Illinois Institute of Technology

ILLINOIS TECH

Armour College of Engineering

# Project 2

*Student :*
Carlos Martin
A20595921

*Course :*
ECE 566 Machine and Deep Learning
Due Date: 12/01/2024

November 26, 2024

# Contents

## 1 Introduction

The purpose of this project is to design, train, and evaluate a convolutional neural network (CNN) for multi-class classification of handwritten digits from the MNIST dataset. The dataset includes grayscale images of handwritten digits (0–9) with added correlated noise, making it a challenging classification task. The goal is to identify the most effective neural network architecture and training strategy to achieve high classification accuracy while ensuring the model generalizes well to unseen test data.

This report outlines the step-by-step process of exploring different CNN architectures, layer combinations, and training settings to determine the best-performing model for this task. Key aspects of the project include:

- Designing multiple CNN architectures inspired by popular networks such as AlexNet, ResNet, and GoogleNet, with modifications for the MNIST input size.
- Systematic experimentation with layer configurations, optimization techniques, and regularization to prevent overfitting.
- Evaluation of the models using training, validation, and test datasets to measure accuracy and loss.

The report includes detailed plots of training and validation loss/accuracy over epochs. An analysis of the results, total training time, and rationale for architectural and hyperparameter choices is provided to demonstrate a comprehensive understanding of CNN design principles and their application to noisy image classification.

Through this project, the aim is to develop a CNN that achieves high accuracy on the test set while remaining robust to noise, thereby showcasing the practical application of deep learning techniques to complex real-world problems.

It is crucial to ensure that the designed network does not overfit the training data. Overfitting can result in a model that performs exceptionally well on the training set but fails to generalize to unseen data, leading to poor performance in real-world conditions. The goal is to develop a robust network that achieves a balance between training accuracy and generalization.

## 2 Methodology

The methodology followed in this project involves the design, training, and evaluation of four different convolutional neural networks (CNNs), each inspired by widely recognized architectures in the field of deep learning. These networks have been adapted to the characteristics of the MNIST dataset. Each network was designed with specific architectural principles in mind, leveraging the strengths of the corresponding base architecture while making appropriate modifications for MNIST's smaller image size and simpler classification task.

## 2.1 AlexNet Based Network

The first network in this project is inspired by the AlexNet architecture. This implementation adapts the key principles of AlexNet to the MNIST dataset, which consists of smaller grayscale images and is less complex than the datasets for which AlexNet was originally designed. As such, the network was adjusted to reduce its depth and computational requirements while maintaining its core feature extraction capabilities.

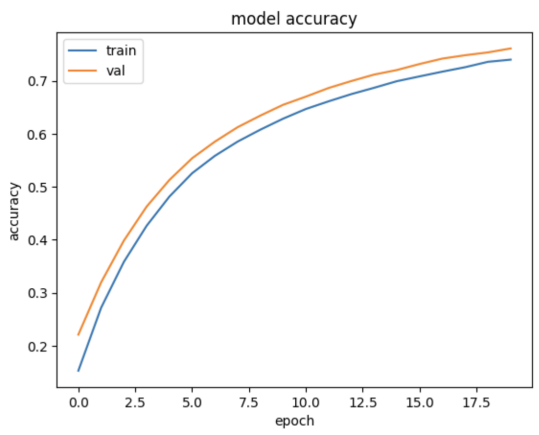
The adapted network begins with a convolutional layer with 32 filters of size 3 x 3, followed by a max pooling layer for spatial downsampling and a batch normalization layer to stabilize training. This pattern is repeated with a second convolutional layer with 64 filters, again followed by max pooling and batch normalization. These layers extract key features from the MNIST images while reducing overfitting and speeding up convergence through normalization.

In the deeper portion of the network, three consecutive 1 x 1 convolutional layers with 128 filters each are used to refine the extracted features. This technique reduces the computational load while focusing on channel-wise interactions between features, a strategy that aligns with modern CNN design principles. The extracted features are then flattened and passed through a dense layer with 128 neurons using the ReLU activation function. Finally, a softmax layer is used to output probabilities for the 10 digit classes, completing the classification task.

