 doi:10.20944/preprints202104.0523.v1

Article

Comparison of Backpropagation and Kalman Filter-based Training for Neural Networks

Laurin Luttmann [1,†,‡] and Paolo Mercorelli [1,‡]

[1] Institute of Product and Process Innovation, Leuphana University of Lueneburg, Universitaetsallee 1, D-21335 Lueneburg, Germany, mercorelli@uni.leuphana.de

Abstract: This work describes and compares the backpropagation algorithm with the extended Kalman filter, a second-order training method which can be applied to the problem of learning neural network parameters and is known to converge in only a few iterations. The algorithms are compared with respect to their effectiveness and speed of convergence using simulated data for both, a regression and a classification task.

Keywords: Backpropagation Algorithm; Kalman Filter; Neural Networks

1. Introduction

Neural networks (NN) have been successfully used in noise filtering and state estimation tasks [1]. Neural networks use backpropagation algorithm [2] in order to update the parameters such that the difference between the prediction of the network and the observed data is minimized [3]. However, such neural networks require a large amount of data. In addition, the structure of the network, defined by the number of hidden layers and the number of nodes in each hidden layer, needs to be large if the network is to successfully estimate a non-linear signal. One of the key downside of large number of data and complex structure required to train neural larger networks is also that the training time, and the prediction time, tends to be long.

Kalman filter enables inference of unmeasured variables from indirect and noisy measurement [4]. As a result, it is arguably one of the most important discoveries in the field of mathematical engineering and has been used to solve various engineering problems in the area of monitoring and control of complex dynamic systems such as manufacturing processes, aircraft navigation, ships, and spacecrafts. While the Kalman filter is used to estimate of a state vector in a linear model in a dynamical system, the extended Kalman filter (EKF) is used in order to estimate a non-linear model [5]. Since neural networks can be described as a composition of successive nonlinear functions, the EKF may be used to estimate the parameters of this non-linear model. As a learning method for neural networks, the EKF could overcome the drawbacks of standard neural networks and through the incorporation of second order information is likely to converge faster to an optimum.

In order to investigate whether the EKF can overcome the shortcomings of the backpropagation algorithm, this work describes and compares the backpropagation algorithm with the extended Kalman filter for learning neural network parameters. The Paper is organized in the following way. Section 2 is dedicated to the background of Feedforward neural networks. Section 3 introduces the Backpropagation algorithm and section 4 in the same way introduces the EKF. At the end the paper compares the results using simulated data. The conclusions close the paper.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

2. Feedforward neural networks

Feedforward neural networks, often also called Multilayer Perceptrons (MLPs), are compositions of many different functions. They are called feedforward, since the output of preceding functions is passed on to their successors. For example, the neural network might consists of the three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$, where the subscript denotes the layer of the network. The full model in this case would be of the form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, where x is a vector of input features [2]. The first layer of the neural network constructs M linear transformations of the input variables $x_1, ..., x_D$, where D is the dimensionality of the feature space describing an observation:

$$z_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b^{(1)}, \quad j = 1, ..., M \tag{2.1}$$

where j denotes the unit, often also referred to as neuron, of the layer (1) and $w_{ji}^{(1)}$ is a learnable weight which passes from the i-th unit of the previous layer (the i-th input feature in this case) to the j-th neuron of the current layer [11] . Moreover, $b_j^{(1)}$ is referred to as bias. In order to introduce nonlinearity to the network and thus be able to model all kinds of functions, the weighted inputs $z_j^{(1)}$ are transformed using a nonlinear activation function $\sigma(\cdot)$ which yields the activation or output of a neuron:

$$a_j^{(1)} = \sigma\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)}\right), \quad j = 1, ..., M\,. \tag{2.2}$$

The output of this layer is then passed to the next layer, leading to a function of the following form:

$$a_k^{(2)} = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} a_j^{(1)} + b_k^{(2)}\right), \quad k = 1, ..., K \tag{2.3}$$

where k denotes the neuron of the second layer of the network. Note, that the output activations of the preceding layer $a_j^{(1)}$ serve as input for this layer.

For ease of notation, these functions are typically defined in matrix notation, such that $\mathbf{a}^{(l)}$ refers to the output vector of the l-th layer, $\mathbf{w}^{(l)}$ is the matrix of weights and $\mathbf{b}^{(l)}$ is the vector of biases. Using this notation, the previously described network of 3 layers can then be described by the following set of functions [11]: The last layer is the output layer, which outputs the value of y. In between are the hidden layers, which are called hidden because the training data does not show the desired output for each of these layer [2]. Hence, a typical 3 layer feedforward neural network would look as follows (the input layer is not counted as layer):

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{w}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \tag{2.4}$$

$$\mathbf{a}^{(2)} = \sigma\left(\mathbf{w}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\right) \tag{2.5}$$

$$\hat{\mathbf{y}} = \sigma\left(\mathbf{w}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}\right). \tag{2.6}$$

Functions (2.4) and (2.5) are the hidden layers, which are called hidden because the training data does not show the desired output for each of these layers [2]. Function (2.6) is the output layer of the network which computes the approximation $\hat{\mathbf{y}}$ of the true label $\mathbf{y}$ corresponding to the input $\mathbf{x}$. The Forward-Propagation algorithm (see algorithm 1) generalizes the calculation of the output for any feedforward neural network with an arbitrary amount of layers L [2].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

