# Lecture 7: Time Series Forecasting with Machine Learning Models

Isaiah Hull[1,2]

[1]BI Norwegian Business School
[2]CogniFrame

December 5, 2023

## Lecture 7: Overview

1. Time Series Forecasting with Machine Learning.
2. Applications.

# 1. Time Series

## Time Series

- Empirical economics focuses on causal inference and hypothesis testing.
- Machine learning emphasizes prediction.
- Overlap in objectives when forecasting in economics and finance.

## Time Series and Machine Learning

- Coulombe et al. (2019) discusses machine learning for time series econometrics.
- Potentially valuable tools: nonlinear models, regularization, cross validation, and alternative loss functions.
- Will focus on TensorFlow and deep learning models for time series.

**Sequential Models in Machine Learning**

- Specialized layers for neural networks to handle sequential data.
- Originally developed for natural language processing (NLP).
- Also applicable to time series contexts.
- Early example in Nakamura (2005) used neural networks for forecasting inflation.

**Dense Neural Networks**

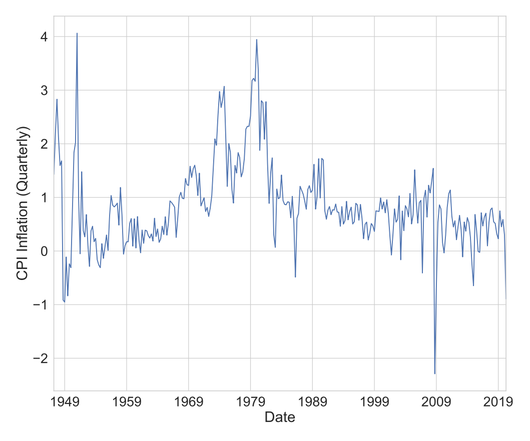
- Discussed earlier.
- Not previously adapted for sequential data.
- Forecasting exercise: Predicting quarterly inflation similar to Nakamura (2005).

## Forecasting Inflation

- Data on U.S. quarterly inflation from 1947:Q2 to 2020:Q2.
- Following Nakamura (2005), consider univariate models with lags of inflation.

# Time Series

**CPI Inflation: 1947:Q2 - 2020_Q2 (U.S. BLS)**




Source: Hull (2021).

## Pre-Processing Sequential Data

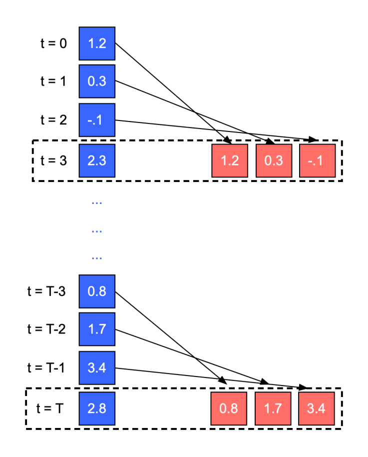
- Similar to text and image data, need to pre-process sequential data.
- Transform time series into fixed-length sequences.
- Decide on sequence length, e.g., number of lags for inputs.
- For a sequence length of three, use realizations in periods t, t-1, and t-3.

## Illustrating Pre-Processing

- Split single time series into overlapping sequences of consecutive observations.
- Use sequences to predict inflation for a single quarter ahead.

# Time Series

## Partition of Time Series




Source: Hull (2021).

## Sequence Generator for Inflation

```python
# Import packages.
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

# Load CSV data as pandas dataframe.
inflation = pd.read_csv('inflation.csv')

# Convert inflation column into numpy array.
inflation = np.array(inflation['Inflation'])

# Instantiate sequence generator.
generator = TimeseriesGenerator(inflation, inflation, length=4, batch_size=12)
```

## Model Training

- Using the generator object for batches of data.
- Use Keras Sequential() model.
- Sequential API for layer stacking.
- Define input, hidden, and output layers.
- Compile model with mean squared error and adam optimizer.
- Use fit_generator() for training.

