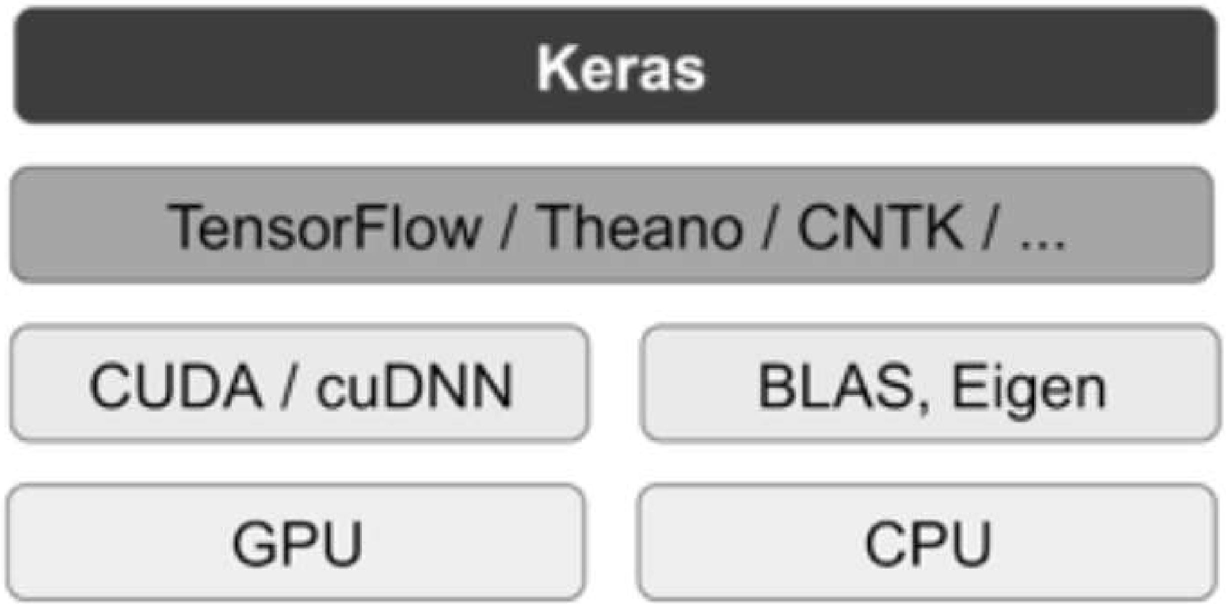
SDU Summer School

Deep Learning

Fall 2022

Introduction to KERAS

What is Keras?



- Keras is a high-level API providing easy to use elements for deep learning
- Can work with several backends
- Programs can easily deployed on CPUs, GPUs without changing the code

Who makes Keras? Contributors and backers

633 contributors

Google

Microsoft

NVIDIA

aws

The Keras user experience

- Keras API is easy to understand.
  - Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- Easy to learn.
  - As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.
- This ease of use does not come at the cost of reduced flexibility:
  - Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows.

Multi-Backend, Multi-Platform

- Develop in Python, R
  - On Unix, Windows, OSX

- Run the same code with…
  - TensorFlow
  - Cognitive Toolkit (CNTK) - Microsoft
  - Theano
  - MXNet
  - PlaidML
  - ??

- Run on CPU, NVIDIA GPU, AMD GPU, TPU...

How to use Keras: An introduction

Three API Styles

- **The Sequential Model**
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases

- **The functional API**
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases

- **Model subclassing**
  - Maximum flexibility
  - Larger potential error surface

The Sequential API

```
Import tensorflow as tf
from tensorflow.keras import layers

model = tf.keras.Sequential()
model.add(layers.Dense(20, activation=‘relu’, input_shape=(10,)))
model.add(layers.Dense(20, activation=‘relu’))
model.add(layers.Dense(20, activation=‘softmax’))

model.fit(x, y, epochs=10, batch_size=32)
```

Options for Layers

- Core Layers
- Convolutional Layers
- Pooling Layers
- Locally-connected Layers
- Recurrent Layers
- Reshape Layers
- Dropout Layers
- Merge Layers
- Normalization Layers
- Noise layers

Options for Layers

- The core layers perform the most basic operations

- They are enough to built FFN networks

- **Core Layers**
  - Input Layers
  - Dense Layers
  - Activation Layer
  - Embedding Layers
  - Masking layers
  - Lambda Layers

