# MN911 - Advanced Biometry Security Mini-Project Report

***Topic 6:*** **CNN Model Based on ResNet Architecture**

**Author: Hoang-Nhat TRAN and Thu-Hien LE**

**March 26, 2025**

**Keywords**:
**cnn, resnet**

Conducted under the supervision of:

Prof. Djana Petrovska

# Acknowledgements

This mini-project is part of the MN911 - Advanced Security module, within the M2 Multimedia Networking training program jointly coordinated by Université Paris-Saclay and Telecom SudParis - Institut Polytechnique de Paris.

We would like to thank Assoc. Prof. Hai Vu and his research group at Hanoi University of Science and Technology for generously providing us with a device with accelerated computing units to run the experiments.

# Abstract

Moreover, within the context of biometric security, we adopted one of the margin-based loss functions widely used in biometric face verification and evaluated its robustness for training the ResNet on CIFAR-10. We show that these methods can produce comparable results on small classification tasks, but a much more discriminative feature space. The code will be made publicly available at `https://github.com/hihien99/mn911`.

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# 1. Introduction

This mini-project addresses the challenge of enhancing image classification on the CIFAR-10 dataset using deep learning techniques. Firstly, a baseline convolutional neural network was developed and analysed to provide essential insights into CNN behavior for multi-class tasks. Building on these insights, our work then explores advanced architectures—specifically ResNet-18 and ResNet-50 with data normalization, augmentation, and enhanced evaluation metrics such as precision, recall, and F1-score. Additionally, inspired by recent research in cancellable biometrics, margin-based loss functions were integrated to boost the model's discriminative power. Together, these methodologies contribute to a more robust and reliable classification system.

The rest of this report contains:

- In Chapter 2, we fulfill the tasks of the lab session, consisting of the implementation of a simple CNN.
- In Chapter 3, we extend the work in the lab as a final mini-project, featuring the ResNet architectures trained with different loss functions.

# 2. Baseline

In this chapter, we address the tasks and questions in the lab session. The practical work entails training a simple convolutional neural network (CNN), dubbed "baseline", on the image classification dataset CIFAR-10. Therefore, the chapter is presented in sections that answer questions in the lab.

The baseline results will be further compared with our extension in Chap. 3.

## 2.1. Question 1: Define the CNN model

In this lab, we define a simple CNN with the architecture illustrated in Fig. 2.1, and its hyperparameters listed in Tab. 2.1.

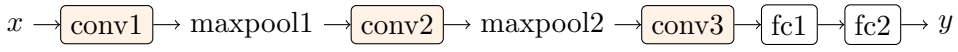



Figure 2.1: Baseline CNN Architecture

| **Layer** | Baseline |
|---|---|
| conv1 | $3 \times 3, 32$ conv |
| maxpool1 | $2 \times 2$ max pooling, stride 2 |
| conv2 | $3 \times 3, 64$ conv |
| maxpool2 | $2 \times 2$ max pooling, stride 2 |
| conv3 | $3 \times 3, 64$ conv |
| fc1 | $1024 \rightarrow 64$-d fully-connected |
| fc2 | $64 \rightarrow n$-d fully-connected output layer, softmax |

Table 2.1: Hyperparameters of Baseline CNN architecture.

Programmatically, the lines to define this model using `keras` library are as in Lst. 2.1.

Listing 2.1: Code to initialize the baseline CNN in `keras`.

```
from tensorflow.keras import datasets, layers, models, Model

def baseline():
    random_state = 0
    np.random.seed(random_state)
    keras.utils.set_random_seed(random_state)
    tf.config.experimental.enable_op_determinism()
    model = models.Sequential()
```

```
9     model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
10    model.add(layers.MaxPooling2D((2, 2)))
11    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
12    model.add(layers.MaxPooling2D((2, 2)))
13    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
14    model.add(layers.Flatten())
15    model.add(layers.Dense(64, activation='relu'))
16    model.add(layers.Dense(10))
17    return model
18
19 model = baseline()
```

In which:

- Line $4-6$: Setting the random state and operator determinism among different libraries to ensure reproducibility.
- Line 8: Creating a sequential model.
- Line 9: Defining the first layer (a convolutional layer with 32 filters ($3 \times 3$), *ReLU* activation function, apply on specific input image size).
- Line 10: Max-pooling layer with a pool size of $2 \times 2$.
- Line 11: Defining the second layer (a convolutional layer with 64 filters ($3 \times 3$), *ReLU* activation function).
- Line 12: Max-pooling layer with a pool size of $2 \times 2$.
- Line 13: Defining the third layer (a convolutional layer with 64 filters ($3 \times 3$), *ReLU* activation function).
- Line 14: Flatten the output to prepare it for the dense layers.
- Line 15: Defining dense layer with 64 units and *ReLu* activation function.
- Line 16: Defining the final dense layer with 10 units and $Softmax$ function for classification.

Note that while we have set a fixed random seed, the reproducibility of the training procedure is only guaranteed between different runs on the same device. The results can differ if executed on different devices. This is a problem related to `keras`.

## 2.2. Question 2: Interpret the obtained results

Before interpreting the results, we train the baseline model using the same parameter settings as instructed in the TP. The script to train the baseline model on CIFAR-10 is in Lst. 2.2.

Listing 2.2: Code to train the baseline CNN.

```python
def compile_and_train(model,
                      train_images,
                      train_labels,
                      test_images,
                      test_labels,
                      epochs=10):
    model.compile(optimizer='adam',
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
    history = model.fit(train_images, train_labels, epochs=epochs,
                        validation_data=(test_images, test_labels))
    return history

print('Training simple CNN')
history_cnn = compile_and_train(
    model, train_images, train_labels, test_images, test_labels)
```

The results obtained after the training procedure can be seen in Fig. 2.2

```
Training simple CNN
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 17s 9ms/step - accuracy: 0.3623 - loss: 1.7268 - val_accuracy: 0.5586 - val_loss: 1.2357
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.5737 - loss: 1.1894 - val_accuracy: 0.6181 - val_loss: 1.0787
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.6457 - loss: 1.0053 - val_accuracy: 0.6491 - val_loss: 0.9996
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.6825 - loss: 0.8995 - val_accuracy: 0.6540 - val_loss: 1.0026
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7106 - loss: 0.8257 - val_accuracy: 0.6633 - val_loss: 0.9853
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7320 - loss: 0.7660 - val_accuracy: 0.6735 - val_loss: 0.9666
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7492 - loss: 0.7145 - val_accuracy: 0.6958 - val_loss: 0.9235
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.7686 - loss: 0.6647 - val_accuracy: 0.6985 - val_loss: 0.9308
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7815 - loss: 0.6216 - val_accuracy: 0.7013 - val_loss: 0.9370
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7959 - loss: 0.5860 - val_accuracy: 0.6973 - val_loss: 0.9842
```

Figure 2.2: The obtained results of the training procedure.

- **epoch**: indicates the number of full visits through all samples of the training set.
- **accuracy**: Average accuracy on the train set; it increases across epochs (from 0.36 to 0.79), indicating that the model is indeed fitting.
- **val_accuracy**: Accuracy on the test dataset; it increases across epochs (0.55 to 0.69) and indicates how good the model is on the unseen data.
- **loss**: Coexisting with the increase of accuracy metric, the decrease in loss (calculated using cross entropy) in train and test sets means the loss function converges to an optimal.

## 2.3. Question 3: Interpret the obtained curve.

While training the baseline model, the training and testing accuracies are saved into the *history*. We use the following lines of code in Lst. 2.3 to plot the training and testing curves to observe how well the model learns after 10 epochs. The resulting curves can be seen in Fig. 2.3.