## Train Neural Network with Sequences

```
# Define sequential model.
model = tf.keras.models.Sequential()

# Add input layer.
model.add(tf.keras.Input(shape=(4,)))

# Define dense layer.
model.add(tf.keras.layers.Dense(2, activation="relu"))
```

## Train Neural Network with Sequences

```
# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))

# Compile the model.
model.compile(loss="mse", optimizer="adam")

# Train the model.
model.fit_generator(generator, epochs=100)
```

## Train Neural Network with Sequences

*#Train for 25 steps*
Epoch 1/100
25/25 [==============================] - loss: 4.3247

Epoch 100/100
25/25 [==============================] - loss: 0.3816

**Model Performance**

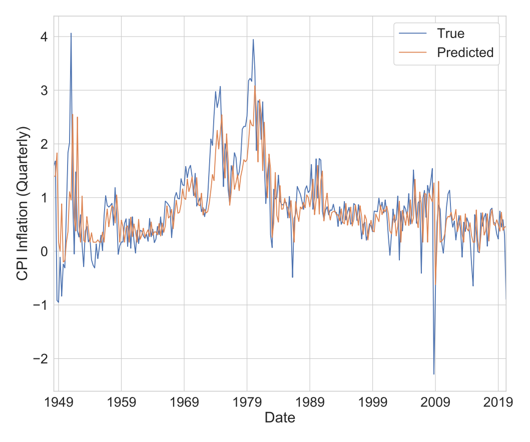
- Between epochs 1-100: Mean squared error drops from 4.32 to 0.38.
- No regularization or test sample split.
- Possible overfitting, but model has only 13 trainable parameters.

## Model Architecture Summary

```
print(model.summary())

_________________________________________________________________
Layer (type)          Output Shape          Param #
=================================================================
dense_1 (Dense)          (None, 2)            10
_________________________________________________________________
dense_1 (Dense)          (None, 1)             3
=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________
```

## True Values of Inflation vs. Model's Prediction

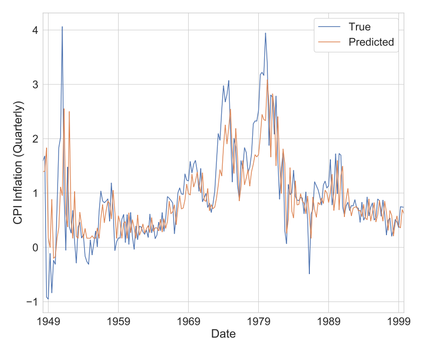



Source: Hull (2021).

## Model Evaluation

- Use model.predict_generator(generator) for predictions.
- Examine overfitting.
- Use pre-2000 values for training and post-2000 for testing.
- Construct a separate generator for predictions.

## Examining Potential Overfitting with Post-2000 Data




Source: Hull (2021).

**Recurrent Neural Networks (RNN)**

- RNN processes sequences using dense layers and recurrent layers.
- Inputs can be word vectors, musical notes, inflation measurements.
- Treatment follows Goodfellow et al. (2017).

## Recurrent Layer Structure

- Recurrent layer consists of cells.
- Each cell takes input value $x(t)$ and state $h(t-1)$.
- Produces an output value $o(t)$.

## Multiplication in RNN Cell

$$a(t) = b + Wh(t-1) + Ux(t)$$

- Take the state of the series, h(t-1).
- Multiply by weights, W.
- Take the input value, x(t), and multiply by a separate weights, U.
- Sum both terms together, along with a bias term, b.

## Activation Function in RNN Cell

$$h(t) = \mathsf{tanh}(a(t))$$

- Take output of multiplication step.
- Pass output to hyperbolic tangent activation function.
- Output is the updated state of the system, h(t).

## Output from RNN Cell

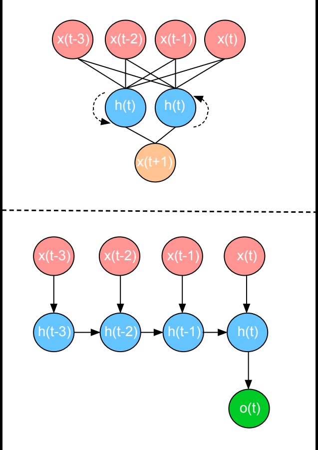
$$o(t) = c + Vh(t)$$

- Multiply updated state by a separate set of weights, V.
- Add bias term, c.