Dense Layer

```
tensorflow.keras.layers.Dense(units, #Number of units in the layer
     activation=None,              #Standard: use linear output
     use_bias=True,                #Add a bias vector
     kernel_initializer='glorot_uniform’, #How to initialize the weights
     bias_initializer='zeros’,     #How the biases
     kernel_regularizer=None,      #For example, apply L2 regularization
     bias_regularizer=None,        #For example, apply L2 regularization
     activity_regularizer=None,    #For example, apply L2 regularization
     kernel_constraint=None,       #For example, non-negative constraint
     bias_constraint=None          #For example, non-negative constraint
)
```

Activation Function

```
model.add(Dense(64))
model.add(Activation('tanh’))
#This is equivalent to:
model.add(Dense(64, activation='tanh'))
```

- Available Activations:
  - softmax
  - elu: (Exponential linear unit.)
    - x if $x > 0$ and $\alpha * (\exp(x) - 1)$ if $x < 0$.
  - selu: Scaled Exponential Linear Unit
  - softplus
    - $\log(\exp(x) + 1)$
  - relu
    - relu(x, alpha=0.0, max_value=None)
  - sigmoid
  - tanh

Compiling the Model

```
from tensorflow.keras import optimizers

model.compile(
     optimizer=optimizers.RMSprop(lr=0.001),
     loss=‘binary_crossentropy’,
     metrics=['accuracy’]
)
```

- Before training a model, you need to configure the learning process, which is done via the compile method, defining
  - **An optimizer**. This could be the string identifier of an existing optimizer (such as rmsprop or adagrad), or an instance of the Optimizer class
  - **A loss function**. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function, or it can be an objective function.
  - **A list of metrics.** A metric could be the string identifier of an existing metric or a custom metric function.

Examples: Compiling Models

```
# For a multi-class classification problem
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
from tensorflow.keras import optimizers
model.compile(optimizer=optimizers.RMSprop(lr=0.0025),
              loss='mse')
```

What does compile do?

- Compile defines the loss function, the optimizer and the metrics. That's all.
  - It has nothing to do with the weights and you can compile a model as many times as you want without causing any problem to pretrained weights.
  - You need a compiled model to train (because training uses the loss function and the optimizer). But it's not necessary to compile a model for predicting.
  - Do you need to use compile more than once? Only if:
    - You want to change one of these:
      - Loss function
      - Optimizer / Learning rate
      - Metrics
    - You loaded (or created) a model that is not compiled yet. Or your load/save method didn't consider the previous compilation.
- Consequences of compiling again:
  - If you compile a model again, you will lose the optimizer states.

Loss Functions

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- **binary_crossentropy**
- **categorical_crossentropy**
- sparse_categorical_crossentropy
- …

Metrics

- Can be any of the loss functions

- Some standard metrics like
  - F1
  - Precision
  - Recall
  - accuracy

Train the Model

```
model.fit(x=None, y=None,   # Input and desired outcome
    batch_size=None,        # Number of samples per gradient update. If
                            none, it defaults to 32
    epochs=1,               # Number of runs over the complete x and y
    verbose=1,              # Verbosity mode. 0 = silent, 1 = progress
                            bar, 2 = one line per epoch.
    callbacks=None,         # List of functions to call during training
    validation_split=0.0,   # Part of dataset set aside for validating
    validation_data=None,   # Validation dataset, tuple (x_val, y_val)
    shuffle=True,           # shuffle the training data before each
                            epoch
    class_weight=None,      # Give some classes more/less weight
    sample_weight=None,     # Give some samples more/less weight
    ...
)
```

In Context

```
model.compile(optimizer='rmsprop’,
    loss='binary_crossentropy’,
    metrics=['acc’])

history = model.fit(x=partial_x_train, y=partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(x_val, y_val))
```

The History Object

- Note that the call to model.fit() returns a History object. This object has a member history, which is a dictionary containing data about everything that happened during training.

```
>>> history_dict = history.history
>>> history_dict.keys()
[u’accuracy', u'loss', u'val_accuracy', u'val_loss']
```

