Neural Networks: Tensorflow

Andrzej Kordecki

Neural Networks for Classification and Identification (ML.EM05): Exercise 10
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Table of Contents

Tensorflow + Keras

Tensorflow

TensorFlow is an open-source machine learning library for research and production.

- allows to perform specific machine learning operations on huge matrices with large efficiency,
- support of multi CPU cores, GPU cores, or even multiple devices like multiple GPUs.
- provides APIs for Python, C++, Haskell, Java, Go, Rust, and R.

Tensorflow

Importing library into Python code:

```
import tensorflow as tf
```

Tensorflow webpage:

http://www.tensorflow.org

Tutorials:

https://www.tensorflow.org/tutorials/

Keras

Keras is a high-level API to build and train deep learning models. It's used for fast prototyping, advanced research, and production, with three key advantages:

- User friendly and simple interface optimized for common use cases.
- Easy to extend. Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models. Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- The library can be used separately, but the Tensorflow-integrated library will be discussed during exercise.

Keras

Importing library into Python code:

```
import keras
# OR
from tensorflow import keras
```

Keras webpage:

http://keras.io

Tutorials:

https://keras.io/#guiding-principles

Tensorflow

Computations in TensorFlow are done with data flow graphs:

- Nodes represent mathematical operations,
- In TensorFlow, there are two types of edge:
    - Normal Edges - they are carriers of data structures (tensors), where an output of one operation becomes the input for another operation.
    - Special Edges - these edges are not data carriers between the output and the input. A special edge indicates a control dependency between two nodes.
- Communication is between these nodes with use of the edges.

Build a simple NN solution

Build a neural network need to:

1. Load input dataset,
2. Set neural network model,
3. Train model,
4. Evaluate results.

All examples use in header:

```
import tensorflow as tf
from tensorflow import keras
```

Input data

Loading data - directly from files (small weight datasets):

1. Read dataset from folder path or read with use of library, e.g.

   ```
   images = tf.keras.datasets.mnist
   ```

2. Data scaling to fixed range of values (mostly $< 0, 1 >$ or $< -1, 1 >$), e.g.

   ```
   (x_train, y_train), (x_test, y_test) =
       images.load_data()
   x_train, x_test = x_train / 255.0, x_test / 255.0
   ```

All import data must have the same format and type.

Input data

Loading data - Tensorflow data generator:

1. Define data generator (data augmentation)

```
# Scaling image value to range $<0,1>$
image_gen = ImageDataGenerator(
            rescale=1./255)
```

2. Define input data path and data reading order

```
# We set directory to images and image scaling
data_gen = image_gen.flow_from_directory(
            batch_size=batch_size,
            directory=train_dir,
            shuffle=True,
            target_size=(IMG_HEIGHT, IMG_WIDTH))
```

The structure of the data folders is very important.

Input data

The ImageDataGenerator selected arguments:

- rescale - rescaling factor,
- rotation_range - degree range for random rotations,
- horizontal_flip (vertical_flip) - randomly flip inputs horizontally/vertically,
- zoom_range - range for random zoom
- brightness_range - range for picking a brightness shift value change,
- width_shift_range / height_shift_range - pixels or percent of image width/height random change.

Build a simple NN

Building a neural network model:

1. Define neural network type, e.g.

```
# Linear stack of layers
model = keras.Sequential()
```

2. Add input layer of network, e.g.

```
# input_shape is a shape tuple of
# input image of size 28x28,
# the batch dimension is not included
model.add(tf.keras.layers.Flatten(
                                input_shape=(28, 28)))
```

Build a simple NN

Building a neural network model:

3. Add all hidden layers of the network, e.g.

```
# 3D convolution layer
model.add(tf.keras.layers.Conv2D(
                32, 3, activation='relu'))

# Fully connected layer
model.add(tf.keras.layers.Dense(
                64, activation='relu'))

# Dropout layer - generalization
model.add(tf.keras.layers.Dropout(0.5))
```

Build a simple NN

Building a neural network model:

4. Add Output part of the network in accordance to the goal of network, e.g. neural network for image classification into 5 classes:

```
# Classification part of network
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(
            512, activation='relu'))

# Output layer with 5 neurons
# - each for one class
model.add(tf.keras.layers.Dense(
            5, activation='sigmoid'))
```

Build a simple NN

Keras layers selected arguments:

- units - dimensionality of the output space (number of neurons or layer resolution).
- activation - is the element-wise activation function.
- use_bias - whether the layer uses a bias vector.
- activation - Set the activation function for the layer.
- kernel_initializer and bias_initializer - the initialization schemes that create the layer's weights (kernel and bias).
- kernel_regularizer and bias_regularizer - the regularization schemes that apply the layer's weights, such as L1 or L2 regularization (kernel and bias).

