Keras Notes

[2]:
```
import tensorflow as tf
```

[3]:
```
print(tf.__version__)
```

```
2.16.2
```

[5]:
```
# create a constant tensor A
A= tf.constant([[4,2],[6,1]])
A
```

[5]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[4, 2],
       [6, 1]])>
```

[9]:
```
V=tf.Variable([[9,2],[6,5]])
V
```

[9]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[9, 2],
       [6, 5]])>
```

[17]:
```
AV=tf.concat(values=[A,V],axis=0)
av=tf.concat(values=[A,V],axis=1)
print(AV,'\n\n',av)
```

```
tf.Tensor(
[[9 2]
 [6 5]
 [9 2]
 [6 5]], shape=(4, 2), dtype=int32)

 tf.Tensor(
[[9 2 9 2]
 [6 5 6 5]], shape=(2, 4), dtype=int32)
```

[19]:
```
zero_tflw=tf.zeros(shape=[3,4],dtype=tf.int32)
zero_tflw
```

[19]:
```
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])>
```

[28]:
```
one_tflw=tf.ones(shape=[3,4],dtype=tf.float32)
one_tflw
```

[28]:
```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)>
```

[68]:
```
# identity matrix
IM=tf.constant([[1,2],[3,4],[5,6]])
rows, columns= IM.shape
print('Row:',rows,'\nColumns:' ,columns)

IM_indent=tf.eye(num_rows=rows,num_columns=columns, dtype=tf.int32)
print('indentity matrix of IM is:\n',IM_indent)
```

```
Row: 3
Columns: 2
indentity matrix of IM is:
 tf.Tensor(
[[1 0]
 [0 1]
 [0 0]], shape=(3, 2), dtype=int32)
```

[34]:
```
random_generate=tf.random.uniform(shape=[2,4],dtype=tf.float32)
random_generate
```

[34]:
```
<tf.Tensor: shape=(2, 4), dtype=float32, numpy=
array([[0.7430968 , 0.52440953, 0.8917074 , 0.6367301 ],
       [0.8461467 , 0.8472326 , 0.55651677, 0.8393111 ]], dtype=float32)>
```

[35]:
```
reshape_tensor=tf.reshape(tensor= random_generate
                          ,shape= [4,2])
reshape_tensor
```

[35]:
```
<tf.Tensor: shape=(4, 2), dtype=float32, numpy=
array([[0.7430968 , 0.52440953],
       [0.8917074 , 0.6367301 ],
       [0.8461467 , 0.8472326 ],
       [0.55651677, 0.8393111 ]], dtype=float32)>
```

[39]:
```
# typecast a tensor
x=tf.constant([[9,2],[6,5]],dtype=tf.float32)
print(x)

X=tf.cast(x,tf.int32)
print('\n',X)
```

```
tf.Tensor(
[[9. 2.]
 [6. 5.]], shape=(2, 2), dtype=float32)

 tf.Tensor(
[[9 2]
 [6 5]], shape=(2, 2), dtype=int32)
```

[43]:
```
t=tf.transpose(X)
print('transpose of X:\n',t)
```

```
transpose of X:
 tf.Tensor(
[[9 6]
 [2 5]], shape=(2, 2), dtype=int32)
```

[49]:
```
A=tf.constant([[2,3],[4,5]])
V=tf.constant([[4],[2]])
```

[52]:
```
# matmul matrix multiplier
AV=tf.matmul(A,V)
AV
```

[52]:
```
<tf.Tensor: shape=(2, 1), dtype=int32, numpy=
array([[14],
       [26]])>
```

[54]:
```
# Element wise multiply
av=tf.multiply(A,V)
av
```

[54]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 8, 12],
       [ 8, 10]])>
```

Import the necessary libraries

[97]:
```
import datetime,warnings,scipy
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
plt.rcParams["patch.force_edgecolor"]=True
plt.style.use('fivethirtyeight')
mpl.rc('patch',edgecolor='dimgray', linewidth=1)
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "last_expr"
pd.options.display.max_columns= 50
%matplotlib inline
warnings.filterwarnings("ignore")
```

Import the functions from Keras and sklearn libraries

[106]:
```
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import mean_squared_error
```

Notes by keras application and me

You will also need to install a backend framework – either JAX, TensorFlow, or PyTorch:

And

pip install --upgrade keras-cv

pip install --upgrade keras-nlp

pip install --upgrade keras

[4]:
```
import os
os.environ["KERAS_BACKEND"]= "jax"
import keras
```

Note: The backend must be configured before importing Keras, and the backend cannot be changed after the package has been imported.

[5]:
```
import os
os.environ["TF_USE_LEGACY_KERAS"] = "1"
import tensorflow
```

Note: These line wouldneed to be before any import tensorfow

Introduction

Keras 3 is a deep learning framework works with TensorFlow, JAX, and PyTorch interchangeably. This notebook will walk you through key Keras 3 workflows.

```
Keras, a popular deep learning library in Python, is divided into several key modules:

Layers API: Provides various types of neural network layers, like Dense, Conv2D, LSTM, etc., which can be used to build the architecture of a neural network.