Listing 2.3: Code to plot the accuracy curve of baseline model after training.

```
plt.style.use('ggplot')

fig, ax = plt.subplots()
cmap = plt.get_cmap('tab10')
x = np.arange(1, 11)
ax.plot(x, history.history['accuracy'], marker='x', color=cmap(0), label='train')
ax.plot(x, history.history['val_accuracy'], marker='x', color=cmap(1), label='test')

ax.grid(True, which='both')
ax.set_xlabel('epoch')
ax.set_ylabel('accuracy')
ax.set_ylim([0, 1])
ax.set_xlim([1, 10])
ax.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

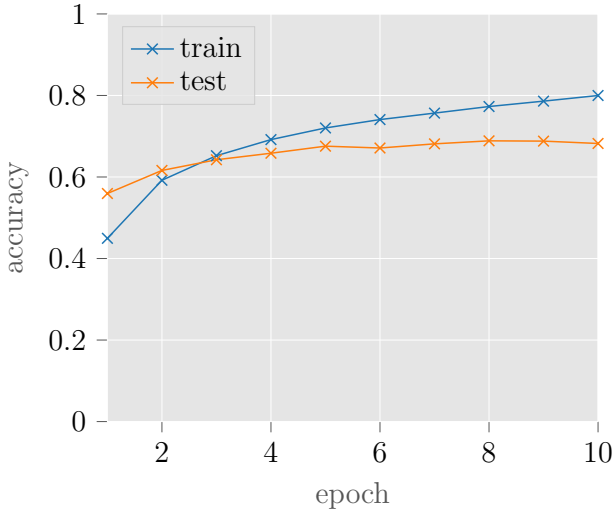



Figure 2.3: Training and testing accuracy curves of baseline CNN model over 10 epochs.

- **Training Accuracy Curve**: In the early epochs, the accuracy is relatively low and rises to higher ones as epochs increase. It means that the model is learning to correctly classify more

examples from the training set. Along the increasing number of epochs, the model learns and refines its weights.

- **Testing Accuracy Curve**: This curve indicates how well the model performs on unseen data. Generally, it might be see a similar increase as with training accuracy. One thing that can be noticed here is testing accuracy starts to flatten while the traning accuracy keeps improving because the test set is different from the training set. This is normal, the small swings might indicate that the model's performance is stable over epochs.

## 2.4. Question 4: Interpret the obtained results.

In this section, we utilize the saved model to classify a new image and assign it the appropriate label. The code in Lst. 2.4 reads an image and converts it to the correct color format for displaying using `cv2` library.

Listing 2.4: Code to read and show a new image.

```
img = cv2.imread('chien.jpg')
# convert to RGB for compatibility with matplotlib
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
plt.axis('off')
plt.show()
```

The output of the above code block can be seen in Fig. 2.4.

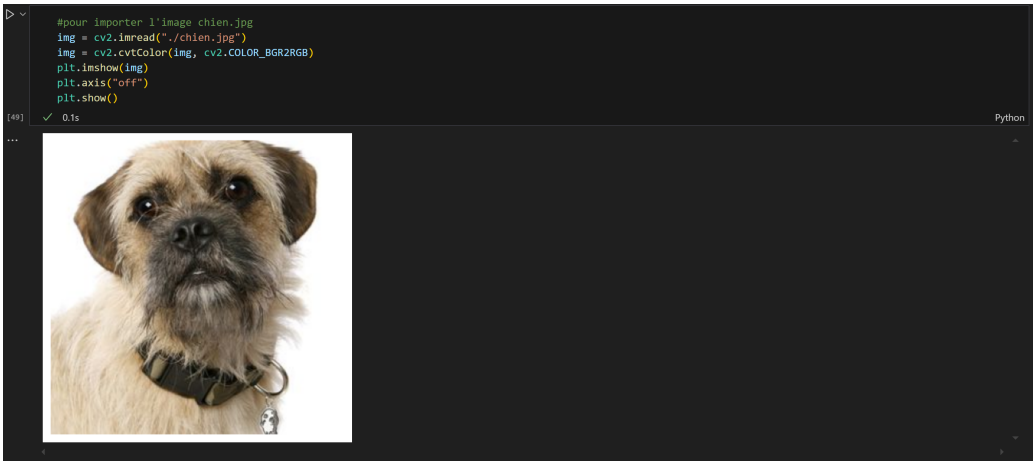



Figure 2.4: The new dog image we used to test the model.

After that, we run the next code block in Lst. 2.5, which utilizes the saved model to classify the image of *dog*:

Listing 2.5: Code to infer the baseline model on a new image.

```
# Pour la prédiction de l'image chien
img = cv2.resize(img_cvt, (32,32))
img = img.astype(np.float32) / 255.0
imgs = tf.expand_dims(img, axis=0, name=None)
preds = model.predict(imgs)
print(class_names[np.argmax(preds)])
```

This code resizes the image to $32 \times 32$ pixels, normalizes it, and formats it correctly for batch processing. After preprocessing, we have a tensor that has the shape of $(1 \times 32 \times 32 \times 3)$ as the input of the model.

The output *preds* is a numpy array with shape $(10 \times 1)$. It is the array of probabilities for each class. For this case, the result is shown in figure 2.5, the probability of the $5^{th}$ class (*dog*) is ranked first, which is nearly 42.8%.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
[[5.0987503e-05 2.0602366e-04 2.4212722e-01 2.5268067e-03 2.5779667e-04
  4.2870650e-01 3.2429051e-01 1.4733798e-03 2.8844281e-05 3.3190148e-04]]
predicted label:  dog
```

Figure 2.5: The tensor indicates the output of the model on a test image.

## 2.5. Question 5: Interpret the images obtained for each layer.

In this section, we extract and visualize feature maps at each layer of the baseline model. From the results, we can understand what parts of an input image each layer focuses on when it processes the data.

Listing 2.6: Code to extract and visualize baseline model feature maps of different layers.

```
for layer_id in range(len(model.layers)):
    # create feature extractor
    model_feature_extractor = Model(inputs=model.inputs, outputs=model.layers[layer_id].output
    model_feature_extractor.summary()
    feature_maps = model_feature_extractor.predict(imgs)
    square = 5
    idx = 1
    for _ in range(square):
        for _ in range(square):
            # specify subplot and turn of axis
            ax = plt.subplot(square, square, idx)
            ax.set_xticks([])
            ax.set_yticks([])
            # plot filter channel in grayscale
```

```
            plt.imshow(feature_maps[0, :, :, idx - 1])
            idx += 1
    plt.show()
```

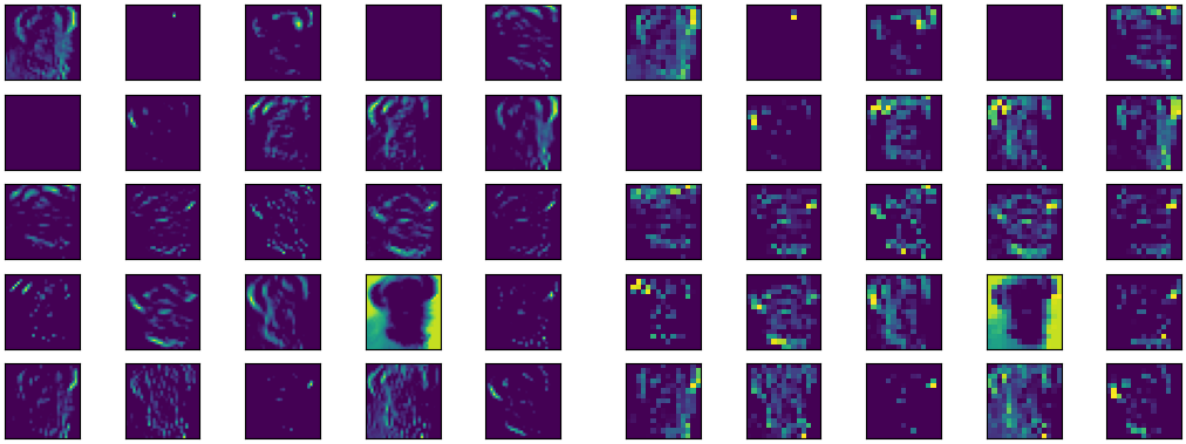
The results obtained from executing Lst. 2.6 can be seen in Fig. 2.6, at the very first layers, basic features and simple patterns such as edges, lines, and color are learned. when we go deeper into the next layers (max-pooling layer, convolutional layers,..), the feature maps are reduced in shape to reduce computational cost and are typically less intuitive than the first layers. They start to capture more intricate patterns and form a more holistic understanding of the image content (in other words, a larger **receptive field** is captured). At the deeper layers, to highlight which regions of an image are emphasized, various **Explainable AI (XAI) Techniques** [4] can be employed.

## 2.6. Question 6: Try to test the trained model with other images belonging to the ten classes of CIFAR-10.

In this section, 100 images were randomly selected from the test set to evaluate the baseline model's performance (see Lst. 2.7).

Listing 2.7: Code to sample 100 images from CIFAR-10 dataset and infer using baseline.

