A Comprehensive Survey on Hyperparameter Tuning, Regularization, and Optimization in Deep Neural Networks

Harsh Gupta
Independent Researcher
harshmail281199@gmail.com

2 June, 2025

Abstract

This survey presents a detailed and structured examination of hyperparameter tuning, regularization methods, and optimization techniques in deep neural networks. It begins by laying a theoretical foundation of what hyperparameters are, how they influence model behavior, and their categorization into architecture, training, regularization, optimization, and exponential weighted averages (EWA). The paper explores modern tuning strategies such as grid search, random search, Bayesian optimization, and AutoML. It also covers essential regularization techniques like dropout, L1/L2 penalties, early stopping, and batch normalization. Various optimization algorithms including SGD, momentum, RMSProp, and Adam are examined along with best practices for robust experimentation. The survey concludes with practical guidelines for hyperparameter selection, learning curve analysis, experiment tracking, and ensuring model robustness. This paper aims to serve as a theoretical and practical guide for researchers and practitioners in the deep learning community.

Keywords: Hyperparameters, Regularization, Optimization, Bias-Variance, Learning Rate Decay, Exponential Weighted Averages, Batch Normalization

Contents

1 Introduction **5**

1.1 Scope of the Survey 5

1.2 Learning Outcomes of this Survey Paper 6

1.3 Motivation 7

2 Foundations of Hyperparameters **7**

2.1 What are Hyperparameters? 7

2.2 Hyperparameters vs. Parameters 8

2.3 Role in Model Performance 8

2.4 Bias-Variance Tradeoff 9

3 Categories of Hyperparameters **11**

3.1 Model Architecture Hyperparameters 11

3.2 Training Hyperparameters 12

3.3 Regularization Hyperparameters 12

3.4 Optimization Hyperparameters 13

3.5 Hyperparameters Specific to EWA (Exponential Weighted Averages) 14

4 Regularization Techniques in Deep Learning **14**

4.1 L1 and L2 Regularization 14

4.2 Elastic Net 15

4.3 Dropout 16

4.4 Early Stopping 16

4.5 Data Augmentation 17

4.6 Batch Normalization . . . 17

4.7 Normalization Techniques: When to Use What . . . 19

4.8 Summary . . . 20

5 Optimization Algorithms and Learning Techniques 20

5.1 Mini-Batch Gradient Descent . . . 21

5.2 Stochastic Gradient Descent (SGD) . . . 21

5.3 Momentum-Based Methods . . . 22

5.4 RMSProp . . . 22

5.5 Adam . . . 23

5.6 Exponential Weighted Averages (EWA) . . . 24

5.7 Learning Rate Scheduling and Decay . . . 25

6 Hyperparameter Tuning Strategies 27

6.1 Manual Search . . . 27

6.2 Grid Search . . . 28

6.3 Random Search . . . 29

6.4 Adequate Search . . . 29

6.5 Coarse to Fine Search . . . 30

6.6 Bayesian Optimization . . . 31

6.7 Automated Machine Learning (AutoML) . . . 32

7 Training Challenges and Mitigation Strategies 33

7.1 Vanishing and Exploding Gradients . . . 33

7.2 Curse of Dimensionality . . . 34

7.3 Overfitting to Validation Set . . . 34

7.4 Choosing an Appropriate Scale for Hyperparameters 35

8 Best Practices and Guidelines 36

8.1 Choosing Initial Hyperparameter Ranges 36

8.2 Using Learning Curves 37

8.3 Tracking Experiments 38

8.4 Hyperparameter Robustness 39

9 Conclusion 40

10 Acknowledgements 40

1 Introduction

Deep neural networks (DNNs) have become the cornerstone of modern artificial intelligence, enabling breakthroughs in computer vision, natural language processing, speech recognition, and more. Their success, however, is not solely due to architectural innovations or large datasets. A significant and often underestimated factor in their performance is the selection and configuration of hyperparameters—variables that are not learned from the data but are defined prior to the training process. These include learning rates, batch sizes, dropout probabilities, weight decay coefficients, momentum factors, and many more.

Tuning hyperparameters is both an art and a science. Poorly chosen values can lead to slow convergence, overfitting, underfitting, or even complete failure of the training process. Conversely, properly tuned hyperparameters can dramatically improve training speed, generalization, and final model performance. Moreover, concepts such as the bias-variance tradeoff, regularization, and optimization techniques are closely interwoven with hyperparameter selection. As such, a deep understanding of how each hyperparameter impacts the training dynamics is essential for any practitioner or researcher.

This survey aims to provide a comprehensive and detailed examination of hyperparameter tuning, regularization methods, and optimization algorithms in deep learning. It not only outlines the theory behind these components but also dives deep into their mathematical foundations, practical implications, and visualizations to foster a rigorous understanding. Throughout this paper, formulas, pseudocode, and graphical explanations will accompany each concept to bridge the gap between theoretical understanding and practical implementation.

1.1 Scope of the Survey

This survey focuses on the following core areas:

- The foundational understanding of what hyperparameters are and why they matter.
- Categories of hyperparameters and their influence on model training and generalization.
- Exhaustive and automated search strategies for hyperparameter tuning, including Bayesian methods and AutoML frameworks.

- Regularization techniques such as L1/L2 regularization, dropout, early stopping, and batch normalization, with mathematical derivations.
- Optimization strategies including gradient descent variants, learning rate scheduling, and Exponential Weighted Averages (EWA).
- Training challenges like vanishing/exploding gradients, overfitting, and hyperparameter scaling, along with best practices for mitigation.

