Online Adaptation of Deep Architectures with Reinforcement Learning

Thushan Ganegedara, Lionel Ott and **Fabio Ramos** [1]

Abstract. Online learning has become crucial to many problems in machine learning. As more data is collected sequentially, quickly adapting to changes in the data distribution can offer several competitive advantages such as avoiding loss of prior knowledge and more efficient learning. However, adaptation to changes in the data distribution (also known as covariate shift) needs to be performed without compromising past knowledge already built in into the model to cope with voluminous and dynamic data. In this paper, we propose an online stacked Denoising Autoencoder whose structure is adapted through reinforcement learning. Our algorithm forces the network to exploit and explore favourable architectures employing an estimated utility function that maximises the accuracy of an unseen validation sequence. Different actions, such as *Pool*, *Increment* and *Merge* are available to modify the structure of the network. As we observe through a series of experiments, our approach is more responsive, robust, and principled than its counterparts for non-stationary as well as stationary data distributions. Experimental results indicate that our algorithm performs better at preserving gained prior knowledge and responding to changes in the data distribution.

1 Introduction

Over the past decade, Deep Architectures [5], [1] have become a widely-discussed topic in machine learning. One key reason being the ability to jointly perform feature-extraction and classification on raw data, outperforming many other techniques in various domains including object recognition [7], [2], hand-writing recognition [5] and speech recognition [4]. A deep network can be understood as a neural network consisting of many hidden layers [3]. While the interest in deep networks arose quite early, only the recent hardware and optimisation developments (e.g. Graphical Processing Units (GPUs), Greedy pre-training) sparked the practicality of deep architectures.

Despite the note-worthy learning capacity, deep architectures are still susceptible to the past-knowledge being overridden due to *Covariate Shift* [15]. Covariate shift is a common phenomenon that transpires in online settings. Covariate shift essentially refers to the difference in training and testing data distributions. Successful exploitation of adaptive capabilities of deep networks to minimise the adverse effects of the covariate shift will lead to new frontiers in data science.

While many algorithms (especially Support Vector Machines (SVM)) have been enhanced with online learning capabilities [9], [11], only few attempts of incorporating online learning for Neural Networks have been proposed in the literature, notably in [19], [13], [10], and [14]. Of these, only [14] and [19] focus on changing the structure of the network, where the others focus on adapting a fixed architecture accordingly. [14] proposes an intriguing approach to evolve neural networks using genetic algorithm, by mutating weights and nodes in the network and crossing over existing networks to generate more fit off-springs. However, this technique is not scalable for deep networks and requires many repetitive runs through the data. [19] proposes a structural adaptation technique for deep architectures relying on simple heuristic (i.e. immediate performance convergence). [19] does not seek a long-term reward and lacks in responsiveness, as it waits for a pool of data to be filled in order to add nodes to the structure. These limitations motivate the question of how to explore the space of different architectures in an online setting in a more responsive, robust and principled manner.

In this paper, we introduce a state-of-the-art mechanism to modify deep architectures (specifically Denoising Autoencoders [18]) based on reinforcement learning. The decision making behaviour exploits and explores possible actions to discover favourable modifications to the structure (i.e. adding/removing nodes) by maximising a stipulated reward over time. Adding nodes helps to accommodate new features, while removing nodes helps to remove redundant features. An additional pooling operation fine-tunes the network with previously observed data. The method keeps track of a continuously updated utility (long-term reward) function to decide which action is best for a given state, whose estimation will improve over time. The experimental results on three datasets clearly show that our algorithm outperforms its counterparts in both stationary and non-stationary situations.

2 Background

2.1 Online Learning

By online learning we refer to the ability to accommodate new knowledge (i.e. features) without overriding previously acquired knowledge (i.e. features) [17]. This is becoming more popular due to the explosive growth of data. Online learning has the ability to learn from a continuous stream of data without a loss of past knowledge and attempts to address the non-stationary nature of data by allowing more flexibility in the model. For this reason, online algorithms perform significantly better in handling problems with covariate shift.

2.2 Deep Networks

We begin the presentation of the method by introducing the following notation:

- $\mathbf{x}$ - Inputs
- $\mathbf{y}$ - Input labels
- K - Number of classes

[1] University of Sydney, Australia email: tgan4199@uni.sydney.edu.au, lott4241@uni.sydney.edu.au and fabio.ramos@sydney.edu.au

- $\tilde{\mathbf{x}}$ - Noise-corrupted input
- $\hat{\mathbf{x}}$ - Reconstructed input
- W - Weights of a neuron layer
- b - Bias of a neuron layer
- b' - Reconstruction bias of a neuron layer

2.2.1 Autoencoder

An Autoencoder [6] maps a set of inputs $\mathbf{x} = \{\mathbf{x}_i \in [0,1]^D\}\ \forall i = 1, ..., N$ where $\mathbf{x_i} = \{x_i^1, x_i^2, ..., x_i^D\}$ and D is dimensionality of data to a latent feature space H with $h_{W,b}(\mathbf{x}) = sig(W\mathbf{x}+b)$, where $W \in \mathbb{R}^{H\times D}$, $b \in \mathbb{R}^H$ and $sig(s) = \frac{1}{1+\exp -s}$. An autoencoder can reconstruct the input $\hat{\mathbf{x}}_i\ \forall i = 1, ..., N$ from the latent feature space H with $\hat{\mathbf{x}} = sig(W^T \times h_{W,b}(\mathbf{x}) + b')$ where superscript ^{T} denotes transpose and $b' \in \mathbb{R}^D$. For simplicity we assume tied weights.

2.2.2 Denoising Autoencoder

The Denoising Autoencoder (DAE) is a variant of autoencoder which uses a corrupted (noisy) version of the example as the input [18]. This forces the algorithm to become more robust to noise. DAE works in the following manner.

First, the inputs are corrupted by introducing noise using a binomial distribution with probability p. Let us call the corrupted input $\tilde{\mathbf{x}}$. Next, $\tilde{\mathbf{x}}$ is mapped to a hidden representation using $h_{W,b}(\tilde{\mathbf{x}}) = sig(W\tilde{\mathbf{x}}+b)$ where $W \in \mathbb{R}^{H\times D}$, $b \in \mathbb{R}^H$ and $sig(s) = \frac{1}{1+\exp -s}$. Finally, the decoding function retrieves the reconstructed input, $\hat{\mathbf{x}} = sig(W^T \times h_{W,b}(\tilde{\mathbf{x}}) + b')$, where $b' \in \mathbb{R}^D$. In this work, we assume tied weights for encoding and decoding. *Cross entropy* is used as the cost function (Equation 1),

$$L_{gen}(\mathbf{x}, \hat{\mathbf{x}}) = \sum_{j=1}^{D} x^j \log(\hat{x}^j) + (1 - x^j)\log(1 - \hat{x}^j). \quad (1)$$

The optimal values for parameters W, b, b' are found by minimising the cost function,

$$W_{opt}, b_{opt}, b'_{opt} = \text{argmin}_{W,b,b'} L_{gen}(\mathbf{x}, \hat{\mathbf{x}}).$$

2.2.3 Stacked Denoising Autoencoders

A Stacked Denoising Autoencoder (SDAE) [18] is a set of connected autoencoders. A SDAE undergoes two main processes; pre-training and fine-tuning. In the pre-training process, the network is considered as a set of autoencoders $AE^1, ..., AE^L$. The output of $AE^l = \{W^l, b^l, b'^l\}$, $h^l_{W,b}$ where l is the current layer, is calculated as follows,

$$h^l_{W,b} = \begin{cases} sig(W^l\tilde{\mathbf{x}} + b^l); & \text{if } l = 1 \\ sig(W^l h^{l-1}_{W,b} + b^l); & \text{Otherwise.} \end{cases}$$

In the fine-tuning phase, the network is treated as a single deep autoencoder and trained using labelled data $\mathcal{D}$. Assuming labelled data in the format $\mathcal{D} = (\mathbf{x_i}, \mathbf{y_i}), \forall i = 1, ..., N$ where $\mathbf{y_i} \in \{0,1\}^K$ such that if y_i^j are the elements of $\mathbf{y}_i$ then $\sum_j y_i^j = 1$, we can use a *softmax* layer with parameters $\{W^{out}, b^{out}, b'^{out}\}$. The output of the network is defined as $\hat{\mathbf{y}} = \text{softmax}(W^{out} h^L_{W,b} + b^{out})$, where $\text{softmax}(a_k) = \frac{\exp(a_k)}{\sum_{k'} \exp(a_{k'})}$. Then the cost function becomes,

$$L_{disc}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{j=1}^{K} (y^j \log \hat{y}^j + (1 - y^j)\log(1 - \hat{y}^j)). \quad (2)$$

Finally, from Equation 2, we can formulate the optimisation problem to learn W, b and b' as,

$$W_{opt}, b_{opt}, b'_{opt} = \text{argmin}_{W,b,b'} L_{disc}(\mathbf{y}, \hat{\mathbf{y}}),$$

where W_{opt}=$(W^1_{opt}, ..., W^L_{opt}, W^{out}_{opt})$, b_{opt}=$(b^1_{opt}, ..., b^L_{opt}, b^{out}_{opt})$ and b'_{opt}=$(b'^1_{opt}, ..., b'^L_{opt}, b'^{out}_{opt})$.

2.3 Incremental Feature Learning for Denoising Autoencoders

Merge-Incremental Denoising Autoencoders (MI-DAE) is an online learning stacked denoising autoencoder proposed in [19]. Initially, the network is pre-trained using a pool of data (typically first 12,000 examples). Then, for every batch of data b_t, add hard examples (i.e. $\mathbf{x_i}$ if $L_{gen}(\mathbf{x_i}, \hat{\mathbf{x}}_\mathbf{i}) > \frac{\sum_{\forall \mathbf{x_j} \in b_t} L_{gen}(\mathbf{x_j}, \hat{\mathbf{x}}_\mathbf{j})}{|b_t|}$) to a pool, B. The method then performs merging of nodes within the same layer or adds new nodes to the network. Once the number of points in B exceeds a threshold, τ, retrieve previously calculated pairs of nodes with the highest similarity (ΔMrg) and add ΔInc new nodes to the network. Next, use B to greedily train newly added features. Afterwards, update ΔMrg and ΔInc [20] and remove all data from B. Finally repeat this process for all the batches in the sequence. Pseudo-code for this algorithm is presented in Algorithm 1.

Algorithm 1 MergeInc Algorithm

procedure MERGEINC(b_t,ΔMRG,ΔINC)
  Define: μ - Average reconstruction error for the most recent 10,000 examples
  Define: τ - Pool threshold (10,000 examples)
  Compute objective $L_{disc}(\mathbf{y_j}, \hat{\mathbf{y}}_\mathbf{j}), \forall \{\mathbf{x_j}, \mathbf{y_j}\} \in b_t$
  Add hard example $\mathbf{x_j}$ to B if $L_{gen}(\mathbf{x_j}, \hat{\mathbf{x}}_\mathbf{j}) > \mu$ of b_t
  if $|B| > \tau$ **then**
    Merge 2ΔMrg candidates to ΔMrg
    Add ΔInc nodes and fine-tune ΔInc new nodes with $\{\mathbf{x_j}, \mathbf{y_j}\} \in B$ while keeping rest of the network constant
    Update ΔMrg and ΔInc (Heuristic-based [20])
    Set $B = \emptyset$
  end if
  Fine-tune all the features (with ΔMrg and ΔInc) with b_t
end procedure

2.4 Reinforcement Learning and Markov Decision Processes

After describing SDAE, we now introduce notation and the basics of reinforcement learning (RL). RL enables an agent to learn a *policy*, π (a function that defines which action to take in a given state), by interacting with its environment, preferably trading-off between exploration and exploitation. A reinforcement learning task that satisfies the *Markov Property* can be formulated as a Markov decision process (MDP) [16]. Formally a Markov Decision Process can be defined using the following,

- A set of states - S
- A set of actions - A
- A transition function - $T : S \times A \times S \to [0, 1]$
- A reward function - $R : S \times A \times S \to \mathbb{R}$.

Table 1. The notations and definitions used in Section 3

Notation	Description	Notation	Description
N	Number of data points	B_r	Pool of data containing most recent τ examples
D	Dimensionality of data	B_{ft}	Pool of data containing dissimilar inputs
K	Number of classes	Λ	Distance threshold for B_{ft}
p	Number of data points in one batch	$\tilde{\mathcal{L}}^n(m)$	Exponential Moving Average of error L in the window $n-m$ to n
n	sequence number of the current batch of data	ν_l^n	Ratio between the current count of neurons and the initial count for neuron layer l for n^{th} data batch
τ	Size of data pools	ΔInc	Number of neurons to add at a given time
$\mathbf{x}_i$	i^{th} data point	ΔMrg	Number of neurons to remove at a given time
$\mathbf{y}_i$	Vectorized label of $\mathbf{x}_i$ s.t $\forall\ y_i^j \in \mathbf{y}_i\ y_i^j \in \{0,1\}$ s.t. $\sum_j y_i^j = 1$	r^n	The reward for the n^{th} batch of data
$\mathcal{D}$	Dataset containing $\{\{\mathbf{x}_1,\mathbf{y}_1\},\{\mathbf{x}_2,\mathbf{y}_2\},\ldots\}$	γ	Discount rate for Q value update
$\mathcal{D}^n$	n^{th} batch of data ($\mathcal{D}^n \subset \mathcal{D}$)	$Q(s,a)$	Utility function
L_g^n	Generative error for n^{th} batch of data	η_1	The duration until beginning to collect state-action pairs
L_c^n	Classification error for n^{th} batch of data	η_2	The duration until beginning to exploit Q-values

In this paper, RL is used to find the policy to adapt the structure of the network, given the current network configuration or state. Therefore, at a given instance i, from state s^i an action a^i is performed and the network transits to state s^{i+1}. Actions are modifications to the network such as adding new nodes or removing existing nodes. The state is a function of the network performance and will be defined in Section 3. The reward r^i for going from state s^i to s^{i+1} by taking action a^i is calculated based on the errors produced on the learning task. State s^{i+1} depends on the current state s^i and current action a^i, and is conditionally independent of all the previous states and actions, thus satisfying the *Markov Property*. The ultimate goal is to learn an optimal policy $\pi^*(s^i,a^i)$ that recommends the best action a^i for a given state s^i.

In order to learn the policy to select the best action for a given state, Q-Learning is used. Q-Learning (a variant of Temporal difference [16]) is an off-policy model-free approach to finding the optimal policy, π^*. Q-Learning estimates the utility value in an online manner and, as an off-policy learning, it learns a value function independent of the agent's experience. This leads to exploring new tactics the agent has not tried. Furthermore, Q-Learning can be employed for MDPs with unknown transition and reward functions. Q-Learning proceeds as follows,

1. Define $Q(s^n,a^n)$, where $s^n \in S$ and $a^n \in A$.
2. Initialise $Q^0(s^i,a^i)=0, \forall s^i \in S$ and $\forall a^i \in A$.
3. Update $Q^{t+1}(s^n,a^n) = (1-\alpha)\times Q^t(s^n,a^n) + \alpha \times [R(s^n,a^n,s^{n+1}) + \gamma(max_{a'}(Q(s^{n+1},a')))]$ where γ is the discount rate, α is the learning rate, and s^{i+1} is the state after action a^i.

One of the applications of using Q-learning is to train a multi-layer perceptron as found in [13]. More recently, a variant of Q-Learning was successfully used in a Convolutional Deep Network when the network was trained to play the *Atari* games using raw pixel images [10].

3 Reinforced Adaptive Denoising Autoencoder (RA-DAE)

3.1 Limitations of MI-DAE

MI-DAE (Algorithm 1) introduces several interesting concepts useful for online learning such as, pooling data and update rules for ΔMrg and ΔInc. However, the approach has several limitations: (1) The response of the algorithm to changes is slow as it waits for a pool of data (B) to be filled in order to execute an operation; (2) While the algorithm incorporates an intuitive criteria (performance convergence) to modify the network (update rules), the method is based on simple heuristics such as the immediate future reward that does not generally reflect a holistic view of the effect an action has on the network.

3.2 Overview of RA-DAE

Motivated by the drawbacks in MI-DAE, we propose a more robust and principled solution which relies on RL. In essence, our algorithm estimates an utility function $Q(s,a)$ for each state-action pair by sampling from the environment, where actions are modifications in the network structure. Using $Q(s,a)$, the algorithm selects the best action for a given state. The utility function is based on the accuracy measured on an unseen validation batch. Our approach is beneficial as,

- Actions are taken for every batch of data, resulting in fast response to sudden changes in the data distribution;
- The utility function ensures that actions are taken based on the long-term benefit they incur on the accuracy;
- A new pool operation refreshes the network's knowledge by fine-tuning the network using a pool of data containing data points *significantly* different from each other.

Notation: An input data stream is denoted as $\mathcal{D} = \{\{\mathbf{x}_1,\mathbf{y}_1\},\{\mathbf{x}_2,\mathbf{y}_2\},\ldots\}$, where $\mathbf{x}_i$ is a normalized data point, $\mathbf{x}_i \in [0,1]^D$, $\mathbf{y}_i \in \{0,1\}^K$ and y_i^j are the elements of $\mathbf{y}_i$ with $\sum_j y_i^j = 1$. The n^{th} data batch is written as $\mathcal{D}^n = \{\{\mathbf{x}_{(n-1)\times p}, \mathbf{y}_{(n-1)\times p}\},...,\{\mathbf{x}_{n\times p},\mathbf{y}_{n\times p}\}\}$, where p is the number of examples per batch. Denote the generative error as $\mathrm{L}_g^n = \frac{\sum_{\forall \mathbf{x_i}\in\mathcal{D}^n} L_{gen}(\mathbf{x_i},\hat{\mathbf{x}}_\mathbf{i})}{p}$ and the classification (or discriminative) error as $\mathrm{L}_c^n = \frac{\sum_{\forall \mathbf{y_i}\in\mathcal{D}^n} \mathbb{1}_{\hat{k}_i=k_i}}{p}$, where $\mathbb{1}$ is the *indicator* function and $k_i = argmax_{k'}(\{y_i^{k'}\})$, $\forall k' = 1,...,K$ of the n^{th} batch. r^n denotes the reward for the n^{th} batch. Finally define two pools

$B_r = \{\mathcal{D}^{n-\tau}, ..., \mathcal{D}^n\}$ and

$$B_{ft} = \begin{cases} \mathcal{D}^n \text{ if } B_{ft} = \emptyset \\ \mathcal{D}^n \cup B_{ft} \text{ if } d(\mathcal{D}^n, \mathcal{D}^j) > \Lambda \ \forall \mathcal{D}^j \in B_{ft} \\ B_{ft} - \mathcal{D}^j \ j = argmin_{j'}(\forall \mathcal{D}^{j'} \in B_{ft}) \text{ if } |B_{ft}| > \tau \\ B_{ft} \text{ otherwise} \end{cases} \tag{3}$$

for some d distance measure and a similarity threshold $\Lambda \in [0, 1]$. η_1 and η_2 are pre-defined thresholds for starting to collect observed state-action pairs and exploiting Q-values respectively. α is the learning rate for Q-learning. A summary of the notation is in Table 1 for quick reference.

3.3 RL Definitions

To calculate when and which actions to take, we employ a MDP formulation. We define a set of states S, a set of actions A, and a reward function r^n below.

3.3.1 State Space

The state space S is defined as follows. For the n^{th} batch,

$$S = \{\tilde{\mathcal{L}}_g^n(m), \tilde{\mathcal{L}}_c^n(m), \nu_1^n\} \in \mathbb{R}^3 \tag{4}$$

where the moving exponential average ($\tilde{\mathcal{L}}$) is defined as $\tilde{\mathcal{L}}^n(m) = \alpha \mathrm{L}^n + (1-\alpha)\tilde{\mathcal{L}}^{n-1}(m-1)$, $n \geq m$ and m is a pre-defined constant. $\tilde{\mathcal{L}}_g$ and $\tilde{\mathcal{L}}_c$ denote $\tilde{\mathcal{L}}$ w.r.t. L_g and L_c, respectively, and $\nu_l^n = \frac{\text{Node Count}_{current}}{\text{Node Count}_{initial}}$ for the l^{th} hidden layer. $\tilde{\mathcal{L}}$ is defined in terms of recursive decay to respond rapidly to immediate changes.

This state space takes into account the following attributes:

- Ability of RA-DAE to classify an unseen batch of data;
- Difference between current data distribution and previously observed distributions;
- Complexity of RA-DAE's current structure.

The justification for the choice of state space is discussed in Section 4.2.1.

3.3.2 Action Space

The actions space is defined as,

$$A = \{Pool, Increment(\Delta\text{Inc}), Merge(\Delta\text{Mrg})\}, \tag{5}$$

where $Increment(\Delta\text{Inc})$ adds ΔInc new nodes and greedily initialise them using pool B_r. The $Merge(\Delta\text{Mrg})$ operation is performed by merging the 2ΔMrg nodes. *Merge* operation is executed by selecting the closest pairs (e.g. minimum Cosine distance) of ΔMrg nodes and merging each pair to a single node. The $Pool$ operation fine-tunes the network with B_{ft}. Both operations (i.e. Increment and Merge) are performed in the 1st hidden layer. Equations 6, 7 and 8 outline the calculations for ΔInc and ΔMrg,

$$\Delta = \lambda \exp^{\frac{-(\nu-\hat{\mu})}{2\sigma^2}} |L_c^n - L_c^{n-1}| \tag{6}$$

$$\Delta\text{Inc} = \begin{cases} \Delta; \text{ if } a = Increment \\ 0; \text{ Otherwise} \end{cases} \tag{7}$$

$$\Delta\text{Mrg} = \begin{cases} \Delta; \text{ if } a = Merge \\ 0; \text{ Otherwise} \end{cases} \tag{8}$$

Algorithm 2 RA-DAE algorithm

```
procedure RA-DAE
    define : n - Current batch ID
    Initialise Q(s, a) = 0 ∀s ∈ S, a ∈ A
    s, a = null
    while D_n ≠ NULL do
        s', a', Q', ΔMrg, ΔInc, = GetCtrlParam(n, Q, s, a)
        if a' = Pool then
            Fine-tune using B_ft
        else if a' = Increment then
            Add ΔInc new nodes to the network
            Train the ΔInc nodes greedily using B_r
        else if a' = Merge then
            Merge 2ΔMrg nodes into ΔMrg
        end if
        Train the network with D_n
        s = s', a = a', Q = Q'
        n = n + 1
    end while
end procedure
```

where λ is a coefficient controlling the amount of change, $\hat{\mu}$ and σ are chosen depending on how large or small the network is allowed to grow, and a is the current action chosen by Algorithm 3. We defined ΔMrg and ΔInc as a function of ν_1^n and L_c^n. The objective of Equation 6 is to minimise the error while preventing the network from growing too large or too small. For example, if the error is high, the algorithm increases Δ to reduce the error. If the error has converged, i.e. has not changed for two consecutive batches, Δ will be small.

The need for two pools, B_r and B_{ft} is justified as follows. The pool operation is designed to revise the existing knowledge. Thus, B_{ft} is composed of a diverse set of data batches that differ in the distribution of the data. The objective of the increment operation is to add the most recent features. B_r is ideal for this purpose as it contains the most recent data.

3.3.3 Reward Function

The reward function r^n is defined as,

$$r^n = \begin{cases} e^n - |\hat{\mu} - \nu_1^n| \text{ if } \nu_1^n < V_1 \\ e^n - |\hat{\mu} - \nu_1^n| \text{ if } \nu_1^n > V_2 \\ e^n; \text{ Otherwise,} \end{cases} \tag{9}$$

$$\text{where} \qquad e^n = (1 - (\mathrm{L}_c^n - \mathrm{L}_c^{n-1})) \times (1 - \mathrm{L}_c^n) \tag{10}$$

and V_1 and V_2 are predefined thresholds. e^n is specified so that the reward will be higher for lower errors and higher rates of error change (Equation 10). Equation 9 penalises r^n if the network grows too large or too small.

3.4 RA-DAE Algorithm

With S, A and r^n defined, we present the general approach used to solve the MDP (Algorithm 3). Q-Learning was utilised with the following steps,

For the n^{th} iteration, with data batch $\mathcal{D}^n$,

1. Until adequate samples are collected (i.e. $n \leq \eta_1$), train with B_r.

Figure 1. Random examples from the extended MNIST, CIFAR-10 and MNIST-rot-back datasets, respectively

2. With adequate samples collected (i.e. $n > \eta_1$), start calculating Q-values for each state-action pair observed $\{s^n, a^n\}$, where $s^n \in S$, and $a^n \in A$ as defined in Algorithm 3.
3. During $\eta_1 < n \leq \eta_2$, uniformly perform actions from $A = \{$*Increment*,*Merge*,*Pool*$\}$ to develop a fair utility estimate for all actions in A.
4. With an accurate estimation of Q (i.e. $n > \eta_2$), the best action a' is selected by $a' = argmax_{a'}(Q(s^n, a'))$ with a controlled amount of exploration (ϵ-greedy).
5. if $a' =$*Increment*, calculate ΔInc from Equation 7, add randomly initialised ΔInc nodes and greedily initialise *only* the new nodes with B_r, while keeping the rest constant.
6. if $a' =$*Merge* calculate ΔMrg from Equation 8 and average the closest pairs of ΔMrg nodes to amalgamate 2ΔMrg nodes to ΔMrg nodes.
7. if $a' =$*Pool* fine-tune the network with B_{ft}.
8. Train the network with $\mathcal{D}^n$.
9. Calculate the new state, s^{n+1} (Equation 4) and the reward r^n (Equation 10).
10. Update the Value (Utility) Function $Q(s, a)$ as,

$$Q^{(t+1)}(s^{n-1}, a^{n-1}) = (1-\alpha) \times Q^t(s^{n-1}, a^{n-1}) + \alpha \times q, \quad (11)$$

where $q = r^n + \gamma \times max_{a'}(Q^t(s^n, a'))$.

3.5 Function Approximation for Continuous Space

For clarity of presentation, Algorithms 2 and 3 assume discrete state space. However, the same algorithms can be extended for continuous state space. The idea is to, for a given action a and an unseen state $\tilde{s}$, predict the utility value $Q(\tilde{s}, a) = \hat{f}(\tilde{s}, \mathbf{w})$ through function approximation where $\hat{f}$ is the function and $\mathbf{w}$ is the approximated parameter vector [16]. In this paper, Gaussian Process Regression (GPR) [12] with squared exponential kernel, $k_{SE}(x, x') = \sigma^2(exp(-\frac{(x-x')^2}{2l^2}))$ has been used for this regression task. The hyperparameters σ and l are optimised by maximising the marginal likelihood w.r.t. the hyperparameters [12]. Formally, we collect at least $\eta_2 - \eta_1$ observed states and corresponding value pairs $\{s^n, Q(s^n, a^n)\}$. Next, for each $a \in A$, separate curves are fitted with GPR for the $\{s^n, Q(s^n, a^n)\}$ collection of pairs by separating pairs w.r.t. a^n, so that there are $|A|$ curves. Then, for an unseen state $\tilde{s}$ and a given action a', $Q(\tilde{s}, a')$ is calculated using the curve fitted for action a'. The continuous space is preferred as it provides a detailed representation of the environment with fewer variables, as opposed to the discrete space. This is sensible as the information extracted is continuous (e.g. L_g, L_c, ν).

Algorithm 3 Control Parameter Calculation algorithm

procedure GETCTRLPARAM(n, Q, s, a)
  define : n - Current batch ID
  define : Q - Utility function
  define : s, a - Previous state,action
  define : γ - Discount rate
  define : α - Learning rate
  if $n < \eta_1$ **then**
    return $null, Pool, Q, 0, 0$
  end if
  Calculate current state, s' (Equation 4)
  if $s, a \neq null$ **then**
    $q = r^n + \gamma \times max_{a'}(Q(s', a'))$
    $Q(s, a) = (1 - \alpha) \times Q(s, a) + \alpha \times q$
  end if
  if $n < \eta_2$ **then**
    Evenly chose action a' from $\in A$
  else
    Explore with ϵ-greedy ($\epsilon = 0.1$)
    OR
    $a' = argmax_{a'}(Q(s', a'))$
  end if
  Calculate ΔMrg and ΔInc (Eq. 7 and 8)
  return s', a', Q, ΔMrg,ΔInc
end procedure

3.6 Summary

Our proposed solution is detailed in Algorithm 2 and can be seen is a repeated application of Algorithm 3. For each batch of data $\mathcal{D}^n$, the state s^{n+1} and reward r^n is calculated using Equations 4 and 10 respectively. Next, the best action a' for the new state is retrieved by $a' = argmax'_a(Q(s^{n+1}, a')$. To calculate $Q(s^{n+1}, \hat{a})$ for some action $\hat{a} \in A$, GPR is employed as explained in Section 3.5. Next the action a' is performed. Then the network is fine-tuned using $\mathcal{D}^n$. This process is repeated until the end of the data stream.

4 Experiments

4.1 Overview and Setup

The experiments are based on extended versions of three datasets (i.e. MNIST[2], MNIST-rot-back[3] and CIFAR-10 [4]). Random samples from each dataset are depicted in Figure 1. The extended versions of each dataset consist of 1,000,000 examples. Examples were masked with noise during the generation to make them unique. We generated non-stationary distributions for each dataset using Gaussian processes (GP) [12], simulating the covariate shift effect. Formally, the ratio for each class of labels is generated using $ratio_k(t) = \frac{exp\{a_k(t)\}}{\sum_{j=1}^{K} exp\{a_j(t)\}}$ where $a_k(t)$ is a random curve generated by the GP.

Experiments were conducted with three different types of deep architectures; SDAE (Standard Denoising Autoencoders), MI-DAE (Merge-Incremental Denoising Autoencoders) and RA-DAE (our approach). For MI-DAE, we used a modified "update rule I" introduced

[2] http://yann.lecun.com/exdb/mnist/
[3] www.iro.umontreal.ca/ lisa/twiki/bin/view.cgi/Public/MnistVariations
[4] http://www.cs.toronto.edu/ kriz/cifar.html

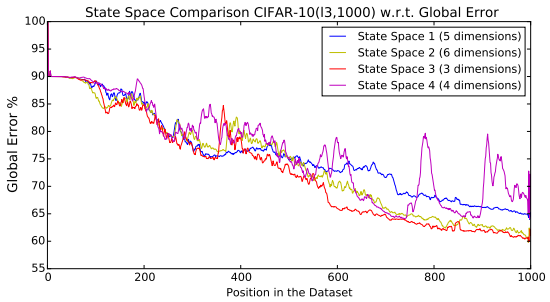

Figure 2. Analysis of Global Error E_{glb} for different state spaces for CIFAR-10 and a network with 3 layers with 1000 neurons on each. The mathematical definitions of state spaces 1,2,3 and 4 can be found in Section 4.2.1. It is clear that State Space 4 shows a steeper reduction of error compared to its counterparts.

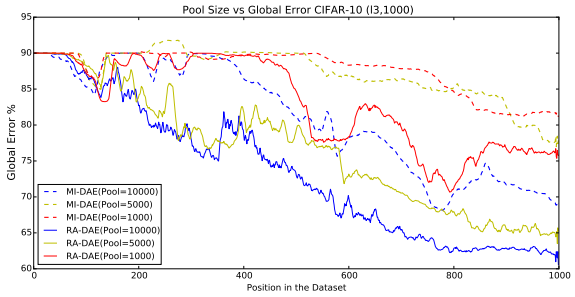

Figure 3. Performance of RA-DAE and MI-DAE for different pool sizes. Pool size of 10000 yielded the best results. It can be seen that RA-DAE with a pool size of 1000 performs similarly to MI-DAE with pool size 10000. This can be attributed to the learned policy and the pooling technique.

in [20] as they claim the performance is fairly robust to different update rules as follows,

$$\Delta \mathrm{N}_{t+1} = \begin{cases} \Delta \mathrm{N}_t + 30; \ , \ \frac{\mathrm{e}_t}{\mathrm{e}_{t-1}} < (1-\epsilon_1) \\ \Delta \mathrm{N}_t / 2; \ , \ \frac{\mathrm{e}_t}{\mathrm{e}_{t-1}} > (1-\epsilon_2) \\ \Delta \mathrm{N}_t, \ \text{Otherwise} \end{cases}$$

ΔMrg = $\lceil \gamma_{ratio} \Delta \mathrm{Inc} \rceil$; for $\gamma_{ratio} = 0.2$, as these modifications produced better performance.

Several initial layer configurations (hidden layer sizes) were used, as outlined in Table 2. To refer to a certain algorithm, we use the following notation. We use the superscript for the number of layers and the subscript to indicate the size of each layer. For example, $\mathrm{SDAE}^{l3}_{1500}$ denotes a SDAE with three layers and 1500 nodes in each layer. The configurations in Table 2 maximise the performance of the algorithms tested. The continuous state space (Equation 4) was used for all the experiments. We define two error measures for evaluating performance. A local error $\mathrm{E}_{lcl} = L_c^{n+1}$, measured on a validation set, $\mathcal{D}^{n+1}$ (batch succeeding the current batch) and a global error $\mathrm{E}_{glb} = \frac{\sum_{\forall i} L_c^i}{|\mathcal{D}_{test}|}$ s.t. $\mathcal{D}^i \in \mathcal{D}_{test}$ measured on an unseen independent test set $\mathcal{D}_{test}$, which contains an approximate uniform distribution of all the classes. These two sets of data enable us to respectively, evaluate how the network preserve immediate past knowledge and the globally accumulated knowledge.

All experiments were carried out using a Nvidia GeForce GTX TITAN GPU and Theano[5]. For all experiments we used 20% corruption level, 0.2 learning rate, batch size of 1000. We empirically chose $\gamma = 0.9$ (Equation 11) m (Equation 4) 30, and η_1 and η_2 (Algorithm 3) to be 30 and 60 respectively. Λ (Equation 3) was selected as 0.7 and 0.995 for non-stationary and stationary experiments respectively. $\tau = 10,000$ (for B_r, B_{ft} and B) was chosen from a set of sizes {1000, 5000, 10000} as 10,000 produced the best results. Results are depicted in Figure 3.

4.2 Results

4.2.1 *Evaluation of State Spaces*

As mentioned in Section 3.3.1 the state space was chosen while paying close attention to the performance against an unseen data batch, difference between observed data distributions and complexity of the network. We utilised various quantifiable measures. L_g^{n+1} and L_c^{n+1} were employed to evaluate RA-DAE's ability to classify an unseen batch of data (i.e. $\mathcal{D}^{n+1}$). Kullback-Leibler divergence ($D_{KL}(P^n||Q^n)$) [8] was used to measure the divergence between the distribution of current data and previously fed data; $D_{KL}(P^n||Q^n) = \sum_i^K P^n(i) log(\frac{P^n(i)}{Q^n(i)})$, where $P^n(i) = \frac{Count_i^n}{p}$, $Count_i^n$ is the number of data points with class i in $\mathcal{D}^n$, p is as defined in Table 1 and $Q^n(i) = \frac{\sum_{j-m}^{j} P^j(i)}{m}$. Finally the complexity of RA-DAE at a given time is captured by ν^n.

Table 2. Initial layer configurations for different datasets. The superscript of the algorithm name specifies the number of layers and the subscript indicates the size of each layer.

MNIST	CIFAR-10	MNIST-rot-back
SDAE^{l1}_{500}	$\mathrm{SDAE}^{l1}_{1000}$	$\mathrm{SDAE}^{l1}_{1500}$
SDAE^{l3}_{500}	$\mathrm{SDAE}^{l3}_{1000}$	$\mathrm{SDAE}^{l3}_{1500}$
$\mathrm{MI\text{-}DAE}^{l1}_{500}$	$\mathrm{MI\text{-}DAE}^{l1}_{1000}$	$\mathrm{MI\text{-}DAE}^{l1}_{1500}$
$\mathrm{MI\text{-}DAE}^{l3}_{500}$	$\mathrm{MI\text{-}DAE}^{l3}_{1000}$	$\mathrm{MI\text{-}DAE}^{l3}_{1500}$
$\mathrm{RA\text{-}DAE}^{l1}_{500}$	$\mathrm{RA\text{-}DAE}^{l1}_{1000}$	$\mathrm{RA\text{-}DAE}^{l1}_{1500}$
$\mathrm{RA\text{-}DAE}^{l3}_{500}$	$\mathrm{RA\text{-}DAE}^{l3}_{1000}$	$\mathrm{RA\text{-}DAE}^{l3}_{1500}$

With the aforementioned quantities defined, the following state spaces were defined:

- State Space 1 - $\{\tilde{\mathcal{L}}_g(m_3), \tilde{\mathcal{L}}_c(m_1), \tilde{\mathcal{L}}_c(m_2), \tilde{\mathcal{L}}_c(m_3), \nu\}$
- State Space 2 - $\{\tilde{\mathcal{L}}_g(m_3), \tilde{\mathcal{L}}_c(m_1), \tilde{\mathcal{L}}_c(m_2), \tilde{\mathcal{L}}_c(m_3), \nu, D_{KL}(P^n||Q^n)\}$
- State Space 3 - $\{\tilde{\mathcal{L}}_g(m), \tilde{\mathcal{L}}_c(m), \nu\}$
- State Space 4 - $\{\tilde{\mathcal{L}}_g(m), \tilde{\mathcal{L}}_c(m), \nu, D_{KL}(P^n||Q^n)\}$

The constants m_1, m_2, m_3 and m were chosen empirically and set to 5,15,30 and 30 respectively. The reason for calculating $\tilde{\mathcal{L}}$ for several m values is to learn whether augmenting the state space of L_g and L_c contribute additional information. However, from the experimental results, it was evident that a simpler state space yields the best results. Furthermore, it was surprising to verify that $D_{KL}(P^n||Q^n)$ had no significant positive impact on the results. The performance of different state spaces is depicted in Figure 2.

4.2.2 *Analysis of Structure Adaptation*

We studied the adaptation pattern of RA-DAE and MI-DAE in both stationary and non-stationary environments. Figure 5(c) depicts the number of nodes in the first layer for MI-DAE and RA-DAE as they adapt to data distributions changes with the CIFAR-10 dataset. In non-stationary problems, RA-DAE exhibits repeated peaks in the number of nodes. This can be explained by the changes in class distribution in Figure 5(f). Node number changes in Figure 5(c) align with

[5] http://deeplearning.net/software/theano/

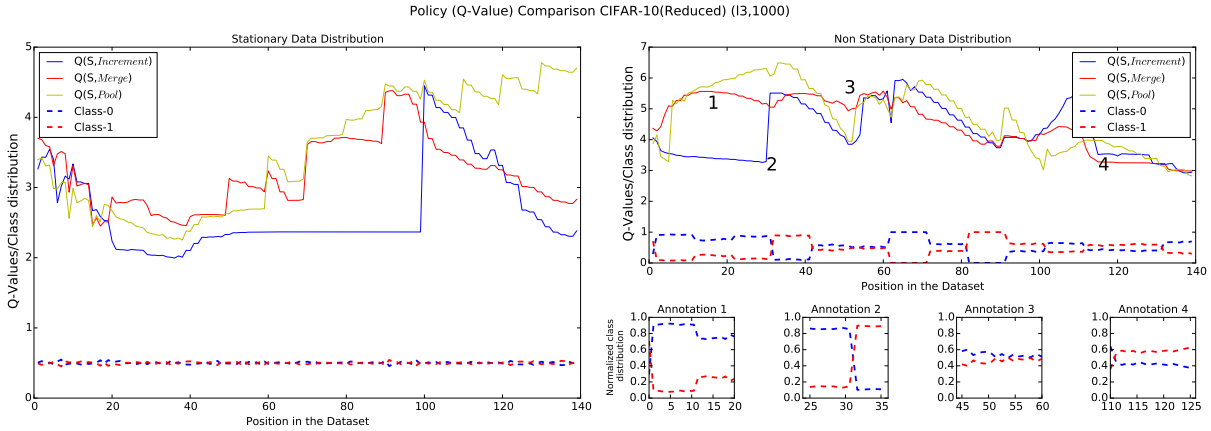

Figure 4. Visualisation of the value function ($Q(s, a)$) evolution for stationary and non-stationary distributions. Annotations on the top-right graph indicate note-worthy behaviours of the value function. The left and top-right graphs depict the complete progression of the data distribution. For clarity, a reduced version of CIFAR-10 with only two classes and 200,000 examples was used. For stationary data distribution, the graph indicates how *Pool* and *Merge* operations dominate the behaviour as there are no significant data distribution changes. In the non-stationary setting, the value function for action *Increment* surges in face of a sharp distribution change (Annotation 2). *Merge* and *Pool* operations take over when data distributions are consistent (Annotation 3 and 4 respectively).

the peaks appearing for various class distributions in Figure 5(f). For sharp distribution changes, RA-DAE quickly increases the number of neurons. However, MI-DAE shows a moderate growth in the number of nodes, despite the rapid changes in the data distribution. This demonstrates that RA-DAE is more responsive than MI-DAE in adapting the architecture in the face of changes. For a stationary data distribution, MI-DAE shows a constant node count after the first few hundred batches, where RA-DAE increases the number of nodes over time. This can be attributed to the fact that reducing the number of neurons tends to increase the error, occasionally making the reduction operation not preferable to RA-DAE. This is acceptable as RA-DAE will not increase nodes unnecessarily as it would lead to poor results due to *overfitting*. An alternative is to perform the pool operation after reduce, which would reduce the error at an increased computational cost.

4.2.3 Analysis of Local and Global Error

Finally, the capability to preserve past knowledge, balancing immediate and global rewards for the algorithms was assessed by using the local error, E_{lcl}, and the global error, E_{glb}. We used the hybrid objective function ($L_{disc} + \lambda L_{gen}$ for $\lambda = 0.2$) [19] to fine-tune the network.

Figure 5 depicts several interesting results. Figure 5(a) illustrates the behaviour of the E_{lcl}. RA-DAE$^{l1}_{500}$ shows a clear improvement w.r.t E_{lcl} over time. Note how in RA-DAE$^{l1}_{500}$ the fluctuations shrink over time. Moreover, Figure 5(d) delineates a significant E_{glb} error margin maintained by RA-DAE$^{l1}_{500}$ compared to SDAE$^{l1}_{500}$ and MI-DAE$^{l1}_{500}$. RA-DAE's ability to grow the network faster compared to MI-DAE explains this significantly lower error. Figure 5(b) and (e) portray the performance of the algorithm in a stationary environment (CIFAR-10). Though we expected all algorithms to perform comparably well in the stationary environment, RA-DAE$^{l3}_{1000}$ achieves the lowest E_{lcl} and E_{glb} and the steepest error reduction. Both RA-DAE$^{l3}_{1000}$ and MI-DAE$^{l3}_{1000}$ demonstrate better performance than SDAE$^{l3}_{1000}$. This highlights that structure adaptation strategies enhance the performance of deep networks in both stationary and non-stationary environments.

Table 3 summarises the errors (mean$\pm$standard deviation of the last 250 batches) for various datasets. The number 250 was chosen, as the last 250 batches displayed a consistent performance in most instances. There are several key observations from Table 3. First, RA-DAE^{l3} has outperformed its counterparts in both stationary and non-stationary scenarios, where RA-DAE^{l1} and MI-DAE^{l1} have performed equally well. By observing the performance of RA-DAE^{l1} and RA-DAE^{l3} it is evident that the performance of RA-DAE has improved as the network becomes deeper. MI-DAE has exhibited the same property in most occasions. The rationale being, not only deep networks are more robust to structural modifications in terms of error, but also they are able to learn more descriptive representations as depth increases. However, performance of SDAE^{l3} is worse than SDAE^{l1} in both cases. This observation justifies the need for better techniques to leverage deep architectures in online scenarios.

A surprising observation can be made in $\{$SDAE,MI-DAE,RA-DAE$\}^{l1}$ for MNIST-rot-back. Even though we expected RA-DAE to perform the best, SDAE^{l1} shows the best performance with a $52.8 \pm 7.0\%$ and $65.7 \pm 2.7\%$ for E_{lcl} and E_{glb} respectively. Close examination of the behaviours of E_{lcl} and E_{glb} of SDAE, MI-DAE and RA-DAE, shows that MI-DAE and RA-DAE do not perform as well as SDAE. This is due to the fluctuation of E_{lcl} being fast, which causes the algorithm to increase the number of nodes unnecessarily. Consequently, MI-DAE and RA-DAE lead to poor accuracy due to *overfitting*. This issue alleviates as the network becomes deeper.

4.2.4 Analysis of the Policy Learnt

In order to analyse the policy learnt by RA-DAE, it is imperative to take a close look at the value function (i.e. $Q(s, a)$) learnt by RA-DAE. Figure 4 depicts the evolution of the value function over time with note-worthy behaviours annotated. For the purpose of visualisation, a simplified version of CIFAR-10 dataset (CIFAR-10-bin) has been used. CIFAR-10-bin comprises only two classes and has a total of 200,000 data points. Figure 4 depicts the value function for two settings; stationary and non stationary. The annotation graphs at top-right highlight the changes in data distribution at the points of interest in the top graph.

In the stationary setting, it can be seen that *Pool* and *Merge* operations have dominated the policy, Figure 4(right). This is sensible as the data distribution stays constant throughout and a necessity to increase the number of nodes hardly emerges.

Table 3. This table presents the E_{lcl} and E_{glb} obtained for various datasets and depths. Errors are in the format of mean±standard deviation for the last 250 batches. The lowest errors are highlighted in bold. RA-DAE has shown the best performance (smallest local and global errors) in most occasions (for both stationary and non-stationary).

	MNIST		CIFAR-10		MNIST-rot-back		CIFAR-10 (Stationary)	
	$E_{lcl}\%$	$E_{glb}\%$	$E_{lcl}\%$	$E_{glb}\%$	$E_{lcl}\%$	$E_{glb}\%$	$E_{lcl}\%$	$E_{glb}\%$
SDAE^{l1}	10.9 ± 5.8	27.2 ± 5.7	65.9 ± 4.9	82.8 ± 1.1	$\mathbf{52.8 \pm 7.0}$	$\mathbf{65.7 \pm 2.7}$	67.9 ± 1.5	70.2 ± 0.6
MI-DAE^{l1}	6.4 ± 3.2	23.9 ± 4.4	$\mathbf{50.4 \pm 4.7}$	$\mathbf{74.9 \pm 3.0}$	61.8 ± 9.0	72.0 ± 2.6	55.5 ± 1.9	$\mathbf{61.4 \pm 0.8}$
RA-DAE^{l1}	$\mathbf{5.1 \pm 1.4}$	$\mathbf{11.3 \pm 0.7}$	$\mathbf{50.6 \pm 7.2}$	$\mathbf{74.0 \pm 2.4}$	62.3 ± 8.8	69.6 ± 2.8	59.8 ± 1.9	$\mathbf{61.9 \pm 1.1}$
SDAE^{l3}	11.2 ± 5.9	31.6 ± 5.4	76.3 ± 6.6	88.4 ± 1.9	67.6 ± 8.9	77.1 ± 3.2	71.8 ± 1.4	72.7 ± 0.7
MI-DAE^{l3}	5.4 ± 4.4	31.3 ± 4.0	43.7 ± 8.5	71.0 ± 1.6	56.0 ± 9.2	65.5 ± 2.1	56.1 ± 1.8	58.9 ± 1.1
RA-DAE^{l3}	$\mathbf{4.1 \pm 3.0}$	$\mathbf{13.4 \pm 0.1}$	$\mathbf{32.4 \pm 8.0}$	$\mathbf{62.7 \pm 0.7}$	$\mathbf{48.2 \pm 9.2}$	$\mathbf{60.6 \pm 3.4}$	$\mathbf{50.6 \pm 2.1}$	$\mathbf{53.6 \pm 2.1}$

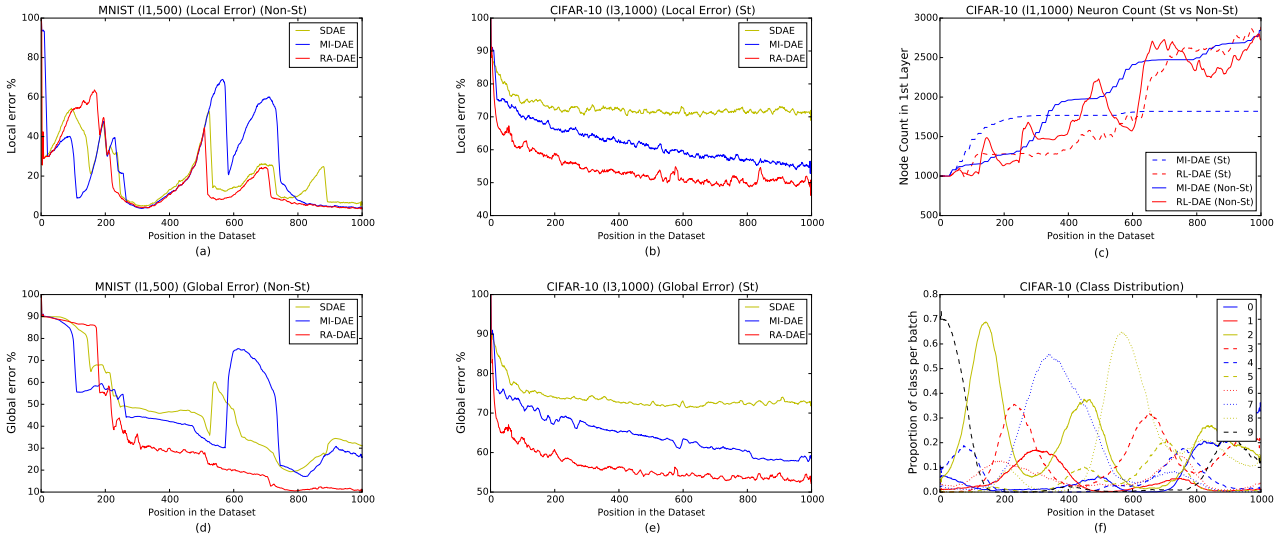

Figure 5. (a) and (d) show the behaviour of E_{lcl} and E_{glb} in a non-stationary (Non-St) situation, where (b) and (e) show the behaviour of E_{lcl} and E_{glb} in a stationary (St) situation. The titles consist of the name of the dataset followed by number of hidden layers and neuron count in each layer, within parenthesis. RA-DAE exhibits the lowest E_{lcl} and E_{glb} at the end, and a more consistent reduction compared to SDAE and MI-DAE. (c) presents node adaptation patterns of MI-DAE and RA-DAE in both stationary and non-stationary situations. (f) shows the class distribution of data over time and each curve denotes a single class. By comparing to (f), (c) clearly indicates that RA-DAE is more sensitive to changes in data distribution than MI-DAE in terms of the neuron adaptation. The horizontal axis represents the number of batches in the training dataset.

For the non-stationary setting, it can be seen how *Pool* and *Merge* operations have a high value as the algorithm has not seen an significant data distributions, thus suppressing *Increment* operation. Next, at annotation 2 it can be seen how the value of *Increment* operation boosts up due to the massive data distribution change. Then, at point 3, *Merge* operation takes over as data distribution is somewhat consistent. And finally, at point 4, *Pool* operation dominates the graph due to the consistency of the distribution of data.

5 Conclusion

Online learning can be widely beneficial for deep architectures as it allows network adaptation for streaming data problems. However, defining the structure of the network, including number of nodes, can be difficult to do in advance. To address this, [19] introduces MI-DAE which can dynamically change the structure of the network but relies on simple heuristics. The novelty of this work is an online learning stacked denoising autoencoder which leverages reinforcement learning to modify the structure of the deep network. In this, we use a model-free reinforcement learning approach and calculate a utility function for actions by sampling from the incoming states.

Compared to the counterpart, our approach is more principled and responsive in adapting to new information. The method leverages RL to make decisions in a dynamic fashion. The control behaviour combined with powerful pooling techniques allows our approach to preserve past-knowledge effectively. Finally, our solution make decisions based on long-term versus immediate reward. Experimental results indicate that our solution often outperforms its counterparts with a lower classification error, and the performance improves as the network becomes deeper. Also, the approach is more sensitive to changes in the data distribution. Future work will address other deep learning architectures such as convolutional neural nets and deep Boltzmann machines.

Acknowledgements

This research was supported by funding from the Faculty of Engineering & Information Technologies, The University of Sydney, under the Faculty Research Cluster Program. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GPU used for this research.

REFERENCES

[1] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al., 'Greedy layer-wise training of deep networks', *Advances in neural information processing systems*, **19**, 153, (2007).

[2] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber, 'Multi-column deep neural network for traffic sign classification', *Neural Networks*, **32**, 333–338, (2012).

[3] Li Deng and Dong Yu, 'Deep learning: Methods and applications', Technical Report MSR-TR-2014-21, (May 2014).

[4] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al., 'Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups', *Signal Processing Magazine, IEEE*, **29**(6), 82–97, (2012).

[5] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh, 'A fast learning algorithm for deep belief nets', *Neural computation*, **18**(7), 1527–1554, (2006).

[6] Geoffrey Hinton and Ruslan Salakhutdinov, 'Reducing the dimensionality of data with neural networks', *Science*, **313**(5786), 504–507, (2006).

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 'Imagenet classification with deep convolutional neural networks', in *Advances in neural information processing systems*, pp. 1097–1105, (2012).

[8] Solomon Kullback and Richard A Leibler, 'On information and sufficiency', *The annals of mathematical statistics*, **22**(1), 79–86, (1951).

[9] Xinwang Liu, Guomin Zhang, Yubin Zhan, and En Zhu, 'An incremental feature learning algorithm based on least square support vector machine', in *Frontiers in Algorithmics*, 330–338, Springer, (2008).

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, 'Playing atari with deep reinforcement learning', *arXiv preprint arXiv:1312.5602*, (2013).

[11] Tomaso Poggio and Gert Cauwenberghs, 'Incremental and decremental support vector machine learning', *Advances in neural information processing systems*, **13**, 409, (2001).

[12] Carl Edward Rasmussen, 'Gaussian processes for machine learning', (2006).

[13] Martin Riedmiller, 'Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method', in *Machine Learning: ECML 2005*, 317–328, Springer, (2005).

[14] Kenneth O Stanley and Risto Miikkulainen, 'Evolving neural networks through augmenting topologies', *Evolutionary computation*, **10**(2), 99–127, (2002).

[15] Masashi Sugiyama and Motoaki Kawanabe, *Machine learning in non-stationary environments: Introduction to covariate shift adaptation*, MIT Press, 2012.

[16] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*, volume 1, MIT press Cambridge, 1998.

[17] Sebastian Thrun and Lorien Pratt, *Learning to learn*, Springer Science & Business Media, 2012.

[18] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol, 'Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion', *The Journal of Machine Learning Research*, **11**, 3371–3408, (2010).

[19] Guanyu Zhou, Kihyuk Sohn, and Honglak Lee, 'Online incremental feature learning with denoising autoencoders', in *International Conference on Artificial Intelligence and Statistics*, pp. 1453–1461, (2012).

[20] Guanyu Zhou, Kihyuk Sohn, and Honglak Lee, 'Supplementary material: Online incremental feature learning with denoising autoencoders', in *International Conference on Artificial Intelligence and Statistics*, (2012).