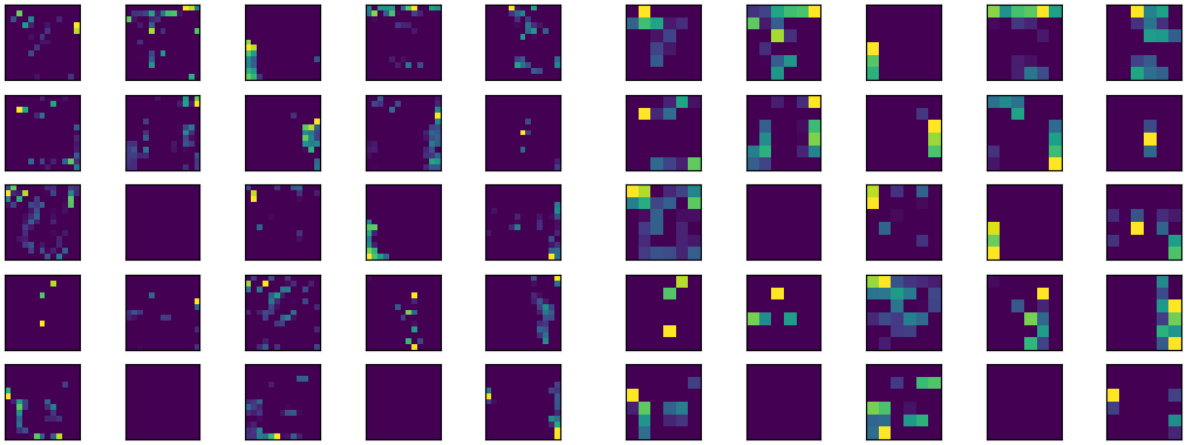
```
random_indices = np.random.choice(len(test_images), size=100, replace=False)

plt.figure(figsize=(20, 20))
for i, idx in enumerate((random_indices)):
    image = test_images[idx]
    true_label = test_labels[idx].astype(int)
    if true_label.ndim > 0 and true_label.shape[0] > 1:  # check one-hot
        true_label = np.argmax(true_label)

    preds = model.predict(np.expand_dims(image, axis=0))
    pred_label = np.argmax(preds).astype(int)

    ax = plt.subplot(10, 10, i+1)
    ax.imshow(image)
    ax.set_xticks([]); ax.set_yticks([])
    if true_label[0] == pred_label:
        plt.setp(ax.spines.values(), linewidth=3, color='lime')
    else:
        plt.setp(ax.spines.values(), linewidth=3, color='red')
    ax.set_title(f'Label: {class_names[true_label[0]]}\nPred: {class_names[pred_label]}')

plt.tight_layout()
plt.show()
```



(a) conv1 output



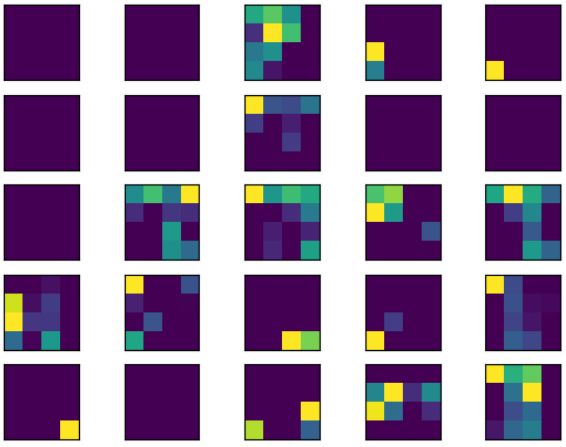
(b) maxpool1 output



(c) conv2 output



(d) maxpool2 output



(e) conv3 output

(f) input image
*(for reference)*

Figure 2.6: Feature maps extracted at each layer of the baseline model.

The results are presented in Fig. 2.7, and correct predictions are highlighted in green while incorrect ones are highlighted in red. These misclassifications indicate that the baseline model's performance is suboptimal.


Figure 2.7: Quantitative performance of the baseline model on the test set.

# 3. Mini-project: Deep Residual Neural Networks

## 3.1. Background

### 3.1.1. ResNet

#### Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of deep learning model designed for various types of data, such as images or time series data. The underlying mechanism of CNNs is using layers that apply filters to the input data and capture local features. These filters slide through the input to create feature maps, which provide high-level information.

The basic units of each convolutional block are a convolutional layer, a ReLU activation function, and max-pooling operation. Each convolutional layer uses a $k \times k$ kernel and a ReLU activation function

#### Gradient Vanishing in Deep Neural Networks

Deeper feed-forward neural networks are harder to train due to a phenomenon called gradient vanishing. The typical formula of a building block of traditional NNs is described in Eq. (3.1).

$$y = \text{relu}\left(\mathcal{F}\left(x\right)\right) \tag{3.1}$$

where $\mathcal{F}$ is a parametrized mapping such as a convolutional layer. Using the chain rule, the gradient back-propagated to the preceding layer $\frac{\partial \mathcal{L}}{\partial x}$ is computed as follows:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial \mathcal{F}\left(x\right)}{\partial x} \cdot \mathbb{1}_{\mathcal{F}(x)>0} \tag{3.2}$$

Here, $\mathcal{L}$ is an arbitrary loss function, $\frac{\partial \mathcal{L}}{\partial y}$ denotes the computed gradient at the output of the layer, and $\mathbb{1}_{\mathcal{F}(x)>0}$ is the indicator function accounting for the gradient of the relu non-linearity. We can observe that the back-propagated gradient is multiplied by a term $\frac{\partial \mathcal{F}(x)}{\partial x}$ which is typically small. Consequently, the gradients at the beginning of the network have negligible magnitude, leading to difficulty in parameter updating.

### Residual Learning

Residual learning [3] was introduced to address the aforementioned problem. The primary contribution consists of proposing a novel building block for deep neural networks, described in Eq. (3.5). This is equivalent to learning an approximation $\mathcal{F}$ of the residual $\mathcal{F}(x) = \mathcal{H}(x) - x$, thus the name.

$$y = \text{relu}(\mathcal{H}(x)) = \text{relu}(\mathcal{F}(x) + x) \tag{3.3}$$

Thanks to the shortcut connection, the gradient signal is assured to have an identity side channel (Eq. (3.4)).

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \left( \frac{\partial \mathcal{F}(x)}{\partial x} + 1 \right) \cdot \mathbb{1}_{\mathcal{F}(x)>0} \tag{3.4}$$

Furthermore, Equation (3.5) restricts $x$ and $\mathcal{F}(x)$ to have the same dimensionality, which is usually not interesting. The authors then introduced a linear projection $W_s$ for the shortcut connection:

$$y = \text{relu}\left(\mathcal{F}(x) + xW_s^T\right) \tag{3.5}$$

Note that the form of the primary mapping $\mathcal{F}$ is flexible. It usually composes two or more convolutional layers and also other normalization techniques. The detailed architectural design will be studied in the next section.

### ResNet Architectures

In the original paper, the authors used the residual block presented above to build a deep CNN architecture called "ResNet". Figure 3.1 shows the typical building block of ResNet in which $\mathcal{F}(x)$ consists of two (BasicBlock) or three (Bottleneck) convolutional layers. Additionally, each convolution layer is accompanied by a Batch Normalization layer [5], which is not shown in the figure.

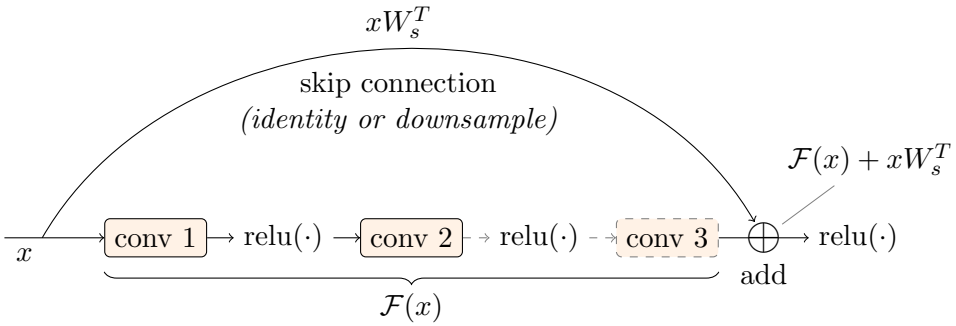



Figure 3.1: ResNet's block: BasicBlock consists of 2 layers and Bottleneck consists of 3 layers.

Figure 3.2 represents the blueprint of ResNet architecture. It comprises one stem convolution, max pooling, four-layer groups that stack multiple blocks as in Fig 3.1, one global average pooling to aggregate high-level features, and a final linear classifier. Notably, the inclusion of the avgpool before the last fc allows the network to admit an almost arbitrary input image size. In the original publication, five network variants were proposed that differed in their total number of learnable layers, namely ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152. Their configurations are detailed in Tab. 3.1.

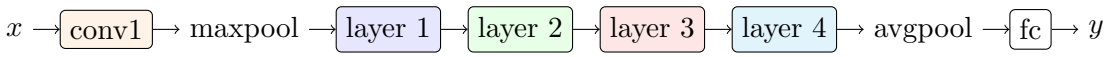



Figure 3.2: ResNet Architecture.

| **Layer** | ResNet-18 | ResNet-34 | ResNet-50 | ResNet-101 | ResNet-152 |
|---|---|---|---|---|---|
| conv1 | $7 \times 7, 64$ conv, stride 2 | | | | |
| maxpool | $3 \times 3$ max pooling, stride 2 | | | | |
| layer1 | $\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$ |
| layer2 | $\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$ | $\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$ | $\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$ | $\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$ |
| layer3 | $\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$ | $\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$ | $\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$ | $\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$ |
| layer4 | $\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$ |
| avgpool | $1 \times 1$ global average pooling | | | | |
| fc | $n$-d fully-connected output layer, softmax | | | | |

Table 3.1: Hyperparameters of ResNet architecture variants proposed in [3].

