Consiglio Nazionale delle Ricerche

Polimi - Financial Engineering AY 2023/2024

Lab: Electricity Price and Load Forecasting - L3

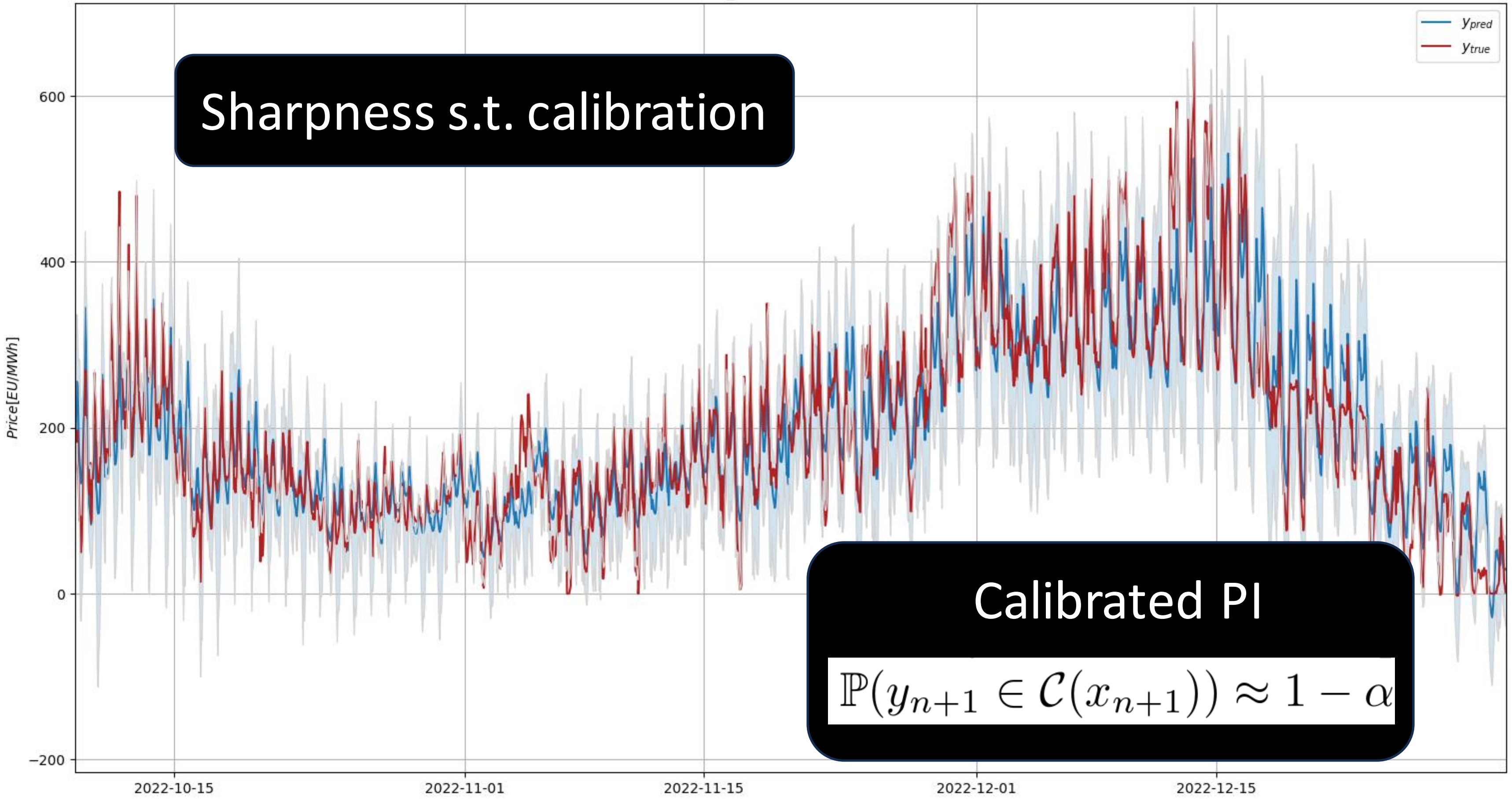
STIIMA
Alessandro Brusaferri

Summary

Goal L3:

- Recap lesson 2
- Intro to probabilistic forecasting
- Intro to quantile regression NNs
- Intro to distributional NNs
- Implementation in Tensorflow probability

Goal of Probabilistic Forecasting (PF)



