# Regularization and Optimizer

KUCSEPotato
a23493307@gmail.com

June 2025

*This paper is written for a project that develops ML/DL models. Since I am a learner, not an expert, if there are any errors, please kindly let me know.*

## 1 Introduction

Regularization and Optimizer are essential components in the machine learning and deep learning model training process. In this paper, I will provide a detailed overview of various regularization techniques and optimization strategies, explain their mathematical formulations, and discuss their roles in state-of-the-art models such as Convolutional Neural Networks (CNNs) and Transformers including Vision Transformers (ViTs).

## 2 Regularization

Regularization refers to a set of techniques designed to improve the generalization ability of models by preventing overfitting. Overfitting occurs when a model learns the noise and specific patterns of the training data instead of capturing the underlying distribution, leading to poor performance on unseen data. Regularization methods such as $L_1$, $L_2$ penalties, dropout, label smoothing, and stochastic depth help constrain the model's capacity or encourage smoother solutions, thereby enabling the model to generalize better to new data.

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

The above equation represents a loss function incorporating regularization. It illustrates how regularization is applied within the loss computation. In detail, the term $\lambda R(W)$ denotes the regularization component, where $R(W)$ is a regularization function (e.g., $L_1$ or $L_2$ norm of the weights), and $\lambda$ is a hyperparameter that controls the strength of the regularization. This term penalizes complex or large models by adding a constraint to the weight parameters, thereby helping to prevent overfitting and improving generalization performance. Briefly, regularization prevents the model from doing too well on training data.

## 2.1 $L_1$ Regularization

$L_1$ regularization is a technique used to prevent overfitting by adding a penalty proportional to the sum of the absolute values of the model weights. The modified loss function is given by:

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda \|W\|_1$$

Here,

- $N$ is the total number of training samples,
- $x_i$ is the input of the $i$-th sample,
- $y_i$ is the corresponding label,
- $f(x_i, W)$ is the model output (prediction) for input $x_i$ with parameters $W$,
- $L_i$ is the individual loss function (e.g., cross-entropy, MSE) for the $i$-th sample,
- $\|W\|_1 = \sum_j |W_j|$ is the $L_1$ norm of the weight vector $W$,
- $\lambda > 0$ is the regularization coefficient controlling the penalty strength.

This form of regularization encourages sparsity by pushing some weights toward exactly zero, which can lead to simpler, more interpretable models and can implicitly perform feature selection.

## 2.2 $L_2$ Regularization

$L_2$ regularization, also known as weight decay or ridge regularization, adds a penalty proportional to the sum of the squared values of the weights:

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda \|W\|_2^2$$

In this formulation:

- $\|W\|_2^2 = \sum_j W_j^2$ is the squared $L_2$ norm of the weights.

Unlike $L_1$, $L_2$ regularization does not encourage sparsity, but rather penalizes large weights more strongly. This leads to a smoother solution by distributing weights more evenly and avoiding over-reliance on any particular input feature.

## 2.3 Elastic Net

Elastic Net regularization is a combination of $L_1$ and $L_2$ penalties:

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda_1 \|W\|_1 + \lambda_2 \|W\|_2^2$$

where:

- $\lambda_1$, $\lambda_2$ are the weights for the $L_1$ and $L_2$ penalties, respectively.

Elastic Net provides a trade-off between sparsity and smoothness. It is particularly effective when input features are correlated or when the number of features exceeds the number of samples.

## 2.4 Dropout

Dropout is a stochastic regularization technique applied during training, where neurons are randomly "dropped" (i.e., their output set to zero) with a probability $p$:

$$\tilde{h}_j = h_j \cdot z_j, \quad z_j \sim \text{Bernoulli}(1-p)$$

Here:

- $h_j$ is the activation of neuron $j$,
- $z_j$ is a binary mask (0 or 1) sampled from a Bernoulli distribution,
- $p$ is the dropout rate, typically between 0.1 and 0.5.

During inference, dropout is disabled and all neurons are used. Their outputs are scaled by $(1-p)$ to maintain expected values between training and testing.