Algorithm 1: Forward-Propagation algorithm

Function FORWARDPROPAGATION($\mathbf{x}$, θ)
Set the corresponding activation $\mathbf{a}^{(0)} = \mathbf{x}$ for the input layer
for $l = 1, 2, ..., L$ **do**
  $\mathbf{z}^{(l)} = \mathbf{w}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$
  $\mathbf{a}^{(l)} = \sigma(z^{(l)})$
end
return $\mathbf{z}^{(l)}$, $\mathbf{a}^{(l)}$ $\quad \forall l \in \{1, ..., L\}$

During training of the neural network one seeks to find the parameters $\mathbf{w}^{(l)}$ and $\mathbf{b}^{(l)}$ that make the network best approximate the true function $f^* : \mathbf{x} \mapsto \mathbf{y}$. To guide this learning behavior the network requires a loss function that quantifies errors in the prediction process. A typical loss function is the quadratic loss:

$$C(\theta) = \frac{1}{2} \sum_{n=1}^{N} ||f(\mathbf{x}_n, \theta) - \mathbf{y}_n||^2, \tag{2.7}$$

where N denotes the total number of training observation, θ is the collection of parameters of the model and $f(\cdot)$ describes the composition of functions in the neural network [11].

3. Backpropagation Algorithm

In order to adjusts the weights and biases, i. e. the parameters of the model, so that the loss function C arrives at a minimum, the derivatives of the loss function with respect to the weights and biases are needed. To compute these derivatives one can make use of the chain rule of calculus, which states that for a function composition $f(g(x))$ the derivative of $f(\cdot)$ w.r.t. x can be written as $\partial f(x)/\partial g(x) \times \partial g(x)/\partial x$ [2]. For the cost function of a neural network, which is a function composition of the many functions building up the hidden and the output layers, the derivative with respect to the weights can be computed as follows [12]:

$$\frac{\partial C}{\partial w_{ji}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \times \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} \tag{3.1}$$

Since $z_j^{(l)}$ is the weighted sum of the inputs plus a bias term, the derivative of it with respect to the weight w_{ji} is the input $a_i^{(l-1)}$ and 1 if the derivative is computed with respect to the bias. In order to determine the derivative of the cost function with respect to the weighted input $z_j^{(l)}$, the following notation is introduced:

$$\delta_j \equiv \frac{\partial C}{\partial z_j^{(l)}}, \tag{3.2}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

where δ_j is usually referred to as error, since for the output units this term simplifies to the difference between the true value y and its estimate $\hat{y}$ as we will see shortly [13]. In order to compute $\delta_j^{(l)}$, again the chain rule can be used:

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} \tag{3.3}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \times \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \tag{3.4}$$

$$= \sum_k \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \delta_k^{(l+1)} . \tag{3.5}$$

Since $z_k^{(l+1)}$ is defined as

$$z_k^{(l+1)} = \sum_j w_{kj}^{(l+1)} \sigma(z_j^{(l)}) + b_k^{(l+1)} \tag{3.6}$$

its derivative with respect to $z_j^{(l)}$ is equal to $w_{kj}^{(l+1)} \sigma'(z_j^{(l)})$. Substituting this back into (3.5) yield:

$$\delta_j^{(l)} = \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)} \sigma'(z_j^{(l)}) \tag{3.7}$$

Now, the error of a layer (l) depend on the error of the succeeding layer $(l+1)$, raising the question as to how it is computed for the output layer of the network, which has no such successor. Instead of using the weighted input of the next layer, the derivative of the cost function is computed w.r.t. the output activation $a_j^{(L)}$:

$$\delta_j^{(L)} = \sum_k \frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} , \tag{3.8}$$

where the sum goes over all neurons k in the output layer [13]. Since the output activation $a_k^{(L)}$ depends only on the weighted input of the same neuron, $\partial a_k^{(L)} / \partial z_j^{(L)}$ vanishes for all $k \neq j$. Hence, the equation simplifies to

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} . \tag{3.9}$$

As $a_j^{(L)} = \sigma(z_j^{(L)})$, the second term on the right hand side of equation (3.8) is equivalent to the derivative of the activation function of the final layer with respect to its input $z_j^{(L)}$. Hence, this term can be written as $\sigma'(z_j^{(L)})$, which yields:

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) . \tag{3.10}$$

Assuming a quadratic loss function as described in (2.7), the derivative of C w.r.t. the network's output $a_j^{(L)} \equiv \hat{y}_j$ is equal to $(\hat{y}_j - y_j)$ and hence the name error for the δ_j terms [11].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

Again, these equations can be rewritten using the matrix notation which was introduced in section 2. The error of the output layer then becomes

$$\boldsymbol{\delta}^{(L)} = \nabla_{\hat{\mathbf{y}}} C \odot \sigma'(\mathbf{z}^{(L)}), \tag{3.11}$$

where $\nabla_{\hat{\mathbf{y}}} C$ is the gradient of the cost function with respect to the output of the network and $\odot$ is the pairwise multiplication operator, usually referred to as Hadamard product [13]. For all other layers, function (3.7) changes to:

$$\boldsymbol{\delta}^{(l)} = \left((\mathbf{w}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)} \right) \odot \sigma'(\mathbf{z}^{(l)}). \tag{3.12}$$

Given the errors it is easy to compute the derivatives of the cost function with respect to the weights and biases. As shown above, the error term $\delta^{(l)}$ must be multiplied with the input of the respective layer $a^{(l-1)}$ when computing the derivative with respect to the weights and with unity when computing it with respect to the biases. The backpropagation algorithm now serves an efficient means to compute these derivatives recursively (see algorithm 2).

