# **Course Materials for GEN-AI** *Northeastern University* [2em]

**Instructor: Ramin Mohammadi** `r.mohammadi@northeastern.edu`

*Thank you for your understanding and collaboration.*

# Regularization

## Stein's Unbiased Risk Estimator

### Model Selection

The general task in ML is to find estimated function $f'(x)$ from a class $f(x)$ - real function that maps $x \to y$.

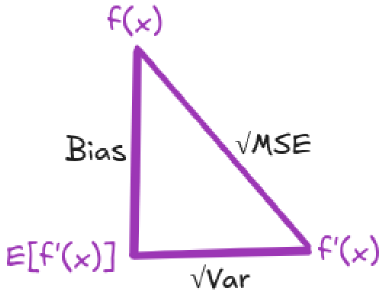


Figure 1: Bias-Variance

**Let us assume:**

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$$

**Definitions:**

- $f(\cdot)$: True $f$
- $f'(\cdot)$: Estimated $f$

**Real Model:**

$$y_i = f(x_i) + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2) \quad \text{(noise)}$$

**Estiamted Model:**

$$\hat{y}_i = f'(x_i)$$

In reality we are interested to calculate True Error $= E[f_i' - f_i]$ but we can only calculate Empirical Error $= E[\hat{y_i} - y_i]$

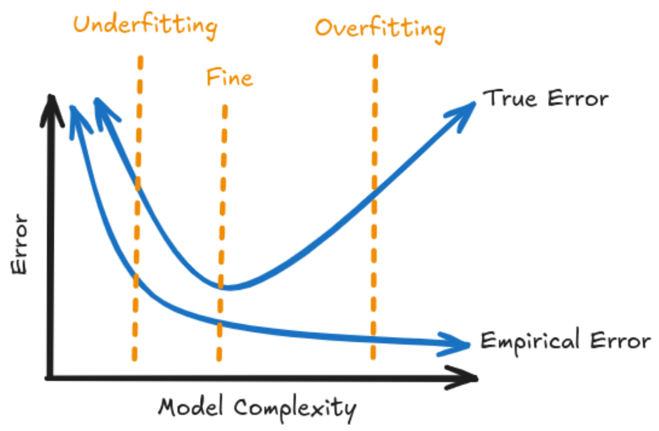


Figure 2: Complexity vs Error

$$\mathbb{E}\left[(\hat{y}_i - y_i)^2\right] = \mathbb{E}\left[\hat{f}(x_i) - (f(x_i) + \varepsilon_i)^2\right]$$
$$= \mathbb{E}\left[(\hat{f}(x_i) - f(x_i) - \varepsilon_i)^2\right] = \mathbb{E}\left[(\hat{f}(x_i) - f(x_i))^2 + \varepsilon_i^2 - 2\varepsilon_i(\hat{f}(x_i) - f(x_i))\right]$$
$$= \mathbb{E}\left[(\hat{f}(x_i) - f(x_i))^2\right] + \mathbb{E}\left[\varepsilon_i^2\right] - 2\mathbb{E}\left[\varepsilon_i(\hat{f}(x_i) - f(x_i))\right]$$
$$= \mathbb{E}\left[(\hat{f}(x_i) - f(x_i))^2\right] + \sigma^2 - [\text{Case 1 or Case 2}]$$

$\mathbb{E}\left[\varepsilon_i^2\right] = \sigma^2$ (expected value of $\mathcal{N}(0, \sigma^2)$).

## Case 1: $(x_i, y_i) \notin \mathcal{D}_{\textbf{training}}$

$$\mathbb{E}[\varepsilon_i(\hat{f}(x_i) - f(x_i))] = \mathbb{E}[(y_i - f(x_i))(\hat{f}(x_i) - f(x_i))] = 0$$

Expectation is 0 because ($\hat{f}(x_i) - f(x_i)$ and $\hat{f}(y_i) - f(x_i)$ are independent.) - This is the same as $Covariance(y_i and \hat{f}_i)$

