Student Research Project

Telecommunication Engineering

Estimating thread densities in X-rays of old canvases with PyTorch

Autor: Max Leitner

Tutor: Juan José Murillo Fuentes

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Student Research Project
Telecommunication Engineering

Estimating thread densities in X-rays of old canvases with PyTorch

Autor:
Max Leitner

Tutor:
Juan José Murillo Fuentes

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Student Research Project: Estimating thread densities in X-rays of old canvases with PyTorch

Autor: Max Leitner

Tutor: Juan José Murillo Fuentes

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El secretario del Tribunal

Abstract

The goal of the following student work is to rewrite a model, implemented in Keras, to PyTorch. Those models execute a thread counting task on images, representing 1cm squared patches of radiographs, which are taken from old paintings. Both models area trained and compared in terms of accuracy and efficiency, which includes resource consumption in form of memory and runtime. By evaluating those metrics, it can be determined how both frameworks behave and if a performance boost can be achieved through the change of the framework.

To answer this question, structural differences between the model implementations of both frameworks are analyzed and the gained knowledge is used to recreate the model and the corresponding training structures. Each model is trained 10 times with the same training parameters, after which the best performing model is selected to perform an additional comparison on the test dataset. During the training, performance measurements are taken.

The experiment showed that models from both frameworks equally performed in terms of accuracy. PyTorch does perform worse regarding training speed. On average, it´s models take 1,7 times as long to finish an epoch of their training. On the other hand, PyTorch thrives in the matter of memory consumption, consuming only one third of the available GPU memory with no additional allocation. Keras utilized the full 16GB of GPU memory, while still allocating 12GB of memory additionally.

Without an individual adaptation of training parameters, no overall improvement could be achieved on the dataset, through a change of the framework.

Keywords:

Keras, PyTorch, Model migration, Thread counting, Radiograph analysis, Performance evaluation, Memory utilization, Training speed, Accuracy

Abstracto

Título: Estimación de la densidad de hilos en radiografías de lienzos antiguos con PyTorch

El objetivo del siguiente trabajo de estudiante es reescribir un modelo, implementado en Keras, a PyTorch. Estos modelos ejecutan una tarea de recuento de hilos en imágenes en recortes de 1cm cuadrado de radiografías, tomados de pinturas antiguas. Ambos modelos se entrenaron y compararon en términos de precisión y eficiencia, lo que incluye el consumo de recursos en forma de memoria y tiempo de ejecución. Evaluando esas métricas, se puede determinar cómo se comportan ambos entornos y si se puede conseguir un aumento del rendimiento mediante el cambio de entorno.

Para responder a esta pregunta, se analizan las diferencias estructurales entre las implementaciones del modelo de ambos entornos y los conocimientos adquiridos se utilizan para recrear el modelo y las estructuras de entrenamiento correspondientes. Cada modelo se entrena 10 veces con los mismos parámetros de entrenamiento, tras lo cual se selecciona el modelo con mejor rendimiento para realizar una comparación adicional en el conjunto de datos de prueba. Durante el entrenamiento se realizan mediciones del rendimiento.

El experimento demostró que los modelos de ambos entornos obtuvieron los mismos resultados en términos de precisión. PyTorch obtiene peores resultados en cuanto a velocidad de entrenamiento. De media, sus modelos tardan 1,7 veces más en terminar una época de entrenamiento. Por otro lado, PyTorch destaca en lo que respecta al consumo de memoria, ya que sólo consume un tercio de la memoria disponible en la GPU sin ninguna asignación adicional. Keras utilizó la totalidad de los 16 GB de memoria de la GPU, sin dejar de asignar 12 GB de memoria adicional.

Sin una adaptación individual de los parámetros de entrenamiento, no se pudo lograr ninguna mejora global en el conjunto de datos mediante un cambio de entorno.

Palabras clave:

Keras, PyTorch, Migración de modelos, Recuento de hilos, Análisis radiográfico, Evaluación del rendimiento, Utilización de memoria, Velocidad de entrenamiento, Precisión

Index

CODE INDEX

FIGURE INDEX

TABLE INDEX

1 INTRODUCTION

With the rise of machine learning and deep learning within the last years, machine learning algorithms are applied in many fields of the daily life (Zeolearn 2023). Since 2022, the topic is also omnipresent in the media due to the publication of ChatGPT (Introducing ChatGPT 2023). The topic in this form is discussed under the title of "artificial intelligence". Machine learning also is used in scientific contexts to facilitate and improve analysations.

Through the collaboration of the Universidad de Sevilla and the Museo del Prado in Madrid, a combination of the machine learning topic and the painting analysis is created. Machine learning and digital analysis can facilitate the work of thread counting, which is needed to describe the structure of the canvas, on which the painting is painted on. For the execution of this work, image data of radiographs is used. By counting the number of horizontal and vertical threads on each square centimetre, conclusions can be drawn regarding the canvas structure and consequently the place of manufacturing (Delgado et al. 2023a).

Already two previous works are published by Delgado et al., presenting two different algorithms, which are meant to replace the manual counting technique or their atomization using Fast Fourier transformation, since this technique suffers from irregularities in the image data.

Goal of this work is the translation of the Inception Regression model based on the *VGG16* – structure, which is presented in the second paper and implemented in Keras, into a lower-level framework, which in our case is PyTorch. By evaluating the performance differences in between the two implementations, we aim to select the better framework for the model structure and dataset used.

To achieve this comparison, at first the background of the thread counting topic is analysed, presenting the already published projects. The model architecture to implement is taken from the second publication, while the spatial counting algorithm, presented in the first publication, serves for the label generation of the dataset. Secondly, previous publications on framework comparisons between Keras and PyTorch will be evaluated, to formulate an expectancy on the outcome of the conducted experiment. In this form possible performance gains or declines can be previously identified. It follows an analysis of the structural differences in between the two frameworks, to identify how the model can be correctly translated, that both implementations refer to the exact same network structure. This will be necessary to assure comparability of the measurement outcomes. Also, the dataset and needed steps of data generation are presented to obtain an understanding of the data processed by the models.

Using this knowledge, the model will be implemented, utilizing the PyTorch API. In addition to the model and training process, different measurement systems are implemented to evaluate the performance, after the training and prediction. The main performance indicators will be training and prediction timers, model accuracy and memory consumption.

According to those measurements, the better framework for the data and the model structure can be picked. The outcome of this selection is put in relation to the expectations formulated in the pre-examinations, taking the results of previous projects into account. That way, questions about the different behaviour of the frameworks can be answered and possible improvements through a change of the framework can be identified.

2 PROJECT BACKGROUND

This student research is based on the previous works of A. Delgado and Juan. J. Murillo-Fuentes from the Dep. Teoría de la Señal y Comunicaciones at ETSI Universidad de Sevilla in collaboration with Laura Alba-Carcelén by the Dep. Restauración y Documentación Técnica Museo Nacional del Prado. Objective of this previous work was the creation of a Deep learning model, allowing to analyse radiographs of paintings owned by the Museo Nacional del Prado to gain information about ownership and dating of those respective paintings.

2.1 Painting Analysis

When it comes to analyzing paintings, different approaches can be used, to assign a painting to a certain period. Besides the classic analysis of the painting itself and consideration of the style, in which it has been painted, as well as the historic properties already known about the painting, like ownership etc., also the materials can be explored. Depending on the desired information, different techniques can be used in the analysis of the materials, ranging from chemical or visual analysis to techniques like radiocarbon dating (Hendriks et al. 2019).

In this work and the previous projects, we are specifically referring to, we stick to the analysis of the canvas of the painting. As pointed out in the work of Delgado and Murillo-Fuentes the most important features for canvas analysis are the type of the fabric, the fabric material, the number of threads per centimeter, in both horizontal and vertical direction and the deviation of angles between those threads in consideration of their horizontal or vertical orientation (Delgado et al. 2023b).

The model mainly dealt with, analyzes the thread density of plain weave fabrics. Plain weave fabrics is considered the simplest and most common of the basic textile weaves. It characterizes by its construction as the fabric is produced by intertwining the horizonal and vertical threads (weft and warp yarn). Warp yarn titles the threads arranged at the loom in parallel from the front to the back. The weft then describes the yarn, which is introduced by the weaver to construct the fabric itself (Burnham 1980).

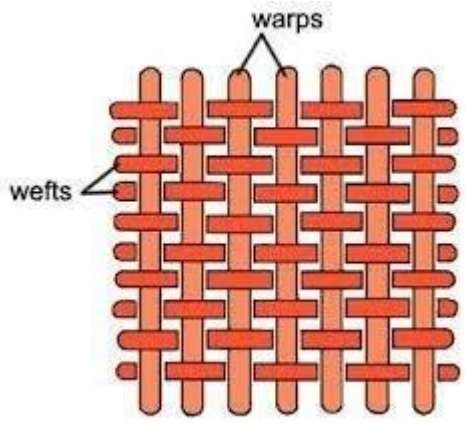

Figure 2-1. Weft and warp yarn of woven fabric (S Anila et al. 2018)

Hereby the weft yarn is passed over and under each warp yarn, changing in each row. This leads to a high number of intersections and gives them their characteristics as resistance against raveling, but therefore a tendency to wrinkle as well as less capacity to absorb paint compared to other weave forms (Encyclopedia Britannica 2023).

To run the fabric data through image analysis the X-ray of the painting is used, as the direct observation of the fabric normally is not possible, due to a piece of cloth, which is stuck to the back of the painting to reinforce the support. The quality of the obtained X-ray images heavily depends on factors like the used amount of paint and primer, as well as wood stretcher, nails and other objects present in the image area (Delgado et al. 2023b).

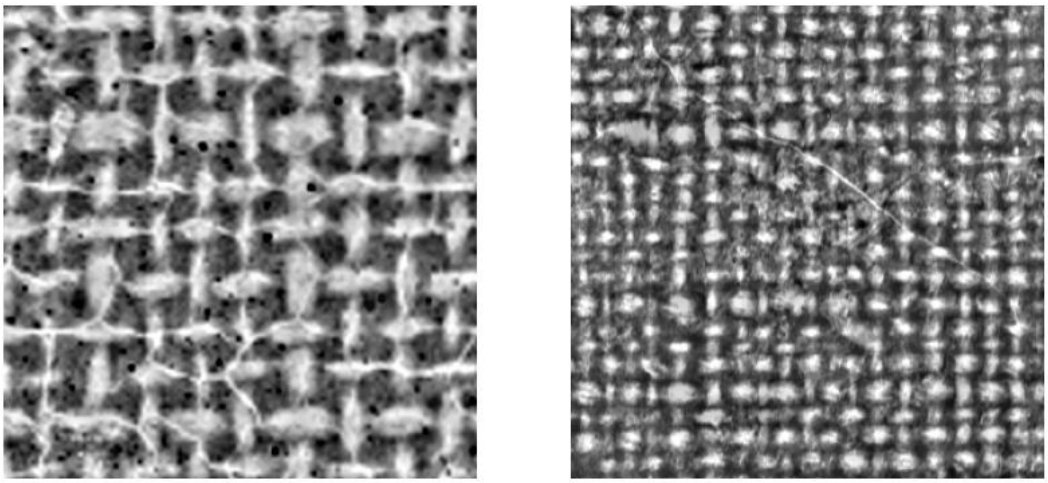

Figure 2-2. Example patches of radiographs with low and high thread density

Utility of those images opens up through the fact, that the separation and distances between the threads in plain weave are not regularly distributed but depend on the construction process. If two identical distributions are to be found it can be concluded that both canvases come from the same exact production. This helps curators to assign paintings to their painter if references can be found to other already studied paintings with secured knowledge of their origin (Delgado et al. 2023b).

2.2 Crossing Point Detection

The training of the model on the radiographic images needs all images in our dataset to be labeled. To avoid assigning every image a number of vertical threads by hand we refer to a previous work conducted by Delgado et al. (2023a) using a different system for thread counting and use their results as our labeled training data. As an underlying concept, spatial counting is used in the model, after identifying the crossing points of the horizontal and vertical threads in the image section. Therefore, two different algorithms were used in the process. The first one is the implementation of a Deep Learning model, to identify the crossing points and create an image representation of the image section, marking the crossing points.

The data patches, which will be processed by our own model, come with assigned labels, already marking the locations of horizontal and vertical threads. Consequently, in our case, the image processing with the deep learning model by Delgado et al. (2023a) can be skipped. Instead, the given labels are used to calculate the crossing points, by putting the labels on top of each other and calculating their intersection. Through finding the centroids of the intersections, we can obtain the exact locations of the crossing points. After this step, the same data is available, as if the images had been processed by the deep learning model without given labels. To complete the labeling task, the locations of the crossing points are given to the second algorithm.

Just mentioned algorithm is called spatial counting. To obtain the number of horizontal and vertical threads in our selected image section, every detected crossing point is taken and its closest neighbors are searched. The number of neighbors assigned to each crossing point is determined beforehand. Out of the neighbor selection the closest neighbor to the left and right, and at the top and the bottom is found. Their relation to the original crossing point is determined by the respective angle of a vector between those points. In the next step, the average

distance of those crossing points to their neighbors is used, to receive an estimation of the average thread density in the image section (Delgado et al. 2023a).

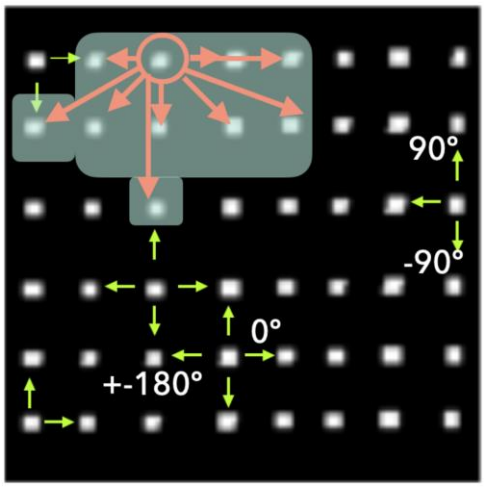

Figure 2-3. Visualization of the spatial counting algorithm (Delgado et al. 2023a)

The in this way obtained data is saved and used as our labeled training data. Additionally, to the thread density, in the referred paper also the thread angle was calculated and used for evaluation. Since our model just serves for thread counting, this technique is not further treated in this work.

2.3 U-Net Regression Model

The model we work on is called a U-Net regression model, which returns the vertical density of threads per centimeter. The first implementation of this model in Keras got presented by Delgado et al. (2023b). From this work we take the base structure of the model and translate it into a PyTorch model to evaluate.

Inspiration for this model was taken from the U-Net model proposed by Delgado et al. (2023a). As the original U-Net model, also the regression model takes advantage of the convolution applied with the help of the inception module. The name comes from the fact that the model structure is similar to the U-Net model, without including the decoder branch, but instead in the deepest stage of the U-Net, the tensor is flattened and, through processing the data with Dense layers, a single number is emitted from the output layer, representing the number of vertical threads in the analyzed image section.

The exact model to translate is an Inception VGG-Based Regression Model which represents an inception regression model, while being adapted to the VGG16 Simonyan and Zisserman architecture. This combination generates a regression model, using the inception model for the convolution, but the number of dense layers and the number of their respective neurons is designed according to the VGG architecture. According to Delgado et al. the designed structure aims to perform the feature extraction with the help of convolutional layers in the encoder path. In each convolutional layer two successive inception blocks and 2D Max Pooling are used. The regression itself is then performed in the second part, processing the data through a flatten layer and six following dense layers. All hidden layers apply the ReLU activation function, while the output layer is relying on linear activation.

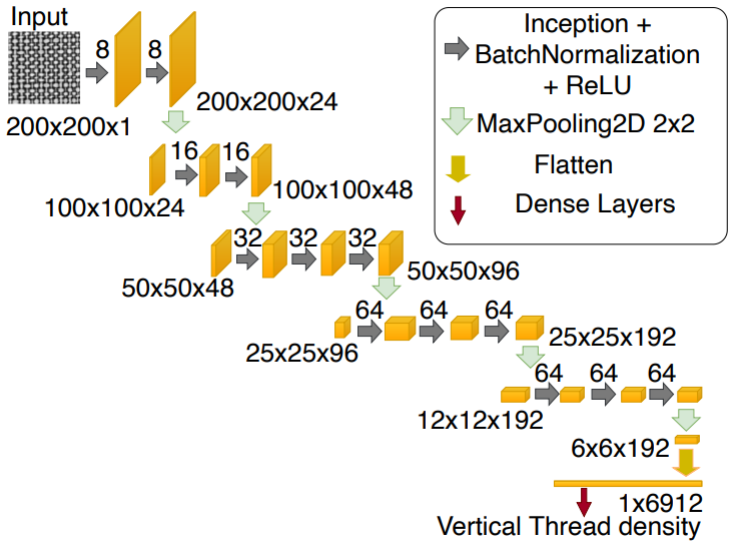

Figure 2-4. U-Net regression model with VGG16 structure (Delgado et al. 2023b)

2.4 Inception Module

In previous sections the inception module was mentioned as part of the U-Net regression model. This module contains the core functionality of the convolution performed in the encoder branch. Firstly, the inception module designed for the U-Net model processing painting radiographs was proposed in (Delgado et al. 2023a). The motivation behind this approach falls back to a possibility of great range of thread densities within an image section. Models with fixed kernel sizes struggle with a range, declared to lie between 6 to 23 threads per centimeter. As a consequence, crossing points cannot be located reliably. To counteract this problem the inception methodology and therefore multiple convolutional kernels of different sizes are used. In the presented paper, the quadratic kernels have sizes of 3, 5 and 7. Within every inception block, every kernel is applied separately, returning the same number of features, which are specified for every layer. The produced feature maps then are concatenated and further processed with batch normalization during training and a ReLU activation function. By concatenating, the resulting tensor has three times the number of features, specified beforehand. During the convolution itself, no reduction of the image size is taking place. This is because before the convolution step a zero padding is applied to every input image, so that the image size of input and output matches. Reduction of the image size is achieved by the MaxPooling2D layer after the inception module.