Algorithm 2: Backpropagation algorithm

Function BACKPROPAGATION$(\mathbf{x}, \theta)$
Calculate all $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$ with FORWARDPROPAGATION$(\mathbf{x}, \theta)$
$\boldsymbol{\delta}^{(L)} = \nabla_{\hat{\mathbf{y}}} C \odot \sigma'(\mathbf{z}^{(L)})$
for $l = L-1, L-2, ..., 1$ **do**
  $\boldsymbol{\delta}^{(l)} = ((\mathbf{w}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$
end
return $\frac{\partial C}{\partial \mathbf{w}^{(l)}} = \mathbf{a}^{(l-1)} \boldsymbol{\delta}^{(l)}$ and $\frac{\partial C}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \quad \forall l \in \{1, ..., L\}$

Given a set of weights and input features, the backpropagation algorithm starts with calculating the weighted inputs $\mathbf{z}^{(l)}$ and activations $\mathbf{a}^{(l)}$ iteratively for each layer in ascending order by using forward propagation (see algorithm 1) Then, the error of the output layer is computed using formula (3.11). Given $\boldsymbol{\delta}^{(L)}$, all other errors can be computed by iterating through all layers in reversed order and applying formula (3.7). Finally, the derivatives of the cost function with respect to the weights and biases can be returned by multiplying the error terms with the input of the respective layers and with a vector of ones respectively [12].

Given a set of randomly initialized weights and biases, the networks weights can be learned by an iterative optimization algorithm in combination with backpropagation. Typically, the cost function is minimized using stochastic gradient descent or a variant thereof (see algorithm 3). For a specified number of training iterations, called epochs, this algorithm draws random examples from the set of training observations and calculates the gradients using backpropagation on those. The gradients are then used to update the weights by multiplying them with a learning rate $\alpha \ll 1$ and subtracting the resulting products from the respective weights. For each training iteration, this process is repeated until all training observations have been used for calculating the gradient [14].

4. Kalman Filter Training

4.1. Extended Kalman Filter

The Extended Kalman Filter (EKF) addresses the general problem of trying to estimate the state of a discrete time controlled process that is governed by the nonlinear discrete time system of the form

$$\mathbf{w}_{k+1} = f(\mathbf{w}_k, \mathbf{x}_k) + \text{!}_k \tag{4.1}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

Algorithm 3: Stochastic Gradient Descent

Function StochasticGradientDescent($\mathbf{x}$)
Randomly initialize weights and biases
$\theta \leftarrow \{\mathbf{w}^{(l)}, \mathbf{b}^{(l)}\}_{1 \leq l \leq L}$
repeat
  Randomly permute data $\mathbf{x}$
  for $i = 1,\ldots,N$ **do**
    $\nabla_\theta C \leftarrow$ BackPropagation($\mathbf{x}_i, \theta$)
    $\theta \leftarrow \theta - \alpha \nabla_\theta C$
  end
until *stopping criterion is reached*
yield Learned weights and biases $\hat{\mathbf{w}}^{(l)}, \hat{\mathbf{b}}^{(l)}, \ 1 \leq l \leq L$

with the observations or measurements:

$$\mathbf{y}_k = h(\mathbf{w}_k) + \ ^\circ{}_k, \tag{4.2}$$

where $\mathbf{w}_k$ is the state of the system at time-step k, $\mathbf{x}_k$ is an input of forces controlling the system and $!_k$ and $^\circ{}_k$ are the process and observation noises respectively [15]. Both noise terms are assumed to be zero mean multivariate Gaussian noises with covariance $\mathbf{Q}_k$ and $\mathbf{R}_k$ respectively. The nonlinear function $f(\cdot)$ relates the state at the current time step k to the next time step $k+1$ using the additional information $\mathbf{x}_k$ about the process. Likewise, $h(\cdot)$ relates the state $\mathbf{w}_k$ to the observation $\mathbf{y}_k$ [16]. The goal of the Extended Kalman Filter is to find an estimate $\hat{\mathbf{w}}_{k+1}$ of $\mathbf{w}_{k+1}$ given the observations or measurements $\{\mathbf{y}_j\}_{0 \leq j \leq k}$ [6].

One can show that this estimate can be obtained by the recursion

$$\mathbf{K}_k = \mathbf{P}_k \mathbf{H}_k [\mathbf{R}_k + \mathbf{H}_k^T \mathbf{P}_k \mathbf{H}_k]^{-1} \tag{4.3}$$

$$\hat{\mathbf{w}}_{k+1} = \hat{\mathbf{w}}_k + \mathbf{K}_k [y_k - h(\hat{\mathbf{w}}_k)] \tag{4.4}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{K}_k \mathbf{H}_k^T \mathbf{P}_k + \mathbf{Q}_k \tag{4.5}$$

where $\mathbf{K}_k$ is referred to as the Kalman gain which specifies how much weight should be attributed to the difference of the new measurement oberserved in time step k and the estimate of that measured value when computing the new state estimate for $k+1$ [16]. Note, that the estimate of the observed value is simply defined as $h(\hat{\mathbf{w}}_k)$, since the expected value of the measurement noise $^\circ{}_k$ is equal to zero. The Kalman gain itself is defined in terms of the error covariance matrix of the state $\mathbf{P}_k = E[(\mathbf{w}_k - \hat{\mathbf{w}}_k)(\mathbf{w}_k - \hat{\mathbf{w}}_k)^T]$, the measurement covariance $\mathbf{R}_k$ and the Jacobian of the measurement equation with respect to the state estimates $\hat{\mathbf{w}}_k$ [15]:

$$\mathbf{H}_k = \frac{\partial h(\cdot)}{\partial \hat{\mathbf{w}}_k}. \tag{4.6}$$

