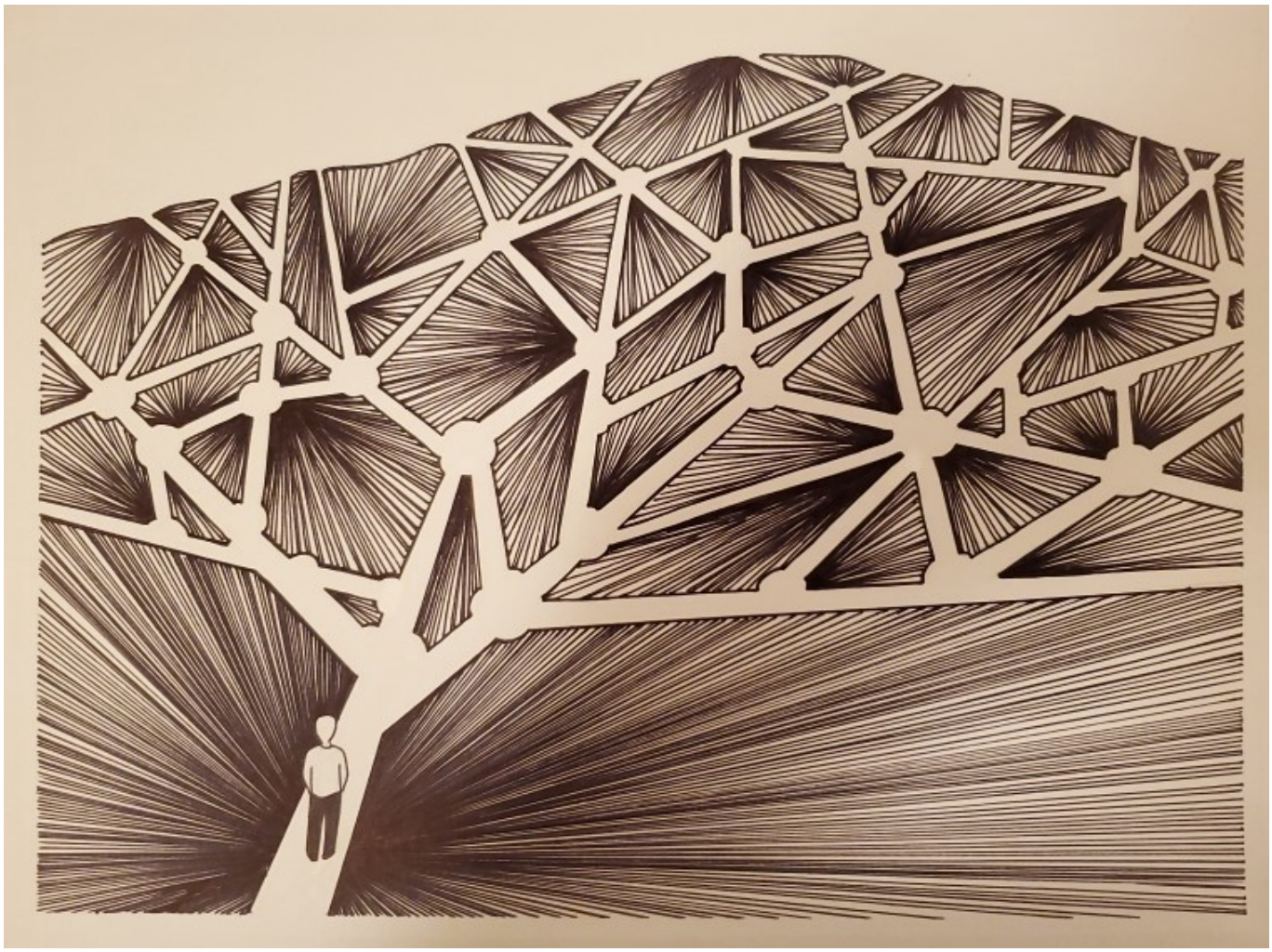

Data artist Wendy Angus, used with permission

Automated Feature Engineering Using Neural Networks

Michael Malin · Feb 26 · 12 min read ★

Few will deny that feature engineering is one of the most important steps for producing accurate models. Some people love it, but I am definitely not one of them. I find this step very tedious and it is my firm belief that anything tedious can be automated. While my solution doesn't entirely eliminate the need for manual work, it does greatly reduce it and produces better results. **It also produces a model that consistently beats gradient boosted methods on structured datasets.**

This article will answer the following questions:

- What exactly is being automated?
- How does it work?
- How do you build it?
- How does it compare to other models?

