Rules-of-thumb for building a Neural Network

In this article, we will get a starting point to build an initial Neural Network. We will learn the thumb-rules, e.g. the number of hidden layers, number of nodes, activation, etc., and see the implementations in TensorFlow 2.

Deep Learning provides a wide variety of models. With them, we can build extremely accurate predictive models. However, with the wide variety and a multitude of setting parameters it may be daunting to find a starting point.

In this article, we will find a starting point for building a Neural Network, more specifically a Multilayer Perceptron as an example but most of it is generally applicable.

The idea here is to go over the thumb-rules to build a first neural network model. Tune and optimize this first model if it performs reasonably (minimally acceptable accuracy). Otherwise, it is better to look in the problem, data, or a different approach.

In the following, we have,

- Rules of thumb for building a Neural Network, and
- their implementation codes for a Binary Classification in TensorFlow 2.

Neural Networks

Neural Network has advanced tremendously with CNNs, RNNs, etc. and several subtypes within each of them developed over time. With each development we have successfully improved our prediction capabilities.

But at the same time, we have successfully made it harder to find a starting point for model building.

Each model has its own sky-reaching claims. With so many billboards around, it is easy to get distracted.

In the following, we will sail through the distractions by using a few rules-of-thumb to build a first model.

Rules of Thumb

We have a variety of neural networks. Among them, a multilayer perceptron is the “hello world” of Deep Learning. And, therefore, a good place to start when you are learning about or developing a new model in Deep Learning.

Following are the thumb-rules for building an MLP. However, most of them are applicable on other Deep Learning models.

1. **Number of Layers**: Start with two hidden layers (this does not include the last layer).
2. **Number of nodes (size) of intermediate layers**: a number from the geometric progression of 2, e.g., 4, 8, 16, 32, … . The first layer should be around half of the number of input data features. The next layer size as half of the previous.
3. **Number of nodes (size) of output layer for Classification:** If binary classification then the size is one. For a multi-class classifier, the size is the number of classes.
4. **Size of output layer for regression**: If single response then the size one. For multi-response regression, the size is the number of responses.
5. **Activation for intermediate layers**: Use `relu` activation.
6. **Activation for output layer:** Use `sigmoid` for binary classification, `softmax` for multi-class classifier, and `linear` for regression. For Autoencoders, the last layer should be `linear` if the input data is continuous, otherwise, `sigmoid` or `softmax` for binary or multi-level categorical input.
7. **Dropout layers:** Add Dropout after every layer, except the Input layer (if defining the Input layer separately). Set **Dropout rate to 0.5**. Dropout rate > 0.5 is counter-productive. If you believe a rate of 0.5 is regularizing too many nodes, then increase the size of the layer instead of reducing the Dropout rate to less than 0.5. I prefer to not set any Dropout on the Input layer. But if you feel compelled to do that, set the Dropout rate < 0.2.
8. **Data preprocessing**: I am assuming your predictors *X* is numeric and you have already converted any categorical columns into one-hot-encoding. Before using the data for model training, perform data scaling. Use`MinMaxScaler` from `sklearn.preprocessing`. If this does not work well, do `StandardScaler` present in the same library. The scaling is needed for *y* in regression.
9. **Split data to train, valid, test**: Use `train_test_split` from `sklearn.model_selection`. See example below.
10. **Class weights:** If you have unbalanced data, then set class weights to balance the loss in your `model.fit` . For a binary classifier, the weights should be: {0: number of 1s / data size, 1: number of 0s / data size}. For extremely unbalanced data (rare events), class weight may not work. Be cautious adding it.
11. **Optimizer**: Use `adam` with its default learning rate.
12. **Loss in classification:** For binary classification use `binary_crossentropy`. For multiclass, use `categorical_crossentropy` if the labels are one-hot-encoded, otherwise use `sparse_categorical_crossentropy` if the labels are integers.
13. **Loss in regression**: Use `mse`.
14. **Metrics for Classification:** Use `accuracy` that shows the percent of correct classifications. For unbalanced data, also include `tf.keras.metrics.Recall()` and `tf.keras.metrics.FalsePositives()`.
15. **Metric for Regression**: Use `tf.keras.metrics.RootMeanSquaredError()`.
16. **Epochs**: Start with 20 to see if the model training shows decreasing loss and any improvement in accuracy. If there is no minimal success with 20 epochs, move on. If you get some minimal success, make epoch as 100.
17. **Batch size**: Choose the batch size from the geometric progression of 2. For unbalanced datasets have larger value, like 128, otherwise start with 16.

Few Extras for Advanced Practitioners

1. **Oscillating loss**: If you encounter oscillating loss upon training then there is a convergence issue. Try reducing the learning rate and/or change the batch size.
2. **Oversampling and undersampling**: If your data is unbalanced, use SMOTE from `imblearn.over_sampling` .
3. **Curve shifting:** If you have to do a shifted prediction, for example an early prediction, use curve shifting. An implementation `curve_shift` is shown below.
4. **Custom Metric**: An important metric for unbalanced binary classification is the False Positive Rate. You can build this and similarly other custom metrics as shown below in `class FalsePositiveRate()` implementation below.
5. **Selu activation**: `selu` activation has been deemed as better than all other existing activations. I have not observed that always, but if you want to use `selu` activation then use `kernel_initializer='lecun_normal'` and `AlphaDropout`. In `AlphaDropout` use the rate as 0.1, `AlphaDropout(0.1)` . Example implementation is shown

below.

Example Multilayer Perceptron (MLP) in TensorFlow 2

I have implemented the MLP Neural Network on the paper sheet break data set I have used in my previous articles (see Extreme Rare Event Classification using Autoencoders in Keras).

In this implementation, we will see examples for the elements we mentioned in the above rules-of-thumb.

The implementation is done in TensorFlow 2. I highly recommend migrating to TensorFlow 2, if not already. It has all the simplicity of Keras and significantly better computational efficiency.

Follow Step-by-Step Guide to Install Tensorflow 2 for installation.

In the following, I am not attempting to find the best model. The idea is to learn the implementations. No step is skipped to favor brevity. Instead, the steps are verbose to help the reader apply them directly.

Libraries

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as snsimport pandas as pd
import numpy as np
from pylab import rcParamsfrom collections import Counterimport tensorflow as tffrom
tensorflow.keras import optimizers
from tensorflow.keras.models import Model, load_model, Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, AlphaDropout
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoardfrom sklearn.preprocessing
import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc, roc_curve
from sklearn.metrics import precision_recall_fscore_support, f1_scorefrom numpy.random import
seed
seed(1)SEED = 123 #used to help randomly select the data points
DATA_SPLIT_PCT = 0.2rcParams['figure.figsize'] = 8, 6
LABELS = ["Normal","Break"]
```

To test whether you are with the correct TensorFlow version, run this,

```
tf.__version__
```

Reading and Preparing the Data

Download the data here.

```
'''
Download data here:
https://docs.google.com/forms/d/e/1FAIpQLSdyUk3lfDl7I5KYK_pw285LCApc-_RcoC0Tf9cnDnZ_TWzPAw/view
```

```
form
'''
df = pd.read_csv("data/processminer-rare-event-mts - data.csv")
df.head(n=5)  # visualize the data.
```

Convert categorical columns to one-hot-encoding

```
hotencoding1 = pd.get_dummies(df['x28'])  # Grade&Bwt
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['x61'])  # EventPress
hotencoding2 = hotencoding2.add_prefix('eventpress_')df=df.drop(['x28', 'x61'],
axis=1)df=pd.concat([df, hotencoding1, hotencoding2], axis=1)
```

Curve Shift

This is a timeseries data in which we have to predict the event (y = 1) ahead in time. In this data, consecutive rows are 2 minutes apart. We will shift the labels in column y by 2 rows to do a 4 minute ahead prediction.

```
sign = lambda x: (1, -1)[x < 0]def curve_shift(df, shift_by):
    '''
    This function will shift the binary labels in a dataframe.
    The curve shift will be with respect to the 1s.
    For example, if shift is -2, the following process
    will happen: if row n is labeled as 1, then
    - Make row (n+shift_by):(n+shift_by-1) = 1.
    - Remove row n.
    i.e. the labels will be shifted up to 2 rows up.
    Inputs:
    df       A pandas dataframe with a binary labeled column.
             This labeled column should be named as 'y'.
    shift_by An integer denoting the number of rows to shift.
    Output
    df       A dataframe with the binary labels shifted by shift.
    '''vector = df['y'].copy()
    for s in range(abs(shift_by)):
        tmp = vector.shift(sign(shift_by))
        tmp = tmp.fillna(0)
        vector += tmp
    labelcol = 'y'
    # Add vector to the df
    df.insert(loc=0, column=labelcol+'tmp', value=vector)
    # Remove the rows with labelcol == 1.
    df = df.drop(df[df[labelcol] == 1].index)
    # Drop labelcol and rename the tmp col as labelcol
    df = df.drop(labelcol, axis=1)
    df = df.rename(columns={labelcol+'tmp': labelcol})
    # Make the labelcol binary
    df.loc[df[labelcol] > 0, labelcol] = 1return df
```

Shift up by two rows,

```
df = curve_shift(df, shift_by = -2)
```

Remove the time column now. It won't be required from here.

```
df = df.drop(['time'], axis=1)
```

Divide the data into train, valid, and test

```
df_train, df_test = train_test_split(df, test_size=DATA_SPLIT_PCT, random_state=SEED)
df_train, df_valid = train_test_split(df_train, test_size=DATA_SPLIT_PCT, random_state=SEED)
```

And separate the X and y.

```
x_train = df_train.drop(['y'], axis=1)
y_train = df_train.y.valuesx_valid = df_valid.drop(['y'], axis=1)
y_valid = df_valid.y.valuesx_test = df_test.drop(['y'], axis=1)
y_test = df_test.y
```

Data Scaling

```
scaler = MinMaxScaler().fit(x_train)
# scaler = StandardScaler().fit(x_train)
x_train_scaled = scaler.transform(x_train)
x_valid_scaled = scaler.transform(x_valid)
x_test_scaled = scaler.transform(x_test)
```

MLP Models

Custom metric: FalsePositiveRate()

We will develop a `FalsePositiveRate()` metric that we will use in each model below.

```
class FalsePositiveRate(tf.keras.metrics.Metric):
    def __init__(self, name='false_positive_rate', **kwargs):
        super(FalsePositiveRate, self).__init__(name=name, **kwargs)
        self.negatives = self.add_weight(name='negatives', initializer='zeros')
        self.false_positives = self.add_weight(name='false_negatives', initializer='zeros')
    def update_state(self, y_true, y_pred, sample_weight=None):
        '''
        Arguments:
        y_true  The actual y. Passed by default to Metric classes.
        y_pred  The predicted y. Passed by default to Metric classes.
        '''
        # Compute the number of negatives.
        y_true = tf.cast(y_true, tf.bool)
        negatives = tf.reduce_sum(tf.cast(tf.equal(y_true, False), self.dtype))
        self.negatives.assign_add(negatives)
        # Compute the number of false positives.
        y_pred = tf.greater_equal(y_pred, 0.5)  # Using default threshold of 0.5 to call a
prediction as positive labeled.
        false_positive_values = tf.logical_and(tf.equal(y_true, False), tf.equal(y_pred, True))
        false_positive_values = tf.cast(false_positive_values, self.dtype)
        if sample_weight is not None:
            sample_weight = tf.cast(sample_weight, self.dtype)
            sample_weight = tf.broadcast_weights(sample_weight, values)
            values = tf.multiply(false_positive_values, sample_weight)
        false_positives = tf.reduce_sum(false_positive_values)
        self.false_positives.assign_add(false_positives)
    def result(self):
        return tf.divide(self.false_positives, self.negatives)
```

Custom performance plotting functions

We will write two plot function to visualize the progress in loss, and accuracy measures. We will use them for the models built below.

```
def plot_loss(model_history):
    train_loss=[value for key, value in model_history.items() if 'loss' in key.lower()][0]
    valid_loss=[value for key, value in model_history.items() if 'loss' in key.lower()][1]fig,
ax1 = plt.subplots()color = 'tab:blue'
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss', color=color)
    ax1.plot(train_loss, '--', color=color, label='Train Loss')
    ax1.plot(valid_loss, color=color, label='Valid Loss')
    ax1.tick_params(axis='y', labelcolor=color)
    plt.legend(loc='upper left')
    plt.title('Model Loss')plt.show()def plot_model_recall_fpr(model_history):
    train_recall=[value for key, value in model_history.items() if 'recall' in key.lower()][0]
    valid_recall=[value for key, value in model_history.items() if 'recall' in
key.lower()][1]train_fpr=[value for key, value in model_history.items() if
'false_positive_rate' in key.lower()][0]
    valid_fpr=[value for key, value in model_history.items() if 'false_positive_rate' in
key.lower()][1]fig, ax1 = plt.subplots()color = 'tab:red'
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Recall', color=color)
    ax1.set_ylim([-0.05,1.05])
    ax1.plot(train_recall, '--', color=color, label='Train Recall')
    ax1.plot(valid_recall, color=color, label='Valid Recall')
    ax1.tick_params(axis='y', labelcolor=color)
    plt.legend(loc='upper left')
    plt.title('Model Recall and FPR')ax2 = ax1.twinx()  # instantiate a second axes that shares
the same x-axiscolor = 'tab:blue'
    ax2.set_ylabel('False Positive Rate', color=color)  # we already handled the x-label with
ax1
    ax2.plot(train_fpr, '--', color=color, label='Train FPR')
    ax2.plot(valid_fpr, color=color, label='Valid FPR')
    ax2.tick_params(axis='y', labelcolor=color)
    ax2.set_ylim([-0.05,1.05])fig.tight_layout()  # otherwise the right y-label is slightly
clipped
    plt.legend(loc='upper right')
    plt.show()
```

Model 1. Baseline.

```
n_features = x_train_scaled.shape[1]
mlp = Sequential()
mlp.add(Input(shape=(n_features, )))
mlp.add(Dense(32, activation='relu'))
mlp.add(Dense(16, activation='relu'))
mlp.add(Dense(1, activation='sigmoid'))
mlp.summary()mlp.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.Recall(), FalsePositiveRate()]
           )history = mlp.fit(x=x_train_scaled,
                  y=y_train,
                  batch_size=128,
                  epochs=100,
                  validation_data=(x_valid_scaled, y_valid),
                  verbose=0).history
```

See the model fitting loss and accuracies (recall and FPR) progress.

```
plot_loss(history)
```

```
plot_model_recall_fpr(history)
```

Model 2. Class weights.

Define the class weights as mentioned in the rules of thumb.

```
class_weight = {0: sum(y_train == 1)/len(y_train), 1: sum(y_train == 0)/len(y_train)}
```

Now, we will train the model.

```
n_features = x_train_scaled.shape[1]mlp = Sequential()
mlp.add(Input(shape=(n_features, )))
mlp.add(Dense(32, activation='relu'))
mlp.add(Dense(16, activation='relu'))
mlp.add(Dense(1, activation='sigmoid'))mlp.summary()mlp.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.Recall(), FalsePositiveRate()]
           )history = mlp.fit(x=x_train_scaled,
                  y=y_train,
                  batch_size=128,
                  epochs=100,
                  validation_data=(x_valid_scaled, y_valid),
                  class_weight=class_weight,
                  verbose=0).historyplot_loss(history)
```

```
plot_model_recall_fpr(history)
```

Model 3. Dropout Regularization.

