GMDH Neural Network for Short-Term Electricity Load Forecasting

KOSTAS PASSADIS ~ EUROSCIPY 2018

About me

- Senior Electrical & Computing Engineer at the Greek Transmission System Operator (IPTO)
- PhD in Electromagnetics
- Senior Technical Development Editor @manning (by night)

Presentation Outline

- Group Method of Data Handling – Idea and model overview
- Implementation in Python and alternative ways to fit the model
- Advantages and Disadvantages
- Case Study: Short-term load forecast
- Time series forecasting principles and challenges
- Data preparation and simulation results
- Possible improvements

GMDH Neural Networks - Principle

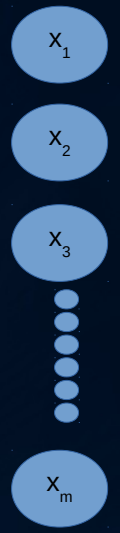
- Group Method of Data Handling
- Early work developed in the seventies (Ivakhnenko 1966)
- Based on the principle that every function f(x) can approximated by an infinite Volterra-Kolmogorov-Gabor polynomial:

$$y = a + \sum_{i=1}^{m} b_i * x_i + \sum_{i=1}^{m} \sum_{j=1}^{m} c_{ij} * x_i * x_j + \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{k=1}^{m} d_{ijk} * x_i * x_j * x_k + ...$$

- GMDH algorithms consider various component subsets of the base function called partial models
- It turns out that the VKG can be represented by constructing a cascade of second order polynomials (Madala & Ivakhnenko, 1994)

GMDH - Overview

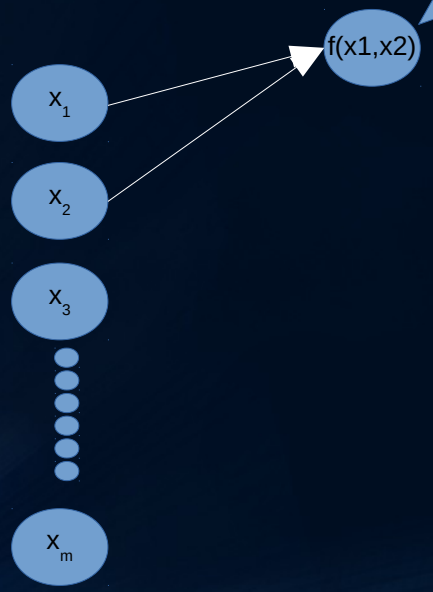
- Multilayered structure not determined before hand but is automatically determined during the training procedure (self-organizing).
- The network structure consists of layers of partial descriptors (quadratic polynomials). Every node performs its own transfer function and passes its output to neurons in the next layer.



GMDH - Overview

- Multilayered structure not determined before hand but is automatically determined during the training procedure (self-organizing).
- The network structure consists of layers of partial descriptors (quadratic polynomials). Every node performs its own transfer function and passes its output to neurons in the next layer.

$$f(x_1, x_2) = w_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_1 * x_2 + w_4 * x_1^2 + w_5 * x_2^2$$

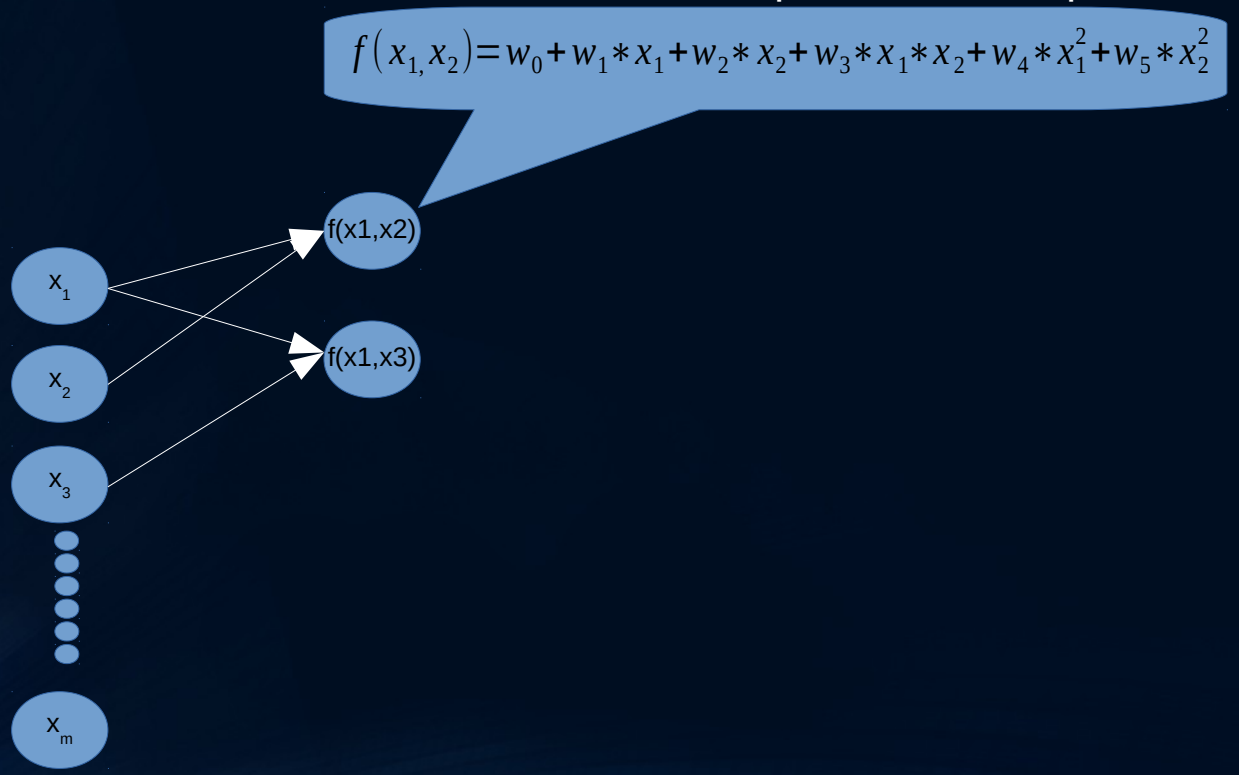


Each node has its own:

