Lecture 8 - Student Version

Recurrent neural networks can handle time series data of different lengths. In this demo notebook we will first look deeper into Deep Knowledge Tracing, before showing examples of different types of neural network models for tracing and time series tasks. The learning objectives of this notebook are as follows:

1. Explore the differences between deep learning architectures for time-series data with LSTMs, GRUs and RNNs.

2. Implement hyperparameter tuning for a deep learning pipeline.

3. Contrast two behavioral time-series data settings: a model that makes a prediction at every time interval vs. a model that makes an overall prediction at the end of the time series.

If you are using EPFL's Noto, this notebook will need to use the `tensorflow` kernel for the dependencies to be installed appropriately. Change the kernel in the upper right corner of Noto. Select `tensorflow`.

```
# Load standard imports for the rest of the notebook.
import seaborn as sns
import pandas as pd
import numpy as np
import scipy as sc
import tensorflow as tf

# In this demo, we use a lot of SciKit-Learn functions, as imported
below.
from sklearn import feature_extraction, model_selection
from sklearn.metrics import mean_squared_error, roc_auc_score,
balanced_accuracy_score
from sklearn.model_selection import ParameterGrid, train_test_split
from sklearn.preprocessing import MinMaxScaler

DATA_DIR = "./../../data/"

# Setting this variable to true will train the DKT model fitting,
evaluation and
# hyperparameter tuning from scratch, which will take ~1 hour on
Colab.
train_from_scratch = False

def create_iterator(data):
    '''
    Create an iterator to split interactions in data into train and
test, with the same student not appearing in two diverse folds.
    :param data:        Dataframe with student's interactions.
```

```
    :return:            An iterator.
    '''
    # Both passing a matrix with the raw data or just an array of
indexes works
    X = np.arange(len(data.index))
    # Groups of interactions are identified by the user id (we do not
want the same user appearing in two folds)
    groups = data['user_id'].values
    return model_selection.GroupShuffleSplit(n_splits=1,
train_size=.8, test_size=0.2, random_state=0).split(X, groups=groups)
```

Deep Knowledge Tracing (DKT)

We begin by loading the data of the ASSISTments dataset (that we have explored in previous lectures).

The ASSISTments data sets are often used for benchmarking knowledge tracing models. We will play with a simplified data set that contains the following columns:

Name	Description
user_id	The ID of the student who is solving the problem.
order_id	The temporal ID (timestamp) associated with the student's answer to the problem.
skill_name	The name of the skill associated with the problem.
correct	The student's performance on the problem: 1 if the problem's answer is correct at the first attempt, 0 otherwise.

```
data = pd.read_csv(DATA_DIR + 'assistments.csv',
low_memory=False).dropna()
data.head()
```

```
   user_id  order_id       skill_name  correct
0    64525  33022537  Box and Whisker        1
1    64525  33022709  Box and Whisker        1
2    70363  35450204  Box and Whisker        0
3    70363  35450295  Box and Whisker        1
4    70363  35450311  Box and Whisker        0
```

Next, we print the number of students and skills in the dataset.

```
print("Number of unique students in the dataset:",
len(set(data['user_id'])))
print("Number of unique skills in the dataset:",
len(set(data['skill_name'])))
```

```
Number of unique students in the dataset: 4151
Number of unique skills in the dataset: 110
```

Data Preparation

Since the data needs to be fed into the model in batches, we need to specify in advance how many elements per batch the DKT model will receive. DKT also requires that all sequences need to be of the same length in order to be used as model input.

Given that students have different number of opportunities across skills, we need to define a scheme such that the sequences will be the same length. We choose to pad our values to the maximum sequence length and determine a masking value (for the model to ignore) for those entries that are introduced as a padding into the student's sequences.

```
def prepare_seq(df):
    '''
    Extract user_id sequence in preparation for DKT. The output of
this function
    feeds into the prepare_data() function.
    '''
    # Enumerate skill id as a categorical variable
    # (i.e. [32, 12, 32, 45] -> [0, 1, 0, 2])
    df['skill'], skill_codes = pd.factorize(df['skill_name'],
sort=True)

    # Cross skill id with answer to form a synthetic feature
    df['skill_with_answer'] = df['skill'] * 2 + df['correct']

    # Convert to a sequence per user_id and shift features 1 timestep
    seq = df.groupby('user_id').apply(lambda r:
(r['skill_with_answer'].values[:-1], r['skill'].values[1:],
r['correct'].values[1:],))

    # Get max skill depth and max feature depth
    skill_depth = df['skill'].max()
    features_depth = df['skill_with_answer'].max() + 1

    return seq, features_depth, skill_depth

def prepare_data(seq, params, features_depth, skill_depth):
    '''
    Manipulate the data sequences into the right format for DKT with
padding by batch
    and encoding categorical features.
    '''

    # Get Tensorflow Dataset
    dataset = tf.data.Dataset.from_generator(generator=lambda: seq,
output_types=(tf.int32, tf.int32, tf.float32))

    # Encode categorical features and merge skills with labels to
compute target loss
    dataset = dataset.map(
```

```
        lambda feat, skill, label: (
            tf.one_hot(feat, depth=features_depth),
            tf.concat(values=[tf.one_hot(skill, depth=skill_depth),
tf.expand_dims(label, -1)], axis=-1)
        )
    )

    # Pad sequences to the appropriate length per batch
    dataset = dataset.padded_batch(
        batch_size=params['batch_size'],
        padding_values=(params['mask_value'], params['mask_value']),
        padded_shapes=([None, None], [None, None]),
        drop_remainder=True
    )

    return dataset.repeat(), len(seq)
```