Feature Engineering

> "Doing this step correctly can involve doing weeks of EDA. Luckily, finding interactions is what neural networks excel at!"

When a model is fed a group of features, it has to learn which ones interact with each other and how they interact. With large datasets, there can be an endless number of combinations to test and these models tend to focus on interactions that provide quick success. With feature engineering, you can manually create or combine features to ensure that the model gives them proper focus.

- Data cleaning: Some people consider this feature engineering but it is really its own step. In short, you need to make sure the data is even useable before feature engineering is even possible. It involves fixing errors in the data, handling missing values, handling outliers, one-hot encoding, scaling features ,and countless other things. In my opinion, data cleaning is the only step worse than feature engineering so anyone who finds a way to automate this step will be my new hero.

- Mean encoding: This step involves transforming categorical features like zip code into information useable by the model. For example, you might create a column that shows the average sales revenue for a zip code. I already largely removed the need for this step in my previous article on feature embedding.

- Lag variables: It can often help to add a time series element to your data. By adding values from previous periods, a model can figure out how things trend over time (e.g. last month's sales, the month before that, etc). This process is not overly complicated and can be automated with simple loops.

- Interactions: This step involves combining features in different ways. For example, you might measure conversion for online ads by dividing ad-driven customer purchases by the total number of times an ad was clicked. But what if conversion varied greatly depending on the price of the product? Now you might create separate columns based on price thresholds. It can also by very difficult to spot and know how to handle 3rd (or higher) order interactions (e.g. ad conversion varying on price and category). **This is by far the most nuanced and time consuming step in feature engineering**. Doing this step correctly can involve doing weeks of EDA. Luckily, finding interactions is what neural networks excel at! The trick is ensuring that the model actually looks for them which will be the focus hereafter.

The Concept

consider certain combinations by engineering them. But what if we could force the neural network to consider them instead? What if we could ensure that the neural network engineers these features in a way that yields the best accuracy for the target output? The key is to train the model to focus on the features first.

Lets say we have features A, B, C and D with a target output Y. The first step to this problem is to create a model that **predicts each feature**. Why do we care about predicting features? Because we want the neural network to learn interactions specific to each feature.

Example feature network diagram by author

The interesting thing about this step is that we are not concerned about the model outputs. The last hidden layer (green nodes in diagram) that contain our new engineered features are what we will be extracting. These can feed into our final model (along with the original features) to predict our target output Y.

Example Full feature network diagram by author

The trick is making sure that the feature networks train ***with*** the final model rather than a separate process. Trickier still is training an embedding layer that feeds into each feature layer (see my article on why that's important). The good news for you: after months of struggling, I was able to develop a solution and **it exceeded my expectations**.

The Code

To demonstrate these methods, we will be trying to predict the probability that a person with COVID-19 will have a severe reaction. You can find the "Cleaned-Data.csv" dataset here: https://www.kaggle.com/iamhungundji/covid19-symptoms-checker?select=Cleaned-Data.csv

Lets pull in the data and create training, validation, and test datasets:

```
from sklearn.model_selection import train_test_split
from tensorflow import feature_column
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ModelCheckpoint
from sklearn.metrics import log_loss

X_train = pd.read_csv('covid_data.csv')
y_train = X_train.pop('Severity_Severe').to_frame()
X_train = X_train.iloc[:,:23]

X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train,test_size=0.2,random_state=42)

X_val, X_test, y_val, y_test = train_test_split(
    X_val, y_val,test_size=0.5,random_state=42)
```

Now, we will want to define which features we want to create feature models for. Since we do not have many features, we might as well use them all (except for *Country* which will be used for the embedding). When models contain hundreds of features, it is good practice to explicitly define only the top features like I do here:

```
model_cols = ['Fever','Tiredness','Dry-Cough',
              'Difficulty-in-Breathing',
              'Sore-Throat','None_Sympton',
              'Pains','Nasal-Congestion',
              'Runny-Nose','Diarrhea',
              'None_Experiencing','Age_0-9',
              'Age_10-19','Age_20-24','Age_25-59',
              'Age_60_','Gender_Female','Gender_Male',
              'Gender_Transgender','Contact_Dont-Know',
              'Contact_No','Contact_Yes']
```

