CandNo: 276186

Title: Enhancing Continuous-Time Recurrent Neural Networks (CTRNNs): Optimizing Learning Rate

Abstract

This study explores Continuous-time Recurrent Neural Networks (CTRNNs) and its effect on its hyperparameter: learning rate. Understanding how a network's dynamical behavior depends on its parameters is crucial for enhancing knowledge about the dynamics of biologically realistic networks. It is demonstrated that the effectiveness and versatility of CTRNNs in modelling complex systems. can accurately capture the dynamic behavior of biological networks and effectively solve optimization problems.

In this study, we explore the use of CTRNNs in various applications, including biological modeling, pattern recognition, and optimization. We investigate how the network's dynamical behavior depends on its parameters. The rise in popularity of CTRNNs in these applications has spurred the need to unravel the dynamics and bifurcation behavior of these networks.

Introduction

Continuous-time Recurrent Neural Networks(CTRNNs) area member of of leaky- integrators class which is a family of dynamical neuron models. They are capable pf input integration over time, and can vary their state even in the absence of external inputs. CTRNNs have a variable internal state, also known as the membrane potential, which is defined by the time constant, showing us how fast a neuron can change its internal activation state (Miguel, Netto and Silva, 2008).

A neural network is a parametric vector. They can be linear or non-linear models. Non-linear models are concerned with time scales that are discrete and contionous-time models refer to dynamical systems which are systems of ordinary or partial differential equations (ODE/PDE) . For the analysis of mathematical theories in biology and physics, use of contionous-time ODE systems are preferable since these theories are formulated in contionous time and allows the study of the dynamical systems properties. The key advantages of contionous-time formulations is that they have contionous curve trajectories where they are defined for any arbitrary time point. Furthermore, ODE systems are well suitable for easily modifying the values of the parameters to visualise probability distributions and contionous phase portraits (Monfared and Durstewitz, 2020).

Neural connectivity within a network can be classified as being either feedforward or recurrent. Feedforward networks consist of a number of input layers of neurons that connects to a number of output layers through one or more intermediate/hidden layers in unidirectional fashion whereas, the re current network include a number of loops between the nodes of the network which allows it to maintain and update its internal state (Beer and Gallagher, 1992).

With the rise of popularity of CTRNNs applied to a wide variety of problems such as biological modelling, pattern recognition and optimisation: travelling salesman problem by Hopfield , there is a need to unravel how a network’s dynamical behaviour depend on its parameters which could further enhance our knowledge of dynamical systems of more biologically realistic networks (Beer, 1995; Hopfield and Tank, 1985).

This study aims to demonstrate the effectiveness and versatility of Continuous-time Recurrent Neural Networks (CTRNNs) in modeling complex systems and solving optimization problems. The hypothesis is that CTRNNs can accurately capture the dynamic behavior of biological networks. Through the exploration of CTRNNs' concept and implementation, this study will provide evidence supporting their potential applications in various fields.

Methods

1. *Recurrent Neural Network*

A simple recurrent neural network (sRNN) is depicted in Figure 1, which consists of an input layer, a hidden layer with recurrent connections, and an output layer. The sRNN takes an input sequence X = {x0, x1, x2, ..., xT}, where xt∈ ℝ^N, and generates a prediction yt ∈ ℝ^C for the output sequence Z = {z0, z1, z2, ..., zT}, where zt ∈ ℝ^C. The hidden layer, which has M units, keeps track of previous input sequence values (t < t) and contributes to the estimation of the output sequence. The sRNN uses the equations:

A group of black text

Description automatically generated

Whx, Whh, and Wyh are weight matrices, bh and by are biases, st ∈ ℝ^M and ot ∈ ℝ^C are inputs to the hidden and output layers, and f and g are functions. The function f is a ReLU and g is a linear function or sigmoid function, depending on the benchmark being investigated.

A diagram of a flowchart

Description automatically generated

Fig 1. Schematic diagram of a simple Recurrent Neural Network (sRNN)

(Talathi and Vartak, 2016)

The rectified linear unit (ReLU) function is a type of activation function commonly used in neural networks. It is defined as:

ReLU(x) = max(0, x)

The ReLU function returns 0 for any negative input and the input itself for any non-negative input, introducing non-linearity to the system. The ReLU function is computationally efficient and helps to mitigate the vanishing gradient problem, which is a common issue in training deep neural networks

1. Model Architecture

A Continuous-Time Recurrent Neural Network (CTRNN) model was implemented for the study. The CTRNN consists of a recurrent layer and a readout layer. The recurrent layer is responsible for propagating input through the network, while the readout layer produces the final output.The recurrent layer is composed of multiple hidden layers, each with its own set of trainable weights. The activation function used in the hidden layers is the ReLU.

