scenario if the learning rate was not reduced — it would have jumped over the minima.

Pros

- Well-suited for dealing with **sparse data**.
- Significantly **improves robustness** of SGD.
- Lesser need to manually tune learning rate.

Cons

- **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

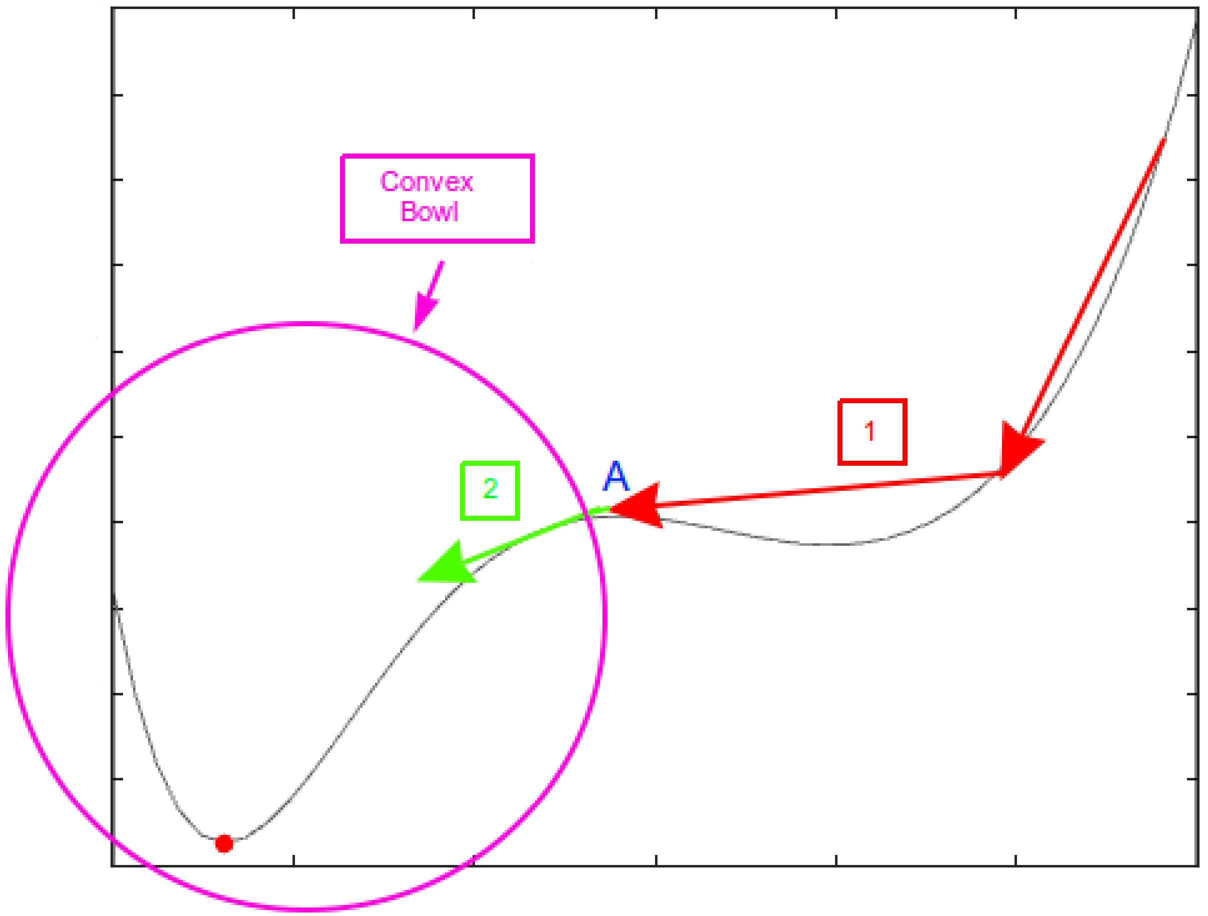
Optimization Algorithms: RMSProp

- **RMSProp** addresses the problem caused by accumulated gradients in AdaGrad.
- It modifies the gradient accumulation step to an **exponentially weighted moving average** in order to **discard history from the extreme past**. The RMSProp update is given by:

$$\text{Compute gradient: } \boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}).$$
$$\text{Accumulate squared gradient: } \boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}.$$
$$\text{Compute parameter update: } \Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}. \quad \left(\frac{1}{\sqrt{\delta + \boldsymbol{r}}} \text{ applied element-wise}\right)$$

