École national Supérieure d'Informatique, Algiers, Algeria
2CSSID (2025/2026)
Machine Learning (ML)

Workshop 03: Neural Networks' tools (Tensorflow)

Mr. Abdelkrime ARIES

Get familiar with a tool to create MLPs and train them.
Learn how to create custom MLPs using this tool.
Understand how this tool works by programming low level MLPs

1 Data Preparation

We will use the **satellite** dataset:

- Import the training dataset.
- Separate the input data and normalize it by dividing by **255**.
- Separate the output data and encode it in OneHot format using **LabelBinarizer** from **scikit-learn**.
- Transform **X_train** and **Y_train** to **tf.constant** with **dtype=tf.float32**.
- Redo the same with test dataset, except do not transform **Y_test** to **tf.constant**.

2 Keras

In this section, we will create a project to classify satellite images. We will try to use high level APIs to create an MLP.

2.1 Sequential model

Create a sequential model (**nn1**). Then, add:

- an input layer having a shape (number of features of the past dataset,);
- a dense layer having 10 neurons and an activation function 'relu';
- a dense layer having 10 neurons and an activation function 'relu';
- and a dense layer having number-of-classes neurons and an activation function 'softmax'.

Use its method **summary** to print its structure.

2.2 Model training

- Compile **nn1** by calling the method **compile**:
  - Use cross entropy as loss function.
  - Define Adam as an optimization function.
- Train **nn1** using its method **fit**.
  - Use **X_train** and **Y_train** as its obligatory arguments.
  - Use a number of epochs of your choice.

2.3 Model testing

- Generate the estimations of **X_test** using **nn1**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

3 High level with a custom class

We want to create a custom model which adds layers one after the other. It verifies if the layers are compatibles. Then, when finished, we cannot add any more layers (we lock it).

3.1 Custom Layer

- Create a class called **MyLayer** which inherits from **keras.layers.Dense**.
- Its obligatory arguments are: the number of inputs and the number of outputs.
- Its optional arguments are: a boolean for bias (True by default) and a string of activation function to choose from 'relu', 'sigmoid' and 'linear' ('linear' by default).
- Use **assert** to verify that both the number of inputs and outputs are more than 0.
- In this case the number of outputs is the number of units (neurons). To specify the number of inputs, call the method **build((number of inputs,))**.
- You have some unitary tests, adapt them to your model to test it.

3.2 Custom Net

- Create a class called **MyMLP** which inherits from **keras.Model**.
- It has no arguments; but, it contains 2 attributes: a lock initialized to False and a list to store our layers.

- Add a method **add_layer** which has one argument: a layer.
  - It raises an exception when the object is locked.
  - It raises an exception when the previous layer's output is not as the current layer's input.
  - In case this is the first layer to add, no need to check.
  - It adds the layer when all is right.
  - Then, it returns **self**; this allows us to call this method in one line
- Add a method **compile** which has these optional arguments: number of inputs of the output layer (=1), number of outputs of the output layer (=1), bias of the output layer (=True), a boolean indicating if it is a multi-class classification (=False), and learning rate (=1.).
  - If there are past layers, it modifies the number of inputs of the output layer to be the output of the last layer.
  - The output layer will have a sigmoid activation function
  - If it is multi-class and the number of outputs is more than 1, this function must be softmax.
  - The loss function must be BCE.
  - If it is multi-class and the number of outputs is more than 1, The loss function must be CE.
  - The optimizer must be Adam, taking this model's parameters and the bias.
  - Lock the model.
- Add the **forward** method taking **X** as argument.
- Override **__call__** which returns the backward pass's result.

3.3 Model training

- Create a model (**nn2**) composed of:
  - a hidden layer having a shape (number of features of the past dataset, 10) and ReLU activation function;
  - a hidden layer having a shape (10, 10) and a ReLU activation function;
  - an output layer having a shape (10, number of classes) for multi-class classification.
- Call **summary** to print its structure.
- Fit it on **satellite** train dataset.

3.4 Model testing

- Generate the estimations of **X_test** using **nn2**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

4 Low Level

4.1 Activation functions

- Create functions **simple_sigmoid**, **simple_ReLU** and **simple_softmax** which takes **X** as argument.
- You have to use **tf** functions.

4.2 Loss functions

- Create classes **SimpleBCE** and **SimpleCE** which inherit from **keras.Loss**.
- No constructor is required.
- Add the **call** method which takes **X, Y** as arguments.

4.3 Optimization functions

- Create a class **SimpleGD** which inherits from **keras.Optimizer**.
- The constructor takes **lr** as argument.
- It passes it to the super class as **super().__init__(learning_rate=lr)**
- Override the **apply_gradients** method which takes one argument: **grads_and_vars**.
  - The argument is a list of tuples (grads, vars).
  - Update the parameters (vars) using their method **assign_sub** which is the original value minus the value in the arguments.