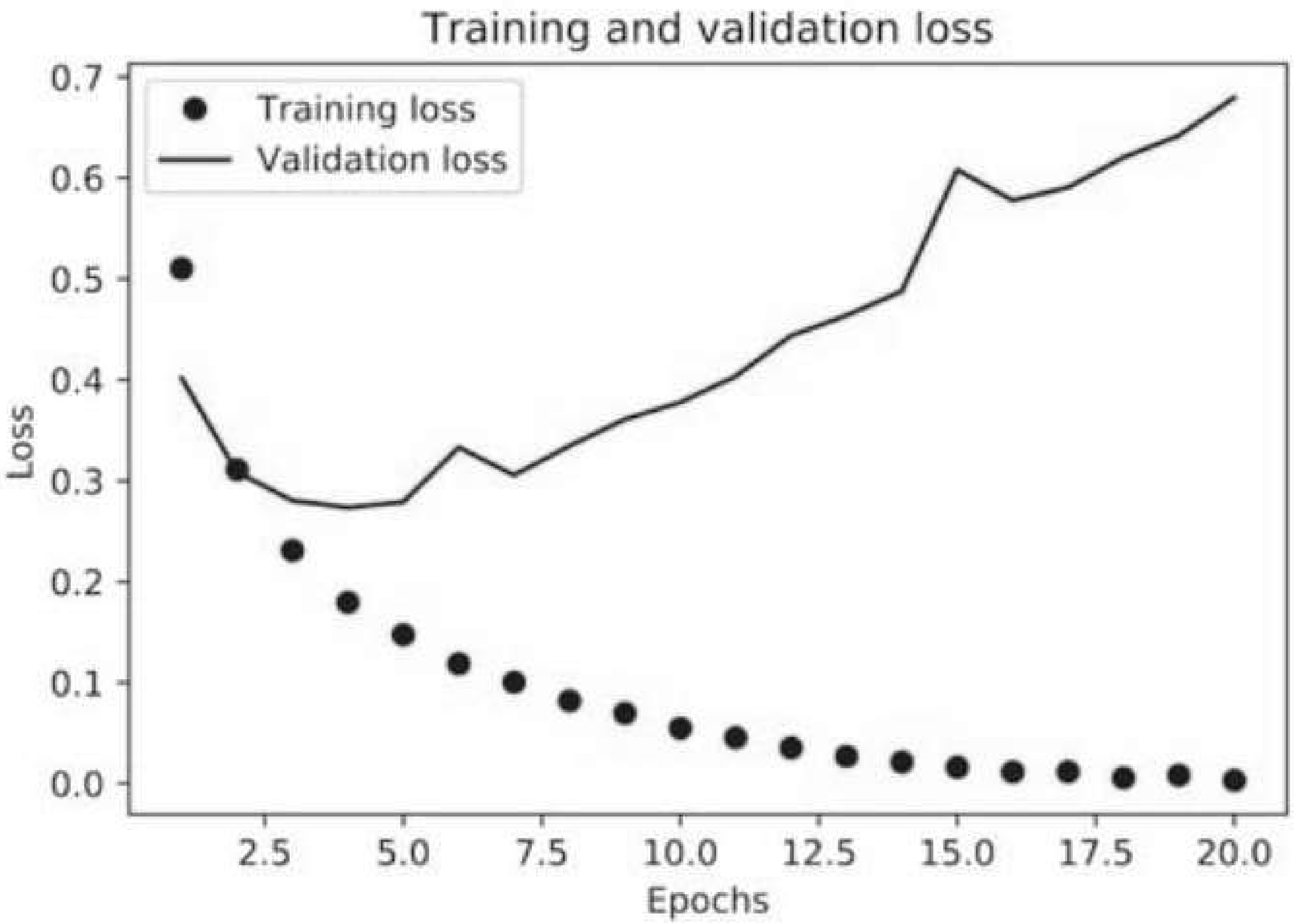
- The dictionary contains four entries: one per metric that was being monitored during training and during validation.

- You can now plot these to get Information about your performance

Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss’]

epochs = range(1, len(loss_values) + 1)

#’bo’ is for blue dot, ‘b’ is for solid blue line
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Plotting the training and validation loss



Practical Recommendations

- For lower data amounts, you should train smaller and shallower networks in order to prevent overfitting
- Preprocessing
  - Take small values - Typically, most values should be in the 0–1 range.
  - Be homogenous- That is, all features should take values in roughly the same range.

Load and Save models

- You save a Keras model into a single HDF5 file which will contain:
  - the architecture of the model, allowing to re-create the model
  - the weights of the model
  - the training configuration (loss, optimizer)
  - the state of the optimizer, allowing to resume training exactly where you left off.

```
from tensorflow.keras.models import load_model

model.save('my_model.h5') # creates a HDF5 file 'my_model.h5'
del model                 # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```