Regularization Techniques in Multilayer Perceptrons

Naman Goel

January 7, 2026

Abstract

Regularization techniques constitute essential tools for preventing overfitting and improving generalization in neural networks. This report presents a rigorous examination of L1 and L2 regularization, dropout, batch normalization, layer normalization, and early stopping. Each technique is motivated through bias-variance analysis and supported with mathematical foundations, practical implementations, and empirical guidance.

1 Introduction

The fundamental challenge in machine learning is to construct models that generalize well to unseen data. While a sufficiently complex model can fit training data perfectly through memorization, such models typically fail on test data. Regularization techniques encourage simpler models or more robust learning processes, trading some training accuracy for improved generalization.

2 The Bias-Variance Tradeoff

2.1 Formal Characterization

Prediction error can be decomposed into three components: bias, variance, and irreducible error. Bias represents systematic error from using a simplified model that cannot capture the true relationship. Variance represents sensitivity to fluctuations in training data.

Simpler models exhibit high bias (systematically wrong) but low variance (stable across different training sets). Complex models exhibit low bias (can fit complex patterns) but high variance (unstable across different training sets). Regularization effectively increases model bias to reduce variance, optimizing the tradeoff.

2.2 Regularization as Bias Inducement

By constraining the complexity of the model, regularization induces bias toward simpler solutions. Although training error increases, test error typically decreases through variance reduction. This is the central principle underlying all regularization techniques.

3 L2 Regularization (Weight Decay)

3.1 Motivation and Formulation

Large weights amplify input signals, enabling the network to implement complex, nonlinear decision boundaries. Conversely, small weights tend to produce smoother, simpler decision boundaries. L2 regularization penalizes large weights, encouraging simpler models.

The regularized loss function augments the original loss with a penalty term proportional to the squared magnitude of all weights:

$$\mathcal{L}_{\text{regularized}} = \mathcal{L}_{\text{original}} + \lambda \sum_{\mathbf{W}} \|\mathbf{W}\|_F^2 \tag{1}$$

where λ is the regularization strength hyperparameter and $\|\mathbf{W}\|_F^2$ denotes the Frobenius norm (sum of squared elements). The Frobenius norm is convenient for computation and provides consistent penalization.

3.2 Effect on Optimization

When computing gradients of the regularized loss, the weight gradient includes an additional term proportional to the weights themselves:

$$\frac{\partial \mathcal{L}_{\text{regularized}}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}_{\text{original}}}{\partial \mathbf{W}} + 2\lambda \mathbf{W} \tag{2}$$

The update rule consequently includes a term decaying weights toward zero independent of data gradients:

$$\mathbf{W} \leftarrow (1 - 2\lambda\eta)\mathbf{W} - \eta \frac{\partial \mathcal{L}_{\text{original}}}{\partial \mathbf{W}} \tag{3}$$

This multiplicative decay factor (less than one when $\lambda > 0$) encourages weights toward zero while still allowing increases when data signals justify them.

3.3 Implementation

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

model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)

# L2 regularization: weight_decay parameter
optimizer = optim.Adam(
    model.parameters(),
    lr=0.001,
    weight_decay=0.0001  # Lambda = 0.0001
)

criterion = nn.CrossEntropyLoss()
```

4 L1 Regularization (Lasso)

4.1 Sparse Solutions Through Absolute Penalty

L1 regularization penalizes the sum of absolute values of weights:

$$\mathcal{L}_{\text{L1}} = \mathcal{L}_{\text{original}} + \lambda \sum_{\mathbf{W}} |\mathbf{W}| \tag{4}$$

The crucial distinction from L2 is that the penalty on each weight is linear rather than quadratic. This creates a qualitatively different optimization landscape where exact zeros are favored solutions.

4.2 Geometric Interpretation

Geometrically, L2 regularization constrains weights to lie within a ball (circular constraint), while L1 constrains them to lie within a diamond-shaped region. The corners of the diamond correspond to sparse solutions where many coordinates are exactly zero. During optimization, solutions tend toward these corners.

4.3 Sparsity Benefits

Sparse networks where many weights are zero provide multiple benefits: reduced memory requirements, faster inference through skipping zero computations, and inherent feature selection where unused features receive zero weights. However, implementing sparse operations efficiently requires specialized techniques.

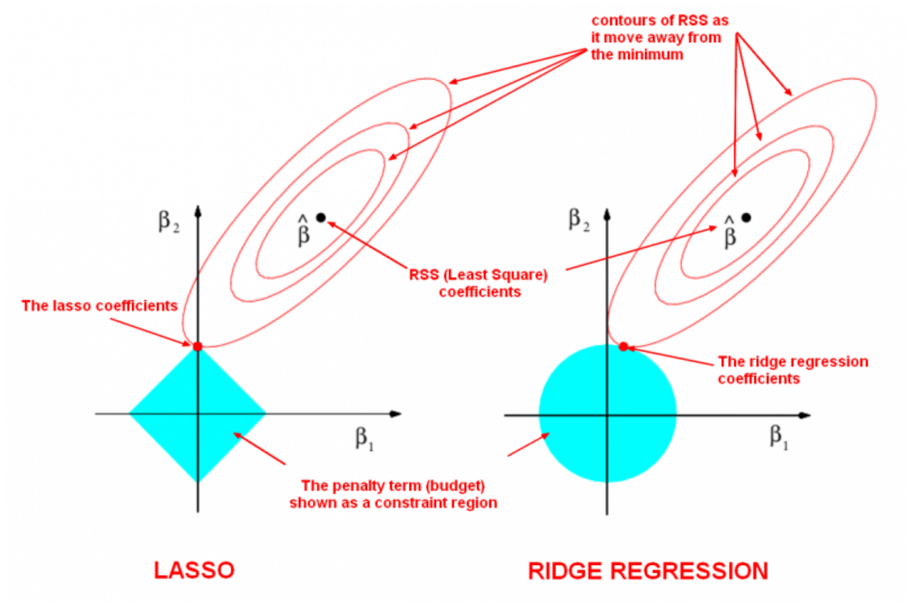
contours of RSS as it move away from the minimum
β_2
$\hat{\beta}$
The lasso coefficients
RSS (Least Square) coefficients
β_2
$\hat{\beta}$
The ridge regression coefficients
β_1
β_1
The penalty term (budget) shown as a constraint region
LASSO
RIDGE REGRESSION

Figure 1: Geometric Interpretation of L1 and L2 regularization

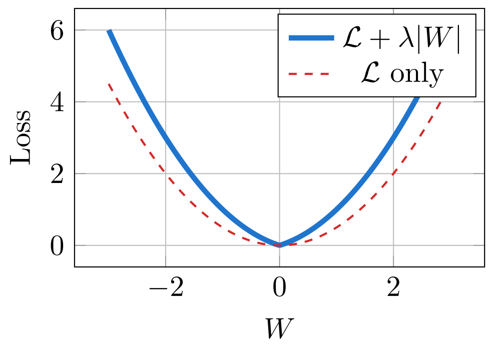

Figure 2: L1 regularization creates a sharp corner at $W = 0$, encouraging sparse solutions. The data loss (dashed) is quadratic; L1 adds the V-shaped term.

5 Dropout: Regularization Through Stochasticity

5.1 Principle

Dropout randomly deactivates neurons during training with a fixed probability p. During inference, all neurons are active but their outputs are scaled to account for the probabilistic averaging during training.

5.2 Mechanism

In forward propagation, each neuron independently has probability p of being zeroed and probability $1 - p$ of being multiplied by $1/(1 - p)$. This scaling ensures that the expected value of each unit's contribution remains constant, which is critical for inference consistency.

The scaling can be applied either during training (inverted dropout) or during inference, though inverted dropout is more common in practice.

5.3 Why Dropout Regularizes

Dropout prevents learned co-adaptation of neurons. If neurons depend on specific partners activating, those partnerships must be robust to the absence of partners. The network effectively trains many thinned subnetworks sharing parameters. At test time, predictions from all subnetworks are averaged (through the multiplicative scaling), similar to ensemble averaging.

This interpretation reveals why dropout helps generalization: ensemble methods reduce variance by combining diverse models. Dropout creates diversity through different random architectures during training.

5.4 Implementation