- Weights
- Score

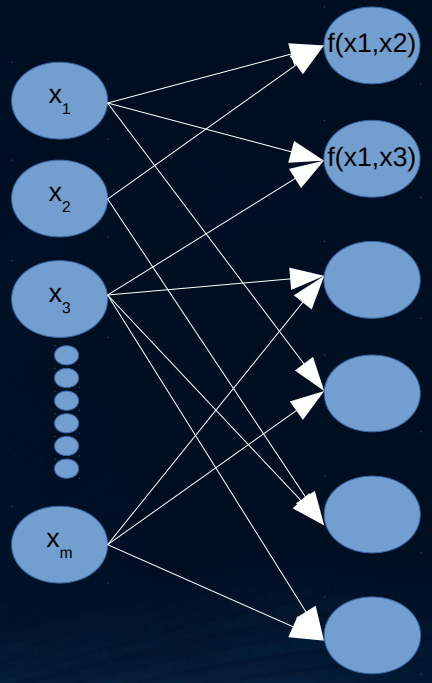
GMDH - Overview

- Multilayered structure not determined before hand but is automatically determined during the training procedure (self-organizing).
- The network structure consists of layers of partial descriptors (quadratic polynomials). Every node performs its own transfer function and passes its output to neurons in the next layer.



GMDH - Overview

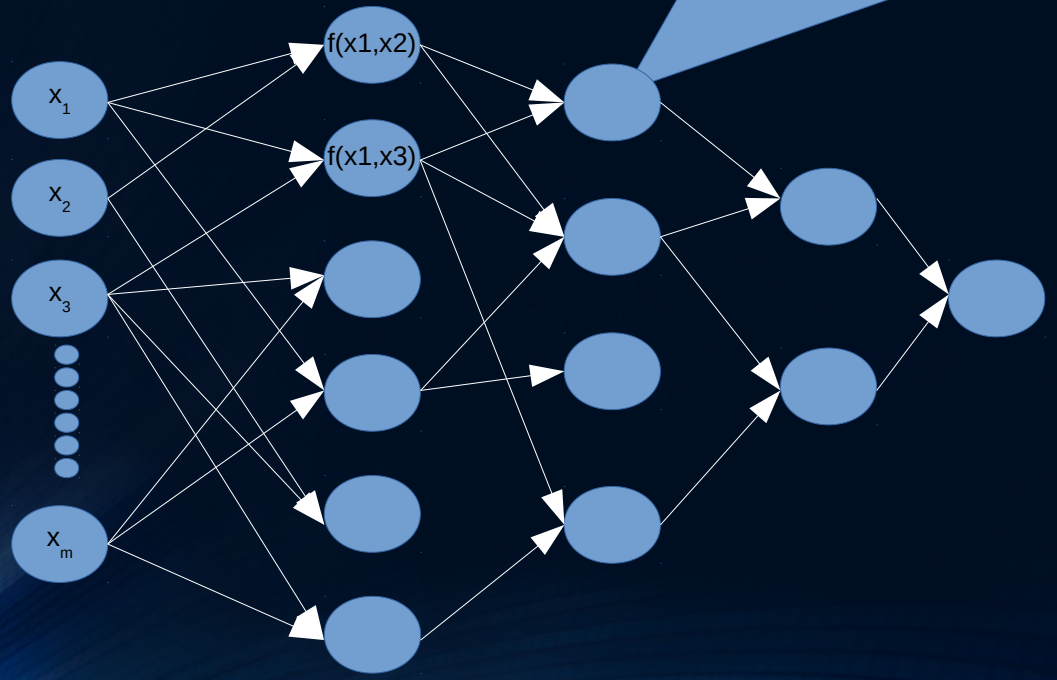
- Multilayered structure not determined before hand but is automatically determined during the training procedure (self-organizing).
- The network structure consists of layers of partial descriptors (quadratic polynomials). Every node performs its own transfer function and passes its output to neurons in the next layer.



GMDH - Overview

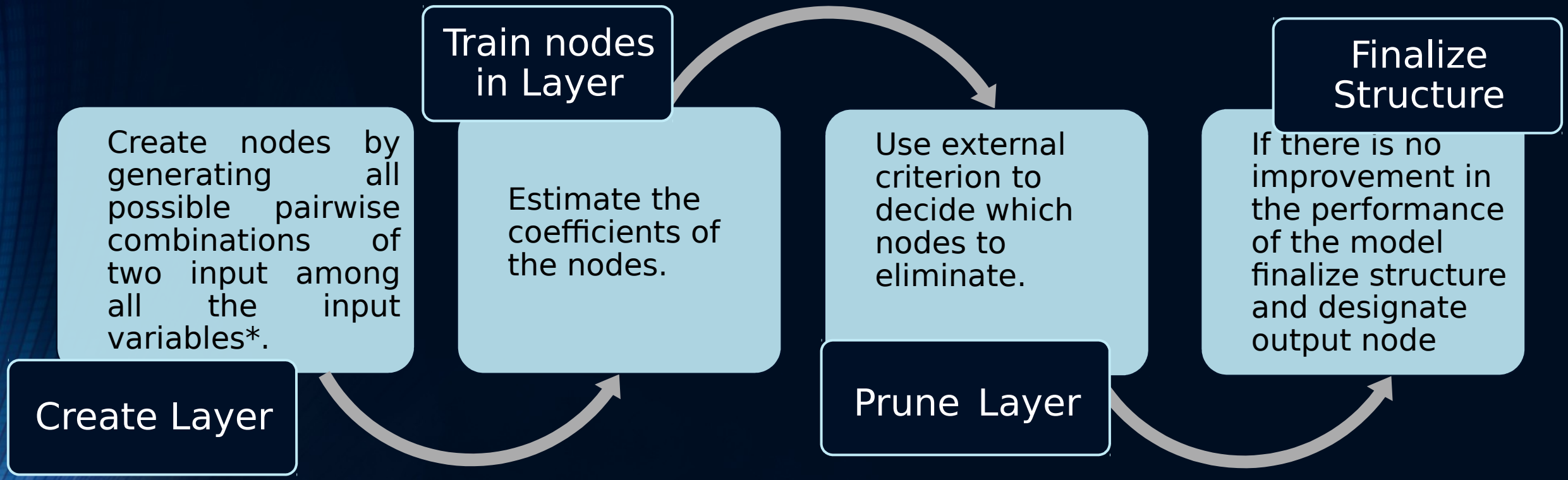
- Multilayered structure not determined before hand but is automatically determined during the training procedure (self-organizing).
- The network structure consists of layers of partial descriptors (quadratic polynomials). Every node performs its own transfer function and passes its output to neurons in the next layer.

$$f(x_i, x_j) = w_0 + w_1 * x_i + w_2 * x_j + w_3 * x_i * x_j + w_4 * x_i^2 + w_5 * x_j^2$$




```
class GMDH:

    def __init__(self, solver='ridge'):
        self.nodes = []
        self.best_score = 0.0
        self.predict_node = None
        self.solver = solver
```

GMDH – Training Procedure



* n = m*(m-1)/2

GMDH – Building Block

```python
@total_ordering
class PolynomialNode(Node):

    def __init__(self, node_id, inputNodes, outputNode):
        """The polynomial node is the building block of the neural network. The node maintains a list of
        input nodes and the target values (output node). The polynomial unit contains six parameters (weights),
        the optimal values of which will be determined during the training process."""
        self.inputNodes = inputNodes
        self.outputNode = outputNode
        self.predictions = []
        self.w = np.zeros(6)
        self.score = np.Inf
        super().__init__(node_id)

    def _transfer(self, X):
        """The transform function takes a design matrix X as input. The node retries the two inputs it depends
           on by asking its input nodes to predict. Then, it expands the feature space by constructing
           second order terms of the inputs"""

        X1 = self.inputNodes[0].predict(X)
        X2 = self.inputNodes[1].predict(X)
        X_t = np.hstack((X1, X2))
        X_t = np.array([[1, x[0], x[1], x[0]*x[1], x[0] * x[0], x[1] * x[1]] for x in X_t])
        return X_t
```

The transfer function takes two inputs and constructs a quadratic feature matrix. The output of the neuron is still linear in terms of The weight coefficients and is equal to the dot product of the weights and the new feature matrix.

GMDH – Fitting a Node (1)

Ridge Regression (Closed form solution)

- Minimize $J(w)=(y-Xw)^T(y-Xw)+\alpha w^T w$, where α is a positive scalar.
- Advantages (over Maximum Likelihood): Numerically stable, better generalization properties.
- Solution $\hat{w}=(X^T X+\alpha I)^{-1} X^T y$
- We have to make sure line fits the points but we also want small parameter values.
- Extremes: If α is very high we get zero values for w. If α is zero we get the Maximum Likelihood solution.
- How dow we choose a value for α?

GMDH – Scikit-learn to the rescue

```
def fit(self, X, y, param_grid = {'alpha':[0.1, 0.5]}):
    X = self._transfer(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    estimator = Ridge(solver='cholesky', fit_intercept=False)
    grid_search = GridSearchCV(estimator, param_grid=param_grid, cv=cv)
    grid_search.fit(X_train, y_train)
    self.estimator = grid_search.best_estimator_
    self.score = grid_search.score(X_test, y_test)
```

GridSearchCV is called a meta-estimator. It behaves like any other estimator in that we can call fit on it. If we call fit on grid search not only is the best hyper-parameter found using cross-validation but a new model built on the whole training set using this best hyper-parameter is fit. We we call predict() or score() on the grid search object the best parameter values are used.

GMDH – Fitting a Node (2)

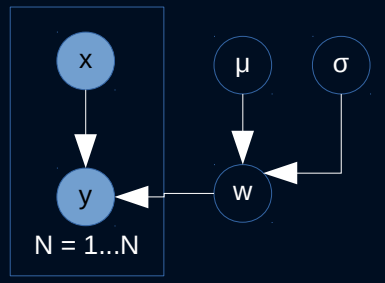
Bayesian Linear Regression

- In the Bayesian setting we treat the regression problem using probability distributions rather than point estimates. Bayesian thinking is very intuitive: we have some prior beliefs and then as we collect more and more data these prior beliefs either get reinforced or disproved.