Sharpness s.t. calibration

Calibrated PI

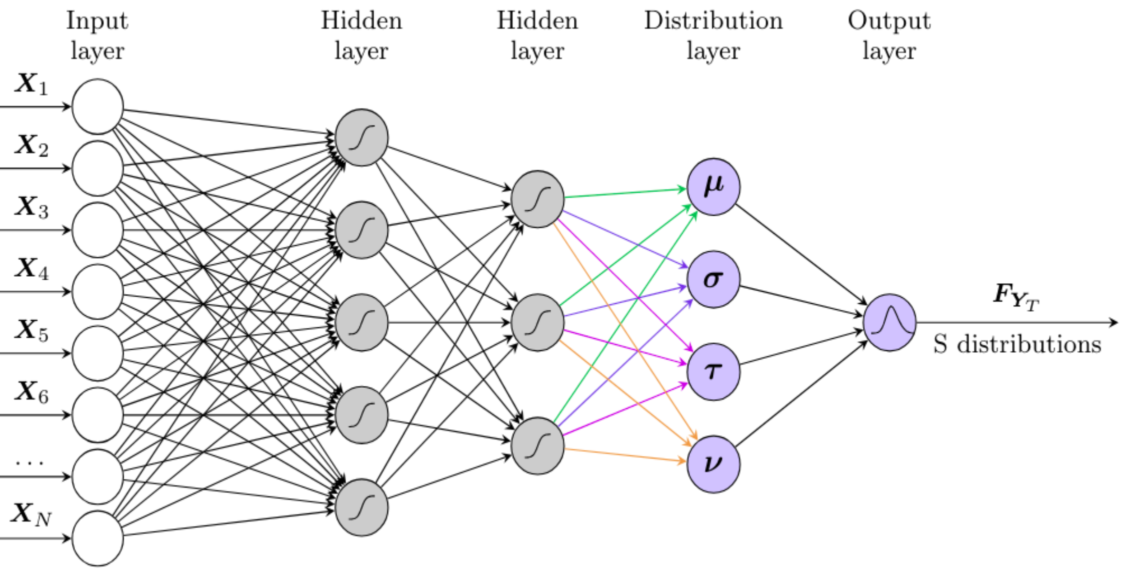
$$\mathbb{P}(y_{n+1} \in \mathcal{C}(x_{n+1})) \approx 1 - \alpha$$

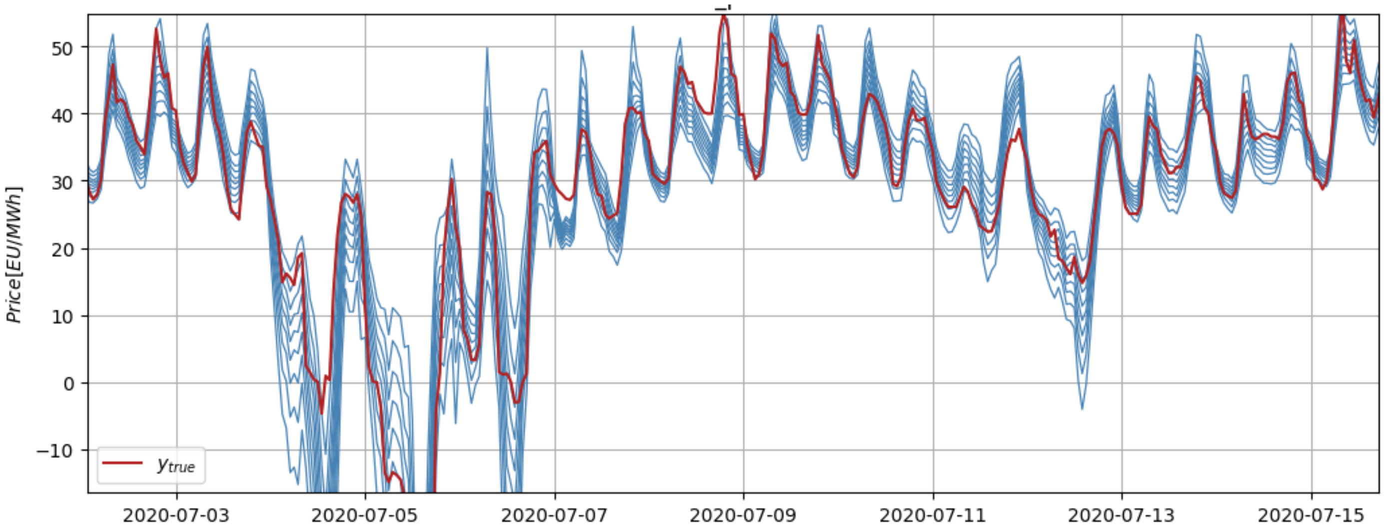
Broad set of approaches:

- Marcjasz, G., Narajewski, M., Weron, R., and Ziel, F., 2023. Distributional neural networks for electricity price forecasting, Energy Economics, 125, 106843.
- Nowotarski, J. and Weron, R., 2018. Recent advances in electricity price forecasting: A review of probabilistic forecasting, Renewable and Sustainable Energy Reviews, 81, 1548–1568.

We focus on two PF methods in this course

- Flexible distribution parameterization by a NN conditioning
- Quantile Regression NNs: approximate multiple quantiles
    - **Multi-step** and **multi-quantile** settings (unique backbone NN)

