Arrival Prediction

Arrival Prediction

We can now frame our arrival prediction problem

We want to predict the number of arrivals in the next interval

- We will focus on predicting the total number of arrivals
- The same models can be applied to any of the individual counts

Which ML task is this? Which loss function should we use?

Arrival Prediction

We can now frame our arrival prediction problem

We want to predict the number of arrivals in the next interval

- We will focus on predicting the total number of arrivals
- The same models can be applied to any of the individual counts

Which ML task is this? Which loss function should we use?

This is a regression problem

- ...Which does not imply that the MSE is the best choice
- In fact, we should always check the target distribution first

Which Distribution

We might be tempted to:

- Consider the target attribute (e.g. number of arrivals in bin)
- Run a statistical tests for multiple distributions

...But it is technically wrong

Which Distribution

We might be tempted to:

- Consider the target attribute (e.g. number of arrivals in bin)
- Run a statistical tests for multiple distributions

...But it is technically wrong

...Since regressors are trained to learn a conditional distribution

Therefore, what we should do instead is:

- Partition the target data based on the value of one or more relevant features
- ...Then proceed as above for each group

Unfortunately, this is tricky in practice

- What if we don't know which features are important?
- What if there are a lot of relevant features

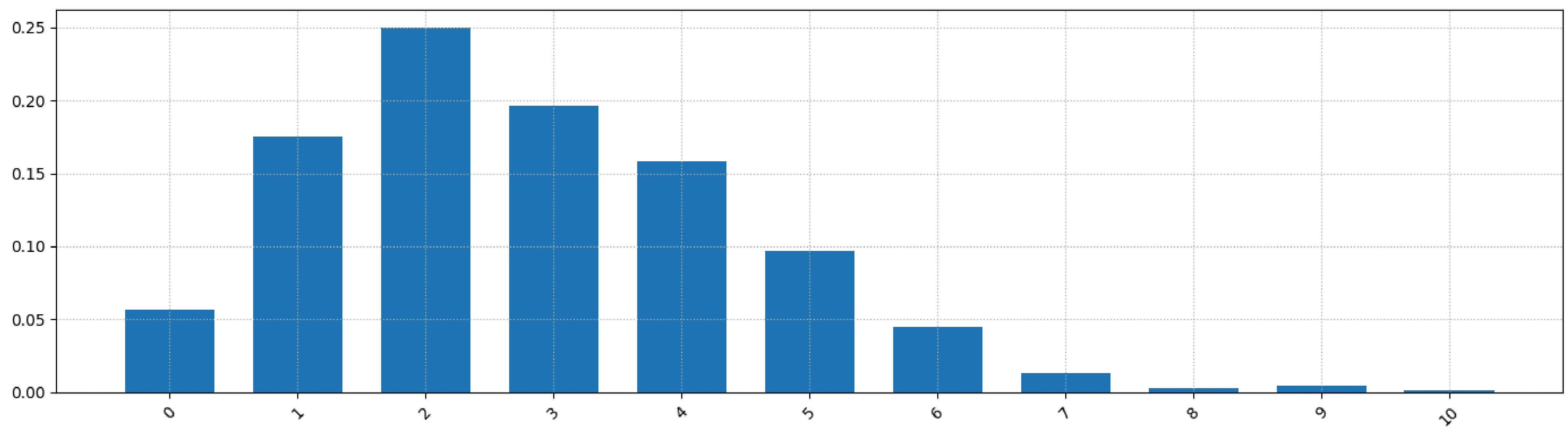
In practice, the first approach is often used as an approximation

Analyzing the Conditional Arrival Distribution

...But in our case we know that the hour of the day is a good predictor

Let's check the (conditional) distribution for a few values (here 6m):

In [2]:
```
tmp = codes_b[codes_b.index.hour == 6]['total']
tmpv = tmp.value_counts(sort=False, normalize=True).sort_index()
util.plot_bars(tmpv, figsize=figsize)
```



- This is not a normal distribution

Poisson Distribution

When we need to count occurrences over time...

It's almost always worth checking the Poisson distribution, which models:

- The number of occurrences of a certain event in a given interval
- ...Assuming that these events are independent
- ...And they occur at a constant rate

In our case:

- The independence assumption is reasonable (arrivals do not affect each other)
- The constant rate is true for the conditional probability
- ...Assuming that we condition using the right features
- I.e. those that have an actual correlation with the arrivals

Poisson Distribution

The Poisson distribution is defined by a single parameter λ

λ is the rate of occurrence of the events

- The distribution has a discrete support
- The Probability Mass Function is:

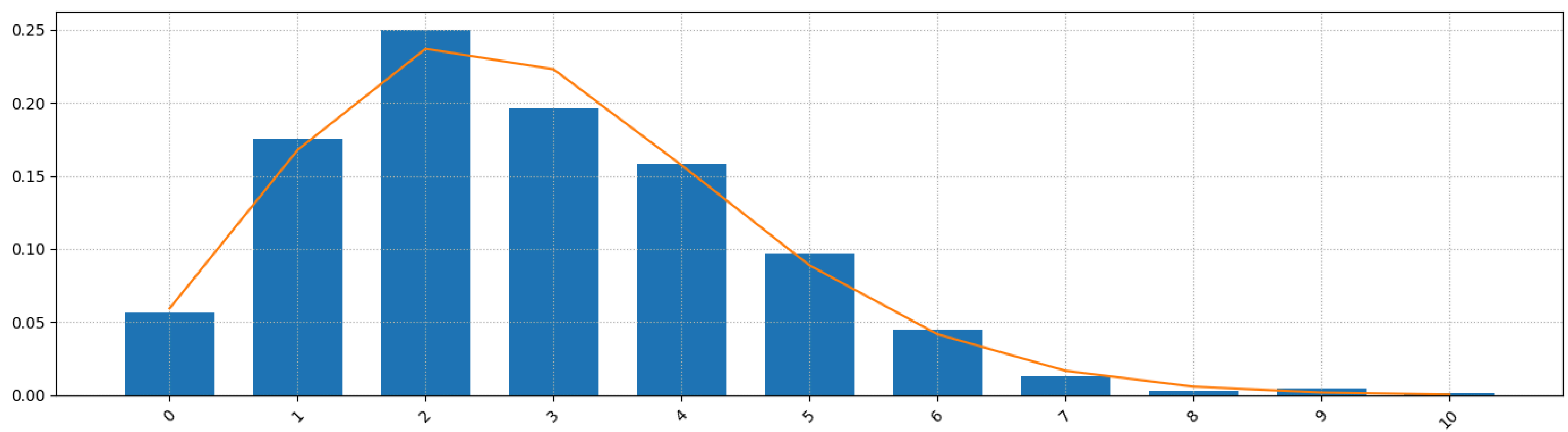
$$p(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

- Both the mean and the standard deviation have the same value (i.e. λ)
- The distribution skewness is $\lambda^{-\frac{1}{2}}$
  - For low λ values, there is a significant positive skew (to the left)
  - The distribution becomes less skewed for large λ

Fitted Poisson Distribution

Let's try to fit a Poisson distribution over our target

In [3]:
```
mu = tmp.mean()
dist = stats.poisson(mu)
x = np.arange(tmp.min(), tmp.max()+1)
util.plot_bars(tmpv, figsize=figsize, series=pd.Series(index=x, data=dist.pmf(x)))
```