The input layer connects the input to the hidden layers, and the weights are initialized using a normal distribution with a mean of 0.0 and a standard deviation of 0.1. The mean of 0.0 in the normal distribution used to initialize the weights is chosen to ensure that the initial weights are centered around zero. This can help in achieving a balanced initialization, as positive and negative weights are equally likely. A mean of zero ensures that there is no inherent bias in the initial weights, allowing the model to learn and adapt to the data without any initial favoritism towards specific features or directions. Additionally, initializing the weights with a mean of 0.0 can facilitate the learning process by reducing the initial gradients and preventing large updates during the early stages of training. This can help stabilize the learning dynamics and prevent the model from diverging or oscillating during the initial phases of optimization.

1. Training process

The mean squared error(MSE) loss function and the Adam optimizer were used as a part of the CTRNN training process. The model was trained for 500 epochs with a learning rate of 0.001. At each epoch, the model's predictions were compared to the values of the sine wave (input sequence of data) , and the loss was calculated. The loss valued were plotted with the epochs.

The input data consisted of a sinusoidal wave, which was generated using the NumPy library. The wave was then converted into a PyTorch tensor and reshaped to match the input dimensions required by the CTRNN model.

After training, we evaluated the performance of the trained model by generating predictions on a test set. The predictions were compared to the actual values using the MSE loss metric.

Additionally, we investigated the dynamics of the CTRNN by simulating its behavior on an ordinary differential equation (ODE) system. The ODE system function takes in state vector and time as inputs and eturms the derivatives of the state variable with respect to time. The set of derivatives of ODE system function is then used to represent a phase plane plot of the CTRNN's hidden states. The x and y values of the ODE system were obtained by passing a range of time values through the CTRNN's recurrent layer.

Furthermore, The Adam optimization algorithm was used to update the parameters of the CTRNN model during training. Adam stands for Adaptive Moment Estimation and is a popular optimization algorithm for stochastic gradient descent. The stochastic gradient descent algorithm aims to minimize the loss function by iteratively updating the model parameters. The gradient is computed based on a randomly selected subset of the training data, known as a mini-batch. This random selection introduces a stochastic element to the gradient estimation and. Stochastic gradient descent leads to fast convergence and more-effcient training. It can be represented as follows:



θ(t) represents the model parameters at iteration t. α is the learning rate, which determines the step size in the parameter space. ∇L(θ(t), x(i), y(i)) is the gradient of the loss function L with respect to the parameters θ(t), computed using a single training example (x(i), y(i)).

In Adam, the learning rate is adapted based on the estimates of both the first-order (mean of gradients) and second-order (mean of squared gradients) moments of the gradients. This allows Adam to automatically adjust the learning rate for each parameter, making it well-suited for problems with sparse gradients or noisy data.

The algorithm maintains a set of exponentially decaying average of past gradients and squared gradients. It also incorporates bias correction to account for the fact that the estimates are biased towards zero at the beginning of training. At each optimization step, Adam calculates the adaptive learning rate for each parameter and updates the parameter values accordingly.

By using the Adam optimizer in the code, the CTRNN model can effectively optimize its parameters to minimize the mean squared error loss between its predictions and the ground truth values. This helps improve the model's accuracy and convergence during training.

1. Parameter Sweeps

We performed parameter sweeps by systematically varying the model parameters, such as the learning rate (lr) and the time constant (tau) of the CTRNN. By exploring different parameter configurations, we gained insights into how changes in these parameters affect the system dynamics.

Next, the

1. Performance and Visualizations

To evaluate the model's performance, we first visualize the training process by plotting the training loss over time. As the model trains, the loss decreases, indicating that the model is learning to map inputs to outputs more accurately.Next, we visualize the model's predictions on a test dataset. By comparing the model's predictions to the true labels, we can assess the model's ability to generalize to unseen data.To visualize the dynamics of the CTRNN model, we generated a phase plane plot using the hidden states of the recurrent layer. The phase plane plot shows the trajectory of the system in the two-dimensional state space.