Thanks to the introduction of the skip connection, ResNet was able to go much deeper in terms of depth. This results in a breakthrough in classification error rate in the ImageNet dataset [6] whereas previous architectures [7, 8] tend to saturate as the depth grows. In this work, due to computational frugality, we will only investigate the ResNet-18 and ResNet-50 architectures.

### 3.1.2. Softmax Loss Function

In classification problems, the most widely used loss function is categorical Cross-Entropy (BCE for binary classification problems), also sometimes referred to as the softmax loss.

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{e^{x_i W_{y_i}^T + b_{y_i}}}{\sum_{j=1}^{N} e^{x_i W_{y_j}^T + b_j}} \tag{3.6}$$

**Margin-based Loss Functions**

Margin-based variants of the aforementioned loss function are commonly adopted in face recognition models. By adding a margin to the softmax loss, they encourage the learned feature space to be more discriminative. This in turn allows modern face recognition methods to achieve relatively high verification accuracy.

Over the last decade, there have been a myriad of such loss functions proposed in the literature. The key idea of such a class of loss functions is to let the row of the weight in the last fully connected layer serve as the class centers in the angular space and then penalize the intra-class angles. To be more specific, in the traditional scheme, the logits are computed as follows:

$$y = zW^T + b \tag{3.7}$$

Whereas in margin-based cross-entropy-guided networks, the bias term $b$ is omitted for simplicity, and the logits for classification decision is the cosine of $\theta$, i.e. the angular distances between the deep features and the corresponding weights:

$$y = \cos\theta = \frac{zW^T}{|z| \cdot |W|} \tag{3.8}$$

The first of their kind is SphereFace [9], which penalizes the loss as in Eq. (3.9).

$$\mathcal{L}_{\text{SphereFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos\left(m\theta_{y_i}\right)}}{e^{s\cos\left(m\theta_{y_i}\right)} + \sum_{j\neq y_i} e^{s\cos\theta_j}} \tag{3.9}$$

Two later extensions namely CosFace [10] and ArcFace [11] proposed different manners for inserting the margin $m$ into the softmax criteria, which are called "additive margin" (Eq. (3.10)) for CosFace and "angular margin" (Eq. (3.11)) for ArcFace.

$$\mathcal{L}_{\text{CosFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos\theta_{y_i}+m}}{e^{s\cos\theta_{y_i}+m} + \sum_{j\neq y_i} e^{s\cos\theta_j}} \tag{3.10}$$

$$\mathcal{L}_{\text{ArcFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos\left(\theta_{y_i}+m\right)}}{e^{s\cos\left(\theta_{y_i}+m\right)} + \sum_{j\neq y_i} e^{s\cos\theta_j}} \tag{3.11}$$

CurricularFace [12] introduced the notions of curriculum learning:

$$\mathcal{L}_{\text{CurricularFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos\left(\theta_{y_i}+m\right)}}{e^{s\cos\left(\theta_{y_i}+m\right)} + \sum_{j\neq y_i} e^{N(t,\cos\theta_j)}} \tag{3.12}$$

where

$$N\left(t, \cos\theta_j\right) = \begin{cases} \cos\left(\theta_j\right) & \text{if } s\cos\left(\theta_{y_i} + m\right) \geq \cos\theta_j \\ \cos\left(\theta_j\right)\left(t + \cos\theta_j\right) & \text{if } s\cos\left(\theta_{y_i} + m\right) < \cos\theta_j \end{cases} \tag{3.13}$$

and $t$ is a parameter that increases as the training progresses.

More recently, one of the reigning state-of-the-art techniques called AdaFace [1] proposed using image quality as guiding information.

$$\mathcal{L}_{\text{AdaFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\left(\cos\left(\theta_{y_i}+g_{\text{angle}}\right)-g_{\text{add}}\right)}}{e^{s\left(\cos\left(\theta_{y_i}+g_{\text{angle}}\right)-g_{\text{add}}\right)} + \sum_{j\neq y_i} e^{s\cos\theta_j}} \tag{3.14}$$

where $g_{\text{angle}}$ and $g_{\text{add}}$ are image quality indicators computed from the normalized feature $\widehat{\|z\|}$:

$$\widehat{\|z\|} = \left\lfloor \frac{\|z\| - \mu_z}{\sigma_z/h} \right\rceil_{-1}^{1} \quad \text{(where } \lfloor\cdot\rceil_{-1}^{1} \text{ indicates } [-1,1] \text{ clipping function)} \tag{3.15}$$

$$g_{\text{angle}} = -m \cdot \widehat{\|z\|} \tag{3.16}$$

$$g_{\text{add}} = m \cdot \widehat{\|z\|} + m \tag{3.17}$$

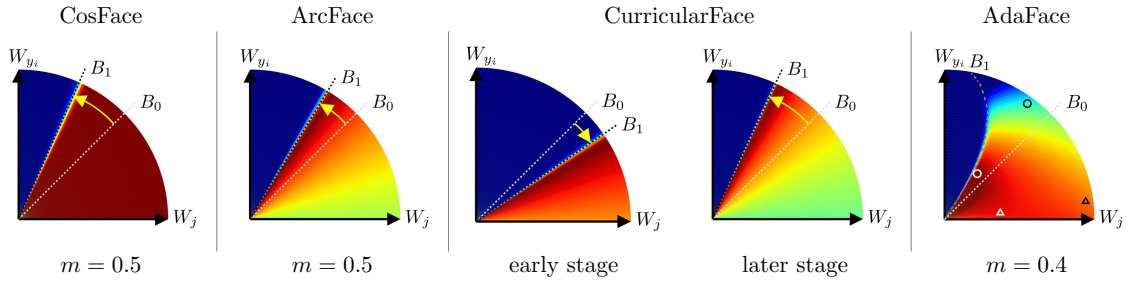



Figure 3.3: Faint illustration of how margin-based loss functions impact the decision boundary from $B_0$ to $B_1$. Images are taken from [1].

The behaviors of these loss functions are illustrated in Fig. 3.3. Besides, there have been many other techniques that belong to this research direction. Since it is not the primary aim of our project, we will only stop at a very high-level idea and not dive deeper into their details.

## 3.2. Methods

### 3.2.1. Architecture

We used a slightly modified version of the ResNet-18 and ResNet-50 architectures. We observe that the target images that we will be using have a relatively small resolution of $32 \times 32$. We

argue that with such small input dimensionality, the initial conv layer with a smaller receptive field would produce more detailed features. Therefore, we downscaled the hyperparameters of the stem convolution layer from the original of kernel_size $= (7 \times 7)$, stride $= (3 \times 3)$, padding $= (2 \times 2)$ to kernel_size $= (3 \times 3)$, stride $= (2 \times 2)$, padding $= (1 \times 1)$. Note that this modification does not affect the number of parameters in the subsequent layers since after the convolutional layers, the features are squashed to a single vector with a global average pooling operation.

### 3.2.2. Loss Functions

First, we train the ResNet models with the vanilla multiclass Cross-Entropy to show the effectiveness of residual architectures compared to traditional single-stream convolutional neural networks.

Then, we further compare it with a state-of-the-art margin-based criterion, AdaFace [1]. Although it was originally introduced to tackle face recognition at different image quality, we argue that the quality indicator can still be useful even in uniform-quality datasets because different augmentations are applied. The objective of this investigation is to validate whether a discriminative loss function widely used in biometrics can help improve performance in a simple classification task.

### 3.2.3. Data Pre-processing and Augmentation

#### Normalization

Opposite to the training of the baseline model, we standardize the pixel values such that they have zero-mean and unit-variance before being fed into the neural network.

$$X = \frac{X - \mu}{\sigma^2} \tag{3.18}$$

where $\mu$ and $\sigma$ are the channel-wise mean and standard deviation computed from the training set.

#### Augmentation

Data augmentation has become a crucial technique for reducing the generalization gap of deep neural networks. For this purpose, we use a composition of random horizontal flipping, and random color jitter. These augmentations are applied in a cascaded manner with probability 1 on every sample, and can be easily implemented using the `torchvision` library as in Lst. 3.1. We empirically observed that such simple transforms work even better for our setting than AutoAugment [13], in which complicated augmentations are learned specifically for a chosen dataset.

Listing 3.1: Code to initialize image augmentation.

```python
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=(0.8, 1.2),
                           contrast=(0.8, 1.2),
                           saturation=(0.8, 1.2),
                           hue=(-0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize(  # statistics computed from the training set
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2471, 0.2435, 0.2616]),
])
```

## 3.3. Experiments

### 3.3.1. Dataset

Following the baseline in the lab session, we conducted experiments on the CIFAR-10 dataset [2]. The dataset is pre-split into training and testing subsets, consisting of 50000 and 10000 $32 \times 32$ RGB images, respectively. There are 10 classes with an evenly distributed number of samples. Figure 3.4 showcases 100 images (10 for each class) sampled from the dataset.

### 3.3.2. Evaluation Metrics

We employ the standard metrics for evaluating classification models. Their formulas are given in the following with $\mathrm{TP}, \mathrm{FP}, \mathrm{TN}, \mathrm{FN}$ denoting true positives, false positives, true negatives, and false negatives, respectively.

**Accuracy**

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} \tag{3.19}$$

**Precision**

$$\text{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{3.20}$$

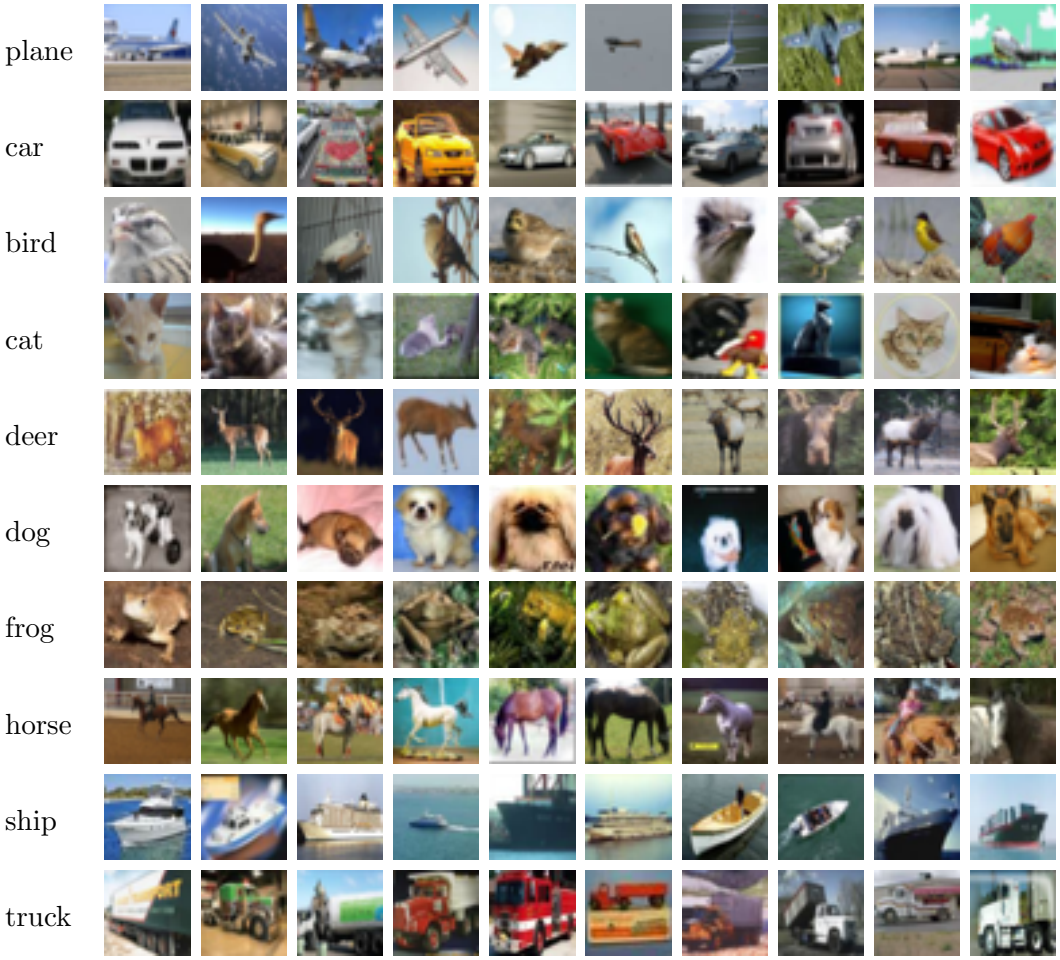



Figure 3.4: Example images in CIFAR-10 dataset. Images are taken from [2].

### Recall

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.21}$$

### F1 Score

$$\text{F1\_score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.22}$$

### Area Under ROC Curve (AUROC)

To draw a ROC curve, True Positive Rate (TPR) and False Positive Rate (FPR) are needed and can be computed as in Eq. (3.24). Each prediction instance of represents one point in the ROC space. We then compute the area under the drawn curve, and the perfect classifier has its auroc of 1.

$$\text{TPR} = \frac{\text{TP}}{\text{P}} \tag{3.23}$$

$$\text{FPR} = \frac{\text{FP}}{\text{N}} \tag{3.24}$$

**Area Under Precision-Recall Curve (AUPRC)**

A Precision-Recall curve consists of points corresponding to different thresholds. Their locations represent the resulting Precision and Recall when we choose that threshold. Similar to auroc, the higher the auprc, the better the model.

### 3.3.3. Implementation Details

We ran all experiments on a Linux device with an AMD Ryzen 9 3950X (32) @ 3.500GHz CPU and an NVIDIA GeForce RTX 3090 GPU. We implemented the architecture using the PyTorch framework [14]. We used Kaiming initialization [15] for all conv layers except those at the residual initialized with zero following [16]. The models are trained for 250 epochs using AdamW optimizer [17] with Cosine annealing learning rate scheduler [18]. All instances utilized the same random seed for the sake of fair comparison and reproducibility.

### 3.3.4. Results on CIFAR-10

**Qualitative results**

Table 3.2 lists the evaluation metrics computed from the test set of CIFAR-10 using the baseline CNN presented in Chapter 2, two ResNet models trained using CE and different margin-based loss functions. Since the accuracy, precision, recall, and f1 score of all models are accidentally identical, we only report accuracy. Regarding accuracy, both ResNet-18 models achieve above 90% while ResNet-50 are around 92%. Interestingly, CE still produces the best classifiers. We hypothesize that it is because CIFAR-10 is a small dataset in terms of the number of classes, meanwhile, metric learning-based loss functions are more suited for tasks with multiple thousands of classes.

| **Metrics** | Baseline | ResNet-18 | | | | ResNet-50 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | CE | ArcFace | CurrFace | AdaFace | CE | ArcFace | CurrFace | AdaFace |
| accuracy | 0.6977 | **0.9079** | 0.9013 | 0.9025 | 0.9048 | **0.9217** | 0.9193 | 0.9199 | 0.9203 |
| auroc | 0.8321 | **0.9943** | 0.9860 | 0.9804 | 0.9844 | **0.9957** | 0.9925 | 0.9942 | 0.9945 |
| auprc | 0.5252 | **0.9625** | 0.9477 | 0.9394 | 0.9469 | **0.9708** | 0.9631 | 0.9682 | 0.9691 |

Table 3.2: Results of Baseline, ResNet-18, and ResNet-50 models on CIFAR-10 dataset.

We plot the accuracy evolution of ResNet-18 over the training loop in Fig. 3.5. It can be seen that margin-based losses such as AdaFace converge slower and thus are harder to train. However, after epoch 200, both models approach 100% accuracy on the training set and only improve very slightly on the test set afterward.

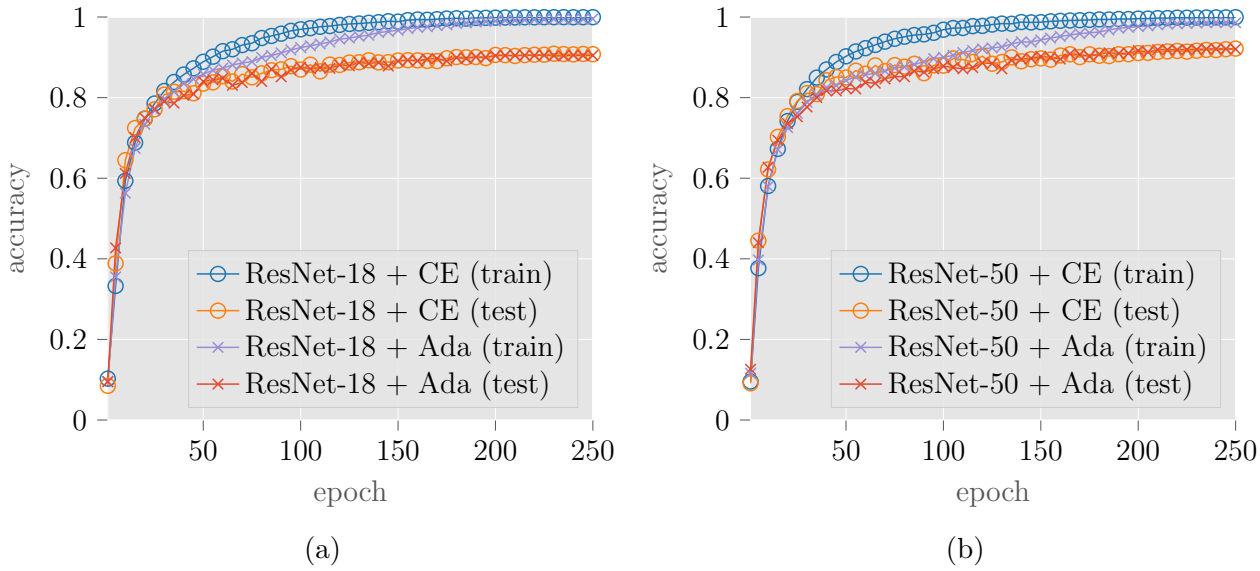



Figure 3.5: Accuracy curves of (a) ResNet-18 and (b) ResNet-50 trained CE and AdaFace loss functions.

