Lab report – TNM112

Deep learning for media technology, Lab 2

Filip Kayar (Filka428)

Friday 13th December, 2024 (17:35)

Abstract

This lab focused on designing, training, and evaluating convolutional neural networks (CNNs) for image classification tasks. Through three structured tasks, a deeper understanding of CNN architectures and their applications was developed. Task 1 involved implementing a custom CNN and validating it against a prebuilt Keras model, confirming the correctness of the implementation. Task 2 explored regularization techniques such as L2 regularization, dropout, and data augmentation, achieving strong generalization on a minimal dataset. Task 3 applied advanced augmentation and binary classification methods to the PatchCamelyon dataset, highlighting the importance of metrics like AUC for imbalanced datasets. Overall, the lab emphasized the critical role of regularization, thoughtful architecture design, and effective preprocessing in building robust and accurate models, with insights applicable to real-world challenges in fields such as medical imaging.

1 Introduction

Todays AI and deep learning models, convolutional neural networks (CNN) are a very popular and important part of teaching a model the act of "seeing". By using CNN we let the neural network see and classify. CNNs are particularly important due to their exceptional ability to extract objects and features from data, which makes them the backbone of many state-of-the-art applications in computer vision. Their architecture, inspired by the human visual system, has revolutionized the way machines interpret and analyze complex datasets. By extracting features from images such as edges and at higher levels even eyes or hands, for example.

This lab aims to design, train, and evaluate CNN models for image classification tasks. By implementing our own CNN, a deeper understanding of how CNNs work and their architecture will be achieved. This lab also aims is to understand the practical challenges of CNN implementation, such as architecture design, data processing, and model evaluation with the help of using Keras and tensorflow. By testing different methods and architectures a better understanding of CNNs will be achieved and learn the standard signs of overfitting. This knowledge is critical for building robust and accurate models that can generalize well to real-world applications.

This lab focuses on implementing a custom CNN and using it on the same dataset as a prebuilt Keras model to ensure the implementation is correct. Additionally, two datasets are used to construct and train two separate models,

one dataset is a simpler dataset of numbers while the second one is one containing tissue samples from breast lymph nodes. By comparing the results of the custom CNN with the Keras model and exploring challenges such as architecture design and overfitting, this lab provides a deeper understanding of CNNs and their practical applications in building accurate and robust classification systems.

2 Background

CNNs are utilized with tools such as Keras and TensorFlow, which provide powerful frameworks for designing, training, and evaluating machine learning models. Data preprocessing techniques, model evaluation, and the concepts of overfitting and generalization are central to the lab, ensuring a comprehensive exploration of CNNs and their practical applications. Overfitting means that the model adjusts itself to fit the training data very well and is not generalized, which leads to it giving bad results for the validation set and test set. By altering the input data, higher robustness can be achieved with the help of some Keras layers like rotation, brightness and translation, etc. The implemented CNN in this lab will not be used to evaluate with, due to its lack of ability to do parallel computations.

3 Method

This section will review the work done on the different tasks for this lab. The lab was implemented in Python using some already existing code. Keras, TensorFlow, and NumPy are some of the most used libraries for this lab and makes the work much easier.

3.1 Task 1

This first task was to implement our own CNN in `cnn.py` where the functions needed to be implemented was `activation`, `conv2d_layer`, `pool2d_layer`, `flatten_layer`, `dense_layer`, and `evaluate`.Additionally, compute the number of weights in the model, in the function `setup_model` [1]. This task will be using the MNIST dataset containing images of handwritten numbers where each image is 28x28 pixels.

The first function implemented was `activation` which was copied from lab 1 and consisted of an if-statement to determine what activation function is called. Within each if-statement the activation function is applied respectively, see the code below.

```
def activation(x, activation):
    if activation == 'linear':
        return x
    elif activation == "sigmoid":
        return 1/(1+np.exp(-x))
    elif activation == "relu":
        return np.maximum(0, x)
    elif activation == "softmax":
        return np.exp(x)/np.sum(np.exp(x), keepdims=True)
    else:
```

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

The second function implemented was `conv2d_layer` which takes in four parameters. The first parameter is activations from the previous layer, followed by convolution kernels with size of [kernel height, kernel width, channels prev. layer, channels this layer], then the bias vector, and lastly what kind of activation function. The function first assigns CI and CO which is the channels in and out respectively from the weight matrix W. Then we take out the size of the input and set the output with the same shape but with zeros. The output should be the same size since this function does not do pooling. The for loop goes through all of the output channels and calculates what the output for each channel should be. It starts with creating an empty output containing zeros and then goes over all the input channels. For each input channel, we get the corresponding kernel and flip it. Then we use the kernel to do convolution on the input channel j. All the convolutions are summed up and the bias is added to the output channel. Finally, we apply the activation function and set the corresponding output channel (i) in the output variable.

```
def conv2d_layer(h, W, b, act):
    CI = W.shape[2]
    CO = W.shape[3]
    x_out,y_out = h[:,:,0].shape
    output = np.zeros((x_out,y_out,CO))
    for i in range(CO):
        conv_sum = np.zeros(h[:,:,0].shape)
        for j in range(CI):
            kernel = W[:, :, j, i]
            flippedKernel = np.flipud(np.fliplr(kernel))
            conv_result = signal.convolve2d(h[:, :, j],
    flippedKernel, mode="same")
            conv_sum += conv_result
        conv_sum +=  b[i]
        output[:, :, i] = activation(conv_sum, act)
    return output
```

The `pool2d_layer` function is used to scale down channels and get a wider field of understanding for the model. The pooling function is implemented as below where the only input value is the activations from the convolution layer and consists of all the channels. We calculate the shape from the input channels and set the output for the pooling layer to be half the size, meaning we pool a 2x2 to one value. The first for-loop goes through all the channels, the second goes through the x-axis and the third goes through the y-axis. For each 2x2 space, we get the pooling values and choose the maximum value, this is called max pooling. Lastly, we set this value in the output in the corresponding position.

```
def pool2d_layer(h):
    sx, sy, c = h.shape
    sy = int(sy/2)
    sx = int(sx/2)
    h_out = np.zeros((sx, sy, c))
    for i in range(c):
        for j in range(sx):
            for k in range(sy):
                pooling_values = h[2*j:2*j+2, 2*k:2*k+2, i]
                max_value = np.max(pooling_values)
                h_out[j,k,i] = max_value
    return h_out
```

The simplest function implemented in this lab was `flatten_layer` which takes the output from a convolution network and aligns it as a vector. This is done to use it as an input to the dense layers. In NumPy there is a predefined function named flatten that aligns the input as a vector, see the code below.

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

The `dense_layer` uses the same theory as the feedforward function implemented in lab1. The underlying math behind `dense_layer` can be seen in equation 1 where everything inside the σ function is the same as z in the code below. The activation function is then applied to z to create the output for the dense layer.

```
def dense_layer(h, W, b, act):
    z = np.matmul(W,h) + np.squeeze(b)
    y = activation(z, act)
    return y
```

$$y^{(l)} = \sigma\left(W^{(l)}h^{(l-1)} + b^{(l)}\right), \tag{1}$$

To know how good the model is we implement `evaluate` function to determine the train- and test accuracy but also the train- and test loss. This function uses the same setup as for lab1 but instead of using mean square error, cross-entropy is used. The code below starts with calculating the predicted values for the training- and test set by feeding the dataset through feedforward. When we have the predicted values we use clip to get values between ϵ and $1 - \epsilon$, this is to avoid numerical instability, since we use log next in the code. Calculating the loss we use cross entropy which can be seen in equation 2. We sum over all the predictions and for each prediction we multiply the true one hot encoded vector by the logarithmic predicted one. Finally, it is divided by the number of samples. For accuracy, we sum up all the correct predicted samples and then divide it by the number of samples. This is done the same for both the test- and train set. See the code below for the implementation.

```
def evaluate(self):
        print('Model performance:')
        pred_x_train = self.feedforward(self.dataset.x_train)
        pred_x_test = self.feedforward(self.dataset.x_test)

        epsilon = 1e-10
        pred_x_train = np.clip(pred_x_train, epsilon, 1. - epsilon)
        pred_x_test = np.clip(pred_x_test, epsilon, 1. - epsilon)

        train_loss = (-np.sum(self.dataset.y_train_oh * np.log(
    pred_x_train)))/self.dataset.N_train
        train_correct = sum(np.argmax(pred_x_train, axis=1) == self
    .dataset.y_train)
        train_acc = train_correct/self.dataset.N_train
        print("\tTrain loss:     %0.4f"%train_loss)
        print("\tTrain accuracy: %0.2f"%train_acc)

        test_loss = (-np.sum(self.dataset.y_test_oh * np.log(
    pred_x_test)))/len(self.dataset.x_test)
        test_correct = sum(np.argmax(pred_x_test, axis=1) == self.
    dataset.y_test)
        test_acc = test_correct/len(self.dataset.x_test)
        print("\tTest loss:      %0.4f"%test_loss)
```

```
        print("\tTest accuracy:  %0.2f"%test_acc)
```

