UNIVERSITÀ DEGLI STUDI DI MESSINA

Machine Learning Project Report

Deep Learning Classification

Guided by: Prof. Giancarlo Iannizzotto

By: Naveen Tiwari

Index:

Introduction:

Machine Learning (ML) and Deep Learning (DL) stand at the forefront of technological advancements, revolutionizing how computers learn and make decisions. At their core, these fields empower algorithms to discern patterns from data, enabling them to perform tasks without being explicitly programmed. In this project, I delve into the realms of TensorFlow and Keras to construct a deep neural network for a classification task.
The primary objective of this project is to harness the power of deep neural networks to construct a robust and accurate classification model. Through the use of TensorFlow and Keras, I aim to leverage the capabilities of modern deep learning frameworks to create a model that excels at discerning intricate patterns within input data, ultimately achieving high classification accuracy.
The project's core revolves around the development and training of a deep neural network for classification tasks. By choosing classification as the focus, I aim to showcase the versatility and power of deep learning in accurately categorizing input data. TensorFlow and Keras, with their userfriendly interfaces and powerful abstractions, provide the tools necessary to experiment with various neural network architectures and optimize the model's performance.

Key Terms:

Machine Learning:

Machine Learning, a subset of artificial intelligence, encompasses a diverse set of algorithms and techniques designed to enable machines to learn from data. Rather than relying on explicit programming, ML models iteratively learn patterns from data and improve their performance over time. Supervised learning, one of the key branches of ML, involves training models on labeled datasets, where the algorithm learns to map inputs to corresponding outputs.

Deep Learning:

Deep Learning, a specialized field within ML, introduces artificial neural networks inspired by the structure and function of the human brain. These deep neural networks, characterized by multiple layers (deep layers), excel at learning intricate representations and hierarchical features from raw data. DL has achieved remarkable success in complex tasks such as image and speech recognition, natural language processing, and, notably, classification.

NumPy:

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Classification:

Classification, a quintessential task in ML, involves categorizing input data into predefined classes or labels. It forms the basis for numerous realworld applications, including spam email filtering, image recognition, and medical diagnosis. Deep Learning, with its ability to automatically learn hierarchical features, has proven particularly effective in enhancing classification accuracy.

TensorFlow:

TensorFlow stands as a pinnacle in the realm of opensource machine learning libraries. Developed by the Google Brain team, TensorFlow facilitates the creation and training of intricate machine learning models. Its core strength lies in the concept of computational graphs, enabling efficient optimization and parallelization during the training process. TensorFlow's versatility extends to a wide array of applications, making it a cornerstone in the development of deep learning models.

Pandas:

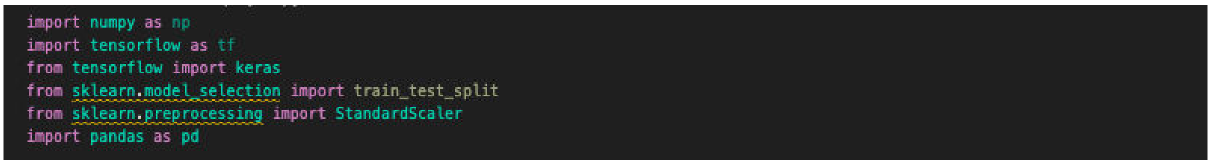
Pandas is a Python package that provides fast, flexible, and expressive data structures designed to make working with "relational" or "labelled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language. It is already well on its way towards this goal.

Keras:

Keras, built as a highlevel neural networks API, serves as a userfriendly interface atop TensorFlow. Its design philosophy revolves around simplicity and ease of use, allowing practitioners to rapidly prototype and experiment with neural network architectures. Keras abstracts the complexities associated with lowlevel operations, making it an ideal tool for both beginners and seasoned machine learning practitioners.

The synergy between TensorFlow and Keras in this implementation provides a robust foundation for constructing and training neural networks, streamlining the process of turning abstract machine learning concepts into practical, executable code.