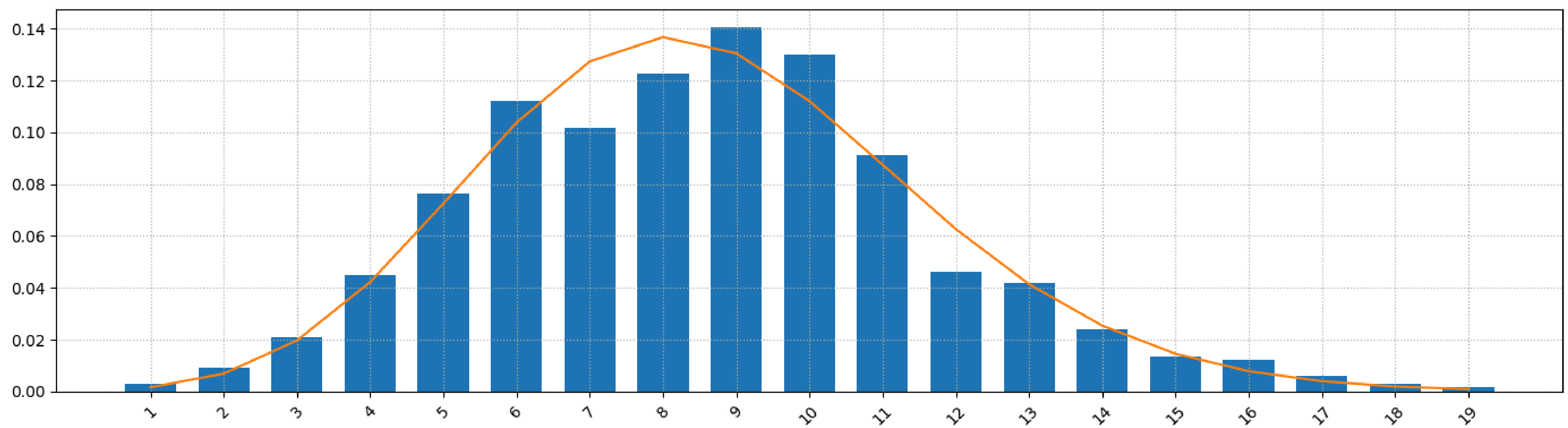


It's a very good match!

Fitted Poisson Distribution

Let's try for 8AM (closer to the peak)

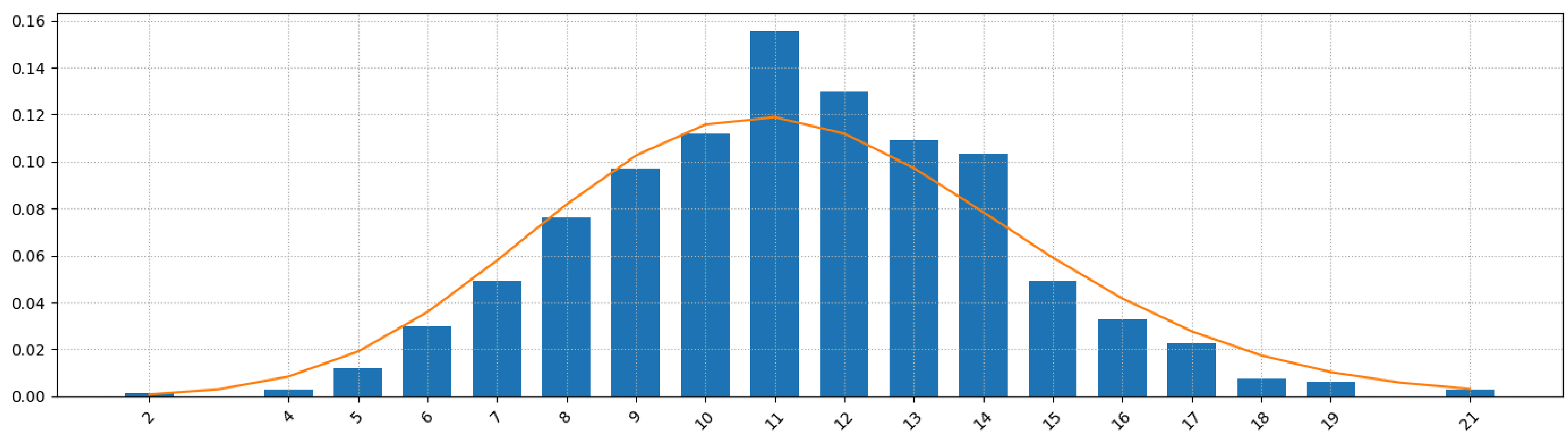
In [4]:
```
tmp = codes_b[codes_b.index.hour == 8]['total']
tmpv = tmp.value_counts(sort=False, normalize=True).sort_index()
mu = tmp.mean()
dist = stats.poisson(mu)
x = np.arange(tmp.min(), tmp.max()+1)
util.plot_bars(tmpv, figsize=figsize, series=pd.Series(index=x, data=dist.pmf(x)))
```



Fitted Poisson Distribution

...And finally for the peak itself (11am)

In [5]:
```
tmp = codes_b[codes_b.index.hour == 11]['total']
tmpv = tmp.value_counts(sort=False, normalize=True).sort_index()
mu = tmp.mean()
dist = stats.poisson(mu)
x = np.arange(tmp.min(), tmp.max()+1)
util.plot_bars(tmpv, figsize=figsize, series=pd.Series(index=x, data=dist.pmf(x)))
```



Neuro-Probabilistic Models

Learning and Estimator

How can we build an estimator for our problem?

Learning and Estimator

How can we build an estimator for our problem?