```python
import torch.nn as nn

class DropoutNetwork(nn.Module):
    def __init__(self, dropout_rate=0.5):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.dropout1 = nn.Dropout(p=dropout_rate)
        self.fc2 = nn.Linear(256, 256)
        self.dropout2 = nn.Dropout(p=dropout_rate)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout1(x)  # Randomly deactivate
        x = torch.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x

# Important: switch modes
model.train()   # Dropout active
model.eval()    # Dropout inactive
```

6 Batch Normalization

6.1 Internal Covariate Shift

During training, the distribution of inputs to each layer changes as upstream layers' parameters are updated. This internal covariate shift forces each layer to continuously readapt to changing input distributions, slowing convergence.

6.2 Normalization Strategy

Batch normalization normalizes layer inputs to have zero mean and unit variance. For each layer and each mini-batch, it computes statistics across the batch dimension and normalizes accordingly. Subsequently, it applies learnable scaling and shifting parameters to allow the network to undo normalization if beneficial.

Mathematically, for each feature j within a mini-batch:

$$\text{mean}_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij} \tag{5}$$

$$\text{variance}_j = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \text{mean}_j)^2 \tag{6}$$

$$\text{normalized}_{ij} = \frac{x_{ij} - \text{mean}_j}{\sqrt{\text{variance}_j + \epsilon}} \tag{7}$$

Finally, learnable parameters γ_j and β_j provide affine transformation:

$$y_{ij} = \gamma_j \cdot \text{normalized}_{ij} + \beta_j \tag{8}$$

6.3 Training vs Inference

During training, normalization uses mini-batch statistics. During inference, using mini-batch statistics is inappropriate when batch size is small or the batch may not be representative. Instead, batch normalization maintains exponentially weighted moving averages of mean and variance computed during training, using these running statistics at inference time.

6.4 Regularization Effect

Beyond stabilizing training, batch normalization provides regularization benefits. The stochasticity introduced by computing statistics from mini-batches rather than the full training set acts as a regularizer, reducing generalization gap. This is why models with batch normalization often require less additional regularization.

6.5 Implementation

```python
import torch.nn as nn

class BatchNormNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)

        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)

        x = self.fc3(x)
        return x

# Critical: use correct mode
model.train()     # BatchNorm uses batch statistics
model.eval()      # BatchNorm uses running statistics
```

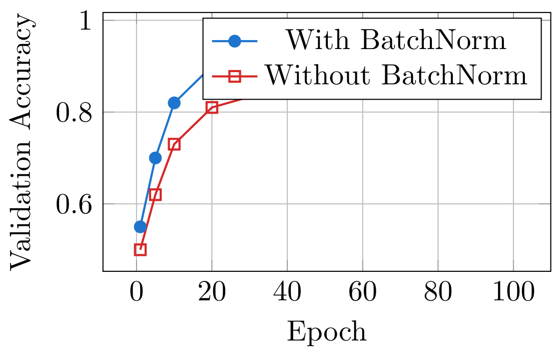

Figure 3: Batch normalization typically results in faster convergence and better generalization. A network without BatchNorm requires more epochs to reach the same validation accuracy.

7 Layer Normalization

7.1 Normalization Across Features Rather Than Batch

Layer normalization normalizes across the feature dimension rather than the batch dimension. For each sample independently, it computes mean and variance across all features and normalizes accordingly:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{9}$$

where μ and σ^2 are computed across features for a single sample, γ and β are learnable scale and shift parameters, and $\odot$ denotes element-wise multiplication.

7.2 Advantages in Specific Contexts

Layer normalization is superior to batch normalization in several important scenarios:

1. Small batch sizes: Batch statistics become unreliable with few samples
2. Recurrent neural networks: Batch normalization applies poorly across time steps
3. Variable-length sequences: Different sequence lengths complicate batch statistics
4. Transformer networks: Layer normalization has become standard in attention-based architectures

7.3 Implementation

