All Articles

Hacker's Guide to Hyperparameter Tuning

TL;DR Learn how to search for good Hyperparameter values using Keras Tuner in your Keras and scikit-learn models

Hyperparameter tuning refers to the process of searching for the best subset of hyperparameter values in some predefined space. For us mere mortals, that means - should I use a learning rate of 0.001 or 0.0001?

In particular, tuning Deep Neural Networks is notoriously hard (that's what she said?). Choosing the number of layers, neurons, type of activation function(s), optimizer, and learning rate are just some of the options. Unfortunately, you don't really know which choices are the ones that matter, in advance.

On top of that, those models can be slow to train. Running many experiments in parallel might be a good option. Still, you need a lot of computational resources to do that on practical datasets.

Here are some of the ways that Hyperparameter tuning can help you:

- Better accuracy on the test set
- Reduced number of parameters
- Reduced number of layers
- Faster inference speed

None of these benefits are guaranteed, but in practice, some combination often is true.

Run the complete code in your browser

What is a Hyperparameter?

Hyperparameters are never learned, but set by you (or your algorithm) and govern the whole training process. You can think of Hyperparameters as configuration variables you set when running some software. Common examples of Hyperparameters are learning rate, optimizer type, activation function, dropout rate.

Adjusting/finding good values is really slow. You have to wait for the whole training process to complete, evaluate the results and adjust the value(s). Unfortunately, you might have to complete the whole search process when your data or model changes.

Don't be a hero! Use Hyperparameters from papers or other peers when your datasets and models are similar. At least, you can use those as a starting point.

When to do Hyperparameter Tuning?

Changing anything inside your model or data affects the results from previous Hyperparameter searches. So, you want to defer the search as much as possible.

Three things need to be in place, before starting the search:

- You have intimate knowledge of your data
- You have an end-to-end framework/skeleton for running experiments
- You have a systematic way to record and check the results of the searches (coming up next)

Hyperparameter tuning can give you another 5-15% accuracy on the test data. Well worth it, if you have the computational resources to find a good set of parameters.

Common strategies

There are two common ways to search for hyperparameters:

Improving one model

This option suggest that you use a single model and try to improve it over time (days, weeks or even months). Each time you try to fiddle with the parameters so you get an improvement on your validation set.

This option is used when your dataset is very large and you lack computational resources to use the next one. (Grad student optimization also falls within this category)

Training many models

You train many models in parallel using different settings for the hyperparameters. This option is computationally demanding and can make your code messy.

Luckily, we'll use the Keras Tuner to make the process more managable.

Finding Hyperparameters

We're searching for multiple parameters. It might sound tempting to try out every possible combination. Grid search is a good option for that.

However, you might not want to do that. Random search is a better alternative. It's just that Neural Networks seem much more sensitive to changes in one parameter than another.

Another approach is to use Bayesian Optimization. This method builds a function that estimates how good your model is going to be with a certain choice of hyperparameters.

Both approaches are implemented in Keras Tuner. How can we use them?

Remember to occasionaly re-evaluate your hyperparameters. Over time, you might've improved your algorithm, your dataset might have changed or the hardware/software has changed. Because of those changes the best settings for the hyperparameters can get stale and need to be re-evaluated.

Data

We'll use the Titanic survivor data from Kaggle:

The competition is simple: use machine learning to create a model that predicts which passengers survived the Titanic shipwreck.

Let's load and take a look at the training data:

```
!gdown --id 1uWHjZ3y9XZKpcJ4fkSwjQJ-VDbZS-7xi --output titanic.csv
```

```
df = pd.read_csv('titanic.csv')
```

Exploration

Let's take a quick look at the data and try to understand what it contains:

```
df.shape
```

