FTDL — Take Home Exam 2

Fundamentals and Tools of Deep Learning, 2022–2023

Solutions

Subham Shome

subham.shome@estudiante.uam.es

IPCV 2022-24

Escuela Politécnica Superior — Universidad Autónoma de Madrid

May 18, 2023

1. **(1 pt) Indicate functions used respectively in Tensorflow and Keras for the selection of the model, the learning rate, the activation, the loss function and the optimizer.**

Ans: In TensorFlow and Keras, the functions used for various model-related tasks are as follows:

- Model Selection:
  - TensorFlow: It provides various APIs for model selection, such as `tf.keras.Sequential` for creating a linear stack of layers, or the functional API `tf.keras.Model` for more complex model architectures.
  - Keras: In Keras, the model selection is typically done using the `keras.models.Sequential` class for linear stack models or the functional API `keras.models.Model` for more complex models.
- Learning Rate:
  - TensorFlow: It offers various optimizers, and the learning rate is typically set within the optimizer. For example, the `tf.keras.optimizers.Adam` optimizer accepts a learning rate argument (`learning_rate`) that can be set explicitly.
  - Keras: In Keras, the learning rate can be set using the `lr` parameter when instantiating an optimizer.
    For example, `keras.optimizers.Adam(lr=0.001)` sets the learning rate to 0.001.
- Activation Function:

  - TensorFlow: It provides activation functions as part of the `tf.nn` module. Commonly used activation functions include `tf.nn.relu`, `tf.nn.sigmoid`, `tf.nn.tanh`, etc.
  - Keras: Keras includes activation functions within the `keras.activations` module. You can specify the activation function as a string or as a callable object when defining a layer or a model.

- Loss Function:
  - TensorFlow: It provides various loss functions in the `tf.losses` module. Examples include `tf.losses.mean_squared_error`, `tf.losses.softmax_cross_entropy`, etc.
  - Keras: Keras offers a wide range of loss functions within the `keras.losses` module. You can specify the loss function as a string or as a callable object when compiling the model.

- Optimizer:
  - TensorFlow offers various optimizers in the `tf.keras.optimizers` module. Examples include `tf.keras.optimizers.SGD`, `tf.keras.optimizers.Adam`, etc.
  - Keras: Keras provides a variety of optimizers within the `keras.optimizers` module. You can instantiate an optimizer by specifying its name as a string or by directly calling its constructor, e.g., `keras.optimizers.SGD()`, `keras.optimizers.Adam()`.

P.T.O.

2. **(1,5 pt) Write the code both in Tensorflow and Keras for building a MLP with 10 inputs, 2 outputs, 2 hidden layers of 50 units each, that uses Stochastic Gradient Descent as optimizer, Average Square error as loss function and Relu as activation function.**

Ans: Please find below the code snippet for the same.

```
import tensorflow as tf

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(10,)),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(2, activation='linear')
])

optimizer = tf.keras.optimizers.SGD()
model.compile(optimizer=optimizer, loss='mean_squared_error')

model.summary()
```

✓ 0.0s

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 50)                550

 dense_4 (Dense)             (None, 50)                2550

 dense_5 (Dense)             (None, 2)                 102

=================================================================
Total params: 3,202
Trainable params: 3,202
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: TensorFlow

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(50, activation='relu', input_shape=(10,)))
model.add(Dense(50, activation='relu'))
model.add(Dense(2, activation='linear'))

optimizer = SGD()
model.compile(optimizer=optimizer, loss='mean_squared_error')

model.summary()
```

✓ 0.0s

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 50)                550

 dense_7 (Dense)             (None, 50)                2550

 dense_8 (Dense)             (None, 2)                 102

=================================================================
Total params: 3,202
Trainable params: 3,202
Non-trainable params: 0
_________________________________________________________________
```

Figure 2: Keras

The full code is attached as a Jupyter notebook along with this file.

3. **(1,5 pt) Suppose we have a database with 100000 input samples. Write the code in Tensorflow for using mini-batches of 100 input samples each.**

Ans: Please find below the code snippet for the same.
The full code is attached as a Jupyter notebook along with this file.

```python
import tensorflow as tf
import numpy as np

data_size = 100000
input_size = 10

inputs = np.random.rand(data_size, input_size)
targets = np.random.rand(data_size, 2)
train_data = tf.data.Dataset.from_tensor_slices((inputs, targets))

batch_size = 100
training_data = train_data.batch(batch_size)

for batch_inputs, batch_targets in training_data:
    with tf.GradientTape() as tape:
        predictions = model(batch_inputs)
        loss = tf.losses.mean_squared_error(batch_targets, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

✓ 4.9s

Figure 3: Question 3

4. **(1 pt) In the same previous database, write the code for normalizing and standardizing the whole dataset.**

Ans: Please find below the code snippet for the same.

