Neural Networks: Tensorflow model operation

Andrzej Kordecki

Neural Networks for Classification and Identification (ML.EM05): Exercise 12
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

Improving network training process

Improving network training process

Improving network training process of big neural network models:

- Early stopping criterion.
- Learning rate and momentum methods.
- Generalization methods.

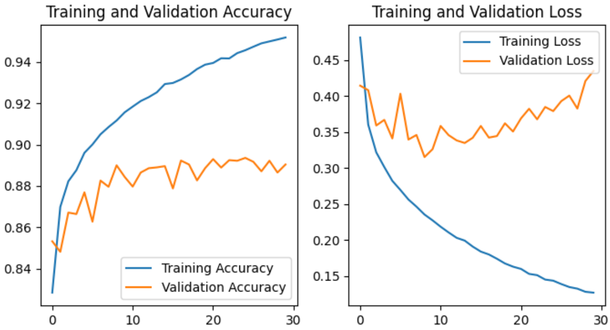
Big model classification structure from previous exercises.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(10, activation=None)
])
```

Improving network training process

The obtained results:

```
Epoch 30/30
loss: 0.1774 - accuracy: 0.9343 \
- val_loss: 0.4009 - val_accuracy: 0.8786
```



Early Stopping

The early stopping function allows you to specify the performance measure to monitor in order to end training before assumed number of epochs.

- Training will stop when the chosen performance measure stops improving.
- The “mode” argument will need to be specified as whether the objective of the chosen metric is to increase or to decrease (e.g. different for accuracy and loss).
- The calculation of measures on the validation dataset will have the ‘val_‘ prefix, such as ‘val_loss‘ for the loss on the validation dataset.

Early Stopping

Stop training when a monitored metric has stopped improving.

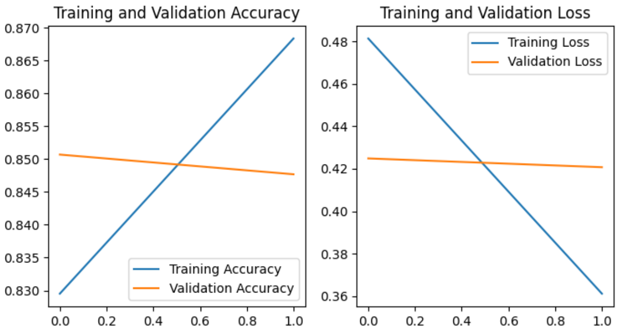
```
es = tf.keras.callbacks
            .EarlyStopping(monitor='val_loss',
                           min_delta=0,
                           patience=0,
                           mode='auto',
                           restore_best_weights=False,
                           verbose=1)

history = model.fit(train_images, train_labels,
                    epochs=30,
                    validation_data=(test_images,
                                     test_labels),
                    callbacks=es)
```

Early Stopping

The neural training is not always a smooth training and may end much faster than we would expect. The results:

```
Epoch 00002: early stopping
loss: 0.3612 - accuracy: 0.8684
- val_loss: 0.4207 - val_accuracy: 0.8477
```



Early Stopping

Importnat parameter to change is "patience" and "restore_best_weights".

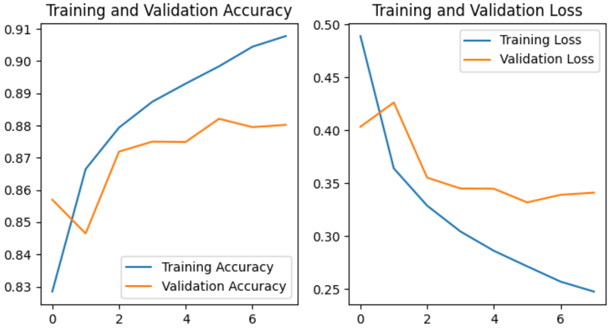
```
es = tf.keras.callbacks
              .EarlyStopping(monitor='val_loss',
                             min_delta=0,
                             patience=2,
                             mode='min',
                             restore_best_weights=True,
                             verbose=1)
```

The value of the "min_delta" parameter depends on knowledge of model characteristics.

Early Stopping

The results of new early stopping settings:

```
Epoch 00008: early stopping. Restoring model
weights from the end of the best epoch: 6.
Epoch 6/30: loss: 0.2715 - accuracy: 0.8984
- val_loss: 0.3320 - val_accuracy: 0.8821
```



Checkpoint model saving

ModelCheckpoint is a callback function used to save the Keras model or model weights at some frequency.

