2025

Introduction to Deep Learning

FINAL PROJECT

AYESHA ZAFAR

Contents

1. Introduction

In this project, I explored image classification using Convolutional Neural Networks (CNNs). The main goal was to train a CNN from scratch on a subset of the **CIFAR-100** dataset and experiment with various model configurations to improve classification accuracy. Since training deep learning models on large datasets can be time consuming and memory intensive - I decided to work with a smaller, more manageable portion of the data (as suggested in the project guidelines).

To keep things practical, I randomly selected a subset of **10 classes** from CIFAR-100 and limited the number of samples per class. This way, I could run multiple training and tuning experiments without overwhelming system resources.

Later on, I repeated similar experiments using a subset of **ImageNet**, specifically through a lightweight alternative called **Imagenette**, which is available via TensorFlow Datasets. This allowed me to compare model behavior across two different datasets and get a better sense of how well the same CNN architecture generalizes.

2. Dataset

I worked with two image classification datasets: **CIFAR-100** and a smaller version of ImageNet called **Imagenette**. Both contain colored images, but they differ in size, diversity, and complexity. Since I wanted to train CNNs from scratch without overwhelming my resources, I only used a subset of each dataset—both in terms of the number of classes and the number of images per class.

2.1 CIFAR-100

CIFAR-100 is a well-known dataset that consists of **60,000** color images (32x32 pixels) across **100 different classes**, such as animals, vehicles, and objects. Each image is labeled with one fine-grained category.

2.1.1 Preparation

The dataset is already split into **50,000 training** and **10,000 test images**, but using all 100 classes wasn't practical for my experiments. So I randomly selected **10 classes** and focused only on those.

I wrote custom functions to:

- **Map class names to their indices** using the official class list.
- **Filter out images** that didn't belong to the selected classes.
- **Remap the labels** so that the new class indices range from 0 to 9.
- **Reduce the dataset size** to make training more manageable:

    - 300 images per class for training.
    - 50 images per class for testing.

This left me with a nicely balanced subset of the dataset: **3,000 training images** and **500 test images**.

2.1.2 Preprocessing

Before training the CNN:

- I **normalized** the pixel values to the range [0, 1].
- I applied **one-hot encoding** to the class labels so the model could output a probability for each class.

This helped the model converge faster and treat all classes fairly during training.

2.2 Imagenette (Subset of ImageNet)

I also wanted to test how well my model generalizes to a different dataset, so I repeated the experiments on **Imagenette**—a lighter version of ImageNet with **10 classes**, curated specifically for quicker experiments.

2.2.1 Preparation

Using TensorFlow Datasets, I loaded both the training and validation splits. Imagenette originally contains high-resolution images (320x320), but I resized them to **32x32** to match the CIFAR-100 format and keep the model lightweight.

To maintain consistency with my CIFAR setup:

- I randomly selected **10 classes**.
- For each class, I extracted:
    - **300 training images**.
    - **50 validation images**.
- I also **remapped the class indices** to range from 0 to 9.

This gave me a similar dataset size and balance as CIFAR-100.

2.2.2 Preprocessing

Just like with CIFAR-100:

- I **normalized** all pixel values.
- I applied **one-hot encoding** to the labels.

3. Baseline CNN Model

For the base experiment, I selected a subset of 10 classes from the CIFAR-100 dataset, including a diverse mix of natural and man-made objects: ['skyscraper', 'motorcycle', 'poppy', 'squirrel', 'dinosaur', 'worm', 'plain', 'tractor', 'keyboard', 'rocket'].

To classify these classes, I built a Convolutional Neural Network model from scratch using TensorFlow/Keras. This model served as the baseline architecture for evaluating the classification performance on the selected subset.

3.1 Model Architecture

The model consists of three convolutional blocks followed by fully connected layers:

- **Input Layer:** Accepts images of shape (32, 32, 3).
- **Convolutional Blocks:**
    - **Block 1:** Two convolutional layers with 32 filters, followed by max pooling and dropout (25%).
    - **Block 2:** Two convolutional layers with 64 filters, followed by max pooling and dropout (25%).
    - **Block 3:** Two convolutional layers with 128 filters, followed by max pooling and dropout (25%).
- **Fully Connected Layers:**
    - A dense layer with 512 units and ReLU activation, followed by dropout (50%).
    - An output dense layer with 10 units (corresponding to 10 classes) and softmax activation.
- **Compilation:** The model was compiled using the Adam optimizer, categorical cross-entropy loss, and accuracy as the evaluation metric.

This architecture is inspired by classic CNN designs used in small-scale image classification tasks, balancing model complexity and computational cost.

3.2 Training Details

- **Epochs:** 25
- **Batch Size:** 64
- **Validation Split:** A fixed test set was used for validation after data preprocessing.

- **Data Augmentation:** Not used in this baseline model (could be considered for future improvements).
- **Preprocessing:** Input images were normalized and labels were one-hot encoded.

3.3 Results

During training, the model achieved the following metrics:

- **Final Training Accuracy**: 91.68 %
- **Final Validation Accuracy**: 75.80%
- **Final Test Accuracy**: 77.70 %
- **Test Loss**: 0.7539

The training accuracy showed a consistent upward trend, ultimately surpassing 91%, indicating strong model convergence. Validation accuracy remained relatively stable throughout training, fluctuating between 72% and 76%. While the final validation and test accuracies were close, the difference from the training accuracy suggests mild overfitting. Nonetheless, achieving 77.70% test accuracy is promising, especially considering the relatively simple model architecture and the absence of advanced regularization techniques such as data augmentation, dropout, or learning rate scheduling.

3.4Observations

- The model demonstrated solid learning performance, with the training accuracy reaching over 91%, reflecting successful convergence and the model's capacity to fit the 10-class subset.
- A noticeable gap (~14–16%) between training and validation/test accuracy indicates **moderate overfitting**, likely due to:
    - A limited dataset size, which reduces the ability to generalize to unseen data.
    - Lack of regularization strategies like data augmentation or early stopping.
- Validation accuracy showed less volatility peaking at **76.20%** in the final epochs. However, the **validation loss trend remained somewhat inconsistent**, with a rise in later epochs despite the increase in validation accuracy. This behaviour suggests that while classification performance improved, the model's confidence in its predictions may have varied.

4. Experiments to Improve Accuracy

4.1 Increasing the Size and Depth of Inner Layers

To improve the model's ability to learn complex patterns, I extended the architecture by:

- Adding a fourth convolutional block with 512 filters.
- Increasing the number of filters in existing layers (64 → 128 → 256 → 512).
- Expanding the fully connected layers with 512 and 256 units and applying L2 regularization.

4.1.1 Model Summary:

- 4 convolutional blocks (each with 2 Conv layers + BatchNorm + ReLU + Dropout + MaxPooling)
- Dense layers: 512 → 256 → Output
- Regularization: Dropout & L2
- Optimizer: Adam, Loss: Categorical Crossentropy

4.1.2 Results:

After training for 25 epochs:

- **Final Training Accuracy:** 89.45%
- **Final Validation Accuracy:** 68.80%
- **Final Test Accuracy:** 74.96%
- **Final Test Loss:** 1.3025

While the deeper network showed a consistent increase in training accuracy, reaching nearly 90%, the validation accuracy fluctuated and ended significantly lower. This discrepancy highlights a clear case of overfitting. Validation loss remained relatively high throughout, further suggesting that the model struggled to generalize despite its increased learning capacity. These limitations are likely due to the modest dataset size and the lack of advanced regularization techniques.

4.1.3 Data Augmentation

To reduce overfitting and improve generalization, I applied data augmentation techniques such as:

- Random rotation (±10°)
- Width and height shifts (±5%)

- Horizontal flips
- Minor zooming

The model was first trained without augmentation for 20 epochs, then fine-tuned with augmentation for 50 epochs using ImageDataGenerator.

Results after Fine-Tuning:

- **Peak Validation Accuracy:** 94.13%
- **Final Validation Accuracy:** 71.60%
- **Final Test Accuracy:** 75.09%
- **Final Test Loss:** 1.2022

Fine-tuning with extended training and data augmentation led to moderate improvements. Validation accuracy peaked at 81.40%, and the test accuracy improved slightly compared to the earlier model. However, performance gains were not sustained in the final epochs, indicating diminishing returns from further training. While augmentation introduced more diversity and helped the model generalize better initially, overfitting began to reemerge as training progressed.

4.1.4 Key Takeaways:

- Deeper architectures can enhance learning capacity, but overfitting remains a challenge without sufficient data or regularization.
- Data augmentation led to some improvement in generalization, but gains were less stable and tapered off by later epochs.
- A balance between model complexity, dataset size, and regularization (e.g., early stopping, dropout, or learning rate scheduling) is crucial for maximizing performance.
-

4.2 Using Fewer or More Convolutional/Max Pooling Layers

To evaluate the impact of CNN architecture depth (number of convolutional and pooling layers) on classification performance. I tested two models:

- A **simpler CNN** with fewer convolutional layers.
- A **deeper CNN** with more convolutional layers and increased filter capacity.

4.2.1 Model Architectures:

Simpler CNN:

- 2 convolutional blocks:

  - Block 1: Conv2D(32) → MaxPooling
  - Block 2: Conv2D(64) → MaxPooling
- Flatten → Dense(128) → Dropout(0.3) → Output

Deeper CNN:

- 3 convolutional blocks, each with 2 Conv2D layers:
  - Block 1: Conv2D(64) ×2 → MaxPooling
  - Block 2: Conv2D(128) ×2 → MaxPooling
  - Block 3: Conv2D(256) ×2 → MaxPooling
- Flatten → Dense(256) → Dropout(0.3) → Output

4.2.2 Results:

Model	**Final Train Accuracy**	**Final Validation Accuracy**	**Final Test Accuracy**	**Validation Loss Trend**
Simpler	92.33%	71.80%	75.06%	Gradual increase, somewhat stable
Deeper	96.67%	73.40%	76.60%	Increasing, clear signs of overfitting

4.2.3 Observations:

- **Train Accuracy:**
  The deeper model achieved higher training accuracy (96.67%) compared to the simpler model (92.33%), indicating its stronger capacity to fit training data due to more parameters and layers.
- **Validation and Test Accuracy:**
  Despite higher training performance, the deeper model underperformed slightly on the validation and test sets compared to the simpler model. This suggests that the deeper network may have started to **overfit** the training data.
- **Validation Loss:**
  The deeper model showed a significant increase in validation loss over epochs, which is a typical sign of overfitting. In contrast, the simpler model had a more stable validation loss.

4.3 Using Different Activation Functions in Convolutional and Inner Layers

Activation functions introduce non-linearity to the model, influencing the network's ability to learn complex patterns. The activation functions I evaluated include: ReLU, Sigmoid, Tanh, Leaky ReLU, Swish, Swish combined with ReLU, Swish combined with Leaky ReLU.

Each model shared the same overall architecture (three convolutional blocks followed by a dense layer and a softmax output layer) but differed in the activation functions applied.

4.3.1 Methodology

I used custom CNNs for each activation function. The architecture and optimizer (Adam) were kept consistent, and all models were trained for 25 epochs using the same training and testing datasets.

4.3.2 Results & Observations

Activation Function	**Test Accuracy**	**Final Test Loss**	**Observations**
ReLU	78.32%	0.9312	Fast convergence and solid baseline; good balance between accuracy and loss.
Tanh	79.79%	0.8294	Best overall performance; achieved the highest accuracy and lowest loss.
Sigmoid	50.44%	1.3363	Poor performance due to vanishing gradient; significantly lower accuracy and higher loss.
Leaky ReLU	77.98%	0.9384	Comparable to ReLU; small gain in stability due to non-zero gradient in negative inputs.
Swish	78.24%	1.3668	Accuracy close to ReLU, but high loss suggests overfitting or poor generalization.
Swish + ReLU	75.27%	0.8538	Lower accuracy but lowest loss; may indicate better generalization despite fewer correct predictions.
Swish + Leaky ReLU	78.07%	0.9429	Accuracy slightly better than Swish alone; leaky ReLU helped reduce vanishing gradient issues.

4.4 Using Different Optimizers and Learning Rates

In this part I trained the same CNN model architecture using four popular optimizers: **Adam**, **SGD**, **RMSprop**, and **Adagrad**, each tested with three different learning rates: **0.01**, **0.001**, and **0.0001**.

4.4.1 Results Summary

Optimizer	**Learning Rate**	**Test Accuracy (%)**
Adam	0.01	10.00
Adam	0.001	77.80
Adam	0.0001	66.60
SGD	0.01	45.20
SGD	0.001	18.40
SGD	0.0001	9.40
RMSprop	0.01	10.00
RMSprop	0.001	75.20
RMSprop	0.0001	60.60
Adagrad	0.01	54.60
Adagrad	0.001	29.40
Adagrad	0.0001	10.60

4.4.2 Observations

- **Adam** with a learning rate of **0.001** achieved the highest accuracy (**77.80%**), confirming its strong adaptive capabilities and robustness to sparse gradients.
- **RMSprop** also performed very well, reaching **75.20%** accuracy at **0.001**, making it a competitive alternative to Adam.
- **High learning rates (0.01)** still led to **training failure** for **Adam** and **RMSprop** (both at **10%**), likely due to unstable weight updates.
- **SGD** showed better performance at **0.01 (45.20%)**, but degraded sharply at lower learning rates, highlighting its sensitivity and need for momentum or learning rate schedules.

- **Adagrad** achieved **moderate performance** at **0.01 (54.60%)**, but suffered at lower learning rates, suggesting limited adaptability over time due to aggressive learning rate decay.

4.5 Using Different Batch Sizes and Epochs

Next, I tested how different combinations of **batch sizes** and **number of training epochs** affect the model's final accuracy.

- Batch Sizes: 32, 64, 128
- Epochs: 10, 20, 30

4.5.1 Results Summary

Batch Size	Epochs	Test Accuracy (%)
32	10	68.00
32	20	73.00
32	30	71.40
64	10	66.60
64	20	73.80
64	30	71.00
128	10	63.20
128	20	73.40
128	30	73.40

4.5.2 Observations

- Increasing the number of **epochs** generally improved test accuracy across all batch sizes, though the benefit plateaued or slightly declined after 20 epochs in some cases.
- The highest accuracy (**73.80%**) was achieved with a **batch size of 64 and 20 epochs**, indicating a good balance between batch size and training duration.
- **Batch size 128** performed comparably well at **20 and 30 epochs** (both **73.40%**), suggesting it can still generalize well with sufficient training.
- **Batch size 32** peaked at **73.00% with 20 epochs**, but did **not improve further** with 30 epochs, possibly due to overfitting or reduced gradient diversity.

- Smaller batches (e.g., 32) can help escape sharp minima but may require careful tuning of epochs, while larger batches (e.g., 128) offer stable convergence but might need more epochs to reach optimal performance.

5. Repeating Experiments on a Different CIFAR-100 Subset

To evaluate the generalizability of model architecture and optimization strategies, I repeated all experiments from Task 5 using a new subset of CIFAR-100 classes:

Selected Classes: ['orchid', 'otter', 'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine']

5.1 Baseline Model

Using the original architecture, the model achieved:

- **Test Accuracy**: **76.02%**

5.2 Effect of Increasing Layer Size and Depth

Enhancing the number and size of dense layers:

- **Test Accuracy**: **72.89%**

Observation: Larger and deeper layers did not improve the accuracy, suggesting the new subset didn't much benefit from a richer model capacity.

5.3 Varying Convolutional/Pooling Layers

Experiments with different configurations:

Architecture Type	**Test Accuracy**
Simpler (fewer layers)	73.62%
Deeper (more layers)	77.20%

Observation: A simpler architecture performed well but not as good as the deeper one, likely due to less complexity.

5.4 Exploring Different Activation Functions

Activation Function	**Test Accuracy**
Tanh	77.09%

Activation Function	Test Accuracy
Sigmoid	54.71%
Leaky ReLU	77.29%
Swish	73.60%
ReLU	76.60%
Swish + ReLU	74.11%
Swish + Leaky ReLU	78.67%

Observation: Leaky ReLU and Swish + Leaky ReLU provided the best performance. Sigmoid significantly underperformed due to its vanishing gradient effect.

5.5 Optimizer and Learning Rate Experiments

Optimizer	Learning Rate	Test Accuracy
Adam	0.01	10.00%
Adam	0.001	76.00**%**
Adam	0.0001	67.20%
SGD	0.01	48.00%
SGD	0.001	12.00%
SGD	0.0001	10.00%
RMSprop	0.01	10.00%
RMSprop	0.001	74.00%
RMSprop	0.0001	61.00%
Adagrad	0.01	63.20%
Adagrad	0.001	18.80%
Adagrad	0.0001	14.40%

Observation: Once again, **Adam with 0.001** emerged as the most reliable choice. RMSprop was a competitive alternative.

5.6 Batch Size and Epoch Experiments

Batch Size	Epochs	Test Accuracy
32	10	72.40%
32	20	72.80%
32	30	74.80%
64	10	69.80%
64	20	75.20%
64	30	74.40%
128	10	66.20%
128	20	74.00%
128	30	74.20%

Observation: The accuracy plateaued at **75.80%** across multiple configurations, highlighting a robust generalization in this subset. Small batch sizes and longer training (epochs) consistently achieved higher performance.

5.7 Observations

Repeating the experiments on a different subset validated the earlier findings:

- **Leaky ReLU/Swish+ Leaky ReLU**, **Adam optimizer with lr=0.001**, and **20 epochs with batch size 64** provide consistent performance across different class subsets.
- Architecture changes had a moderate effect; however, activation function and optimizer choice showed more significant influence.

6. Repeating Experiments on a Imagenet Subset

To explore the performance of convolutional neural networks on a more complex dataset, I also experimented with a subset of the **Imagenette** dataset, which is a smaller, 10-class subset of ImageNet. The images were resized to 32×32 pixels to maintain consistency with previous experiments.

6.1 Baseline Model

A basic CNN model achieved a test accuracy of:

- **Accuracy**: **62.81%**
- **Loss**: 1.6697

6.2 Effect of Increasing Layer Size and Depth

Increasing the number of neurons and depth in the inner layers changed model capacity:

- **Accuracy**: **61.8%**
- **Loss**: 1.8820

Observation: Increasing inner layer size slightly worsened performance, suggesting the dataset didn't benefit from enhanced feature representation capacity.

6.3 Varying Convolutional/Pooling Layers

Architecture Type	**Test Accuracy**
Simpler (fewer layers)	60.20%
Deeper (more layers)	59.60%

Observation: The deeper architectures underperformed a little, potentially due to overfitting on the limited subset.

6.4 Exploring Different Activation Functions

Activation Function	**Test Accuracy**
Tanh	61.40%
Sigmoid	38.40%
Leaky ReLU	61.00%
Swish	57.40%
ReLU	64.80%
Swish + ReLU	62.80%
Swish + Leaky ReLU	60.60%

Observation: ReLU and Swish + ReLU gave the best results. Sigmoid suffered due to vanishing gradients.

6.5 Optimizer and Learning Rate Experiments

Optimizer	**Learning Rate**	**Test Accuracy**
Adam	0.01	10.00%
Adam	0.001	78.40**%**
Adam	0.0001	68.40%
SGD	0.01	50.80%
SGD	0.001	16.40%
SGD	0.0001	8.80%
RMSprop	0.01	10.00%
RMSprop	0.001	73.00%
RMSprop	0.0001	60.40%
Adagrad	0.01	64.40%
Adagrad	0.001	32.20%
Adagrad	0.0001	11.60%

Observation: **Adam with lr=0.001** emerged as the most reliable optimizer. RMSprop was a solid runner-up.

6.6 Batch Size and Epoch Experiments

Batch Size	**Epochs**	**Test Accuracy**
32	10	68.00%
32	20	75.60%
32	30	75.60%
64	10	72.40%
64	20	74.80%
64	30	75.20**%**

Batch Size	Epochs	Test Accuracy
128	10	68.00%
128	20	73.40%
128	30	73.80%

Observation: **Batch size 32 with 20 and 30 epochs** achieved the highest accuracy. However, smaller batches with more epochs offered stable and high-performing alternatives.

6.7 Observations

Repeating similar experiments on the Imagenette subset reaffirmed previous findings:

- **Best performing combinations**:
  - **Activation Function**: ReLU / Swish + ReLU
  - **Optimizer**: Adam (lr = 0.001)
  - **Training Setup**: 20–30 epochs with batch size 32
- **Impact Comparison**:
  - Optimizer and activation function choice had the **largest effect** on performance.
  - Changes to convolutional architecture (deeper vs. simpler) had **marginal or negative effects**, possibly due to the small dataset size.
  - Learning rate sensitivity remained high; overly large or small rates degraded performance.