Your Turn (In-Class Discussion)

What do these hyperparameters mean?

```
# Specify the model hyperparameters. Full descriptions included in the
demo notebook!
params = {}

params['batch_size'] = 32
params['mask_value'] = -1.0
params['verbose'] = 1
params['best_model_weights'] = 'weights/bestmodel'
params['optimizer'] = 'adam'
params['recurrent_units'] = 16
params['epochs'] = 20
params['dropout_rate'] = 0.1
```

We then split the data into a train, a validation and a test set.

```
# Obtain indexes for training and test sets
train_index, test_index = next(create_iterator(data))

# Split the data into training and test
X_train, X_test = data.iloc[train_index], data.iloc[test_index]

# Obtain indexes for training and validation sets
train_val_index, val_index = next(create_iterator(X_train))

# Split the training data into training and validation
X_train_val, X_val = X_train.iloc[train_val_index],
X_train.iloc[val_index]

# Build TensorFlow sequence datasets for training, validation, and
test data
```

```
seq, features_depth, skill_depth = prepare_seq(data)
seq_train = seq[X_train_val.user_id.unique()]
seq_val = seq[X_val.user_id.unique()]
seq_test = seq[X_test.user_id.unique()]

# Prepare the training, validation, and test data in the DKT input format
tf_train, length = prepare_data(seq_train, params, features_depth, skill_depth)
tf_val, val_length  = prepare_data(seq_val, params, features_depth, skill_depth)
tf_test, test_length = prepare_data(seq_test, params, features_depth, skill_depth)

# Calculate the length of each of the train-test-val sets and store as parameters
params['train_size'] = int(length // params['batch_size'])
params['val_size'] = int(val_length // params['batch_size'])
params['test_size'] = int(test_length // params['batch_size'])
```

```
2023-06-29 18:59:51.688341: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory
2023-06-29 18:59:51.688412: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2023-06-29 18:59:51.688457: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (noto.epfl.ch): /proc/driver/nvidia/version does not exist
2023-06-29 18:59:51.689086: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Model Creation

First, we train DKT using an LSTM architecture and default parameter settings. We use a validation set to monitor prediction accuracy of the model and store the model with the best weights.

Considering that we padded the sequences such that all have the same length, we need to remove predictions for the time steps that are based on padded data. To this end, we implement a function called get_target.

```
def get_target(y_true, y_pred, mask_value=params['mask_value']):
    '''
```

```
    Adjust y_true and y_pred to ignore predictions made using padded
values.
    '''
    # Get skills and labels from y_true
    mask = 1. - tf.cast(tf.equal(y_true, mask_value), y_true.dtype)
    y_true = y_true * mask

    skills, y_true = tf.split(y_true, num_or_size_splits=[-1, 1],
axis=-1)

    # Get predictions for each skill
    y_pred = tf.reduce_sum(y_pred * skills, axis=-1, keepdims=True)

    return y_true, y_pred
```

While training and evaluating the model, we will monitor the following performance metrics. Please, note that we need to process our targets before using the default TensorFlow metric functions.

```
class AUC(tf.keras.metrics.AUC):
    # Our custom AUC calls our get_target function first to remove
predictions on padded values,
    # then computes a standard AUC metric.
    def __init__(self):
        # We use a super constructor here just to make our metric name
pretty!
        super(AUC, self).__init__(name='auc')

    def update_state(self, y_true, y_pred, sample_weight=None):
        true, pred = get_target(y_true, y_pred)
        super(AUC, self).update_state(y_true=true, y_pred=pred,
sample_weight=sample_weight)

class RMSE(tf.keras.metrics.RootMeanSquaredError):
    # Our custom RMSE calls our get_target function first to remove
predictions on padded values,
    # then computes a standard RMSE metric.
    def update_state(self, y_true, y_pred, sample_weight=None):
        true, pred = get_target(y_true, y_pred)
        super(RMSE, self).update_state(y_true=true, y_pred=pred,
sample_weight=sample_weight)

def CustomBinaryCrossEntropy(y_true, y_pred):
    # Our custom binary cross entropy loss calls our get_target
function first
    # to remove predictions on padded values, then computes standard
binary cross-entropy.
    y_true, y_pred = get_target(y_true, y_pred)
    return tf.keras.losses.binary_crossentropy(y_true, y_pred)
```

We define an LSTM and a GRU model.

```
def create_model_lstm(nb_features, nb_skills, params):

    # Create an LSTM model architecture
    inputs = tf.keras.Input(shape=(None, nb_features), name='inputs')

    # We use a masking layer here to ignore our masked padding values
    x = tf.keras.layers.Masking(mask_value=params['mask_value'])
(inputs)

    # This LSTM layer is the crux of the model; we use our parameters
to specify
    # what this layer should look like (# of recurrent_units, fraction
of dropout).
    x = tf.keras.layers.LSTM(params['recurrent_units'],
return_sequences=True, dropout=params['dropout_rate'])(x)

    # We use a dense layer with the sigmoid function activation to map
our predictions
    # between 0 and 1.
    dense = tf.keras.layers.Dense(nb_skills, activation='sigmoid')

    # The TimeDistributed layer takes the dense layer predictions and
applies the sigmoid
    # activation function to all time steps.
    outputs = tf.keras.layers.TimeDistributed(dense, name='outputs')
(x)
    model = tf.keras.models.Model(inputs=inputs, outputs=outputs,
name='DKT')

    # Compile the model with our loss functions, optimizer, and
metrics.
    model.compile(loss=CustomBinaryCrossEntropy,
                  optimizer=params['optimizer'],
                  metrics=[AUC(), RMSE()])

    return model

# Create our DKT model using an LSTM
dkt_lstm = create_model_lstm(features_depth, skill_depth, params)

def create_model_gru(nb_features, nb_skills, params):

    # Create a GRU model architecture
    inputs = tf.keras.Input(shape=(None, nb_features), name='inputs')

    # We use a masking layer here to ignore our masked padding values
    x = tf.keras.layers.Masking(mask_value=params['mask_value'])
(inputs)
```

```
    # This GRU layer is the crux of the model; we use our parameters 
to specify
    # what this layer should look like (# of recurrent_units, fraction 
of dropout).
    x = tf.keras.layers.GRU(params['recurrent_units'], 
return_sequences=True, dropout=params['dropout_rate'])(x)

    # We use a dense layer with the sigmoid function activation to map 
our predictions
    # between 0 and 1.
    dense = tf.keras.layers.Dense(nb_skills, activation='sigmoid')

    # The TimeDistributed layer takes the dense layer predictions and 
applies the sigmoid
    # activation function to all time steps.
    outputs = tf.keras.layers.TimeDistributed(dense, name='outputs')
(x)
    model = tf.keras.models.Model(inputs=inputs, outputs=outputs, 
name='DKT')

    # Compile the model with our loss functions, optimizer, and 
metrics.
    model.compile(loss=CustomBinaryCrossEntropy,
                  optimizer=params['optimizer'],
                  metrics=[AUC(), RMSE()])

    return model

# Create our DKT model using a GRU
dkt_gru = create_model_gru(features_depth, skill_depth, params)
```

Model Fitting and Evaluation

Next we train the models and then evaluate them on the test data.

```
# This cell takes 8 minutes to run. On default, we will not run the 
training experiments below.
# However, if you would like to run it from scratch, you can modify 
train_from_scratch=True
# at the beginning of the notebook.

if train_from_scratch:
  # This line tells our training procedure to only save the best 
version of the model at any given time.
  ckp_callback = 
tf.keras.callbacks.ModelCheckpoint(params['best_model_weights'], 

save_best_only=True, save_weights_only=True)
```

```
  # Let's fit our LSTM model on the training data. This cell takes 8
minutes to run.
  history = dkt_lstm.fit(tf_train, epochs=params['epochs'],
steps_per_epoch=params['train_size']-1,
                          validation_data=tf_val,
validation_steps=params['val_size'],
                          callbacks=[ckp_callback],
verbose=params['verbose'])

if train_from_scratch:
  # We load the LSTM model with the best performance, and evaluate it
on the test set.
  dkt_lstm.load_weights(params['best_model_weights'])
  dkt_lstm.evaluate(tf_test, steps=params['test_size'],
verbose=params['verbose'], return_dict=True)

if train_from_scratch:
  # This line tells our training procedure to only save the best
version of the model at any given time.
  ckp_callback =
tf.keras.callbacks.ModelCheckpoint(params['best_model_weights'],

save_best_only=True, save_weights_only=True)