## Model Details

- In our example: inflation is the only feature.
- $x(t)$, $W$, $U$, and $V$ are scalars.
- Weights are shared across time periods.
- Only five parameters needed for one RNN cell layer.

## RNN (top) and unrolled RNN cell (bottom)




Source: Hull (2021).

**RNN Illustration**

- Pink nodes: input values (lags of inflation).
- Orange node: inflation for the next quarter (target).
- Blue nodes: individual RNN cells in an RNN layer.

**Unrolled RNN Cell**

- State combined with input to get next state.
- Last step gives output $o(t)$.
- Output is input to a final dense layer for next quarter's inflation prediction.

## RNN Characteristics

- Retains a state for sequential data.
- Updates state at each step.
- Reduces parameters through weight-sharing.
- No time-specific weights needed.
- Can handle sequences of varying length.

## Model Definition

- Define RNN model.
- Model similar in complexity to previous dense network.
- Uses ‘SimpleRNN‘ layer with 2 RNN cells.

## Sequence Generator for Inflation

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

# Load data.
inflation = pd.read_csv(data_path+'inflation.csv')

# Convert to numpy array.
inflation = np.array(inflation['Inflation'])
```

## Sequence Generator for Inflation

```
# Add dimension.
inflation = np.expand_dims(inflation, 1)

# Instantiate time series generator.
train_generator = TimeseriesGenerator(
inflation[:211], inflation[:211],
length = 4, batch_size = 12)
```

## Define RNN Model in Keras

```
# Define sequential model.
model = tf.keras.models.Sequential()

# Define recurrent layer.
model.add(tf.keras.layers.SimpleRNN(2, input_shape=(4, 1)))

# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))
```

## Define RNN Model in Keras

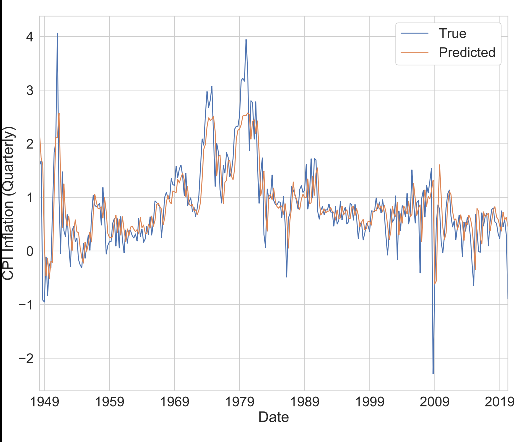
```
# Compile the model.
model.compile(loss="mse", optimizer="adam")

# Fit model to data using generator.
model.fit_generator(train_generator, epochs=100)
```

## Model Compilation and Training

- Mean squared error (MSE) is lower compared to dense network.
- Test sample performance post-2000 remains stable.

## Test Sample Performance Post-2000




Source: Hull (2021).

**Evaluate RNN Model in Keras**

```
Epoch 1/100
18/18 [==============================] - 1s 31ms/step - loss: 0.9206

Epoch 100/100
18/18 [==============================] - 0s 2ms/step - loss: 0.2594
```

## RNN vs. Dense Networks

- RNN has fewer parameters than dense networks.
- Single RNN cell layer requires 5 parameters.
- Examine architecture using ‘model.summary()‘.
- RNN model in this example: 11 parameters.
- Dense network used previously had more parameters.

## RNN Architecture in Keras

```
# Print model summary.
print(model.summary())
```

```
________________________________________________________________
Layer (type)                 Output Shape        Param #
================================================================
simple_rnn_1 (SimpleRNN)     (None, 2)           8
________________________________________________________________
dense_1 (Dense)              (None, 1)           3
================================================================
Total params: 11
Trainable params: 11
Non-trainable params: 0
________________________________________________________________
```

## RNN in Practice

- Challenges with basic RNN:
  - Vanishing gradient problem.
  - Original RNN struggles with long data sequences.
  - Doesn't account for distant temporal relationships.

## Long short-term memory (LSTM)

- RNNs suffer from vanishing gradient with long sequences.
- Solution: Use gated RNN cells.
- Two common gated RNN cells:
  - Long short-term memory (LSTM)
  - Gated recurrent units (GRUs)

**Long short-term memory (LSTM)**

- Introduced by Hochreiter and Schmidhuber (1997).
- Uses operators to limit flow of information in long sequences.

## Operations in LSTM

- Following Goodfellow et al. (2017) for LSTM operations.
- Three gates in LSTM:
  - Forget gate.
  - External input gate.
  - Output gate.
- Gates control flow of information in LSTM cell.

## Equations for LSTM Gates

- Forget gates:

$$f(t) = \sigma(b^f + W^f h(t-1) + U^f x(t))$$

- External input gates:

$$g(t) = \sigma(b^g + W^g h(t-1) + U^g x(t))$$

- Output gates:

$$q(t) = \sigma(b^q + W^q h(t-1) + U^q x(t))$$

## Internal States of LSTM

- Each gate has unique weights and biases.
- Gating procedure can be learned.
- Internal states are updated using definitions provided.
- Incorporates forget gate, external input gate, input sequence, and state.

## Equations for LSTM Gates

- Internal state:

$$s(t) = f^t s(t-1) + g(t)\sigma(b + Wh(t-1) + Ux(t))$$

- Hidden state:

$$h(t) = tanh(s(t))q(t)$$

## Use of Gates in LSTM

- Gates increase the number of model parameters.
- Significant improvement in handling long sequences.
- LSTM is typically the baseline ML model in time series analysis.

## LSTM Model Definition and Training

- Use tf.keras.layers.LSTM() instead of tf.keras.layers.SimpleRNN().
- Train for 100 epochs.
- Mean squared error is higher for LSTM than RNN after 100 epochs.
  - More weights in LSTM require more training epochs.
- Benefit from LSTM grows in sequence length.

## Train an LSTM Model in Keras

```
# Define sequential model.
model = tf.keras.models.Sequential()

# Define recurrent layer.
model.add(tf.keras.layers.LSTM(2, input_shape=(4, 1)))

# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))
```

## Train an LSTM Model in Keras

```
# Compile the model.
model.compile(loss="mse", optimizer="adam")

# Train the model.
model.fit_generator(train_generator, epochs=100)
```

## Train an LSTM Model in Keras

Epoch 1/100
18/18 [==============================] - 1s 62ms/step - loss: 3.1697

Epoch 100/100
18/18 [==============================] - 0s 3ms/step - loss: 0.5873

## LSTM Model's Architecture

- LSTM cells introduce additional operations:
  - Forget gate
  - External input gate
  - Output gate
- Each gate requires its own set of parameters.
- LSTM layer uses 32 parameters, which is 4 times as many as the RNN.

## Summarize LSTM Architecture in a Keras Model

```
# Print model architecture.
print(model.summary())
```

```
_________________________________________________________________
Layer (type)          Output Shape         Param #
=================================================================
lstm_1 (LSTM)           (None, 2)           32
_________________________________________________________________
dense_1 (Dense)         (None, 1)            3
=================================================================
Total params: 35
Trainable params: 35
Non-trainable params: 0
_________________________________________________________________
```

## Intermediate Hidden States

- By convention, LSTM only uses the final value of the hidden state.
- For instance:
  - Model uses $h(t)$.
  - Does not use $h(t-1), h(t-2), h(t-3)$.

## Use of Intermediate Hidden States

- Recent research shows benefits from using intermediate hidden states.
- Especially effective for modeling long-term dependencies in NLP.
- Typically done within attention models.

## Incorporating Hidden States in LSTM

- Modify LSTM cells to return hidden states.
- Set return_sequences to True.

## Code Implementation