$y_i$ is test data point and $\hat{f}_i$ is model developed our training data

$$\mathbb{E}[(\hat{y}_i - y_i)^2] = \mathbb{E}[(\hat{f}(x_i) - f(x_i))^2] + \sigma^2$$

For all points that are not part of $\mathcal{D}_{\text{training}}$ (test data):

$$\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 = \sum_{i=1}^{m}(\hat{f}(x_i) - f(x_i))^2 + \sum_{i=1}^{m}\sigma^2$$

Empirical error: $\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$

True error: $\sum_{i=1}^{m}(\hat{f}(x_i) - f(x_i))^2$

Noise variance: $\sum_{i=1}^{m}\sigma^2$

$$\text{True error } \approx \text{Empirical error} - m * \sigma^2$$

## Case 2: $(x_i, y_i) \in \mathcal{D}_{\textbf{training}}$

$$\mathbb{E}[\varepsilon_i(\hat{f}(x_i) - f(x_i))] \neq 0$$

(Because they are no longer independent as points are part of $\mathcal{D}_{\text{training}}$ and impact the learning of $\hat{f}_i$.)

$$2\mathbb{E}[\varepsilon_i(\hat{f}(x_i) - f(x_i))] = 2\sigma^2\mathbb{E}\left[\frac{\partial(\hat{f}(x_i) - f(x_i))}{\partial\varepsilon_i}\right]$$

(Using Stein's Lemma for Gaussian noise $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$.)

## Stein's Lemma:

If $x \sim \mathcal{N}(\theta, \sigma^2)$ and $g(x)$ is differentiable, then:

$$\mathbb{E}[g(x)(x - \theta)] = \sigma^2\mathbb{E}\left[\frac{\partial g(x)}{\partial x}\right].$$

$$= 2\sigma^2\mathbb{E}\left[\frac{\partial(\hat{f}(x_i) - f(x_i))}{\partial\varepsilon_i}\right]$$

As $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ and $(\hat{f}(x_i) - f(x_i)) = g(x)$

$$2\sigma^2\mathbb{E}\left[\frac{\partial\hat{f}_i}{\partial\varepsilon_i} - \frac{\partial f_i}{\partial\varepsilon_i}\right].$$

Using $f(x_i)$ is true function and independent of $\varepsilon_i$

$$= 2\sigma^2\mathbb{E}\left[\frac{\partial\hat{f}_i}{\partial\varepsilon_i}\right]$$

$$= 2\sigma^2\left[\frac{\partial\hat{f}_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial\varepsilon_i}\right]$$

(Using $y_i = f(x_i) + \varepsilon_i$):

$$= \frac{\partial y_i}{\partial\varepsilon_i} = 1.$$

$$= 2\sigma^2\left[\frac{\partial\hat{f}_i}{\partial y_i}\right]$$

$$= 2\sigma^2\left[\mathcal{D}_i\right]$$

$$\mathbb{E}[(\hat{y}_i - y_i)^2] = \mathbb{E}[(\hat{f}_i - f_i)^2] + \sigma^2 - 2\sigma^2 D_i$$

(for all training points).

$$= \sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \sum_{i=1}^{n}(\hat{f}_i - f_i)^2 + n\sigma^2 - 2\sigma^2 \sum_{i=1}^{n} \frac{\partial \hat{f}_i}{\partial y_i}.$$

$$\sum_{i=1}^{n}(\hat{f}_i - f_i)^2 = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2 - n\sigma^2 + 2\sigma^2 \sum_{i=1}^{n} \frac{\partial \hat{f}_i}{\partial y_i}.$$

**Key Terms:**

- True error: $\sum_{i=1}^{n}(\hat{f}_i - f_i)^2$.
- Empirical error: $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$ (small for a simpler model, smallest for an optimal model, increases for a complex model).
- Model complexity: $2\sigma^2 \sum_{i=1}^{n} \frac{\partial \hat{f}_i}{\partial y_i}$ (small for simpler models, large for complex models).

