Limitations of Gradient Decent and Alternative Optimization Algorithms

Limitations of Gradient Decent

- Sensitive to Learning Rate
- Local Minima
- Slow Convergence
- Saddle Points
- Sensitivity to Initialization
- Computational Intensity
- Lack of Momentum

Exponentially Weighted Average

- A fast and efficient way to compute moving averages implemented in the different optimization algorithms

Example: Temperature θ_t over days, calculate the moving averages

v_t: Moving average value at day 't'

$v_0 = 0$

$v_1 = 0.9v_0 + 0.1\theta_1$

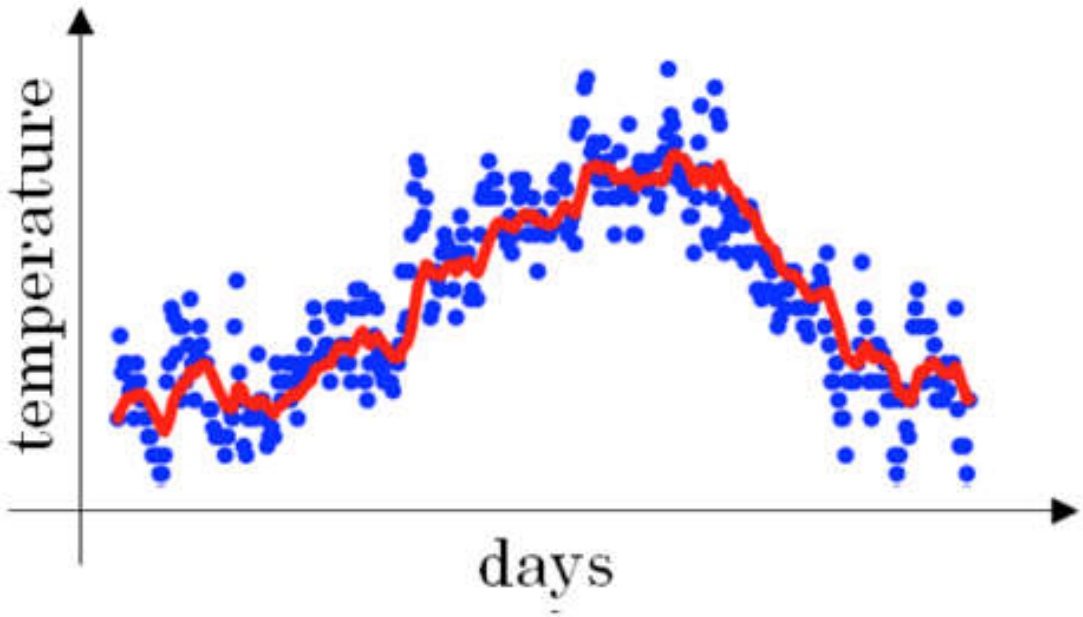
$v_1 = 0.9v_1 + 0.1\theta_2$

..

$v_t = 0.9v_{t-1} + 0.1\theta_t$

if $\beta = 0.9$

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$



Andrew NG

Exponentially Weighted Average

What does v_t and β means

v_t: averaging over $\frac{1}{1-\beta}$ days (approx)

For ex. , For β=0.9, $\frac{1}{1-\beta}$ ~=10 ;

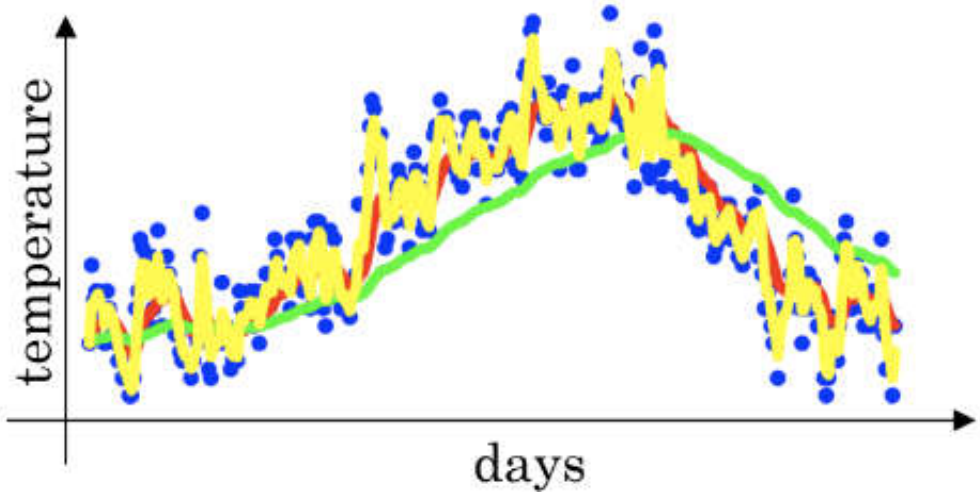
β = 0.9 averages over 10 days (smooth curve: Red Line)