Each topic is accompanied by in-depth explanations, mathematical formulations, and implementation strategies to serve as a practical reference for both beginners and experienced deep learning researchers.

1.2 Learning Outcomes of this Survey Paper

Upon completing this survey, readers will be able to:

- Differentiate between parameters and hyperparameters in neural networks.
- Understand the impact of each hyperparameter on convergence, generalization, and stability.
- Analyze and apply various tuning strategies in real-world projects.
- Implement regularization methods from scratch and understand their mathematical rationale.
- Apply and fine-tune optimization algorithms like SGD, Adam, RMSProp, and their variants.
- Tackle common training challenges through a deeper understanding of model dynamics and hyperparameter interactions.

This survey acts as a bridge between theoretical learning and applied machine learning engineering, making it especially useful for researchers developing custom architectures or designing new optimization algorithms.

1.3 Motivation

In the age of pre-trained transformers and large-scale foundation models, many developers focus on using off-the-shelf architectures without deeply understanding the mechanics that drive training performance. While APIs and frameworks have abstracted much of the process, true innovation and mastery come from understanding the fine-tuned components behind the scenes. Hyperparameters are among the most sensitive and influential components of this process.

The motivation behind this paper is to demystify hyperparameters by unpacking their role in the training pipeline. This includes how they relate to optimization stability, convergence rate, generalization ability, and model robustness. By providing detailed insights, this paper intends to empower readers to approach model training not as a black-box procedure but as a deliberate and informed process.

Furthermore, there is a lack of consolidated, in-depth survey papers that rigorously cover all these intertwined topics—tuning, regularization, and optimization—from both a theoretical and applied perspective. This paper fills that gap and serves as both a teaching tool and a practical reference.

2 Foundations of Hyperparameters

2.1 What are Hyperparameters?

Hyperparameters are external configuration variables set before the training process begins. Unlike model parameters, which are learned from the data through optimization, hyperparameters guide and control the training dynamics. Their values profoundly influence the performance, convergence speed, and generalization ability of neural networks.

Common hyperparameters in deep learning include:

- **Learning rate** (η) — the step size for each weight update.

- **Batch size** — the number of training examples processed per forward/backward pass.
- **Number of epochs** — how many times the model sees the entire training data.
- **Number of layers and units per layer** — architectural decisions.
- **Dropout rate** — fraction of units randomly deactivated to prevent overfitting.
- **Weight decay (L2 regularization)** — penalizes large weights.

For example, consider a simple feedforward neural network. If the learning rate is too large, the network may overshoot the minima during gradient descent. If it's too small, convergence will be extremely slow or the network might get stuck in local minima.

2.2 Hyperparameters vs. Parameters

It's critical to distinguish between model *parameters* and *hyperparameters*:

- **Parameters** are learned from the training data through optimization algorithms. They include weights (w) and biases (b) in neural networks.
- **Hyperparameters** are preset by the practitioner and define the behavior of the learning algorithm or the model structure itself.

Example:

In the function:

$$y = \sigma(w_1 x_1 + w_2 x_2 + b)$$

- w_1, w_2, b are parameters (learned via backpropagation). - The choice of activation function σ, learning rate η, and batch size are hyperparameters.

2.3 Role in Model Performance

Hyperparameters play a central role in determining how well a deep neural network learns and generalizes. The performance of a model is often highly sensitive to these choices.

- **Underfitting:** Too few layers or low capacity can fail to capture the underlying patterns.
- **Overfitting:** Excessive capacity or poor regularization can cause the model to memorize the training data.
- **Slow convergence:** A learning rate that is too low can slow down training.
- **Instability:** A large learning rate or poor batch size selection can cause the loss to fluctuate or diverge.

Visual example:

Consider training a neural network for image classification. Two different learning rates, $\eta_1 = 0.001$ and $\eta_2 = 0.1$, can produce completely different convergence behavior:

- $\eta_1 = 0.001$: The loss slowly decreases and may plateau before reaching an optimal value.
- $\eta_2 = 0.1$: The loss may oscillate wildly or diverge.

Choosing hyperparameters intelligently—often through tuning techniques such as grid search or Bayesian optimization—can dramatically improve the final model.

2.4 Bias-Variance Tradeoff

The bias-variance tradeoff is a fundamental concept in supervised learning and directly relates to model generalization. It provides insight into how the complexity of a model and the choice of hyperparameters affect prediction error.

Definition

The total expected prediction error can be decomposed as:

$$\text{Total Error} = \underbrace{\text{Bias}^2}_{\text{Error due to wrong assumptions}} + \underbrace{\text{Variance}}_{\text{Sensitivity to training data}} + \underbrace{\text{Irreducible Error}}_{\text{Noise in data}}$$

Bias refers to error due to overly simplistic assumptions in the model. High bias can cause underfitting.

Variance refers to error due to high sensitivity to small fluctuations in the training set. High variance can cause overfitting.

Examples in Hyperparameter Context:

- A shallow network (few layers, few units) has high bias and low variance.
- A deep network with many units and low regularization has low bias but high variance.
- Regularization hyperparameters (e.g., L2 penalty, dropout) are used to reduce variance.
- Learning rate scheduling and batch size can help balance stability and generalization.

Goal: Find the sweet spot where both bias and variance are reasonably low.