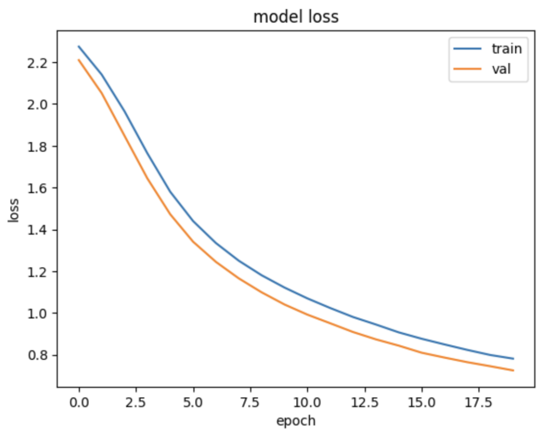
With the following parameters:

- **Batch Size :** 3
- **Validation Split:** 0.2 (20% of the data is used for validation)
- **Number of Epochs:** 20

the performance obtained with this network reaches an accuracy of 74% . Moreover, in the next graph, it is possible to see how the accuracy improves while training the model and going around the number of epochs decided.



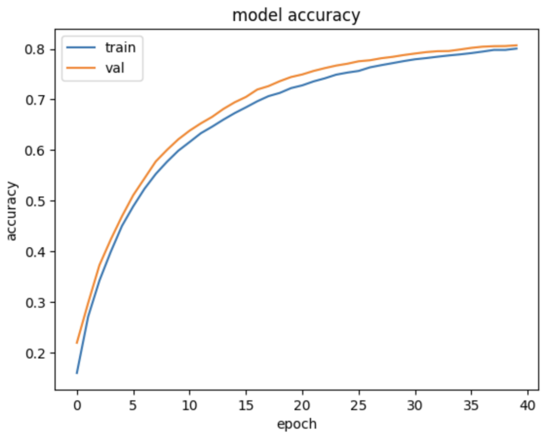
(a) Accuracy obtained while training the AlexNet modified model with 20 epochs



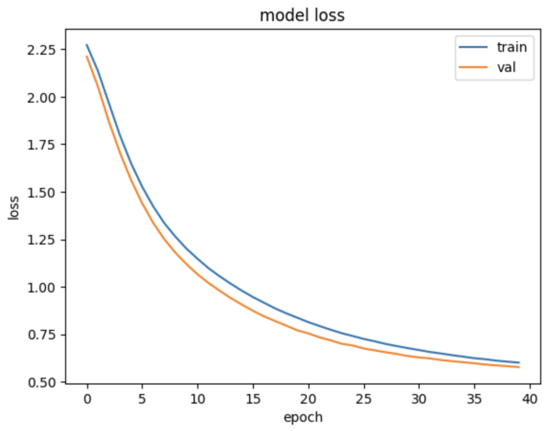
(b) Loss evolution while training with 20 epochs

Figure 1: Results obtained after training a modified AlexNet with 20 epochs.

To further improve the results obtained after modifying the training parameters (batch_size, validation_split, and n_epochs), the number of epochs was increased to 40 to observe how the model's performance evolves over a longer training period.



(a) Accuracy obtained while training the AlexNet modified model with 40 epochs



(b) Loss evolution while training with 40 epochs

Figure 2: Results obtained after training a modified AlexNet with 40 epochs.

The primary difference between the two models lies in the number of training epochs. The first model was trained for **20 epochs**, while the second model was trained for**40 epochs**. This change allows the second model to undergo additional optimization steps, potentially improving its ability to generalize to unseen data.

As shown in Figure 1 and Figure 2, the model trained for 40 epochs demonstrates how the accuracy and loss evolve. The accuracy curves for both training and validation continue to rise steadily between 20 and 40 epochs. This indicates that the network benefits from the additional training time, further optimizing its feature extraction and classification capabilities.

The loss curves for the second model show continued decline during the additional epochs, suggesting that the model is converging better compared to the first model.

However, it is crucial to monitor the training process closely to ensure that the extended training does not lead to overfitting. While the additional epochs improve performance in this case, excessive training beyond the optimal point could result in diminishing returns or even degrade the validation accuracy.

With the results obtained from testing accuracy, we can conclude that the model is not overfitting to the training data. After training for 20 epochs, the test accuracy was 0.7462, and this increased to 0.8022 when the number of epochs was extended to 40. The steady improvement in test accuracy demonstrates that the additional training time allowed the model to learn more effectively without degrading its generalization performance. This indicates that the model is optimizing its ability to classify unseen data, rather than memorizing the training set, confirming that overfitting has been successfully avoided in this scenario.

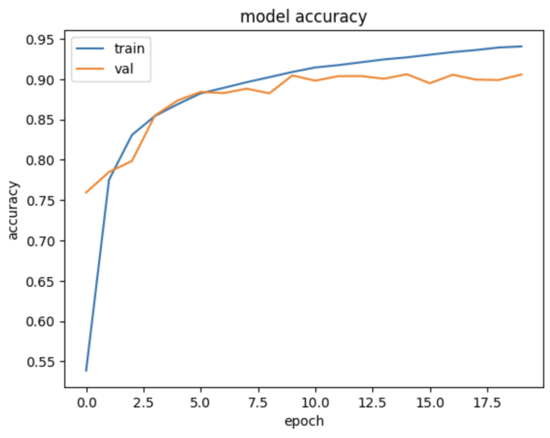
## 2.2 ResNet-Inspired Neural Network

This neural network is inspired by the original ResNet (Residual Network) architecture, which is known for its use of residual connections to address the vanishing gradient problem in deep neural networks. Residual connections allow gradients to propagate effectively through deeper layers, enabling the network to learn richer feature representations. However, this implementation has been adapted for the MNIST dataset, which consists of smaller grayscale images, by simplifying the architecture while retaining the core idea of residual learning.

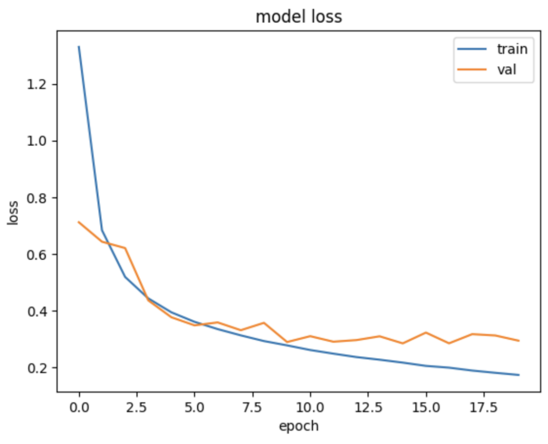
The network begins with an input layer of shape $28 \times 28 \times 1$ to accommodate the grayscale MNIST images. A convolutional layer with 32 filters and a kernel size of $3 \times 3$ is applied, followed by the first residual block. The residual block consists of two $3 \times 3$ convolutional layers with Batch Normalization layers added after each convolution to stabilize the learning process.

After the first residual block, a max pooling layer is used to reduce the spatial dimensions. The second residual block follows, operating with 64 filters. The features extracted by this block are then passed through a global average pooling layer, which reduces the spatial dimensions to a single feature vector for each filter, thus significantly reducing the number of parameters and computation required for the dense layers. The final output layer is a dense layer with a softmax activation function, which produces probabilities for the 10 classes in the MNIST dataset. The network was compiled using the Adam optimizer.

This adaptation simplifies the original ResNet architecture by reducing the number of residual blocks and adjusting the filter sizes, making it more computationally efficient while still leveraging the advantages of residual learning.



(a) Accuracy obtained while training the ResNet modified model with 20 epochs



(b) Loss evolution while training with 20 epochs

Figure 3: Results obtained after training a modified ResNet with 20 epochs.

With the following parameters:

- **Batch Size :** 3
- **Validation Split:** 0.2 (20% of the data is used for validation)

- **Number of Epochs:** 20

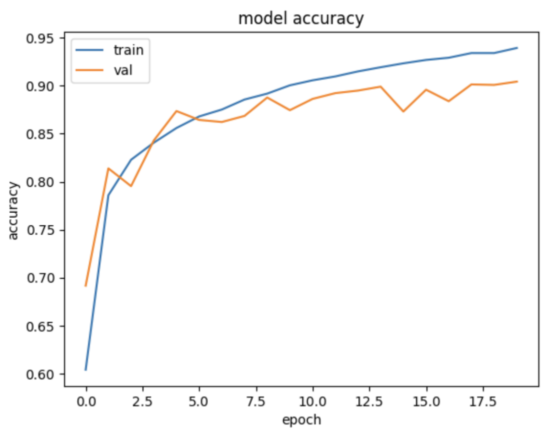
the performance obtained with this network reaches an accuracy of 94% during training and 90% over testing and validation as it is shown in the figure shown above.

To evaluate the impact of batch size on the performance of the ResNet-inspired model, the batch size was increased from 3 to 10 while keeping other parameters constant. This modification revealed several key differences:

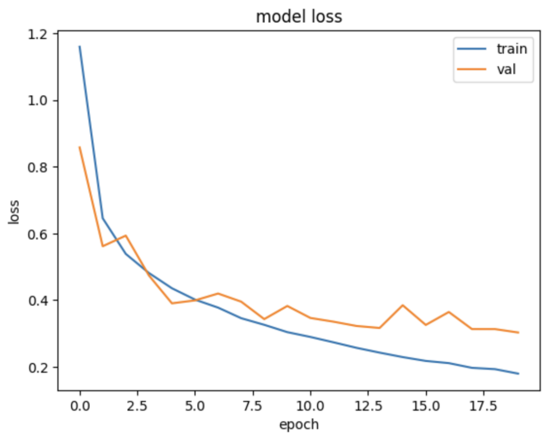
- **Stability of Training:** A larger batch size resulted in smoother and more stable convergence of the training and validation loss curves. This is because a larger batch size provides a more representative estimate of the gradient at each step.
- **Training Speed:** The model trained faster per epoch with a batch size of 10 due to fewer weight updates per epoch. This improved computational efficiency compared to using a smaller batch size of 3.
- **Generalization Performance:** With a batch size of 10, the validation accuracy improved slightly, indicating better generalization. The smaller batch size of 3 caused higher variance in validation performance due to noisier gradient updates.
- **Model Convergence:** The larger batch size allowed the model to converge more predictably across epochs, whereas the smaller batch size led to more fluctuations in both training and validation metrics.

These observations highlight that while a smaller batch size provides finer updates to the model's weights, it may introduce noise that slows convergence and affects generalization. Increasing the batch size to 10 strikes a balance between computational efficiency and training stability for this specific task.

To sum up, the results obtained in this experiment are the one shown below.



(a) Accuracy obtained while training the ResNet modified model with batch size=10



(b) Loss evolution while training with batch size = 10

Figure 4: Results obtained after training a modified ResNet with batch size = 10.

These figures illustrate the accuracy and loss curves obtained after training the ResNet-inspired model with a batch size of 10. In the accuracy graph , it can be observed that while the training accuracy steadily improves and converges towards higher values, the validation accuracy fluctuates significantly and does not converge to a stable value. Similarly, in the loss graph (Figure 1b), the validation loss shows an inconsistent trend, diverging from the smooth reduction observed in the training loss. These results indicate the need to further fine-tune the model parameters or incorporate additional regularization techniques to improve validation performance.