Models API: Includes the Sequential model for building simple linear stacks of layers and the Functional API for creating more complex models.

Losses API: Offers various loss functions like mean_squared_error, binary_crossentropy, etc., which are used to measure the difference between the predicted and actual outputs.

Optimizers API: Contains different optimization algorithms like SGD, Adam, RMSprop, etc., which are used to adjust the weights of the neural network to minimize the loss function.

Metrics API: Provides different metrics like accuracy, precision, recall, etc., to evaluate the performance of a model.

Callbacks API: Enables the use of custom actions at different stages of training, such as early stopping, learning rate scheduling, etc.

Datasets API: Includes pre-built datasets like MNIST, CIFAR-10, IMDB, etc., for training and evaluating models.

Utilities API: Contains utility functions for various tasks like image preprocessing, text tokenization, and saving/loading models.
```

```
# Here are some common Keras syntax examples for different tasks:

1. Building a Sequential Model:
from keras.models import Sequential
from keras.layers import Dense

# Initialize the model
model = Sequential()

# Add layers
model.add(Dense(units=64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(units=10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
2. Building a Model using the Functional API:
from keras.models import Model
from keras.layers import Input, Dense

# Define the input layer
inputs = Input(shape=(input_dim,))

# Add layers
x = Dense(64, activation='relu')(inputs)
outputs = Dense(10, activation='softmax')(x)

# Create the model
model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
3. Training a Model:
# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
4. Evaluating a Model:
# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test, batch_size=32)
print(f'Test loss: {loss}')
print(f'Test accuracy: {accuracy}')
5. Making Predictions:
# Make predictions
predictions = model.predict(x_new)
6. Using Callbacks:
from keras.callbacks import EarlyStopping

# Define early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3)

# Train the model with callbacks
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2, callbacks=[early_stopping])
7. Saving and Loading a Model:
# Save the model
model.save('my_model.h5')

# Load the model
from keras.models import load_model
model = load_model('my_model.h5')
These examples cover the basics of defining, training, evaluating, and saving models in Keras.
```

[2]:
```python
import numpy as np
import os

os.environ["KERAS_BACKEND"] = "jax"

# Note that Keras should only be imported after the backend
# has been configured. The backend cannot be changed once the
# package is imported.
import keras
```

A first example: A MNIST convnet Let's start with the Hello World of ML: training a convnet to classify MNIST digits.

[3]:
```python
# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 13s 1us/step
x_train shape: (60000, 28, 28, 1)
y_train shape: (60000,)
60000 train samples
10000 test samples
```

Certainly! Here's a step-by-step explanation of what each part of the provided code does:

1. Load the Data

python

Copy code

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

What It Does: This line loads the MNIST dataset from Keras. MNIST is a collection of 28x28 grayscale images of handwritten digits (0-9).

Outputs:

x_train: The training images (60,000 images).

y_train: The labels corresponding to the training images (60,000 labels).

x_test: The test images (10,000 images).

y_test: The labels corresponding to the test images (10,000 labels).

2. Scale Images to the [0, 1] Range

python

Copy code

```
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

What It Does: This normalizes the pixel values of the images. Originally, pixel values range from 0 to 255. Dividing by 255 scales these values to the range [0, 1], which is more suitable for neural network training.

Result: Each pixel value in the images is now a float between 0 and 1.

3. Ensure Images Have Shape (28, 28, 1)

python

Copy code

```
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
```

What It Does: Adds an extra dimension to the image arrays to represent the number of color channels. Since MNIST images are grayscale, this dimension will be 1.

Result:

The shape of x_train changes from (60,000, 28, 28) to (60,000, 28, 28, 1).

The shape of x_test changes from (10,000, 28, 28) to (10,000, 28, 28, 1).

4. Print Shapes and Number of Samples

python

Copy code

```
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
```

What It Does: Prints the shapes of the image and label arrays and the number of samples in the training and test sets.

Outputs:

x_train shape: Shows the shape of the training image array (should be (60000, 28, 28, 1)).

y_train shape: Shows the shape of the training labels array (should be (60000,)).

x_train.shape[0]: Prints the number of training samples (60,000).

x_test.shape[0]: Prints the number of test samples (10,000).

[7]:

```python
# Model parameters
num_classes = 10
input_shape = (28, 28, 1)

model = keras.Sequential(
    [
        keras.layers.Input(shape=input_shape),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.MaxPooling2D(pool_size=(2, 2)),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.GlobalAveragePooling2D(),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(num_classes, activation="softmax"),
    ]
)
```

[8]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 64)	640
conv2d_1 (Conv2D)	(None, 24, 24, 64)	36,928
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
conv2d_2 (Conv2D)	(None, 10, 10, 128)	73,856
conv2d_3 (Conv2D)	(None, 8, 8, 128)	147,584
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0
dropout (Dropout)	(None, 128)	0
dense (Dense)	(None, 10)	1,290

Total params: 260,298 (1016.79 KB)

Trainable params: 260,298 (1016.79 KB)

Non-trainable params: 0 (0.00 B)

The code snippet you provided defines a Convolutional Neural Network (CNN) using Keras. Here's a detailed step-by-step explanation of each part:

1. Define Model Parameters

python

Copy code

```
num_classes = 10
input_shape = (28, 28, 1)
```

num_classes: Number of output classes for classification. MNIST has 10 classes (digits 0-9).

input_shape: Shape of the input images, which are 28x28 pixels with 1 color channel (grayscale).