- We can build some intuition using the simple linear regression model. The simple linear regression can be written in probabilistic terms as follows:

$$\mu_i = \alpha + \beta * x_i$$
$$y_i \sim N(\mu_i, \sigma)$$

$$posterior \propto likelihood\ x\ prior$$
$$f(\alpha, \beta, \sigma | y, x) \propto \prod_{i=1}^{N} N(y_i | \alpha + \beta x_i, \sigma) f_\alpha(\alpha) f_\beta(\beta) f_\sigma(\sigma)$$



- The dependent variable y follows a normal distribution. In the Bayesian approach instead of maximizing the likelihood we assume prior distributions of the parameters and then using Bayes theorem we can obtain the posterior distribution of the parameters. The result of performing Bayesian Linear Regression is a distribution of possible model parameters based on the data and the prior.

- Get a whole range of inferential solutions rather than a single point estimate for each parameter

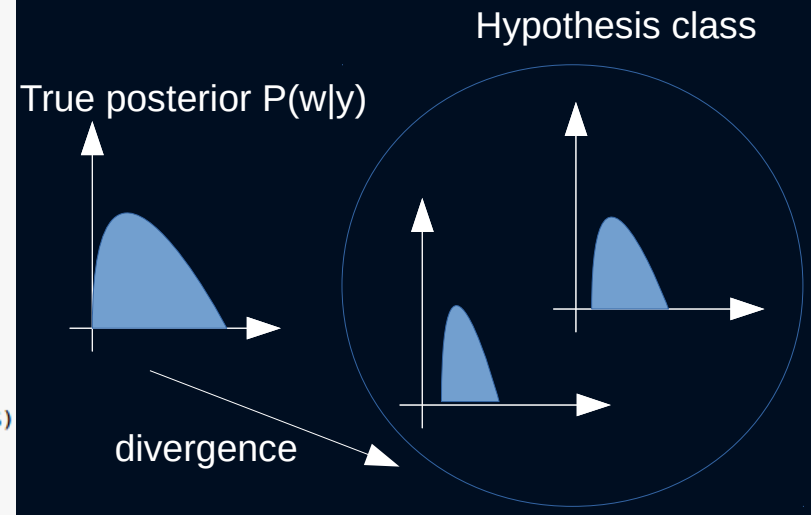
- How do we estimate the posterior?

GMDH – Variational Inference to the rescue

```
def fit(self, X, y, l=0.5, a_0=1e-3, b_0=1e-3, max_iter=500, epsilon_conv=1e-5, cv = 5, batch
    X_t = self._transfer(X)
    D = X_t.shape[1]
    I = X_t.shape[0]
    XX = np.dot(X_t.T, X_t)
    Xy = np.dot(X_t.T, y)
    yy = np.dot(y.T, y)
    L = np.repeat(-np.inf, max_iter)
    a = a_0 + D / 2
    E_a = a_0 / b_0
    for i in range(2, max_iter):
        diag_mat = np.zeros((D,D))
        np.fill_diagonal(diag_mat, E_a)
        S = np.linalg.pinv(diag_mat + l * XX)
        m = l * np.dot(S, Xy)
        # Update expectation of E[w'w]
        E_ww = np.dot(m.T, m) + np.matrix.trace(S)
        # Update b_N parameter of Gamma factor
        b = b_0 + 0.5 * E_ww
        # Compute expectation of E[a]
        E_a = a / b
        # Compute lower bound
        lb_py = 0.5 * I * np.log(l / (2 * math.pi)) - 0.5 * l * yy + l * np.dot(m.T, Xy) \
            - 0.5 * l * np.matrix.trace(np.dot(XX,(np.dot(m.reshape(m.shape[0], 1), m.reshape(m.shape[0], 1).T) + S)
        lb_pw = -0.5 * D * np.log(2*math.pi) + 0.5 * D * (scipy.special.digamma(a) - np.log(b)) \
            - 0.5 * E_a * E_ww
        lb_pa = a_0*np.log(b_0) + (a_0 - 1) * (scipy.special.digamma(a) - np.log(b)) - b_0*E_a \
            - np.log(scipy.special.gamma(a_0))
        lb_qw = -0.5 * np.log(np.linalg.det(S)) - 0.5 * D * (1 + np.log(2*math.pi))
        lb_qa = -np.log(scipy.special.gamma(a)) + (a - 1)*scipy.special.digamma(a) + np.log(b) - a
        L[i] = lb_py + lb_pw + lb_pa - lb_qw - lb_qa
        #print("It:\t",i,"\tLB:\t", L[i],"\tLB_diff:\t", L[i] - L[i - 1],"\n")
        # Check if lower bound decreases
        if L[i] < L[i - 1]:
            print("Lower bound decrease \n")
        # Check for convergence
        if (L[i] - L[i - 1] < epsilon_conv):
            break
        # Has VI converged in the max iterations?
        if (i == max_iter) :
            print("Oops VI did not converge!\n")
    self.m, self.S, self.a, self.b, self.l, self.X, self.I, self.D, self.L = m, S, a, b, l, X, I, D, L[2:i]
    y_hat = self.predict(X)
    self.score = self.evaluate(y.reshape((y.shape[0], 1)), y_hat)
```

Variational principle:
General family of methods for approximating complicated densities by a simpler class of densities

(full derivation @ Pattern Recognition and Machine Learning by Bishop, chapter 10)

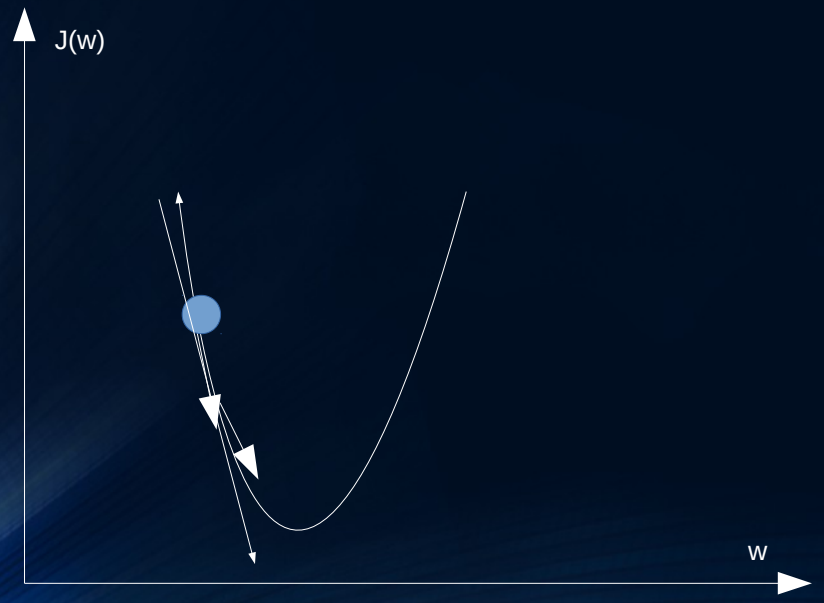


The immediate benefit of this is that after taking into account any data we can quantify an uncertainty in our parameters via the posterior distribution

GMDH – Fitting a node (3)

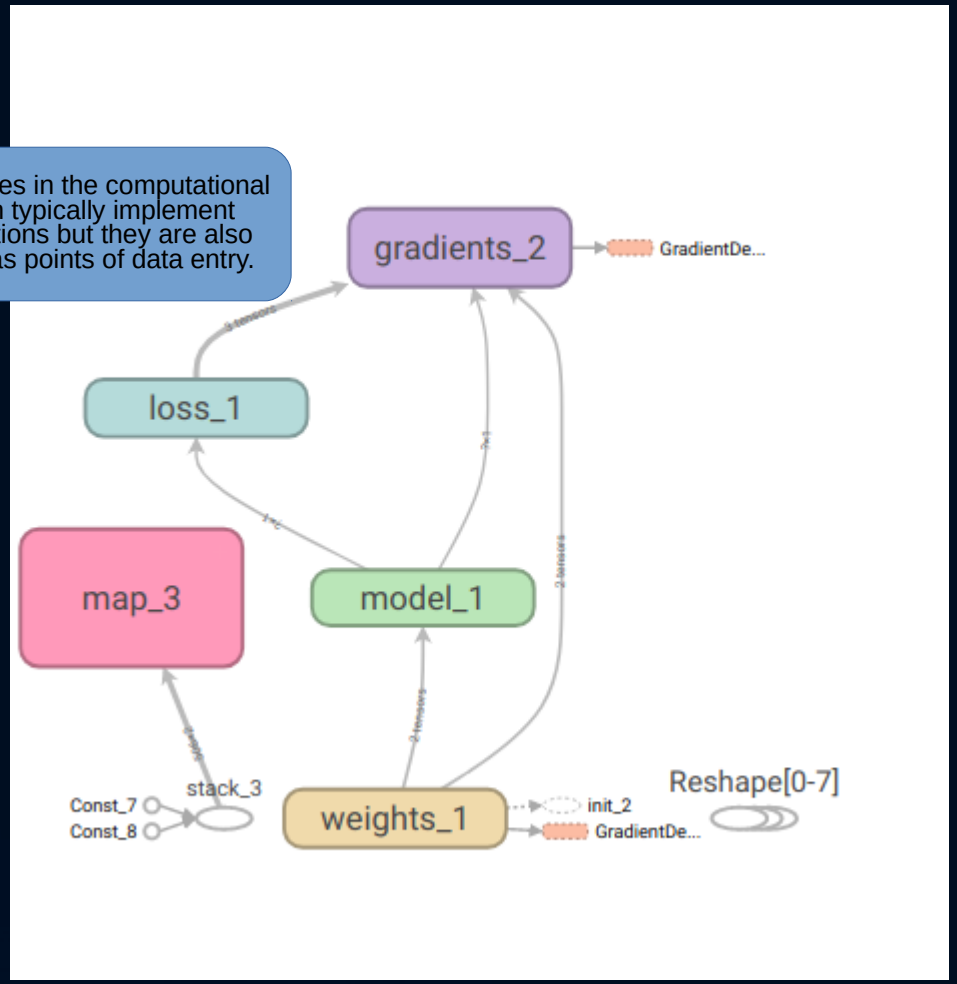
Gradient Descent

- In each iteration take a step in the opposite direction of the gradient.
- Batch gradient descent computes the gradient with respect to the entire training set: $\theta = \theta - \eta \nabla_\theta J(\theta)$
- Stochastic gradient descent performs parameter update after each training example: $\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$
- Mini batch gradient descent performs an update after n training examples: $\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}, y^{(i:i+n)})$