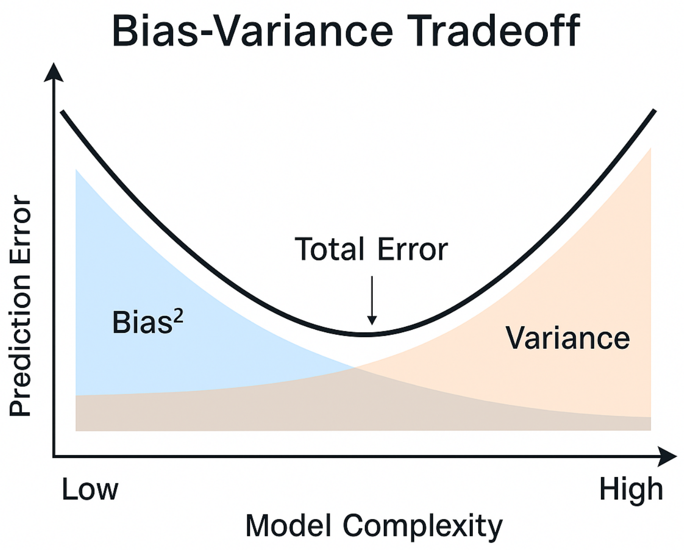

Figure 1: Bias-Variance Tradeoff: showing how total error is a combination of bias and variance depending on model complexity.

In practice, plotting the training and validation loss curves helps identify whether a model is overfitting (high variance) or underfitting (high bias), and hyperparameters can be adjusted accordingly.

3 Categories of Hyperparameters

Hyperparameters in deep learning can be broadly categorized based on their function and scope. Understanding these categories is essential for organizing tuning workflows, debugging training issues, and improving generalization. This section breaks hyperparameters down into five functional categories: model architecture, training, regularization, optimization, and exponential weighted averages (EWA).

3.1 Model Architecture Hyperparameters

Model architecture hyperparameters define the structural design of the neural network. They determine how many learnable parameters the model will have and how capable it is of capturing patterns in the data.

- **Number of Layers:** Determines the depth of the model. Deeper networks can learn more abstract features but are harder to train due to vanishing gradients and overfitting.
- **Number of Units per Layer:** Controls the width of each layer. Wider layers have more capacity but increase computation and overfitting risk.
- **Activation Functions:** Common choices include ReLU, Leaky ReLU, Tanh, and Sigmoid. These affect non-linearity and gradient flow.
- **Type of Layers:** For example, fully-connected, convolutional (CNN), recurrent (RNN), transformer layers, etc.

Example: A fully connected (dense) neural network for tabular data classification may have the following architectural hyperparameters:

- 1 input layer with 30 features (input neurons).
- 3 hidden layers with 128, 64, and 32 units respectively.
- ReLU activation function used in all hidden layers.
- 1 output layer with a sigmoid activation for binary classification.

Architectural hyperparameters often interact with other categories like regularization (e.g., dropout between layers) and optimization (e.g., weight initialization).

3.2 Training Hyperparameters

Training hyperparameters define how the learning algorithm proceeds over time.

- **Batch Size** (m)**:** Number of examples processed in one forward/backward pass. Typical values are 32, 64, 128. Smaller batch sizes lead to noisier updates but can generalize better.
- **Number of Epochs** (E)**:** Total passes over the full training dataset. Larger values can overfit; early stopping is often used.
- **Learning Rate** (η)**:** Controls the step size of weight updates:

$$w := w - \eta \frac{\partial L}{\partial w}$$

- **Weight Initialization:** e.g., Xavier (Glorot), He initialization — affects convergence.

Choosing an appropriate learning rate is arguably the most critical training hyperparameter. Too low: slow convergence. Too high: divergence.

3.3 Regularization Hyperparameters

These are used to prevent overfitting by reducing the model's capacity or encouraging simpler solutions.

- **L2 Regularization (Weight Decay):** Adds a penalty term to the loss:

$$J(w) = L(y, \hat{y}) + \lambda \sum_i w_i^2$$

- **L1 Regularization:** Encourages sparsity:

$$J(w) = L(y, \hat{y}) + \lambda \sum_i |w_i|$$

- **Dropout Rate:** Fraction of neurons randomly disabled during training.
- **Batch Normalization Momentum:** Affects how batch stats are updated.
- **Early Stopping Patience:** Number of epochs to wait before stopping when validation loss stops improving.
- **Data Augmentation Factors:** Rotation, scaling, flipping parameters for image tasks.

Regularization hyperparameters are often used together. For example, applying both dropout and L2 weight decay is common in image models.

3.4 Optimization Hyperparameters

These affect how the model learns, particularly how gradients are used to update the weights.

- **Optimizer Choice:** SGD, Adam, RMSProp, AdaGrad, etc.
- **Momentum** (β)**:** Speeds up SGD by averaging gradients:

$$v_t = \beta v_{t-1} + (1 - \beta)\frac{\partial L}{\partial w}$$

$$w := w - \eta v_t$$

- **Learning Rate Decay:** Reduce η over time:

$$\eta_t = \eta_0 \cdot \frac{1}{1 + \text{decay rate} \cdot t}$$

- **Gradient Clipping:** Prevents exploding gradients by thresholding large gradients.
- **Epsilon** (ϵ)**:** A small constant to prevent division by zero in optimizers like Adam.

The interaction between optimizers and learning rate decay can have a large impact. For instance, Adam with cosine decay can outperform vanilla SGD in many settings.