For β=0.98, $\frac{1}{1-\beta}$ ~=50 ;

β = 0.98 averages over 50 days (smoother curve: Green Line)

For β=0.5, $\frac{1}{1-\beta}$ ~=2 ;

β = 0.5 averages over 2 days (Fluctuations: Yellow Line) - Much more noisy



Implementing Exponentially Weighted Average

v_θ: v is computing exponentially weighted average of parameter θ.

day 0: $v_\theta = 0$ day 1: $v_\theta = \beta v + (1-\beta)\theta_1$
day 2: $v_\theta = \beta v + (1-\beta)\theta_2$
...

Algorithms: $v_\theta = 0$ Repeat: {

Get next θ_t

$v_\theta := \beta v_\theta + (1-\beta)\theta_t$

}

Single line implementation for fast and efficient calculation of exponentially weighted moving average.

Example

- Calculate Moving Average for the following:
- [15,21,24,27,28,30,31,34]
- β=0.5

Solution

Starting with $v_0 = 0.0$, we can calculate v_t for each time step using the EMA formula:

1. $v_1 = 0.5 * 0.0 + (1 - 0.5) * 15 = 7.5$
2. $v_2 = 0.5 * 7.5 + (1 - 0.5) * 21 = 14.25$
3. $v_3 = 0.5 * 14.25 + (1 - 0.5) * 24 = 19.625$
4. $v_4 = 0.5 * 19.625 + (1 - 0.5) * 27 = 23.8125$
5. $v_5 = 0.5 * 23.8125 + (1 - 0.5) * 28 = 26.90625$
6. $v_6 = 0.5 * 26.90625 + (1 - 0.5) * 30 = 28.953125$
7. $v_7 = 0.5 * 28.953125 + (1 - 0.5) * 31 = 29.9765625$
8. $v_8 = 0.5 * 29.9765625 + (1 - 0.5) * 34 = 31.48828125$

Bias correction in Exponentially Weighted Moving Average

- Average (EMA) to mitigate an initial bias in the moving average values.
- When you first start computing the EMA, the initial values can be skewed because there's no prior information to provide a smooth start.
- Bias correction helps address this issue by making the early EMA values more accurate.
- The EMA formula is:

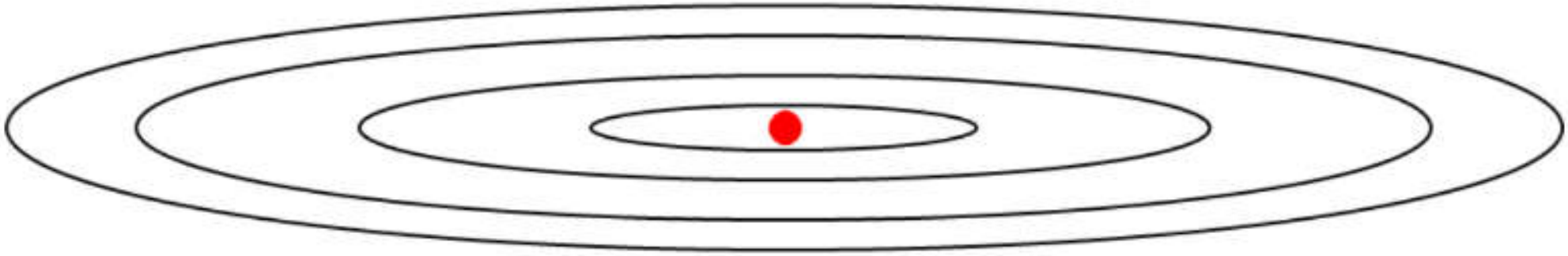
$$v_t = \beta_{vt-1} + (1-\beta)\theta_t$$

Bias correction in Exponentially Weighted Moving Average

- Initially, $v0$ is often set to 0. However, this can introduce a bias, especially when β is relatively high. The bias correction adjusts the initial $v0$ to account for this bias. The corrected formula is:

$$v_t = \frac{\beta_{vt-1} + (1-\beta)\theta_t}{1-\beta^t}$$

Momentum in Optimization



On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$

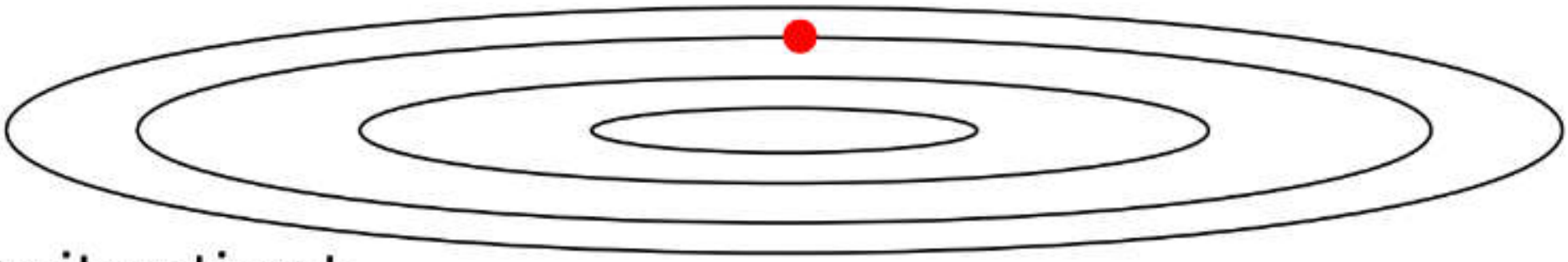
$$v_{db} = \beta v_{db} + (1-\beta)db$$

$$W = W - \alpha v_{dW},\ \ b = b - \alpha v_{db}$$

RMSprop: Root Mean Square Prop

- RMSprop is designed to address some of the limitations of traditional gradient descent by adapting the learning rates for each parameter separately.
- It adapts the learning rates individually for each parameter based on the historical behavior of their gradients.
- This helps in cases where some parameters have very different scaling or dynamic ranges.
- It also helps to prevent learning rates from becoming too small, as the squared gradients serve as a form of automatic learning rate annealing

RMSprop: Root Mean Square Prop



On iteration t

Compute dW and db on the mini batch

$$s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2$$

$$s_{db} = \beta_2 s_{db} + (1 - \beta_2) db^2$$

$$W = W - \alpha \frac{dW}{\sqrt{s_{dW} + \epsilon}}$$

$$b = b - \alpha \frac{db}{\sqrt{s_{db} + \epsilon}}$$

Adaptive Moment Estimation (ADAM)

- RMSprop only maintains the moving average of the squared gradients, which is used to adapt the learning rates for each parameter. It uses a single moving average (*vt*) to capture the second moment of the gradients.
- Adam, on the other hand, maintains two separate moving averages: one for the first moment of the gradients (*mt*) and another for the second moment of the gradients (*vt*). This means Adam keeps track of both the average gradient and the average of the squared gradient.

Adaptive Moment Estimation (ADAM)

- Adam incorporates bias correction for both *mt* and *vt* to account for the initialization of these moving averages. This bias correction helps in the early stages of optimization when the moving averages have a bias towards zero.
- RMSprop does not typically include bias correction, and this can lead to issues with the initial moving average estimates.

Adaptive Moment Estimation (ADAM)

- Adam has additional hyperparameters:
- $\beta 1$ (for the exponential moving average of the first moment)
- $\beta 2$ (for the exponential moving average of the second moment)
- ϵ (a small constant to prevent division by zero).

Adaptive Moment Estimation (ADAM)

$s_{dW} = 0, v_{dW} = 0, s_{db} = 0, v_{db} = 0$

On iteration t

Compute dW and db on the mini batch

$s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2, v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW$

$s_{db} = \beta_2 s_{db} + (1 - \beta_2) db^2, v_{db} = \beta_1 v_{db} + (1 - \beta_1) db$

$$s_{dW}^{corrr} = \frac{s_{dW}}{1 - \beta_2^t}, s_{db}^{corrr} = \frac{s_{db}}{1 - \beta_2^t}$$

$$v_{dW}^{corrr} = \frac{v_{dW}}{1 - \beta_1^t}, v_{db}^{corrr} = \frac{v_{db}}{1 - \beta_1^t}$$

$$W = W - \alpha \frac{v_{dW}^{corrr}}{\sqrt{s_{dW}^{corrr} + \epsilon}}, b = b - \alpha \frac{v_{db}^{corrr}}{\sqrt{s_{db}^{corrr} + \epsilon}}$$

- α needs to be tuned
- β_1 is by defaut 0.9
- β_2 is by defaut 0.999
- ε is by default 10^{-8}

Overfitting

- Memorizing is not learning!
- The word overfitting refers to a model that models the training data too well. Instead of learning the general distribution of the data, the model learns the expected output for every data point.
- This is the same a memorizing the answers to a math's quiz instead of knowing the formulas. Because of this, the model cannot generalize.
- Everything is all good as long as you are in familiar territory, but as soon as you step outside, you're lost.