Most of layer parameters have default value.

Build a simple NN

Layers examples:

```
# Create a sigmoid layer:
layers.Dense(64, activation='sigmoid')
layers.Dense(64, activation=tf.sigmoid)

# A linear layer with L1 regularization of
# factor 0.01 applied to the kernel matrix:
layers.Dense(64,
    kernel_regularizer=keras.regularizers.l1(0.01))

# A linear layer with a kernel initialized
# to a random orthogonal matrix:
layers.Dense(64, kernel_initializer='orthogonal')
```

Build a simple NN

Whole neural network model for image classification - Part 1:

```
model = Sequential([
    # Input layer
    tf.keras.layers.Conv2D(16, 3, activation='relu',
             input_shape=(IMG_HEIGHT, IMG_WIDTH ,3)),

    # Feature extraction part
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(64, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
```

Build a simple NN

Whole neural network model for image classification - Part 2:

```
    # Classification part
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),

    # Output layer
    tf.keras.layers.Dense(5, activation='sigmoid')
])
```

In most networks, we can distinguish certain segments responsible for individual local tasks of network. To print network structure we can use:

```
model.summary()
```

Model training

We can configure learning process by calling the compile and fit method:

```
# Model was defined before as model = Sequential([.])
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels, epochs=10, batch_size=32)
```

Compile function set:

- Loss function,
- Optimization algorithm,
- Evaluation measure calculated during training.

Fit method set information about input data.

Model training

Tensorflow compile function selected arguments:

- optimizer - object specifies the training procedure. Pass it optimizer instances from the tf.train module, such as AdamOptimizer, RMSPropOptimizer, or GradientDescentOptimizer.
- loss - function to minimize during optimization. Common choices include: mean square error (mse), categorical_crossentropy, and binary_crossentropy. Loss functions are specified by name or by passing a callable object from the tf.keras.losses module.
- metrics - used to monitor training. These are string names or callables from the tf.keras.metrics module.

We can set custom loss function and metrics.

Model training

Examples of configuring a model for training:

```
# Configure a model for mean-squared error regression.
model.compile(optimizer=tf.train.AdamOptimizer(0.01),
        loss='mse',        # mean squared error
        metrics=['mae'])   # mean absolute error

# Configure a model for categorical classification.
model.compile(
        optimizer=tf.train.RMSPropOptimizer(0.01),
        loss=keras.losses.categorical_crossentropy,
        metrics=['accuracy','mse'])
        # accuracy based on confusion matrix
```

Model training

The model fit (or fit_generator) function is used to start training:

```
# Data for classification into 10 classes
data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))
model.fit(data, labels, epochs=10)
```

Fit function set:

- input data (with desired output values),
- validation data,
- size of input data batch,
- number of epochs or additional stop criterion,
- storing information about training process.

Model training

Examples of fit functions:

```
model.fit(x=x_train,
          y=y_train,
          epochs=5,
          batch_size=32,
          validation_data=(x_test, y_test))
model.fit_generator(
    train_data_gen,
    steps_per_epoch=total_train // batch_size,
    validation_data=val_data_gen,
    validation_steps=total_val // batch_size,
    epochs=15,
    callbacks=[cp_callback])
```

Model training

Tensorflow fit function selected arguments:

- epochs - training is structured into epochs. An epoch is one iteration over the entire input data (this is done in smaller batches).
- batch_size - specifies the size of each batch. The model slices the data into smaller batches and iterates over these batches during training.
- validation_data - allows to easily monitor its performance on neural network. Passing this argument—a tuple of inputs and labels—allows the model to display the loss and metrics in inference mode for the passed data, at the end of each epoch.

Model training

Stop criterio:

- Number of epochs (default),
- Minimum change in the monitored quantity (mostly loss function).

We can stop training before set number of epoch by monitoring changes in loss function (or other selected quantity) values:

```
earlystopper = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', mode='auto', patience=15)
model.fit(train_data_gen,
    steps_per_epoch=total_train // batch_size,
    epochs=15,
    callbacks=[cp_callback, earlystopper])
```

Model training

Function EarlyStopping selected arguments:

- monitor - quantity to be monitored.
- min_delta - minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change..
- patience: number of epochs that produced the monitored quantity with no improvement after which training will be stopped.
- mode: one of auto, min, max. In min/max mode, training will stop when the quantity monitored has stopped decreasing/increasing.
- restore_best_weights: whether to restore model weights from the epoch with the best value of the monitored quantity.