Each of these features will be a different auxiliary output of our overall model along with the target feature we are trying to predict (*Severity_Severe*). As we create our TensorFlow datasets, we will also have to define these as output features as well. Note that we rename each of these features by adding '*_out*' to the end so that TensorFlow does not get confused by duplicate names. Notice that we also add an additional '*_aux_out*' column for our target output. This is so that we can train a separate feature model around the target feature that will also feed into the final model as well. This is a

```
Y_train_df = X_train[model_cols].copy()
Y_train_df.columns = Y_train_df.columns + "_out"
Y_train_df['Severity_Severe_out'] = y_train['Severity_Severe']
Y_train_df['Severity_Severe_aux_out'] = y_train['Severity_Severe']
trainset = tf.data.Dataset.from_tensor_slices((
    dict(X_train),dict(Y_train_df))).batch(256)

Y_val_df = X_val[model_cols].copy()
Y_val_df.columns = Y_val_df.columns + "_out"
Y_val_df['Severity_Severe_out'] = y_val['Severity_Severe']
Y_val_df['Severity_Severe_aux_out'] = y_val['Severity_Severe']
valset = tf.data.Dataset.from_tensor_slices((
    dict(X_val),dict(Y_val_df))).batch(256)

Y_test_df = X_test[model_cols].copy()
Y_test_df.columns = Y_test_df.columns + "_out"
Y_test_df['Severity_Severe_out'] = y_test['Severity_Severe']
Y_val_df['Severity_Severe_aux_out'] = y_val['Severity_Severe']
testset = tf.data.Dataset.from_tensor_slices((
    dict(X_test),dict(Y_test_df))).batch(256)
```

The first function we are going to create is *add_model*. We will feed this function our feature names, define the number and size of layers, signify if we want to use batch normalization, define the name of the model, and choose the output activation. The hidden_layers variable will have a separate list for each layer with the first number being the number of neurons and the second being the dropout rate. The output of this function will be the output layer and the final hidden layer (engineered features) which will feed on to the final model. This function allows for easy hyperparameter tuning when using tools like hyperopt.

```
def add_model(
    feature_outputs=None,hidden_layers=[[512,0],[64,0]],
    batch_norm=False,model_name=None,activation='sigmoid'):

    if batch_norm == True:
        layer = layers.BatchNormalization()(feature_outputs)
    else:
        layer = feature_outputs

    for i in range(len(hidden_layers)):
```

```
            if batch_norm == True:
                layer = layers.BatchNormalization()(layer)
            if hidden_layers[i][1] > 0:
                layer = layers.Dropout(hidden_layers[i][1])(layer)

        output_layer = layers.Dense(1, activation=activation,
                                    name=model_name+'_out')(layer)

        return last_layer, output_layer
```

This next function is for creating an embedding layer. This will be helpful because *Country* is a sparse categorical feature. This function will take a dictionary of the features we will be converting to an embedding along with a list of the unique possible values for that feature defined here:

```
emb_layers = {'Country':list(X_train['Country'].unique())}
```

We also feed in model inputs which will be defined later. For the dimensions parameter, I have chosen to follow the default rule-of-thumb of using the 4th root of the length of unique features.

```
def add_emb(emb_layers={},model_inputs={}):
    emb_inputs = {}
    emb_features = []

    for key,value in emb_layers.items():
        emb_inputs[key] = model_inputs[key]
        catg_col = feature_column
            .categorical_column_with_vocabulary_list(key, value)
        emb_col = feature_column.embedding_column(
            catg_col,dimension=int(len(value)**0.25))
        emb_features.append(emb_col)

    emb_layer = layers.DenseFeatures(emb_features)
    emb_outputs = emb_layer(emb_inputs)

    return emb_outputs
```