How to detect overfitting?

- To test this ability, a simple method consists in splitting the dataset into two parts: the training set and the test set.

	Low *Training* Error	High *Training* Error
Low *Testing* Error	The model is learning!	Probably some error in your code. Or you've created a *psychic* AI.
High *Testing* Error	OVERFITTING	The model is not learning.

Bias vs Variance

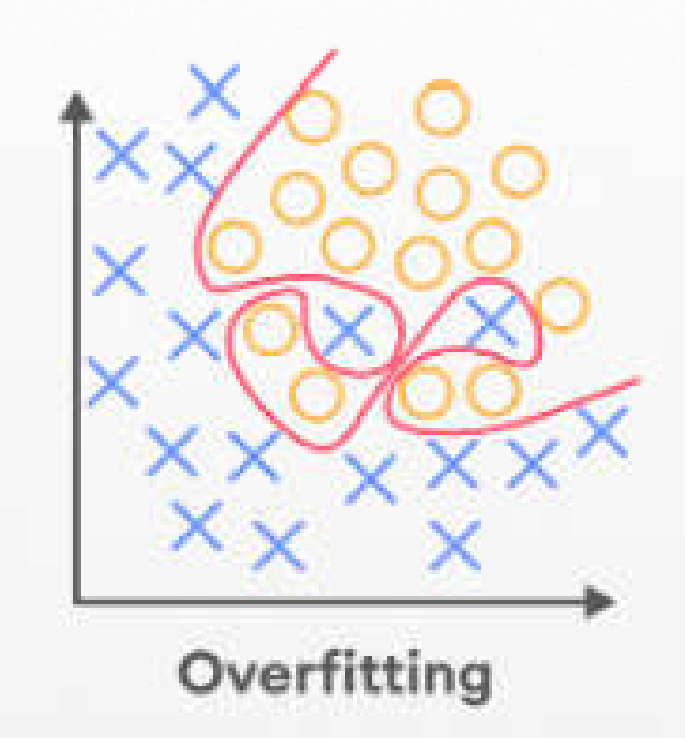
High Bias

- Problem with training data performance
- Bigger network
- More units in a layer

High variance

- More data
- Regularization

Bias variance tradeoff



How to overcome overfitting?

- Collect more Data
- Simplify the model
- Regularization
- Dropout Regularization
- Data Augmentation and noise
- Early stopping

Regularization

- Regularization is a technique in machine learning and statistics used to prevent overfitting and improve the generalization of a model.
- Overfitting occurs when a model learns the training data too well, capturing noise and fluctuations in the data rather than the underlying patterns.
- Regularization introduces a penalty on the complexity of the model, encouraging it to be simpler and less prone to overfitting.

Regularization

- There are two common types of regularization techniques: L1 (Lasso) regularization and L2 (Ridge) regularization, but there are other variations and techniques as well.

