# An Adaptive Recurrent Neural-Network Controller using a Stabilization Matrix and Predictive Inputs to Solve a Tracking Problem under Disturbances[☆]

Michael Fairbank[a], Shuhui Li[b], Xingang Fu[b], Eduardo Alonso[a], Donald Wunsch[c]

[a]*School of Informatics, City University London, London EC1 V0B, UK.*
[b]*Department of Electrical & Computer Engineering, the University of Alabama, Tuscaloosa, AL 35487, USA.*
[c]*Department of Electrical & Computer Engineering, Missouri University of Science and Technology, Rolla, MO 65409-0040, USA.*

## Abstract

We present a recurrent neural-network (RNN) controller designed to solve the tracking problem for control systems. We demonstrate that a major difficulty in training any RNN is the problem of exploding gradients, and we propose a solution to this in the case of tracking problems, by introducing a stabilization matrix and by using carefully constrained context units. This solution allows us to achieve consistently lower training errors, and hence allows us to more easily introduce adaptive capabilities. The resulting RNN is one that has been trained off-line to be rapidly adaptive to changing plant conditions and changing tracking targets.

The case study we use is a renewable-energy generator application; that of producing an efficient controller for a three-phase grid-connected converter. The controller we produce can cope with random variation of system parameters and fluctuating grid voltages. It produces tracking control with almost instantaneous response to changing reference states, and virtually zero oscil-

[☆]This work was supported in part by the U.S. National Science Foundation under Grant EECS 1059265/1102159, the Mary K. Finley Missouri Endowment, and the Missouri S&T Intelligent Systems Center.

*Email addresses:* `michael.fairbank@virgin.net` (Michael Fairbank), `sli@eng.ua.edu` (Shuhui Li), `xfu@crimson.ua.edu` (Xingang Fu), `E.Alonso@city.ac.uk` (Eduardo Alonso), `dwunsch@mst.edu` (Donald Wunsch)

lation. This compares very favorably to the classical proportional integrator (PI) controllers, which we show produce a much slower response and settling time. In addition, the RNN we propose exhibits better learning stability and convergence properties, and can exhibit faster adaptation, than is achievable with adaptive critic designs.

*Keywords:* Tracking Problem, Stabilization Matrix, Recurrent Neural Networks, Exploding Gradients, Vector Control

## 1. Introduction

In this paper we propose a recurrent neural-network controller to solve the tracking problem. We consider a real-world test problem from electrical power and energy applications, and this forms the motivation for development of the neural-controller presented in this paper. The energy application we consider is that of a three-phase grid-connected dc/ac voltage-source converter, or grid-connected converter (GCC) for short.

A GCC is usually employed to interface between the dc and ac sides of an electric power system. Typical converter configurations containing a GCC include: 1) a dc/dc/ac converter for solar, battery and fuel cell applications (Figueres et al., 2009; Wang and Nehrir, 2007), 2) a dc/ac converter for STATCOM applications (Luo et al., 2009; Carrasco et al., 2006), and 3) an ac/dc/ac converter for wind power and HVDC applications (Carrasco et al., 2006; Xu and Wang, 2007; Mullane et al., 2005; Pena et al., 1996; Rabelo et al., 2009).

In all these applications, controlling the GCC efficiently and making it maintain a desired state (a tracking problem) is crucial for the reliability and stability of both the ac and the dc subsystems. The controller must be able to track any reference command variations quickly. For example, these might occur in wind power and photovoltaic applications as a result of sudden variations in the wind speed or solar irradiation levels.

Classically the tracking problem has been addressed using proportional integrator (PI) controllers (Qiao et al., 2009b; Pena et al., 1996). Limitations of these methods are that they can have slow response times to changing reference commands, can take considerable time to settle down from oscillating around the target reference state (Dannehl et al., 2009), and have difficulty recovering from short-circuit faults in either the generator or the power-grid. Hence neural-network based solutions have been proposed to overcome these difficulties, in this control problem and related ones (Qiao et al., 2008b, 2009a; Li et al., 2012; Venayagamoorthy et al., 2002; Park et al., 2004; Qiao et al., 2008a; Venayagamoorthy et al., 2003).

These neural-network approaches have mainly been based on Adaptive Critic Designs (ACDs) (Wang et al., 2009; Prokhorov and Wunsch, 1997; Werbos, 1992). ACDs use two neural networks: an action network and a critic network. The critic network provides feedback to the action network, allowing the action network to be trained on-line and in real-time, and therefore to be continually learning and adaptive during plant operation. However useful this double network design may be, proving convergence of the two continually learning networks at once is challenging. In fact, just proving the convergence of the critic network on its own is not trivial, since critic learning algorithms generally are not true gradient descent (Barnard, 1993). The general instability in this case is proven by Werbos (1998), and divergence examples of concurrent actor-critic learning exist (Fairbank and Alonso, 2012). In practice, the best course of action is not to allow such a system to be continually autonomously learning while controlling a delicate or critical industrial system. Qiao et al. (2008b, 2009a) and Venayagamoorthy et al. (2003) overcome this problem by first training the action and critic networks concurrently off-line, and then freezing the action neural network and dispensing with the critic network for on-line operation of the plant. This solution of course neutralizes the adaptive benefits of the ACD architecture. Adaptive behavior is often recreated by using lagged state inputs for the action network (e.g. Venayagamoorthy et al., 2003), effectively creating a time-delay neural network. Modest improvements over PI controllers are made using ACDs, for example, see Qiao et al. (2008b, 2009a).

To improve on this situation further, we are using an architecture that uses an action network only, but which is trained off-line through backpropagation through time (BPTT) (Werbos, 1990). This approach has the advantage that the learning algorithm is true gradient descent on the cost-to-go function, and so convergence is assured (assuming a smooth error minimization surface, and a sufficiently small learning rate). Also, since the BPTT algorithm is true gradient descent, learning is guaranteed to find a true local minimum of the training error. In contrast, the ACD learning algorithms used by the aformentioned references are not true gradient descent, and hence the learning progress appears stochastic, and the minimum obtained is often not as low as that obtained by BPTT.

Recent studies show how a single action network can be trained with

BPTT to control a GCC under fixed plant behavior (Li et al., 2012). However, for real-life applications, the plant behavior can change; system parameters can exhibit random variations; voltages coming into the system from the power grid can fluctuate; short circuits can occur. Hence the action network needs to become more adaptive than demonstrated by Li et al. (2012).

Adaptive behavior can be enabled by modifying the action network to have neural-context units which respond to the changing behavior of the plant, thus making the action network into a RNN. This design for adaptation is potentially much faster than the adaptation carried out by ACDs, in that the weights of the RNN do not need to change to accommodate adaptation. This is referred to as fixed-weight adaptive behavior by Prokhorov et al. (2002), and can produce almost instantaneous adaptation. In contrast, ACD adaptive behavior takes place by retraining the two neural networks involved, and this kind of learning is slow.

A major difficulty with using a RNN for the controller is that because data cycles around the RNN many times, learning gradients may decay rapidly to zero, or alternatively, the learning gradients may rapidly become excessively large, and both of these problems cause difficulties for learning by gradient descent. These problems are known as "vanishing" or "exploding" gradients, respectively, in the RNN literature (Hochreiter and Schmidhuber, 1997). While Hochreiter and Schmidhuber (1997) address the problem of vanishing gradients, our paper attempts to minimize the problem of exploding gradients for the tracking problem domain, through the introduction of a "stabilization matrix", and carefully constrained context units.

The novelties of this paper include: 1) the stabilizing matrix, which is a hand-picked neural weight matrix which represents some pre-learned basic control behavior, allowing the learning algorithm to concentrate on learning the more advanced nuances of behavior and thus to acquire improved solutions than otherwise possible, 2) a theoretical discussion on the importance of handling the problem of exploding gradients in RNNs, and 3) a design for using predicted as well as previous inputs that allows the neural network to behave adaptively on-line, despite the training process having taken place entirely off-line.

The rest of the paper is structured as follows: the basic topology of the GCC neural-network vector controller, and how to train it to solve the tracking problem using BPTT, is presented in Section 2. Section 3 shows the stabilization matrix approach, which enhances neural-network training speed and stability when the system matrix and the control voltage matrix are fixed. Section 4 presents how a RNN is trained to behave adaptively on-line when these matrices vary, which relies upon novel extra context inputs to the neural controller. **Simulation experiments** are given in Section 5. These include GCC experiments for the neural vector controller, under variable and dynamic conditions, and a comparison to two conventional control methods, showing the advantages of our method. Also an experiment is included that demonstrates how the stabilization-matrix method can be extended to the case of non-invertible matrices. The paper concludes in Section 6 with a summary and a discussion of further work, and Appendix A which proves that the method for adaptation which we used is flexible enough to work in a greater variety of applications than just our chosen experiments.

## 2. Neural-Network Vector-Control Architecture

Fig. 1 shows schematics of the GCC, in which a dc-link capacitor appears on the left, and a three-phase voltage source, representing the voltage at the Point of Common Coupling (PCC) of the ac system, appears on the right. In this diagram the capacitor would be connected to the electrical generator (for example the wind turbine, or photovoltaic array) and has a dc voltage represented by $V_{dc}$, and the three voltages $v_a$, $v_b$ and $v_c$ would represent the three-phase voltage of the electric power grid.

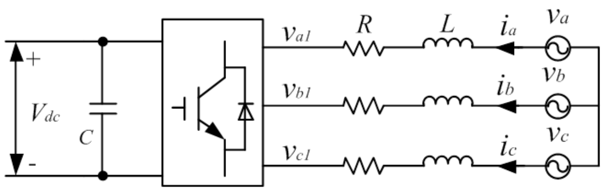


Figure 1: Grid-connected converter schematic.

The powers transferred between the grid and the converter include active power and reactive power. The purpose of the GCC controller is to control the active and reactive power transferred between the grid and the GCC.

The circuit contains 3-phase ac-voltages $v_a$, $v_b$, and $v_c$, with corresponding 3-phase ac-currents $i_a$, $i_b$ and $i_c$. By transforming to a rotating frame of reference with axes $d$ and $q$, as described by Li et al. (2011), it is possible to largely eliminate the ac-sinusoidal variations, and to transform these three dimensions down to just two, i.e. to currents $i_d$ and $i_q$, and voltages $v_d$ and

$v_q$. In this simpler d-q reference frame, the voltages and currents evolve according to the following 2-dimensional vector differential equation (see Li et al. (2011, 2012) for further derivation):

$$\begin{bmatrix} v_d \\ v_q \end{bmatrix} = R \begin{bmatrix} i_d \\ i_q \end{bmatrix} + L\frac{\partial}{\partial t} \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \omega_s L \begin{bmatrix} -i_q \\ i_d \end{bmatrix} + \begin{bmatrix} v_{d1} \\ v_{q1} \end{bmatrix} \tag{1}$$

Here $\omega_s$ is the angular frequency of the PCC voltage (i.e. the angular frequency of the rotating dq reference frame, which is also the angular frequency of the ac grid), and $L$ and $R$ are the inductance and resistance of the grid filter. $v_{d1}$ and $v_{q1}$ are the "control voltages" which are added into the system, in the dq reference frame, through the GCC controller. The purpose of these control voltages is to influence the current transferred between the grid and the GCC, a process referred to as *vector control.* Fig. 2 shows schematics for the GCC controller, with a neural network to make the control decisions, plus circuits to transform both to and from the dq reference frame, as necessary, and a pulse-width-modulation (PWM) scheme to convert the neural-network control decisions to the high voltage control signals in the system. **In the figure, $\theta_e$ is the instantaneous rotating angle of the three-phase grid voltage in the space vector domain.**

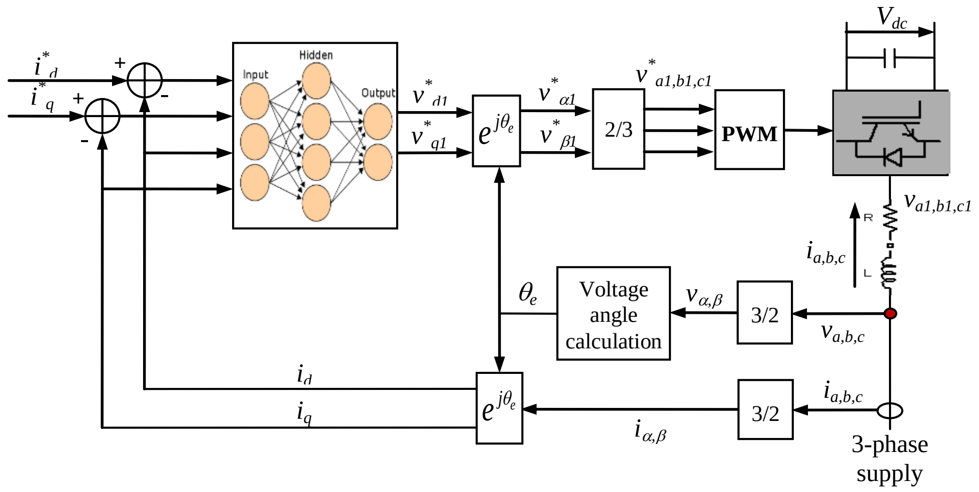


Figure 2: GCC neural vector-control structure.

From now on we will just consider the voltage and current variables in the simpler dq reference frame. The job of the GCC neural-controller is to output control voltages $v_{d1}$ and $v_{q1}$ which will make the actual dq-axis currents, $i_d$ and $i_q$ match as closely as possible some externally set reference currents, $i_d^*$ and $i_q^*$.

Since the state variables are $i_d$ and $i_q$, Eq. (1) can be rearranged to give:

$$\frac{\partial}{\partial t}\begin{bmatrix} i_d \\ i_q \end{bmatrix} = -\begin{bmatrix} R/L & -\omega_s \\ \omega_s & R/L \end{bmatrix}\begin{bmatrix} i_d \\ i_q \end{bmatrix} - \frac{1}{L}\begin{bmatrix} v_{d1} - v_d \\ v_{q1} - v_q \end{bmatrix} \tag{2}$$

For digital control implementations, a discrete-time equation is required:

$$\begin{bmatrix} i_d(kT_s + T_s) \\ i_q(kT_s + T_s) \end{bmatrix} = \mathbf{A}\begin{bmatrix} i_d(kT_s) \\ i_q(kT_s) \end{bmatrix} + \mathbf{B}\begin{bmatrix} v_{d1}(kT_s) - v_d \\ v_{q1}(kT_s) - v_q \end{bmatrix} \tag{3}$$

where $k$ is an integer time step, $\mathbf{A}$ is the system matrix, and $\mathbf{B}$ is the matrix associated with the control voltage. Our experiments used a zero-order-hold mechanism (Franklin et al., 1998) with sampling time $T_s = 0.001$ to obtain the discrete-time equation (Eq. (3)) from the continuous-time one (Eq. (2)). This means the components of the matrices $\mathbf{A}$ and $\mathbf{B}$ are only available numerically.

Using conventional control-system notation, we define the state vector to be $\vec{x} := \left[\begin{smallmatrix} i_d \\ i_q \end{smallmatrix}\right]$, and the control vector to be $\vec{u} := \left[\begin{smallmatrix} v_{d1} \\ v_{q1} \end{smallmatrix}\right]$ (the "control voltages"), and $\vec{c}$ to be a system vector (the "grid voltage") defined by $\vec{c} := \left[\begin{smallmatrix} v_d \\ v_q \end{smallmatrix}\right]$. Then, the system state evolution Eq. (3) can be written in a more concise form:

$$\vec{x}_{k+1} = \mathbf{A}\vec{x}_k + \mathbf{B}(\vec{u}_k - \vec{c}). \tag{4}$$

### *2.1. A Basic NN Controller and Optimization Function*

The tracking objective is to make the actual current state, $\vec{x}_k$, track the given (possibly moving) target state $\vec{x}_k^* := \left[\begin{smallmatrix} i_d^* \\ i_q^* \end{smallmatrix}\right]$. Actions are chosen by

$$\vec{u}_k = k_{pwm}\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{w}), \tag{5}$$

where the function $\pi(\cdot)$ is a fully connected multi-layer perceptron (MLP, (Bishop, 1995)) with dimensions 4-6-6-2, i.e. four inputs, two hidden layers of six nodes each, and two output nodes, and where $\vec{w}$ is a vector of all of the weights in the neural network. The MLP contained shortcut connections between all pairs of layers. All nodes had a bias weight, and a hyperbolic tangent activation function. The input vectors were rescaled to be $\tanh(\vec{x}_k/1000)$ and $\tanh((\vec{x}^*_k - \vec{x}_k)/100)$, respectively. We define the function $\pi(\cdot)$ to include these input rescalings.

$k_{pwm} = V_{dc}/2$ is a scalar constant, referred to as the gain of the voltage source dc/ac PWM converter (Mohan et al., 2002). From a neural-network point of view, $k_{pwm}$ can be understood simply to be a constant for rescaling the neural-network output. This will ensure each component of the action vector, $u^i$, lies in the following range:

$$u^i \in [-k_{pwm}, k_{pwm}]. \tag{6}$$

We train the weights of the action network to solve the tracking problem by doing gradient descent with respect to $\vec{w}$ on the following cost function:

$$J(\vec{x}_0, \vec{w}) := \sum_{k=0}^{K-1} \gamma^k U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k) \tag{7}$$

where

$$U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k) := |\vec{x}_k - \vec{x}^*_k|^m \tag{8}$$

and $m$ is some constant power (we used $m = 1$ in our experiments), $|\cdot|$ denotes the modulus of a vector, and $\gamma \in [0, 1]$ is the constant discount factor (we used $\gamma = 1$ throughout). The trajectory duration used for training was $K = 1000$ time steps (i.e. 1 second of real time). Lines 1-7 of Alg. 1 illustrate how Eqs. (4)-(8) can be used to simulate a single trajectory and calculate its total cost, $J$.

To train the action network, we used BPTT, as described in the following subsection.

## *2.2. Backpropagation Through Time Algorithm*

The action network was trained to minimize the cost in Eq. (7) by using BPTT (Werbos, 1990). BPTT efficiently calculates the gradient of $J(\vec{x}_0, \vec{w})$ with respect to the weight vector of the action network, $\vec{w}$, for a given trajectory with arbitrary initial state $\vec{x}_0$. This gradient can thus be used to optimize the vector-control strategy, for example by using gradient descent. In general, the BPTT algorithm consists of two steps: a forward pass which unrolls a trajectory, followed by a backward pass along the whole trajectory,

**Algorithm 1** BPTT for tracking control problem, with fixed $\mathbf{A}$ and $\mathbf{B}$ matrices

```
 1: $J \leftarrow 0$
 2: {Unroll a full trajectory:}
 3: **for** $k = 0$ to $K - 1$ **do**
 4:     $\vec{u}_k \leftarrow k_{pwm}\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{w})$ {Neural-network output}
 5:     $\vec{x}_{k+1} \leftarrow \mathbf{A}\vec{x}_k + \mathbf{B}(\vec{u}_k - \vec{c})$ {Calculate next state}
 6:     $J \leftarrow J + \gamma^k U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)$
 7: **end for**
 8: {Backwards pass along trajectory:}
 9: $J\_\vec{w} \leftarrow \vec{0}$
10: $J\_\vec{x}_K \leftarrow \vec{0}$
11: **for** $k = K - 1$ to $0$ step $-1$ **do**
12:     $J\_\vec{u}_k \leftarrow (\mathbf{B}^T)J\_\vec{x}_{k+1} + \gamma^k \left(\frac{\partial U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)}{\partial \vec{u}_k}\right)$
13:     $J\_\vec{x}_k \leftarrow k_{pwm}\left(\frac{d\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{w})}{d\vec{x}_k}\right) J\_\vec{u}_k + (\mathbf{A}^T)J\_\vec{x}_{k+1} + \gamma^k \left(\frac{\partial U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)}{\partial \vec{x}_k}\right)$
14:     $J\_\vec{w} \leftarrow J\_\vec{w} + k_{pwm}\left(\frac{\partial \pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{w})}{\partial \vec{w}}\right) J\_\vec{u}_k$
15: **end for**
16: {On exit, $J\_\vec{w}$ holds $\frac{\partial J}{\partial \vec{w}}$ for the whole trajectory.}
```

which accumulates the gradient-descent derivative. Alg. 1 gives pseudocode for both stages of this process.

The second half of the algorithm calculates the desired gradient, $\frac{\partial J}{\partial \vec{w}}$. In this code, the variables $J\_\vec{x}_k$, $J\_\vec{u}_k$ and $J\_\vec{w}$ are workspace column vectors of dimension 2, 2, and $\dim(\vec{w})$, respectively. These variables hold the "ordered partial derivatives" of $J$ with respect to the given variable name, so that for example $J\_\vec{x}_k = \frac{\partial^+ J}{\partial \vec{x}_k}$. This ordered partial derivative, as defined by Werbos (Werbos, 1990; Werbos et al., 1992), represents the derivative of $J$ with respect to $\vec{x}_k$, assuming all other variables which depend upon $\vec{x}_k$ in lines 3-7 of Alg. 1 are not fixed, and thus their derivatives will influence the value of $\frac{\partial^+ J}{\partial \vec{x}_k}$ via the chain rule. The derivation of the gradient computation part of the algorithm (lines 8-16) is exact, using the method known as generalized backpropagation (Werbos, 1990), or automatic-differentiation (Werbos, 2005; Rall, 1981). **For full details of this derivation, see Werbos et al. (1992).**

The vector and matrix notation is such that all vectors are columns and differentiation of a scalar by a vector gives a column. Differentiation of a vector function by a vector argument gives a matrix defined by the transpose of the usual Jacobian notation, such that for example, $\left(\frac{\partial \pi}{\partial \vec{w}}\right)^{ij} := \frac{\partial \pi^j}{\partial \vec{w}^i}$.

The algorithm refers to the derivatives $\frac{\partial \pi}{\partial \vec{x}}$ and $\frac{\partial \pi}{\partial \vec{w}}$. These would be calculated by ordinary neural-network backpropagation. These derivatives are only required as an inner product with the vector $J\_\vec{u}_k$, and thus each evaluation of these derivatives could be performed in asymptotic time $\mathrm{O}\big(\dim(\vec{w})\big)$. These neural-network backpropagation calculations should be considered as a sub-module necessary to implement Alg.1, and should not be confused with the BPTT backpropagation itself, which is what the rest of Alg. 1 implements. Therefore the asymptotic running time for the whole BPTT algorithm applied to a full trajectory with $K$ time steps will be $\mathrm{O}\big(K \dim(\vec{w})\big)$.