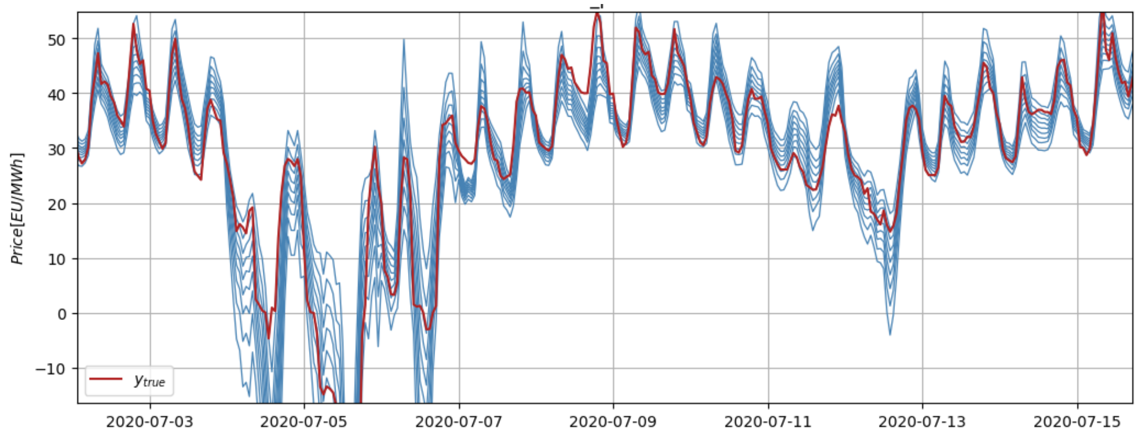


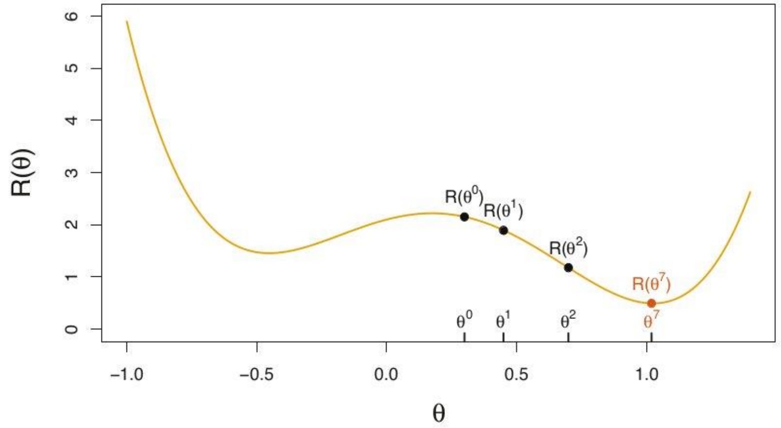


QR-DNN: extend the DNN output layer

DNN parameterization:

$$
\left\{
\begin{aligned}
&\ell_1 = \mathrm{g}(x_i W_1 + b_1)\\
&\ell_2 = \mathrm{g}(\ell_1 W_2 + b_2) W_3 + b_3\\
&W_1 \in \mathbb{R}^{n_x \times n_{u_1}},\quad W_2 \in \mathbb{R}^{n_{u_1} \times n_{u_2}},\\
&W_3 \in \mathbb{R}^{n_{u_2} \times H \cdot n_p},\quad n_{u_1}, n_{u_2} \in \mathbb{Z}^+\\
&b_1 \in \mathbb{R}^{n_{u_1}}, b_2 \in \mathbb{R}^{n_{u_2}}, b_3 \in \mathbb{R}^{\boxed{H \cdot n_p}}
\end{aligned}
\right.
$$





Average Pinball loss function:

$$
\left\{
\begin{aligned}
&n_p = \#\Gamma \ (\text{e.g., } \#\Gamma = 10 \text{ for deciles approximation})\\
&\sum_i \sum_h \sum_\gamma (y_i^h - \hat{q}_\gamma^h(x_i))\gamma 1\{y_i^h > \hat{q}_\gamma^h(x_i)\} + (\hat{q}_\gamma^h(x_i) - y_i^h)(1-\gamma)1\{y_t^h \leq \hat{q}_\gamma^h(x_i)\}
\end{aligned}
\right.
$$

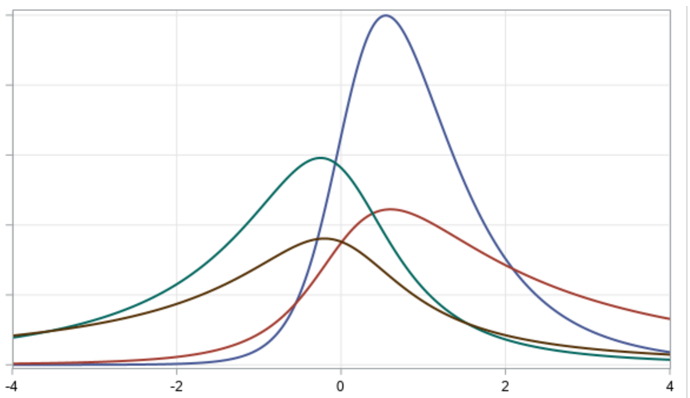
Distributional NNs

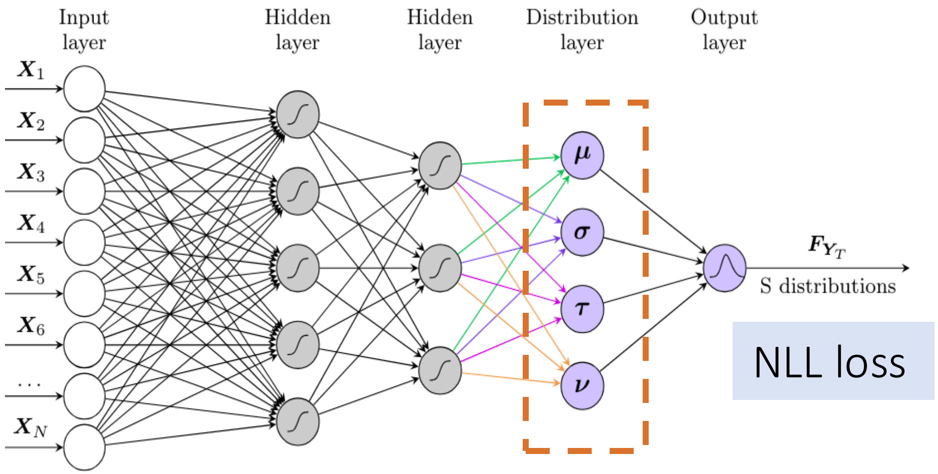
DNN parameterization:

$$
\begin{cases}
\ell_1 = \mathrm{g}(x_i W_1 + b_1) \\
\ell_2 = \mathrm{g}(\ell_1 W_2 + b_2) W_3 + b_3 \\
W_1 \in \mathbb{R}^{n_x \times n_{u_1}}, \; W_2 \in \mathbb{R}^{n_{u_1} \times n_{u_2}}, \\
W_3 \in \mathbb{R}^{n_{u_2} \times H \cdot n_p}, \; n_{u_1}, n_{u_2} \in \mathbb{Z}^+ \\
b_1 \in \mathbb{R}^{n_{u_1}}, b_2 \in \mathbb{R}^{n_{u_2}}, b_3 \in \mathbb{R}^{\boxed{H \cdot n_p}}
\end{cases}
$$

Output distributional layer:

$$
\begin{cases}
\lambda_i^h = \ell_2^{[h]} \\
\sigma_i^h = \epsilon + \gamma \text{ Softplus}\left(\ell_2^{[H+h]}\right) \\
\tau_i^h = 1 + \gamma \text{ Softplus}\left(\ell_2^{[2 \cdot H+h]}\right) \\
\zeta_i^h = \ell_2^{[3 \cdot H+h]} \\
\text{Softplus}(x) = \log\left(1 + e^x\right)
\end{cases}
$$





Parameterized density (e.g., Johnson's SU):

$$
f^h(\chi) = \frac{\tau_i^h}{\sigma_i^h \sqrt{2\pi}} \frac{1}{\sqrt{1 + \left(\frac{\chi - \lambda_i^h}{\sigma_i^h}\right)^2}} e^{-\frac{1}{2}\left[\zeta_i^h + \tau_i^h \sinh^{-1}\left(\frac{\chi - \lambda_i^h}{\sigma_i^h}\right)\right]^2}
$$

$\lambda_i^h, \sigma_i^h, \tau_i^h, \zeta_i^h$: conditional JSU loc, scale, tailweight and skewness

γ : 3, ϵ : 1e-3, correction factors (computation purpose)

Marcjasz, G., Narajewski, M., Weron, R., and Ziel, F., 2023. Distributional neural networks for electricity price forecasting, Energy Economics, 125, 106843.

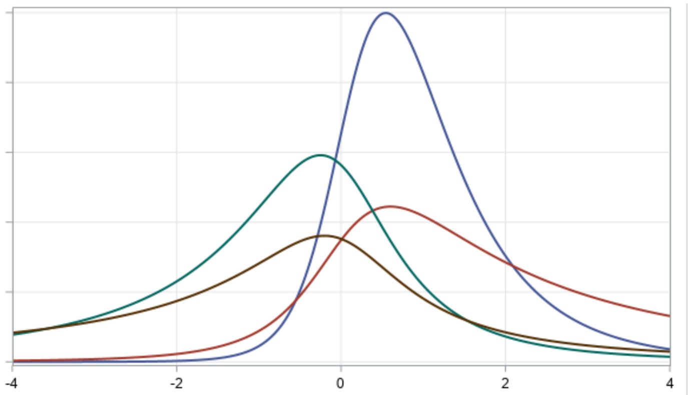
Distributional NNs

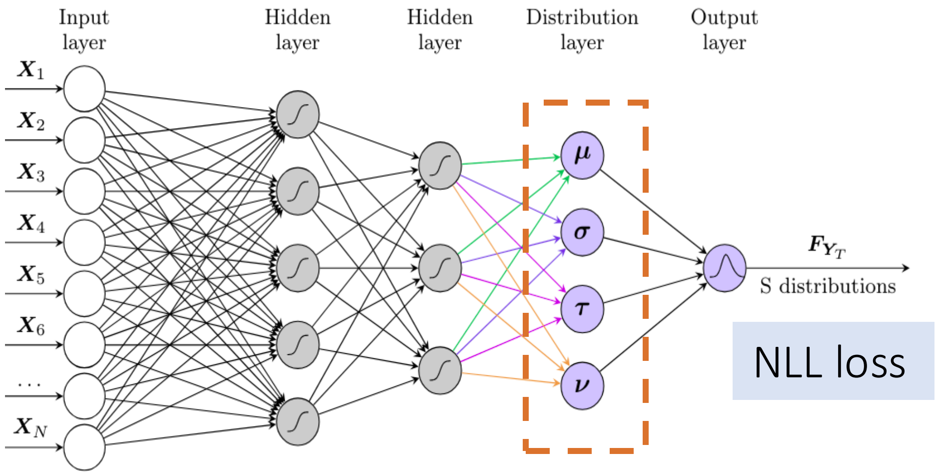
DNN parameterization:

$$\begin{cases} \ell_1 = g(x_i W_1 + b_1) \\ \ell_2 = g(\ell_1 W_2 + b_2) W_3 + b_3 \\ W_1 \in \mathbb{R}^{n_x \times n_{u_1}}, \quad W_2 \in \mathbb{R}^{n_{u_1} \times n_{u_2}}, \\ W_3 \in \mathbb{R}^{n_{u_2} \times H \cdot n_p}, \quad n_{u_1}, n_{u_2} \in \mathbb{Z}^+ \\ b_1 \in \mathbb{R}^{n_{u_1}}, b_2 \in \mathbb{R}^{n_{u_2}}, b_3 \in \mathbb{R}^{\boxed{H \cdot n_p}} \end{cases}$$