- ρ is the weighing used for exponential averaging. As more updates are made, the contribution of past gradient values are reduced since $\rho < 1$ and $\rho > \rho^2 > \rho^3 \ldots$

Optimization Algorithms: RMSProp

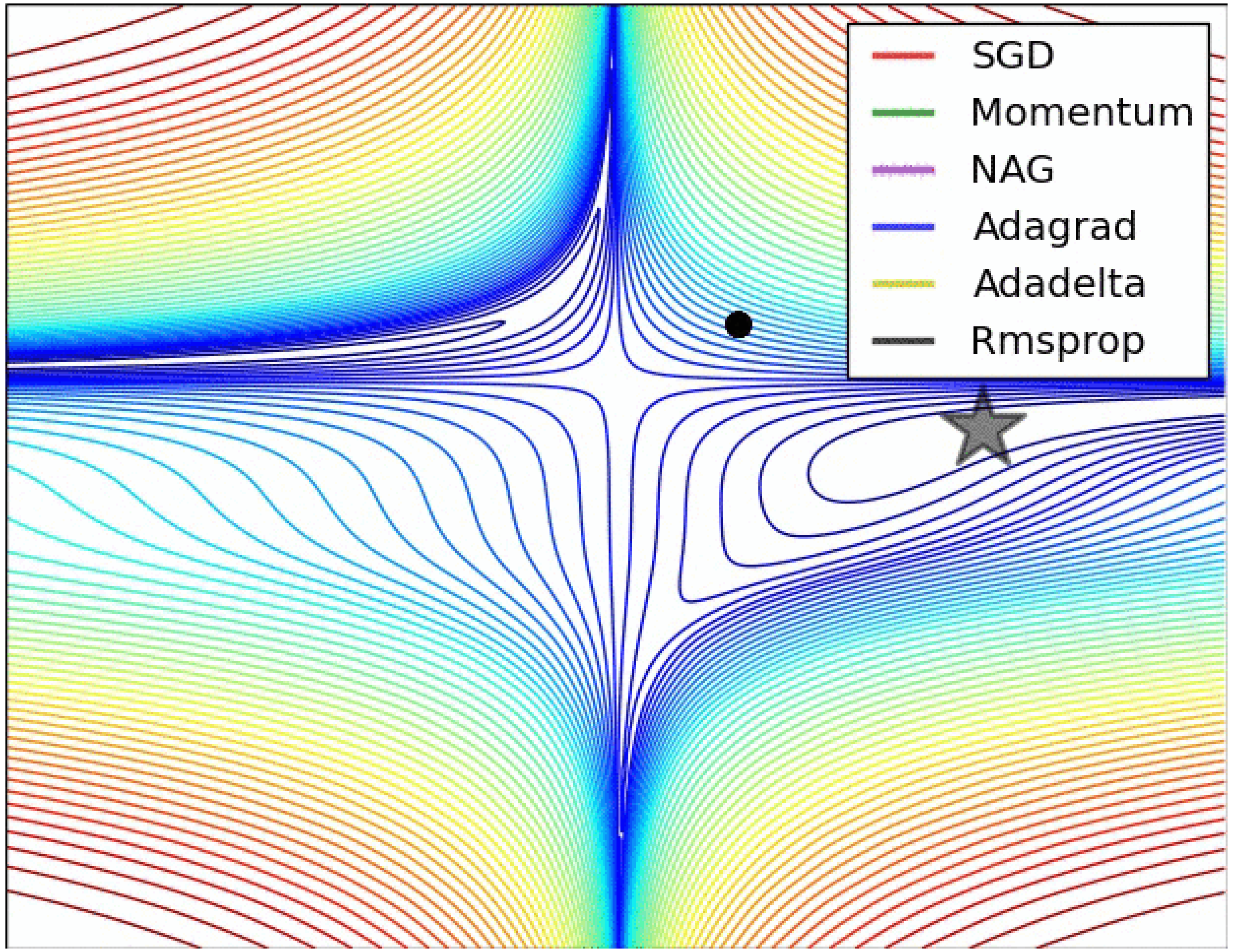


Intuition behind RMSProp. 1) Usual parameter updates 2) Once it reaches the convex bowl, exponentially weighted averaging would cause the effect of earlier gradients to reduce and to simplify, we can assume their contribution to be zero. This can be seen as if AdaGrad had been used with the training initiated inside the convex bowl