We could build a table

For example, we could compute average arrivals for every hour of the day

- These correspond to λ for that hour, so we target the correct distribution
- ...But the approach has trouble scaling to multiple features

Learning and Estimator

How can we build an estimator for our problem?

We could build a table

For example, we could compute average arrivals for every hour of the day

- These correspond to λ for that hour, so we target the correct distribution
- ...But the approach has trouble scaling to multiple features

We could train a regressor as usual

For example a Linear Regressor or a Neural Network, with the classical MSE loss

- If we do this, it's easy to include multiple input features
- ...But we would be targeting the wrong type of distribution!

Neuro-Probabilistic Models

In practice there is an alternative

Let's start by build a probabilistic model of our phenomenon:

$$y \sim \text{Pois}(\lambda(x))$$

- The number arrivals in a 1-hour bin (i.e. y)
- ...Is drawn from a Poisson distribution (parameterized with a rate)
- ...But the rate is a function of known input, i.e. $\lambda(x)$

Neuro-Probabilistic Models

In practice there is an alternative

Let's start by build a probabilistic model of our phenomenon:

$$y \sim \text{Pois}(\lambda(x))$$

- The number arrivals in a 1-hour bin (i.e. y)
- ...Is drawn from a Poisson distribution (parameterized with a rate)
- ...But the rate is a function of known input, i.e. $\lambda(x)$

Then we can approximate lambda using an estimator, leading to:

$$y \sim \text{Pois}(\lambda(x, \theta))$$

- $\lambda(x, \theta)$ can be any model, with parameter vector λ

This is a hybrid approach, combining statistics and ML

Neuro-Probabilistic Models

How do we train this kind of model?

Just as usual, i.e. for (empirical) maximum log likelihood:

$$\operatorname*{argmin}_{\theta} - \sum_{i=1}^{m} \log f(\hat{y}_i, \lambda(\hat{x}_i, \theta))$$

- Where $f(\hat{y}_i, \lambda)$ is the probability of value $\hat{y}_i$ according to the distribution
- ...And $\lambda(\hat{x}_i, \theta)$ is the estimate rate for the input $\hat{x}_i$

In detail, in our case we have:

$$\operatorname*{argmin}_{\theta} - \sum_{i=1}^{m} \log \frac{\lambda(\hat{x}_i, \theta)^{\hat{y}_i} e^{-\lambda(\hat{x}_i, \theta)}}{\hat{y}_i!}$$

...Which is differentiable and can be solved via gradient descent!

Building a Neuro-Probabilistic Model

We can build this class of models by using custom loss functions

...But it's easier to use a library such as TensorFlow Probability

- TFP provides a layer the abstracts a generic probability distribution:

```
tfp.layers.DistributionLambda(distribution_function, ...)
```

- And function (classes) to model many statistical distributions, e.g.:

```
tfp.distributions.Poisson(log_rate=None, ...)
```

About the `DistributionLambda` layer

- Its input is a symbolic tensor (like for any other layer)
- Its output is tensor of probability distribution objects
- ...Rather than a tensor of numbers

Building a Neuro-Probabilistic Model

The `util` module contains code to build our neuro-probabilistic model

```python
def build_nn_poisson_model(input_shape, hidden, rate_guess=1):
    model_in = keras.Input(shape=input_shape, dtype='float32')
    x = model_in
    for h in hidden:
        x = layers.Dense(h, activation='relu')(x)
    log_rate = layers.Dense(1, activation='linear')(x)
    lf = lambda t: tfp.distributions.Poisson(rate=rate_guess * tf.math.exp(t))
    model_out = tfp.layers.DistributionLambda(lf)(log_rate)
    model = keras.Model(model_in, model_out)
    return model
```

- An MLP architecture computes the `log_rate` tensor (corresponding to $\log \lambda(x)$)
- Using a log, we make sure the rate is strictly positive
- A `DistributionLambda` yield the output (a distribution object)

Building a Neuro-Probabilistic Model

The `util` module contains code to build our neuro-probabilistic model

```python
def build_nn_poisson_model(input_shape, hidden, rate_guess=1):
    model_in = keras.Input(shape=input_shape, dtype='float32')
    x = model_in
    for h in hidden:
        x = layers.Dense(h, activation='relu')(x)
    log_rate = layers.Dense(1, activation='linear')(x)
    lf = lambda t: tfp.distributions.Poisson(rate=rate_guess * tf.math.exp(t))
    model_out = tfp.layers.DistributionLambda(lf)(log_rate)
    model = keras.Model(model_in, model_out)
    return model
```

- The `DistributionLambda` layer is parameterized with a function
- The function (`lf` in this cse) constructs the distribution object
- ...Based on its input tensor (called `t` in the code)

Building a Neuro-Probabilistic Model

We need to be careful about initial parameter estimates

```
def build_nn_poisson_model(input_shape, hidden, rate_guess=1):
    ...
    lf = lambda t: tfp.distributions.Poisson(rate=rate_guess * tf.math.exp(t))
    ...
```

- Assuming standardized/normalized input, under default weight initialization
- ...The `log_rate` tensor will be initially close to 0
- Meaning out rate λ would be initially close to $e^0 = 1$

We need to make sure that this guess is meaningful for our target

- In the code, this is achieve by scaling the rate
- ...With a guess that must be passed at model construction time

Training a Neuro-Probabilistic Model

Training the model requires to specify the loss function

...Which in our case is the negative log-likelihood

- So, it turns out we do need a custom loss functions
- ...But with TFP this is easy to compute

In particular, as loss function we always use:

```
negloglikelihood = lambda y_true, dist: -dist.log_prob(y_true)
```

- The first parameter is the observed value (e.g. actual number of arrivals)
- The second is the distribution computed by the `DistributonLambda` layer
- ...Which provides the method `log_prob`

Compatibility

There is currently a compatibility issue between TFP and Keras 3

- This can be solved by relying on the `tf-keras` package
- ...Which is a TF-specific implementation of the Keras API

In the `util` module, `tf-keras` is imported instead of regular `keras`

```
import tf_keras as keras
from tf_keras import ...
```

- This is not a very elegant solution
- ...But it will do while waiting for a better approach

Data Preparation

Let's see the approach in practice

We will start by preparing our data:

- As input we will use the field `weekday` in natural form
- ...And the field `hour` using a one-hot encoding

Let's perform the encoding:

In [16]:
```
np_data = pd.get_dummies(codes_bt, columns=['hour'], dtype='int32')
np_data.iloc[:2]
```

Out[16]:

	green	red	white	yellow	total	month	weekday	hour_0	hour_1	hour_2	...	hour_14	hour_15	hour_16	hour_17	hour
Triage																
2018-01-01 00:00:00	2	0	2	0	4	1	0	1	0	0	...	0	0	0	0	0
2018-01-01 01:00:00	7	1	1	1	10	1	0	0	1	0	...	0	0	0	0	0

2 rows × 31 columns

Data Preparation

Now we can separate the training and test data

```
In [17]: sep = '2019-01-01'
         np_tr = np_data[np_data.index < sep]
         np_ts = np_data[np_data.index >= sep]
```

...And then the input and output

```
In [18]: in_cols = [c for c in np_data.columns if c.startswith('hour')] + ['weekday']
         out_col = 'total'

         np_tr_in = np_tr[in_cols].copy()
         np_tr_in['weekday'] = np_tr_in['weekday'] / 6
         np_tr_out = np_tr[out_col].astype('float64')

         np_ts_in = np_ts[in_cols].copy()
         np_ts_in['weekday'] = np_ts_in['weekday'] / 6
         np_ts_out = np_ts[out_col].astype('float64')
```

Data Preparation

The input data need to be standardized/normalized as usual

In our case, we do this only for weekday (the hours are already $\in \{0, 1\}$)

```
np_tr_in['weekday'] = np_tr_in['weekday'] / 6
```

The output does not require standarization

...But we need to represent it using floating point numbers

```
np_tr_out = np_tr[out_col].astype('float64')
```

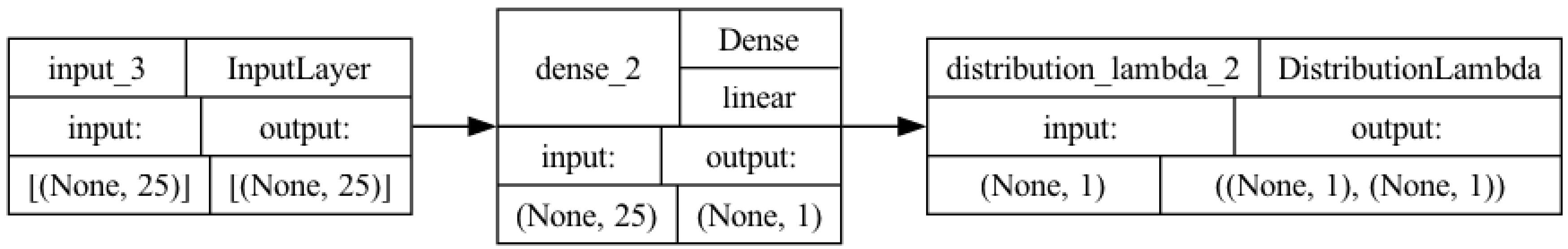
- This is an implementation requirement for TensorFlow