Output distributional layer:

$$\begin{cases} \lambda_i^h = \ell_2^{[h]} \\ \sigma_i^h = \epsilon + \gamma \text{ Softplus}\left(\ell_2^{[H+h]}\right) \\ \tau_i^h = 1 + \gamma \text{ Softplus}\left(\ell_2^{[2 \cdot H+h]}\right) \\ \zeta_i^h = \ell_2^{[3 \cdot H+h]} \\ \text{Softplus}(x) = \log\left(1 + e^x\right) \end{cases}$$





Parameterized density (e.g., Johnson's SU):

```
tfd.JohnsonSU(skewness=[1, 4], tailweight=[2, 5],
              loc=[3, 6], scale=[11, 22.])
```

$\lambda_i^h, \sigma_i^h, \tau_i^h, \zeta_i^h$: conditional JSU loc, scale, tailweight and skewness

γ : 3, ϵ : 1e-3, correction factors (computation purpose)

Marcjasz, G., Narajewski, M., Weron, R., and Ziel, F., 2023. Distributional neural networks for electricity price forecasting, Energy Economics, 125, 106843.

Extend DNN output layer into PF form

```python
class DNNRegressor:
    def __init__(self, settings, loss):
        self.settings = settings
        self.__build_model__(loss)

    def __build_model__(self, loss):
        x_in = tf.keras.layers.Input(shape=(self.settings['input_size'],))
        x_in = tf.keras.layers.BatchNormalization()(x_in)
        x = (tf.keras.layers.Dense(self.settings['hidden_size'],
                                   activation=self.settings['activation'],
                                   )(x_in))
        for hl in range(self.settings['n_hidden_layers'] - 1):
            x = tf.keras.layers.Dense(self.settings['hidden_size'],
                                      activation=self.settings['activation'],
                                      )(x)
        if self.settings['PF_method'] == 'point':
            out_size = 1
            logit = tf.keras.layers.Dense(self.settings['pred_horiz'] * out_size,
                                          activation='linear',
                                          )(x)
            output = tf.keras.layers.Reshape((self.settings['pred_horiz'], 1))(logit)
```

Simply changing the output layer leads to a probabilistic forecaster from the point-DNN !!

QR-DNN

```python
elif self.settings['PF_method'] == 'qr':
    out_size = len(self.settings['target_quantiles'])
    logit = tf.keras.layers.Dense(self.settings['pred_horiz'] * out_size,
                                  activation='linear'
                                  )(x)
    output = tf.keras.layers.Reshape((self.settings['pred_horiz'], out_size))(logit)
    # Fix quantile crossing by sorting
    output = tf.keras.layers.Lambda(lambda x: tf.sort(x, axis=-1))(output)
```

Normal Distribution DNN

```python
elif self.settings['PF_method'] == 'Normal':
    out_size = 2
    logit = tf.keras.layers.Dense(self.settings['pred_horiz'] * out_size,
                                  activation='linear',
                                  )(x)
    output = tfp.layers.DistributionLambda(
        lambda t: tfd.Normal(
            loc=t[..., :self.settings['pred_horiz']],
            scale=1e-3 + 3 * tf.math.softplus(0.05 * t[..., self.settings['pred_horiz']:])))(logit)
```

Map the related PF loss functions

STIIMA

```python
class PinballLoss(keras.losses.Loss):
    def __init__(self, quantiles: List, name="pinball_loss"):
        super().__init__(name=name)
        self.quantiles = quantiles

    def call(self, y_true, y_pred):
        loss = []
        for i, q in enumerate(self.quantiles):
            error = tf.subtract(y_true, y_pred[:, :, i])
            loss_q = tf.reduce_mean(tf.maximum(q * error, (q - 1) * error))
            loss.append(loss_q)
        L = tf.convert_to_tensor(loss)
        total_loss = tf.reduce_mean(L)
        return total_loss

    def get_config(self):
        return {
            "num_quantiles": self.quantiles,
            "name": self.name,
        }
```

Average Pinball loss implementation in Tensorflow

$$\sum_i \sum_h \sum_\gamma (y_i^h - \hat{q}_\gamma^h(x_i))\gamma 1\{y_i^h > \hat{q}_\gamma^h(x_i)\} + (\hat{q}_\gamma^h(x_i) - y_i^h)(1-\gamma)1\{y_t^h \leq \hat{q}_\gamma^h(x_i)\}$$

```python
class TensorflowRegressor():
    """
    Implementation of the Tenforflow regressor
    """
    def __init__(self, settings, sample_x):
        self.settings = settings
        self.x_columns_names = settings['x_columns_names']
        self.pred_horiz = settings['pred_horiz']

        tf.keras.backend.clear_session()
        # Map the loss to be used
        if settings['PF_method'] == 'qr':
            loss = qt.PinballLoss(quantiles=settings['target_quantiles'])
        elif settings['PF_method']=='point':
            loss = 'mae'
        elif (settings['PF_method'] == 'Normal'
              or settings['PF_method'] == 'JSU'
        ):
            loss = lambda y, rv_y: -rv_y.log_prob(y)
        else:
            sys.exit('ERROR: unknown PF_method config!')
```