```
n_features = x_train_scaled.shape[1]mlp = Sequential()
mlp.add(Input(shape=(n_features, )))
mlp.add(Dense(32, activation='relu'))
mlp.add(Dropout(0.5))
mlp.add(Dense(16, activation='relu'))
mlp.add(Dropout(0.5))
mlp.add(Dense(1, activation='sigmoid'))mlp.summary()mlp.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.Recall(), FalsePositiveRate()]
           )history = mlp.fit(x=x_train_scaled,
                  y=y_train,
                  batch_size=128,
                  epochs=100,
                  validation_data=(x_valid_scaled, y_valid),
                  class_weight=class_weight,
                  verbose=0).historyplot_loss(history)
```

```
plot_model_recall_fpr(history)
```

Model 4. Oversampling-Undersampling

Using SMOTE resampler.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=212)
x_train_scaled_resampled, y_train_resampled = smote.fit_resample(x_train_scaled, y_train)
print('Resampled dataset shape %s' % Counter(y_train_resampled))
```

```
n_features = x_train_scaled.shape[1]mlp = Sequential()
mlp.add(Input(shape=(n_features, )))
mlp.add(Dense(32, activation='relu'))
mlp.add(Dropout(0.5))
mlp.add(Dense(16, activation='relu'))
mlp.add(Dropout(0.5))
mlp.add(Dense(1, activation='sigmoid'))mlp.summary()mlp.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.Recall(), FalsePositiveRate()]
           )history = mlp.fit(x=x_train_scaled_resampled,
                  y=y_train_resampled,
                  batch_size=128,
                  epochs=100,
                  validation_data=(x_valid, y_valid),
                  class_weight=class_weight,
                  verbose=0).historyplot_loss(history)
```

```
plot_model_recall_fpr(history)
```

Model 5. Selu activation.

We use the `selu` activation that got popular due its self-normalizing properties.

Note:

- We used a `kernel_initializer=’lecun_normal’` , and,
- Dropout as `AlphaDropout(0.1)` .

```
n_features = x_train_scaled.shape[1]mlp = Sequential()
mlp.add(Input(shape=(n_features, )))
mlp.add(Dense(32, kernel_initializer='lecun_normal', activation='selu'))
mlp.add(AlphaDropout(0.1))
mlp.add(Dense(16, kernel_initializer='lecun_normal', activation='selu'))
mlp.add(AlphaDropout(0.1))
mlp.add(Dense(1, activation='sigmoid'))mlp.summary()mlp.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy', tf.keras.metrics.Recall(), FalsePositiveRate()]
           )history = mlp.fit(x=x_train_scaled,
                  y=y_train,
                  batch_size=128,
                  epochs=100,
                  validation_data=(x_valid, y_valid),
                  class_weight=class_weight,
                  verbose=0).historyplot_loss(history)
```

```
plot_model_recall_fpr(history)
```

Conclusion

- With all the predictive modeling abilities that Deep Learning has offered, it can also be overwhelming to begin.
- The rules-of-thumb in this article provides a starting point to build an initial Neural Network.
- The model built from here should be further tuned to improve the performance.
- If the model performance built with these rules-of-thumb does not have some minimal performance. Tuning further may not bring much improvement. Try another approach.
- The article shows steps to implement the neural network in TensorFlow 2.
- If you do not have TensorFlow 2, it is recommend to migrate to it as it brings the ease of (keras) implementation and high performance. See instructions here, Step-by-Step Guide to Install Tensorflow 2.

Written by

Chitta Ranjan

Follow

Ph.D., Director of Science at ProcessMiner, Inc. https:%%//%%www.linkedin.com/in/chitta-ranjan-b0851911/

Follow