inzva DLSG

Regularization and Hyperparameter Tuning

Week 2

Regularization

Regularization, in the context of machine learning and deep learning, is a collection of techniques used to prevent a model from overfitting the training data. Overfitting occurs when a model learns to fit the training data too closely, capturing noise and irrelevant patterns that do not generalize well to unseen data. Regularization techniques play a crucial role in improving the generalization performance of machine learning and deep learning models by controlling their complexity and preventing overfitting. Now, let's discuss how to identify overfitting and when to use regularization.

Performance Measuring

Effective Data Splitting in Machine Learning: Train, Validation, and Test Sets

Data splitting is a crucial step in the development of machine learning models. Properly partitioning the data into training, validation, and test sets ensures that the model is trained on a diverse range of examples, validated on unseen data to prevent overfitting, and rigorously tested to assess robustness and accuracy. This process not only aids in hyperparameter tuning and model selection but also builds trust in the model's predictions through a thorough evaluation.

Evolution of Data Splitting

In the early days of machine learning, computational power and data availability were limited. It was common practice to divide the dataset into two segments: a training set and a test set. Typical ratios for this split were 70/30% or 80/20%, where the larger portion was used for training and the smaller for testing.

With the exponential growth in data and computational resources, more sophisticated splitting strategies have become feasible. For instance, it is now common to see splits where a larger percentage of data, sometimes as high as 99%, is allocated to the training set, assuming a corresponding increase in total data volume. This approach allows for extensive training without sacrificing the representativeness of the test set.

Today's vast data and computational power enable us to use an even more robust splitting strategy: train/validation/test sets. By having another set for the validation of the models, we can use the validation set (also called the development set or dev set) to measure the performance of the models and improve them. After tuning, the untouched test set provides a final, unbiased evaluation metric. The advantage of this split is that we can get a more accurate measure of the model's performance since the test set remains unseen until the final assessment and the model does not overfit to the test data. However, if only the train and test sets were used, we would use the test set to fine-tune the model and that would cause the model to exploit the test data for fine-tuning, possibly introducing information not representative of real-world data and some overfitting.

In short, the modern standard involves splitting the dataset into three parts: the training set, the validation set, and the test set. This method provides several advantages over the traditional train/test split:

- Training Set: Used to train the model, adjusting weights and features to fit the data.
- Validation Set: Also known as the development set, it is used to tune hyperparameters and make decisions about model adjustments without impacting the test set. This helps avoid overfitting the model to the test data.
- Test Set: Serves as an untouched dataset used only for the final model evaluation. This ensures that the performance metrics are unbiased and representative of real-world performance.

Benefits of Using a Validation Set

Using a validation set allows for:

- Better hyperparameter tuning without compromising the test set.
- Early stopping to prevent overfitting if the model performance on the validation set starts to degrade.
- Selecting the best model after comparing different architectures or configurations based on their performance on the validation set.