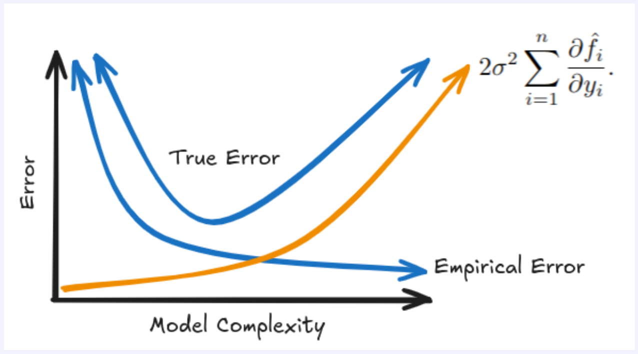


Figure 3: Model Complexity vs Error

**Notes:**

- Empirical error always goes down with increasing complexity, but model complexity term increases.
- For complex $\hat{f}_i$, the model complexity term is larger.
- True error goes up after some point as complexity increases.

## Regularization

A central problem in machine learning is to make models perform not only on the training data but also on new or unseen data. **Overfitting** occurs when a model learns not just the underlying pattern in the data but also noise and details specific to the training set, which leads to poor performance on new, unseen data. **Regularization** is a critical tool in ensuring that models generalize well and do not just memorize the training data.

By adding **regularization**, the model becomes more robust to variations in the data and is less likely to make incorrect predictions on unseen data due to **overfitting**.

## Weight Decay Regularization

The **weight decay** regularizer works by adding a penalty to the loss function based on the size (magnitude) of the weights, effectively "decaying" or shrinking them toward zero over time. This encourages the model to prefer smaller weights, leading to simpler models with improved generalization to unseen data.

The general loss function $J(\theta, x, y)$ is modified by adding a penalty term $\lambda f(\theta)$ where $\theta$ represents the model parameters(weights) and $\lambda$ controls the strength of the regularization.

$$J(\theta, x, y) = J(\theta, x, y) + \lambda f(\theta)$$

The penalty term behaves similarly to model complexity. As the weights $\theta$ increase, the penalty increases, which means the model becomes more complex. Larger weights are associated with models that have more capacity to overfit the training data.

Now calculating the **gradient of the regularized loss function:**

$$\nabla_\theta J(\theta, x, y) = \lambda w + \Delta_\theta J(\theta, x, y)$$

This indicates that the weight decay term $\lambda w$ is added to the gradient of the original loss function, which affects how the weights are updated during optimization.

Now **updating the weight**:

$$w := w - \alpha(\lambda w + \Delta_\theta J(\theta, x, y))$$

This shows that the weights are updated by both the original gradient of the loss function and the weight decay term, which shrinks the weights by a factor of $\lambda$.

## Why does Regularization work?

Let's understand why regularization works mathematically, deriving insights from the Taylor expansion of the loss function and the influence of regularization on the model's parameter space.

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + ...$$

$$J(w) = J(w^*) + \nabla J(w^*)(w - w^*) + \frac{1}{2}\nabla^2 J(w^*)(w-w^*)^2$$

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w-w^*)^T H (w - w^*)$$

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w-w^*)^T H (w - w^*) + \lambda \frac{1}{2}|w|_2^2$$

$$\nabla_w \hat{J}(w) = \nabla_w J(w^*) + H(w-w^*) + \lambda w$$

-

$$\frac{\partial}{\partial w_k}\frac{1}{2}(w-w^*)^T H(w-w^*) = \frac{\partial}{\partial w_k}\frac{1}{2}\sum_i\sum_j (w_i - w_i^*)H_{ij}(w_j - w_j^*)$$

$$= \frac{\partial}{\partial w_k}\sum_i\sum_j (w_i - w_i^*)H_{ij}(w_j - w_j^*)$$

**Two Cases:**

- When $i = k$:

$$\frac{\partial}{\partial w_k}\big((w_k - w_k^*)H_{kj}(w_j - w_j^*)\big) = H_{kj}(w_j - w_j^*).$$

- When $j = k$:

$$\frac{\partial}{\partial w_k}\big((w_i - w_i^*)H_{ik}(w_k - w_k^*)\big) = (w_i - w_i^*)H_{ik}.$$

