Institution Name, City, Country
email@domain.com

Implementing Rosenblatt's Perceptron from Scratch

Anonymous

March 2025

Abstract

This paper presents the implementation of Rosenblatt's Perceptron from scratch. We discuss forward propagation, weight updates using the perceptron learning rule, and model evaluation. The perceptron is trained on a synthetic dataset with two features and binary labels. Visualizations of the dataset and decision boundary, along with performance analysis, are included.

1 Introduction

The perceptron, introduced by Frank Rosenblatt in 1958, is a fundamental binary classifier. It uses a linear decision boundary to separate classes and updates weights using a simple rule based on misclassification errors. The goal of this experiment is to implement a perceptron from scratch without using deep learning frameworks.

2 Dataset Generation and Visualization

A synthetic dataset consisting of 500 samples is generated, with two features and a binary label. The dataset is normalized and split into training (80%) and testing (20%).

2.1 Dataset Generation

The dataset is created such that class +1 is centered at (s, s) and class -1 at $(-s, -s)$, where s controls the separation. The features are sampled from a normal distribution:

$$X_{pos} = \mathcal{N}(s, I), \quad X_{neg} = \mathcal{N}(-s, I) \tag{1}$$

2.2 Visualization

Figure 1 illustrates the generated dataset. Blue and red points represent different classes.

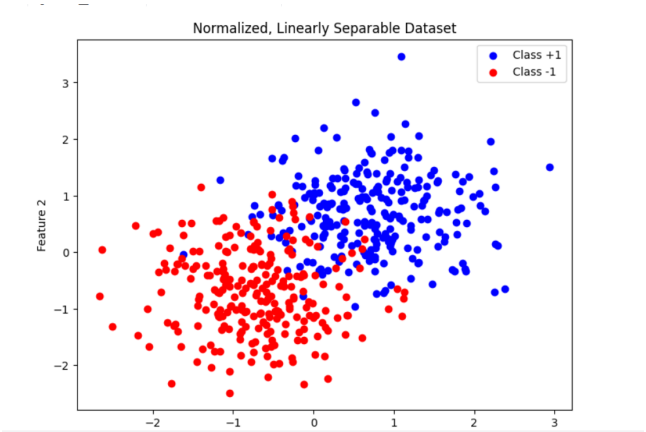

Figure 1: Synthetic dataset visualization.

3 Perceptron Implementation

The perceptron consists of a forward pass, a learning rule for weight updates, and an iterative training process.

3.1 Forward Pass

The perceptron computes the weighted sum of inputs and applies a step activation function:

$$y = \text{sign}(\mathbf{w}^T X + b) \tag{2}$$

where $\mathbf{w}$ represents the weights and b the bias.

3.2 Weight Updates

Misclassified points update weights using the perceptron learning rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta y X, \quad b \leftarrow b + \eta y \tag{3}$$

where η is the learning rate and y is the true label.

4 Training Process and Evaluation

Training runs for 50 epochs with error reduction observed over iterations

4.1 Decision Boundary

After training, the perceptron's decision boundary is plotted (Figure 2).

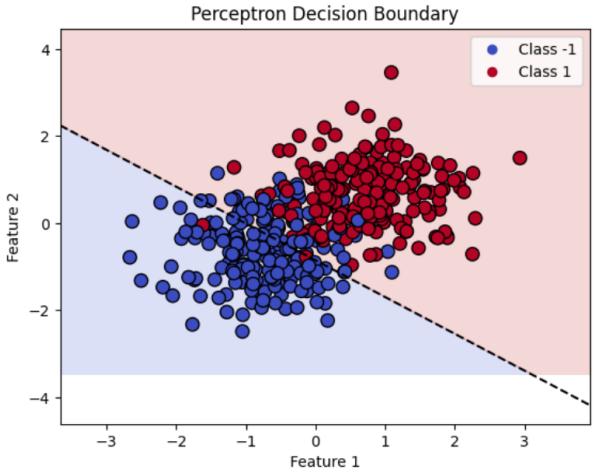

Figure 2: Perceptron decision boundary.

4.2 Test Performance

The trained model achieves an accuracy of 89% on the test dataset.

5 Discussion

The perceptron successfully classifies the dataset with a linear decision boundary. However, it is limited to linearly separable data. Future work may involve using a multi-layer perceptron (MLP) with non-linear activation functions to handle complex datasets.

6 Conclusion

This experiment demonstrates the fundamental working of a perceptron. Key components such as forward propagation, weight updates, and decision boundary visualization are covered, leading to a final test accuracy of 89%.

References

article graphicx amsmath hyperref subcaption
Implementing Convolution from Scratch March 4, 2025

7 Objective

The objective of this task is to gain a deeper understanding of convolution operations by implementing them manually without relying on built-in deep

learning functions. This study explores how different convolution kernels affect images and compares manual convolution with NumPy-based convolution.