2. Define the Model

python

Copy code

```
model = keras.Sequential(
    [
        keras.layers.Input(shape=input_shape),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.MaxPooling2D(pool_size=(2, 2)),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.GlobalAveragePooling2D(),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(num_classes, activation="softmax"),
    ]
)
```

Explanation of Each Layer:
Input Layer:

python
Copy code
keras.layers.Input(shape=input_shape)
Purpose: Specifies the shape of the input data (28x28 pixels with 1 color channel). This is the first layer of the model.
First Convolutional Layer:

python
Copy code
keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu")
Purpose: Applies 64 convolutional filters of size 3x3 to the input image.
Activation Function: ReLU (Rectified Linear Unit) introduces non-linearity.
Second Convolutional Layer:

python
Copy code
keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu")
Purpose: Applies another set of 64 convolutional filters of size 3x3.
Activation Function: ReLU.
Max Pooling Layer:

python
Copy code
keras.layers.MaxPooling2D(pool_size=(2, 2))
Purpose: Reduces the spatial dimensions of the feature maps by taking the maximum value in each 2x2 block.
Third Convolutional Layer:

```
python
Copy code
keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu")
Purpose: Applies 128 convolutional filters of size 3x3.
Activation Function: ReLU.
Fourth Convolutional Layer:

python
Copy code
keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu")
Purpose: Applies another set of 128 convolutional filters of size 3x3.
Activation Function: ReLU.
Global Average Pooling Layer:

python
Copy code
keras.layers.GlobalAveragePooling2D()
Purpose: Reduces each feature map to a single value by averaging over all spatial dimensions. This flattens the output from the previous layer while
preserving the number of feature maps.
Dropout Layer:

python
Copy code
keras.layers.Dropout(0.5)
Purpose: Applies dropout regularization with a rate of 0.5, randomly setting 50% of the input units to 0 during training to prevent overfitting.
Dense Output Layer:

python
Copy code
keras.layers.Dense(num_classes, activation="softmax")
Purpose: A fully connected layer with num_classes (10) units. Each unit represents a class probability.
Activation Function: Softmax converts the output into a probability distribution over the 10 classes.
```

[9]:
```
# We use the compile() method to specify the optimizer, loss function,
# and the metrics to monitor. Note that with the JAX and TensorFlow
# backends, XLA compilation is turned on by default.

model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="acc"),
    ],
)
```

[*]:
```
# Let's train and evaluate the model.
# We'll set aside a validation split of 15% of the data during training
# to monitor generalization on unseen data.

batch_size = 128
epochs = 20

callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="model_at_epoch_{epoch}.keras"),
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=2),
]

model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.15,
    callbacks=callbacks,
)
score = model.evaluate(x_test, y_test, verbose=0)
```

The provided code snippet outlines the process to train and evaluate the CNN model. Here's a detailed step-by-step explanation of each part:

1. Set Training Parameters

python

Copy code

```
batch_size = 128
epochs = 20
```

batch_size: The number of samples processed before the model is updated. Here, 128 samples are used per batch.

epochs: The number of times the entire training dataset will pass through the model. Here, the model will train for 20 epochs.

2. Define Callbacks

python

Copy code

```
callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="model_at_epoch_{epoch}.keras"),
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=2),
]
```

ModelCheckpoint: Saves the model's weights at the end of each epoch. The filepath specifies where to save the model and includes {epoch} to save different files for each epoch.

EarlyStopping: Stops training early if the validation loss does not improve for a specified number of epochs (patience=2). This helps prevent overfitting and reduces unnecessary computation.

3. Train the Model

python

Copy code

```
model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.15,
    callbacks=callbacks,
)
```

x_train and y_train: The training data and labels.

batch_size: The number of samples processed before updating the model weights.

epochs: The total number of epochs to train the model.

validation_split: A fraction of the training data (15%) used for validation during training. This helps monitor the model's performance on unseen data while training.

callbacks: List of callbacks to apply during training. Here, it includes saving the model and early stopping.

4. Evaluate the Model

python

Copy code

```
score = model.evaluate(x_test, y_test, verbose=0)
```

x_test and y_test: The test data and labels.

verbose=0: Suppresses detailed output during evaluation.

```
[*]: # During training, we were saving a model at the end of each epoch.
     # You can also save the model in its latest state like this:
     model.save("final_model.keras")

     # And reload it like this:
     model = keras.saving.load_model("final_model.keras")

     # Next, you can query predictions of class probabilities with predict():
     predictions = model.predict(x_test)
```