```
# Modify LSTM to return hidden states
LSTM(return_sequences=True)
```

- After modification, we can review model's architecture using the summary() method.

## Code Implementation

```
# Define sequential model.
model = tf.keras.models.Sequential()

# Define recurrent layer to return hidden states.
model.add(tf.keras.layers.LSTM(2, return_sequences=True, input_shape=(4, 1)))

# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))

# Summarize model architecture.
model.summary()
```

## Code Implementation

```
Layer (type)          Output Shape          Param #
=================================================================
lstm_1 (LSTM)         (None, 4, 2)          32

dense_1 (Dense)       (None, 4, 1)          3
=================================================================
Total params: 35
Trainable params: 35
Non-trainable params: 0
```

## **Unusual Model Architecture**

- Model outputs a 4x1 vector, not a scalar prediction for each observation.
- This is due to the LSTM layer:
  - Outputs 4x1 vectors from its two LSTM cells.

## Making Use of LSTM Output

- Several methods to incorporate the LSTM output.
- Example: **Stacked LSTM** (Graves et al., 2013).
  - Pass full sequence hidden states to a second LSTM layer.
  - Adds depth to network for multiple levels of representation.

## Stacked LSTM Implementation

- Define a model with two LSTM layers.
- First LSTM layer:
  - Three LSTM cells.
  - Input shape of (4,1).
  - Set return_sequences to True.
  - Outputs 4x1 sequence of hidden states.

## Second LSTM Layer

- Accepts 3-tensor (4x1x3) from the first LSTM layer.
- Contains two cells.
- Only returns the final hidden states.
- Does not return intermediate state values.

## Define a Stacked LSTM Model

```python
# Define sequential model.
model = tf.keras.models.Sequential()

# Define recurrent layer to return hidden states.
model.add(tf.keras.layers.LSTM(3, return_sequences=True, input_shape=(4, 1)))

# Define second recurrent layer.
model.add(tf.keras.layers.LSTM(2))

# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))
```

## Summarize Stacked LSTM Architecture

```
# Summarize model architecture.
model.summary()
```

```
_________________________________________________________________
Layer (type)           Output Shape            Param #
=================================================================
lstm_1 (LSTM)          (None, 4, 3)            60
_________________________________________________________________
lstm_2 (LSTM)          (None, 2)               48
_________________________________________________________________
dense_1 (Dense)        (None, 1)               3
=================================================================
Total params: 111
Trainable params: 111
Non-trainable params: 0
_________________________________________________________________
```

**Focus and Context**

- Apply forecasting methods.
- Based examples on univariate inflation forecasting (Nakamura, 2005).
- All methods applicable to multivariate settings.

## Multivariate Forecasting

- Use both LSTM model and Gradient Boosted Trees.
- Forecast inflation monthly.
- Use five features instead of one.

## Data Preview and Model Implementation

- Prepare data and define model.
- Perform multivariate forecast using:
  - LSTM
  - Gradient boosted trees

## Features Details

- New features added:
  - Unemployment (measured in first differences).
  - Hours worked in manufacturing.
  - Hourly earnings in manufacturing.
  - Money supply measure (M1).
- Level variables transformed using percentage changes from previous period.

## Load and Preview Inflation Forecast Data

```
import pandas as pd

# Load data.
macroData = pd.read_csv(data_path+'macrodata.csv', index_col = 'Date')

# Preview data.
print(macroData.round(1).tail())
```

| Date | Inflation | Unemp | Hours | Earnings | M1 |
|---|---|---|---|---|---|
| 12/1/19 | -0.1 | 0.1 | 0.5 | 0.2 | 0.7 |
| 01/1/20 | 0.4 | 0.6 | -1.7 | -0.1 | 0.0 |
| 02/1/20 | 0.3 | -0.2 | 0.0 | 0.4 | 0.8 |
| 03/1/20 | -0.2 | 0.8 | -0.2 | 0.4 | 6.4 |
| 04/1/20 | -0.7 | 9.8 | -6.8 | 0.5 | 12.9 |

## LSTM Overview

- Recall data preparation for LSTM:
  - Instantiate a generator.
  - Convert target and features to np.array() objects.

## Data Generators

- Two data generators:
  - Training data
  - Test data

## Sequence Lengths

- Previous setup:
  - Quarterly data
  - 4-quarter sequence lengths
- Current setup:
  - Monthly data
  - 12-month sequence lengths

## Prepare Data for Use in LSTM Model

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

# Define target and features.
target = np.array(macroData['Inflation'])
features = np.array(macroData)

# Define train generator.
train_generator = TimeseriesGenerator(features[:393],
            target[:393], length = 12, batch_size = 6)
```