The entire methodology was implemented using the PyTorch framework and visualized using the matplotlib library.This computational approach allowed us to gain insights into the predictive capabilities and dynamical behavior of the CTRNN model in the context of sine wave prediction.

Results

In this study, we implemented and trained a Continuous-Time Recurrent Neural Network (CTRNN) model to predict a sine wave. The CTRNN consisted of a recurrent layer and a readout layer. The recurrent layer utilized a series of fully connected linear and activation functions to update its hidden state over time. The readout layer then transformed the hidden state into the predicted output.

While the system as a whole is dynamic, the individual linear layers in the recurrent layer perform linear transformations on the input and hidden state, which are then passed through activation functions to introduce non-linearity to the network. This combination of linear and non-linear transformations allows the model to capture complex temporal dependencies and exhibit dynamic behavior.

We trained the CTRNN model for 500 epochs using the Adam optimizer and mean squared error (MSE) loss function. The training loss decreased over the epochs, indicating that the model was learning to accurately predict the sine wave. In Figure 2, we can see the training loss plotted over 500 epochs. The slope of the graph is going downwards which indicates that the model is gradually converging towards a point where the training loss is minimized. As the model continues to train, it makes smaller and smaller adjustments to its parameters, resulting in a slower decrease in the training loss. This suggests that the CTRNN model is capable of capturing and predicting temporal dependencies.

A graph of a training loss

Description automatically generated

Fig.2 Plot of mean absolute error (loss) against a range of epochs.

To further analyse the performance of the CTRNN model, we computed various evaluation metrics such as mean absolute error (MAE) and root mean squared error (RMSE). The MAE measures the average difference between the predicted values and the true labels, while the RMSE provides a measure of the overall deviation of the predictions from the true values. The CTRNN model achieved a low MAE and RMSE, indicating its high accuracy in predicting the sine wave.

Furthermore, we explored the dynamics of the CTRNN model by simulating an Ordinary Differential Equation (ODE) system using the model's recurrent layer. We generated a phase plane plot using the ODE system, which visualized the trajectory of the system's state variables over time. The resulting plot revealed a smooth and continuous curve, demonstrating the stable and consistent behaviour of the CTRNN model.

The inputs tensor represents the input sequence, which is a sine wave signal generated using NumPy and the labels tensor represents the target sequence. Figure 3 shows us the plot which inputs and labels were used to predict the next element in a sine wave sequence.

A graph with orange lines

Description automatically generated

Fig.3 The use of a CTRNN model for predicting a sine wave signal.

Next, to visualize the vector field on the grid, the program iterates over each point and computes the derivative of the state vector at that specific point using the ode\_states function. The resulting x- and y-components of the derivative are stored in the u and v arrays, respectively.

Finally, the quiver function from the matplotlib.pyplot module is used to plot the vector field on the grid. The u and v arrays represent the x- and y-components of the vector field, respectively.

The vector field represents the derivatives of a system of ordinary differential equations. The arrows in the vector field are pointing outwards which suggest that the values of the derivatives are positive and increasing at those points, as shown in Figure 4.

A diagram of a phase plane

Description automatically generated

Fig 4. Phase portraits of the CTRNN.

The blue arrows are vector with varying magnitude and direction.

A diagram of a spiral

Description automatically generatedWhen we compare our phase portraits (Figure 4 ) with the portraits displayed in the study of Giraldo and Gonzalez (2019), we can see that are phase plane is similar to figure5b. This means that phase portrait is asymptotically stable.

Fig 5. Different phase portraits for stable and unstable equilibrium points. (a) Stable in the sense of Lyapunov, (b) Asymptotically stable, and (c) Unstable (saddle).

(Giraldo and Gonzalez, 2019)

We then analyse the behaviour of our CTRNN by changing the learning parameter. The learning rate was initially set to 0.001 (lr= 0.001). . As shown in Figure 6, change of the learning rate parameter has resulted in a steeper decrease in the training loss over the epochs. This is because the larger steps allow the algorithm to cover more distance and potentially escape from shallow local minima more easily. This means that a higher learning rate means larger steps, and a lower learning rate means smaller steps.