exclude the feature being predicted (data leakage), and the features used for the embeddings. You should also be careful to remove features that can directly be used to calculate the output feature. For example, a model will quickly discover that it can get 100% accuracy for a feature like *Gender_Female* simply by looking at the values of the other gender columns and ignoring all other features. This would not be a very helpful model! To fix this, we will exclude the other gender, age, and contact features from the corresponding feature models.

```
feature_layers = {col:[col,'Country'] for col in model_cols}

feature_layers['Gender_Female'] += ['Gender_Male',
                                    'Gender_Transgender']
feature_layers['Gender_Male'] += ['Gender_Female',
                                  'Gender_Transgender']
feature_layers['Gender_Transgender'] += ['Gender_Female',
                                         'Gender_Male']

feature_layers['Age_0-9'] += ['Age_10-19','Age_20-24',
                              'Age_25-59','Age_60_']
feature_layers['Age_10-19'] += ['Age_0-9','Age_20-24',
                                'Age_25-59','Age_60_']
feature_layers['Age_20-24'] += ['Age_0-9','Age_10-19',
                                'Age_25-59','Age_60_']
feature_layers['Age_25-59'] += ['Age_0-9','Age_10-19',
                                'Age_20-24','Age_60_']
feature_layers['Age_60_'] += ['Age_0-9','Age_10-19',
                              'Age_20-24','Age_25-59']

feature_layers['Contact_Dont-Know'] += ['Contact_No','Contact_Yes']
feature_layers['Contact_No'] += ['Contact_Dont-Know','Contact_Yes']
feature_layers['Contact_Yes'] += ['Contact_Dont-Know','Contact_No']
```

We area also going to want to add a *feature_layer* for our auxiliary skip connection model:

```
feature_layers['Severity_Severe_aux'] = ['Country']
```

the *hidden_layer* structure described at the *add_model* function and an indicator if batch normalization should be used.

First, the function will define the input features the way TensorFlow likes to read them. A major strength of using TensorFlow inputs is that we only need to define the features once and they can be reused over and over again in each of the feature models. Next, we will determine if any embedding columns were defined and create an embedding layer (optional). For each feature model, we will create the *DenseFeatures* input layer (excluding the features defined above) and create a separate model using the *add_model* function. Just before the return, we check to see if the loop is running on the skip connection model. If so, we append the input features so that the final model can train using the original features as well. Finally, this function will return a dictionary of the model inputs, a list of each feature model output layer, and a list of each of the final hidden layers (i.e. the new engineered features).

```
def feature_models(
    output_feature=None,all_features=[],feature_layers={},
    emb_layers={},hidden_layers=[],batch_norm=False):

    model_inputs = {}
    for feature in all_features:
        if feature in [k for k,v in emb_layers.items()]:
            model_inputs[feature] = tf.keras.Input(shape=(1,),
                                                   name=feature,
                                                   dtype='string')
        else:
            model_inputs[feature] = tf.keras.Input(shape=(1,),
                                                   name=feature)

    if len(emb_layers) > 0:
        emb_outputs = add_emb(emb_layers,model_inputs)

    output_layers = []
    eng_layers = []
    for key,value in feature_layers.items():
        feature_columns = [feature_column.numeric_column(f)
                           for f in all_features if f not in value]

        feature_layer = layers.DenseFeatures(feature_columns)
        feature_outputs = feature_layer({k:v for k,v in
                                         model_inputs.items()
                                         if k not in value})
```

```
            last_layer, output_layer = add_model(
                feature_outputs=feature_outputs,
                hidden_layers=hidden_layers,
                batch_norm=batch_norm,
                model_name=key)

            output_layers.append(output_layer)
            eng_layers.append(last_layer)

            if key == output_feature + '_aux':
                eng_layers.append(feature_outputs)

    return model_inputs, output_layers, eng_layers
```

Note that if an embedding layer is used, it will be concatenated with each of these models' inputs. That means these embeddings will not only train to maximize overall model accuracy, but will train on each of these feature models as well. This leads to very robust embeddings and is a significant upgrade to the process described in my previous article.

