Lab report – TNM112

Deep learning for media technology, Lab 2

Emil Djurson (emidj236)

Lab partner: Astrid Helzén (asthe501)

Sunday 14th December, 2025 (16:02)

Abstract

This lab provided comprehensive practical insight into the design, implementation and optimization of Convolutional Neural Networks (CNNs). The work began with implementing a custom CNN from a given skeleton code, where the core layers had to be implemented. The result was then verified for numerical equivalence against a Keras reference model. This foundational phase established a pipeline that achieved 92.92% *and an absolute difference of* 7.32×10^{-5}. *The second experiment focused on mitigating challenges posed by data scarcity, successfully demonstrating that combined regularization and data augmentation strategies could prevent overfitting on a minimal 128-image MNIST subset, yielding a robust average test accuracy of* $93.03\% \pm 0.71\%$.

The final experiment addressed complex image classification using the PatchCamelyon dataset. This involved implementing a ResNet inspired architecture and comparing its performance across the Tensorflow and PyTorch frameworks. The optimized model achieved the highest validation accuracy of 82.15% *using the PyTorch framework and* 80.78% *using the TensorFlow framework.*

1 Introduction

In deep learning focused on image analysis and pattern recognition, Convolutional Neural Networks (CNNs) are an essential architecture. This lab aimed to provide deep practical insight into both the manual implementation of CNN core components and the optimization of their performance under varied data conditions. The work is structured as a progression through three distinct challenges.

The initial phase focused on foundational knowledge, requiring the implementation of core network layers such as, convolution, max pooling and dense layers from scratch. With the results then being compared to a Keras baseline to verify numerical correctness. The second phase investigated how robust regularization strategies and data augmentation could be utilized to achieve high classification accuracy despite the use of a limited training dataset. The third and final phase required architectural and optimization advancements, focusing on a more complex problem. tumor classification from the PatchCamelyon dataset.

2 Background

In order to implement the CNN and conduct the experiments, Python was used together with a Jupyter Notebook environment. For the first task the Keras API was used to verify the results of the custom implemented CNN. Later on in task 3, PyTorch was also used in order to compare different frameworks and utilize photometric data augmentation. Task 3 also utilized GPU computation in order to speed up the training process.

3 Method

To systematically investigate the implementation, optimization and advanced architecture design of CNNs, the following tasks were done in sequential order.

3.1 Task 1

The first task was to understand and implement a custom CNN model from the given skeleton code. The results from the custom implemented CNN model was then compared to the results from the Keras CNN.

3.1.1 Activation functions

The CNN was implemented with four different activation functions which are sigmoid, softmax, linear and ReLu. The exact same implementation from the previous lab was used as seen in listing 1.

```python
def activation(x, activation):
    if activation == 'linear':
        return x

    elif activation == 'relu':
        return np.maximum(0, x)

    elif activation == 'sigmoid':
        return 1 / (1 + np.exp(-x))

    elif activation == 'softmax':
        exps = np.exp(x - np.max(x))
        return exps / np.sum(exps)

    else:
        raise Exception("Activation function is not valid",
    activation)
```

Listing 1: Activation functions.

3.1.2 Convolution layer

In order to extract spatial features, a Convolutional layer was implemented. The computation within the layer was defined by equation 1.

$$H_j^{(l)} = \sigma\left(\sum_{i=1}^{C_I} W_{i,j}^{(l)} * H_i^{(l-1)} + b_j^{(l)}\right), \tag{1}$$

Where the output features map $H_j^{(l)}$ is calculated by summing the contributions from all input channels (C_I) to produce a single output channel. j. $W_{i,j}^{(l)}$ is the two dimensional kernel that is convolved over the previous layer's feature map $H_i^{(l-1)}$. The summation $\sum_{i=1}^{C_I}(...)$ sums the filtered information across all input channels. Finally the the kernel specific bias $b_j^{(l)}$ is added before the entire result is passed through the non-linear activation function σ. Listing 2 shows the code implementation of the equation.

```
def conv2d_layer(h, W, b, act):
    CI = h.shape[2] # Number of input channels
    CO = W.shape[3] # Number of output channels

    h_j = np.zeros((h.shape[0], h.shape[1], CO))

    for i in range(CO):
        z = 0

        for j in range(CI):
            kernel = W[:, :, j, i]
            kernel = np.flipud(np.fliplr(kernel))
            convolved = signal.convolve2d(h[:, :, j], kernel, mode=
    'same')
            z += convolved

        h_j[:, :, i] = activation(z + b[i], act)

    return h_j
```

Listing 2: Convolve 2D function.