Libraries Used:



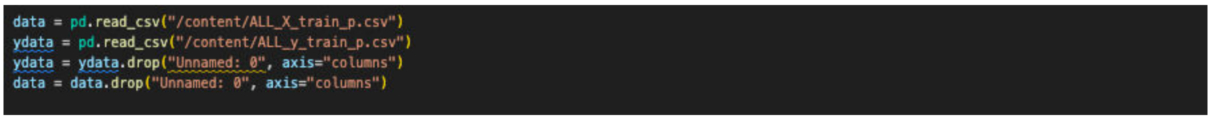
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd
```

Data Loading and Preprocessing:

The efficacy of a deep learning model heavily relies on the quality and preparation of the input data. This project emphasizes meticulous data handling to ensure a robust foundation for model training.

Data Loading:

The journey begins with the loading of training and test datasets, a crucial step to acquire representative samples for model development and evaluation. Two CSV files, “ALL_X_train_p.csv” and “ALL_y_train_p.csv”, are employed to house input features and labels, respectively.

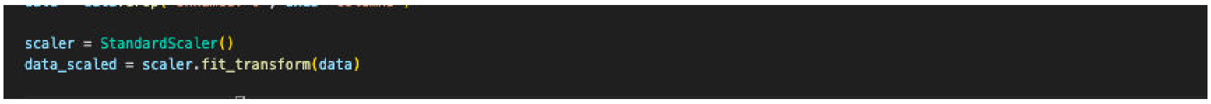


```
data = pd.read_csv("/content/ALL_X_train_p.csv")
ydata = pd.read_csv("/content/ALL_y_train_p.csv")
ydata = ydata.drop("Unnamed: 0", axis="columns")
data = data.drop("Unnamed: 0", axis="columns")
```

These datasets encapsulate the information necessary for the model to learn and generalize patterns during training.

Preprocessing:

Once loaded, the input features undergo a careful preprocessing phase. The “StandardScaler” from scikitlearn is enlisted to standardize the data, ensuring that all features share a similar scale. This step is crucial in preventing certain features from dominating the learning process due to differences in magnitude.



```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)
```

Standardization enhances the stability and efficiency of the training process, promoting a more uniform convergence of the model.

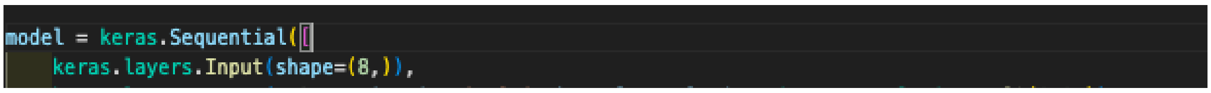
This meticulous approach to data loading and preprocessing sets the stage for a wellconditioned neural network. By providing the model with standardized and normalized data, I aim to enhance its ability to discern meaningful patterns, ultimately leading to improved classification accuracy.

Model Architecture:

The success of a deep learning project often hinges on the thoughtful design of the neural network architecture. In this implementation, I employ the Sequential API from Keras, a versatile tool for building models layer by layer.

Input Layer:

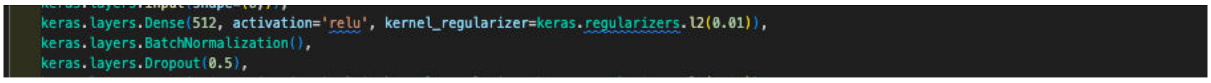
The neural network begins with an input layer, specified with the “Input” function, indicating that our data has eight features.



```
model = keras.Sequential([
    keras.layers.Input(shape=(8,)),
```

Dense Layers:

The core of the architecture consists of densely connected layers. These layers learn intricate representations from the input data through a set of weighted connections. Here, I use multiple dense layers with varying numbers of neurons and activation functions.

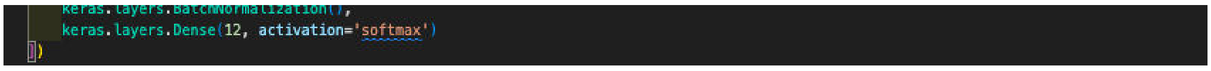


```
    keras.layers.Dense(512, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
```

The first dense layer comprises 512 neurons with rectified linear unit (ReLU) activation. Regularization is applied using L2 regularization, promoting generalization.
Batch normalization follows, enhancing stability during training.
Dropout is introduced to prevent overfitting, randomly setting a fraction of input units to zero during training.
This pattern repeats with subsequent dense layers, each contributing to the model's ability to capture and learn intricate patterns in the data.

Output Layer:

The architecture culminates in the output layer, which is tailored to the specific requirements of the classification task. For this multiclass classification problem, I utilize the softmax activation function with 12 neurons.