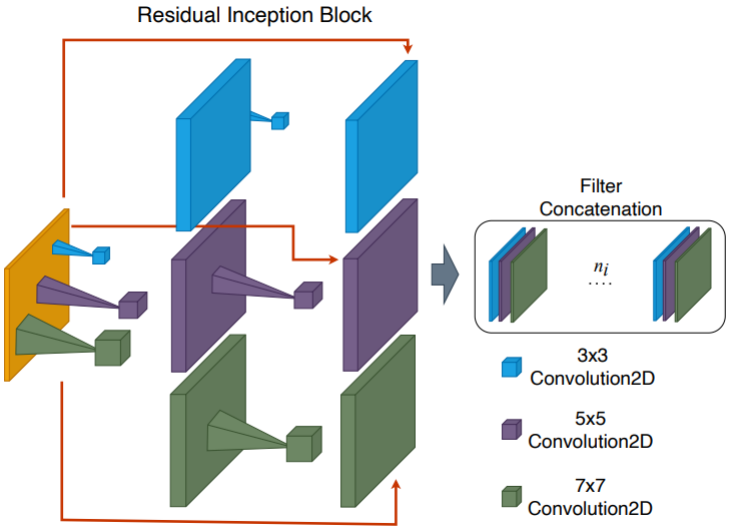

Figure 2-5. Schema of Inception Module (Delgado et al. 2023b)

3 KERAS – PYTORCH COMPARISON

As we translate the model from Keras to PyTorch at first we have to identify the biggest differences in terms of design and capabilities of both frameworks. Therefore, we compare the history of their respective developments, as well as the design differences, which must be considered when creating Deep Neural Networks with each of those frameworks.

Furthermore, we analyse the current research state in terms of speed and needed resources of each framework, to obtain an outlook on performance results we can expect after translating the model into PyTorch. By researching compatibility and equivalencies between the frameworks we can identify equivalent structures, to receive an exact translation of the model and preserve comparability.

3.1 Historic Development

The Keras framework was developed by François Chollet in 2015. It started out as a research project called ONEIROS, which stands for Open-ended Neuro-Electronic Intelligent Robot Operating System (Team 2023a) at the University of Montreal. Since the initial success of the *AlexNet* Model in 2012 in the area of computer vision, the development of new machine learning frameworks was driven by big companies, like Google, Facebook and Amazon. With the involvement of Google by developing TensorFlow, it enabled the development of Keras, since it relies on TensorFlow and Theano as its respective backends. It is written in Python and abstracts the more complex features of the underlying frameworks (Yuan 2020).

Two years after its official release, Keras got integrated into the TensorFlow library, to simplify the configuration of the framework. Then in 2018 Keras got added as well to the TensorFlow 2.0 library allowing it to make use of the new features and performance benefits introduced by this upgrade. It allows accessing TensorFlow functionality through a high-level API, which Keras is representing (The History of Keras: From Research Project to Industry Standard – TS2 SPACE 2023).

PyTorch's release is dated one year after the Keras release. In October 2016 it got introduced as an internship project by Adam Paszke. It is based on the Torch project, a machine learning framework written in the Lua programming language. Finally, it got released by the Facebook AI research team and is mainly written in Python and C++. Today deep learning projects by Facebook, Uber and Twitter are created with PyTorch (Kurama 2021; Moltzau 2020).

3.2 Structural Analysis

The main structural differences between Keras and PyTorch conclude out of their basic use-case. Keras is a python based high level API, which is very beginner friendly and requires less background understanding than other frameworks for the construction of Deep Neural Networks (DNNs). It runs on a different possible backends, like TensorFlow or Theano.

PyTorch on the other hand is a machine learning library, which can be implemented into Python and C++ code. Therefore its API operates on a much lower level, which complicates certain structures, but at the same time

offers better capabilities for debugging, adjusting DNNs to personal preferences or influencing the computational workflow (Keras vs PyTorch 2020).

3.2.1 Model Creation

Both frameworks allow different approaches to create deep learning model. The Keras framework offers three different possibilities for model creation. Those are the use of the sequential model, the object-oriented approach and the functional API. PyTorch offers similar techniques with its object-oriented approach, as well the possibility to create sequential models. A functional API, like Keras has, is missing and for this makes Keras more beginner friendly (Agarwal 2019).

3.2.1.1 Functional API

The functional API from Keras allows to create complex models without the need to create those respective models with the object-oriented approach through class definition. Like the object-oriented definition it relies on the *tf.keras.Model* class. For the custom deep neural network there is no need to declare a new class. We begin by declaring an input layer and provide a shape for this input tensor. Then we chain the following layers, by handing them the outputs of previous layers and catch the output of our output layer in a variable. Finally, we instantiate our model by calling the *tf.keras.Model* class and providing the input and output layer as arguments of the class call. For readability of the code, those steps can be united within a *"create_model()"* function, which then contains the chaining of layers and the model class construction. The API is not limited to create complete models from the input to the output layer but allows at the same time to construct partial models from intermediate layers, which would then share those layers. This allows for example to create an autoencoder, but at the same time access the encoder or decoder separately (Team 2023b).

```
import tensorflow as tf

inputs = tf.keras.Input(shape=(3,))
x = tf.keras.layers.Dense(4, activation=tf.nn.relu)(inputs)
outputs = tf.keras.layers.Dense(5, activation=tf.nn.softmax)(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Code 3-1. Example code for model creation with Keras functional API (Team 2023c)

3.2.1.2 The sequential class

The sequential class is the easiest way to create a model in both frameworks. It is used to create simple models, which just have one input and one output tensor. It creates a stack of layers, through which the input gets passed. In Keras we need the *keras.Sequential* class. The respective pendant in PyTorch is the *torch.nn.Sequential* class. Both work same sense. On construction of the model class with the *Sequential* constructor, all contained layers get passed to the model in order. The class itself then takes over the part of connecting the layers and passing tensors through themselves, when the model is called. In this approach for Keras no input or output size has to be given, as the model takes tensors of every input size and then sequentially chains the outputs of previous layers to the following ones. PyTorch still needs the specification for input and output sizes. While in PyTorch the layers are given comma-separated to the constructor or as a ordered dictionary *OrderedDict*, Keras' models takes a list of layers (Sequential — PyTorch 2.0 documentation 2023; Team 2023c; Chng 2022).

```
import tf.keras as keras
import tf.keras.layers as layers

model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(3, activation="relu"),
        layers.Dense(4),
    ]
)
```

Code 3-2. Example creation of a sequential model in Keras (Team 2023d)

```
import torch.nn as nn

model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
```

Code 3-3. Example creation of a sequential model in PyTorch (Sequential — PyTorch 2.0 documentation 2023)

Both classes allow the appending of layers to the end of the model. In Keras this can be done with the *add()* function of the model, while in PyTorch the *append()* function of the model fulfils this functionality.

3.2.1.3 Object-oriented approach

The object-oriented approach represents the most complex way to create a model in both frameworks. At the same time, it allows the highest level of flexibility and customizability, when it comes to designing neural networks. In this design for each model a class must be created. The class then needs to inherit either *tf.keras.Model* in Keras or *torch.nn.Model* in PyTorch.

The *__init__()* function contains the model description itself. In this section all used layers have to be defined according to the frameworks' syntax. In both versions every layer gets defined as a new attribute of the object class itself. Within the initialization no linking between the layers is done. The main difference hereby lies in the properties of the layers themselves, as in PyTorch input and output sizes must be defined, whereas in Keras it is sufficient to declare layer sizes. Both can be seen in the Code examples (Code 3-4, Code 3-5). In the basic models, the initialization function just takes the model itself as an argument. If needed by design, additional arguments can be added, which have to be passed to the constructor on model creation.

```
import tensorflow as tf

class MyModel(tf.keras.Model):

  def __init__(self):
    super().__init__()
    self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
    self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

  def call(self, inputs):
    x = self.dense1(inputs)
    return self.dense2(x)

model = MyModel()
```

Code 3-4. Example code for creating a Keras model by class definition (Team 2023c).

```
import torch

class TinyModel(torch.nn.Module):

    def __init__(self):
        super(TinyModel, self).__init__()

        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x

tinymodel = TinyModel()
```

Code 3-5. Example code for creating a PyTorch model by class definition (Building Models with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation 2023)

To link the layers and allow the training on data, as well as the processing of data a function *call()* in Keras or *forward()* in PyTorch must be added. The function takes the model itself and the input tensor as arguments. This input tensor then must get passed through the layers. For this, the input gets passed as argument to the layers predefined in the initialization function and their output handed to the following layer. The output of the last layer is at the same time the output tensor of our model and therefore the return value of the *forward()* function (Chng 2022; Team 2023b; Building Models with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation 2023).

3.2.2 Model Training and Test

While the creation of models in Keras and PyTorch is similar and differs mainly when the functional API of Keras is used, the way of training those models is hardly similar.

3.2.2.1 Training models in Keras

In Keras three main functions of the model are used to fulfil the training and test purpose. The first one is the *compile()* function. After creating the model, it must be configured for training. By calling the *compile()* function we can define configurations, like the optimizer, the loss function or the evaluation metrics to use.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.FalseNegatives()])
```

Code 3-6. Example code for Keras model compile function (Team 2023b)

In the next step the actual training of the model takes place. It is handled by the *fit()* function of the model. The handed parameters include the training data in form of the input tensors, as well as target data and training parameters like batch size, number of epochs and validation data, if needed. Also call back functionality, like early stopping, learning rate scheduling and training checkpoints can be handed to the *fit()* function.

```
history = model.fit(
    x_train,
    y_train,
    batch_size=64,
    epochs=2,
    validation_data=(x_val, y_val),
)
```

Code 3-7. Example code for Keras model fit function
(Training and evaluation with the built-in methods)

The last needed functionality is the possibility to test the model on new data. For this purpose exists the *evaluate()* function. If called, it takes the test data in form of the input tensors, as well as the target data together with properties for the test run, like batch size, initial sample weights or call back functionality. Based on the metrics already provided in the compile step, the accuracy of the model on the test data can be evaluated (Training and evaluation with the built-in methods).

3.2.2.2 Training models in PyTorch

Creating a training setup in PyTorch requires more knowledge of the training process, as it must be set up by hand, without the comfort of calling a function like *fit()* in Keras. Instead, we have to write a training loop, in which we pass the data to our model and calculate the loss function to update the optimizer accordingly.

Therefore, we define the optimizer and the loss function before our training loop with their respective properties. In the best case, the data is already organized in form of a *Dataset*, which can be handled by a *DataLoader*. This allows the easier handling of data, when training the data in batches. If not, it is possible to define a custom dataset and corresponding data loader, to handle the data. If the dataset is small enough, the training data can be directly passed to the model, as seen in Code 3-8. Before starting the training, the model has be set into training mode by calling *model.train()*, to enable gradient tracking and batch normalization. Within the training loop, which tracks the number of training epochs, the following steps are minimally necessary to enable proper model training:

- Increase the epoch counter
- Set the gradients of the optimizer to zero
- Pass the training data through the model
- Compute the loss function for the calculated and given output
- Compute the gradients of the loss function by back tracking in the model layers
- Adjust the learning weights of the optimizer

```
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=0.01)

epochs=500
final_losses=[]

model.train()

for i in range(epochs):

    i= i+1
    optimizer.zero_grad()

    y_pred=model.forward(X_train)
    loss=loss_function(y_pred,y_train)
    final_losses.append(loss)

    loss.backward()
    optimizer.step()
```

Code 3-8. Example code of a simple training loop in PyTorch (Red Hat Developer 2022)

Additionally, to this basic functionality, more advanced features, like early stopping or learning rate adaptation can be defined before the training loop and get adjusted after each epoch. If validation data is considered, a second validation run has to be performed after the training epoch, where the model is put into evaluation mode by calling *model.eval()*. Then the validation data gets passed through the model and based on the output the validation metrics can be calculated (Red Hat Developer 2022; Training with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation 2023).

3.3 Performance Analysis

3.3.1 Referenced Work

Both Keras and PyTorch are considered powerful tools for the creation of deep neural networks. Due to their structural differences and their intention of usage, they also have differences in terms of performance results. Since Keras is a higher-level API and written in Python it lacks the performance lower-level frameworks like PyTorch can offer. The first performance aspect to compare is training and prediction speed. PyTorch is considered to be faster and more suitable for bigger datasets, but at the same time also requires more complex preparation and is less suited for quick setups in prototyping (Terra 2020).

The second performance aspect which needs to be considered is the performance in terms of accuracy, learning capabilities and overfitting. To get more insights into performance differences we can compare results of previous setups, which implemented their models in both frameworks.

A comparison between the frameworks was done in the work by Kim et al. 2022. In this project the Dogs and Cats Dataset by Kaggle got analyzed, which contains 25000 labeled images of dogs and cats. The results of the model implementation were obtained with a 75-25 training-test split. Next to PyTorch and Keras, also the MXNet library got analyzed, which will not be considered in our comparison. With each framework a CNN model was created, trying to keep similarities within the model layers as close as possible. Also, for the models the same hyperparameters got used, to keep the models comparable. The proposed model consisted of stacked convolutional layers with three-by-three filters. Those were combined with max-pooling layers and a ReLU activation function. For the output layer, a sigmoid function was used, to deal with the binary classification. To determine the performance of the models, the accuracy and F1-score (Korstanje 2021) of each model got

calculated. Since the goal was binary classification, true and false values got assigned to the cat/dog attribute. Next to the named metrics, also the cross-entropy loss (Koech 2020) got calculated. After the training of ten epochs, it was observable, that the Keras model showed better accuracy values at 78%, but suffered overfitting, while the PyTorch model came with a 70.8% accuracy and therefore having a good fit on the data. In a final test of the models on testing image samples they observed that, throughout different sample sizes, the Keras model achieved high accuracy ranging from 89-98%. The PyTorch model on the other hand always got outperformed, through achieving worse accuracies than the Keras model, ranging in between 85-92%. The same tendency could be observed, when comparing the F1 scores. Although the better performance of the Keras model can be observed in all given sample sizes, the authors of the work do not ultimately claim a superiority in performance for the Keras model as, with the random sampling, accuracy and F1-scores differed in every test run (Kim et al. 2022).

A second project, which dealt with comparing the frameworks PyTorch and Keras, was published by KABAKUŞ in 2020. Both frameworks got examined by creating three different types of neural networks: a Feedforward Neural Network (FNN), a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN). Each models' performance got evaluated by using three different measurements: training time, testing time and prediction accuracy. The FNN was constructed out of three Dense layers with the sizes 64-64-10, where the first two worked with the ReLU activation function and the last one with SoftMax activation. As a dataset, the MNIST dataset consisting of handwritten digits in greyscale from zero to nine with a size of each 28x28 pixels was used (MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges 2013). For the training of the neural network, the researchers used the *Adam* optimizer. A second model type, which got analyzed, was a CNN, relying on the *VGG16* architecture. The structure of the *VGG16* architecture can be seen in Figure 3-1. To train the CNN, the *CIFAR-10* (CIFAR-10 and CIFAR-100 datasets 2017) dataset was used. This dataset includes a collection of colored images, which can be assigned to ten different classes. Within those classes different animal labels, like cat, frog and deer, but also vehicles, like trucks and airplanes can be found. Model type number three were the RNNs. In this example a LSTM (Long short-term memory) implementation was used. The structure of the LSTM model can be seen in Figure 3-2. To benchmark this model, it got trained on the *IMDB Movie Review* dataset (N 2019). It consists of IMDb movie reviews which got labelled either as *positive* or *negative*. Each movie review hereby is represented by an encoding in form of a list of word indexes.

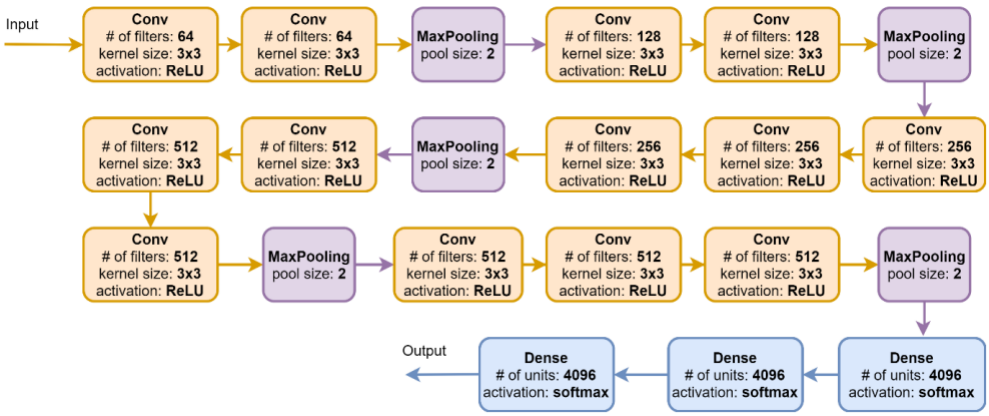

Figure 3-1. Architecture of the *VGG16* CNN model (KABAKUŞ 2020)

Figure 3-2. Architecture of the LSTM model (KABAKUŞ 2020)

The researchers evaluated the models, by running them on the Google Colab platform, to benefit from the available GPU acceleration. 60,000 training and 10,000 test images were used out of the *MNIST* dataset to train and evaluate the FNN model. In the results Keras performed better than PyTorch in every of the compared categories. With an accuracy of 97.24% to 96.69% by PyTorch it reached slightly better results. In terms of speed, the Keras model was 3.8 times faster during the training and 2.4 times during the testing. Based on those results the researchers concluded, that Keras would be the better choice for the design of Feed Forward Networks.

