L9 Regularization Techniques

Regularization Techniques

- Similar to cloth, which has to be a right fit, the problem of overfitting and underfitting happens in ML & DL models
- There are techniques to tackle this overfitting and under fitting issue and these techniques are called regularization techniques.
- Regularization is a technique used to prevent overfitting and improve the generalization performance of a model.
- It involves adding a penalty term to the loss function during training.
- This penalty discourages the model from becoming too complex or having large parameter values, which helps in controlling the model's ability to fit noise in the training data.

Weight regularization as an approach to reduce overfitting for neural networks

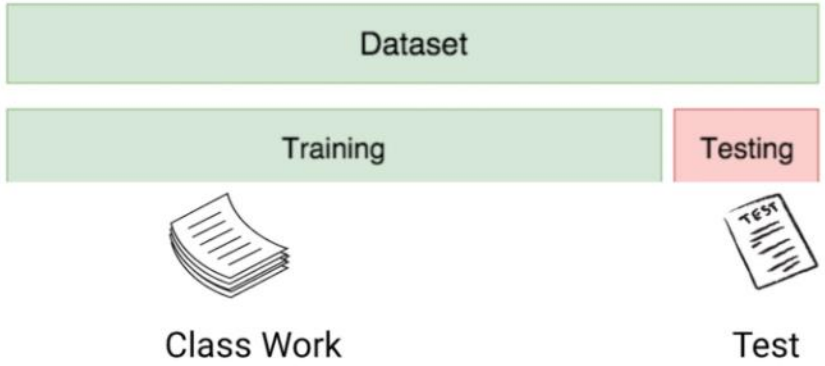
- Neural networks learn a set of weights that best map inputs to outputs.
- A network with large network weights can be a sign of an unstable network where small changes in the input can lead to large changes in the output.
- This can be a sign that the network has overfit the training dataset and will likely perform poorly when making predictions on new data.
- A solution to this problem is to update the learning algorithm to encourage the network to keep the weights small.
- This is called weight regularization and it can be used as a general technique to reduce overfitting of the training dataset and improve the generalization of the model.

Regularization Techniques

- Regularization in deep learning methods include
- L1 and L2 regularization,
- Dropout,
- Early stopping,
- Data augmentation
- By applying regularization, models become more robust and better at making accurate predictions on unseen data.

Overfitting and underfitting Model - Example

- Ex: Test Performance in academic setting



	Not interested in learning	Memorizing topics	Conceptual learning
Model-type	Underfitting	Overfitting	Best-fit
Sample Test performance on new data point	<50%	60-70%	>80%
Sample test performance on training data	<50%	98%	>80%

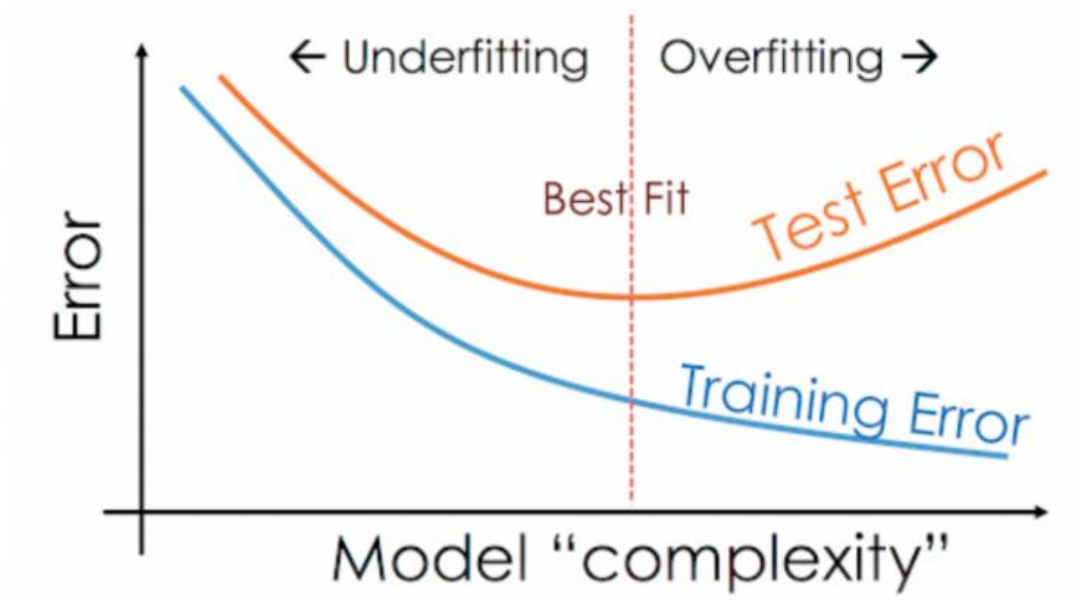
Overfitting Model

- If a given model is performing too well on the training data but the performance drops significantly over the test set is called an overfitting model.
- Ex: non-parametric models like decision trees, KNN are very prone to overfitting as these models can learn very complex relations which can result in overfitting
- Overfitting happens when a model becomes overly intricate, essentially memorizing the training data.
- While this might lead to high accuracy on the training set, the model may struggle with new, unseen data due to its excessive focus on specific details.

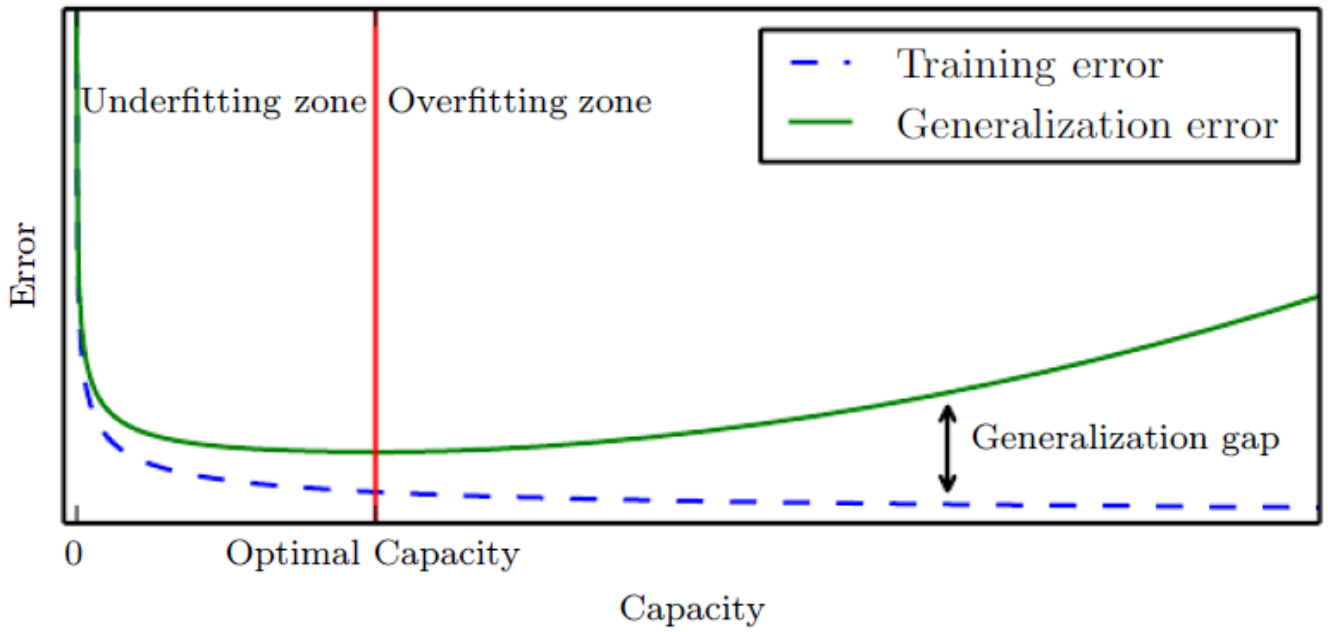
Underfitting model

- If the model is performing poorly over the test and the train set, then it is called an underfitting model.
- Ex: Building a linear regression model over non-linear data

Overfit vs Underfit vs Best-fit



Overfit vs Underfit vs Best-fit



Typical relationship between capacity and error.

Training and test error behave differently.

At the left end of the graph, training error and generalization error are both high. This is the underfitting regime.

As we increase capacity, training error decreases, but the gap between training and generalization error increases.

Eventually, the size of this gap outweighs the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity

Overfit vs Underfit vs Best-fit

- Simpler functions are more likely to generalize (to have a small gap between training and test error)
- we must still choose a sufficiently complex hypothesis to achieve low training error.
- Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming the error measure has a minimum value).
- Typically, generalization error has a U-shaped curve as a function of model capacity as shown in graph

Introduction

- The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task.
- We do so by building a set of preferences into the learning algorithm
- When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.
- For example, increasing or decreasing the degree of a polynomial for a regression problem.
- The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions
- Linear regression, has a hypothesis space consisting of the set of linear functions of its input

Introduction

- These linear functions can be very useful for problems where the relationship between inputs and outputs truly is close to linear.
- They are less useful for problems that behave in a very nonlinear fashion.
- For example, linear regression would not perform very well if we tried to use it to predict sin(x) from x
- We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

Introduction

- We can also give a learning algorithm a preference for one solution in its hypothesis space to another.
- This means that both functions are eligible, but one is preferred.
- The unpreferred solution will be chosen only if it fits the training data significantly better than the preferred solution.
- For example, we can modify the training criterion for linear regression to include weight decay.

Introduction

- To perform linear regression with weight decay, we minimize a sum comprising both the mean squared error on the training and a criterion J (w) that expresses a preference for the weights to have smaller squared L-2 norm.
- Minimize the mean squared error on the training set known as MSEtrain
- Loss function = Loss (say, binary cross entropy) + Regularization term

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^{\top} \boldsymbol{w},$$

- loss = loss + weight decay parameter * L2 norm of the weights
- Weight decay is a regularization technique by adding a small penalty, usually the L2 norm of the weights (all the weights of the model), to the loss function
- L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero)

Introduction

- where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights
- When λ = 0, we impose no preference, and larger λ forces the weights to become smaller
- Minimizing **J(w)** results in a choice of weights that make a tradeoff between fitting the training data and being small.

Why do we use weight decay?

- To prevent overfitting
- Because the L2 norm of the weights are added to the loss, each iteration of our network will try to optimize/minimize the model weights in addition to the loss.
- This will help keep the weights as small as possible, preventing the weights to grow out of control, and thus avoid exploding gradient.

Overfitting and underfitting

- We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set.
- Under this process, the expected test error is greater than or equal to the expected value of training error.
- The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.
2. Make the gap between training and test error small.