3.5 Hyperparameters Specific to EWA (Exponential Weighted Averages)

EWA is a technique used in optimization algorithms (e.g., Momentum, RMSProp, Adam) to smooth gradients or second-order statistics.

- **Beta Values** (β_1, β_2)**:**
  - β_1 is used for the first moment estimate (mean of gradients).
  - β_2 is used for the second moment estimate (uncentered variance).
- **Epsilon** (ϵ)**:** Added to the denominator to avoid division by zero.
- **Bias Correction:** Compensates for initialization bias:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- **Learning Rate Schedule with EWA:** Can be coupled with warm restarts or decay.

Default values like $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are generally good but can be fine-tuned for improved convergence in specific tasks.

4 Regularization Techniques in Deep Learning

Regularization is a set of techniques used to prevent overfitting in machine learning models by discouraging complex models that may memorize the training data. In deep neural networks, regularization becomes even more important due to the high capacity of these models. This section explores major regularization strategies used in deep learning, including mathematical formulations and implementation insights.

4.1 L1 and L2 Regularization

These are classic regularization techniques that penalize large weights in the model.

L2 Regularization (Weight Decay):

L2 regularization adds a penalty term proportional to the square of the weights to the loss function:

$$J(w) = L(y, \hat{y}) + \lambda \sum_{i=1}^{n} w_i^2$$

Where:

- $J(w)$ is the regularized loss,
- $L(y, \hat{y})$ is the original loss (e.g., cross-entropy),
- λ is the regularization strength,
- w_i are the model weights.

L2 regularization discourages large weights, leading to smoother models with better generalization.

L1 Regularization:

L1 regularization adds a penalty proportional to the absolute values of weights:

$$J(w) = L(y, \hat{y}) + \lambda \sum_{i=1}^{n} |w_i|$$

It encourages sparsity (i.e., many weights become zero), which is useful for feature selection.

4.2 Elastic Net

Elastic Net combines both L1 and L2 regularization:

$$J(w) = L(y, \hat{y}) + \lambda_1 \sum_{i=1}^{n} |w_i| + \lambda_2 \sum_{i=1}^{n} w_i^2$$

This balances sparsity and weight shrinkage, and is useful in settings where groups of correlated features exist.

4.3 Dropout

Dropout is a stochastic regularization technique that randomly deactivates a subset of neurons during training. This forces the network to learn redundant (backup) representations and prevents co-adaptation of features.

Mechanism:

- During training, each neuron's output is set to zero with probability p.
- During inference, all neurons are used, but their outputs are scaled by $1 - p$.

Mathematically: If h is a hidden layer output, dropout applies:

$$\tilde{h} = h \cdot \mathbf{m}, \quad \text{where } m_i \sim \text{Bernoulli}(1 - p)$$

Dropout is most effective in fully connected layers. Typical values of p range from 0.2 to 0.5.

4.4 Early Stopping

Early stopping monitors model performance on a validation set and halts training when performance stops improving.

Key Parameters:

- **Patience:** Number of epochs to wait after last improvement.
- **Monitored Metric:** Validation loss or accuracy.

This technique prevents the model from continuing to learn patterns that only exist in the training set, which would lead to overfitting.

4.5 Data Augmentation

Data augmentation artificially expands the training set by applying transformations to input data. It is especially useful in image and audio tasks.

Examples (for images):

- Horizontal/vertical flipping
- Random cropping
- Rotation and scaling
- Color jitter and brightness shifts

Example (for text):

- Synonym replacement
- Sentence shuffling
- Back-translation

Augmentation helps the model become invariant to small perturbations in input and generalize better.

4.6 Batch Normalization

Batch Normalization (BN) is a technique to normalize the inputs to each layer within a mini-batch. The motivation is to mitigate the *internal covariate shift* — the change in the distribution of inputs to each layer as training progresses, which can slow down convergence or destabilize learning.

Normalization Step

Given a mini-batch $\mathcal{B} = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ for some layer's input activations, batch normalization computes (mean and variance):

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu_{\mathcal{B}}\right)^2$$

Then, the inputs are normalized:

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

Here, ϵ is a small constant (e.g., 10^{-5}) added for numerical stability.

Scaling and Shifting: Learnable Parameters

Unlike traditional normalization, BN introduces two learnable parameters for each activation:

- γ (gamma): scale parameter
- β (beta): shift parameter

The final output of BN is:

$$y^{(i)} = \gamma \hat{x}^{(i)} + \beta$$

These parameters allow the network to **recover the original representation if necessary** (i.e., when learning that normalization is not needed).

Parameter Initialization and Learning

Typically:

- γ is initialized to 1 (preserves variance).
- β is initialized to 0 (zero mean).

They are updated via backpropagation just like other weights in the network. The gradients of γ and β are:

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y^{(i)}} \cdot \hat{x}^{(i)}, \quad \frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y^{(i)}}$$

Training vs Inference

During training: BN uses the mini-batch mean and variance to normalize the data.

During inference: BN uses a running average of the mean and variance collected during training:

$$\mu_{\text{running}} \leftarrow \rho \cdot \mu_{\text{running}} + (1 - \rho) \cdot \mu_{\mathcal{B}}$$

$$\sigma^2_{\text{running}} \leftarrow \rho \cdot \sigma^2_{\text{running}} + (1 - \rho) \cdot \sigma^2_{\mathcal{B}}$$

Here, ρ is a momentum parameter (commonly 0.9 or 0.99). These running statistics are then used at inference time to ensure consistency.