  # Let's fit our GRU model on the training data. This cell takes 8
minutes to run.
  history = dkt_gru.fit(tf_train, epochs=params['epochs'],
steps_per_epoch=params['train_size']-1,
                          validation_data=tf_val,
validation_steps=params['val_size'],
                          callbacks=[ckp_callback],
verbose=params['verbose'])

if train_from_scratch:
  # We load the GRU model with the best performance, and evaluate it
on the test set.
  dkt_gru.load_weights(params['best_model_weights'])
  dkt_gru.evaluate(tf_test, steps=params['test_size'],
verbose=params['verbose'], return_dict=True)
```

Hyperparameter Tuning

As we have seen, we need to specify a lot of hyperparameters. In a next step, we perform a small grid search for the number of recurrent units in the LSTM: {8, 16, 32, 64}.

```
# Modify the dictionary of parameters so that each parameter maps to a
list of possibilities.
# In this case, we're only searching over the recurrent_units and
leaving the rest of the
# parameters fixed to their default values.
params_space = {param: [value] for param, value in params.items()}
```

```
params_space['recurrent_units'] = [8, 16, 32, 64]
params_grid = ParameterGrid(params_space)

# For each combination of candidate parameters, fit a model on the
training set
# and evaluate it on the validation set (as we've seen in Lecture 5).

# NOTE: This cell will take 40 minutes to run from scratch.
if train_from_scratch:
  results = {}

  # For each parameter setting in the grid search of parameters
  for params_i in params_grid:

      # Create a LSTM model with the specific parameter setting
params_i
      dkt_lstm = create_model_lstm(features_depth, skill_depth,
params_i)

      save_model_name = params_i['best_model_weights'] +
str(params_i['recurrent_units'])

      # Save the best version of the model through the training epochs
      ckp_callback =
tf.keras.callbacks.ModelCheckpoint(save_model_name,

save_best_only=True, save_weights_only=True)

      # Fit the model on the training data with the appropriate
parameters
      dkt_lstm.fit(tf_train,
                  epochs=params_i['epochs'],
                  steps_per_epoch=params_i['train_size']-1,
                  validation_data=tf_val,
                  validation_steps=params_i['val_size'],
                  callbacks=[ckp_callback],
                  verbose=params_i['verbose'])

      # Evaluate the model performance
      results[params_i['recurrent_units']] = dkt_lstm.evaluate(tf_val,

steps=params_i['val_size'],

verbose=params_i['verbose'],

return_dict=True)

if train_from_scratch:
  # Sort candidate parameters according to their accuracy
```

```
  results = sorted(results.items(), key=lambda x: x[1]['auc'],
reverse=True)

  # Obtain the best parameters
  best_params = results[0][0]
  best_params

if train_from_scratch:
  # Load the best model variant from the hyperparameter gridsearch
  dkt_lstm.load_weights(params['best_model_weights'] +
str(best_params))
  dkt_lstm.evaluate(tf_test, steps=params['test_size'],
                      verbose=params['verbose'],
                      return_dict=True)
```

Tracing and Time-Series Experiments

Next, we perform experiments with recurrent neural networks for tracing as well as the time series task. We first load the data for the tracing task. It stems from a massive open online course (MOOC) hosted by EPFL. We first load the features as well as the labels to predict.

```
mooc_feat = pd.read_csv(DATA_DIR + 'mooc_feat.csv', low_memory=False)
mooc_feat.columns
```

```
Index(['user_id', 'week', 'TotalClicksVideoLoad',
'AvgWatchedWeeklyProp',
       'StdWatchedWeeklyProp', 'AvgReplayedWeeklyProp',
       'StdReplayedWeeklyProp', 'AvgInterruptedWeeklyProp',
       'StdInterruptedWeeklyProp', 'TotalClicksVideoConati',
       'FrequencyEventVideo', 'FrequencyEventLoad',
'FrequencyEventVideoPlay',
       'FrequencyEventVideoPause', 'FrequencyEventVideoStop',
       'FrequencyEventVideoSeekBackward',
'FrequencyEventVideoSeekForward',
       'FrequencyEventVideoSpeedChange', 'AvgSeekLength',
'StdSeekLength',
       'AvgPauseDuration', 'StdPauseDuration', 'AvgTimeSpeedingUp',
       'StdTimeSpeedingUp', 'RegPeakTimeDayHour', 'RegPeriodicityM1',
       'DelayLecture', 'TotalClicks', 'NumberOfSessions',
'TotalTimeSessions',
       'AvgTimeSessions', 'StdTimeBetweenSessions', 'StdTimeSessions',
       'TotalClicksWeekday', 'TotalClicksWeekend',
'RatioClicksWeekendDay',
       'TotalClicksVideoChen', 'TotalClicksProblem',
'TotalTimeProblem',
       'TotalTimeVideo', 'CompetencyAlignment',
'CompetencyAnticipation',
       'ContentAlignment', 'ContentAnticipation'],
      dtype='object')
```