- These two factors correspond to the two central challenges in machine learning:
- underfitting and overfitting
- Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set.
- Overfitting occurs when the gap between the training error and test error is too large

Capacity

- We can control whether a model is more likely to overfit or underfit by altering its capacity.
- Informally, a model's capacity is its ability to fit a wide variety of functions.
- Models with low capacity may struggle to fit the training set.
- Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set

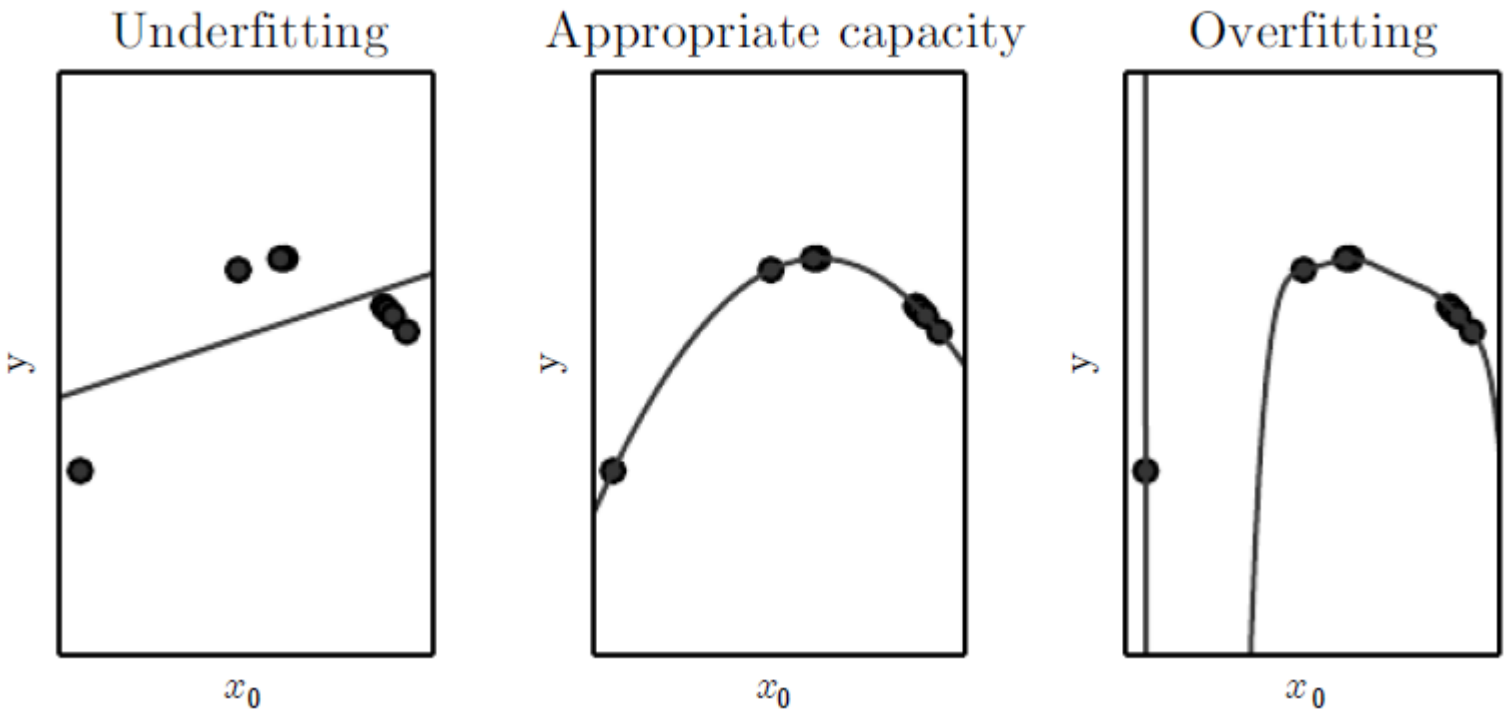
Hypothesis space

- One way to control the capacity of a learning algorithm is by choosing its hypothesis space
- Hypothesis space is the set of functions that the learning algorithm is allowed to select as being the solution.
- For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space.
- We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space.
- Doing so increases the model's capacity

Capacity

- Machine learning algorithms will generally perform best
  - when their capacity is appropriate for the true complexity of the task and
  - The amount of training data is also appropriate
- Models with insufficient capacity are unable to solve complex tasks.
- Models with high capacity can solve complex tasks
- But when their capacity is higher than needed to solve the present task they may overfit.

Capacity - Illustration



$$\hat{y} = b + wx$$

$$\hat{y} = b + w_1 x + w_2 x^2$$

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i$$

Here, a linear, quadratic and degree-9 predictor attempting to fit a problem where the true underlying function is quadratic.
The linear function is unable to capture the curvature in the true underlying problem, so it underfits.
The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass exactly through the training points because we have more parameters than training examples.
In this example, the quadratic model is perfectly matched to the true structure of the task so it generalizes well to new data

Capacity - Illustration

- We fit three models to the example training set.
- The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function.
- (Left)A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data.
- (Center)A quadratic function fit to the data generalizes well to unseen points.
- It does not suffer from a significant amount of overfitting or underfitting.
- (Right)A polynomial of degree 9 fit to the data suffers from overfitting.
- The solution passes through all of the training points exactly, but we have not been able to extract the correct structure.
- It now has a deep valley in between two training points that does not appear in the true underlying function.
- It also increases sharply on the left side of the data, while the true function decreases in this area.

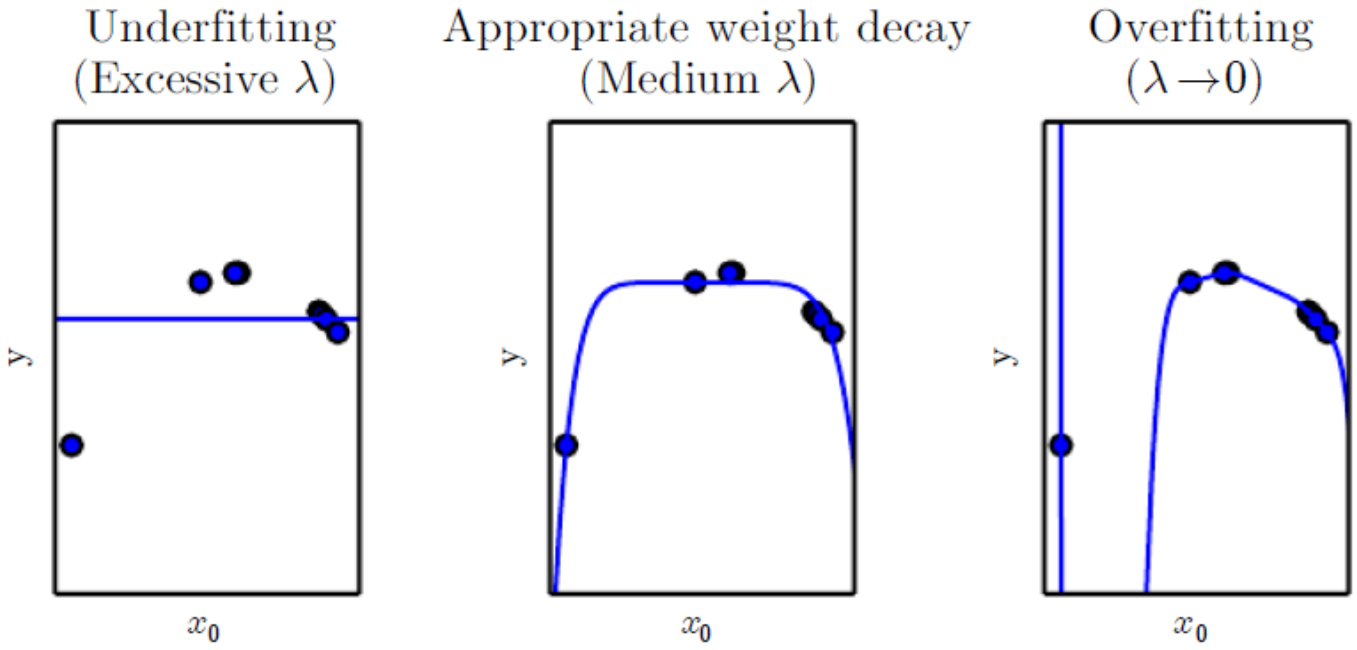
Test error

- The challenge in machine learning is that the model must perform well on new, previously unseen inputs—not just those on which our model was trained.
- The ability to perform well on previously unobserved inputs is called generalization.
- Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the training error, and we reduce this training error.
- we want the generalization error, also called the test error, to be low as well.
- The generalization error is defined as the expected value of the error on a new input.
- Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

Effect of weight decay on overfitting and underfitting

As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of λ

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w},$$



Effect of weight decay on overfitting and underfitting

- We fit a high-degree polynomial regression model to our example training set
- The true function is quadratic, but here we use only models with degree 9.
- We vary the amount of weight decay to prevent these high-degree models from overfitting.
- (Left)With very large λ, we can force the model to learn a function with no slope at all.
- This underfits because it can only represent a constant function.
- (Center)With a medium value of λ, the learning algorithm recovers a curve with the right general shape.
- Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients.
- (Right)With weight decay approaching zero the degree-9 polynomial overfits significantly

Introduction : Regularization and hyperparameter

- More generally, we can regularize a model that learns a function $f(x; \theta)$ by **adding a penalty called a regularizer to the cost function**.
- In the case of weight decay, the regularizer is $\Omega(w) = w^T w$.
- There are many other ways of expressing preferences for different solutions, both implicitly and explicitly.
- Together, these different approaches are known as **regularization**
- Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm.
- These settings are called hyperparameters

Introduction : Regularization and hyperparameter

- Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm.
- These settings are called hyperparameters.
- The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm).
- In the polynomial regression example, there is a single hyperparameter: the degree of the polynomial, which acts as a capacity hyperparameter.
- The λ value used to control the strength of weight decay is another example of a hyperparameter.

Introduction : Regularization and hyperparameter

- Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize
- More frequently, the setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set.
- This applies to all hyperparameters that control model capacity.
- If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting
- For example, we can always fit the training set better with a higher degree polynomial and a weight decay setting of $\lambda = 0$ than we could with a lower degree polynomial and a positive weight decay setting.

Introduction: Training and Validation

- To solve this problem, we need a validation set of examples that the training algorithm does not observe.
- It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters.
- For this reason, no example from the test set can be used in the validation set.
- Therefore, we always construct the validation set from the training data.

Introduction: Training and Validation

- Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters.
- The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly.
- We always construct the validation set from the training data.
- The validation set contains examples coming from the same distribution as the training set

Regularization

- Training a model involves two critical steps:
  - Optimization, when we need the loss to decrease on the training set;
  - Generalization, when the model has to work not only on the training set but also on data it has not seen before, like the validation set.
- *"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."*
- The mathematical tools aimed at easing these two steps are sometimes subsumed under the label regularization.

Regularization

- The first way to stabilize generalization is to add a regularization term to the loss
- This term is crafted so that the weights of the model tend to be small on their own, limiting how much training makes them grow.
- In other words, it is a penalty on larger weight values.
- This makes the loss have a smoother topography, and there's relatively less to gain from fitting individual samples
- The most popular regularization terms of this kind are L2 regularization, which is the sum of squares of all weights in the model, and L1 regularization, which is the sum of the absolute values of all weights in the model

Regularization - Summary

- In our weight decay example, we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize.
- There are many other ways of expressing preferences for different solutions, both implicitly and explicitly.
- Together, these different approaches are known as regularization.
- Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Regularization - Summary

- The no free lunch theorem has made it clear that there is no best machine learning algorithm, and,
- in particular, no best form of regularization.
- Instead we must choose a form of regularization that is well-suited to the particular task we want to solve
- Regularization is an indirect and forced simplification of the model.
- The regularization term requires the model to keep parameter values as small as possible, so requires the model to be as simple as possible.
- Complex models with strong regularization often perform better than initially simple models, so this is a very powerful tool.

Regularization - Summary

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

- L1 Regularization and L2 Regularization or Ridge regression
- Dropout
- Data Augmentation
- Early stopping

L1 & L2 Regularization

- L2 weight decay is the most common form of weight decay, other ways to penalize the size of the model parameters is to use L1 regularization.
- L2 Regularization - is the sum of squares of all weights in the model.

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

- L1 Regularization –is the sum of the absolute values of all weights in the model.

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

- Note: These weights can be positive or negative and hence taking the absolute values

L2 Regularization in Pytorch

```
# Compute the loss and its gradients
loss = loss_fn(outputs, labels)

l2_lambda = 0.001
l2_norm = sum(p.pow(2.0).sum()
              for p in model.parameters())
loss = loss + l2_lambda * l2_norm
```

Note: replace pow(2.0) with abs() for L1 regularization

Weight decay parameter in PyTorch .SGD() method

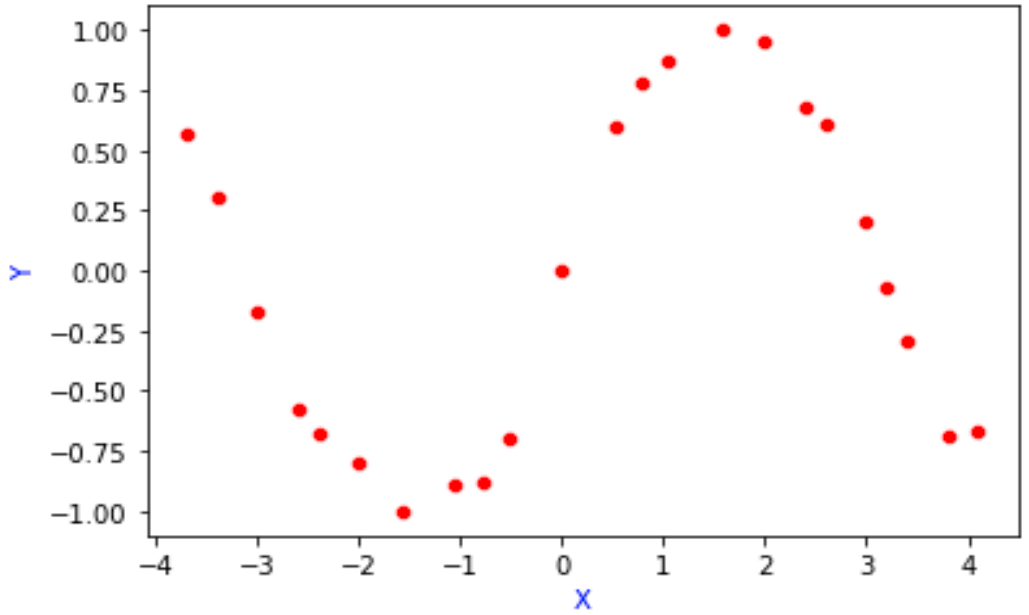
```
LAMBDA = 2
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, weight_decay=LAMBDA)
```

Bias and Variance

- In machine learning, we collect data and build models using training data.
- We apply that model to test data, which the model has not seen, and do predictions.
- Our main aim is to reduce the prediction error.
- We build the model by minimizing training error but we are more concerned about test error/prediction error.
- Prediction error depends on bias and variance.
- There are two types of error in machine learning. Reducible error and Irreducible error. Bias and Variance come under reducible error.
- Note: **Irreducible Error **- Those error cannot be reduced irrespective of any algorithm that we use in the model. It is caused by unusual variables that have a direct influence on the output.

Bias Error

- Model Building – Illustrate how the training error and prediction error differ when we increase the model complexity.
- Ex: we have below data points. We have to find the relationship between X and Y.



Bias Error- Model Building

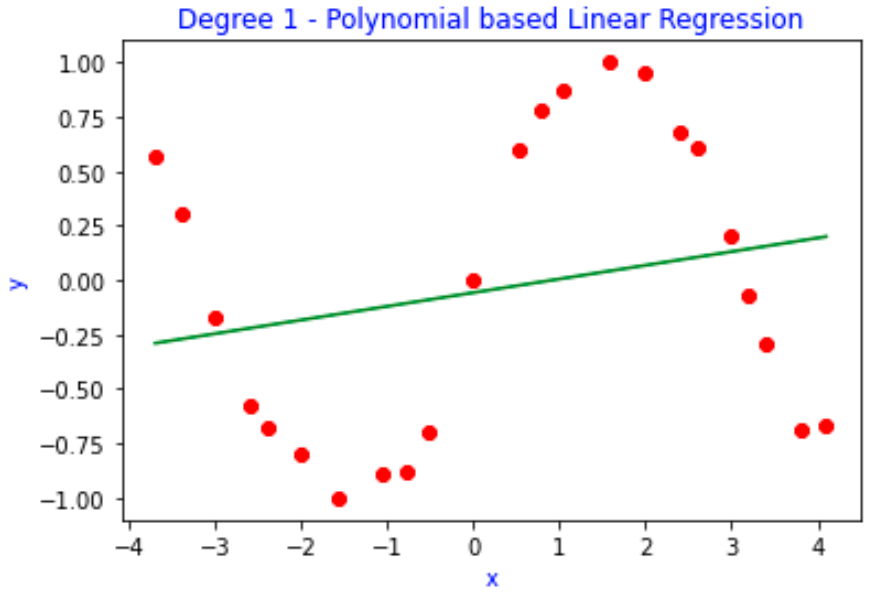
- True relationship or True function between X and Y is denoted as f(X). This function is unknown
- $Y=f(X)+\varepsilon$
- Now, we have to build a model which depicts the relationship between X and Y.
- Input → Model → Output
- Learning Algorithm: The learning algorithm will accept input and returns a function that depicts the relationship between X and Y.
- Input → Learning Algorithm → $\hat{f}(X)$
- Ex: In Linear Regression, the learning algorithm is gradient descent, which finds the best fit line based on cost function

Bias Error - Model Building

- Given a data set, we split it into training data and test data.
- Training data — Build the model using training data
- Test data — Predicting the output using the model chosen.
- consider 4 models build on the training data making an assumption of how y is related to x

Bias Error- Model Building

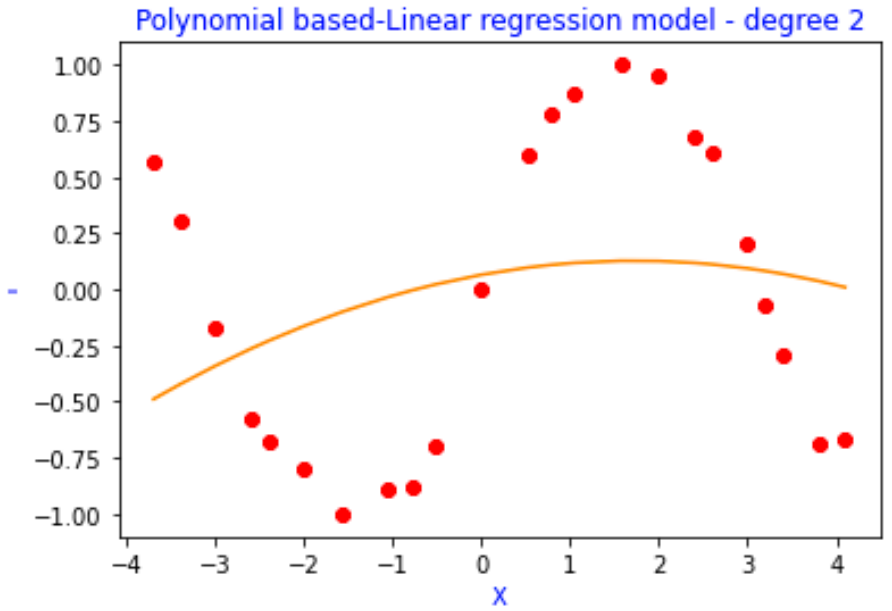
1. **Simple Model → Degree 1 → $y=\hat{f}(x) = w0 + w1x$**



Here in this simple model, the fitted line is far away from the data points, so the training error will be high.

2. Degree 2 polynomial

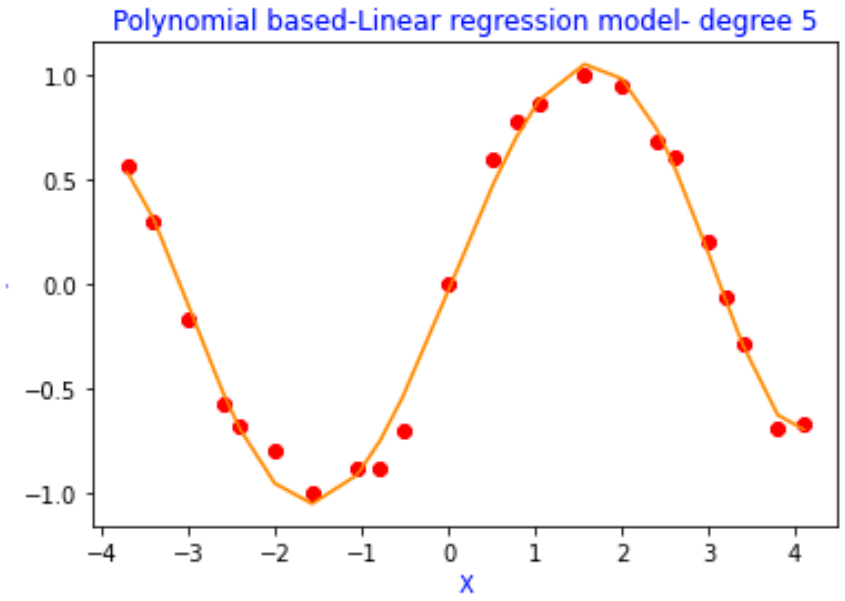
$y=\hat{f}(x) = w0 + w1x + w2x^2$



Bias Error- Model Building

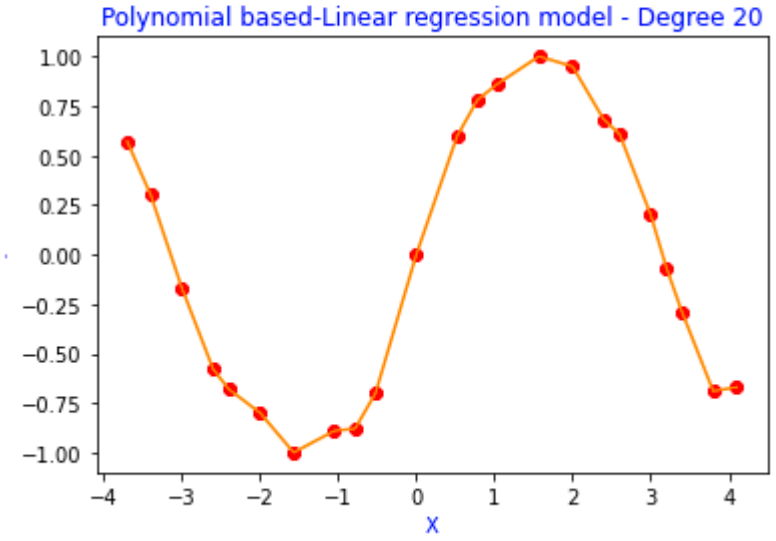
3. Degree 5 polynomial

$$y=\hat{f}(x) = \sum_{i=1}^{5} \omega_i x^i + w_0$$



4. Complex Model → Degree 20

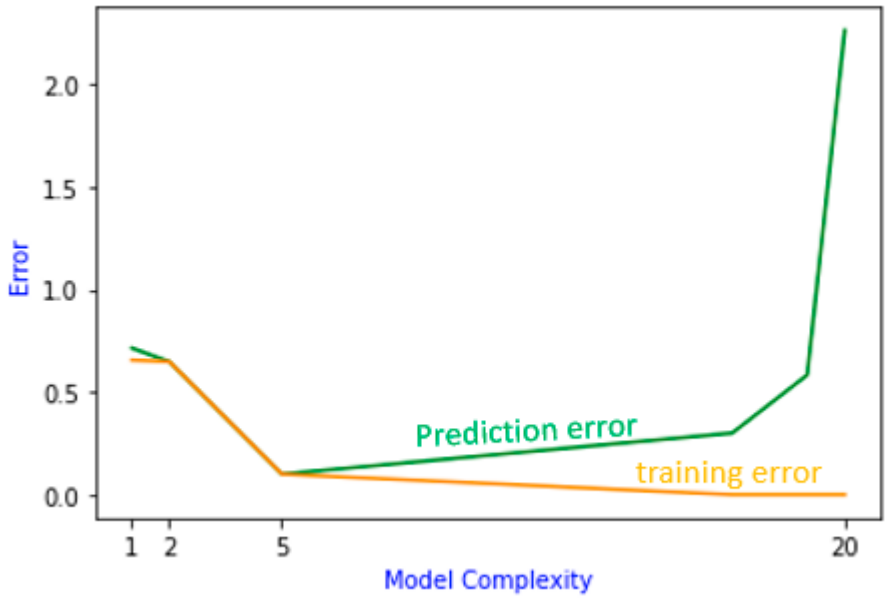
$$y=\hat{f}(x) = \sum_{i=1}^{20} \omega_i x^i + w_0$$



the fitted curve passes through all the data points, so the training error will be close to zero. This model tries to memorize the data along with the noise instead of generalizing it. So, this model will not perform well on test data/validation data which is unseen. This scenario is known as Overfitting.

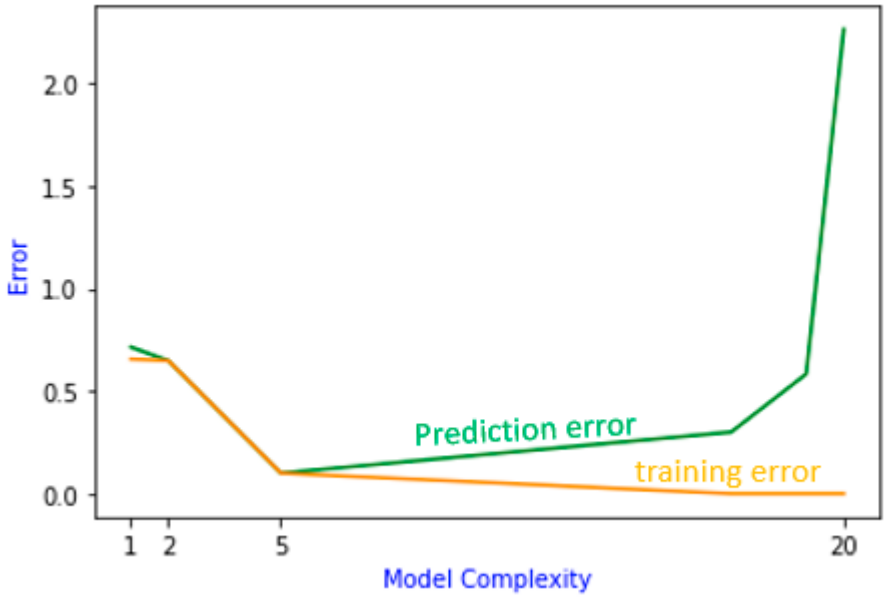
Bias Error– Prediction using models

- If we do prediction using these 4 models on validation data, we will get different prediction errors.
- Now plotting training error and prediction error vs model complexity(in our case, degree of polynomial)



Bias Error– Prediction using models

- From the graph, as model complexity increases[degree 1, degree2, degree 5, degree 20], training error tends to decrease.
- But prediction error decreases to some extent and when the model becomes more complex, it increases.
- There is a trade-off between training error and prediction error.
- At the end of the two curves, there is a high bias at one end and high variance at another end.
- So, there is a trade-off between bias and variance to achieve the ideal model complexity.



Bias and Variance

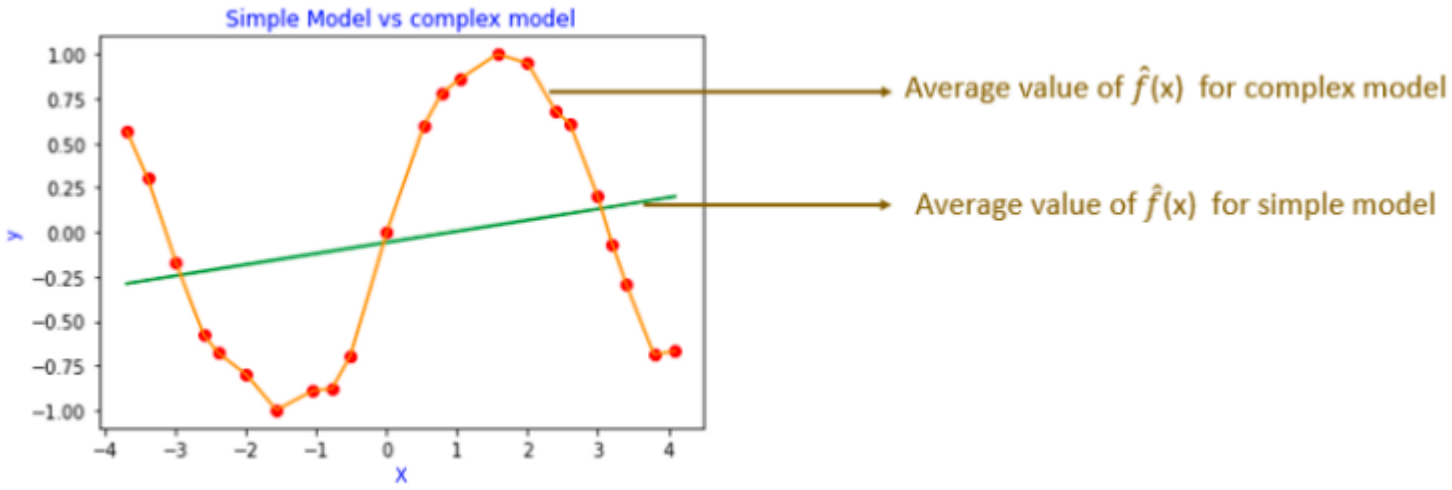
- Bias: say f(x) is the true model and f̂(x) is the estimate of the model
- Bias(f̂(x))= E[f̂(x)]-f(x)

$$\text{Bias}(\hat{Y}) = E(\hat{Y}) - Y$$

- Bias is the difference between the expected value and the true function.
- E[f̂(x)] → Expected value of the model.

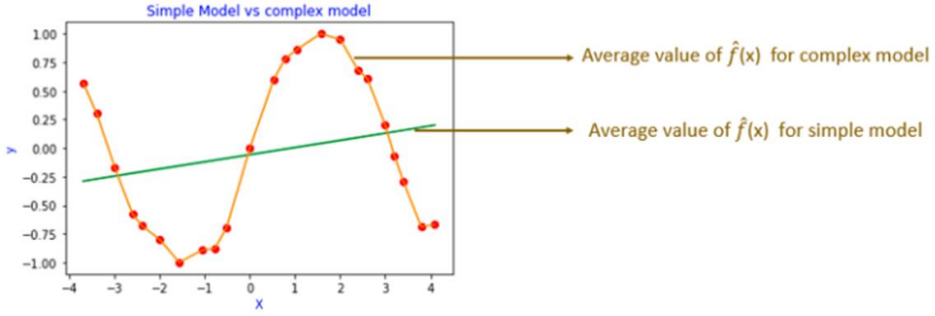
How to calculate the expected value of the model

- Build the model ($\hat{f}(x)$) using the same form(ex. polynomial degree 1) on different random samples drawn from the training data.
- Then calculate the expected value of all the functions which is denoted as $E[\hat{f}(x)]$.



Bias and Variance

- the orange fitted curve is the average of all the complex models (degree=20) performed on different random samples drawn from the training data.
- the green fitted line is the average of all the simple models(degree=1) performed on different random samples drawn from the training data.
- we can see that simple model have a high bias. Because the average function is far away from the true function.
- Complex models have low bias. They fit the data perfectly.



Variance

- Variance is the measure of spread in data from its mean position.
- In machine learning variance is the amount by which the performance of a predictive model changes when it is trained on different subsets of the training data.
- More specifically, variance is the variability of the model that how much it is sensitive to another subset of the training dataset.
- i.e. how much it can adjust on the new subset of the training dataset.

Variance

- Let Y be the actual values of the target variable
- Y^ be the predicted values of the target variable.
- Then the variance of a model can be measured as the expected value of the square of the difference between predicted values and the expected value of the predicted values.

- Variance = E[(Y^ - E[Y^]) ^2]

- where E[Y^] is the expected value of the predicted values.
- Here expected value is averaged over all the training data.
- So variance tells how f̂(x) differs from the expected value of the model E(f̂(x)).

Bias and Variance

- So, for complex models, variance tends to be higher because a small change in the training sample will lead to different $\hat{f}(x)$.
- Because complex models, memorize the data points.
- For simple models, there will not be much difference in $\hat{f}(x)$, if we change the training sample a little.
- Simple models generalize the pattern.

Bias vs variance

- Simple models may have high bias and low variance -> underfitting
- Complex models may have low bias and high variance -> overfitting
- Best fit models will have low bias and low variance.
- There is a trade-off between bias and variance because both contribute to error.

- Expected Prediction Error: Expected Prediction Error depends on three errors

- Bias
- Variance
- Noise (Irreducible Error)

Bias and Variance

- The prediction error is high when bias is high and when variance is high.
- degree 1 polynomial → training error and the prediction error is high → Underfitting
- degree 2 polynomial → training error and prediction error high →Underfitting
- degree 5 polynomial → training error is less and the difference between training error and the prediction error is less. → Best fit
- degree 20 polynomial → training error is less and prediction error is very high →Overfitting

Bias- variance trade-off

- Bias- variance trade-off is needed for the following scenarios.

- To overcome underfitting and overfitting condition
- To have consistencies in predictions.

Summary: Bias and Variance

- Bias and variance measure two different sources of error in an estimator
- Bias measures the expected deviation from the true value of the function or parameter.
- Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

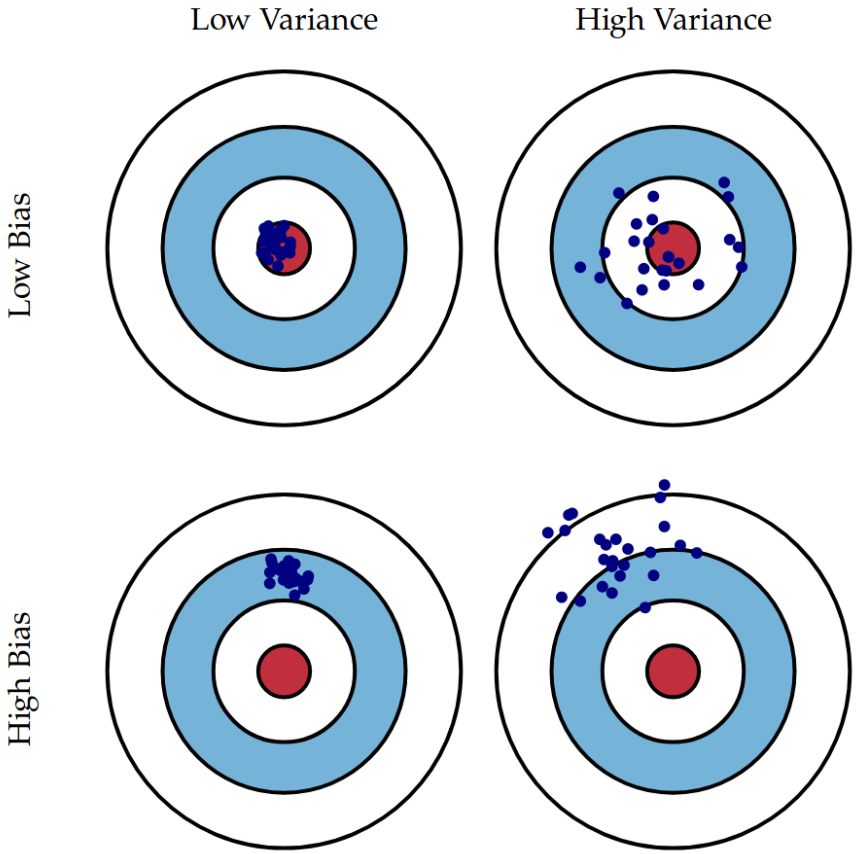
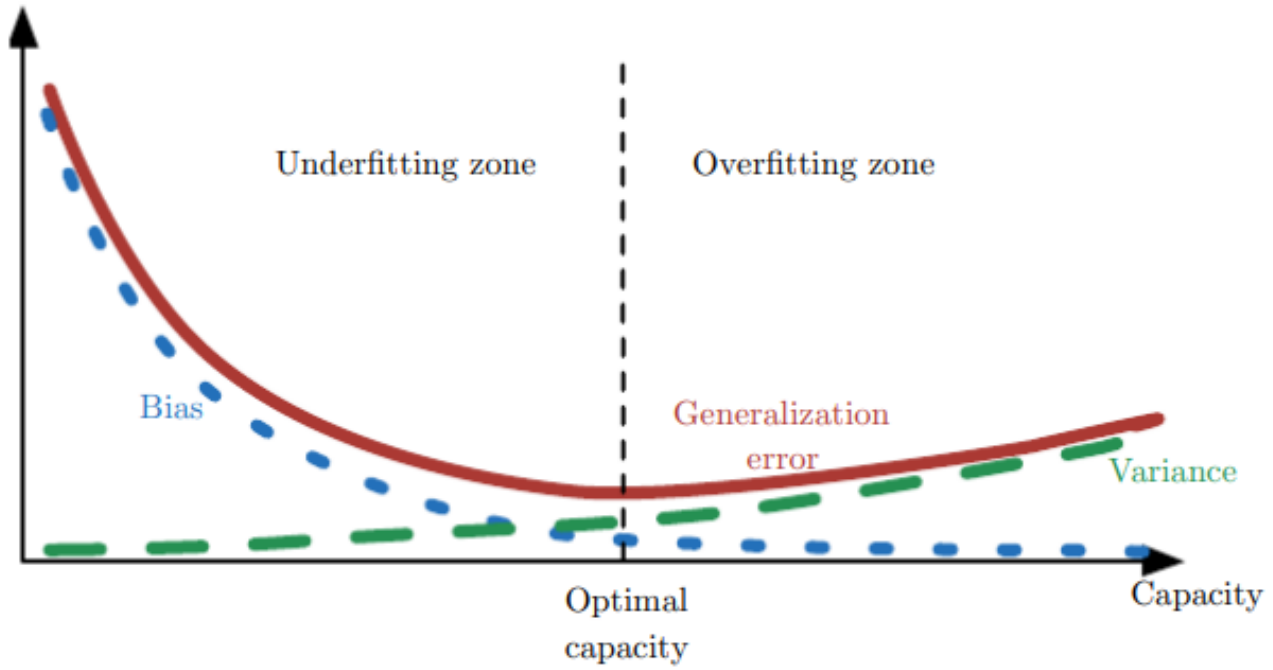


Fig. 1 Graphical illustration of bias and variance.

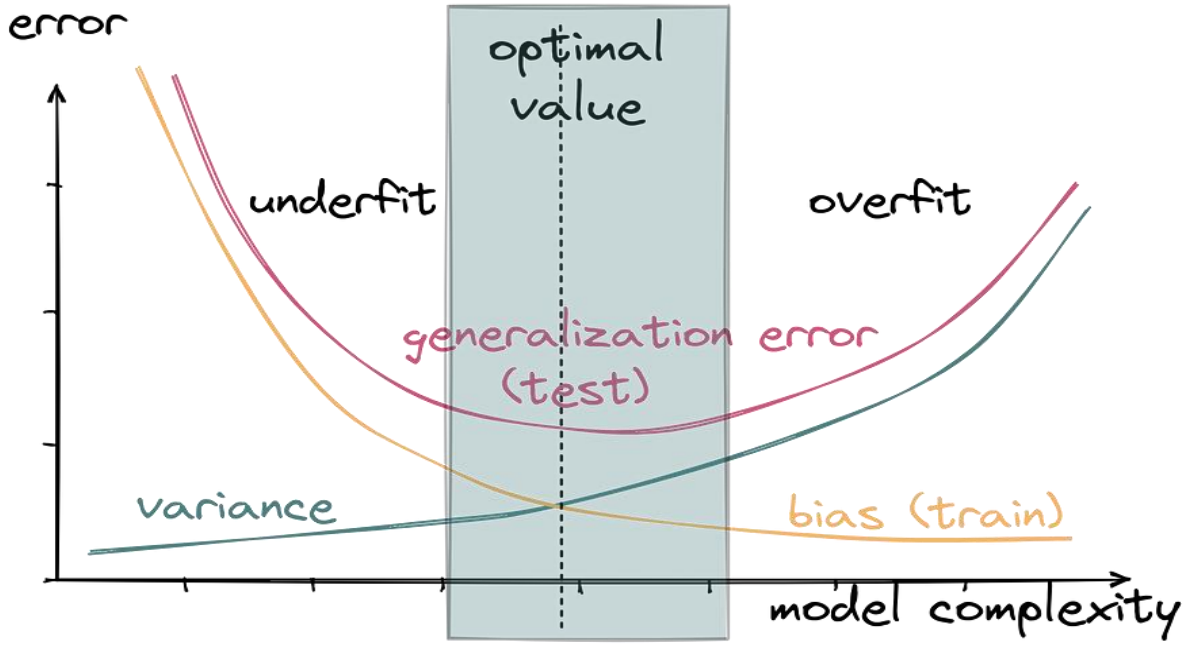
Note: The central point represents the target.
So red circle is y=f(x)
Blue circle is f(x)^

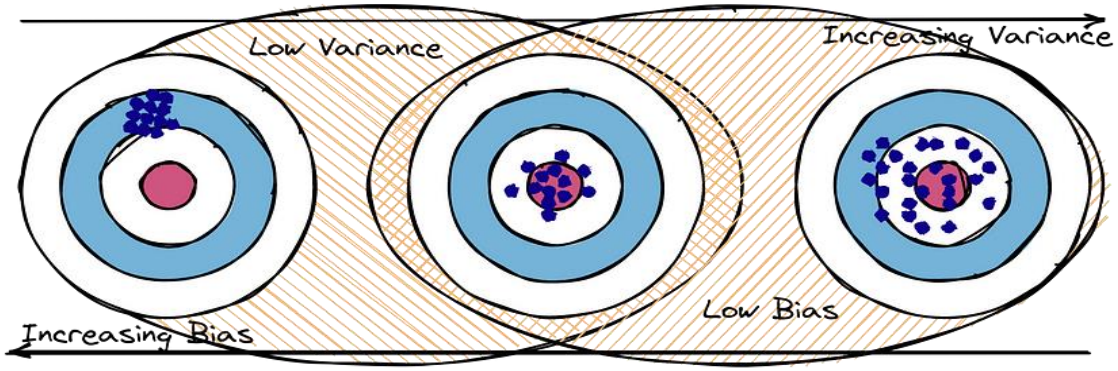
Summary : Bias and Variance

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting



Summary : Bias, Variance and Error





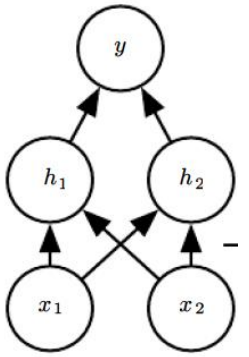
Dropout

Dropout

- Dense layers often have a lot of parameters.
- Ex: If a first hidden layer has 784 × 300 connection weights, plus 300 bias terms, then we have 235,500 parameters!
- This gives the model quite a lot of flexibility to fit the training data
- but it also means that the model runs the risk of overfitting, especially when we do not have a lot of training data.
- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models
- Bagging involves training multiple models, and evaluating multiple models on each test example.
- This seems impractical when each model is a large neural network
- since training and evaluating such networks is costly in terms of runtime and memory.
- It is common to use ensembles of five to ten neural networks—Szegedy et al. (2014a) used six to win the ILSVRC—but more than this rapidly becomes unwieldy.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Dropout

- Dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network (dropout is applied to non-output units)
- In most modern neural networks, based on a series of transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero
- This does not mean that we physically remove those neurons. We can make it act as those are removed by setting the output of those dropout neuron to be zero
- the neurons which are "dropped out" in this way do not contribute to the forward pass and do not participate in back propagation. So every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.

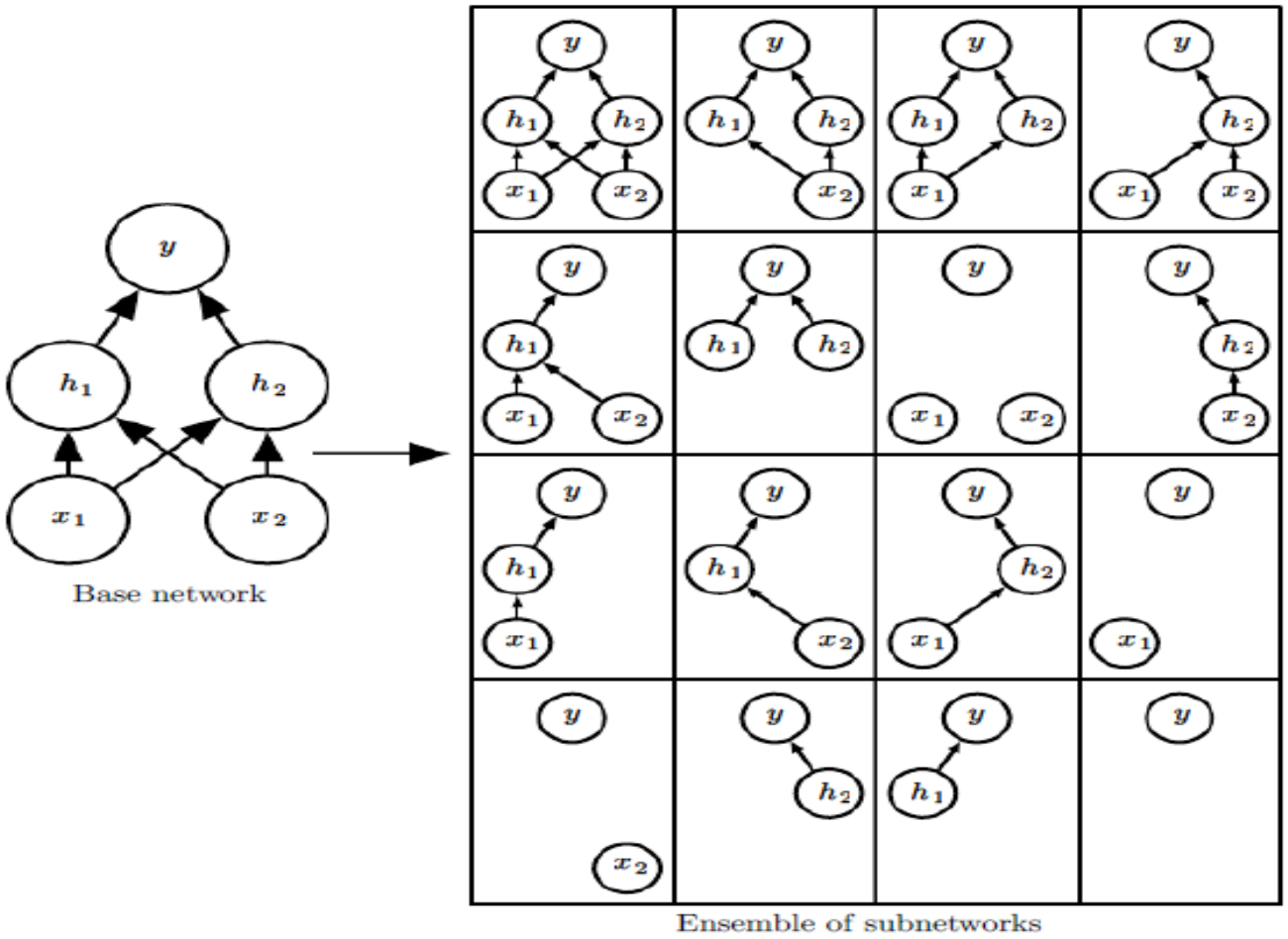


Base network

Ensemble of subnetworks

- Dropout trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network.
- Here, we begin with a base network with two visible units and two hidden units.
- There are sixteen possible subsets of these four units.
- Show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network.
- In this small example, a large proportion of the resulting networks have no input units or no path connecting the input to the output.
- This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.

Ensemble of subnetworks



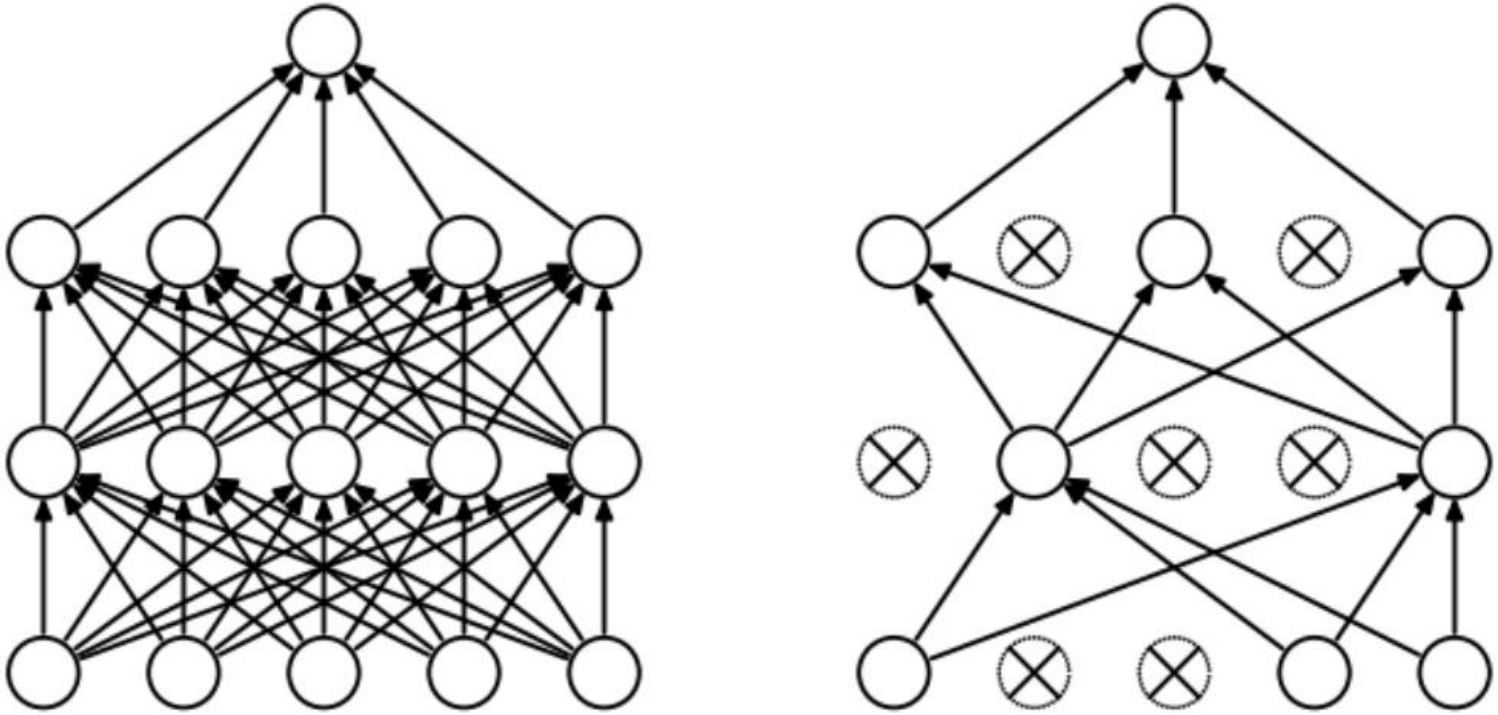
Ensemble of subnetworks

- In bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i
- Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent.
- In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network.
- This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.
- In the case of bagging, each model is trained to convergence on its respective training set.
- In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible subnetworks within the lifetime of the universe.
- Instead, a tiny fraction of the possible sub-networks are each trained for a single step, and the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters.
- dropout follows the bagging algorithm.
- Ex: the training set encountered by each sub-network is indeed a subset of the original training set sampled with replacement.

Dropout

- Dropout is a regularization technique for neural networks
- Dropout is a regularization technique for neural networks that drops a unit (along with connections) at training time with a specified probability (a common value is 0.5) using samples from a Bernoulli distribution during training.
- At test time, all units are present, but with weights scaled by p (i.e. w becomes w*p).
- The idea is to prevent co-adaptation
- where the neural network becomes too reliant on particular connections
- This could be symptomatic of overfitting.
- Intuitively, dropout can be thought of as creating an implicit ensemble of neural networks.

Dropout

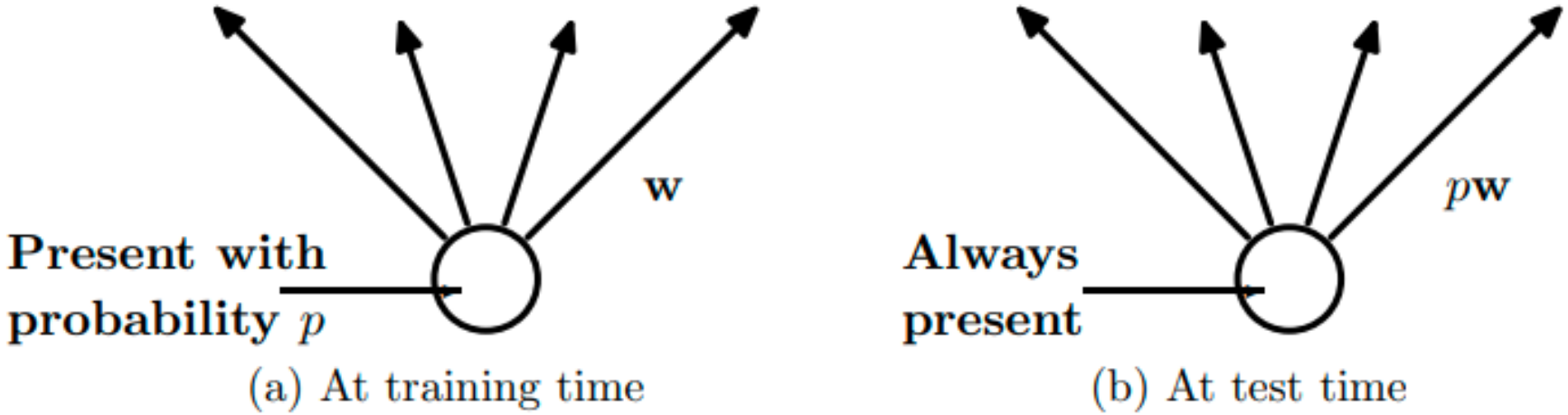


(a) Standard Neural Net

(b) After applying dropout.

- During training of a neural network model, it will take the output from its previous layer, randomly select some of the neurons and zero them out before passing to the next layer, effectively ignored them.
- This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass.
- When the model is used for inference, dropout layer is just to scale all the neurons constantly to compensate the effect of dropping out during training.

Dropout



(a) At training time (b) At test time

(a) A unit at training time that is present with probability p and is connected to units
in the next layer with weights w.
(b): At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Pytorch- Dropout

- **torch.nn.Dropout(*p: float = 0.5, inplace: bool = False*)**

During training, it randomly zeroes some of the elements of the input tensor with probability **p**

The zeroed elements are chosen independently for each forward call and are sampled from a Bernoulli distribution.

Furthermore, the outputs are scaled by a factor of $1/(1-p)$ during training. This means that during evaluation the module simply computes an identity function.

- Input: $(*)(*)$. Input can be of any shape
- Output: $(*)(*)$. Output is of the same shape as input

Note:An identity function is a mathematical function that returns its input unchanged. For any input value x, the output of the identity function is equal to x. The notation for the identity function is represented as f(x) = x. During evaluation/test/inference time, the dropout layer becomes an identity function and makes no change to its input.

Pytorch- Dropout

- nn.Dropout()
- dropout_layer = nn.Dropout(p=0.1)
- randomly zeroes some of the elements of the input tensor based on the given probability, p using samples from a Bernoulli distribution.
- When this occurs, a portion of the output will be lost on every forward call
- To account for this, the outputs are also scaled by a factor of $1/_{(1\_p)}$
- Because dropout is active only during training time but not inference time
- without the scaling, the expected output would be larger during inference time because the elements are not being randomly chosen to be dropped (set to 0).
- But we want the expected output with and without going through the dropout layer to be the same.
- Therefore, during training, we compensate by making the output of the dropout layer larger by the scaling factor of 1/(1-p).
- Note: The scaling makes the input mean and output mean roughly equivalent.

Using nn.Dropout()

```
import torch
import torch.nn as nn

# generate 100 ones
x = torch.ones(100)
print("x=", x)
dropout_layer = nn.Dropout(p=0.1)
output = dropout_layer(x)
print("output=", output)
```

Intuition - scaling

- With a dropout rate of p = 0.1, approximately 10 of the values should be 0.
- The scale rate = 1/(1-0.1)= 1/0.9 =1.1.
- this is the value that each output should be.
- ten of the values are zeroed-out completely,
- and the result are scaled to ensure the input and output have the same mean — or as close to it as possible.

Using nn.Dropout()- Output

```
import torch
import torch.nn as nn


# generate 100 ones
x = torch.ones(100)
print("x=", x)
dropout_layer = nn.Dropout(p=0.1)
output = dropout_layer(x)
print("output=", output)
print(x.mean(), output.mean())
```

```
x= tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
output= tensor([1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000, 0.0000, 1.1111,
        1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 0.0000, 0.0000, 1.1111, 1.1111,
        0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 0.0000, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111])
tensor(1.) tensor(0.9556)
```

Using torch.Bernoulli – Implement nn.Dropout

- dropout_mask = torch.bernoulli(torch.full_like(x, 0.9))
- Here, use torch.bernoulli to generate a binary dropout mask.
- torch.full_like(x, 0.9) creates a tensor of the same shape as x filled with the value 0.9
- Representing the probability of keeping each element.
- torch.bernoulli then converts these probabilities into a binary mask, where each element has a 0.9 probability of being 1 (kept) and a 0.1 probability of being 0 (dropped).
- output = x * dropout_mask / 0.9: We apply the dropout mask to the original tensor x.
- The division by 0.9 is a scaling factor to compensate for the fact that during training, we drop elements with a certain probability (1 - p).
- By dividing the output by 1 - p, we ensure that the expected value of the tensor remains the same during training and testing.

Using torch.Bernoulli – Implement nn.Dropout

```
import torch
# Generate 100 ones
x = torch.ones(100)
print("x=", x)

# Generate a dropout mask using torch.bernoulli with a probability of 0.1
dropout_mask = torch.bernoulli(torch.full_like(x, 0.9))  # p = 0.9 for keeping values
#dropout_mask = torch.bernoulli(torch.ones_like(x) * 0.9)

# Apply dropout scaling
output = x * dropout_mask / 0.9  # Scale by 1/p
print("output=", output)
print(x.mean(), output.mean())
```

Using torch.Bernoulli - Output

```
x= tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
output= tensor([1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000, 1.1111, 1.1111,
        0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000,
        0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 0.0000,
        1.1111, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        0.0000, 1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        1.1111, 0.0000, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111, 1.1111,
        0.0000])
tensor(1.) tensor(0.9778)
```

Dropout - Summary

- In PyTorch, while we call model.eval() on our model, it switches the model to evaluation mode, which disables dropout.
- So it is crucial to set the model to evaluation mode before making predictions during testing.
- By calling eval() on the models, dropout will be turned off, and
- the models will be used in their entirety for making predictions during testing.
- This ensures that the dropout behavior is consistent with what was used during training.

Data Augmentation

Data Augmentation

- Data preprocessing
- Deep learning model's success largely depends on the quality of the data that we feed into it, since it determines the performance and generalizations of a model.
- In most cases, raw data is rarely in the form that we need
- Data augmentation
- In practice, the amount of data we have is limited.
- The best way to make a machine learning model generalize better is to train it on more data.
- One way to get around this problem is to create fake data and add it to the training set.
- So transforms provide the opportunity for two helpful functions:

1. Data preprocessing: allows to transform data into a suitable format for training
2. Data augmentation: allows to generate new training examples by applying various transformations on existing data

Data Augmentation – Applicability

- Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition.
  - Images are high dimensional and include an enormous variety of factors of variation, many of which can be easily simulated.
- Not effective to apply transformations that would change the correct class.
  - Ex: optical character recognition tasks require recognizing the difference between ‘b’ and ‘d’ and the difference between ‘6’ and ‘9’,
  - so horizontal flips and 180◦ rotations are not appropriate ways of augmenting datasets for these tasks.
- Note: Augmentation can be applied to audio, video, text, image

Image Augmentation

- Geometric transformations: randomly flip, crop, rotate, stretch, and zoom images.
- Color space transformations: randomly change RGB color channels, contrast, and brightness.
- Kernel filters: randomly change the sharpness or blurring of the image.
- Random erasing: delete some part of the initial image.
- Mixing images: blending and mixing multiple images.
- Note: There are over 30 different augmentations available in the torchvision.transforms module

torchvision.transforms.Compose

- Image transformation is available in the torchvision
- torchvision supports common computer vision transformations in the

1. torchvision.transforms and
2. torchvision.transforms.v2 modules.

- Transforms can be used to transform or augment data for training or inference of different tasks (image classification, detection, segmentation, video classification).
- A standard way to use these transformations is in conjunction with torchvision.transforms.Compose, which allows to stack multiple transformations sequentially

Image Processing - PIL

- PIL - Python Image Library - open() and show() function to read and display the image

```
from PIL import Image
img = Image.open('./data/cat.3.jpg')
img.show()
```

- writes the image to a temporary file and the temporary file will be deleted once the program execution is completed.
- Also show() will block the Execution Environment until we close the image.
- Pillow first converts the image to a .png format (on Windows OS) and stores it in a temporary buffer and then displays it.
- Due to the conversion of the image format to .png some properties of the original image file format might be lost (like animation).
- So use show() method only for test purposes

```
from IPython.display import display
```
– display() function to display image

```
display(img)  //displays on the console output
```

- Note: PIL has been discontinued from 2011
- Python Pillow is built on the top of PIL (Python Image Library) and is considered as the fork for the same (fork means that we still need to use PIL)

Image Processing - PIL

Image class attributes .format, .size, and .mode – img.format, img.size, img.mode, img.width, img.height

- The format of an image shows the type of image such as 'JPEG’.
- The size shows the width and height of the image in pixels.
- The mode could be 'RGB’.

```
import torchvision
from torchvision import transforms
from PIL import Image
# Loading a Sample Image
img = Image.open('./data/cat.3.jpg')
img.show()
print(img.size)
print(img.format)
print(img.mode)
```

```
(500, 414)
JPEG
RGB
```

Converting Images to Tensors-.ToTensor()

- Converting Images to Tensors
- By using the transforms.ToTensor() transformation, we are able to easily convert data (such as images) to tensors.
- Tensors provide many different benefits:
- Seamless Integration: Deep learning models, especially those built using PyTorch, expect input data in tensor format. Converting data to tensors enables smooth integration into the model.
- Efficient Computations: Tensors allow for efficient mathematical operations and computations required during model training and inference.
- Automatic Differentiation: Tensors in PyTorch enable automatic differentiation, a fundamental concept for training neural networks using gradient-based optimization algorithms.

Converting Images to Tensors-.ToTensor()

```
import torchvision
from torchvision import transforms
from PIL import Image
from IPython.display import display
img = Image.open('./data/cat.3.jpg')
display(img)
#img.show()
tensor_transform = transforms.ToTensor()
tensor_image = tensor_transform(img)
print(tensor_image)
print(tensor_image.shape)
```

```
tensor([[[0.6275, 0.6431, 0.6667,  ..., 0.7255, 0.7216, 0.7176],
         [0.4941, 0.5137, 0.5373,  ..., 0.7216, 0.7176, 0.7137],
         [0.4980, 0.5137, 0.5412,  ..., 0.7176, 0.7137, 0.7098],
         ...,
         [0.7765, 0.7569, 0.6980,  ..., 0.8078, 0.7647, 0.6824],
         [0.6902, 0.7176, 0.6863,  ..., 0.7373, 0.5647, 0.4902],
         [0.7451, 0.7451, 0.6157,  ..., 0.7843, 0.5686, 0.5647]],

        [[0.5922, 0.6078, 0.6275,  ..., 0.6745, 0.6706, 0.6667],
         [0.4588, 0.4824, 0.4980,  ..., 0.6706, 0.6667, 0.6627],
         [0.4667, 0.4824, 0.5020,  ..., 0.6627, 0.6588, 0.6549],
         ...,
         [0.7294, 0.7098, 0.6510,  ..., 0.8000, 0.7569, 0.6745],
         [0.6431, 0.6706, 0.6392,  ..., 0.7333, 0.5608, 0.4863],
         [0.6980, 0.6980, 0.5686,  ..., 0.7765, 0.5608, 0.5569]],

        [[0.5569, 0.5725, 0.5922,  ..., 0.6118, 0.6078, 0.6039],
         [0.4235, 0.4392, 0.4627,  ..., 0.6078, 0.6039, 0.6000],
         [0.4235, 0.4314, 0.4627,  ..., 0.6118, 0.6078, 0.6039],
         ...,
         [0.6353, 0.6157, 0.5569,  ..., 0.7098, 0.6667, 0.5843],
         [0.5490, 0.5765, 0.5451,  ..., 0.6627, 0.4902, 0.4157],
         [0.6039, 0.6039, 0.4745,  ..., 0.7176, 0.5020, 0.4980]]])
torch.Size([3, 414, 500])
```

Normalization - .Normalize

- Normalization allows to ensure that input features are scaled and centered consistently, which leads to better convergence during training.
- Allows to standardize input data, while ensuring that the underlying data distribution remains intact with following benefits.
- Stable Training: Normalized data can lead to more stable and faster convergence during training, as it mitigates the issue of varying scales among input features.
- Reduced Sensitivity: Neural networks are less sensitive to input features that vary within different scales, leading to improved generalization.
- Numerical Stability: Normalized data can prevent numerical instability issues that might arise during computations involving large or small numbers.

Normalization - .Normalize()

```
import torchvision
from torchvision import transforms
from PIL import Image
from IPython.display import display
# Normalizing an Image with PyTorch transforms
normalize_transform = transforms.Normalize(
    mean=[0.5, 0.5, 0.5],
    std=[0.3, 0.3, 0.3]
)
img = Image.open('./data/cat.3.jpg')
#img.show()
display(img)
normalized_image = normalize_transform(tensor_image)
# Transform the image to a PIL Image
normalized_image_show = transforms.ToPILImage()(normalized_image)
display(normalized_image_show)
```

.Resize()

- transforms.Resize(). allows to pass in a tuple containing the size to resize.

```
import torchvision
from torchvision import transforms
from PIL import Image
from IPython.display import display
# Loading a Sample Image
img = Image.open('./data/cat.3.jpg')
display(img)
resize_transform = transforms.Resize((150, 150))
resized_image = resize_transform(img)
display(resized_image)
```

.CenterCrop()

```
import torchvision
from torchvision import transforms
from PIL import Image
from IPython.display import display
img = Image.open('./data/cat.3.jpg')
display(img)
# define an transform, height=180 width=300
transform = transforms.CenterCrop((180, 300))
```

.CenterCrop()

- CenterCrop() method
- Accepts images like PIL Image, Tensor Image, and a batch of Tensor images.
- The tensor image is a PyTorch tensor with [C, H, W] shape
- Here C represents a number of channels and H, W represents height and width respectively.
- Transforming the image at the center, to get a square image as output.
- transform = transforms.CenterCrop(200)
- Transforming the image with a height of 180 and a width of 300 pixels
- transform = transforms.CenterCrop((180, 300))

.Compose()

```
import torchvision.transforms as T
from PIL import Image
img = Image.open('./data/cat.3.jpg')
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
      mean=[0.485, 0.456, 0.406],
      std=[0.229, 0.224, 0.225])
])
x = transform(img)
img.show() # x.show() -'Tensor' object has no attribute 'show'
print(x.shape)
```

Output - torch.Size([3, 224, 224])

```
'''
transform= T.Compose([
  T.Resize(256),
  T.CenterCrop(224),
  T.ToTensor(),
  T.Normalize(
     mean=[0.485, 0.456,
0.406],
     std=[0.229, 0.224,
0.225] )
])
'''
```

Data Augmentation Steps

1.Load the Dataset:

- Load training dataset, typically divided into images and corresponding labels.