```
# Path set also file format [cpkt, h5]
cp_path = "training/cp-{epoch:04d}.h5"

# Create a callback that saves the whole model
mc = tf.keras.callbacks
              .ModelCheckpoint(filepath=cp_path,
                               monitor='val_loss',
                               mode='min',
                               save_weights_only=False,
                               save_freq='epoch',
                               verbose=1)
```

Checkpoint model saving

We have to always remember to make changes also in fit function

```
history = model.fit(train_images, train_labels,
                    epochs=30,
                    validation_data=(test_images,
                                     test_labels),
                    callbacks=[es, mc])
```

We are using both early stopping and model checkpoint saving.

Checkpoint model saving

The results:

```
Epoch 1/30
1868/1875 [==========>.] - ETA: 0s - loss: 0.4849 ...
Epoch 00001: saving model to training\cp-0001.h5
1875/1875 [==========>.] - 4s 2ms/step - loss: 0.4849 .
Epoch 2/30
1858/1875 [==========>.] - ETA: 0s - loss: 0.3653 ....
Epoch 00002: saving model to training\cp-0002.h5
1875/1875 [==========>.] - 3s 2ms/step - loss: 0.3652 .
Epoch 3/30
...
```

Checkpoint model saving

Loading model:

```
new_model = tf.keras.models.load_model(
                            "training/cp-0007.h5")

test_loss, test_acc = new_model.evaluate(test_images,
                            test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

It always a good practice to check loaded model accuracy.

Learning rate and momentum

Learning rate and in back-propagation algorithm:

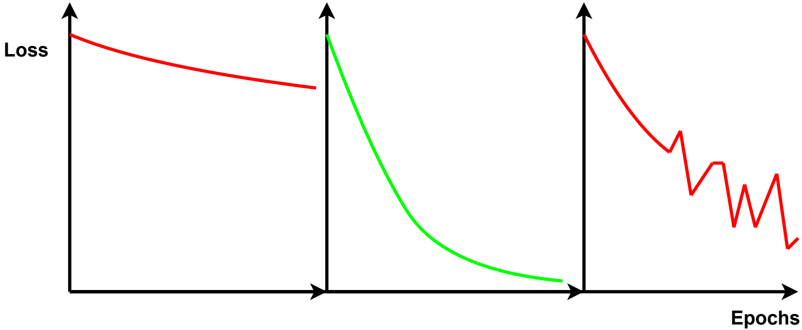
- The amount that the weights are updated during training is referred to as the step size or the "learning rate."
- The amount that previous changes in the weights should influence the current direction of movement in weight space is referred to as the momentum.

Both momentum and learning rate:

- They are a configurable hyperparameters used in the training of neural networks that has a small positive value.
- Both are directly implemented in the revised weight-update rule.
- The parameter value affects the calculation time and the accuracy of the obtained solution

Learning rate and momentum

We want to set learning rate and momentum to achieve a steep decrease in the model loss.



This task is not as trivial as to find one optimal value.

Learning rate and momentum

The training parameters in Tesnorflow are set in model function fit:

```
model.compile(optimizer='SGD',
              loss=tf.keras.losses.MSE)
```

The optimization method is defined by "optimizer" argument e.g.:

- class Adam: Optimizer that implements the Adam algorithm.
- class RMSprop: Optimizer that implements the RMSprop algorithm.
- class SGD: Gradient descent (with momentum) optimizer.

Learning rate and momentum

The each "optimizer" class have a different set of arguments and its default values. In case of stochastic gradient descent we can set:

```
opt = tf.keras.optimizers.SGD(lr=0.001, momentum=0.9)

model.compile(optimizer=opt,
              loss=tf.keras.losses
                .SparseCategoricalCrossentropy(
                                  from_logits=True),
              metrics=['accuracy'])
```

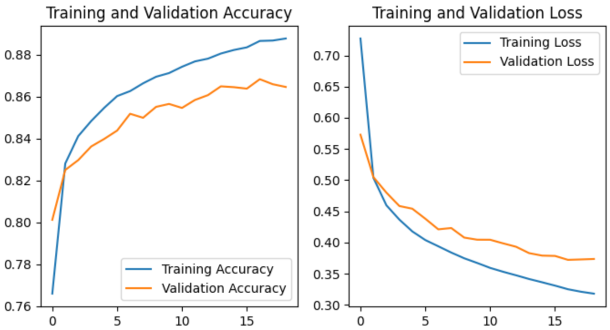
Update rule for parameter w with gradient g:

```
velocity = momentum * velocity - learning_rate * g
w = w + velocity
```

Learning rate and momentum

The results:

```
Epoch 00019: early stopping. Restoring model
weights from the end of the best epoch: 17.
loss: 0.3251 - accuracy: 0.8866
- val_loss: 0.3722 - val_accuracy: 0.8683
```



Learning rate and momentum

Learning rate schedule allows to adjust the learning rate for a given number of training epochs

```
def scheduler(epoch, lr):
  if epoch < 10:
    return 0.9*lr
  else:
    return 0.001

lrs = tf.keras.callbacks
              .LearningRateScheduler(scheduler)

opt = tf.keras.optimizers.SGD(lr=0.01, momentum=0.9)
```

Learning rate and momentum

The changes must be also in the model fit function.

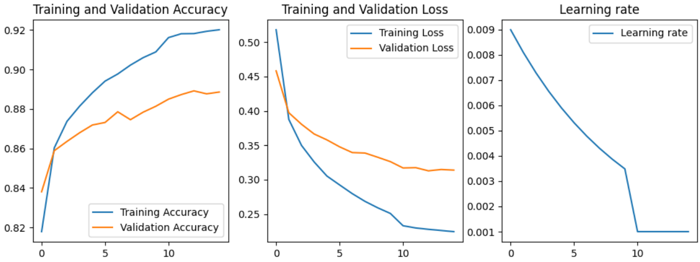
```
model.compile(optimizer=opt,
              loss=tf.keras.losses
                     .SparseCategoricalCrossentropy(
                                    from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels,
                    epochs=30,
                    validation_data=(test_images,
                                     test_labels),
                    callbacks=[es, mc, lrs])

lr_train = history.history['lr'] # Lr history
```

Learning rate and momentum

The results:



The learning rate settings are useful, if we have already trained model at least few times.

Learning rate and momentum

Adam optimization is based on adaptive estimation of first-order and second-order moments.

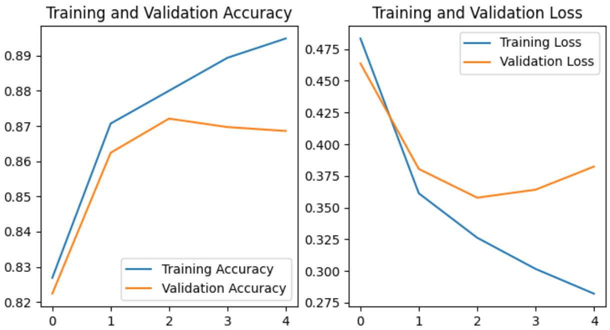
```
model.compile(optimizer=tf.keras.optimizers.Adam(
                learning_rate=0.001,
                beta_1=0.9,
                beta_2=0.999),
              loss=tf.keras.losses
                .SparseCategoricalCrossentropy(
                from_logits=True),
              metrics=['accuracy'])
```

The calculation without use of learning rate schedule.