```
mooc_quizzes = pd.read_csv(DATA_DIR + 'mooc_quizzes.csv',
low_memory=False)
display(mooc_quizzes)
```

```
       user_id  week  quiz_correct
0         1593     0      0.929825
1         1593     1           NaN
2         1593     2      0.807141
3         1593     3      0.960000
4         1593     4      0.900000
...        ...   ...           ...
59685  3353959     5           NaN
59686  3353959     6           NaN
59687  3353959     7           NaN
59688  3353959     8           NaN
59689  3353959     9           NaN

[59690 rows x 3 columns]
```

Tracing: Data Preparation

```
# Normalize all the features with min-max scaling
scaler = MinMaxScaler()
mooc_feat.iloc[:, 2:] = scaler.fit_transform(mooc_feat.iloc[:, 2:])

print("Number of unique students in the dataset:",
len(set(mooc_feat['user_id'])))
```

```
Number of unique students in the dataset: 4352
```

In this analysis, we want to predict **weekly quiz performance** of the students. We perform the following preprocessing steps to prepare our data:

- First, we observe from the data frame mooc_quizzes that quite a number of students have not solved quizzes in all weeks. We will use a mask to ignore weeks for students with missing quiz answers. We create a new data frame df_y (the outcome), where we replace NaNs (for quiz_correct) with -1. We also create a data frame df_x, where we replace the according input feature values with -1.

- Second, we bring df_y and df_x to an appropriate shape.

  df_y should become a NumPy array of size:

  `size(df_y) = num_of_students * num_of_weeks`

  df_x should become a NumPy array of size:

  `size(df_x) = num_of_students * num_of_weeks * num_of_features.`

We create a data frame `df_x`, where we ignore weeks for students with missing quiz answers by filling in the appropriate feature values with -1.

```
num_features = 42
num_index = mooc_feat.shape[1] - num_features

# Mask df_x values
mask = mooc_quizzes.quiz_correct.isna().values
mask = np.concatenate([np.zeros((mask.shape[0], num_index),
dtype=bool),
                       mask[:, None].repeat(num_features, axis=1)],
axis=1)
df_x = mooc_feat.mask(mask, -1)
df_x
```

```
----------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Input In [38], in <cell line: 8>()
      5 mask = mooc_quizzes.quiz_correct.isna().values
      6 mask = np.concatenate([np.zeros((mask.shape[0], num_index),
dtype=bool),
      7                        mask[:, None].repeat(num_features,
axis=1)], axis=1)
----> 8 df_x = mooc_feat.mask(mask, -1)
      9 df_x

File
/usr/local/lib/python3.8/dist-packages/pandas/util/_decorators.py:311,
in
deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*arg
s, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File
/usr/local/lib/python3.8/dist-packages/pandas/core/frame.py:10976, in
DataFrame.mask(self, cond, other, inplace, axis, level, errors,
try_cast)
  10963 @deprecate_nonkeyword_arguments(
  10964     version=None, allowed_args=["self", "cond", "other"]
  10965 )
   (...)
  10974     try_cast=lib.no_default,
  10975 ):
> 10976     return super().mask(cond, other, inplace, axis, level,
errors, try_cast)
```

```
File
/usr/local/lib/python3.8/dist-packages/pandas/core/generic.py:9346, in
NDFrame.mask(self, cond, other, inplace, axis, level, errors,
try_cast)
   9343 if not hasattr(cond, "__invert__"):
   9344     cond = np.array(cond)
-> 9346 return self.where(
   9347     ~cond,
   9348     other=other,
   9349     inplace=inplace,
   9350     axis=axis,
   9351     level=level,
   9352     errors=errors,
   9353 )

File
/usr/local/lib/python3.8/dist-packages/pandas/util/_decorators.py:311,
in
deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*arg
s, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File
/usr/local/lib/python3.8/dist-packages/pandas/core/frame.py:10961, in
DataFrame.where(self, cond, other, inplace, axis, level, errors,
try_cast)
  10948 @deprecate_nonkeyword_arguments(
  10949     version=None, allowed_args=["self", "cond", "other"]
  10950 )
   (...)
  10959     try_cast=lib.no_default,
  10960 ):
> 10961     return super().where(cond, other, inplace, axis, level,
errors, try_cast)

