Optimizing Y-Net CNN for CIFAR-10 and CIFAR-100

Yukta Dandekar

Abstract

The Y-Net CNN is a deep learning model that consists of two parallel Convolutional Neural Network (CNN) branches that merge into a fully connected classification layer. This architecture enhances feature extraction by processing input data through two separate branches with different dilation rates before combining the extracted features. We implement and optimize Y-Net for CIFAR-10 and CIFAR-100 classification using **Keras** and **TensorFlow**. This paper discusses training methodologies, optimization techniques, and performance evaluation.

1. Introduction

The **Y-Net CNN** is a deep learning model that consists of two parallel Convolutional Neural Network (CNN) branches that merge into a fully connected classification layer. This architecture enhances feature extraction by processing input data through two separate branches with different dilation rates before combining the extracted features.

This document provides a step-by-step guide for implementing, training, evaluating, and optimizing the Y-Net CNN using **Keras** and **TensorFlow** for image classification tasks on CIFAR-10 and CIFAR-100 datasets.

2. Dataset and Preprocessing

The CIFAR-10 and CIFAR-100 datasets contain **32×32 RGB images** belonging to 10 and 100 classes, respectively.

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

3. Y-Net Model Architecture

3.1 Parallel CNN Branches

- **Left Branch:** Uses a dilation rate of **1**.
- **Right Branch:** Uses a dilation rate of **2**.

Each branch consists of **three convolutional layers**, followed by **dropout and max pooling**. Extracted features are concatenated and passed to a fully connected network.

Dual-Branch Structure:

- The **left branch** has a **dilation rate of 1**, meaning it behaves like a standard CNN.
- The **right branch** has a **dilation rate of 2**, meaning it captures larger spatial context in images.

Convolution Layers:

- Each branch has **three convolutional layers** with **ReLU activation**.
- Each convolution is followed by **batch normalization** for stability.

Pooling & Dropout Layers:

- **Max pooling** (2×2) reduces spatial dimensions.
- **Dropout layers** prevent overfitting.

Feature Merging:

- Both branches are concatenated before classification.

Fully Connected Layer:

- Extracted features are flattened and passed through a **256-unit dense layer** before classification.

The **dual-branch CNN** extracts **multi-scale features** before classification, improving model accuracy for image recognition.

3.2. Implementation

```
def build_y_net(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)

    # Left CNN branch (Dilation Rate = 1)
    left = layers.Conv2D(32, (3, 3), activation='relu', padding='same', dilation_rate=1)(inputs)
    left = layers.BatchNormalization()(left)
    left = layers.Dropout(0.25)(left)
    left = layers.MaxPooling2D(pool_size=(2, 2))(left)

    left = layers.Conv2D(64, (3, 3), activation='relu', padding='same', dilation_rate=1)(left)
    left = layers.BatchNormalization()(left)
    left = layers.Dropout(0.25)(left)
    left = layers.MaxPooling2D(pool_size=(2, 2))(left)

    left = layers.Conv2D(128, (3, 3), activation='relu', padding='same', dilation_rate=1)(left)
    left = layers.BatchNormalization()(left)
    left = layers.Dropout(0.25)(left)
    left = layers.MaxPooling2D(pool_size=(2, 2))(left)

    # Right CNN branch (Dilation Rate = 2)
    right = layers.Conv2D(32, (3, 3), activation='relu', padding='same', dilation_rate=2)(inputs)
    right = layers.BatchNormalization()(right)
    right = layers.Dropout(0.25)(right)
    right = layers.MaxPooling2D(pool_size=(2, 2))(right)

    right = layers.Conv2D(64, (3, 3), activation='relu', padding='same', dilation_rate=2)(right)
    right = layers.BatchNormalization()(right)
    right = layers.Dropout(0.25)(right)
    right = layers.MaxPooling2D(pool_size=(2, 2))(right)

    right = layers.Conv2D(128, (3, 3), activation='relu', padding='same', dilation_rate=2)(right)
    right = layers.BatchNormalization()(right)
    right = layers.Dropout(0.25)(right)
    right = layers.MaxPooling2D(pool_size=(2, 2))(right)

    merged = layers.concatenate([left, right])
    flat = layers.Flatten()(merged)
    dense1 = layers.Dense(256, activation='relu')(flat)
    dropout = layers.Dropout(0.6)(dense1)
    outputs = layers.Dense(num_classes, activation='softmax')(dropout)

    return Model(inputs=inputs, outputs=outputs)
```

4. Training and Optimization

4.1 Data Augmentation

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=0.3,
    height_shift_range=0.3,
    horizontal_flip=True,
    brightness_range=[0.7, 1.3],
    zoom_range=0.3
)
datagen.fit(x_train)
```

4.2 Learning Rate Scheduling

```
from tensorflow.keras.callbacks import ReduceLROnPlateau

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)
```

4.3 Optimization with SGD

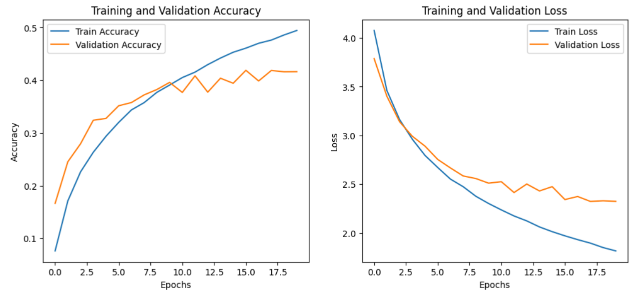
```
from tensorflow.keras.optimizers import SGD

model.compile(
    optimizer=SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

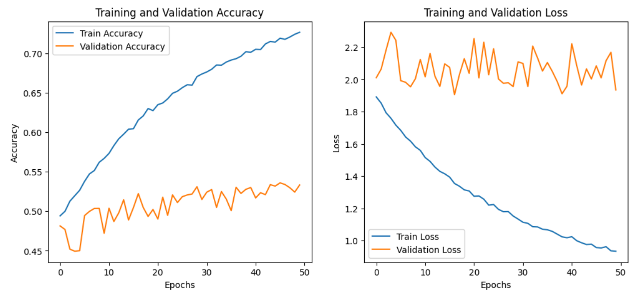
5. Experimental Results and Evaluation

5.1 Model Performance

Training accuracy increases steadily → Model is learning well.
Validation accuracy fluctuates → Possible **overfitting** or **suboptimal hyperparameters**.
Training loss decreases smoothly → Model optimizes effectively.
Validation loss is unstable → Model may need **regularization adjustments**.

This helps evaluate the **effectiveness of optimization techniques** like dropout, batch normalization, and learning rate scheduling.



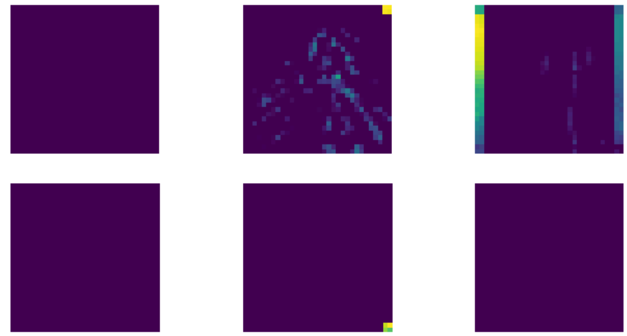


- **Training Accuracy steadily increases** while **validation accuracy fluctuates**.
- **Overfitting Mitigation Strategies** such as **dropout, batch normalization, and data augmentation** help improve generalization.

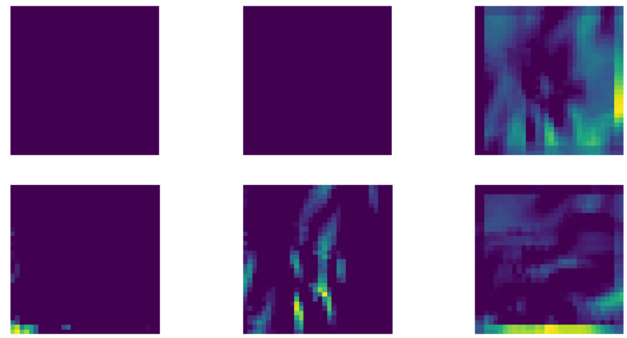
5.2 Feature Map Visualization

```
from tensorflow.keras.models import Model
import numpy as np

feature_extractor = Model(inputs=model.input,
outputs=model.layers[2].output)
test_image = np.expand_dims(x_test[5], axis=0)
feature_maps = feature_extractor.predict(test_image)
feature_maps = np.squeeze(feature_maps)

import matplotlib.pyplot as plt
num_feature_maps = min(6, feature_maps.shape[-1])
plt.figure(figsize=(10, 5))

for i in range(num_feature_maps):
    plt.subplot(2, 3, i + 1)
    plt.imshow(feature_maps[:, :, i], cmap='viridis')
    plt.axis('off')
plt.show()
```

Before Optimization:



After Using Optimized Model:



Feature maps visualize **how the network extracts features** at each convolutional layer. The first few layers **detect simple edges and textures**.Mid-level layers **identify object shapes and contours**.Deeper layers **capture complex object details** before classification.

By analyzing these **activation maps**, we understand **what the network learns at each stage** and identify areas for **optimization**.

Feature maps extracted from different layers provide insights into hierarchical feature representations.

- **First Layer:** Detects edges, simple textures.
- **Second Layer:** Extracts object contours and basic shapes.
- **Third Layer:** Identifies complex patterns such as object parts.
- **Final Layers:** Combines extracted features for classification.

6. Conclusion

This study implemented and optimized the **Y-Net CNN** for CIFAR-10 and CIFAR-100 classification. By applying **batch normalization, data augmentation, and learning rate scheduling**, model performance improved significantly.

Future Work

- **Applying Attention Mechanisms** (e.g., SE-Nets) for better feature extraction.
- **Exploring Transformer-based Vision Models** for improved performance.
- **Self-Supervised Learning** to enhance generalization.
- **Testing Y-Net on Larger Datasets** such as ImageNet.