HO CHI MINH NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

GROUP CQ2021-01

Seminar Keras Report

Introduction to Big Data

H Chí Minh, May 26th, 2024

SEMINAR OVERVIEW

INSTRUCTOR: Mr. Bui Huynh Trung Nam, Mrs. Nguyen Ngoc Thao

GROUP INFORMATION

Student ID	Full name
21120275	Huynh Cao Khoi
21120308	Pham Le Tu Nhi
21120496	Chu Hai Linh
21120533	Le Thi Minh Phuong

ACKNOWLEDGEMENT:

We would like to express our sincere gratitude to Mr. Bui Huynh Trung Nam and Mrs. Nguyen Ngoc Thao for his/her dedicated guidance in this seminar. We are also deeply thankful to Ms. Nguyen Ngoc Thao for her dedication to teaching and her inspiring contributions to our understanding of Big Data in this course Introduction to Big Data.

Contents

1 Problem Statement

Keras is a high-level, cross-backend deep learning framework that can run on TensorFlow, JAX, and PyTorch.

Keras is used to build various **deep learning** models. Therefore, a solid understanding of deep learning is essential to get to know Keras. In order to support our introduction to and demonstration of Keras later, the objectives of this section are to:

- Provide a fundamental and basic background on deep learning for readers who may not be familiar with the topic, using an interesting and easy-to-understand approach.
- Introduce a simple deep learning problem that we will solve using the Keras framework in the Demonstration Section.

1.1 MNIST Handwritten Digit Classification

The problem stated in this work is MNIST Handwritten Digit Classification, which is an image classification task. The MNIST database [3] is a benchmark in the field of machine learning and deep learning, consisting of 70,000 images of handwritten digits (0-9) in size 28x28 (pixels). This dataset serves as an excellent starting point to learn and experiment with image classification tasks. Therefore, we use it in our work as a use case problem to learn how to use Keras.

Figure 1: MNIST database (source)

The task here is to classify an image into one of 10 classes, corresponding to the digits from 0 to 9. The deep learning model used in this work is Convolutional Neural Network (CNN), because CNNs are particularly effective for image classification tasks.

1.2 Convolutional Neural Network Intuition

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for processing structured grid data, such as images. CNNs have been highly successful in various image classification tasks due to their ability to capture spatial hierarchies and patterns effectively.

To understand CNNs intuitively, it helps to draw an analogy to the human visual system. Our **visual cortex**, the part of the brain responsible for processing visual information, has a structure that processes visual input hierarchically and locally.

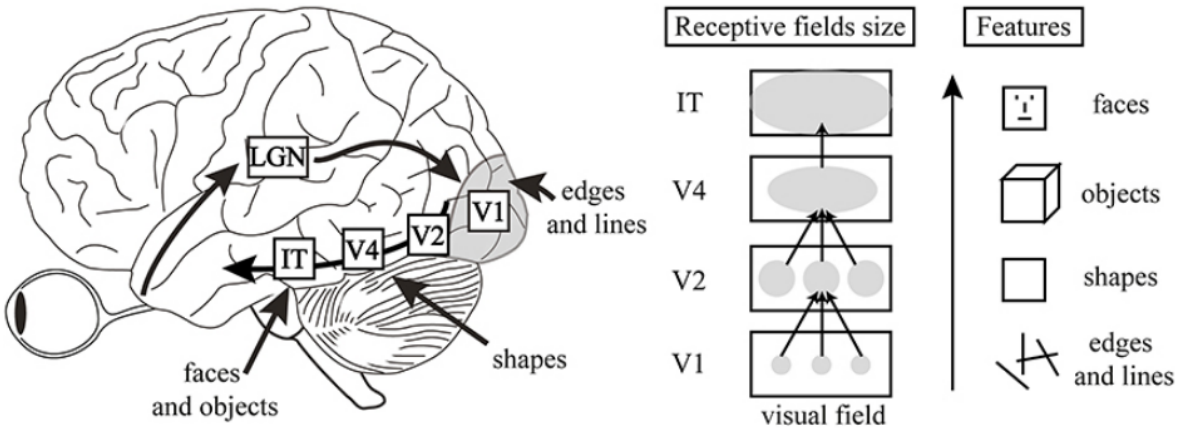

Figure 2: Visual cortex processes visual input hierarchically and locally (source)

The human visual system processes visual information in stages:

- **Local Receptive Fields:** Neurons in the visual cortex respond to a small, localized region of the input image at a time. This small region is the neuron's receptive field. By combining information from these local receptive fields, it can understand the whole image.
- **Hierarchical Processing:** The visual system processes information hierarchically in orderly areas of the visual cortex (namely V1, V2, V3, V4, etc.). Early areas detect simple features like edges and corners, while later areas combine these features to recognize complex patterns such as shapes and objects.

Similarly, CNNs process images using layers that mimic these principles.

- **Local Receptive Fields:** In a CNN, the concept of local receptive fields is implemented using convolutional layers. Each neuron in a convolutional layer only looks at a small region of the input image, called a local receptive field, rather than the entire image. This local processing helps the network learn features such as edges, textures, and simple shapes, which are crucial for recognizing more complex patterns in later layers.
- **Hierarchical Processing:** CNNs build up from detecting simple features to recognizing complex patterns through multiple layers. Early layers extract simple

features like edges and corners. Intermediate layers combine these simple features to detect more complex shapes and patterns. Higher layers integrate these complex patterns to recognize entire objects or specific regions of interest.