3.1.3 Max pooling layer

To reduce the number of feature map dimensions and decrease computational complexity a max pooling layer was implemented. The layer reduces the feature map to half of its original height and width.

The code implementation is shown in listing 3, where the function starts by calculating the new height and width using integer division of the original dimensions.

```
def pool2d_layer(h):
    sy, sx = h.shape[0] // 2, h.shape[1] // 2

    ho = np.zeros((sy, sx, h.shape[2]))

    for i in range(h.shape[2]):
        ho[:, :, i] = skimage.measure.block_reduce(h[:, :, i], (2,
    2), np.max)
    return ho
```

Listing 3: Pool 2D layer function.

The core operation uses a 2×2 sliding window with a stride of 2. The sliding window is applied to each input channel of the input feature map, for every position the window covers the maximum value within the 2×2 region is selected as the output.

3.1.4 Flatten layer

The flatten_layer function was implemented to convert the feature maps which are multi-dimensional to a one-dimensional vector, the code implementation is shown in listing 4.

```
def flatten_layer(h):
    return h.flatten()
```

Listing 4: Flatten layer function.

3.1.5 Dense layer

In order to make any classifications with the CNN the dense layer was implemented. It receives the flattened features that have been extracted by the previous convolution and pooling layers, and map them to the final output classes.

The code implementation, shown in listing 5 is based on the standard feedforward matrix operations. The code firstly verifies that h is a one-dimensional vector and reshapes it from a row vector to a column vector. The core transformation is then performed by calculating equation 2.

$$Z = Wh + b \tag{2}$$

The input vector h is multiplied by the weight matrix W and the bias vector b is added. The result Z is then passed through the non-linear activation function.

```
def dense_layer(h, W, b, act):
    h = h.reshape(len(h), 1)

    z = W @ h + b
    h = activation(z, act)
    return h[:,0]
```

Listing 5: Dense layer function.

3.1.6 CNN setup and loss calculation

The code skeleton did not have a calculation for the total number of weights which was simply implemented in the setup_model function, the code implementation is shown in listing 6.

```
self.N = len(W) + len(b)
```

Listing 6: CNN setup function.

Then in order to calculate how well the model performs, the loss was calculated with the Cross-Entropy Loss function, which is the standard measure for multi-classification tasks. The code implementation is shown in listing 7.

```
train_loss = -1 * np.mean(self.dataset.y_train_oh * np.log(
    output_train))
test_loss = -1* np.mean(self.dataset.y_test_oh * np.log(output_test
    ))
```

Listing 7: Evaluate function.

3.2 Task 2

The objective of this task was to implement and train a CNN effectively on a minimal MNIST dataset consisting of only 128 images. In order to counteract the sever data limitation, a data augmentation pipeline was implemented as shown in listing 8.

```
x = layers.Input(shape=data.x_train.shape[1:])
shear = keras.layers.RandomShear(0.1, 0.1)(x)
rot = keras.layers.RandomRotation(0.1)(shear)
zoom = keras.layers.RandomZoom(0.1)(rot)
trans = keras.layers.RandomTranslation(0.1, 0.1)(zoom)
```

Listing 8: Data augmentation.

The core network structure was built from three feature-extracting convolutional blocks, where each block consisted of $N = 2$ convolutional layers, followed by a 2×2 max-pooling operation. The channels progressed from 8 to 16 and finally to 32. After the feature extraction and flattening, the network transitioned into three dense layers consisting of 256, 128, and 64 neurons. Listing 9.

```
conv1  = conv_block(trans, N=2, channels=8, kernel_size=(3,3),
    activation='relu', padding='same')
conv2  = conv_block(conv1, N=2, channels=16, kernel_size=(3,3),
    activation='relu', padding='same')
conv3  = conv_block(conv2, N=2, channels=32, kernel_size=(3,3),
    activation='relu', padding='same')
flat1  = layers.Flatten()(conv3)

dense1 = layers.Dense(256, activation='relu', kernel_regularizer=
    keras.regularizers.L2(l2=0.001))(flat1)
dropd1  = keras.layers.Dropout(0.2)(dense1)
dense2 = layers.Dense(128, activation='relu', kernel_regularizer=
    keras.regularizers.L2(l2=0.001))(dropd1)
dropd2  = keras.layers.Dropout(0.2)(dense2)
dense3 = layers.Dense(64, activation='relu', kernel_regularizer=
    keras.regularizers.L2(l2=0.001))(dropd2)
dropd3  = keras.layers.Dropout(0.2)(dense3)
y = layers.Dense(data.K, activation='softmax')(dropd3)
```