```
    keras.layers.BatchNormalization(),
    keras.layers.Dense(12, activation='softmax')
])
```

The softmax activation assigns probabilities to each class, allowing the model to make predictions by selecting the class with the highest probability.

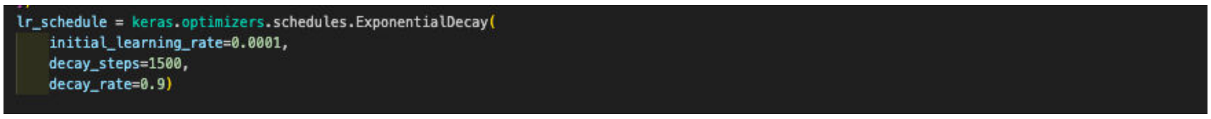
This meticulously crafted architecture, with its interconnected layers and carefully chosen activation functions, is poised to learn and extract meaningful features from the input data, paving the way for accurate and reliable classification.

Learning Rate Schedule:

The learning rate is a critical hyperparameter that influences the convergence and optimization performance of a deep learning model. Here crafted learning rate schedule is employed to strike a balance between rapid convergence and finetuning.

Exponential Decay:

The learning rate schedule implemented here utilizes the ExponentialDecay functionality from TensorFlow's optimizers. This adaptive learning rate mechanism allows the model to automatically adjust its learning rate during training.

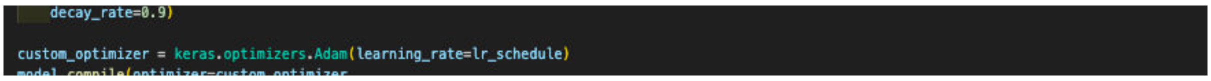


```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.0001,
    decay_steps=1500,
    decay_rate=0.9)
```

 "initial_learning_rate": The starting learning rate, set at 0.0001, provides a conservative yet efficient initiation for the model.
 "decay_steps": This parameter dictates how often the learning rate should be adjusted. In this case, the learning rate is updated every 1500 steps, allowing for adaptability as the model progresses through training.
 "decay_rate": The decay rate governs the rate at which the learning rate diminishes. With a decay rate of 0.9, the learning rate undergoes a gradual reduction, preventing drastic changes that may hinder convergence.

Custom Optimizer:

At the heart of model compilation is the choice of optimizer. An optimizer is responsible for adjusting the model's weights during training, steering the model towards optimal performance. In this implementation, a custom optimizer is employed, integrating a learning rate schedule for adaptive learning.
The Adam optimizer, known for its adaptive learning rate methodology, is wellsuited for a wide range of deep learning tasks. The ExponentialDecay schedule is then incorporated into a custom optimizer, specifically the Adam optimizer.



```
    decay_rate=0.9)