- MAE for point
- NLL for distributional
- Pinball for QR

Define the output handler to store predictions

```python
class TensorflowRegressor():
    """
    Implementation of the Tenforflow regressor
    """
    def __init__(self, settings, sample_x):
        self.settings = settings
        self.x_columns_names = settings['x_columns_names']
        self.pred_horiz = settings['pred_horiz']

        tf.keras.backend.clear_session()
        # Map the loss to be used
        if settings['PF_method'] == 'qr':
            loss = PinballLoss(quantiles=settings['target_quantiles'])
        elif settings['PF_method']=='point':
            loss = 'mae'
        elif (settings['PF_method'] == 'Normal'
        ):
            loss = lambda y, rv_y: -rv_y.log_prob(y)
        else:
            sys.exit('ERROR: unknown PF_method config!')

        # Instantiate the model
        if settings['model_class']=='DNN':
            # get input size for the chosen model architecture
            settings['input_size']=DNNRegressor.build_model_input_from_series(x=sample_x,
                                                                              col_names=self.x_columns_names,
                                                                              pred_horiz=self.pred_horiz).shape[1]
            # Build the model architecture
            self.regressor = DNNRegressor(settings, loss)

        elif  settings['model_class']=='ARX':
            # get input size for the chosen model architecture
            settings['input_size']=ARXRegressor.build_model_input_from_series(x=sample_x,
                                                                              col_names=self.x_columns_names,
                                                                              pred_horiz=self.pred_horiz).shape[1]
            # Build the model architecture
            self.regressor = ARXRegressor(settings, loss)
        else:
            sys.exit('ERROR: unknown model_class')

        # Map handler to convert distributional output to quantiles or distribution parameters
        if (settings['PF_method'] == 'Normal'):
            self.output_handler = self.__pred_Normal_params__
        elif settings['PF_method'] == 'JSU':
            self.output_handler = self.__pred_JSU_params__
        else:
            self.output_handler =self.__quantiles_out__

    1 usage (1 dynamic)
    def predict(self, x):
        return self.output_handler(self.regressor.predict(x))
```


```python
# Map handler to convert distributional output to quantiles or distribution
if (settings['PF_method'] == 'Normal'):
    self.output_handler = self.__pred_Normal_params__
elif settings['PF_method'] == 'JSU':
    self.output_handler = self.__pred_JSU_params__
else:
    self.output_handler =self.__quantiles_out__
```


```python
def __quantiles_out__(self, preds):
    # Expand dimension to enable concat in ensemble
    return tf.expand_dims(preds, axis=2)

def __pred_Normal_params__(self, pred_dists: tfp.distributions):
    loc = tf.expand_dims(pred_dists.loc, axis=-1)
    scale = tf.expand_dims(pred_dists.scale, axis=-1)
    # Expand dimension to enable concat in ensemble
    return tf.expand_dims(tf.concat([loc,scale], axis=-1), axis=2)
```

Store step-wise (e.g., day-ahead hour) predictions as:

- Predicted distribution parameters for Distributional NNs
- Predicted quantiles for the QR setup