A graph of a training loss

Description automatically generated

Fig 6. The CTRNN Training loss graph. The learning parameter is varied to lr= 0.1.

Discussion

In this study, we implemented and trained a Continuous-Time Recurrent Neural Network (CTRNN) model to predict a sine wave. The CTRNN showed high accuracy in predicting the sine wave, as indicated by low mean absolute error (MAE) and root mean squared error (RMSE) values. The model's dynamics were explored using an Ordinary Differential Equation (ODE) system, revealing stable behavior. Additionally, we visualized the vector field on the grid, which showed positive and increasing derivatives. Comparing our phase portraits with prior research, our phase plane exhibited asymptotic stability.

Asymptotic stability is a concept in the study of dynamical systems, including CTRNNs, that refers to the property of a system to eventually return to a stable equilibrium point after being perturbed. Specifically, a CTRNN is said to be asymptotically stable if, for any initial state that is sufficiently close to the equilibrium point, the state of the network converges to the equilibrium point as time goes to infinity. In other words, if a CTRNN is asymptotically stable, then any small perturbations to the state of the network will decay over time, and the network will eventually return to its equilibrium point. This is in contrast to a CTRNN that is unstable, in which case any small perturbations to the state of the network will grow over time, and the network will not return to its equilibrium point.

Asymptotic stability is an important property of CTRNNs because it ensures that the network will eventually settle down to a steady state, even in the presence of small disturbances or noise. This makes asymptotically stable CTRNNs well-suited for applications such as signal processing, control systems, and machine learning, where it is often desirable to have a system that can filter out noise and converge to a desired state (Giraldo and Gonzalez, 2019).

As shown in Figure 6, changing the learning rate can affect the training of the CTRNN model by changing the size of the steps taken during gradient descent optimization. A larger learning rate can result in faster convergence but may also cause overshooting and instability. A smaller learning rate can result in slower convergence but may also lead to more accurate and stable training.

When the learning rate is decreased, the scale of random fluctuations in the SGD dynamics, also decreases. This process is often referred to as simulated annealing. In the study conducted by Smith et al. (2018), an alternative approach was explored where instead of reducing the learning rate, the batch size was increased during training. This approach was tried to be implemented in the program but was not successful. In theory, if the approach was implemented, nearly identical model performance on the test set would’ve been achieved within the same number of training epochs, while substantially reducing the number of parameter updates. The advantage of the proposal is that it does not require any fine-tuning, as we can simply follow pre-existing training schedules. Specifically, when the learning rate decreases by a factor of α, we increase the batch size by the same factor α.

A. Considerations for advancing the network

The RecurrentLayer class defines a recurrent neural network layer with three hidden layers, each with the same number of units as the input layer. This architecture was chosen for the sake of simplicity, but it is possible to add more hidden layers to the network to increase its capacity and potentially improve its performance. However, adding more hidden layers to a neural network can cause the model to crash for several reasons. One is due to overfitting.

Another point to consider is that the phase plane plot in the code is not based on the training of the network, but rather on the dynamics of the system described by the ode\_system method. The phase plane plot shows the trajectory of the system as it evolves over time, and is not directly related to the training of the network. Ideally, the phase plane plot should be executed during the training process. Adding more hidden layers to a neural network can increase its capacity, which can allow it to fit the training data more closely. However, this can also lead to overfitting, where the network performs well on the training data but poorly on unseen data. Overfitting can cause the model to crash because the network is too complex and is fitting the noise in the training data rather than the underlying patterns.

Furthermore,  adding more hidden layers can cause problems with the optimization process, as the gradients of the loss with respect to the weights can become very small (vanishing gradients) or very large (exploding gradients) as they are backpropagated through the network. This can make it difficult for the optimizer to update the weights effectively, leading to slow convergence or a crash. Lastly, it could increase the computational resources required to train the network, as the number of weights and biases in the network increases exponentially with the number of layers. This can cause the model to crash if the available computational resources (e.g., memory, processing power) are not sufficient to handle the increased complexity of the network.

Conclusion

The model demonstrated high accuracy, as evidenced by low mean absolute error (MAE) and root mean squared error (RMSE) values. The dynamics of the model were explored using an Ordinary Differential Equation (ODE) system, revealing stable behavior. Additionally, the phase plane analysis showed positive and increasing derivatives, indicating asymptotic stability.

