Manual training with TensorFlow

October 6, 2025

This notebook demonstrates the implementation of a churn prediction model using a manual training loop with `tf.GradientTape`.

Although this approach is not part of the main project workflow, it is included to illustrate an advanced technique in model training and optimization.

This approach provides insight into how weights are manually updated, how the loss is computed, and how the optimizer is applied.

Although `Keras` abstracts these steps, understanding them in detail is valuable for custom architectures, non-standard loss functions, or deep debugging.

[12]:
```
import numpy as np
import pandas as pd
```

[14]:
```
# For this Project Appendix the original training and test sets were exported␣
↪from the main notebook 'Churn prediction with Keras.ipynb'

# X
Xtrain = np.load(r"C:\Users\Usuario\Xtrain_scaled.npy")
Xtest = np.load(r"C:\Users\Usuario\Xtest_scaled.npy")

# y
y_train = pd.read_csv(r"C:\Users\Usuario\y_train.csv").values.ravel()
y_test = pd.read_csv(r"C:\Users\Usuario\y_test.csv").values.ravel()
```

[16]:
```
# We define the model with the same parameters as the original one

import tensorflow as tf
from tensorflow.keras import Input, Sequential
from tensorflow.keras.layers import Dense

n_features = Xtrain.shape[1]

model = Sequential([
    Input(shape=(n_features,)),
    Dense(10, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Manual Configuration

Training with a manual loop provides full control over every step of the learning process—from the forward pass and loss computation to backpropagation and weight updates.

It allows for seamless integration of custom logic, tailored metrics, and advanced techniques such as multiple optimizers, direct gradient manipulation, conditional computations, and non-standard loss functions.

This approach is especially valuable in research, experimentation, or scenarios where the standard training pipeline is insufficient

[54]:
```
# Manual configuration

loss_fn = tf.keras.losses.BinaryCrossentropy() # This function measures the␣
↪difference between the actual probability distribution (labels 0 or 1)
                                                # and the probability predicted␣
↪by the model.

optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3) # The same learning␣
↪rate we used in .compile() on the main notebook.

train_ds = tf.data.Dataset.from_tensor_slices((Xtrain, y_train)).batch(32) #␣
↪This creates a dataset that produces 32 samples with tag per iteration.
```

[42]:
```
# To see how the weights were calculated for this particular dataset
# please refer to '3.1) Importance of class weight' on the main Notebook.

class_weights = {0: 0.6810433884297521, 1: 1.8808844507845934}
```

[60]:
```
# Training Loop with GradientTape

for epoch in range(30): # Using 30 epochs as the original
    epoch_loss = 0
    for step, (x_batch, y_batch) in enumerate(train_ds): # Every step will␣
↪obtain a batch with 'x_batch' caracteristics, and its tags 'y_batch'.

        with tf.GradientTape() as tape: # This opens a context for recording␣
↪automatic gradient calculation
                                        # (this will be need for the␣
↪backpropagation training).

            logits = model(x_batch, training=True) # obtains the predictions (or␣
↪logits).
            logits = tf.squeeze(logits)  # Asures compatible form.

            loss = loss_fn(y_batch, logits) # calculates loss comparing true␣
↪labels agains the predictions.
```

```
            weights = tf.where(y_batch == 1, class_weights[1], class_weights[0])␣
↪# Creates a vector of weights per class for every batch.
            weighted_loss = tf.reduce_mean(loss * weights) # calculates de␣
↪average of every element on the tensor
                                                            # formed by the␣
↪vectors: loss & weights.

        grads = tape.gradient(weighted_loss, model.trainable_weights) #␣
↪calculates loss gradients with respect of the trainable weights.
        optimizer.apply_gradients(zip(grads, model.trainable_weights)) # applies␣
↪the calculates gradients to the traing weights in every epoch.
        epoch_loss += weighted_loss.numpy() # accumulates the further obtained␣
↪the average loss per epoch.

    print(f"Epoch {epoch+1}, Loss: {epoch_loss/len(train_ds):.4f}") # loss per␣
↪epoch
```

```
Epoch 1, Loss: 0.4385
Epoch 2, Loss: 0.4357
Epoch 3, Loss: 0.4334
Epoch 4, Loss: 0.4316
Epoch 5, Loss: 0.4301
Epoch 6, Loss: 0.4287
Epoch 7, Loss: 0.4274
Epoch 8, Loss: 0.4264
Epoch 9, Loss: 0.4255
Epoch 10, Loss: 0.4247
Epoch 11, Loss: 0.4240
Epoch 12, Loss: 0.4233
Epoch 13, Loss: 0.4227
Epoch 14, Loss: 0.4221
Epoch 15, Loss: 0.4215
Epoch 16, Loss: 0.4211
Epoch 17, Loss: 0.4207
Epoch 18, Loss: 0.4203
Epoch 19, Loss: 0.4198
Epoch 20, Loss: 0.4195
Epoch 21, Loss: 0.4191
Epoch 22, Loss: 0.4188
Epoch 23, Loss: 0.4186
Epoch 24, Loss: 0.4183
Epoch 25, Loss: 0.4180
Epoch 26, Loss: 0.4177
Epoch 27, Loss: 0.4174
Epoch 28, Loss: 0.4172
Epoch 29, Loss: 0.4169
```

Epoch 30, Loss: 0.4167

Evaluation of the Model

[79]:
```
# Metrics and prediction

from sklearn.metrics import confusion_matrix, roc_auc_score,␣
↪precision_recall_curve, auc

probs = model.predict(Xtest)[:,0]
preds_bin = (probs > 0.5).astype(int)

print("probs min/max/mean:", probs.min(), probs.max(), probs.mean())
print("Unique binary preds and counts:", np.unique(preds_bin,␣
↪return_counts=True))
print("Confusion matrix:\n", confusion_matrix(y_test, preds_bin))
roc = roc_auc_score(y_test, probs)
print("ROC AUC:", roc)
prec, rec, thr = precision_recall_curve(y_test, probs)
print("PR AUC:", auc(rec, prec))
```

```
probs min/max/mean: 0.003331948 0.9032699 0.24757144
Unique binary preds and counts: (array([0, 1]), array([1443,  315],
dtype=int64))
Confusion matrix:
 [[1192   99]
 [ 251  216]]
ROC AUC: 0.8454022826452943
PR AUC: 0.6567113634864129
```

With the manual configuration of the loop and metrics not only we obtained the same results in ROC-AUC & PR-AUC curves but also, we obtained small improvement on the values in the targeted results of the confusion matrix: True positives & False positives.

While using pure TensorFlow requires a deeper understanding of the training process, it offers insight into the internal workflow of model learning—beneficial for both customization and optimization.