Optimization Algorithms: RMSProp

- The algorithm converges rapidly after finding a **convex bowl**, as if it were an instance of AdaGrad initialized within that bowl.
- In the Figure, the region represented by **1** indicates usual **RMSProp** parameter updates as given by the update equation, which is actually **exponentially averaged AdaGrad updates**.
- Once the optimization process lands on **A**, it essentially lands at the top of a convex bowl. At this point, intuitively, all the updates before **A** can be **seen to be forgotten due** to the exponential averaging.

Comparison



Comparison between the various optimization methods. It can be clearly seen that algorithms with adaptive learning rates provide faster convergence. NAG here refers to Nesterov Accelerated Gradient which is the same as Nesterov Momentum. Source: http://ruder.io/optimizing-gradient-descent/index.html#adam

Approximate Second-Order Methods

- The **optimization algorithms** that we've discussed till now involved computing **only the first derivative**. But there are many methods which involve higher order derivatives as well.
- The **main problem** with these algorithms are that they are not practically feasible in their vanilla form and so, certain methods are used to approximate the values of the derivatives.

Approximate Second-Order Methods

- The following are some second-order methods for optimization all of which use empirical risk as the objective function:
- Newton’s Method
- Conjugate Gradients
- Broyden–Fletcher–Goldfarb–Shanno (BFGS) Algorithm.
- Very high memory requirement for these second-order algorithms make it **impractical for most modern deep learning models** that typically have **millions of parameters**.

Optimization Strategies… Batch Normalization

Batch Normalization

- Batch normalization (BN) is one of the most exciting innovations in Deep learning that has significantly stabilized the learning process and allowed faster convergence rates.
- **The intuition:** Most of the deep models are compositions of many layers (or functions) and the **gradient** with respect to **one layer** is taken considering the **other layers** to be **constant**.

Optimization Strategies… Batch Normalization

- However, in practice all the layers are **updated simultaneously** and this can lead to unexpected results.

- **The idea:** To normalize the **inputs of each layer** in such a way that they have a **mean output activation of 0 and standard deviation of 1**. This is analogous to how the inputs to networks are standardized.