From DNNs to predicted quantiles aggregation

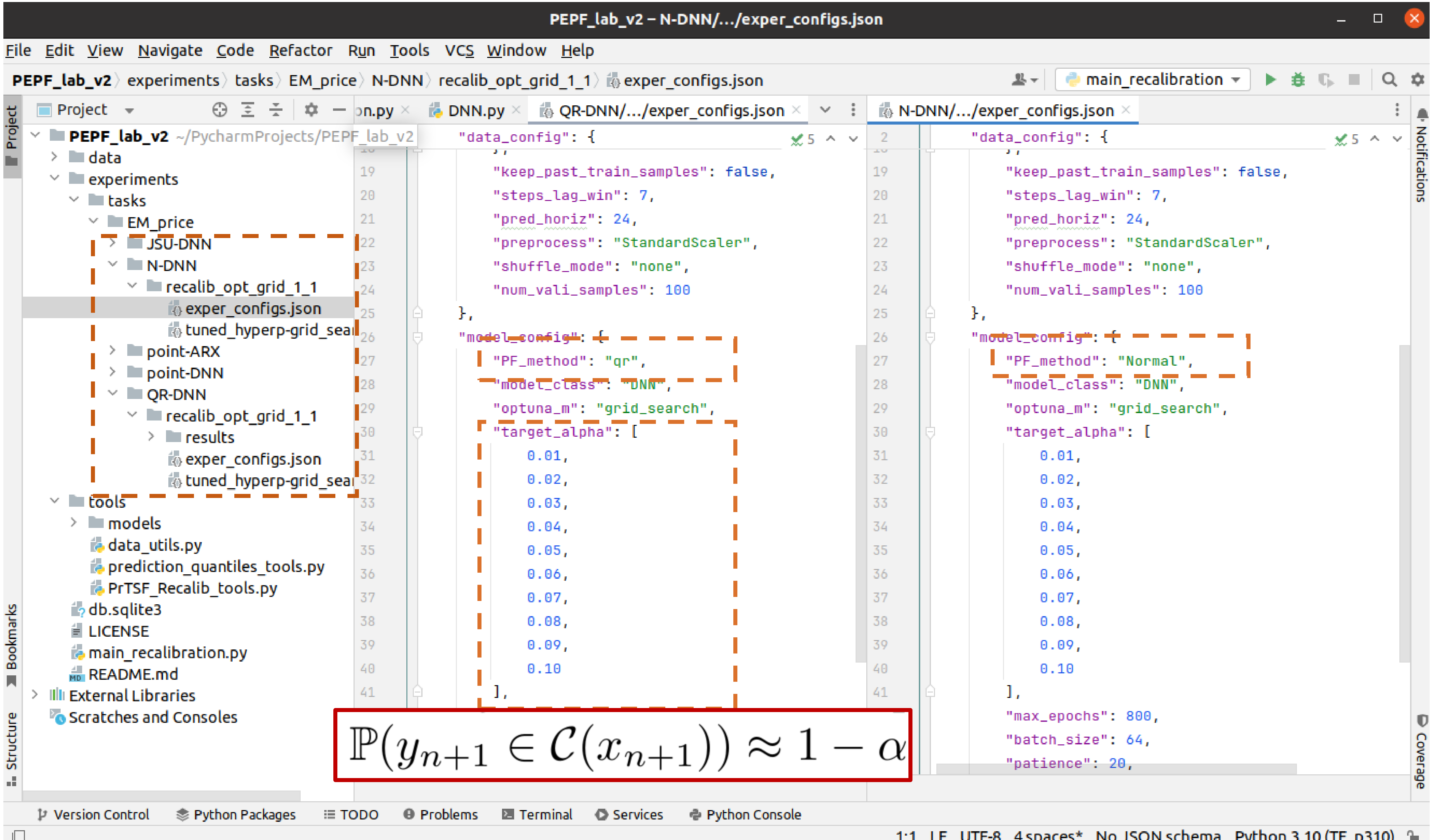
```python
class Ensemble():
    """
    Tensorflow ensemble wrapper
    """
    def __init__(self, settings):
        # store configs for internal use
        self.settings = settings
        # map the methods to use for aggretation and quantile building depending on the configs
        if (self.settings['PF_method'] == 'point'):
            self.ensemble_aggregator = self.__aggregate_de_quantiles__
            self._build_test_PIs = self.__get_qr_PIs__
        elif (self.settings['PF_method'] == 'qr'):
            self.ensemble_aggregator = self.__aggregate_de_quantiles__
            self._build_test_PIs = self.__get_qr_PIs__
        elif (self.settings['PF_method'] == 'Normal'):
            self.ensemble_aggregator = self.__aggregate_de__
            self._build_test_PIs = self.__build_Normal_PIs__
        elif (self.settings['PF_method'] == 'JSU'):
            self.ensemble_aggregator = self.__aggregate_de__
            self._build_test_PIs = self.__build_JSU_PIs__
        else:
            sys.exit('ERROR: Ensemble config not supported!')
```

```python
@staticmethod
def __get_qr_PIs__(preds_test, settings):
    # simply flatten in temporal dimension
    return preds_test.reshape(-1, preds_test.shape[-1])

@staticmethod
def __build_Normal_PIs__(preds_test, settings):
    # for each de component, sample, aggregate samples and compute quantiles
    pred_samples = []
    for k in range(preds_test.shape[2]):
        pred_samples.append(tfd.Normal(
            loc=preds_test[:,:,k,0],
            scale=preds_test[:,:,k,1]).sample(10000).numpy())
    return np.transpose(np.quantile(np.concatenate(pred_samples, axis=0),
                                    q=settings['target_quantiles'], axis=0),
                        axes=(1, 2, 0)).reshape(-1, len(settings['target_quantiles']))

@staticmethod
def __aggregate_de__(ens_comp_preds):
    # aggregate by concatenation, for point a distributional settings
    return np.concatenate(ens_comp_preds, axis=2)

@staticmethod
def __aggregate_de_quantiles__(ens_comp_preds):
    # aggregate by a uniform vincentization
    return np.mean(np.concatenate(ens_comp_preds, axis=2), axis=2)
```

- From point EPF model ensemble to PEPF ensemble (e.g., in case of multiple DNNs, num_ense > 1)
- Simple aggregation by equally weighted (i.e., uniform) quantile averaging
- From distributional NNs samples to prediction quantiles

Add PF method folder, set json and target alpha

STIIMA

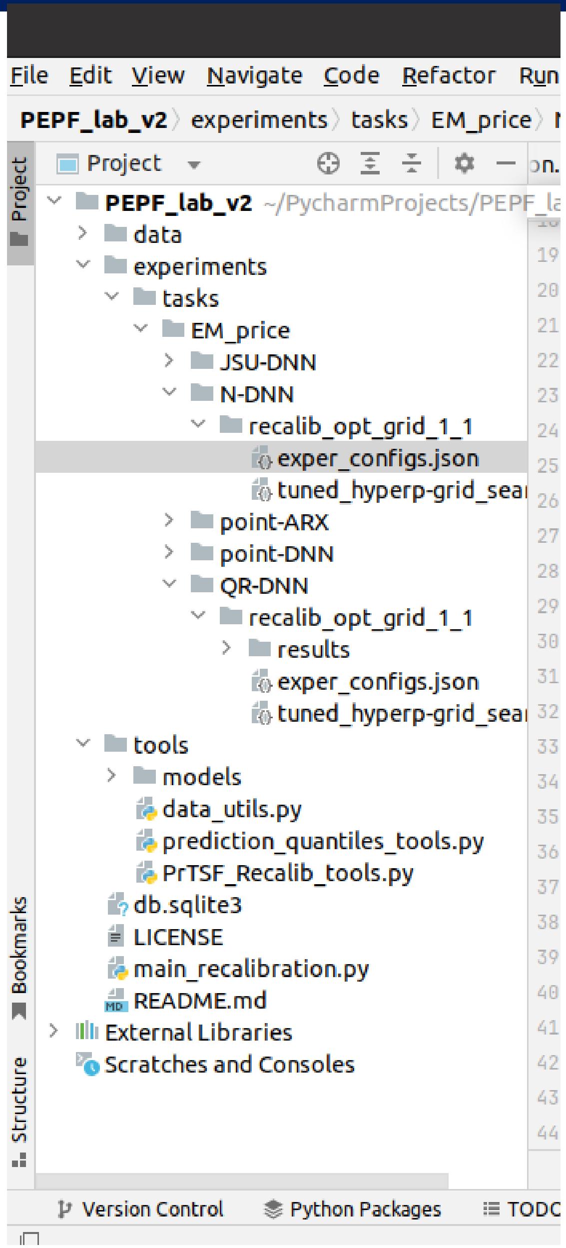


$$\mathbb{P}(y_{n+1} \in \mathcal{C}(x_{n+1})) \approx 1 - \alpha$$

Set experiment and run

STIIMA

```python
#------------------------------------------------------------------------
# Set PEPF task to execute
PF_task_name = 'EM_price'
# Set Model setup to execute
exper_setup = 'QR-DNN'

#------------------------------------------------------------------------
# Select run
run_id = 'recalib_opt_grid_1_1'
# Load hyperparams from file (select: load_tuned or optuna_tuner)
hyper_mode = 'optuna_tuner'
# Plot train history flag
```



- Model form dedicated folder (QR-DNN, etc)
  - Run_id folder (different experiments)
    - json configs
    - recalib results folder

Raw dataframe (e.g., pre-processing, model chain, etc.)

STIIMA

main_recalibration.py × | DNN.py × | prediction_quantiles_tools.py × | models_tools.py × | PrTSF_Recalib_tools.py ×

```python
import os
import pandas as pd

os.environ["TF_USE_LEGACY_KERAS"]="1"

from tools.PrTSF_Recalib_tools import PrTsfRecalibEngine, load_data_model_configs
from tools.prediction_quantiles_tools import plot_quantiles

#-------------------------------------------------------------------------------------------------------
# Set PEPF task to execute
PF_task_name = 'EM_price'
# Set Model setup to execute
exper_setup = 'QR-DNN'

#-------------------------------------------------------------------------------------------------------
# Select run
run_id = 'recalib_opt_grid_1_1'
# Load hyperparams from file (select: load_tuned or optuna_tuner)
hyper_mode = 'load_tuned'
# Plot train history flag
plot_train_history=False
plot_weights=False
#-------------------------------------------------------------------------------------------------------

# Load experiments configuration from json file
configs=load_data_model_configs(task_name=PF_task_name, exper_setup=exper_setup, run_id=run_id)

# Load dataset
dir_path = os.getcwd()
ds = pd.read_csv(os.path.join(dir_path, 'data', 'datasets', configs['data_config'].dataset_name))
ds.set_index(ds.columns[0], inplace=True)

# Instantiate recalibratione engine
PrTSF_eng = PrTsfRecalibEngine(dataset=ds,
                               data_configs=configs['data_config'],
                               model_configs=configs['model_config'])
```

ds

	ep	IDX__sub_step	TARG__EM_price	FUTU__EM_load_f	FU
0		0	52.10000	5947.00000	0.
1		1	47.84000	5406.00000	0.
2		2	43.66000	4937.00000	0.
3		3	43.66000	4592.00000	0.
4		4	41.98000	4391.00000	0.
5		5	43.93000	4282.00000	0.
6		6	46.02000	4260.00000	0.
7		7	50.92000	4326.00000	0.
8		8	55.57000	4426.00000	0.
9		9	58.10000	4602.00000	29
10		10	59.80000	5061.00000	11
11		11	59.51000	5550.00000	18
12		12	59.00000	5924.00000	22
13		13	58.13000	6091.00000	23
14		14	56.90000	5988.00000	23
15		15	52.69000	5878.00000	19
16		16	52.10000	5843.00000	13

ds Format: %s

☑ Colored cells

☑ Resize automatically

Close

Assignment PEF3 (deadline 13-5)

- Implement Johnson's SU distributional DNN
- Compare QR-DNN, Normal-DNN, JSU-DNN (Pinball score)
    - Random hyperparameter search on 'hidden size' and 'learning rate'
    - Test recalibration (1 time) on May 2017

Facultative:

- Implement Winkler's score (see Readings prelab 4.2.1) and compute on results above
- Preprocess the data by the arcsinh(.) transformation (see Readings prelab 3 and references therein) and perform the QR-DNN experiments above on the transformed data (results on the inverse transformation)

Consiglio Nazionale delle Ricerche

Thanks

Alessandro Brusaferri