1.3 Basic Layers of a Simple CNN Model

A Convolutional Neural Network (CNN) typically consists of several key types of layers, each serving a specific function in the network's operation:

Input Layer: The input layer takes the raw image data as input.

Convolutional Layer: Convolutional layers are the core building blocks of a CNN. They apply filters (**kernels**) to the input image to produce feature maps. Each kernel detects specific features, such as edges, textures, or more complex patterns.

The filtering is performed on the input image using an operation called **convolution**. The kernel (a smaller matrix of weights) slides over the input image `s` pixels at a time (`s` is called **stride**). At each position, the kernel performs element-wise multiplication with the corresponding pixels of the input image (this region is called receptive field as mentioned above). The resulting values are summed to produce a single value in the output feature map.

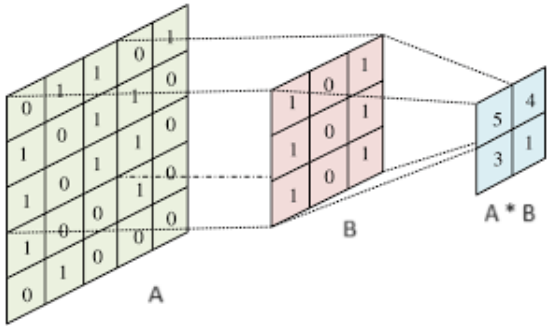

Figure 3: Convolution operation (source)

Pooling Layer: Pooling layer is used to progressively reduce the spatial dimensions (width and height) of the input feature maps, thereby reducing computational cost. Besides, pooling layers contribute significantly to expanding the receptive fields in CNNs, enable subsequent layers to capture more abstract and high-level features over larger

areas of the input image.

There are different types of pooling: max pooling, min pooling and average pooling. Max pooling is most commonly used, which selects the maximum value from each window of the feature map, thus, captures the most prominent feature within the receptive field.

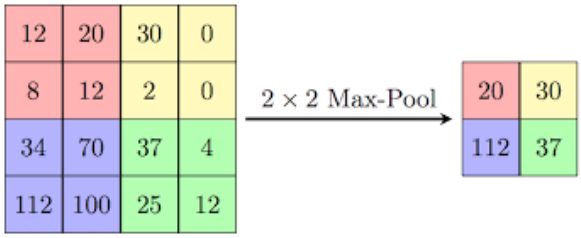

Figure 4: Max pooling (source)

Dense Layer: Dense layer is also known as fully connected layer. In a dense layer, each neuron is connected to every neuron in the preceding layer. This dense connectivity pattern allows the layer to capture complex relationships between input features and output targets. In many CNN architectures, dense layers are commonly placed at the end of the model, serving as the final layers responsible for performing classification tasks.

Output Layer: The output layer is the final layer of the model, often a dense layer with a number of neurons equal to the number of classes.

Despite this understanding of the fundamental concepts of CNNs, diving into the detailed implementation of CNNs can seem complex, especially for beginners. It is the reason for high-level APIs like Keras to simplify implementation.

2 Keras History and Development

2.1 History

The history of Keras started from the boom of Deep Learning, ImageNet and Google (as we can get the intuition from our problem statement section above). Let's unravel its complexity and try to understand the how and why Keras exist.

2.1.1 ImageNet and the rise of Deep Learning

ImageNet is a image dataset created by Dr. Fei-Fei Li along with her colleagues and students. There are 2 things that make ImagineNet special:

1. ImageNet is a large data set. The images of ImageNet is organized according to the WordNet hierarchy. There are over 100,000 synsets in WordNet, and ImagineNet tries to provides on average 1000 images to illustrate each[2]. By doing simple calculation of multiplying the number of subsets and images, there should be about 100,000,000 (that's 100 millions) images in the dataset.

2. Each image in ImageNet is hand-annotated and label. This makes it possible to run ML/DL models on the dataset.

In 2010, ImageNet Large Scale Visual Recognition Challenge (ILSVRC) started. ImageNet Challenge is an annual Computer Science competition that ran from 2010 to 2017. The goal of the competition is to create and evaluate the best computer vision algorithm.

The task of the project is to classify images to the correct object represented in the image. For the first 2 years of the competition, the winner was both non-Deep Learning algorithm, with error rates of 28% - 26% respectively.

Comes 2012, and the winner of the competition was a Deep Learning algorithm called AlexNet. AlexNet is special because it's one of the first successful Deep Learning

algorithm. Its error rates was 16.4%, a lot less than 2011's, winning the competition with virtually no competition. That was one of the first time people realize the potential and power of a Deep Learning model.

After 2012, all winning models are Deep Learning model, and their accuracy keeps getting better and better. Today, we have a lot of sophisticated models. LLM has become a normal part of many people lives, and Deep Learning models are used in nearly every fields.

2.1.2 Francois Chollet and the birth of Keras

When AlexNet won ImageNet 2012, there was one person who inspired. That person is the future creator of Keras - Fracois Chollet. At the time, Chollet was a researcher at the University of Tokyo[1]. After seeing the success of Deep Learning model, Chollet decided to learn more about such approach to solve problem.

Fast forward to 2014, Chollet was an AI researcher. He was interested in studying RNN models, but there wasn't any good libraries on the market. Theano and Caffee are popular choices. But Theano is a low-level library, anything above tensor computation and some neural manipulation is not supported; Caffee is written in C and quite difficult to use to implement complex Neural Network.

Seeing that there is no solution for his problems in the market, Chollet decided to create his own solution. He started working on Keras and publish it to open source. At the time, Keras uses Theano as backend for its operations.

2.1.3 Google involvement and TensorFlow

Sometime in 2015, Francois Chollet joined Google as a Software Engineer - which is independent from his work with Keras. At Google, he took note of a developing project called Tensor Flow, and though to incorporate it into Keras. Google eventually noticed Chollet work with Keras, and ask him to start a team creating Tensor Flow incorporate version of Tensor Flow, which he agrees.

This eventually becomes the release of Keras 2.0. After this, the Google continues to back the development of Keras. Chollet stays in Keras team to fully focus on develop Keras. In 2023, Keras 3.0 was released, with more functionalities incorporating different backend technologies.

2.2 Development

Current versions of Keras: Keras 3.3.3

2.2.1 Popularity of Keras

Keras, one of the most popular deep learning libraries in the world, boasts an extensive user base and is widely utilized in countless projects. With millions of users worldwide, Keras has become an integral tool for a vast array of applications.

Over the past few years, Keras has experienced a significant surge in popularity. This can be attributed to several factors, including its remarkable ease of use, flexibility, and outstanding performance.

One of the primary reasons for Keras's widespread adoption is its user-friendly nature. Keras provides a simple and intuitive API that enables users to effortlessly construct and train deep learning models. Even individuals with limited experience in machine learning can quickly grasp the fundamentals and begin leveraging the power of Keras.

Furthermore, Keras offers unparalleled flexibility. Users can create complex network architectures, tailor models to suit their specific requirements, and experiment with various configurations. This level of customization ensures that Keras can seamlessly accommodate a wide range of applications and enables researchers and developers to push the boundaries of what is achievable in deep learning.

Moreover, Keras's exceptional performance is another key factor driving its popularity. Built on top of TensorFlow, one of the most high-performing deep learning frameworks available, Keras inherits its efficiency and reliability. This integration with TensorFlow allows Keras to leverage its extensive computational capabilities, making it an excellent

choice for projects that demand optimal performance.

2.2.2 Comparison with other solutions

Firstly, let's examine a line chart illustrating the level of interest over time for several deep learning libraries. This chart is generated from data obtained from Google Trends spanning from 2015 to the present. The libraries included in this analysis are Keras, TensorFlow, PyTorch, Caffe, and Theano.

Upon observing the chart, we can deduce that Theano, developed by the University of Montréal in 2009, and Caffe, created by the Berkeley Vision and Learning Center (BVLC) in 2013, garnered early attention but did not experience significant growth. In 2015, the advent of TensorFlow attracted heightened interest, leading to rapid development and increased attention. Subsequently, the emergence of Keras and PyTorch also contributed to their growing popularity. Notably, TensorFlow and PyTorch received the most attention due to their practical applications. Keras, on the other hand, maintained a moderate level of interest.

Next, we will compare **Keras** with two popular deep learning libraries, **PyTorch and TensorFlow**. We will briefly introduce these two libraries and highlight their differences with Keras.

TensorFlow

TensorFlow is an open-source library developed by Google in 2015. It was created to support the construction and deployment of complex machine learning models. At its core, TensorFlow provides a symbolic math library that allows users to define and execute computational operations on data graphs called "tensors". Tensors are multi-dimensional data objects (e.g., arrays) and are the central component for data processing in TensorFlow. TensorFlow supports multiple levels of abstraction for building and training machine learning models.

PyTorch

PyTorch is a deep learning framework based on the Torch library. It was developed by Facebook's artificial intelligence research team in 2016 and was publicly released on GitHub in 2017. One of the strengths of PyTorch is its ability to handle dynamic computation graphs, allowing users to build and modify models flexibly during the training process. PyTorch is also known for its use of Python as the primary language, making it easy for users to perform computations and operations on data.

Comparison

- **API Level:** Keras is a library that provides a high-level API. In contrast, PyTorch provides a low-level API. TensorFlow, on the other hand, is more extensive, providing both high-level and low-level APIs depending on the user's needs.
- **Language:** Keras is written in the Python language, while PyTorch and TensorFlow also use additional languages such as CUDA and C++.
- **Ease of Use:** Keras uses the Python language with basic syntax, making it easy for beginners to understand and use. PyTorch has a user-friendly interface and relatively easy-to-understand syntax. In contrast, TensorFlow is considered to be difficult to use, requiring users to have knowledge and skills to use it effectively.
- **Development Speed and Flexibility:** Keras has pre-built models, allowing users to quickly use and deploy models. However, this reduces the flexibility of Keras, making it difficult for users to modify models according to their preferences. In the case of PyTorch and TensorFlow, many models need to be built from scratch, so the deployment speed may be slower. However, these libraries offer higher flexibility in terms of fine-tuning and modifications of models.
- **Community support:** Community support is strong across all three libraries.

However, with its earlier inception and higher adoption rate, TensorFlow boasts the largest community support among the three libraries.

When to use?

- PyTorch and Keras are both good choices if you are starting to work with deep learning frameworks and need to quickly implement deep learning models.
- Keras helps reduce programming and model optimization work, allowing you to focus on building and experimenting with models rather than implementation details. On the other hand, TensorFlow and PyTorch are preferred by researchers for their flexibility in models and good performance for large datasets.
- Use TensorFlow if you need high scalability and want to deploy models in real-world environments.