### Confusion matrix

In addition to scalar metrics, we compute the confusion matrix of the trained models to analyze per-class performance. Table 3.4 and Table 3.5 show the row-normalized confusion matrices of the ResNet-18 model trained with CE and AdaFace losses, respectively. The diagonal represents the percentage of correct classification, whose elements generally exceed 0.9 for all classes except `cat` and `dog` which can sometimes cause confusion in both models. For comparison, we also plot the confusion matrix of the baseline CNN model in Tab. 3.3.

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.789 | 0.02 | 0.047 | 0.008 | 0.012 | 0.005 | 0.015 | 0.015 | 0.063 | 0.026 |
| car | 0.023 | 0.834 | 0.007 | 0.005 | 0.008 | 0.003 | 0.008 | 0.007 | 0.026 | 0.079 |
| bird | 0.073 | 0.003 | 0.622 | 0.042 | 0.091 | 0.055 | 0.058 | 0.029 | 0.017 | 0.01 |
| cat | 0.03 | 0.016 | 0.085 | 0.431 | 0.084 | 0.182 | 0.086 | 0.037 | 0.026 | 0.023 |
| deer | 0.028 | 0.004 | 0.074 | 0.041 | 0.695 | 0.027 | 0.029 | 0.074 | 0.016 | 0.012 |
| dog | 0.019 | 0.003 | 0.09 | 0.15 | 0.053 | 0.575 | 0.039 | 0.052 | 0.012 | 0.007 |
| frog | 0.01 | 0.007 | 0.067 | 0.048 | 0.083 | 0.02 | 0.737 | 0.011 | 0.011 | 0.006 |
| horse | 0.022 | 0.005 | 0.049 | 0.027 | 0.071 | 0.062 | 0.006 | 0.731 | 0.005 | 0.022 |
| ship | 0.108 | 0.031 | 0.014 | 0.012 | 0.009 | 0.008 | 0.002 | 0.004 | 0.793 | 0.019 |
| truck | 0.042 | 0.085 | 0.015 | 0.009 | 0.007 | 0.011 | 0.01 | 0.011 | 0.04 | 0.77 |

Table 3.3: Confusion matrix of the baseline CNN model.

### Visualization of Learned Feature Spaces

Since the feature dimensions of models are very high and inconsistent, we utilized the UMAP [19] to perform dimensionality reduction first. Figure 3.6 illustrates the feature space of Baseline, ResNet18 + CE, and ResNet-18 + AdaFace; both with 500 random samples (50 for each class, denoted by different colors) taken from the training set of CIFAR-10.

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.926 | 0.005 | 0.015 | 0.011 | 0.006 | 0.002 | 0.002 | 0.006 | 0.019 | 0.008 |
| car | 0.005 | 0.957 | 0.001 | 0.001 | 0.001 | 0. | 0.004 | 0.001 | 0.005 | 0.025 |
| bird | 0.018 | 0. | 0.88 | 0.024 | 0.029 | 0.019 | 0.02 | 0.007 | 0.001 | 0.002 |
| cat | 0.006 | 0.001 | 0.036 | 0.808 | 0.022 | 0.079 | 0.02 | 0.016 | 0.003 | 0.009 |
| deer | 0.007 | 0.002 | 0.016 | 0.018 | 0.906 | 0.011 | 0.011 | 0.027 | 0.002 | 0. |
| dog | 0.003 | 0.002 | 0.014 | 0.087 | 0.021 | 0.846 | 0.009 | 0.014 | 0.002 | 0.002 |
| frog | 0.006 | 0.002 | 0.015 | 0.024 | 0.008 | 0.007 | 0.935 | 0.002 | 0. | 0.001 |
| horse | 0.006 | 0.001 | 0.009 | 0.016 | 0.02 | 0.016 | 0.001 | 0.926 | 0.002 | 0.003 |
| ship | 0.024 | 0.007 | 0.002 | 0.005 | 0. | 0. | 0.002 | 0. | 0.948 | 0.012 |
| truck | 0.004 | 0.03 | 0.003 | 0.003 | 0. | 0.001 | 0.001 | 0.001 | 0.01 | 0.947 |

Table 3.4: ResNet-18 + CE

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.927 | 0.004 | 0.019 | 0.01 | 0.004 | 0.003 | 0.002 | 0.004 | 0.017 | 0.01 |
| car | 0.006 | 0.948 | 0.001 | 0.003 | 0.001 | 0.002 | 0.002 | 0.001 | 0.008 | 0.028 |
| bird | 0.02 | 0. | 0.868 | 0.029 | 0.037 | 0.02 | 0.012 | 0.012 | 0.002 | 0. |
| cat | 0.013 | 0.001 | 0.021 | 0.804 | 0.024 | 0.096 | 0.021 | 0.011 | 0.005 | 0.004 |
| deer | 0.003 | 0.001 | 0.024 | 0.025 | 0.894 | 0.017 | 0.013 | 0.023 | 0. | 0. |
| dog | 0.003 | 0. | 0.017 | 0.085 | 0.02 | 0.852 | 0.003 | 0.015 | 0.001 | 0.004 |
| frog | 0.002 | 0. | 0.013 | 0.029 | 0.007 | 0.009 | 0.934 | 0.003 | 0.001 | 0.002 |
| horse | 0.007 | 0. | 0.006 | 0.018 | 0.018 | 0.013 | 0.002 | 0.932 | 0.002 | 0.002 |
| ship | 0.022 | 0.006 | 0.005 | 0.01 | 0.001 | 0.001 | 0.001 | 0.001 | 0.943 | 0.01 |
| truck | 0.008 | 0.028 | 0.004 | 0.004 | 0. | 0.001 | 0.001 | 0.003 | 0.008 | 0.943 |

Table 3.5: ResNet-18 + AdaFace

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.921 | 0.002 | 0.019 | 0.011 | 0.007 | 0.001 | 0.002 | 0.005 | 0.027 | 0.005 |
| car | 0.003 | 0.968 | 0. | 0.002 | 0. | 0. | 0.001 | 0. | 0.01 | 0.016 |
| bird | 0.01 | 0. | 0.904 | 0.023 | 0.021 | 0.013 | 0.018 | 0.006 | 0.004 | 0.001 |
| cat | 0.008 | 0.002 | 0.022 | 0.842 | 0.015 | 0.069 | 0.025 | 0.012 | 0.003 | 0.002 |
| deer | 0.004 | 0.001 | 0.021 | 0.018 | 0.915 | 0.012 | 0.012 | 0.015 | 0.001 | 0.001 |
| dog | 0.003 | 0. | 0.014 | 0.072 | 0.01 | 0.883 | 0.006 | 0.01 | 0.002 | 0. |
| frog | 0.003 | 0. | 0.014 | 0.026 | 0.005 | 0.008 | 0.938 | 0.003 | 0.001 | 0.002 |
| horse | 0.003 | 0.001 | 0.005 | 0.009 | 0.011 | 0.017 | 0.001 | 0.948 | 0.001 | 0.004 |
| ship | 0.017 | 0.003 | 0.006 | 0.007 | 0. | 0.001 | 0.002 | 0. | 0.957 | 0.007 |
| truck | 0.008 | 0.028 | 0.003 | 0.005 | 0. | 0.001 | 0. | 0.001 | 0.013 | 0.941 |

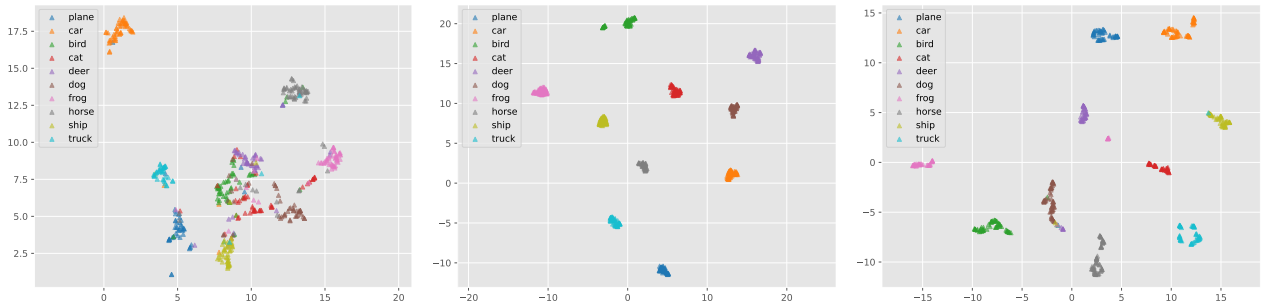
Table 3.6: ResNet-50 + CE

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.936 | 0.005 | 0.017 | 0.002 | 0.003 | 0.001 | 0.003 | 0.004 | 0.02 | 0.009 |
| car | 0.002 | 0.971 | 0. | 0. | 0. | 0.001 | 0. | 0. | 0.003 | 0.023 |
| bird | 0.012 | 0.001 | 0.9 | 0.018 | 0.023 | 0.021 | 0.013 | 0.006 | 0.002 | 0.004 |
| cat | 0.006 | 0.002 | 0.023 | 0.81 | 0.026 | 0.084 | 0.025 | 0.015 | 0.002 | 0.007 |
| deer | 0.007 | 0.001 | 0.02 | 0.017 | 0.923 | 0.011 | 0.005 | 0.016 | 0. | 0. |
| dog | 0.003 | 0. | 0.011 | 0.062 | 0.012 | 0.891 | 0.004 | 0.012 | 0.001 | 0.004 |
| frog | 0.006 | 0.003 | 0.019 | 0.02 | 0.004 | 0.008 | 0.936 | 0.001 | 0.002 | 0.001 |
| horse | 0.005 | 0. | 0.007 | 0.012 | 0.011 | 0.016 | 0.002 | 0.942 | 0.003 | 0.002 |
| ship | 0.025 | 0.009 | 0.004 | 0.006 | 0. | 0.001 | 0.001 | 0.001 | 0.945 | 0.008 |
| truck | 0.005 | 0.028 | 0.004 | 0.004 | 0. | 0. | 0.001 | 0.002 | 0.007 | 0.949 |

Table 3.7: ResNet-50 + AdaFace

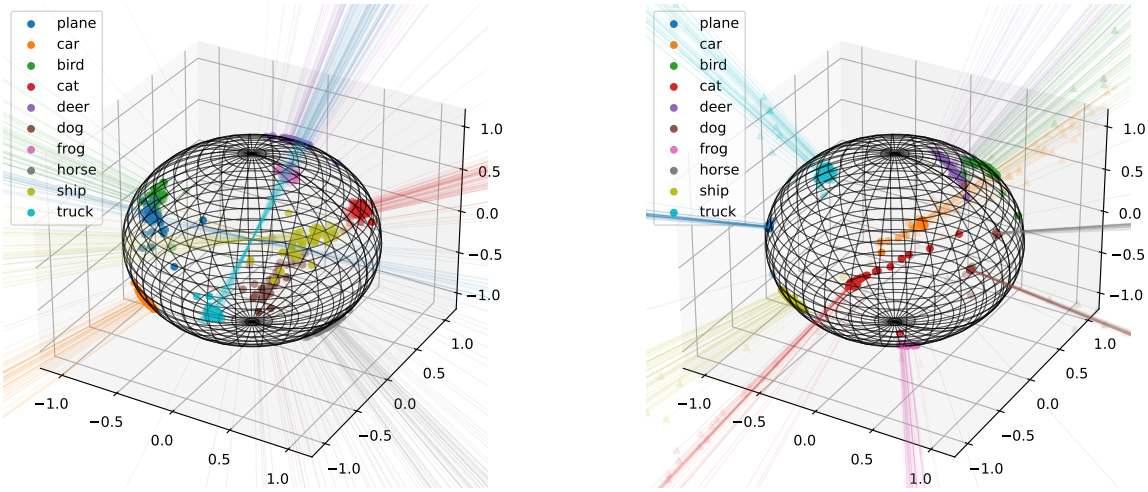
At first glance, the feature space produced by CE may seem to have more concentrated class blobs. However, UMAP visualization can be misleading because it uses Euclidean distances. Meanwhile, both loss functions work on angular space; hence, we can obtain a hugely contrasting perspective by using a more appropriate visualization approach. We modified the last linear transform of the ResNet-18 model before training. This allows us to extract 2 or 3-D features while preserving the structure created by the loss functions. The visualization procedure is further detailed in Appendix A.3.

Analogously, we visualize 500 samples fed through ResNet-18 trained with CE and AdaFace in




(a) (b) (c)

Figure 3.6: UMAP(`n_neighbors=6`)-transformed feature space of (a) Baseline, (b) ResNet-18 + CE, and (c) ResNet-18 + AdaFace.




(a) (b)

Figure 3.7: Illustration of normalized feature space of ResNet-18 model trained using (a) CE and (b) AdaFace.

Fig. 3.7. We then cast the original deep features (translucent △ markers) onto a unit sphere (◯ markers) to obtain the normalized version. Consequently, we can clearly observe the within-class compactness of the normalized feature space of AdaFace in terms of angular discrepancy compared to that of CE.

### 3.3.5. Low-Resolution Stem Convolution

As mentioned in Sec. 3.2.1, we reduced the hyperparameters of the first convolutional layer. Table 3.8 shows the results of this experiment, in which we compare our modded ResNet-18 model with the original ResNet-18 architecture used in [3] under CE loss.

| **Metrics** | ResNet-18† | ResNet-18 (our) |
|---|---|---|
| accuracy | 0.8581 | 0.9079 |
| auroc | 0.9870 | 0.9943 |
| auprc | 0.9271 | 0.9625 |

Table 3.8: Comparison between our modified ResNet-18 and the original ResNet-18†.

### 3.3.6. Complexity Analysis

While ResNet models outperform the baseline by orders of magnitude, they exhibit significant memory and computational complexity costs. We report the total number of parameters, TFLOPs (number of floating point operations in trillions), and TMACs (number of multiply-

add accumulations in trillions) of each model in Tab. 3.9. In which, #params [1] can represent the model's size in memory if considering each parameter to be a 32-bit float number, and the latter two metrics are commonly used to quantify computational complexity.

| **Metrics**↓ | Baseline | ResNet-18 | ResNet-50 |
|---|---|---|---|
| #params | 122,570 | 11,173,962 ($\sim \times 91$) | 23,520,842 ($\sim \times 192$) |
| TFLOPs | 0.0184 | 0.5621 ($\sim \times 30$) | 1.3086 ($\sim \times 71$) |
| TMACs | 0.0091 | 0.2804 ($\sim \times 31$) | 0.6516 ($\sim \times 72$) |

Table 3.9: Number of parameters, TFLOPS, and TMACs of baseline CNN vs ResNet models. The number in parentheses of ResNet presents the relative scale between their metrics and those of the baseline.

## 3.4. Discussion

We observe that ResNet-18 and ResNet-50 can easily achieve very high accuracy on CIFAR-10 at more than a 0.9 correction rate. The results can further be improved using more complex training strategies, for instance: using pre-training techniques, transfer learning of a pre-trained model on a larger dataset such as ImageNet, or simply adding more augmentations.

Also, combining them with margin-based losses moderately deteriorates the performance of the models but produces a much more compact deep feature space, which can be interesting in certain use cases.

[1]The patching of the output layer in margin-based losses following Eq. (3.8) vaguely alters the computed values reported above.

# 4. Conclusion

This project presents a comprehensive analysis and improvement of the image classification task on the CIFAR-10 dataset through two approaches.

• Chapter 2, we developed and evaluated a baseline convolutional neural network model. Detailed analysis of the performance through quantitative metric (*accuracy*) and the feature visualization part. From the baseline model, we gained basic knowledge about CNNs, deep learning models, and classification tasks for multiple classes.

• Chapter 3 is built on the insights gained from the baseline model, our mini-project focuses on enhancing performance by implementing ResNet architectures which are ResNet-18 and ResNet-50, and compares their performance against the baseline model. Data normalization and data augmentation techniques are used to ensure the standardized and diversified input data. Additionally, we expanded our evaluation framework by including metrics beyond accuracy which are precision, recall, and F1-score to provide a more comprehensive understanding of the model's performance. Furthermore, another interesting point is inspired by recent research in cancellable biometrics for face verification [20] that we have reviewed as part of the security module, we tried to integrate margin-based loss functions to enhance the model's discriminative ability.

Overall, the integration of these methodologies resulted in a more robust classification system that outperformed the baseline in terms of both performance and reliability.

# Bibliography


[1] M. Kim, A. K. Jain, and X. Liu, "Adaface: Quality adaptive margin for face recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18750–18759, June 2022.

[2] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 and cifar-100 datasets (canadian institute for advanced research)," 2009.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[4] S. Sun, W. An, F. Tian, F. Nan, Q. Liu, J. Liu, N. Shah, and P. Chen, "A review of multimodal explainable artificial intelligence: Past, present and future," *arXiv preprint arXiv:2412.14056*, 2024.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[9] W. Liu, Y. Wen, Z. Yu, M. Li, B. Raj, and L. Song, "Sphereface: Deep hypersphere embedding for face recognition," *CoRR*, vol. abs/1704.08063, 2017.

[10] H. Wang, Y. Wang, Z. Zhou, X. Ji, Z. Li, D. Gong, J. Zhou, and W. Liu, "Cosface: Large margin cosine loss for deep face recognition," *CoRR*, vol. abs/1801.09414, 2018.

[11] J. Deng, J. Guo, and S. Zafeiriou, "Arcface: Additive angular margin loss for deep face recognition," *CoRR*, vol. abs/1801.07698, 2018.

[12] Y. Huang, Y. Wang, Y. Tai, X. Liu, P. Shen, S. Li, J. Li, and F. Huang, "Curricularface: Adaptive curriculum learning loss for deep face recognition," *CoRR*, vol. abs/2004.00288, 2020.

[13] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data," *CoRR*, vol. abs/1805.09501, 2018.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[16] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[17] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[18] I. Loshchilov and F. Hutter, "SGDR: stochastic gradient descent with restarts," *CoRR*, vol. abs/1608.03983, 2016.

[19] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020.

[20] K. Ito, T. Kozu, H. Kawai, G. Hanawa, and T. Aoki, "Cancelable face recognition using deep steganography," *IEEE Transactions on Biometrics, Behavior, and Identity Science*, vol. 6, no. 1, pp. 87–102, 2023.

[21] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008.

# A. Appendix

## A.1. A Fair Experiment

The results reported in Sec. 3.3.4 are undoubtedly unfair for the baseline CNN model. This is because we only trained the baseline for 10 epochs while ResNets are trained for 250 epochs with other techniques such as data standardization, augmentations, and learning rate scheduling. In Tab. A.1 and Tab. A.2, we report the accuracy and confusion matrix of the baseline model but trained with exactly the configurations we used for ResNet, dubbed "Baseline+".

| **Metrics**↑ | baseline | baseline+ |
|---|---|---|
| accuracy | 0.6977 | 0.8336 |
| auroc | 0.8321 | 0.9842 |
| auprc | 0.5252 | 0.9096 |

Table A.1: Results of Baseline+ on CIFAR-10.

| Actual \ Predicted | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 0.85 | 0.01 | 0.035 | 0.018 | 0.005 | 0.004 | 0.004 | 0.008 | 0.049 | 0.017 |
| car | 0.012 | 0.915 | 0.002 | 0.004 | 0.001 | 0.003 | 0.004 | 0. | 0.021 | 0.038 |
| bird | 0.029 | 0.001 | 0.767 | 0.035 | 0.062 | 0.039 | 0.04 | 0.013 | 0.009 | 0.005 |
| cat | 0.017 | 0.005 | 0.048 | 0.667 | 0.047 | 0.131 | 0.039 | 0.023 | 0.01 | 0.013 |
| deer | 0.008 | 0.002 | 0.043 | 0.025 | 0.829 | 0.017 | 0.029 | 0.039 | 0.007 | 0.001 |
| dog | 0.009 | 0. | 0.036 | 0.107 | 0.031 | 0.77 | 0.011 | 0.028 | 0.001 | 0.007 |
| frog | 0.005 | 0.001 | 0.031 | 0.04 | 0.014 | 0.012 | 0.885 | 0.006 | 0.004 | 0.002 |
| horse | 0.009 | 0. | 0.017 | 0.026 | 0.041 | 0.042 | 0.002 | 0.856 | 0.001 | 0.006 |
| ship | 0.038 | 0.012 | 0.009 | 0.007 | 0.001 | 0.003 | 0.003 | 0.003 | 0.902 | 0.022 |
| truck | 0.021 | 0.043 | 0.005 | 0.004 | 0.003 | 0.001 | 0.001 | 0.006 | 0.021 | 0.895 |

Table A.2: Confusion matrix of Baseline+.

Surprisingly, being able to reach 83% accuracy, Baseline+'s performance is not too far away from ResNet models. Its accuracy curve is plotted in Fig. A.1.

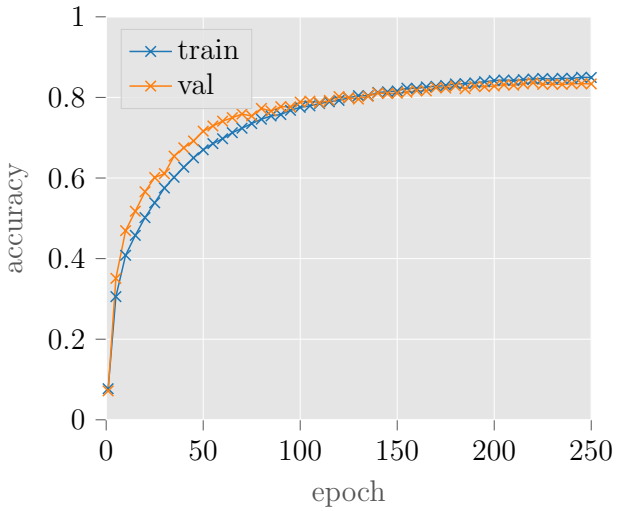



Figure A.1: Accuracy curve of Baseline+.

## A.2. Importance of Skip Connections

In this section, we analyze the importance of skip connections in the building block of ResNets (Fig. 3.1). We remove all skip connections to create a variant of ResNet called PlainNet. This idea has also been coined by the authors in [3]. We then train PlainNet-50 to compare its performance against ResNet-50, both using the vanilla CE loss function and identical configurations. The new 50-layered deep model turns out to be even less robust than ResNet-18, with an accuracy of 89.7%. This experiment proves the significant role of skip connections (or residual learning). The results are reported in Tab. A.3 and Tab. A.4.

| **Metrics**↑ | ResNet-50 | PlainNet-50 |
|---|---|---|
| accuracy | 0.9217 | 0.8970 |
| auroc | 0.9957 | 0.9923 |
| auprc | 0.9708 | 0.9556 |

Table A.3: Results of PlainNet-50 on CIFAR-10.

| | | **Predicted** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *plane* | *car* | *bird* | *cat* | *deer* | *dog* | *frog* | *horse* | *ship* | *truck* |
| **Actual** | plane | 0.914 | 0.006 | 0.022 | 0.009 | 0.003 | 0.001 | 0.003 | 0.005 | 0.026 | 0.011 |
| | car | 0.003 | 0.957 | 0. | 0.001 | 0.001 | 0.001 | 0.001 | 0. | 0.004 | 0.032 |
| | bird | 0.023 | 0.001 | 0.859 | 0.03 | 0.023 | 0.027 | 0.02 | 0.01 | 0.004 | 0.003 |
| | cat | 0.007 | 0.002 | 0.044 | 0.77 | 0.027 | 0.096 | 0.026 | 0.013 | 0.006 | 0.009 |
| | deer | 0.005 | 0. | 0.026 | 0.018 | 0.899 | 0.014 | 0.015 | 0.02 | 0.002 | 0.001 |
| | dog | 0.004 | 0.002 | 0.017 | 0.086 | 0.024 | 0.846 | 0.004 | 0.014 | 0.001 | 0.002 |
| | frog | 0.005 | 0.001 | 0.021 | 0.028 | 0.008 | 0.013 | 0.921 | 0. | 0.001 | 0.002 |
| | horse | 0.005 | 0. | 0.011 | 0.02 | 0.016 | 0.023 | 0. | 0.921 | 0.003 | 0.001 |
| | ship | 0.025 | 0.008 | 0.001 | 0.006 | 0.001 | 0.001 | 0.004 | 0.002 | 0.943 | 0.009 |
| | truck | 0.006 | 0.031 | 0.002 | 0.007 | 0. | 0.001 | 0.003 | 0.002 | 0.008 | 0.94 |

Table A.4: Confusion matrix of PlainNet-50.

## A.3. Details of Visualization

In Section 3.3.4, instead of using dimensionality reduction techniques such as T-SNE [21] or U-Map [19] for visualizing high-dimensional feature space, we split the final linear layer `fc` into two linear layers `fc1` and `fc2` before training. Let $W \in \mathbb{R}^{(d_{out} \times d_{in})}$ and $b \in \mathbb{R}^{d_{out}}$ denote the weight and bias of the classification layer, the logits can be computed as:

$$y = zW^T + b \tag{A.1}$$
$$= zW_1^T W_2^T + b_2 \tag{A.2}$$

where $W_1 \in \mathbb{R}^{(3 \times d_{in})}$ is the weight of `fc1` (which is applied without bias), $W_2 \in \mathbb{R}^{(d_{out} \times 3)}$ is the weight of `fc2`, and $b_2 = b$ is bias of `fc2`. In the case of pre-trained weight initialization, one can choose random $W_2$ and solve for $W_1$ using linear system solvers like `torch.linalg.lstsq` (or vice versa):

$$W_2 W_1 = W \tag{A.3}$$

By doing so, one can visualize the 3D feature space $z' = zW_1^T$ without loss of structure imposed by the loss function.