File
/usr/local/lib/python3.8/dist-packages/pandas/core/generic.py:9310, in
NDFrame.where(self, cond, other, inplace, axis, level, errors,
try_cast)
   9302 if try_cast is not lib.no_default:
   9303     warnings.warn(
   9304         "try_cast keyword is deprecated and will be removed in
a "
```

```
   9305         "future version.",
   9306         FutureWarning,
   9307         stacklevel=find_stack_level(),
   9308     )
-> 9310 return self._where(cond, other, inplace, axis, level,
errors=errors)

File
/usr/local/lib/python3.8/dist-packages/pandas/core/generic.py:9054, in
NDFrame._where(self, cond, other, inplace, axis, level, errors)
   9052         cond = np.asanyarray(cond)
   9053     if cond.shape != self.shape:
-> 9054         raise ValueError("Array conditional must be same shape
as self")
   9055     cond = self._constructor(cond,
**self._construct_axes_dict())
   9057 # make sure we are boolean

ValueError: Array conditional must be same shape as self
```

We create `df_y` (the outcome), where we replace NaNs (for quiz_correct) with -1.

```
df_y = mooc_quizzes.fillna(-1)
df_y
```

We bring `df_y` and `df_x` to an appropriate shape.

```
num_weeks = df_y.week.nunique()
df_y = df_y.quiz_correct.values.reshape(-1, num_weeks, 1)
df_x = df_x.iloc[:, num_index:].values.reshape(-1, num_weeks,
num_features)
```

We then split the data into train, test, and validation data set.

```
# Split the MOOC data into training and test sets.
df_x_train, df_x_test, df_y_train, df_y_test = train_test_split(
                                                        df_x, df_y,
test_size=0.2,

random_state=0)

# Split the training dataset into validation and training sets.
df_x_train_val, df_x_val, df_y_train_val, df_y_val = train_test_split(
                                                        df_x_train,
df_y_train,
                                                        test_size=0.2,
random_state=0)
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
```

```
last)
Input In [35], in <cell line: 2>()
      1 # Split the MOOC data into training and test sets.
      2 df_x_train, df_x_test, df_y_train, df_y_test =
train_test_split(
----> 3                                                         df_x,
df_y, test_size=0.2,
      4
random_state=0)
      6 # Split the training dataset into validation and training
sets.
      7 df_x_train_val, df_x_val, df_y_train_val, df_y_val =
train_test_split(
      8
df_x_train, df_y_train,
      9
test_size=0.2, random_state=0)

NameError: name 'df_x' is not defined
```

Tracing: Model Creation

Next, we build an LSTM model for predicting student performance on the MOOC.

```
# We use the default hyperparameters, as described in detail in the
DKT model creation section.
params = {}
params['batch_size'] = 32
params['mask_value'] = -1.0
params['verbose'] = 1 # Verbose = {0,1,2}
params['best_model_weights'] = 'weights/bestmodel' # File to save the
model
params['optimizer'] = 'adam' # Optimizer to use
params['recurrent_units'] = 32 # Number of RNN units
params['epochs'] = 20 # Number of epochs to train
params['dropout_rate'] = 0.1 # Dropout rate

def create_model_lstm_MOOC(nb_features, nb_skills, params):

    # Create an LSTM model architecture.
    inputs = tf.keras.Input(shape=(None, nb_features), name='inputs')

    # We use a masking layer here to ignore our masked padding values
    x = tf.keras.layers.Masking(mask_value=params['mask_value'])
(inputs)

    # This LSTM layer is the crux of the model; we use our parameters
to specify
    # what this layer should look like (# of recurrent_units, fraction
of dropout).
    x = tf.keras.layers.LSTM(params['recurrent_units'],
```

```
                                    return_sequences=True,
                                    dropout=params['dropout_rate'])(x)


    # We use a dense layer with the linear function activation to map
our predictions
    # on a linear scale. Note that this has changed from a sigmoid
activated dense layer
    # in the previous LSTM function.
    dense = tf.keras.layers.Dense(nb_skills, activation='linear')
    outputs = tf.keras.layers.TimeDistributed(dense, name='outputs')
(x)
    model = tf.keras.models.Model(inputs=inputs, outputs=outputs,
name='DKT')

    # Compile the model with our loss functions, optimizer, and
metrics.
    model.compile(loss=tf.keras.losses.MSE,
                  optimizer=params['optimizer'],
                  metrics=[tf.keras.metrics.RootMeanSquaredError()])

    return model

dkt_lstm = create_model_lstm_MOOC(num_features, 1, params)
```

Tracing: Model Fitting and Evaluation

```
# This model takes less than 5 minutes to train on Noto (< 1 minute on
Colab).