Building the Model

We can now build the Neuro-Probabilistic model

In [19]:
```python
nnp = util.build_nn_poisson_model(input_shape=(len(in_cols),), hidden=[], rate_guess=np_tr_
util.plot_nn_model(nnp)
```

Out[19]:



As a rate guess we use the average over the training set

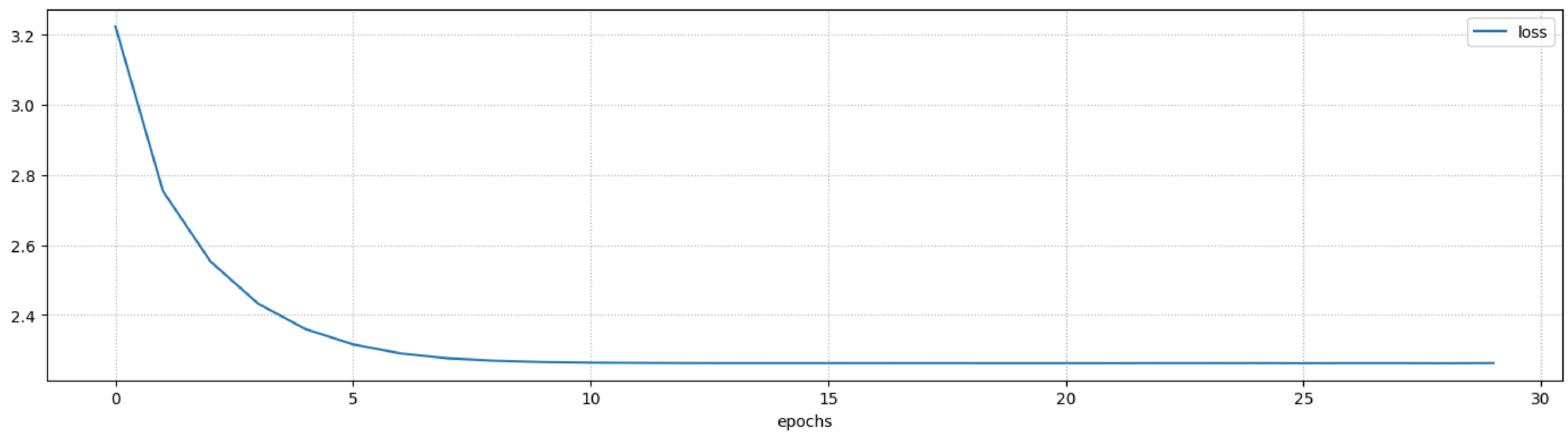
- This is easy to compute
- ...And will provide a better starting point for gradient descent

Training the Model

We can train the model (mostly) as usual

...Except that we need to use the mentioned custom loss function

In [20]:
```
negloglikelihood = lambda y_true, dist: -dist.log_prob(y_true)
nnp = util.build_nn_poisson_model(input_shape=len(in_cols), hidden=[], rate_guess=np_tr_out.
history = util.train_nn_model(nnp, np_tr_in, np_tr_out, loss=negloglikelihood, validation_sp
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 2.2630 (training)
```

Predictions

When we call the `predict` method on the model we obtain samples

This means that the result of `predict` is stochastic

```
In [21]: print(str(nnp.predict(np_tr_in, verbose=0)[:3]).replace('\n', ' '))
         print(str(nnp.predict(np_tr_in, verbose=0)[:3]).replace('\n', ' '))

[[4.]  [3.]  [2.]]
[[6.]  [2.]  [2.]]
```

We can obtain the distribution object by simply calling the model

```
In [22]: nnp(np_tr_in.values)

Out[22]: <tfp.distributions._TensorCoercible 'tensor_coercible' batch_shape=[8760, 1] event_shape=
         [] dtype=float32>
```