A similar tendency could be found for the design of the Convolutional Neural Network. Keras was found to be more accurate, by having 78.43% accuracy compared to 76.54% by PyTorch. Also, in terms of speed Keras was found to be superior. Training was completed 1.9 times and testing 1.4 times faster with Keras than with PyTorch. Because of this, the researchers again found Keras to be better suited to design the *VGG16* model, representing the CNN architecture.

Lastly, also for the Recurrent Neural Network a comparison was made. 25,000 reviews were used for training and the same amount again for testing, using the 5000 most frequent words in the embedding. In this scenario PyTorch performed better in terms of accuracy, with an accuracy of 87.08% compared to 85.83% achieved by Keras. The model training was completed 1.3 times faster with PyTorch than with Keras. In the testing phase, Keras was able to be 1.6 times faster than PyTorch. Since Keras was only favorable, when it came to test speed, the researchers found PyTorch to be better suited to design and train RNNs.

In summary, Keras seemed to be superior to PyTorch when it came to the design of FNNs and CNNs, both in terms of accuracy and speed. For the creation of RNNs, PyTorch provided a higher level of accuracy and was faster on training the data (KABAKUŞ 2020).

In 2021 the research team around Elshawi et al. published a benchmarking of the at this time most popular deep learning frameworks TensorFlow, MXNet, PyTorch, Theano, Chainer, and Keras. From this benchmark we will pick the data of Keras and PyTorch for our own comparison. The comparison was made with three different architectures: Convolutional Neural Networks, Faster Region-based Convolutional Neural Networks (Faster R-CNN) and Long-Short-Term Memory (LSTM). Used evaluation metrics were accuracy, training time, convergence and resource consumption patterns. Experiments were run both on CPU and GPU, analyzing different datasets. To train the CNNs, the datasets *MNIST, CIFAR-10, CIFAR-100* (CIFAR-10 and CIFAR-100 datasets 2017) and *SVHN (The Street View House Numbers (SVHN) Dataset 2011)* were used (*SVHN* only for GPU accelerated computations). For the Faster R-CNN architecture, models were trained on the *VOC2012* dataset (The PASCAL Visual Object Classes Challenge 2012 (VOC2012) 2020). Three different datasets were used to evaluate the LSTM models: *IMDB Reviews*, *Penn Treebank* (Treebank-3 - Linguistic Data Consortium 2023) and *Many things: English to Spanish* (Tab-delimited Bilingual Sentence Pairs from the Tatoeba Project (Good for Anki and Similar Flashcard Applications) 2023).

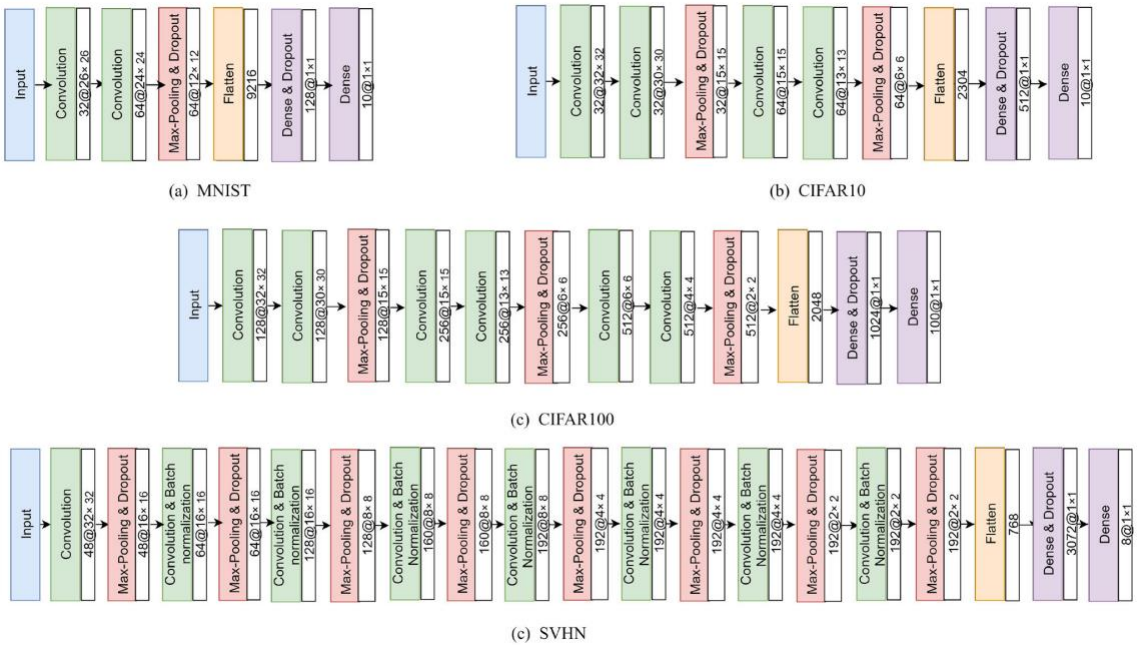

Figure 3-3. CNN Architectures for different data sets by Elshawi et al.

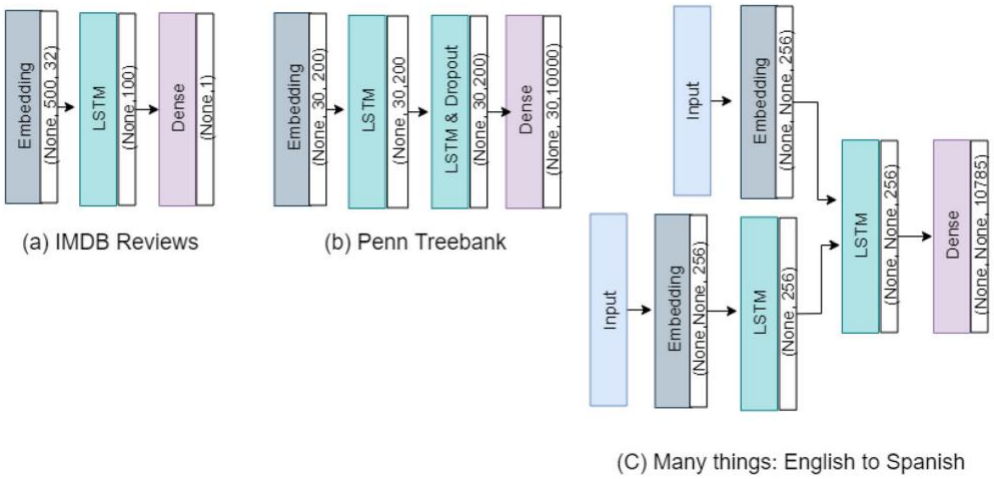

Figure 3-4. LSTM Architectures for different data sets by Elshawi et al.

The results of this paper indicate that the performance of the models heavily depends on the underlying dataset and the model type. Also, they show that the different frameworks have different abilities to use the additional computing power when a GPU is accessible. When it comes to accuracy, running on the CPU in the given setup, Keras overall achieves higher accuracy on the CNN models for the three tested datasets. On the MNIST dataset the difference is not statistically significant since both models achieve an accuracy of approximately 98%. Especially on the *CIFAR-10*, but as well on the *CIFAR-100* dataset Keras gains a significant advantage by achieving respectively 80% and 53% of accuracy, compared with 72% and 50% by PyTorch. The results with the LSTM models are closer together, as both models achieve comparable accuracy values. Only on the *Many things* dataset PyTorch performs slightly better. After testing the Faster R-CNN model is can be concluded that Keras' accuracy performance is also superior in this testing scenario. The same tendencies can be seen when both models run on a GPU.

In terms of training time on the CPU, Keras is up to 5 times faster than PyTorch when training the CNN. PyTorch trains it´s LSTM models faster on the *IMDB* and *Penn Treebank* sets but struggles training on the *Many Things* dataset. When accelerating the computation with a GPU PyTorch gains an advantage on the training time for the MNIST dataset, finishing faster than Keras. On the other datasets Keras is still faster than PyTorch, up to 4.3 times faster on the *CIFAR-10* dataset. Comparing the training times on the GPU for the LSTM it shows, that PyTorch here gains most benefits from GPU acceleration. Throughout all datasets PyTorch is faster than Keras, having the best performance on the *IMDB* dataset, which is trained 22 times faster.

The next measurement taken analyzed in this work refers to resource consumption. For the CPU training it can be observed that PyTorch's CPU usage is higher than Keras' with exception of the *CIFAR-100* and the *Many things* dataset. The opposite can be seen when comparing memory consumption. Throughout most of the datasets Keras occupies more memory than PyTorch. Exceptions were found training the Faster R-CNN on the *VOC2012* dataset and the LSTM on the *Many Things* dataset. After introducing the GPU, we see that for the CNN training the Keras models mostly have a higher usage of the GPU. On the other hand, training the LSTM models, the PyTorch framework needs more resources. Also, during the training PyTorch relies much more on the CPU when training the CNN, while it can outsource more resources of the LSTM training to the GPU. Training the Faster R-CNN, PyTorch has a higher GPU usage than Keras, but utilizes less of the CPU, Additionally, PyTorch gains a significant advantage when it comes to memory consumption. In general, Keras consumes more memory than PyTorch during the GPU accelerated training.

The last comparison between the frameworks was made for their convergence behavior. Mostly, models from both frameworks find their peak performance after a similar number of training epochs. Biggest differences could be found at the training of the LSTM models on the *Penn Treebank* and *Many Things* dataset. At the first mentioned Keras already peaks between 10 to 20 epochs, while PyTorch needs around 40. At the second one PyTorch converges way faster, already after less than 20 epochs, while it takes Keras between 60 to 80 epochs to reach its maximum accuracy level (Elshawi et al. 2021).

A comparison solely on Feed-Forward Neural Networks was made by Munjal et al., testing FNNs with different hidden layer sizes on data obtained analyzing the impedance and inductance spectra of Bi-Metallic Coins. The

compared frameworks were Keras, PyTorch, TensorFlow and CNTK. Again, we just use the data regarding Keras and PyTorch to work with. Each model constructed was implemented identically in both frameworks. All models have an input layer of 6 neurons and an output layer with 1 neuron. Also, every model had two hidden layers. In the conducted experiments the number of neurons in the hidden layers variated between 8, 16, 32 and 64. In the training process two Dropout layers with a dropout ratio of 0,3 were included. The optimizer used for weight tuning was the Adam optimizer. All frameworks use the sigmoid activation function in the output layer and trained 10 epochs each, with exception of the CNTK framework, which for out comparison has no relevance. Next to the model size, model training time, prediction time and production accuracy were evaluated.

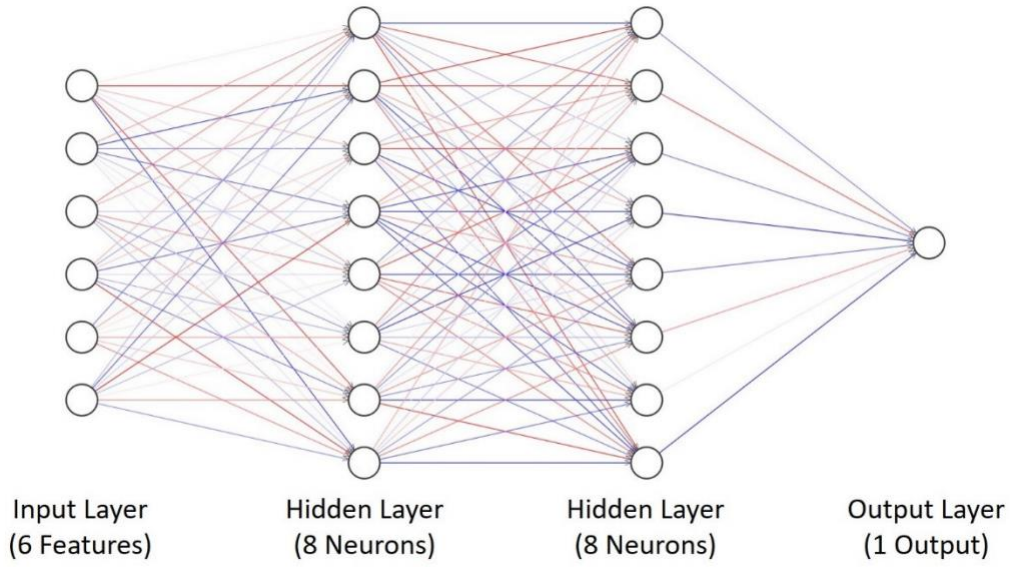

Figure 3-5. Model structure with 8 Neurons in hidden layers by Munjal et al.

Due to being a high-level API, Keras models had the biggest sizes, consuming between 32 and 53kB depending on the number of hidden neurons. PyTorch models are way smaller only needing 2,7 to 7,5kB of memory. With increasing number of neurons, the training times of every model got up. For all model sizes Keras was slightly faster taking 13,0 to 15,6ms for the training, compared to 17,3 to 22,6ms in PyTorch. Performance results came in reversed comparing the prediction times. For the models with 8 and 16 neurons PyTorch only needed 0,15 and 1,5ms to predict the results of the training data. The same models took Keras 59,4 and 65,9ms to process. Even though the prediction times for the bigger models rose significantly in PyTorch, it still outperformed Keras, taking 21,2 and 33,8ms in PyTorch and 68,5/75,7ms in Keras. To compare the model accuracy, two different types of generated data, representing two different targets, were used for classification, varying their similarity during the experiment. Therefore, all models fall to an accuracy of 50% at high similarity, as the model randomly classifies. The experiments concluded that for the 8-neuron network Keras has the lower accuracy, overall having lower accuracies on all similarity levels. On the 16-neuron model in the beginning both frameworks have a comparable accuracy at 99,8% in Keras and 99,51% in PyTorch. But with rising difficulty the PyTorch model provides more accurate results, dropping later to 50%. The same behavior can be observed in the more complex models with 32 and 64 neurons per layer. With rising difficulty the PyTorch accuracy performs better in a wider range, than the Keras model (Munjal et al. 2022).

3.3.2 Conclusion from previous works

Based on the results of the previous works, no absolute conclusion can directly be drawn to predict the performance of the model implemented in PyTorch, since in the past experiments no clear superior framework could be found. The results heavily depend on the processed data and the type of model.

The initial statement declaring the higher speed capabilities of PyTorch due to its lower-level API cannot be completely supported without reservation by the works previously intending a comparison. Considering the results of the papers previously presented in this section, the intended performance boost of the regression model must be at least partially questioned.

One common outcome which can be observed in all conducted experiments is Keras' superiority in terms of

accuracy. In our own experiments we can therefore expect a decline in terms of training accuracy compared to the implementation in Keras. This does not necessarily lead to worse results with the PyTorch implementation since previous works also pointed out a tendency of Keras in terms of overfitting to training data. A direct transition from the training to the test accuracy cannot be made. Even if an accuracy decline is to be found, we can assume it to be in the same scale as the results obtained with Keras.

The implemented regression model contains mostly convolutional elements and in the final layers implements a Feed forward network. In both types of networks, Keras was found to be faster during the training of the data. Because of this, we can expect the PyTorch model to need more time for the model training in this case as well. Since the speedup differed a lot in the analysed works, no concrete speedup difference can be predicted for our model. We can expect the Keras model to be faster than the following PyTorch version by a factor within a range of 1,9 and 5 according to previous results.

When it comes to the measurement of test speed, there is no uniform tendency regarding our selected model types. In most cases PyTorch was found to be faster than Keras during testing. Therefore, we can expect the new implementation to outperform the Keras model in this section.

In terms of memory consumption, it was clearly observed that PyTorch requires bigger memory resources, and we can deduce that the same results will be shown in our own model evaluation. To verify this, the memory consumption on the GPU is tracked in every epoch.

Deriving a supposition from the results of the analysed works, we can expect the PyTorch model and the Keras model to converge after a similar number of epochs. A slight edge for the PyTorch model can be assumed, as the models implemented with PyTorch in the experiments by Elshawi et al. (2021) converged slightly faster computing CNN models, since our model´s structure is mostly dominated by convolutional layers.

Results and predictions made in this section can also vary, because as Elshawi et al. (2021) pointed out that PyTorch is the greater beneficiary of the GPU acceleration and this can lead to better computational performance than expected. Also, how well the model can fit on the painting data can have a high influence on the obtained performance metrics.

4 DATASET AND DATA PREPARATION

4.1 Dataset

The dataset we use is the same, which was used by Delgado et al. in both published papers. In this sense, the model research is based on radiographs of 37 paintings provided by the Museo Nacional del Prado in Madrid. The selection of paintings features painters, like Rubens, Velázquez, Lorena, Swanevelt, Dughet, Poussin, Both, Lemaire, and Ribera. From those paintings, we get 240 labeled samples which are sections of the original paintings with a size of 1,5cm × 1,5cm. Those samples can be characterized with the thread density they represent, which lies in a range of 6 to 23 threads per cm. As already described in Delgado et al. (2023b), the samples come in different conditions, regarding resolution and noise.