$$\text{Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log \hat{y}_i \tag{2}$$

The `setup_model` sets all the parameters for the model. First, it sets what kind of activation function, then the names for each layer, followed by the weight matrix and biases. When calculating the number of weights in the model, which includes convolutional kernels, weight matrices, and bias vectors we use a for-loop. The loop goes over each layer name to know what kind of layer it is. If the layer is a convolution one or a dense layer we take the shape of the weight in position i and then multiply the dimensions to get the total number of weights. Then we add the biases for this layer and add it to the total number of weights. If the layer is not convolutional or dense then it does not contain any weights and we can skip that layer. An example of a layer we can skip is the pooling layer since it does not contain weights.

```
def setup_model(self, W, b, lname, activation):
        self.activation = activation
        self.lname = lname
        self.W = W
        self.b = b
        self.N = 0
        for i, layer_name in enumerate(self.lname):
            if layer_name == 'conv' or layer_name == 'dense':
                weight_shape = W[i].shape
                num_weights = np.prod(weight_shape)
                num_biases = len(b[i])
                self.N += num_weights + num_biases
        print('Number of model weights: ', self.N)
```

When the `cnn.py` is fully implemented we run the code that is provided with the lab files to see that the result should be the same as the Keras implementation. Since the weights and structure is taken directly from the Keras model the accuracy and loss should be equal within machine precision. To test it fast we can use the feedforward sample function with a random initialized network where the weights are randomized. The results will be shown later in the result part of the report.

3.2 Task 2

This task uses the same dataset as previous section, which is the MNIST dataset of numbers. Although a small subset of 128 randomly selected images from the dataset is used to explore the effects of overfitting. How can regularization strategies and the structure of the network help to avoid overfitting. The objective is to expand a given network and apply techniques such as data augmentation, dropout, weight decay, and batch normalization to improve the generalization of the model and avoid overfitting. The given model from the lab files will be modified and the final model that gave good results can be seen in the code below.

```
def conv_block(x, N, chan, kernel_size, activation,padding='same'):
    for i in range(N):
```

```
        x = layers.Conv2D(chan, kernel_size=kernel_size, activation=None, padding=padding, kernel_regularizer=keras.regularizers.l2(1e-4))(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
    return layers.MaxPooling2D(pool_size=(2, 2))(x)

epochs = 110
batch_size = 16
K = 5
acc = np.zeros((K,2))
for k in range(K):
    print("Running step ", k+1, " of ", K)
    data = data_generator.DataGenerator()
    data.generate(dataset='mnist', N_train=128)

    keras.backend.clear_session()

    x      = layers.Input(shape=data.x_train.shape[1:])

    #Augmentation
    x = layers.RandomRotation(factor=0.05)(x)
    x = layers.RandomTranslation(height_factor=0.05, width_factor=0.05)(x)
    x = layers.RandomBrightness(factor=0.1)(x)

    #conv/dropout/norm
    conv1  = conv_block(x, N=2, channels=64, kernel_size=(3,3), activation='relu', padding='same')
    conv2  = conv_block(conv1, N=2, channels=128, kernel_size=(3,3), activation='relu', padding='same')
    dropout1 = layers.Dropout(0.1)(conv2)
    conv3  = conv_block(dropout1, N=2, channels=128, kernel_size=(3,3), activation='relu', padding='same')
    dropout2 = layers.Dropout(0.1)(conv3)

    #Flat
    flat1  = layers.Flatten()(dropout2)

    #Dense layer
    dense1 = layers.Dense(512, activation='relu')(flat1)
    dropout3 = layers.Dropout(0.1)(dense1)

    y = layers.Dense(data.K, activation='softmax')(dropout3)

    model = keras.models.Model(inputs=x, outputs=y)
    model.summary()

    opt = keras.optimizers.Adam()
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    log = model.fit(data.x_train, data.y_train_oh, batch_size=batch_size, epochs=epochs, validation_data=(data.x_valid, data.y_valid_oh), validation_freq=10, verbose=True)

    util.plot_training(log)
    acc[k,:] = util.evaluate(model, data, final=True)

print("Avarage performance over ",len(acc[:,1]) ," evaluations: ", np.mean(acc[:,1]))
```