custom_optimizer = keras.optimizers.Adam(learning_rate=lr_schedule)
model.compile(optimizer=custom_optimizer
```

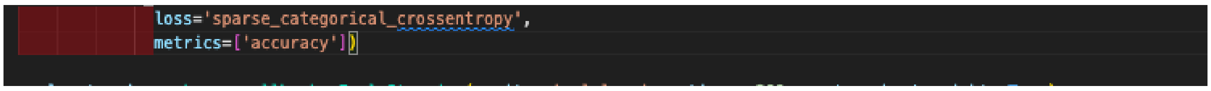
The Adam optimizer is renowned for its adaptive learning rate approach, making it wellsuited for a variety of deep learning tasks.
This learning rate schedule plays a pivotal role in enhancing the model's convergence by allowing for a dynamic adjustment of the learning rate based on the progression of training. It helps navigate the delicate balance between rapid initial learning and finetuning as the model approaches convergence.

Model Compilation:

The compilation of a deep learning model involves the configuration of essential components that define how the model should learn and optimize during training. Attention is paid to the compilation phase to ensure the model is equipped with the right tools for effective learning. With the components in place, the model is compiled using the “compile” method.
This step finalizes the model's configuration, making it ready for training. The optimizer, loss function, and metrics collectively define how the model will learn from the data and how its performance will be assessed. This meticulous compilation process ensures that the model is equipped with the necessary tools to navigate the complex landscape of the dataset and adapt its parameters effectively during training.

Loss Function:

The choice of a loss function is critical, especially in classification tasks. Here, the sparse categorical crossentropy is selected, aligning with the multiclass nature of the problem.

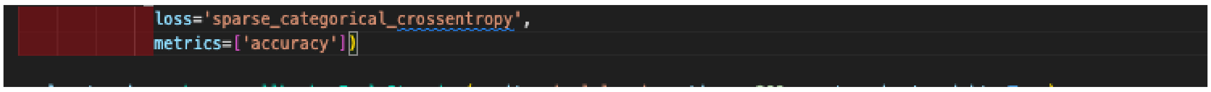


```
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Sparse categorical crossentropy is suitable when the target labels are integers (as opposed to onehot encoded vectors) and is commonly used for classification problems with multiple classes.

Metrics:

Metrics provide a quantitative measure of the model's performance during training and evaluation. In this project, accuracy is chosen as a metric, offering insight into the model's ability to correctly classify instances.

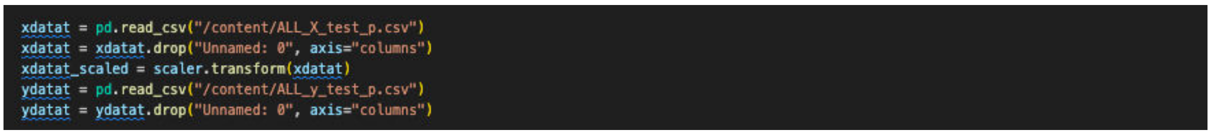


```
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Accuracy is a fundamental metric for classification tasks, indicating the proportion of correctly classified instances over the total number of instances.

Test Data Loading:

Before evaluating the model, it is essential to load the test data for a fair and unbiased assessment.

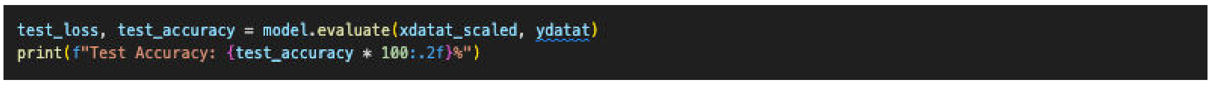


```
xdatat = pd.read_csv("/content/ALL_X_test_p.csv")
xdatat = xdatat.drop("Unnamed: 0", axis="columns")
xdatat_scaled = scaler.transform(xdatat)
ydatat = pd.read_csv("/content/ALL_y_test_p.csv")
ydatat = ydatat.drop("Unnamed: 0", axis="columns")
```

The test dataset, separate from the training data, serves as a critical benchmark for assessing how well the model generalizes to new, unseen data.

Model Evaluation:

The evaluation phase is a crucial step in understanding the performance of a trained deep learning model. In this project, the model is subjected to rigorous evaluation, aiming to gain insights into its accuracy and generalization capabilities.
The “evaluate” method is employed to quantify the model's performance on the test data.

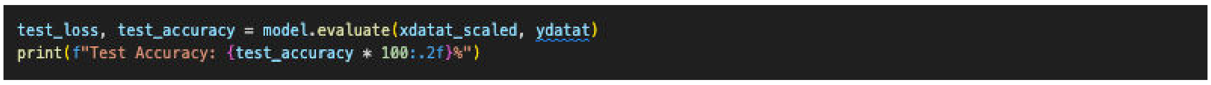


```
test_loss, test_accuracy = model.evaluate(xdatat_scaled, ydatat)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

“test_loss”: Represents the loss on the test set, indicating the difference between predicted and true labels. A lower loss value signifies better alignment between predictions and ground truth.
“test_accuracy”: Denotes the accuracy achieved on the test set. It represents the proportion of correctly classified instances over the total number of instances.