## 2.3 Personal Neural Network

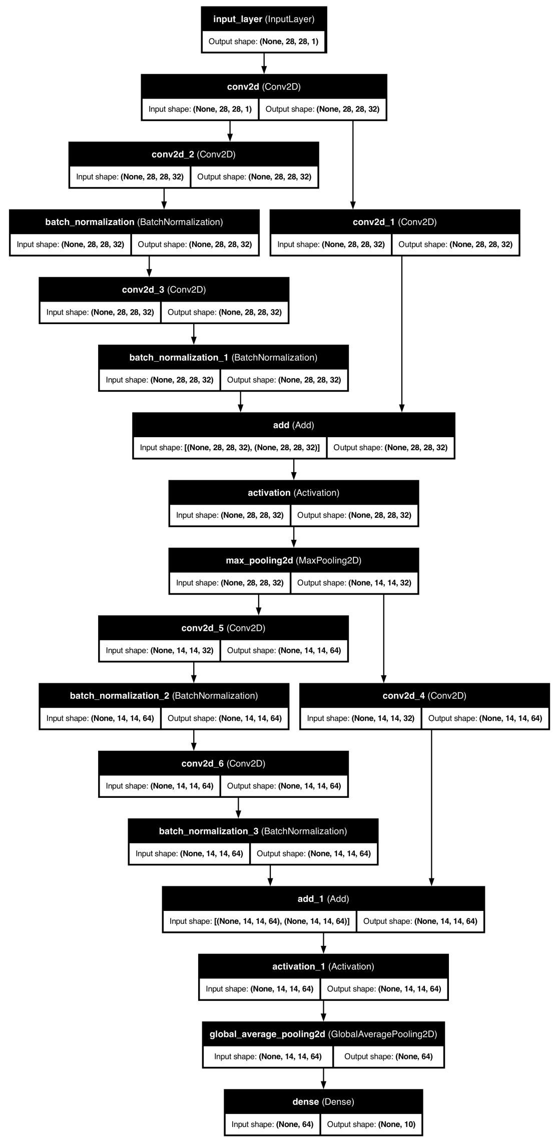
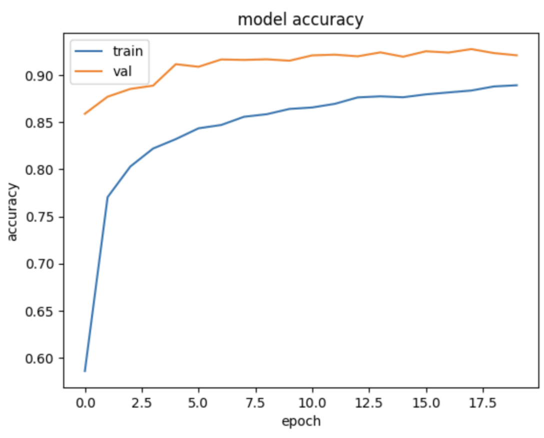

input_layer (InputLayer)
Output shape: (None, 28, 28, 1)
conv2d (Conv2D)
Input shape: (None, 28, 28, 1) | Output shape: (None, 28, 28, 32)
conv2d_2 (Conv2D)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
batch_normalization (BatchNormalization)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
conv2d_1 (Conv2D)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
conv2d_3 (Conv2D)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
batch_normalization_1 (BatchNormalization)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
add (Add)
Input shape: [(None, 28, 28, 32), (None, 28, 28, 32)] | Output shape: (None, 28, 28, 32)
activation (Activation)
Input shape: (None, 28, 28, 32) | Output shape: (None, 28, 28, 32)
max_pooling2d (MaxPooling2D)
Input shape: (None, 28, 28, 32) | Output shape: (None, 14, 14, 32)
conv2d_5 (Conv2D)
Input shape: (None, 14, 14, 32) | Output shape: (None, 14, 14, 64)
batch_normalization_2 (BatchNormalization)
Input shape: (None, 14, 14, 64) | Output shape: (None, 14, 14, 64)
conv2d_4 (Conv2D)
Input shape: (None, 14, 14, 32) | Output shape: (None, 14, 14, 64)
conv2d_6 (Conv2D)
Input shape: (None, 14, 14, 64) | Output shape: (None, 14, 14, 64)
batch_normalization_3 (BatchNormalization)
Input shape: (None, 14, 14, 64) | Output shape: (None, 14, 14, 64)
add_1 (Add)
Input shape: [(None, 14, 14, 64), (None, 14, 14, 64)] | Output shape: (None, 14, 14, 64)
activation_1 (Activation)
Input shape: (None, 14, 14, 64) | Output shape: (None, 14, 14, 64)
global_average_pooling2d (GlobalAveragePooling2D)
Input shape: (None, 14, 14, 64) | Output shape: (None, 64)
dense (Dense)
Input shape: (None, 64) | Output shape: (None, 10)

Figure 5: Model Diagram

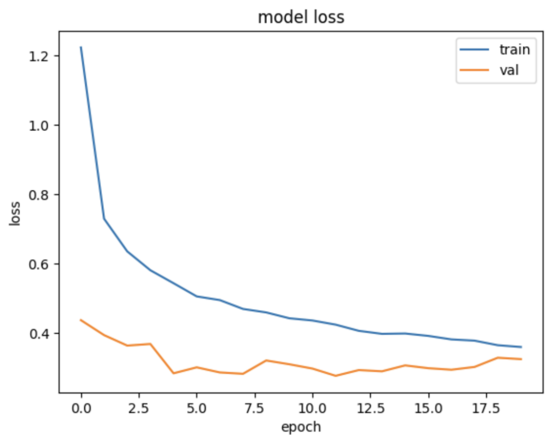
The designed convolutional neural network (CNN) for MNIST dataset consists of three main convolutional blocks, followed by fully connected layers. The input layer accepts grayscale images of size $28 \times 28 \times 1$, suitable for the MNIST dataset. The first convolutional block uses two $3 \times 3$ convolutional layers with 32 filters, followed by batch normalization and max pooling. This block extracts features such as edges and corners while reducing spatial dimensions to prevent overfitting.

The subsequent block doubles the number of filters to 64, applying the same sequence of operations to capture more complex patterns, such as textures and shapes, while further downsampling the feature maps.

The third convolutional block increases the number of filters to 128 and introduces a global average pooling layer instead of flattening. Global average pooling significantly reduces the parameter count by summarizing each feature map into a single value, improving generalization and computational efficiency. Batch normalization is applied after each convolution to prevent exploding or vanishing gradients. This progression from low-level to high-level feature extraction ensures the model captures meaningful information while handling the noisy input effectively.



(a) Accuracy obtained while training the personal neural network created.



(b) Loss evolution while training the personal neural network.

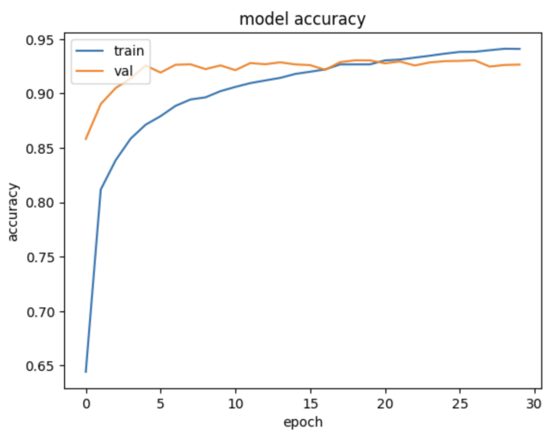
Figure 6: Results obtained after training a personalized neural network.

Finally, the dense layers translate the extracted features into class probabilities for digit classification. A fully connected layer with 128 neurons and ReLU activation refines the feature representations, followed by a dropout layer to reduce overfitting. The output layer uses 10 neurons with softmax activation to assign probabilities for each digit class (0–9).

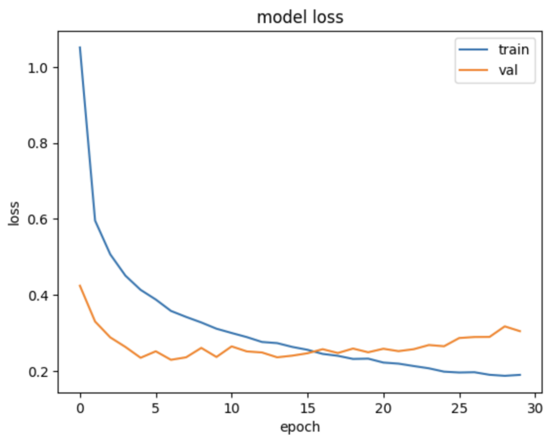
The training results with parameters (Batch size = 3, validation split = 20% and 20 epochs)show that the model progressively improves its training accuracy and reduces its training loss over 20 epochs, reaching a final training accuracy of 88.98% with a loss of 0.3532. The validation accuracy stabilizes around 92%, with the lowest validation loss of 0.2769 achieved at epoch 12. This indicates that the model has effectively learned from the noisy MNIST training data, achieving strong performance without significant overfitting. However, the fluctuations in validation loss after epoch 12 suggest that further fine-tuning, such as learning rate adjustments or additional regularization, may stabilize the performance.

When evaluated on the test set, the model achieves an accuracy of 92.29%, closely matching the validation accuracy. This demonstrates that the model generalizes well to unseen data, effectively handling the noise present in the MNIST dataset. Overall, the results indicate that the chosen architecture and training approach successfully capture the essential features of the noisy digit classification task.

Using a batch size of 5, a validation split of 20%, and training for 30 epochs, the model's accuracy over training and validation data is shown in the figure above. It can be observed that the validation accuracy initially improves and surpasses the training accuracy during the early epochs. However, as the training progresses, the training accuracy continues to improve steadily, while the validation accuracy plateaus and fluctuates around a specific value. Eventually, the training accuracy overtakes the validation accuracy.



(a) Accuracy obtained while training the personal neural network.



(b) Loss evolution while training the personal neural network.

Figure 7: Results obtained after training with 30 epochs a personalized neural network.

This pattern indicates the possibility of overfitting. Overfitting occurs when the model learns to perform exceptionally well on the training data, capturing noise and irrelevant details, but struggles to generalize to unseen data. The divergence between the training and validation accuracy after approximately 10 epochs suggests that the model is beginning to memorize the training data rather than learning generalizable patterns. To address this issue, regularization techniques such as increasing dropout, reducing the number of epochs, or utilizing early stopping should be considered. Additionally, experimenting with a larger batch size or data augmentation could further improve generalization and reduce overfitting.

## 3 Conclusion

This project explored the design, training, and evaluation of three distinct convolutional neural networks (CNNs) for classifying MNIST handwritten database. Each network was inspired by known architectures, including AlexNet, ResNet, and a custom-designed neural network, with modifications to suit the MNIST dataset inout size. The results achieved are:

- **AlexNet-Inspired Network:** The modified AlexNet achieved a test accuracy of **74.62%** after 20 epochs, which improved to **80.22%** after 40 epochs. The extended training enabled better feature extraction and classification while avoiding overfitting, as indicated by consistent improvements in test accuracy.
- **ResNet-Inspired Network:** Utilizing residual connections to enhance gradient flow, the adapted ResNet achieved a training accuracy of **94%** and a test accuracy of **90%** over 20 epochs. Increasing the batch size to 10 further stabilized training and slightly enhanced generalization, demonstrating the advantages of efficient residual learning and batch size optimization.
- **Custom Neural Network:** The personalized CNN demonstrated strong generalization, achieving a test accuracy of **92.29%** over 20 epochs with a validation accuracy plateauing around **92%**. Extended training to 30 epochs highlighted the potential for overfitting, suggesting further fine-tuning of regularization and hyperparameters could improve stability.

Overall, the experiments underscore the importance of architectural design, hyperparameter tuning, and training strategies in achieving robust performance on image datasets.

# 4 Annex 1: Code used

## 4.1 AlexNet Modified Net

```
import os
import numpy as np
import pickle
import matplotlib.pyplot as plt

from keras.utils import to_categorical
from keras.callbacks import ReduceLROnPlateau, ModelCheckpoint,
    EarlyStopping
from keras.optimizers import Adadelta, Adam, SGD
from keras.layers import Input, Conv2D, Dense, MaxPooling2D, Dropout,
    Flatten, AveragePooling2D, Conv2DTranspose, UpSampling2D,
    BatchNormalization, Add, Input, GlobalAveragePooling2D, Activation,
    Concatenate
from keras.models import Sequential
from tensorflow.keras.models import Model
from keras.losses import categorical_crossentropy
from tensorflow.keras.preprocessing.image import ImageDataGenerator

data = np.load('./MNIST_CorrNoise.npz')

x_train = data['x_train']
y_train = data['y_train']

num_cls = len(np.unique(y_train))
print('Number of classes: ' + str(num_cls))
print('Example of handwritten digit with correlated noise: \n')

k = 3000
plt.imshow(np.squeeze(x_train[k,:,:]))
plt.show()
print('Class: '+str(y_train[k])+'\n')

# RESHAPE and standarize
x_train = np.expand_dims(x_train/255,axis=3)

# convert class vectors to binary class matrices
y_train = to_categorical(y_train, num_cls)

print('Shape of x_train: '+str(x_train.shape))
print('Shape of y_train: '+str(y_train.shape))

model_name='CNN_1_20ep' # To compare models, you can give them different
     names

pweight='./weights/weights_' + model_name  + '.keras'

if not os.path.exists('./weights'):
  os.mkdir('./weights')

## EXPLORE VALUES AND FIND A GOOD SET
b_size = 3 # batch size
val_split = 0.2 # percentage of samples used for validation (e.g. 0.5)
```

```
ep = 20 # number of epochs

input_shape = x_train.shape[1:4] #(28,28,1)
model = Sequential()


model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape
    =(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(128, kernel_size=(1, 1), activation='relu'))
model.add(Conv2D(128, kernel_size=(1, 1), activation='relu'))
model.add(Conv2D(128, kernel_size=(1, 1), activation='relu'))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_cls, activation='softmax'))

model.compile(loss = categorical_crossentropy,
              optimizer=Adadelta(),
              metrics=['accuracy'])

model.summary()

checkpointer = ModelCheckpoint(filepath=pweight, verbose=1,
    save_best_only=True)
callbacks_list = [checkpointer] # explore adding other callbacks such as
     ReduceLROnPlateau, EarlyStopping

history=model.fit(x_train, y_train,
                        epochs=ep,
                        batch_size=b_size,
                        verbose=1,
                        shuffle=True,
                        validation_split = val_split,
                        callbacks=callbacks_list)

print('CNN_1 weights saved in ' + pweight)

# Plot loss vs epochs
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()

# Plot accuracy vs epochs
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()

from keras.models import load_model

## LOAD DATA
data = np.load('./MNIST_CorrNoise.npz')

x_test = data['x_test']
y_test = data['y_test']

num_cls = len(np.unique(y_test))
print('Number of classes: ' + str(num_cls))

# RESHAPE and standarize
x_test = np.expand_dims(x_test/255,axis=3)

print('Shape of x_train: '+str(x_test.shape)+'\n')

## Define model parameters
model_name='CNN_1' # To compare models, you can give them different
    names
pweight='./weights/weights_' + model_name  + '.keras'

model = load_model(pweight)
y_pred = model.predict(x_test)
# Convert predicted probabilities to class indices
y_pred_classes = np.argmax(y_pred, axis=1)

# Ensure y_test is also in class indices (not one-hot encoded)
if len(y_test.shape) > 1:  # If y_test is one-hot encoded
    y_test = np.argmax(y_test, axis=1)

Acc_pred = sum(y_pred_classes == y_test) / len(y_test)
#Acc_pred = sum(y_pred == y_test)/len(y_test)

print('Accuracy in test set is: '+str(Acc_pred))
```

## 4.2 ResNet Modified Net

```

def residual_block(x, filters):
    shortcut = Conv2D(filters, (1, 1), padding='same')(x)  # Match the
    number of filters
    x = Conv2D(filters, (3, 3), padding='same', activation='relu')(x)
    x = Conv2D(filters, (3, 3), padding='same')(x)
    x = Add()([shortcut, x])
    return x

    input_shape = x_train.shape[1:4]  # (28, 28, 1)

input_layer = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
x = residual_block(x, 32)  # First residual block
x = MaxPooling2D(pool_size=(2, 2))(x)
x = residual_block(x, 64)  # Second residual block
x = GlobalAveragePooling2D()(x)
```

```
output_layer = Dense(num_cls, activation='softmax')(x)

model = Model(input_layer, output_layer)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()

checkpointer = ModelCheckpoint(filepath=pweight, verbose=1, save_best_only=True)
callbacks_list = [checkpointer]
```

### 4.3 Personal CNN

```

def residual_block(x, filters):
    shortcut = Conv2D(filters, (1, 1), padding='same')(x)  # Align number of filters
    x = Conv2D(filters, (3, 3), padding='same', activation='relu')(x)
    x = BatchNormalization()(x)
    x = Conv2D(filters, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Add()([shortcut, x]) # Add shortcut to the residual path
    x = Activation('relu')(x)
    return x

# Input Layer
input_layer = Input(shape=(28, 28, 1))

# Convolutional Layers with Residual Blocks
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
x = residual_block(x, 32)  # First residual block
x = MaxPooling2D(pool_size=(2, 2))(x)
x = residual_block(x, 64)  # Second residual block
x = GlobalAveragePooling2D()(x)

# Output Layer
output_layer = Dense(num_cls, activation='softmax')(x)

# Define and Compile Model
model = Model(input_layer, output_layer)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Summary and Training
model.summary()
```