2.Define Transformation Pipeline:

- Create a set of transformations that will be applied to the input data. Common image data augmentation transformations include:
  - Random flips (horizontal or vertical)
  - Random rotations
  - Random crops and resizing
  - Changes in brightness, contrast, and saturation
  - Gaussian noise addition
- Use a library like **torchvision.transforms** in PyTorch to define the transformations.

3.Apply Transformations:

- Iterate through the training dataset and apply the defined transformations to each input sample.
- For each training example, generate multiple augmented samples by applying different combinations of transformations.

4.Model Training:

- Train machine learning model using the augmented dataset.
- The augmented samples contribute to the model's ability to generalize well to unseen data.

Data Augmentation Steps

5. **Validation and Testing:**

- For validation and testing, use the original, non-augmented data to evaluate the model's performance.

6. Monitoring:

- Monitor the training process, including loss and accuracy, to ensure the model is learning effectively.

7. Repeat:

- Repeat the training process with the augmented data until the model converges or reaches satisfactory performance.

8. Evaluate on Unseen Data:

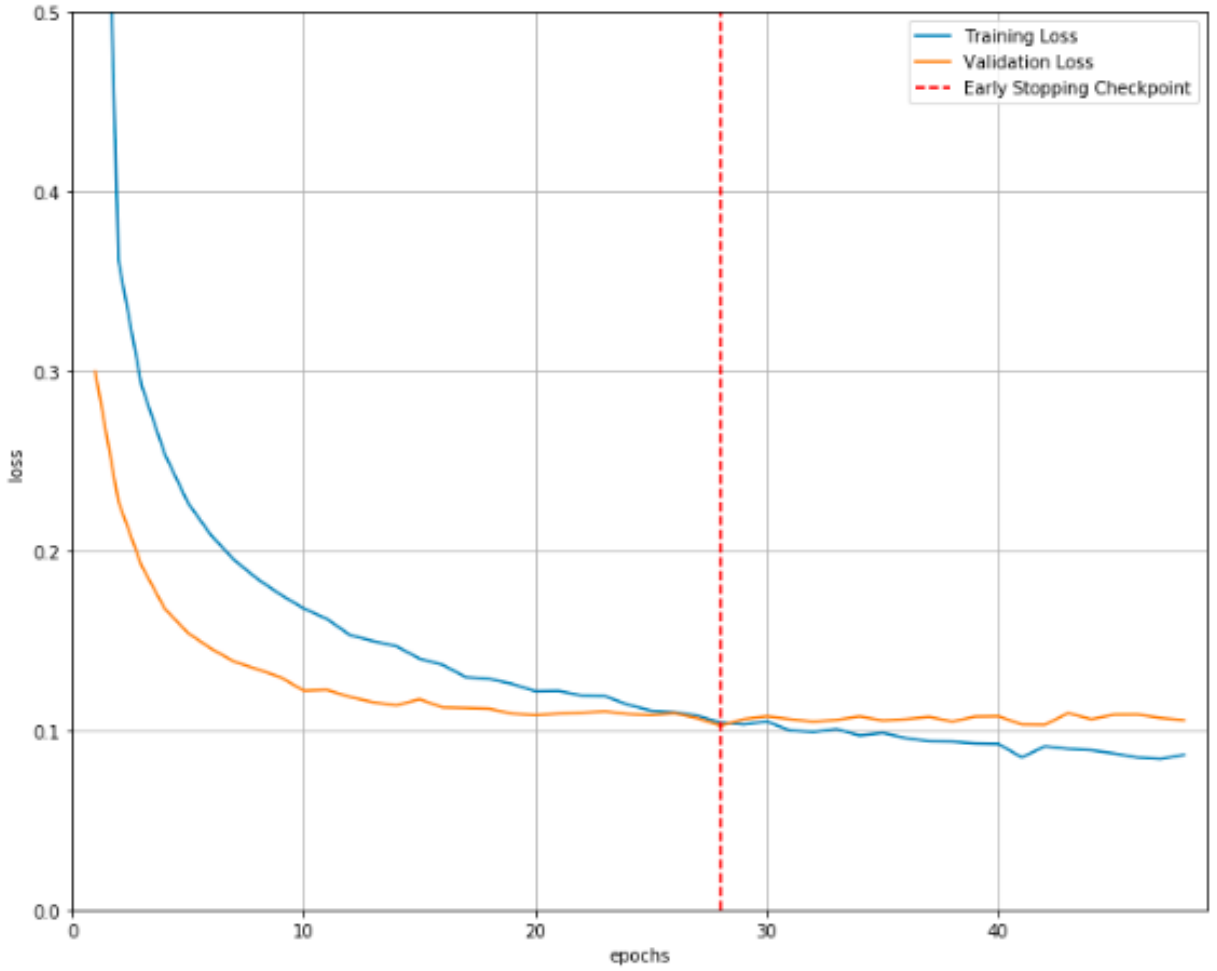
- Finally, evaluate the trained model on completely unseen data (testing set) to assess its generalization performance.

Early Stopping

Early Stopping

- A problem with training neural networks is in the choice of the number of training epochs to use.
- Too many epochs can lead to overfitting of the training dataset
- too few may result in an underfit model.
- Early stopping is a method that allows to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

Early Stopping



Training loss – General strategy

# to track the training loss as the model trains
train_losses = []

```
for epoch in range(1, n_epochs + 1):

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for batch, (data, target) in enumerate(train_loader, 1):
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # record training loss
        train_losses.append(loss.item())
```

Validation loss- general strategy

# to track the validation loss as the model trains
valid_losses = []

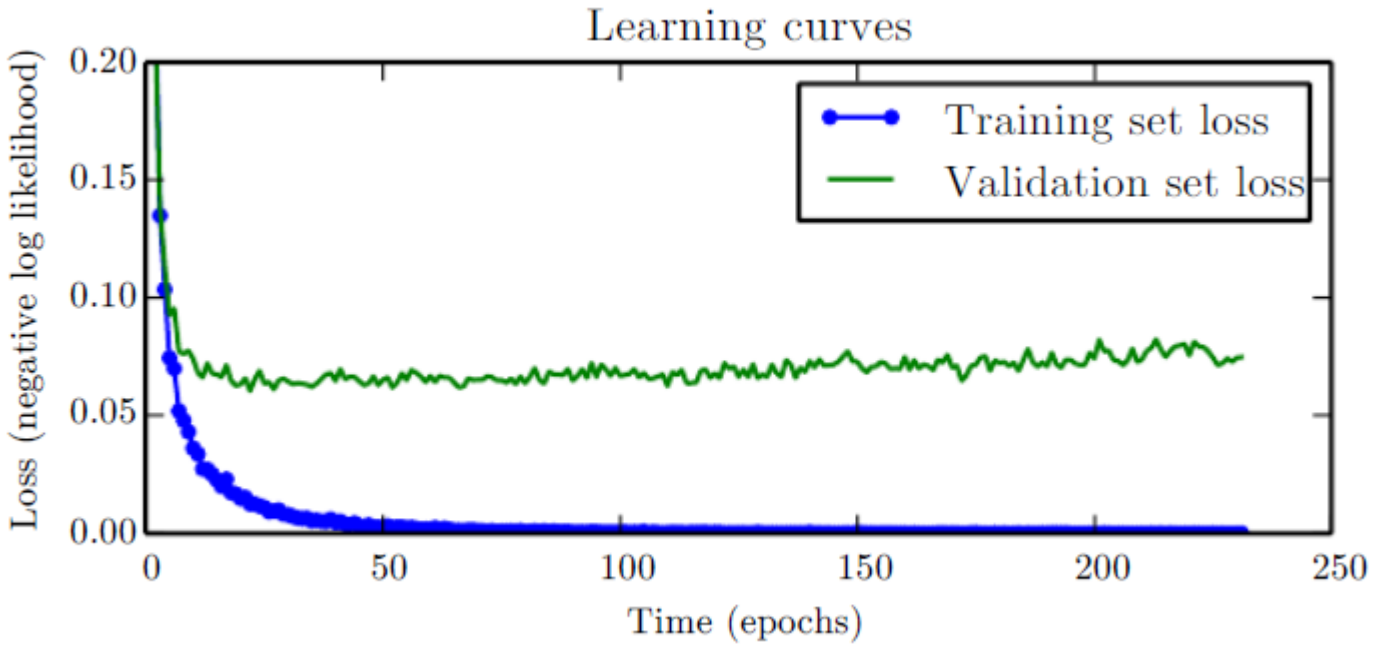
```
######################
# validate the model #
######################
model.eval() # prep model for evaluation
for data, target in valid_loader:
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the loss
    loss = criterion(output, target)
    # record validation loss
    valid_losses.append(loss.item())
```

Early Stopping

- When training large models with sufficient representational capacity to overfit the task, we often observe 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
- Instead of running our optimization algorithm until we reach a (local) minimum of validation error, we run it until the error on the validation set has not improved for some amount of time

Early Stopping

- Every time the error on the validation set improves, we store a copy of the model parameters
- When the training algorithm terminates, we return these parameters, rather than the latest parameters.



Early Stopping – General strategy

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define your neural network model
class YourModel(nn.Module):
    # ... model architecture ...

# Initialize your model, loss function, and optimizer
model = YourModel()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Early Stopping - Patience

- Patience
- To monitor the validation loss during training and
- stop training early if the validation loss does not improve for a certain number of consecutive iterations (patience).
- Ex: if we set patience=5, the training process will stop if the validation metric does not improve for five consecutive epochs.
- "patience" refers to the number of consecutive epochs with no improvement on the chosen metric (e.g., validation loss or accuracy) before the training is halted

Early Stopping – General strategy

```
# Define early stopping parameters
patience = 5
best_validation_loss = float('inf')
current_patience = 0
# Training loop
for epoch in range(num_epochs):
    # Training steps ...

    # Validation steps
    model.eval()
```

Early Stopping – General strategy

```
with torch.no_grad():
    validation_loss = 0.0
    for inputs, labels in validation_dataloader:
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        validation_loss += loss.item()

    # Average validation loss
    validation_loss /= len(validation_dataloader)
```

Early Stopping – General strategy

```
# Check for improvement in validation loss: condition true means there is an improvement
        if validation_loss < best_validation_loss:
            best_validation_loss = validation_loss
            current_patience = 0
            # Save the model if desired
            torch.save(model.state_dict(), 'best_model.pth')
        else:
            current_patience += 1

        # Check if early stopping criteria are met
        if current_patience > patience:
            print("Early stopping! No improvement for {} epochs.".format(patience))
            break
    # Continue with the next epoch
```

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

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