## 2.5 Batch Normalization

Batch Normalization normalizes the inputs to each layer based on the statistics of the current mini-batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta$$

Where:

- $x_i$ is the input activation to be normalized,
- $\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$ is the mini-batch mean,
- $\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$ is the mini-batch variance,
- $\epsilon$ is a small constant for numerical stability,
- $\gamma$, $\beta$ are learnable affine parameters.

This normalization stabilizes and accelerates training and often leads to better generalization. Though not originally designed for regularization, it has a regularizing effect similar to dropout.

## 2.6 Stochastic Depth

Stochastic Depth is used primarily in very deep residual networks. During training, residual blocks are randomly skipped:

$$\text{output} = \begin{cases} x + f(x), & \text{with probability } p \\ x, & \text{with probability } 1 - p \end{cases}$$

Here:

- $x$ is the input to the residual block,
- $f(x)$ is the transformation applied by the residual block,
- $p$ is the probability of keeping the block (survival rate).

This method prevents overfitting by introducing stochasticity and enables the training of very deep architectures (e.g., ¿1000 layers) without degradation in performance.

## 2.7 Label Smoothing

Label Smoothing is a regularization technique used in classification tasks to prevent the model from becoming overly confident in its predictions. It modifies the target label distribution by replacing hard one-hot vectors with softened probability distributions. The modified loss function is defined as:

$$\mathcal{L}_i = -\sum_{k=1}^{K} y_i^{\text{smooth}}(k) \log \hat{p}_i(k)$$

Here,

- $K$ is the total number of classes,
- $y_i^{\text{smooth}}(k)$ is the smoothed label probability assigned to class $k$ for the $i$-th sample,
- $\hat{p}_i(k)$ is the model's predicted probability for class $k$ on sample $i$,
- $\varepsilon$ is the label smoothing factor, a small positive scalar (e.g., 0.1),
- $y_i$ is the ground-truth class index for sample $i$.

The smoothed label distribution is computed as:

$$y_i^{\text{smooth}}(k) = \begin{cases} 1 - \varepsilon, & \text{if } k = y_i \\ \frac{\varepsilon}{K-1}, & \text{otherwise} \end{cases}$$

This approach distributes a small portion of the target probability mass across all classes, rather than assigning full probability to the correct class. As a result, it prevents the model from outputting extreme confidence scores (i.e., probability 1.0), and instead encourages learning more calibrated and generalizable probability distributions.

## 2.8 When to Use Each Regularization Technique

Different regularization techniques offer complementary benefits depending on the nature of the dataset, model architecture, and training regime. The following guidelines summarize when each method is most appropriate:

- $L_1$ **Regularization:** Recommended when the goal is to achieve model sparsity or perform automatic feature selection. It is especially useful in high-dimensional settings where only a subset of input features are expected to be informative (e.g., text classification or genomics).
- $L_2$ **Regularization:** Suited for general-purpose regularization when all features are potentially useful. It is effective in preventing large weight magnitudes and is commonly used in conjunction with gradient-based optimizers (e.g., SGD, AdamW). Often applied in computer vision and speech tasks.
- **Elastic Net:** Ideal when features are highly correlated or when both sparsity and smoothness are desired. Frequently used in tabular data problems or regression settings with collinear variables.
- **Dropout:** Particularly effective in fully connected networks or small to mid-sized convolutional networks, where co-adaptation between neurons can be a problem. It acts as an ensemble method and is most beneficial when training data is limited or noisy.
- **Batch Normalization:** Used to stabilize training and accelerate convergence. It is most useful in deep feedforward and convolutional networks with large batch sizes. Though not designed as a regularizer, it often reduces the need for dropout.
- **Stochastic Depth:** Recommended for very deep residual networks (e.g., ResNet-152 or deeper). It reduces the effective depth during training while maintaining full capacity at inference, leading to better generalization and convergence.
- **Label Smoothing:** Useful in classification tasks involving a large number of classes or noisy labels (e.g., ImageNet, language modeling). It improves calibration of predicted probabilities and mitigates overconfident predictions.

In practice, combinations of these techniques are often used. For example, modern deep learning models may simultaneously apply label smoothing, batch normalization, and $L_2$ regularization. The choice of regularization method should be guided by the model architecture, dataset size, label quality, and computational budget.

# 3 Optimizer

Optimizers are algorithms that update the parameters of a model in order to minimize a given loss function. The choice of optimizer significantly influences the convergence speed, stability, and overall performance of the learning process. Gradient Descent, Stochastic Gradient Descent (SGD), Momentum, RMSProp, Adam, and AdamW are some of the most commonly used optimization algorithms in modern deep learning. Each optimizer incorporates different strategies to handle learning rate adjustment, gradient smoothing, and parameter regularization.

## 3.1 Gradient Descent

Gradient Descent is the most basic optimization algorithm. It updates the model's parameters by computing the gradient of the loss function with respect to the parameters and moving in the opposite direction of the gradient. This direction indicates where the loss decreases the fastest.

**Concept:** Imagine you are at the top of a hill (high loss) and want to reach the bottom (low loss). The gradient tells you which direction is steepest down. Gradient Descent takes a small step in that direction.

$$\theta = \theta - \eta \cdot \nabla_\theta L$$

- $\theta$: current model parameters,
- $\eta$: learning rate (step size),
- $\nabla_\theta L$: gradient of the loss with respect to $\theta$.

**Limitations:** Requires computing gradients over the entire dataset, which can be slow and inefficient for large datasets.

## 3.2 Stochastic Gradient Descent (SGD)

To improve efficiency, SGD computes gradients using only a small subset (mini-batch) of the training data. This allows the model to learn faster, although the direction of each step is more noisy.

$$\theta = \theta - \eta \cdot \nabla_\theta L_i$$

- $L_i$: loss computed from a single example or mini-batch.

**Benefit:** Faster and uses less memory.
**Drawback:** Updates can fluctuate due to random sampling, which sometimes helps escape poor local minima.

### 3.3 Momentum

Momentum builds upon SGD by adding memory. Instead of only relying on the current gradient, it also considers past gradients, which helps smooth the updates and accelerate convergence.

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta L_t, \quad \theta = \theta - v_t$$

- $v_t$: accumulated velocity (moving average of gradients),
- $\gamma$: momentum coefficient (usually 0.9).

**Concept:** Like pushing a ball downhill—it accelerates as it rolls, especially when the slope is consistent.

### 3.4 RMSProp

RMSProp adapts the learning rate for each parameter individually based on how large its gradients have been. Parameters with consistently large gradients get smaller learning rates, and those with small gradients get larger learning rates.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)(\nabla_\theta L)^2$$
$$\theta = \theta - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot \nabla_\theta L$$

- $\rho$: decay rate (typically 0.9),
- $E[g^2]_t$: running average of squared gradients.

**Used in:** Recurrent Neural Networks (RNNs) and situations where gradient scales vary widely.

### 3.5 Adam (Adaptive Moment Estimation)

Adam combines the strengths of Momentum and RMSProp. It maintains a moving average of both the gradients and the squared gradients to adapt learning rates while smoothing updates.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta L)^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta = \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

**Advantages:**

- Fast convergence,
- Adaptive learning rates,
- Works well with sparse data and deep networks.

### 3.6 AdamW

AdamW is a variant of Adam that corrects how weight decay (regularization) is applied. In Adam, weight decay is mixed with gradient updates, which can interfere with optimization. AdamW separates the two, improving generalization.

$$\theta = \theta - \eta \cdot \left( \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta \right)$$

- $\lambda$: weight decay coefficient.

**Used in:** Transformer-based models like BERT, ViT.
**Benefit:** Better generalization, now standard in many frameworks.

# 4 Conclusion

In this paper, I have reviewed essential regularization and optimization techniques that are commonly used in modern machine learning and deep learning models. These methods play a crucial role in improving training stability, accelerating convergence, and enhancing generalization performance.

Regularization techniques such as $L_1/L_2$ penalties, Dropout, Batch Normalization, Label Smoothing, and Stochastic Depth each address different challenges such as overfitting, unstable gradients, and overconfident predictions. Similarly, optimization algorithms—including SGD, Momentum, RMSProp, Adam, and AdamW—offer different strategies for efficient and robust parameter updates.