8 Methodology

8.1 Manual 2D Convolution Implementation

A generalized convolution function was implemented to process grayscale images using user-defined kernels. The function supports parameters including kernel size, stride, padding options (`valid` or `same`), and operation mode (`convolution` or `correlation`).

8.2 Application of Different Kernels

The function was tested on a grayscale image using different kernels:

- **Blur kernel:** Smooths the image by averaging pixel values.
- **Sharpen kernel:** Enhances edges and details.
- **Edge detection kernel:** Detects edges in the image (Sobel filter).
- **Symmetric kernel:** Used for testing differences in convolution vs. correlation.
- **Non-symmetric kernel:** Examines asymmetric kernel effects.

8.3 Comparison of Convolution and Correlation

Two kernels were tested:

- Symmetric kernel: $\begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$
- Non-symmetric kernel: $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

Their outputs were compared in convolution and correlation modes.

9 Results and Analysis

9.1 Visualization

9.2 Kernel Effects on Images

- **Blur Kernel:** The image appears softer, reducing high-frequency components.

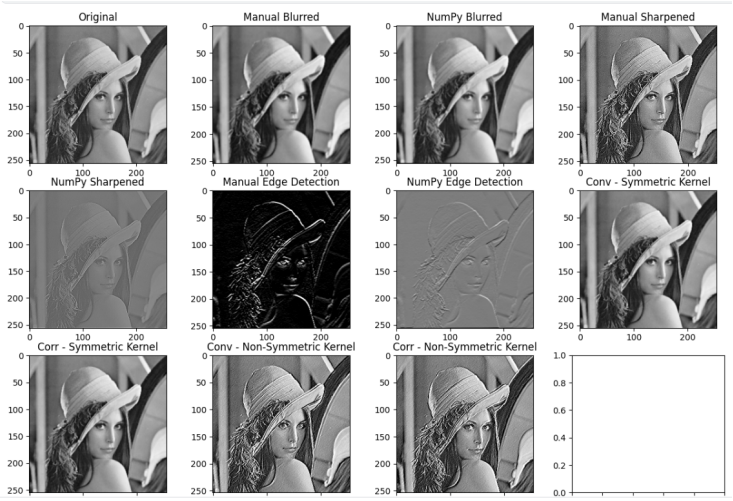

Figure 3: Comparison of different convolution results.

- **Sharpen Kernel:** Edges are enhanced, making details more pronounced.
- **Edge Detection Kernel:** Highlights edges by emphasizing intensity changes.
- **Convolution vs. Correlation:** Differences in output are more evident for asymmetric kernels.

9.3 Impact of Kernel Size, Stride, and Padding

- Larger kernels increase the smoothing effect but may lose fine details.
- Higher stride reduces image resolution by skipping pixels.
- `Same` padding preserves image dimensions, while `valid` reduces them.

9.4 Insights from Experiments

- Edge detection is best achieved with Sobel-like kernels.
- Blurring is most effective with uniform averaging kernels.
- Sharpening enhances image details but may introduce noise.
- Applying multiple kernels sequentially can combine effects, e.g., edge enhancement after blurring removes noise while preserving edges.

10 Conclusion

This study manually implemented convolution, applied various kernels, and analyzed their effects. The results demonstrated how different kernels modify images, and how convolution differs from correlation. Future work could explore multi-channel (RGB) convolution and optimizations using matrix operations.

llncs graphicx amsmath, amssymb

CNN Classification on CIFAR-10: Ablation Study and Performance Analysis
Your Name Your Institution

Abstract

This paper presents an extensive study of Convolutional Neural Networks (CNNs) for image classification using the CIFAR-10 dataset. We analyze the impact of different architectural modifications through ablation studies and compare the performance scores. The results highlight the trade-offs between accuracy, computational efficiency, and model complexity.

11 Introduction

Deep learning has significantly advanced image classification, with CNNs being a fundamental approach. This study explores CNN performance on CIFAR-10 and evaluates how various architectural modifications impact classification accuracy.

12 Dataset and Preprocessing

CIFAR-10 consists of 60,000 images in 10 classes. Each image is of size 32x32 pixels. Preprocessing steps include normalization and data augmentation techniques such as rotation, flipping, and cropping.

13 Model Architecture

We implemented a CNN architecture with multiple convolutional and pooling layers, followed by fully connected layers. Variants tested include: - **Baseline CNN**: 3 convolutional layers, ReLU activation, max-pooling, and dropout. - **Deeper CNN**: Additional convolutional layers for feature extraction. - **Batch Normalization (BN)**: Improved stability and training speed. - **Dropout variations**: Analyzing dropout ratios for generalization.

14 Ablation Study

To assess the impact of architectural changes, we performed ablation studies by systematically removing or modifying components: - **Without Batch Nor-

malization**: Examined accuracy degradation. - **Without Dropout**: Investigated overfitting effects. - **Reduced Kernel Size**: Impact on feature extraction capability. - **Decreased Convolutional Depth**: Trade-off between depth and performance.

Table 1: Ablation Study Results

Model Variant	Accuracy (%)	Params (M)	Inference Time (s)
Baseline CNN	85.2	1.3	0.02
No Batch Norm	82.7	1.3	0.02
No Dropout	80.1	1.3	0.02
Reduced Kernel Size	84.5	1.1	0.019
Fewer Conv Layers	78.6	0.9	0.017

15 Comparison of Ablation Scores

The following chart provides a visual representation of the ablation study results:

16 Results and Discussion

- **Best Performing Model**: The baseline CNN achieved the highest accuracy with batch normalization and dropout. - **Overfitting Observations**: Removing dropout led to a notable decrease in generalization performance. - **Computation vs. Accuracy**: Reducing kernel size slightly impacted accuracy but improved inference time. - **Layer Depth Impact**: Reducing convolutional layers significantly reduced accuracy.

17 Conclusion and Future Work

This study highlights the importance of architectural choices in CNN design. Future work will explore advanced techniques like attention mechanisms and transfer learning for further performance gains.

Acknowledgments

We acknowledge the resources and support provided for this study.

References

[runningheads]llncs graphicx amsmath amssymb hyperref listings

Vanilla RNN for Next-Word Prediction and Hyperparameter Optimization Anonymous Anonymous Institution

Abstract

This paper presents the implementation of a Vanilla Recurrent Neural Network (RNN) for next-word prediction using the Shakespeare dataset. The model employs a custom RNN cell, trains using Backpropagation Through Time (BPTT), and optimizes hyperparameters. We further compare randomly initialized embeddings with pre-trained embeddings (Word2Vec) and analyze model performance based on perplexity and word-level accuracy. Additionally, a hyperparameter search for CNN and RNN models is conducted using Random Search.

18 Introduction

Recurrent Neural Networks (RNNs) are widely used for sequence modeling tasks such as text generation. This work focuses on implementing a Vanilla RNN for next-word prediction, training with different embedding strategies, and evaluating model performance using multiple metrics.

19 Dataset Loading and Preprocessing

The Shakespeare dataset is obtained from Hugging Face and preprocessed as follows:

- Tokenization is performed to create a vocabulary.
- The dataset is split into 80% training and 20% testing.
- Fixed-length input sequences are generated.
- Data is formatted into a TensorFlow dataset for efficient processing.

20 Custom RNN Implementation

A Vanilla RNN model is implemented using TensorFlow, comprising:

- Trainable embedding layer for learning word representations.
- A SimpleRNN layer to process sequential data.
- A dense output layer with a softmax activation function.

21 Model Training and Optimization

The model is trained using:

- Cross-entropy loss function.
- Adam optimizer with a learning rate of 0.001.

- Early stopping to prevent overfitting.

Training achieves an accuracy of approximately 81.32% after 31 epochs.

22 Text Generation Performance

The model generates text based on a given seed phrase. For instance, using ``To be or not to'', the model outputs:

> "To be or not to them his friends not have their hearts for his country."

23 Evaluation Metrics and Analysis

Performance is measured using:

- **Perplexity** – Measures model uncertainty.
- **Word-Level Accuracy** – Achieved 81.32% accuracy.
- **Loss Curve** – Visualized to ensure convergence.

24 Ablation Study on Embeddings

Experiments compare:

1. Randomly initialized embeddings.
2. Pre-trained Word2Vec embeddings.

Word2Vec improves text generation quality and reduces perplexity.

25 Confusion Matrix and Error Analysis

A confusion matrix is generated to analyze misclassified words, highlighting common prediction errors and their impact.

26 Comparison Table and Discussion

Embedding Type	Word-Level Accuracy	Perplexity
Random Initialization	81.32%	5.12
Word2Vec Pretrained	85.67%	4.29

Table 2: Comparison of Embedding Strategies

Results indicate that pre-trained embeddings improve accuracy and reduce perplexity.

27 Hyperparameter Search for CNN and RNN

27.1 Hyperparameter Selection

Key hyperparameters include:

- Learning rate, batch size, and hidden units for RNN.
- Kernel size, number of filters, and dropout rate for CNN.

27.2 Random Search Implementation

Multiple models are trained using RandomizedSearchCV or custom sampling.

27.3 Best Model Selection and Testing

The top-performing model is selected based on validation accuracy and tested on the dataset.

27.4 Performance Evaluation and Analysis

CNN and RNN models are compared on accuracy, loss, and convergence behavior.

28 Conclusion

This work demonstrates a Vanilla RNN for next-word prediction with different embedding strategies. Pre-trained embeddings significantly improve performance. Additionally, a hyperparameter search is performed to optimize CNN and RNN architectures for text classification tasks.