PW4 - Deep Neural Networks

ARN

Jarod Streckeisen, Timothée Van Hove

May 8, 2024

Contents

Introductive questions

Question: *What is the learning algorithm being used to optimize the weights of the neural networks, and what are the parameters (arguments) being used by that algorithm?*

The algorithm used to optimize the weight is RMSprop (Root Mean Square Propagation). We can find the arguments from the Keras documentation:

learning_rate: A float, a keras.optimizers.schedules.LearningRateSchedule instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.001.

rho: float, defaults to 0.9. Discounting factor for the old gradients.

momentum : float, defaults to 0.0. If not 0.0., the optimizer tracks the momentum value, with a decay rate equals to 1 - momentum.

epsilon: A small constant for numerical stability. Defaults to 1e-7.

centered: Boolean. If True, gradients are normalized by the estimated variance of the gradient; if False, by the uncentered second moment. Setting this to True may help with training, but is slightly more expensive in terms of computation and memory. Defaults to False.

name: String. The name to use for momentum accumulator weights created by the optimizer.

weight_decay: Float. If set, weight decay is applied.

clipnorm: Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.

clipvalue: Float. If set, the gradient of each weight is clipped to be no higher than this value.

global_clipnorm: Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.

use_ema: Boolean, defaults to False. If True, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.

ema_momentum: Float, defaults to 0.99. Only used if use_ema=True. This is the momentum to use when computing the EMA of the model's weights: new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value.

ema_overwrite_frequency: Int or None, defaults to None. Only used if use_ema=True. Every ema_overwrite_frequency steps of iterations, we overwrite the model variable by its moving average. If None, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling optimizer.finalize_variable_values() (which updates the model variables in-place). When using the built-in fit() training loop, this happens automatically after the last epoch, and you don't need to do anything.

loss_scale_factor: Float or None. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, keras.optimizers.LossScaleOptimizer will automatically set a loss scale factor.

gradient_accumulation_steps: Int or None. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every gradient_accumulation_steps steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Question: *What loss function is being used ? Please, give the equation(s)*

The categorical crossentropy. This loss for a single sample with true label in one-hot encoded form and the predicted probabilities can be defined as follows:

$$L = -\sum_{c=1}^{N} y_{o,c} \log(p_{o,c})$$

Where:

- `L` is the loss for a single example.
- `N` is the number of classes.
- `y o,c` is a binary indicator (0 or 1) if class label `c` is the correct classification for observation `o`
- `p o,c` is the predicted probability of observation `o` being of class `c`.

Mean Categorical Crossentropy Loss: To calculate the loss over a batch of data, the mean loss across all observations `N` is used:

$$L_{\text{mean}} = -\frac{1}{N} \sum_{o=1}^{N} \sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Where:

- `L mean` is the mean loss over a batch of data.
- `N` is the number of observations in the batch.
- `M` is the number of classes.
- `y o,c` is a binary indicator (0 or 1) if class label `c` is the correct classification for observation `o`.
- `p o,c` is the predicted probability of observation `o` being of class `c`.

This form aggregates the losses for each class prediction into a single scalar value, which represents the average loss per observation in the batch.

Experiment 1: Digit recognition from raw data

This experiment aims to develop and evaluate a `shallow neural network model` to classify handwritten digits from the `MNIST` dataset.

1. Model tuning

Those parameters have been tuned to enhance the model performance:

- Activation function sigmoid -> ReLu
- Number of epoch 3 -> 20
- Optimizer RMSProp -> Adam

Adding two layers of 6 neurons was a good compromise, as more neurons didn't improve accuracy. We also tried more than 2 layers, but we didn't have better results. We didn't feel the need to use dropout regularization, as the model didn't show any signs of overfitting.

2. Model Architecture

- **Input Layer:** Receives flattened 784-dimensional vectors (28x28 pixels per image).
- **First Hidden Layer:** 2 neurons, uses ReLU activation function. This layer is relatively small, testing the capability of minimal neurons to capture essential features.
- **Second and Third Hidden Layers:** Each with 6 neurons and ReLU activation, aimed at gradually processing features extracted by the preceding layer.
- **Output Layer:** 10 neurons (one for each digit) with a softmax activation function to output probabilities for each class.

The model uses a sequential layout in Keras, facilitating a straightforward stack of layers with direct forward connections.

3. Weights

Input : 1570

First layer : 6 * 2 + 6 bias Second layer : 6 * 6 + 6 bias Output : 6 * 10 + 10 bias

Total bias : 22 bias

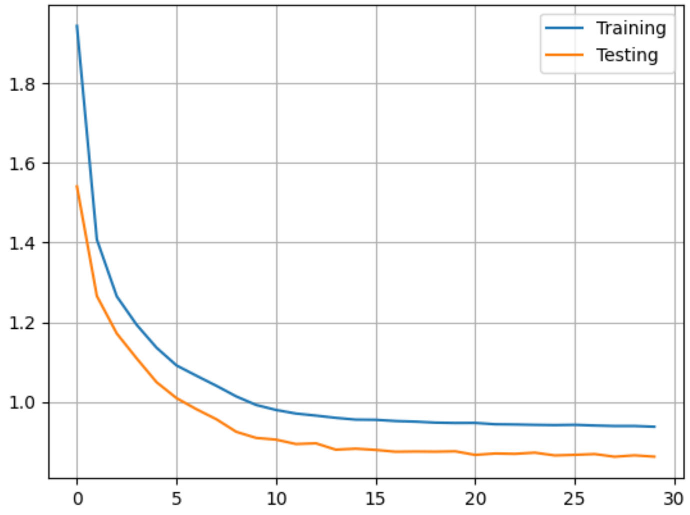
Total number of weights (incl. bias) : 1700

4. Model Summary

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	1570
dense_1 (Dense)	(None, 6)	18
dense_2 (Dense)	(None, 6)	42
dense_3 (Dense)	(None, 10)	70

5. Training performance

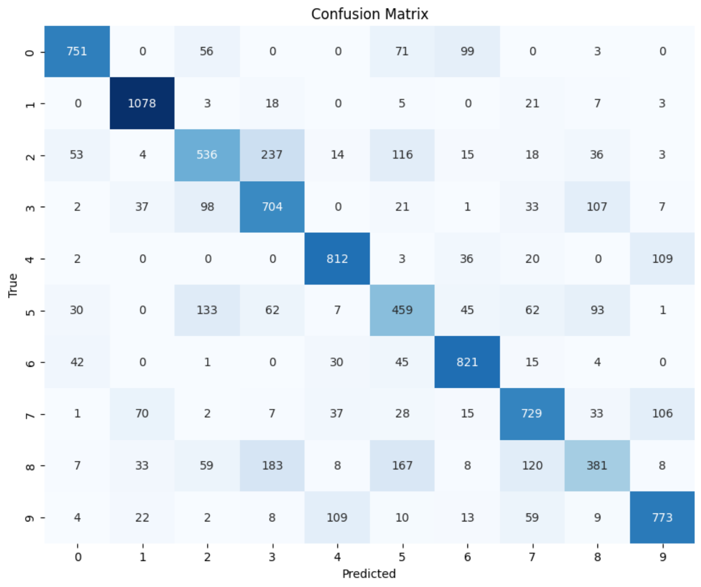
The shallow neural network was trained for 30 epochs with a batch size of 128. The training process shows an initial rapid decrease in loss. The last epochs shows slower improvement, indicating diminishing returns from additional training.