$$\lambda \sum_{j=0}^{M} |W_j| \qquad \lambda \sum_{j=0}^{M} W_j^2$$

L1 Penalty L2 Penalty

L1 (Lasso) Regularization

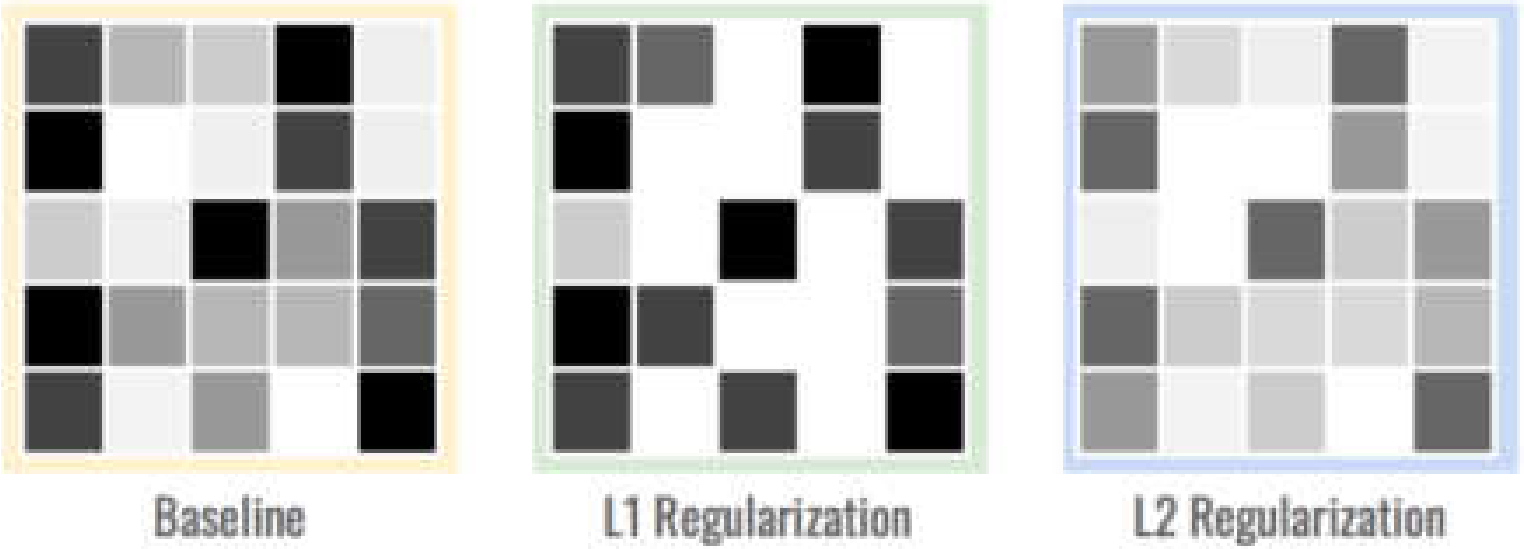
- In L1 regularization, a penalty term is added to the loss function, which is proportional to the absolute values of the model's coefficients.
- L1 regularization encourages sparsity in the model, meaning it pushes some of the feature weights to exactly zero, effectively eliminating those features from the model.
- By promoting sparsity, L1 regularization can help with feature selection, which can be useful in high-dimensional datasets where many features may be irrelevant.
- L1 regularization is often used when you want a simpler model with a reduced number of influential features

L2 (Ridge) Regularization

- In L2 regularization, a penalty term is added to the loss function, which is proportional to the squared values of the model's coefficients.
- L2 regularization encourages small weights for all features without forcing them to be exactly zero. It smooths the model and reduces the impact of any single feature.
- L2 regularization can help prevent the model from fitting the noise in the training data and makes it more robust to variations in the data.
- L2 regularization is often used when you want to prevent large weight values and reduce the risk of overfitting.

Regularization

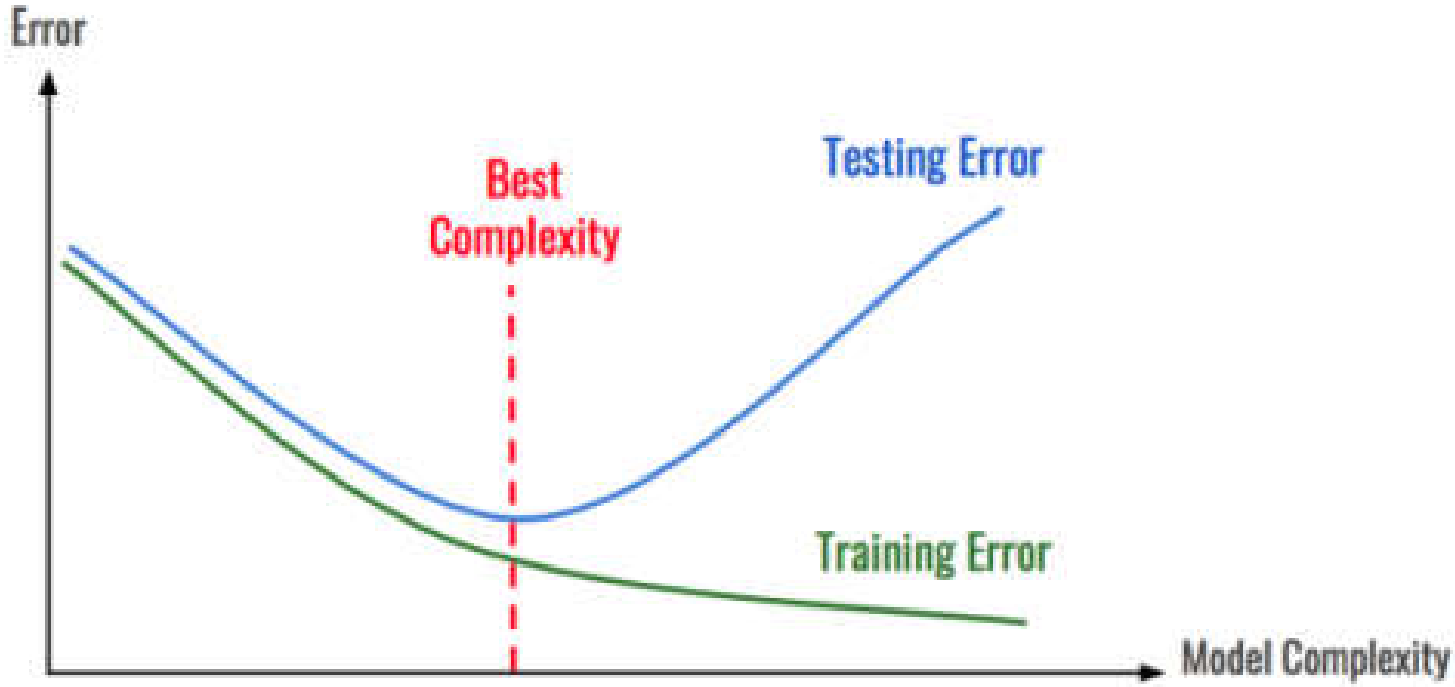
Here is what the weight matrixes would look like. Note how the **L1** matrix is **sparse** with many zeros, and the **L2** matrix has *slightly* **smaller weights**.