Impact on Optimization

Batch normalization has several notable effects:

- Allows for higher learning rates due to stabilized gradient flow.
- Acts as a regularizer — often reducing or removing the need for dropout.
- Reduces sensitivity to weight initialization.
- Mitigates vanishing/exploding gradient problems in deep networks.

4.7 Normalization Techniques: When to Use What

In the context of this survey's scope, which focuses on regularization strategies used alongside deep neural networks, **Batch Normalization** (BN) is the most widely used normalization technique.

Where to use Batch Normalization:

- Works best in feedforward networks and convolutional neural networks (CNNs).

- Should be placed **after the linear transformation** and **before the activation function**.
- Not ideal for very small batch sizes — in such cases, alternatives like Layer Normalization (not covered here) might be more stable.
- Often used in place of or alongside Dropout in deeper models.

When to use:

- Use BN when facing unstable gradients or slow convergence.
- Particularly helpful if learning rate tuning is difficult — BN can stabilize learning across a wider range of learning rates.
- If Dropout is causing underfitting, try replacing it or combining it with BN in hidden layers.

4.8 Summary

When paired appropriately with other methods covered in this section, such as Dropout, L2 regularization, and Data Augmentation, Batch Normalization can significantly enhance the generalization ability and training efficiency of deep neural networks.

5 Optimization Algorithms and Learning Techniques

Optimization algorithms are at the heart of training deep neural networks. They determine how the model's parameters (weights and biases) are updated based on gradients computed from the loss function. This section covers key gradient-based optimization techniques and enhancements such as momentum, adaptive learning rates, and scheduling strategies.

5.1 Mini-Batch Gradient Descent

Instead of computing gradients over the entire training set (as in batch gradient descent) or a single example (as in stochastic gradient descent), mini-batch gradient descent computes the gradient over small subsets of the data.

Update rule:

$$w := w - \eta \cdot \nabla_w L_{\text{mini-batch}}$$

Where:

- η: learning rate
- $L_{\text{mini-batch}}$: loss computed on a mini-batch

Advantages:

- Faster convergence than full batch
- Less noisy than SGD
- Allows for efficient parallelization on GPUs

Common mini-batch sizes: 32, 64, 128.

5.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates weights using one training example at a time.

Update rule:

$$w := w - \eta \cdot \nabla_w L(x^{(i)}, y^{(i)})$$

Pros:

- Noisy updates help escape local minima
- Requires very little memory

Cons:

- High variance in updates
- Slower convergence and more fluctuations

In practice, mini-batch SGD is used instead of pure SGD.

5.3 Momentum-Based Methods

Momentum accelerates SGD by accumulating an exponentially weighted moving average of past gradients, leading to smoother and faster updates.

Update rules:

$$v_t = \beta v_{t-1} + (1 - \beta) \cdot \nabla_w L$$

$$w := w - \eta v_t$$

Where:

- $\beta \in [0.9, 0.99]$: momentum coefficient
- v_t: velocity vector

Intuition: Think of a ball rolling down a hill — it builds up velocity, and small obstacles don't stop it.

Nesterov Accelerated Gradient (NAG): An improved version that computes gradients at the "look-ahead" position:

$$v_t = \beta v_{t-1} + \eta \cdot \nabla_w L(w - \beta v_{t-1})$$

5.4 RMSProp

RMSProp (Root Mean Square Propagation) adapts the learning rate for each parameter based on a moving average of squared gradients.

Update rules:

$$s_t = \beta s_{t-1} + (1-\beta)(\nabla_w L)^2$$

$$w := w - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot \nabla_w L$$

Where:

- s_t: accumulated squared gradient
- ϵ: small constant to avoid division by zero

Benefits:

- Efficient in dealing with non-stationary objectives
- Especially good for RNNs and sequence models

5.5 Adam

Adam (Adaptive Moment Estimation) combines momentum and RMSProp. It maintains an exponentially decaying average of both gradients and squared gradients.

Update rules:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_w L$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_w L)^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w := w - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Default hyperparameters:

- $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Strengths:

- Combines the best of momentum and RMSProp
- Works well out of the box for most problems

5.6 Exponential Weighted Averages (EWA)

Exponential Weighted Averages (EWA) are used to smooth noisy sequences, particularly gradients or squared gradients, by giving exponentially decreasing weight to past observations. EWA is a fundamental component in optimizers like Momentum, RMSProp, and Adam.

Why Use EWA?

Gradient estimates can fluctuate wildly between iterations due to noise or high curvature regions. EWA helps to:

- Reduce noise in gradient updates.
- Retain memory of past gradients for better directionality (momentum).
- Adaptively adjust the learning rate based on gradient magnitude.

This smoothing effect leads to more stable and reliable convergence, especially in high-dimensional or ill-conditioned loss surfaces.

Formula

The general EWA update is:

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

Where:

- v_t: exponentially weighted average at time t
- x_t: current observation (e.g., gradient or squared gradient)
- $\beta \in (0, 1)$: decay rate (closer to 1 = smoother average)

Why Bias Correction is Needed

At the beginning of training, v_t is biased toward zero since $v_0 = 0$. This is especially noticeable in early iterations, where the running average underestimates the true value.

To correct this bias, we use:

$$\hat{v}_t = \frac{v_t}{1 - \beta^t}$$

This bias correction is crucial in algorithms like Adam, especially during the initial steps, to ensure the first few parameter updates are not too small.

Effect on Optimization

Using EWA allows optimizers to:

- ”Remember” useful past gradients (momentum-like behavior).
- Reduce sensitivity to sudden spikes or noise.
- Create directionally consistent updates in valleys or along long ridges of the loss surface.