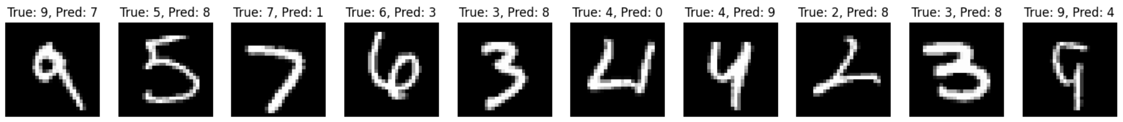
- **Final Training Loss:** ~ 0.905
- **Final Training Accuracy:** ~ 0.720

These scores suggest that the model struggles to capture the complexity of the data, likely due to the shallow architecture.

The confusion matrix and the mismatch sample give insight about the classification performance:



Mismatch sample:



Accuracy of every digits:

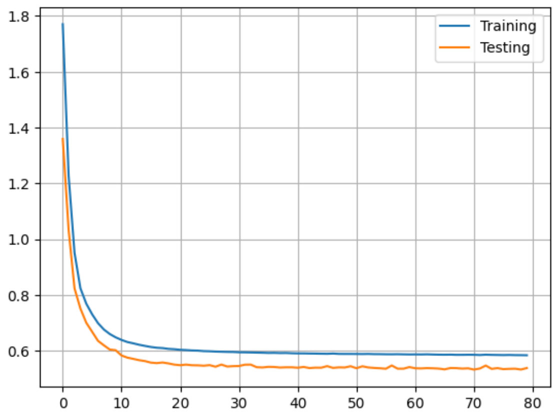
```
Digit Accuracies from Most to Least Accurate:
Digit 1: 94.98%
Digit 6: 85.70%
Digit 4: 82.69%
Digit 0: 76.63%
Digit 9: 76.61%
Digit 7: 70.91%
Digit 3: 69.70%
Digit 2: 51.94%
Digit 5: 51.46%
Digit 8: 39.12%
```

Experiment 2: Digit recognition from features of the input data

1. Enhanced model pix_p_cell = 4

- Add hidden layer of 8 neurons
- RMSProp -> ADAM
- 3 epoch -> 80 epoch
- pix_p_cell = 4
- n_orientations = 8
- batch size = 128

Training history

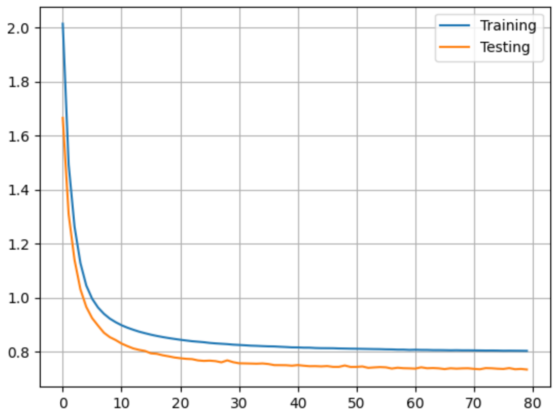


```
Test score: ~0.563
Test accuracy: ~0.814
```

2. Enhanced model pix_p_cell = 7

- Add hidden layer of 8 neurons
- RMSProp -> ADAM
- 3 epoch -> 80 epoch
- pix_p_cell = 7
- n_orientations = 8
- batch size = 128

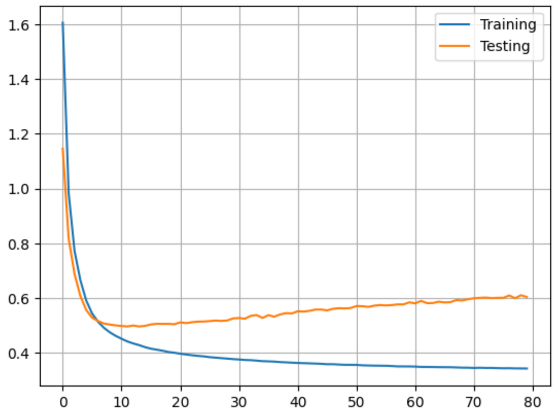
Training history



```
Test score: ~0.774
Test accuracy: ~0.720
```

3. Changing n_orientations

Too much n_orientations (i.e 32) reduce model accuracy



4. Best Model: pix_p_cell = 2

Those parameters have been tuned to enhance the model performance:

- Add hidden layer of 8 neurons
- RMSProp -> ADAM
- 3 epoch -> 80 epoch
- pix_p_cell = 2
- n_orientations = 8
- batch size = 128

Architecture

The 2 pix_p_cell model architecture consists of a sequence of densely connected layers :

- **Input Layer:** This layer accepts a flat input with `hog_size` features. Here, `hog_size` is determined by the image size, number of orientations (`n_orientations`), and the number of pixels per cell (`pix_p_cell`). For this experiment, the input layer receives HOG-transformed data rather than raw pixel values, which aims to improve the model ability to recognize patterns related to the shapes and edges of digits.
- **Hidden Layers:**
  - The first dense layer consists of 2 neurons.
  - The second layer has 8 neurons, potentially capturing more complex relationships in the input data after being initially processed by the first layer.
- **Output Layer:** A dense layer with 10 neurons, corresponding to the 10 possible classes (digits 0-9). This layer uses a `softmax` activation function to output a probability distribution over the 10 classes.

Weights

To get the number of weight in the first layer, we have to calculate the hog size:

$$\text{hog\_size} = \left(\frac{\text{height}}{\text{pix\_p\_cell}} \times \frac{\text{width}}{\text{pix\_p\_cell}}\right) \times \text{n\_orientations} = \left(\frac{28}{2} \times \frac{28}{2}\right) \times 8 = 14 \times 14 \times 8 = 1568$$

$$\text{input\_layer} = \text{hog\_size} \times \text{neurons} + \text{biases} = 1568 \times 2 + 2 = 3138$$

The second layer has 8 neurons. Because the previous layer has 2, we have 2 × 8 + 8 = 24 weights

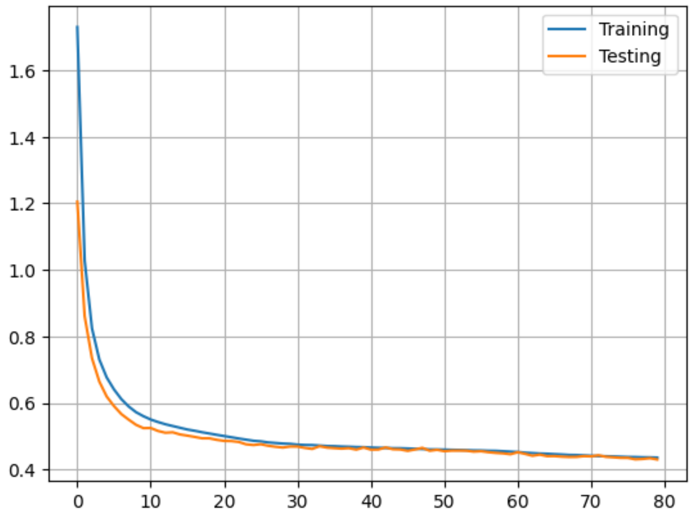
The output layer has 10 neurons. Because the previous layer has 8 neurons, we have 8 × 10 + 10 = 90 weights

The total number of weights is 3138 + 24 + 90 = 3252.