## Prepare Data for Use in LSTM Model

```
# Define test generator.
test_generator = TimeseriesGenerator(features[393:],
                 target[393:], length = 12, batch_size = 6)
```

## Generators Defined

- Train model.
- Model Specifications:
  - Two LSTM cells.
  - Input shape adjusted:
    - 12 elements in each sequence.
    - Five features.

## Training Outcome

- Concerns in macroeconomic forecasting:
  - Longer sequences and more variables.
  - Typically, concerns about number of parameters.

## Model Parameters

- Despite longer sequence length:
  - Doesn't increase the number of parameters.
- Model's total parameters:
  - Only 67 parameters.

## Load and Preview Inflation Forecast Data

```
# Define sequential model.
model = tf.keras.models.Sequential()

# Define LSTM model with two cells.
model.add(tf.keras.layers.LSTM(2, input_shape=(12, 5)))

# Define output layer.
model.add(tf.keras.layers.Dense(1, activation="linear"))

# Compile the model.
model.compile(loss="mse", optimizer="adam")
```

## Load and Preview Inflation Forecast Data

```
# Train the model.
model.fit_generator(train_generator, epochs=100)

Epoch 1/20
64/64 [==============================] - 2s 26ms/step - loss: 0.3065

Epoch 20/20
64/64 [==============================] - 0s 6ms/step - loss: 0.0663
```

## Use MSE to Evaluate Train and Test Sets

```
# Evaluate training set using MSE.
model.evaluate_generator(train_generator)

0.06527029448989197

# Evaluate test set using MSE.
model.evaluate_generator(test_generator)

0.15478561431742632
```

**Model Evaluation**

- Compare training sample results to test sample results.
- Training set performance generally better than test set.
- Not uncommon, but indicative of overfitting.

## Addressing Performance Gap

- If gap between training and test set performance is large:
  - Consider regularization.
  - Consider terminating training earlier.
  - Reduce number of epochs.

## Gradient Boosted Trees

- Compare GBT to deep learning.

## Data Preparation for Gradient Boosting

- Similar to LSTMs, data splitting into sequences is required.
- In TensorFlow, GBT trained using Estimator API.
- Must define feature columns for each of the five features.

## Data Generation Functions

- Define functions to generate data for both training and testing.
- Evaluate overfitting, analogous to the LSTM example.
- Sample split:
  - Train set: years before 2000.
  - Test set: years post-2000.

## Use MSE to Evaluate Train and Test Sets

```
# Define lagged inflation feature column.
inflation = tf.feature_column.numeric_column("inflation")

# Define unemployment feature column.
unemployment = tf.feature_column.numeric_column("unemployment")

# Define hours feature column.
hours = tf.feature_column.numeric_column("hours")
```

## Use MSE to Evaluate Train and Test Sets

```
# Define earnings feature column.
earnings = tf.feature_column.numeric_column("earnings")

# Define M1 feature column.
m1 = tf.feature_column.numeric_column("m1")

# Define feature list.
feature_list = [inflation, unemployment, hours, earnings, m1]
```

## Define the Data Generation Functions

```
# Define input function for training data.
def train_data():
    train = macroData.iloc[:392]
    features = {"inflation": train["Inflation"],
    "unemployment": train["Unemployment"],
    "hours": train["Hours"],
    "earnings": train["Earnings"],
    "m1": train["M1"]}
    labels = macroData["Inflation"].iloc[1:393]
    return features, labels
```