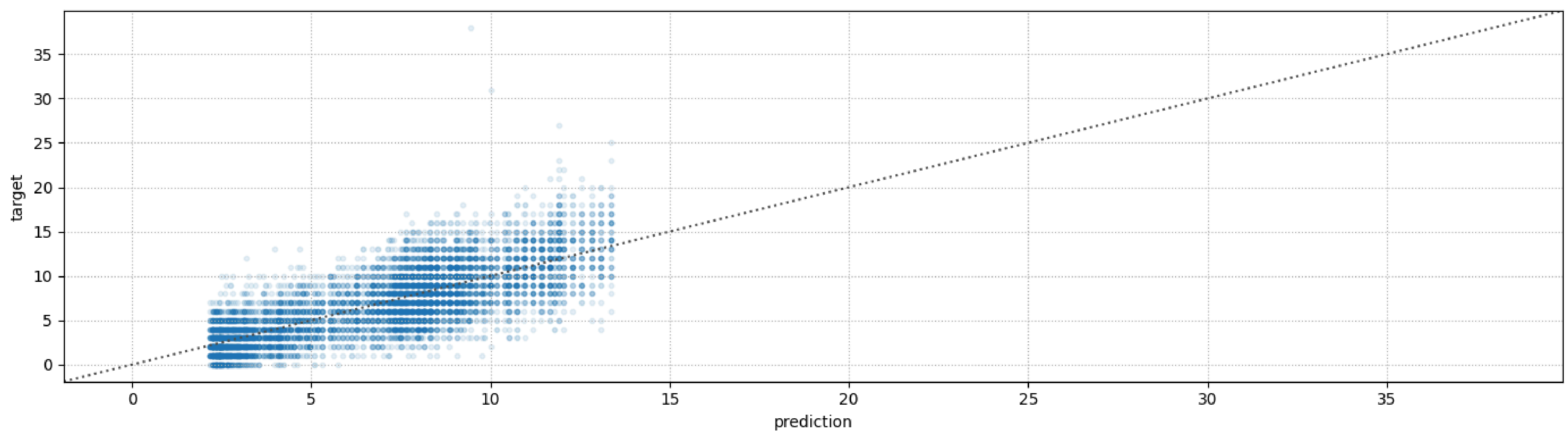
- Then we can call methods over the distribution objects
- ...To obtain means, standard deviations, and any other relevant statistics

Evaluation

Using the predict means, let's check the quality of our results

```
In [23]: tr_pred = nnp(np_tr_in.values).mean().numpy().ravel()
         util.plot_pred_scatter(np_tr_out, tr_pred, figsize=figsize)
```

```
R2: 0.60
MAE: 1.93
```



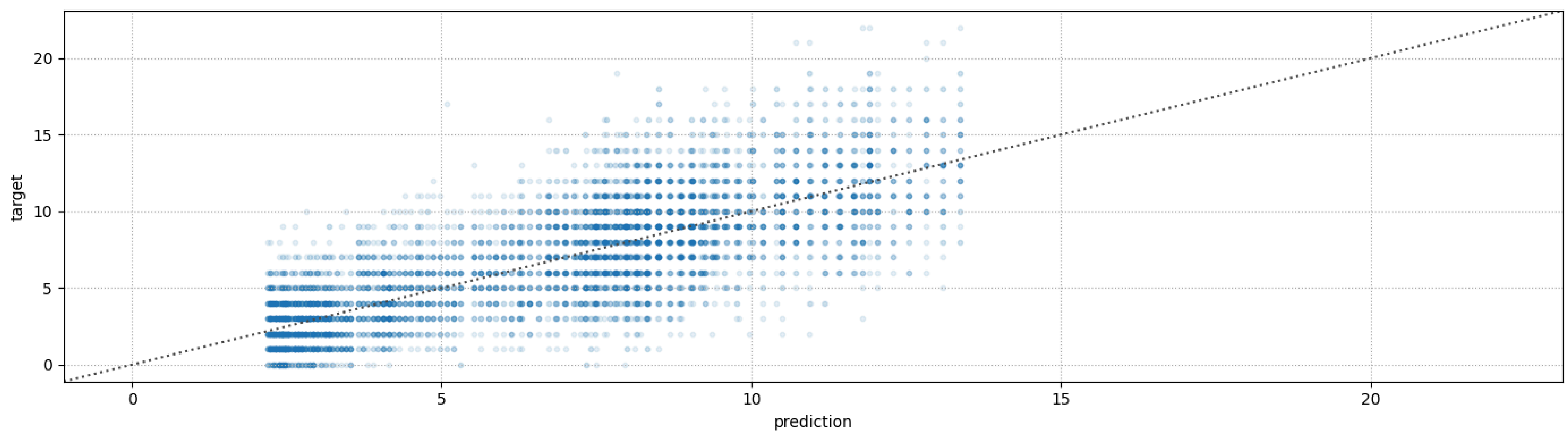
- This is a stochastic process, making this R^2 value very good
- When the stochasticity is too high, using the R^2 might not even be viable

