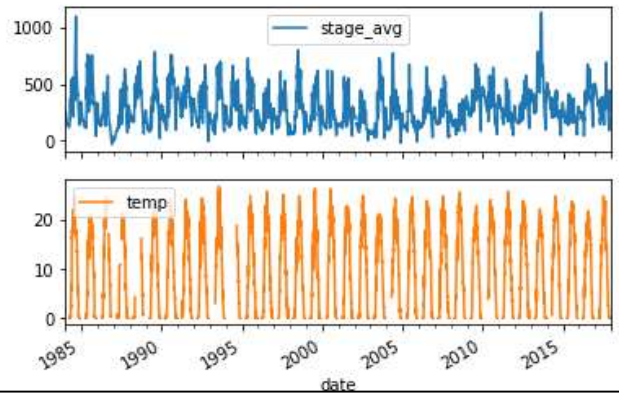
# Решение задачи NoFloodWithAI: Flash floods on the Amur River

Команда «HTAR»

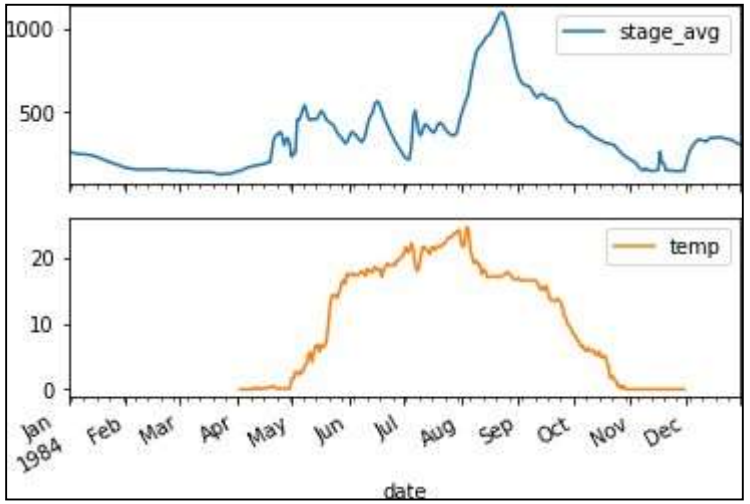
# Визуализация входных данных

- Графики средних по станциям уровней воды и температуры

```
[7]  for i in stations[:1]:
         plot_cols=['stage_avg', 'temp']
         station = daily[daily.station_id == i]
         plot_features = station[plot_cols]
         plot_features.index=station['date']
         _ = plot_features.plot(subplots=True)

         plot_features = station[plot_cols][:365]
         plot_features.index = station['date'][:365]
         _ = plot_features.plot(subplots=True)
```