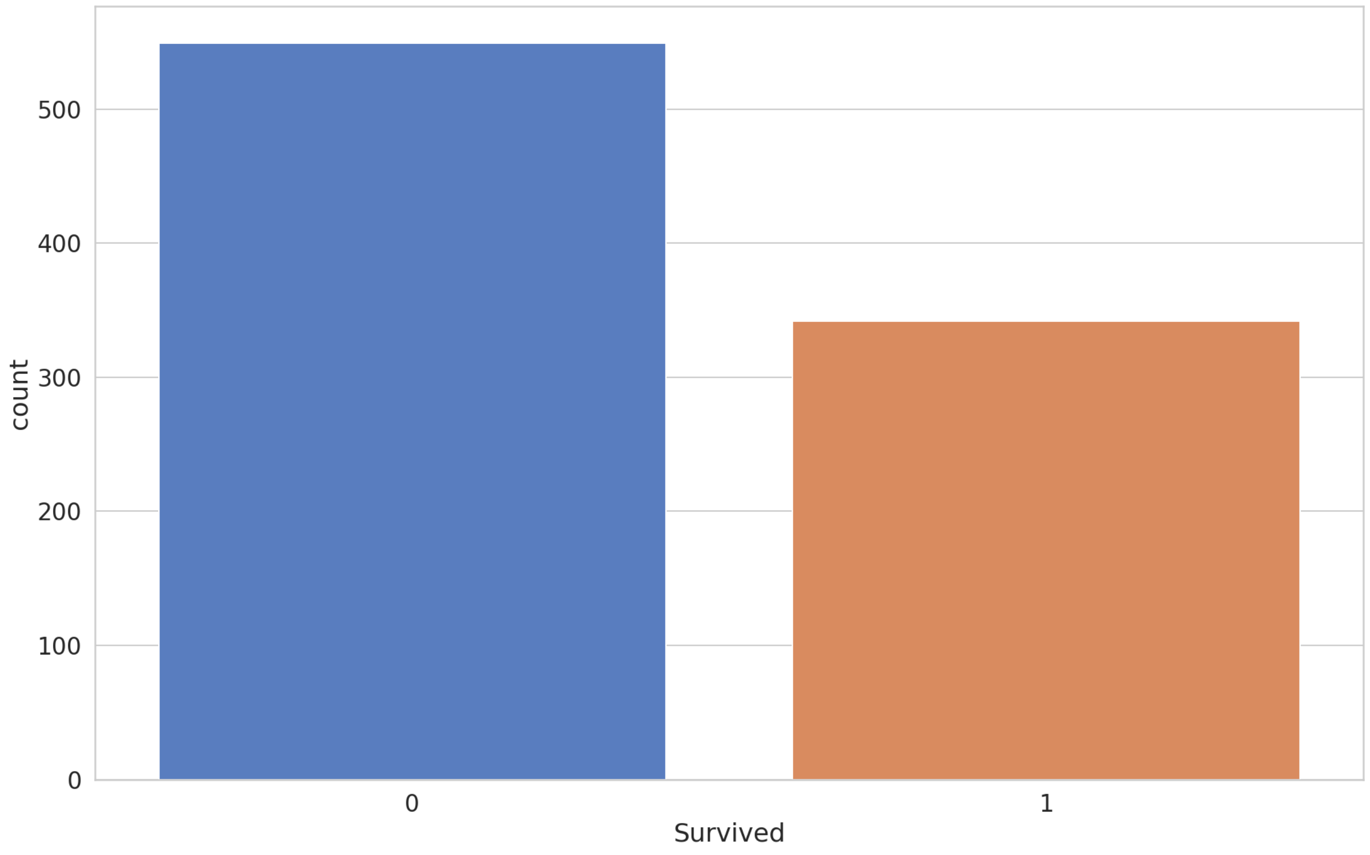
```
(891, 12)
```

We have 12 columns with 891 rows. Let's see what the columns are:

```
df.columns
```