GMDH – Tensorflow to the rescue

```
def _transfer(self, X):
    X1 = self.inputNodes[0].predict(X)
    X2 = self.inputNodes[1].predict(X)
    X_t = tf.stack([X1, X2], axis=1)
    X_t = tf.map_fn(lambda x: [1., x[0], x[1], x[0]*x[1], x[0] * x[0], x[1] * x[1]], X_t, \
                    dtype=[tf.float32, tf.float32, tf.float32, tf.float32, tf.float32, tf.float32])
    return X_t

def predict(self, X, sess):
    X_t = np.array(sess.run(self._transfer(X)))
    X_t = np.transpose(X_t)
    y_pred = sess.run(tf.matmul(self.x, self.w), feed_dict = {self.x:X_t})
    return y_pred


def fit(self, X, y, sess):
    X_t = np.array(sess.run(self._transfer(X)))
    X_t = np.transpose(X_t)
    with tf.name_scope("model"):
        y_hat = tf.matmul(self.x, self.w)
    with tf.name_scope("loss"):
        loss = tf.reduce_mean(tf.square(y - y_hat))
    optimizer = tf.train.GradientDescentOptimizer(0.1)
    train_op = optimizer.minimize(loss)
    for i in range(200):
        sess.run(train_op, feed_dict={self.x:X_t})
        if i%20 == 0:
            [l, w] = sess.run([loss, self.w], feed_dict={self.x:X_t})
            print(l)
```



- Built around the idea of constructing and manipulating a computational graph which is a symbolical representation of the operations to be performed.

  y_hat = tf.matmul(x, W)
  loss = tf.reduce_mean(tf.square(y-y_hat))

GMDH – Advantages / Disadvantages

Advantages

- Simple to parallelize as nodes in each layer are fit independently of one another.
- Avoid intermediate computations during forward propagation by caching layer predictions.
- Simple to implement.
- Self-organizing structure generated during the training process.
- Only include the relevant terms (feature selection).
- Sparse connectivity which means that a discovered structure can be trained fast.

Disadvantages

- Tendency to generate quite complex polynomials for relatively simple systems
- Low accuracy has been observed for long range predictions
- For more than one output multiple models need to be built

GMDH – Load Forecasting (1)

- Day-ahead scheduling proceeds through an electricity market that the TSO operates. The generating units supply price-energy bids to a market pool and it is the responsibility of the TSO to produce a unit schedule for the day ahead (Day-ahead market).
- The problem of deciding whether a generating unit should be ON or OFF for a specified time schedule is called a unit commitment problem. Hence, It is a multiple-time optimization problem the goal of which is to produce ON/OFF schedules for all generating units that participate in the market in the most economical constrained to satisfy the load demand.
- How is the load demand for the next day determined => Short-term load forecast
- So it is all about determining resource requirements (load forecast) and then using efficiently those resources.

GMDH – Load Forecasting (2)

Forecasting – Basic steps

1. Define the problem: It involves an understanding of how the forecasts will be used in the organization.

2. Gather information: There are two types of information available: (a) the historical data and (b) the domain knowledge.

3. Data preparation: Tidy up your data.

4. Exploratory data analysis: We always start by graphing the data for visual inspection. Are there any consistent patterns? Is there a significant trend in the time series? Is seasonality important? Are there any outliers?

5. Choosing and fitting a model: There are always some inherent set of assumptions when building a model. Do they fit the problem at hand?

6. Evaluating the model performance: There is a variety of accuracy measures to evaluate the performance of a model. It is important to know how the forecasting method has performed in the particular context.

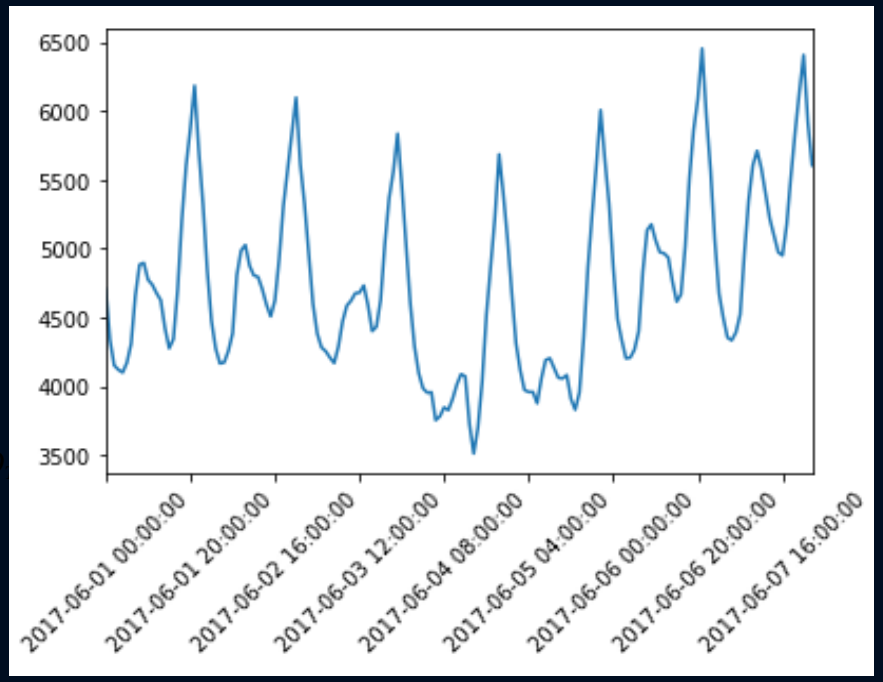
GMDH – Load Forecasting (3)