Listing 9: CNN structure

Training was conducted over 350 epochs with a batch size of 16. The final performance of the configuration was determined by extracting the average result from 5 independent runs.

3.3 Task 3

The goal of Task 3 was to implement a CNN for tumor classification from the PatchCamelyon dataset, either from scratch or using the model from Task 2 as a starting point. Development focused on two main areas, data augmentation and the network architecture.

Firstly, the data augmentation pipeline was expanded compared to Task 2 by implementing more geometric data augmentation and experimenting with photometric data augmentation, the code implementation shown in listing 10.

```
data_augmentation = keras.Sequential([
    layers.RandomFlip(mode="horizontal_and_vertical"),
    layers.RandomRotation(factor=0.25),
    layers.RandomTranslation(height_factor=0.1, width_factor=0.1),
    layers.RandomZoom(height_factor=(-0.15, 0.15)),
    layers.RandomShear(x_factor=0.05, y_factor=0.05),
    # layers.RandomColorJitter(brightness_factor=0.1,
    contrast_factor=0.2, saturation_factor=0.2, hue_factor=0.01),

    # layers.RandomFlip(mode="horizontal"),
    # layers.RandomFlip(mode="vertical"),

    # layers.RandomContrast(0.05),
    # layers.RandomBrightness(0.05),
])
```

Listing 10: Geometric and photometric data augmentation.

Secondly, the network architecture was upgraded to a deeper, more powerful model by implementing a ResNet-inspired structure within the Keras functional API. This design utilized residual skip connections to facilitate training and optimization. While the model maintained the same count of three main convolutional blocks as in Task 2, their internal configuration was changed to support deeper learning.

The channel depth was substantially increased across the blocks, progressing from 32 to 64 and finally to 128 channels. Each block was implemented using the same pattern of two convolutional layers with 3×3 kernels and $'same'$ padding, utilizing Glorot Normal kernel initialization, followed by batch normalization and LeakyReLU activations. To establish the skip connection, the block's input was channeled through a 1×1 convolution and then concatenated with the feature map output. Listing 11 shows a residual block with 32 channels and the 1×1 convolution.

```
residual_1 = x
residual_1 = layers.Conv2D(32, (1, 1), padding='same',
    kernel_initializer='glorot_normal')(residual_1)

x = layers.Conv2D(32, (3, 3), padding='same', kernel_initializer='
    glorot_normal')(x)
# ... (BatchNormalization and LeakyReLU)

x = layers.Add()([x, residual_1])
# ... (MaxPooling2D and Dropout)
```

Listing 11: Residual block 1.

The second and third blocks maintained this residual structure shown in listing 11. Following the first two blocks, a max pooling layer was used for downsampling, and a high dropout rate was applied to all three blocks. After the third residual block before the final dense layers a global average pooling layer was implemented to reduce the number of dimensions in the feature map.

Finally, the optimized network structure and augmentation were tested and compared using both the TensorFlow framework and the PyTorch framework, leveraging GPU computation in both environments.

4 Results

This section will present the result from all of the different tasks.

4.1 Task 1

When comparing the custom CNN that was implemented to the Keras model, the process confirmed both the numerical correctness and functional equivalence of the manual implementation. Numerical verification showed extremely high fidelity, showed in table 1.

Abs sum of Keras model	Abs sum of our model	Abs difference
804.32874	804.3285975679755	7.32×10^{-5}

Table 1: Evaluation of custom CNN convolutional layer implementation.

Ultimately, the custom CNN successfully matched the performance of the Keras reference model on the full MNIST dataset, as both achieved an identical test accuracy of 97.92%, as shown in table 2.

Model	Test loss	Test Accuracy (%)
Keras CNN	0.0638	97.92
Custom CNN	0.0027	97.92

Table 2: Performance comparison between Keras CNN and custom implemented CNN.

4.2 Task 2

The data augmentation and structural network changes successfully mitigated overfitting on the limited 128-image MNIST subset, exceeding the required 90% average test accuracy. Over five independent runs, the model achieved an average test accuracy of 93.07% $\pm$ 0.71% as shown in table 3.

Test run	Train loss	Train accuracy (%)	Test loss	Test accuracy (%)
1	0.1379	99.22	0.5540	92.32
2	0.1413	99.22	0.6129	92.43
3	0.1235	100.00	0.4755	94.21
4	0.1214	100.00	0.5045	92.85
5	0.1483	100.00	0.4643	93.55

Table 3: Performance comparison for the 5 runs.

The consistency of the results confirms the effectiveness of the regularization approach. Although the model often reach 100% training accuracy, indicating high fitting to the small training data, the final test accuracy remained hight and stable above 92% across all runs. Figure 1 shows test run 1 and how the model learned generalizable features rather than simply memorizing the limited dataset.

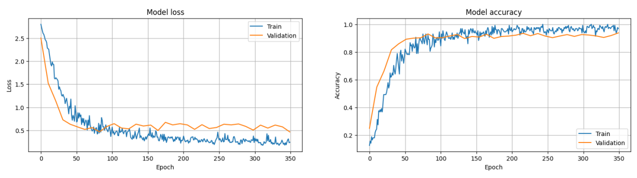


Figure 1: Model performance from test run 1.

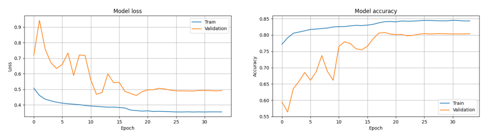
4.3 Task 3

The PyTorch implementation of the model significantly outperformed the TensorFlow implementation in all metrics visible in table 4, achieving the highest validation accuracy of 82.15% and the lowest loss of 0.3949.

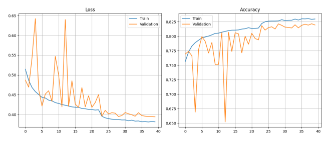
Framework	Train loss	Train acc. (%)	Val. loss	Val. acc. (%)
PyTorch	0.3823	82.93	0.3949	82.15
TensorFlow	0.4208	81.87	0.4601	80.78

Table 4: Evaluation of custom CNN convolutional layer implementation.

When further inspecting the plots in figure 2, the PyTorch implementation demonstrates superior generalization. This is evident as the validation loss curve tracks the training loss curve more closely than the TensorFlow implementation does, indicating reduced overfitting. A similar, tighter correlation is observed when comparing the training and validation accuracy curves between the two frameworks.



(a) TensorFlow (Max Validation Acc: 80.78%).



(b) PyTorch (Max Validation Acc: 82.15%).

Figure 2: Training Loss and Accuracy curves for the implemented CNN in TensorFlow (left) and PyTorch (right).

A crucial difference in performance was traced back to the implementation of photometric augmentation. When applied to the TensorFlow model, photometric augmentation failed and resulted in validation accuracy dropping to around $49\% - 55\%$, indicating the layer was effectively disrupting the input data. However, applying the identical photometric augmentation approach within the PyTorch framework was successful, leading to an immediate improvement in model performance.

5 Conclusion

Through three separate tasks, this lab provided knowledge into the design, implementation and optimization of CNNs. Task 1 that involved implementing custom layers that successfully matched the Keras baseline and ahcieved 97.92% accuracy on the MNIST dataset.

A key finding confirmed the necessity of effective regularization, demonstrating that adding data augmentation makes it possible to utilize a small dataset and successfully prevent overfittin as shown in task 2. However, the effectiveness of augmentation proved dependent on the framework, in task 3 the attempt to introduce photometric augmentation in TensorFlow led to a failure, causing the model's performance on the PatchCamelyon dataset to collapse. When implementing the same photometric within the PyTorch environment was successful.

Use of generative AI

Generative AI has been used to assist in writing the abstract based on the method, results and conclusion of this report. It has also been used to explore alternatives to improve the performance of the CNN in task 3.

References