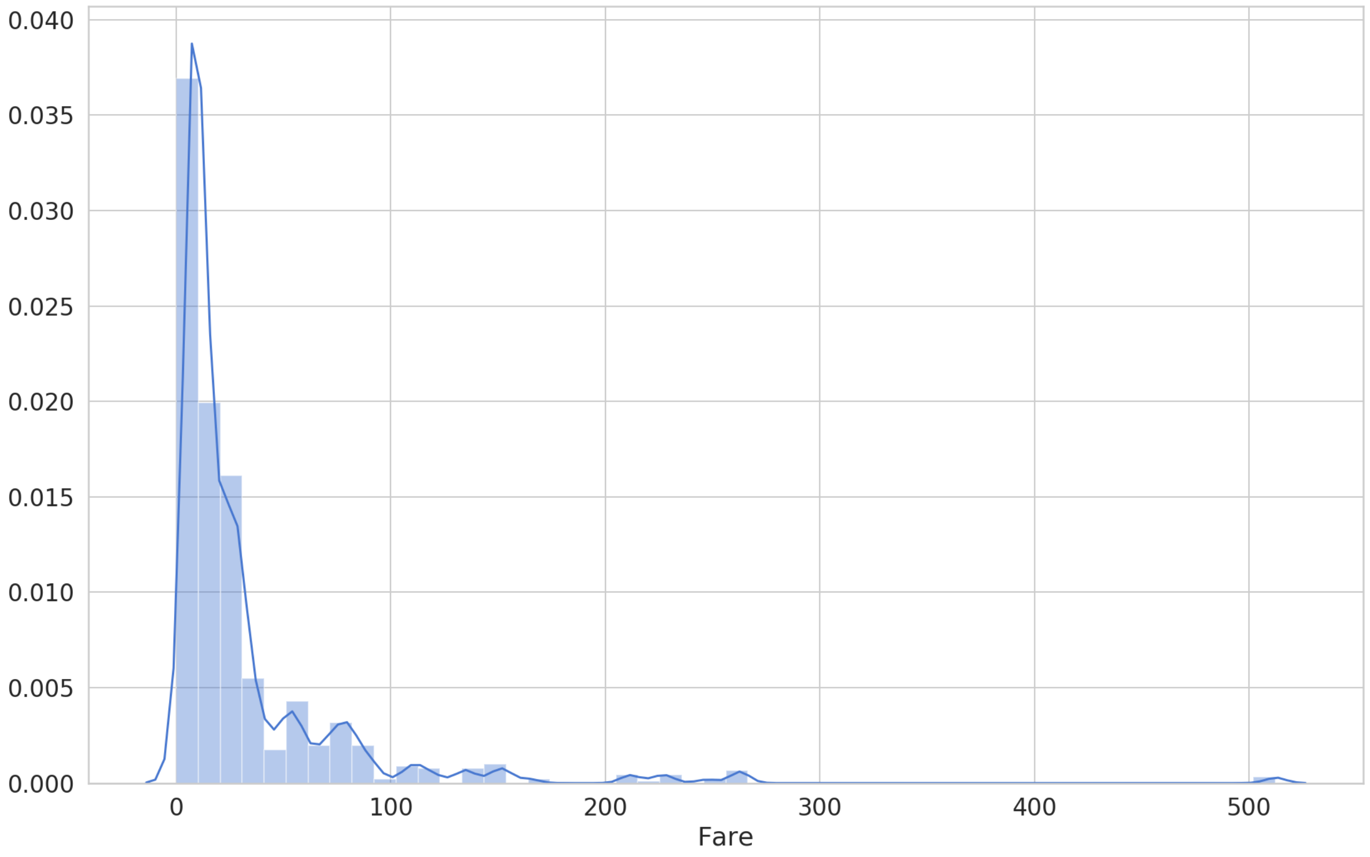
```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

All of our models are going to predict the value of the `Survived` column. Let's have a look its distribution:



While the classes are not well balanced, we'll use the dataset as-is. Read the Practical Guide to Handling Imbalanced Datasets to learn about some ways to solve this issue.

Another one that might interest you is the `Fare` (the price of the ticket):



About 80% of the tickets are priced below 30 USD. Do we have missing data?

Preprocessing

```
missing = df.isnull().sum()
missing[missing > 0].sort_values(ascending=False)
```