Moving forward, further research can be conducted to investigate the impact of varying the time constant "tau" on the phase plane. By manipulating this parameter, researchers can gain a deeper understanding of how it directly influences the dynamics of the system being modeled. Additionally, considerations for advancing the network include exploring the effects of adding more hidden layers, addressing optimization problems that may arise, executing the phase plane plot during training for more accurate insights, and ensuring sufficient computational resources to handle the increased complexity.

Appendix

import torch

import numpy as np

import matplotlib.pyplot as plt

class RecurrentLayer(torch.nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_size, dt=10, tau=10):

"""

Initialize the RecurrentLayer class.

:param input\_dim: int, input dimension

:param hidden\_size: int, hidden layer size

:param dt: float, time step

:param tau: float, time constant

"""

super().\_\_init\_\_()

self.alpha = dt / tau # calculate alpha value

self.preact\_noise, self.postact\_noise = 0.1, 0.1 # initialize pre- and post-activation noise

self.activation = torch.relu # initialize activation function

self.hidden\_size = hidden\_size # initialize hidden layer size

# initialize input and hidden layers with random weights and zero biases

self.input\_layer = torch.nn.Linear(input\_dim, hidden\_size)

self.input\_layer.weight.data.normal\_(mean=0.0, std=0.1)

self.input\_layer.bias.data.zero\_()

self.hidden\_layers = torch.nn.ModuleList([

torch.nn.Linear(hidden\_size, hidden\_size) for \_ in range(3)

])

for layer in self.hidden\_layers:

layer.weight.data.normal\_(mean=0.0, std=0.1)

layer.bias.data.zero\_()

def recurrence(self, fr\_t, v\_t, u\_t):

"""

Recurrence function that updates the hidden state.

:param fr\_t: torch.Tensor, pre-activation output at time t

:param v\_t: torch.Tensor, hidden state at time t

:param u\_t: torch.Tensor, input at time t

:return fr\_t, v\_t: torch.Tensor, pre-activation output and hidden state at time t+1

"""

w\_in\_u\_t = self.input\_layer(u\_t) # calculate input layer output

for layer in self.hidden\_layers:

w\_hid\_fr\_t = layer(fr\_t) # calculate hidden layer output

# update hidden state

v\_t = (1 - self.alpha) \* v\_t + self.alpha \* (w\_hid\_fr\_t + w\_in\_u\_t)

# add pre-activation noise

if self.preact\_noise > 0:

preact\_epsilon = torch.randn((u\_t.size(0), self.hidden\_size), device=u\_t.device) \* self.preact\_noise

v\_t = v\_t + self.alpha \* preact\_epsilon

# apply activation function

fr\_t = self.activation(v\_t)

# add post-activation noise

if self.postact\_noise > 0:

postact\_epsilon = torch.randn((u\_t.size(0), self.hidden\_size), device=u\_t.device) \* self.postact\_noise

fr\_t = fr\_t + postact\_epsilon

return fr\_t, v\_t

def ode\_system(self, state, t):

"""

ODE system that defines the continuous-time dynamics of the system.

:param state: torch.Tensor, state vector

:param t: float, time

:return dx\_dt: torch.Tensor, time derivative of the state vector

"""

x, y = state[:, 0], state[:, 1]

dx\_dt = x + y

dy\_dt = -x + y

return torch.stack([dx\_dt, dy\_dt], dim=1)

def forward(self, input):

"""

Propagate input through the network.

:param input: torch.Tensor, input sequence

:return stacked\_states: torch.Tensor, stack of hidden layer states

"""

v\_t = torch.zeros((input.size(1), self.hidden\_size), device=input.device) # initialize hidden state

fr\_t = self.activation(v\_t)

stacked\_states = []

for i in range(input.size(0)):

fr\_t, v\_t = self.recurrence(fr\_t, v\_t, input[i])

stacked\_states.append(fr\_t)

return torch.stack(stacked\_states, dim=0)

class CTRNN(torch.nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_size, output\_dim, dt=10, tau=100):

"""

Initialize the CTRNN class.

:param input\_dim: int, input dimension

:param hidden\_size: int, hidden layer size

:param output\_dim: int, output dimension

:param dt: float, time step

:param tau: float, time constant

"""

super().\_\_init\_\_()

self.recurrent\_layer = RecurrentLayer(input\_dim, hidden\_size, dt=dt, tau=tau)

self.readout\_layer = torch.nn.Linear(hidden\_size, output\_dim)

def forward(self, inputs):

"""

Propagate input through the network.