The algorithm also refers to the derivatives $\frac{\partial U}{\partial \vec{x}}$ and $\frac{\partial U}{\partial \vec{u}}$. By differentiating Eq. (8) directly, these are given by

$$\frac{\partial U}{\partial \vec{x}_k} = m(\vec{x}_k - \vec{x}^*_k)U^{1-2/m}$$

and,

$$\frac{\partial U}{\partial \vec{u}_k} = \vec{0}.$$

## *2.3. Training the Neural Controller*

We first trained the network to control the plant in a situation where the **A** and **B** matrices were fixed. To choose these constants, we employed a standard arrangement for the integrated GCC and grid system, as used in renewable energy conversion system applications (Mullane et al., 2005; Pena et al., 1996; Li et al., 2011). These include 1) a three-phase 60Hz, 690V voltage source signifying the grid (i.e. $\omega_s = 120\pi$, $\begin{bmatrix} v_d \\ v_q \end{bmatrix} = \begin{bmatrix} 690\text{V} \\ 0\text{V} \end{bmatrix}$), 2) a reference voltage of 1200V for $V_{dc}$, and 3) a normal resistance of $R = 0.012\Omega$ and a normal inductance $L = 0.002\text{H}$ for the grid filter. These constants are used to calculate the fixed **A** and **B** matrices in Eq. (3).

For training purposes, the reference current $\vec{x}^* := \begin{bmatrix} i_d^* \\ i_q^* \end{bmatrix}$ was changed every 0.05s (i.e. every 50 time steps), within the range $[-500\text{A}, 500\text{A}]$, in a fixed pattern. Fig. 3 shows this reference-current trajectory used for training. The objective of training is to make the actual trajectory match this reference trajectory, by minimization of Eq. (7). The actual trajectory was made to start from a fixed point, chosen arbitrarily to be $\vec{x} = \begin{bmatrix} 120 \\ 10 \end{bmatrix}$.

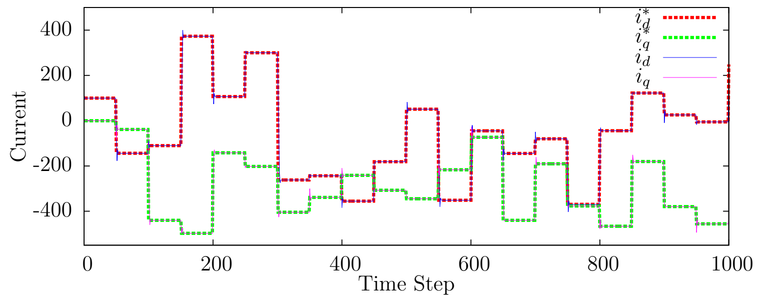


Figure 3: Training data and neural-network output in the GCC tracking problem. The dotted lines indicate the reference currents that were used during training, and how these varied over the one-second training trajectory. The solid lines show the performance of the trained neural network (trained using a stabilization matrix, as described in Sec. 3.2) in controlling the actual currents, which lie very closely on top of the reference current curve, indicating that the tracking performance is good.

During training, the total gradient $\frac{\partial J}{\partial \vec{w}}$ was accumulated over the full trajectory length, using Alg. 1. This gradient was accelerated by RPROP (Riedmiller and Braun, 1993) before each weight update was finally made. After 800 iterations, the average trajectory cost per time step along the whole trajectory was calculated and noted in Table 1, in the first row (without stabilization matrix). This was repeated for 10 different experiments. Each experiment started with a different initial neural-weights randomization in the range $[-0.1, 0.1]$.

Table 1: Results with and without the "stabilization matrix", with fixed known **A** and **B** matrices.

| Results showing different $J$ values for different learning trials from 10 different initial-weight randomizations. | | | | | |
|---|---|---|---|---|---|
| Without stabilization matrix | 15.99 | 20.56 | 14.38 | 17.86 | 14.75 |
| | 14.34 | 11.87 | 14.01 | 11.92 | 14.43 |
| With stabilization matrix ignoring constraint (6) | 9.49 | 9.07 | 526.05 | 9.51 | 10.19 |
| | 9.38 | 9.63 | 10.18 | 9.83 | 9.33 |
| With stabilization matrix and truncated actions by (6) | 10.23 | 9.96 | 1183.4 | 10.48 | 11.00 |
| | 10.16 | 10.38 | 10.95 | 10.74 | 10.20 |

The neural controllers obtained by the best results in the first row of Table 1 replicate the neural controller performance described by Li et al. (2012), which can outperform PI and ACD methods in tracking ability (as we will show in the experiments of Section 5.1). As can be seen in the table, the results are not as good as when the stabilization-matrix method, described in the next section, is used.

## 3. Training Neural Network with a Stabilization Matrix

As indicated in Section 1, the control of a GCC normally faces the challenges of rapidly changing target states, random variation of system parameters, and oscillation of grid voltage. These issues cause a difficulties in training the action network to meet a variety of GCC control requirements. In this section we consider RNN action networks, and look at making more robust RNN controllers by trying to solve the problem of "exploding gradients", through the introduction of the stabilization matrix.

### *3.1. Why is training the action network so hard?*

Training the action network is difficult because every time a component of the weight vector $\vec{w}$ is changed, the actions chosen by Eq. (5) will change at every time step. However what makes things even more challenging than this

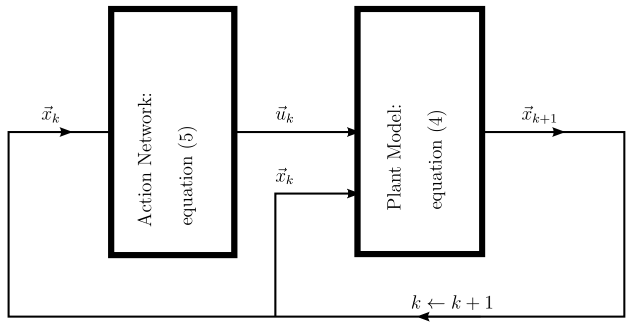


Figure 4: Power plant recurrence

is that each changed action ($\Delta\vec{u}_k$) will consequently change the next state ($\Delta\vec{x}_{k+1}$) the system passes through, by Eq. (4). And each changed state will further change the next action chosen, $\Delta\vec{u}_{k+1}$, by Eq. (5). Clearly this creates an on-going cascade of changes. Hence changing one component of $\vec{w}$, even by the tiniest finite amount, can completely scramble the trajectory generated by Eqs. (4)-(5). This feedback is shown in Fig. 4. This difficulty is analogous to one of the major difficulties in training RNNs, since the model Eq. (4) can be interpreted to be just another layer of a neural network. In that case Fig. 4 becomes identical to the schematics of a RNN.

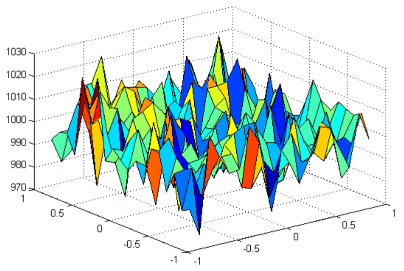
Consequently the function $J(\vec{x},\vec{w})$ can be over-sensitive to tiny changes in $\vec{w}$, chaotic even. In other words, the surface of the function in $\vec{w}$-space can be extremely crinkly, as illustrated by Fig. 5(a). Gradient-descent methods find it hard to traverse such a rugged surface, and hence it is difficult to train the action network effectively. This is one major reason why training the action network, or any recurrent network, is hard. This is referred to as the problem of "exploding" gradients by Hochreiter and Schmidhuber (1997).

Our approach of using a stabilization matrix attempts to smooth out the surface of Fig. 5(a).

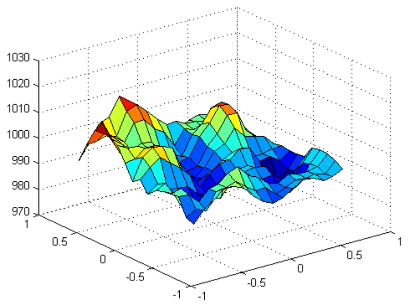
### *3.2. Using the stabilization matrix*

Fig. 6 shows the way the state vector would evolve if the action $\vec{u} \equiv \vec{0}$ and if $\vec{c} \equiv \vec{0}$, i.e. if the action network chose completely passive actions. In this case the state vector would drift around the state space like a cork floating on an ocean current.

To solve the tracking problem, the task of the action network can be split into two stages: Firstly to fight against moving with the arrows in Fig. 6,



(a) An error surface that is difficult for gradient-descent algorithms.



(b) A smooth error surface that is easy for gradient-descent algorithms.

Figure 5: Types of error surfaces commonly encountered. In our problem this is the surface of the function $J(\vec{x}, \vec{w})$ in $\vec{w}$ space.

which will most likely take the state away from the target state $\vec{x}^*$. Then, secondly, to actively head towards the tracking target point $\vec{x}^*$. The idea of the stabilization matrix is to make the first of these two objectives automatic. This should make the action network's task much simpler. The presence of the stability matrix should make the arrows in Fig. 6 vanish.

To achieve this, we first find the fixed point of Eq. (4) with respect to the control action by:

$$\begin{aligned}\vec{x} &= \mathbf{A}\vec{x} + \mathbf{B}(\vec{u} - \vec{c})\\ \vec{0} &= (\mathbf{A} - \mathbf{I})\vec{x} + \mathbf{B}(\vec{u} - \vec{c})\\ \vec{u} - \vec{c} &= -\mathbf{B}^{-1}(\mathbf{A} - \mathbf{I})\vec{x}\\ \vec{u} &= -\mathbf{B}^{-1}(\mathbf{A} - \mathbf{I})\vec{x} + \vec{c}\end{aligned}$$

where $\mathbf{I}$ is the identity matrix.

Choosing this action will keep the plant in exactly the same state. Hence we change the action chosen by the action network from Eq. (5) into Eq. (9):

$$\vec{u}_k = k_{pwm}\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{w}) + \mathbf{W_0}\vec{x}_k + \vec{c} \tag{9}$$

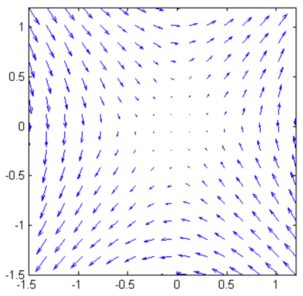


Figure 6: The motion which would occur from the equation $\frac{\partial \vec{x}}{\partial k} = \mathbf{A}\vec{x} + \mathbf{B}(\vec{u} - \vec{c})$ if $\vec{u} - \vec{c} \equiv \vec{0}$. In this case the motion simplifies to $\frac{\partial \vec{x}}{\partial k} = \mathbf{A}\vec{x}$.

where

$$\mathbf{W_0} = -\mathbf{B}^{-1}(\mathbf{A} - \mathbf{I}) \tag{10}$$

is the constant "stabilization matrix", and $\pi(\vec{x}, \vec{x}^* - \vec{x}, \vec{w})$ is the output of the neural network. $\mathbf{W_0}$ acts like an extra weight matrix in the neural network that connects the input layer directly to the output layer. By the phrase "the stabilization matrix", we mean both the $\mathbf{W_0}$ and $\vec{c}$ terms in Eq. (9). This combination can be justified since $\vec{c}$ could be included into the stabilization matrix as a bias term, as is conventional in neural networks.

The stabilization matrix effectively is a hand-picked weight matrix which helps the neural network do its job more easily. It aims to give the controller a default behavior of being able to hold the system-state steady. This should minimize the on-going cascade of changes that was described in the previous subsection as making training the action network difficult. Hence the stabilization matrix aims to smooth out the crinkles of the $J(\vec{x}, \vec{w})$ surface; making the surface more like Fig.5(b) than Fig. 5(a).

Results for the stabilization-matrix method are shown in Table 1. The results corresponding to the middle row of this table are displayed graphically as the solid curves in Fig. 3.

When training with the stabilization matrix, we had to ignore the constraint of Eq. (6) since the stabilized actions (9) naturally have a different range. We couldn't simply force the constraint (6) through truncation, since truncation would make the learning gradients vanish. The second row of Table 1 shows these results when ignoring the constraint of Eq. (6). However in the actual plant, the actions must obey that constraint, so the trajectory costs using this constraint are given in the third row of the table. In this case, the same fully-trained neural networks created for row 2 of the table were used again, but in reporting the final trajectory costs, the trajectories were generated while enforcing Eq. (6) through direct truncation of each component of $\vec{u}_k$.

Comparing the results using the stabilization matrix in the third row of the table, to the results without the stabilization matrix (given in the first row of the same table) we can see that using the stabilization matrix has produced consistently lower $J$ values (ignoring the single extreme outlier). This is thought to be for the reasons given above. Further results also showing the effectiveness of the stabilization matrix are presented in Section 4.2.

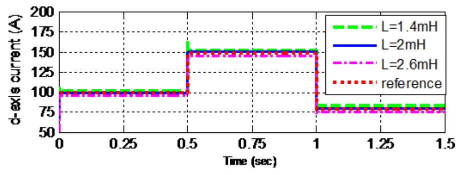
The stabilization-matrix method has improved learning performance, and it was designed to do this by making the optimization surface smoother and therefore easier to navigate. Of course more sophisticated learning optimizers might be able to navigate a crinkly surface better than RPROP did (such as the multi-stream extended Kalman filter algorithm (Feldkamp and Puskorius, 1998), or conjugate gradient descent), but it would be expected that the stabilization-matrix method would assist these second-order algorithms to achieve better RNNs than they otherwise would.

## 4. Producing Adaptive Behaviour: Learning off-line to be adaptive on-line

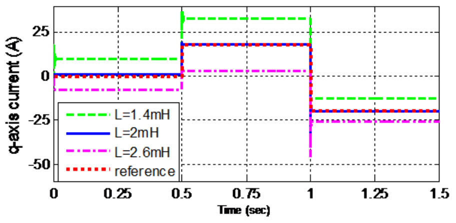
The previous results are valid for a fixed inductance constant, $L$, and fixed grid voltage $\vec{c}$. However, when $L$ or $\vec{c}$ vary, the **A** and **B** matrices in Eq. (4) will also change, and so the fixed behavior learned by the action network will no longer be optimal, and not solve the tracking problem correctly. Fig. 7 shows the resulting constant tracking error in several cases when $L$ differs from the $L = 2$mH value used in training. In this section we describe how to remedy this, by creating a neural network that can adapt to changing unknown **A** and **B** matrices.

### *4.1. Adaptive control with variable L*

We note that $L$ is not a variable that can be read– it is unknown to the controller, as are the matrices **A** and **B** (since they depend on $L$ in Eq. (2)), but we want the controller to adapt to such circumstances and to optimize the plant behavior nonetheless. With this aim in mind, we propose to train one flexible action network that can adapt in real time without any need for



(a) d-axis current



(b) q-axis current

Figure 7: Constant tracking-error problem. The curves are all meant to lie on top of the "reference" curve (the tracking target). However only the $L = 2$mH curve does this properly; the $L = 2.6$mH and $L = 1.4$mH curves show a constant tracking error, especially noticeable in the $q$-axis. This problem is caused by the action network's inability to adapt to the unknown and changing value of the constant $L$.

retraining, unlike traditional ACD methods which require continuous on-line training. Hence we are training the controller off-line to be adaptive on-line, following Prokhorov et al. (2002).

In this experiment, to train an adaptive controller, we regularly cycle the $L$ values through a sequence, such as $L = 2$mH, $L = 2.6$mH, and $L = 1.4$mH, changing every 0.1 seconds, so the controller can learn to adapt to a standard range of conditions. Since the $L$ value depends on the time step $k$, we can denote the time dependent $L$ value as $L_k$, and the corresponding time dependent $\mathbf{A}$ and $\mathbf{B}$ matrix values as $\mathbf{A}_k$ and $\mathbf{B}_k$.

The adaptive behavior tracking problem would easily be solved if we were allowed to define the action network as $\pi(\vec{x}_k, \vec{x}^*_k, L_k, \vec{w})$, but the value of $L_k$ is hard to observe in reality and thus the neural network must deduce it for itself. In addition, we must not change the $\mathbf{W_0}$ stabilization matrix depending on $L_k$; since $L_k$ is unknown to the neural controller, $\mathbf{W_0}$ must remain constant.

Summarizing, when we switch to variable $L_k$, the problem of controlling the neural-network output function $\pi(\vec{x}, \vec{x}^* - \vec{x}, \vec{w})$ becomes significantly harder, as can be seen in the first row of Table 2. Comparing these results to those previously shown for the equivalent non-adaptive problem (in the second row of Table 1), we see that they are much worse. This worsening of results manifests itself an inability to control the plant as intended, as shown by a constant tracking error in Fig. 7.

Table 2: Results for adaptive behavior controllers. All are using the fixed stabilization matrix, ignoring constraint (6), with changing unknown $\mathbf{A}_k$ and $\mathbf{B}_k$ matrices.

| Results showing different $J$ values for different learning trials from 10 different initial weight randomizations. | | | | | |
|---|---|---|---|---|---|
| With no adaptive behavior | 28.14 | 25.40 | 28.31 | 28.39 | 28.24 |
| $\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{w})$ | 28.53 | 421.08 | 27.33 | 25.10 | 27.29 |
| With extra input $\vec{x}_k - \hat{x}_k$ | 17.04 | 17.90 | 16.21 | 20.45 | 22.52 |
| | 14.45 | 13.96 | 17.35 | 14.32 | 15.75 |
| With extra inputs $\vec{x}_k - \hat{x}_k$ | 13.52 | 12.03 | 12.14 | 13.14 | 11.78 |
| and $\vec{s}_k$ | 13.18 | 13.46 | 12.77 | 705.66 | 13.33 |

To make the neural network become adaptive, it needs to have some idea on how the actual plant behavior is differing from its expected behavior, so that the controller can recalibrate its behavior intelligently during run time, and try to eliminate the constant tracking error shown in Fig. 7. For example, if we consider the situation of the controller representing a marksman shooting at a target under strong cross-wind conditions, then by observing the deflection the wind causes to the first shot, the marksman can make a compensatory adjustment to the subsequent shooting angle to try to cancel out the effect of the wind. Hence we give the neural network an extra input $\vec{x}_k - \hat{x}_k$ which reflects the difference between the actual current state ($\vec{x}_k$) and the predicted current state ($\hat{x}_k$) calculated with the fixed model, such that $\hat{x}_k$ is defined by:

$$\hat{x}_k := \bar{\mathbf{A}}\vec{x}_{k-1} + \bar{\mathbf{B}}(\vec{u}_{k-1} - \vec{c}), \tag{11}$$

where $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ are constant matrices of (4) chosen for the default $L = 2\text{mH}$ value. In the real plant, $L_k$ often differs from this default value, and so the $\mathbf{A}_k$ and $\mathbf{B}_k$ matrices will differ from $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$. Hence the difference $\vec{x}_k - \hat{x}_k$ will be non-zero, and it will give useful feedback for telling the controller how to adapt to the dynamically changing plant conditions.

Adding this extra input $\vec{x}_k - \hat{x}_k$ to the neural network changes the arguments of its function $\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{w})$ into $\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{w})$. This alteration causes the results to improve to those in the second row of Table

2, which show a big improvement over the previous row. In this case, and in the following experiments, the stabilization matrix is defined by the fixed $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ matrices, i.e. $\mathbf{W_0} := -\bar{\mathbf{B}}^{-1}(\bar{\mathbf{A}} - \mathbf{I})$, since the changing $\mathbf{A}_k$ and $\mathbf{B}_k$ matrices are not known to the controller. Also, the new input was rescaled into $\tanh\left(\frac{\vec{x}_k - \hat{x}_k}{1000}\right)$ and this extra vector input meant the neural-network architecture needed modifying to now have 6 input units.

Theorem 1 in Appendix A proves that adding this as an extra input is sufficient, in principle, to fix the constant tracking-error problem previously seen, under any mild unknown perturbations to the system and control matrices $\mathbf{A}$ and $\mathbf{B}$, and under various tight assumptions stated in the theorem. In practice good action-network performance holds under more liberal conditions than those listed in the theorem, as the results in this section and the experiments of Section 5 show. Furthermore, Theorem 2 in the appendix shows that it is not possible to solve the problem if this extra input is removed.

Finally, to enhance performance further, an extra neural input pair is used to describe the previous action taken, $\vec{u}_{k-1}$. This allows the network to observe whether this previous action led to the anticipated outcome in $\vec{x}_k - \hat{x}_k$, and if not, an appropriate adjustment could be made to the next action, $\vec{u}_k$, to compensate for this unexpected behavior. This is explicitly to handle the uncertainty in the $\mathbf{B}_k$ matrix. To ensure this new input was rescaled suitably, and also to avoid the scrambling effect of the stabilization matrix, we used the previous direct output of the neural-network function $\pi(\cdot)$, instead of the output of Eq. (9). Lines 4, 5 and 8 of Alg. 2 give the exact details of how this new input was calculated.

We stress that this input, which we denote $\vec{s}_k$, was not intended to act as a general purpose "context" unit such as is conventionally used in RNN architectures. Unlike a conventional context unit, our input $\vec{s}_k$ is heavily constrained in that it also determines the action vector which controls the power plant, so it cannot freely be used to retain arbitrary context information from one time step to the next. This design decision was made deliberately to constrain the way the RNN can generate feedback; the intention being to simplify the training process (since there are less possibilities of feedback mechanism to explore) and also to make the final controller more robust by being less likely to encounter a context that has never been seen before, possibly causing a malfunction at run-time (because the feedback space is reduced, this is less likely to happen). These constraints did not hinder our controller from producing very effective solutions, as the results in this section and the next show. If a more complicated plant was to be controlled then more elaborate feedback units could be introduced, as and when required.

Combining all of these new features, i.e. the two extra neural input pairs to allow adaptive behavior, plus the stabilization matrix, means a trajectory can now be generated by lines 1-10 of Alg. 2. Adding these features changes the results to those shown in the final row of Table 2, which shows a further improvement over the previous row (ignoring the single extreme outlier result). Of course, the performance under changing $L$ is always going to be a bit worse than with fixed $L$ (Table 1), since it is always going to take some time for the neural network to make enough observations of the plant's actual behavior to deduce the values of the changing hidden matrices $\mathbf{A}_k$ and $\mathbf{B}_k$. However, we show in Section 5.2 that the neural controller is able to make an almost instantaneous adaptation to the plant's changing conditions, and a virtual elimination of the constant tracking error previously shown.

The pseudocode in Alg. 2 gives the correct gradient calculation by BPTT, which was again generated using the method of Werbos et al. (1992). With both of the new extra neural input pairs, the final MLP has dimensions 8-6-6-2.

*4.2. Effectiveness of the stabilization matrix in the adaptive control*

All of the experiments in the previous subsection used the stabilization-matrix method to enhance neural training– without it the results deteriorate significantly, as can be seen in Table 3. This is further evidence that the crinkly error surface is being smoothed out by the stabilization matrix.

Table 3: Effect of the stabilizing matrix on the adaptive behavior controllers. All are using the extra inputs $\vec{x}_k - \hat{x}_k$ and $\vec{s}_k$, for the vector-control problem with changing unknown $\mathbf{A}_k$ and $\mathbf{B}_k$ matrices.

| Results showing different $J$ values for different learning trials from 10 different initial weight randomizations. | | | | | |
|---|---|---|---|---|---|
| With stabilizing matrix, | 14.48 | 13.61 | 13.81 | 13.97 | 13.32 |
| and action truncation by (6) | 14.28 | 14.38 | 14.10 | 1446.4 | 14.23 |
| Without stabilizing matrix | 35.46 | 39.32 | 39.96 | 52.47 | 65.92 |
| | 31.50 | 87.58 | 19.89 | 37.58 | 32.01 |

Of course the stabilization matrix $\mathbf{W_0}$ was only calculated for the constant $L = 2$mH value, so it would not provide full stabilization when $L$ took on a

**Algorithm 2** Enhanced BPTT for tracking control problem, with time varying $\mathbf{A}$ and $\mathbf{B}$ matrices, using a stabilization matrix and adaptive controller

```
1: $J \leftarrow 0$, $\hat{x}_0 \leftarrow \vec{x}_0$, $\vec{s}_0 \leftarrow \vec{0}$
2: {Unroll a full trajectory:}
3: for $k = 0$ to $K-1$ do
4:     $\vec{y}_k \leftarrow \pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{s}_k, \vec{w})$ {Neural-network output}
5:     $\vec{u}_k \leftarrow k_{pwm}\vec{y}_k + \mathbf{W_0}\vec{x}_k + \vec{c}$ {Stabilized control action}
6:     $\vec{x}_{k+1} \leftarrow \mathbf{A}_k\vec{x}_k + \mathbf{B}_k\left(\vec{u}_k - \vec{c}\right)$ {Next state, using the time dependent $\mathbf{A}_k$
       and $\mathbf{B}_k$ matrices}
7:     $\hat{x}_{k+1} \leftarrow \bar{\mathbf{A}}\vec{x}_k + \bar{\mathbf{B}}\left(\vec{u}_k - \vec{c}\right)$ {Predicted next state, according to the fixed
       $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ matrices}
8:     $\vec{s}_{k+1} \leftarrow \vec{y}_k$ {Previous network output}
9:     $J \leftarrow J + \gamma^k U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)$
10: end for
11: {Backwards pass along trajectory:}
12: $J\_\vec{w} \leftarrow \vec{0}$
13: $J\_\vec{x}_K \leftarrow \vec{0}$, $J\_\hat{x}_K \leftarrow \vec{0}$, $J\_\vec{s}_K \leftarrow \vec{0}$
14: for $k = K-1$ to $0$ step $-1$ do
15:     $J\_\vec{u}_k \leftarrow (\mathbf{B}_k)^T J\_\vec{x}_{k+1} + \bar{\mathbf{B}}^T J\_\hat{x}_{k+1} + \gamma^k \left(\frac{\partial U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)}{\partial \vec{u}_k}\right)$
16:     $J\_\vec{y}_k \leftarrow k_{pwm} J\_\vec{u}_k + J\_\vec{s}_{k+1}$
17:     $J\_\vec{x}_k \leftarrow \left(\frac{d\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{s}_k, \vec{w})}{d\vec{x}_k}\right) J\_\vec{y}_k + \mathbf{W_0}^T J\_\vec{u}_k + (\mathbf{A}_k)^T J\_\vec{x}_{k+1} +$
        $\bar{\mathbf{A}}^T J\_\hat{x}_{k+1} + \gamma^k \left(\frac{\partial U(\vec{x}_k, \vec{x}^*_k, \vec{u}_k)}{\partial \vec{x}_k}\right)$
18:     $J\_\hat{x}_k \leftarrow \left(\frac{\partial\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{s}_k, \vec{w})}{\partial \hat{x}_k}\right) J\_\vec{y}_k$
19:     $J\_\vec{s}_k \leftarrow \left(\frac{\partial\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{s}_k, \vec{w})}{\partial \vec{s}_k}\right) J\_\vec{y}_k$
20:     $J\_\vec{w} \leftarrow J\_\vec{w} + \left(\frac{\partial\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{s}_k, \vec{w})}{\partial \vec{w}}\right) J\_\vec{y}_k$
21: end for
22: {On exit, $J\_\vec{w}$ holds $\frac{\partial J}{\partial \vec{w}}$ for the whole trajectory.}
```

different value. But it seems that the stabilization matrix was still effective in improving results on this problem. It remains to be seen how different the matrices **A** and **B** can become before the stabilization matrix stops working.

Fig. 8. shows the average cost per trajectory time step for successful trainings of the action network corresponding to the four conditions shown in Table 2 and 3, i.e., without stabilization matrix (case 1), with stabilization matrix (case 2), with extra input $\vec{x}_k - \hat{x}_k$ (case 3), and with extra inputs $\vec{x}_k - \hat{x}_k$ and $\vec{s}_k$ (case 4). As the figure indicates, without the stabilization matrix, it is hard for the neural network to learn; but the overall average cost dropped to a small number very quickly with the stabilization matrix and the extra inputs $\vec{x}_k - \hat{x}_k$ and $\vec{s}_k$.

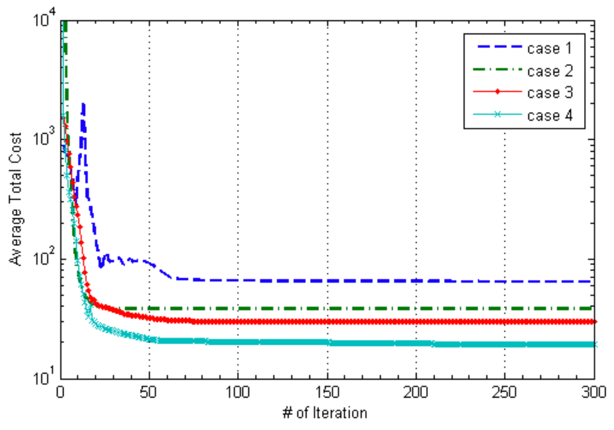


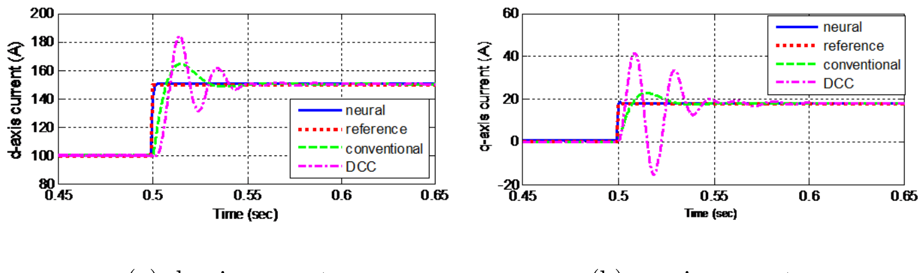
Figure 8: Average cost per trajectory time step for training neural controller

## 5. Simulation Experiments

The simulation experiments conducted so far in this paper have shown a systematic development of the stabilization-matrix method, and of the neural inputs used for adaptive behaviour. These experiments were all conducted on the training data-set, i.e. the fixed reference target curve given in Fig. 3. In this section we provide further simulation experiments for the GCC controller. These are based on the final neural network developed in Section 4, and applied the action constraint of Eq. (6), and these experiments operate out-of-sample from the training data. Also a non-GCC controller experiment is included which demonstrates how the stabilization matrix can also be used when the **B** matrix is non-invertible.

## *5.1. Comparison of Neural Controller with Conventional Standard and DCC Vector-Control Mechanisms*

In this comparison study, the performance of the neural controller is compared against standard PI controllers, under the same plant conditions as described in Section 2.3 (i.e. no adaptation to changing plant behavior was required). Fig. 9 shows the results. One of the result curves is for the neural controller, another is for a tuned PI conventional GCC controller (Luo et al., 2009; Carrasco et al., 2006; Dannehl et al., 2009), and a third is for a "direct-current vector control" structure (DCC, (Li et al., 2010, 2011)). The PI controllers were tuned until the controller performance was acceptable (Li et al., 2011). The reference currents are given a step-change at 0.5s into the experiment, as shown in the figure.



(a) d-axis current (b) q-axis current

Figure 9: Comparison of two PI controllers ("conventional", and "DCC") with the neural vector controller.

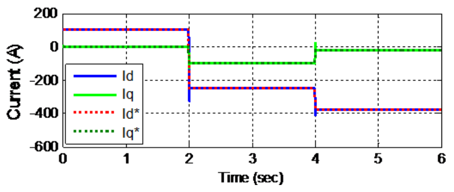
The figure indicates that among the three vector-control strategies, the neural controller has the fastest response time, low overshoot, and best performance in tracking the changing reference current. The contrast between the tracking ability of the neural controller and the PI controllers is striking, with the neural controller solving the tracking problem almost perfectly, whereas the PI controllers oscillate around the reference current noticeably before settling down. This contrasts to published results for ACD designs, where ACDs only make a marginal improvement over PIs (for example see the results graphs of Qiao et al. (2008b, 2009a)).

## *5.2. Control Evaluation under Variable Parameters of GCC System*

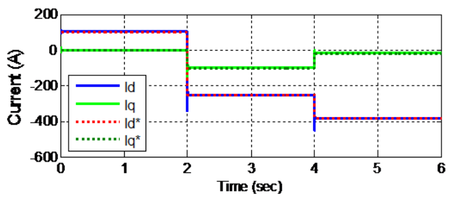
The previous experiment was for fixed $\mathbf{A}$ and $\mathbf{B}$ matrices. We now see how the controller can adapt when these matrices drift from their original values. Fig. 10 compares how the neural controller is affected when there is an increase or a decrease of inductance $L$ by 30% from its nominal value. For all the cases, a disturbance voltage $\left[\begin{smallmatrix}\Delta v_d \\ \Delta v_q\end{smallmatrix}\right]$ is also added to the grid voltage, and Eq. (9) is used to compute the final control voltage applied to the system.

The results of Fig. 10 show a virtual elimination of the constant tracking error previously seen in Fig. 7, and the neural controller successfully making almost-instantaneous adaptation (with almost zero rise time and settling time) to the changing $L$, $\mathbf{A}_k$ and $\mathbf{B}_k$ system variables, with an adequate overshoot. The results show successful tracking over a larger (and therefore more challenging) range of reference current values ($i_d^* \in [-400, 100]$, $i_q^* \in [-100, 0]$) than was attempted in Fig. 7.

This experiment shows the RNN is successfully making adaptation to the changing plant behavior, while maintaining the tight tracking of the reference current that was seen in the previous experiment.



(a) $L = 2.6$mH, $\Delta v_d = -0.3 * v_d$, $\Delta v_q = -0.1 * v_d$



(b) $L = 1.4$mH, $\Delta v_d = -0.3 * v_d$, $\Delta v_q = -0.1 * v_d$

Figure 10: Performance of neural vector controller under variable grid-filter inductance and PCC voltage conditions.

## *5.3. Ability to Track Fluctuating Reference Current*

GCCs are typically used to connect wind turbines and solar photovoltaic (PV) arrays to the grid. Due to variable weather conditions, the power transferred from a wind turbine or a PV array changes rapidly, making the GCC reference current (the target state) vary constantly over time. To represent such conditions, a variable d-axis reference current is generated while the q-axis reference current is set at zero (i.e., zero reactive power), with the standard fixed inductance value, $L = 2\text{mH}$. Fig. 11 shows that the neural network, with the stabilization matrix, predictive error signal $\vec{x}_k - \hat{x}_k$, and control action history $\vec{s}_k$, performs a near perfect match to the rapidly changing reference current. This kind of rapid close tracking would not be possible by PI controllers, as indicated by their sluggish performance in the previous experiments (e.g. Fig. 9).

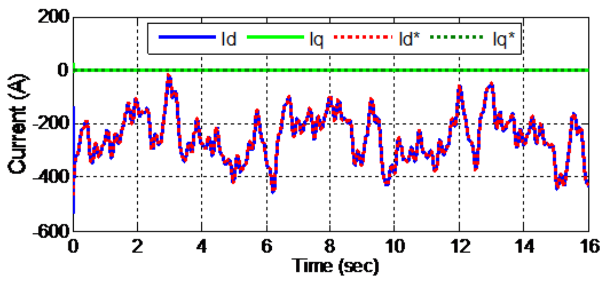


Figure 11: Performance of neural vector controller under a variable reference current condition.

## *5.4. Experiment with Non-invertible Control Matrix*

We now present an experiment where the matrix $\mathbf{B}$ is rectangular and non-invertible. The system considered here is taken from (Kirk, 2004, Ex.5.2-3):

$$\frac{\partial \vec{x}}{\partial t} = \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix} \vec{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, \tag{12}$$

for a state vector $\vec{x} := \left[\begin{smallmatrix} x^0 \\ x^1 \end{smallmatrix}\right]$ and control $u \in \Re$. This equation was discretized using a sampling time $T_s = 0.01$, producing matrices $\mathbf{A}$ and $\mathbf{B}$ analogous to those used in Eq. (3). Since $\mathbf{B}$ is non-square here, and therefore non-invertible, in order to use the stabilization matrix, we replaced the matrix inverse operation in Eq. (10) by the Moore-Penrose pseudoinverse (Golub and Van Loan, 1983) to obtain the results in the following experiment.

The problem is to optimize the cost-to-go function $J$ (Eq. 7) for,

$$U(\vec{x}_k, \vec{x}_k^*, \vec{u}_k) := \left(x^0(k) - 1\right)^2 + 0.0025\left(u(k)\right)^2. \tag{13}$$

Optimizing this cost function will move the state vector towards $x^0 = 1$ as quickly as possible, while also penalizing excessively large actions $u$. Although this cost function does not explicitly specify a target for the state vector component $x^1$, an implicit target for it is $x^1 = 0$, since the only way to obtain a fixed point of Eq. (12) is by achieving $x^1 = 0$. Hence in this problem, the reference state can be considered to be $\vec{x}^* = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$.

The action network had a layered structure 4-6-1, with bias-weights for each node, extra short-cut connections from the input to the output layer, hyperbolic tangent activation functions at all network nodes, and weights initially randomized in the range $[-.1, .1]$. The 4 inputs to the neural network were $\tanh(\vec{x}_k/10)$ and $\tanh(\vec{x}_k^* - \vec{x}_k)$. The output of the neural network $y$ was used to generate the action via the relationship $u = 10y$ when no stabilization matrix was used, and $u = 10y + \mathbf{W_0}\vec{x}$ when the stabilization matrix was used.

The trajectory start state was always $\vec{x} = \left[\begin{smallmatrix}2\\1\end{smallmatrix}\right]$, and the trajectory duration was 3 seconds (i.e. 300 time steps), with discount factor $\gamma = 1$. In this problem no adaptive behaviour was required, since the **A** and **B** matrices are fixed and known. Training took place for 800 iterations, from 10 different random weight initializations, using BPTT with RPROP, both with and without a stabilization matrix. The exact derivatives of Eq. (13) were made available to the BPTT algorithm. Results are shown in Table 4 and Fig. 12 shows a trajectory generated with the stabilization matrix, i.e. effectively solving the problem by following the reference state $\vec{x}^*$.

Table 4: Results with and without the "stabilization matrix", for the problem defined in Sec. 5.4.

| Results showing different $J$ values for different learning trials from 10 different initial weight randomizations. | | | | | |
|---|---|---|---|---|---|
| Without stabilization matrix | 0.2508 | 0.2508 | 0.2508 | 0.2506 | 0.2507 |
| | 0.2507 | 0.2508 | 0.2507 | 0.2509 | 0.2508 |
| With stabilization matrix | 0.2030 | 0.2030 | 0.2030 | 0.2029 | 0.2029 |
| | 0.2039 | 0.2029 | 0.2028 | 0.2029 | 0.2029 |

## 6. Conclusions

In this paper we have shown how a RNN could be used to control a three-phase grid-connected rectifier/inverter for renewable, micro-grid and

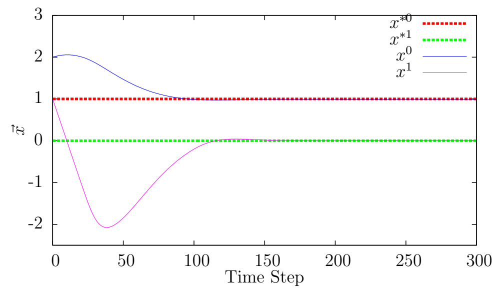


Figure 12: Trajectory after training with a stabilization matrix, for the problem defined in Sec. 5.4.

electric power system applications. We described how conventional neural controllers, which require on-line retraining to achieve their adaptation, can have limitations in adaptation speed and convergence. In contrast, the controller we have proposed does not need any on-line retraining, and can adapt effectively and almost instantaneously to changing plant conditions and/or changing reference commands. Compared to standard vector-control methods, including recently developed direct-current vector-control techniques, the neural vector-control approach produces good response time, low overshoot, and in general, excellent performance.

We have proposed some novel RNN inputs that allow for rapid adaptation and have discussed the motivations for using these unusual inputs as opposed to conventional generic context units. We have also introduced the stabilization-matrix method to ease RNN training in these experiments. The benefits of this method have been shown in numerical experiments, in Sections 3 to 5, in the case-study GCC problem and in a different problem domain demonstrating a situation with a non-invertible control matrix. It is our contention that this RNN controller architecture can have generic control applications to other kinds of plant, and produce a competitive alternative method to ACD and PI controllers.

We also have tried to emphasise the importance of addressing the problem of "exploding" gradients in RNN training. **The results of our experiments show that the use of the stabilization matrix has correctly addressed this significant issue, and the concept has been used to attack the long-standing problem of efficiently training RNNs to solve the tracking problem. Our approach contrasts** to conventional RNN research which tends to focus on the complementary problem known as "vanishing" gradients (Hochreiter and Schmidhuber, 1997), or superior optimizers (Feldkamp and Puskorius, 1998). We have described how the method we proposed can be used in conjunction with these more powerful optimizers.

In this research we have concentrated on situations where the $\mathbf{A}$ and $\mathbf{B}$ matrices are fixed or changing with small deviations over time. In future research, it will be necessary to investigate more problem domains including situations where the $\mathbf{A}$ and $\mathbf{B}$ matrices are functions of both $\vec{x}$ and $t$.

## Appendix A. Proof of sufficiency of the arguments chosen to solve the adaptive tracking problem

In this appendix, in Theorem 1 we prove that a neural network with inputs $(\vec{x}_k^* - \vec{x}_k)$ and $(\vec{x}_k - \hat{x}_k)$ is theoretically capable of solving the tracking problem, under mildly perturbed system and control matrices, $\mathbf{A}$ and $\mathbf{B}$. In Theorem 2 we show that it is not possible for a neural network with only the inputs $\vec{x}_k$ and $\vec{x}_k^*$ to solve the same problem.

**Theorem 1.** *A neural network of the form $\pi(\vec{x}_k, \vec{x}_k^* - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{w})$ can solve the tracking problem for some unknown constant system matrices $\mathbf{A}$ and $\mathbf{B}$, which are suitably close approximations to some known constant system matrices $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$, if the following 4 conditions are all met:*

1. *The known constant matrices $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ are sufficiently close approximations to the unknown constant matrices $\mathbf{A}$ and $\mathbf{B}$.*
2. *The matrix $\bar{\mathbf{B}}$ is square and invertible.*
3. *The tracking target $\vec{x}^*$ is sufficiently slowly moving such that it can be treated as constant.*
4. *The discrete-time matrices $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ were generated from an underlying continuous-time process with sufficiently small sampling time.*

Proof. The proof is split into two steps. Firstly, in "Proof Part-A", it is proven that we can simplify the situation to the case where both $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ are equal to identity matrices. Secondly, in "Proof Part-B", this simplified situation is analysed and a choice of action, constructed only from the arguments to the neural network, is proven to make the system state $\vec{x}_k$ converge to the tracking target $\vec{x}^*$.

In both of these proof steps, we split the unknown matrices $\mathbf{A}$ and $\mathbf{B}$ into known parts ($\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$) and unknown parts ($\mathbf{A}_\triangle$ and $\mathbf{B}_\triangle$), respectively, such that $\mathbf{A} := \bar{\mathbf{A}}+\mathbf{A}_\Delta$ and $\mathbf{B} := \bar{\mathbf{B}}+\mathbf{B}_\Delta$. Under this notation, the state-evolution equation (4) changes into

$$\vec{x}_{k+1} = (\bar{\mathbf{A}} + \mathbf{A}_\Delta)\vec{x}_k + (\bar{\mathbf{B}} + \mathbf{B}_\Delta)(\vec{u}_k - \vec{c}). \tag{A.1}$$

Also, by Condition 3, we can assume the tracking target $\vec{x}^*_k$ is fixed and hence drop the $k$ subscript and just use $\vec{x}^*$.

The two main proof parts now follow:

Proof Part-A. We define a transformed control vector $\vec{v}_k$, which is related to the original control vector $\vec{u}_k$ by the relationship:

$$\vec{u}_k = \bar{\mathbf{B}}^{-1}\left(\vec{v}_k - (\bar{\mathbf{A}} - \mathbf{I})\vec{x}_k\right) + \vec{c}. \tag{A.2}$$

Substituting this into (A.1) gives:

$$\begin{aligned}
\vec{x}_{k+1} =& (\bar{\mathbf{A}} + \mathbf{A}_\Delta)\vec{x}_k + (\bar{\mathbf{B}} + \mathbf{B}_\Delta)\bar{\mathbf{B}}^{-1}\left(\vec{v}_k - (\bar{\mathbf{A}} - \mathbf{I})\vec{x}_k\right) && \text{(by (A.2))}\\
=& (\bar{\mathbf{A}} + \mathbf{A}_\Delta)\vec{x}_k + (\vec{v}_k - (\bar{\mathbf{A}} - \mathbf{I})\vec{x}_k) + \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}\left(\vec{v}_k - (\bar{\mathbf{A}} - \mathbf{I})\vec{x}_k\right)\\
=& (\mathbf{A}_\Delta)\vec{x}_k + (\vec{v}_k + \vec{x}_k) + \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}\left(\vec{v}_k - (\bar{\mathbf{A}} - \mathbf{I})\vec{x}_k\right)\\
=& \left(\mathbf{I} + \mathbf{A}_\Delta - \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}(\bar{\mathbf{A}} - \mathbf{I})\right)\vec{x}_k + \left(\mathbf{I} + \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}\right)\vec{v}_k\\
=& \left(\mathbf{I} + \mathbf{A}'_\Delta\right)\vec{x}_k + \left(\mathbf{I} + \mathbf{B}'_\Delta\right)\vec{v}_k && \text{(A.3)}
\end{aligned}$$

where

$$\mathbf{A}'_\Delta := \mathbf{A}_\Delta - \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}(\bar{\mathbf{A}} - \mathbf{I}), \tag{A.4}$$

and

$$\mathbf{B}'_\Delta := \mathbf{B}_\Delta\bar{\mathbf{B}}^{-1}. \tag{A.5}$$

Therefore we only need consider control systems of the form (A.3). $\vec{v}_k$ is now the control vector. Next we must show that the two perturbation matrices, $\mathbf{A}'_\Delta$ and $\mathbf{B}'_\Delta$, in (A.3) are both small.

The matrices $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ were derived from an exact discrete-time system of the form of Eq. (4), i.e.

$$\vec{x}_{k+1} = \bar{\mathbf{A}}\vec{x}_k + \bar{\mathbf{B}}(\vec{u}_k - \vec{c}), \tag{A.6}$$

which itself was derived from an underlying continuous-time system, of the form of Eq. (2),

$$\frac{\partial \vec{x}(t)}{\partial t} = \mathbf{F}\vec{x}(t) + \mathbf{G}(\vec{u}(t) - \vec{c}), \tag{A.7}$$

with sampling time $T_s$.

Note that the discrete-time system (Eq. (A.6)) is related to the continuous-time system (Eq. (A.7)) by a first-order Taylor-series expansion, as follows:

$$\begin{aligned}
\vec{x}_{k+1} &= \vec{x}_k + \frac{\partial \vec{x}}{\partial t} T_s + \mathrm{O}\big((T_s)^2\big) \\
&= \vec{x}_k + (\mathbf{F}\vec{x}_k + \mathbf{G}(\vec{u}_k - \vec{c}))\, T_s + \mathrm{O}\big((T_s)^2\big) \qquad \text{(by Eq. (A.7))} \\
&= (\mathbf{I} + \mathbf{F}T_s)\vec{x}_k + \mathbf{G}(\vec{u}_k - \vec{c})T_s + \mathrm{O}\big((T_s)^2\big) \qquad (A.8)
\end{aligned}$$

By comparing Eqs. (A.8) and (A.6), we can see that $\bar{\mathbf{A}} = \mathbf{I} + \mathbf{F}T_s + \mathrm{O}\big((T_s)^2\big)$, and $\bar{\mathbf{B}} = \mathbf{G}T_s + \mathrm{O}\big((T_s)^2\big)$. Therefore as $T_s \to 0$, we must have $\bar{\mathbf{A}} \to \mathbf{I}$ and $\bar{\mathbf{B}} \to 0$. Therefore by Condition 4, we have

$$\bar{\mathbf{A}} - \mathbf{I} \approx 0 \tag{A.9}$$

Also, by Conditions 1 and 2, we have,[1]

$$\mathbf{B}_\Delta \bar{\mathbf{B}}^{-1} \approx 0. \tag{A.10}$$

We can now see that $\mathbf{A}'_\Delta$ is small, since all of the terms in equation (A.4) are small (by Eqs. (A.9) and (A.10) and Condition 1 which implies that both $\mathbf{A}_\Delta$ and $\mathbf{B}_\Delta$ are small). Also, $\mathbf{B}'_\Delta$ is small too, by Eqs. (A.5) and (A.10).

So working with Eq. (A.3), we can relabel $\mathbf{A}'_\Delta$ to $\mathbf{A}_\Delta$ and $\mathbf{B}'_\Delta$ to $\mathbf{B}_\Delta$ and $\vec{v}$ to $\vec{u}$, and therefore work with

$$\vec{x}_{k+1} = (\mathbf{I} + \mathbf{A}_\Delta)\,\vec{x}_k + (\mathbf{I} + \mathbf{B}_\Delta)\,\vec{u}_k, \tag{A.11}$$

where both $\mathbf{A}_\Delta$ and $\mathbf{B}_\Delta$ are small. This equation is similar to (A.1), except that we can now assume $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ are both identity matrices.

[1]This equation takes a bit of consideration, since $T_s \to 0$ implies both $\mathbf{B}_\Delta$ and $\bar{\mathbf{B}}$ have magnitudes proportional to $T_s$, and this will make $\bar{\mathbf{B}}^{-1}$ large, implying a possible violation of Eq. (A.10). However when the product $\mathbf{B}_\Delta \bar{\mathbf{B}}^{-1}$ is formed, the two proportionalities to $T_s$ cancel each other, and therefore $T_s$ does not influence the final magnitude of this product. Hence we only need consider Conditions 1 and 2 to justify Eq. (A.10).

Proof Part-B. The simplified state-evolution equation is given by (A.11), or equivalently,

$$\vec{x}_k = (\mathbf{I} + \mathbf{A}_\Delta)\vec{x}_{k-1} + (\mathbf{I} + \mathbf{B}_\Delta)\vec{u}_{k-1}. \tag{A.12}$$

In this system (A.12), the "predicted" state defined by (11) simplifies down into:

$$\hat{x}_k := \vec{x}_{k-1} + \vec{u}_{k-1}. \tag{A.13}$$

We now prove that it is possible to control the above system, assuming sufficiently small perturbation matrices $\mathbf{A}_\Delta$ and $\mathbf{B}_\Delta$, by making our action choice a function of the inputs of the neural network, as follows. Consider the following choice of action, which is permitted because it is purely a function of the arguments of $\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{w})$:

$$\begin{aligned}
\vec{u}_k =& (\vec{x}^* - \vec{x}_k) - (\vec{x}_k - \hat{x}_k) && \text{(A.14)}\\
=& \vec{x}^* - 2\vec{x}_k + \vec{x}_{k-1} + \vec{u}_{k-1} && \text{(by (A.13))}\\
=& \vec{x}^* - 2(\vec{x}_{k-1} + \mathbf{A}_\Delta\vec{x}_{k-1} + \vec{u}_{k-1} + \mathbf{B}_\Delta\vec{u}_{k-1}) + \vec{x}_{k-1} + \vec{u}_{k-1} && \text{(by (A.12))}\\
=& \vec{x}^* - 2\vec{x}_{k-1} - 2\mathbf{A}_\Delta\vec{x}_{k-1} - 2\vec{u}_{k-1} - 2\mathbf{B}_\Delta\vec{u}_{k-1} + \vec{x}_{k-1} + \vec{u}_{k-1} &&\\
=& \vec{x}^* - \vec{x}_{k-1} - 2\mathbf{A}_\Delta\vec{x}_{k-1} - \vec{u}_{k-1} - 2\mathbf{B}_\Delta\vec{u}_{k-1} &&\\
=& \vec{x}^* - (\mathbf{I} + 2\mathbf{A}_\Delta)\vec{x}_{k-1} - (\mathbf{I} + 2\mathbf{B}_\Delta)\vec{u}_{k-1} && \text{(A.15)}
\end{aligned}$$

Combining (A.12) and (A.15) into one discrete-time system gives:

$$\begin{pmatrix}\vec{x}_k\\ \vec{u}_k\end{pmatrix} = \mathbf{E}\begin{pmatrix}\vec{x}_{k-1}\\ \vec{u}_{k-1}\end{pmatrix} + \begin{pmatrix}0\\ \vec{x}^*\end{pmatrix} \tag{A.16}$$

where

$$\mathbf{E} := \begin{pmatrix}(\mathbf{I} + \mathbf{A}_\Delta) & (\mathbf{I} + \mathbf{B}_\Delta)\\ -(\mathbf{I} + 2\mathbf{A}_\Delta) & -(\mathbf{I} + 2\mathbf{B}_\Delta)\end{pmatrix}. \tag{A.17}$$

We must prove that (A.16) converges. Note that a fixed point of this is $\begin{pmatrix}\vec{x}^* & -(\mathbf{I} + \mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^*\end{pmatrix}^T$, because then

$$\begin{pmatrix}\vec{x}^*\\ -(\mathbf{I} + \mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^*\end{pmatrix} = \mathbf{E}\begin{pmatrix}\vec{x}^*\\ -(\mathbf{I} + \mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^*\end{pmatrix} + \begin{pmatrix}0\\ \vec{x}^*\end{pmatrix} \tag{A.18}$$

This fixed point satisfies $\vec{x}_k = \vec{x}^*$, so it is a correct solution to the tracking problem.

So we next must show that this fixed point is an attractor. Substituting shifted coordinates $\vec{x}_k := \vec{x}'_k + \vec{x}^*$ and $\vec{u}_k := \vec{u}'_k - (\mathbf{I}+\mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^*$ into (A.16) moves the fixed point to the origin:

$$\begin{aligned}
\begin{pmatrix} \vec{x}'_k + \vec{x}^* \\ \vec{u}'_k - (\mathbf{I}+\mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^* \end{pmatrix} &= \mathbf{E}\begin{pmatrix} \vec{x}'_{k-1} + \vec{x}^* \\ \vec{u}'_{k-1} - (\mathbf{I}+\mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^* \end{pmatrix} + \begin{pmatrix} 0 \\ \vec{x}^* \end{pmatrix} \\
&= \mathbf{E}\begin{pmatrix} \vec{x}'_{k-1} \\ \vec{u}'_{k-1} \end{pmatrix} + \mathbf{E}\begin{pmatrix} \vec{x}^* \\ -(\mathbf{I}+\mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^* \end{pmatrix} + \begin{pmatrix} 0 \\ \vec{x}^* \end{pmatrix} \\
&= \mathbf{E}\begin{pmatrix} \vec{x}'_{k-1} \\ \vec{u}'_{k-1} \end{pmatrix} + \begin{pmatrix} \vec{x}^* \\ -(\mathbf{I}+\mathbf{B}_\Delta)^{-1}\mathbf{A}_\Delta\vec{x}^* \end{pmatrix} \qquad \text{(by Eq. (A.18))}
\end{aligned}$$

$$\Rightarrow \begin{pmatrix} \vec{x}'_k \\ \vec{u}'_k \end{pmatrix} = \mathbf{E}\begin{pmatrix} \vec{x}'_{k-1} \\ \vec{u}'_{k-1} \end{pmatrix} \tag{A.19}$$

Hence we just need to prove that (A.19) converges to the origin.

We split the matrix $\mathbf{E}$ in (A.17) into two parts: a core matrix $\mathbf{C} := \left(\begin{smallmatrix} \mathbf{I} & \mathbf{I} \\ -\mathbf{I} & -\mathbf{I} \end{smallmatrix}\right)$ and a small perturbation matrix $\mathbf{P} := \left(\begin{smallmatrix} \mathbf{A}_\Delta & \mathbf{B}_\Delta \\ -2\mathbf{A}_\Delta & -2\mathbf{B}_\Delta \end{smallmatrix}\right)$, such that $\mathbf{P} + \mathbf{C} \equiv \mathbf{E}$, and also define $\vec{y}_k := \left(\begin{smallmatrix} \vec{x}'_k \\ \vec{u}'_k \end{smallmatrix}\right)$, so that (A.19) can be rewritten as

$$\vec{y}_k = (\mathbf{C} + \mathbf{P})\vec{y}_{k-1}.$$

Applying two steps of this iteration, and noting that $\mathbf{C}^2 = \left(\begin{smallmatrix} \mathbf{I} & \mathbf{I} \\ -\mathbf{I} & -\mathbf{I} \end{smallmatrix}\right)\left(\begin{smallmatrix} \mathbf{I} & \mathbf{I} \\ -\mathbf{I} & -\mathbf{I} \end{smallmatrix}\right) = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$, gives:

$$\vec{y}_k = (\mathbf{CP} + \mathbf{PC} + \mathbf{P}^2)\vec{y}_{k-2}. \tag{A.20}$$

Since every term in this product has a factor of $\mathbf{P}$, which is assumed small, clearly this equation will converge to zero as $k \to \infty$. This completes the proof that it is possible to solve the tracking problem using a function of the form $\pi(\vec{x}_k, \vec{x}^*_k - \vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{w})$, under the assumed conditions of the theorem.

Remark. Note that this theorem proof does not attempt to show that the action defined by Eq. (A.14) is in any way an optimal choice, or that deciding to base control decisions using an arbitrary function of the form $\pi(\vec{x}_k, \vec{x}^*_k -$

$\vec{x}_k, \vec{x}_k - \hat{x}_k, \vec{w})$ is in any way optimal; just that this choice produces a possible solution, which is guaranteed to converge to the tracking target, eventually, and under the given assumptions. In fact, the results in the third row of Table 2 show that additional inputs can improve tracking performance over the basic function considered in Theorem 1. These experiments also show the neural network can cope with fast-moving tracking targets (e.g. in Section 5.3), and finite $T_s$ values, so the four strict conditions of this theorem may be stretched, to a certain extent, in practical applications.

A key motivation of this proof has been to show that the disturbances to the $\mathbf{A}$ and $\mathbf{B}$ matrices can be to all matrix components, and tracking will still be possible provided those disturbances are sufficiently small.

**Theorem 2.** *It is not possible for a feed-forward neural network with only the inputs $\vec{x}_k$ and $\vec{x}^*_k$ to solve the tracking problem, when the system and control matrices $\mathbf{A}$ and $\mathbf{B}$ have undergone a mild disturbance.*

Proof. Suppose we had actions $\vec{u}_k$ generated by a function $\vec{u}_k := \pi(\vec{x}_k, \vec{x}^*)$, and suppose this control action was able to move the plant's state vector to the tracking target $\vec{x}^*$ and hold it there, so that $\vec{x}_k = \vec{x}^*$ for all $k > k_0$, for some constant $k_0$. At this target state, i.e. at $\vec{x}_{k+1} = \vec{x}_k = \vec{x}^*$, the state-evolution equation, in the simplified frame of reference given by (A.11), becomes:

$$\begin{aligned} \vec{x}^* &= (\mathbf{I} + \mathbf{A}_\Delta)\, \vec{x}^* + (\mathbf{I} + \mathbf{B}_\Delta)\, \vec{u}_k \\ \Rightarrow \vec{u}_k &= -\left(\mathbf{I} + \mathbf{B}_\Delta\right)^{-1} \mathbf{A}_\Delta \vec{x}^* \end{aligned}$$

This is the action required to hold the plant at the target state, once there. So, when $\vec{x}_k = \vec{x}^*$, we must have $\pi(\vec{x}_k, \vec{x}^*)$ equal to

$$\pi(\vec{x}^*, \vec{x}^*) = -\left(\mathbf{I} + \mathbf{B}_\Delta\right)^{-1} \mathbf{A}_\Delta \vec{x}^*$$

The right-hand side (RHS) of this equation has a dependency on the perturbation matrices, $\mathbf{A}_\Delta$ and $\mathbf{B}_\Delta$. This means, in different plants with different perturbation matrices, the RHS will evaluate to different values. The left-hand side (LHS) however is constant for the given $\vec{x}^*$ vector. So this is a contradiction; it is not possible for the constant LHS to equal the varying possible RHS values. Consequently, it is not possible to solve this tracking problem by choosing actions based only on $\vec{x}_k$ and $\vec{x}^*$.

## References

Barnard, E., 1993. Temporal-difference methods and markov models. IEEE Transactions on Systems, Man, and Cybernetics 23 (2), 357–365.

Bishop, C. M., 1995. Neural Networks for Pattern Recognition. Oxford University Press.

Carrasco, J. M., Franquelo, L. G., Bialasiewicz, J. T., Galván, E., Guisado, R. C. P., Prats, M. A. M., León, J., Moreno-Alfonso, N., August 2006. Power-electronic systems for the grid integration of renewable energy sources: A survey. IEEE Transactions on Industrial Electronics 53 (4), 1002–1016.

Dannehl, J., Wessels, C., Fuchs, F. W., October 2009. Limitations of voltage-oriented PI current control of grid-connected PWM rectifiers with *lcl* filters. IEEE Transactions on Industrial Electronics 56 (2), 380–388.

Fairbank, M., Alonso, E., June 2012. The divergence of reinforcement learning algorithms with value-iteration and function approximation. In: Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12). IEEE Press, pp. 3070–3077.

Feldkamp, L. A., Puskorius, G. V., 1998. A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering, and classification. Proceedings of the IEEE 86 (11), 2259–2277.

Figueres, E., Garcerá, G., Sandia, J., Gonzalez-Espin, F., Rubio, J. C., March 2009. Sensitivity study of the dynamics of three-phase photovoltaic inverters with an LCL grid filter. IEEE Transactions on Industrial Electronics 56 (3), 706–717.

Franklin, G. F., Workman, M. L., Powell, D., 1998. Digital control of dynamic systems, 3rd Edition. Addison-Wesley Longman Publishing Co., Inc.

Golub, G. H., Van Loan, C. F., 1983. Matrix Computations. Johns Hopkins University Press, Baltimore, Maryland.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9 (8), 1735–1780.

Kirk, D. E., 2004. Optimal control theory: an introduction. Dover Publications.

Li, S., Fairbank, M., Wunsch, D., Alonso, E., June 2012. Vector control of a grid-connected rectifier inverter using an artificial neural network. In: Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12). IEEE Press, pp. 1783–1789.

Li, S., Haskew, T. A., Hong, Y. K., Xu, L., February 2011. Direct-current vector control of three-phase grid-connected rectifier-inverter. Electric Power Systems Research 81 (2), 357–366.

Li, S., Haskew, T. A., Xu, L., December 2010. Control of HVDC light system using conventional and direct current vector control approaches. IEEE Transactions on Power Electronics 25 (12), 3106–3118.

Luo, A., Tang, C., Shuai, Z., Tang, J., Xu, X. Y., Chen, D., July 2009. Fuzzy-PI-based direct-output-voltage control strategy for the STATCOM used in utility distribution systems. IEEE Transactions on Industrial Electronics 56 (7), 2401–2411.

Mohan, N., Undeland, T. M., Robbins, W. P., 2002. Power electronics: converters, applications, and design, 3rd Edition. John Wiley & Sons.

Mullane, A., Lightbody, G., Yacamini, R., November 2005. Wind-turbine fault ride-through enhancement. IEEE Transactions on Power Systems 20 (4), 1929–1937.

Park, J., Harley, R., Venayagamoorthy, G., 2004. New external neuro-controller for series capacitive reactance compensator in a power network. IEEE Transactions on Power Systems 19 (3), 1462–1472.

Pena, R., Clare, J. C., Asher, G. M., May 1996. Doubly fed induction generator using back-to-back PWM converters and its application to variable-speed wind-energy generation. In: Electric Power Applications, IEE Proceedings-. Vol. 143. IET, pp. 231–241.

Prokhorov, D., Wunsch, D., 1997. Adaptive critic designs. IEEE Transactions on Neural Networks 8 (5), 997–1007.

Prokhorov, D. V., Feldkamp, L. A., Tyukin, I. Y., 2002. Adaptive behavior with fixed weights in RNNs: an overview. In: Proceedings of the 2002 International Joint Conference on Neural Networks, 2002. IJCNN'02. Vol. 3. IEEE, pp. 2018–2022.

Qiao, W., Harley, R., Venayagamoorthy, G., 2008a. Fault-tolerant optimal neurocontrol for a static synchronous series compensator connected to a power network. IEEE Transactions on Industry Applications 44 (1), 74–84.

Qiao, W., Harley, R. G., Venayagamoorthy, G. K., 2009a. Coordinated reactive power control of a large wind farm and a STATCOM using heuristic dynamic programming. IEEE Transactions on Energy Conversion 24 (2), 493–503.

Qiao, W., Venayagamoorthy, G. K., Harley, R. G., 2008b. Optimal wide-area monitoring and nonlinear adaptive coordinating neurocontrol of a power system with wind power integration and multiple FACTS devices. Neural Networks 21 (2), 466–475.

Qiao, W., Venayagamoorthy, G. K., Harley, R. G., 2009b. Real-time implementation of a STATCOM on a wind farm equipped with doubly fed induction generators. IEEE Transactions on Industry Applications 45 (1), 98–107.

Rabelo, B. C., Hofmann, W., da Silva, J. L., de Oliveira, R. G., Silva, S. R., October 2009. Reactive power control design in doubly fed induction generators for wind turbines. IEEE Transactions on Industrial Electronics 56 (10), 4154–4162.

Rall, L. B., 1981. Automatic differentiation: Techniques and applications. Lecture Notes in Computer Science 120.

Riedmiller, M., Braun, H., 1993. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Proc. of the IEEE Intl. Conf. on Neural Networks. San Francisco, CA, pp. 586–591.

Venayagamoorthy, G., Harley, R., Wunsch, D., 2003. Implementation of adaptive critic-based neurocontrollers for turbogenerators in a multimachine power system. IEEE Transactions on Neural Networks 14 (5), 1047–1064.

Venayagamoorthy, G. K., Harley, R. G., Wunsch, D. C., 2002. Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator. IEEE Transactions on Neural Networks 13 (3), 764–773.

Wang, C., Nehrir, M. H., November 2007. Short-time overloading capability and distributed generation applications of solid oxide fuel cells. IEEE Transactions on Energy Conversion 22 (4), 898–906.

Wang, F.-Y., Zhang, H., Liu, D., 2009. Adaptive dynamic programming: An introduction. IEEE Computational Intelligence Magazine 4 (2), 39–47.

Werbos, P. J., 1990. Backpropagation through time: What it does and how to do it. In: Proceedings of the IEEE. Vol. 78, No. 10. pp. 1550–1560.

Werbos, P. J., 1992. Approximating dynamic programming for real-time control and neural modeling. In: White, Sofge (Eds.), Handbook of Intelligent Control. Van Nostrand Reinhold, New York, Ch. 13, pp. 493–525.

Werbos, P. J., 1998. Stable adaptive control using new critic designs. eprint arXiv:adap-org/9810001, sections 7.7–7.8.

Werbos, P. J., 2005. Backwards differentiation in AD and neural nets: Past links and new opportunities. In: Bücker, H. M., Corliss, G., Hovland, P., Naumann, U., Norris, B. (Eds.), Automatic Differentiation: Applications, Theory, and Implementations. Lecture Notes in Computational Science and Engineering. Springer, pp. 15–34.

Werbos, P. J., McAvoy, T., Su, T., 1992. Neural networks, system identification, and control in the chemical process industries. In: White, Sofge (Eds.), Handbook of Intelligent Control. Van Nostrand Reinhold, New York, Ch. 10, sec. 10.6.1–10.6.2, pp. 339–343, version on author's homepage with errata.
URL `www.werbos.com`

Xu, L., Wang, Y., February 2007. Dynamic modeling and control of DFIG-based wind turbines under unbalanced network conditions. IEEE Transactions on Power Systems 22 (1), 314–323.