**Symmetry of $H$:**

$$H_{ik} = H_{ki}, \quad H_{kj} = H_{jk} \quad \text{(as $H$ is symmetric and $i,j$ are both the same).}$$

$$\frac{\partial}{\partial w_k} = \frac{1}{2}\sum_{i=1} H_{kj}(w_j - w_j^*) + (w_i - w_i^*)H_{ik}.$$

$$= \frac{1}{2}\sum_{i=1} 2H_{ik}(w_i - w_i^*).$$

**Final Simplification:**

$$= H(w - w^*).$$

$$H(w - w^*) + \lambda w = 0$$

$$H(w) - H(w^*) + \lambda w = 0$$

$$(H + I\lambda)w = Hw^*$$

$$w = (H + I\lambda)^{-1}Hw^*$$

$$w = (H{+}I\lambda)^{-1}Hw^* \Rightarrow \Big\{ if \lambda = 0 \Rightarrow w = w^* \quad \text{(No regularization)} if \lambda > 0 \Rightarrow \text{First SVD on H} \Rightarrow H = Q \wedge Q^T$$

-

## Eigen Decomposition of a Symmetric Hessian Matrix

The eigen decomposition of $H$ is given by:

$$H = Q\Lambda Q^T,$$

where:

- $Q \in \mathbb{R}^{d\times d}$ is an orthogonal matrix ($QQ^T = Q^TQ = I$), whose columns are the eigenvectors of $H$.
- $\Lambda \in \mathbb{R}^{d\times d}$ is a diagonal matrix, where the diagonal elements $\lambda_1, \lambda_2, \ldots, \lambda_d$ are the eigenvalues of $H$.
- $Q^THQ = \Lambda$ is the diagonalized form of $H$.

## Singular Value Decomposition (SVD) of a Symmetric Hessian Matrix

The Singular Value Decomposition of $H$ is given by:

$$H = U\Sigma V^T,$$

where:

- $U \in \mathbb{R}^{d\times d}$: The left singular vectors, which are the eigenvectors of $HH^T$.
- $V \in \mathbb{R}^{d\times d}$: The right singular vectors, which are the eigenvectors of $H^TH$.
- $\Sigma \in \mathbb{R}^{d\times d}$: A diagonal matrix containing the singular values of $H$, where $\sigma_i = |\lambda_i|$, the absolute values of the eigenvalues of $H$.

For a symmetric Hessian matrix ($H = H^T$):

- $U = V$, meaning the left and right singular vectors are identical.
- The SVD simplifies to:
$$H = U\Sigma U^T,$$
which is equivalent to the eigen decomposition:
$$H = Q\Lambda Q^T.$$
Here, $Q = U = V$, and $\Sigma = |\Lambda|$ (the diagonal matrix of the absolute eigenvalues of $H$).

-

$$\begin{aligned}
w &= (Q \wedge Q^T + QQ^T\lambda)^{-1} Q \wedge Q^T w^* \\
&= \left[Q(\wedge + I\lambda)Q^T\right]^{-1} Q \wedge Q^T w^* \\
&= Q(\wedge + I\lambda)^{-1} Q^T Q \wedge Q^T w^* \\
&= Q(\wedge + I\lambda)^{-1} \wedge Q^T w^*
\end{aligned}$$

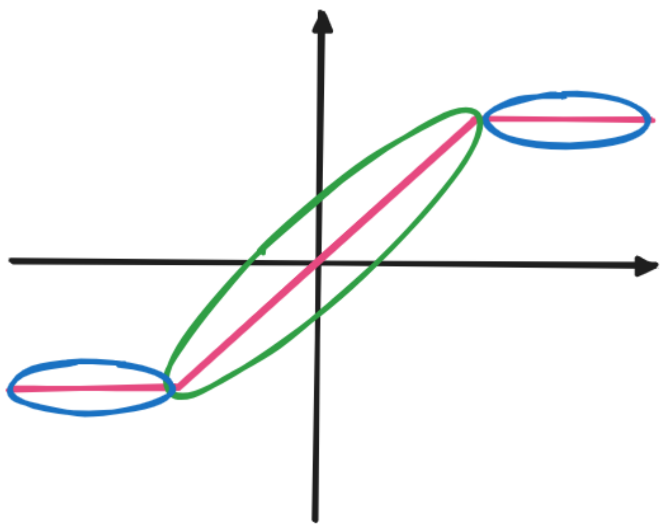
$$\left(\begin{bmatrix} s_1+\lambda & 0 & \cdots & 0 \\ 0 & s_2+\lambda & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_d+\lambda \end{bmatrix}\right)^{-1} = \begin{bmatrix} \frac{1}{s_1+\lambda} & 0 & \cdots & 0 \\ 0 & \frac{1}{s_2+\lambda} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{s_d+\lambda} \end{bmatrix}$$

$$\Rightarrow *\wedge = \begin{bmatrix} \frac{s_1}{s_1+\lambda} & 0 & \cdots & 0 \\ 0 & \frac{s_2}{s_2+\lambda} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{s_d}{s_d+\lambda} \end{bmatrix}.$$

Now,

$$\frac{s_i}{s_i+\lambda} \Rightarrow \begin{cases} \text{if } s_i \gg \lambda, & \frac{s_i}{s_i+\lambda} \approx 1 \quad \text{corresponding direction of search in the landscape is important,} \\ \text{if } s_i \ll \lambda, & \frac{s_i}{s_i+\lambda} \approx 0 \quad \text{corresponding direction of search in the landscape is not important.} \end{cases}$$

## Understanding the search space



### Activation Functions and Weights

- The behavior of activation functions (e.g., ReLU, sigmoid, tanh) depends on the magnitude of the inputs, which are determined by the weights in the network.
- **Large weights:** Inputs to the activation function often fall into non-linear regions, resulting in complex, non-linear transformations across layers. Corresponding direction of search in the landscape is not important
- **Small weights:** Inputs tend to stay within regions where the activation function behaves more linearly, simplifying the transformations. Corresponding direction of search in the landscape is important

### Linear vs Non-linear Behavior

- Non-linear activation functions (e.g., ReLU's "knee" or sigmoid's saturated regions) are critical for the expressiveness of neural networks.
- When weights are small, the network behaves closer to a linear model, as activation functions remain in their linear regions, reducing complexity.

### Simplification of Model Behavior

- Keeping weights small prevents the model from using complex, non-linear transformations, limiting its representational capacity.
- This linear-like behavior can be beneficial for early training or when overfitting or high complexity is undesirable.

### Connection to Search Directions

- When the model is closer to linear behavior, gradient-based optimization is more predictable, as the error surface is smoother and less rugged.
- Small weights act as a natural regularizer, simplifying the optimization landscape and reducing the importance of exact search directions.

## Regularization Techniques

Following we will discuss some of the most commonly used regularization techniques to tackle the overfitting problem.

### Data Augmentation

The goal is to improve the generalization ability of the model by exposing it to a wider variety of data. **Data augmentation** helps prevent overfitting, where a model performs well on the training data but poorly on unseen data, by increasing the diversity of the training samples without needing to collect new data.

Data augmentation involves creating modified versions of the original training data by applying various transformations, such as:

- **Rotation**: Rotating the images by a random degree.
- **Translation**: Shifting the image horizontally or vertically.
- **Scaling**: Zooming in or out on the image.
- **Flipping**: Mirroring the image horizontally or vertically.
- **Shearing**: Applying a distortion that tilts the image.

## Noise Injection

Injecting noise into data acts as a form of regularization by making the model more robust and less sensitive to small variations in the input data. When noise is added to the input data, the model learns to generalize across slightly perturbed versions of the same data point.

- **Adding noise to the input data:** For small noise, it has the effect to reduce the sensitivity of the output with respect to small variation in x. It is similar to adding additional regularization term to $J(w)$.
- **Adding noise to the weights:** It is similar to stochastic implementation of baysian inference over weights.

