Table of Contents

[Neural Network Model(s) 1](#_Toc137643466)

[Improving NNs 1](#_Toc137643467)

[Loss Functions 2](#_Toc137643468)

[Q to ChatGPT Is there a loss function that does not penalised the outliers? but rather ignores them. 2](#_Toc137643469)

[Q to ChatGPT How about this loss function the race\_loss function ? What are your thoughts on this ? 3](#_Toc137643470)

[Model 4](#_Toc137643471)

[Q to CGPT, analyse my model code and explain. 4](#_Toc137643472)

[Q The predicted values can assume any value from 1 to 8 5](#_Toc137643473)

[Optimisers 6](#_Toc137643474)

[Q to CGPT: Is the adam optimiser best for such cases? 6](#_Toc137643475)

[Errors 7](#_Toc137643476)

[Q to CGPT : Nans in loss function, due to not normalising the variables. 7](#_Toc137643477)

[Architecture 8](#_Toc137643478)

[Q CGPT is the architecture good for a regression task? 8](#_Toc137643479)

[Validation 10](#_Toc137643480)

[Q to CGPT my validation loss is lower than training loss, what does that mean? 10](#_Toc137643481)

[Q CGPT … model history related ? 11](#_Toc137643482)

# Neural Network Model(s)

## Improving NNs

Improving a neural network model typically involves optimizing its architecture, training process, and input data. Here are some steps you can follow:

1. Collect more and diverse data: Having more and diverse data helps the model generalize better. Make sure your dataset includes a wide range of examples and represents the target problem well. Consider data augmentation techniques to increase dataset size and variability.
2. Split the data: Divide your data into training, validation, and test sets. This helps to evaluate the model's performance and prevent overfitting.
3. Preprocess the data: Clean and preprocess the data by handling missing values, normalizing features, and encoding categorical variables. This can improve the model's ability to learn from the data.
4. Choose an appropriate architecture: Experiment with different neural network architectures, including the number of layers, types of layers (e.g., convolutional, recurrent, or transformers), and the number of neurons in each layer.
5. Initialize weights properly: Initialize the model's weights using techniques like Xavier initialization or He initialization to avoid issues with vanishing or exploding gradients.
6. Optimize the learning rate: Choose an appropriate learning rate, which determines how much the model weights are updated during training. Adaptive learning rate methods like Adam, RMSprop, or AdaGrad can help.
7. Regularize the model: Apply regularization techniques like L1, L2, or dropout to prevent overfitting and improve generalization.
8. Use batch normalization: Incorporate batch normalization layers to stabilize and speed up training by normalizing the inputs across each mini-batch.
9. Use transfer learning: Leverage pre-trained models and fine-tune them on your specific task. This can save time and improve performance, especially when dealing with limited data.
10. Evaluate and optimize: Monitor the model's performance using appropriate metrics like accuracy, precision, recall, or F1 score. Use techniques like cross-validation, grid search, or random search to fine-tune hyperparameters.
11. Ensemble methods: Combine multiple models to improve overall performance. Techniques like bagging, boosting, or stacking can be employed.

Remember that iterating and experimenting is crucial when improving neural network models. Continuously monitor performance, identify areas of improvement, and apply new techniques to refine the model.

# Loss Functions

### Q to ChatGPT Is there a loss function that does not penalised the outliers? but rather ignores them.

Yes, there are loss functions that are more robust to outliers and less sensitive to them. One such loss function is the Huber loss, also known as the smooth mean absolute error. The Huber loss combines the advantages of the mean squared error (MSE) and mean absolute error (MAE) by introducing a threshold parameter (δ).

The Huber loss function is defined as follows:

L(y\_true, y\_pred) = 1/2 \* (y\_true - y\_pred)^2 if |y\_true - y\_pred| <= δ = δ \* |y\_true - y\_pred| - 1/2 \* δ^2 otherwise

For errors smaller than the threshold (δ), the Huber loss behaves like the mean squared error, whereas for larger errors, it behaves like the mean absolute error. This makes it less sensitive to outliers and can help prevent the model from overfitting to those outliers.

When using Huber loss in deep learning frameworks, the δ value can be adjusted to control the balance between sensitivity to small errors and robustness to outliers.

Another alternative is to use the quantile loss function, which is less sensitive to outliers, as it does not penalize all errors equally:

L(y\_true, y\_pred) = (q - 1) \* max(y\_true - y\_pred, 0) + q \* max(y\_pred - y\_true, 0)

Here, q is the quantile value, typically between 0 and 1. The quantile loss function essentially computes the weighted sum of the positive and negative errors. For instance, if q=0.5, the model focuses on minimizing the median error, which is less sensitive to outliers than the mean error. Adjusting the q value allows you to control the balance between underestimation and overestimation errors.

When choosing a loss function, consider the specific problem you are working on and whether robustness to outliers is a priority. Note that these loss functions may not completely ignore outliers, but they can help reduce their impact on the model.

### Q to ChatGPT How about this loss function the race\_loss function ? What are your thoughts on this ?

def race\_loss(y\_true, y\_pred):

    race\_loss = K.abs(y\_true - y\_pred)/( 1 + y\_true)

    return race\_loss

The custom loss function you provided, race\_loss, computes the absolute difference between the true and predicted values, then divides it by (1 + y\_true). This loss function has some interesting properties:

1. **Asymmetric weighting:** The loss function assigns different weights to errors based on the true value. When the true value is small, the loss function will penalize errors more heavily. When the true value is larger, the penalty for errors will be smaller. This asymmetric weighting may be suitable for some problems where errors for smaller true values are more important than those for larger true values.
2. **Non-negative output:** The loss function is always non-negative because both the numerator and denominator are non-negative. This is a desirable property for a loss function, as it ensures that the minimum loss is zero.
3. **Bounded gradient:** The gradient of this loss function is bounded, unlike the previous one you mentioned. This means that it is less likely to encounter issues like exploding gradients during training.
4. **Sensitivity to small true values:** The loss function can be sensitive to small true values, as the denominator will be small, leading to a larger loss. If your data has a significant number of small true values, you may want to apply some form of normalization or transformation to address this issue.

Overall, the race\_loss function seems to be more suitable for training neural networks than the previous one you mentioned. However, it's important to carefully consider the specific problem you are working on and evaluate whether the asymmetric weighting of errors is appropriate for your task. You may need to experiment with different loss functions and compare their performance to determine which one works best for your problem.

# Model

### Q to CGPT, analyse my model code and explain.

The model you're using is a deep feedforward neural network with 5 hidden layers, each containing 2048 units, and a single-unit output layer. You're using the custom 'race\_loss' function as the loss function and the Adam optimizer for training. The model also includes a custom callback to plot the learning curves during training.

Here are a few suggestions to improve the model and its training process:

1. **Data preprocessing:** Before feeding the data to the model, ensure that you have properly preprocessed it. This includes handling missing values, normalizing or scaling features, and encoding categorical variables if necessary.
2. **Regularization:** To prevent overfitting, consider adding regularization techniques such as L1, L2, or dropout. For example, you can add dropout layers or use the kernel\_regularizer parameter in the Dense layers to apply L1 or L2 regularization.
3. **Batch normalization:** Adding batch normalization layers between the Dense layers can help stabilize and speed up training.
4. **Learning rate scheduling:** You can use learning rate scheduling techniques to adjust the learning rate during training. This can help the optimizer converge faster and achieve better results.
5. **Early stopping:** To prevent overfitting and save time, you can use early stopping, which stops training when the validation loss stops improving for a specified number of epochs.
6. **Hyperparameter tuning:** Experiment with different hyperparameters, such as the number of layers, units per layer, learning rate, and optimizer. You can use techniques like grid search or random search to find the optimal hyperparameter values.
7. **Model architecture:** Consider trying different model architectures, such as convolutional neural networks (CNN) or recurrent neural networks (RNN) if your problem domain suggests that these may be beneficial (e.g., image or sequence data).
8. Monitor performance: Keep track of relevant metrics during training, such as loss, accuracy, precision, recall, or F1 score, depending on your problem.

Remember that iterating and experimenting with different configurations is key to improving your model's performance. Regularly evaluate the model and identify areas that need improvement.

Q The predicted values can assume any value from 1 to 8. I want to modify the race loss function to not worry about errors when the predicted values are close to 8 but instead focus on values when the predicted values are close to 1

You can modify the race loss function to focus more on errors when predicted values are close to 1 and less on errors when predicted values are close to 8. One way to achieve this is by introducing a custom weighting factor based on the predicted values. Here's an example:

import tensorflow.keras.backend as K

def custom\_race\_loss(y\_true, y\_pred):

weight\_factor = K.abs(1 - K.clip(y\_pred, 1, 8) / 8)

race\_loss = weight\_factor \* K.abs(y\_true - y\_pred) / (1 + y\_true)

return race\_loss

In this implementation, the weight\_factor is calculated based on the predicted values y\_pred. The factor ranges between 0 and 1, where it is close to 1 when the predicted value is close to 1 and close to 0 when the predicted value is close to 8.

This weighting factor is then multiplied by the original race\_loss formula, which scales the loss value based on the predicted value. As a result, errors for predicted values close to 1 will have a more significant impact on the overall loss, while errors for predicted values close to 8 will have a smaller impact.

Update the loss function in your model:

model\_hyperopt.compile(loss=custom\_race\_loss, optimizer=Adam(), metrics=[custom\_race\_loss])

Remember to monitor your model's performance during training and fine-tune the loss function or other hyperparameters as needed to achieve the desired results.

# Optimisers

Q to CGPT: Is the adam optimiser best for such cases? and also could you suggest some other optimisers that will work best in these cases

The choice of the optimizer depends on the specific problem and model architecture. Adam is a popular choice due to its adaptiveness to different problems and its ability to converge faster than some other optimizers. However, there are several other optimizers that you can experiment with to find the one that works best for your case.

1. Stochastic Gradient Descent (SGD): A simple but powerful optimizer that updates the model's parameters using a fixed learning rate. It can be slower to converge but is known for its ability to find good minima in some cases. You can also add momentum or Nesterov momentum to speed up convergence and improve stability.

from tensorflow.keras.optimizers import SGD optimizer = SGD(learning\_rate=0.01, momentum=0.9, nesterov=True)

1. RMSprop: An adaptive learning rate optimizer that adjusts the learning rate for each parameter individually. It can converge faster than SGD and is well-suited for problems with non-stationary objectives.

from tensorflow.keras.optimizers import RMSprop optimizer = RMSprop(learning\_rate=0.001, rho=0.9)

1. Adagrad: Another adaptive learning rate optimizer that scales the learning rate based on the history of gradients for each parameter. It can be well-suited for sparse data but may have issues with a diminishing learning rate in long training runs.

from tensorflow.keras.optimizers import Adagrad optimizer = Adagrad(learning\_rate=0.01)

1. Adadelta: An extension of Adagrad that tries to address the diminishing learning rate issue by accumulating a window of past gradient updates.

from tensorflow.keras.optimizers import Adadelta optimizer = Adadelta(learning\_rate=1.0, rho=0.95)

1. Adamax: A variation of Adam that uses the infinity norm (max) instead of the L2 norm for updating the learning rates. It can be more stable than Adam in some cases.

from tensorflow.keras.optimizers import Adamax optimizer = Adamax(learning\_rate=0.002, beta\_1=0.9, beta\_2=0.999)

1. Nadam: A combination of Adam and Nesterov momentum, which can provide faster convergence in some cases.

from tensorflow.keras.optimizers import Nadam optimizer = Nadam(learning\_rate=0.002, beta\_1=0.9, beta\_2=0.999)

It's essential to experiment with different optimizers and their hyperparameters to find the one that works best for your problem. You may also need to adjust other aspects of the model, such as the architecture, loss function, and regularization, to achieve the desired results.

# Errors

### Q to CGPT : Nans in loss function, due to not normalising the variables.

Yes, not normalizing the variables can potentially lead to **NaN** values in the loss function during training. Neural networks are sensitive to the scale of input features, and using unnormalized variables can cause numerical instability and difficulties in optimization.

Here's how not normalizing the variables can lead to **NaN** values in the loss function:

1. **Magnitude of Variables**: If the input variables have significantly different magnitudes, such as one variable ranging from 0 to 1 and another variable ranging from 100 to 1000, the neural network might have difficulties learning the appropriate weights and biases. This can result in large gradients during backpropagation, which can lead to numerical instability and **NaN** values in the loss function.
2. **Activation Functions**: Neural networks typically use activation functions like sigmoid, tanh, or ReLU. These functions have finite ranges, and if the input variables are not properly scaled, the activations can go beyond these ranges, causing numerical issues and **NaN** values.

To avoid these problems, it's generally recommended to normalize the input variables. Normalization ensures that the variables have similar scales and ranges, usually by scaling them to a range like 0 to 1 or using techniques such as z-score normalization.

By normalizing the variables, you can help stabilize the training process, improve convergence, and reduce the likelihood of encountering **NaN** values in the loss function.

# Architecture

## Q CGPT is the architecture good for a regression task?

Yes, the architecture you provided is suitable for regression tasks, like predicting a continuous variable. You have a deep neural network with multiple layers and the correct output (a single neuron for your single continuous output).

Few things to consider:

1. The last layer of your model does not use an activation function. That's generally a good practice for regression problems since you don't want to restrict the output to a specific range like [0,1] (which would be the case if you use a sigmoid or softmax activation function, for example).
2. You have several layers with a large number of neurons (1024, 1024, 512, 256, 128). This might be fine, but depending on the complexity of your task and the amount of data you have, it could lead to overfitting (the model learns the training data too well and performs poorly on unseen data). If you see overfitting during training (training loss keeps decreasing but validation loss starts to increase), you might want to consider simplifying your model (fewer layers or fewer neurons per layer).
3. The use of batch normalization can help stabilize the training process and speed up learning. However, note that it also adds some complexity to your model.
4. You might also want to consider adding some dropout layers for regularization. This can help to prevent overfitting.
5. Remember to compile your model with a suitable loss function for regression tasks. Mean Squared Error (MSE) or Mean Absolute Error (MAE) are commonly used for regression.

def create\_model():

model = Sequential([

Input(shape=(TRAIN\_FEATURES,)),

Dense(1024, activation='relu'),

BatchNormalization(),

Dropout(0.2), # adding dropout for regularization

Dense(1024, activation='relu'),

BatchNormalization(),

Dropout(0.2), # adding dropout for regularization

Dense(512, activation='relu'),

BatchNormalization(),

Dropout(0.2), # adding dropout for regularization

Dense(256, activation='relu'),

BatchNormalization(),

Dropout(0.2), # adding dropout for regularization

Dense(128, activation='relu'),

Dense(1)

])

# compiling with suitable loss for regression

model.compile(optimizer='adam', loss='mean\_squared\_error')

return model

The number **0.2** in the Dropout layer signifies the fraction of the input units to drop during training. It's a hyperparameter that you can tune.

To be more specific, Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. It's a regularization method that approximates training a large number of neural networks with different architectures in parallel.

If you set the dropout rate to **0.2**, that means approximately 20% of the input units to the Dropout function will be dropped out, or in other words, during each training phase, 20% of the neurons will not be updated. This helps to prevent overfitting.

You might need to experiment with this value depending on your dataset and problem at hand. Commonly used values are in the range of 0.2 to 0.5. Note that increasing the dropout rate might increase underfitting, while decreasing it might increase overfitting. As with many hyperparameters in machine learning, finding the right balance is part of the challenge.

# Validation

## Q to CGPT my validation loss is lower than training loss, what does that mean?

ChatGPT

Normally, we would expect the training loss to be lower than the validation loss because the model is trained on the training data and therefore should perform better on it. It has seen this data and adjusted its parameters to reduce the error on this data.

However, there can be scenarios where the validation loss is lower than the training loss, which can occur due to the following reasons:

1. **Regularization:** If you have applied certain types of regularization (like dropout) during training, these are typically turned off when calculating the loss on the validation set. Since regularization is a form of deliberate adding noise to prevent overfitting and makes the training harder, the loss without it during validation can be lower.
2. **Size of the datasets:** If your validation set is significantly smaller than your training set, it's possible that it may not be representative of the overall distribution of data, and the model might perform better on it just by chance.
3. **Early stopping:** In some cases, the validation loss might be lower than the training loss if you're using early stopping techniques, which stop training when the validation loss stops decreasing. In such a case, it's possible to stop training at a point where validation loss is less than the training loss.
4. **Batch Normalization:** Like dropout, batch normalization also behaves differently during training and inference. It uses mini-batch statistics during training, and population statistics during inference (which are estimates computed during training). This difference can also sometimes lead to lower validation loss.

It is important to check that your validation set is large and diverse enough to be representative of your data. If you observe consistently lower validation loss over multiple runs or with different initializations, it may be due to the reasons above.

As always, the most important metric is the performance of your model on a test set, a set of data that has not been used during the training or validation phases. This gives you the most unbiased estimate of your model's ability to generalize to new, unseen data.

## Q CGPT … model history related ?

When you save a model using Keras's **model.save()** or **model.save\_weights()** methods, the model's weights are saved, but the training history is not saved. So when you load the model from a file using **keras.models.load\_model()**, you get back the model with its weights, but without any of its training history.

The **history** object is created and returned when you call **model.fit()**. It contains the loss and any metrics you specified when compiling the model, recorded at each epoch of training. These values are in-memory and are not saved to disk when you save the model.

If you want to access the training history after loading a model from a file, you have a couple of options:

1. Save the **history.history** dictionary (or the whole **history** object) to a file after training, then load it back when you load the model.

Here is an example of how to save the **history** object using the **pickle** module:

pythonCopy code

import pickle # Save with open('history.pkl', 'wb') as f: pickle.dump(history.history, f) # Load with open('history.pkl', 'rb') as f: loaded\_history = pickle.load(f)

1. Re-train the model for at least one epoch on your data. This will generate a new **history** object with at least one entry per metric. This option may not be suitable if re-training the model is not desired or feasible.

Please note that option 1 is a common practice, where important training metrics are saved for later analysis while model development and testing.