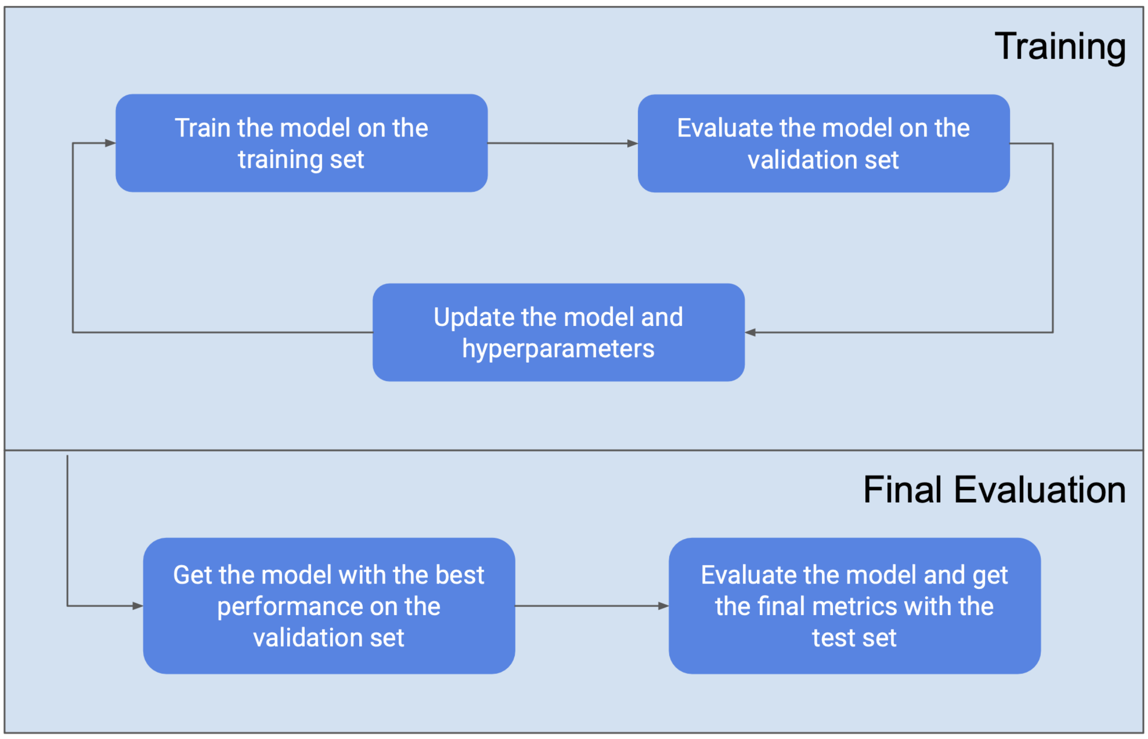
Training
Train the model on the training set
Evaluate the model on the validation set
Update the model and hyperparameters
Final Evaluation
Get the model with the best performance on the validation set
Evaluate the model and get the final metrics with the test set

Figure 1: Training - evaluation - tuning cycle.

Bias/Variance

Bias refers to the error introduced in a model due to overly simplistic assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). For instance, a linear regression model would have high bias when trying to capture non-linear relationships because it assumes that the relationship between variables is linear.

Variance refers to the error introduced in a model due to excessive sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

Underfitting and Overfitting

Underfitting and overfitting are two fundamental issues in both machine learning and deep learning, and understanding them is crucial for developing models that perform well on new, unseen data. Let's explore both concepts, their implications, and their similarities across the fields of machine learning and deep learning.

Underfitting

Underfitting occurs when a model is too simple to learn the underlying pattern of the data. This typically happens if the model does not have enough parameters (underparameterized) or if it is trained with too few features. An underfitted model performs poorly on the training data and lacks the capacity to generalize well on new data.

Characteristics of Underfitting:

- High bias and low variance: The model makes strong assumptions about the shape of the data and fails to capture the underlying trends.
- It performs poorly on both training data and unseen test data because it cannot model the data complexity.
- Simplistic model: Often, linear models can underfit complex datasets that require non-linear solutions. (in the context of machine learning)

Causes of Underfitting:

- Insufficient Model Complexity: The model does not have enough parameters or layers (in the context of neural networks) to capture the complex patterns in the data.
- Poor Feature Selection: Using features that do not capture the important aspects of the data can lead the model to ignore significant patterns. (This can be restricted to the context of machine learning)
- Too Much Regularization: While regularization is used to prevent overfitting, setting it too high can overly simplify the model, leading to underfitting.

Overfitting

Overfitting occurs when a model is too complex relative to the amount and noisiness of the training data. This complexity allows the model to fit not only the underlying pattern but also the random noise in the data. As a result, while the model performs exceptionally well on the training data, it performs poorly on new, unseen data.

Characteristics of Overfitting:

- High variance: The model's performance varies significantly with small fluctuations in the training data.
- Good performance on training data but poor generalization to new data.
- Complex model: Often, non-linear models with many parameters can overfit, especially if the training data is sparse and noisy.

Causes of Overfitting:

- Excessive Model Complexity: A model with too many parameters (like a deep neural network with many layers) can capture intricate patterns as well as noise.
- Insufficient Training Data: Without enough data, a complex model may learn noise instead of the underlying pattern.
- Lack of Regularization: Regularization techniques help to prevent overfitting by penalizing overly complex models, and not using these can lead to overfitting.