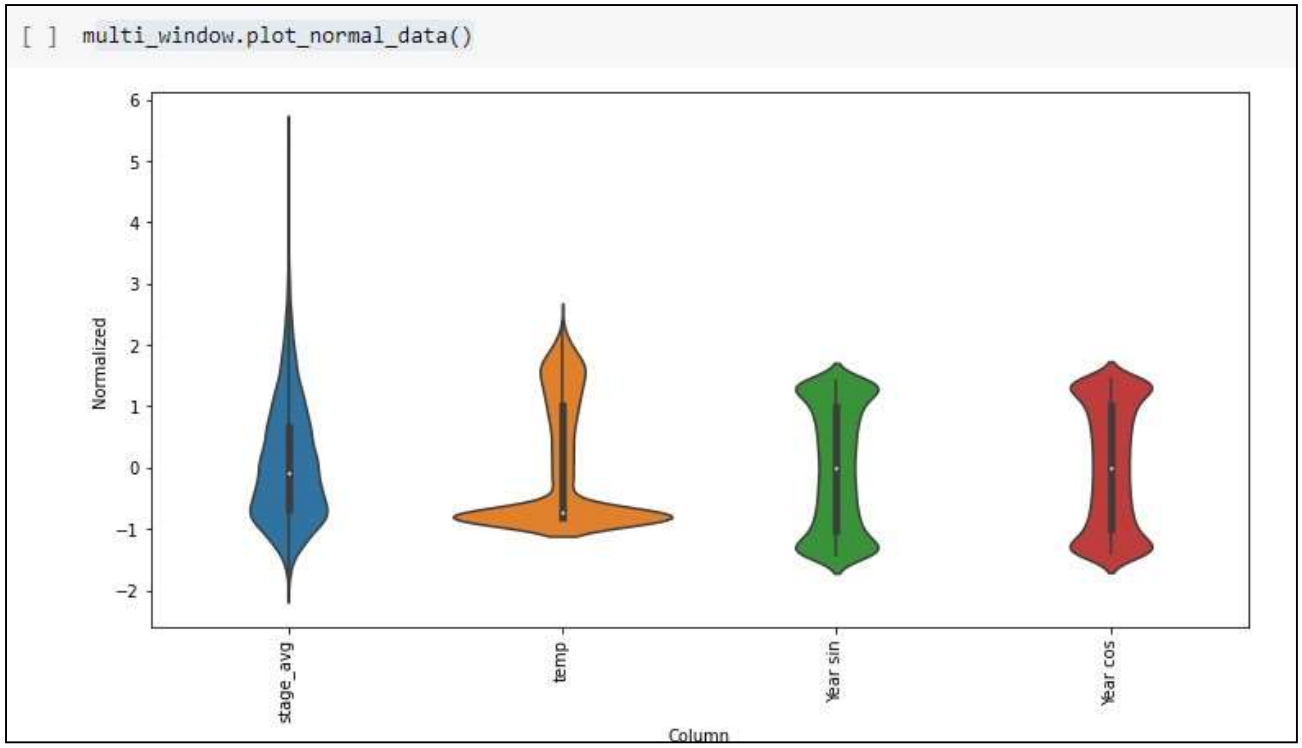



- Графики среднегодовых уровней воды и температуры



# Визуализация входных данных

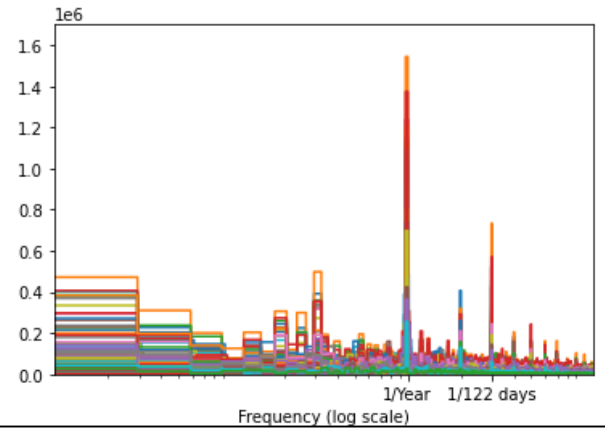
- Нормализованные данные



# Визуализация входных данных

- Визуализация основных цикличных промежутков

```
[8]  for i in stations:
         station = daily[daily.station_id == i]
         fft = tf.signal.rfft(station['stage_avg'])
         f_per_dataset = np.arange(0, len(fft))

         n_samples_d = len(station['stage_avg'])
         days_per_year = 365.254
         years_per_dataset = n_samples_d/(days_per_year)
         f_per_year = f_per_dataset/years_per_dataset
         plt.step(f_per_year, np.abs(fft))
         plt.xscale('log')
         plt.ylim(0, 1700000)
         plt.xlim([0.01, max(plt.xlim())])
         plt.xticks([1,3], labels=['1/Year','1/122 days'])
         _ = plt.xlabel('Frequency (log scale)')
```



# Реализация нейронной сети

- Класс, описывающий окно данных

```
[49] class WindowGenerator():
       def __init__(self, input_width, label_width, shift, station,
                    daily=daily,
                    label_columns=None):
         # Store the raw data.
         self.daily_station = daily[daily.station_id==station].drop(columns='station_id')
         n = len(self.daily_station)
         self.train_daily = self.daily_station[0:int(n*0.7)]
         self.val_daily = self.daily_station[int(n*0.7):int(n*0.9)]
         self.test_daily = self.daily_station[int(n*0.9):]
         self.train_mean = self.train_daily.mean()
         self.train_std = self.train_daily.std()
         self.train_daily = (self.train_daily - self.train_mean) / self.train_std
         self.val_daily = (self.val_daily - self.train_mean) / self.train_std
         self.test_daily = (self.test_daily - self.train_mean) / self.train_std

         # Work out the label column indices.
         self.label_columns = label_columns
         if label_columns is not None:
           self.label_columns_indices = {name: i for i, name in
                                         enumerate(label_columns)}
         self.column_indices = {name: i for i, name in
                                enumerate(self.train_daily.columns)}

         # Work out the window parameters.
         self.input_width = input_width
         self.label_width = label_width
         self.shift = shift

         self.total_window_size = input_width + shift

         self.input_slice = slice(0, input_width)
         self.input_indices = np.arange(self.total_window_size)[self.input_slice]

         self.label_start = self.total_window_size - self.label_width
         self.labels_slice = slice(self.label_start, None)
         self.label_indices = np.arange(self.total_window_size)[self.labels_slice]

       def __repr__(self):
         return '\n'.join([
             f'Total window size: {self.total_window_size}',
             f'Input indices: {self.input_indices}',
             f'Label indices: {self.label_indices}',
             f'Label column name(s): {self.label_columns}'])
```

# Реализация нейронной сети

- Методы класса WindowGenerator

```
def split_window(self, features):
  inputs = features[:, self.input_slice, :]
  labels = features[:, self.labels_slice, :]
  if self.label_columns is not None:
    labels = tf.stack(
        [labels[:, :, self.column_indices[name]] for name in self.label_columns],
        axis=-1)
  inputs.set_shape([None, self.input_width, None])
  labels.set_shape([None, self.label_width, None])

  return inputs, labels

WindowGenerator.split_window = split_window
```

```
[ ] def plot_normal_data(self):
        df_std = (self.daily_station - self.train_mean) / self.train_std
        df_std = df_std.melt(var_name='Column', value_name='Normalized')
        plt.figure(figsize=(12, 6))
        ax = sns.violinplot(x='Column', y='Normalized', data=df_std)
        plot = ax.set_xticklabels(self.daily_station.keys(), rotation=90)
    WindowGenerator.plot_normal_data = plot_normal_data
```

```
[ ] def print_nans(self):
        print(self.daily_station[self.daily_station.isna().any(axis=1)])
    WindowGenerator.print_nans = print_nans
```

```
[ ] def plot(self, model=None, plot_col='stage_avg', max_subplots=3):
      inputs, labels = self.example
      plt.figure(figsize=(12, 8))
      plot_col_index = self.column_indices[plot_col]
      max_n = min(max_subplots, len(inputs))
      for n in range(max_n):
        plt.subplot(3, 1, n+1)
        plt.ylabel(f'{plot_col} [normed]')
        plt.plot(self.input_indices, inputs[n, :, plot_col_index],
                 label='Inputs', marker='.', zorder=-10)

        if self.label_columns:
          label_col_index = self.label_columns_indices.get(plot_col, None)
        else:
          label_col_index = plot_col_index

        if label_col_index is None:
          continue

        plt.scatter(self.label_indices, labels[n, :, label_col_index],
                    edgecolors='k', label='Labels', c='#2ca02c', s=64)
        if model is not None:
          predictions = model(inputs)
          plt.scatter(self.label_indices, predictions[n, :, label_col_index],
                      marker='X', edgecolors='k', label='Predictions',
                      c='#ff7f0e', s=64)

        if n == 0:
          plt.legend()

      plt.xlabel('Time [h]')

    WindowGenerator.plot = plot
```

```
[ ] def make_dataset(self, data):
      data = np.array(data, dtype=np.float32)
      ds = tf.keras.preprocessing.timeseries_dataset_from_array(
          data=data,
          targets=None,
          sequence_length=self.total_window_size,
          sequence_stride=1,
          shuffle=True,
          batch_size=32,)

      ds = ds.map(self.split_window)

      return ds

    WindowGenerator.make_dataset = make_dataset
```