```
Cabin       687
Age         177
Embarked      2
```

Yes, we have a lot of cabin data missing. Luckily, we won't need that feature for our model. Let's drop it along with other columns:

```
df = df.drop(['Cabin', 'Name', 'Ticket', 'PassengerId'], axis=1)
```

We're left with 8 columns (including `Survived`). We still have to do something with the missing `Age` and `Embarked` columns. Let's handle those:

```
df['Age'] = df['Age'].fillna(df['Age'].mean())
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
```

The missing `Age` values are replaced with the mean value. Missing `Embarked` values are replaced with the most common one.

Now that our dataset has no missing values, we need preprocess the categorical features:

```
df = pd.get_dummies(df, columns=['Sex', 'Embarked', 'Pclass'])
```

We can start with building and optimizing our models. What do we need?

Keras Tuner

Keras Tuner is a new library (still in beta) that promises:

Hyperparameter tuning for humans

Sounds cool. Let's have a closer look.

There are two main requirements for searching Hyperparameters with Keras Tuner:

- Create a model building function that specifies possible Hyperparameter values
- Create and configure a Tuner to use for the search process

The version of Keras Tuner we're using in this writing is 7f6b00f45c6e0b0debaf183fa5f9dcef824fb02f. Yes, we're using the code from the `master` branch.

There are four different tuners available:

- RandomSearch
- Hyperband
- BayesianOptimization
- Sklearn

The scikit-learn Tuner is a bit special. It doesn't implement any algorithm for searching Hyperparameters. It rather relies on existing strategies to tune scikit-learn models.

How can we use Keras Tuner to find good parameters?

Random Search

Let's start with a complete example of how we can tune a model using Random Search:

```
def tune_optimizer_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(
      units=18,
      activation="relu",
      input_shape=[X_train.shape[1]]
    ))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    optimizer = hp.Choice('optimizer', ['adam', 'sgd', 'rmsprop'])

    model.compile(
        optimizer=optimizer,
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Everything here should look familiar except for the way we're choosing an Optimizer. We register a Hyperparameter with the name of `optimizer` and the available options. The next step is to create a Tuner:

```
MAX_TRIALS = 20
EXECUTIONS_PER_TRIAL = 5

tuner = RandomSearch(
    tune_optimizer_model,
    objective='val_accuracy',
    max_trials=MAX_TRIALS,
    executions_per_trial=EXECUTIONS_PER_TRIAL,
    directory='test_dir',
    project_name='tune_optimizer',
    seed=RANDOM_SEED
)
```

The Tuner needs a pointer to the model building function, what objective should optimize for (validation accuracy), and how many model configurations to test at most. The other config settings are rather self-explanatory.

We can get a summary of the different parameter values from our Tuner:

```
tuner.search_space_summary()
```

```
Search space summary
|-Default search space size: 1
optimizer (Choice)
|-default: adam
|-ordered: False
|-values: ['adam', 'sgd', 'rmsprop']
```

Finally, we can start the search:

```
TRAIN_EPOCHS = 20

tuner.search(x=X_train,
             y=y_train,
             epochs=TRAIN_EPOCHS,
             validation_data=(X_test, y_test))
```

The search process saves the trials for later analysis/reuse. Keras Tunes makes it easy to obtain previous results and load the best model found so far.

You can get a summary of the results:

```
tuner.results_summary()
```

```
Results summary
|-Results in test_dir/tune_optimizer
|-Showing 10 best trials
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.7519553303718
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.7430167198181
|-Objective: Objective(name='val_accuracy', direction='max') Score: 0.7273743152618
```

That's not helpful since we can't get the actual values of the Hyperparameters. Follow this issue for resolution of this.

Luckily, we can obtain the Hyperparameter values like so:

```
tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
```

```
{'optimizer': 'adam'}
```

Even better, we can get the best performing model:

```
best_model = tuner.get_best_models()[0]
```

Ok, choosing an Optimizer looks easy enough. What else can we tune?

Learning rate and Momentum

The following examples use the same RandomSearch settings. We'll change the model building function.