Before we move onto our final function, lets define each of the parameters we will feed in. Most of these have already been described above or are typical to all TensorFlow models. In case you are not familiar with the *patience* parameter, it is used to stop training the model when the validation accuracy hasn't improved within the designated number of epochs.

```
params = {'all_features': list(X_train.columns),
          'output_feature':y_train.columns[0],
          'emb_layers':emb_layers,
          'feature_layers':feature_layers,
          'hidden_layers':[[256,0],[128,0.1],[64,0.2]],
          'batch_norm': True,
          'learning_rate':0.001,
          'patience':3,
          'epochs':20
         }
```

layers/features and feed them into the final model. Finally, we build, compile, train, and test the model.

```
def final_model(params,test=True):

    print(params['batch_norm'],params['hidden_layers'])

    model_inputs, output_layers, eng_layers = feature_models(
        all_features=params['all_features'],
        feature_layers=params['feature_layers'],
        emb_layers=params['emb_layers'],
        hidden_layers=params['hidden_layers'],
        batch_norm=params['batch_norm'],
        output_feature=params['output_feature'])

    concat_layer = layers.concatenate(eng_layers)
    last_layer, output_layer = add_model(
        feature_outputs=concat_layer,
        hidden_layers=params['hidden_layers'],
        batch_norm=params['batch_norm'],
        model_name=params['output_feature'])

    output_layers.append(output_layer)

    model = tf.keras.Model(
        inputs=[model_inputs],
        outputs=output_layers)

    aux_loss_wgt = 0.5 / len(params['feature_layers'])
    loss_wgts = [aux_loss_wgt for i in
                 range(len(params['feature_layers']))
    loss_wgts.append(0.5)

    model.compile(loss='binary_crossentropy',
                  optimizer=tf.keras.optimizers.Adam(
                      lr=params["learning_rate"]),
                  loss_weights=loss_wgts,
                  metrics=['accuracy'])

    es = tf.keras.callbacks.EarlyStopping(
        monitor='val_loss',mode='min',verbose=1,
        patience=params['patience'],restore_best_weights=True)

    history = model.fit(
        trainset,validation_data=valset,
        epochs=params['epochs'], verbose=0, callbacks=[es])

    yhat = model.predict(testset)
```

```
    print('Binary Crossentropy:',loss)

    if test==True:
        sys.stdout.flush()
        return {'loss': loss, 'status': STATUS_OK}
    else:
        return history, model
```

Notice that one of the inputs to this function is called *test*. This input allows you to switch between using hyperopt to solve for the best parameters (*test=True*), or train and return your final model (*test=False*). You also might not be familiar with the *loss_weights* parameter when compiling the model. Because we have several auxiliary outputs, we need to tell TensorFlow how much weight to give each one in determining how to adjust the model to improve accuracy. I personally like to give 50% weight to the auxiliary predictions (total) and 50% the the target prediction. Some might find it strange to give any weight to the auxiliary predictions since they are discarded at the *loss* calculation step. The problem is, if we do not give them any weight, the model will mostly ignore them, preventing it from learning useful features.

Now we just need to run *final_model* using the parameters we defined above:

```
history, model = final_model(params,test=False)
```

Now that we have a trained model, we can optionally extract the new features to be used in other models using the keras *get_layer()* function. I'll save this step for a future article if this generates enough interest.

The Results

> conventional wisdom that gradient boosted models are superior for structured datasets."

As you can imagine, this is a computationally expensive model to train. The good news is that it typically will converge on a more accurate answer in far fewer trials than a typical MLP. When you include the time saved from not spending weeks tediously engineering features, its far faster. Also, prediction latency is small enough to make this a production model (as opposed to the typical Kaggle 50+ meta model). If you extract the features and retrain a neural network with them, then it gets even faster.

The question still remains, **is it accurate?** In every case that I have applied this model, it has been the most accurate. **It consistently beats XGBoost,** going against conventional wisdom that gradient boosted models are superior for structured datasets. Lets see how it did on this problem!

Methodology and Accuracy

I tested three different models:

- XGBoost
- A standard MLP with embedding
- The Auto-feature model trained above

For the auto-feature model, I ran 20 trials using hyperopt to experiment with different network sizes. For the two competing models, I ran 40 trials because of their faster training time. Here are the results:

Accuracy scores by model

better than marginal gains. When I work with massive datasets utilizing hundreds of features, its not uncommon for the auto-feature model to beat XGBoost by 5–10%.

This has become a go-to production model for me when top accuracy is very important. It has yielded much success for me in my career and now I hope it can do the same for you.

About me

I am a Data Science freelancer with over 10 years of experience. I am always looking to connect so please feel free to:

- Connect with me on LinkedIn
- Follow me on Twitter
- Visit my website

Please feel free to comment below if you have any questions

Data Science AI Machine Learning Python TensorFlow