# Реализация нейронной сети

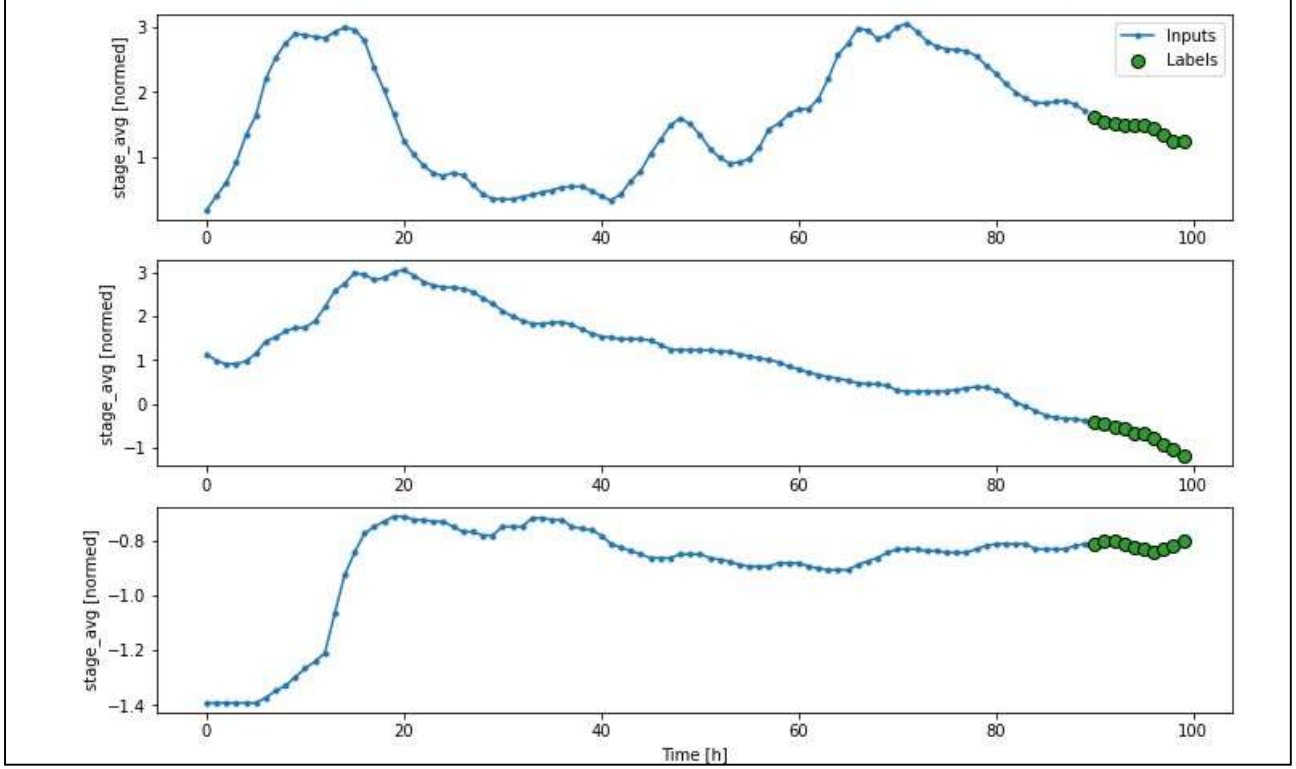
- Методы класса WindowGenerator

```
[ ]  @property
     def train(self):
       return self.make_dataset(self.train_daily)

     @property
     def val(self):
       return self.make_dataset(self.val_daily)

     @property
     def test(self):
       return self.make_dataset(self.test_daily)

     @property
     def example(self):
       """Get and cache an example batch of `inputs, labels` for plotting."""
       result = getattr(self, '_example', None)
       if result is None:
         # No example batch was found, so get one from the `.train` dataset
         result = next(iter(self.train))
         # And cache it for next time
         self._example = result
       return result

     WindowGenerator.train = train
     WindowGenerator.val = val
     WindowGenerator.test = test
     WindowGenerator.example = example
```

# Реализация нейронной сети

- Входные данные и ожидаемые результаты

```
[ ]  OUT_STEPS = 10
     multi_window = WindowGenerator(input_width=90,
                                    label_width=OUT_STEPS,
                                    shift=OUT_STEPS, station=5001, label_columns=['stage_avg'])

     multi_window.plot()
     multi_window
     num_features = 4
```