4.2. Neural Network Training with the EKF

Feedforward neural networks can be described as nonlinear systems by letting the weight parameters of the network constitute the hidden state of such a system which should then be estimated [6]. Therefore, we let all weights and biases of the network be arranged in the state vector:

$$\mathbf{w} = [\{\mathbf{w}^{(l)}, \mathbf{b}^{(l)}\}_{1 \leq l \leq L}]^T. \tag{4.7}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

According to [9], a neural network's behavior can then be described by the following nonlinear discrete-time system

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{!}_k \tag{4.8}$$

$$\mathbf{y}_k = \mathbf{a}_k^{(L)} + \mathbf{^\circ}_k = h(\mathbf{w}_k, \mathbf{x}_k) + \mathbf{^\circ}_k\,, \tag{4.9}$$

where $\mathbf{x}_k$ again is a vector of input variables $x_1, ..., x_D$ describing an observation, $\mathbf{y}_k$ is the target vector and $\mathbf{a}_k^{(L)}$ is the vector of outputs of the neural network, which is produced by the nonlinear function composition

$$h(\mathbf{w}_k, \mathbf{x}_k) = f^{(L)}(f^{(L-1)}(\cdots f^{(1)}(\mathbf{x}_k, \mathbf{w}_k^{(1)}), \mathbf{w}_k^{(L-1)}), \mathbf{w}_k^{(L)})\,. \tag{4.10}$$

It follows, that the weights of the neural network can be estimated using the EKF recursions described by (4.3 – 4.5). The EKF algorithm performs an online optimization procedure, meaning that training instances are processed one at a time, and tries to find weights that minimize the squared error between $\mathbf{y}_k$ and $\mathbf{a}_k^{(L)}$ [9].

The EKF training algorithm for neural networks (see algorithm 4) again requires a random initialization of the state $\hat{\mathbf{w}}_0$, constituting of all weights and biases of the network. Moreover, $\mathbf{P}_0$, $\mathbf{R}_0$ and $\mathbf{Q}_0$ have to be specified by the user.

Given the state and an observed input vector $\mathbf{x}_k$, the forward propagation described in algorithm 1 can be used to determine an estimate for the measurement value. Moreover, the backpropagation algorithm (see algorithm 2) can be used to derive the partial derivatives of $h(\cdot)$ with respect to the state estimates $\hat{\mathbf{w}}_k$. Note here, that instead of computing the Jacobian of the cost function w.r.t. the weights of the network, the backpropagation is used to determine the Jacobian of the network's output w.r.t. its weights. Therefore, the initial error term $\delta^{(L)}$ described in formula (3.11) reduces to the derivative of the activation function of the last layer w.r.t. its input $\mathbf{z}^{(L)}$ [10].

After $\hat{\mathbf{y}}_k$ and $\mathbf{H}_k$ are computed, the weights of the network are updated using the EKF recursions described by (4.3 – 4.5). Formula (4.4) acts here as the weight update equation, similar to the one used in stochastic gradient descent described in algorithm 3 (see line 8). But unlike the standard backpropagation, which uses the "delta rule" to move the error in the estimate through all neurons of the network, this procedure uses the Kalman gain to propagate the error $(\mathbf{y}_k - \hat{\mathbf{y}}_k)$ to the weights [6]. The Kalman gain makes use of entire gradient matrix and the error covariance matrix which models the interactions of each pair of weights in the network [9]. Hence, the Kalman gain uses global information to compute the weight updates and consequently should be able to converge much faster then the standard backpropagation algorithm, which only considers first-order information.

4.3. Choice of Parameters and Initialization

In contrast to the standard stochastic gradient descent with backpropagation algorithm, whose only hyperparameter is the learning rate α, the EKF algorithm relies on the parameters $\mathbf{R}_k$, $\mathbf{P}_k$ and $\mathbf{Q}_k$.

[10] note that the measurement noise $\mathbf{R}_k$ is equivalent to the inverse of the learning rate α of standard backpropagation, i. e. $\mathbf{R}_k \equiv \frac{1}{\alpha}$. That is, if the measurement noise is high, new observations y_k are trusted less and consequently the Kalman gain is low, leading to the new state estimate being mostly influences by the a priori estimate. Likewise, a small learning rate corresponds to small steps in the direction of the steepest descent of a single training observation. In general it is not possible to calculate an optimal learning rate a priori [17]. Hence, this parameter must be tuned, using for example a grid search. Typically, a grid search for the learning rate α involves picking values on a logarithmic scale, e.g. from the set $\{0.1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}\}$ [2]. The learning algorithm is then applied with each one of the learning rates and the performance is evaluated on a test data set [18]. However, in scenarios of signal processing, where data

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

Algorithm 4: EKF-Training

Function KALMANTRAINING($\mathbf{x}, \mathbf{P}_0, \mathbf{R}_0, \mathbf{Q}_0$)
Randomly initialize weights and biases
$\hat{\mathbf{w}}_0 \leftarrow [\{\mathbf{w}^{(l)}, \mathbf{b}^{(l)}\}_{1 \leq l \leq L}]^T$
$k \leftarrow 0$
repeat
  observe training pattern with label $\mathbf{y}_k$ and input vector $\mathbf{x}_k$
  $\hat{\mathbf{y}}_k \leftarrow$ FORWARDPROPAGATION$(\mathbf{x}_k, \hat{\mathbf{w}}_k)$
  $\mathbf{H}_k \leftarrow$ BACKPROPAGATION$(\mathbf{x}_k, \hat{\mathbf{w}}_k)$
  $\mathbf{K}_k \leftarrow \mathbf{P}_k \mathbf{H}_k [\mathbf{R}_k + \mathbf{H}_k^T \mathbf{P}_k \mathbf{H}_k]^{-1}$
  $\hat{\mathbf{w}}_{k+1} \leftarrow \hat{\mathbf{w}}_k + \mathbf{K}_k [\mathbf{y}_k - \hat{\mathbf{y}}_k]$
  $\mathbf{P}_{k+1} \leftarrow \mathbf{P}_k - \mathbf{K}_k \mathbf{H}_k^T \mathbf{P}_k + \mathbf{Q}_k$
  $k \leftarrow k + 1$
until *stopping criterion is reached*
yield Learned weights and biases $\hat{\mathbf{w}}^{(l)}, \hat{\mathbf{b}}^{(l)}, \ 1 \leq l \leq L$

arrives continuously, ad hoc adaptation of the learning rate is required if the learning algorithm does not yield satisfying results [17]. [19] presents a method to update the estimate for the measurement noise during each iteration of the Kalman recursion. The authors use a residual based approach, where the residual $\boldsymbol{\epsilon}_k$ is defined as the difference between the actual measurement and its a posteriori estimate at time step k, hence $\boldsymbol{\epsilon}_k = \mathbf{y}_k - \hat{\mathbf{y}}_k^+ = \mathbf{y}_k - h(\hat{\mathbf{w}}_{k+1}, \mathbf{x}_k)$. The update equation for $\mathbf{R}_k$ is then defined as follows:

$$\mathbf{R}_k = \eta \mathbf{R}_{k-1} + (1 - \eta)(\boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T \mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^T), \tag{4.11}$$

where η is a forgetting factor that is used to average the sum of squared residuals $\boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T$ over time and thus approximate its expected value. [19] recommend setting this parameter to a value of 0.3. For the initial measurement noise $\mathbf{R}_0$ this work follows [10], who recommend setting the learning rate α to a small value of around 0.01, which corresponds with a $\mathbf{R}_0$ of 100, due to its inverse relation.

The process noise $\mathbf{Q}_k$ is an important factor as it can help to prevent the error covariance matrix $\mathbf{P}_k$ from converging towards zero, which would imply a Kalman filter of zero and thus that no learning is taking place anymore [20]. [19] therefore derived an adaptive estimation of $\mathbf{Q}_k$ that uses the innovation $\xi_k = (\mathbf{y}_k - h(\mathbf{x}_k, \hat{\mathbf{w}}_k))$. The authors show, that the process noise covariance can be update using the following equation:

$$\mathbf{Q}_k = \eta \mathbf{Q}_{k-1} + (1 - \eta)(\mathbf{K}_k \xi_k \xi_k^T \mathbf{K}_k^T), \tag{4.12}$$

where again η is a forgetting factor. The initial value $\mathbf{Q}_k$ is usually set to $q\mathbf{I}$, with q being a small value. In this work, q is chosen to be 10^{-2} to initialize the process noise.

Since the state covariance matrix $\mathbf{P}_k$ is already iteratively updated by the extended Kalman Filter, its initial value is not as important as the former two [21]. However, [10] recommend setting it to $\epsilon^{-1}\mathbf{I}$, where ϵ^{-1} is a small number from the range 0.001–0.01 and $\mathbf{I}$ is the identity matrix. The authors state that by setting $\mathbf{P}_0$ to a diagonal matrix, the fact that weights are initialized randomly without correlating on each other is reflected. Moreover, due to the random initialization, the diagonal entries are set to rather high values to account for the resulting uncertainty associated with the initial state $\hat{\mathbf{w}}_0$.

In fact, for neural networks, the initial state $\hat{\mathbf{w}}_0$ should generally be drawn independently from a uniform or normal distribution. [22] for example recommend to draw weights $\mathbf{w}^{(l)}$ independently from a normal distribution with zero mean and standard deviation equal to $\sqrt{2/n^{(l-1)}}$, with $n^{(l-1)}$ being the number of neurons in layer $(l-1)$. Moreover, the authors state that biases should be initialized with zero.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

5. Computational Experiments

To evaluate the computational efficiency and predictive effectiveness of the extended Kalman Filter for training neural networks, it will be used together with the standard backpropagation (SBP) algorithm to fit neural networks on different data sets.

The network architecture regarding the hidden layers is the same for every experiment and comprises of two hidden layers with 20 and 10 neurons respectively. The activation function for each neuron in the hidden layers is a Rectified Linear Unit (ReLU), which is defined as $\sigma(x) = \max(0, x)$ [12]. The activation function in the output layer is dependent on the problem at hand. For regression tasks, a linear activation (i.e. $\sigma(x) = x$) is used, whereas for classification tasks the sigmoid function $\sigma(x) = e^x/(1 + e^x)$ determines the final network output.

First, the two presented optimization algorithms have been used to fit a neural network to a regression task with one-dimensional input, where the true function $f(x)$ is a 5th order Legendre polynomial [23]:

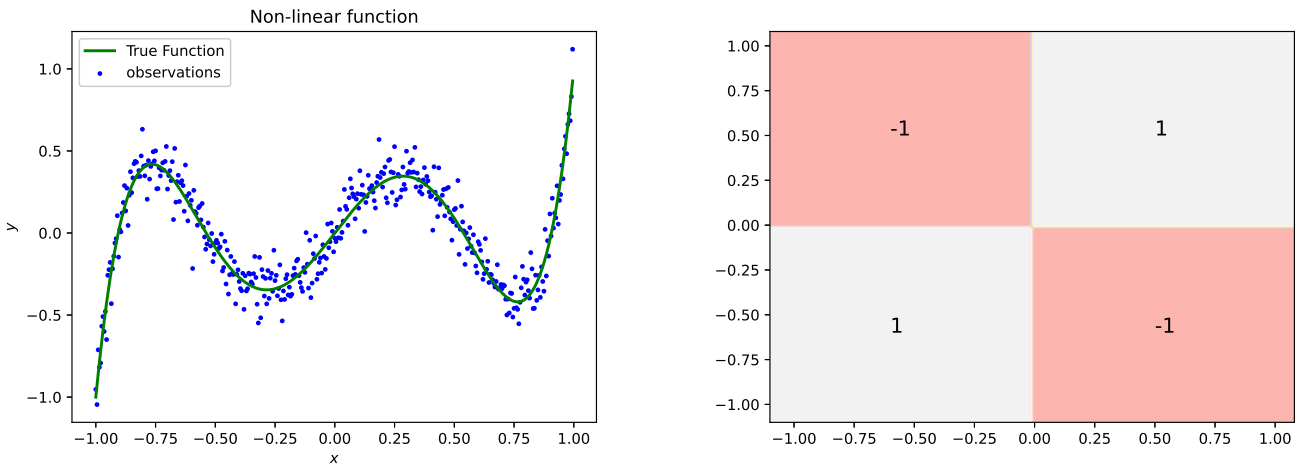
$$f(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x)\,. \tag{5.1}$$

To generate the training data, n input values x are randomly drawn from a uniform distribution limited by the interval $[-1, 1]$ and the corresponding labels are calculated using the function $f(x)$ and corrupting the result with zero mean Gaussian noise, i. e. $y = f(x) + \nu$, where ν is drawn from $\mathcal{N}(0, \sigma^2\mathbf{I}_{n\times n})$. The resulting data can be seen in figure 1a.

Next to the regression problem, the classical XOR ("exclusive or") classification problem is considered. Given two input features x_1 and x_2, the XOR function computes:

$$f(x_1, x_2) = sign(x_1) * sign(x_2)\,, \tag{5.2}$$

where $sign(x) = +1$ if $x > 0$ and -1 otherwise. The decision boundary for this function is visualized in figure 1b.

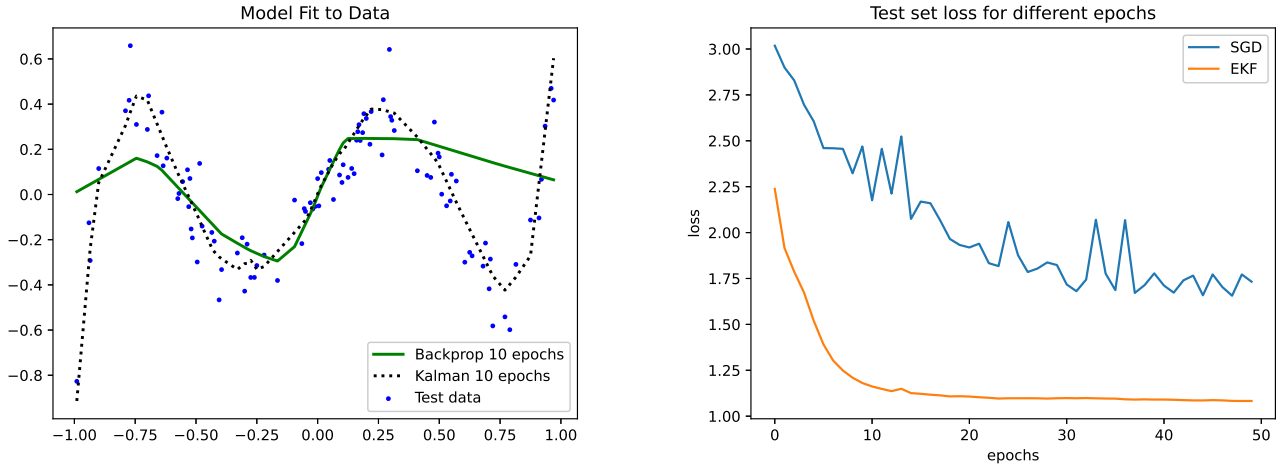


(a) Function and data for regression task **(b)** XOR Function

Figure 1. Regression and classification task

For the regression problem it can be observed, that the neural network fitted using the EKF method achieved very good results after only 10 epochs, while the network trained with the SBP algorithm results in a rather poor fit to the data (see figure 2a). The EKF method also outperforms the SBP algorithm during the course of 50 training epochs considering the loss on the test set (see figure 2b).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1



(a) Fit to regression task **(b)** Evolution of test set loss

Figure 2. Model performance on regression task

The quality of the fit resulting from the EKF algorithm can also be attributed to the recursive update of the measurement noise $\mathbf{R}_k$ and process noise $\mathbf{Q}_k$ through formulas (4.11) and (4.12) respectively. Note that by setting $\eta = 1$, the respective noise parameters are not updated, hence leading to the standard EKF recursions. Comparing the results for $\eta = 0.3$ and $\eta = 1$ shows that the recursive estimates of these parameters contribute to better fits of the neural network to the data, especially if the noise level σ^2 is high. Figure 3 shows the fit for $\eta = 1$ (left) and $\eta = 0.3$ (right). For several different training patterns with different noise levels σ^2 from the interval $[0.3, 0.4]$, the EKF method with recursive updates of $\mathbf{R}_k$ and $\mathbf{Q}_k$ achieved on average a 13% lower mean squared error compared to the case where $\eta = 1$.

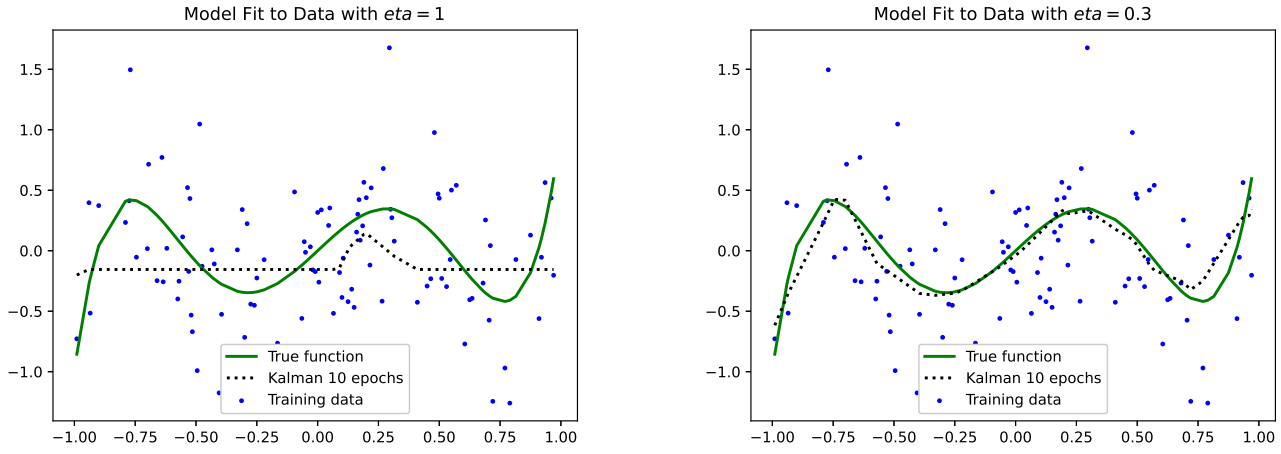


Figure 3. Effect of the adaptive noise estimates

Lastly, the performance of the SBP and EKF training methods for the classification task can be seen in figure 4 and 5 respectively. In this case, the SBP achieves better results than the EKF method and gets very close to a loss of zero. The decision boundaries determined by SBP after 50 epochs are very smooth, while those of the EKF are rather uneven.

The reason for the inferiority of the EKF method are probably due to the fact, that this method implicitly minimizes the quadratic loss, while for the SBP any loss function can be minimized. For classification problems, the binary cross entropy is usually

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 April 2021 doi:10.20944/preprints202104.0523.v1

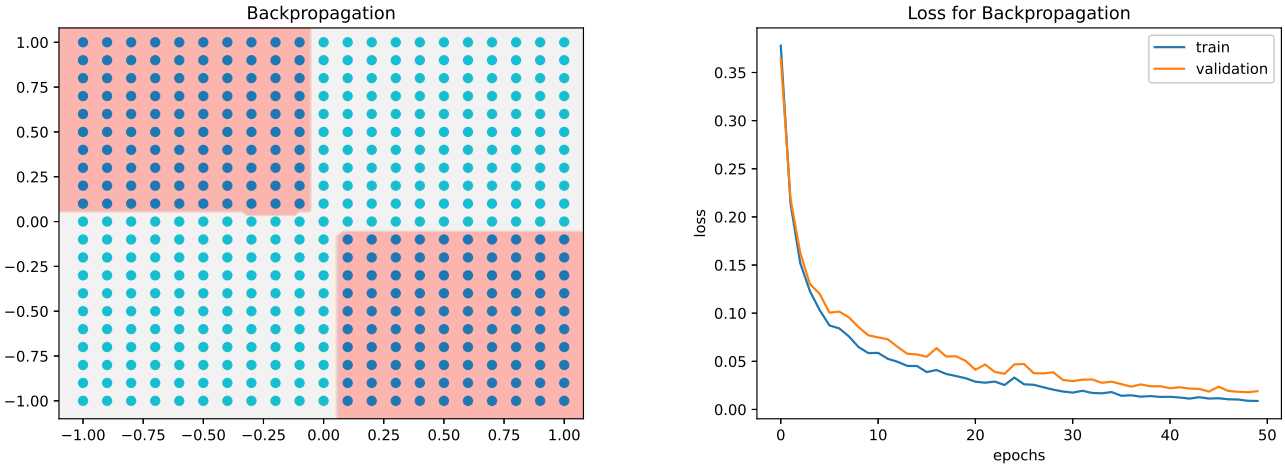


Figure 4. SBP Performance on XOR data

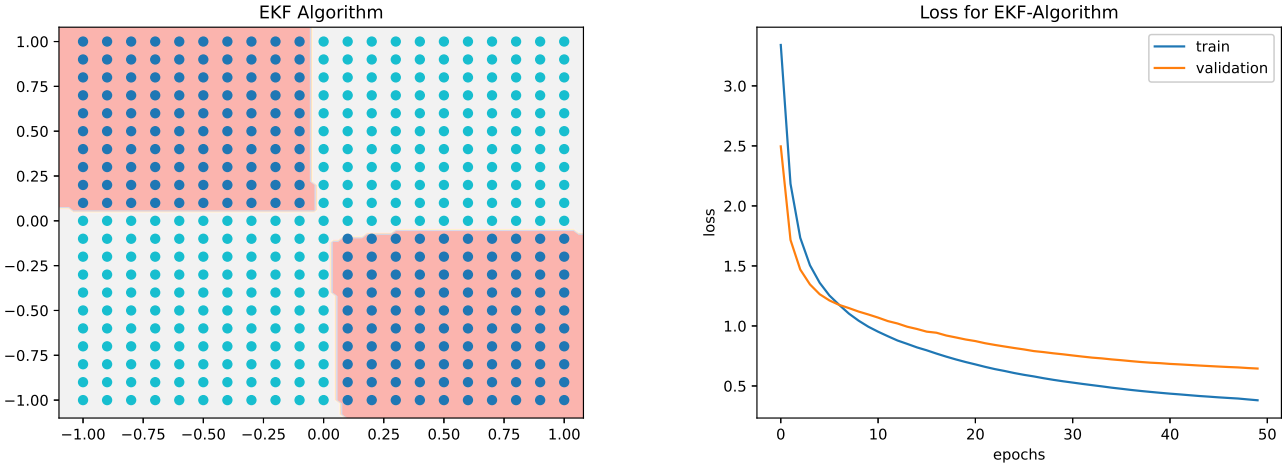


Figure 5. EKF Performance on XOR data

minimized and has also been used to produce the results seen in figure 4. Indeed, when changing the loss of SBP to the mean squared error, the EKF even produces a smaller error than the SBP. However, the implicit assumption of a quadratic loss presents a major flaw of the EKF algorithm for training neural networks, as also noted by [10].

6. Conclusions

Throughout this paper it was shown, that learning the parameters of a feedforward NN can be translated into a state estimation problem for a nonlinear dynamic system and hence be solved by the EKF. The EKF procedure has shown to converge extremely fast to an optimum compared to the standard backpropagation algorithm typically employed to train NNs. This makes the EKF-based training of such networks especially powerful in applications, where the parameters need to be learned fast, as it is the case if labeled data are scarce or the data generating process is highly non-stationary [9]. Moreover, the recursive update strategy of the noise parameters $\mathbf{R}_k$ and $\mathbf{Q}_k$ makes the model more robust against improper initial values of these parameters, which especially in noisy scenarios has lead to better predictive performance. Future research should now investigate how the EKF method can also be used to learn the parameters of neural networks with loss functions other than the mean square error.

References

1. Bai, Y.t.; Wang, X.y.; Jin, X.b.; Zhao, Z.y.; Zhang, B.h. A Neuron-Based Kalman Filter with Nonlinear Autoregressive Model. *Sensors* **2020**, *20*. doi:10.3390/s20010299.
2. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press, 2016. http://www.deeplearningbook.org.
3. Iiguni, Y.; Sakai, H.; Tokumaru, H. A real-time learning algorithm for a multilayered neural network based on the extended Kalman filter. *IEEE Transactions on Signal Processing* **1992**, *40*, 959–966. doi:10.1109/78.127966.
4. Grewal, M.S.; Andrews, A.P. *Kalman filtering: Theory and Practice with MATLAB*; John Wiley & Sons, 2014.
5. Haykin, S. *Kalman filtering and neural networks*; Vol. 47, John Wiley & Sons, 2004.
6. Singhal, S.; Wu, L. Training Multilayer Perceptrons with the Extende Kalman Algorithm. NIPS, 1988, Vol. 1, pp. 133–140.
7. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* **2012**, *25*, 1097–1105.
8. Schmidhuber, J. Deep Learning in Neural Networks: An Overview. *CoRR* **2014**, *abs/1404.7828*, [1404.7828].
9. Vural, N.M.; Ergut, S.; Kozat, S.S. An efficient and effective second-order training algorithm for lstm-based adaptive learning. *arXiv preprint arXiv:1910.09857* **2019**.
10. Puskorius, G.V.; Feldkamp, L.A. Parameter-based kalman filter training: theory and implementation. *Kalman filtering and neural networks* **2001**, p. 23.
11. Bishop, C.M. *Pattern recognition and machine learning*; Springer, 2006.
12. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *nature* **2015**, *521*, 436–444.
13. Nielsen, M.A. *Neural networks and deep learning*; Vol. 25, Determination press San Francisco, CA, 2015.
14. Murphy, K.P. *Machine learning: a probabilistic perspective*; MIT press, 2012.
15. Haykin, S. Kalman filters. *Kalman Filtering and Neural Networks* **2001**, pp. 1–21.
16. Welch, G.; Bishop, G. *An introduction to the Kalman filter*; Chapel Hill, NC, USA, 1995.
17. Reed, R.; MarksII, R.J. *Neural smithing: supervised learning in feedforward artificial neural networks*; Mit Press, 1999.
18. Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research* **2012**, *13*.
19. Akhlaghi, S.; Zhou, N.; Huang, Z. Adaptive adjustment of noise covariance in Kalman filter for dynamic state estimation. 2017 IEEE power & energy society general meeting. IEEE, 2017, pp. 1–5.
20. Bar-Shalom, Y.; Li, X.R.; Kirubarajan, T. *Estimation with applications to tracking and navigation: theory algorithms and software*; John Wiley & Sons, 2004.
21. Zhao, S.; Huang, B. On initialization of the Kalman filter. 2017 6th International Symposium on Advanced Control of Industrial Processes (AdCONIP). IEEE, 2017, pp. 565–570.
22. He, K.; Zhang, X.; Ren, S.; Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Proceedings of the IEEE international conference on computer vision, 2015, pp. 1026–1034.
23. Hobson, E.W. *The theory of spherical and ellipsoidal harmonics*; CUP Archive, 1931.