Regularization Methods

Early Stopping

One method of regularization is the early stopping technique. If the model overfits the data and causes high variance, we can stop the training early to prevent overfitting. For early stopping, a threshold is designated, and if the training loss doesn't decrease by more than the threshold, or if the test loss increases by more than the threshold at any step, the training is stopped.

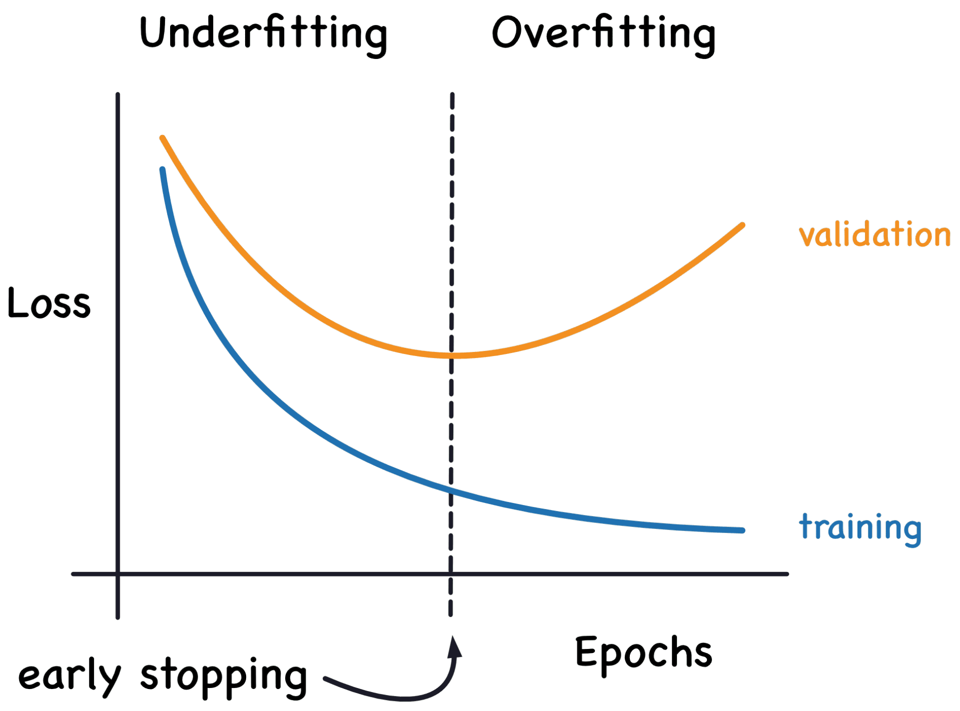

Figure 2: Early Stopping with Training & Validation Losses [3]

Data Augmentation

Data augmentation is another regularization technique that is widely used to prevent overfitting. By using data augmentation, we can address the problem of insufficient training data, as the augmented data can introduce variance that the original data may lack. This technique helps the model learn from both the augmented and original data.
Different augmentation techniques can be applied depending on the data type. For example, for image data, we commonly use the following techniques for data augmentation:

- Flipping
- Rotation
- Cropping
- Distortion

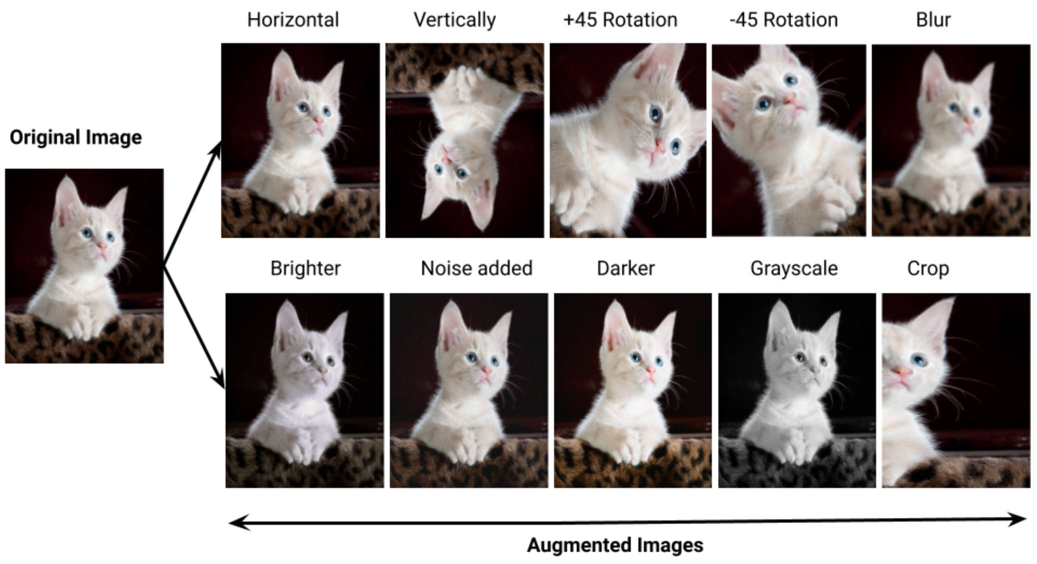

Figure 3: Data Augmentation [4]

Data augmentation should be used carefully, and augmented data should not be included in the test set, as the final evaluation results from the test set should reflect real-world performance. The distribution of the test set should match that of real-world data, which does not include augmented data.

Although data augmentation can help with the overfitting problem, it may also introduce disadvantages if used incorrectly. Excessive data augmentation might introduce noise, resulting in decreased test accuracy. Additionally, overusing augmentation can increase computational costs and substantially extend the model's training time.

L1 and L2 Regularization

L1 and L2 regularization techniques are extensively used to prevent overfitting. In these techniques, an additional term is added to the loss function. This term penalizes overfitting and forces the model parameters to become smaller.
The L1 loss has the following formula:

$$\text{L1 loss term} = \lambda \sum_i |w_i|, \text{ and Total loss} = \text{Original loss} + \lambda \sum_i |w_i|,$$

where w_i represents each weight in the model, and λ is the regularization strength, a hyperparameter that controls the amount of regularization applied.
Similarly, the L2 loss is given by the formula:

$$\text{L2 loss term} = \frac{\lambda}{2}\sum_i w_i^2, \text{ and Total loss} = \text{Original loss} + \frac{\lambda}{2}\sum_i w_i^2,$$

L1 regularization forces some parameters to approach zero, resulting in sparsity among neurons. Therefore, L1 regularization acts similarly to dropout regularization for neural networks and prevents overfitting by shrinking the network. On the other hand, L2 regularization discourages large weights but doesn't encourage sparsity like L1. Instead, L2 shrinks these coefficients to non-zero values and tends to distribute the regularization effect more evenly across all weights.

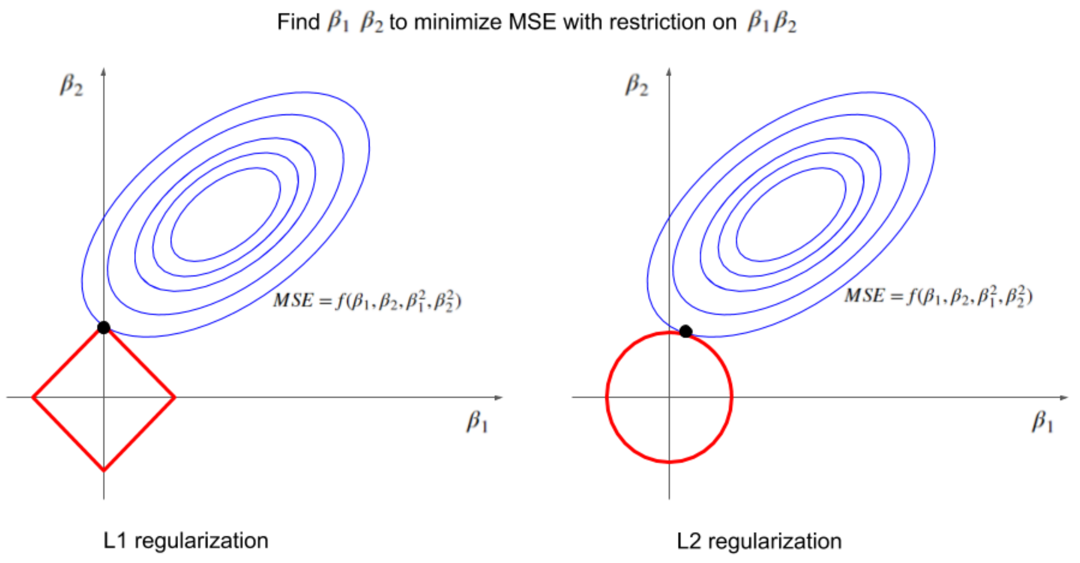
Find β_1 β_2 to minimize MSE with restriction on $\beta_1 \beta_2$
β_2
$MSE = f(\beta_1, \beta_2, \beta_1^2, \beta_2^2)$
β_1
L1 regularization
β_2
$MSE = f(\beta_1, \beta_2, \beta_1^2, \beta_2^2)$
β_1
L2 regularization