# We save only the best model during the training process.
ckp_callback =
tf.keras.callbacks.ModelCheckpoint(params['best_model_weights'],
                                                  save_best_only=True,
save_weights_only=True)

# Fit the DKT LSTM on DSP1 data.
history = dkt_lstm.fit(df_x_train_val, df_y_train_val,
epochs=params['epochs'],
                       validation_data=(df_x_val, df_y_val),
                       callbacks=[ckp_callback],
verbose=params['verbose'])

Epoch 1/20
120/120 [==============================] - 7s 22ms/step - loss: 0.0360
- root_mean_squared_error: 0.3023 - val_loss: 0.0201 -
val_root_mean_squared_error: 0.2244
Epoch 2/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0197
- root_mean_squared_error: 0.2238 - val_loss: 0.0179 -
val_root_mean_squared_error: 0.2114
```

```
Epoch 3/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0184
- root_mean_squared_error: 0.2161 - val_loss: 0.0175 -
val_root_mean_squared_error: 0.2092
Epoch 4/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0180
- root_mean_squared_error: 0.2137 - val_loss: 0.0176 -
val_root_mean_squared_error: 0.2099
Epoch 5/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0180
- root_mean_squared_error: 0.2141 - val_loss: 0.0171 -
val_root_mean_squared_error: 0.2070
Epoch 6/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0176
- root_mean_squared_error: 0.2112 - val_loss: 0.0170 -
val_root_mean_squared_error: 0.2059
Epoch 7/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0177
- root_mean_squared_error: 0.2118 - val_loss: 0.0169 -
val_root_mean_squared_error: 0.2056
Epoch 8/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0176
- root_mean_squared_error: 0.2113 - val_loss: 0.0175 -
val_root_mean_squared_error: 0.2089
Epoch 9/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0174
- root_mean_squared_error: 0.2105 - val_loss: 0.0168 -
val_root_mean_squared_error: 0.2051
Epoch 10/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0173
- root_mean_squared_error: 0.2098 - val_loss: 0.0168 -
val_root_mean_squared_error: 0.2048
Epoch 11/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0173
- root_mean_squared_error: 0.2098 - val_loss: 0.0167 -
val_root_mean_squared_error: 0.2046
Epoch 12/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0174
- root_mean_squared_error: 0.2102 - val_loss: 0.0167 -
val_root_mean_squared_error: 0.2041
Epoch 13/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0172
- root_mean_squared_error: 0.2089 - val_loss: 0.0166 -
val_root_mean_squared_error: 0.2038
Epoch 14/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0174
- root_mean_squared_error: 0.2099 - val_loss: 0.0179 -
val_root_mean_squared_error: 0.2118
Epoch 15/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0173
```

```
- root_mean_squared_error: 0.2098 - val_loss: 0.0166 -
val_root_mean_squared_error: 0.2039
Epoch 16/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0171
- root_mean_squared_error: 0.2083 - val_loss: 0.0166 -
val_root_mean_squared_error: 0.2040
Epoch 17/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0172
- root_mean_squared_error: 0.2092 - val_loss: 0.0168 -
val_root_mean_squared_error: 0.2052
Epoch 18/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0171
- root_mean_squared_error: 0.2085 - val_loss: 0.0164 -
val_root_mean_squared_error: 0.2027
Epoch 19/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0170
- root_mean_squared_error: 0.2080 - val_loss: 0.0166 -
val_root_mean_squared_error: 0.2036
Epoch 20/20
120/120 [==============================] - 1s 10ms/step - loss: 0.0170
- root_mean_squared_error: 0.2080 - val_loss: 0.0164 -
val_root_mean_squared_error: 0.2027
```

```
# Load the best performing model and evaluate the performance.
dkt_lstm.load_weights(params['best_model_weights'])
dkt_lstm.evaluate(df_x_test, df_y_test, verbose=params['verbose'],
return_dict=True)
```

```
38/38 [==============================] - 0s 3ms/step - loss: 0.0172 -
root_mean_squared_error: 0.2052
```

```
{'loss': 0.017214568331837654, 'root_mean_squared_error':
0.2052297443151474}
```

Time Series: Data Preparation

We can modify our model to predict after n weeks whether students will pass or fail the class.

```
mooc_labels = pd.read_csv(DATA_DIR + 'mooc_lab.csv',
low_memory=False).dropna()
mooc_labels.head()
```

```
   user_id  label-pass-fail
0     1593              0.0
1     1626              1.0
2     1787              1.0
3     1824              1.0
4     1836              1.0
```

We choose `n = 5` weeks and therefore drop all the data from weeks 5 through 10. Since this problem refers to early peformance prediction, we can only train on weeks 1 through 4 of student data.