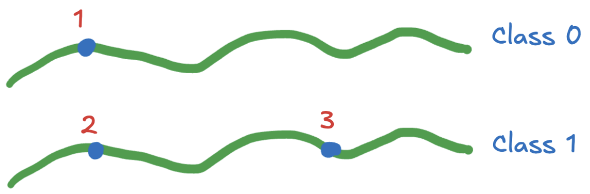
## Manifold Tangent Classifier

The **manifold tangent regularizer** is a technique that encourages a machine learning model to learn representations of data that are aligned with the underlying data manifold. It operates under the assumption that high-dimensional data often lies on a lower-dimensional manifold, meaning that the data has an intrinsic structure that can be expressed in fewer dimensions than the input space suggests.

The **tangent space** at a data point $x$ represents the directions in which the data can vary without leaving the manifold.

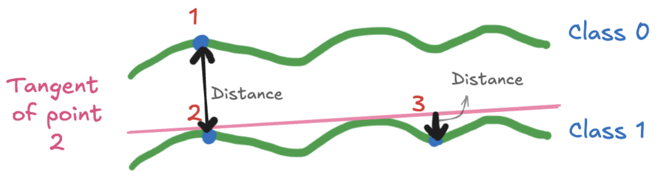
For example,

- If we draw points on a paper, the points have an intrinsic dimensionality of 2 but are on a 3D manifold.
- Assume taking pictures of your hand, at each photo you will rotate your hand. Now, If you take 100 photos, you will 100 dimensions of the photo, but as the actual degree of freedom of the hand is 1. Rotating only, so all images have intrinsic dimensionality of 1.



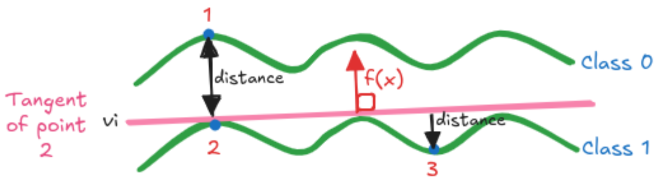
if we use nearest neighbor 1 and 2 will be together(Small euclidean).

So the idea here is that, instead of just looking at distances, look at distances between each point and the tangent space.



So we will look at the distance of each point and the tangent space of other point.

The idea here is that the derivative of $f(x)$ with respect to x should be orthogonal to tangent space $v_i$.



The ideal neural networks should be orthogonal to $v_i$. Orthogonal means perturbation of $x_i$ in the $v_i$ direction shouldn't change the value of the $f(x)$. Output remains the same for any $x_i$ on the $v_i$, but if $i$ go to the other manifold the second class then the decision should change. Our regularizer will be,

$$Regularizer = \lambda \sum_i \left( \frac{\partial f(x)}{\partial x} \cdot v_i \right)^2$$

Quite hard in practice as we don't know $v_i$.

## Early Stopping

**Early stopping** is a regularization technique in machine learning that helps prevent overfitting by halting the training process when the model's performance on a validation set starts to degrade. At each epoch we evaluate our model's performance against validation subset and if the efficacy improves, we store a copy of the model parameters.

Recall that,

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w^*)^T H (w - w^*)$$

$$\nabla_w \hat{J}(w) = H(w - w^*)$$

$$w^{t+1} = w^t - \lambda \nabla \hat{j}(w) = w^t - \lambda H(w^t - w^*)$$

$$w^{t+1} - w^* = w^t - w^* - \lambda H(w^t - w^*)$$

$$= (I - \lambda H)(w^t - w^*)$$

$$w^{t+1} = (I - \lambda H)(w^t - w^*) + w^*$$

If we continue for k steps:

$$\begin{aligned} w^k &= (I - \lambda Q \wedge Q^T)^k (w^0 - w^*) + w^* \\ &= (QQ^T - \lambda Q \wedge Q^T)^k (w^0 - w^*) + w^* \\ &= \left[ Q(I - \lambda \wedge) Q^T \right]^k (w^0 - w^*) + w^* \\ &= Q(I - \lambda \wedge)^k Q^T (w^0 - w^*) + w^* \end{aligned}$$

$$Q(I - \lambda \wedge)^k Q^T = \begin{bmatrix} (1 - \lambda s_1)^k & 0 & \cdots & 0 \\ 0 & (1 - \lambda s_2)^k & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & (1 - \lambda s_d)^k \end{bmatrix}.$$

$$\Rightarrow (1 - \lambda s_i)^k = \begin{cases} \text{if } \lambda s_i \approx small, \quad small \to 1 \text{if } \lambda s_i \approx large, \quad (1 - \lambda s_i)^k \to 0 \end{cases}$$