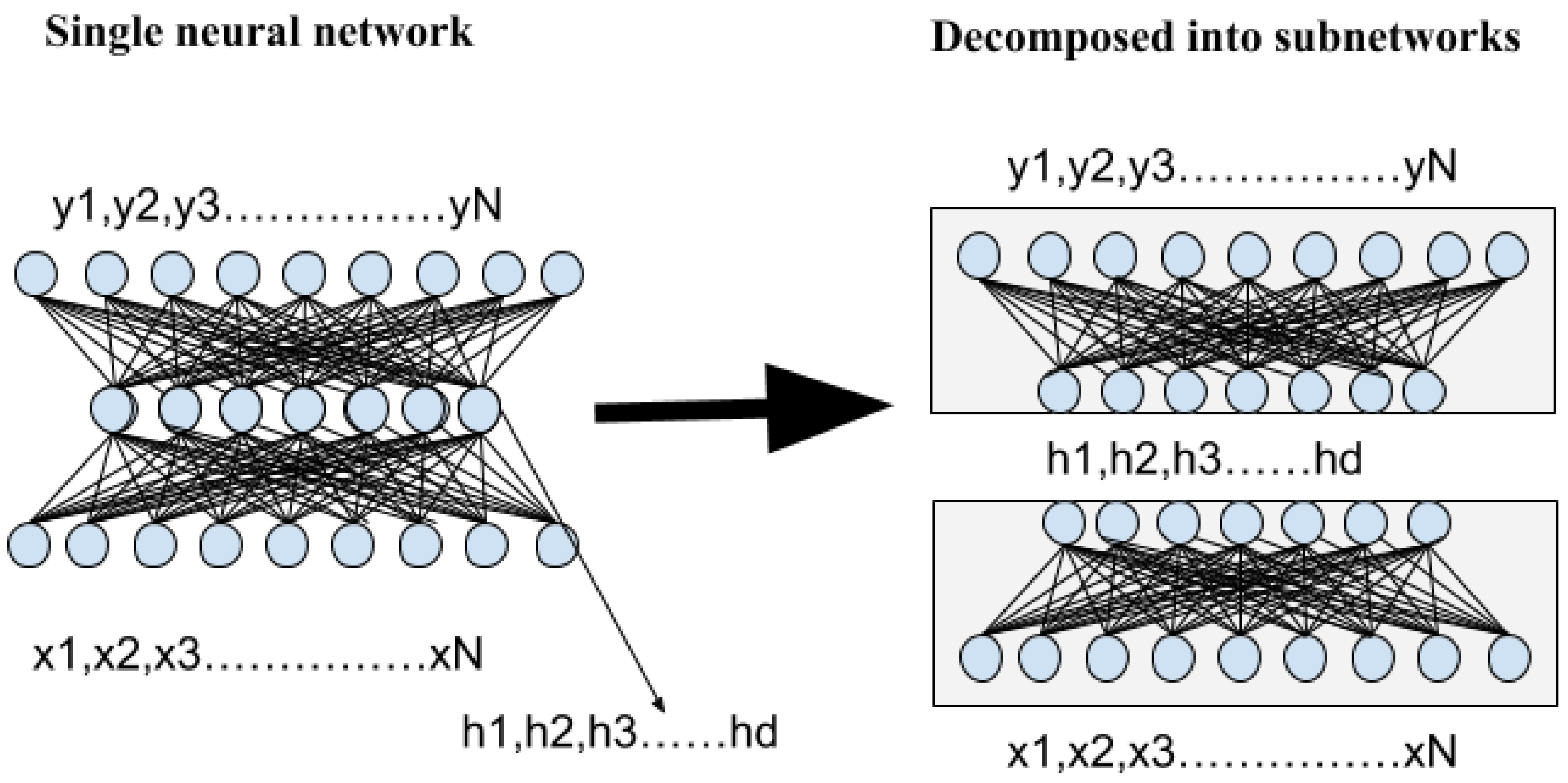
Optimization Strategies… Batch Normalization

- **How does this help?** We know that normalizing the inputs to a network helps it learn. But a network is just a series of layers, where the output of one layer becomes the input to the next. That means we can think of any layer in a neural network as the first layer of a smaller subsequent network.
- Thought of as a series of neural networks feeding into each other, we **normalize the output of one layer before applying the activation function**, and then feed it into the following layer (sub-network).

Optimization Strategies… Batch Normalization

- **Mathematical Intuition:** BN is about normalizing the hidden units activation values so that the distribution of these activations remains same during training.
- During training of any deep neural network if the hidden activation distribution changes because of the changes in the weights and bias values at that layer, they cause rapid changes in the layer above it.
- This slows down the training a lot. The change in distribution of the hidden activations during is called **internal covariate shift** which effect the training speed of the network.

Optimization Strategies… Batch Normalization



We look at the one single Deep neural network as multiple subnetworks

Optimization Strategies… Batch Normalization

- **How to Normalize the Hidden Units?**
- Consider we have **d** number of hidden units in a hidden layer of any Deep neural network.
- We can represent the activation values of this layer as **x=[x1,x2,………xd].**
- Now we can normalize the **k$_{\textbf{th}}$** hidden unit activation using the formula bellow.