Evaluation

Let's repeat the exercise on the test set

In [24]:
```
ts_pred = nnp(np_ts_in.values).mean().numpy().ravel()
util.plot_pred_scatter(np_ts_out, ts_pred, figsize=figsize)
```

```
R2: 0.60
MAE: 1.94
```



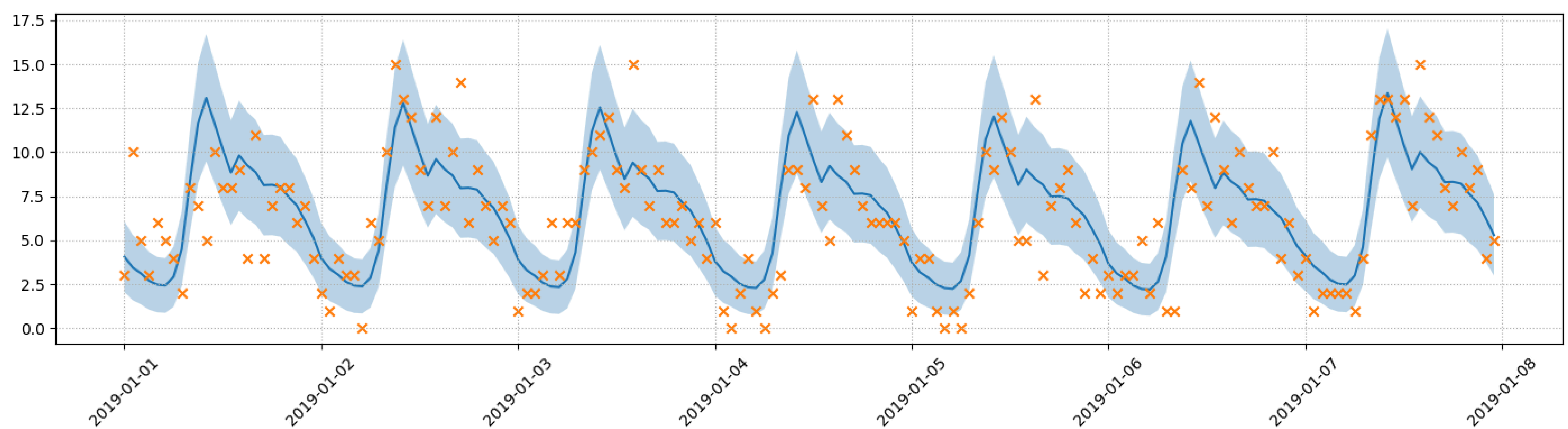
- No overfitting, which is again very good

Confidence Intervals

Since our output is a distribution, we have access to all sort of statistics

Here we will simply show the mean and stdev over one week of data:

In [25]:
```
ts_pred_std = nnp(np_ts_in.values).stddev().numpy().ravel()
util.plot_series(pd.Series(index=np_ts_in.index[:24*7], data=ts_pred[:24*7]), std=pd.Series
plt.scatter(np_ts_in.index[:24*7], np_ts_out[:24*7], marker='x');
```



Some Remarks

This is a very flexible approach

...And it is not restricted to the Poisson distribution

- If you are investigating extreme phenomema
  - Then it is typical to aggregate target values using a maximum
  - ...And you can use a Gumbel or GEV distribution
- If you are interested in inter-arrival times
  - Then you may try and exponential distribution
- Even when you expect a Normal distribution
  - ...You may want your model to estimate a stddev, rather than just a mean
  - There will be an example in the next notebook
- If you are studying survival (e.g. medical applications or equipment)
  - Then you may want to use a Negative Binomial Distribution
  - ...Or you can use the other approach from the next notebook