Figure 4: L1 and L2 regularization [1]

While the figure above perfectly illustrates the difference between the contour lines of L1 and L2 regularizations, it also shows why L1 induces sparsity while L2 does not. The L1 contours are not continuous when some parameters are zero, whereas the L2 contours are always continuous due to their elliptical geometry. Because of this geometry, L1 regularization ends up in its 'corners', forcing some model coefficients to zero, while L2 regularization only shrinks the coefficients to non-zero values.

Dropout

Dropout is a regularization technique used extensively in deep learning to prevent overfitting. The concept was introduced by Srivastava et al. in their 2014 paper as a surprisingly simple but effective way to improve the performance of neural networks. At its core, dropout involves randomly "dropping out" a subset of neurons during the training phase, meaning that these neurons are temporarily removed from the network along with all their incoming and outgoing connections.
During training, dropout is applied at each training step:

- For each layer where dropout is applied, each neuron (including its connections) is retained with a probability p (typically 0.5 for hidden layers), or dropped with probability $1 - p$.

- The decision to drop or keep a neuron is made independently for each neuron and each batch during the training process.
- Neurons that are retained are scaled by $\frac{1}{p}$ during training to account for the reduced number of active neurons in the network. This scaling compensates for the fact that more neurons are active at test time than during training.

Benefits of Dropout

- Reduces Overfitting: By randomly dropping neurons, the network cannot rely on any single neuron or combination of neurons because they might be dropped at any time during training. This encourages the network to distribute the learned representations across multiple neurons, leading to a more robust model that generalizes better to new data.
- Ensemble Effect: Dropout can be seen as training a large number of thin networks (each with different neurons dropped). At test time, it is akin to averaging the predictions from all these networks, which is similar to the effect of an ensemble of different models.
- Network Thinning: It effectively creates a thinner network on each forward pass, yet utilizes the full network at test time for more robust predictions.

Implementation and Usage

Dropout is typically applied to fully connected layers or densely connected regions within a network, although it can be adapted for use in convolutional layers as well. It's particularly useful in large networks that are prone to overfitting due to having a large number of parameters relative to the number of training samples.
In practice, dropout is straightforward to implement with modern deep learning frameworks like TensorFlow or PyTorch. In PyTorch, for example, dropout can be added to a network by including *torch.nn.Dropout* layers at desired points in the network architecture, specifying the dropout probability for each layer.

Considerations

- Tuning Dropout Rate: The probability of retaining a neuron (pp) is a hyperparameter that can significantly affect the performance and is usually tuned using a validation set.
- Impact on Convergence: While dropout helps in reducing overfitting, it might also make the convergence of the training process slower, requiring potentially more epochs or a slightly higher learning rate.
- Not Always Beneficial: Dropout has been found to be less effective or even detrimental in certain cases, particularly for tasks or datasets where overfitting is not a major concern or where data is not abundant.

Normalizing Inputs

Normalizing inputs is a widely used preprocessing step in machine learning and deep learning. This method standardizes the distribution of input features, allowing the model to converge faster with fewer steps required.