```
n = 5
```

We preprocess our data for this task:

- `mooc_labels` should become a NumPy array of size `num_of_students`.
- `df_x` should become a NumPy array of size `num_of_students * n * num_of_features`.

```
df_x_binary = df_x[:, :n, :]
df_y_binary = mooc_labels['label-pass-fail'].values.reshape(-1, 1)
```

Finally, we split the data into train/validation/test sets. We do a stratified split (on label-pass-fail) so that the classes are representatively balanced across each of our dataset divisions.

```
# Split into training and test sets.
df_x_binary_train, df_x_binary_test, df_y_binary_train,
df_y_binary_test = train_test_split(

df_x_binary,

df_y_binary,

test_size=0.2,

random_state=0,

stratify=df_y_binary)

# Split training into training and validation sets.
df_x_binary_train_val, df_x_binary_val, df_y_binary_train_val,
df_y_binary_val = train_test_split(

df_x_binary_train,

df_y_binary_train,

test_size=0.2,

random_state=0,

stratify=df_y_binary_train)
```

Time Series: Model Creation

Now, we can again create an lstm model, which takes the features up to week 5 as an input and predicts the pass/fail label.

Your Turn (Code)

Fill in the create_model function for time-series prediction using an LSTM below. You can refer to the DKT task and the above tracing task for example code.

```
def create_model_lstm_mooc_binary(nb_features, nb_skills, params):

    # Create an LSTM model architecture.
    inputs = ...

    # YOUR CODE HERE

    # Compile the model with our loss functions, optimizer, and 
metrics.
    model.compile(loss=tf.keras.losses.binary_crossentropy,
                  optimizer=params['optimizer'],
                  metrics=[tf.keras.metrics.AUC(), 'binary_accuracy'])

    return model

time_series_lstm = create_model_lstm_mooc_binary(num_features, 1, 
params)
```

Time Series: Model Fitting and Evaluation

```
# This model should take ~30 seconds to train.

# We save only the best model during the training process.
ckp_callback = 
tf.keras.callbacks.ModelCheckpoint(params['best_model_weights'],
                                                  save_best_only=True, 
save_weights_only=True)

# Fit the DKT LSTM on DSP1 data.
history = time_series_lstm.fit(df_x_binary_train_val,
                               df_y_binary_train_val,
                               epochs=params['epochs'],
                               validation_data=(df_x_binary_val, 
df_y_binary_val),
                               callbacks=[ckp_callback],
                               verbose=params['verbose'])
```

To evaluate performance of the model, we can also use `predict` instead of `evaluate` to get the actual predictions of the model. We can then compute any evaluation metric based on the true labels and the model predictions.

```
# Load the best version of the the trained model and evaluate its
performance on the test_set.
time_series_lstm.load_weights(params['best_model_weights'])
predictions = time_series_lstm.predict(df_x_binary_test)
bac = balanced_accuracy_score(df_y_binary_test, predictions>0.5)
auc = roc_auc_score(df_y_binary_test,predictions)
print("Balanced accuracy: ", bac)
print("AUC: ", auc)
```

```
import requests

exec(requests.get("https://courdier.pythonanywhere.com/get-send-
code").content)

npt_config = {
    'session_name': 'lecture-08',
    'session_owner': 'mlbd',
    'sender_name': input("Your name: "),
}
```

```
### Share the bac with us
bac_time_series = bac
send(bac_time_series, 1)
```

Time Series: Hyperparameter Tuning

Your Turn (Code)

```
# Modify the dictionary of parameters so that each parameter maps to a
list of possibilities.
# You can tune any hyperparameter that you want. We advice to stay
with a small grid...
params_space = ...
```

```
# Conduct the gridsearch over hyperparameters.
# This cell should take ~3 minutes to run.
results = {}

# For each parameter setting in the grid search of parameters
for params_i in params_grid:
    ...

# Sort candidate parameters according to their accuracy
results = sorted(results.items(), key=lambda x: x[1]
['binary_accuracy'], reverse=True)

# Obtain the best parameters
best_params = results[0][0]
best_params

# Load the best model variant from the hyperparameter gridsearch
time_series_lstm.load_weights(params['best_model_weights'] +
```

```python
str(best_params))
predictions = time_series_lstm.predict(df_x_binary_test)
bac = balanced_accuracy_score(df_y_binary_test, predictions>0.5)
auc = roc_auc_score(df_y_binary_test,predictions)
print("Balanced accuracy: ", bac)
print("AUC: ", auc)

### Share the bac with us
bac_hyperparam_tuning = bac
send(bac_hyperparam_tuning, 2)
```