Neural Network using Tensorflow.keras

**521H0489 – Hồ Hữu An**

**TensorFlow** is a robust library for creating and training machine learning models. Keras is a premium TensorFlow API package that allows users to effortlessly design and train neural network models. In this report, we will learn about layers and fundamental parameters in TensorFlow Keras.

1. **Sequential model**

**Sequential Model** enables us to create sequential neural network models. The sequential model consists of stacked layers through which data travels in one direction from start to conclusion.

Main features:

* The **Sequential** model in Keras is called "sequential" because the layers are stacked in a specific order. Data passes through the layers in the exact order in which they are declared in the model. This creates an end-to-end linear neural network architecture.
* **One input and one output**: The Sequential model has a single input and a single output. This is appropriate for situations with fixed-size input and output data.
* **Ease of use**: Sequential models are simple to employ, particularly for individuals new to deep learning. To train the model, we simply specify the layers in the proper sequence and then execute the compile and fit procedures.

1. **Layer: Dense**

**Layer Dense** is the most fundamental layer type in TensorFlow Keras. In a neural network, it is a completely connected layer (or dense layer) in which each neuron from the previous layer is linked to all neurons from the current layer. Layer Dense generates output by doing linear computations and applying an activation function.

1. **Activations**

**Activations** are functions that are used after layers in a neural network to provide non-linear results. Some examples include:

* ReLU (Rectified Linear Unit): relu(x) = max(0, x)
* Sigmoid:
* Tanh:
* Softmax:
* Softplus: softplus(x) = log(exp(x) + 1)
* Softsign:
* SELU (Scaled Exponential Linear Unit)

scale \* x if x > 0

scale \* alpha \* (exp(x) - 1) if x < 0

where alpha and scale are pre-defined constants (alpha=1.67326324 and scale=1.05070098)

* ELU (Exponential Linear Unit): with alpha > 0

x if x > 0

alpha \* exp(x) - 1 if x < 0

note: ELUs have negative values which pushes the mean of the activations closer to zero.

* Silu (or Swish): swish(x) = x \* sigmoid(x)

1. **Compile and arguments**

In TensorFlow Keras, the **compile()** function is used to **configure the model's training process**. Once the model structure and layers have been built, we must utilize the compile() function to set crucial parameters relevant to the training process, such as:

* **Optimizer**: The optimization algorithm governs how the weights of the model are modified to minimize loss.
  + **Stochastic Gradient Descent (sgd)**: optimizer='sgd'. This is a basic gradient descent optimization approach. It updates the weights by traveling in the opposite direction of the loss function's gradient.
  + **Adaptive Moment Estimation(adam):** optimizer='adam'. It is an optimization algorithm that self-tunes by combining gradient descent and momentum. To update the weights, it combines first- and second-order gradient information.
  + **Root Mean Square Propagation(rmsprop):** optimizer='rmsprop'. It is an optimization algorithm based on gradient descent with learning rate adjustment technique. It performs weight updates by dividing the learning rate by a gradient magnitude.
  + **Adaptive Gradient Algorithm(adagrad):** optimizer='adagrad'. It is an optimization algorithm that automatically adjusts the learning rate for each parameter based on the history of the gradient. It prioritizes updating parameters with smaller gradients.
  + **Adadelta:** optimizer='adadelta'.It is an optimization algorithm that automatically adjusts learning rate based on the history of the gradient, similar to Adagrad. However, Adadelta uses a different method to calculate learning rate.
  + **Adamax:** optimizer ='adamax'. Adamax is a variant of the Adam algorithm, where the infinity norm of the gradient is used instead of the quadratic norm as in Adam.
* **Loss**:
  + loss='mean\_squared\_error', calculates the mean squared error between the predicted output and the actual values.
  + loss='binary\_crossentropy', the predicted output is a probability value and is compared with the actual label.
  + loss='categorical\_crossentropy', used for multi-class classification problem, where the actual label is encoded as a one-hot vector.
  + loss='sparse\_categorical\_crossentropy', Similar to categorical crossentropy, but in this case the actual labels do not need to be encoded as one-hot vectors, but are just integer values ​​representing the classes.
  + loss='kullback\_leibler\_divergence', used in deep learning problems and processing probability distribution data.
  + loss='mean\_absolute\_error', calculates the average error between the predicted output and the actual value, but does not square the error.
* **loss\_weights:** A list or dictionary of scalar coefficients (Python floats) to weight the loss contributions of distinct model outputs is optional. loss\_weights = [1.0, 1.0, 1.0]
* **metrics:** “accuracy” “precision”, “recall”, “F1-score”, ….,list of metrics to be evaluated by the model during training and testing
* **weighted\_metrics:** “accuracy” “precision”, “recall”, “F1-score”, …., list of metrics to be evaluated and weighted by sample\_weight or class\_weight during training and testing.
* **run\_eagerly:** Boolean. If True, the forward pass of this model will never be compiled. It is advised that this be set to False when training (for best performance) and True when troubleshooting.
* **steps\_per\_execution:** Integer, to control the number of steps performed before updating the weights during model training.
* **jit\_compile:** boolean or "auto".When compiling a model, you can choose whether to use XLA compilation. jit\_compile="auto" for jax and tensorflow backends enables XLA compilation if the model supports it and disables it otherwise.
* **auto\_scale\_loss:** Boolean. If True and the model dtype policy is "mixed\_float16", the given optimizer will be wrapped in a LossScaleOptimizer, which will dynamically scale the loss to prevent underflow.

1. **Fit and arguments**

The **fit()** function in TensorFlow Keras is used to **execute the model's training process**. We use the fit() function to train the model on the training data once we have configured the model and defined crucial parameters using the compile() method.

The main parameters of the fit() method include:

* **x**: input data
* **y**: output data
* **batch\_size**: Integer or None. Samples used per gradient update. If unspecified, batch\_size will default to 32

Note: If your data is in the form of datasets, generators, or keras.utils.PyDataset instances (which produce batches), do not mention the batch\_size.

* **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire x and y data provided (unless the steps\_per\_epoch flag is set to something other than None).
* **verbose:** "auto", 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. "auto" becomes 1 for most cases. Defaults to "auto".

Note: the progress bar is not particularly useful when logged to a file, so verbose=2 is recommended when not running interactively (e.g., in a production environment).

* **Callbacks:** List of keras.callbacks.Callback instances. List of callbacks to apply during training.
* **validation\_split:**

Float between 0 and 1. Fraction of the training data to be used as validation data.

The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch

The validation data is selected from the last samples in the x and y data provided, before shuffling

* **validation\_data:** Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.
* **shuffle:** Boolean, whether to shuffle the training data before each epoch
* **class\_weight:** Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only)
* **sample\_weight:** Optional NumPy weights array for training samples, used for weighting the loss function (just during training).

can either pass a flat (1D) NumPy array with the same length as the input samples (1:1 mapping between weights and samples) or, in the case of temporal data, a 2D array with shape (samples, sequence\_length) to apply a distinct weight to each timestep of each sample.

* **initial\_epoch:** Integer. Epoch at which to start training
* **steps\_per\_epoch:** Either an integer or None. Total number of steps (sample batches) taken before declaring one epoch complete and beginning the next.
* **validation\_steps:** Only applicable if validation\_data is supplied. The total number of steps (batches of samples) to draw before ending validation at the conclusion of each epoch.
* **validation\_batch\_size:** Integer or None. Number of samples per validation batch. If unspecified, will default to batch\_size.
* **validation\_freq:** Validation data is only meaningful if it is provided. Specifies how many training epochs to run before performing a new validation run, e.g. validation\_freq=2 does validation every 2 epochs.