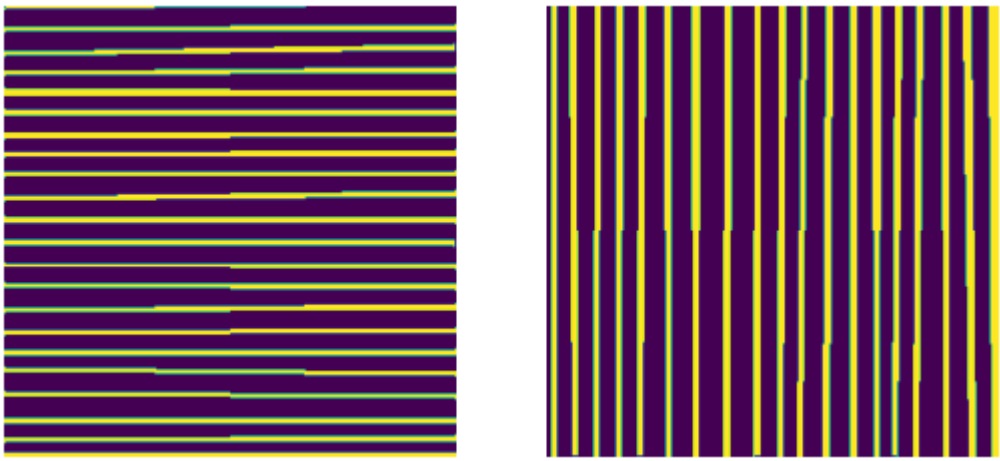

Figure 4-1. Labels for horizontal and vertical threads

4.2 Data preparation

Different steps must be taken, to make the dataset properly accessible to our regression model. This includes preparing the data for the spatial counting algorithm, which delivers the labeling for our data. Therefore, in the first step, the labeled patches, where we possess the information about the location of the horizontal and vertical threads, have to be assigned to each painting, we have data from. In the end we obtain a list for each painting, with the names of their available patches assigned. At first, the images are adjusted by creating a histogram of the grey color scale and equalizing the distribution of the color channel values, changing the contrast and especially creating higher contrast in low contrast regions. Some images appear more than ones in the index lists for training, validation and test data. If the algorithm finds to process the same painting twice, the patches are rotated 90° to increase the data availability.

Since a meaningful training can only be achieved with higher data availability, we process the available patches, to inflate the size of the training data through data augmentation. The initial patch size is 1,5cm × 1,5cm from which we obtain smaller patches with the size of 1cm × 1cm. As the data is processed in Numpy arrays, this means a change from 300 × 300 pixels to 200 × 200. Those are obtained, by selecting different image sections and rotating or flipping them around different axes. In each step, eight different sections of the patch are taken. Also, a small rotation is applied to the image in the range of 1° to 6°. This rotation allows generating training data for the model, to learn distorted structures around nails etc.

In the next step, the labels for those images have to be created, by creating a separate image of the crossing points of the patch. For this reason, the available markers for horizontal and vertical threads are placed on top of each other. The in this way created image is processed with the identical data augmentation steps, to get labels for all new created "sub-patches". Since now more data patches are available, the indices for training, validation and test data have to be adjusted. In the last step, the training data is normalized and translated into a range of 0 – 1, which makes the data easier to train in the learning process.

The actual target value of the regression model is the number of vertical threads in each 1cm × 1cm patch. This data is needed during the training and validation of the model, as well as a reference, when predicting on the test dataset. For the calculation of those numbers, the earlier described spatial counting algorithm is applied to the crossing point maps, returning one scalar value for each patch.

After the data augmentation, the model can be trained on a total of 24360 patches. Additional 4368 patches are available for validation during the training process. In the final evaluation of the best performing model, 1512 patches make up a test dataset. In total this makes 30240 patches, originals and augmented, to work with. The total data preparation process takes around 520s in the Keras script, which translates to 8min 40s, during which the spatial counting algorithms takes most of the time. Executing the preprocessing with PyTorch, the process takes on average 6s longer as, in the measurement, the translation of the data to tensors and the creation of the datasets and corresponding data loaders are included.

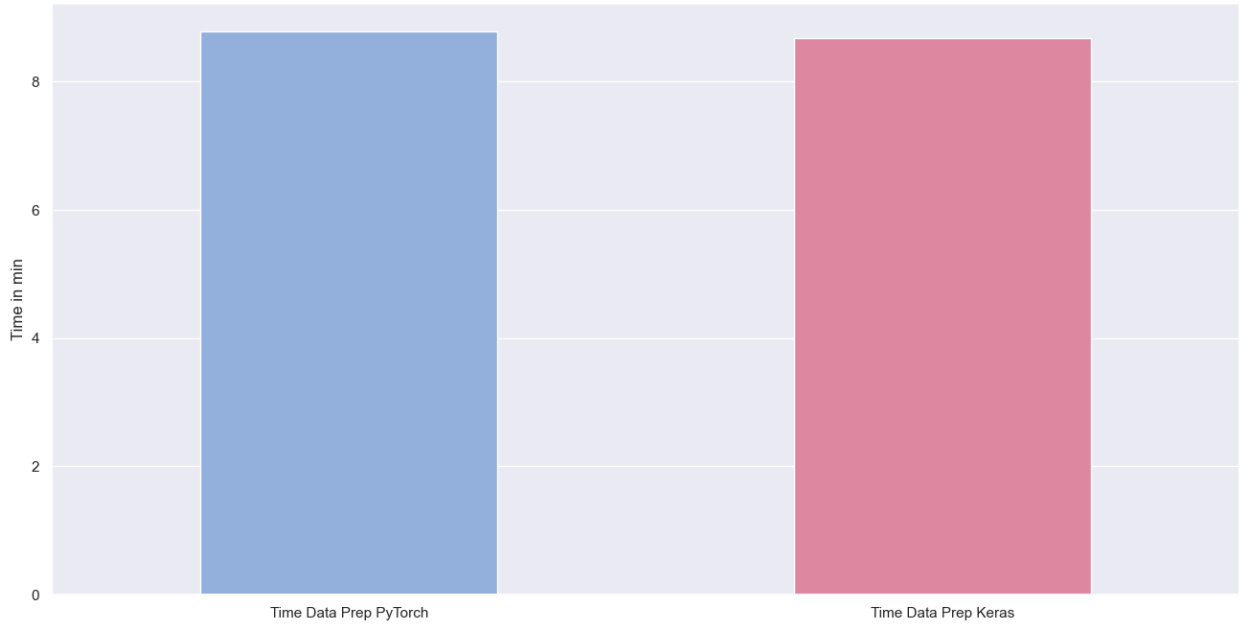

Figure 4-2. Data preparation times

The Python implementation, realising the selection of smaller patches and data augmentation, is taken from the preprocessing steps created in (Delgado et al. 2023b). In each training round, the patches are returned by the augmentation algorithm in form of Numpy arrays, together with an array of labels, after applying the spatial counting algorithm.

5 HARDWARE AND SOFTWARE DETAILS

5.1 Hardware conditions

To accelerate our computations, we rely upon the NVIDIA Tesla P100 GPU with 16GB built-in memory. It is the world's first GPU designed for AI supercomputing in data centers. The NVIDIA Pascal™ GPU architecture is the backbone of the computing power delivered by the GPU. Characteristics of this architecture are the 16 nanometer FinFET fabrication technology, in which 150 billion transistors are placed on the GPU chip. This architecture allows, together with memory optimization of the built-in memory, a performance boost and acceleration for the training of our neural network implementations (NVIDIA 2023).

5.2 Software environment

The implementation and testing of the models are performed with Python 3.9.7 in a Jupyterlab 3.6.1 environment. For the creation of the PyTorch model, an installation of PyTorch 1.13.0 is used. The set against Keras model is written with Keras-Tensorflow 2.7.0. To enable the GPU acceleration on die NVIDIA graphics card, CUDA 11.3.1 is used. Additional memory allocation is measured with the help of the *tracemalloc* module. While times are measured with the built-in Python *time* module, the measurement of the GPU memory consumption is done with Nvidia-smi 0.1.3. The results of the time managements are stored in the form of Numpy arrays, using Numpy 1.21.2. To represent and visualize the collected data, plots are created with Matplotlib 3.4.3 and Seaborn 0.12.2.

6 MODEL TRANSLATION AND TRAINING

With the knowledge gained in section 3 we can get to the model translation step itself. This chapter is separated into the presentation of the steps taken to translate the model itself, according to the previously evaluated differences between Keras and PyTorch, and then the translation of the training code. This code enables the execution and consequently the evaluation of the model. The parameters used in the model training are also presented within this chapter.

6.1 Model Setup

The model implementation created by Delgado et al. (2023b) in Keras is built by using the functional API, presented in chapter 3.2.1.1. To call the model a function called *regVGG* is called in the original implementation. Within this function all needed Keras layers are defined and linked to one another. As arguments the function takes the input shape of the tensor in form of a tuple, the number of filters/features applied in the inception module and an adjustable dropout rate. Like in every Keras model, the first layer has to be a layer of the *Input* type (Team 2023d). The layer takes the previously given input shape as an argument.

The following definition of the Inception module is excluded and declared within a separate function. This *InceptionModule2* function takes and *input* parameter, which is needed, since it represents the tensor which will be handed to the convolutional layers and connects the layers within the inception function to the layers outside of the modules. In the function three so called "towers" are defined, allowing the processing of the given input tensor by the three convolutional layers with different sized kernels. Each tower consists of three layers. The first one is the *Convolutional2D* layer (TensorFlow 2023e). In our case this layer is specified with the number of filters/features declared by the parameter in the Inception module function. Also, every convolutional layer gets the $k \times k$ kernel size parameter. Since a size reduction of the tensor is not wished after applying the convolution, the padding option *'same'* is given as an argument, to apply zero padding if needed and return a tensor with the same size as the input tensor. The kernels get initialized by selecting the *'he_normal'* option (TensorFlow 2023a). This kernel initializer takes its values from a truncated normal distribution. This normal distribution is characterized by being centered around zero and having a standard deviation of:

$$stddev = sqrt\left(\frac{2}{fan_{in}}\right)$$

$$fan_{in} \ldots number\ of\ input\ units\ in\ the\ weight\ tensor$$

Every convolutional layer is followed by a layer for Batch normalization and a ReLU activation function (TensorFlow 2023c, 2023d). The results of the three towers are concatenated in the next step along the features axis. After this step, the inception module returns the concatenated tensor for further processing in the module.

To create the setup for an identical model in PyTorch we first create our model class called *RegVGGTorch*. Here we can directly observe the main difference emphasized in chapter 3.2.1., since we go from the usage of a functional API to a class-based approach. This main class inherits the PyTorch *Module* class (Module — PyTorch 2.0 documentation 2023). The *__init__* function contains the model description and takes our design parameters. Inspired by the Keras model, the values taken are the input shape of the tensor, the number of filters

and the dropout rate. In PyTorch no input layer is required and the first layer declared is the inception module.

The inception module itself is implemented in a separate class. Since it is an independent module, it also inherits the *Module* class. As parameters we have to hand to the module the number of input channels the tensor will have, when it is given data to process, and the number of filters applied by the convolutional layers. To obtain better readability for each "tower" in the Keras model, we create a separate model class containing each convolutional kernel.

```
class InceptionModule(nn.Module):
    def __init__(self,
                 in_channels,
                 num_filters) -> None:
        super().__init__()

        self.inception_3x3 = Inception3x3(in_channels, num_filters)
        self.inception_5x5 = Inception5x5(in_channels, num_filters)
        self.inception_7x7 = Inception7x7(in_channels, num_filters)

    def forward(self, x):
        x_1 = self.inception_3x3(x)
        x_2 = self.inception_5x5(x)
        x_3 = self.inception_7x7(x)

        inception = torch.cat([x_1, x_2,x_3], dim=1)

        return inception
```

Code 6-1. Inception module

To perform the convolution we choose the *nn.Conv2D* module from PyTorch (Conv2d — PyTorch 2.0 documentation 2023). Those layers get initialized, with the handed number of input channels/features, the number of filters to apply and their kernel size, depending on the kernel the respective class is representing. As a padding option we select *'same'* and therefore the same like in Keras, to enable zero padding, which is the standard padding option in PyTorch. The according effect is the unchanged size of the processed image section. Like in the original model, the convolution is followed by a batch normalization and the ReLU activation function. For this purpose, we use the *nn.BatchNorm2d* module, which requires the number of features as parameter, the tensor returned by the convolutional layer has, as well as the *nn.ReLU* module (BatchNorm2d — PyTorch 2.0 documentation 2023; ReLU — PyTorch 2.0 documentation 2023). Every separately implemented kernel module returns its result to the inception module, in which those results are concatenated along the feature axis.

```
class Inception3x3(nn.Module):
    def __init__(self,
                 in_channels,
                 num_filters) -> None:
        super().__init__()

        self.conv_2D = nn.Conv2d(in_channels, num_filters, kernel_size=(3,3),
                                 padding='same')
        self.batch_normalization = nn.BatchNorm2d(num_filters)
        self.activation = nn.ReLU()


    def forward(self, x):
        x = self.conv_2D(x)
        x = self.batch_normalization(x)
        x = self.activation(x)

        return x
```

Code 6-2. Inception submodule with 3x3 kernel

According to the model structure, within the model one block layers is created for every level of depth in the U-Net regression model. In our case the regression model has a depth of five blocks, until we reach the flatten layer. Every block consists of two or three successive Inception modules, followed by a Max Pooling layer and Dropout layer. In Keras those layers are represented by the modules *MaxPooling2D* (TensorFlow 2023i) and *Dropout* (TensorFlow 2023h) and get substituted in the PyTorch model with *nn.MaxPool2d* and *nn.Dropout2d* (MaxPool2d — PyTorch 2.0 documentation 2023; Dropout2d — PyTorch 2.0 documentation 2023). In terms of parameters both versions are treated the same. The Max Pooling layer gets provided with the kernel size, which in our case is a 2×2 kernel. The parameter for the Dropout layer is the dropout rate given to the model instance on creation. In the first and second stage of the of the U-Net, two inception modules are used, while in the following stages three inception modules get applied one after another. Within the same depth of the U-Net model, the number of filters is the same for each inception module. In the first stage eight filters are applied and the number of filters gets doubled after every stage up to the fourth stage, where 64 filters get applied by every inception module. In the fifth stage again 64 filters are used.

```
        self.inception_1 = InceptionModule(in_channels, num_filters)
        self.inception_2 = InceptionModule(3*num_filters, num_filters)
        self.max_pooling_1 = nn.MaxPool2d((2,2))
        self.dropout_1 = nn.Dropout2d(p=dropout)
```

Code 6-3. Layer 1 of the U_Net structure

After processing the data through the stages with the inception modules, the resulting tensor is flattened. In the original Keras model, this is done by the *Flatten* layer (TensorFlow 2023b). Similar functionality can be applied in PyTorch by using the *nn.Flatten* layer (Flatten — PyTorch 2.0 documentation 2023). The usage of those layers is similar and no arguments are handed to the Flatten layers on creation. Next layers in line are the Dense layers of the Keras model, responsible for the regression itself. In our model two Dense layers are applied with 512 neurons each, followed by an output layer with one neuron, providing the counting result of the thread counting model. In the Keras framework the *Dense* module is used and is initialized with the number of desired neurons and their following activation function (TensorFlow 2023f). After each of the hidden layers, the ReLU activation function gets applied, while the output layer uses a linear activation function. PyTorchs equivalent of the *Dense* module, is the *nn.Linear* class (Linear — PyTorch 2.0 documentation 2023). As the Keras version, it gets the number of filters handed as a parameter, but additionally the number of input features needs to be precalculated and given as a parameter. The *nn.Linear* module does not include the activation function. Therefore after this layer, the *nn.ReLU* layer has to be added, to introduce the activation function.

```
        self.flatten = nn.Flatten()
        self.linear_1 = nn.Linear(round(in_side_length/2**5)**2 * 24*num_filters,
                                  64*num_filters)
        self.relu_1 = nn.ReLU()
        self.linear_2 = nn.Linear(64*num_filters, 64*num_filters)
        self.relu_2 = nn.ReLU()
        self.linear_3 = nn.Linear(64*num_filters, 1)
```

Code 6-4. Feedforward structure of the model

Since the convolutional layers in PyTorch do not have a parameter to specify the weight initialization, we have to apply this manually. For this purpose our model class gets another function called *_init_weights()*. Within the initialization function, we can call this function with the *self.apply()* command, which then applies the given function to all layers of the model. The weight initialization function checks, if the given layer is of the *nn.Conv2D* type and if this is the case, the weights of this layer get initialized with the normal distributed Kaiming initialization (torch.nn.init — PyTorch 2.0 documentation 2023). The Kaiming initialization is another name for the He initialization used in the Keras model.

```
def _init_weights(self, module):
    if isinstance(module, nn.Conv2d):
        torch.nn.init.kaiming_normal_(module.weight)
```

Code 6-5. Weight initialization function

To connect all defined layers in the Keras framework, using the functional API, every layer gets handed the output of its predecessor as an argument calling the initialized layer. The resulting output is catched with a new variable. When every layer and connection is defined, the model has to be completed by calling the Keras *Model* class and setting the previously defined input and output layer as inputs and outputs. This resulting model then is returned by the *regVGG* function and can be used in the script to process the data.

In PyTorch the connection of the previously defined layers is realized in the *forward* function of the model class. On call, the handed data is processed through the layers in the order defined in the *forward* function. The returned value of this function is data output of the model, in our case the number determined by the model to be the number of vertical threads in the image section.

In this way, the realized PyTorch model is identical to the model previously proposed by Delgado et al. (2023b) in the Keras framework. Their similarity can be proven, next to the structure, by comparing their respective number of trainable parameters. In our case, both implementations have 10.166.505 trainable parameters, which is an indication for their identicality.

6.2 Data Loader

To facilitate the data handling within the training loop we can set up a data loader for the training, validation and testing of the PyTorch model (torch.utils.data — PyTorch 2.0 documentation 2023). At the same time this allows the training of the data in batches. The *DataLoader* class provided by PyTorch needs the data to process in form of a *Dataset* class. Therefore, we implement our own *Dataset* class, containing the previously created dataset.

```
class CropsDataset(Dataset):
    def __init__(self, X, Y) -> None:
        super().__init__()
        self.X = X
        self.Y = Y

        if len(self.X) != len(self.Y):
            raise Exception("The length of X and Y do not match")

    def __len__(self):
        return len(self.X)

    def __getitem__(self, index):

        x = self.X[index]
        y = self.Y[index]

        return x,y
```

Code 6-6. Custom Dataset Class

Our class is based on the basic implementation made by Black (2020). The class implemented is named *CropsDataset* and inherits from the PyTorch *Dataset* class. To create a functional D*ataset* class, we have to overwrite the *__init__*, *__len__* and *__getitem__* functions. On initialization the *__init__* function gets the data handed in form of a tensor, which we create from a numpy array, and for every data entry the belonging label in a separate tensor. Therefore, we check if the length of the data tensor and the label tensor match and raise an error if this is not the case.

With the *__len__* function we return the number of entries in our dataset. Lastly, the *__getitem__* function returns the data entry and the corresponding label based on an index. By organizing our data in this class, the data loader can iterate the dataset and create the batches needed for training in our wished size.

Afterwards we create three different data loaders, one for each type of usage: training, validation and test. Alongside the dataset we define the batch size and activate the shuffling of our data.

```
train_loader = DataLoader(CropsDataset(
                    torch.Tensor(x_rot_train_full[train_idx][:,None,:]),
                    torch.Tensor(y_train_full[train_idx])),
                    batch_size=batch_size, shuffle=True)
```

Code 6-7. Initialization DataLoader for Training Data

The data, used to initialize the data loaders, is obtained, after executing the data augmentation and labelling steps, which were presented in section 4. Therefore, the data is handled in form of Numpy arrays and converted into tensors, before being stored within the *Dataset* class. To ensure that both models train on the same data, a seed value is given to the augmentation function. It consists out of a fixed seed number, set in the beginning of the Python script, combined with the number of the training, which is executed. Consequently, in each of the ten trainings, a slightly different dataset is returned by the augmentation algorithm, to the application of gaussian filters in the augmentation process. Before each training, a new variation of the training and validation data is created in form of a Numpy array and handed to the dataset and data loader classes. Then, the data is used for the training and validation of the model.

6.3 Early Stopping

The next functionality, which we have to realize manually is the early stopping. It allows us to stop the training

process, before we reach the previously defined number of training epochs, if no further improvement can be observed. This form of regularization helps to avoid overfitting the model to the training data (Mustafeez 2020). In Keras the *EarlyStopping* class is already defined and can be handed to the *fit* function of the Keras model as a callback property (TensorFlow 2023g). Since early stopping is not natively implemented in the PyTorch framework, we create our own class to handle this task.

The custom *EarlyStopping* class can be initialized with customizable values for the patience and the minimum delta. Patience is the number of epochs, in which consecutively no improvement must be found, until the early stopping is triggered. With the minimum delta, the minimal improvement for each epoch can be set, to be recognized as improvement by the early stopping class.

```
class EarlyStopping():
    def __init__(self, patience=20, min_delta=0) -> None:
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf

    def __call__(self, validation_loss):
        if validation_loss < (self.min_validation_loss - self.min_delta):
            self.min_validation_loss = validation_loss
            self.counter = 0

        else:
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False
```

Code 6-8. Early Stopping Class

After each epoch we must call the early stopping instance with the validation loss previously calculated. On call, the instance compares the validation loss with the minimal validation loss achieved during the training process, taking into account the minimal delta value. If an improvement is detected, the patience counter is reset, and the minimal validation loss is updated. Otherwise, the patience counter is increased by one and compared with the preset patience value. On reaching the patience level, the early stopping is triggered, interrupting the training process.

6.4 Training Setup

The training process in PyTorch must be implemented manually as well. That being the case, a training loop has to be created, in which one loop iteration represents one training epoch of the model. We design our training loop in a way, that equals the training done with the Keras framework and is backed by the information obtained in section 3.2.2.

Before entering the training loop, we must create all instances needed to execute the training process. The first and most important is the model itself and will be created, as described in the previous chapter, with its required parameters. Since the execution of the model will be done on a GPU, according to PyTorch semantics, the model must be moved to the GPU. For that reason, in case the GPU is available, we move the model to the GPU by calling *model.cuda()* (CUDA semantics — PyTorch 2.0 documentation 2023). The same step needs to be done for each data and label tensor, processed batchwise in the training loop.

```
model = torch_models.RegVGGTorch(num_filters=number_of_filters,
                                 dropout=pb_dropout)
if torch.cuda.is_available():
      model = model.cuda()
```

Code 6-9. Model initialization with CUDA

As optimizer we select the *Adam* optimizer, same as the one, used in the given Keras model. The *Adam* optimizer is implemented in the PyTorch framework and is connected to our model, by giving the model parameters as an argument on initialization (Adam — PyTorch 2.0 documentation 2023). Furthermore, we change the learning rate to the same value, the Keras model is using, in this case taking the value *1,0e-3*.

Even though the current implementation in Keras relies on a fixed learning rate and also the final comparison between the frameworks will be executed with a constant learning rate during the training process, in the former training script, a function enabling the variation of the learning rate was given. To also enable this option in the PyTorch implementation, we use the *LambdaLR* learning rate scheduler (LambdaLR — PyTorch 2.0 documentation 2023). To make use of its functionality, we define a lambda function, which returns varying results based on the number of epochs already accomplished in the training process. The function is handed to the learning rate scheduler, which is connected to the optimizer. After each epoch in the training the loop the learning rate is updated by calling *scheduler.step()*.

```
lambda_function = lambda epoch: 1/(1 + epoch/10)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,lambda_function,
            verbose=True)
```

Code 6-10. Learning Rate scheduler

The loss function to evaluate the performance of our model is the "normalized mean absolute error". Since there is no pre-implemented loss function in PyTorch, which normalizes the error on the target data, we implement a custom loss function. It takes the prediction and target data and returns the mean absolute error, where the absolute error is normalized on the value of the target data, before calculating the mean. The function must return a single value, to allow the backtracking of the optimizer.

```
def normalizedMAE_torch(y_true, y_pred):
    return torch.mean(torch.abs(y_pred - y_true)/y_true)
```

Code 6-11. Custom Loss function

To enable the early stopping functionality, we create an instance of the custom *EarlyStopping* class, implemented earlier. For the early stopping and the model selection after the training process, we need to keep track of the minimal validation loss achieved during the training, as we want to save the best performing version of the model. Consequently, we initiate the minimal valid loss with *infinity*, as a starting point for the variable. Also, we set the model into training mode, before entering the training loop for the first time.

In the beginning of each epoch, we must reset the training loss. After that, we enter a second loop, iterating the dataset batchwise. We resort to the data loader, created in the preparation step, which returns the data and label tensor in the batch size defined on creation. Within the loop, we clear the gradients of all optimized variables, by calling *optimizer.zero_grad()*. In the next step we pass the data through our model and catch the outputs in a variable, so that we can calculate the loss of this iteration with our loss function, comparing the model output with the target labels. By using the backward pass, the gradient of the loss gets calculated with respect to the model parameters. Based on those gradients, the model parameters get updated by the optimizer through calling *optimizer.step()*. Lastly, in each batch processing, we accumulate the train loss.

```
train_loss = 0.0
model.train()

# Iterate training data
for batch_idx, (data, labels) in enumerate(train_loader):
      optimizer.zero_grad()

      if torch.cuda.is_available():
            data, labels = data.cuda(), labels.cuda()

      output = model(data)

      loss = criterion(labels, output)
      loss.backward()
      optimizer.step()

      train_loss += loss.item()
```

Code 6-12. Batchwise Training loop

After processing all data in training mode, the validation data has to be looped, so that we can evaluate the progress of the training. Before, we instantiate a variable for the validation loss, on which we will accumulate the validation loss of each processed batch. On processing the validation data, the model is put into validation mode, to change the processing of the batch normalization to using the full data and deactivate the dropout layers. The then following loop is similar to the loop iterating the training data, without integrating the optimizer. Batch wise validation data is loaded to the GPU and processed by the model. The created output is compared with the labels with the help of the loss function. Both, the validation loss and the training loss, are getting saved to allow a statistical evaluation and representation.

After each validation run, the validation loss of the current run is compared to the minimal validation loss of previous epochs. If an improvement can be found, the model is saved to later have access to the model with its best performance capabilities.

A test loop after finishing the training has the same structure as the validation loop. For testing, we load the best performing model according to the evaluation during the training. Then the test data, like the validation data, is loaded batchwise from its data loader, moved to the GPU and passed through the model. A final evaluation takes place, by calculating again the loss on this partition of the data.

6.5 Performance Analysis Utilities

The actual comparison between the model implementations in PyTorch and Keras is based on the measures of achieved accuracy, training time, validation time and allocated memory during training. Next to separated measurement of training and validation time, we also measure the time per epoch and the total training time. To measure the different times, we use the python *time* module (Python documentation 2023). We obtain timestamps, by calling *time.perf_counter()* before and after each training step, we need to measure. Later we can subtract the timestamps from one another to calculate the time required to execute this step. All measurements are saved in a list, to later get the statistical analysis.

Since the model applies a regression instead of a classification, we measure the preciseness of our model by comparing the results of the loss function applied during the training and validation phase, using the mean absolute error as loss function. With those we can evaluate how well the models are able to fit the training data. By also comparing the error on the processing of the test data, we can generate a conclusion on how well the models can perform on the dataset in general.

The consumption of memory is evaluated based on two measurements. With the *tracemalloc* module, the

general allocation of memory on the RAM can be measured every training. The value we concentrate on is the maximum memory allocation during one complete training. Secondly, the consumption of memory on the GPU can be measured separately with the *nvidia-smi* module. Using this module allows the independent analysis of the behavior of the two frameworks through third party software, without the need for the use of framework specific analysis.

For PyTorch the measurement calls can be placed directly within the training loop. This way, timestamps and memory information can be directly obtained and saved. As the Keras training takes places within the *model.fit()* function call, to measure the times for training and validation steps as well as GPU memory consumption, a separate callback has to be written and handed to the training function.

While every measurement is evaluated separately, we try to draw a joined conclusion using all analyses together to select a framework, which is the better fit, to implement and process the x-ray data of the paintings.

To allow full comparability, the training parameters for both models are identical. Accordingly, both models use the same model structure, optimizer, and loss function. The optimizer is the *Adam* optimizer with a learning rate of 10^{-3}. With the normalized mean squared error as a loss function, the models were trained batchwise with a batch size of 32 inputs per cycle. The early stopping used, to prematurely end the training, has a latency of 20 epochs. If early stopping is not terminating the training process, the maximum number of epochs available for training for both models is 100 epochs.

7 PERFORMANCE RESULTS

7.1 Model performance

After a total execution time of two days, taking nearly 48 hours of calculation time, the training of the PyTorch model was completed. For the training evaluation in total ten separate trainings were executed during this period. Per Training this leads to a total training time of 4h 43min on average. The times of the individual trainings have a high variation, ranging from 3h 8min to 6h 16min. This high variation is caused by the different number of epochs, the model was trained for in each training, until reaching either the maximum number of epochs or the early stopping requirements. On average, the model was trained for 75,3 epochs, until the early stopping requirements got reached. The fastest training run finished already after 50 epochs, whereas in three trainings, the early stopping criteria never got reached and therefore all 100 training epochs were made use of. By taking the number of epochs into account, we can calculate the average duration per epoch, which leads to the result of 225,34s per epoch. This number can be confirmed by calculating the average value of the epoch time measurement during the training. A slight difference in value of 0,01s per epoch can be explained by the additional calculations regarding time measurements and the print of the outputs by the end of each epoch, which are not included in the pure epoch time measurements.

Each of the conducted trainings achieves a value within a comparable range as a result for the achieved minimal training and validation loss. Since during the training the performance of the model is evaluated on the achieved validation loss, the selection of the best performing training run is also done by comparing the validation losses. We observe that the best performance is achieved by the model in training 7, achieving a minimal validation loss of $9{,}89 \times 10^{-3}$. The best fit on the training data was achieved in training 2, resulting in a train loss of $8{,}91 \times 10^{-3}$. In training 7 the minimum value was obtained within a training of 66 epochs, while in training 2 all 100 epochs were executed. While with a growing number of epochs generally a better fit on the training data can be achieved, no such correlation can be found for the validation data, as seen in figure 7-2.

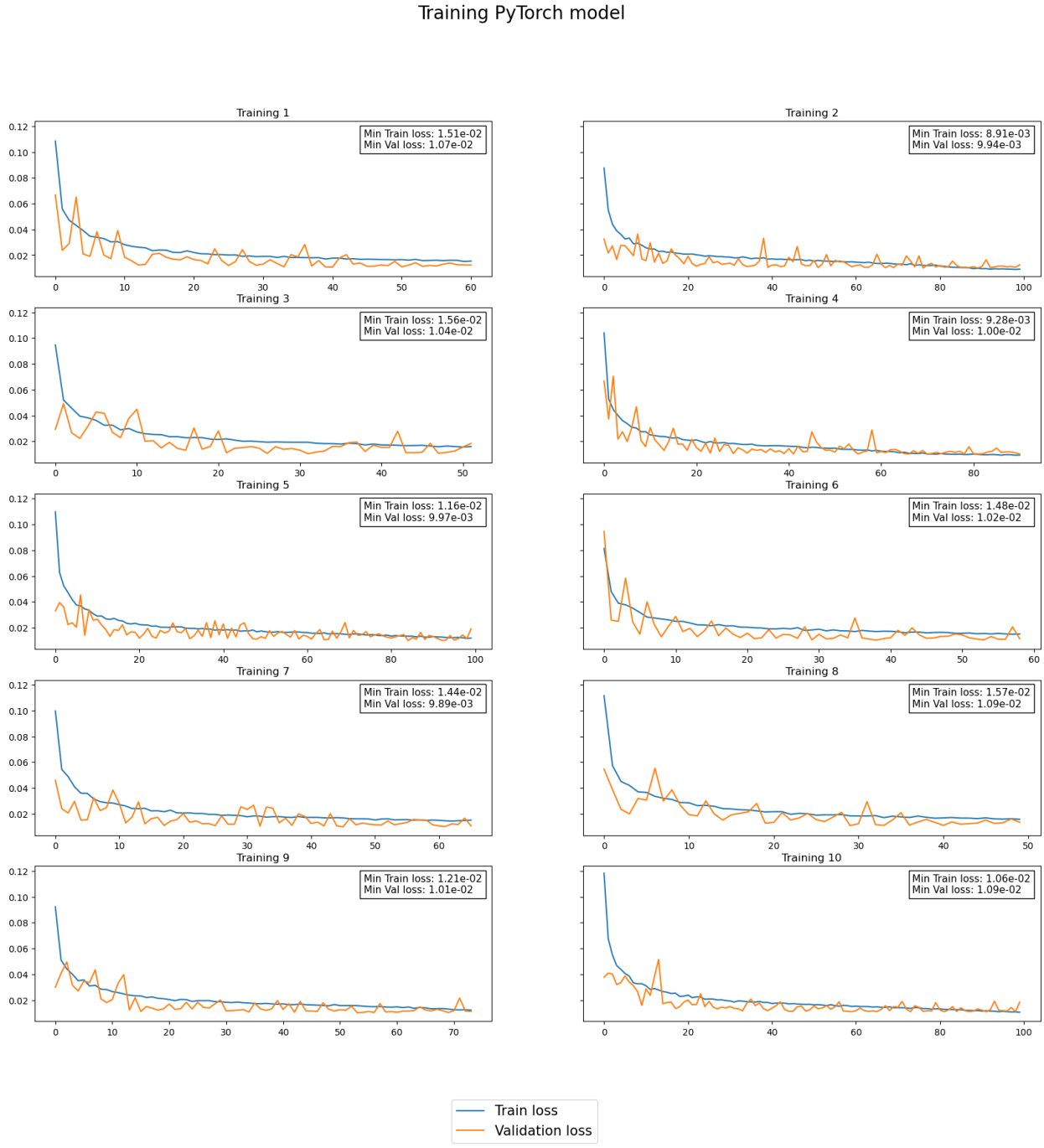

Figure 7-1. Training development PyTorch

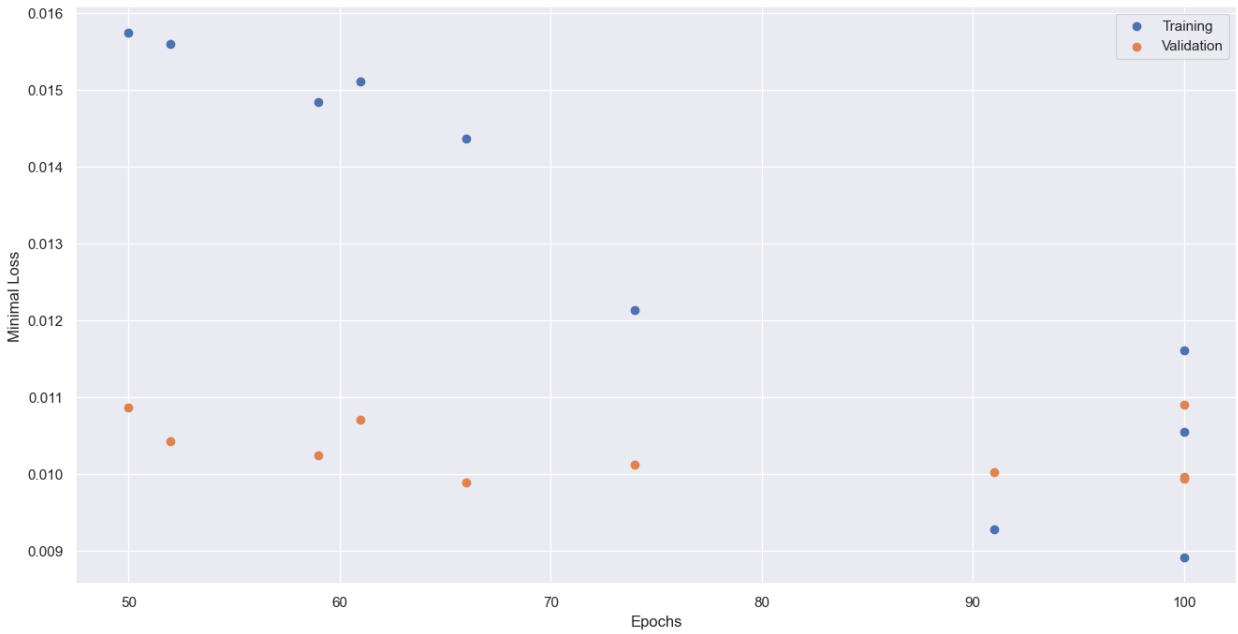

Figure 7-2. Minimal Train-/Validation loss with total training epochs

Within each epoch, which takes 225,33s, training and validation must be executed. The highest proportion is needed for the training of the model. On average, the training per epoch takes 217,61s, which is equivalent to 3min 37s and makes up 96,57% of the total epoch time. The validation is much faster and only takes 7,61s to complete on average.