Performance Reporting:

The results of the evaluation are then reported for a comprehensive understanding of the model's effectiveness.

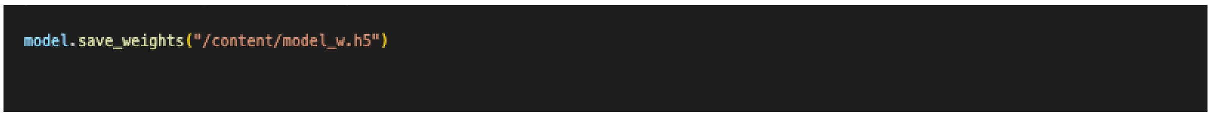


```
test_loss, test_accuracy = model.evaluate(xdatat_scaled, ydatat)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

These metrics provide a clear picture of how well the model generalizes to unseen data. A high accuracy and low loss indicate that the model has successfully learned meaningful patterns from the training data and can apply this knowledge to new instances.

Model Saving:

Finally, the model weights are saved for potential future use.
This ensures that the trained model, with its optimized parameters, can be employed without retraining, offering efficiency in deployment scenarios.
Saving the model after training is a crucial step to preserve its optimized parameters and architecture for future use or deployment. In this project, the model is saved in a format that retains its structure and weights.



```
model.save_weights("/content/model_w.h5")
```

Model Saving Process:

The saving process is initiated after the model has undergone training and evaluation.
The “save_weights” method is utilized to save the model's weights to a specified file.
“/content/model_w.h5”: This is the file path where the model weights are saved. The “.h5” extension is common for storing weights in the Hierarchical Data Format (HDF5), a file format widely used for storing large amounts of numerical data.

Importance of Model Saving:

1. Reproducibility: Saving the model allows for reproducibility. The saved weights encapsulate the knowledge acquired during training, enabling the exact state of the model to be reconstructed later.

2. Deployment: The saved model can be deployed in various environments without the need for retraining. This is especially crucial in production scenarios where efficiency and speed are paramount.

3. Transfer Learning: The saved model can serve as a starting point for transfer learning. It can be used as a foundation for training on a related task, leveraging the knowledge gained from the original training.

4. Sharing Models: The saved model can be easily shared with collaborators or across different projects, promoting collaboration and knowledge dissemination.

File Format and Compatibility:

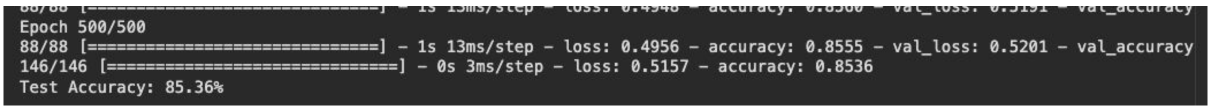
The choice of the HDF5 format ensures compatibility with a variety of tools and frameworks. It is a widely adopted format for storing and exchanging large amounts of numerical data, making it suitable for deep learning model weights.

Loading Saved Model:

To use the saved model in a different script or environment, the model can be reconstructed and the saved weights loaded.
This flexibility in loading saved models contributes to the seamless integration of deep learning models into diverse applications.

Test Accuracy:

I was able to generate a test accuracy of 85.36% using 500 epochs.



```
88/88 [==============================] - 1s 13ms/step - loss: 0.4948 - accuracy: 0.8500 - val_loss: 0.5191 - val_accuracy
Epoch 500/500
88/88 [==============================] - 1s 13ms/step - loss: 0.4956 - accuracy: 0.8555 - val_loss: 0.5201 - val_accuracy
146/146 [==============================] - 0s 3ms/step - loss: 0.5157 - accuracy: 0.8536
Test Accuracy: 85.36%
```

Conclusion:

In conclusion, this project demonstrates the potential of deep learning for classification tasks. The model, with its carefully crafted architecture and optimization strategies, achieved commendable accuracy on unseen data. The combination of TensorFlow and Keras, coupled with meticulous data handling and advanced techniques, has yielded a model ready for deployment and further exploration in future endeavors.

THE-END