```python
import torch.nn as nn

class LayerNormNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.ln1 = nn.LayerNorm(256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.ln1(x)  # Normalize per sample
        x = torch.relu(x)
        x = self.fc2(x)
        return x
```

8 Early Stopping

8.1 Principle

During training, validation loss typically decreases initially, reaches a minimum, then increases as the model begins overfitting to training data. Early stopping terminates training when validation performance plateaus or begins degrading.

8.2 Algorithm

The algorithm maintains a ”patience” counter. If validation loss improves by at least a minimum delta, the counter resets. If no improvement occurs for a number of consecutive epochs equal to patience, training terminates. The best model (at lowest validation loss) is restored.

8.3 Implementation

```
class EarlyStopping:
    def __init__(self, patience=10, min_delta=0.001):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.best_model_state = None

    def __call__(self, val_loss, model):
        if self.best_loss is None:
            self.best_loss = val_loss
            self.best_model_state = model.state_dict()
        elif val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.best_model_state = model.state_dict()
            self.counter = 0
        else:
            self.counter += 1

        if self.counter >= self.patience:
            model.load_state_dict(self.best_model_state)
            return True  # Stop training

        return False
```

9 Data Augmentation

9.1 Expansion Through Transformation

Data augmentation artificially expands the training set through task-preserving transformations. The key constraint is that augmentations must not change the true label. For images, rotations, crops, and color shifts preserve content while increasing dataset diversity.

9.2 Augmentation Strategies

1. Geometric: Rotations, translations, flips, scaling
2. Photometric: Brightness, contrast, saturation, color jitter
3. Advanced: Mixup (blending images), CutMix (cutting and pasting regions), AutoAugment (learned augmentation policies)

9.3 Regularization Mechanism

Augmentation acts as regularization by increasing effective dataset size and encouraging invariance to transformations. Networks trained with augmentation learn representations robust to variations seen in the augmented data.

9.4 Implementation

```python
from torchvision import transforms

train_transform = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

10 Combining Regularization Techniques

10.1 Synergistic Effects

Regularization techniques work synergistically. Batch normalization stabilizes training, allowing larger learning rates, while dropout prevents co-adaptation. L2 regularization penalizes magnitude while dropout ensures robustness. Early stopping prevents overtraining while data augmentation expands effective training data.

10.2 Practical Guidelines

A well-balanced regularization approach typically includes:

1. Batch normalization in hidden layers for training stability
2. Dropout (rate 0.3-0.5) before dense layers
3. L2 regularization with strength 0.0001 to 0.001
4. Data augmentation appropriate to the task
5. Early stopping with patience 10-20 epochs

10.3 Hyperparameter Tuning

Technique	Typical Range	Tuning Strategy
L2 strength	10^{-5} to 10^{-3}	Grid search
Dropout rate	0.3 to 0.7	Increase if overfitting
Batch norm momentum	0.1 to 0.9	Default 0.9 usually works
Early stopping patience	5 to 30	Dataset dependent

Table 1: Regularization hyperparameter ranges and tuning guidance.

11 Conclusion

Regularization techniques are indispensable for practical deep learning. L1 and L2 regularization encourage simplicity through weight constraints. Dropout prevents co-adaptation through stochastic deactivation. Batch normalization stabilizes training while providing regularization. Early stopping prevents overtraining. Data augmentation expands effective training data. The combination of these techniques enables construction of models that both achieve high training accuracy and maintain strong generalization to unseen data.

References

- Ioffe, S., Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. JMLR, 15(56), 1929-1958.
- Ba, J. L., Kiros, J. R., Hinton, G. E. (2016). Layer normalization. arXiv preprint arXiv:1607.06450.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep learning. MIT press.
- Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society.