Normalization is performed using the following formula (Z-Score Normalization):

$$x' = \frac{x - \mu}{\sigma},$$

where x is the original data, μ is the mean, σ is the standard deviation of the data, and x' is the normalized value. This transformation rescales the data so that it has a mean of 0 and a standard deviation of 1, which is particularly useful when we want the data to follow a standard normal distribution.
Without normalization, the gradients of larger parameters dominate the updates; however, normalizing inputs standardizes the proportions of the parameters and corrects the step directions. This effect is also illustrated in the figure below, where the x-axis and y-axis represent different parameters, and the contours show the loss curves:

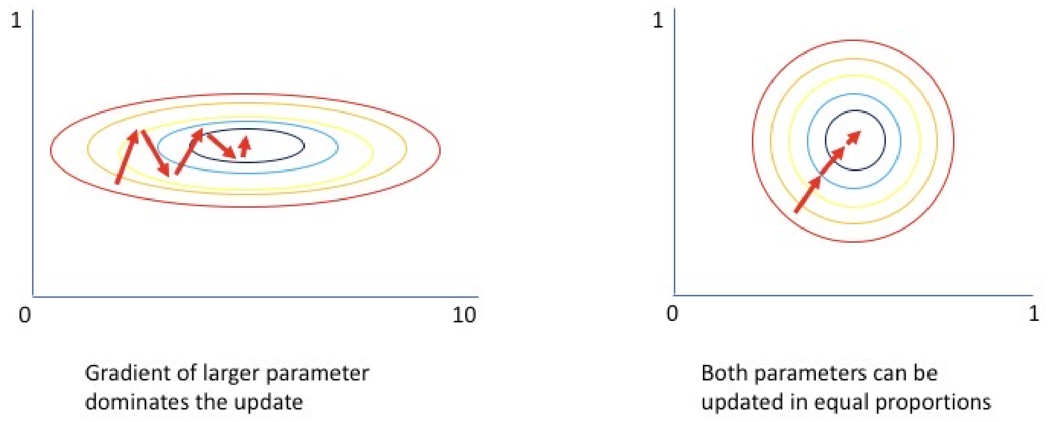
1
0
10
Gradient of larger parameter dominates the update
1
0
1
Both parameters can be updated in equal proportions

Figure 5: Normalizing inputs [2]

Weight Initialization for Deep Networks

Weight initialization is a crucial step in training deep neural networks, as it can significantly affect the speed of convergence and the performance of the model. Proper weight initialization ensures that the gradients are appropriately scaled across the layers, preventing issues like vanishing or exploding gradients, especially when the network is deep. Weight initialization is important because:

- **Prevents Vanishing/Exploding Gradients**: In deep networks, gradients tend to either shrink (vanish) or grow (explode) as they propagate backward through layers during training. Poor weight initialization can worsen these problems, and make learning inefficient or causing the network to not train at all by causing overflow errors in Python and other programming languages.

- **Speeds up Convergence**: Proper initialization can make training faster by ensuring that the activations and gradients are neither too large nor too small.

Random Initialization: In this initialization, weights are initialized randomly from a Gaussian distribution. Initializing all weights to the same value can lead to symmetry if they are all assigned to zero, where all neurons in a layer learn the same features. Random initialization helps break this symmetry, allowing the model to learn diverse features.
However, random initialization can lead to vanishing or exploding gradients, especially in deep networks. So, random values need to be carefully scaled to avoid these issues.

Xavier, and He initializations are scaled forms of random initialization that address vanishing or exploding gradients problems.
Xavier Initialization: Xavier initialization draws the weights from a distribution with a mean of 0 and a variance that depends on the number of input and output units of the layer. This initialization works well with sigmoid and tanh activation functions. Weights W are initialized as:

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}\right)$$

Where n_{in} and n_{out} are the number of input and output units in the layer.
This initialization works well with sigmoid or tanh activation functions because it keeps the variance of activations consistent across layers.
He Initialization: He initialization is similar to Xavier initialization, but is scaled for ReLU and its variants like Leaky ReLU activation functions. Its formula is:

$$W \sim \mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{n_{\text{in}}}}\right)$$

ReLU has an asymmetric activation that can result in dead neurons (neurons that output zero for all inputs) if not initialized carefully. He initialization helps avoid this issue with its initialization technique.
In smaller networks, initialization isn't as crucial since the backpropagated gradients are less likely to vanish or explode. Random initialization with a reasonable scale often works. However, in deeper networks, improper initialization can cause serious problems with gradients. If weights are initialized too large, activations and gradients can explode, making training unstable. If weights are too small, activations and gradients can vanish, leading to slow or stalled training.
In conclusion, proper weight initialization is key to successful deep learning training. It ensures stable gradient propagation, faster convergence, and better overall model performance. Choosing the correct initialization technique depends on the activation function and architecture used in your network.