```python
import tensorflow as tf
import numpy as np

data_size = 100000
input_size = 10

inputs = np.random.rand(data_size, input_size)
targets = np.random.rand(data_size, 2)

# Normalize the input data
inputs_normalized = (inputs - np.min(inputs, axis=0)) / (np.max(inputs, axis=0) - np.min(inputs, axis=0))

# Standardize the input data
inputs_standardized = (inputs - np.mean(inputs, axis=0)) / np.std(inputs, axis=0)

train_data_normalized = tf.data.Dataset.from_tensor_slices((inputs_normalized, targets))
train_data_standardized = tf.data.Dataset.from_tensor_slices((inputs_standardized, targets))

batch_size = 100
training_data_normalized = train_data_normalized.batch(batch_size)
training_data_standardized = train_data_standardized.batch(batch_size)

# Continue with the training dataset
for batch_inputs, batch_targets in training_data_normalized:
    with tf.GradientTape() as tape:
        predictions = model(batch_inputs)
        loss = tf.losses.mean_squared_error(batch_targets, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

# Using the standardized dataset
for batch_inputs, batch_targets in training_data_standardized:
    with tf.GradientTape() as tape:
        predictions = model(batch_inputs)
        loss = tf.losses.mean_squared_error(batch_targets, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

✓ 10.0s

Figure 4: Question 4

The full code is attached as a Jupyter notebook along with this file.

5. **(5 pt) In some cases, for example in linear regression, we can obtain an explicit formula for the optimal weights.**

(a) Show that if we have the inputs as a $X \in M_{n\times m}$ matrix of n samples $x = x_1, x_2, ..., x_m$ with m features each and a n-dimensional vector $y = y_1, y_2, ..., y_n$ with the targets for each sample, the optimal values $w = w_1, w_2, ..., w_m$ for the linear regression model $y = w_1x_1 + w_2x_2 + \ldots + w_{m\times m}$ (this is $y = Xw$) using the average square error as loss function are:

$$W = (X^T X)^{-1} X^T Y$$

(Note: Provide a complete proof, using matrixes and elements of the matrixes, the general proof using Matrix Algebra provided in the lectures is not enough)

(b) Which method do you think is the most costly in computational terms for obtaining the weights for the previous problem, the explicit formula or the neural network method.

(c) What method gives the most exact value of the optimal weights?

(d) Can you always find the values of the weights using the exact formula shown in question a?

Justify your answers.

Ans:

(a) Let $X \in M_{n\times m}$ be a matrix of n samples $x = x_1, x_2, \ldots, x_m$ with m features each. Let $y = y_1, y_2, \ldots, y_n$ be an n-dimensional vector with the targets for each sample.

The linear regression model can be represented as $y = w_1x_1 + w_2x_2 + \ldots + w_mx_m$, which can be written as $y = Xw$, where $w = w_1, w_2, \ldots, w_m$.

To find the optimal values of w using the average square error as the loss function, we need to minimize the following objective function:

$$E(w) = \frac{1}{n}\|y - Xw\|^2$$

Expanding the expression and simplifying in matrix terms, we have:

$$E(W) = \frac{1}{n}(Y - XW)^T(Y - XW)$$

Let's find the w that minimizes this objective function. To do so, we differentiate $E(W)$ with respect to w, set the derivative equal to zero, and solve for W.

$$\frac{\partial E(W)}{\partial W} = \frac{1}{n}\left(\frac{\partial (Y - XW)^T(Y - XW)}{\partial W}\right)$$

Using matrix calculus identities, we have:

$$\frac{\partial E(W)}{\partial W} = \frac{1}{n}\left(-2X^T(Y - XW)\right)$$

Setting the derivative equal to zero and solving for W, we have:

$$-2X^T(Y - XW) = 0$$

Multiplying both sides by X^T, removing -2 and rearranging the terms, we get:

$$X^T XW = X^T Y$$

Solving ths equation to find W, we get:

$$W = (X^T X)^{-1} X^T Y$$

Hence proved.

(b) The explicit formula for obtaining the weights in linear regression, which involves matrix inversion, can be computationally costly. The computational complexity of matrix inversion is approximately cubic in the number of features, making it less efficient when dealing with large datasets or high-dimensional feature spaces. In contrast, the neural network method requires iterative training using optimization algorithms, which can also be computationally demanding but offers more flexibility in handling complex relationships. Overall, the explicit formula tends to be more computationally costly in terms of time complexity, especially for problems with a large number of features.

(c) The method that gives the most exact value of the optimal weights for linear regression is the explicit formula. When using the explicit formula, the optimal weights can be calculated analytically by solving the equation directly, providing an exact solution. This method computes the weights without the need for iterative optimization algorithms or approximations. However, the explicit formula requires the inverse of the matrix, which may be computationally costly for large feature spaces.

While the explicit formula provides the most exact values for the optimal weights in linear regression, if we have a well-adjusted iterative method, it is possible to achieve similar results. Iterative optimization algorithms, such as gradient descent, can converge to a high-precision solution with sufficient iterations and careful parameter tuning.

(d) No, it is not always possible to find the values of the weights using the exact formula $W = (X^T X)^{-1} X^T Y$ for linear regression. The exact formula requires the matrix $X^T X$ to be invertible, which means that it must be a full-rank matrix. However, if $X^T X$ is not invertible, which can occur in scenarios such as multicollinearity or when the number of features exceeds the number of samples, then the exact formula cannot be applied. In

such cases, alternative methods like regularization techniques (e.g., Ridge regression, Lasso regression) or numerical optimization algorithms are typically used to estimate the weights.

References

- Lecture slides
- Michael Nielsen. Neural Networks and Deep Learning. Online book, 2016