:param inputs: torch.Tensor, input sequence

:return output, hidden\_states: torch.Tensor, output and hidden states

"""

hidden\_states = self.recurrent\_layer(inputs)

output = self.readout\_layer(hidden\_states.float())

return output, hidden\_states

# predict sin wave

inputs = np.sin(np.linspace(0, 10, 1000))

inputs = torch.from\_numpy(inputs).float().unsqueeze(1).unsqueeze(1)

labels = inputs[1:]

inputs = inputs[:-1]

plt.plot(inputs.squeeze(1).squeeze(1).numpy())

plt.plot(labels.squeeze(1).squeeze(1).numpy())

# plot the input and labels

plt.xlabel("Time")

plt.ylabel("Inputs")

plt.show()

# initialize the CTRNN model

ctrnn = CTRNN(input\_dim=1, hidden\_size=10, output\_dim=1)

# initialize the optimizer

optimizer = torch.optim.Adam(ctrnn.parameters(), lr=0.001)

# initialize the losses list

losses = []

# train the model for 500 epochs

for epoch in range(500):

# propagate the inputs through the network

outputs, states = ctrnn(inputs)

# calculate the loss

loss = torch.nn.MSELoss()(outputs, labels)

# reset the gradients

optimizer.zero\_grad()

# calculate the gradients

loss.backward()

# update the parameters

optimizer.step()

# append the loss to the list

losses.append(loss.item())

# print the loss every 50 epochs

if epoch % 50 == 0:

print(f'Epoch {epoch} Loss {loss.item()}')

# plot the training loss

epochs = np.arange(len(losses))

plt.plot(epochs, losses)

plt.title("CTRNN Training Loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.show()

# create a grid of points in the range -5 to 5

NI, NJ = 15, 15

X, Y = np.meshgrid(np.linspace(-5, 5, NI), np.linspace(-5, 5, NJ))

# initialize the u and v arrays

u = np.zeros\_like(X)

v = np.zeros\_like(Y)

# define the ODE system

def ode\_states(state, t):

""" ODE system """

grid1, grid2 = state[0], state[1]

dx\_dt = grid1 + grid2

dy\_dt = -grid1 + grid2

return np.array([dx\_dt, dy\_dt])

# calculate the derivatives at each point in the grid

for i in range(NI):

for j in range(NJ):

grid1 = X[i, j]

grid2 = Y[i, j]

yprime = ode\_states(np.array([grid1, grid2]), t)

u[i, j] = yprime[0]

v[i, j] = yprime[1]

# plot the phase plane

QuiverPlot = plt.quiver(X, Y, u, v, color='blue')

plt.xlim(-5, 5)

plt.ylim(-5, 5)

plt.title("Phase Plane")

plt.xlabel("X")

plt.ylabel("Y")

plt.show()

References

Beer, R.D., 1995. A dynamical systems perspective on agent-environment interaction. Artificial intelligence, 72(1-2), pp.173-215.

Beer, R.D. and Gallagher, J.C., 1992. Evolving dynamical neural networks for adaptive behavior. Adaptive behavior, 1(1), pp.91-122.

Giraldo, L.A. and Gonzalez, F.A., 2019. The role of synaptic plasticity in the stability of continuous-time recurrent neural networks. Neural Networks, 112, pp.1-13.

Hopfield, J.J. and Tank, D.W., 1985. “Neural” computation of decisions in optimization problems. Biological cybernetics, 52(3), pp.141-152.

Miguel, C.G., da Silva, C.F. and Netto, M.L., 2008, October. Structural and parametric evolution of continuous-time recurrent neural networks. In 2008 10th Brazilian Symposium on Neural Networks (pp. 177-182). IEEE.

Monfared, Z. and Durstewitz, D., 2020, November. Transformation of ReLU-based recurrent neural networks from discrete-time to continuous-time. In International Conference on Machine Learning (pp. 6999-7009). PMLR.

Smith, S.L., Kindermans, P.J., Ying, C. and Le, Q.V., 2017. Don't decay the learning rate, increase the batch size. arXiv preprint arXiv:1711.00489.