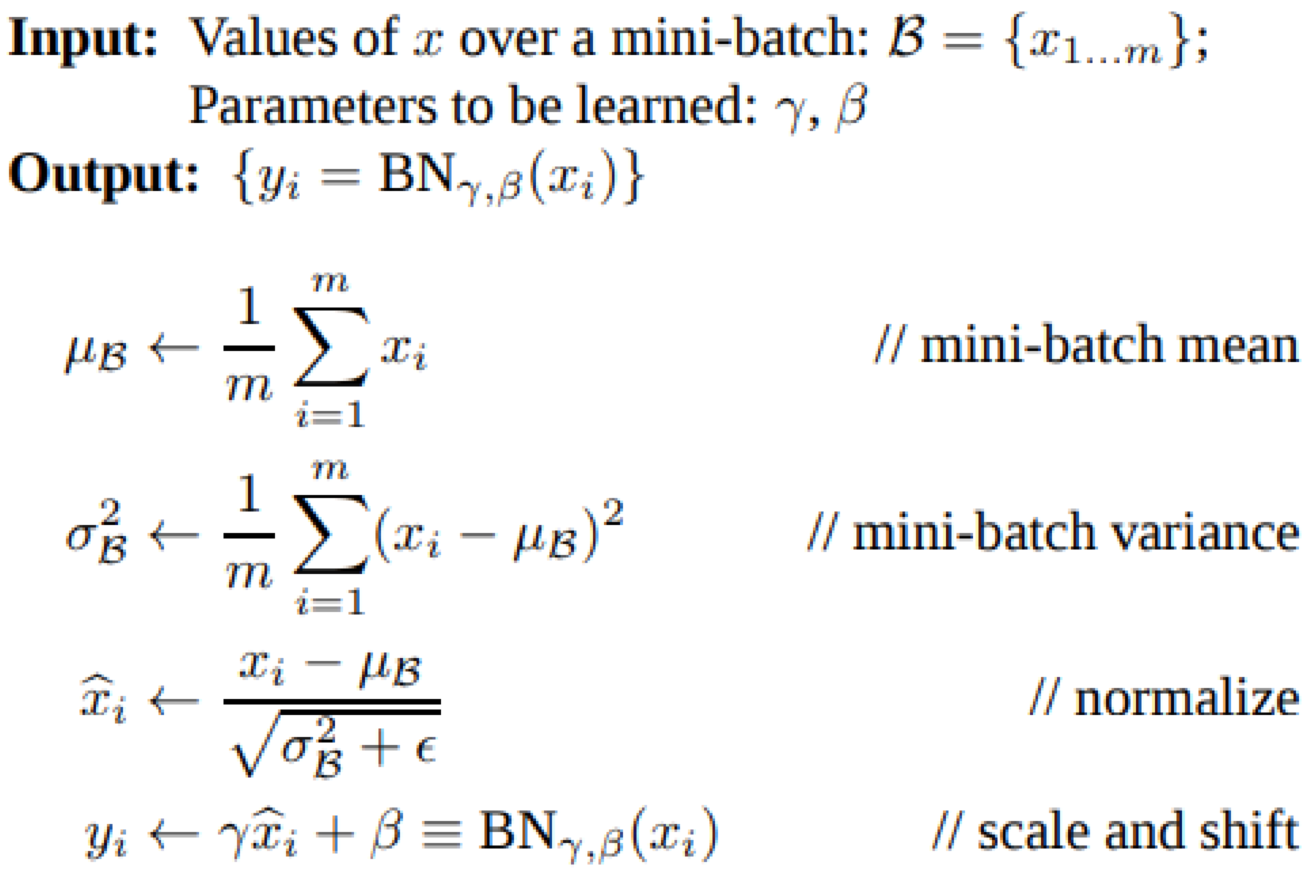
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Optimization Strategies... Batch Normalization

- For this, we introduce **2 new variables**, one for **learning the mean** and other for **variance**.
- These parameters are learned and updated along with weights and biases during training. The final normalized scaled and shifted version of the hidden activation for the **k$_{\text{th}}$** hidden unit is given bellow.