Model Summary

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	3138
dense_1 (Dense)	(None, 8)	24
dense_2 (Dense)	(None, 10)	90

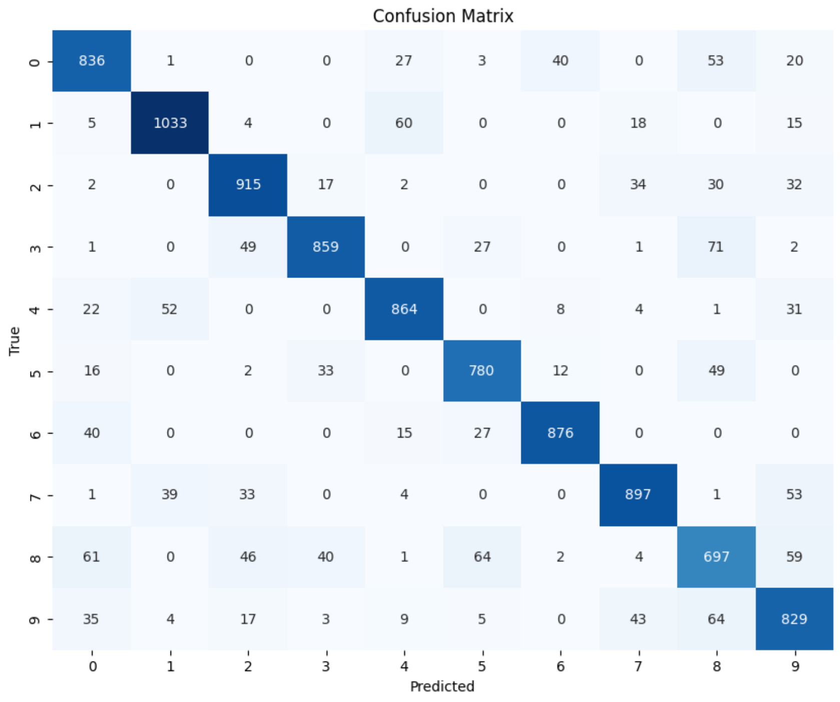
Training performance



```
Test score: ~0.449
Test accuracy: ~0.858
```

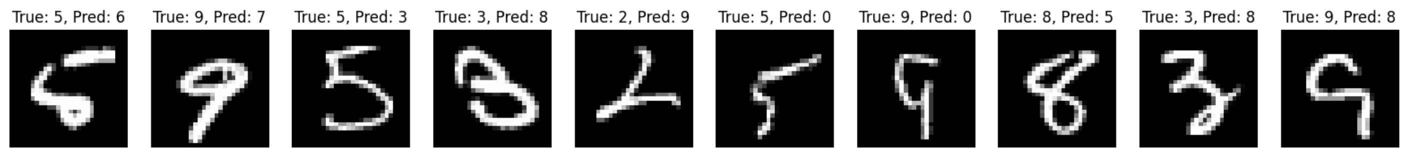
The training loss started relatively high but decreased consistently, indicating that the model was effectively learning and optimizing its weights. The validation accuracy improved, reflecting that adjustments to the model were beneficial.

The confusion matrix and the mismatch sample give insight about the classification performance:



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	836	1	0	0	27	3	40	0	53	20
1	5	1033	4	0	60	0	0	18	0	15
2	2	0	915	17	2	0	0	34	30	32
3	1	0	49	859	0	27	0	1	71	2
4	22	52	0	0	864	0	8	4	1	31
5	16	0	2	33	0	780	12	0	49	0
6	40	0	0	0	15	27	876	0	0	0
7	1	39	33	0	4	0	0	897	1	53
8	61	0	46	40	1	64	2	4	697	59
9	35	4	17	3	9	5	0	43	64	829

Mismatch sample:



Accuracy of every digits:

```
Digit Accuracies from Most to Least Accurate:
Digit 6: 91.44%
Digit 1: 91.01%
Digit 2: 88.66%
Digit 4: 87.98%
Digit 5: 87.44%
Digit 7: 87.26%
Digit 0: 85.31%
Digit 3: 85.05%
Digit 9: 82.16%
Digit 8: 71.56%
```

Experiment 3: CNN

The goal of this experiment is to train a CNN using the `MNIST` dataset. We are going to try to identify a good neural network configuration that automatically extracts and learns the optimal features necessary for accurate digit recognition. The best performing model will then be analyzed.

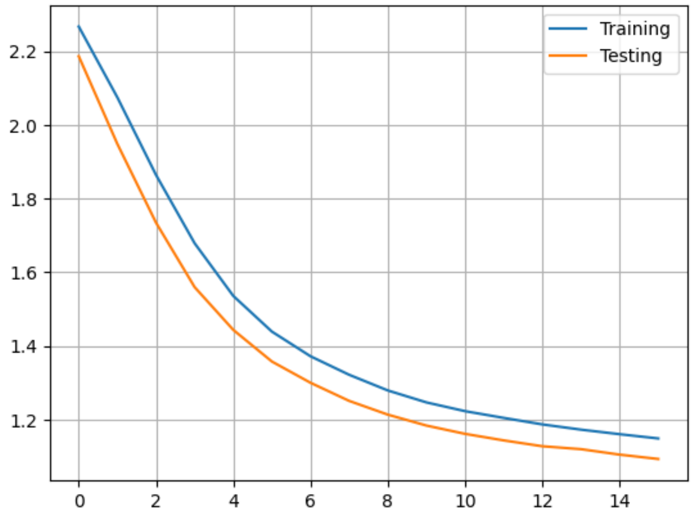
1. Modifying training duration: increasing epochs

One of the first observations from the given training results is the low number of training epochs. Originally set to 3, this low count result in undertraining. To fix this, we increased the number of epochs to 16. This gives the model more iterations over the training data, giving it a chance to learn and refine its weights and bias.

Impact on Performance:

- **Initial Model Accuracy**: 46%
- **Improved Model Accuracy**: 59%

The graph below shows a more stable convergence and a reduction in loss, that indicates an better learning process.



2. Increasing the number of neurons in the fully connected (dense) layer

An good way to increase the model capacity to learn complex patterns is to increase the number of neurons in the fully connected layers. Initially, our model used only 2 neurons in the fully connected layer, which was a limiting factor for the classification.

Changes made to the dense layer of our CNN model:

```
l0 = Input(shape=(height, width, 1), name='l0')

l1 = Conv2D(2, (2, 2), padding='same', activation='relu', name='l1')(l0)
l1_mp = MaxPooling2D(pool_size=(2, 2), name='l1_mp')(l1)
l2 = Conv2D(2, (2, 2), padding='same', activation='relu', name='l2')(l1_mp)
l2_mp = MaxPooling2D(pool_size=(2, 2), name='l2_mp')(l2)
l3 = Conv2D(2, (2, 2), padding='same', activation='relu', name='l3')(l2_mp)
l3_mp = MaxPooling2D(pool_size=(2, 2), name='l3_mp')(l3)
flat = Flatten(name='flat')(l3_mp)
l4 = Dense(200, activation='relu', name='l4')(flat)
l5 = Dense(n_classes, activation='softmax', name='l5')(l4)

model = Model(inputs=l0, outputs=l5)
model.summary()
```

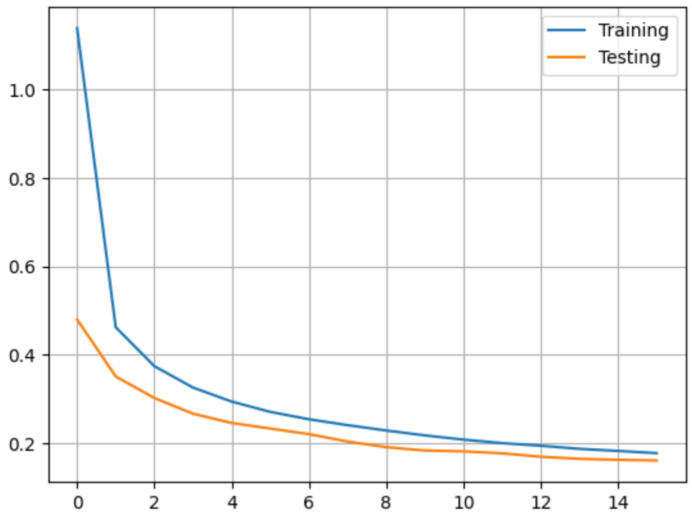
Model summary:

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 2)	10
l1_mp (MaxPooling2D)	(None, 14, 14, 2)	0
l2 (Conv2D)	(None, 14, 14, 2)	18
l2_mp (MaxPooling2D)	(None, 7, 7, 2)	0
l3 (Conv2D)	(None, 7, 7, 2)	18
l3_mp (MaxPooling2D)	(None, 3, 3, 2)	0
flat (Flatten)	(None, 18)	0
l4 (Dense)	(None, 200)	3800
l5 (Dense)	(None, 10)	2010

Results:

```
Test score: ~0.165
Test accuracy: ~0.927
```

The following graph shows a stable convergence that indicate an effective learning process:



3. Simplifying the network by removing the third convolutional layer

In our previous model configuration, we observed that the output of the third convolutional layer resulted in a very small dimension (3x3 pixels). This small dimension can restrict the network from learning and keeping hierarchies in the image data, which can lead to problems during classification.

We decided to remove the third convolutional layer, because we though that fewer layers with more significant dimensions could enhance learning and generalization :

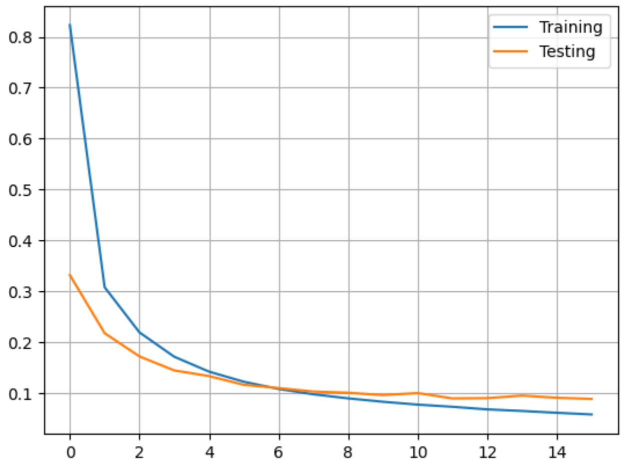
```
l0 = Input(shape=(height, width, 1), name='l0')

l1 = Conv2D(2, (2, 2), padding='same', activation='relu', name='l1')(l0)
l1_mp = MaxPooling2D(pool_size=(2, 2), name='l1_mp')(l1)
l2 = Conv2D(2, (2, 2), padding='same', activation='relu', name='l2')(l1_mp)
l2_mp = MaxPooling2D(pool_size=(2, 2), name='l2_mp')(l2)
flat = Flatten(name='flat')(l2_mp)
l4 = Dense(200, activation='relu', name='l4')(flat)
l5 = Dense(n_classes, activation='softmax', name='l5')(l4)

model = Model(inputs=l0, outputs=l5)
model.summary()
```

Model summary:

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 2)	10
l1_mp (MaxPooling2D)	(None, 14, 14, 2)	0
l2 (Conv2D)	(None, 14, 14, 2)	18
l2_mp (MaxPooling2D)	(None, 7, 7, 2)	0
flat (Flatten)	(None, 98)	0
l4 (Dense)	(None, 200)	19800
l5 (Dense)	(None, 10)	2010

Impact on Performance

This change reduced the complexity of the model but also its ability to generalize as showed by the higher test accuracy. However, the training and validation loss curves indicate potential overfitting. The graph below shows that the validation loss stabilize around 0.1 while the training loss continues to decline. This suggest that its performance on new, unseen data might not improve further without adjustments to prevent overfitting.



```
Test score: ~0.082
Test accuracy: ~0.974
```

4. Increasing convolution layers and adjusting kernel size

To enhance our model's capability to capture more features from the `MNIST` dataset, we decided to increase the number of filters in the convolution layers and adjust the kernel size. The first convolution layer's filter count was increased to 8, and the second to 16, both using larger 3x3 kernels instead of the previous 2x2.

```
l0 = Input(shape=(height, width, 1), name='l0')

l1 = Conv2D(8, (3, 3), padding='same', activation='relu', name='l1')(l0)
l1_mp = MaxPooling2D(pool_size=(2, 2), name='l1_mp')(l1)
l2 = Conv2D(16, (3, 3), padding='same', activation='relu', name='l2')(l1_mp)
l2_mp = MaxPooling2D(pool_size=(2, 2), name='l2_mp')(l2)
flat = Flatten(name='flat')(l2)
l4 = Dense(200, activation='relu', name='l4')(flat)
l5 = Dense(n_classes, activation='softmax', name='l5')(l4)

model = Model(inputs=l0, outputs=l5)
model.summary()
```

Model summary:

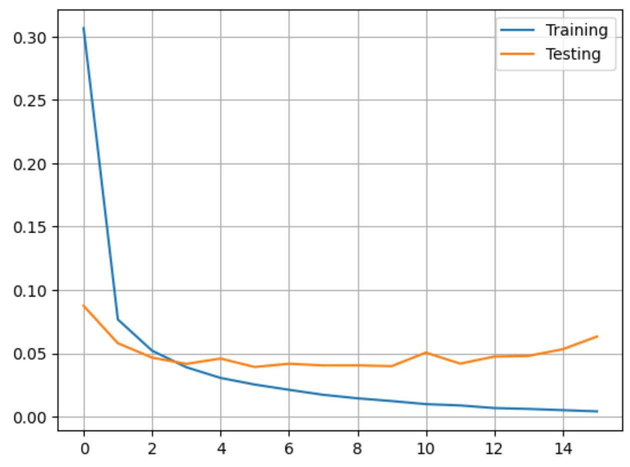
Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 8)	80
l1_mp (MaxPooling2D)	(None, 14, 14, 8)	0
l2 (Conv2D)	(None, 14, 14, 16)	1168
flat (Flatten)	(None, 3136)	0
l4 (Dense)	(None, 200)	627400
l5 (Dense)	(None, 10)	2010

Impact on Performance

Moving from a 2x2 to a 3x3 kernel allows the model to get broader feature representations at each layer, improving its capacity to recognize patterns in the image data. On the other hand, by increasing the number of filters, the network can extract a more diverse set of features from the input images, so it can better differentiate the different digits.