EWA is also a way to embed temporal information into optimization, smoothing both variance and directional updates in dynamic environments.

5.7 Learning Rate Scheduling and Decay

The learning rate η is one of the most critical hyperparameters in deep learning. A fixed learning rate often results in suboptimal training — either convergence is slow, or the training becomes unstable. Learning rate scheduling dynamically adjusts the learning rate throughout training to achieve faster convergence and better generalization.

Why Use Scheduling?

- Early training benefits from a higher learning rate to explore the parameter space.
- Later training benefits from a smaller learning rate to fine-tune around local minima.

- Decaying or cyclic learning rates reduce the risk of getting stuck in poor local minima.

Common Scheduling Strategies

- **Step Decay:**
$$\eta_t = \eta_0 \cdot \gamma^{\lfloor \frac{t}{s} \rfloor}$$
Where s is the number of epochs between drops, and γ is the decay factor (e.g., 0.5 or 0.1).

- **Exponential Decay:**
$$\eta_t = \eta_0 \cdot e^{-kt}$$
Where k is the decay rate.

- **1/t Decay:**
$$\eta_t = \frac{\eta_0}{1 + kt}$$

- **Warm Restarts (SGDR):** Reset learning rate to high value after a scheduled decay, often paired with cosine annealing to simulate multiple restarts.

- **Discrete Staircase Decay:** A simple piecewise constant strategy where the learning rate drops sharply at predefined epochs:
$$\eta_t = \begin{cases} \eta_0 & \text{if } t < t_1 \\ \eta_1 & \text{if } t_1 \leq t < t_2 \\ \eta_2 & \text{if } t_2 \leq t < t_3 \\ \cdots \\ \eta_n & \text{otherwise} \end{cases}$$
Often used in practice with manual tuning — effective and simple.

.

6 Hyperparameter Tuning Strategies

Hyperparameter tuning is the process of systematically searching for the best configuration of hyperparameters to maximize a model's performance. Since deep learning models are highly sensitive to hyperparameters, choosing the right tuning strategy can significantly affect model accuracy, training stability, and generalization. Below are the most widely used strategies, along with their mechanisms, benefits, and trade-offs.

6.1 Manual Search

Manual search involves tweaking hyperparameters one at a time based on domain knowledge, intuition, or visual inspection of performance metrics.

How it Works:

- Start with a reasonable initial guess or defaults.
- Adjust one hyperparameter at a time (e.g., learning rate).
- Evaluate the model's performance.
- Iterate based on trial and error.

Pros:

- Simple and easy to execute.
- Can be guided by expert knowledge.

Cons:

- Time-consuming and inefficient.
- Prone to bias and suboptimal results.
- Not scalable to large search spaces.

Use Case: Small models, few hyperparameters, or preliminary experiments.

6.2 Grid Search

Grid search performs an exhaustive search over a predefined, discrete set of hyperparameter values.

How it Works:

- Define a grid of values for each hyperparameter.
- Train and evaluate models on all possible combinations.
- Select the configuration with the best validation performance.

Example:

- Learning rate: $\{0.001, 0.01, 0.1\}$
- Batch size: $\{16, 32, 64\}$
- Total models: $3 \times 3 = 9$ models trained

Pros:

- Simple and systematic.
- Guarantees full exploration of the defined space.

Cons:

- Computationally expensive.
- Inefficient in high-dimensional spaces.
- Wastes effort on unimportant parameters.

Use Case: Low-dimensional search spaces, or when computation is cheap.

6.3 Random Search

Random search samples hyperparameters randomly from specified distributions, rather than exhaustively evaluating all combinations.

How it Works:

- Specify a range or distribution for each hyperparameter.
- Randomly sample and train/evaluate n configurations.
- Return the best-performing configuration.

Example:

- Learning rate: log-uniform(1e-5, 1e-1)
- Dropout: uniform(0.2, 0.5)
- Sample 30 random combinations.

Pros:

- More efficient than grid search.
- Can discover better-performing regions faster.

Cons:

- No coverage guarantees.
- Randomness introduces variability in results.

Use Case: Preferred baseline method for tuning over large spaces.

6.4 Adequate Search

Adequate search focuses on searching just long enough to find diminishing returns, saving compute by stopping early.

How it Works:

- Combine random/grid search with early-stopping of tuning trials.
- Monitor validation performance.
- Stop when additional tuning yields little improvement.

Pros:

- Saves compute without sacrificing performance.
- Avoids overfitting to the validation set.

Cons:

- Risk of premature stopping.
- Might miss global optima if cut too early.

Use Case: Tuning large models under a limited computational budget.

6.5 Coarse to Fine Search

Coarse-to-fine search uses a two-phase approach: explore broadly with coarse granularity, then refine around promising regions.

How it Works:

1. **Phase 1:** Broad grid or random search across large ranges.
2. **Phase 2:** Narrow search around top-performing values.

Example:

- Coarse: Learning rate {0.001, 0.01, 0.1}

- Fine: Learning rate {0.006, 0.007, 0.008}

Pros:

- Focuses effort on promising regions.
- Combines exploration and exploitation.

Cons:

- Requires manual intervention or heuristics to define fine search bounds.

Use Case: Effective for tuning one or two sensitive hyperparameters like learning rate or weight decay.

6.6 Bayesian Optimization

Bayesian optimization uses probabilistic modeling (usually Gaussian Processes) to intelligently explore the hyperparameter space with fewer evaluations.