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$

Optimization Strategies… Batch Normalization



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

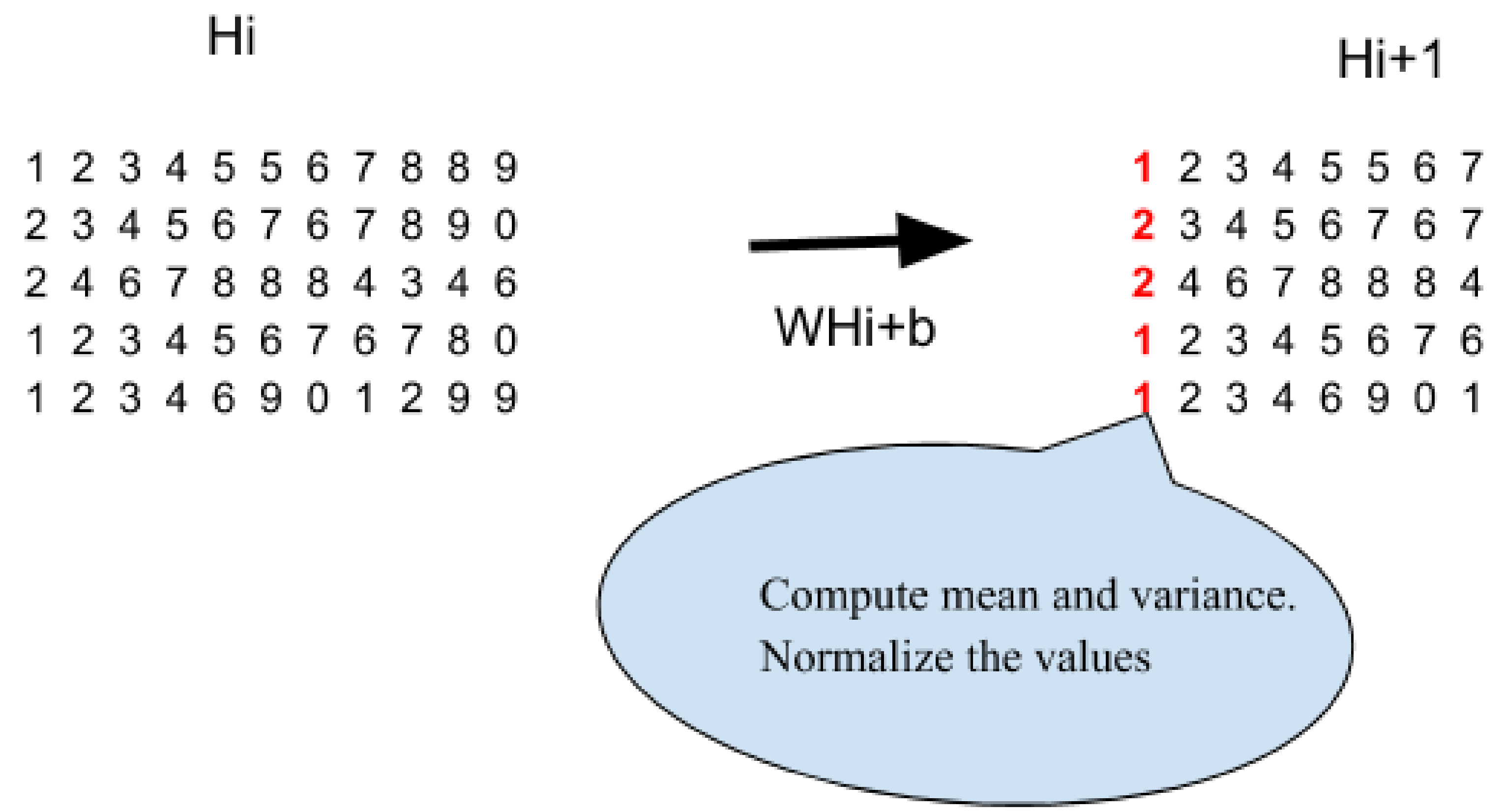
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

Optimization Strategies… Batch Normalization



Hi

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 5 & 6 & 7 & 8 & 8 & 9 \\ 2 & 3 & 4 & 5 & 6 & 7 & 6 & 7 & 8 & 9 & 0 \\ 2 & 4 & 6 & 7 & 8 & 8 & 8 & 4 & 3 & 4 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 6 & 7 & 8 & 0 \\ 1 & 2 & 3 & 4 & 6 & 9 & 0 & 1 & 2 & 9 & 9 \end{matrix}$$

Hi+1

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 5 & 6 & 7 \\ 2 & 3 & 4 & 5 & 6 & 7 & 6 & 7 \\ 2 & 4 & 6 & 7 & 8 & 8 & 8 & 4 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 6 \\ 1 & 2 & 3 & 4 & 6 & 9 & 0 & 1 \end{matrix}$$

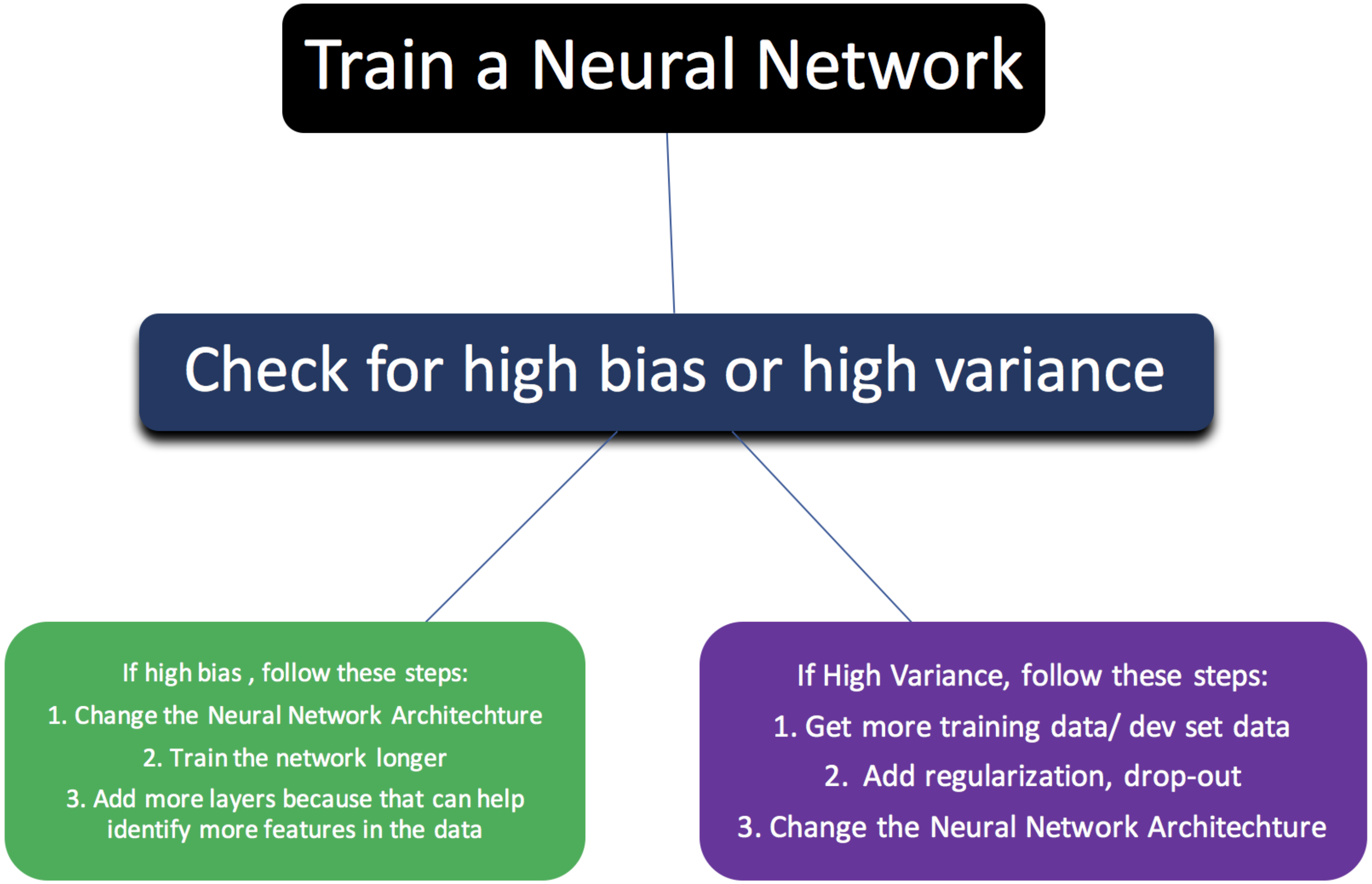
- Assume we have a minibatch of **m** training examples. We pass this minibatch to our neural network. At layer ***i*** we get The hidden activations matrix **Hi**.
- We then compute the mean and variance for each column as shown in figure and apply batch normalization transformation .