- Time series forecasting problem
- There are two major types of forecasting models:

  Time series and Explanatory models.
- Explanatory models assume that the quantity to be forecasted exhibits an explanatory relationship with one or more independent variables. For example:

  Load = f(temperature, time of day, season, is_holiday,...error)
- In time series models prediction of the future is based on past values of the quantity to be forecasted. The objective of time series models is to extrapolate a pattern into the future:

  $Load_{t+1} = f\left(Load_t, Load_{t-1}, Load_{t-2}, ..., error\right)$



GMDH – Preparing the dataset (1)

Arranging the data

	TRUNC_TIMESTAMP	MW	MW_1	MW_2	MW_3	MW_4	MW_5	MW_6
24	2017-06-02 00:00:00	4842	5320.0	5711.0	6183.0	5871.0	5603.0	5210.0
25	2017-06-02 01:00:00	4475	4842.0	5320.0	5711.0	6183.0	5871.0	5603.0
26	2017-06-02 02:00:00	4274	4475.0	4842.0	5320.0	5711.0	6183.0	5871.0
27	2017-06-02 03:00:00	4166	4274.0	4475.0	4842.0	5320.0	5711.0	6183.0
28	2017-06-02 04:00:00	4173	4166.0	4274.0	4475.0	4842.0	5320.0	5711.0

Inputs

Data points are shifted to the right to create lags

```
def add_lags(df, number_of_lags, merge_var = 'TRUNC_TIMESTAMP', y_var = 'MW'):
    y = df[y_var]
    y_shifted = pd.concat([df[merge_var]]+[y.shift(i) for i in range(1, number_of_lags + 1)], axis = 1)
    colnames = [merge_var] + ['{}_{}'.format(y_var, i) for i in range(1, number_of_lags + 1)]
    y_shifted.columns = colnames
    df_shifted = pd.merge(df, y_shifted, on=merge_var)
    df_shifted.dropna(inplace=True)
    return df_shifted
```

GMDH – Preparing the dataset (2)

Partition the dataset

```
X_train, X_test = X[0:4000,:], X[4000:,:]
y_train, y_test = y[0:4000], y[4000:]
```

This might seem a naive method to split the data but with time series splitting into training and test sets is more complicated as the time ordering has to be maintained.

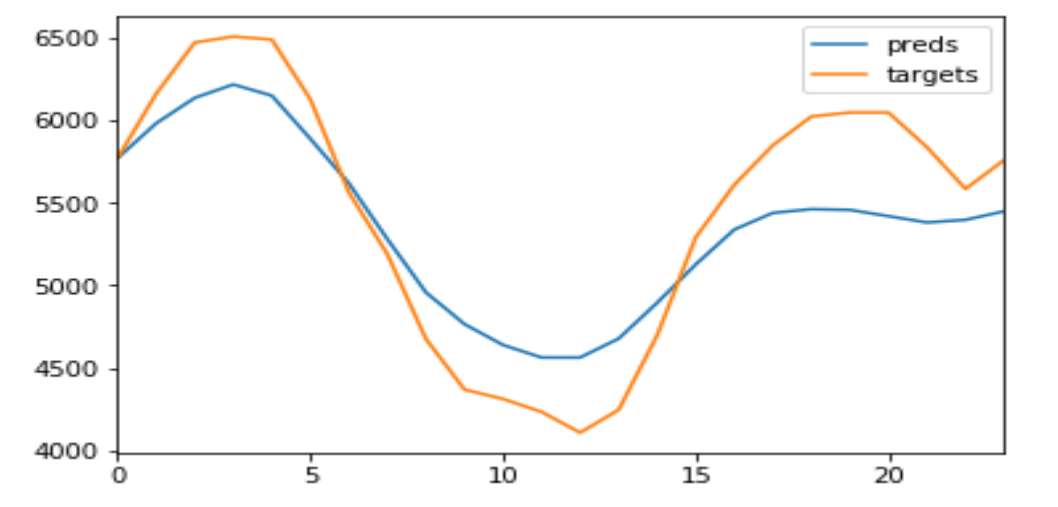
Normalize the data

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
from sklearn.preprocessing import MinMaxScaler

X_scaler = MinMaxScaler()
X_scaler.fit(X_train)

X_train = X_scaler.transform(X_train)
X_test = X_scaler.transform(X_test)

y_scaler = MinMaxScaler()
y_scaler.fit(y_train.reshape(-1,1))

y_train = y_scaler.transform(y_train.reshape(-1,1))
y_train = y_train.flatten()