2.2.3 The applications of Keras in real life

Keras has found widespread use in both industry and the research community. It boasts over one million individual users as of late 2021, making it one of the most popular deep learning solutions alongside TensorFlow 2.

Major companies like Netflix, Uber, Yelp, Instacart, Zocdoc, and Square utilize Keras in their applications. It is particularly favored by startups that prioritize deep learning in their products.

Among researchers, Keras and TensorFlow 2 are highly regarded and frequently mentioned in scientific papers indexed by Google Scholar. Additionally, renowned scientific institutions like CERN and NASA have also adopted Keras for their research endeavors.

3 Keras Architecture

Layer and Model are the building blocks of Keras API. Using them is easy because the way Keras is build mirror the implementation of actual Deep Learning Neural Network implementation and design.

3.1 Layer

Keras Layer consists of a tensor-in tensor-out computation function. The states of the tensor-in and tensor-out are stored as well.

The Keras layer is store in `keras.layer` package. A layer can be created in just one line, and programmers can provides the necessary hyperparameter for each layer as provided by Keras.

Here is an example of using Keras layer:

```
1    import keras
2    from keras import layers
3
4    layer = layers.Dense(32, activation='relu')
```

Beyond every Keras layer, there is a base Layer with basic functionalitis like `add_weight`, `set_weight`, `add_loss`, etc.

Keras provides implementations for many layers, includes: 1D to 3D Convolution layer, 1D to 3D Pooling layer, Recurrent layer, Regularization layers, Reshaping layers, Merge layers, etc., and a wide range of layer activation as well: relu, sigmoid, softmax, softplus, etc.

3.2 Model

Keras models are build on top of Keras layers. There are 3 types of Keras Models:

- Sequential API: Single intput and output layer/model.

- Functional API: Mutiple inputs and outputs layer/model.
- Class subclassing: Keras support for programmers who want to take advantage of Keras architecture but wants to customize the class/model functionalities specific to their tasks.

Let's dive deeper to the Keras models to understand them better.

3.2.1 Sequential API

Figure 5: Sequential API model - Visualized

Sequential API model is single input/output model. This is the most simple model architecture to implement because programmer don't have to specify the input/output of each layer in the model. The order of insertion to the model determine the flow of the data through the mode - which determine the input and output of each layer.

Here is a simple implementation for the Sequential API model:

```
model = Sequential([
    Flatten(input_shape(28, 28)),
    Dense(128, activation='relu'),
    Dense(128, activation='softmax')
])
```

We can see that programmer only need to determine which layer to add to the model and their order. Keras will handle the rest.

Sequential API model can help implements many famous and quite capable models, like LeNet (1998), AlexNet (2012) and VGGNet (2014)[4].

3.2.2 Functional API

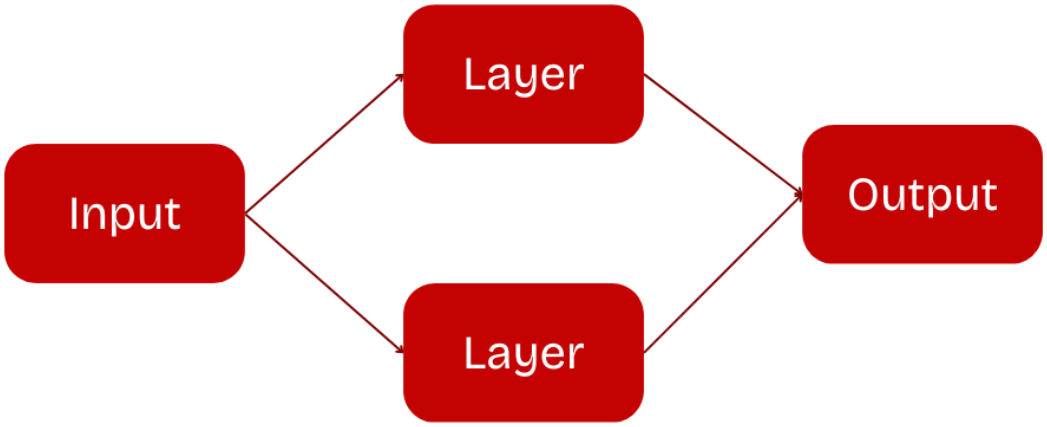

Figure 6: Functional API model - Visualized

Functional API model are multi input/output model. Functional API are more complex than Sequential API model because it's require the programmer to specify the input and output of each layer.

Here is a simple implementation of the Functional API:

```
inputs = layers.Input(shape(28, 28, 1))
layer1 = layers.Conv2D(32, (3,3), activation='relu')(inputs)
layer2 = layers.Conv2D(32, (5,5), activation='relu')(inputs)
merge = layers.concatenate([layer1, layer2])
x = layer.MaxPooling((2,2))(merge)...
ouputs = layers.Dense(10, activation='softmax')(x)
model = Model(inputs=inputs, outputs=outputs)
```

Function API model can help implements many modern models like ResNet, GoogleLeNet/Inception, Xception, SqueezeNet[4].

3.2.3 Model subclassing

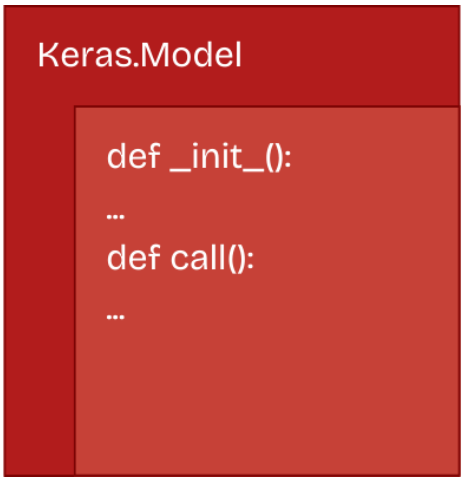

Figure 7: Model Subclassing model - Visualized

With Sequential API model and Functional API mode, we are using the architecture that Keras provide. There can be many reason why the programmer would want to customize the layer/models to their liking. This can be a simple tweek in the layer so that it perform addition computation in layer, or a more common use case, programmer want to customize the *fit function* of their model. This typically mean rewriting the `train_step` function. Whatever the cause, there is a need of using the establish architecture of Keras to build more customized models, and that's why Model subclassing API exist.

Model subclassing let the programmer adjust the functions as they see fit, using Keras architecture. Simply explain, it allows the programmer to rewrite/inherit from the base `keras.layer.Layer` and `keras.Model` class, and write their own class.

Here is a simple implementation of the Functional API:

```
class MyModel(keras.Model):
    def __inti__(self):
        super(MyModel, self).__init__:
        self.flatten = Flatten()
        self.dense1 = Dense(128, activation='relu')
        self.dense2 = Dense(10, activation='softmax')
```

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

3.3 Keras 3.0 Functionalities: Cross backend architecture

Keras 3.0 incorporate 3 backend libraries - this includes: TensorFlow, JAX and Pytorch. It goes one step further and focuses in ways that programmers can benefit from all 3 backends - even if you only know how to use 1, or even none.

3.3.1 Process any backend data input

Keras' `fit`, `predict`, etc. work with any backend (and with the addition of numpy array, Pandas dataframe input. So if programmers is want to use Keras model in their code base in a backend/numpy/pandas, it's easy, simple and require no extra processing step.

3.3.2 Ultilizing `keras.ops`

One of the new features in Keras 3.0 is `keras.ops`. Simply put, `keras.ops` helps you write 1 code and run it in all 3 backend (and even numpy array). Programmer can do this by using the `keras.ops` operator for calculation operations, or any backend specified operation.

So for example, you can run this code:

```
import keras
from keras import ops

TokenAndPositionEmbedding(keras.Layer):
    def __init__(self, max_length, vocab_size, embed_dim):
        super().__init__()
        self.token_embed = self.add_weight(
```

```
                shape=(vocab_size, embed_dim),
                initializer='random_uniform',
                trainable=True,
            )

            self.positional_embed = self.add_weight(
                shape=(max_length, embed_dim),
                initializer='random_uniform',
                trainable=True,
            )

            def call(self, token_ids):
                length = token_ids.shape[-1]
                positions = ops.arange(0, length, dtype='int32')
                position_vectors = ops.take(self.position_embed,
    position, axis=0)

                token_id = ops.cast(token_ids, dtype='int32')
                token_vector = ops.take(self.token_embed, token_ids,
    axis=0)

                embed = token_vectors + positions_vectors

                power_sum = ops.sum(ops.square(embed), axis=1,
    keepdims=True)
                return embed / ops.sqrt(ops.maximum(power_sum, 1e-7))
```

The code above can run on any backend, no mater if you choose Tensor Flow, JAX or Pytorch. This can be beneficial if you want to run your code in certain backend that you don't have experience in. It can save a lot of time since learning a new framework can be time consuming while your work can demand immediate solution.

3.3.3 How do you ultilize different backend in Keras?

The 3 backend incorporate in Keras are TensorFlow, JAX and Pytorch. Each with its strength and weakness. Using Keras, you can take advantage of is cross-backend architecture and potentially able to take advantage of the best that each has to offer.

- For Development, you can take advantage of Pytorch and its development ecosystems - the many processing libraries that are build on top of Pytorch. Many popular choices are fastai, PyTorch geometric, OpenMined, etc.
- For Training, you can use JAX and its support for large scale TPU and parallelism. JAX is a new library focuses on the ability to process massive calculation scheme on distributed, large systems.
- For deployment, you can rely on TensorFlow and its many distribution streams. Developed by Google (though it's a lot older than JAX), TensorFlow offers many solution for deployment in all sort of platform, from web to mobile to cloud.

4 Demonstation

In the previous sections, we have discussed what Keras is and its functionalities. We have also explored some of the factors that help Keras become one of the most popular libraries for Deep Learning, especailly for the beginers

To further highlight the advantages of Keras for users, in the upcoming sections, we will explore how we can usse Keras to implement a simple Convolutional Neural Network (CNN) model for the MNIST dataset[5]

4.1 Import necessary libraries

First, we need to import some modules and classes from Keras:

```
# Import the dataset
from keras.datasets import mnist

# Import helper function for data preprocessing
from keras.utils import to_categorical

# Import the model and load pre-train model method
from keras.models import Sequential, load_model

# Import some layer types
from keras.layers import Conv2D, Flatten, Dropout, Dense,
MaxPooling2D

# Import Stochatic Gradient Descent optimaizer .
from keras.optimizers import SGD
```

Here, we have import all resources we need. Now, let's start to build our model. First, we need to collect our data.

4.2 Load the data

The MNIST dataset consists of 70,000 handwritten digit images (60,000 for training and 10,000 for testing) and their corresponding labels (the digit represented in the image). So with mnist we have imported above, we will use `load_data` method to get the train and test dataset, stored into four separate variables:

```
(train_X,train_y),(test_X,test_y) = mnist.load_data()
```

- `train_X`: Contain 60,000 training images. The shape will be (60000, 28, 28), with each image has 28x28 pixels.
- `train_y`: Contain 60,000 training labels, which are integers from 0 to 9 representing the digit showed in the corresponding image.
- `test_X`: Contain 10,000 test images. The shape will be (10000, 28, 28).
- `test_y`: Contain 10,000 test label of each image in `test_X`.

Also, we can see some example of images in `train_X`:

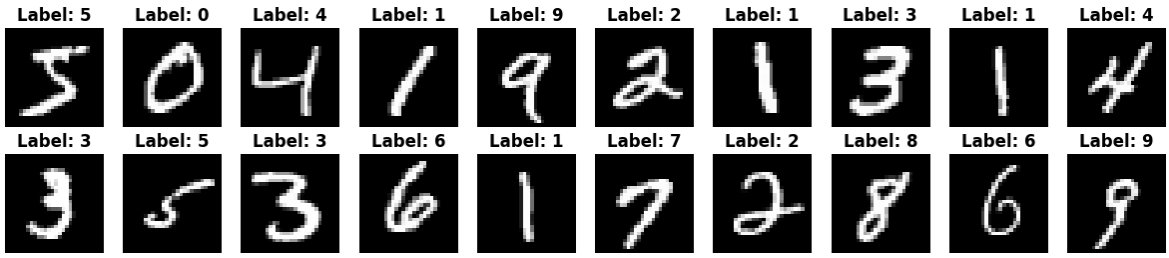

Figure 8: Some examples of MNIST dataset

4.3 Preprocess the data

Before feeding this data into our model, we need to do some preproccessing:

1. Our CNN model expect the input data to have a specific shape, typically (batch_size, height, width, channels). But the original MNIST dataset's shape is (num_samples,

height, width). That's why we need to add an extra dimension to represent the single-channel images.

```
train_X = train_X.reshape(-1,28,28,1)
test_X  = test_X.reshape(-1,28,28,1)

```

2. The pixel values of a image are in the range from 0 to 255, also it can contain integer values. So we will convert all the values into float, also normalize these values to the range $[0, 1]$ by dividing each pixel value by 255. This helps the model learn more effectively and can improve its performance.

```
train_X = train_X.astype('float32')
test_X  = test_X.astype('float32')
train_X = train_X / 255
test_X  = test_X / 255

```

3. The labels, currently, are integers from 0 to 9, representing the digit in the image. However, our model expect the labels to be in a categorical format, where each label is represented as a one-hot encoded vector. The `to_categorical()` function will help us perform this conversion.

```
train_y = to_categorical(train_y)
test_y  = to_categorical(test_y)

```

Now, our data is ready, we will start to build the fist CNN model.

4.4 Build model

We can build a CNN model in Keras with these following steps:

4.4.1 Initialize model

The first step is create a Sequential model,which is simply a stack of layers. This can be done by the following code.

```
model = Sequential()

```

The Sequential model is the simplest way to build a neural network in Keras, where layers are added one after the other to form the model.So the output of the current layer will be the input of the next layer.

4.4.2 Adding necessary layers

Now, we continue the CNN model by adding these layers:

1. **First Convolution layer and MaxPooling layer**:

```
 model.add(Conv2D(filters=32, kernel_size=(3,3), activation='
relu',input_shape=(28,28,1), padding='SAME'))
 model.add(MaxPooling2D(pool_size=(2,2)))

```

   Our first convolutional layer get these parameters:

   - `filters`: Define the numer of kernel used in our model, each kernel is responsible to extract one or more features from the image.
   - `kernel_size` : define the size of each kernel in our model.
   - `input_shape`: The size of input image , which is (height, width, channels)
   - `activation`: The activation function of a layer. We will use **ReLU** function, the most common activation function, and is highly recommended to use in most of deep learning model.
   - `padding`: Determine how the input image is padded before applying the convolution operation:

- `valid`: there is no padding and the output size is smaller than the input size.
- `same`: The output size is the same as the input size. Padding is added to the input image by adding rows and columns of zeros around the border, so that the convolution can be applied to the full input without losing any information at the edges.

After a convolutional layer, we add a max-pooling layer with the parameter `pool_size` of 2x2. This layer helps to reduce the dimensions of the feature, means that the number of parameters in the model will be reduces, making the computation of our model become easier and more efficient.

2. **Second convolutional layer and MaxPooling layer:**

```
1    model.add(Conv2D(64,(3,3),activation='relu',padding='SAME'))
2    model.add(MaxPooling2D(pool_size=(2,2)))
3
```

The next convolutional layer share the same activation, kernel size, etc. with the first one . However, because we want to extract the *high-level* of the image, the number of kernels used here is increased to 64.

After finish this step, we have already defined all layers for features extraction part of a CNN model. Now, we will define layers which will help us compute and get the result from these features.

1. **Dropout layer:**

```
1    model.add(Dropout(0.25))
2
```

Dropout layer is a technique of regularization, with the parameter `rate` (here is 0.25), stands for the percentage of neurals of the input will be randomly ignored before fed to the next layer. This layer helps to prevent the model from overfitting

to the training data, which is a common issue in deep learning models with a large number of parameters, like CNN.

2. **Flatten layer:**

```
model.add(Flatten())

```

The output of the last layer is flattened into a 1D vector, preparing it for the following dense layers.

3. **Dense Layer**

```
model.add(Dense(128,activation='relu'))
model.add(Dropout(0.5))

```

We also add a dense (fully connected) layer with 128 units and ReLU activation function, followed by a dropout layer with a rate of 0.5 for regularization.

Normally, dense layer is also known as fully-connected layers, is responsible for learning higher-level features by taking all the features extracted by the previous convolutional layers and combining them to make a final prediction.

4. **Output layer:**

```
model.add(Dense(10,activation='softmax'))

```

Finally, we add the output layer with 10 units (one for each digit class) and a softmax activation function, which will give us the probabilities of the 10 digit classes. So the output layer of the image will be the index of neural having the highest probability.

We have finished implementing all the layers our CNN model. To use the model for training step, we have to compile it by the following code:

```
model.compile(loss='categorical_crossentropy', optimizer=SGD(0.01), metrics=['accuracy'])

print(model.summary())
```

We need to understand some specific parameters:

- `loss`: In this parameter, we will define our loss function of the model. Here, we will use `categorical_crossentropy` function, a common loss function for multi-class classification problems .
- `optimizer`: Stand for the optimization algorithm used to update the model's weights during training. Here we will use `SGD(0.01)`, means Stochastic Gradient Descent with the learning rate is 0.01.
- `metrics`: The metric we will use to evaluate our model. Here is accuracy metric.

With `model.summary()`, we can see the summary version of our model

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2  (None, 14, 14, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 64)          0
 g2D)

 dropout (Dropout)           (None, 7, 7, 64)          0

 flatten (Flatten)           (None, 3136)              0

 dense (Dense)               (None, 128)               401536

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 421642 (1.61 MB)
Trainable params: 421642 (1.61 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

Figure 9: Model summary

In this summarization, we can see the sequence of layers in our model, also the size and the amount of parameter used in each layer.

4.5 Train

Then, we will use the training and testing dataset to train the model.

```
batch_size = 32
epochs = 10

history_data = model.fit(train_X,
                         train_y,
                         batch_size=batch_size,
                         epochs=epochs,
                         verbose=1,
                         validation_data=(test_X,test_y))
```

But we need tounderstand some hyperparameters:

- **batch_size**: Define the number of samples our model will train at a time.
- **epochs**: Define the number times that our model will learn from our entire training dataset.
- **validation_data**: the validation dataset, which will be used to evaluate the model's performance during training.
- **verbose=1**: Custom parameter to show us the training process.

During the training process, the model will be trained on the `train_X` and `train_y` data, and its performance will be evaluated on the `test_X` and `test_y` data at the end of each epoch.

This process will be repeated 10 times, and the model will improve its ability to classify, also its accuracy.

```
Epoch 1/10
1875/1875 [==============================] - 86s 45ms/step - loss: 0.7242 - accuracy: 0.7678 - val_loss: 0.1618 - val_accuracy: 0.9510
Epoch 2/10
1875/1875 [==============================] - 86s 46ms/step - loss: 0.2409 - accuracy: 0.9276 - val_loss: 0.1032 - val_accuracy: 0.9686
Epoch 3/10
1875/1875 [==============================] - 86s 46ms/step - loss: 0.1781 - accuracy: 0.9468 - val_loss: 0.0800 - val_accuracy: 0.9748
Epoch 4/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.1487 - accuracy: 0.9554 - val_loss: 0.0683 - val_accuracy: 0.9778
Epoch 5/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.1295 - accuracy: 0.9610 - val_loss: 0.0584 - val_accuracy: 0.9810
Epoch 6/10
1875/1875 [==============================] - 86s 46ms/step - loss: 0.1172 - accuracy: 0.9648 - val_loss: 0.0539 - val_accuracy: 0.9825
Epoch 7/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.1062 - accuracy: 0.9683 - val_loss: 0.0513 - val_accuracy: 0.9840
Epoch 8/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.0992 - accuracy: 0.9695 - val_loss: 0.0437 - val_accuracy: 0.9854
Epoch 9/10
1875/1875 [==============================] - 85s 45ms/step - loss: 0.0903 - accuracy: 0.9726 - val_loss: 0.0428 - val_accuracy: 0.9856
Epoch 10/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.0857 - accuracy: 0.9737 - val_loss: 0.0394 - val_accuracy: 0.9868
```

Figure 10: Training process

After each epoch, the accuracy on both the training set and the test set increasing. This indicates that our model is improving its performance in the training progress.