Hyperparameter Tuning

In deep learning, selecting the best set of hyperparameters are important since their values affect the models' performance drastically. This process is called as hyperparameter tuning, and there are several ways these hyperparameters are set, such as grid search, random search and bayesian search.

Grid Search

Grid search is a brute-force technique that involves specifying a grid of hyperparameter values and systematically evaluating the model for every combination of these values.
In grid search, we define a set of possible values for each hyperparameter. These values are set such that each value pairs have the same distance between them, i.e. they construct a grid. Then, we train the model for every possible combination of these hyperparameters. The performance of each combination is evaluated, typically using

cross-validation and the best hyperparamaters are chosen as the final hyperparameter values.
It should be noted that when applying a grid search, it is important to scale the hyperparameters according to their range. For example, if a hyperparameter is in the range [1, 1000] with exponential growth, a 4-point grid should include the points $\{1, 10, 100, 1000\}$ rather than $\{1, 334, 667, 1000\}$. Similarly, if the hyperparameter's growth is linear, the grid should have the points $\{1, 334, 667, 1000\}$.

Random Search

Random search is performed by sampling random combinations of hyperparameters from a predefined range, rather than systematically testing every possible combination.
In this method, we first specify the ranges for each hyperparameter. The algorithm samples random values from these ranges and evaluates the model, and this process is repeated for a fixed number of iterations or until a stopping criterion is met, where the training loss is below or accuracy is higher than a certain threshold.
The advantage of random search over grid search is that using random values broadens the range of the search and it can find better hyperparameter values than grid search. However, it is possible to miss the optimal combination, as the sampling is random. There is no guarantee of finding the best parameters for the random search, but it generally finds a good set.

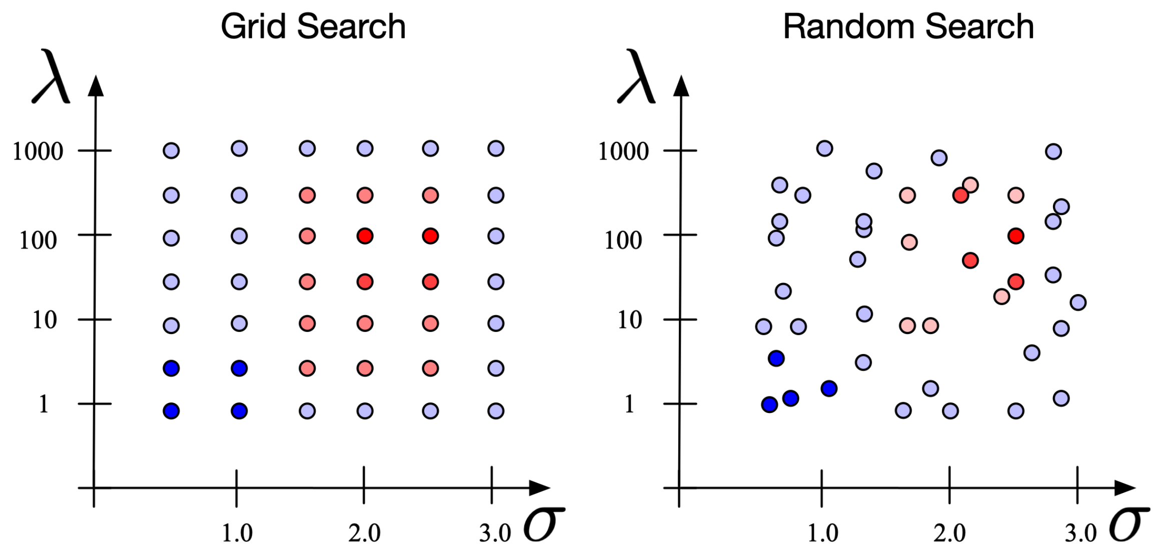

Figure 6: Grid search & random search [5]

References

[1] Chen, X. Intuitive and visual explanation on the differences between l1 and l2 regularization, 2024.

[2] Jordan, J. Normalizing your data, 2024.

[3] Karagiannakos, S. Regularization techniques for training deep neural networks, 2021.

[4] ubiAI. What are data augmentation techniques, 2024.

[5] Weinberger, K. Model selection: Tuning hyperparameters, 2023.