Observations on Overfitting:

The training loss continues to decrease nearly to zero, while the validation loss shows signs of increase. This suggests that while the model has learned the training data exceptionally well, it may not generalize as effectively on new, unseen data.



```
Test score: ~0.050
Test accuracy: ~0.986
```

5. Reducing overfitting with dropout regularization

To mitigate overfitting, we used dropout regularization in our model. This technique is designed to prevent overfitting by randomly "dropping out" a subset of features during the training phase. Drop-out was applied after each convolutional and dense layer before the final classification layer:

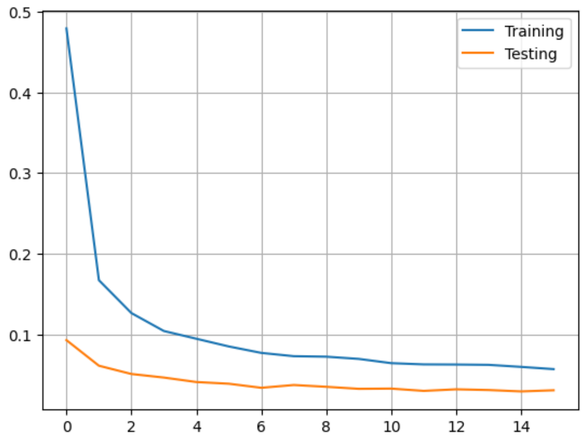
```
l0 = Input(shape=(height, width, 1), name='l0')
l1 = Conv2D(8, (3, 3), padding='same', activation='relu', name='l1')(l0)
l1_mp = MaxPooling2D(pool_size=(2, 2), name='l1_mp')(l1)
l1_drop = Dropout(0.2, name='l1_drop')(l1_mp)
l2 = Conv2D(16, (3, 3), padding='same', activation='relu', name='l2')(l1_drop)
l2_mp = MaxPooling2D(pool_size=(2, 2), name='l2_mp')(l2)
l2_drop = Dropout(0.2, name='l2_drop')(l2_mp)
flat = Flatten(name='flat')(l2_drop)
l4 = Dense(200, activation='relu', name='l4')(flat)
l4_drop = Dropout(0.5, name='l4_drop')(l4)
l5 = Dense(n_classes, activation='softmax', name='l5')(l4_drop)
model = Model(inputs=l0, outputs=l5)
model.summary()
```

Model summary:

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 8)	80
l1_mp (MaxPooling2D)	(None, 14, 14, 8)	0
l1_drop (Dropout)	(None, 14, 14, 8)	0
l2 (Conv2D)	(None, 14, 14, 16)	1168
l2_mp (MaxPooling2D)	(None, 7, 7, 16)	0
l2_drop (Dropout)	(None, 7, 7, 16)	0
flat (Flatten)	(None, 784)	0
l4 (Dense)	(None, 200)	157000
l4_drop (Dropout)	(None, 200)	0
l5 (Dense)	(None, 10)	2010

Impact on Performance

The use of dropout after pooling layers and before the final classification layer makes that the model relies less on any particular set of neurons. By randomly disabling neurons during training, this forces the network to get more robust features. In the following graph, the training curve shows a smooth decline, while the testing curve remains low and stable, converging to the training loss, which signifies an effective learning and generalization across unseen data.



```
Test score: ~0.025
Test accuracy: ~0.991
```

Architecture

- **Input Layer :** l0 receives images of size 28x28 with 1 channel (grayscale images). This is the entry point for data but does not perform any computation.
- **Convolutional Layers :** l1 applies 8 filters of size 3x3 with the ReLU activation function. The output is 28x28x8 (because `same` padding keeps the output height and width equal to the input).
- l2 applies 16 filters of size 3x3 with the ReLU activation function. This layer's output is 14x14x16, also using `same` padding.
- **Pooling Layers :** l1_mp and l2_mp perform max pooling with a 2x2 window, reducing the dimensions by half, thus l1_mp results in 14x14x8, and l2_mp results in 7x7x16.
- **Dropout Layers :** l1_drop**,** l2_drop**, and** l4_drop are used to prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time. Dropout rates are 0.2 after both pooling layers and 0.5 before the final output layer.
- **Dense Layers :** l4 is a fully connected layer with 200 neurons and ReLU activation function. l5 is the final output layer with 10 neurons (one for each digit) using softmax activation function for multi-class classification.

Weights

Convolutional Layer Weights :

- l1: $(3\times3\times1)\times8+8=80(3\times3\times1)\times8+8=$**80** parameters
- l2: $(3\times3\times8)\times16+16=1168(3\times3\times8)\times16+16=$**1168** parameters

Dense Layer Weights :

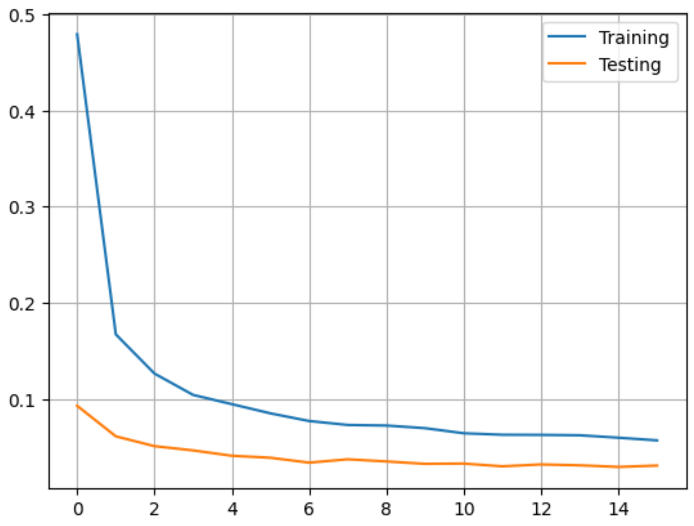
- l4: $784\times200+200=157000784\times200+200=$**157'000** parameters
- l5: $200\times10+10=2010200\times10+10=$**2010** parameters

Total Number of Parameters:

80 + 1168 + 157000 + 2010 = 160'258

Performance discussion

The tuning process showed a good improvement in model accuracy and a decrease in loss, as evidenced by the loss and accuracy curves provided below. The model achieved a final training accuracy of approximately 98.44% and a validation accuracy of 99.10%, proving its effectiveness in recognizing and classifying the digits from the MNIST dataset.



The graph shows a convergence where the validation loss decreases, suggesting that the model is not underfitting or overfitting. The training loss continues to decline slightly, indicating that the model could potentially improve with more epoch, however, the test loss seem to stagnate.

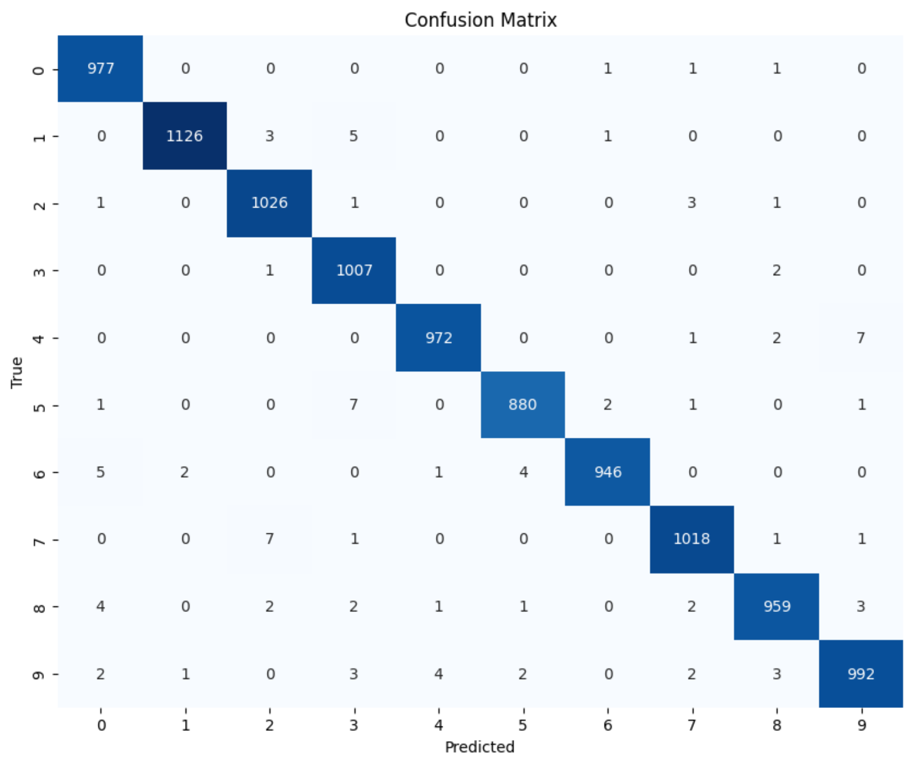
After finishing the training, the model was evaluated on a separate test set:

- **Test Accuracy:** 99.09%
- **Test Loss:** 0.0254

Those numbers show that the model learned the training data effectively and generalized well to new, unseen data. The high test accuracy aligns with the validation accuracy, indicating the good model robustness and capability to perform a reliable classification.