## Define the Data Generation Functions

```
# Define input function for test data.
def test_data():
    test = macroData.iloc[393:-1]
    features = {"inflation": test["Inflation"],
    "unemployment": test["Unemployment"],
    "hours": test["Hours"],
    "earnings": test["Earnings"],
    "m1": test["M1"]}
    labels = macroData["Inflation"].iloc[394:]
    return features, labels
```

## Train and Evaluate Model

```
# Instantiate boosted trees regressor.
model = tf.estimator.BoostedTreesRegressor(feature_columns =
    feature_list, n_batches_per_layer = 1)

# Train model.
model.train(train_data, steps=100)

# Evaluate train and test set.
train_eval = model.evaluate(train_data, steps = 1)
test_eval = model.evaluate(test_data, steps = 1)

# Print results.
print(pd.Series(train_eval))
print(pd.Series(test_eval))
```

## Train and Evaluate Model

```
average_loss         0.010534
label/mean           0.416240
loss                 0.010534
prediction/mean      0.416263
global_step        100.000000
dtype: float64

average_loss         0.145123
label/mean           0.172864
loss                 0.145123
prediction/mean      0.286285
global_step        100.000000
dtype: float64
```

**ML in Economics and Finance**

- ML focuses on prediction.
- Economics and finance often aim for causal inference and hypothesis testing.
- Overlap in fields: forecasting.

## Time Series Forecasting with ML

- Emphasis on deep learning models.
- Gradient boosted trees in TensorFlow.
- Early use of neural network in economics for forecasting: Nakamura (2005).
- Modern models: RNNs, LSTMs, stacked LSTMs.
- Applications extend to areas like NLP.

## Further Reading

- Macroeconomic time series forecasting: Cook and Hall (2017).
- Stock return & bond premium forecasting: Heaton et al. (2016), Messmer (2017), Rossi (2018), and Chen et al. (2019).
- High-dimensional time series regression and nowcasting: Babii et al. (2019, 2020).

# 2. Applications

# Applications

**Colab Tutorial**

- Time Series Prediction

# References I


Babii, A., E. Ghysels, and J. Striaukas (2019) "Inference for High-Dimensional Regressions with Heteroskedasticity and Auto-correlation," *arXiv preprint*.

——— (2020) "Machine Learning Time Series Regressions with an Application to Nowcasting," *arXiv preprint*.

Chen, L., M. Pelger, and J. Zhu (2019) "Deep Learning in Asset Pricing," *arXiv preprint*.

Cook, T.R. and A.S. Hall (2017) "Macroeconomic Indicator Forecasting with Deep Neural Networks," Research Working Paper 17-11, Federal Reserve Bank of Kansas City.

# References II


Coulombe, P.G., M. Leroux, D. Stevanovic, and S. Surprenant (2019) "How is Machine Learning Useful for Macroeconomic Forecasting?" working papers, CIRANO.

Goodfellow, I., Y. Bengio, and A. Courville (2017) *Deep Learning*, Cambridge, MA: MIT Press.

Graves, A., A.-r. Mohamed, and G. Hinton (2013) "Speech Recognition with Deep Recurrent Neural Networks," *arXiv preprint*.

Heaton, J.B., N.G. Polson, and J.H. Witte (2016) "Deep Learning for Finance: Deep Portfolios," *Applied Stochastic Models in Business and Industry*, 33 (1), 3–12.

Hochreiter, S. and J. Schmidhuber (1997) "Long Short-term Memory," *Neural Computation*, 9 (8), 1735–1780.

# References III


Hull, Isaiah (2021) *Machine Learning for Economics and Finance in TensorFlow 2*: Apress, 10.1007/978-1-4842-6373-0.

Messmer, M. (2017) "Deep Learning and the Cross-Section of Expected Returns," ssrn working paper.

Nakamura, Emi (2005) "Inflation Forecasting Using a Neural Network," *Economics Letters*, 85, 373–378.

Rossi, A.G. (2018) "Predicting Stock Market Returns with Machine Learning," working paper.