The `history_data` variable can be used to analyze the model's learning curve and monitor its performance during the training process.

```
history_dict = history_data.history


test_loss = history_dict['val_loss']
training_loss = history_dict['loss']
test_accuracy = history_dict['val_accuracy']
training_accuracy = history_dict['accuracy']
```

We can also visualize the result:

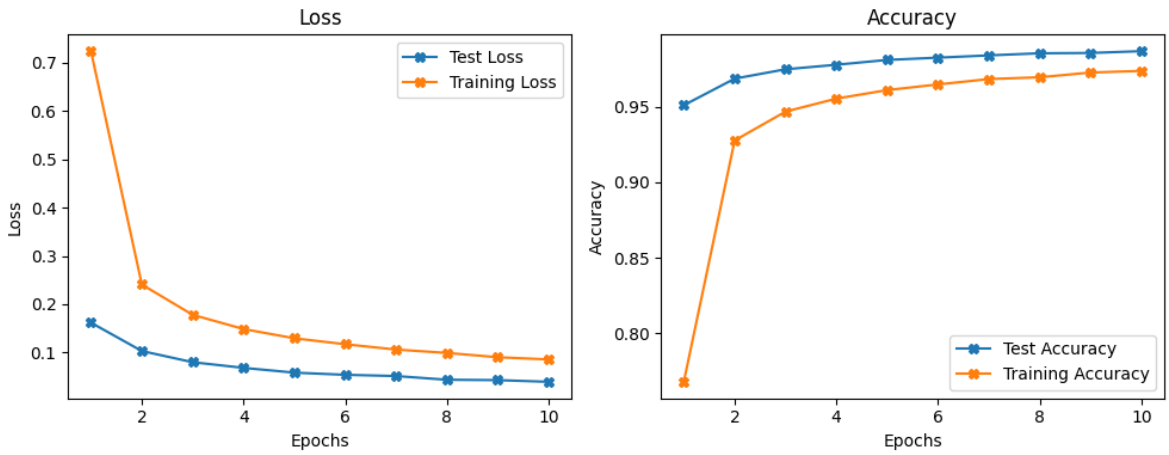

Figure 11: Training process

4.6 Evaluate model

After the training process, we also evaluate the quality of our model by `evaluate()` method.

```
loss,accuracy = model.evaluate(test_X,test_y,verbose=0)

print('Test loss ---> ',str(round(loss*100,2)) +str('%'))
print('Test accuracy ---> ',str(round(accuracy*100,2)) +str('%'))
```

4.7 Save model

Finally, if you want to reuse you model in the future. Keras provide us a simple way to store the model.

```
model.save('MNIST_10_epochs.keras')
print('Model saved.')
```

4.8 Live Prediction

In this final part, we will use our trained model to make a prediction with the real data. By using OpenCV, we builded a simple drawing application on a black canvas. It will allow user to draw a digit on the canvas. This draw will then be processed and fed into our trained model to make a prediction about the number that was drawn.

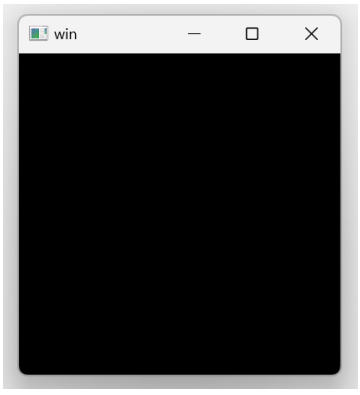

Figure 12: Drawing app

Figure 13: Draw number 1

Figure 14: Predicted result

Detail instructions for using the application:

- Use your mouse or touchpad to draw the number you want the application to predict.
- When you finish drawing, press *Enter* key to see the predicted result.
- Press *C* key to clear the canvas and start drawing a new number.
- Press *Esc* to exit the application.
- To get best the results, we have to draw the number carefully and as clearly as possible.

4.9 Conclusion

In this guide, we have explored the usage of Keras to implement CNN for image recognition tasks. We can easily recognize that Keras is a powerful and user-friendly deep learning library, providing several key benefits that simplify our developing process:

- **Easy to create layers:** In Keras, createing a layers is very simple, by passing in a few parameters such as the size of the kernel, the activation function, the number of input channels, etc. This makes the model building process much easier compared to lower-level libraries where we need to define the layers in a more complex way.
- **Simple training process:** The traning process is also very simple with `fit()` method. Unlike some other libraries such as *TensorFlow* or *PyTorch*, which have the more complex syntax, requires us to carefully define how the model will learn in each epoch.
- **Similarities in syntax with Scikit-learn:** Some methods in Keras, such as `fit()` and `predict()`, are inspired by the syntax of Scikit-learn. This can be

a significant advantage when we are already familiar with Scikit-learn, and our usage of Keras will be much more easier.

Finally, we have go through a process of how to create a deep learning model in Keras, and the way to apply it in a normal scenario. After this demonstration, we can start to implement our own model, and start to have fun in the world of deeplearning and Keras.

References

[1] Sanyam Bhutani. Interview with The Creator of Keras, AI Researcher: François Chollet. https://medium.com/dsnet/interview-with-the-creator-of-keras-ai-researcher-fran%C3%A7ois-chollet-823cf1099b7c. Accessed on 2024-05-24.

[2] ImageNet. About ImageNet. https://www.image-net.org/about.php. Accessed on 2024-05-24.

[3] Yann LeCun, Corinna Cortes, and Christopher Burges. Mnist dataset. *URL http://yann. lecun. com/exdb/mnist*, 6, 1998.

[4] Go Min-su. Deep Learning Bible - 2. Classification - Eng. https://wikidocs.net/165440. Accessed on 2024-05-24.

[5] K. Team. Keras documentation: Simple MNIST convnet. https://keras.io/examples/vision/mnist_convnet/. Accessed on 2024-05-15.