How it Works:

1. Fit a surrogate model to predict performance from past trials.
2. Use an acquisition function (e.g., Expected Improvement) to balance exploration vs. exploitation.
3. Choose the next hyperparameters that optimize the acquisition function.
4. Repeat until convergence or budget is exhausted.

Pros:

- Sample-efficient — finds good values with fewer trials.
- Automatically balances exploration and exploitation.

Cons:

- Computationally expensive per iteration.
- Complex to implement.

Use Case: When each model training is expensive, tuning needs to be sample-efficient.

6.7 Automated Machine Learning (AutoML)

AutoML automates the end-to-end process of model selection, hyperparameter tuning, and sometimes feature engineering.

How it Works:

- Leverages search algorithms (Bayesian, genetic, bandits) behind the scenes.
- Accepts user input like a dataset and an objective metric.
- Returns the best model configuration with minimal human involvement.

Popular Tools:

- Optuna, HyperOpt, Ray Tune, Auto-sklearn, Google AutoML

Pros:

- Minimal human effort.
- Often integrates architecture tuning and ensembling.

Cons:

- High compute and memory usage.
- Limited interpretability.

Use Case: Enterprise and production workflows, or when non-experts need strong baseline models.

7 Training Challenges and Mitigation Strategies

Despite advances in model architectures and optimization algorithms, training deep neural networks remains a challenging task. This section outlines common training obstacles and offers strategies to mitigate them effectively.

7.1 Vanishing and Exploding Gradients

In deep networks, gradients can become excessively small (vanish) or excessively large (explode) as they are backpropagated through many layers, particularly with activation functions like Sigmoid or Tanh.

Mathematical Intuition: During backpropagation, gradients are computed using the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial a_{n-1}} \cdots \frac{\partial a_1}{\partial w}$$

If each derivative $\frac{\partial a_i}{\partial a_{i-1}}$ is less than 1, gradients shrink exponentially $\rightarrow$ **vanishing**. If greater than 1, they grow exponentially $\rightarrow$ **exploding**.

Symptoms:

- Vanishing: weights stop updating; loss stagnates.
- Exploding: unstable updates; loss diverges to NaN.

Mitigation Strategies:

- Use **ReLU** instead of Sigmoid/Tanh.
- Apply **Batch Normalization** to normalize activations.

- Use **gradient clipping** to cap excessively large gradients.
- Choose **careful weight initialization** (e.g., Xavier or He initialization).
- Use **residual connections** (e.g., in ResNets) to shorten gradient paths.

7.2 Curse of Dimensionality

As the number of input features (dimensions) increases, the data becomes sparse and the volume of the input space grows exponentially. This makes it harder for the model to generalize.

Challenges:

- Distance metrics become less meaningful.
- Overfitting becomes more likely with high-dimensional input.
- Requires exponentially more data to maintain model performance.

Mitigation Strategies:

- Perform **feature selection** using domain knowledge or mutual information.
- Apply **dimensionality reduction** techniques (e.g., PCA).
- Use **regularization** (L1, L2) to penalize uninformative weights.
- Collect more data (if possible) or apply data augmentation.

Use Case: Common in domains like genomics, text (TF-IDF vectors), and financial data.

7.3 Overfitting to Validation Set

When hyperparameter tuning is repeatedly performed based on validation performance, the model can inadvertently "memorize" patterns specific to the validation set.

Symptoms:

- High validation accuracy but poor generalization to test data.

- Very small gap between training and validation performance during tuning, but high test error.

Causes:

- Excessive tuning on a small validation set.
- Lack of test set or independent evaluation data.
- Small dataset size with high variance.

Mitigation Strategies:

- Use a separate **test set** that is never touched during training or tuning.
- Use **cross-validation** instead of a single validation split.
- Apply **early stopping** and monitor generalization gap.
- Limit the number of tuning iterations or apply **adequate search** techniques.

7.4 Choosing an Appropriate Scale for Hyperparameters

Hyperparameters like learning rate, regularization coefficients, or dropout are often sensitive to their scale — linear vs logarithmic.

Examples:

- Learning rates: better searched in log space (e.g., 10^{-5} to 10^{-1}).
- Dropout: linear scale is more suitable (e.g., 0.1 to 0.5).
- Weight decay: log scale for tiny values (e.g., 10^{-6} to 10^{-2}).

Why It Matters:

- A linear search might completely miss important regions (e.g., $\eta = 0.001$ is vastly different from 0.01).
- Proper scaling reduces the number of evaluations required in grid or random search.

Best Practices:

- Use `log-uniform` sampling for learning rate and weight decay.
- Normalize hyperparameters when using Bayesian optimization.
- Apply coarse-to-fine search using appropriate scales.

.

8 Best Practices and Guidelines

Choosing and tuning hyperparameters is both a science and an art. While many tuning methods exist, success depends on applying consistent best practices. This section outlines general guidelines and proven strategies that help practitioners improve training efficiency, reduce trial-and-error effort, and achieve more robust and reproducible deep learning models.

8.1 Choosing Initial Hyperparameter Ranges

Selecting effective initial hyperparameter values and their search ranges is critical, especially when using automated tuning methods like random search or Bayesian optimization.