The model consists of convolutional layers and dense layers that can be

trained. The predefined function `conv_block` is used to calculate the convolution easier with built-in functionality in it. It has a for-loop to do convolution N times and for each loop, we apply 2D convolution with the given kernel, for which we use a (3,3) kernel and also regularization. The regularization is used to prevent overfitting, in this case, we use L2 regularization also known as weight decay. This adds a penalty to the loss function proportional to the sum of the squared values of the model's weights. This discourages large weight values, helping to prevent overfitting and improving the model's ability to generalize to new data. After the convolutional layer, we do batch normalization which ensures that the input to the next layer has a consistent distribution of values with mean = 0 and variance = 1. It also allows faster convergence meaning the model can learn faster, potentially leading to better performance in less time. The result is then used with ReLU and a pooling layer with (2,2) pooling kernel which scales down the result by half in each axis, and then we return it.

The models structure is built up by first using augmentation such as random rotation, random translation, and random brightness. Their values are 5% rotation, (x: 5%, y:5%) transaltion and 10% brightness. Since the database consists of numbers, using too much translation or rotation will remove the data's meaning. For example, if we rotate a nine with half a rotation the data will now represent a six instead and lose its true representation. After the augmentation, we use the `conv_block` function to do convolution twice and for the second and third convolutions, we add a small dropout of 10% each. The number of feature maps of each layer is 64 then 128 followed by 128 again. Then we use a flattening layer to feed the output from the convolution layers to a dense layer since dense layers need the input as a vector. The dense layer consists of 512 neurons and ReLU activation function, feeding its output to a dropout layer with 10% and lastly, we feed it to the prediction layer with softmax. In this model, we use Adam optimizer and a cross-entropy loss function. Finally, we use the evaluate function to get how good the model is and all this is done K times to get a weighted average, we use K=5 in this lab.

Performance is calculated on a validation set during development, with the final evaluation conducted on the test set. To account for the stochastic nature of optimization and the random selection of the training subset, the evaluation is repeated for at least five runs. The average of the results is reported to ensure robust conclusions. This experiment highlights the importance of regularization in building accurate and reliable models, even with limited datasets.

3.3 Task 3

This task is similar to Task 2 but instead of using the MNIST dataset, we use the PatchCamelyon dataset containing tissue samples from breast lymph nodes. These could either be healthy or contain tumor tissue, i.e. this is a binary classification problem compared to MNIST [1]. The code for `conv_block` is the same in this task and to start we used the same code from Task 2 to have somewhere to start. See the code below for the implementation that gave good results.

```
def conv_block(...): #Same code as in Task 2 (For code see Task 2)

epochs = 16
batch_size = 70
```

```
5
6  data = data_generator.DataGenerator()
7  data.generate(dataset='patchcam')
8
9  keras.backend.clear_session()
10
11 #Input
12 x = layers.Input(shape=data.x_train.shape[1:])
13
14 #Augmentation
15 x = layers.RandomRotation(factor=0.4)(x)
16 x = layers.RandomTranslation(height_factor=0.1, width_factor=0.1)(x
       )
17 x = layers.RandomBrightness(factor=0.15)(x)
18 x = layers.RandomFlip(mode="horizontal_and_vertical")(x)
19
20 #conv
21 conv1  = conv_block(x, N=2, channels=128, kernel_size=(3,3),
       activation='relu', padding='same')
22
23 conv2  = conv_block(conv1, N=2, channels=128, kernel_size=(3,3),
       activation='relu', padding='same')
24
25 conv3  = conv_block(conv2, N=2, channels=256, kernel_size=(3,3),
       activation='relu', padding='same')
26
27 #Flat
28 flat1 = layers.Flatten()(conv3)
29
30 dropout1 = layers.Dropout(0.3)(flat1)
31 dense1 = layers.Dense(256, activation='relu')(dropout1)
32 dropout2 = layers.Dropout(0.3)(dense1)
33 dense2 = layers.Dense(256, activation='relu')(dropout2)
34
35
36 y = layers.Dense(data.K, activation='softmax')(dense2)
37
38 model = keras.models.Model(inputs=x, outputs=y)
39 model.summary()
40
41 opt = keras.optimizers.AdamW()
42 model.compile(loss='categorical_crossentropy', optimizer=opt,
       metrics=['accuracy','AUC'])
43
44 # Training
45 print(data.x_valid.shape)
46 log = model.fit(data.x_train, data.y_train_oh,
47                 batch_size=batch_size,
48                 epochs=epochs,
49                 validation_data=(data.x_valid, data.y_valid_oh),
50                 validation_freq=4,
51                 verbose=True,
52
53 util.evaluate(model, data)
54 util.plot_training(log)
```

The input is first augmented as we did in the previous task but in this dataset, the images do not have a rotation and can be rotated more and we also add a flip, which flips the images randomly in the horizontal and vertical axis. Except for the rotation and flips we also have the translation and brightness. For rotation we use 40%, translation with 10% in both directions and brightness

of 15%. After the augmentation, the data is fed through three convolutional layers with 128, 128, and 256 channels in corresponding order. The output of the convolutional layers is then flattened and fed to a dense layer with dropout before each dense layer with 30% dropout. This time, two dense layers are used with 256 neurons in each. The dense layer is fed to the output layer with softmax to determine if the tissue is healthy or not. In this task we use the AdamW optimizer with cross-entropy but also look at AUC, which stands for area under curve and is a good measure when the two different classes are unevenly distributed. Finally we evaluate the mode to get the result.

4 Results

This section will go through all of the results that was achieved in the different tasks.

4.1 Task 1

The results from a network trained with Keras on the MNIST dataset can be seen in Table 1 while our own CNN implementation, which copies the weights from the trained model using Keras, can be seen in Table 2. As we can see, the results are identical since we use the same weights and model structure. This shows that our own implemented CNN model works as it should.

Metric	**Value**
Trainable params	4 266
Train Loss	0.0603
Train Accuracy	98.11%
Test Loss	0.0683
Test Accuracy	97.71%

Table 1: Keras model performance metrics and params

Metric	**Value**
Trainable params	4 266
Train Loss	0.0603
Train Accuracy	98.11%
Test Loss	0.0683
Test Accuracy	97.71%

Table 2: Our own CNN model performance metrics and params

In the lab files, we have some code to compare the output of a randomly initialized Keras model and our CNN, for a single image. This is fast to compute, which will be convenient to check that our layers are correctly implemented. We get that our model gets an absolute sum of 1313.510246699676 while the Keras model gets 1313.5103 and an absolute difference of 3.693997859954834e-05 which tells us that our model works well. The difference is due to machine error and the Keras model only gives a precision of four decimals.

4.2 Task 2

The network implemented in Task 2 was run five times and gave results as in Table 3 where the average performance over these five runs was about 91.64%. The runs gave about the same results and about the same training curves and one of the runs can be seen in Figure 1.

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
Train loss:	0.0508	0.0507	0.0505	0.0509	0.0506
Train accuracy:	100.00%	100.00%	100.00%	100.00%	100.00%
Test loss:	0.2806	0.4066	0.2974	0.3567	0.3509
Test accuracy:	93.06%	90.17%	92.80%	91.34%	90.84%

Table 3: Model performance metrics across multiple runs

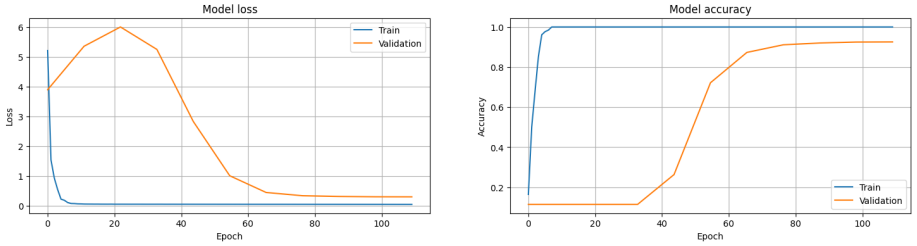

Figure 1: Model loss and model accuracy of the first run

We see that the models perform very well with slightly above 90% accuracy for the test set which is really good when considering the small data set used which only consisted of 128 images. This shows how the model can predict well if it is implemented in a good way.

4.3 Task 3

With the code implemented in Task 3, we evaluate it four times and the accuracy and loss can be seen in Figure 2 for each run. In Table 4, the result of each run can be seen with train and validation for loss, accuracy, and AUC.

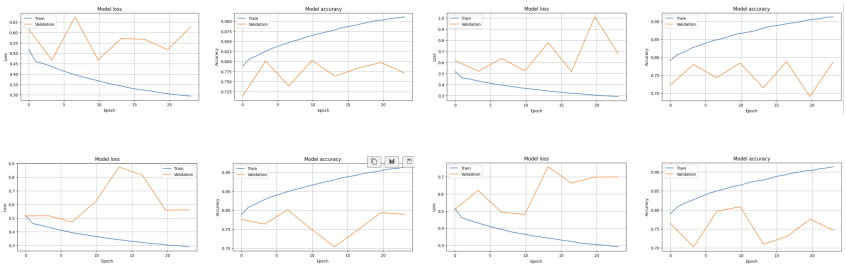

Figure 2: Model accuracy and loss in the four runs

As we can see the model performance well with good accuracy slightly under 80% for the validation set. For this task, the test set is not available since it is part of a competition and will not be shown in this report.

	Run 1	**Run 2**	**Run 3**	**Run 4**
Train Loss	0.3595	0.3178	0.3280	0.3710
Train Accuracy	86.2%	90.14%	88.65%	85.52%
Train AUC	0.94	0.96	0.92	0.94
Validation Loss	0.4671	0.5166	0.4712	0.4801
Validation Accuracy	80.26%	78.8%	80.11%	80.84%
Validation AUC	0.89	0.87	0.89	0.89

Table 4: Performance of the model in four different runs, including Train and Validation AUC.

5 Conclusion

This lab provided valuable hands-on experience in designing, training, and evaluating convolutional neural networks (CNNs) for image classification tasks. By working through the tasks, we gained a deeper understanding of CNN architectures and the challenges involved in building robust models.

In Task 1, implementing a custom CNN from scratch highlighted the core mechanics of layers such as convolution, pooling, and activation. Comparing the results with a prebuilt Keras model validated our implementation, demonstrating its correctness through identical results.

In Task 2, exploring regularization techniques such as L2 regularization, dropout, and data augmentation emphasized their importance in preventing overfitting. The network achieved over 90% accuracy on the test set despite being trained on a minimal subset of 128 images. This shows how well-designed architectures and regularization can lead to strong generalization even with limited data.

In Task 3, working with the PatchCamelyon dataset underscored the differences between binary and multi-class classification. Advanced augmentations like rotations and flips were used effectively to improve model performance. Evaluating metrics such as AUC demonstrated the model's ability to distinguish between classes, achieving high validation accuracy and robust predictions.

Overall, this lab highlighted the importance of regularization, proper architecture design, and thoughtful data preprocessing in training CNNs. It also highlighted the practical trade-offs between model complexity and generalization. These insights are critical for developing reliable models for real-world applications, particularly in domains like medical image analysis.

Use of generative AI

Generative AI has been used in this report to explain some parts that were hard to grasp. Some parts of the method section were structured with the help of AI to get a better structure of the content and help with some sentence building. In conclusion, AI was a big help in summarizing the findings and what was done in each task. This was also done on the abstract and was all generated by AI, since generative AI summarizes the report very well.

References

[1] TNM112. Lab02 - convolutional neural networks, 2024. Lab description – Deep learning for media technology.