In particular, in the context of Transformer-based architectures and Vision Transformers (ViTs), the following combinations have proven to be highly effective:

- **Optimizer: AdamW** is the most widely used optimizer in Transformer-based models. It decouples weight decay from the gradient update and leads to better generalization, especially in large-scale training scenarios.
- **Regularization:**
  - **Layer Normalization** is used instead of Batch Normalization, as it is more suitable for sequence data and independent of batch size.
  - **Label Smoothing** is commonly applied to prevent overconfidence in classification tasks.
  - **Stochastic Depth** is often adopted in deep Vision Transformers to improve generalization and ease optimization.
  - **Dropout** is still employed in various parts of the model, such as attention and feedforward layers.
- **Learning Rate Scheduling:** Warm-up followed by cosine annealing or linear decay is typically used to stabilize early training and improve convergence.

As deep learning models continue to grow in depth and complexity, the importance of choosing appropriate regularization and optimization strategies becomes increasingly critical. Understanding and applying these techniques appropriately is essential for building models that are both effective and robust.

Future work may explore combinations of these methods under different training regimes, as well as novel approaches tailored to emerging architectures such as sparse transformers, diffusion models, or multi-modal learning systems.

# A Basic Terminology in ML/DL

This appendix introduces essential terminology frequently encountered in the field of machine learning (ML) and deep learning (DL). Each term is briefly defined to help readers unfamiliar with the field better understand the concepts discussed in the main text.

**Model** A function or structure that maps inputs to outputs. In ML/DL, a model learns parameters to minimize error.

**Training** The process of adjusting the model's parameters using data and a loss function to improve performance.

**Loss Function** A function that quantifies the error between the predicted and actual values. Common examples include Mean Squared Error (MSE) and Cross-Entropy Loss.

**Optimizer** An algorithm used to minimize the loss function by updating model parameters. Examples include SGD and Adam.

**Epoch** One full pass over the entire training dataset during learning.

**Batch / Mini-batch** A subset of the training data used to compute gradients and update model parameters.

**Overfitting** A situation where the model fits the training data too well and performs poorly on unseen data.

**Underfitting** When the model is too simple to capture the underlying patterns in the data.

**Activation Function** A non-linear function applied at each neuron in a neural network. Examples: ReLU, Sigmoid, Tanh.

**Gradient** The derivative of the loss with respect to model parameters, used to guide optimization.

**Backpropagation** The algorithm for computing gradients in a neural network by propagating errors backward from the output.

**Learning Rate** A hyperparameter that determines the step size when updating model parameters.

**Regularization** Techniques used to prevent overfitting by adding penalty terms to the loss function (e.g., L1, L2, Dropout).

**Neural Network** A model consisting of interconnected layers of artificial neurons that transform inputs to outputs.

**Deep Learning** A subfield of ML involving neural networks with many hidden layers, capable of modeling complex data representations.

# B Activation Functions

Activation functions are essential components of neural networks that introduce non-linearity into the model. Without them, the network would behave like a simple linear model, regardless of its depth. This appendix presents commonly used activation functions, including their definitions, properties, and usage scenarios.

## B.1 Sigmoid Function

The Sigmoid function maps any real-valued input into the range $(0, 1)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

**Properties:**

- Output range: $(0, 1)$
- Smooth and differentiable
- Commonly used in binary classification (e.g., logistic regression)

**Drawbacks:**

- Saturates at both ends (vanishing gradients)
- Output not zero-centered

## B.2 Tanh Function

The Tanh function is similar to sigmoid but outputs values between $(-1, 1)$:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

**Properties:**

- Output range: $(-1, 1)$
- Zero-centered output (advantage over sigmoid)

**Drawbacks:**

- Also suffers from vanishing gradients for large input magnitudes

## B.3 ReLU (Rectified Linear Unit)

ReLU is one of the most widely used activation functions in deep learning:

$$\text{ReLU}(x) = \max(0, x)$$

**Properties:**

- Output range: $[0, \infty)$
- Simple and efficient to compute
- Sparse activation: only positive values pass