Heuristics and Recommendations:

- **Learning Rate** (η)**:**
  - Start with 10^{-3} for Adam and 10^{-2} or 10^{-1} for SGD.
  - Log-scale search: sample from $\eta \in [10^{-5}, 1]$ instead of linear.
- **Batch Size:**
  - Typical values: 32, 64, 128, 256.
  - Trade-off: small batches offer regularization, large batches offer stability.
- **Dropout Rate:**

  - For fully connected layers, start with 0.2–0.5.
  - For convolutional layers, usually lower (0.1–0.3).

- **Regularization Strength** (λ)**:**
  - Start with $\lambda \in [10^{-5}, 10^{-2}]$.

- **Momentum** (β)**:**
  - Typical value: 0.9 for SGD with momentum.

- **Number of Epochs:**
  - Use early stopping; start with a large enough value (e.g., 100–200).

Tip: Use log-uniform sampling for continuous hyperparameters and grid/random search for discrete ones.

8.2 Using Learning Curves

Learning curves are essential diagnostic tools that help evaluate model performance during training. They visualize how metrics like loss or accuracy evolve over epochs for both training and validation datasets. These curves are crucial for identifying common issues such as underfitting or overfitting.

Interpretation Guidelines:

- **Underfitting:** High training and validation loss that decrease very slowly or not at all.
- **Overfitting:** Training loss decreases consistently, but validation loss starts increasing after a certain point.
- **Good Fit:** Both losses decrease and converge to low values with a small gap.

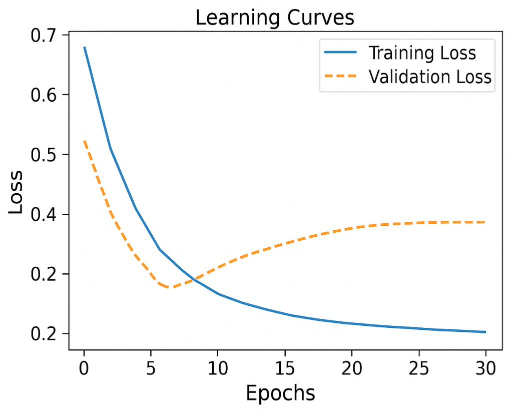

Figure 2: Example learning curves illustrating training and validation loss trends over epochs.

Components of a Typical Learning Curve Plot:

- **X-axis:** Epochs
- **Y-axis:** Loss (or accuracy)
- **Curves:** Training (solid line) and validation (dashed line)

Applications of Learning Curves:

- Tuning regularization strength (e.g., dropout rate, L2 penalty).
- Adjusting model capacity (number of layers or units).
- Determining when to apply early stopping or learning rate scheduling.

8.3 Tracking Experiments

Experiment tracking helps manage reproducibility, comparison, and collaboration in deep learning projects. Without tracking, tuning becomes inefficient and error-prone.

What to Track:

- Hyperparameter configurations (learning rate, batch size, optimizer, etc.)
- Model architecture (layers, activations, etc.)

- Metrics: training/validation/test loss and accuracy

Benefits:

- Visual comparisons between runs
- Automatic logging of metrics
- Easy sharing with collaborators

8.4 Hyperparameter Robustness

A robust model performs well across a range of hyperparameter values, not just a narrow peak. Robustness is critical for model reliability in production environments where retraining or data shifts may occur.

Key Concepts:

- **Flat Minima:** Regions in the loss surface where small changes in weights (and hence hyperparameters) do not degrade performance significantly.
- **Sensitivity Analysis:** Assessing how validation performance changes with small hyperparameter perturbations.
- **Regularization and Robustness:** Dropout, label smoothing, and weight decay typically enhance robustness.

.

Practical Tips:

- Prefer configurations where performance is stable over small changes.
- Use cross-validation when feasible to test generalization across data splits.
- Run tuning experiments with different random seeds and initializations.

Visualization: Contour plots of validation loss as a function of learning rate and batch size can reveal sensitivity patterns and robustness zones.

9 Conclusion

This survey presented a structured and in-depth exploration of hyperparameter tuning, regularization, and optimization techniques in deep neural networks. Beginning with the theoretical foundations, we categorized hyperparameters into architecture, training, regularization, optimization, and exponential weighted average groups, highlighting their roles in model performance and generalization.

We discussed established and emerging regularization strategies, reviewed practical optimization algorithms such as SGD and Adam, and provided guidelines for choosing hyperparameter values, leveraging learning curves, and ensuring robustness through experiment tracking and tuning strategies.

As neural networks continue to scale, understanding and managing hyperparameters remain crucial for building efficient and reliable deep learning models. We hope this paper offers valuable insights and serves as a practical reference for researchers and practitioners alike.

10 Acknowledgements

I would like to sincerely thank the open-source deep learning community and educational platforms that have contributed to my learning journey. In particular, the DeepLearning.AI team and the Stanford CS230 course have been instrumental in shaping the foundational knowledge behind this work.

Gratitude is also extended to the developers of frameworks like TensorFlow, PyTorch, and tools such as Weights & Biases, which made experimentation and visualization efficient and insightful.

This paper is the second in my independent research survey series, and I look forward to continuing this academic pursuit with future explorations in the field.

References

[1] Andrew Ng. *Machine Learning Specialization.* DeepLearning.AI.

[2] Andrew Ng. *Deep Learning Specialization.* DeepLearning.AI.

[3] Andrew Ng, Kian Katanforoosh. *CS230: Deep Learning*, Stanford University.

[4] Wikipedia contributors. *Hyperparameter, Optimizer, and Regularization pages.*

[5] Medium Authors. *Deep Learning and Optimization Articles.* Towards Data Science, Medium.