y_test = y_scaler.transform(y_test.reshape(-1,1))
y_test = y_test.flatten()
```

Notice that the order of preparing the data is very important. First we split the dataset and then we normalize it using the estimator that is fit only in the training set, otherwise we “leak” information into the model.

GMDH – Running the model



Using the model to make predictions for 24 hours ahead

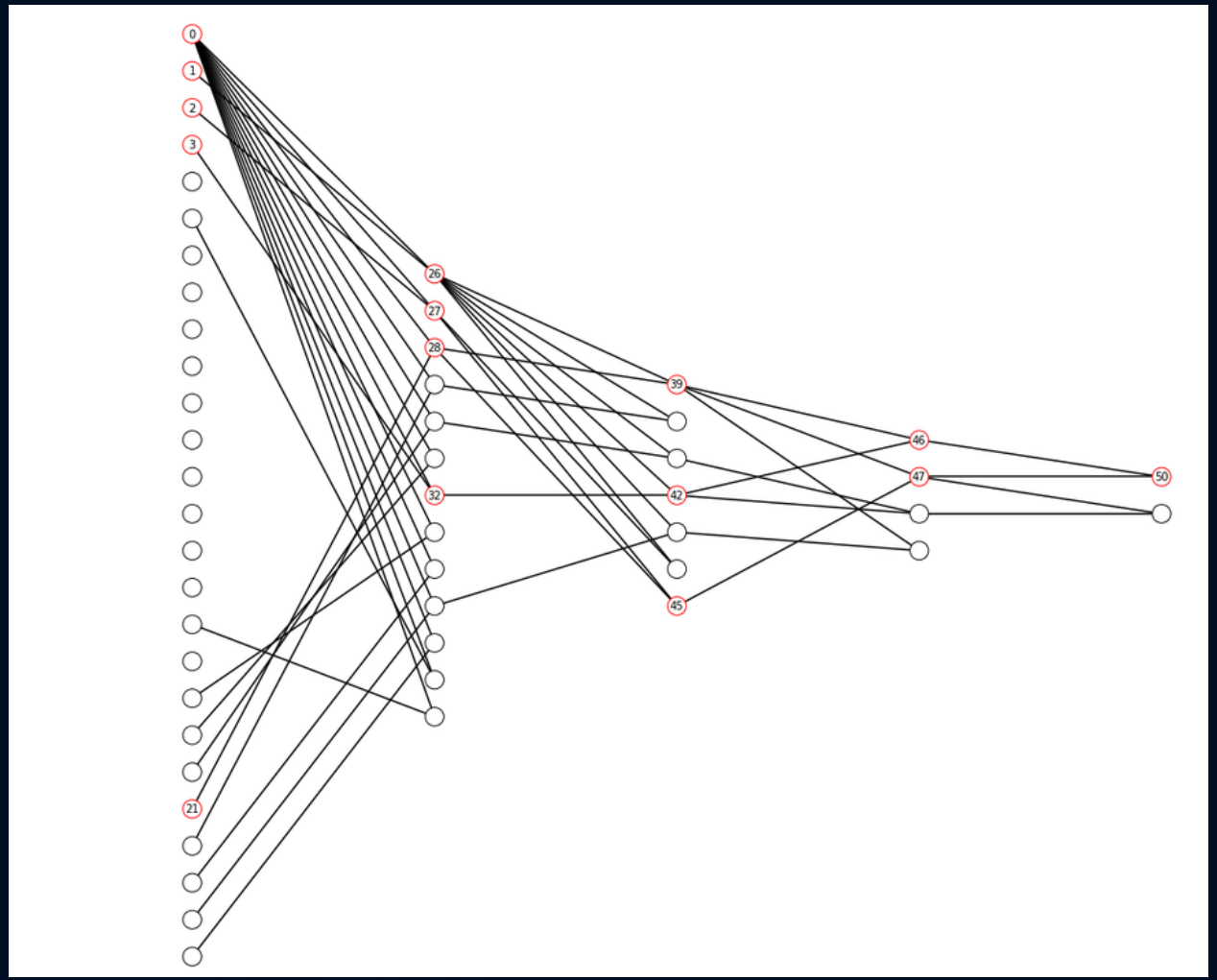
```
def predict_sequence(model, x, steps = 1, num_lags=24):
    y_hat = np.zeros((steps, 1))
    i = 0
    for step in range(steps):
        pred = model.predict(x)
        y_hat[step, 0] = pred

        x[0][1:(num_lags - 1)] = x[0][0:(num_lags-2)]
        x[0][0] = pred


    return y_hat
```

```
def predict(self, X):
    return self.predict_node.predict(X)
```

Predictions	Targets
5772	5771
5982	6162
6136	6471
6217	6508
6150	6490
5888	6128
5619	5564
5280	5189
4958	4676
4639	4313
4564	4235
4563	4109
4678	4274
4695	4699
5128	5292
5339	5610
5440	5849
5462	6024
5456	6048
5420	6040
5382	5838
5397	5585
5449	5759

GMDH – Interpreting the model

```
def get_active_path(self):
    queue = Queue()
    queue.put(self.predict_node)

    num_nodes = self.num_nodes()
    visited = dict()

    while not queue.empty():
        node = queue.get()

        if visited.get(node.node_id) is not None:
            continue

        if isinstance(node, PolynomialNode):

            visited[node.node_id] = (node.inputNodes[0].node_id, node.inputNodes[1].node_id)

            for v in node.inputNodes:
                if visited.get(v.node_id) is None:
                    queue.put(v)

    return visited
```



We can backtrack from the output layer to the input layer to find the active paths and draw our network.
This allows us to see which input features were important in our predictions and which input features were eliminated.

GMDH – Can we do better?

Improvements

1. Use additional predictors. In the example we presented only lagged values were used as input features.
2. Decompose the time series (identify trend and seasonality)
3. Use a different model for each forecast horizon. This will prevent forecast errors from accumulating.
4. Experiment with GMDH hyperparameters (external criterion, percentage elimination at successive layer building).
5. Use Revised Group Method of Data Handling. In a nutshell this method allows to use multiple transfer functions (for example passing the output of the polynomial function from a sigmoid function) for the nodes of the network. The transfer function for each node is determined during the training time.

Thank you!