To achieve this computation, the model and data has to be processed on the GPU, specified in the previous section. Next to the timings, also the memory consumption on the GPU is evaluated. The model is stored in the memory of the GPU at all times and the data is loaded batchwise into the memory in each iteration. During the training the occupancy is constantly around 4,98GB, therefore occupying 31,31% of the total GPU memory of 15,90GB ≈ 16GB.

Next to the GPU, also the RAM memory allocation was traced. It appears, that PyTorch running on the GPU with sufficient memory does not have the need for additional memory allocation during the training. Throughout all trainings, the average peak of memory allocation was only 1,62MB, which can be explained with the storage of measurement variables, to collect and calculate all timers during training.

To obtain an evaluation on the final accuracy of the model, it gets evaluated on a separate test dataset. As a model the earlier determined best model, with the lowest validation loss, is taken to process the data. The prediction of the labels for this dataset took 2,7s and resulted in prediction loss of $11{,}58 \times 10^{-3}$, which is slightly higher than the validation loss by the same model. In comparison, the training loss is even higher, as this specific model from training 7 produced a training loss of $1{,}44 \times 10^{-2}$. Those numbers hint, that the model has a good, generalized fit on the data.

7.2 Comparison to Keras implementation

As a follow-up to the training of the PyTorch model, the Keras model got executed under the same conditions and the same measurements were taken during the training to allow a one-on-one comparison of the frameworks.

The complete training cycle of ten independent trainings was completed already after 22 hours. On average, each training took 2h 12min. In comparison with PyTorch this means, that the same model can be trained in less than half of the time, it takes to train the PyTorch model. Their timings range from a maximum of 2h 43min to a minimum of 1h 23min.

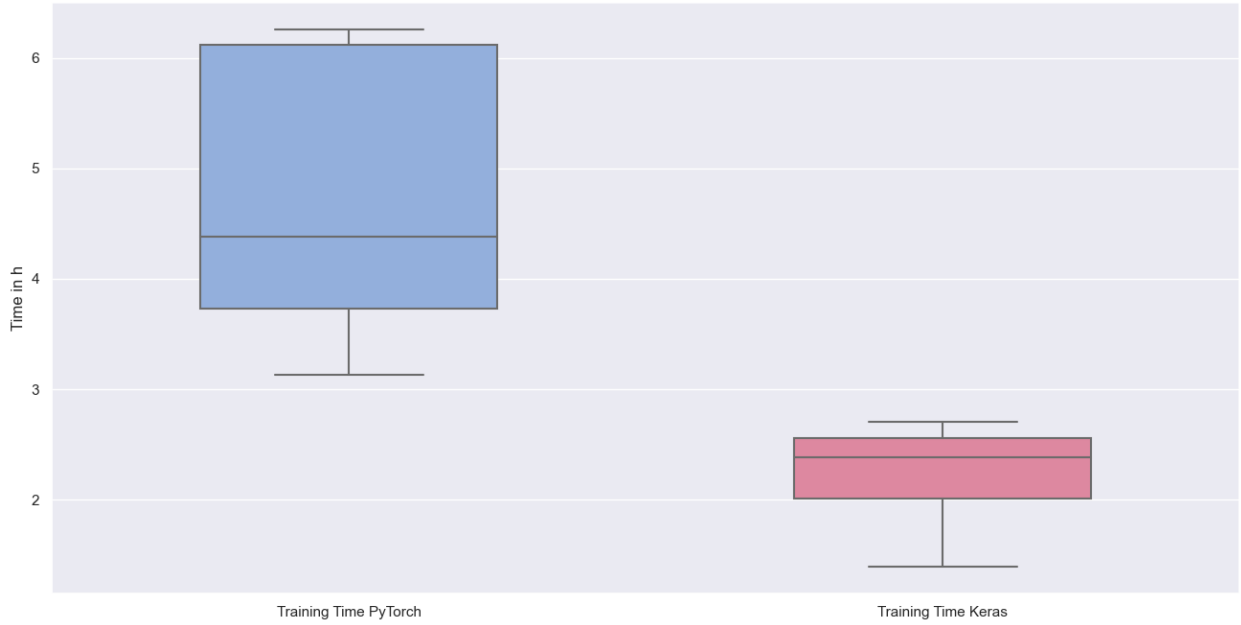

Figure 7-3. Total training times

Again, a big variation in training times can be seen, as those depend on the number of epochs trained in each session. Still, all the Keras trainings were able to finish faster than all compared PyTorch trainings. The shortest training in Keras is already finished after 38 epochs through the early stopping mechanism. The highest number of epochs was found to be 74 epochs, finishing on average after 60,1 epochs. As we can see, none of the trainings took the full available 100 training epochs, as all of them were stopped prematurely by the early stopping criteria. Consequently, Keras finishes the training on average 15 epochs earlier than PyTorch. This hints to a faster convergence of the Keras model. After fewer epochs, and therefore training time, the Keras model reaches the optimal loss.

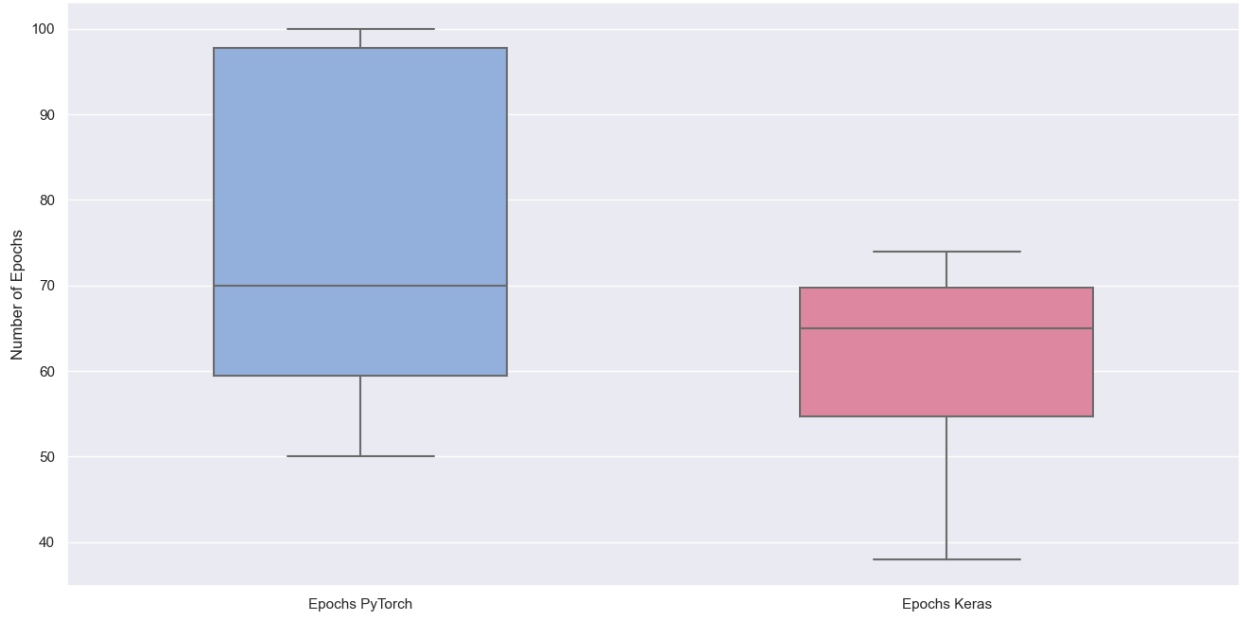

Figure 7-4. Trained epochs

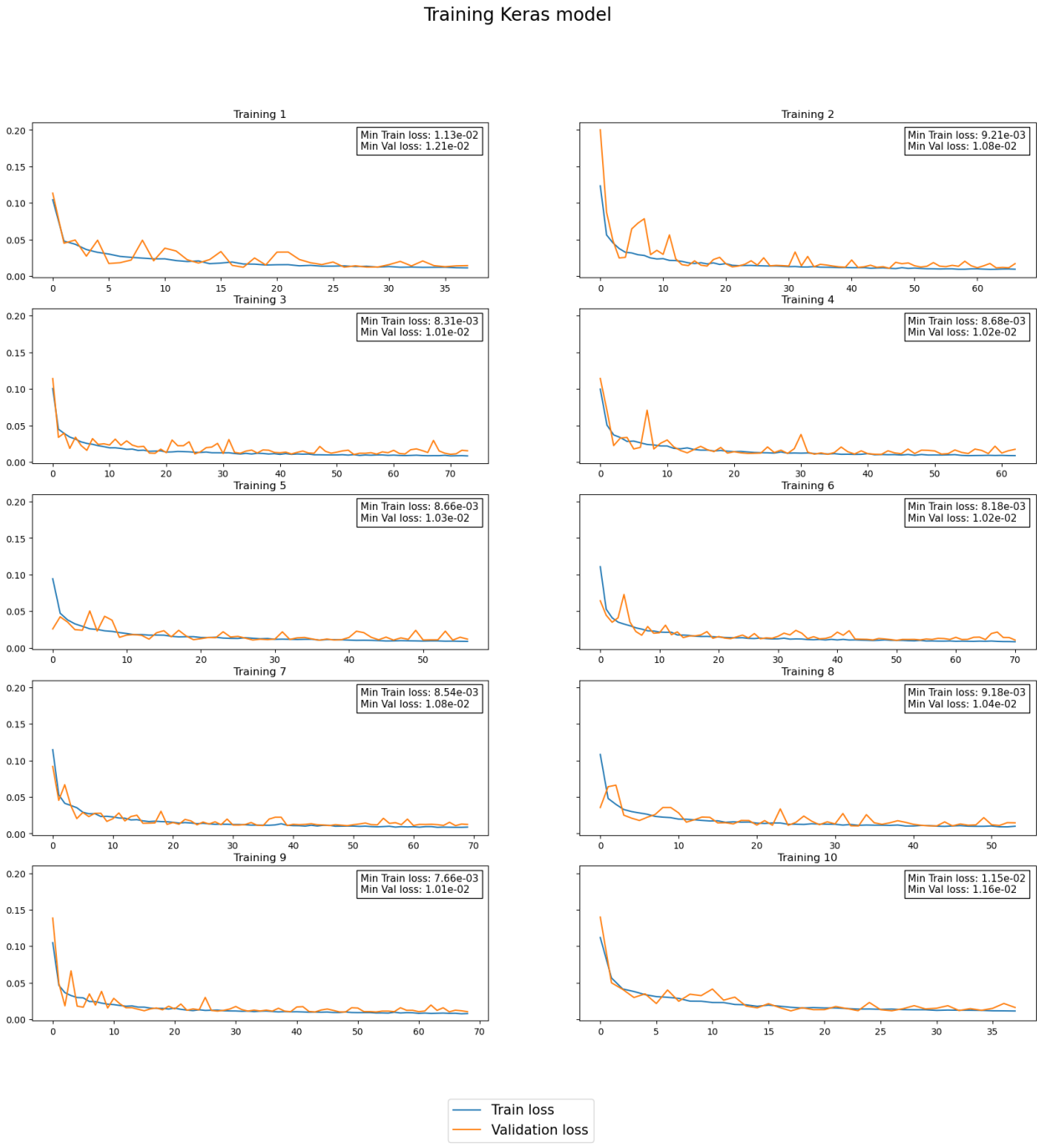

Figure 7-5. Training development Keras

Not only the number the number of epochs is decisive for the duration of the total training. The training steps, as well as the validation steps themselves can be executed faster than in PyTorch. For the time per epoch, we calculate 131,42s which translates to 2min 11s. Most of the period accounts for the training steps with 123,58s, while 6,11s are needed for the validation. The validation in Keras is only 19,7% faster and makes a smaller difference. Training steps, on the other hand, can be completed in 56,79% of the time needed with PyTorch. Those factors lead to a faster completion of every single epoch and give Keras and advantage for a faster completion of the training.

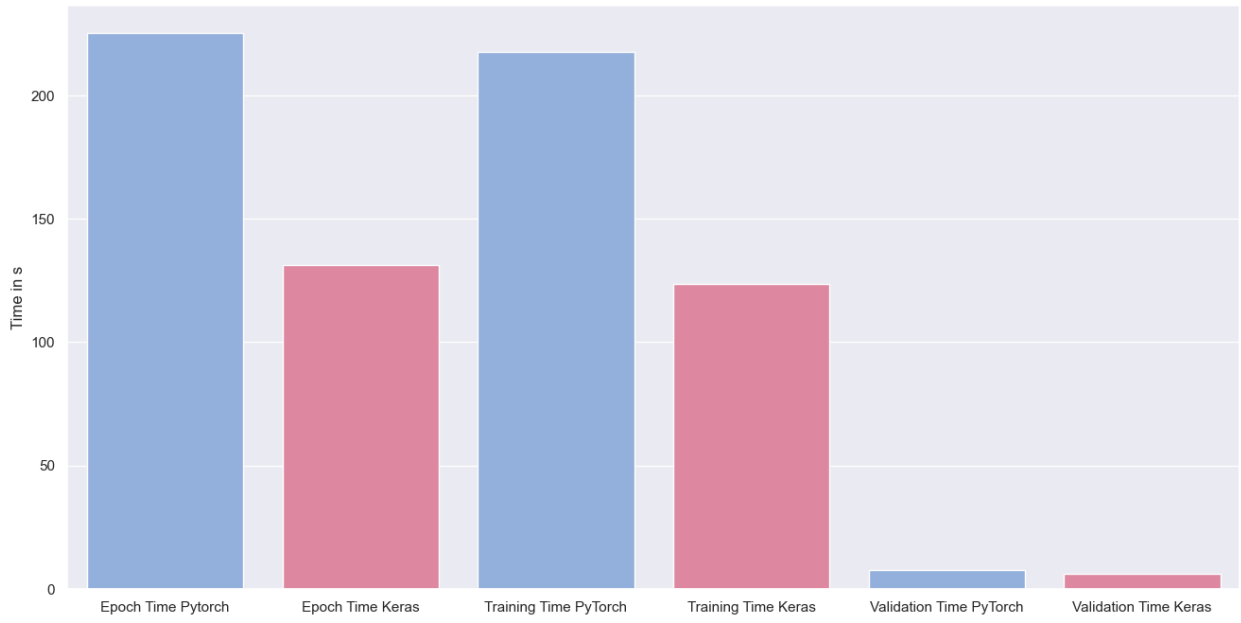

Figure 7-6. Time comparison

Even though Keras has a significant advantage in terms of training time, its performance in terms of accuracy does not significantly suffer from the faster execution and smaller number of calculated epochs. As to be seen in figure 7-7, the minimal training loss in most of the trainings is even smaller than the training loss achieved in PyTorch trainings. Occasionally, PyTorch trainings achieve a better fit on the training data than Keras, but the median loss is in favour of Keras, as is the distribution of values around it. PyTorch gains a small edge on the validation loss, but all the differences are very small and do not have a big impact on the prediction outcome of the respective models.

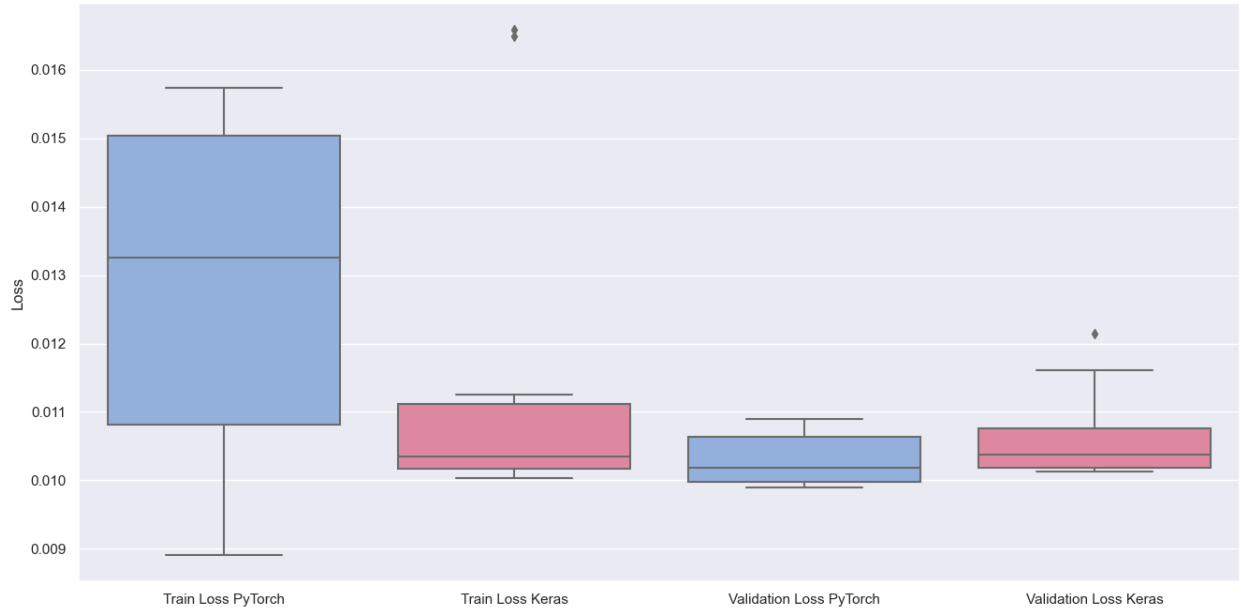

Figure 7-7. Training and validation loss

Like the PyTorch model, also the implementation in the Keras framework got executed on the same GPU. Both frameworks utilize the GPU to speed up their calculations. While PyTorch only used roughly one third of the GPUs memory, Keras uses nearly all the available memory on the GPU. During training the model and tensors occupy 15.67GB on the GPU, which translates to 98,59% of the total available memory.

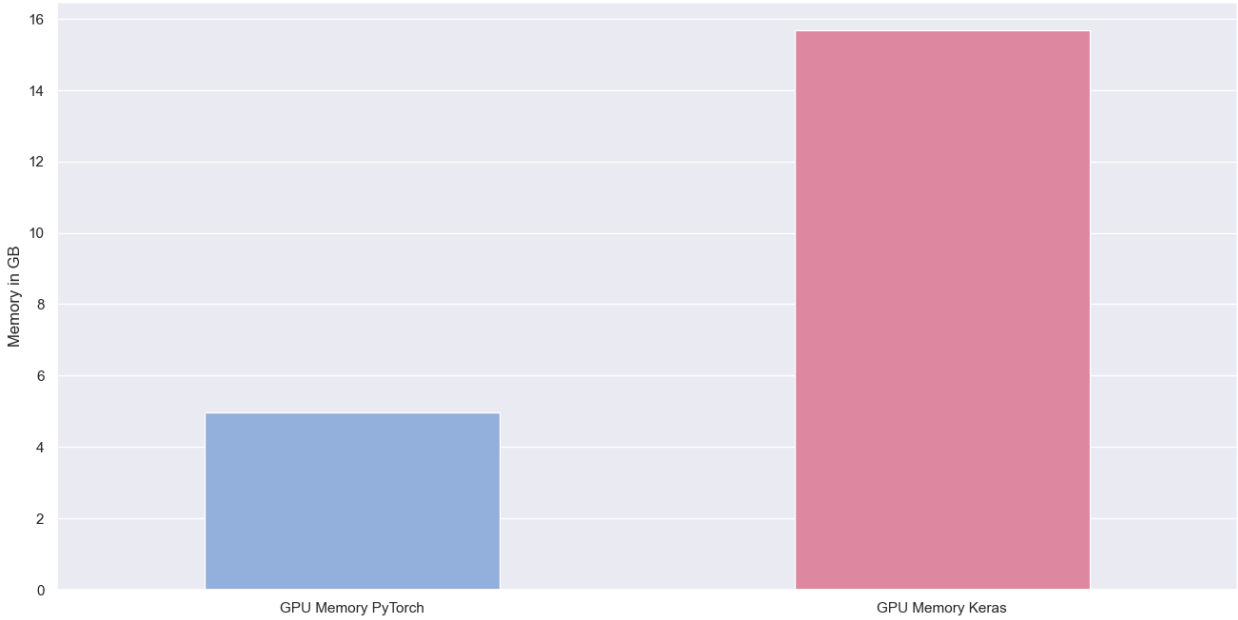

Figure 7-8. Occupied GPU memory

Apart from the GPU memory, the PyTorch model did not occupy additional memory in the RAM, which we were able to measure. This behaviour is not transferable to the Keras model. In each training, the Keras model allocated additionally 12,19GB of memory at its peak to execute its computations. This memory consumption adds up to the already needed 15,67GB on the GPU. Combined, the maximum memory consumption of the Keras model is 27,86GB compared to 4,98GB of the PyTorch model. This results in a memory consumption 5,6-times higher than needed to train the model in PyTorch.

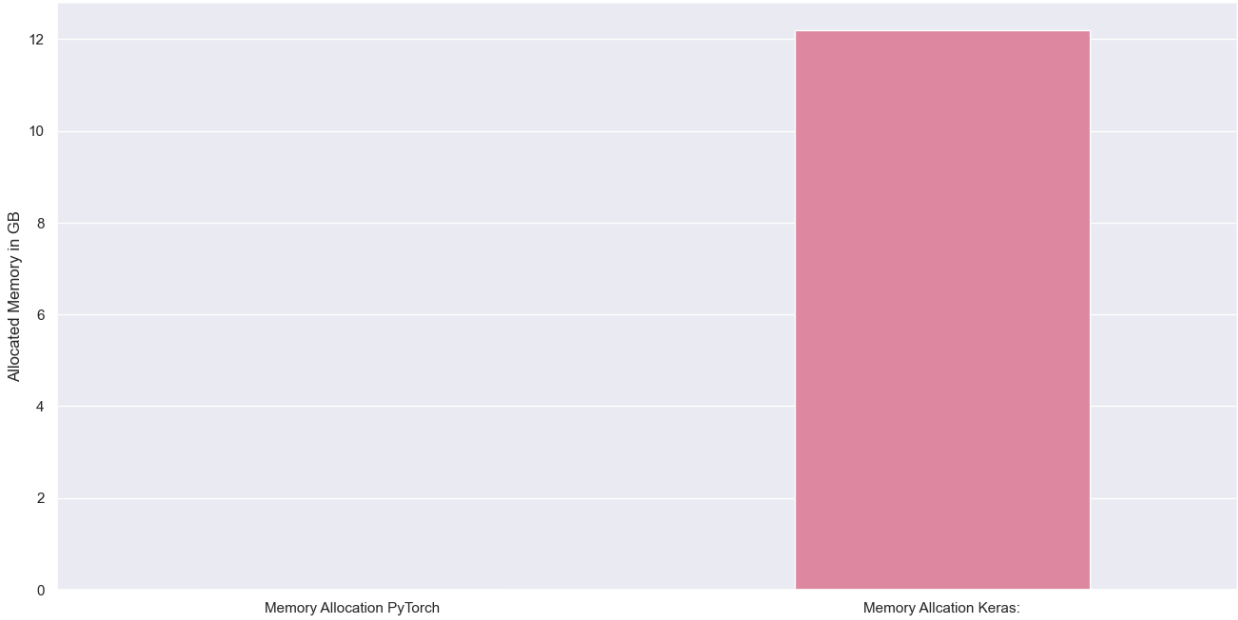

Figure 7-9. Memory allocation

To compare the performance of the Keras model on the test dataset, the best Keras model was selected, based on the achieved validation loss. In this case, the best results were achieved in training 9 with a validation loss of $10{,}12 \times 10^{-3}$. The minimal achieved training loss attained by this model is $10{,}04 \times 10^{-3}$. The validation loss in that case is just insignificantly worse than the one achieved by the PyTorch model. Both were tested on the same test dataset and their prediction time was measured. Keras finished the prediction after 2,91s and attained a test loss of $10{,}89 \times 10^{-3}$. Compared to the PyTorch implementation, the prediction is slightly slower, but results in a slightly better prediction, even though having the slightly worse validation loss.

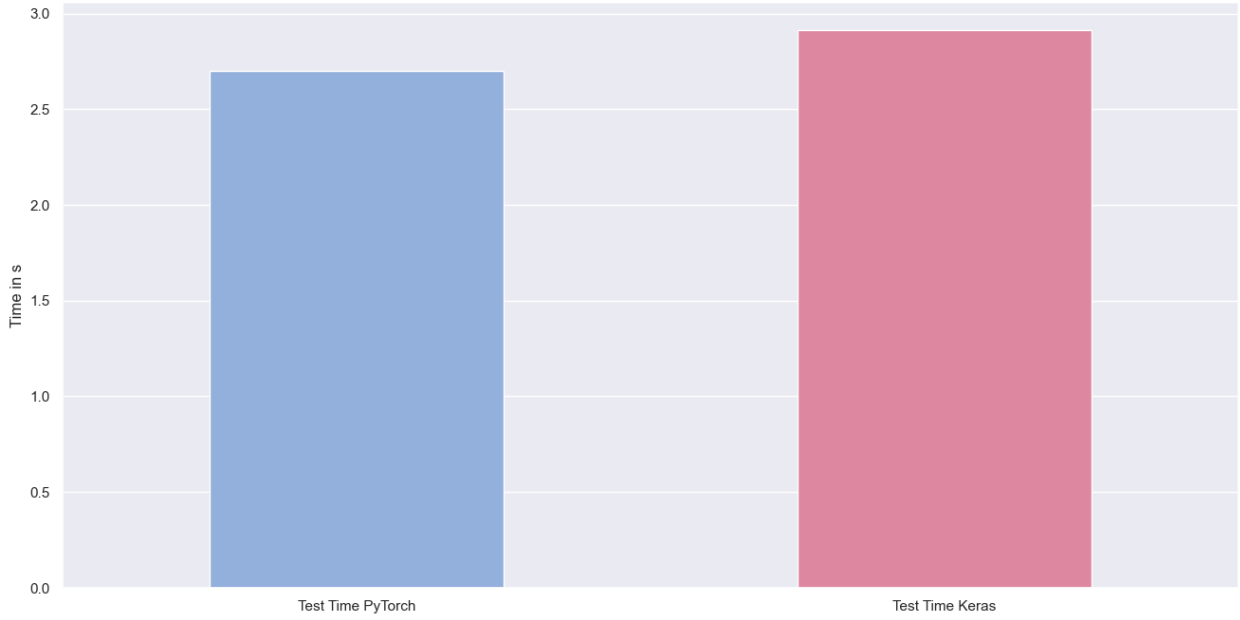

Figure 7-10. Test time

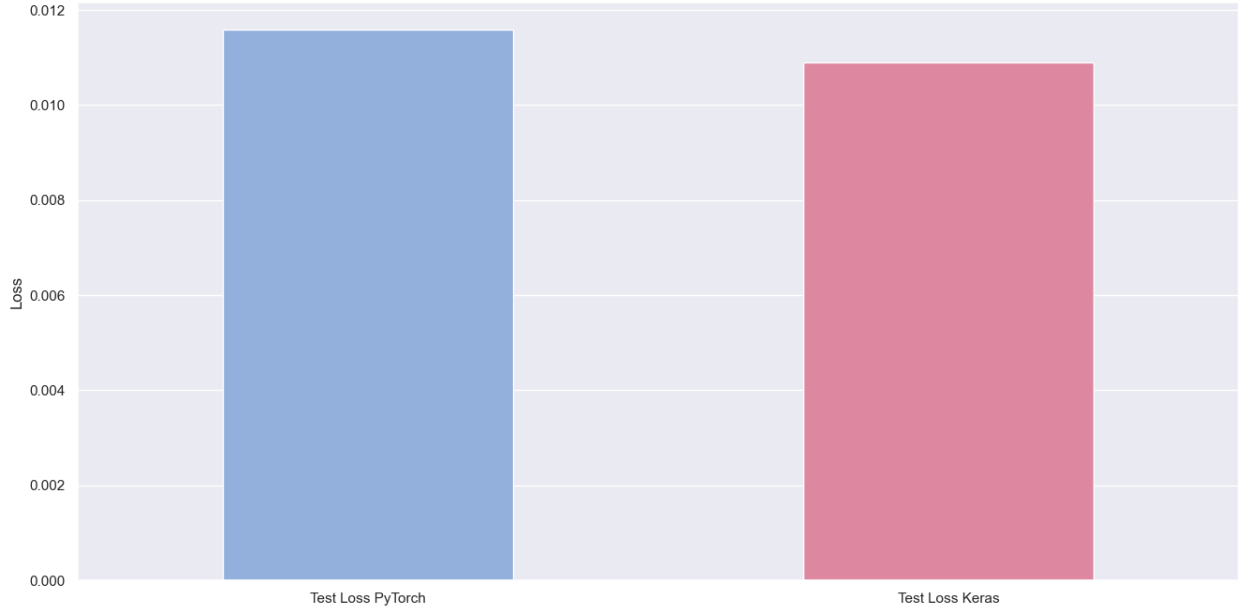

Figure 7-11. Test loss

7.3 Conclusion from framework comparison

The three main evaluation metrics, which have been covered in the previous section, are timers, memory consumption and accuracy. While those were solely compared numerically, in this section, the obtained results are related to the results found in other experiments, which compare the performance of different frameworks implementing identical deep neural networks. For this purpose, the measurements taken with the regression model are compared to the predictions made in section 3.3.2, regarding a convolutional model with a feed-forward section in the end for regression. A summary of the key metrics achieved on our data can be found in table 7-1.

Table 7-1. Performance comparison

	PyTorch	Keras
Average Training Time	4,71 h	2,20 h
Average Training Epochs	75,3	60,1
Time per epoch	225,33 s	131,42 s
Time per training step	217,61 s	123,58 s
Time per validation step	7,61 s	6,11 s
Prediction time (on test set)	2,7 s	2,91 s
GPU memory consumption	4,98 GB (31,3%)	15,67 GB (98,59%)
Allocated memory	1,62 MB	12,19 GB
Minimal train loss	$8{,}91 \times 10^{-3}$	$7{,}66 \times 10^{-3}$
Minimal validation loss	$9{,}89 \times 10^{-3}$	$10{,}13 \times 10^{-3}$
Test loss (with best performing model)	$11{,}58 \times 10^{-3}$	$10{,}89 \times 10^{-3}$

At first, the much faster training time of the Keras model is to note, completing the training and therefore also every epoch around 1,7 times faster, than the PyTorch model. The validation, while being faster, does not display such a grave slowdown, as validation in Keras is only around 1,25 times faster. This outcome supports the knowledge and understanding gained in the pre-examinations, where also the Keras framework was found to be faster in training CNN models. Even though a slowdown of training speed is traced in the PyTorch model, the expectations of a slowdown are not met. In the pre-examinations, a prediction for the Keras model was made, expecting it to be faster in a range of 1,9 to 5 times faster than the PyTorch implementation. Only being close to the range, the slowdown of the PyTorch model is not as grave as anticipated, while still showing the right tendency.

Next to the training speed, also a prediction on the outcome for pure prediction speed was made, so that already in section 3.3.2, an edge of the PyTorch model was assumed. While the difference of the timers is marginal, even on the smaller test set, the faster prediction capabilities of the PyTorch model can be seen in the conducted experiment, proving the statements made previously.

Evaluating the accuracy of both models, it can be observed that the values of training, validation and test loss are all to be found within the same order of magnitude. Also, the results achieved by both models are very close together and do not show many differences. The best model in terms of training loss comes from the Keras implementation, getting the best fit on the training data. At the same time, the best performing Keras model achieved the better accuracy on the test dataset. This test accuracy can be achieved even though, after the model selection, the PyTorch model has the better validation accuracy. Since PyTorch only tops the accuracy in this metric, the better fit on the validation data could be a coincidence due to the composition of the validation set. Although PyTorch needs much longer for its training, it cannot take benefits from this additional computing time. In more than double the average training time, the model is not able to generate results with better accuracy, but only close and in certain metrics still worse results. The better fit on the training data by Keras at some point could also be interpreted as overfitting, but by introducing mechanisms, like the early stopping, it is observable on the validation and test data, that no overfitting is to be found, but generalized results are achieved.

Neither in previous works nor in the results of the comparison made in this work, a superiority of PyTorch in terms of accuracy could be found. All works have in common that they show Keras better capability to find a good fit to the training data, when it comes to the processing of CNN networks. In some cases, this fit on the training data is interpreted as overfitting, but is not the case in this comparison, as proven with the performance on train and test dataset.

Not only does Keras achieve nearly identical accuracy results in shorter training periods, but in this case, it converges faster to its loss function minimum. By finishing on average 15 epochs earlier, the result is contrary to the prediction made in the pre-examinations, where a faster convergence of PyTorch on CNN networks was predicted.

While the achieved accuracy of both models is similar, a clear tendency can be found evaluating the memory consumption metrics. The PyTorch model only consumes one third of the GPU memory, Keras is using. Keras

is utilizing the full GPU memory available and consequently, PyTorch only consumes one third of the total available GPU memory. In addition to the GPU memory, Keras allocates up to 12,19 GB of memory at a time. PyTorch, on the other hand, does not need further memory which needs to be allocated outside of the GPU. Possible causes for Keras' high memory consumption can be a regular swap of tensors from the GPU to memory, since the full model does not fit on the GPU together with the data. Even though both networks are the same, an explanation of the higher memory needs can be Keras' nature of a higher-level API, which creates a memory overhead.

The results of this experiment yield the contrary to the expectations formulated in the pre-examinations. Since PyTorch was found to have a high memory utilization, also in this experiment, the higher memory consumption was anticipated. Adding together GPU utilization and maximal memory allocation, this leads to a consumption 5,6 times as high by Keras. In this case, having Keras as the more memory intensive framework, fits instinctively better to its higher-level API nature. With this knowledge, for the processing of the patch data, PyTorch can be seen as better suited on systems with less memory availability.

For comparability, both models were trained with the same hyperparameters. Considering the possibility of an adjustment, the PyTorch training can be optimized due to its smaller memory consumption. By increasing the batch size, the memory consumption increases but allows a speedup of the training process at the same time. With this change, an edge for PyTorch can be created and compensate the currently worse performance in the time consumption metrics.

On the present setup with the given hard- and software availability, Keras is the better fit for the processing and training of the radiograph data using the current training parameters. It is faster in the training and even achieves better accuracy on the training and test dataset. PyTorch has a slight advantage in terms of prediction speed, but which is marginal compared to the speedup during training achieved by Keras. Next to the faster training per epoch, Keras also shows the better convergence behaviour. The current hardware setup, can deal with the high memory needs Keras demands, making it the overall better choice for this setup and data.

8 CONCLUSION

In this work, we presented a PyTorch implementation of the Inception Regression model, based on the *VGG16* – structure. Before implementing the own model, the results previously published regarding the automatic thread counting analysis were revised.

The parallels in the code implementation for Keras and PyTorch were identified, since the previous Keras model implantation relies on the functional API, which is available for Keras, but lacks in the library of the PyTorch API. Because of this, the model had to be translated to an object-orientated structure. We learned about the needed adjustments in between the training loop of PyTorch and the model training call in Keras and implemented the additional utilities to enable early stopping functionality, the use of a custom loss function and the setup for the batchwise training, which needs changes in the training loop and the organization of the data in the form of data loaders.

This gained knowledge was used, together with a look at the dataset, to construct the new model implementation. Following those understandings, an exact replication of the network structure in the PyTorch framework was made possible. In addition to the model and training structure, measurement functionalities were implemented to evaluate timers for training and validation, memory consumption and accuracy.

Before the evaluation of our own model, an analysis of potential performance improvements or performance declines was made. Those assumptions are based on the outcomes of previously conducted framework comparisons. The analysis showed that, using the PyTorch framework, longer training times have to be expected. Also, the accuracy of the model can suffer from this framework change. Based on the pre-examinations, PyTorch was also found to be the more memory intensive framework. As previous studies showed, PyTorch can benefit more from GPU acceleration, which let open the possibility, that it may perform better in the setup. Biggest influence on the superiority of one framework or the other were the type of network implemented and dataset worked on. While Keras performed better on FFNs or CNNs, PyTorch thrived on LSTM implementations.

To evaluate the two frameworks, each network got trained ten times in separate training runs, storing the best performing model according to the achieved validation loss in each training. Comparability was assured using the identical hyperparameters for each model and in each training run. After picking the best performing model out of those ten, it is tested another time on a separate test set. Measurements of all ten trainings are considered, either by observing their median and range or calculating the mean.

Our own measurements in parts match those findings. Using the PyTorch framework, training per epoch took around 1,7 times longer than with Keras and validation times extended by a factor of around 1,25. As Keras in our case converges faster to its optimal validation loss and consequently finishes trainings on average 15 epochs earlier, in the end the total training times are typically more than twice as long. A small improvement through the framework change can be seen on the prediction timers. The results on accuracy do not differ a lot from the Keras achievements. Training, validation and test loss are within the same range, while Keras achieves slightly better performance on the training and test loss, the model with the smallest validation loss comes from the PyTorch implementation.

In the memory comparison, our own results heavily differ from previous findings. Although PyTorch was said to be the more memory intensive framework, the opposite is the case in this experiment. The compared memory

metric relies on the measurements of the utilized GPU memory and the tracing of allocated memory. During the training, PyTorch only utilized 31,31% of the available GPU memory and did not allocate additional memory. Keras, on the other side, on average made use of 98,59% of the available 16GB of GPU memory and additionally allocated 12,19GB of memory at its peak value. Combining those measurements, Keras needs 5,6 times more memory, than the same model implemented in the PyTorch framework.

Based on those measurements, the preferable model for this type of network architecture, together with the structure of the data, was searched. Finally, Keras was found to be the superior framework for this case, due to its faster training times and equal achievements in terms of accuracy, which in terms of terms of training and test loss were even slightly better. In most of the training runs Keras models got a better fit on the training data. Even though the memory comparison was in favor of the PyTorch framework, for the decision the setup we worked on was also considered. The setup is able to handle Keras' higher memory consumption, while achieving better results on the other metric. Because of this, Keras is chosen as the preferred framework and unfortunately no overall improvement can be achieved through the framework change. PyTorch would be the recommendation for setups with less memory available, since its consumption is much lower, while working with the data.

For future works, PyTorch's low memory consumption can offer room for improvement. The memory availability gives the option to increase the batch size, which can speed up the training. This change could compensate for the current training time deficits. Therefore, we propose to further test the PyTorch model in different training setups, with the help of a hyperparameter optimization. Possibly, with different training parameters, a PyTorch model can be created, which exceeds the performance achieved by the Keras model.

REFERENCES

Adam — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, updated on 6/2/2023, checked on 6/2/2023.

Agarwal, Rahul (2019): Moving from Keras to Pytorch - Towards Data Science. In *Towards Data Science*, 5/28/2019. Available online at https://towardsdatascience.com/moving-from-keras-to-pytorch-f0d4fff4ce79, checked on 5/5/2023.

BatchNorm2d — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html#torch.nn.BatchNorm2d, updated on 5/29/2023, checked on 5/29/2023.

Black, Sam (2020): Generating batch data for PyTorch - Towards Data Science. In *Towards Data Science*, 11/16/2020. Available online at https://towardsdatascience.com/generating-batch-data-for-pytorch-7435b1a02e21, checked on 6/2/2023.

Building Models with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation (2023). Available online at https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html, updated on 5/5/2023, checked on 5/5/2023.

Burnham, Dorothy K. (1980): Warp and weft. A textile terminology. Toronto: Royal Ontario Museum.

Chng, Zhe Ming (2022): Three Ways to Build Machine Learning Models in Keras. In *Machine Learning Mastery*, 6/30/2022. Available online at https://machinelearningmastery.com/three-ways-to-build-machine-learning-models-in-keras/, checked on 5/5/2023.

CIFAR-10 and CIFAR-100 datasets (2017). Available online at https://www.cs.toronto.edu/~kriz/cifar.html, updated on 4/8/2017, checked on 5/10/2023.

Conv2d — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d, updated on 5/29/2023, checked on 5/29/2023.

CUDA semantics — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/notes/cuda.html, updated on 6/2/2023, checked on 6/2/2023.

Delgado, A.; Alba-Carcelén, L.; Murillo-Fuentes, J. J. (2023a): Crossing Points Detection in Plain Weave for Old Paintings with Deep Learning. Available online at https://arxiv.org/pdf/2302.11924.

Delgado, A.; Murillo-Fuentes, Juan J.; Alba-Carcelen, Laura (2023b): Thread Counting in Plain Weave for Old Paintings Using Semi-Supervised Regression Deep Learning Models. Available online at https://arxiv.org/pdf/2303.15999.

Dropout2d — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Dropout2d.html#torch.nn.Dropout2d, updated on 5/31/2023, checked on 5/31/2023.

Elshawi, Radwa; Wahab, Abdul; Barnawi, Ahmed; Sakr, Sherif (2021): DLBench: a comprehensive experimental evaluation of deep learning frameworks. In *Cluster Comput* 24 (3), pp. 2017–2038. DOI: 10.1007/s10586-021-03240-4.

Encyclopedia Britannica (2023): Plain weave | textile. Available online at https://www.britannica.com/technology/plain-weave, updated on 5/4/2023, checked on 5/4/2023.

Flatten — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html#torch.nn.Flatten, updated on 5/31/2023, checked on 5/31/2023.

Hendriks, Laura; Hajdas, Irka; Ferreira, Ester S. B.; Scherrer, Nadim C.; Zumbühl, Stefan; Smith, Gregory D. et al. (2019): Uncovering modern paint forgeries by radiocarbon dating. In *Proceedings of the National*

Academy of Sciences of the United States of America 116 (27), pp. 13210–13214. DOI: 10.1073/pnas.1901540116.

Introducing ChatGPT (2023). Available online at https://openai.com/blog/chatgpt, updated on 7/1/2023, checked on 7/1/2023.

KABAKUŞ, Abdullah Talha (2020): A Comparison of the State-of-the-Art Deep Learning Platforms: An Experimental Study. In *SAUCIS* 3 (3), pp. 169–182. DOI: 10.35377/saucis.03.03.776573.

Keras vs PyTorch (2020). In *GeeksforGeeks*, 2/4/2020. Available online at https://www.geeksforgeeks.org/keras-vs-pytorch/, checked on 5/4/2023.

Kim, Seongsoo; Wimmer, Hayden; Kim, Jongyeop (2022): Analysis of Deep Learning Libraries: Keras, PyTorch, and MXnet. In : 2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA). May 25-27, 2022, Las Vegas, USA. 2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA). Las Vegas, NV, USA, 5/25/2022 - 5/27/2022. Piscataway, NJ: IEEE, pp. 54–62.

Koech, Kiprono Elijah (2020): Cross-Entropy Loss Function - Towards Data Science. In *Towards Data Science*, 10/2/2020. Available online at https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e, checked on 5/9/2023.

Korstanje, Joos (2021): The F1 score | Towards Data Science. In *Towards Data Science*, 8/31/2021. Available online at https://towardsdatascience.com/the-f1-score-bec2bbc38aa6, checked on 5/9/2023.

Kurama, Vihar (2021): The 15 Popular Deep Learning Frameworks for 2022. In *Paperspace Blog*, 11/9/2021. Available online at https://blog.paperspace.com/15-deep-learning-frameworks/, checked on 4/25/2023.

LambdaLR — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.LambdaLR.html#torch.optim.lr_scheduler.LambdaLR, updated on 6/13/2023, checked on 6/13/2023.

Linear — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear, updated on 5/31/2023, checked on 5/31/2023.

MaxPool2d — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d, updated on 5/31/2023, checked on 5/31/2023.

MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges (2013). Available online at http://yann.lecun.com/exdb/mnist/, updated on 5/14/2013, checked on 5/9/2023.

Module — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Module.html, updated on 5/29/2023, checked on 5/29/2023.

Moltzau, Alex (2020): PyTorch Governance and History - Alex Moltzau - Medium. In *Medium*, 8/8/2020. Available online at https://alexmoltzau.medium.com/pytorch-governance-and-history-2e5889b79dc1, checked on 4/23/2023.

Munjal, Rohan; Arif, Sohaib; Wendler, Frank; Kanoun, Olfa (2022): Comparative Study of Machine-Learning Frameworks for the Elaboration of Feed-Forward Neural Networks by Varying the Complexity of Impedimetric Datasets Synthesized Using Eddy Current Sensors for the Characterization of Bi-Metallic Coins. In *Sensors* 22 (4), p. 1312. DOI: 10.3390/s22041312.

Mustafeez, Anusheh Zohair (2020): What is early stopping? In *Educative*, 7/31/2020. Available online at https://www.educative.io/answers/what-is-early-stopping, checked on 6/2/2023.

N, Lakshmipathi (2019): IMDB Dataset of 50K Movie Reviews. Available online at https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews, updated on 3/9/2019, checked on 5/10/2023.

NVIDIA (2023): NVIDIA Tesla P100: der fortschrittlichste Grafikprozessor für Rechenzentren. Available online at https://www.nvidia.com/de-de/data-center/tesla-p100/, updated on 6/21/2023, checked on 6/22/2023.

Python documentation (2023): time — Time access and conversions. Available online at https://docs.python.org/3/library/time.html, updated on 6/4/2023, checked on 6/4/2023.

Red Hat Developer (2022): Build, train, and run your PyTorch model | Red Hat Developer. Available online at https://developers.redhat.com/learn/openshift-data-science/how-create-pytorch-model/build-train-and-run-your-pytorch-model, updated on 10/10/2022, checked on 5/5/2023.

ReLU — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html#torch.nn.ReLU, updated on 5/29/2023, checked on 5/29/2023.

S Anila; K Sheela Sobana; B Saranya (2018): Fabric Texture Analysis and Weave Pattern Recognition by Intelligent Processing. Available online at https://www.researchgate.net/publication/327209131_Fabric_Texture_Analysis_and_Weave_Pattern_Recognition_by_Intelligent_Processing.

Sequential — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html, updated on 5/5/2023, checked on 5/5/2023.

Tab-delimited Bilingual Sentence Pairs from the Tatoeba Project (Good for Anki and Similar Flashcard Applications) (2023). Available online at https://www.manythings.org/anki/, updated on 4/12/2023, checked on 5/10/2023.

Team, Keras (2023a): Keras documentation: About Keras. Available online at https://keras.io/about/, updated on 4/17/2023, checked on 4/23/2023.

Team, Keras (2023b): Keras documentation: The Model class. Available online at https://keras.io/api/models/model/, updated on 4/17/2023, checked on 5/5/2023.

Team, Keras (2023c): Keras documentation: The Sequential model. Available online at https://keras.io/guides/sequential_model/, updated on 4/17/2023, checked on 5/5/2023.

Team, Keras (2023d): Keras documentation: Input object. Available online at https://keras.io/api/layers/core_layers/input/, updated on 5/23/2023, checked on 5/29/2023.

TensorFlow (2023a): tf.keras.initializers.HeNormal | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/initializers/HeNormal, updated on 3/23/2023, checked on 5/29/2023.

TensorFlow (2023b): tf.keras.layers.Flatten | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten, updated on 3/23/2023, checked on 5/31/2023.

TensorFlow (2023c): tf.keras.layers.Activation | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Activation, updated on 5/20/2023, checked on 5/29/2023.

TensorFlow (2023d): tf.keras.layers.BatchNormalization | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization, updated on 5/20/2023, checked on 5/29/2023.

TensorFlow (2023e): tf.keras.layers.Conv2D | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D, updated on 5/20/2023, checked on 5/29/2023.

TensorFlow (2023f): tf.keras.layers.Dense | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense, updated on 5/20/2023, checked on 5/31/2023.

TensorFlow (2023g): tf.keras.callbacks.EarlyStopping | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping, updated on 5/27/2023, checked on 6/2/2023.

TensorFlow (2023h): tf.keras.layers.Dropout | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout, updated on 5/27/2023, checked on

5/31/2023.

TensorFlow (2023i): tf.keras.layers.MaxPool2D | TensorFlow v2.12.0. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D, updated on 5/27/2023, checked on 5/31/2023.

Terra, John (2020): Keras vs Tensorflow vs Pytorch: Key Differences Among Deep Learning. In *Simplilearn*, 7/26/2020. Available online at https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article, checked on 5/6/2023.

The History of Keras: From Research Project to Industry Standard – TS2 SPACE (2023). Available online at https://ts2.space/en/the-history-of-keras-from-research-project-to-industry-standard/, updated on 4/21/2023, checked on 4/23/2023.

The PASCAL Visual Object Classes Challenge 2012 (VOC2012) (2020). Available online at http://host.robots.ox.ac.uk/pascal/VOC/voc2012/, updated on 6/27/2020, checked on 5/10/2023.

The Street View House Numbers (SVHN) Dataset (2011). Available online at http://ufldl.stanford.edu/housenumbers/, updated on 12/16/2011, checked on 5/10/2023.

torch.nn.init — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_normal_, updated on 5/31/2023, checked on 5/31/2023.

torch.utils.data — PyTorch 2.0 documentation (2023). Available online at https://pytorch.org/docs/stable/data.html?highlight=dataloader#torch.utils.data.DataLoader, updated on 5/31/2023, checked on 5/31/2023.

Training and evaluation with the built-in methods. Available online at https://www.tensorflow.org/guide/keras/train_and_evaluate, checked on 5/5/2023.

Training with PyTorch — PyTorch Tutorials 2.0.0+cu117 documentation (2023). Available online at https://pytorch.org/tutorials/beginner/introyt/trainingyt.html, updated on 5/5/2023, checked on 5/5/2023.

Treebank-3 - Linguistic Data Consortium (2023). Available online at https://catalog.ldc.upenn.edu/LDC99T42, updated on 5/10/2023, checked on 5/10/2023.

Yuan, Lin (2020): A Brief History of Deep Learning Frameworks - Towards Data Science. In *Towards Data Science*, 12/8/2020. Available online at https://towardsdatascience.com/a-brief-history-of-deep-learning-frameworks-8debf3ba6607, checked on 4/23/2023.

Zeolearn (2023): Machine Learning- The Complete Guide. Available online at https://www.zeolearn.com/magazine/what-is-machine-learning, updated on 7/1/2023, checked on 7/1/2023.