- Update cost function
- $J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^i, y^i) + \frac{\lambda}{2m}||W^2||_2$
- $J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^i, y^i) + \frac{\lambda}{m}||W||_1$

Early Stopping

- Monitor the model's performance on a validation set during training and stop training when the performance starts to degrade. This helps prevent the model from overfitting the training data.



On the left, the model is too simple. On the right it overfits.

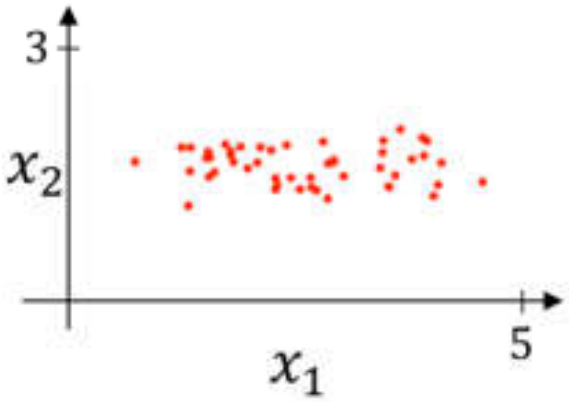
Data Augmentation and Noise

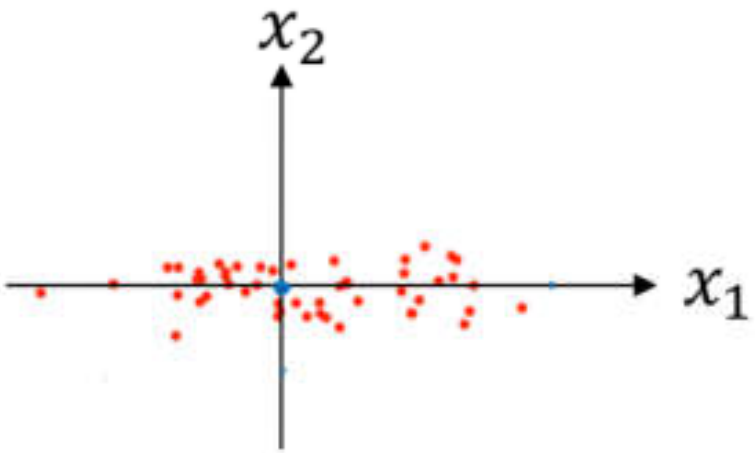
- Generate data using different transformation, scaling etc.

- Another good practice is to add Noise
  - To the input: This serves the same purpose as data augmentation, but will also work toward making the model robust to natural perturbations it could encounter in the wild.
  - To the output: Again, this will make the training more diversified.