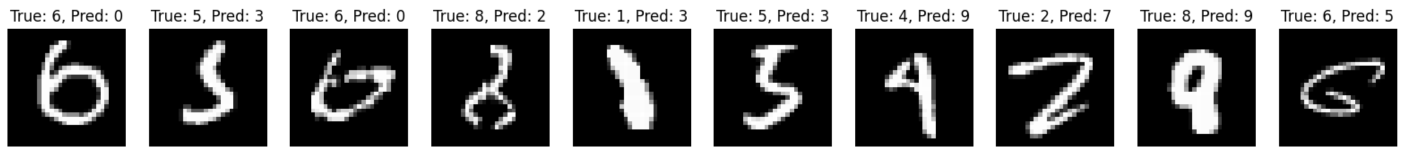
The initial configurations with fewer neurons and layers showed underperformance, likely because the model was less complex and was not able to capture the patterns in the data. The successive tuning, including increasing the number of neurons and introducing dropout layers, helped to achieve better generalization and helped to control overfitting.

The confusion matrix and the mismatch sample give insight about the classification performance:



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	977	0	0	0	0	0	1	1	1	0
1	0	1126	3	5	0	0	1	0	0	0
2	1	0	1026	1	0	0	0	3	1	0
3	0	0	1	1007	0	0	0	0	2	0
4	0	0	0	0	972	0	0	1	2	7
5	1	0	0	7	0	880	2	1	0	1
6	5	2	0	0	1	4	946	0	0	0
7	0	0	7	1	0	0	0	1018	1	1
8	4	0	2	2	1	1	0	2	959	3
9	2	1	0	3	4	2	0	2	3	992

Mismatch sample:



Accuracy of every digits:

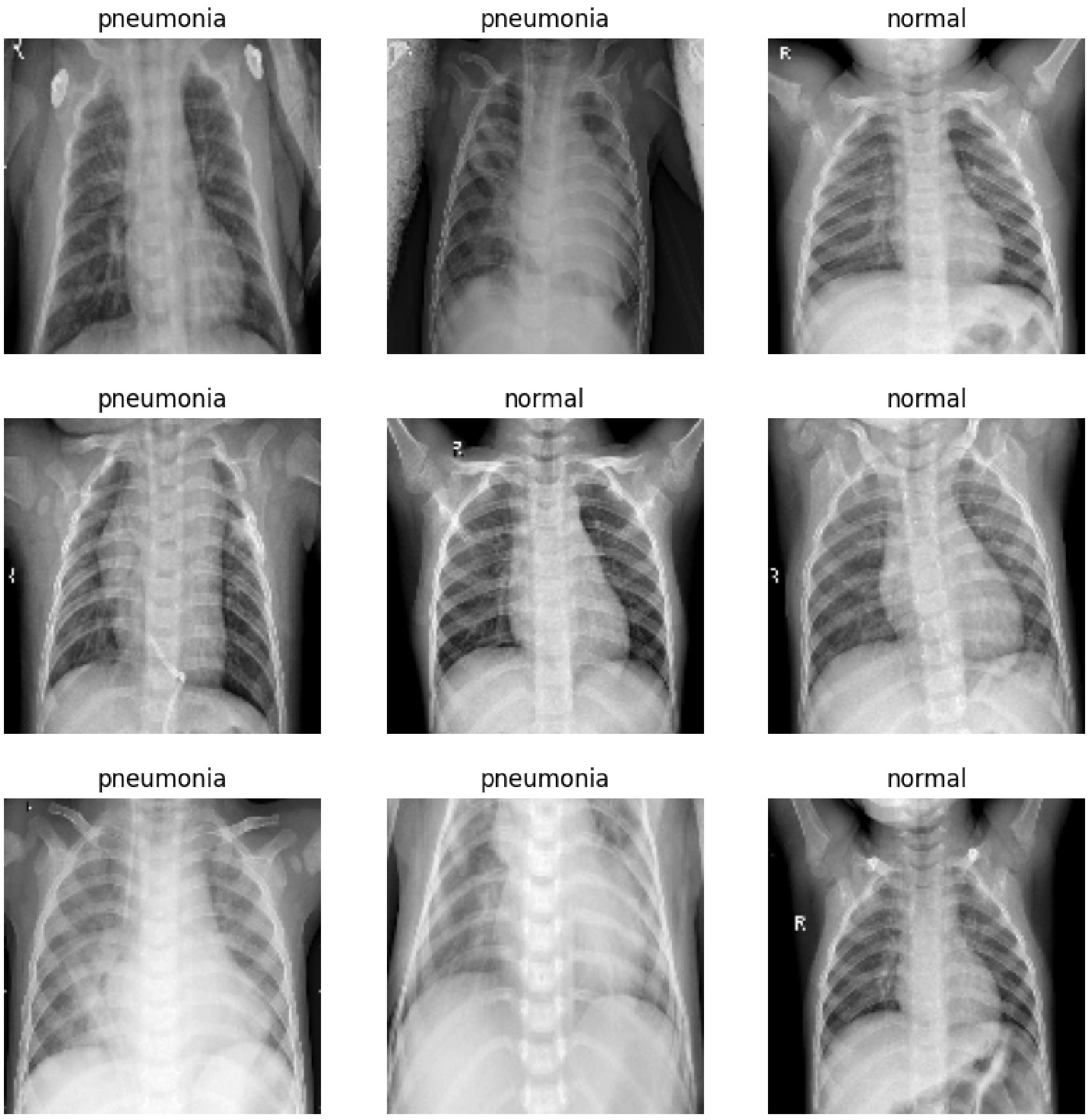
```
Digit Accuracies from Most to Least Accurate:
Digit 1: 99.91%
Digit 0: 99.69%
Digit 2: 99.22%
Digit 3: 99.21%
Digit 5: 99.10%
Digit 7: 98.93%
Digit 4: 98.88%
Digit 8: 98.67%
Digit 6: 98.54%
Digit 9: 98.32%
```

Comparison of the experiments on MNIST dataset

After analyzing those 3 experiments, it is clear that the complexity of the model and the nature of the feature representation are essential to the performance in digit recognition. While shallow networks quickly reach their performance limits, deeper architectures like CNNs with advanced feature engineering (like HOG) or inherent feature extraction capabilities show big improvements in accuracy and loss. However, more complex models require more CPU resources and training time but they yield much better results. The experiment with the CNN showed the power of layer depth, neuron count, and dropout in resulting in high accuracy and making it the best performer.

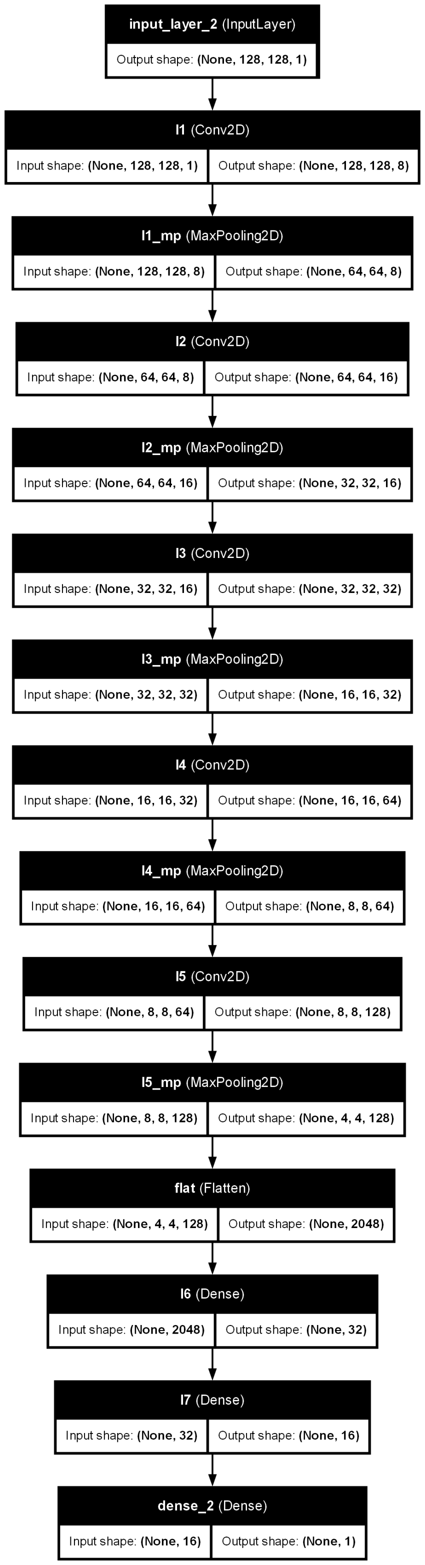
Experiment 4: Pneumonia Detection Model Performance

The goal of this experiment is to evaluate the performance of a CNN designed to classify chest X-ray images into two categories: normal and pneumonia. The dataset that we used contains 5216 images for training, 16 images for validation and 624 images for testing.



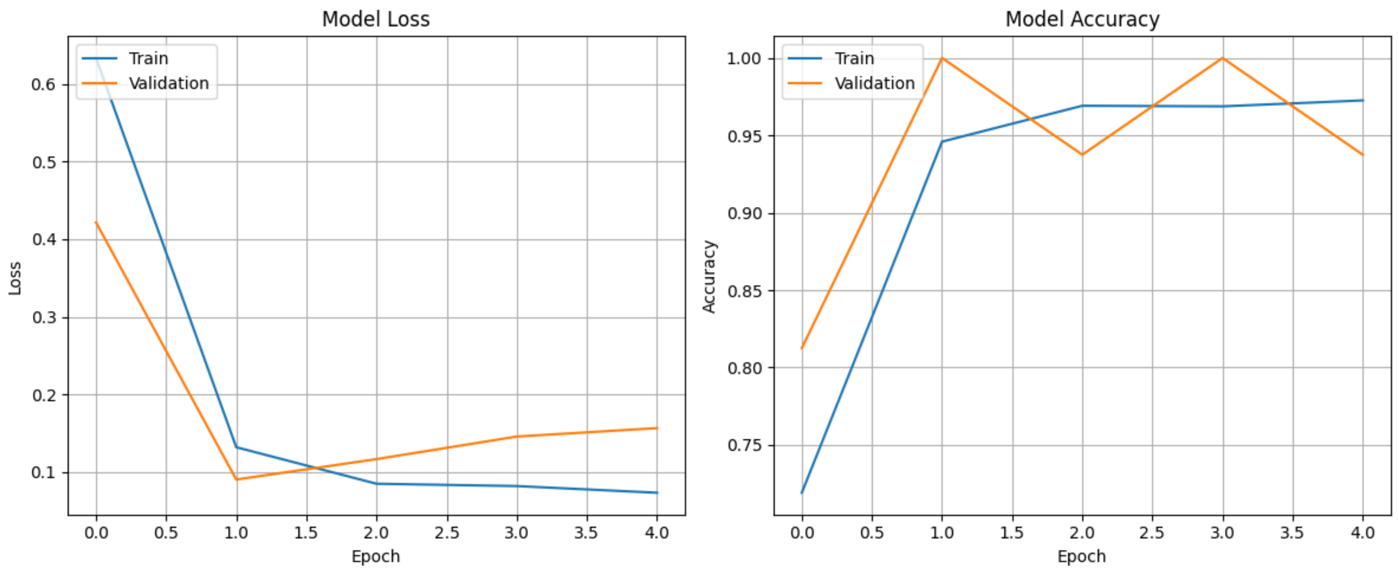
1. Model Architecture

The CNN model consists of 5 convolutional layers, each one followed by max pooling layers, and it follows by 2 fully connected (dense) layers. The convolutional layers are designed to extract spatial hierarchies of features from the images, while the dense layers classify these features into the two categories.


input_layer_2 (InputLayer)
Output shape: (None, 128, 128, 1)
l1 (Conv2D)
Input shape: (None, 128, 128, 1)
Output shape: (None, 128, 128, 8)
l1_mp (MaxPooling2D)
Input shape: (None, 128, 128, 8)
Output shape: (None, 64, 64, 8)
l2 (Conv2D)
Input shape: (None, 64, 64, 8)
Output shape: (None, 64, 64, 16)
l2_mp (MaxPooling2D)
Input shape: (None, 64, 64, 16)
Output shape: (None, 32, 32, 16)
l3 (Conv2D)
Input shape: (None, 32, 32, 16)
Output shape: (None, 32, 32, 32)
l3_mp (MaxPooling2D)
Input shape: (None, 32, 32, 32)
Output shape: (None, 16, 16, 32)
l4 (Conv2D)
Input shape: (None, 16, 16, 32)
Output shape: (None, 16, 16, 64)
l4_mp (MaxPooling2D)
Input shape: (None, 16, 16, 64)
Output shape: (None, 8, 8, 64)
l5 (Conv2D)
Input shape: (None, 8, 8, 64)
Output shape: (None, 8, 8, 128)
l5_mp (MaxPooling2D)
Input shape: (None, 8, 8, 128)
Output shape: (None, 4, 4, 128)
flat (Flatten)
Input shape: (None, 4, 4, 128)
Output shape: (None, 2048)
l6 (Dense)
Input shape: (None, 2048)
Output shape: (None, 32)
l7 (Dense)
Input shape: (None, 32)
Output shape: (None, 16)
dense_2 (Dense)
Input shape: (None, 16)
Output shape: (None, 1)