**Drawbacks:**

- Can cause "dead neurons" if inputs are negative for all training data

## B.4 Leaky ReLU

Leaky ReLU addresses the dying ReLU problem by allowing a small gradient when the input is negative:

$$\text{Leaky ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$

Typical value: $\alpha = 0.01$

## B.5 Softmax

Softmax is used in the output layer of multi-class classification problems. It transforms a vector of raw scores (logits) into a probability distribution:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

**Properties:**

- Output values between $(0, 1)$
- All outputs sum to 1 (probability distribution)

## B.6 Summary Table

| Name | Output Range | Differentiable? | Usage |
|---|---|---|---|
| Sigmoid | (0, 1) | Yes | Binary classification |
| Tanh | (-1, 1) | Yes | RNNs, classical networks |
| ReLU | [0, $\infty$) | Yes (except at 0) | CNNs, feedforward networks |
| Leaky ReLU | $(-\infty, \infty)$ | Yes | Deep CNNs, ResNets |
| Softmax | (0, 1) | Yes | Output layer for multi-class |

# C PyTorch Implementation Examples

This appendix provides practical implementation examples of the regularization and optimization techniques discussed in the main text, using the PyTorch framework. Each example is accompanied by a short explanation for clarity.

## C.1 Regularization Techniques

### C.1.1 $L_2$ Regularization (Weight Decay)

**Description:** Adds a penalty to large weights by applying $L_2$ norm regularization via the optimizer.

```
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.01,
    weight_decay=1e-4  # L2 regularization strength
)
```

Listing 1: L2 regularization using weight_decay in SGD

### C.1.2 $L_1$ Regularization (Manual)

**Description:** PyTorch does not directly support $L_1$ regularization via optimizers. You can manually add the $L_1$ term to the loss function.

```
l1_lambda = 1e-5
l1_norm = sum(p.abs().sum() for p in model.parameters())
loss = base_loss + l1_lambda * l1_norm
```

Listing 2: Manual L1 regularization

### C.1.3 Dropout

**Description:** Randomly zeroes some neuron outputs during training for regularization.

```
model = nn.Sequential(
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(p=0.5),  # 50% dropout
    nn.Linear(64, 10)
)
```

Listing 3: Dropout layer in a sequential model

### C.1.4 Batch Normalization

**Description:** Normalizes layer outputs for each mini-batch to stabilize training.

```
model = nn.Sequential(
    nn.Linear(128, 64),
    nn.BatchNorm1d(64),  # Normalize over batch
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

Listing 4: BatchNorm1d after Linear layer

### C.1.5 Label Smoothing

**Description:** Smooths target labels to prevent overconfidence during classification.

```
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
```

Listing 5: CrossEntropyLoss with label smoothing

### C.1.6 Stochastic Depth

**Description:** Randomly drops entire residual paths during training to improve generalization in deep networks. (Requires timm library)

```
from timm.models.layers import DropPath

class MyBlock(nn.Module):
    def __init__(self, drop_prob=0.2):
        super().__init__()
        self.block = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU()
        )
        self.drop_path = DropPath(drop_prob)

    def forward(self, x):
        return x + self.drop_path(self.block(x))
```

Listing 6: Stochastic Depth via DropPath from timm

## C.2 Optimization Algorithms

### C.2.1 SGD

**Description:** Performs standard stochastic gradient descent with optional momentum.

```
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.01,
    momentum=0.9
)
```

Listing 7: SGD with momentum

### C.2.2 RMSProp

**Description:** Adapts learning rate for each parameter using a moving average of squared gradients.

```
optimizer = torch.optim.RMSprop(
    model.parameters(),
    lr=0.01,
    alpha=0.9  # decay rate
)
```

Listing 8: RMSProp optimizer

### C.2.3 Adam

**Description:** Combines momentum and adaptive learning rate per parameter.

```
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=0.001,
    betas=(0.9, 0.999)
)
```

Listing 9: Adam optimizer

### C.2.4 AdamW

**Description:** Adam with decoupled weight decay. Preferred for Transformer architectures.

```
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.001,
    weight_decay=1e-2
)
```

Listing 10: AdamW optimizer with weight decay