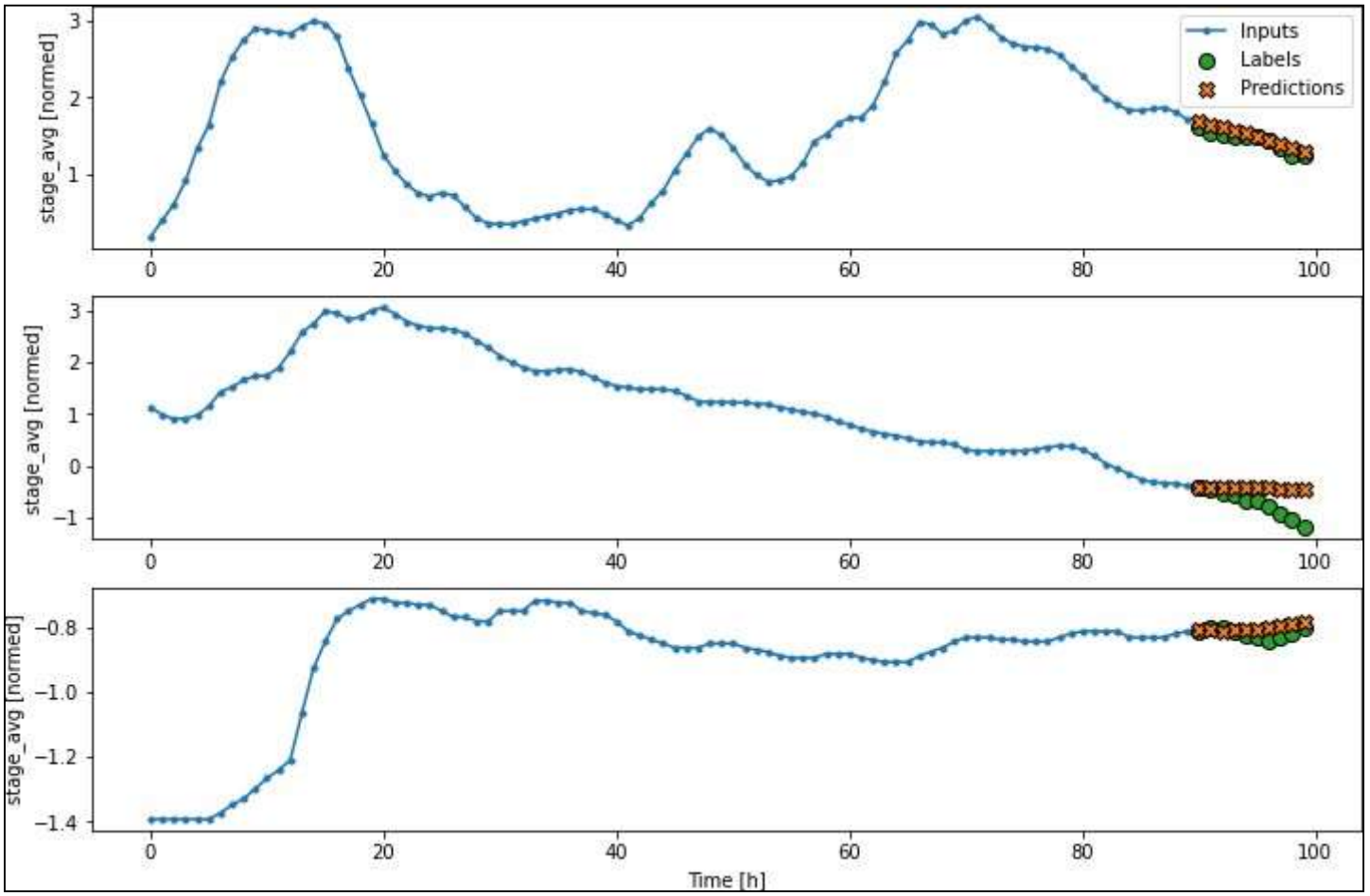
# Реализация нейронной сети

- Модель нейросети (Tensorflow)

```
[ ]  multi_linear_model = tf.keras.Sequential([
        # Take the last time-step.
        # Shape [batch, time, features] => [batch, 1, features]
        tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
        # Shape => [batch, 1, out_steps*features]
        tf.keras.layers.Dense(OUT_STEPS*num_features,
                              kernel_initializer=tf.initializers.zeros),
        # Shape => [batch, out_steps, features]
        tf.keras.layers.Reshape([OUT_STEPS, num_features])
    ])
    multi_val_performance = {}
    multi_performance = {}
    history = compile_and_fit(multi_linear_model, multi_window)


    multi_val_performance['Dense'] = multi_linear_model.evaluate(multi_window.val)
    multi_performance['Dense'] = multi_linear_model.evaluate(multi_window.test, verbose=0)
    multi_window.plot(multi_linear_model)
```

# Реализация нейронной сети

- Результат работы нейросети



# Экономическое обоснование необходимости решения проблемы на реке Амур

- Безусловно, наводнение на реке Амур в 2013 году обострило необходимость повышения внимания к этой проблеме. Согласно официальным данным, на середину октября 2013 года общее число пострадавших в результате наводнения превысило 168 тысяч человек. Десятки тысяч человек переселены из зоны бедствия. Суммарный экономический ущерб на конец октября 2013 года, по официальным данным, составлял 40 млрд. рублей.
- Необходимо отметить, что в последние десятилетия создано новое поколение методов оценки опасности и прогнозирования наводнений, основанных на математических компьютерных моделях. Такие модели позволяют воспроизводить особенности произошедших стихийных бедствий и рассчитывать возможные сценарии развития будущих. В экономически развитых странах они становятся основным инструментом для принятия решений о мерах защиты от наводнений.
- Создание моделей для основных речных систем России, воспроизводящих особенности формирования стока и ориентированных на имеющиеся данные измерений, — первоочередная задача, которая была реализована нами в конкурсе Сбербанк «NoFloodWithAl: паводки на реке Амур».