2. Training Process

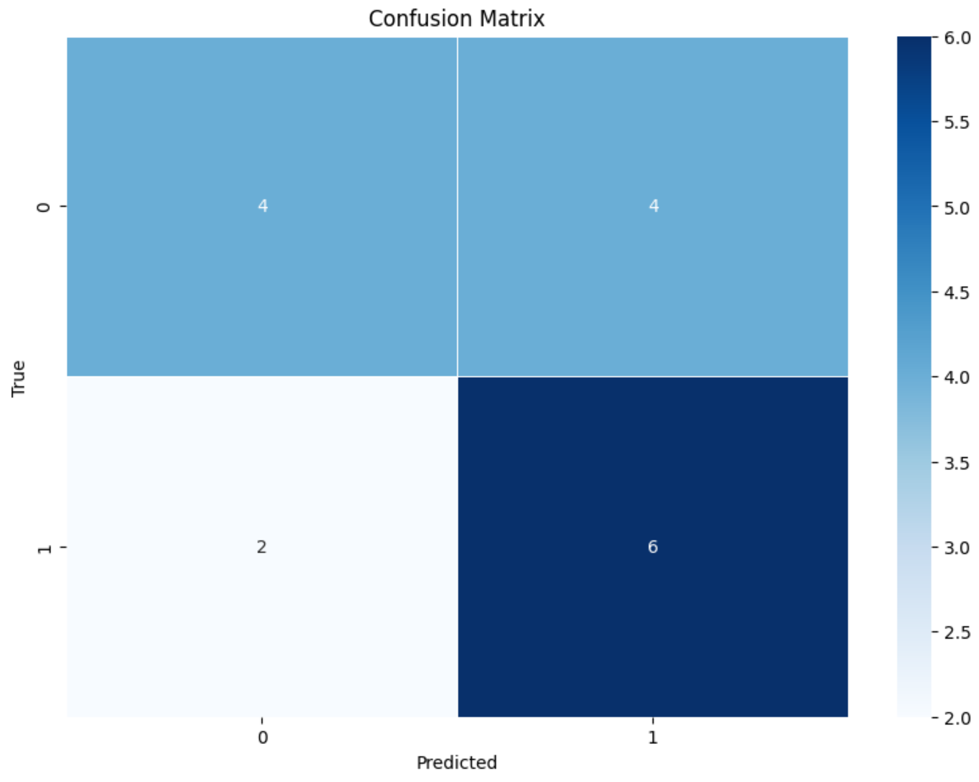
The model was trained over 5 epochs with Adam optimizer, Binary Crossentropy loss function and class weight are applied due to imbalance in the dataset, with more weight given to the minority class.



3. Validation Results

- Accuracy: 62.5%
- F1 Score: 66.7%

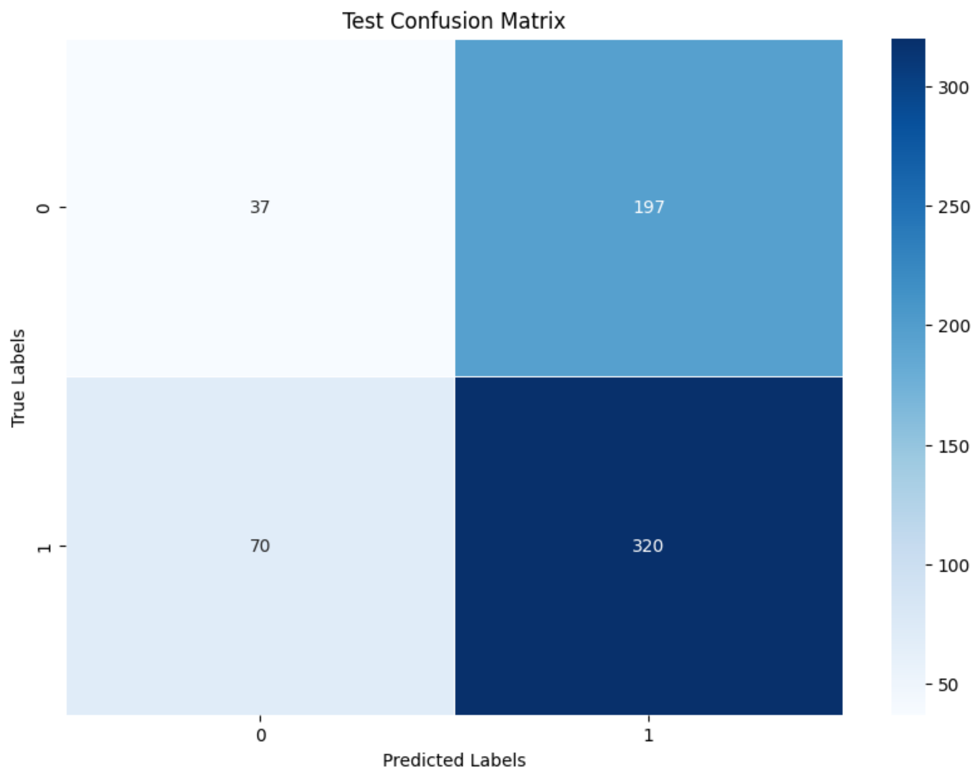
As we can see in the training output above, the model achieved full accuracy in predicting the validation set in earlier epochs, which dropped in later epochs. This variation might be due to overfitting, given the small size of the validation set.



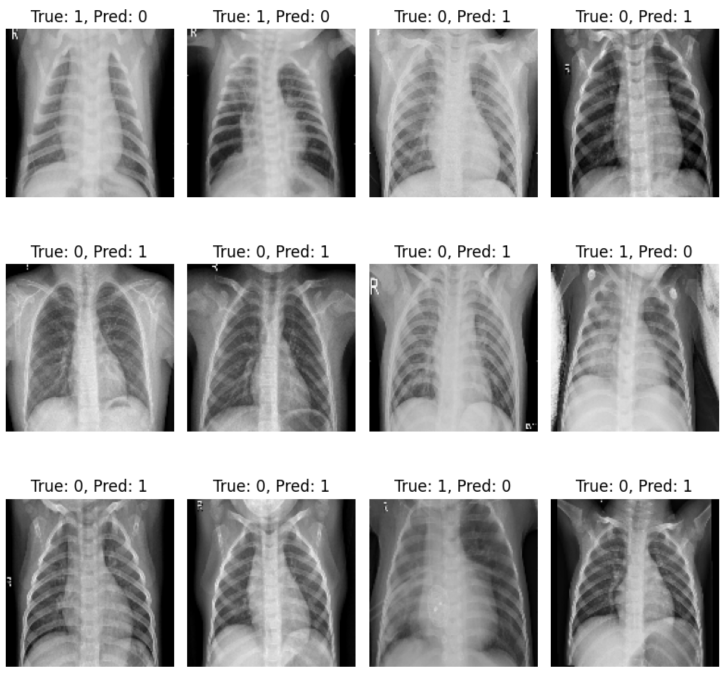
4. Test Results

- Accuracy: 57.2%
- F1 Score: 70.6%

The accuracy on the test set was lower compared to the validation set, which could indicate that the model ability to generalize needs improvement. The higher F1 score compared to accuracy could indicate that the model is better at managing the balance between precision and recall.



5. Some misclassified images



6. Discussion

The good performance on the validation set in the first epochs, followed by a perfect score, shows problems in the model ability to generalize, because we don't see the same level of performance on the test set. This difference is probably due to overfitting, caused by the small size of the validation set.

The test results show that while the model has learned to classify, its accuracy is still poor. We could have better results by further tuning the model, specifically with data augmentation to improve its ability to generalize to new, unseen data.

A bigger validation set would provide a more solid basis for tuning and evaluating the model. Implementing more complex augmentation techniques could help the model generalize better from the training data. We could also use dropout regularization to reduce the overfitting. Finally, longer training with early stopping would allow the model to stabilize its performance on the training and validation datasets.

PW4 Questions and conclusion

Deeper vs. Shallower Models

Deeper models (like the current one with multiple convolution and dense layers) generally have more weights due to increased layers and complexity, which allows them to learn more detailed features from the data. However, deeper models also require more computational resources and are more prone to overfitting, which needs strategies like dropout. **Shallower models** might have fewer parameters, making them quicker to train and less prone to overfitting but potentially less capable of learning complex patterns.

Example:

TODO