Normalization

- Features can have different ranges (scale)
- Normalization helps in efficient learning

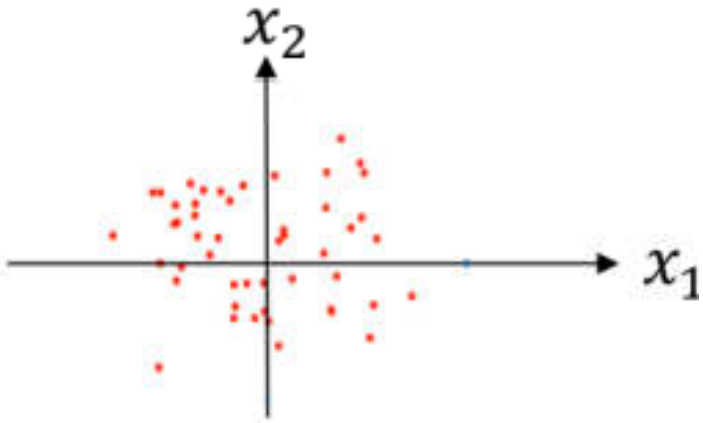




Subtract mean: zero mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^i$$

$$x = x - \mu$$

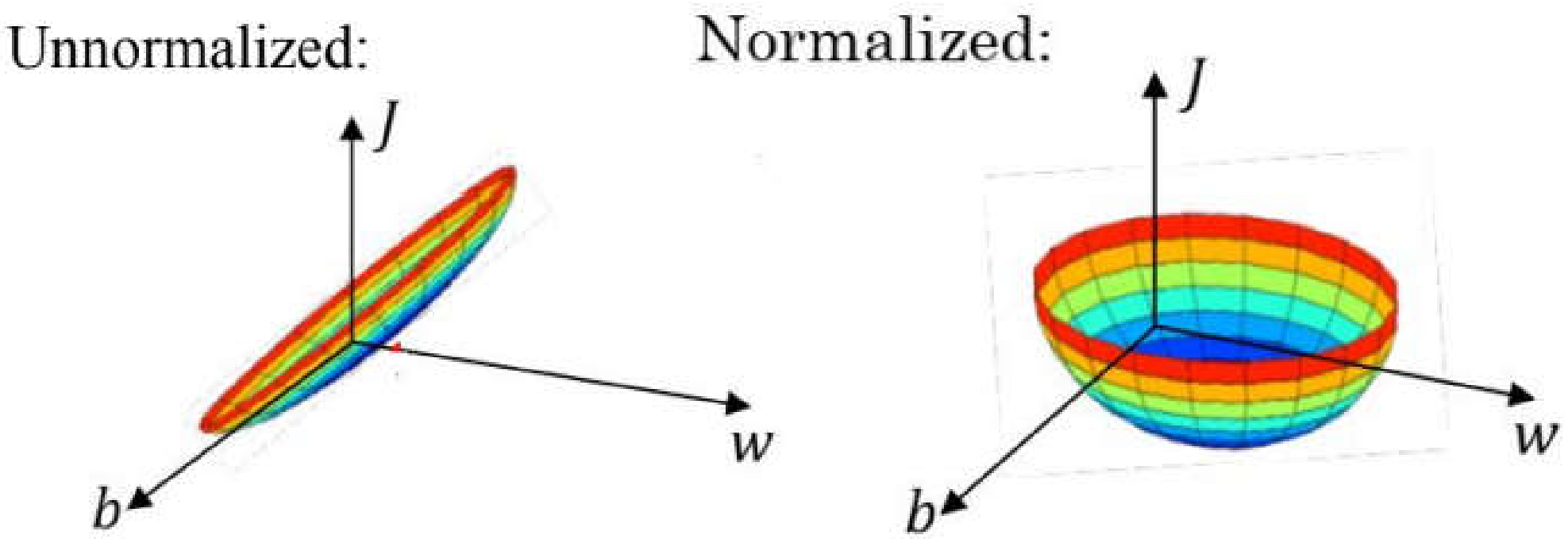


Normalize variance

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{i\,2}$$

$$x /= \sigma^2$$

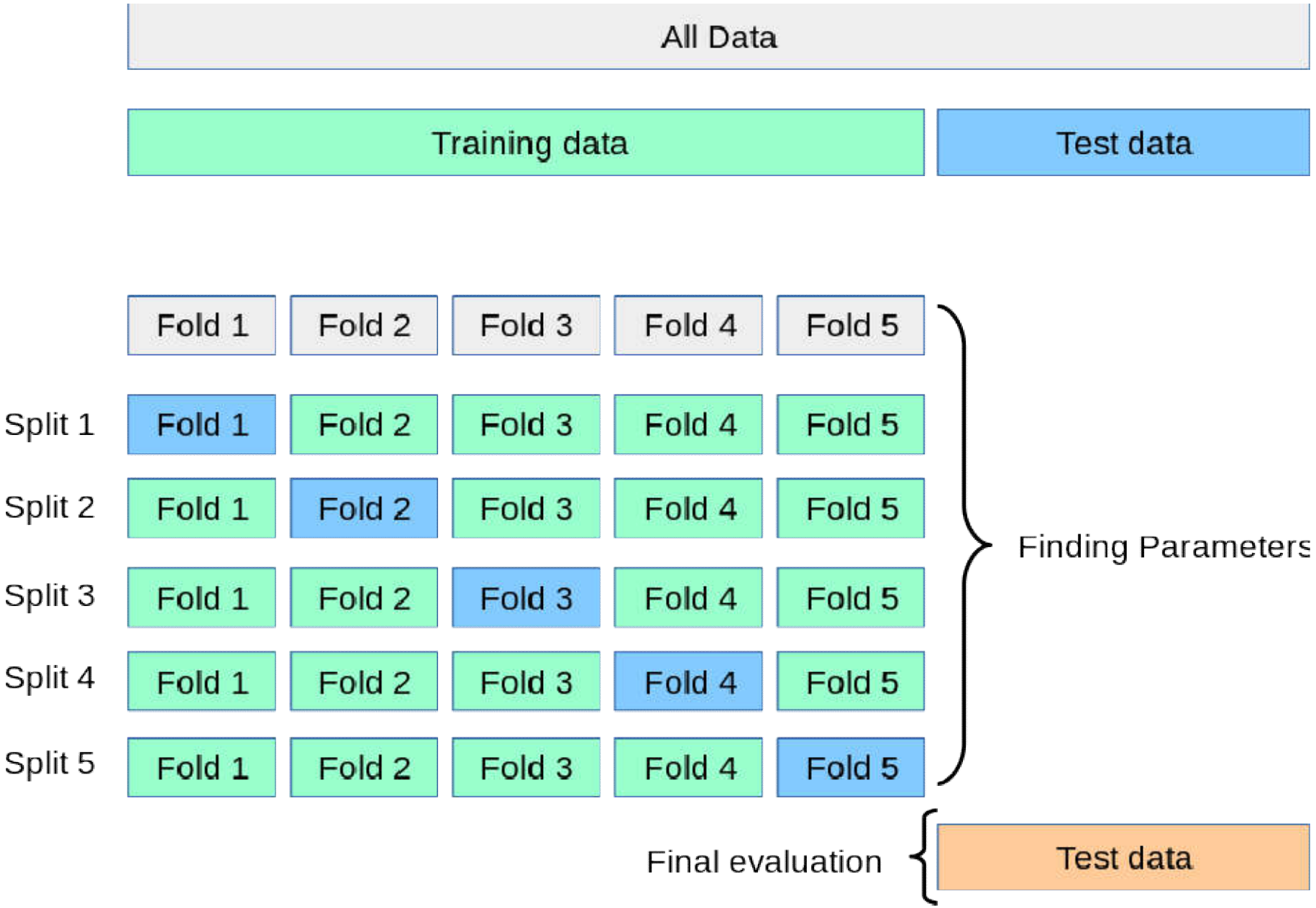
Normalization



Cross-Validation

- Use cross-validation techniques to assess the model's performance on different subsets of the data.
- Cross-validation helps you get a better estimate of how well your model generalizes to unseen data.
- The most common form of cross-validation is k-fold cross-validation, where the dataset is divided into k roughly equal-sized folds. The process involves the following steps:
- **Data Splitting**: The dataset is randomly partitioned into k subsets (or "folds"). Each fold contains a roughly equal proportion of the data.
- **Model Training and Testing**: The following steps are performed k times:
  - A model is trained on k-1 of the folds (the training set).
  - The model is then tested on the remaining 1 fold (the validation set). This process is repeated for each fold.

Cross-Validation



Cross-Validation

- Imagine we have a data sample with 6 observations:
- **[0.1, 0.2, 0.3, 0.4, 0.5, 0.6]**
- The first step is to pick a value for k in order to determine the number of folds used to split the data. Here, we will use a value of k=3. Because we have 6 observations, each group will have an equal number of 2 observations.

For example:

- Fold1: [0.5, 0.2]
- Fold2: [0.1, 0.3]
- Fold3: [0.4, 0.6]
- Three models are trained and evaluated with each fold given a chance to be the held out test set.

For example:

- **Model1**: Trained on Fold1 + Fold2, Tested on Fold3
- **Model2**: Trained on Fold2 + Fold3, Tested on Fold1
- **Model3**: Trained on Fold1 + Fold3, Tested on Fold2