If k is large, because this value already goes to 0 for $\lambda s_i \gg large$, this means that we are no longer learning meaningful insight. After some epochs, the effect of important features (large $s_i$/eigen values) is 0, we have learned them already. It will focus on noise then.

### Parameter tying and parameter sharing

Parameter tying and parameter sharing are techniques used in machine learning to reduce the number of free parameters in a model, thereby acting as regularizers. They are particularly effective in deep neural networks, where the number of parameters can grow exponentially with the network's depth and width.

Sometimes we may need other ways to express our prior knowledge about suitable values of the model parameters. Certain parameters should be close to one another.

CNN is a similar concept, its a feed forward neural network in which some of the weights are tied.

For example, if we have a linear model $y = \theta^T x$, $\theta$ is dense vector means all features are important. If $\theta$ was sparse, only some of the features are important.

$$y = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_d x_d \begin{cases} Dense : ||y - \theta^T x||^2 & \approx \text{it's not a better generalization} \\ Sparse : ||y - \theta^T x||^2 + \lambda ||\theta|| & \approx \text{it's a better generalization} \end{cases}$$

Parameter tying and sharing is one of making the model sparse.

Consider a model $a$ with $w^a$ and a model $b$ with $w^b$. Suppose they both predict different thing(same data) but outputs are related:

$$\hat{Y_a} = f(w_a, X)$$
$$\hat{Y_b} = f(w_b, X)$$

Suppose we believe that the model parameters should be similar, then we use additional regularization of $\lambda||w_a - w_b||_2^2$.

## Bagging and other ensemble methods

**Ensemble methods** combine multiple models to improve prediction accuracy and generalization. The basic idea is that by aggregating the predictions of multiple models, we can reduce overfitting and variance, and achieve more robust predictions. Two major ensemble techniques are **Bagging** and **Boosting**, but there are others as well. Let's dive into them, starting with **Bagging**.

#### Bagging

**Bagging or Bootstrap Aggregating** is a technique that creates multiple versions of a model and combines them to improve accuracy and reduce variance. It works by training each model on a random subset of the training data and then averaging their predictions (for regression) or using majority voting (for classification).

By averaging the predictions of multiple models, bagging reduces the variance, thus mitigating overfitting. Even if some models overfit on their respective training subsets, their combined prediction tends to generalize better. Each model is trained on different subsets of the data, which makes the ensemble less sensitive to noisy data or outliers.

#### Boosting

**Boosting** is another ensemble method that builds models sequentially, where each new model tries to correct the errors of the previous ones. Unlike bagging, which builds models independently, boosting focuses on model dependency. Each subsequent model focuses more on the data points that the previous models misclassified or predicted poorly.

Some of the common boosting algorithms are:

- AdaBoost
- Gradient Boosting
- XGBoost

### Stacking

**Stacking** is an ensemble method that combines different types of models (heterogeneous models) by training a meta-model (a second-level model) to aggregate their predictions. Instead of using simple majority voting or averaging, stacking learns how to best combine the outputs of multiple models.

Stacking allows for the use of very different models (e.g., decision trees, neural networks, SVMs) and can learn to weight them appropriately, which might lead to better generalization compared to simple ensembles like bagging or boosting.

Ensemble methods are powerful techniques to improve model performance by combining the strengths of multiple models. **Bagging** helps reduce variance, while **Boosting** focuses on reducing bias. **Stacking** further enhance performance by combining the predictions of different types of models. These techniques are widely used in practice, especially in high-performing machine learning solutions such as Random Forests and Gradient Boosting Machines.