Optimization Strategies… Batch Normalization

BN during Inference/Testing

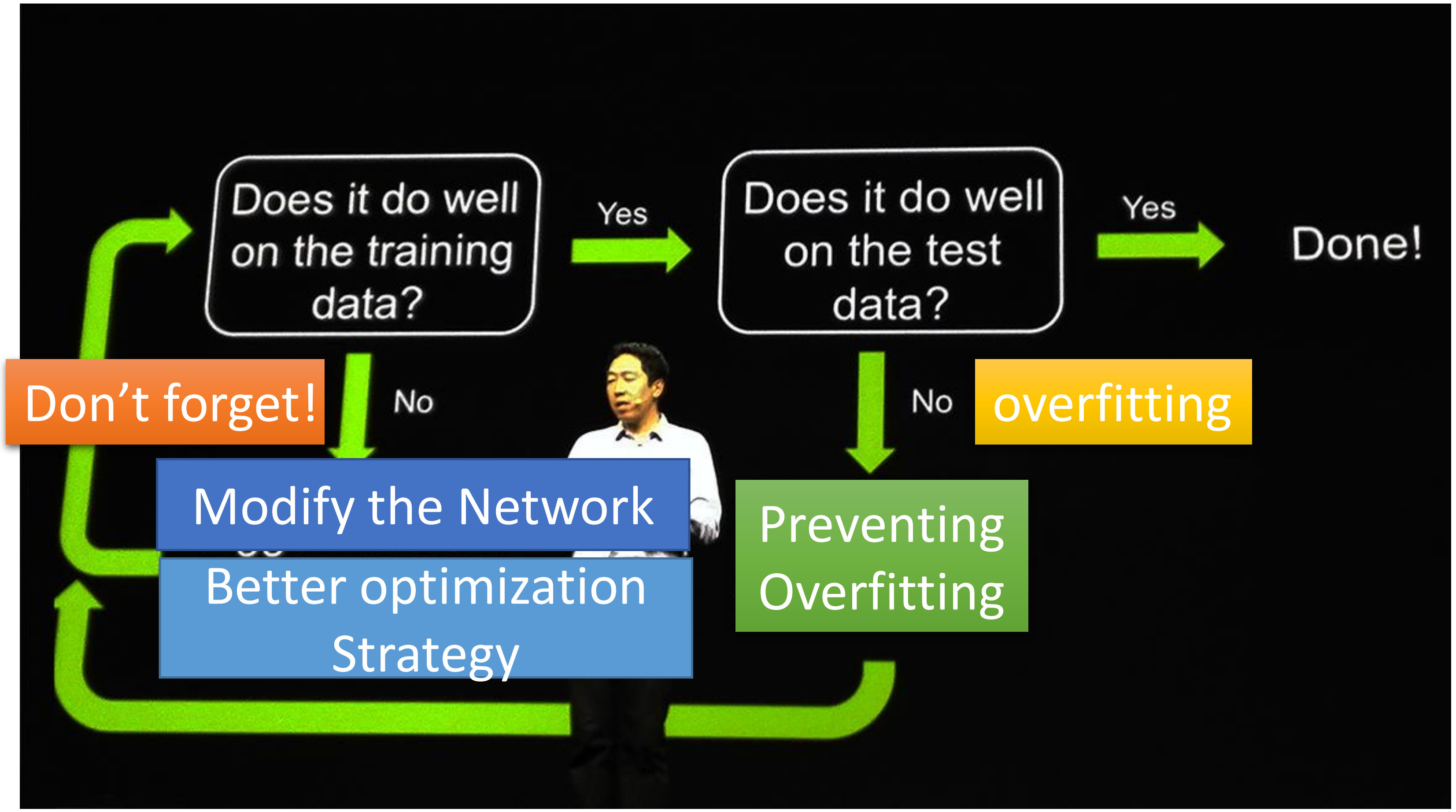
- During testing or inference phase we can’t apply the same **BN** as we did during training because we might pass **only one sample** at a time so it doesn’t make sense to find mean and variance on a single sample.
- We compute the **running average** of **mean and variance** of **k_{th}** unit during training and use those mean and variance values with trained batch-norm parameters during testing phase.

Some Tricks


Train a Neural Network
Check for high bias or high variance
If high bias , follow these steps:
1. Change the Neural Network Architechture
2. Train the network longer
3. Add more layers because that can help identify more features in the data
If High Variance, follow these steps:
1. Get more training data/ dev set data
2. Add regularization, drop-out
3. Change the Neural Network Architechture

How to resolve high bias and high variance

Recipe for Learning


Does it do well on the training data?
Yes
Does it do well on the test data?
Yes
Done!
No
No
Don't forget!
overfitting
Modify the Network
Better optimization Strategy
Preventing Overfitting

http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Thank You!