Learning rate and momentum

The results:

```
Epoch 00005: early stopping. Restoring model
weights from the end of the best epoch: 3.
loss: 0.3262 - accuracy: 0.8800
- val_loss: 0.3578 - val_accuracy: 0.8721
```



Generalization methods

A "simple model" is a model with a small number of parameters or distribution of parameter values has small entropy.

- Weight regularization. We can simplify complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". The method modify loss function of the network by adding a penalty cost function associated with having large weights.
- L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients.
- L2 regularization, where the cost added is proportional to the square of the value of the weights coefficients (weight decay).

Generalization methods

Dropout is one of most commonly used regularization techniques.

- Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training.
- The "dropout rate" is the fraction of the features that are being zeroed-out. It is usually set between 0.2 and 0.5.
- In tf.keras you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before.

Generalization methods

Adding both weight regularization and momentum with change of activation function.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(256, activation='elu',
             kernel_regularizer
                =tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='elu',
             kernel_regularizer
                =tf.keras.regularizers.l2(0.01))
])
```

Generalization methods

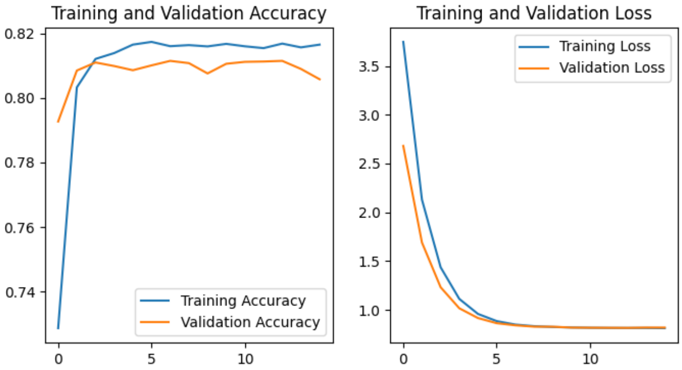
The training settings.

```
opt = tf.keras.optimizers.SGD(lr=0.001, momentum=0.9)

model.compile(optimizer=opt,
              loss=tf.keras.losses
                .SparseCategoricalCrossentropy(
                                    from_logits=True),
              metrics=['accuracy'])
```

Generalization methods

The influence of generalization method on training process.



Transfer learning

Transfer learning is a method where a model developed for a task is reused as the starting point for a model on a second task. Keras library have many neural networks model alongside with pre-trained weights.

Transfer learning application:

- Prediction,
- Feature extraction,
- Fine tuning.

Transfer learning

Image classification with trnsfered VGG16 network.

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16
                                  import preprocess_input
import numpy as np

model = VGG16(weights='imagenet')
img_path = 'bird.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
preds = model.predict(x)
```

Transfer learning

Transfer learning function arguments depend on network model.

- *include_top*: whether to include the fully-connected layers at the top of the network.
- *weights*: one of None (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.

The changes in network structure for fine tuning (e.g. different number of image classes).

Transfer learning

Fine-Tuning:

1. Freeze the convolutional base of transferred model.
2. Add classifier part (layers) to the last layers of the base model.
3. Unfreeze a few of the top layers of a frozen model base (optional) and jointly train both the newly-added classifier layers and the last layers of the base model.

Freezing prevents the weights in a given layer from being updated during training.

```
layer.trainable = False # one layer
```

This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

Transfer learning

We can freeze all layers of predefined base model:

```
base_model = tf.keras.applications
                .MobileNetV2(input_shape=IMG_SHAPE,
                             include_top=False,
                             weights='imagenet')
base_model.trainable = False # whole base model
```

This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

Transfer learning

The whole new model can be build:

```
inputs = tf.keras.Input(shape=(160, 160, 3))
x = base_model(inputs, training=False)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```