Two of the most important parameters for your Optimizer are the Learning rate and Momentum. Let's try to find good values for those:

```
def tune_rl_momentum_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(
      units=18,
      activation="relu",
      input_shape=[X_train.shape[1]]
    ))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    lr = hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])
    momentum = hp.Choice('momentum', [0.0, 0.2, 0.4, 0.6, 0.8, 0.9])

    model.compile(
        optimizer=keras.optimizers.SGD(lr, momentum=momentum),
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

The procedure is pretty identical to the one we've used before. Here are the results:

```
{'learning_rate': 0.01, 'momentum': 0.4}
```

Number of parameters

We can also try to find better value for the number of units in our hidden layer:

```
def tune_neurons_model(hp):
    model = keras.Sequential()
    model.add(keras.layers.Dense(units=hp.Int('units',
                                    min_value=8,
                                    max_value=128,
                                    step=16),
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

We're using a range of values for the number of parameters. The range is defined by a minimum, maximum and step value. The best number of units is:

```
{'units': 72}
```

Number of hidden layers

We can use Hyperparameter tuning for finding a better architecture for our model. Keras Tuner allows us to use regular Python for loops to do that:

```
def tune_layers_model(hp):
    model = keras.Sequential()

    model.add(keras.layers.Dense(units=128,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    for i in range(hp.Int('num_layers', 1, 6)):
      model.add(keras.layers.Dense(units=hp.Int('units_' + str(i),
                                            min_value=8,
                                            max_value=64,
                                            step=8),
                               activation='relu'))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Note that we still test a different number of units for each layer. There is a requirement that each Hyperparameter name should be unique. We get:

```
{'num_layers': 2,
 'units_0': 32,
 'units_1': 24,
 'units_2': 64,
 'units_3': 8,
 'units_4': 48,
 'units_5': 64}
```

Not that informative. Well, you can still get the best model and run with it.

Activation function

You can try out different activation functions like so:

```
def tune_act_model(hp):
    model = keras.Sequential()

    activation = hp.Choice('activation',
                            [
                              'softmax',
                              'softplus',
                              'softsign',
                              'relu',
                              'tanh',
                              'sigmoid',
                              'hard_sigmoid',
                              'linear'
                            ])

    model.add(keras.layers.Dense(units=32,
                                 activation=activation,
                                 input_shape=[X_train.shape[1]]))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Surprisingly we obtain the following result:

```
{'activation': 'linear'}
```

Dropout rate

Dropout is a frequently used Regularization technique. Let’s try different rates:

```
def tune_dropout_model(hp):
    model = keras.Sequential()

    drop_rate = hp.Choice('drop_rate',
                        [
                          0.0,
                          0.1,
                          0.2,
                          0.3,
                          0.4,
                          0.5,
                          0.6,
                          0.7,
                          0.8,
                          0.9
                        ])

    model.add(keras.layers.Dense(units=32,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))
    model.add(keras.layers.Dropout(rate=drop_rate))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Unsurprisingly, our model is relatively small and don’t benefit from regularization:

```
{'drop_rate': 0.0}
```

Complete example

We've dabbled with the Keras Tuner API for a bit. Let's have a look at a somewhat more realistic example:

```
def tune_nn_model(hp):
    model = keras.Sequential()

    model.add(keras.layers.Dense(units=128,
                                 activation="relu",
                                 input_shape=[X_train.shape[1]]))

    for i in range(hp.Int('num_layers', 1, 6)):
      units = hp.Int(
          'units_' + str(i),
          min_value=8,
          max_value=64,
          step=8
      )
      model.add(keras.layers.Dense(units=units, activation='relu'))
      drop_rate = hp.Choice('drop_rate_' + str(i),
                            [
                              0.0, 0.1, 0.2, 0.3, 0.4,
                              0.5, 0.6, 0.7, 0.8, 0.9
                            ])
      model.add(keras.layers.Dropout(rate=drop_rate))

    model.add(keras.layers.Dense(1, activation='sigmoid'))

    model.compile(
        optimizer="adam",
        loss = 'binary_crossentropy',
        metrics = ['accuracy'])
    return model
```

Yes, tuning parameters can complicate your code. One thing that might be helpful is to try and separate the possible Hyperparameter values from the code building code.

Bayesian Optimization

The Bayesian Tuner provides the same API as Random Search. In practice, this method should be as good (if not better) as the Grad student hyperparameter tuning method. Let’s have a look:

```
b_tuner = BayesianOptimization(
    tune_nn_model,
    objective='val_accuracy',
    max_trials=MAX_TRIALS,
    executions_per_trial=EXECUTIONS_PER_TRIAL,
    directory='test_dir',
    project_name='b_tune_nn',
    seed=RANDOM_SEED
)
```

This method might try out significantly fewer parameters than Random Search, but this is highly problem dependent. I would recommend using this Tuner for most practical problems.

scikit-learn model tuning

Despite its name, Keras Tuner allows you to tune scikit-learn models too! Let’s try it out on a RandomForestClassifier:

```
import kerastuner as kt
from sklearn import ensemble
from sklearn import metrics
from sklearn import datasets
from sklearn import model_selection

def build_tree_model(hp):
  return ensemble.RandomForestClassifier(
      n_estimators=hp.Int('n_estimators', 10, 80, step=5),
      max_depth=hp.Int('max_depth', 3, 10, step=1),
      max_features=hp.Choice('max_features', ['auto', 'sqrt', 'log2'])
  )
```

We’ll tune the number of trees in the forest (**n_estimators**), the maximum depth of the trees (**max_depth**), and the number of features to consider when choosing the best split (**max_features**).

The Tuner expects an optimization strategy (Oracle). We'll use Baysian Optimization:

```
sk_tuner = kt.tuners.Sklearn(
  oracle=kt.oracles.BayesianOptimization(
      objective=kt.Objective('score', 'max'),
      max_trials=MAX_TRIALS,
      seed=RANDOM_SEED
  ),
  hypermodel=build_tree_model,
  scoring=metrics.make_scorer(metrics.accuracy_score),
  cv=model_selection.StratifiedKFold(5),
  directory='test_dir',
  project_name='tune_rf'
)
```

The rest of the API is identical:

```
sk_tuner.search(X_train.values, y_train.values)
```

The best parameter values are:

```
sk_tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters.values
```

```
{'max_depth': 4, 'max_features': 'sqrt', 'n_estimators': 60}
```

Conclusion

There you have it. You now know how to search for good Hyperparameters for Keras and scikit-learn models.

Remember the three requirements that need to be in place before starting the search:

- You have intimate knowledge of your data
- You have an end-to-end framework/skeleton for running experiments
- You have a systematic way to record and check the results of the searches

Keras Tuner can help you with the last step.

Run the complete code in your browser

References

- Keras Tuner
- Random Search for Hyper-Parameter Optimization
- Bayesian optimization
- Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization
- Overview of hyperparameter tuning

Published 19 Oct 2019

Neural Networks Deep Learning TensorFlow Machine Learning Python

Show some love

Welcome! I am Venelin, and it`s a great pleasure to have you here!

Venelin Valkov on Twitter

What do you think?

2 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments **curiously** 1 **Login**

Recommend Tweet Share **Sort by Newest**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **CURIOUSILY**

Cryptocurrency price prediction using LSTMs | TensorFlow for Hackers (Part III)

2 comments • 9 months ago

Avatar **Diego** — Could you please provide an example of multivariate time series? Thanks !

What to do when data is missing? - Part II

3 comments • 3 years ago

Avatar **Henry Hargreaves** — you need to use l1_l2

Time Series Forecasting with LSTMs using Keras in Python

4 comments • 2 months ago

Avatar **gokul adethya** — Hey thanks for the article.I have been searching for RNN with tensorflow and finally found it.I came here after your

Predicting the next Fibonacci number with Linear Regression in …

1 comment • 9 months ago

Avatar **xh123** — Is it possible to make a program that as a input gets an index of fibonacci sequence and output is that number in sequence at

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy