UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Gean Marciel dos Santos Stein

MODELO DE REDE NEURAL DE MÚLTIPLAS ESCALAS DE TEMPO APLICADO AO ROBÔ DIMITRI

Santa Maria, RS, Brasil
2017

Gean Marciel dos Santos Stein

MODELO DE REDE NEURAL DE MÚLTIPLAS ESCALAS DE TEMPO APLICADO AO ROBÔ DIMITRI

Trabalho de conclusão apresentado ao Curso de Engenharia de Controle e Automação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do título de **Bacharel em Controle e Automação.**

Orientador: Rodrigo da Silva Guerra

Santa Maria, RS, Brasil
2017

Gean Marciel dos Santos Stein

MODELO DE REDE NEURAL DE MÚLTIPLAS ESCALAS DE TEMPO APLICADO AO ROBÔ DIMITRI

Trabalho de conclusão apresentado ao Curso de Engenharia de Controle e Automação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do título de **Bacharel em Controle e Automação.**

Aprovado em 22 de fevereiro de 2017

_____
Prof. Rodrigo da Silva Guerra, PhD (UFSM)
(Presidente/Orientador)

_____
Prof. Giovani Rubert Librelotto, PhD (UFSM)

_____
Prof. Luís Alvaro de Lima Silva, PhD (UFSM)

Santa Maria, RS, Brasil
2017

RESUMO

MODELO DE REDE NEURAL DE MÚLTIPLAS ESCALAS DE TEMPO APLICADO NO ROBÔ DIMITRI

AUTOR: Gean Marciel dos Santos Stein
ORIENTADOR: Rodrigo da Silva Guerra

A proposta para a implementação e aplicação do modelo de rede neural recorrente de múltiplas escalas de tempo (MTRNN – *Multiple Timescale Recurrent Neural Network*) surgiu de experimentos na área de aprendizagem de máquina (Peniak *et al*. 2011; Nobuta *et al*. 2011), nos quais um robô humanoide aprende atividades complexas através da interação com humanos. O modelo MTRNN (Yamashita e Tani, 2008) se distingue por possuir dois tipos de unidades principais, os neurônios rápidos e os neurônios lentos. Essa característica da rede a torna capaz de codificar e reproduzir em detalhe fenômenos dinâmicos nos neurônios rápidos e ao mesmo tempo representar estes fenômenos em um grau de abstração mais alto nos neurônios mais lentos, trazendo ideia análoga à intencionalidade das ações observadas em humanos e em outros animais. O objetivo geral do trabalho é a aplicação do modelo MTRNN no robô Dimitri, recentemente desenvolvido no Centro de Tecnologia da UFSM. A rede neural foi implementada em Python, tendo sua implementação iniciada em um trabalho anterior. O treinamento da rede utiliza o algoritmo de aprendizagem *Backpropagation Through Time* (BPTT) otimizado pelo método de gradiente acelerado de Nesterov. Os dados usados para o treinamento da rede referem-se à posição dos servo-motores ao serem manipulados por um tutor, o qual irá executar movimentos cíclicos. Foram realizados experimentos a fim de testar a rede e suas características. A rede implementada foi capaz de aprender três padrões de movimentos diferentes, mas foi incapaz de operar em malha fechada devido à sensibilidade a distúrbios, sendo assim a rede não está pronta para ser aplicada diretamente no robô Dimitri.

Palavras-chave: Aprendizado de Máquina. Rede Neural. MTRNN. Dimitri.

ABSTRACT

MULTIPLE TIMESCALE RECURRENT NETWORK MODEL APPLIED ON THE DIMITRI ROBOT

AUTHOR: Gean Marciel dos Santos Stein
ADVISOR: Rodrigo da Silva Guerra

The idea for the implementation and application of the Multiple Timescale Recurrent Network (MTRNN) model started with experiments based on machine learning (Peniak et al., 2011; Nobuta et al., 2011). In which a humanoid robot learns complex activities through interaction with humans. The MTRNN model (Yamashita and Tani, 2008) is distinguished by having two main types of units, the fast neurons and the slow neurons. This characteristic of the network makes it able to encode and reproduce in detail dynamic phenomena in fast neurons and at the same time represent these phenomena in a higher degree of abstraction in the slower neurons, bringing an idea analogous to the intentionality of the actions observed in humans and in others Animals. The main goal of this work is the application of the MTRNN model in the Dimitri robot, recently developed at the UFSM Technology Center. The neural network was implemented in Python, and its implementation started in a previous work. The network training uses the Backpropagation Through Time (BPTT) learning algorithm optimized by Nesterov's accelerated gradient method. The data used for the training of the network refer to the position of the servo motors when being manipulated by a tutor, who will perform cyclical movements. Experiments were carried out to test the network and its characteristics. The implemented network was able to learn three different movement patterns, but was unable to operate in closed loop due to sensitivity to disturbances, so the network is not ready to be applied directly to the Dimitri robot.

Keywords: Neural Network. Machine Learning. MTRNN. Dimitri.

SUMÁRIO

1 INTRODUÇÃO

O primeiro modelo computacional para redes neurais baseado em matemática e algoritmos data de 1943. Desde então, não só o poder de computação têm aumentado, como também os modelos de redes neurais têm avançado. O estudo de redes neurais artificiais pode ser classificado em duas grandes linhas, uma delas focada na aplicação de redes artificiais para solução de problemas ligados à aprendizagem de máquina e à inteligência artificial, enquanto a outra foca nos processos biológicos do cérebro e do sistema nervoso de diferentes animais, o modelo de rede proposto (YAMASHITA & TANI, 2008) que baseia este trabalho, de certa forma especula o funcionamento do cérebro humano, não buscando uma fidelidade com a biologia, mas ainda assim tenta mostrar como, em princípio, uma arquitetura baseada em diferentes escalas de tempo pode organizar tarefas recombinando segmentos de atividades na forma de um plano mais complexo.

O aprendizado de máquina vem sendo usado para a solução de uma variedade de tarefas, envolvendo visão computacional, reconhecimento de fala, classificação de imagens, entre outros. O tema deste trabalho vem da robótica, mais precisamente do experimento realizado por Yamashita e Tani (2008), no qual um robô humanoide foi usado para o aprendizado de atividades complexas usando um modelo de rede neural proposto pelos autores do experimento: a rede neural recorrente de múltiplas escalas de tempo (MTRNN).

Uma atividade complexa pode ser decomposta em uma sequência de atividades mais simples. Por exemplo, o ato de pegar o copo pode ser dividido em estender o braço, abrir mão e agarrar o copo, onde cada movimento desses pode ser considerado um movimento primitivo. O modelo MTRNN se distingue por possuir dois tipos de unidades principais, os neurônios rápidos e os neurônios lentos. Essa característica da rede a torna capaz de codificar e reproduzir em detalhe fenômenos dinâmicos nos neurônios rápidos e ao mesmo tempo representar estes fenômenos em um grau de abstração mais alto nos neurônios mais lentos, trazendo ideia análoga à intencionalidade das ações observadas em humanos e em outros animais. No exemplo anterior a intenção seria pegar o copo, essa intenção seria codificada nas unidades lentas as quais se tornam responsáveis por garantir um objetivo de mais longo prazo, coordenando, na sua dinâmica mais gradual, um plano de sequências de ações nas unidades rápidas responsáveis pelos movimentos primitivos. Essa diferença entre os neurônios rápidos

e lentos dá à rede a capacidade de auto-organização, ou seja, as unidades lentas são capazes de reorganizar os movimentos primitivos previamente aprendidos em novos movimentos complexos.

A habilidade da rede em “aprender a intenção” do sinal faz com que a MTRNN possa ser aplicada em vários campos, sendo o mais pesquisado a interação homem-robô. O crescimento de pesquisas nesta área também traz a necessidade que a interação ocorra de forma segura, detectando a força com que o robô manipula os objetos. Uma parceria firmada entre KAIST e UFSM possibilitou o desenvolvimento do robô humanoide Dimitri, o qual utiliza atuadores de série elástica desenvolvidos no Centro de Tecnologia da UFSM. Esse tipo de atuador oferece um nível de segurança maior, pois insere um elemento elástico entre o motor e a carga, sendo a força medida diretamente pela deformação da peça plástica.

Em um trabalho anterior deste mesmo autor (STEIN, 2016) a rede MTRNN foi implementada para gestos dinâmicos bidimensionais. onde os dados utilizados para o treinamento da rede eram coletados usando uma tela sensível ao toque, nela eram desenhadas figuras bidimensionais, como círculo, quadrado ou triângulo, tendo os deslocamentos do cursor nos eixos x e y gravados para uso no treinamento da rede. O trabalho tinha como objetivo implementar o modelo MTRNN à partir de uma rede recorrente discreta, aumentando aos poucos a complexidade do algoritmo até chegar no modelo proposto, o qual seria adaptado para outras finalidades em trabalhos futuros.

Já para o presente trabalho o objetivo geral é a adaptação e aplicação do modelo MTRNN no robô Dimitri, replicando o experimento descrito em (YAMASHITA; TANI, 2008). A rede neural foi implementada em Python e adaptada de forma a possibilitar o treinamento usando os dados adquiridos com os servo-motores do Dimitri. O treinamento da rede utiliza o algoritmo de aprendizagem *Backpropagation Through Time* (BPTT) (WERBOS, 1990) otimizado pelo método de gradiente acelerado de Nesterov (SUTSKEVER, 2013). Os dados usados para o treinamento da rede referem-se à posição dos servo-motores ao serem manipulados por um tutor, o qual irá demonstrar movimentos cíclicos, movimentando os braços do robô como forma de treinamento.

A estrutura do restante do texto deste trabalho tem a seguinte organização: após a Introdução, no Capítulo 2 é apresentada a Revisão Bibliográfica, nela são abordados os conceitos básicos de uma rede neural, redes recorrente em tempo discreto e contínuo, e também a estrutura e propriedades da MTRNN. Ainda no Capítulo 2 também serão abordadas as especificações do robô Dimitri e os atuadores de série elástica. No Capítulo 3 são

abordados os métodos e os materiais utilizados para a implementação das rede, as adaptações feitas e os experimentos realizados com o Dimitri. No Capítulo 4 os resultados são apresentados e discutidos. Por fim, seguem as Conclusões e Referências.

2 REVISÃO BIBLIOGRÁFICA

A estrutura da revisão bibliográfica foi dividida de modo com que os tópicos forneçam uma base para o entendimento de redes neurais, onde serão apresentadas suas principais características. Em seguida faz-se uma revisão sobre MTRNN, expondo sua arquitetura e propriedades. Por fim, são apresentadas algumas características do robô Dimitri.

2.1 REDES NEURAIS ARTIFICIAIS

Redes neurais artificiais (ANN) são modelos inspirados em redes neurais biológicas, consistindo basicamente de uma rede de neurônios artificiais, os quais são uma aproximação dos neurônios encontrados no cérebro de seres vivos. Uma rede neural pode ser definida como:

> ...um sistema de computação composta de um número de elementos de processamento simples, altamente interligados, que processam informação pela resposta dinâmica de estado às entradas externas. (CAUDILL M., 1989)

Ou seja, um neurônio artificial é uma aproximação muito simples do verdadeiro, mas uma rede dos mesmos pode ser extremamente poderosa em dispositivos computacionais. As conexões entre os neurônios possuem pesos numéricos que podem ser ajustados com base na experiência, o que faz as ANNs adaptáveis às entradas e capazes de aprender. Por essa razão ANNs são geralmente usadas em aplicações ligadas ao aprendizado de máquina e ao reconhecimento de padrões.

2.1.1 Vantagens e características

Uma das principais vantagens das ANNs se deve à estrutura de processamento distribuído paralelo e à habilidade de aprender e, portanto, generalizar. Por generalizar entende-se a produção, pela rede neural, de saídas aceitáveis referentes às entradas que não foram encontradas durante a etapa de treinamento. Outras propriedades das redes neurais (Haykin, S.S. 2009) são: a não linearidade, uma ANN pode ser linear ou não linear, importante propriedade para casos onde o sinal de entrada não é linear; adaptabilidade, uma

rede neural pode mudar o peso de suas sinapses, aprendendo e se auto-organizando com as mudanças de ambiente; é tolerante a falhas, mesmo parcialmente danificada uma rede neural pode continuar a responder de uma forma aceitável; e robustez, ANNs conseguem lidar com pequenas mudanças nos sinais de entrada.

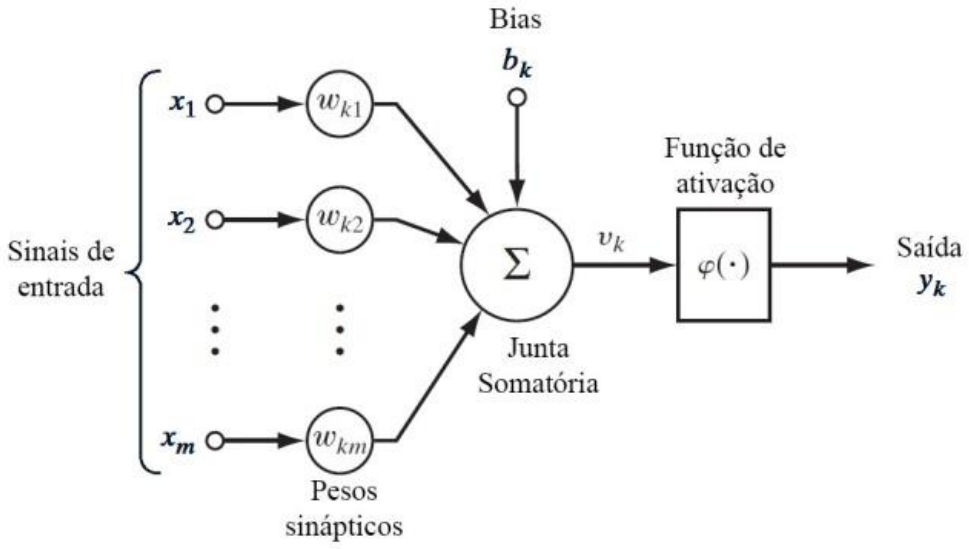
A unidade principal de uma ANN é o neurônio, ilustrado na Figura 1. Nesta figura são mostradas as funções fundamentais do neurônio, bem como suas entradas e sua saída. A função de entrada computa a soma dos valores de entrada multiplicados previamente pelos pesos das conexões. Já a função de ativação é uma função não linear, sendo geralmente uma função sigmoide (Equação 1) ou tangente hiperbólica. A saída do neurônio é representada pela Equação 2.

$$g(x) = 1/(1 + e^{-x}) \tag{1}$$

$$a_i = g(x_i) = g(\sum_j W_{ij} a_j) \tag{2}$$

Como dito anteriormente, as sinapses são representadas por pesos numéricos, que são responsáveis pela transmissão do sinal de uma unidade para outra, ou para ela mesma no caso de redes recorrentes. Podemos reunir os pesos em uma matriz facilitando assim o cálculo das funções de entrada e ativação.

Figura 1 – Esquema de um neurônio



Fonte: Haykin, S. (2009)

Na Tabela 1 são listadas algumas das notações usadas na representação de ANN:

Tabela 1 – Notação para redes neurais

Notação	Significado
x_n^i	Entrada na unidade i no tempo n
a_n^i	Ativação da unidade i no tempo n
b_i	*Bias* da unidade i
g *ou* φ	Função de ativação
g' *ou* φ'	Derivada da função de ativação
E_n^i	Erro (diferença entre a saída observada e a saída alvo) da unidade i
u_n^i	Soma das entradas na unidade i no tempo n após serem pesadas, estado interno da unidade
y_n^i	Ativação na unidade i no tempo n na camada de saída
T_n^i *ou* y_n^*	Valor de saída alvo no tempo n
W_{ij}	Peso na conexão da unidade i vindo da unidade j

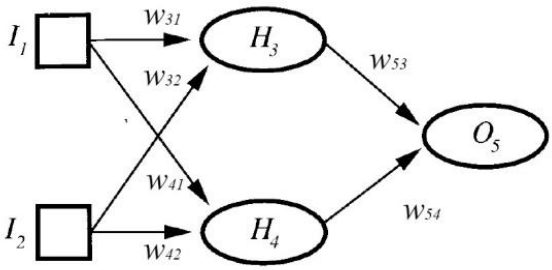
Uma rede neural artificial é tipicamente definida por três parâmetros:

- Topologia, ou seja, o padrão de conexões entre as diferentes camadas de neurônios (os pesos dessas conexões são as chamadas sinapses)
- A função de ativação que converte as entradas do neurônio multiplicadas por pesos em sua saída de ativação
- O processo de aprendizado para atualização dos pesos das conexões

2.1.2 Topologias das redes

Existem vários tipos de topologias de redes neurais, mas os dois tipos principais são *feedforward* e recorrente (RNN). As redes do primeiro modelo possuem conexões unidirecionais e são acíclicas. Geralmente elas são divididas em camadas, onde um neurônio de uma camada tem a saída conectada somente com neurônios da próxima camada, sem conexões com unidades da mesma camada ou da camada anterior. Uma típica rede *feedforward* é dividida em três conjuntos: unidades de entrada (*input units*), camadas ocultas (*hidden units*) e camada de saída (*output units*). Na Figura 2 é representado uma ANN *feedforward*.

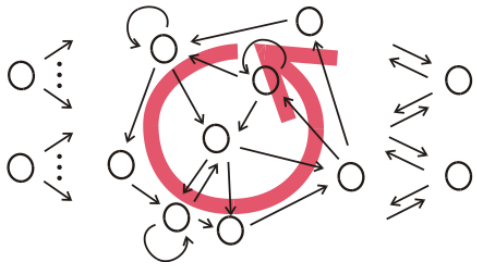
Figura 2 – Modelo de uma ANN *feedforward*



Fonte: Russell, S. J., & Norvig, P. (1995)

Outra estrutura comum nas redes neurais é a rede recorrente, onde um neurônio pode se conectar com outros de camadas diferentes ou com ele mesmo, sendo assim as conexões em uma RNN são ciclos direcionados. RNNs são mais complexas de treinar, podem tornar-se instáveis ou exibir comportamento caótico, mas por outro lado oferecem a possibilidade de modelar sistemas com estados dependentes do tempo ou sequencias. Na Figura 3 é mostrado um exemplo de RNN.

Figura 3 - Exemplo de conexões em uma rede recorrente



Fonte: Guo, J. 2013

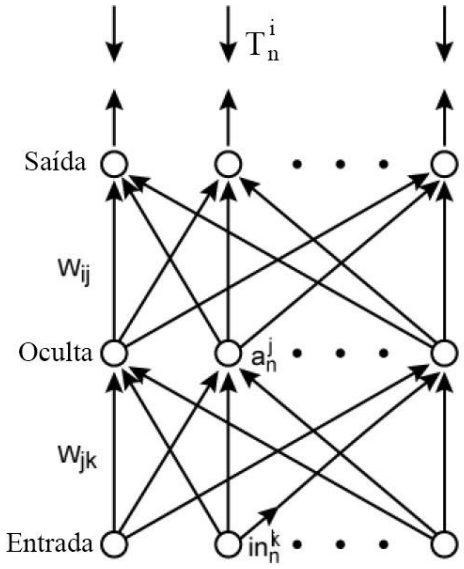
2.1.3 Aprendizado

Pode-se dizer que o conhecimento de uma ANN está nos pesos que conectam os neurônios, pois ao fazer o treinamento da rede, os pesos são constantemente modificadas. O algoritmo de apredizado mais comum para redes *feedforward* é o *backpropagation*. O treinamento da rede *feedforward* pelo por *backpropagation* pode ser dividido em três etapas, na primeira ocorre a propagação *forward* dos dados de treinamento pela rede, ao final dessa etapa a saída da rede será comparada com o valor de saída desejado, essa comparação é feita

com a ajuda de uma função custo a qual varia conforme as regras aplicadas, sendo usualmente o erro quadrático médio entre a saída observada e a saída alvo. O objetivo do treinamento da rede é diminuir o valor da função custo, pois caso a saída observada seja igual a saída desejada, o erro é nulo, ou seja, a rede previu perfeitamente o próximo passo do sinal. Na segunda etapa, o erro encontrado na saída da rede é propagado para as camadas anteriores, com o intuito de calcular os chamados deltas, que ponderam a influência no erro referentes às saídas dos neurônios. Sendo assim, é possível "culpar" individualmente cada unidade pela sua parcela de responsabilidade pelo erro da saída.

Na terceira etapa do algoritmo é feita a atualização dos pesos onde cada sinapse é ajustada de acordo com os deltas calculados na etapa anterior. As equações (Equações 3, 4, 5 e 6) e a Figura 4 a seguir mostram as etapas do algoritmo de aprendizagem *backpropagation*.

Figura 4 – ANN do tipo *feedforward* e *backpropagation*



$$\delta_n^i = -\left(T_n^i - y_n^i\right) . g_i' \quad (3)$$

$$\Delta W_{ij} = -\varepsilon \delta_n^i . a_n^j \quad (4)$$

$$\delta_n^j = \sum_i \left(\delta_n^i . W_{ij}\right) . g_j' \quad (5)$$

$$\Delta W_{jk} = -\varepsilon \delta_n^j . a_n^k \quad (6)$$

Fonte: Tani, J., 2016

2.2 REDE NEURAL RECORRENTE

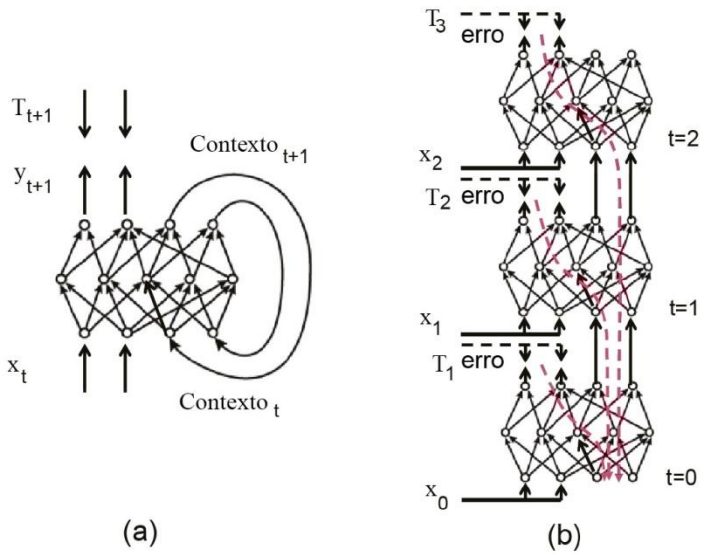
Uma rede neural recorrente é aquele que possui ciclos nas suas conexões, ou seja, os neurônios podem se conectar com unidades de camadas anteriores ou com eles mesmos. A ideia por trás de RNNs é fazer o uso de informação sequencial. Em redes neurais do tipo *feedforward*, por exemplo, cada informação de entrada, e também saída, é tratada como independente umas das outras. Mas existem situações em que para sabermos os dados de

saída atuais, necessitamos dos dados de entrada anteriores. Pode-se pensar nas RNNs como tendo “memória”, guardando informação sobre o que já foi calculado.

2.2.1 Backpropagation through time

O algoritmo de aprendizado mais utilizado para RNNs é o *backpropagation through time* (BPTT). Uma forma de representar uma RNN é “desdobrando-a” no tempo, na Figura 5(a) é mostrada a ativação da rede, enquanto a Figura 5(b) ilustra a RNN desdobrada no tempo. Pode-se ver que cópias da rede são empilhadas e as conexões dentro da rede são redirecionadas para as cópias subsequentes. Ao fazer isso, a RNN é “transformada” em uma *feedforward*, podendo ser usado o algoritmo de aprendizado *backpropagation*, já visto anteriormente.

Figura 5 – Rede neural recorrente



Fonte: Tani, J. (2016)

2.2.2 Rede Neural Recorrente de Tempo Contínuo

O modelo de rede neural recorrente de tempo contínuo (CTRNN, Continuous Time Recurrent Neural Network) é uma rede neural recursiva operando em tempo contínuo. Considerando uma CTRNN em que todas as unidades neurais são interconectadas, a dinâmica de ativação de cada neurônio pode ser descrita na forma das seguintes equações:

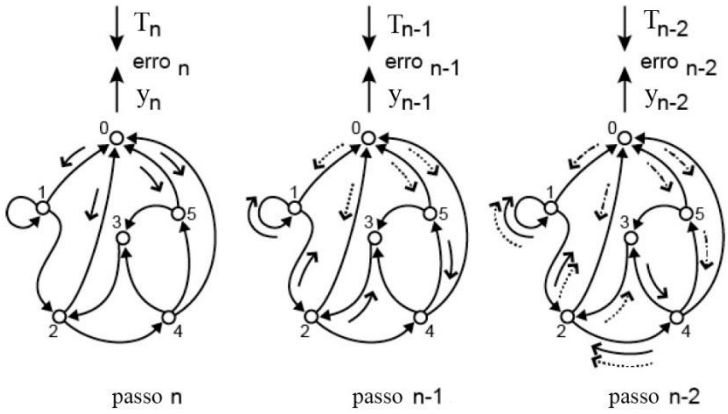
$$\tau \dot{u^i} = -u^i + \sum_j a^j \,.\, W_{ij} + I^j \tag{7}$$

$$a^i = \frac{e^{u^i} - e^{-u^i}}{e^{u^i} + e^{-u^i}} \tag{8}$$

A constante de tempo τ possui o papel de amortecimento com o valor positivo, quanto maior o valor da constante, mais lentamente o potencial de u^i muda.

A Figura 7 demonstra como o algoritmo de aprendizado BPTT, com algumas poucas modificações, atua em uma CTRNN, nela é exemplificado como o erro gerado em um passo continua a se propagar pela rede através do tempo, enquanto outros erros gerados nos passos seguintes também são propagados.

Figura 6 – *Backpropagation* em uma CTRNN.



Fonte: Tani, J. (2016)

Para n passos, a dinâmica de uma CTRNN nas etapas onde a propagação dos valores é feita para frente é computada pela Equação 9, que é obtida diferenciando a Equação 7 usando o método de Euler.

$$u_t^i = \left(1 - \frac{1}{\tau}\right) u_{t-1}^i + \frac{1}{\tau^i} \left(\sum_j a_{t-1}^j . W_{ij} + I_{t-1}^i\right) \tag{9}$$

$$a_t^i = \frac{e^{u_t^i} - e^{-u_t^i}}{e^{u_t^i} + e^{-u_t^i}} \tag{10}$$

Na equação temos um integrador com perdas com uma taxa de decaimento de $1-\frac{1}{\tau}$. Justamente por esse integrador com perdas o cálculo do delta é diferente da BPTT em um RNN convencional. O cálculo dos deltas é computado pela seguinte equação:

$$\frac{\partial E}{\partial u_t^i} = \begin{cases} -\left(Y_t^i - O_n^i\right).g' + \left(1-\frac{1}{\tau}\right)\frac{\partial E}{\partial u_{t+1}^i} & i \in Out \\ \sum_{k\in N}\frac{\partial E}{\partial u_{t+1}^k}\left[\delta_{ik}\left(1-\frac{1}{\tau}\right)+\frac{1}{\tau_k}W_{ki}.g'\right] & i \notin Out \end{cases} \quad (11)$$

Na Equação 11 $\delta_{i,k}$ é o delta de Kronecker, o qual é igual a 1 quando i=k, diferente disso é igual a zero. Nota-se que quando τ é igual a 1, a Equação 10 é a versão em tempo discreto da BPTT. Sendo assim em uma rede com um valor de T alto, a ação do BPTT ocorre com uma pequena taxa de decaimento, ou seja, isso habilita a aprendizagem de correlações de longo termo, pois filtra ações que possuem mudanças rápidas.

2.2.3 Resultados Anteriores

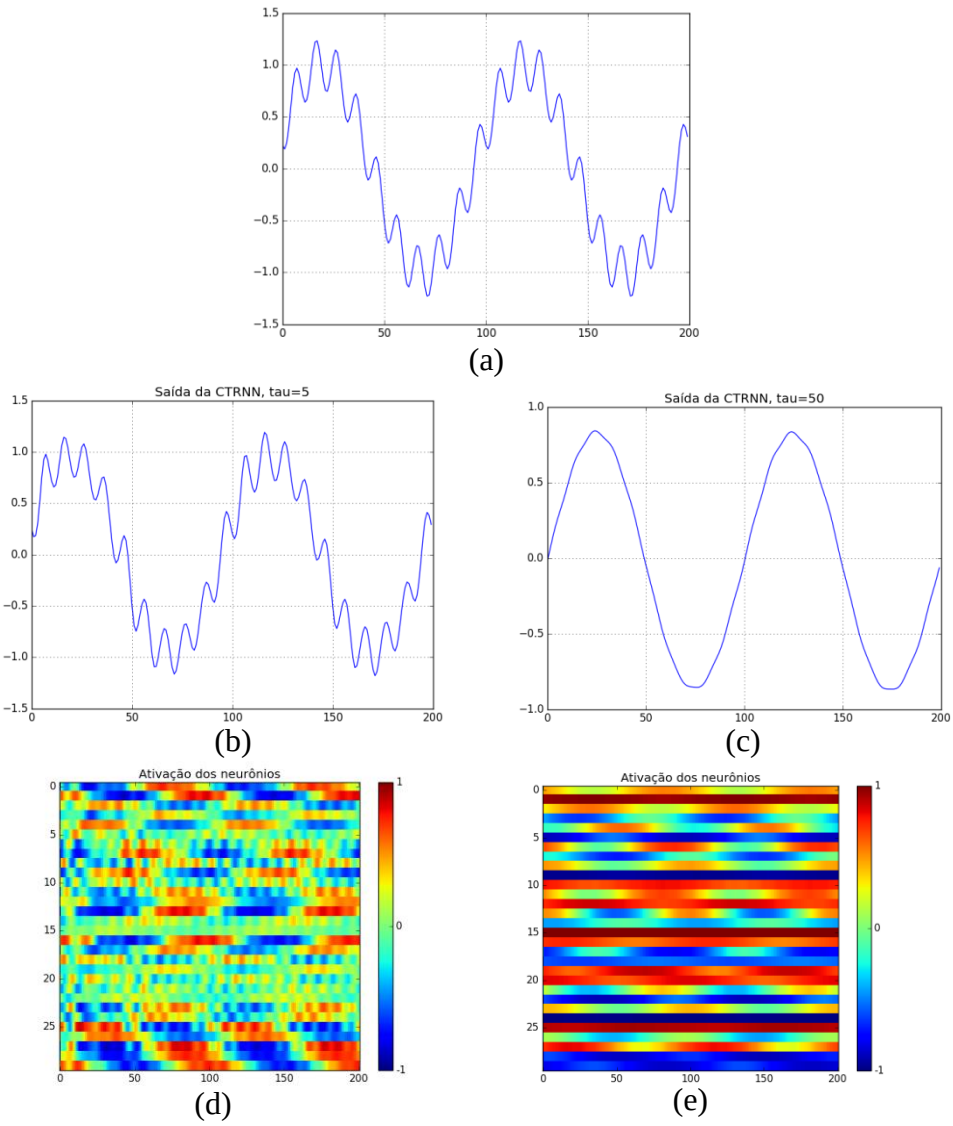
Em um trabalho anterior (STEIN[1], 2016), para demonstrar o efeito da constante de tempo tau, a rede CTRNN foi treinada utilizando duas ondas senoidais somadas, uma possuindo uma frequência maior que a outra. Com um tau menor é esperado que a rede fosse sensível a pequenas variações do sinal, ou seja, que a rede seguisse a onda com frequência menor, por outro lado, com um tau maior é esperado que a rede ignorasse essas pequenas variações e aprendesse com os sinais mais lentos, no caso do teste, a onda com a frequência menor. A Figura 7 é mostrada em: (a) onda senoidal usada para o treinamento da rede, resultante da soma de outras duas ondas com frequências e amplitudes diferentes. (b) e (d): rede com tau=5; (c) e (e): rede com tau=50. Sendo em (b) e (c): saída da rede; (d) e (e): ativação dos neurônios. Cada teste possui 200 *timesteps*, usando 30 neurônios, sendo o tau maior igual a 50 e o tau menor igual a 5.

[1] Projeto Integrador, Engenharia de Controle e Automação, 2016 (não publicado).

2.3 REDE NEURAL RECORRENTE DE MÚLTIPLAS ESCALAS DE TEMPO

O modelo de rede neural recorrente de múltiplas escalas de tempo (MTRNN) desenvolvido por Yamashita e Tani (Yamashita, Y., & Tani J. 2008), vem sendo usado para o aprendizado de ações complexas por máquinas (PENIAK *et al.* 2011, NOBUTA *et al.* 2011, ALNAJJAR *et al.* 2013). Ações complexas podem ser divididas em ações primitivas e recombinadas, formando planos mais complexos, de forma análoga ao que humanos e outros animais são capazes de fazer.

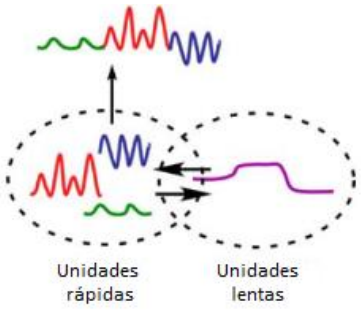
Figura 7– Saída da CTRNN



Fonte: Elaborada pelo autor

O modelo MTRNN tem como núcleo a rede neural de tempo contínuo CTRNN, onde camadas com diferentes escalas de tempo são conectadas, cada uma sendo responsável pelo aprendizado e manipulação dos primitivos, onde camadas mais rápidas, com um τ pequeno, acabam se especializando em codificar aspectos mais fundamentais relativos aos movimentos primitivos e camadas com um τ grande, ou mais lentas, acabam se especializando por manter aspectos mais gerais do plano de ações, cuidando da combinação dos movimentos primitivos. Na Figura 8 é exemplificada a manipulação dos movimentos primitivos pelas unidades da rede, nela os movimentos primitivos são representados pelas curvas de cor azul, verde e vermelha. Os movimentos primitivos são manipulados por unidades rápidas onde a atividade muda rapidamente, já a sequencia de primitivos é manipulado por unidades lentas, onde a atividade muda lentamente.

Figura 8 – Movimentos primitivos nas unidades rápidas e lentas



Fonte: Yamashita e Tani (2008)

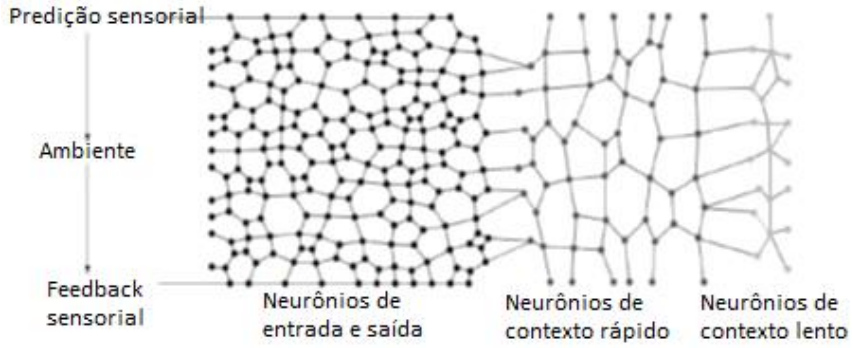
2.3.1 Estrutura

O núcleo de uma rede MTRNN é a rede CTRNN, distribuida em camadas com diferentes escalas de tempo. As entradas da rede são manipuladas por um mapa auto organizável do tipo TPM (*topology preserving map*), também conhecido como Mapa de Kohonen ou *Self-Organizing Map*. Um mapa TPM é usado no início da rede para produzir uma representação distribuída e discretizada do espaço de entrada das amostras de treinamento, tendo o sinal de entrada esparsamente codificado visando a redução da sobreposição entre várias sequencias de movimentos primitivos. Em nosso trabalho, porém,

escolhemos usar a transformação *softmax* (não confundir com a função de ativação *softmax*, Equação 12) ao invés da TPM.

Na Figura 9 é mostrada a rede MTRNN, onde as unidades de entrada e saída recebem os valores enviados pelo mapa auto organizável. Se houver mais de uma fonte em diferentes modalidades, como por exemplo propriocepção e visão, então dois mapas serão usados, uma para cada fonte, sendo que os sinais dos mapas são esparsamente codificados e somente conectam-se com outras unidades do próprio mapa, ou seja, unidades da propriocepção não se conectam com unidades da visão, o mesmo acontece nos neurônios de entrada e saída. Nos neurônios de contexto as unidades são totalmente conectadas, isto é, uma unidade se conecta com todas as outras unidades da rede incluindo ela mesma, mas neurônios de contexto lento não se conectam com os neurônios de entrada e saída, somente indiretamente através dos neurônios de contexto rápido. Valores dos pesos das conexões são geralmente assimétricos, ou seja, a conexão W_{ij} pode ser diferente de W_{ji} .

Figura 9 – Estrutura de uma MTRNN, nela as unidades do contexto rápido se conectam consigo mesmas, com entrada/saída e com contexto lento, e as unidades do contexto lento só conectam com contexto rápido, mas não se conectam com entrada/saída.



Fonte: Peniak M., Marocco F., Tani J. (2011)

A função de ativação do neurônio (Equação 12) depende de onde este pertence, caso pertença à camada de entrada e saída ($i \in Z$), a função será a *softmax*, caso contrário será a sigmoide:

$$y_{i,t} = \begin{cases} \frac{\exp(u_{i,t})}{\sum_{j \in Z} \exp(u_{j,t})} & i \in Z \\ \frac{1}{1+e^{-x}} & i \notin Z \end{cases} \quad (12)$$

2.3.2 Backpropagation through time

Para o treinamento da rede MTRNN o algoritmo BPTT mostrou-se eficiente (HAYKIN, S. S., 2009). Mas algumas modificações no algoritmo foram feitas. O erro não é mais calculado como o erro quadrático médio, e sim usando a divergência de Kullback-Leibler descrita na Equação 12, onde y* é o valor de ativação desejado no i-ésimo neurônio no tempo t, e y é o valor de saída real.

$$E = \sum_t \sum_{i \in O} y^*_{i,t} \log(\frac{y^*_{i,t}}{y_{i,t}}) \tag{13}$$

A atualização dos pesos sinápticos é feita pela Equação 13, onde $\partial E/\partial w$ define o gradiente, α é a taxa de aprendizado da rede e n é o passo da iteração da aprendizagem.

$$w_{ij}(n+1) = w_{ij}(n) - \alpha \frac{\partial E}{\partial w_{ij}} \tag{14}$$

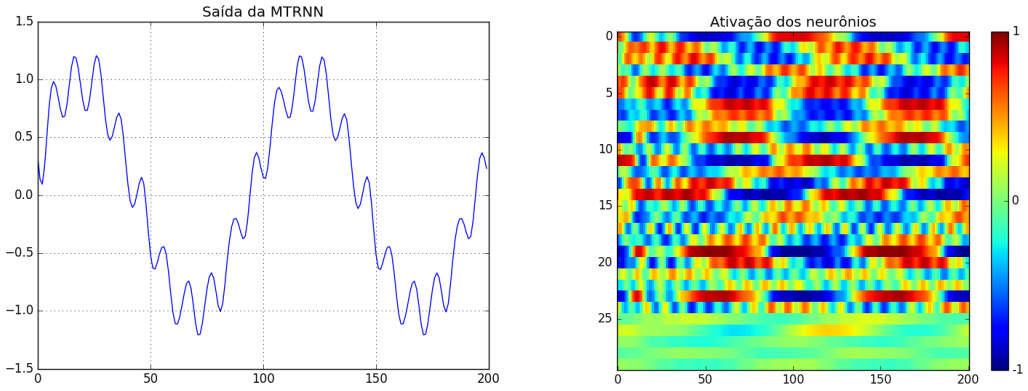
O gradiente $\partial E/\partial w$ é calculado pela Equação 14, sendo a Equação 10 usada para calcular o termo $\partial E/\partial u_{i,t}$

$$\frac{\partial E}{\partial w_{ij}} = \sum_t \frac{1}{\tau_i} \frac{\partial E}{\partial u_{i,t}} x_{j,t-1} \tag{15}$$

2.3.2 Resultados anteriores

Em trabalho anterior deste mesmo autor (STEIN, 2016) e de forma semelhante aos testes feitos com a CTRNN, foi feito um experimento com a MTRNN para demonstrar a diferença de ativação entre os neurônios rápidos e lentos. A mesma senoide da Figura 7(a) foi usada para o treinamento da rede, sendo que esta possuía 25 neurônios rápidos (τ=5) e 5 neurônios lentos (τ=50). Na Figura 10 é mostrado o resultado do experimento. No gráfico inferior esquerdo é mostrada a saída da rede, a qual foi capaz de prever satisfatoriamente as variações do sinal; no gráfico inferior direito é mostrado as ativações de cada neurônio ao longo dos timesteps, note que os últimos cinco neurônios são os neurônios lentos, onde a ativação ocorre de forma mais gradual que nos neurônios rápidos.

Figura 10 – Saída da MTRNN



Fonte: Elaborada pelo autor

2.4 ROBÔ DIMITRI

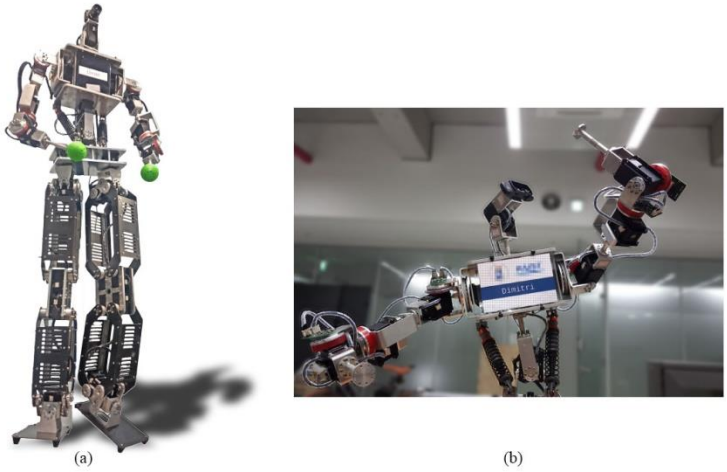
Os movimentos de robôs, principalmente os industriais, são geralmente altamente desacoplados de suas cargas, com juntas rígidas seguindo trajetórias pré-programadas, insensíveis a distúrbios do meio ao longo do caminho. Entretanto com o aumento da interação homem-robô também cresceu a demanda de atuadores mais seguros com complacência ativa ou passiva. Essa complacência permite que o robô reaja dinamicamente a esforços externos, permitindo interação e colisões suaves através do emprego de molas reais ou simuladas. Além de tornar os robôs mais seguros e leves, esse tipo de mecanismo permite o desenvolvimento de algoritmos de manipulação mais avançados, capazes de se adequar de forma inteligente às dinâmicas do ambiente. O robô Dimitri é um robô humanoide de baixo custo desenvolvido com o objetivo de permitir a pesquisa nos campos relacionados a interação homem-robô com o uso de juntas complacentes passivas, que empregam molas reais. Nos seguintes tópicos serão apresentadas as especificações técnicas da parte física do robô Dimitri, o software e os atuadores de série elástica.

2.4.1 Especificações técnicas

O robô humanoide Dimitri (Tatsch *et al.* 2016) (Figura 10, (a) Robô Dimitri de corpo inteiro (b) versão de torso) foi desenvolvido para ser *open-source* e *open-hardware* com o objetivo de ser robusto, simples e de baixo custo, facilitando assim manutenção e

customização, mas que ao mesmo tempo possibilitasse as pesquisas de manipulação inteligente de objetos com *feedback* de força e interação segura humano-robô.

Figura 10 – Robô Dimitri

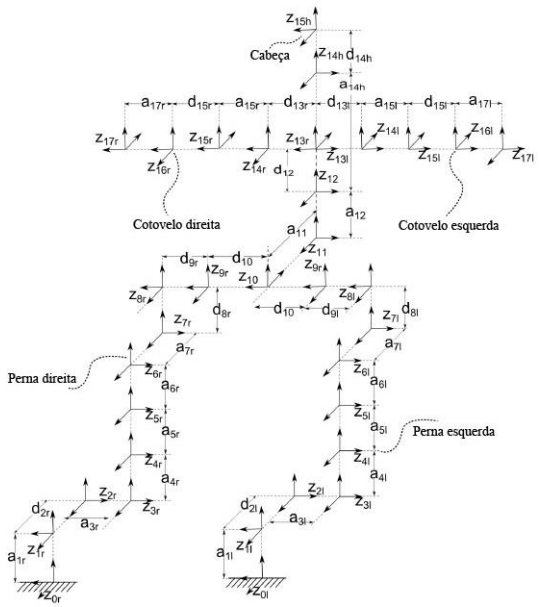


Fonte: Tatsch *et al.* 2016

As partes da estrutura do Dimitri são feitas de folhas de alumínio e fibra de carbono com 3mm de espessura cortadas e dobradas. Os braços também possuem atuadores de série elástica composto molas de poliuretano e montagem de alumínio equipados com um magnetômetro utilizado para medir o deslocamento angular. Na Figura 11 e na Tabela 2 são mostrados os parâmetros Denavit-Hartenberg do robô.

Devido à popularidade no campo de pesquisa de robótica, os atuadores escolhidos são da linha Dynamixel fabricados pela Robotis, sendo os servo-motores MX-106R e MX-64R os modelos escolhidos para as pernas, braços e cintura (primeiro modelo) e pescoço (segundo modelo). Os controladores são conectados juntamente com o circuito do magnetômetro usando barramento RS-485 e protocolo Dynamixel. O computador usado é um NUC (Next Unit of Computing) o qual se comunica com os motores e atuadores de série elástica a uma velocidade de 40kbps. Na Tabela 3 são mostradas as principais características do robô.

Figura 11 – Diagrama de referência para os parâmetros Denavit-Hartenberg



Fonte: Tatsch *et al.* 2016

Tabela 2 – Denavit-Hartenberg parâmetros para perna e braço direitos

i	α_i	a_i(cm)	d_i(cm)	Θ_i
00	0	0	0	Θ_0
01	0	4.5	0	Θ_1
02	π/2	0	5	Θ_2
03	0	4	0	$\Theta_3 = \Theta_4$
04	0	21	0	$\Theta_4 = \Theta_3$
05	0	10.5	0	$\Theta_5 = \Theta_6$
06	0	25	0	$\Theta_6 = \Theta_5$
07	0	3.75	0	Θ_7
08	- π/2	0	3.6	Θ_8
09	π/2	9.4	9.6	Θ_9
10	π/2	0	6	Θ_{10}
11	π/2	0	0	Θ_{11}
12	- π/2	11.5	0	Θ_{12}
13	π/2	0	8	Θ_{13}
14	π/2	0	14.5	Θ_{14}
15	- π/2	18.2	0	Θ_{15}
16	π/2	0	5.5	Θ_{16}
17	0	9.4	0	0

Fonte: Tatsch *et al.* 2016

Tabela 3 – Especificações técnicas do robô Dimitri

Especificações físicas	
Altura do robô	1241.79mm
Alcance	542.5mm
Peso do robô	13.3kg
Graus de liberdade	31
Números de juntas SEA	8 (4 em cada braço)
Carga máxima	2.5kg
Servo motores	MX-106R e MX-64R
Material do frame	Alumínio e fibra de carbono
Computador	
Processador	4ª geração Intel Core i5 4250U
Memória	8GB 1333MHz DDR3L
Armazenamento	120GB SSD mSATA
Barramento de controle	
Interface com PC	USB 2.0
Número de canais	4
Protocolo	RS-485 Dynamixel 1.0
Baudrate	1000kbps
Câmera	
Modelo	FireFlyMV FMVU-13S2C
Resoluçao	até 1280 x 960
Frame Rate	até 60 FPS
Montagem da lente	C / CS
Especificações elétricas	
Tensão de alimentação	12 12.6 Volts CC
Consumo máximo	1960 Wats

Fonte: Tatsch *et al.* 2016

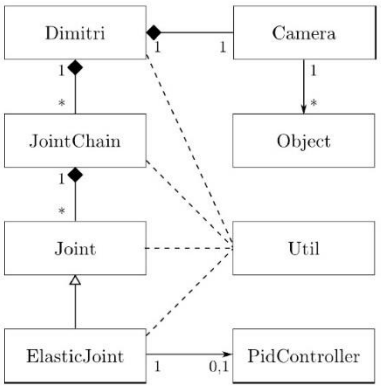
O código para o controle dos motores e SEAs é *open-source*, com orientação a objetos, escrito em C++ e com poucas dependências, sendo o processamento da visão uma delas, a qual depende do OpenCV. Na Figura 12 é exemplificado o diagrama de classes do código para o controle dos motores e SEAs. Para compor os braços, pernas e cintura a classe `JointChain` é usada, sendo que estes objetos pertencem a classe principal `Dimitri`. A classe `ElasticJoint` é uma derivada da classe `Joint` que inclui o feedback das molas e um controlador PID.

2.4.2 Atuadores de série elástica

No robô Dimitri para aplicar a complacência passiva são usados atuadores de série elásticas, os quais consistem em um servo-motor em série com uma mola conectada com a carga. Ao medir a deflexão da mola, a força exercida na saída do elemento complacente pode ser estimada. O uso das juntas complacentes é o aspecto que distingue o Dimitri de outros

robôs humanoides, sendo 4 SEAs em cada braço. O principal elemento da SEA utilizada é uma mola torsional (Martins *et al.* 2015) feita de elastómero de poliuretano (TPU). O material é barato, resistente e fácil de ser usinado. O elemento elástico não foi utilizado neste projeto numa tentativa de diminuir o número de entradas, tornando o treinamento menos complexo, facilitando assim a realização de teste para avaliar o desempenho da rede.

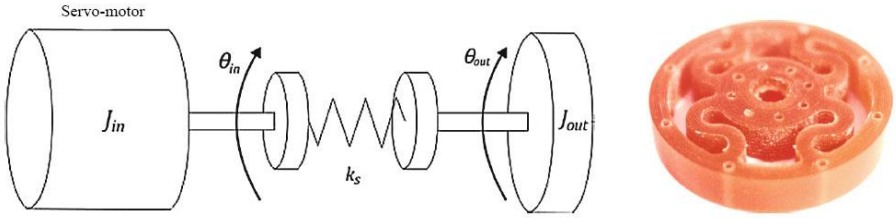
Figura 12 – Diagrama UML do robô Dimitri



Fonte: Tatsch *et al.* 2016

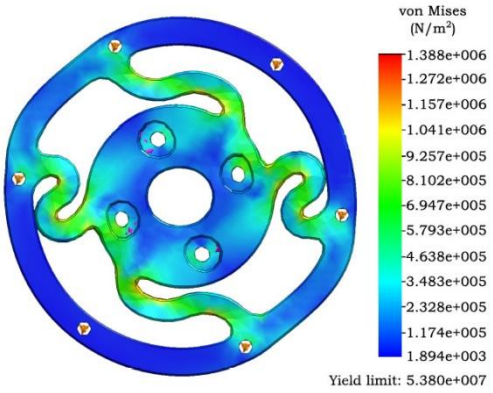
O formato da mola permite um grande deslocamento angular, tanto em sentido horário como anti-horário, e mantém linear a razão entre a força e o ângulo em toda a amplitude do torque suportado pelos motores. A simulação do deslocamento da mola durante o torque máximo do motor é mostrado na Figura 14.

Figura 13 – Atuador de série elástico



Fonte: Martins *et al.* 2015

Figura 7 – Simulação de deslocamento da mola



Fonte: Martins *et al.* 2015

2.5 TRABALHOS RELACIONADOS

Para estudar a aquisição de habilidades perceptivas e motoras como um exemplo de formação de padrão dinâmico, em um estudo realizado por Huys (2004) foram examinados a evolução do balanço postural e dos movimentos dos olhos e cabeça em relação a mudança na performance do aprendizado de malabarismo por indivíduos durante o estudo. Os resultados da pesquisa indicam que a aquisição de habilidades no domínio perceptivo-motor envolvem múltiplas escalas de tempo e dinâmicas multiformes, tanto em termos de desenvolvimento do comportamento alvo, quanto na evolução dos processos que servem ao comportamento alvo.

Baseado nesses achados, o modelo MTRNN foi proposto por Tani e Yamashita (2008) onde uma rede neural recorrente de tempo contínuo, utilizando mais de uma escala de tempo, foi aplicada em um robô humanoide a fim de demonstrar a emergência de uma hierarquia funcional entre os neurônios através de uma espécie de auto-organização. Nesse experimento, atividades complexas foram segmentadas pela MTRNN em primitivos, e estes foram reusados para formar novas sequências. Os resultados do experimento sugerem que as escalas de tempo agem como um importante mecanismo na formação de uma hierarquia funcional em sistemas neurais.

Outros experimentos foram feitos utilizando a MTRNN, Peniak (2011) e Alnajjar (2013) são exemplos de experimentos onde um robô humanoide foi utilizado para o aprendizado de atividades complexas, no primeiro experimento a rede neural foi capaz de aprender oito

padrões de movimento diferentes, já no segundo experimento foi utilizada uma rede neural com três escalas de tempo, na qual dois tipos de memória foram observados: estática, a dinâmica do neurônio muda muito pouco mesmo durante interrupções de tarefas, e dinâmica, na qual a dinâmica do neurônio decresce com o passar do tempo em uma interrupção de tarefa. Mais recentemente o robô Dimitri foi utilizado por Ahmadi (2016) para experimentos similares, os quais serviram de base para alguns dos testes realizados na rede neural implementada neste projeto.

3 MATERIAL E MÉTODOS

A rede neural MTRNN utilizada neste trabalho foi implementada em um projeto anterior do mesmo autor (STEIN, 2016). O algoritmo foi desenvolvido em Python, usando como base uma rede neural recorrente discreta (PETER'S NOTES, 2016). Python foi escolhido para ser a linguagem de programação deste projeto por ser relativamente rápido de programar, tendo várias bibliotecas de fácil utilização e principalmente pela popularidade na pesquisa com inteligência artificial, sendo que muitas das bibliotecas são voltadas para o aprendizado de máquina, como *Theano* e *Tensorflow*, por exemplo. O treinamento da rede utiliza o algoritmo de aprendizagem *Backpropagation Through Time* (BPTT) (WERBOS, 1990) otimizado pelo método do gradiente acelerado de Nesterov (SUTSKEVER, 2013). Os dados usados para o treinamento da rede são adquiridos com a movimentação do robô Dimitri por um tutor, ou seja, o robô é treinado por demonstração, onde o torque de suas juntas é desabilitado e um humano, posicionado em frente ao robô, segura seus braços e executa determinados padrões de movimentos cíclicos em tempo real. A estrutura deste capítulo é dividida em três partes: i) na primeira parte é explicado o modo de aquisição e processamento dos dados de entradas e saídas; ii) na seção seguinte é explicada a implementação da MTRNN; iii) na última parte são comentados os experimentos desenvolvidos usando a MTRNN e o robô Dimitri.

3.1 AQUISIÇÃO E PROCESSAMENTO DAS ENTRADAS E SAÍDAS

Os padrões usados para o treinamento da rede neural são representados pelo deslocamento angular dos servo-motores que movimentam as juntas dos braços do robô

Dimitri. A cada passo de tempo a posição angular do motor é lida pelo computador e armazenada em um *dataset* que depois será processado. Para simplificar os experimentos são usados apenas 8 servo-motores, 4 de cada braço, eliminando-se deste estudo os dados referentes às juntas do tórax e do pescoço.

Ao movimentar as juntas do robô, o torque é desabilitado e o tutor realiza movimentos cíclicos (mais detalhes na seção 3.4) repetindo-os por um determinado número de vezes (quatro vezes nesse trabalho), após a captura dos dados as amostras são segmentadas para conter o mesmo número de passos. No início de cada treinamento ou teste os dados armazenados são carregados para serem usados pela rede neural, esta etapa é feita *off-line*, ou seja, não está conectada diretamente com o robô ou suas juntas, podendo ser realizada em qualquer computador, porém os antes dos dados serem usados na rede neural eles passam por mais um processamento, no modelo proposto por Tani e Yamashita (2008) a fim de diminuir a possibilidade da sobreposição dos movimentos primitivos, os dados são codificados de maneira esparsa utilizando um mapa de topologia (TPM). Entretanto, nessa implementação é utilizada a transformação *softmax*, de maneira similar ao experimento realizado por Ahmadreza Ahmadi (AHMADI & TATSCH, 2016). A transformação é descrita pela seguinte fórmula:

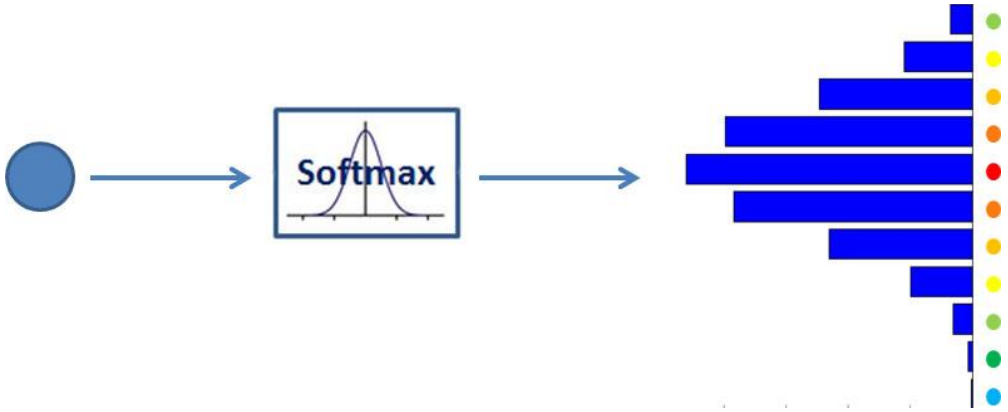
$$p_{i,t} = \frac{\exp\{-\frac{||k_i - k_{sample}||^2}{\sigma}\}}{\sum_{i \in Z} \exp\{-\frac{||k_i - k_{sample}||^2}{\sigma}\}} \tag{16}$$

Onde k_i é o vetor de referência com dimensão N, k_{sample} é o vetor dos dados da amostra e σ é a constante que determina a forma da distribuição de $p_{i,t}$.

Nesse trabalho a dimensão do vetor de referência é igual a 11, com valor mínimo de -1 e máximo de 1, e sigma foi estabelecido em 0,05, sendo assim, cada valor do vetor de entrada, é transformado em um vetor de tamanho 11 que é usado na entrada da MTRNN.

Na Figura 15 podemos visualizar melhor a transformação *softmax*, nela temos um valor proveniente de uma entrada, o qual ao passar pela transformação tem sua dimensão aumentada em onze, distribuindo assim a ativação de um sinal para diferentes neurônios, onde cada neurônio responde se ativando quando o sinal está em determinado intervalo, sendo assim o neurônio acaba representando intervalos adjacentes ao do próximo. Isso serve para distribuir as diferentes faixas de ativação do sinal para diferentes regiões da rede neural, facilitando o treinamento.

Figura 15 – Exemplo do uso da transformação *softmax*



Fonte: Elaborado pelo autor

A rede neural gera como saída vetores de mesmo tamanho dos vetores de entrada, 88 de entrada e 88 de saída, sendo assim, para retomarmos o formato original onde temos os deslocamentos angulares dos servo-motores, é necessário fazer a transformação inversa do *softmax*, descrita pela Equação 17.

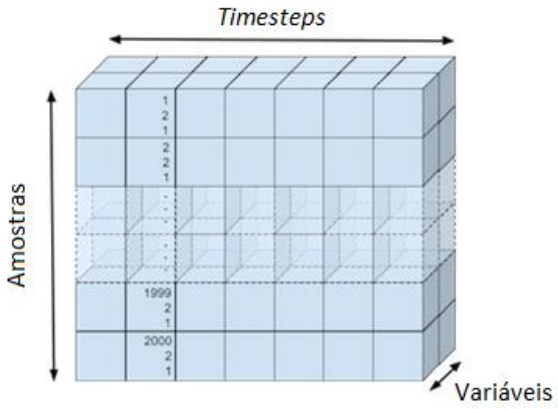
$$k_{out} = \sum_{i \in Z} y_{i,t} k_i \tag{17}$$

Com k_{out} temos os deslocamentos dos oito servo-motores que podem ser então ser utilizados.

Os algoritmos implementados neste trabalho (mais detalhes na seção 3.3) utilizam tensores como elemento base para os cálculos. Um tensor é a generalização que estende os conceitos de matrizes e de vetores para ordens elevadas. Sendo assim, um vetor é um tensor de primeira ordem e uma matriz é um tensor de segunda ordem. Neste trabalho os dados de entrada são organizados em tensores de terceira ordem, como mostrado na Figura 14.

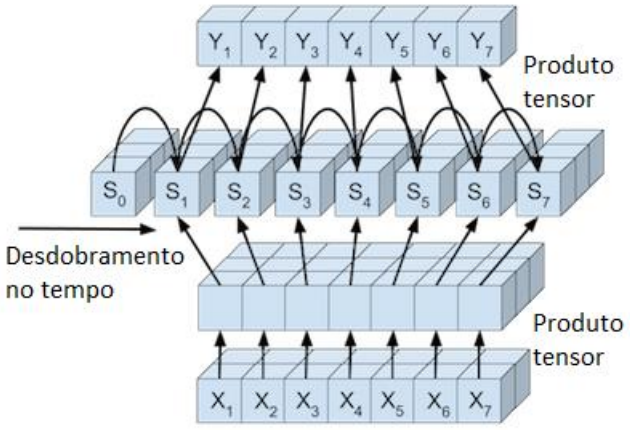
A Figura 16 ilustra as dimensões do tensor referente aos dados de entrada, que são: número de amostras, número de *timesteps* das amostras e número de variáveis. Por exemplo, um *dataset* com um formato (60, 20, 5) terá sessenta amostras, cada uma com vinte passos de tempo e cinco variáveis. Os processos da rede podem ser visualizados na Figura 17, onde se pode constatar o paralelismo alcançado graças à operação com tensores, com desdobramento da rede no tempo e a saída de cada passo de tempo.

Figura 16 - Representação do tensor usado na entrada



Fonte: Adaptado de Peter's Notes (2016)

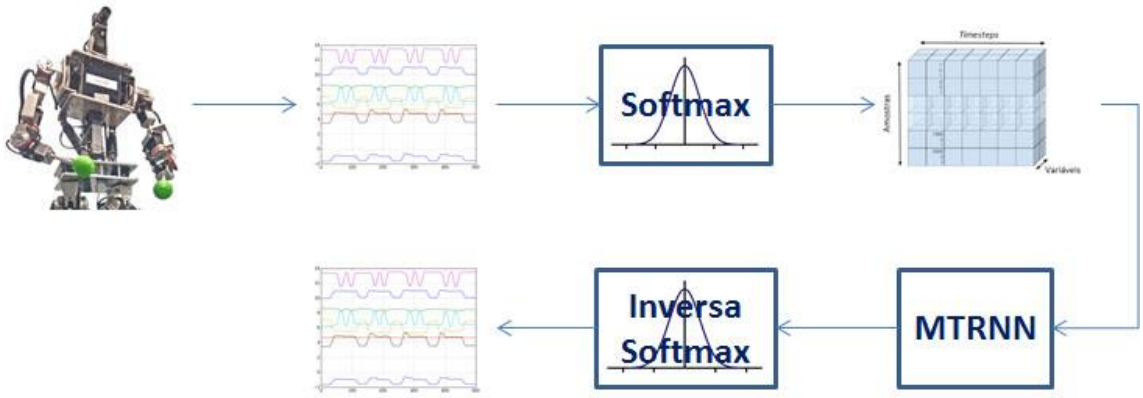
Figura 17 - Representação de como as entradas são propagadas na rede



Fonte: Adaptado de Peter's Notes (2016)

Após a captura dos dados e da transformação dos mesmos pela transformação *softmax*, as amostras são segmentadas em segmentos de igual tamanho, desse modo as amostras podem ser organizadas em um mesmo tensor, sendo assim a primeira dimensão do tensor é o número de amostras, a segunda é o número de *timesteps* e a terceira dimensão são as entradas, sendo 11 entradas para cada motor, logo temos 88 entradas por *timestep*. Na Figura 18 é resumido o processamento das entradas e saídas.

Figura 18 – Resumo da aquisição e processamento das entradas e saídas



Fonte: Elaborado pelo autor.

3.2 MTRNN - ESTRUTURA

Em um projeto anterior (STEIN, 2016), nosso ponto de partida para a implementação do modelo MTRNN, o algoritmo da DTRNN (*Discrete Time Recurrent Neural Network*), foi desenvolvido a partir de uma rede neural *open-source* (PETER'S NOTES, 2016), o projeto tinha como plano usar esse algoritmo como base para implementar as redes DTRNN, CTRNN e MTRNN, nessa ordem, aumentando assim gradativamente a complexidade com o objetivo de compreender melhor tanto o funcionamento da MTRNN, quanto de redes neurais artificiais em geral.

Conforme os modelos de redes neurais foram implementados, algumas mudanças foram feitas para simplificar o uso das redes, as quais continuaram no modelo seguinte, sendo assim, existem algumas diferenças entre o modelo proposto por Tani e Yamshita (2008) e a rede implementada. Além do uso da transformação *softmax*, citada na seção 3.1, duas outras alterações foram feitas, a primeira refere-se às camadas de entrada e saída, as quais são lineares, ao invés do uso da função de ativação *softmax*. Outra mudança foi a troca da função de custo, sendo no modelo proposto a divergência de Kullback-Leibler e na rede implementada o erro quadrático médio, mostrado na Equação 18. As alterações foram feitas em uma tentativa de simplificar a rede usando modelos implementados em trabalhos passados.

$$E = \sum_{t} \sum_{j} (y - y^{*})^{2} \tag{18}$$

Outra pequena alteração é o uso de um algoritmo de otimização para o treinamento da rede. Nesta implementação é utilizado o método de gradiente acelerado de Nesterov (SUTSKEVER, 2013), esse algoritmo é usado para atualizar os parâmetros de aprendizagem da rede e tem como hiperparâmetros (parâmetros que definem a estrutura e o treinamento da rede, não sendo aprendidos ou modificados durante o treinamento) o lambda λ (definido entre zero e um) para o Rmsprop, a taxa de aprendizagem e o número de épocas do treinamento, sendo este feito por *minibatches*, ou seja, o conjunto de amostras é dividido em um grupo menor. O hiperparâmetro λ atua de forma semelhante ao termo de momento (α), ajudando a rede a seguir ajustando os pesos na direção média, evitando que as amostras "puxem" para direções diferentes, devido a variações e ruídos, o que dificultaria a convergência.

O algoritmo Rmsprop mantêm a média móvel (MA) do quadrado de cada parâmetro θ, o gradiente é então normalizado pela raiz quadrada da média móvel. Sendo definido pela seguinte fórmula:

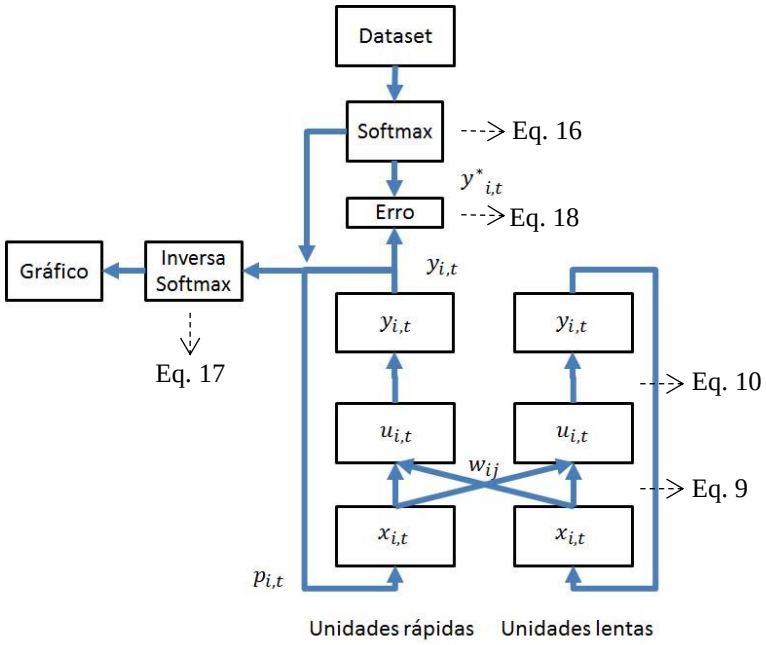
$$\begin{aligned} MA = \ & \lambda * MA + (1 - \lambda) * (\partial\xi / \partial\theta)^2 \\ \nabla(\theta) = & (\partial\xi / \partial\theta) / \sqrt{MA} \end{aligned} \tag{19}$$

O gradiente transformado é usado na atualização dos parâmetros, mas não diretamente, pois é usado para atualizar a velocidade (V_i) do parâmetro, sendo esta multiplicada por α, o qual é definido entre zero e um. Um modo de exemplificar a velocidade do parâmetro é imaginar uma bola rolando para baixo em um vale, quanto mais a bola rola na mesma direção, mais a velocidade da mesma aumenta, o mesmo acontece com os parâmetros, se os mesmos seguem uma mesma direção, a velocidade destes aumenta, diminuindo assim passos necessários para a convergência. O gradiente acelerado de Nesterov usa a velocidade para atualizar os parâmetros, calcula novamente os gradientes, atualiza a velocidade, e então atualiza os parâmetros baseados no gradiente local, o gradiente tem a vantagem de possuir mais informação para a atualização e pode corrigir más velocidades dos parâmetros. As atualizações realizadas por Nesterov são descritas pelas seguintes fórmulas, sendo $\nabla(\theta)$ o gradiente local da posição do parâmetro:

$$\begin{aligned} V_{i+1} &= \alpha V_i - \mu\nabla(\theta_i + \alpha V_i) \\ \theta_{i+1} &= \theta_i + V_{i+1} \end{aligned} \tag{20}$$

Mesmo com essas alterações a principal característica da MTRNN não foi modificada, ou seja, a rede ainda têm como núcleo uma CTRNN com constantes de tempo diferentes. A estrutura da rede implementada é mostrada na Figura 19. Nela podemos ver os estágios de processamento dos dados através da rede. Cada quadrado simboliza uma das etapas, como por exemplo, de $x_{i,t}$ para $u_{i,t}$ e depois $y_{i,t}$. As setas indicam o caminho dos dados. Na figura também são mostradas as equações utilizadas nas etapas, a Equação 9, por exemplo é utilizada para obter o estado interno do neurônio ($u_{i,t}$) através das entradas ($x_{i,t}$).

Figura 19 – Estrutura do modelo MTRNN implementado



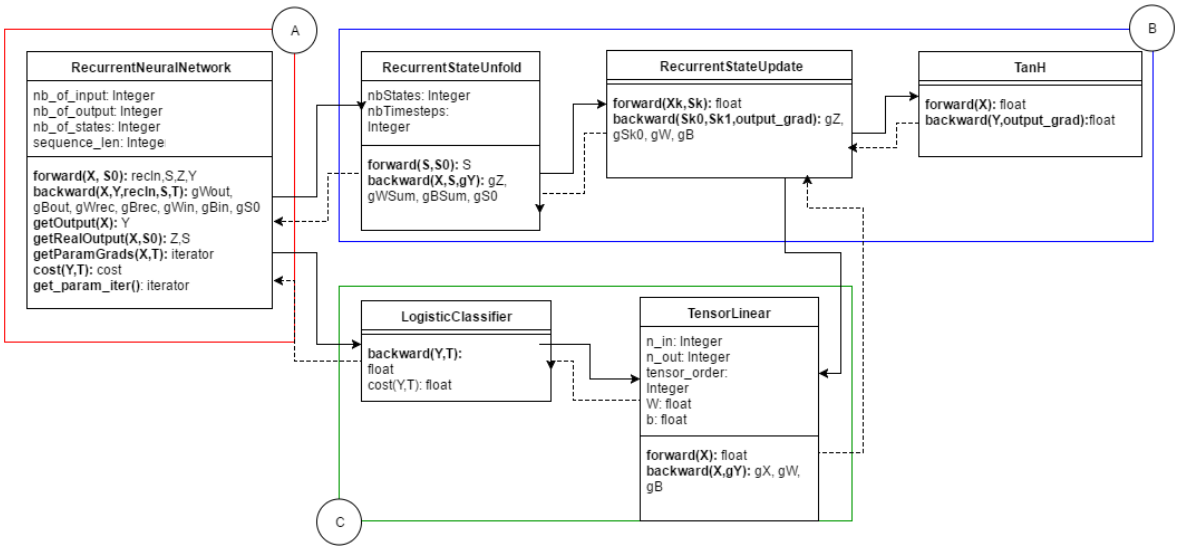
Fonte: Elaborada pelo autor

3.3 MTRNN - ALGORITMO

O algoritmo da MTRNN, foi implementado em Python, usando orientação a objetos. A rede é dividida em seis classes, cada uma com uma parte *forward* e uma *backward* utilizadas para o algoritmo de treinamento *backpropagation*. Pode-se ainda agrupar as classes de acordo com suas funções, são elas: (i) processamento de tensores de entrada e saída; e (ii) desdobramento dos estados da rede e a (iii) rede completa. Além destas também é utilizada uma heurística baseada em momento para a otimização do algoritmo de aprendizagem BPTT.

Na Figura 20 um panorama geral do código é apresentado, ilustrando como as classes e seus métodos são utilizados pelo algoritmo.

Figura 20 – Diagrama de classes da MTRNN



Fonte: Elaborado pelo autor

As classes referentes ao processamento dos tensores de entrada e saída (grupo C na Figura 18) são `TensorLinear` e `LogisticClassification`. Para a primeira são fornecidos como parâmetros do construtor o número de entradas, número de saídas, ordem do tensor, matriz de pesos e vetor de bias. Caso os pesos e o *bias* não sejam dados, estes são iniciados, sendo os valores dos pesos amostrados entre $\pm\sqrt{6.0/(n_{in}+n_{out})}$ (GLOROT e BENGIO, 2010) sendo n_{in} o número de entradas e n_{out} o número de saídas. A classe `TensorLinear` na passagem *forward* efetua a multiplicação das entradas pela matriz de pesos e soma o *bias*, enquanto na *backward* retorna os gradientes dos pesos, do *bias* e da entrada. Já na classe `LogisticClassifier`, a passagem *backward* retorna o gradiente da saída, enquanto a função custo retorna o custo da saída.

Na segunda parte do algoritmo (grupo B na Figura 18) as classes são voltadas para o desdobramento da rede no tempo. Sendo elas `TanH`, `RecurrentStateUpdate` e `RecurrentStateUnfold`, a primeira é usada pela segunda que é então usada pela terceira.

A classe `RecurrenteStateUnfold` desdobra a rede e atualiza os estados iterativamente no tempo, retornando o tensor dos estados resultante, para atualizar os estados é utilizada a classe `RecurrenteStateUpdate`. A função `backward()` retorna o gradiente do estado inicial, o somatório dos gradientes dos pesos e dos *bias*.

A classe `RecurrenteStateUpdate` atualiza os estados dos neurônios, combinando as entradas do neurônio já transformadas, com o estado anterior do mesmo, já na função `backward()` são retornados os gradientes do bias, dos pesos, do estado inicial e dos estados internos dos neurônios. Nesta classe é implementada a constante de tempo, sendo que para cada neurônio é designado um τ. O estado interno atual é salvo para ser usado na próxima passagem *forward*, sendo então multiplicado por $1/\tau$. Já para o *backpropagation* a primeira etapa é feita também salvando o gradiente de saída das camadas internas para ser usado posteriormente, sendo atualizado ao somar o gradiente atual com gradiente anterior multiplicado por $1 - 1/\tau$. A atualização dos gradientes dos estados internos segue o mesmo processo ao armazenar o gradiente do *timestep n*, multiplica-lo por $1 - 1/\tau$ e somar ao produto do gradiente do timestep *n-1* por $1/\tau$. O somatório dos gradientes dos pesos e dos bias também é multiplicado por $1/\tau$.

A classe `TanH` simplesmente aplica a função de ativação na entrada, a qual no caso é a tangente hiperbólica, e como retorno da função `backward`() temos o gradiente de entrada da camada. A função tangente hiperbólica foi escolhida de modo a diminuir as chances do problema da dissipação do gradiente ocorrer.

Finalmente temos a classe que define a rede neural como um todo (grupo A na Figura 18), `RecurrentNeuralNetwork`. O construtor dessa classe tem como parâmetros de entrada o número de entradas, número de saídas, o número de neurônios e o tamanho da sequência utilizada, isto é, o número de *timesteps* em cada amostra. Como nas classes anteriores, esta possui as funções `forward()` e `backward()`, a primeira realiza a transformação linear das entradas para alimentar as camadas internas, a rede é então desdobrada no tempo tendo como retorno a saída de cada neurônio, e novamente é utilizada uma transformação linear para obter as saídas da rede. No caso da função `backward`() primeiro é obtido o gradiente de saída que então é utilizado para o cálculo dos gradientes dos pesos e *bias* da transformação linear de saída e do gradiente de saída dos neurônios internos. Este gradiente é propagado na rede resultando nos gradientes referentes às camadas internas, que serão utilizados no cálculo dos gradientes da transformação linear de entrada.

Além das funções `backward()` e `forward()`, a classe `RecurrentNeuralNetwork` possui as funcões `getOutput()`, a qual retorna somente a saída da rede, a `getStateOutput`(), que retorna não somente a saída da rede mas também o estados dos neurônios, a função `cost()` que retorna a função custo definida na classe `LogisticClassifier` e duas funções utilizadas para a atualização dos parâmetros, `getParamGrads`() e `getParamsIter()`. A primeira utiliza a função `backward()` da mesma classe para obter os parâmetros que serão otimizados, no caso, os gradientes citados anteriormente. Já a função `getParamsIter()` retorna um iterador, o qual permite a atualização e otimização dos parâmetros a medida em que estes são utilizados na função de otimização.

É valido ressaltar que as ligações entre entradas e saídas com os neurônios mais lentos foram cortadas, isto é, o peso da conexão entre eles é mantido em zero. Para esse modelo só foram usadas duas constantes de tempo, caso houvessem mais de duas camadas com constantes de tempo diferentes, as ligações entre os neurônios rápidos e lentos também seriam cortadas, e a comunicação entre eles só seria feita através dos neurônios com o tau médio.

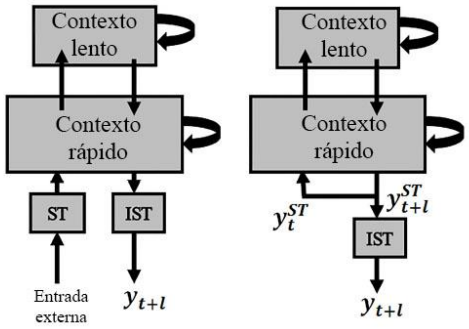
3.4 EXPERIMENTOS

Para testar a rede neural implementada, foram realizados três experimentos: a) aprender um padrão de movimento, b) aprender três padrões de movimento na mesma rede, c) aprender um padrão de movimento e posteriormente com a rede já treinada aprender um segundo movimento permitindo somente que as conexões lentas sejam atualizadas. Os experimentos realizados seguiram duas linhas de geração de saídas, a primeira é a geração em malha aberta (*open-loop*) vista no lado esquerdo da Figura 19, nesse tipo de geração a rede é alimentada com um conjunto de entradas externos a rede, que passarão pela transformação *softmax* (ST), gerando então uma saída na qual será usada a transformação *softmax* inversa (IST) resultando em y_{t+l}, sendo *l* o número de *timesteps* a frente do *timestep* atual *t*.

No lado direito da Figura 21 temos a geração em malha fechada (*closed-loop*), nela a saída da rede y_{t+l}^{ST} é usada para realimentar a rede neural, esse tipo de geração simula o uso direto no robô, chamada por Tani (2008) de “estímulo mental”, pois não passa por um meio físico, ou seja, o robô. O controle dos servo-motores do robô pela rede neural depende do sucesso desta operando em malha fechada, pois ao usar a transformada *softmax* inversa em

y_{t+l}^{ST} temos y_{t+l}, que é a nova posição para o motor, após o motor efetuar o movimento, sua posição é lida, obtendo assim y_{t+l} que após o uso da ST torna-se y_t^{ST}, ou seja, ao ser inserido na rede neural operando em malha fechada, o robô torna-se o meio físico desta. Em resumo, na geração em malha aberta, a entrada da rede não depende da saída, já na malha fechada a entrada da rede é igual a saída da mesma.

Figura 21 – Geração malha aberta e malha fechada



Fonte: Adaptado de Tatsch *et al.*(2016)

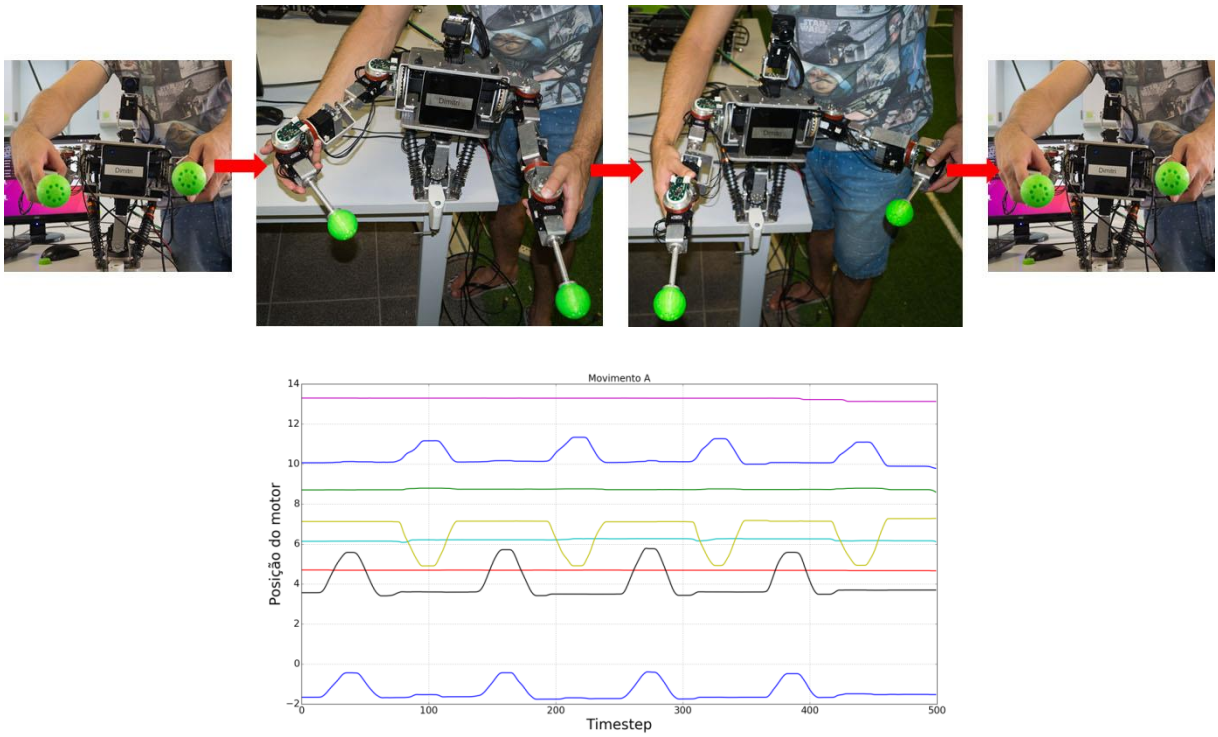
3.4.1 Experimento I

Neste primeiro experimento a rede foi treinada usando somente um padrão de movimento, tendo como objetivo mostrar em detalhes o treinamento da rede, desde a aquisição dos *datasets* usando o Dimitri, até a geração em malha aberta e fechada, previamente explicadas na subseção 3.2. Sendo assim, começamos pela aquisição das sequências de posições angulares dos servo-motores do robô quando enquanto este é movimentado por um tutor. Os dados são capturados usando o software e hardware explicados na seção 2.4.1. Similarmente aos experimentos feitos por Tani (2008) e Ahmadi (2016), todos os movimentos são cíclicos e iniciam e terminam em uma mesma posição inicial (HP-*Home Position*), como mostrado na Figura 22 (superior). Também pode ser observado pela periodicidade das curvas exibidas na Figura 20 (inferior) que o movimento foi repetido quatro vezes.

Após a coleta das amostras, estas foram segmentadas de maneira a ficar com 500 *timesteps* cada. Este número foi escolhido de forma a conter pelo menos quatro repetições do movimento, já que nesta implementação são esperadas amostras de tamanhos iguais.

Para o treinamento foram usados os hiperparâmetros apresentados na Tabela 4, sendo os hiperparâmetros da rede iguais para todos os experimentos, os parâmetros utilizados nos experimentos são similares aos usados por Ahmadi (2016), portanto não foram estudados o impacto do número de neurônios na rede. A rede foi treinada por $4x10^5$ épocas.

Figura 82 – Aquisição das amostras para treinamento



Fonte: Elaborado pelo autor.

Todos os experimentos foram realizados usando o computador próprio do robô, o qual possui um processador Intel Core i5 4250U e memória de 8GB 1333MHz DDR3L, os testes realizados duravam entre dezoito e vinte e quatro horas, a diferença no tempo de treinamento ocorria devido ao número de testes que estavam sendo realizados em paralelo. Durante o treinamento também eram realizados testes de validação, que consistem em apresentar à rede um *dataset* que não é usado durante treinamento, ou seja, um *dataset* que não é familiar a rede. O objetivo ao realizar esse tipo de teste é evitar o sobre treinamento (*Overtraining*), um problema no qual a rede perde a capacidade de generalização. Após o término do treinamento

foram realizados os testes em malha aberta, onde a rede é alimentada com dados de um *dataset* com o mesmo padrão para o qual a rede foi treinada.

Após os testes em malha aberta foram realizados os testes em malha fechada, no qual a rede é inicialmente alimentada com dados externos, similar ao teste em malha aberta, sendo as primeiras saídas descartadas para evitar o estado transitório, e após isso a rede tem suas saídas realimentadas em suas entradas, esperando assim que ela possa reproduzir sozinha o padrão, similar ao que foi inicialmente demonstrado no robô. Os resultados são apresentados na seção 4.1.

Tabela 4 – Hiperparâmetros da rede

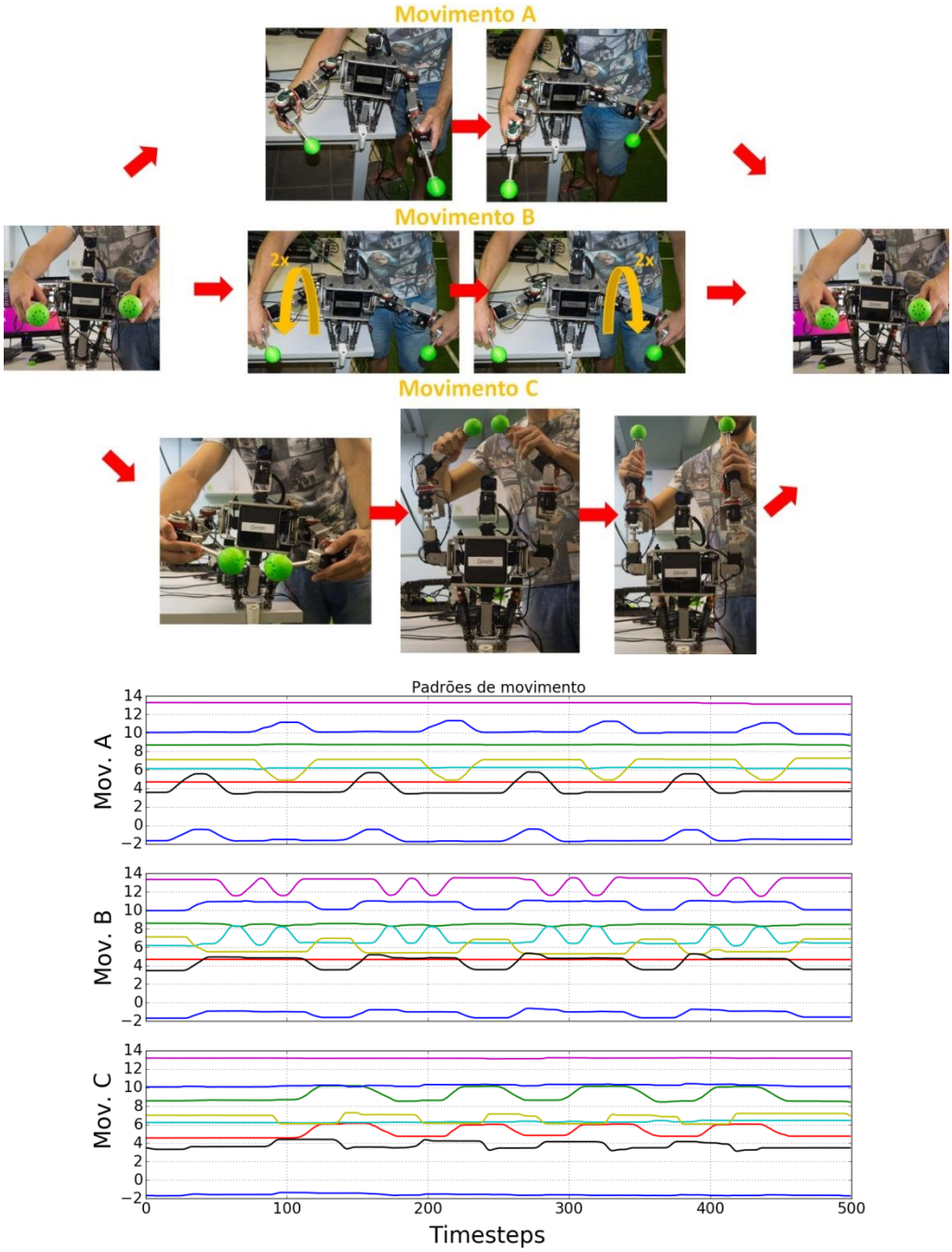
Hiperparâmetro	Valor
Neurônios rápidos	40
Neurônios lentos	20
Tau rápido	3
Tau lento	50
Número de entradas	8 (transformadas em 88)
Número de saídas	8 (transformadas em 88)
Taxa de aprendizagem	0,001
Momento	0,7

3.4.2 Experimento II

No segundo experimento a rede é treinada com três padrões de movimento diferentes, tendo como objetivo mostrar tanto a possibilidade da rede aprender mais de um padrão, quanto as mudanças internas da rede durante a transição de um movimento para outro. Basicamente o processo de aquisição de dados é o mesmo que no Experimento I, repetido para três padrões diferentes, sendo estes cíclicos, começando e terminando todos na mesma posição inicial, como também é feito por Ahmadi (2016), onde é mostrado a mudança da ativação dos neurônios durante a transição entre diferentes movimentos. Na Figura 23(superior) são mostrados os padrões treinados e na Figura 21(inferior) o deslocamento dos motores.

Quanto ao treinamento, os padrões foram apresentados à rede um por vez, ou seja, os parâmetros da rede eram atualizados após o treinamento utilizando um movimento. A rede utilizou os mesmos hiperparâmetros do Experimento I, sendo treinada por $4x10^5$ épocas, demorando em torno de vinte horas, novamente o tempo de treinamento varia conforme o número de testes sendo feitos em paralelo no mesmo computador. Os resultados são apresentados na seção 4.2.

Figura 23 – Aquisição das amostras usando três padrões



Fonte: Elaborado pelo autor.

3.4.3 Experimento III

Neste experimento a rede é treinada, em um primeiro momento, exatamente como no Experimento I, após isso a rede é treinada com um segundo movimento, mas dessa vez

permitindo somente a atualização dos parâmetros referentes ao contexto lento, o objetivo deste experimento é testar a possibilidade da rede aprender um novo padrão usando os movimentos primitivos adquiridos com o primeiro movimento. Tani (2008) realiza este experimento para provar a habilidade da rede em reorganizar os movimentos primitivos em novas atividades complexas.

Para a segunda fase do experimento o treinamento é feito normalmente usando um segundo padrão, entretanto não é permitida a atualização dos parâmetros dos neurônios do contexto rápido, somente os parâmetros referentes ao contexto lento são atualizados. Para a primeira fase a rede foi treinada por $4x10^5$ iterações, já na segunda fase por $2x10^5$ iterações. Como para a primeira fase do treinamento os parâmetros de rede foram aproveitados do Experimento I, a segunda fase durou em torno de quinze horas. Os resultados são apresentados na seção 4.3.

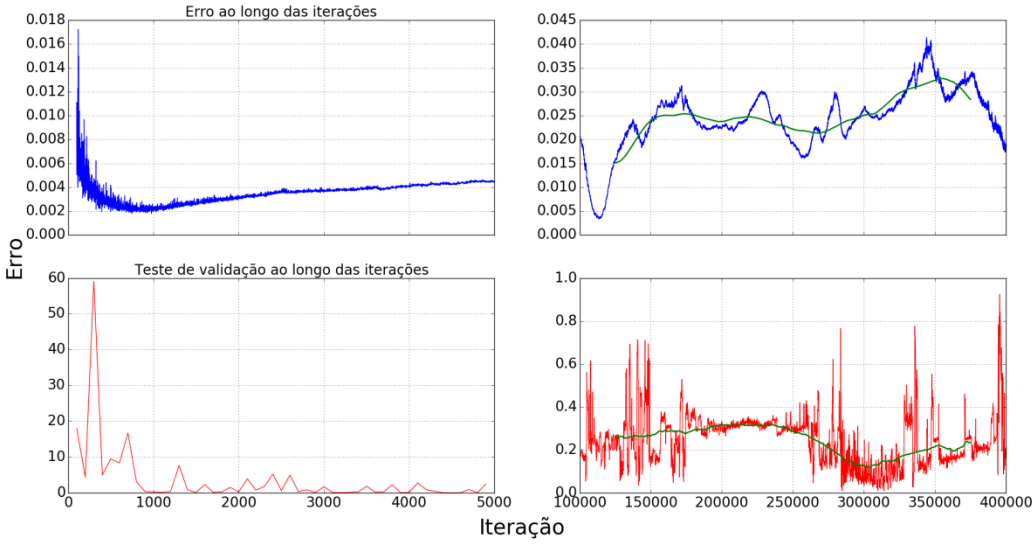
4 APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS

O capítulo de resultados será dividido de forma semelhante à Seção 3.2, sendo assim, os resultados mostrados são os obtidos após os três experimentos descritos serem realizados, os quais se referem a: (1) treinamento de um único padrão, (2) treinamento de três padrões e (3) treinamento de um padrão mais treinamento adicional de um segundo padrão atualizando somente as conexões lentas.

4.1 EXPERIMENTO I

O treinamento da rede neural usando somente um padrão foi feito com $4x10^5$ iterações, o que durou aproximadamente 24 horas. A cada 1000 iterações os parâmetros como peso e bias eram salvos, a cada 100 iterações um teste de validação era feito. Abaixo (Figura 24) é mostrado o erro do treinamento (gráficos superiores) e dos testes de validação ao longo das iterações (gráficos inferiores), sendo os gráficos da direita referentes às primeiras 5000 iterações e no da esquerda são mostrados os dados após a $1x10^5$ iteração, isso foi feito pois a variação dos valores é muito pequena, não sendo visível em um gráfico com todas as iterações, nos gráficos da direita a segunda linha representa a média móvel dos valores.

Figura 24 – Erros durante o treinamento

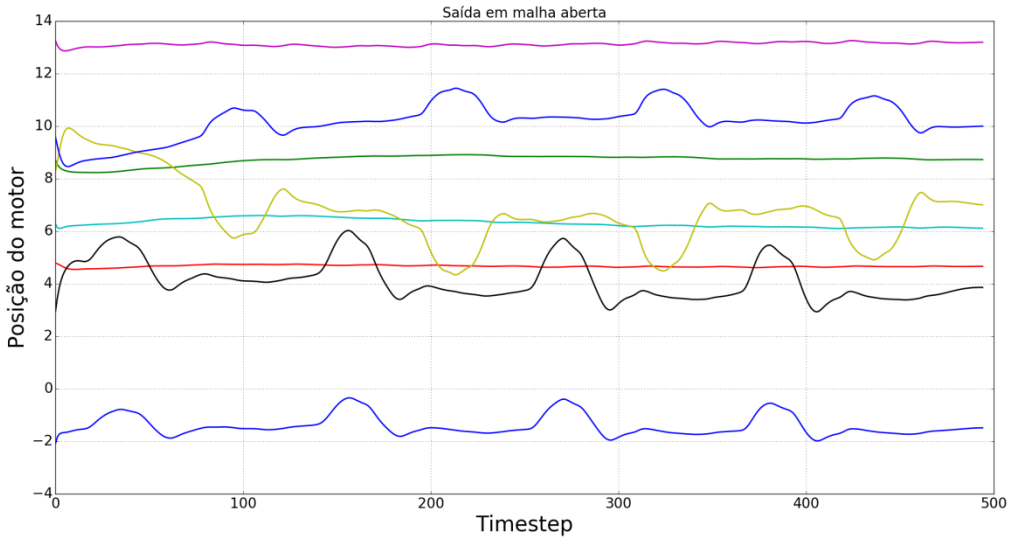


Fonte: Elaborado pelo autor

Nos gráficos superiores é mostrado o erro de treinamento o qual decai rapidamente nas primeiras iterações, mas depois sobe até atingir um erro médio de 0,24. Mesmo com o erro pequeno nas primeiras iterações, o erro no teste de validação ainda é muito alto, estabilizando-se a partir de $2x10^5$ iterações, chegando a um mínimo em $3x10^5$ e depois volta a crescer. Com base nesses resultados o conjunto de pesos e bias escolhidos ficaram $2x10^5$ e $3x10^5$.

Ao usar alguns diferentes conjuntos de pesos e bias no intervalo citado não se notou diferença significativa, sendo assim foi escolhido o conjunto salvo durante a iteração $3x10^5$. Na Figura 25 é mostrado a passagem *forward* de um *dataset*, nota-se que durante os primeiros passos, aproximadamente até o centésimo, a rede possui um efeito transitório devido ao a constante de tempo que não permite uma mudança imediata dos estados internos do neurônios, com base nisso, para fazer a rede operar em malha fechada é preciso ignorar as primeiras saídas, como é feito a seguir.

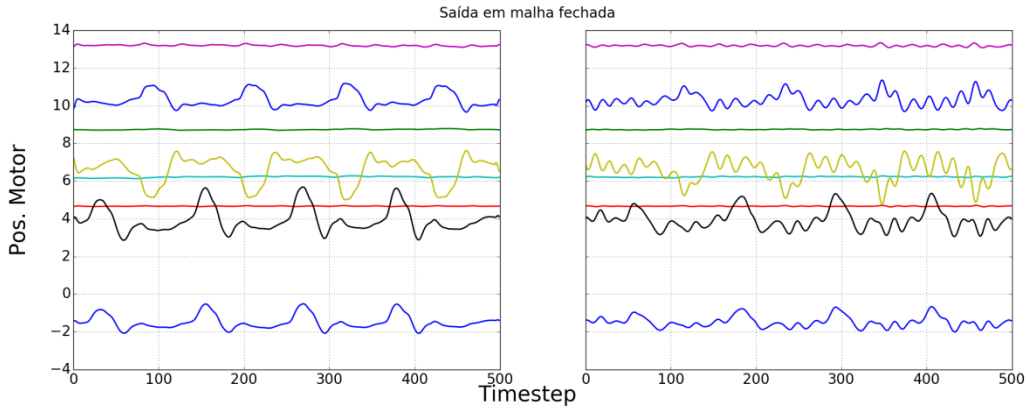
Figura 25 – Saída em malha aberta



Fonte: Elaborado pelo autor

Na Figura 26 (esquerda) é plotada a rede operando em malha fechada, o sinal possui algumas distorções, mas o formato básico do padrão é conservado. Entretanto após alguns ciclos, aproximadamente cinco repetições do padrão, a rede começa a apresentar um comportamento oscilatório, amplificando as distorções como pode ser visto na Figura 26 (direita). No experimento feito tanto por Tani (2008) quanto por Ahmadi (2016) a rede foi capaz de aprender os padrões de movimentos treinados.

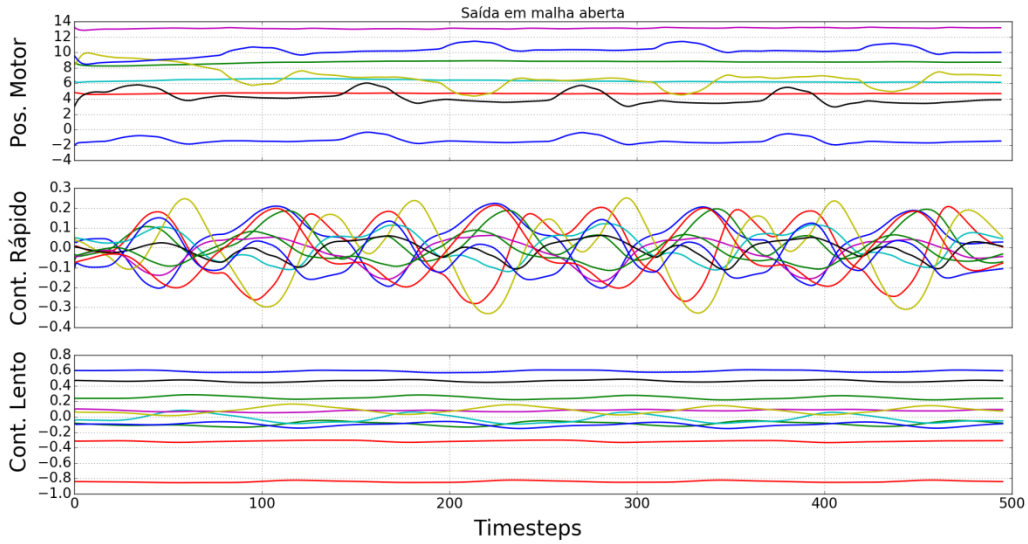
Figura 26 – Saída em malha fechada



Fonte: Elaborado pelo autor

Por último é mostrado na Figura 27 a saída da rede, a ativação de alguns neurônios pertencentes ao contexto lento e ao contexto rápido em malha aberta. Nota-se a diferença na frequência de mudança de estados entre os dois conjuntos de neurônios.

Figura 27 – Ativação dos neurônios



Fonte: Elaborado pelo autor

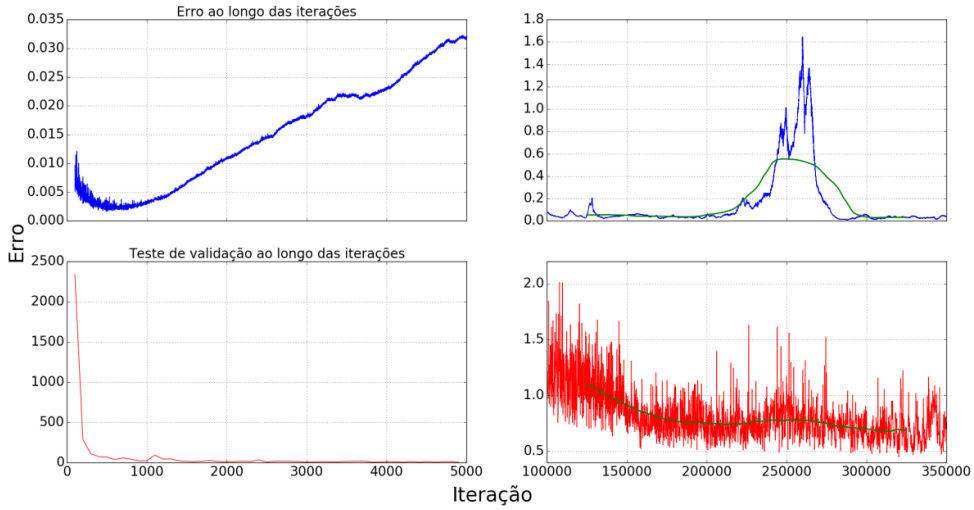
4.2 EXPERIMENTO II

Baseado nos resultados do experimento anterior, para o segundo experimento a rede foi treinada por $3{,}6x10^5$ épocas, usando três padrões de movimentos diferentes. A cada 1000 iterações os parâmetros como peso e bias eram salvos, a cada 100 iterações um teste de validação era feito. Na Figura 28 é mostrado o erro do treinamento (gráficos superiores) e dos testes de validação (gráficos inferiores) ao longo das iterações, novamente os gráficos da direita são referentes às primeiras 5000 iterações e nos da esquerda são mostrados os dados após a $1x10^5$ iteração, nos gráficos da direita a segunda linha representa a média móvel dos valores.

Nos gráficos superiores é mostrado o erro de treinamento o qual decai rapidamente nas primeiras iterações, subindo nas iterações seguintes até se estabilizar perto de $1{,}5x10^5$ iterações, mas perto da iteração $2{,}5x10^5$ ocorre um pico no erro, ao mudar os hiperparâmetros da rede um ou mais picos também ocorreram. Mesmo com o erro pequeno nas primeiras

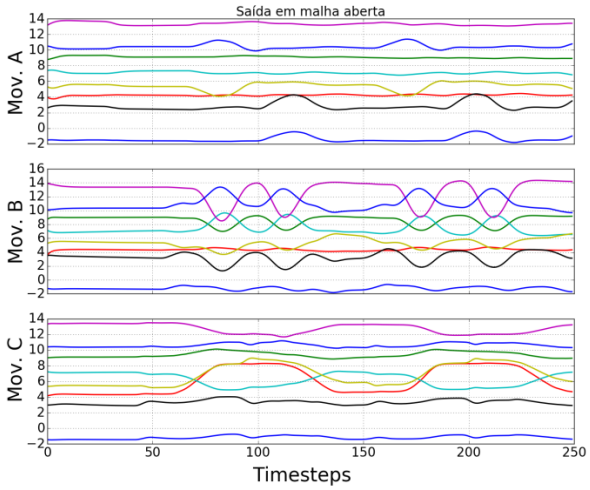
iterações, novamente o erro no teste de validação ainda é muito alto, estabilizando-se a aproximadamente a partir de $2x10^5$ iterações. Considerando o pico foram feitos testes com conjuntos de pesos antes e depois do pico ocorrer, mas a diferença entre ambos é pequena, sendo as saídas em malha aberta dos três movimentos (explicados na subseção 3.3.3) mostradas na Figura 29.

Figura 28 – Erros durante o treinamento



Fonte: Elaborado pelo autor

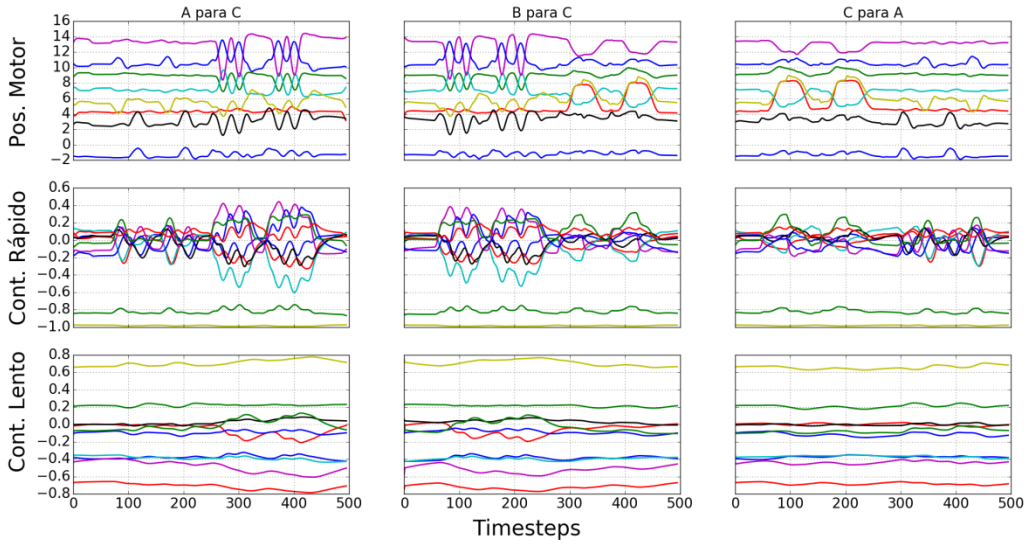
Figura 29 – Ativação dos neurônios durante a transição dos movimentos



Fonte: Elaborado pelo autor

Quanto à ativação dos neurônios foi usado um conjunto de *datasets* onde existe uma transição entre os movimentos, sendo eles do padrão A para B, de B para C e de C para A. Na Figura 30 é mostrada a saída da rede em malha aberta e a ativação dos neurônios do contexto lento e rápido. Podemos notar tanto a diferença na ativação dos neurônios de ambos os contextos nos diferentes movimentos, como também a repetição destes quando o movimento é repetido.

Figura 30 – Ativação dos neurônios durante a transição dos movimentos



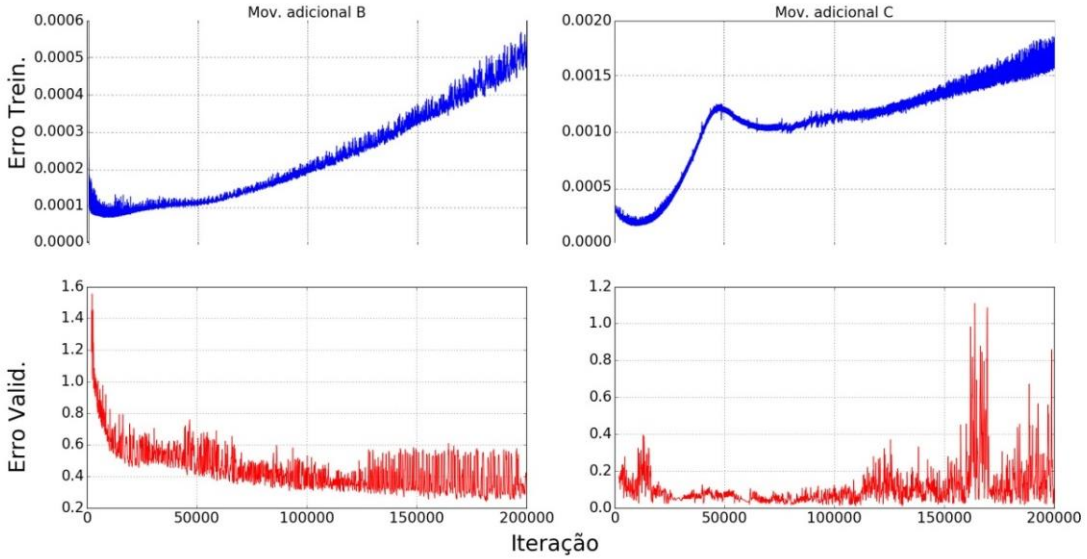
Fonte: Elaborado pelo autor

4.3 EXPERIMENTO III

Para o terceiro experimento o conjunto de pesos e bias usados no Experimento I referentes ao Movimento A também foi usado, a partir dele os parâmetros do contexto lento foram atualizados ao receber um treinamento adicional utilizando o *dataset* do Movimento B e em outro teste do Movimento C. Da mesma forma que nos experimentos anteriores a rede foi treinada por $2x10^5$ épocas. A cada 1000 iterações os parâmetros como peso e bias eram salvos, a cada 100 iterações um teste de validação era feito. Na Figura 31 é mostrado o erro do treinamento e dos testes de validação ao longo das iterações. Mesmo usando outros parâmetros o resultado era semelhante, o erro durante o treinamento crescia ao longo das

iterações. Já o erro no teste de validação usando o Movimento B decresceu, enquanto no teste com o Movimento C não ficou estável.

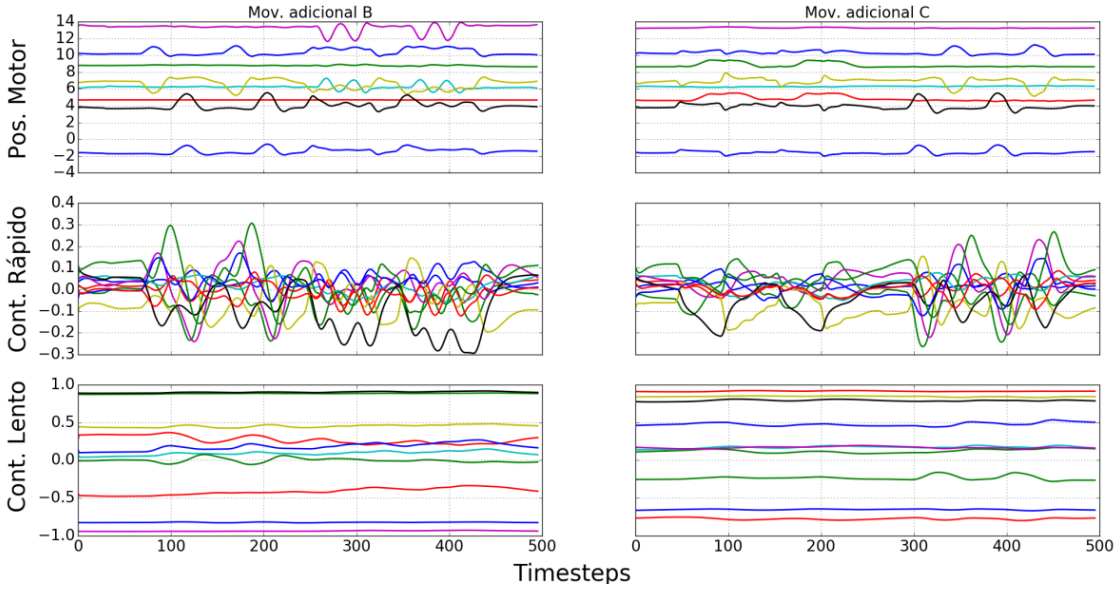
Figura 31 – Ativação dos neurônios durante a transição dos movimentos



Fonte: Elaborado pelo autor

Como o erro durante o treinamento não caiu, foi decidido usar os últimos conjuntos de pesos e bias salvos. Sendo assim na Figura 32 temos a passagem *forward* em malha abera dos testes, sendo o gráfico superior a saída da rede, o do meio a ativação de alguns neurônios do contexto rápido e por último a ativação de alguns neurônios do contexto rápido. Nota-se que o padrão do movimento A ainda é mantido, mas os novos movimentos não são replicados satisfatoriamente, mesmo assim é possível notar algumas características dos movimentos, principalmente no movimento B. No experimento realizado por Tani os movimentos gerados pela rede treinada com mais de um padrão e a rede com treinamento adicional nos neurônios lentos eram muito similares.

Figura 32 – Saída em malha aberta e ativação dos neurônios



Fonte: Elaborado pelo autor

5 CONSIDERAÇÕES FINAIS

Com base nos resultados apresentados, observou-se que o modelo de rede neural recorrente de múltiplas escalas de tempo implementado não apresentou um desempenho satisfatório para ser aplicado diretamente no robô Dimitri, mesmo sendo capaz de aprender mais de um padrão de movimento, a rede não foi capaz de operar em malha fechada de forma eficaz.

No primeiro experimento ao operar em malha aberta, a rede gerou um sinal muito próximo do esperado, entretanto ao ser realimentada a rede gerou saídas com oscilação crescente a cada novo ciclo, o que descaracterizou o movimento e de certa forma impediu a rede de ser aplicada no Dimitri, pois poderia causar danos ao robô.

Já no segundo experimento a rede conseguiu aprender os três padrões que foram apresentados durante o treinamento, cumprindo um dos objetivos do trabalho. Neste experimento também foi possível mostrar a ativação dos neurônios, as quais eram facilmente distinguíveis entre os diferentes movimentos, pois ao serem executados, mesmo durante uma transição, as ativações dos neurônios se repetiram.

No terceiro experimento não se obteve êxito, já que o erro durante o treinamento não foi minimizado, mesmo assim foi possível notar algumas características dos movimentos adicionais na saída da rede, podendo assim demonstrar a possibilidade de a rede implementada de aprender um novo padrão a partir dos movimentos primitivos previamente aprendidos. Uma possível explicação para a falha do experimento, assim como para a operação em malha fechada, são as alterações feitas sobre o modelo original, sendo a principal delas o uso do erro quadrático médio ao invés da divergência de Kullback-Leibler, uma vez que o erro durante o treinamento se aproximava de zero, mas não apresentava melhoras significativas.

Um ponto que também pode ser observado é em relação aos hiperparâmetros da rede e à qualidade das amostras usadas para treinamento, uma vez que a rede implementada mostrou-se sensível a modificação destes, muitas vezes levando o erro durante o treinamento crescer exponencialmente. A única dificuldade para explorar diferentes conjuntos de parâmetros é o tempo de treinamento, que na prática, considerando-se tempo de setup, se aproximava a um dia.

O início do projeto se deu com o aprendizado da linguagem Python juntamente com o básico das redes neurais artificiais, logo alguns detalhes podem ter passado despercebidos ao longo do projeto, como é o caso da importância da escolha correta da medida do erro e como usá-la. Saindo da área de redes neurais, outro fator decisivo na implementação do algoritmo foi a pouca experiência em otimizar o código escrito, dificultando assim qualquer mudança necessária na estrutura da rede.

Para ser aplicada diretamente no robô, possivelmente uma implementação usando a estrutura original da MTRNN deverá ser feita, fazendo uso de outras medidas de erro ou da divergência de Kullback-Leibler como é proposto originalmente. Em uma nova implementação outros recursos de hardware podem ser usados, como a utilização de unidades de processamento gráfico (GPU, Graphics Processing Unit), uso da câmera como é já é feito nos experimentos do Prof. Tani, e principalmente a utilização os atuadores de série elástica, diferencial do robô, aumentando a complexidade da rede neural o que permitirá ao robô “sentir” os objetos manipulados, sendo esta etapa um avanço nas pesquisas do Prof. Jun Tani, o qual possui uma colaboração neste projeto. Mesmo não funcionando integralmente, esperasse que este trabalho sirva como ponto de partida e ajude no desenvolvimento das áreas de robótica e aprendizado de máquina do Centro de Tecnologia da UFSM.

REFERÊNCIAS BIBLIOGRÁFICAS

AHMADI, A.; TATSCH, C.; TANI, J.; GUERRA, R. S. **Dimitri: A Low-cost Compliant Humanoid Torso for Cognitive Robotics Research.** Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR), 2016 XIII Latin American, 2016.

ALNAJJAR, F.; YAMASHITA, Y.; TANI, J. **The hierarchical and functional connectivity of higher order cognitive mechanisms:** neurorobotic model to investigate the stability and flexibility of working memory. Frontiers in Neurorobotics, 2013.

CAUDILL, M. **Neural Network Primer:** Part I. AI Expert, Feb. 1989

GLOROT, X.; BENGIO, Y. **Understanding the Difficulty of Training Deep Feedforward Neural Networks.** DIRO, Universite de Montréal, Montréal, Québec, 2010.

HAYKIN, S. S., & HAYKIN, S. S. **Neural networks and learning machines.** New York, Prentice Hall/Pearson. 2009

HUYS, R.; DAFFERTSHOFER, A.; BEEK, P. J. **Multiple time scales and multiform dynamics in learning to juggle**, Motor Control, vol. 8, pp. 188–212, 2004.

MARTINS. L.; TATSCH, C.; MACIEL, E. H.; HENRIQUES, R. V. B.; GERNDT, R.; GUERRA, R. S. **Polyurethane-based Modular Series Elastic Upgrade to a Robotics Actuator.** RoboCup 2015: Robot World Cup XIX, 2015.

NOBUTA, H.; NISHIDE, S.; OKUNO, H. G.; OGATA, T. **Identification of self-body based on dynamic predictability using neuro-dynamical system.** System Integration International, 2011.

PENIAK M., MAROCCO D., TANI J., YAMASHITA Y., FISCHER K., CANGELOSI A. (2011). **Multiple Time Scales Recurrent Neural Network for Complex Action Acquisition.** In: Comput. Neurosci. Conference 2011.

PETER'S NOTES. Dísponível em: < http://peterroelants.github.io>. Acesso em: 8 mai. 2016

ROJAS, R. (1996). **Neural networks:** a systematic introduction. Berlin, Springer-Verlag.

STEIN, G. M. S. **Implementação de Modelo de Rede Neural Recorrente de Múltiplas Escalas de Tempo,** Projeto Integrador, Engenharia de Controle e Automação, UFSM, 2016.

SUTSKEVER, I.; MARTENS, J.; DAHL, G.; HINTON, G. **On the Importance of Initialization and Momentum in Deep Learning**, 2013.

TANI, J. (2016). **Exploring Robotic Minds:** Actions, Symbols, and Consciousness as Self-Organizing Dynamic Phenomena. Oxford University Press.

TATSCH, C.; AHMADI, A.; BOTTEGA, F.; TANI, J.; GUERRA, R. S. **Dimitri: Na open-source Humanoid With Compliant Joints.** 2016

WERBOS, P. J. **Backpropagation through time: what it does and how to do it.** Proc. IEEE, 78(10):1550-1560. 1990

WILDML: AI, DEEP LEARNING, NLP. Disponível em: < http://www.wildml.com>. Acesso em: 8 mai. 2016.

YAMASHITA, Y.; TANI J. **Emergence of functional hierarchy in a multiple timescale neural network model:** a humanoid robot experiment, PLoS Computational Biology, vol. 4, no. 11, 2008.

ANEXOS

ANEXO A – CÓDIGO FONTE DA MTRNN

```
# coding= utf-8

import itertools
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pickle

#Dataset
nb_train = 3  # numero de amostras
#sequence_len = 100  # tamanho de cada sequencia

#Dados da rede
nb_of_states = 60 # Numero de neuronios
nb_of_fast = 40
nb_inputs = 8*11  # numero de entradas
nb_outputs = 8*11  # numero de saidas

#Hiperparametros da rede
lmbd = 0.5  # Rmsprop lambda
learning_rate = 0.001  # Taxa de aprendizado
momentum_term = 0.7  # Momentum
eps = 1e-6  # Prevenir divisao por zero
mb_size = 1  # tamanho das minibatchs (numero de amostras)
nb_iteration = 10

tau_fast = 2.0
tau_slow = 50.0
# Matriz diagonal com os valores de tau
alfa = tau_fast*np.ones(nb_of_states)
alfa[nb_of_fast:] = tau_slow*np.ones(nb_of_states-nb_of_fast)
tau_inv = np.diag(1/alfa)

# Transformacao Softmax
def softmax_array(x):
    DIM = 11
    maximum = 1.0
    minimum = -1.0
    sigma = 0.20

    intervals = minimum + np.arange(DIM) * float(maximum - minimum) /
float(DIM)
    x = intervals - x
    x = np.multiply(x,x) / sigma
    x = max(x) - x
    x = np.exp(x)
    x = x / sum(x)
    return x

# def load_dataset(file):
    with open(file, "rb") as input_file:
        a = pickle.load(input_file)
        a_in = np.array([frame.values() for frame in a])
```

```
    file_len, nb_inputs = a_in.shape     # Tamanho da sequencia carregada e
numero de entradas

    X = np.zeros((3, file_len-5, nb_inputs*11))     # Inicializa tensor de
entradas
    T = np.zeros((3, file_len-5, nb_inputs*11))     # Inicializa tensor de
saidas

    soft_inputs = np.zeros((file_len-5, nb_inputs*11), dtype=object)    #
Inicializa matriz para os valores em softmax
    soft_outputs = np.zeros((file_len-5, nb_inputs*11), dtype=object)


    for j in range(file_len - 5):
        soft_inputs[j,    :]    =    np.hstack((softmax_array(float(a_in[j,
0]/100)),
                                      softmax_array(float(a_in[j,
1]/100)),
                                      softmax_array(float(a_in[j,
2]/100)),
                                      softmax_array(float(a_in[j,
3]/100)),
                                      softmax_array(float(a_in[j,   4]   /
100)),
                                      softmax_array(float(a_in[j,   5]   /
100)),
                                      softmax_array(float(a_in[j,   6]   /
100)),
                                      softmax_array(float(a_in[j,   7]   /
100))
                                      ))
        soft_outputs[j,   :]   =   np.hstack((softmax_array(float(a_in[j   +5,
0]/100)),
                                      softmax_array(float(a_in[j       +5,
1]/100)),
                                       softmax_array(float(a_in[j      +5,
2]/100)),
                                       softmax_array(float(a_in[j      +5,
3]/100)),
                                       softmax_array(float(a_in[j  +  5,  4]
/ 100)),
                                       softmax_array(float(a_in[j  +  5,  5]
/ 100)),
                                       softmax_array(float(a_in[j  +  5,  6]
/ 100)),
                                       softmax_array(float(a_in[j  +  5,  7]
/ 100))
                                       ))
    for i in range(3):
        X[i, :, :] = soft_inputs
        T[i, :, :] = soft_outputs

    return X, T, file_len

# Raliza a transformada inversa softmax
def soft_inv(x):
    DIM = 11
    maximum = 100
```

```
    minimum = -100
    intervals = minimum + np.arange(DIM) * float(maximum - minimum) /
float(DIM)
    k = intervals*x
    return np.sum(k)

# Divide o vetor de saida em dois vetores
def out_to_plot(x):
    leng = x.shape
    saidas = []
    #aidas = np.zeros((leng[1],leng[2]/11))
    for l in range(leng[1]):
        saida = {}
        saida[LEFT_ARM_PITCH] = soft_inv(x[0,l, :11])
        saida[LEFT_ARM_ROLL] = soft_inv(x[0,l, 11:22])
        saida[LEFT_ARM_YAW] = soft_inv(x[0,l, 22:33])
        saida[LEFT_ELBOW] = soft_inv(x[0,l, 33:44])
        saida[RIGHT_ARM_ROLL] = soft_inv(x[0, l, 44:55])
        saida[RIGHT_ARM_PITCH] = soft_inv(x[0, l, 55:66])
        saida[RIGHT_ARM_YAW] = soft_inv(x[0, l, 66:77])
        saida[RIGHT_ELBOW] = soft_inv(x[0, l, 77:])
        saidas.append(saida)

    return saidas

######################
## Classes da MTRNN ##
######################

# Define a classe TensorLinear
class TensorLinear(object):
    def     __init__(self,     n_in,     n_out,     tensor_order,     W=None,
b=None,in_out=None):
        """Initialse the weight W and bias b parameters."""
        a = np.sqrt(6.0 / (n_in + n_out))
        self.W = (np.random.uniform(-a, a, (n_in, n_out)) if W is None else
W)
        if in_out=='I':
            self.W[:,nb_of_fast:]       =       np.zeros((n_in,nb_of_states-
nb_of_fast))
        if in_out=='O':
            self.W[nb_of_fast:,:]         =         np.zeros((nb_of_states-
nb_of_fast,n_out))
        self.b = (np.zeros((n_out)) if b is None else b)  # Bias
        self.bpAxes = tuple(range(tensor_order-1))   # Axes summed over in
backprop

    def forward(self, X,ini=None):
        """Performa a passage forward"""
        return np.tensordot(X, self.W, axes=((-1),(0))) + self.b

    def backward(self, X, gY, ini=None):
        """Retorna os gradients dos parametros e das entradas"""
        gW = np.tensordot(X, gY, axes=(self.bpAxes, self.bpAxes))
        if ini == 'I':
            gW[:,   nb_of_fast:]   =   np.zeros((nb_inputs,   nb_of_states   -
nb_of_fast))
        if ini == 'O':
            gW[nb_of_fast:,   :]   =   np.zeros((nb_of_states   -   nb_of_fast,
nb_outputs))
```

```
        gB = np.sum(gY, axis=self.bpAxes)
        if (ini != 'I') and (ini != 'O') :
            Wtemp = (np.dot(self.W,tau_inv))
            gX = np.tensordot(gY, Wtemp.T, axes=((-1), (0)))
        else:
            gX = np.tensordot(gY, self.W.T, axes=((-1), (0)))
        return gX, gW, gB

# Define a classe LogisticClassifier
class LogisticClassifier(object):
    """Aplica a funcao logistica nas entradas"""

    #def forward(self, X):
        #"""Performa a transformacao forward."""
        return 1 / (1 + np.exp(-X))

    def backward(self, Y, T):
        """Retorna o gradiente de acordo com a funcao custo."""
        return (2*(Y - T)) / (Y.shape[0] * Y.shape[1])

    def cost(self, Y, T):
        """Calcula a funcao custo."""
        return np.sum(np.square(Y - T)) / (Y.shape[0] * Y.shape[1])

# Define a classe TanH
class TanH(object):
    """APlica a funcao tangent hiperbolico nas entradas."""

    def forward(self, X):
        """Performa a transformacao forward."""
        return np.tanh(X)  # TanH



    def backward(self, Y, output_grad):
        """Retorna o gradiente da camada."""
        gTanh = 1.0 - np.power(Y,2)  # TanH
        return np.multiply(gTanh, output_grad)

 # Define a classe RecurrenteStateUpdate
class RecurrentStateUpdate(object):
    """Atualiza o estado dado."""
    def __init__(self, nbStates, W, b):
        """Inicializa a transfomracao linear a funcao TanH."""
        self.linear = TensorLinear(nbStates, nbStates, 2, W, b)
        self.tanh = TanH()
        self.u2 = np.zeros((mb_size,nb_of_states))

    def forward(self, Xk, Sk):
        """Retorna o estado k+1 a partir das entradas e do estado k."""
        u = np.dot(self.u2,(np.eye(nb_of_states)-tau_inv)) + np.dot((Xk +
self.linear.forward(Sk)),tau_inv)
        C = self.tanh.forward(u)
        self.u2 = u
        return C

    def backward(self, Sk0, Sk1, output_grad):
        """Return the gradient of the parmeters and the inputs of this
layer."""
        gZ = self.tanh.backward(Sk1, output_grad
```

```
        gSk0, gW, gB = self.linear.backward(Sk0, gZ)
        return gZ, gSk0, gW, gB

# Define a classe RecurrentStateUnfold
class RecurrentStateUnfold(object):
    """Desdobra os estados no tempo."""
    def __init__(self, nbStates, nbTimesteps):
        " Inicializa os parametros compartilhados, o estado inicial e a
funcao de atualizacao."
        a = np.sqrt(6.0 / (nbStates * 2))
        self.W = np.random.uniform(-a, a, (nbStates, nbStates))
        self.b = np.zeros((self.W.shape[0]))  # bias compartilhado
        self.S0 = np.zeros(nbStates)  # Estado inicial
        self.nbTimesteps = nbTimesteps  # Timesteps para desdobrar
        self.stateUpdate = RecurrentStateUpdate(nbStates, self.W, self.b)
# funcao de atualizacao dos estados

    def forward(self, X, S0=None):
        """Aplica iterativamente a passagem forward em todos os estados."""
        S = np.zeros((X.shape[0], X.shape[1]+1, self.W.shape[0]))  # State
tensor
        if S0 is None:
            S[:, 0, :] = self.S0
        else:
            S[:, 0, :] = S0
        for k in range(self.nbTimesteps):
            # Atualiza os estados iterativamente
            S[:,k+1,:] = self.stateUpdate.forward(X[:,k,:], S[:,k,:])
        return S

    def backward(self, X, S, gY):
        """Retorna o gradiente dos parametros e das entrads da camada."""
        gSk = np.zeros_like(gY[:,self.nbTimesteps-1,:])  # Inicializa os
gradients de saida dos estados
        gSk_old = np.zeros_like(gY[:,self.nbTimesteps-1,:])
        gZ = np.zeros_like(X)
        gWSum = np.zeros_like(self.W)  # Inicializa os gradients dos pesos
        gBSum = np.zeros_like(self.b)  # Inicializa os gradients dos bias
        # Propaga os gradients iterativamente
        for k in range(self.nbTimesteps-1, -1, -1):
            gSk             =             np.dot(gSk,tau_inv)             +
np.dot(gSk_old,(np.eye(nb_of_states) - tau_inv)) + gY[:, k, :]
            gSk_old = gSk
            # Propaga o gradiente para o tempo anterior
            gZ[:,k,:], gSk, gW, gB = self.stateUpdate.backward(S[:,k,:],
S[:,k+1,:], gSk)
            gWSum += gW # Atualiza o gradiente dos pesos
            gBSum += gB  # Atualiza o gradinte dos bias
        gS0 = np.sum(gSk, axis=0)  # Calcula o gradiente inicial dos
estados
        return gZ, np.dot(gWSum,tau_inv), np.dot(gBSum,tau_inv), gS0


 # Define a rede completa
class RecurrentNeuralNetwork(object):
    """MTRNN."""
    def __init__(self, nb_of_inputs, nb_of_outputs, nb_of_states,
sequence_len):
        """Inicializa as camadas da rede."""
```

```
        self.tensorInput    =    TensorLinear(nb_of_inputs,    nb_of_states,
3,in_out='I')  # Camada de entrada
        self.rnnUnfold  =  RecurrentStateUnfold(nb_of_states,  sequence_len)
# Camada recorrente
        self.tensorOutput   =   TensorLinear(nb_of_states,   nb_of_outputs,
3,in_out='O')  # Transformacao linear
        self.classifier    =    LogisticClassifier()        #    Saida    do
LogisticClassifier
        self.gRecOut = np.zeros((mb_size,sequence_len,nb_of_states))

    def forward(self, X, S0=None):
        """Performa a propagacao da entrada X por todas as camadas."""
        recIn = self.tensorInput.forward(X)
        S = self.rnnUnfold.forward(recIn,S0)
        Z = self.tensorOutput.forward(S[:,1:sequence_len+1,:],'I')
# Propaga a entrada X através da rede desdobrada no tempo
        Y = Z
        # Retorna a entrada para as camadas recorrentes, os estados,
entrada para a LogisticClassifier e saída da rede
        return recIn, S, Z, Y

    def backward(self, X, Y, recIn, S, T):
        """Performa a propagacao backward pela rede."""
        gZ = self.classifier.backward(Y, T)  # Calcula o gradiente de saida
        gRecOut,                gWout,                gBout                =
self.tensorOutput.backward(S[:,1:sequence_len+1,:], gZ,'O')
        self.gRecOut = gRecOut + np.dot(self.gRecOut,(np.eye(nb_of_states)
- tau_inv))
        # REaliza a propagacao backward na rede
        gRnnIn,  gWrec,  gBrec,  gS0  =  self.rnnUnfold.backward(recIn,  S,
self.gRecOut)
        gX, gWin, gBin = self.tensorInput.backward(X, gRnnIn,'I')
        # Retorna os gradients dos parametros: pesos da saída linear, bias
de saída linear, pesos da recursao, bias da recursao, pesos da entrada
linear, bias da entrada linear, estado inicial.
        return gWout, gBout, gWrec, gBrec, gWin, gBin, gS0

    def getOutput(self, X):
        """Retorna somente a saida."""
        recIn, S, Z, Y = self.forward(X)
        return Y

    def getRealOutput(self, X, S0=None):
        """Retorna a saida e os estados."""
        recIn, S, Z, Y = self.forward(X,S0)
        return Z,S

    def getParamGrads(self, X, T):
        """Retorna os gradients com respeito a X e T em foram de lista.
        A  lista  é  ordenada  da  mesma  forma  que  o  iterador  de
get_params_iter."""
        recIn, S, Z, Y = self.forward(X)
        gWout, gBout, gWrec, gBrec, gWin, gBin, gS0 = self.backward(X, Y,
recIn, S, T)
        return [g for g in itertools.chain(
                np.nditer(gS0),
                np.nditer(gWin),
                np.nditer(gBin),
                np.nditer(gWrec),
                np.nditer(gBrec),
```

```
            np.nditer(gWout),
            np.nditer(gBout))]

def cost(self, Y, T):
    """Retorna o custo com respeito a Y e T."""
    return self.classifier.cost(Y, T)

def get_params_iter(self):
    """Retorna um iterador dos parametros.
    Os elementos podem ser editados no local."""
    return itertools.chain(
        np.nditer(self.rnnUnfold.S0, op_flags=['readwrite']),
        np.nditer(self.tensorInput.W, op_flags=['readwrite']),
        np.nditer(self.tensorInput.b, op_flags=['readwrite']),
        np.nditer(self.rnnUnfold.W, op_flags=['readwrite']),
        np.nditer(self.rnnUnfold.b, op_flags=['readwrite']),
        np.nditer(self.tensorOutput.W, op_flags=['readwrite']),
        np.nditer(self.tensorOutput.b, op_flags=['readwrite']))
```

ANEXO B – CÓDIGO PARA TREINAMENTO DA REDE

```
###################################
##Treinamento da rede utilizando  ##
##RMSProp otimizado com gradiente ##
##acelerado de Nesterov           ##
###################################

nbParameters = sum(1 for _ in RNN.get_params_iter()) # Numero de
parametros na rede
maSquare = [0.0 for _ in range(nbParameters)]  # Media movel de Rmsprop
Vs = [0.0 for _ in range(nbParameters)]  # Velocidade

# Cria lista de custos a ser plotada
ls_of_costs = [RNN.cost(RNN.getOutput(Xtest[:mb_size,:,:]),
Ttest[:mb_size,:,:])]
# Itera sobre algumas iteracoes
for i in range(nb_iteration):
    # Itera sobre todas as mini-batches
    for mb in range(nb_train/mb_size):
        X_mb = Xtest[mb*mb_size:(mb*mb_size)+mb_size,:,:] # Input
minibatch
        T_mb = Ttest[mb*mb_size:(mb*mb_size)+mb_size,:,:] # Target
minibatch
        V_tmp = [v * momentum_term for v in Vs]
        # Atualiza cada parametron de acordo com o gradiente anterior
        for pIdx, P in enumerate(RNN.get_params_iter()):
            P += V_tmp[pIdx]
        # Calcula os gradients depois de seguir a velocidade antiga
        backprop_grads = RNN.getParamGrads(X_mb, T_mb) # Calcula os
gradients dos parametros
        # Atualiza cada parametron separadamente
        for pIdx, P in enumerate(RNN.get_params_iter()):
            # Atualiza a media movel RMSProp
            maSquare[pIdx] = lmbd * maSquare[pIdx] + (1-lmbd) *
backprop_grads[pIdx]**2
            # Calcula o gradiente normalizado do RMSProp
            pGradNorm = learning_rate * backprop_grads[pIdx] /
np.sqrt(maSquare[pIdx] + eps)
            # Atualiza a velocidade do momento
            Vs[pIdx] = V_tmp[pIdx] - pGradNorm
            P -= pGradNorm  # Atualiza o parametro
        getoutput = RNN.getOutput(X_mb)
        custo = RNN.cost(getoutput, T_mb)
        ls_of_costs.append(custo)  # Adiciona o custo a lista de custos
```