4.4 Custom Layer

- Copy **MyLayer** and rename it **SimpleLayer**, but it must inherit from **object**.
- Create an attribute **trainable_weights** which is a list to store the variables.
- Create an attribute **W** of type **tf.Variable** initialized to zeros. Add it to **trainable_weights**.
- Create an attribute **W** initialized to zeros. It can be only transformed to **tf.Variable** and stored in **trainable_weights** if bias=True. In this case, it will be subscribed into the module's parameters; thus can be recovered using the method **parameters**.

- Replace all activation functions by our simple versions.
- Add a method **randomize** which randomizes all trainable parameters using a normal law (**tf.random.normal**) having a mean of 0 and a standard deviation of 1. Try to update them using **assign** method.
- Add the **forward** method.
- Override **__call__** to return the result of the past method.

4.5 Custom Net

- Copy **MyMLP** and rename it into **SimpleMLP**, but it must inherit from **object**.
- Create an attribute **trainable_weights** which is a list to store the variables.
- Modify it to handle our new structures.
- Add a method **randomize** which randomizes all layers.
- Add a method **compile** which has these optional arguments: number of inputs of the output layer (=1), number of outputs of the output layer (=1), bias of the output layer (=True), a boolean indicating if it is a multi-class classification (=False), and learning rate (=1.).
  - If there are past layers, it modifies the number of inputs of the output layer to be the output of the last layer.
  - The output layer will have a sigmoid activation function
  - If it is multi-class and the number of outputs is more than 1, this function must be softmax.
  - The loss function must be BCE.
  - If it is multi-class and the number of outputs is more than 1, The loss function must be CE.
  - The optimizer must be Adam, taking this model's parameters and the bias.
  - Lock the model.
  - Update the **trainable_weights**.
- Add the **forward** method taking **X** as argument.
- Add the **backward** method taking **X, Y** as arguments:
  - In the context of a **tf.GradientTape()** called **tape**, perform a forward pass to get the predicted classes. Calculate the error using the **loss** function stored as an attribute.
  - Exit the **tape** context and compute the gradients using **tape.gradient** with the error and the list of parameters as arguments.
  - Use the optimizer to apply the gradients (method **apply_gradients**) which takes the gradients and the zipped list of parameters as arguments.
  - Return the overall error as numpy with **.numpy()**.
- Add a **fit** method taking **X, Y, epochs** as arguments. For each iteration, it shows the cost.
- Override **__call__** which returns the backward pass's result.

4.6 Model training

- Create a model (**nn3**) composed of:
  - a hidden layer having a shape (number of features of the past dataset, 10) and ReLU activation function;
  - a hidden layer having a shape (10, 10) and a ReLU activation function;
  - an output layer having a shape (10, number of classes) for multi-class classification.
- Randomize it.
- Print its parameters. The method **parameters** returns an iterator, so you have to cast it into a list.
- Fit it on **satellite** train dataset.

4.7 Model testing

- Generate the estimations of **X_test** using **nn3**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

Appendix

- pandas.read_csv: `https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html`
- sklearn.preprocessing.LabelBinarizer: `https://scikit-learn.org/1.5/modules/generated/sklearn.preprocess` `LabelBinarizer.html`
- tf.constant: `https://www.tensorflow.org/api_docs/python/tf/constant`

- tf.keras.Sequential: https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
- tf.keras.Input: https://www.tensorflow.org/api_docs/python/tf/keras/Input
- tf.keras.layers.Dense: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
- tf.keras.optimizers.Adam: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam
- tf.keras.losses.CategoricalCrossentropy: https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy
- sklearn.metrics.classification_report: https://scikit-learn.org/1.5/modules/generated/sklearn.metrics.classification_report.html
- assert: https://docs.python.org/3/reference/simple_stmts.html#assert
- tf.keras.Model: https://www.tensorflow.org/api_docs/python/tf/keras/Model
- Exception: https://docs.python.org/3/library/exceptions.html#Exception
- tf.keras.losses.BinaryCrossentropy: https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCr
- tf.keras.Loss: https://www.tensorflow.org/api_docs/python/tf/keras/Loss
- tf.math.exp: https://www.tensorflow.org/api_docs/python/tf/math/exp
- tf.where: https://www.tensorflow.org/api_docs/python/tf/where
- tf.reshape: https://www.tensorflow.org/api_docs/python/tf/reshape
- tf.reduce_sum: https://www.tensorflow.org/api_docs/python/tf/math/reduce_sum
- tf.reduce_mean: https://www.tensorflow.org/api_docs/python/tf/math/reduce_mean
- tf.math.log: https://www.tensorflow.org/api_docs/python/tf/math/log
- tf.keras.Optimizer: https://www.tensorflow.org/api_docs/python/tf/keras/Optimizer
- tf.Variable: https://www.tensorflow.org/api_docs/python/tf/Variable
- tf.zeros: https://www.tensorflow.org/api_docs/python/tf/zeros
- tf.random.normal: https://www.tensorflow.org/api_docs/python/tf/random/normal
- tf.matmul: https://www.tensorflow.org/api_docs/python/tf/linalg/matmul
- tf.GradientTape: https://www.tensorflow.org/api_docs/python/tf/GradientTape