Explanations for Fusion Model

Ansh Menghani

Function Explanations for code (code can be found at https://github.com/anshmenghani/Fusion):

Imports:

```
import os
os.environ["KERAS_BACKEND"] = "tensorflow" # Set the Keras backend environmental variable to Tensorflow
os.environ["TF_ENABLE_ONEDNN_OPTS"] = "0" # Turn off the Tesorflow OneDNN option

import numpy as np
import pandas as pd
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
import joblib
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Layer, Input, Embedding, Flatten, LayerNormalization, Discretization, Dense, GaussianDropout, concatenate, PReLU, Softmax, Cropping1D, Reshape
from tensorflow.keras import backend as K
from tensorflow.keras.saving import register_keras_serializable
from tensorflow.keras.models import Model
from tensorflow.keras.constraints import Constraint
from tensorflow.keras.losses import MeanSquaredLogarithmicError as MSLE, MeanAbsoluteError as MAE, CategoricalCrossentropy as CCE, Loss
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.activations import gelu
from tensorflow.keras.ops import log10, tanh
from tensorflow import TensorShape, constant, cast, clip_by_value, convert_to_tensor, Variable, float32
import tensorflow.keras.callbacks as callbacks
```

Fusion Code:

```
def data_prep(df, inputs, outputs, mod_attrs, func_attrs, funcs):
```

This function prepares the Gaia DR3 data to be used for training. More specifically, it splits the data into training (and validation) and testing data. It then normalizes the data using scikit-learn's RobustScaler, which uses the median and interquartile range to normalize the data. This ensures outliers in data do not skew the dataset too much. Next, the function applies any modifications/preprocessing to the data as necessary. Finally, it returns the finished training and testing data.

```
class UpdateHistory(callbacks.Callback):
```

This class defines the methods used to track validation losses across epochs so that the program can reward the model for decreases in validation loss during training.

```
class LambdaLayerClass(Layer):
```

This class defines the operations that are used to inform the model of the physical laws that are used to govern a stellar system so that it does not violate them when training and making predictions.

```
def lambda_init(in_layer, indices, no_right=False):
```

This function formats the lambda layers (responsible for informing the model of the physical formulas that govern stellar systems) to be compatible with the structure of the model defined in the `createSubModel`, `createModels`, and `fuseModels` functions.

```
class ValLossRewardConstraint(Constraint):
```

The model is rewarded every time its validation loss decreases. This constraint sets limits on how much the model can be rewarded based on how much the validation loss of the model changes between training steps. This prevents the model's loss from going out of control in the early stages of training where changes in validation loss are more random, as the model has not looked at enough data to see patterns in it. A side effect of this validation loss reward is that when the validation loss increases, the model penalizes itself by adding to its loss.

```
class LossRewardOptimizer(Layer):
```

This class defines the method used to train the variable that determines how much the model should be rewarded based on how much validation loss improved between two steps.

```
def RMSLE(y_true, y_pred):
```

This function defines the Square-Root Mean Squared Logarithmic Error loss function (among other loss functions used such as Mean Squared Logarithmic Error and Categorical Crossentropy, this one is not built in). It calculates the logarithmic difference between a predicted value and the "true" value, squares that, finds the mean of each one of those values for all testing data, and then takes the square root of the resultant value:

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(y_i + 1) - log(\widehat{y_i} + 1))^2}$$

```
class DLR(Loss):
```

This class contains the methods used to implement the validation loss reward by subtracting it from the model's loss.

```
def lambda_functors():
```

This function initializes all of the specific physical formulas that govern stellar systems to be added to the model by creating separate instances of the `LamdaLayerClass` class for each formula involved.

```
def createSubModel(shape=None, lambda_layer=None, lambda_inputs=None,
                   norm=True, bound=None, embed=False, embed_dim=None, output_actv=None, output_neurons=1):
```

This function creates submodels for each parameter the final model predicts. Each submodel is different as they have different tasks (some perform regression and others are classifiers), inputs, formulas associated with them, and output types.

```
def createModels():
```

This function calls the `createSubModel` function for each specific parameter the model predicts and returns these separate submodels.

```
def fuseModels(models, name):
```

This function combines (compiles) all of the submodels created from the `createModels` function into one model for training.

```
def Fuse():
```

This function is responsible for properly training the model using a multitude of methods (such as EarlyStopping, which is responsible for stopping training when validation loss is no longer improving and restoring the model's best weights) and many fine-tuned hyperparameters.

Model Processes Flowchart: