UPPSALA
UNIVERSITET

UPTEC F 23033

Examensarbete 30 hp

June 2023

A Bayesian Bee Colony Algorithm for Hyperparameter Tuning of Stochastic SNNs

A design, development, and proposal of a stochastic spiking neural network and associated tuner

Oskar Falkeström

Civilingenjörsprogrammet i teknisk fysik

UPPSALA
UNIVERSITET

A Bayesian Bee Colony Algorithm for Hyperparameter Tuning of Stochastic SNNs

Oskar Falkeström

Abstract

With the world experiencing a rapid increase in the number of cloud devices, continuing to ensure high-quality connections requires a reimagining of cloud. One proponent, edge computing, consists of many distributed and close-to-consumer edge servers that are hired by the service providers. This thesis considers one problem in edge computing, edge user allocation, which involves assigning as many users to as few edge servers as possible. This problem can be formulated as a constraint satisfaction problem (CSP). With many users and edge servers, the complexity of the problem is high, resulting in computationally expensive or possibly infeasible calculations using conventional solvers.

We approach this problem using spiking neural networks. For a spiking neural network supplied with sufficient stochastic noise, the distribution of network states converges to a stationary distribution expressed in terms of an energy function. By appropriately designing the network, it is possible to encode the CSP in a stochastic spiking neural network such that the low energy, high probability, states are solutions to the CSP. To maximize the performance of the stochastic spiking neural network, the synaptic weights and neuron parameters require adjusting or *tuning*. However, the spiking dynamics of the network preclude computation of traditional derivatives, as neurons are governed by discrete and event-based dynamics rather than continuous activation functions. The performance of the spiking neural network is also stochastic. This means that even a poorly tuned network can return good solutions, and vice versa.

In this thesis, a stochastic neural network of spiking neurons is designed to solve the edge user allocation problem. For this network a new hyperparameter tuner is proposed, combining aspects from the explorational artificial bee colony algorithm (ABC) with the exploitation of the tree-structured Parzen estimator algorithm (TPE). This new algorithm, ABC-BA, is designed with the aim of both exploring the solution space and exploiting the promising regions. It is also designed to be less sensitive to the inherent stochasticity of the stochastic spiking neural network.

The network is tuned and evaluated on four problem sizes: 6, 100, 1000, and 10000 users. Results show that the network finds the optimal solution for the smaller problems while finding solutions slightly under optimum for the larger ones. While not guaranteeing optimal solutions, the stochastic network is, compared with the conventional solver, able to find good solutions for the largest problem. The networks with tuned parameters are also tested on unseen problem instances, results suggesting that the tuned parameters function well on similarly sized problems as the one there are tuned to.

To evaluate ABC-BA, the developed algorithm is compared against its two parts. The experiments suggest that ABC-BA outperforms its building blocks in terms of desirable search patterns and parameter performance. An important future research direction is to evaluate whether this conclusion holds for other CSP-solving stochastic spiking neural networks.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Ahsan Javed Awan Ämnesgranskare: Ayca Özcelikkale

Examinator: Tomas Nyberg

Populärvetenskaplig sammanfattning

I dagens moderna samhälle blir antalet moln och mobila enheter allt fler. För att säkerställa fortsatt god anslutningskvalité behövs ny tekik, där data och beräkning flyttas närmare användarna. Detta kan löses med ett större antal servrar, geografiskt utplacerade med delvis överlappande täckning. Varje användare har då möjligheten att ansluta till en av flertal olika servrar, och i det uppstår ett problem. Hur ska tilldelningen ske så att antalet tilldelande användare maximeras samtidigt som antalet använda servrar minimeras?

Ett möjligt tillvägagångsätt för att lösa tilldelningsproblemet är att ta inspiration från vår biologiska hjärna. Hjärnan har tidigare studerats och efterliknats i många olika program och lösningar, där komplicerade sammankopplingar av neuroner och synapser löser svåra problem. I ”den tredje generationen” av dessa nätverk tas ännu ett steg närmare vår biologiska inspirationskälla. Genom att ändra neuronernas beteende från matematiska funktioner till tidsberoende potentialer kan lösningar formas i realtid. De biologiska neuronerna kommunicerar med ”spikar”, korta pulser, som höjer eller sänker potentialen av andra neuroner. Detta ger namnet *spikande neurala nätverk*.

Genom att intelligent utforma det spikande neurala nätverket kan en bra lösningen till ovannämnda tilldelningsproblem tas fram. Inkluderingen av tid i problemlösningen och implementering på speciell "spikande" hårdvara har visat sig ge enorma förbättringar inom energieffektivitet för problem likt användartilldelning, med över tusen gånger längre energiförbrukning.

Det uppstår dock nya utmaningar med dessa nätverk. Traditionella metoder för att optimera prestandan på neurala nätverk fungerar inte på de tidsberoende spikande neurala nätverken. I det här arbetet presenteras, utöver designen av ett spikande neuralt nätverk, en optimeringsmetod. Denna metod är anpassad för de spikande neurala nätverkens unika beteende och funktion, varpå nätverket lösningar förbättras.

Acknowledgements

First and foremost, I would like to express my gratitude to **Ericsson AB** for granting me the opportunity to work on this very interesting and challenging thesis work.

A special thank you goes to my supervisor, **Ahsan Javed Awan**, whose guidance, provision of relevant literature, and valuable insights have been instrumental in the successful completion of this work and the subsequent results.

I would also like to extend my sincere appreciation to my subject reviewer, **Ayca Özcelikkale**, for giving insightful feedback with their expertise in the subject. I am truly grateful for the extra effort they have invested in this role, which has greatly enriched the quality of my thesis.

Finally, I want to convey my deep appreciation to my fellow MSc thesis student, **Nathan Allard**, with whom I share credit on the development of the stochastic spiking neural network. It has been enormously rewarding having another soul as deeply invested in the problem. I do genuinely hope we will be able to keep in contact in the future.

Oskar Falkeström

Contents

Abbreviations

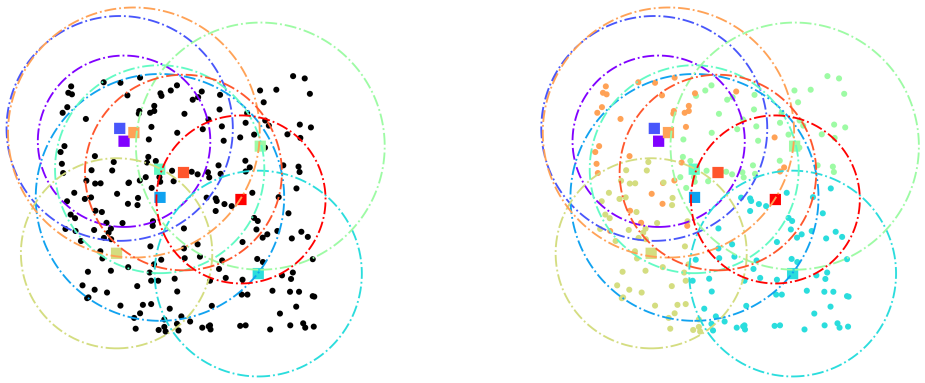
ABC	Artificial Bee Colony
ABC-BA	Bayesian Artificial Bee Colony
ANN	Artificial Neural Network
CSP	Constraint Satisfaction Problem
CWTA	C Winners-take-all
EUA	Edge User Allocation
ILP	Integer Linear Programming
KDE	Kernel Density Estimator
PDF	Probability Density Function
SNN	Spiking Neural Network
TPE	Tree-structured Parzen Estimator
VSVBP	Variable Sized Vector Bin Packing
WTA	Winner-takes-all

1 Introduction

The number of cloud and mobile-connected devices is growing rapidly. According to Ericsson's Mobility Report, it was predicted that there will be around 32 billion such devices by 2023 [1]. To allow continued high bandwidth and low latency connections a reimagining of cloud was proposed, *edge computing*.

In edge computing, online service providers hire edge servers for their services. Edge servers are geographically distributed with the service-ranges, *coverages*, of neighbouring servers usually overlapping, allowing users the possibility to connect to any server that reaches them. This allows for low latency and high bandwidth connections while also allowing clients to offload intensive computational tasks to local servers [2].

This thesis considers the problem of effectively assigning users to the local servers. While perhaps trivial at a glance, maximizing the number of allocated users while minimizing the total amount of hired servers quickly becomes a complex problem. Especially since each server might, based on its resource capacity, only serve a finite number of users. The users in turn cannot be allocated to any server, as some are out-of-range. An example problem and solution can be seen in Figure 1. This problem can be formulated as a *Constraint Satisfaction Problem* (CSP) [1]. In this thesis, this telecommunications-related CSP is approached with an, perhaps, unconventional solver. Here, we look towards adapting nature's neural architecture to computing architecture.



(a) Problem *(b) Solution*

Figure 1: Mock-up allocation problem. Squares and circles represent servers and their coverage and the dots represent users. Unassigned users are shown as black and assigned users show colour matching their server.

The famous quote: "If the human brain were so simple that we could understand it, we would be so simple that we couldn't" [3], has not stopped humanity from using the brain as a source of inspiration, trying to mimic its behaviour. The brain not only accomplishes complex tasks but runs on less than 30 W [4]. Established technologies like neural networks draw inspiration from the complex network of neurons in the brain. However, although their similarity, a big discrepancy between today's artificial neural networks (ANN) and the brain is how the neurons function.

A new generation of ANNs called *Spiking Neural Networks* (SNNs), also tries to incorporate biologically inspired neurons. In an SSN, the neurons are governed by time- and event-dependent behaviour, replacing continuous and differentiable activation functions used in current ANNs. The

inter-neuron communication is facilitated by short pulses, spikes, raising or lowering the membrane voltage of its synapse-connected neighbours. By carefully designing the network structure, stochastic SNNs have been shown to solve CSPs [4]. While not necessarily outperforming ANNs, when run on custom hardware these SNN solvers are orders of magnitude more energy efficient [5].

Regardless, the synaptic strengths and neuron parameters of the SNN still need to be adjusted, *tuned*, for the SNN to work properly and achieve its optimal performance. And while some of the neuron and synaptic strengths are implied from the CSP, there still exists plenty of these hyperparameters that need tuning. This introduced a problem, as traditional approaches such as backpropagation require information about the derivative and the event-driven nature of SNN neurons does not have a traditionally computable derivate. SNNs can instead be tuned by gradient-free algorithms such as evolutionary algorithms, Bayesian algorithms, and particle swarm algorithms. Tuning an SNN is therefore akin to black-box optimization and finding a good tuner/optimizer vital to the network's final performance.

1.1 Background

The driving vision behind this thesis work is two-fold. First, the development and incorporation of these neuromorphic, "brain-like", algorithms in edge computing represent a possibility for quick and energy-efficient solutions. It might then be possible to solve problems in a dynamic setting. Second, by researching and developing a hyperparameter tuner specifically tailored towards stochastic SNNs, future research into other problems will be sped up. With a proper tuner, the true potential of any developed stochastic network can be obtained much quicker.

1.2 Related Work

The topic of stochastic spiking neural networks and algorithms tuning them appear underresearched. Here, similar work is presented and the novelty of this thesis explained.

The idea of using networks of stochastic spiking neurons to solve constraint satisfaction problems was originally proposed by Jonke et al. [4]. It has then been applied in solving problems like Boolean satisfaction and traveling salesman [4], sudoku and map colouring problem [6], and quadratic unconstrained binary optimization [7].

Especially interesting work was made the prior year by my thesis work predecessor Kim Petersson Steenari, where stochastic spiking neural networks are introduced for the edge user allocation problem [8]. Solving the same problem, this thesis uses Petersson's work as an inspiration for fundamental conceptual understanding. This includes the motifs and selection variables, although the motifs are somewhat altered. The novelty of this thesis work is highlighted by an improvement in computed results.

Automatic hyperparameter tuning of *stochastic* spiking neural networks has, to the writer's knowledge, not been attempted prior. All authors in earlier mentioned implementations [4, 6, 7, 8] opt for manual tuning. Automatic tuning of regular spiking neural networks has been done and shown to work. Examples of this include Spike-Timing-Dependent Plasticity with reinforcement learning [9], evolutionary algorithms [10, 11, 12] and swarm-based algorithms [13, 14].

This thesis adapts the *artificial bee colony* algorithm's framework. With the artificial bee colony hyperparameter tuner [15], multiple attempts at improving the algorithm have been researched and proposed. One example with proposed Bayesian characteristics, and a survey on other approaches, can be seen in [16]. Three distinct characteristics, modifications of the artificial bee colony algorithm, are identified and assumed vital for the successful tuning of stochastic spiking neural networks. These are improved local exploitation, adapting the search based on earlier evaluated results, and robustness to the result's inherent stochasticity. Of these three targets, only the first is — perhaps implicitly — considered in [16]. Since the second, and especially third, modifications are very unique requirements stemming from the stochastic spiking neural network,

the need for a new Bayesian bee algorithm remains. The algorithm proposed in this thesis is a hybrid scheme between the artificial bee colony algorithm and the *tree-structured Parzen estimator* algorithm [17]. This hybrid algorithm incorporates Bayesian elements rather differently than [16] and is created with the three characteristics in mind. It is believed that the two underlying algorithms cover each other's weaknesses in hyperparameter tuning stochastic neural networks.

The implementation of the stochastic spiking neural network is done in *Intel Lava*. Earlier work in Lava regarding stochastic spiking neural networks and optimization is perhaps best seen in [7], where quadric unconstrained binary optimization is solved using a stochastic spiking neural network. Lava is currently being developed and is best understood by reading their documentation [18].

1.3 Objective

The goal of this thesis work is to implement an SNN that can satisfactorily solve the user allocation problem of dynamic sizes. The implementation is written such that it can be run and tested on SNN custom, called *neuromorphic*, hardware. With a working SNN, we designed and implemented a tuner that finds parameters that optimize the functionality of the SNN. As a final step the SNN and the tuned parameters are thoroughly tested, investigating how the performance and parameters differed for differently sized problems. The thesis work was considered complete when the following had been achieved:

- A working and implemented stochastic spiking neural network solving the user allocation problem.
- A functioning hyperparameter tuner that improves the performance of the SNN.
- A study investigating the tuner, SNN, and the obtained parameters.

1.4 Limitations

Despite working towards a complete presentation of both the stochastic spiking neural network and the hyperparameter tuner, this thesis still contains some limitations in the research.

Starting with the spiking network, while the implementation function properly the thesis makes no claim presenting this as the best network structure or implementation. No proper research is done investigating major structural changes or different underlying dynamics governing the neurons. The network in this thesis is, therefore, *a* network solving edge user allocation but not necessarily the best one. Additionally, access to neuromorphic hardware is limited and while theoretically energy efficient, no test on energy efficiency is done.

Another limitation arose from the large choice of possible hyperparameter tuners. As tuning stochastic spiking neural networks appears underresearched, the multitude of tuning approaches for non-stochastic spiking neural networks is investigated. Here, the choice to build upon the bee colony framework is, while partly motivated, not the only possible approach. The work is therefore limited, only displaying one working example of stochastic network tuning.

Lastly, as is mentioned throughout the thesis, many of the larger simulations are possibly cut short by the ever-increasing simulation times. This probably limited the larger problems' results. While not displaying bad results, with more computational resources and time, perhaps these results would have been better.

2 Theory

The following section presents an overview of the underlying theory upon which this thesis is written. Here, the edge user allocation problem and its mathematical formulation are introduced followed by a brief introduction to neuromorphic hardware, and lastly the main topic of the

thesis — stochastic spiking neural networks and how to design and tune them to solve constraint satisfaction problems.

2.1 Edge User Allocation

Edge computing refers, in telecommunication, to providing applications with computation and storage close to the end users. Advantages of edge solutions include low latency, high bandwidth, device processing, data offload as well as trusted computing and storage [2].

The edge user allocation (EUA) problem refers to the allocation of users to edge servers. A user may be allocated to any server in range that has enough recourses (e.g., memory, and bandwidth) for that user's resource demand. Based on this, the service provider has two objectives. Without overfilling any server, the number of allocated users should be maximized while minimizing the number of active edge servers. Figure 2 illustrates a trivial example of the EUA problem. It is easy to see that if each server can accommodate three users, a solution can be found where server "2" remains inactive.

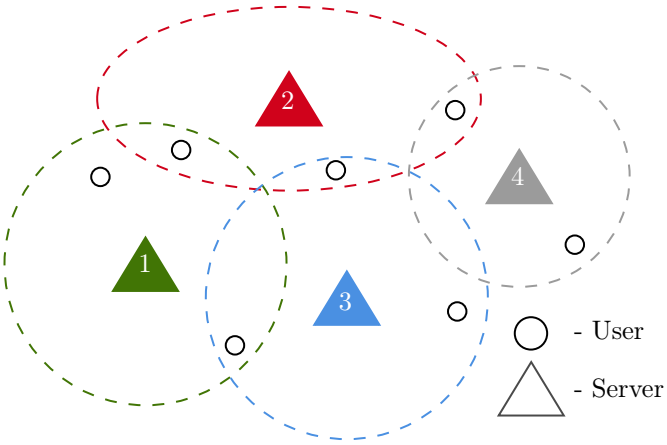


Figure 2: Illustration of a trivial EUA problem.

2.1.1 Mathematical Formulation of EUA

In earlier work, P. Lai et al. formulated the EUA problem as a variable-sized vector bin packing (VSVBP) problem [1]. The VSVBP constitutes a constraint satisfaction problem (CSP). Given an EUA problem with N_u users wishing to be allocated to one of N_s servers, the mathematical formulation contains two optimization objectives and four constraints. As mentioned earlier, the two objectives involve maximizing allocated users while minimizing the number of active edge servers. An edge server is considered active if it has at least one allocated user. These are mathematically expressed in Equations (1a) and (1b).

For the constraints, the *proximity constraint* (2b) dictates that a user may never be allocated to an edge server for which the user is out-of-range. This is formulated such that the distance from a user to its edge server may never surpass the coverage distance of that server. In this thesis, the coverage is assumed circular and a connection is always possible within and never possible outside the coverage. Since each server can only allocate a finite number of users based on the users' resource demand and the server's capacity, the *capacity constraint* (2a) ensures that no active server should allocate users for which their combined demand surpass that server's capacity (in any resource-type). Lastly, the two remaining constraints, *binary value constraint* (2d) and *single allocation constraint* (2c) govern the allocation variables. As a connection either is active or inactive, the variable representing that connection should only take binary values. And since each allocated user should have a unique server connection, the sum of all connection variables

belonging to a user should never exceed one. The full mathematical representation is presented below, Equations (1) and (2) for the objectives and constraints respectively.

$$\max \sum_{j=1}^{N_u} \sum_{i=1}^{N_s} \xi_{ij}, \tag{1a}$$

$$\min \sum_{i=1}^{N_s} \zeta_i, \tag{1b}$$

Subject to constraints,

$$\sum_{j=1}^{N_u} w_j \xi_{ij} \leq C_i \zeta_i, \qquad \forall i \in \{1, \ldots, N_s\} \tag{2a}$$

$$d_{ij} \leq \text{cov}(s_i), \qquad \forall i \in \{1, \ldots, N_s\}; \forall j \in \{1, \ldots, N_u\} \tag{2b}$$

$$\sum_{i}^{N_s} \xi_{ij} \leq 1, \qquad \forall j \in \{1, \ldots, N_u\} \tag{2c}$$

$$\zeta_i, \xi_{ij} \in \{0, 1\}, \qquad \forall i, \forall j \tag{2d}$$

where,

$\zeta_i = 1$ if server s_i is active,
$\xi_{ij} = 1$ if user u_j is allocated to server s_i,
C_i and w_j are the capacity and capacity demand of server s_i and user u_j respectively,
d_{ij} is the distance from user u_j to server s_i,
$\text{cov}(s_i)$ is the coverage of server s_i.

Hence, both server capacity C and user capacity demand w are inferred from the problem and are known prior to solving. Distance variables d and server coverages $\text{cov}(s)$ are also inferred from the problem's geographic setting. The variables ξ and ζ are therefore the decision variables of the problem.

2.2 Spiking Neural Network

Spiking neural networks (SNNs) have been described as the third generation of neural networks [19]. While all artificial neural networks (ANN) are inspired by the human brain, in traditional neural networks the behaviour of the neurons is not biologically feasible, using continuous activation functions. SNNs, therefore, go further, also trying to incorporate biologically-possible neurons [20].

An SNN is structured similarly to ANNs, a network with arbitrary topology consisting of nodes and edges. These nodes and edges are denoted as neurons and synapses, highlighting their biological inspiration. A synapse may be either *excitatory* or *inhibitory*, depending on if it excites or inhibits the post-synaptic neuron. In an SNN the neurons communicate with each other through short pulses — spikes — facilitated by inter-connecting synapses. Generally, a neuron spikes when its membrane potential exceeds its voltage threshold. The membrane potential of a neuron is affected by the incoming spikes, increased with excitatory synapses and decreased with inhibitory. Many neuron models also incorporate a continuous voltage charge or leak, meaning that the membrane voltage change without receiving spikes. To avoid indefinite spiking, after a neuron discharges through a spike the potential is reset to its equilibrium voltage. Information in an SNN is therefore encoded in time, for instance through the rate or timing of the spikes. Unfortunately, due to this,

common ANN training methods are not easily implemented. This is further discussed in Section 2.5.

2.2.1 Neuron Dynamics

As mentioned earlier, SNNs employ spiking neurons inter-connected by excitatory or inhibitory synapses. The underlying dynamics, the inner-working, of the neurons dictate how the neurons behave. There are many different, biologically feasible, neuron dynamics that have been implemented in SNNs [21]. This thesis work opted for simple activation dynamics, reducing the number of hyperparameters within the network. The standard neuron includes a leak parameter τ and a membrane voltage v. The voltage evolves as,

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{1}{\tau}v(t). \tag{3}$$

Equation (3) ensures that the membrane voltage tends towards its equilibrium state, with leak proportional to the membrane potential. The membrane potential of a post-synaptic neuron is affected by the set of pre-synaptic neurons $\mathcal{S}$ according to synaptic weights w. Excitatory and inhibitory connections are implemented as positive and negative weights respectively. Figure 3 shows the parameters affecting the membrane voltage of a neuron. Here, $x(t)$ and $y(t)$ are the outgoing spikes from the pre- and post-synaptic neurons, and w_{ij} is the synaptic weight from neuron j to neuron i.

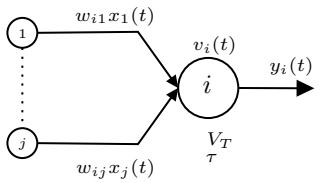


Figure 3: Illustration of parameters affecting the membrane potential of post-synaptic neuron i.

Discretizing the simulation time $t \in [0, T]$, at each timestep t_k the membrane potential is updated according to the pre-synaptic spikes and the voltage leak. This, and discretizing Equation (3), yields the update rules in Equations (4) and (5). These are also visually illustrated in Figure 4. Thus,

$$v_i(t_{k+1}) = \begin{cases} 0 + \sum_{l\in\mathcal{S}} w_{il}\, x_l(t_k), & y_i(t_k) = 1, \\ (1-\tau^{-1})v_i(t_k) + \sum_{l\in\mathcal{S}} w_{il}\, x_l(t_k), & y_i(t_k) = 0. \end{cases} \tag{4}$$

$$y_i(t_{k+1}) = \begin{cases} 0, & v_i(t_{k+1}) < V_T, \\ 1, & v_i(t_{k+1}) \geq V_T. \end{cases} \tag{5}$$

This activation dynamic is deterministic and, as explained in Section 2.4, stochastic behaviour is essential for the functionality of the SNNs solving CSPs. To introduce stochasticity another activation dynamic is introduced, solely defined by a Bernoulli distribution with probability constant p_{IN}. Each timestep these stochastic neuron fire with probability p_{IN}, these spikes increase the membrane voltage of the post-synaptic neuron by the synaptic strength w_{IN}. The mathematically described activation function is thus,

$$\begin{aligned} &y_i(t_{k+1}) = Y, \\ &Y \sim \mathrm{Bern}(p). \end{aligned} \tag{6}$$

Dynamics defined by Equations (4) and (5) are implemented in the principal neuron group. Neurons governed by Equation (6) are only implemented in the input or noise group, only serving to stochastically increase the membrane potential of the principal group. An alternative approach is having the principal neuron spike stochastically, as is done in [4]. It is however believed that these approaches are equivalent.

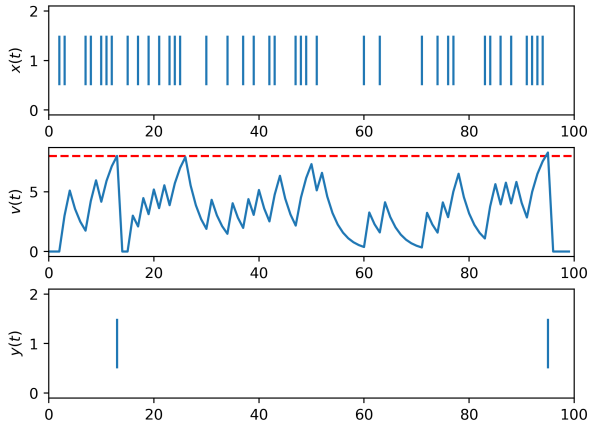


Figure 4: Visual representation of neuron dynamic update rules, Equations (4) and (5). The red line represents the voltage threshold.

2.3 Neuromorphic Hardware

Neuromorphic, or "brain-like", computing seeks to adapt nature's neural architecture to computer architecture. This includes integrated memory-and-computing, fine-grain parallelism, pervasive feedback and recurrence, massive network fan-outs, low precision, and stochastic computations [5]. Communication within the architecture is facilitated by sparse spikes.

The SNN-styled computer architecture, with its asynchronous and event-driven processing, has been shown to massively increase the energy efficiency of computations. [5] presents a comparison between the energy consumption and time to solution when solving the Latin square CSP on a CPU and Intel's neuromorphic hardware *Loihi*. Visible in [5, Figure 7] the neuromorphic hardware is orders of magnitude more energy efficient, with Loihi reporting over a thousand times lower energy than the CPU. When solving CSPs and energy efficiency is important, the potential of neuromorphic hardware is clear.

All code implementation and simulations throughout this thesis project are made using Intel's *Lava* software framework. Lava is an open-source software framework for developing neuro-inspired applications and mapping them to neuromorphic hardware [18]. At the time of writing, Lava supports CPU and Loihi architecture. Unfortunately, as the neuromorphic hardware was not available for testing all simulations and results presented in this thesis are compiled and run on a CPU.

The Lava developers are, as of writing, developing a generic CSP SNN solver [22]. A comparison between a generic and tailored CSP solver is therefore possible in the future.

2.4 Tailoring Stochastic SNNs for Optimization

Jonke et al. [4] propose that a network of stochastic spiking neurons can be utilized in order to solve constrained optimization problems. The methodology is similar to earlier work using

Boltzmann machines to solve CSPs [23].

The state of an SNN consisting of N spiking neurons can at any time be represented by an N-dimensional time-dependent binary vector $\vec{x}(t) = [x_1(t), x_2(t), \ldots, x_N(t)]$. Here $x_k(t)$ represents whether the k:th neuron spiked within the last timestep. With sufficient noise, stochasticity, in the system the network converges to an equilibrium distribution of states $p(\vec{x})$. This distribution is then reformulated as the energy function [4],

$$E(\vec{x}) = -\log\big(p(\vec{x})\big) + C, \tag{7}$$

where C is an arbitrary constant. The network therefore most often exists in low-energy states. An SNN can solve CSPs by encoding the problem variables as neurons and designing the energy function such that low-energy states correspond with viable minimums of the objective function. At equilibrium the network continuously explores these low-energy states, sampling $E(\vec{x})$. Considering a network $\mathcal{N}$ of symmetrically connected neurons, $w_{lk} = w_{kl}$, and no recurrent connections $w_{ll} = 0$. It has been shown [4] that the energy function of network $\mathcal{N}$ is,

$$E_{\mathcal{N}}(\vec{x}) = -\sum_{k=1}^{N} b_k x_k - \frac{1}{2}\sum_{k,l=1}^{N} x_k x_l w_{kl}. \tag{8}$$

In Equation (8), b_k is the bias of neuron k and corresponds with the inherent willingness, or unwillingness, to spike. In an SNN where excitatory noise is supplied by neurons governed by Equation (6) this value is strictly positive. This implies that unless inhibited by other neurons, all neurons fed with stochastic noise eventually spike. This proposed network does however only allow for second-order dependencies between problem variables, see Equation (8), and for many CSPs this is insufficient. Jonke et al. [4] proposes that the dependencies between the problem variables are increased by introducing a set of auxiliary circuits $\mathcal{A}$ to the principal network $\mathcal{N}$. These auxiliary circuits, not constrained by symmetric and non-recurrent connections, serve to increase variable dependencies and modulate the energy function of the full network. To repeat, the principal network is the main building block of the network, adhering to the aforementioned constraints and with a known energy function. The auxiliary networks are added to the principal network and add dependencies between problem variables and help further shape the energy function. While the energy function of this combined network is unknown, Jonke et al. [4, Theorem 1] proves that the energy contributions U of the auxiliary networks are linear and that the total energy function can be written as a linear combination

$$E_{\mathcal{N},\mathcal{A}}(\vec{x}) = E_{\mathcal{N}}(\vec{x}) + \sum_{i\in\mathcal{A}} U_i(\vec{x}). \tag{9}$$

By intelligently constructing motifs of auxiliary circuits, CSP constraints, and dependencies can be modulated. The SNN then continuously samples states from the energy function and, given enough time, finds a solution to the CSP [4]. The methodology has been implemented for the traveling salesman problem [4], Boolean satisfaction problem [4], sudoku problem [6] and map colouring problem [6].

2.5 Hyperparamter Tuning of Stochastic SNNs

When designing a stochastic SNN for solving CSPs, many of the network parameters get inferred from the problem description. But often, and especially with the EUA problem, many parameters still require careful tuning for the SNN to achieve optimal performance. In earlier implementations [4, 6] of CSP-solving stochastic SNNs, tuning is bypassed by manually finding parameters that adequately shape the energy function. With valid parameters, the network will, due to the continuous sampling of the energy function, find the solution given enough time. This works fine when only one problem instance is considered, but not when solving a whole class of problems with

limited runtime. Additionally, a good set of parameters for one EUA problem is not necessarily good for another. It is therefore essential to have a hyperparameter tuner that can optimize the network by finding parameters that give valid and good solutions within the allotted time.

Compared with regular neural networks, the spiking and event-based nature of SNNs does not allow for regular training methods such as backpropagation — as no classical derivative can be calculated. And while there exist SNN-alterations to backpropagation, see for example *SpikeProp* [24], they are more often used for feed-forward networks. To obtain a starting point for the algorithm's development, the tuning of non-stochastic SNNs is considered. These algorithms are used as the foundational framework for an extension to stochastic SNNs.

2.5.1 Tuning on Non-Stochastic SNNs

As no earlier work was found regarding the tuning of stochastic SNNs solving CSPs, the focus is turned to methods of tuning the non-stochastic "regular" SNNs. Note that optimizing network weights and neuron parameters is typically referred to as *training* rather than tuning in the case of regular SNNs. Here, we refer to this as tuning for cohesion. As these experience similar issues, disregarding stochastic performance, it is believed that the research on tuning regular SNNs can be used as a starting point when considering stochastic SNNs.

One biologically inspired method to tune SNNs is called spike-timing-dependent plasticity (STDP). STDP is an online synaptic weight learning rule based on Hebbian learning. It is a biologically feasible learning method and it is widely believed that synaptic plasticity underlies learning and information storage in the brain [25]. STDP updates synaptic weights according to tight temporal correlations between spikes of pre- and post-synaptic neurons. When a neuron spikes, it opens a learning window $W(x)$ for its pre- and post-synaptic neurons. The synaptic weight w_{ji} is then updated by the spike time difference between the pre- and the post-synaptic neuron, t_i and t_j respectively. It has, in combination with reinforcement learning, been used to tune SNNs doing motor control [9].

Another approach involves applying metaheuristics, nature-inspired algorithms. Genetic evolutionary algorithms have been shown to tune SNNs for tasks such as classification and control [10, 11, 12]. Swarm-based algorithms have also previously been used to tune SNNs. Examples include particle swarm optimization (PSO), cuckoo search, ant colony optimization, and artificial bee colony [13, 14].

2.5.2 Choosing a Hyperparamter Tuner

Looking at other implementations, a key difference is that the EUA network is purposely designed. Compared to a dense network, each synaptic connection is a vital and understood part of the network's function. As such, the tuner is restricted to certain elements. Evolutionary algorithms that create, heavily change, or break synapses will therefore most likely not work. Similarly, STDP will probably struggle as there exist multiple positive feedback loops, leading to epileptic behaviour as synaptic strengths grow indefinitely. The metaheuristics swarm intelligence algorithms are chosen as they constitute simple, general, and explorational algorithms. These algorithms are also established, available, and researched. The artificial bee colony algorithm is chosen as the tuner's framework. It has previously been implemented to tune SNNs [13] and has desirable qualities presented in a survey on nature-inspired algorithms for neural networks [26].

2.5.3 Artificial Bee Colony Algorithm

The artificial bee colony (ABC) algorithm is a natured-inspired swarm-based method meant to mimic the strategy bees use to find food. It was originally proposed by Karaboga in 2005 [15] and has been applied when tuning spiking neural networks for classification tasks, see [13]. The ABC algorithm has three types of workers: employed, onlookers, and scouts. The employed bees are all tied to a food source, parameters, and recruit onlookers to exploit that food source. The better a

food source appears the more onlookers are recruited. The employed bee applies a greedy selection process when trying a new food source and internally keeps track of its currently best parameters. To avoid stagnation, each employed bee has an internal "abandonment counter", and after a few iterations of not finding an improvement that food source is considered depleted. The employed bee then instead becomes a scout bee and randomly chooses new parameters in the parameter space, resetting its abandonment counter.

Initialization of the ABC algorithm consists of randomly placing n_b employer bees. Afterward follows an arbitrary amount of iterations, each consisting of the employer, onlooker, and scout phase.

The employer phase involves each employer bee trying new parameters, where those parameters $\vec{\rho}_i'$ are generated as follows,

$$\vec{\rho}_i' = \vec{\rho}_i + r(\rho_{jd} - \rho_{id}), \tag{10}$$

here ρ_{jd} is the d:th element of employer bee j's parameters, r is a uniformly sampled value and,

$$j \in \{1, \ldots, N\} \backslash i,$$
$$d \in \{1, \ldots, \dim(\vec{\rho})\}.$$

Greedy selection is then applied and if the new parameters are superior, the abandonment counter is reset. A fitness score is assigned to each employed bee's parameters. For a minimization problem with objective f, the fitness function is

$$\text{fit}(\vec{\rho}) = \begin{cases} \frac{1}{1+f(\vec{\rho})} & \text{if } f(\vec{\rho}) \geq 0, \\ 1 + \text{abs}(f(\vec{\rho})) & \text{if } f(\vec{\rho}) < 0. \end{cases} \tag{11}$$

To attract onlookers each employed bee gets a probability based on the hive's total fitness,

$$p_m = \frac{\text{fit}(\vec{\rho}_m)}{\sum_{i=1}^{n_b} \text{fit}(\vec{\rho}_i)}. \tag{12}$$

Each onlooker bee is then assigned to a food source based on the computed probabilities. The onlooker bees search, evaluate, and apply greedy selection similarly to the employer phase, Equation (10). This structure ensures good parameters are prioritized for exploitation.

The last step of each iteration is to check whether any employed bee's abandonment counter is over a set threshold n_a. If that is the case, the employer bee's parameters are randomly reassigned, and the abandonment counter is reset.

2.5.4 Tree-structured Parzen estimator

Tree-structured Parzen estimator (TPE) is a Bayesian optimization algorithm that uses a tree-structured model to approximate the likelihood distribution [17]. This is achieved by continuously saving the result from all tested parameters $\vec{\rho}$ and their corresponding score P. After an initial phase of random sampling, it selects the next set of parameters by fitting two probability density functions (PDFs) $\alpha(\vec{\rho})$ and $\beta(\vec{\rho})$ to the "good" and "bad" parameters of the earlier evaluations. This good-bad-split is done by setting a predetermined divider quantile γ, obtaining a subsequent divider score $\hat{P}$. By choosing γ and using all earlier evaluations, $\hat{P}$ is obtained as the value satisfying $p(P < \hat{P}) = \gamma$. For the single-objective case, the tree-structured split is trivial. For multi-objective optimization the split is more complicated, as seen in [27].

$$p(\vec{\rho}|P) = \begin{cases} \alpha(\vec{\rho}) & \text{if } P < \hat{P}, \\ \beta(\vec{\rho}) & \text{if } P \geq \hat{P}. \end{cases} \tag{13}$$

The two PDFs $\alpha(\vec{\rho})$ and $\beta(\vec{\rho})$ are approximated using the Parzen kernel density estimator (KDE), placing kernels at the parameters for each respective PDF. The resulting PDFs' values at any point in the parameter space are obtained as the sum of all kernels. These distributions are adaptive, meaning each iteration, $\alpha(\vec{\rho})$ and $\beta(\vec{\rho})$ are updated.

To obtain the next parameters for evaluation, candidates are sampled from the good distribution $\alpha(\vec{\rho})$. From this set of candidates, the parameters maximizing the *expected improvement* $\mathrm{EI}_{\hat{P}}(\vec{\rho})$ is chosen. Using the the utility function $\max(\hat{P} - P, 0)$ and the likelihood approximation from Equation (13), the expected improvement is [17],

$$\begin{aligned}
\mathrm{EI}_{\hat{P}}(\vec{\rho}) &= \int_{-\infty}^{\infty} \max(\hat{P} - P, 0)\, p(P|\vec{\rho})\, \mathrm{d}P \\
&= \int_{-\infty}^{\hat{P}} (\hat{P} - P)\, \frac{p(\vec{\rho}|P)p(P)}{p(\vec{\rho})}\, \mathrm{d}P \\
&= \alpha(\vec{\rho}) \int_{-\infty}^{\hat{P}} (\hat{P} - P) \frac{p(P)}{\gamma\alpha(\vec{\rho}) + (1-\gamma)\beta(\vec{\rho})}\, \mathrm{d}P \\
&= \frac{\alpha(\vec{\rho})}{\gamma\alpha(\vec{\rho}) + (1-\gamma)\beta(\vec{\rho})} \int_{-\infty}^{\hat{P}} (\hat{P}p(P) - Pp(P))\, \mathrm{d}P \\
&= \frac{\alpha(\vec{\rho})}{\gamma\alpha(\vec{\rho}) + (1-\gamma)\beta(\vec{\rho})} \hat{P}p(P < \hat{P}) \int_{-\infty}^{\hat{P}} -Pp(P)\, \mathrm{d}P \\
&= \frac{\alpha(\vec{\rho})\hat{P}\gamma - \alpha(\vec{\rho}) \int_{-\infty}^{\hat{P}} Pp(P)\, \mathrm{d}P}{\gamma\alpha(\vec{\rho}) + (1-\gamma)\beta(\vec{\rho})} \\
&\propto \left(\gamma + \frac{\beta(\vec{\rho})}{\alpha(\vec{\rho})}(1-\gamma)\right)^{-1}.
\end{aligned} \tag{14}$$

The expected improvement is, as seen above, maximized when minimizing the quotient $\beta(\vec{\rho})/\alpha(\vec{\rho})$. Hence, this quotient is used as a surrogate, working as an approximation of the real function. When the underlying optimization target is computationally expensive the surrogate can reduce the number of bad evaluations.

3 Methodology

The methodology section presents how the theory in the earlier section is applied to solve the edge user allocation problem. It consists of the design for the stochastic SNN and a proposal for a new network tuner based on the ABC algorithm. Additionally, to validate the results of the SNN, the EUA problem is reformulated as an integer linear programming and solved with an established solver.

3.1 Designing an SNN for EUA

To design an appropriate SNN to solve the edge user allocation problem from Section 2.1.1 a principal network of neurons, governed by Equation (4), is placed in a grid structure. The size of the grid is $N_u \times N_s$ where N_u and N_s are the number of users and servers respectively. Each row corresponds with a user and each column a server. Thus, if the neuron in column i and row j spikes it is interpreted as user j connecting to server i.

Each of the neurons in the principal network receives excitatory spikes with strength w_{IN} from the input neurons, described by Equation (6), driving the SNN's search. To guide the network's state

towards solutions of the EUA, objective functions and constraints are encoded through motifs and network structure, shaping the energy function.

3.1.1 Principal Network — Objective Functions

The first objective, to maximize allocations, is ensured by the input neurons. With sufficient noise, the energy function corresponding to unassigned users is very high. The second objective is the minimization of active servers. Each column, corresponding to the same server is connected by symmetric excitatory synapses to a low-threshold neuron, here called *utilization* neuron, with synaptic strength w_{UTIL}. This excitatory connection reduces the energy function of the states where multiple users are allocated to the same server.

The behaviour of the principal network $\mathcal{N}$, just described and visualized in Figure 5, lacks inhibitory synapses and will result in epileptic behaviour, that is, all neurons will constantly fire. As each column is independent and unaffected by the other columns, the symmetric excitatory connections with the utilization neurons result in every column trying to have all its neurons firing. The energy minimum is when all neurons are active, which is obtained by considering the energy function of the principal network, Equation (8), in the case of only excitatory synapses $w > 0$.

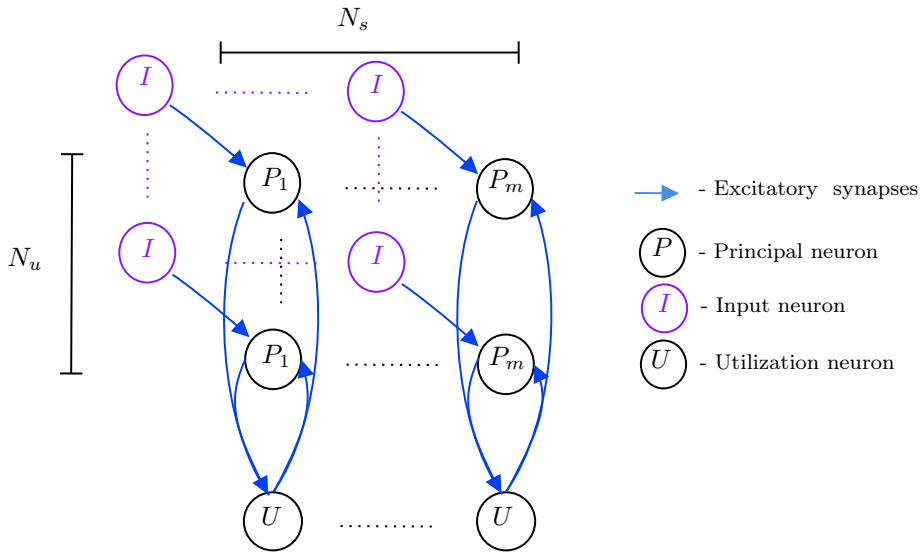


Figure 5: Principal network; Principal, utilization, and input neurons.

3.1.2 Auxillary Network — Constraints

To change the behaviour of the network, and enforce constraints, auxiliary network motifs are designed to shape the energy function. The binary value constraint (2d) is inherently enforced by the neuron's binary states — active and inactive. Secondly, proximity constraint (2b) is enforced by greatly increasing the voltage threshold of the neurons corresponding with out-of-range connections. These neurons fire incredibly seldom, corresponding with very high energy states.

Both the capacity constraint and the single-allocation constraint, Equations (2a) and (2c), involve limiting the number of active neurons in the rows and columns of the aforementioned grid structure. In each row, corresponding with the same user, only one neuron should be active. This is accomplished with auxiliary winner-take-all (WTA) motifs. In the WTA, the neurons are connected to an auxiliary *control* neuron with a strong excitatory connection, such that it reaches its voltage threshold following a single spike. It is then connected back with inhibitory connections, synaptic strength w_{WTA}. When a neuron within the WTA spikes the control neuron returns inhibitory signals so no further spike occurs. Disregarding that the noise encourages the neurons

to spike (bias), the energy function of the auxiliary WTA circuit containing a subset of principal neurons $\mathcal{K} \subset \{1, \ldots, N\}$ is approximated as [28],

$$E_{WTA,[\mathcal{K}]} \approx \begin{cases} 0, & \sum_{k\in\mathcal{K}} x_k \leq 1, \\ w_{WTA}(\sum_{k\in\mathcal{K}} x_k), & \sum_{k\in\mathcal{K}} x_k > 1. \end{cases} \tag{15}$$

Again, the w_{WTA} is the synaptic weight from the control neurons to the principal neurons, x_k represents whether neuron k is spiking, and $\mathcal{K}$ is the set of principal neurons belonging to the WTA.

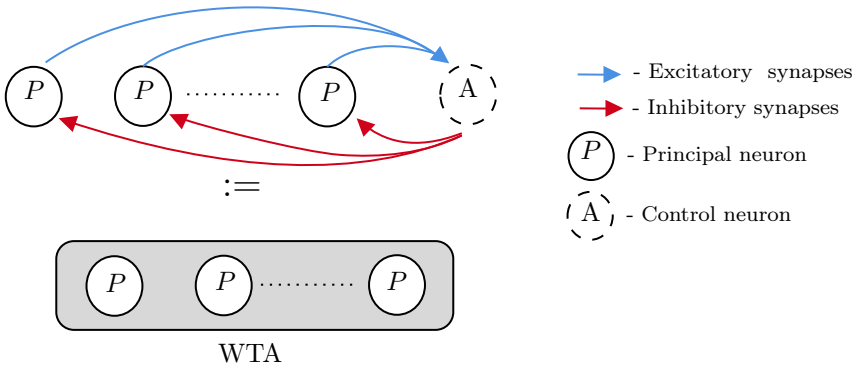


Figure 6: WTA motif.

Likewise with the capacity constraint, in each column, the energy function is raised for states where any of the servers' resource capacities are broken. The synaptic structure is similar to the WTA but the capacity-control neurons' dynamic differs, as they should not imminently spike upon excitation. It should also not spike if excited multiple times by the same neuron. It needs to distinguish the input coming from different neurons and only spike when enough neurons are active, i.e., the capacity constraint is violated. Implemented, the capacity neuron has a voltage vector that is booleanized with a low voltage threshold. When the sum of the dot product between this booleanized vector and the vector, or matrix, containing information regarding the users' resource demand exceed capacity the neuron spikes, inhibiting all neuron corresponding with that server. This setup ensures that each spike from the principal neuron is translated as a user establishing a connection with a server, a change to the current state. Each user starts in an unassigned state but is quickly picked up by any server, and changes server each time a neuron in that user's row spikes.

For the capacity neurons, this raises a problem as the neuron only receives information when a user is assigned to that server but not when a user has been reassigned to another server. As such, the capacity neurons must communicate their booleanized vectors to all other capacity neurons whenever they detect a change, that is when they receive a spike from a neuron in the principal network. When a capacity neuron receives information that a user has been assigned to another server it discharges the voltage belonging to that user. This structure removes the capacity neurons' need for the leak parameter τ, setting τ^{-1} to zero. And since boolean values are easily interpreted as spikes, the communication remains facilitated by spikes. For the framework in which the implementation is made, vector-shaped ports and voltages are possible. If this is not possible, then all vector-based logic can instead be translated to multiple single-valued neurons and synapses.

Disregarding the additional complexity to ensure that the capacity neurons function properly, this additional motif worked like a WTA with multiple winners. Calling this motif C winners-take-all (CWTA), this auxiliary network's contribution to the energy function is calculated by adapting

Equation (15). With the subset $\mathcal{K}$ of principal neurons belonging to the CWTA, disregarding the principal neurons' inherent willingness to spike, their energy contribution becomes,

$$E_{CWTA,\,[\mathcal{K}]}(\vec{x}) \approx \begin{cases} 0, & \sum_{k\in\mathcal{K}} x_k \leq C, \\ w_{CWTA}(\sum_{k\in\mathcal{K}} x_k), & \sum_{k\in\mathcal{K}} x_k > C. \end{cases} \tag{16}$$

Here, w_{CWTA} is the synaptic strength of the returning inhibitory spikes from the capacity neuron to the principal neurons.

3.1.3 Approach for Monitoring Network States and EUA Solutions

As the energy function represents a distribution, the network does not converge to a single state representing the solution but constantly explore states sampled from the energy function. And while the state representing the optimal solution might correlate to the lowest point in the energy function, just the number of low-energies states can make sampling the lowest energy unlikely. For this reason, a way to detect good solutions is required, introducing a monitor.

The naive approach for a monitoring system is having a continuous reading of the network's state. After the simulation the naive monitor goes through and calculates the score of each state, returning the best state. This is both costly and represents a bottleneck in the computation. The ideal approach is having an internal logic check for optimal solutions, only once reading the state or locking the network with inhibitory signals when the optimum is reached. In earlier implementations this has been achieved with the 3-Boolean satisfiability problem [4] and the Latin square problem [5]. These two problems are however inherently different since a state fulfilling the constraints is automatically an optimal solution. For problems like traveling salesman [4] and, most relevantly EUA, the optimum is not known in advance. And therefore it is not, to the writer's knowledge, possible to formulate this kind of logic check. Instead, some kind of monitoring is required. But to avoid the naive approach, the monitor is implemented to reduce overhead computations. This is accomplished by exploiting that the aforementioned booleanized vectors in the capacity neurons already contain the information required to calculate the current score, even without knowing the current solution. The number of users is equal to the sum of all Boolean vectors, and the number of active servers equals the number of non-zero Boolean vectors, both incredibly cheap operations. Computing these allows the monitor to know whether the current state is a better solution without monitoring the entire network. Then, only when the current solution represents an improvement does the monitor read the state of the principal network and save it in memory. Also, any capacity neuron assigned too many users (violated capacity constraint) inhibited the monitor, ensuring no invalid states are saved in memory. This likely gives the same results as continuous monitoring of the state, but at a reduction in computations.

3.1.4 Full Network solving EUA

By following the presented methodology, a stochastic SNN is created solving instances of the EUA problem. The SNN continuously samples states from the energy function distribution and the monitor detects when the current state is an improvement to the one in memory. When the simulation time is over, the network returns the current solution in memory. An illustration of the final network is seen in Figure 7.

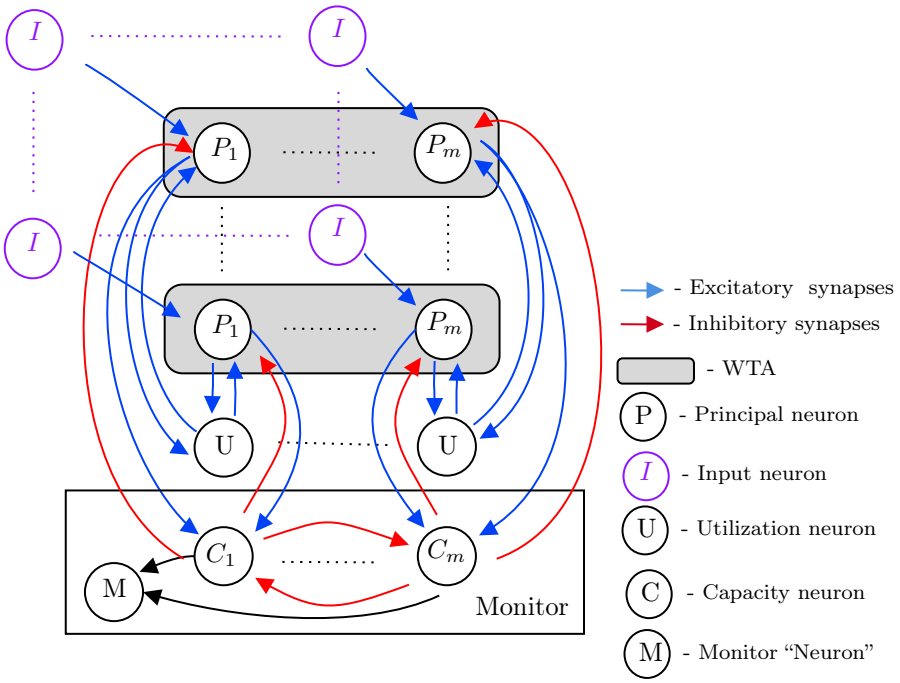


Figure 7: Figure of full network solving the edge user allocation problem.

An approximation of the energy function for the full network gives an idea of how to balance the weights to obtain the desired distribution. By defining Γ and Λ as the set of neuron sets belonging to the auxiliary motifs, WTA and CWTA, and assuming that the motifs constitute the biggest contributor to the auxiliary energy function, the total energy function is approximated with Equations (8), (9), (15), and (16),

$$E_{\mathcal{N},\mathcal{A}}(\vec{x}) \approx -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}x_i x_j + \sum_{\mathcal{K}\in\Lambda} E_{WTA,[\mathcal{K}]} + \sum_{\mathcal{K}\in\Gamma} E_{CWTA,[\mathcal{K}]}. \tag{17}$$

For this thesis work, the hyperparameters are tuned in groups meaning that, for example, all control neurons' inhibitory strengths w_{WTA} are numerically equivalent. This is a large generalization but serves to keep the number of hyperparameters at a reasonable range. Without this generalization, the number of hyperparameters for larger SNNs becomes huge. Below, in Table 1, are all changeable hyperparameters from the implementation presented in this section.

Table 1: List of all SNN hyperparameters.

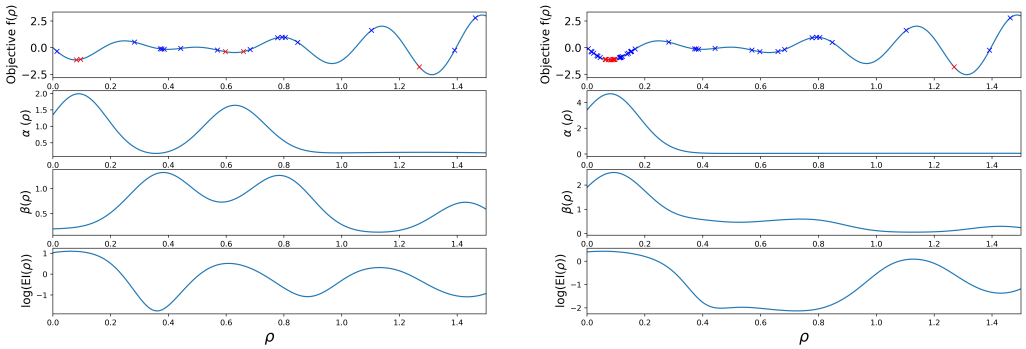
Variable	Description
V_P	Principal neuron voltage threshold
τ_P	Principal neuron leak parameter
w_{IN}	Input neuron excitatory strength
p_{IN}	Input neuron spike probability
w_{WTA}	Control neuron inhibition strength
w_{UTIL}	Utilization neuron excitation strength
w_{CWTA}	Capacity neuron inhibition strength

3.2 Proposed Improvement to ABC

The ABC algorithm, while being simple, intuitive, and proven to solve a multitude of problems has difficulties with the exploitation of solutions. Also, important for this thesis, the EUA problem solved with the SNN represents a computationally expensive and timely optimization problem. The ABC algorithm does not use all earlier evaluations to guide its search, as each employed bee only stores its best parameters in memory. For the SNN, a lot of information is contained within the failed evaluations. Some regions of the parameter space completely break the SNN by shifting the energy function too much, perhaps placing the energy function minima in states corresponding to constraint-breaking solutions. A tuning algorithm preferably avoids excessive evaluation of these regions. Additionally, the performance of the SNN is stochastic and ABC has no real way to deal with or interpreted stochasticity. These observations constitute the three qualities believed vital for the tuning algorithm's success. The proposed algorithm is designed to trade some exploration for local exploitation, better search based on earlier evaluations, and robustness to the evaluation's stochastically over- or underperformance. Different ways of improving the exploitation of the ABC algorithm have been implemented earlier, modifying the bees' flight, fitness function, recruitment strategy, etc. These alternatives have been shown to improve the algorithm for certain problems [16]. However, there is no optimal optimizer and, more importantly, the alteration in [16] does not fulfill the three aforementioned criteria. Working with a computationally costly and highly stochastic system differs from deterministic functions evaluated under a millisecond. It is therefore relevant to develop another Bayesian modification to the ABC algorithm, specifically for the hyperparameter tuning of CSP-solving stochastic SNN. The proposed modification to the ABC algorithm changes the behaviour of the onlooker bees, incorporating aspects from the tree-structured Parzen estimator, presented in Section 2.5.4.

3.2.1 Sortcommings of TPE

The TPE algorithm does greatly increase exploitation and uses a surrogate function built on all earlier evaluations. However, a shortcoming of TPE is getting stuck in a local minimum, as the true minimum might have been missed by the initial iterations or if a large part of points building $\alpha(\vec{\rho})$ are clustered around the false minima. This is more apparent when the minima are far apart, and an example of this is seen in Figure 8. TPE is, contrary to ABC, very exploitative but less explorational. With only TPE there is too little exploration. TPE exploits promising regions but tends to get stuck in local minimums, not exploring further. This tendency might be even more apparent when considering stochastic results, especially early lucky results. Using a single large surrogate function, getting stuck in a local minimum stops the searching process of the entire algorithm. The combination with ABC is believed to negate these problems, as the bees explore and act semi-individually. A single bee "stuck" in a local minimum can escape by other bees finding equivalent, or better, regions outside the local minimum. The bees are also forced to abandon a solution after enough iterations without improvement. This also ensures continuous exploration, even after finding the global minimum.



(a) Initial distributions *(b) Distributions after ∼50 iterations*

Figure 8: Example of how the TPE algorithm might get stuck in local minimums

3.2.2 Hybrid Scheme, ABC-BA

The proposed algorithm is fundamentally a hybrid scheme between the ABC and TPE algorithms. In standard ABC, the onlooker bees search the parameter space with the same strategy as the employer bees, Equation (10). The exploitation arises from more onlookers being recruited to better parameters. The modified algorithm, hereafter called *ABC-BA*, replaces the onlookers with accountant bees. These accountant bees keep track of all earlier evaluations, trying to find good parameters locally to the employer bee they are assigned to. This is achieved by continuously adding the result from all tested parameters $\vec{\rho}$ and corresponding computed score P to a dataset $\mathcal{D} = \{\{\vec{\rho}, P\}_i\}$. The accountants, when assigned to an employer bee, splits the larger dataset into a smaller dataset $\mathcal{D}_c$ containing a fraction of the parameters closest to the employer bee's current parameters $\vec{\rho}$. The size of the smaller dataset can be varied, but all datasets should be created such that no parameters outside the smaller set $\mathcal{D}_c$ are closer (Euclidean distance) than the points within the set. Thus,

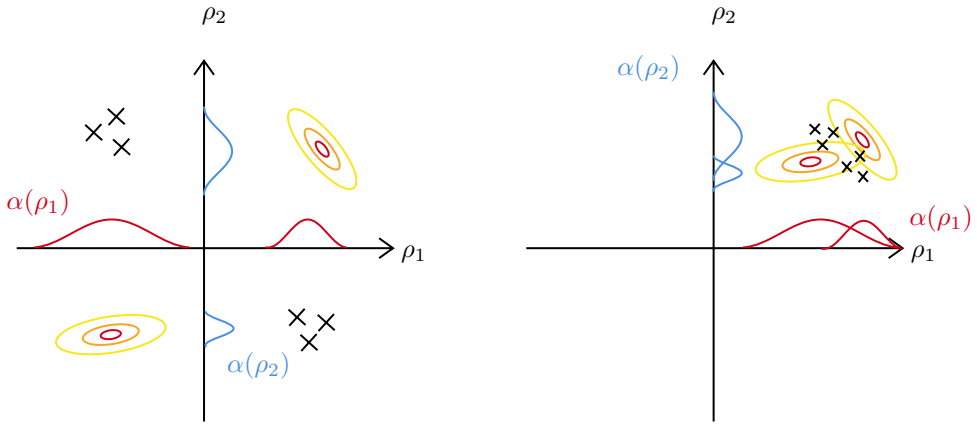
$$\mathcal{D}_c \subset \mathcal{D}, \quad \text{s.t} \mid\mid \vec{\rho} - \vec{\rho}_i \mid\mid \leq \mid\mid \vec{\rho} - \vec{\rho}_j \mid\mid \ \ \forall \vec{\rho}_i \in \mathcal{D}_c, \vec{\rho}_j \in \mathcal{D} \setminus \mathcal{D}_c. \tag{18}$$

With these smaller datasets, unique for each employer bee, the algorithm works with adaptive local surrogates. These surrogates are built similarly to TPE, Equation 13, but as the dataset is local the accountant bees will likely select the next parameters from their local region. Then, as the employer bees move, the "local" region subsequently changes. When an employer bee encounters a region previously, or currently, local to another employer bee the earlier evaluations of that bee are used to form the local surrogate, increasing its accuracy. The bees therefore implicitly share information as all evaluations are added to the dataset $\mathcal{D}$.

To build the local surrogates the, now local, PDFs $\alpha(\vec{\rho})$ and $\beta(\vec{\rho})$ are created with the Parzen estimator KDE. These distributions are fitted to the top γ and bottom $(1-\gamma)$ quantiles of $\mathcal{D}_c$ respectively. One can interpret this as the local dataset being split further into good and bad datasets $\mathcal{D}_{c,\alpha}$ and $\mathcal{D}_{c,\beta}$. For the Parzen estimator KDE, Gaussian kernels are used. These kernels are split into multiple one-dimensional kernels, with a standard deviation in each dimension equal to the distance to the closest point. That is, in ABC-BA the closest points are calculated separately for each dimension. The accountant bees use these distributions to sample candidates from the $\alpha(\vec{\rho})$ PDF, similar to TPE. The accountant bee chooses, for each dimension, the sampled value maximizing the expected improvement, Equation (14). This effectively reduces a d-dimensional problem to d one-dimension problems. For each one-dimension kernel, upper and lower bounds are enforced on the standard deviations, keeping the kernels' shape reasonable. The accountant bees thus select the next parameters as,

$$\rho_i = \arg\max_{\rho_i} \mathrm{EI}_{\hat{P}}(\rho_i) \qquad \forall i \in \{1, \ldots, \dim(\vec{\rho})\} \tag{19}$$

This splitting of the dimensions appears counterintuitive, losing high-dimensional information. A rightful concern is wrongfully inferring information of good, or bad, parameters on unseen parameters. This concern is visualized in Figure 9a, crosses representing regions untruthfully being considered good. However, due to the uniqueness of using local surrogates, these kernels' distributions are necessarily close to each other. Then, a possibly more accurate representation is seen in Figure 9b. Now the crosses bridge the small gap between the two kernels, encouraging the algorithm to evaluate points between them. This increases local exploration, as the two kernels might correspond to the same minimum or have the global minimum between them.



(a) Global surrogate — distant kernels

(b) Local surrogate — close kernels

Figure 9: Visualization of how multiple one-dimensional distributions result in information incorrectly inferred on unseen points. $\alpha(\vec{\rho})$ is the good distribution and its one-dimensional parts are seen along each axis. Crosses show regions where information is inferred and full-dimensional kernels are shown as contour plots.

Moving on, after evaluating the chosen parameters on the SNN, the accountant phase is concluded by adding the computed results to the dataset $\mathcal{D}$. Since the Parzen estimator requires some already calculated values the accountant phase only starts after a few iterations I_{nb}. The number of samples drawn from $\alpha(\vec{\rho})$ each iteration is a hyperparameter of the accountant bee. If the sample size is too large, the slight randomness from sampling $\alpha(\vec{\rho})$ is lost. Too small and the local surrogate might return poor parameters. The accountant bee steers the employer bees away from trying bad parameters and massively increases exploitation, especially when the dataset $\mathcal{D}$ has gotten large. This behaviour is desirable as other alterations of ABC implemented an explicit counter that increases exploitation at later iterations [16].

Lastly, even with identical parameters the SNN's performance can vary wildly. To get an idea of the parameters' consistency, all runs are made multiple times. Even so, since the EUA objective function is discrete, multiple parameters might obtain the same score. At the very end when the algorithm should return the best set of parameters, it chooses the parameters with the highest score from the region with the densest collection of top-performing parameters. As a small step in the parameter space is hypothesized not to affect the SNN's performance, the runs of neighbouring parameters grant information about the parameters' consistency. A small region with intermixed good and bad parameters is then interpreted as a region of inconsistent but lucky parameters. A region with a dense collection of good parameters then suggests consistent parameters and is seen favourable when deciding between multiple parameters with the same reported performance. Pseudo-code of the full ABC-BA algorithm is shown in Algorithm 1.

Algorithm 1 Pseudo-code for the ABC-BA algorithm

Initialize hyperparameters, seen and explained in Table 2
Initialize parameter space, number of iterations $Iter$, and objective function $f(\vec{\rho})$.
* Initialization phase
Employer bees randomly choose a set of parameters and compute score $\frac{1}{I_c}\sum_0^{I_c} f(\vec{\rho})$
Calculate fitness of each employer bee, Equation 11
Add evaluated parameters and computed score to global dataset $\mathcal{D}$
for $i = 0 : Iter$ **do**
    *Employer Phase
    for $j = 0 : n_b$ **do**
        Generate new parameters $\vec{\rho}$ with Equation (10) for bee j and compute score $\frac{1}{I_c}\sum_0^{I_c} f(\vec{\rho})$
        Add evaluated parameter and computed score to global dataset $\mathcal{D}$
        if New parameters are better than current **then**
            Apply greedy selection for employer bee j
            Update fitness of employer bee j, Equation (11)
            Reset abandonment counter of employer bee j
        else
            Increase abandonment counter of employer bee j
        end if
    end for
    Calculate attraction probabilities p, Equation (12)
    if $i \geq I_{nb}$ **then**
        *Accountant Phase
        for $j = 0 : N$ **do**
            Assign accountant bee j to employer bee k based on attraction probability p
            Create $\mathcal{D}_c$ from $\frac{d_{\%}}{n_b}$ of points closest to parameters $\vec{\rho}_k$ (Euclidean distance)
            Split local dataset $\mathcal{D}_c$ into datasets $\mathcal{D}_{c,\alpha}$ and $\mathcal{D}_{c,\beta}$ based on divider γ, Equation (13)
            for Datapoint in local datasets $\mathcal{D}_{c,\alpha}$ and $\mathcal{D}_{c,\beta}$ **do**
                Calculate std $\vec{\sigma}$ as the distance to the closest point in each dimension
                Enforce upper and lower bounds on $\vec{\sigma}$.
                Place kernel at datapoint in $\alpha(\vec{\rho})$ or $\beta(\vec{\rho})$ respectively
            end for
            Sample $\{\vec{\rho}\}$ from $\alpha(\vec{\rho})$
            for $d = 1 : \dim(\vec{\rho})$ **do**
                Calculated $EI(\vec{\rho})$ of d:th parameter in $\vec{\rho}$ for each sample in $\{\vec{\rho}\}$, Equation (14)
            end for
            Obtain the new set of parameters $\vec{\rho}$ as all parameters with highest $EI(\vec{\rho})$
            Compute score $\frac{1}{I_c}\sum_0^{I_c} f(\vec{\rho})$ of new set of parameters
            Add evaluated parameters and computed score to global dataset $\mathcal{D}$
            if New solution is better than current **then**
                Apply greedy selection for employer bee k
                Reset abandonment counter of employed bee k
            else
                Increase abandonment counter of employer bee k
            end if
        end for
    end if
    Report currently best parameters, score, and fitness
    *Scout Phase
    if Any abandonment counter $> n_a$ **then**
        Re-initialize and reset abandonment counter of that employer bee
    end if
end for
Return parameters with the best score in the region densest with good results

3.3 Obtaining a Reference Solution for EUA

To better understand the performance of the SNN, the edge user allocation problem is also solved with a conventional solver. Since the problem displays linear characteristics, and since there exist good solvers for linear optimization, the problem is rewritten and solved with integer linear programming (ILP).

An integer linear programming problem is defined as,

$$\min f^\intercal z. \tag{20}$$

Where the solution vector z is subject to constraints,

$$\begin{aligned} Az &\leq b, \\ A_{eq} &= b_{eq}, \\ lb &\leq z \leq ub, \\ z_i &\in \mathbb{Z}. \end{aligned} \tag{21}$$

Here f, b_{eq}, b, lb and ub are vectors and A and A_{eq} are matrices.

Following the definitions given in Section 2.1.1, the solution vector z is built by stacking the binary user allocation and server usage variables, ξ and ζ. The ordering of ξ_{ij} is user-wise, starting with all variables corresponding with the first user. This dimension of this z-vector is thus $N_u N_s + N_s$.

$$\vec{z} = \begin{bmatrix} \vec{\xi} \\ \vec{\zeta} \end{bmatrix} = \begin{bmatrix} \xi_{11}, & \dots & \xi_{1N_s}, & \dots & \xi_{N_u N_s}, & \zeta_1, & \dots \end{bmatrix}^\intercal \in \mathbb{R}^{N_u N_s + N_s \times 1}. \tag{22}$$

The cost vector f encodes both optimization targets, Equation (1), as the first $N_s N_u$ elements reduce cost and the latter increases. It is therefore simply a vector containing differently signed ones. Namely,

$$f = [-1, -1, \dots, 1, 1, \dots]^\intercal \in \mathbb{R}^{N_u N_s + N_s \times 1}. \tag{23}$$

The inequality constraint encodes the capacity constraint, as well as binds the server usage variables ζ to the user allocation variables — as they are entirely dependent on each other. This is accomplished by a $2N_s \times N_s N_u + N_s$ matrix where the first rows ensure that the sum of variables representing connections to the same server is less than that server's capacity. The remaining rows ensure that the binary server variables are correct, depending on whether a user is allocated to that server. A looping N_s-diagonal submatrix $\tilde{A}$ with ones, effectively stride N_u, is the main building block of the A matrix. The corresponding vector b is servers' capacity $\vec{C}$ and zero for the user and server variables respectively.

$$\begin{aligned} \tilde{A} &= \begin{bmatrix} 1 & 0 & \dots & 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 1 & \dots & 0 \\ \vdots & & \ddots & & & \ddots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{N_s \times N_s N_u}, \\ A &= \left[\begin{array}{c|c} \tilde{A} & \vec{0} \\ \hline \tilde{A} & -(\vec{C}+1) I_{N_s \times N_s} \end{array}\right] \in \mathbb{R}^{2N_s \times N_s N_u + N_s}, \\ b &= \begin{bmatrix} \vec{C} \\ \vec{0} \end{bmatrix} \in \mathbb{R}^{2N_s \times 1}. \end{aligned} \tag{24}$$

The equality constraint encodes the single-allocation constraint, Equation (2c). When summarizing ξ, each user should only be connected to one server. The matrix A_{eq} has N_u rows, wherein in each row the N_s elements corresponding to the same user are non-zero. As the server variables ζ

do not affect this constraint, the last N_s columns are zero. The vector b_{eq}'s elements are one for all users where at least one server exists within range.

$$\begin{aligned} \tilde{A}_{eq} &= \begin{bmatrix} 1 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 1 \end{bmatrix} \in \mathbb{R}^{N_u \times N_s N_u}, \\ A_{eq} &= \left[\ \tilde{A}_{eq} \ \middle| \ \vec{0}, \ \right] \in \mathbb{R}^{N_u \times N_u N_s + N_s}, \\ b_{eq} &= [b_1, \dots, b_{N_u}]^\intercal \in \mathbb{R}^{N_u \times 1}, \end{aligned} \tag{25}$$

where,

$$b_k = \begin{cases} 1, & \text{if } \exists j \ \text{ s.t } \ d_{kj} \leq \text{cov}(s_j) \\ 0, & \text{else.} \end{cases}$$

The last constraint, proximity, seen in Equation (2b) is enforced by setting the upper bound of out-of-bounds variables to zero. This ensures that these variables are always inactive. Mathematically,

$$\begin{aligned} lb &= [0, 0, \dots, 0]^\intercal \in \mathbb{R}^{N_U N_s + N_s \times 1}, \\ ub &= [b_{11}, \dots, b_{ij}, \dots, b_{N_u N_s}, 1, \dots, 1]^\intercal \in \mathbb{R}^{N_U N_s + N_s \times 1}, \end{aligned} \tag{26}$$

where,

$$b_{ij} = \begin{cases} 1, & d_{ij} < \text{cov}(s_i), \\ 0, & \text{else.} \end{cases} \tag{27}$$

The presented ILP is then solved with Scipy's optimization packages, in turn using HigHs dual simplex method [29].

4 Numerical Results

This section presents the numerical results obtained after implementing and hyperparameter-tuning the stochastic spiking neural network. As mentioned, implementation is done in Intel Lava, and simulations are compiled and run on a CPU.

The dynamics of the input neurons, Equation (6), are set with probability constant $p_{IN} = 0.3$ and synaptic strength $w_{IN} = 3.0$. These parameters are chosen as they supplied sufficient noise in the spiking network. Furthermore, results from prior testing show that the dynamics of the principal neurons and the set of synaptic strengths result in an over-dependent system. E.g., halving the voltage threshold has the same effect as doubling the synaptic strengths. To reduce redundant variables, the neuron parameters are set before tuning.

To fairly test and evaluate the performance of both the SNN structure and ABC-BA the following section is divided into multiple tests. The specific optimization problems are presented in figures, looking like the example from Figure 2. The hyperparameters of ABC-BA are displayed below in Table 2

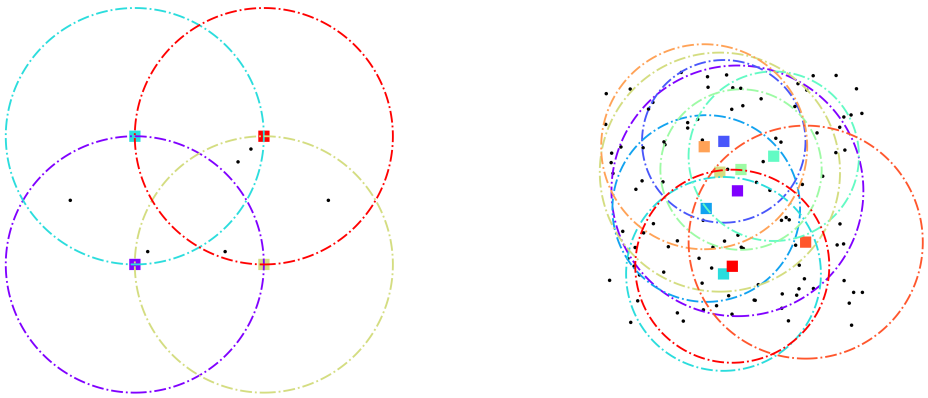
Table 2: Values of used ABC-BA hyperparameters.

Variable	Name	Value
n_b	Number of bees	8
n_a	Trials until abandonment	3
$d_{\%}$	Percentage of point used from local dataset	150%
γ	Quantile	10%
n_i	Number of samples from $l(x)$	5
I_{nb}	Iterations without accountant phase	5
I_c	Times each parameter is tested	5
σ_{min}	Kernel minimum variance	0.05
σ_{max}	Kernel maximum variance	5

4.1 Experiment Scenarios

As the designed SNN is made to work on a dynamic problem size, four differently sized problems are generated. These problems serve as benchmarks for tuning and testing. The four problem sizes are referenced as small, medium, large, and large+.

The small and medium problem sizes are chosen as they appear in the earlier work [8]. The small problem consists of six users and four servers where each server has capacity for three users and the medium problem has 100 users and 10 servers with capacity for 20 users. The generated problems are visualized and displayed in Figure 10. To achieve the best possible comparison, the small problem is the identical problem to the one in [8]. This is found by investigating the code and does explain the uniqueness of that problem's symmetrical server placement.

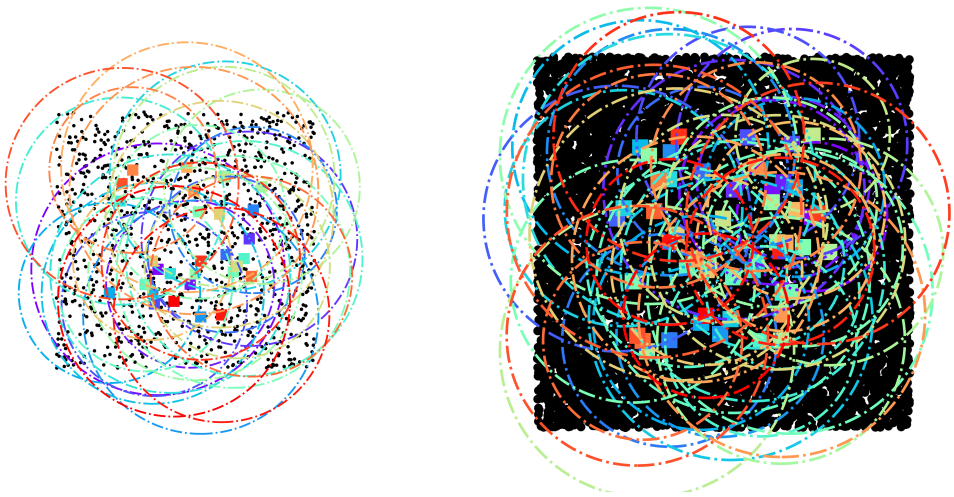


(a) Small, specific from [8] *(b) Medium*

Figure 10: Small and medium-sized benchmark problems. Squares and circles represent servers and their coverage and the dots represent users.

The large problem is randomly generated with 1000 users and 30 servers, all with a capacity for 75 users. The large+ problem has 10000 users and 50 servers with a capacity for 500 users. These larger problem sizes are generated as an implementation in the real world probably requires the network to be able to solve problems of larger sizes. However, perhaps differing from a real-world

problem, the servers' placements are too dense and overlapping. This results in a more difficult problem and is an artifact of the random problem generation. These two larger problems are seen in Figure 11.



(a) Large *(b) Large+, borderline unillustratable*

Figure 11: Large and large+ benchmark problems. Squares and circles represent servers and their coverage and the dots users.

With a larger and more complex problem, the number of possible states increases rapidly, and likewise the number of low-energy but non-optimal states. Resulting of how the SNN function, by setting too few simulation steps the SNN will usually not have time to find a solution. This results in the hyperparameter tuner mistakenly optimizing for parameters that gave quick-to-find sub-optimal solutions. Too long simulation time is however both a waste of computations and neglect to reward parameters that are both good and quick to converge. The number of simulation steps is therefore chosen to reflect the problem size. Summarization of all problem sizes and the decided number of simulation steps are shown in Table 3.

Table 3: The maximum number of simulation steps and problem sizes used in validation and tuning.

Problem size	**Number of simulation step**	**Users**	**Servers**	**Capacity**
Small	500	6	4	3
Medium	2000	100	10	20
Large	5000	1000	30	75
Large+	5000*	10000	50	500

**Limited by long computation time*

For these four scenarios, the tuned parameters for the small, medium, large, and large+ problems are denoted as p_s, p_m, p_l, and p_{l+} respectively. After running ABC-BA for each problem, the resulting tuned parameters are

$$
p_s = \begin{cases} w_{WTA} & = -6.5 \\ w_{CTA} & = -6 \\ w_{Util} & = 5 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}, \quad p_m = \begin{cases} w_{WTA} & = -10.6 \\ w_{CTA} & = -3.5 \\ w_{Util} & = 4.5 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases},
$$

$$
p_l = \begin{cases} w_{WTA} & = -34.1 \\ w_{CTA} & = -0.55 \\ w_{Util} & = 1.3 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}, \quad p_{l^+} = \begin{cases} w_{WTA} & = -100.0 \\ w_{CTA} & = -1.26 \\ w_{Util} & = 1.40 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}. \tag{28}
$$

4.2 Hyperparameter Tuning Convergence

The hyperparameter tuning is done with the proposed ABC-BA algorithm. The used hyperparameters (of the algorithm) and simulation times are presented in Tables 2 and 3. The tuning is run on the problems presented in Section 4.1.

As the EUA problem contains two different objective functions, Equations (1a) and (1b) the optimization is a multi-objective optimization. However, a good set of parameters often succeed in assigning the users to *some* servers, at worst doing a random assignment. Usually, only when the network failed is the number of assigned users less than the total number of eligible users (That is, users in coverage of at least one server). This results in that the more interesting objective is the active server minimization, and user maximization only serves to distinguish the worst parameters. Also, the SNN structure does not allow users to be unassigned, as the noise is excitatory the neurons always eventually spike. Hence, as the two objective functions do not represent conflicting goals they are merged into one single objective function. Following the notation from Equation (1) this new minimization is,

$$
\min \left(- \sum_{j=1}^{N_u} \sum_{i=1}^{N_s} \xi_{ij} + \sum_{i=1}^{N_s} \zeta_i \right). \tag{29}
$$

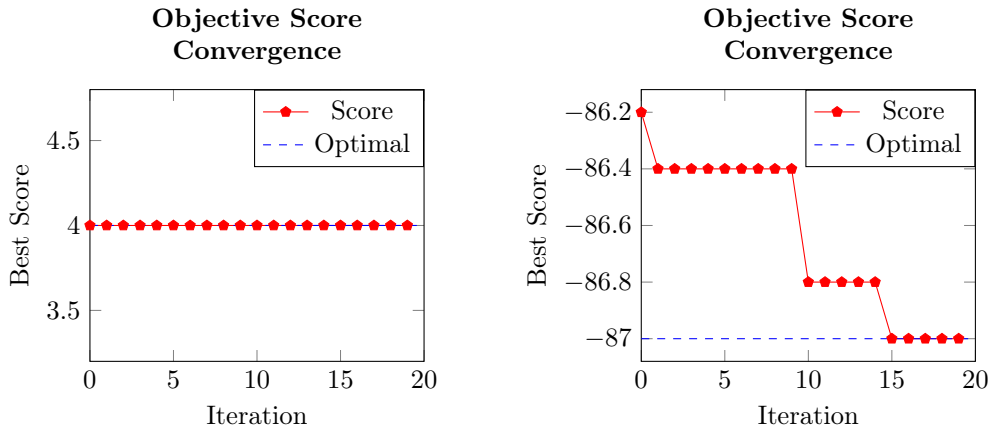
Following the above discussion, when presenting the score using Equation (29), the interesting optimization is at the range of N_s. The often much larger N_u mostly serves to heavily punish the score of parameters that are unable to find a solution with all users. As such, the difference between parameters scoring, for example, -200 and -30 is less interesting than the difference between parameters scoring -200 and -204.

Using the ILP reference solver, the optimal score of all except the large+ problem is calculated. For this size, the problem is no longer solvable with the naive ILP solver, with over a million decision variables. These optimal scores are shown in Table 4.

Table 4: Optimum score calculated with reference ILP solver, expressed using the objective function in Equation (29).

Problem size	**Optimal Score**
Small	-4
Medium	-87
Large	-985
Large+	N/A

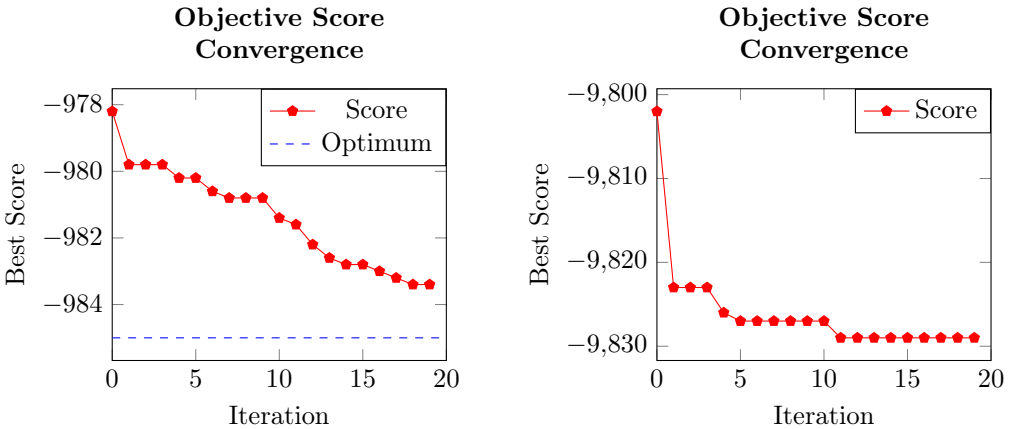
The hyperparameter-tuner is for each problem to run 20 iterations, the rather low iteration count is a necessity due to the time required to run the SNN. The scores' convergences over the iterations are seen in Figure 12 for the small and medium problems and Figure 13 for the two larger problems.



(a) Small *(b) Medium*

Figure 12: Convergence over the tuning iterations, the blue line showing the optimal score.

For the two smaller problems the tuner managed to find the optimal score during tuning. For the small problem, this is found in the first iteration, suggesting that finding a set of parameters managing this is easy as they are obtained by initial random guessing. It is therefore fair to contribute this to the SNN internal search capabilities rather than the tuner's ability to tune the parameters. For the medium problem, the tuner is shown to work properly, as the score steadily decreased until finding the optimum at the fifteenth iteration.



(a) Large *(b) Large+, optimum cannot be calculated*

Figure 13: Convergence over the tuning iterations, the blue line showing the optimal score.

For the larger problems, the optimal scores are never found, although showing steady convergence. For both the problems in Figure 13 it can be argued that the number of iterations should have been increased, seeing that the score does not converge. This is a fair observation and is primarily,

and perhaps only, done since the tuning took such a long time for the larger problems. It is also clear that the larger problems are more difficult to solve, as is expected.

4.3 Validation of the Differently Sized Problems

To evaluate the tuned parameters, seen in Equation (28), each set of parameters is run 100 times on their respective problem. This displays how the stochasticity affected the SNN's performance, and how consistent the results are. Given 100 runs, the variance in the performance is catalogued better than during tuning when much fewer runs are made.

Starting with the small problem. In [8], the optimal score of two active servers and six assigned users is found roughly 10% of runs. These results, translated to objective score, Equation (29), are seen in Table 5. The same problem run on the new SNN with parameters p_s found the optimal score every single run, vastly outperforming the earlier results on this rather trivial problem. These results are seen in Table 6 for completeness. As mentioned earlier, this great improvement is likely due to the new SNN, as parameters yielding an optimal score are found by the initial random pick.

Table 5: Results from [8] solving the small problem 100 runs.

Assigned Users	**Active Servers**	**Score**	**Score - Optimum**	**Occurrence (%)**
6	4	-2	2	45
6	3	-3	1	48
6	2	-4	0	7

Table 6: Results when running the small network after tuning.

Assigned Users	**Active Servers**	**Score**	**Score - Optimum**	**Occurrence (%)**
6	2	-4	0	100

Moving to the medium problem, with 100 users and 10 servers. In [8] the network never found any server reduction at this problem size. While not the identical problem, running the medium problem 100 times with parameters p_m gave the optimal score 99 times. The one other time the SNN instead found a solution with one more active server, seemingly an unlucky event. Simulated results from the validation on the medium problem are in Table 7.

Table 7: Results when running the medium network after tuning.

Assigned Users	**Active Servers**	**Score**	**Score - Optimum**	**Occurrence (%)**
92	6	-86	1	1
92	5	-87	0	99

Scaling upwards, the large problem with 1000 users and 30 servers has not been tested earlier and is therefore only compared against the reference solver. As hinted in the tuning convergence graph, Figure 13a, the SNN never obtains the optimum solution. It does instead most often find a solution that has one or two additional active servers, results in Table 8.

Table 8: Results when running the large network after tuning.

Assigned Users	Active Servers	Score	Score - Optimum	Occurrence (%)
999	17	-982	3	2
999	16	-983	2	24
999	15	-984	1	74

Lastly, the large+ problem with 10000 users and 50 servers is run. First, as the optimum can not be computed with the ILP solver, this column is emitted in Table 9. The results show a very different-looking performance than for the other problems. Instead of having the best value be the most common followed by cases of worse performance, these results are spread around the average performance.

Table 9: Results when running the large+ network after tuning.

Assigned Users	Active Servers	Score	Occurrence (%)
9823	31	-9821	2
9823	30	-9822	2
9823	29	-9823	4
9823	28	-9824	17
9823	27	-9825	30
9823	26	-9826	23
9823	25	-9827	16
9823	24	-9828	4
9823	23	-9829	2

These results differ since they appear more stochastic than earlier results, and probably are. It is plausible that this results from too short a simulation time and too few iterations. The tuner has optimized toward parameters that quickly find a tolerable solution, instead of parameters often finding a good solution. This also explains the incredible discrepancy between the parameters, looking at Equation 28.

The large randomness in the results is then attributed to dependency on the random initialization, that is, the state the network is in after converging to the distribution described by the energy function. This explains the Gaussian appearance of the performance, and a good SNN is not that dependent on initial noise. This Gaussian shape is seen by plotting the performance, Figure 14. Tiny favouritism is still seen for lower scores, so the network still gives some preference for lower scores. Following this reasoning, this test might seem invalid. It is however kept as the performance of these radically different parameters is still interesting.

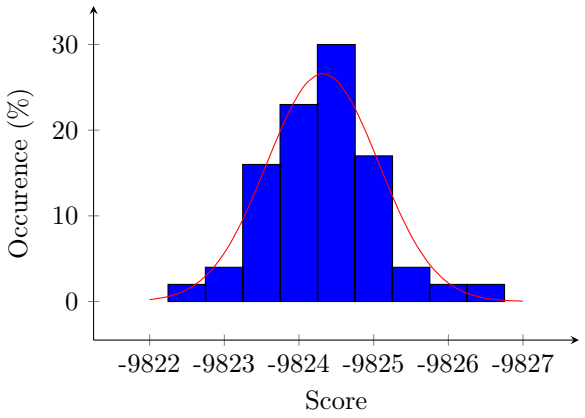


Figure 14: Bar plot of results from Table 9, with Gaussian silhouette.

4.4 Robustness

As tuning takes a long time, a tuned set of parameters preferably solve multiple problems. The worst case is the parameters being overfitted to the specific problem the parameters are tuned to, and therefore only usable for one problem instance. Robustness is, in this case, defined as how well the tuned parameters generalize to unseen problems. To evaluate the robustness of the parameters they are run on different unseen problems, starting with problems having the same number of users and servers. Five new problems are randomly generated for the small, medium, and large problem sizes. The performance is presented as the 10-run average of the computed score minus the optimum score. The large+ parameters and problem size are excluded since the reference solution is unavailable and because the p_{l^+} parameters do not really work for the problem they are tuned to. The simulated results are in Figure 10.

Table 10: Robustness results, testing obtained parameters on new problems with the same number of users and servers. 10-run average.

Parameters	**Number of simulation step**	**Average: Score - Optimum**				
		Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
p_s	500	0.0	0.0	0.0	0.0	0.0
p_m	2000	0.0	0.0	0.0	0.2	0.1
p_l	5000	2.2	146.7	1.8	1.2	1.8

Looking at the presented results in Table 10, the parameters only show slightly worse performance than reported in Section 4.3, suggesting that the tuned parameters work for multiple problems. The only true exception is one test of p_l, where the average is way larger. This highlights two things: first, the average gets very skewed by having a couple of failed runs. This happens when the user maximization objective is unfulfilled and is considered a failed run. Second, having the same number of users and servers does not guarantee equivalent problems and, consequently, results. Another metric, server coverage κ, is hereafter introduced as the average number of servers each user has access to. This adds further insight into the specific problems, not captured by merely the number of users and servers. As might be interesting, the aforementioned test has a $\kappa = 8.29$ while p_l is tuned on a problem with $\kappa = 13.13$.

Lastly, to further evaluate the robustness, all four sets of parameters are run on unseen problems of different sizes. The results are all presented in Table 11. The first three rows, separated by a line, are the exact problems from Section 4.3.

Table 11: Robustness test results, testing obtained parameters on new problem sizes. 10-run average, best scores in bold font.

Users/ Servers/ Capacity	**Number of simulation step**	**Coverage κ**	**Optimal**	**Average Score**			
				p_s	p_m	p_l	p_{l+}
6/4/3	500	2.67	-4	**-4.0**	**-4.0**	-3.9	-3.9
100/10/20	2000	4.45	-87	-86.2	**-87.0**	-85.5	-84.9
1000/30/75	5000	13.13	-985	-264.2	-976.6	**-983.8**	-972.2
50/8/15	2000	2.35	-39	**-39.0**	**-39.0**	**-39.0**	-38.8
100/10/20	2000	5.25	-95	-93.9	**-94.0**	-93.6	-92.5
200/15/30	3000	4.54	-174	-172.4	**-173.9**	-173.2	-171.8
500/20/45	4000	5.79	-481	-260.8	**-476.9**	-439.4	-402.5
600/25/45	4000	4.16	-508	-501.3	-502.0	**-506.0**	-479.8
1000/30/75	5000	5.81	-858	-847.3	-847.1	**-856.2**	-855.2
1500/30/100	5000	13.32	-1485	-405.6	-693.7	**-1478.3**	1470.1
2000/35/100	5000	9.57	-1954	-397.8	-862.4	**-1855.6**	-1251.4

Again, as was suggested earlier, it seems that the parameters work quite well on problems of similar sizes. In Table 11, there exists a clear phase transition of which parameters are the best. And when coverage and problem size is close to the values of the original problem, the parameters show great performance. From this, it seems possible to obtain a general rule, or set of equations, that describes good parameters. And in these equations, the coverage κ appears important but is not the sole deciding parameter, as p_l outperformed p_m on the 600 users problem. It might be that the average number of users per server is equally important, or that an average is insufficient to describe the problem.

Additionally, generally, it seems like the parameters corresponding to the larger problems work better on the smaller problem than vice versa. On the larger problems, the average-skewing effect of failed runs is visible. This does not seem to happen for the smaller problems. A possibility is the excitatory utilization strengths, which for the smaller problems' are larger. As the network scales so does the number of neurons connected to the utilization neurons, resulting in more firing. Too large w_{UTIL} and the network might have a risk of never finding a valid state, always having overfilled servers. This also explains the opposite effect, as the parameters with low w_{UTIL} give distributed, suboptimal, but valid solutions.

4.5 Hyperparameter Tuner Comparison

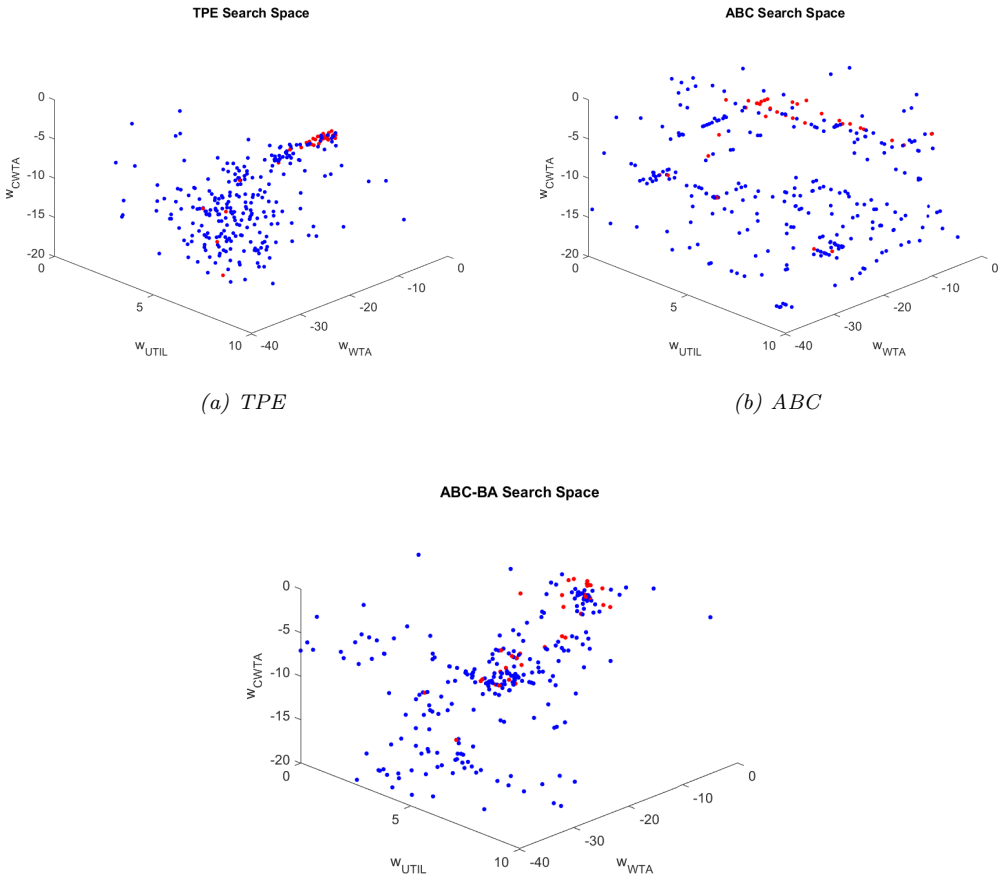
As discussed in Section 3.2, the ABC-BA is developed with the specific SNN-EUA problem in mind, believing it to be a good optimizer. Looking at the computed results from earlier sections, the ABC-BA algorithm seems to have worked fine. However, to validate this claim the ABC-BA algorithm is compared against other tuners. The two that made sense to compare ABC-BA against are the ABC and TPE algorithm, being the algorithms from which ABC-BA is built. If either of the parts outperformed the ABC-BA algorithm then the proposed algorithm has to be considered a failure. This, although trivial at a glance, is more difficult than imagined. The stochastic behaviour of both the algorithms and the SNN made proper comparison hard. All algorithms can find a set of parameters yielding good results, but it is important to separate systematic search from lucky find. The approach is to visualize the parameter space with all tested parameters marked. This approach will distinguish the lucky guesses, as the visualization gives an insight into how the final parameters are found.

As the TPE algorithm has not been implemented in isolation, an existing TPE algorithm provided by the Optuna package [30] is used. This TPE algorithm seems more sophisticated and developed

than the rudimentary TPE presented in Section 3.2, but the main features should still be the same.

The evaluation is done on the medium and large problem sizes. Results are presented as the found parameters, their performance, and a 3D slice of the parameter space with marked trials. However, as mentioned earlier, the parameters are not evaluated in isolation but in the context of how the search space looked. Starting with the medium problem, the plotted search spaces are in Figure 15, and the obtained parameters are

$$p_m^{TPE} = \begin{cases} w_{WTA} & = -14.57 \\ w_{CTA} & = -5.35 \\ w_{Util} & = 6.81 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}, \qquad p_m^{ABC} = \begin{cases} w_{WTA} & = -13.26 \\ w_{CTA} & = -3.28 \\ w_{Util} & = 3.67 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}. \tag{30}$$



(a) TPE *(b) ABC*

(c) ABC-BA

Figure 15: 3D slice of parameter space — Medium problem size. Red dots highlighting the top 10% of results.

Looking at Figure 15, the different search styles of the three algorithms are clear. Unaltered ABC gave a very sparse and explorative search while TPE almost oppositely seems to heavily exploit a small region around where it has the best parameters. ABC-BA inherits aspects from both

its parent algorithms, resulting in multiple smaller clusters of exploitation. The TPE and ABC-BA clusters are centred in similar regions, although the ABC-BA also explores the surrounding region. For many continuous objective functions, a small change can represent an improvement. But in this case, it is assumed that small changes do not alter results for the discrete step-shaped objective function (at least not as much as the SNN stochasticity). Fine-grained exploitation might therefore not be as important when tuning the SNN. When imposing a minimum value on the kernels' variances the performance of a datapoint is also inferred on the surrounding area. Also, as seen in Figure 15c, the ABC-BA algorithm's best points are found on the boundary of the upper cluster. This cluster is likely an early found local minimum that ABC-BA tried to exploit. While this might correspond to a good region, finding consistently performant parameters is tricky. Especially since each parameter is only run a finite number of times. Escaping this and finding better results outside the clusters is a plus for ABC-BA, as TPE seems stuck in a local minimum.

Evaluating the parameters 100 times, results shown in Table 12, shows that p_m^{TPE}, while seemingly the most confident managed to find the worst parameters, failed in assigning all users 24 times. It is however clear that the parameters can give lucky results as good solutions are found in roughly 75% of runs. Running each parameter additional times and this inconsistent solution would have been detected. ABC found quite good parameters with a consistently good score, but from Figure 15b it looks like it found it by accident. The best parameters are found by ABC-BA, while also displaying the most favourable search pattern.

Table 12: Results from running the TPE, ABC, and ABC-BA parameters. Medium problem.

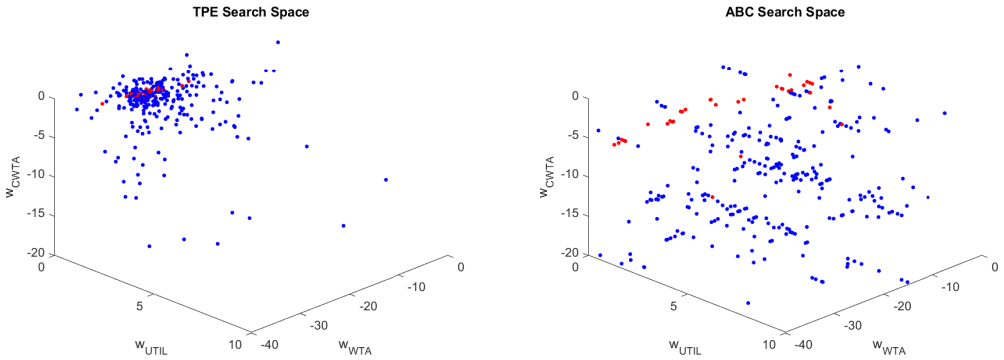
Parameters	**Score**	**Score - Optimum**	**Occurrence (%)**
p_m	-87	0	99
p_m	-86	1	1
p_m^{TPE}	-87	0	15
p_m^{TPE}	-86	1	49
p_m^{TPE}	-85	2	11
p_m^{TPE}	-84	3	1
p_m^{TPE}	-11	76	1
p_m^{TPE}	-8	79	1
p_m^{TPE}	-7	80	2
p_m^{TPE}	-6	81	2
p_m^{TPE}	-5	82	6
p_m^{TPE}	-4	83	2
p_m^{TPE}	-3	84	6
p_m^{TPE}	-2	85	4
p_m^{ABC}	-87	0	14
p_m^{ABC}	-86	1	86

The same tests and comparisons are done for the large problem. The found parameters are in Equation (31), the visualized search spaces in Figure 16, and the parameters' performance in Table 13.

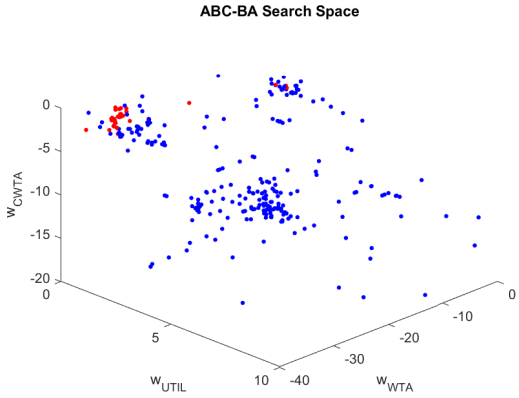
Looking at Figure 16, the algorithms display similar behaviour as when tuning the medium problem. ABC still seems completely random, almost looking identical to the search in Figure 15b. TPE again has one large cluster and ABC-BA has several small. This time the TPE algorithm displays a much broader distribution, probably resulting from the algorithm finding several good, but spaced, parameters before going into deeper exploitation. The best parameters that TPE found are not in the center of the cluster, instead being the red dot left of the cluster (in Figure 16a). It is this upper-left region that ABC-BA exploits, as seen in Figure 16c.

Comparing performance, the parameters found by TPE and ABC-BA are very similar and therefore show very similar performance. The ABC-BA does give a slightly more consistent score, which makes sense seeing that it exploited the region while TPE seems "lucky" in obtaining parameters from that region. ABC gave the worst result, finding parameters that consistently found subpar solutions.

$$p_l^{TPE} = \begin{cases} w_{WTA} & = -35.54 \\ w_{CTA} & = -0.43 \\ w_{Util} & = 1.32 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}, \qquad p_l^{ABC} = \begin{cases} w_{WTA} & = -11.44 \\ w_{CTA} & = -3.12 \\ w_{Util} & = 2.71 \\ {\tau_P}^{-1} & = 0.1 \\ V_P & = 8 \end{cases}. \tag{31}$$



(a) TPE *(b) ABC*



(c) ABC-BA

Figure 16: 3D slice of parameter space — Large problem size. Red dots highlighting the top 10% of results.

Table 13: Results from running the TPE, ABC, and ABC-BA parameters. Large problem.

Parameters	Score	Score - Optimum	Occurrence (%)
p_l	-984	1	74
p_l	-983	2	24
p_l	-982	3	2
p_m^{TPE}	-984	1	62
p_m^{TPE}	-983	2	33
p_m^{TPE}	-982	3	5
p_m^{ABC}	-983	2	1
p_m^{ABC}	-982	3	1
p_m^{ABC}	-981	4	45
p_m^{ABC}	-980	5	40
p_m^{ABC}	-979	6	12
p_m^{ABC}	-978	7	1

5 Discussion

The potential for stochastic spiking neural networks and neuromorphic hardware is big, while this thesis has focused on their application in solving constraint satisfaction problems, SNNs have numerous other uses. Especially regarding CSPs, the research suggests that this is an area with a lot to gain in energy efficiency.

All implementations presented throughout this thesis are written and simulated using Intel Lava. Unfortunately, as the actual neuromorphic hardware was not made available all simulations had to be compiled and run on a CPU. The implementation should however compile and run on Loihi chips, but investigations on the energy efficiency must be left to future research.

Observing the results, it must be said that the network can solve the simpler edition of the EUA problem rather well. The SNN does not always find the optimum solution, especially for larger problems, but that was never the ambition. The SNN is instead useful for finding a satisfactory solution cheaply. It usually misses the optimum since the SNN has trouble filling all servers to absolute capacity, usually finding solutions where servers are slightly below capacity, resulting in a couple of additional active servers. This should also be the case when extending the resource demand and capacity to vectors.

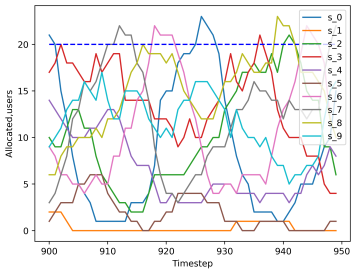
5.1 Improvement from Earlier Implementation

What cannot be understated is the monumental improvement to the network, compared with the earlier implementation. As the networks are conceptually the same, one can wonder what changes made this difference. A key change is the split between the principal and auxiliary networks. For example, the WTA in the earlier network consisted of directly interconnected neurons. Disregarding that the research and mathematical formulations on the principal network required symmetrical connections, this allows massive and sudden voltage changes. In this system, adding an additional server increases the number of WTA-related synapses by $2(N_s - 1)$ instead of only 2. This allows the theoretical max change of voltage resulting from the WTA to be $w_{WTA} \times (N_s - 1)$ instead of w_{WTA}, resulting in difficult scaling. Furthermore, the prior WTA does not inhibit all neurons within the WTA, not affecting the neuron that triggered the WTA. This skewed the energy distribution, allowing the initial noise to completely dominate the entire behaviour of the network. This claim can be tested by randomly assigning users to servers within range, to mimic the randomness of the initial noise. Comparing with the presented results, Table 5, running this random assignment one hundred thousand times for the small problem, Figure

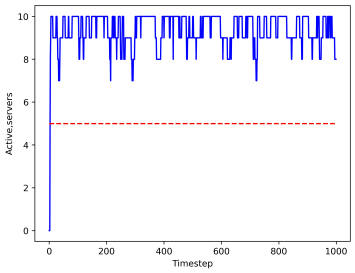
10a, give results corresponding to two, three, and four active servers 5.6%, 50.5% and 43.8% of the times, respectively. This also explains why no server reduction is ever found for the medium-sized problem. The reintroduction of the energy distribution, treating the process as a guided stochastic search of the states yields a substantially better solver (as is done in the underlying theory, see [4, Figure 2]). Lastly, the change to the capacity neurons and incorporation of an internal state monitor allowed the search process to proceed whilst using the incoming spikes to deduce the solver's best solution.

5.2 Dynamic of the Network

The SNN network is however not without flaws, the root of these problems is found in the dynamics of the utilization and capacity motifs. These two motifs receive and send spikes to the same neurons but utilization is excitatory and capacity is inhibitory. This results in a tug-of-war, where the balancing act is quite delicate. Too strong excitatory synapses and the network never finds a solution adhering to the capacity constraint and too weak results in solutions that do not fully utilize the servers. What often seems to happen is that utilization makes multiple neurons fire simultaneously (same simulation step) which overshoots capacity. Capacity then puts a cooldown on those neurons, making sure no more connections are made to that server. The other servers then sequentially steal users from the overfull server, maybe resulting in another server overfilling. The network, therefore, when testing solutions with the minimum amount of servers, usually exists in invalid states. It then finds a valid solution almost by chance when the users "accidentally" get correctly distributed.



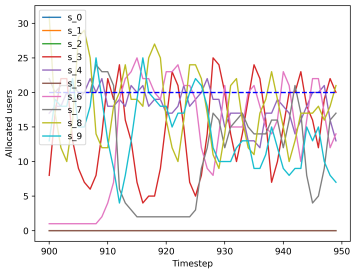
(a) Slice of a simulation showing capacity of all servers. Blue line illustrating capacity constraint



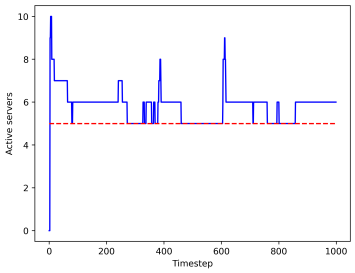
(b) Active servers at any time. The red line illustrates the optimum amount

Figure 17: Exploration of the SNN, medium-sized problem. Calm SNN.

This overfilling behaviour is both good and bad. It is undesirable since it spends a needlessly long time in invalid states, apparently only attempting to distribute users. It also sometimes never finds the solution that is right in front of it, due to bad luck. But on the contrary, it seems like this behaviour is needed to obtain the best results, the servers aggressively trying to take the users. The servers' tur-of-war, between taking as many users as possible and freezing from overshooting capacity is what drives the removal of redundant servers while still allowing new servers to try allocating the users. Also, the placement of users and servers might require some servers to be active; the aggressive behaviour gives servers with only a few users a fighting chance.



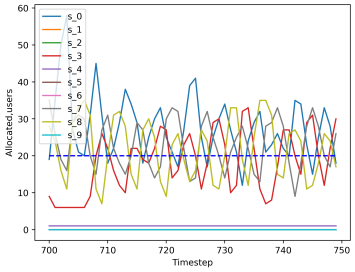
(a) Slice of a simulation showing capacity of all servers. The blue line illustrates capacity constraint.



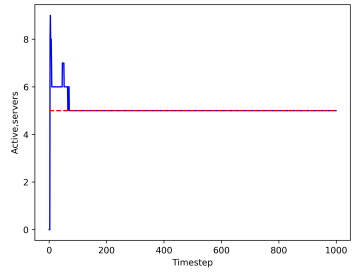
(b) Active servers at any time. The red line illustrates the optimum amount.

Figure 18: Exploration of the SNN, medium-sized problem. Aggressive SNN. Note that the minimum amount of servers not necessarily means that the currently active server can obtain the optimum score.

To visualize this, the number of active servers and the capacity of each server are plotted for three networks with different characteristics: calm, aggressive, and hyper-aggressive. These are in Figures 17 – 19. As guessed, the calm network allows for server switching but never fills up the servers while the hyper-aggressive overfills the servers and practically locks in active servers. This is worse than the calm network, as even though the active number of servers is optimal, these currently active servers might not be able to fit the users, resulting in a network that never finds a valid state. The best are the aggressive networks, filling up and reducing servers while still constantly allowing the active servers to change.



(a) Slice of a simulation showing capacity of all servers. Slice of a simulation showing capacity of all servers. The blue line illustrates capacity constraint.



(b) Active servers at any time. The red line illustrates the optimum amount

Figure 19: Exploration of the SNN, medium-sized problem. Hyper-aggressive SNN.

5.3 Reference Solver ILP

Moving on, the decision to implement a reference solver was fruitful. The resulting ILP solver is a great tool to evaluate the performance of the SNN. This too is easy to scale up to for a resource vector, only needing to extend the inequality matrix.

The ILP solver with the dual simplex solver has superior performance to the SNN for smaller problems, but since the number of decision variables grows like $N_u \cdot N_s$ the ILP solver is not, on the writer's computer, able to run the large+ problem. The ILP is however used for the small,

medium, and large sized problem. This made it possible to evaluate the SNN, but also to see if the tuner succeeded in finding the optimum value. This testing is called validation and results are presented in Section 4.3.

5.4 Validation and Robustness

Starting with the small problem, this specific problem is chosen mainly for comparison purposes. As written earlier, multiple parameter sets allow the SNN to find the optimum. As no convergence existed the final choice of parameters became arbitrary. Regardless, the parameters in Equation (28) successfully obtained the best solution for all validation runs.

With the medium-size problem, the parameters obtained the optimal score for almost all validation runs. The single run with the non-optimal score is, as mentioned earlier, probably a case where the SNN got unlucky with the overfilled servers. Here the effect of the tuner is visible, as seen by the convergence graph in Figure 12b.

With the large problem, the convergence in Figure 13a clearly shows the effect of the tuner. The obtained parameters never found the optimum score but rather the second-best score most of the time. This can be either because the optimum is hard to find, the simulation is too short, or perhaps the truly optimal parameters are not found.

The large+ problem, as mentioned in Section 4.3, underwent flawed tuning. With such a large problem, the lava framework has unreliable performance and slow computations. The maximum number of simulation steps is therefore probably not as high as it should have been. This results in the tuner finding parameters that reliably and quickly find subpar solutions, a "calm" network as described earlier. And due to how the "aggressive" networks find their solutions, this problem size might require an exceedingly long simulation time.

Results from running the sets of parameters on newly generated problems, Tables 10 and 11, suggest that the parameters work well on similar problems. This is excellent as it means that the SNN does not need tuning for every single problem instance, which is unsustainable. In Table 11, the clear phasing of the best parameters (bold font) displays the trend of "larger" parameters working better on larger problems. Following this, it appears possible to use earlier tuned parameters as starting guesses, perhaps massively reducing the parameter space before doing a fine-grained tuning. However, for this, a good way to characterize "similar" is required. Using the number of users and servers gives a relatively good understanding of this, but fails to capture the denseness of the servers. Introducing the coverage parameter κ gave a way to quantify the geographic denseness of the servers. And while κ seems important, it is not the sole determining factor for problem similarity (As p_l outperforms p_m on the 600-user-problem).

Looking at all these results, it seems possible to find a trend of what constitutes good parameters. Generally, it seems that increased size requires a larger w_{WTA} and smaller w_{CWTA} and w_{Util}. Making a qualified guess, as both w_{WTA} and w_{CWTA} represent exploration cooldowns, when a user establishes a new connection, w_{WTA} gives time for the network to explore the solution, stopping that user from rapidly changing servers. With a larger problem size then perhaps more exploration time is needed and thus an increase in w_{WTA}. w_{CWTA} stops new connections from being made to an overfilled server, giving time for the other servers to allocate users. With users having access to more servers (larger problems) this time might be short. In this case, having large w_{CWTA} might result in the users being too spread out. A study trying to find or explore these relationships is both important and interesting but is left out of this thesis work. The derived approximation of the energy function might also be useful for this, if not only to obtain bounds for the parameters. For this thesis, the tuner remained the tool for finding good parameters.

5.5 Hyperparameter Tuners

The three considered tuners: ABC, TPE, and their combination ABC-BA are compared on the medium and large problems. As mentioned earlier, fair comparisons between the tuners are difficult when only looking at the quality of the solution. Since the SNN is itself an optimizer, many parameters work given sufficient luck with the stochastic network. And the longer the runtime is, the more chances the network has to sample the optimum. To remove some of the stochastic performance, multiple runs of the same configurations are made. For the presented results, this number is five. The idea is to separate the sometimes-lucky from the often-lucky parameters. The tuner comparison mainly consists of looking at how the algorithms searched the parameter space, revealing their search strategy, and then seeing how lucky solutions affected the search. Choosing suitable runtimes is also important, theoretically, all parameters can find the optimum with enough time. By setting a maximum time shorter than infinity, the network is implicitly tuned for quick convergence. As good aggressive networks usually require some time to sample a good solution, with too short runtime the network is implicitly tuned for calm parameters that reliably find subpar solutions.

A general observation regarding the algorithms' search is that the TPE and ABC algorithms have extremely different approaches. ABC displayed no clear exploitation, instead just appearing like random search, Figures 15b and 16b. The best parameters are for ABC found by chance rather than intelligent search. TPE has great exploitation but seems to converge to a too-restricted region, especially since the other algorithms clearly found good parameters outside this region. Looking at Figure 15a especially, TPE seems completely stuck in a small region. For function optimization, this is desired as the optimizer should converge to the point minimum. For this problem, tiny variations do not seem to change the SNN's behaviour and since the objective function is step-like it is more beneficial to do wider exploitation.

The algorithm combination, ABC-BA displays characteristics of both the parent algorithms. It has multiple small regions of exploitation, clusters, as well as the ABC's random search between them. It found the best solutions in the same region as TPE but kept exploring. This is perhaps most clear looking at Figure 15c, four regions can be seen of which two are properly exploited. Furthermore, it seems like the Bayesian elements constitute the intelligent search while the ABC elements broaden its view by adding more unique data points. While partly due to design, the ABC-BA algorithm wastes computations in the bad clusters. This is likely the trade-off to having bees compute local distributions, some bees will unknowingly pick up the parameters corresponding with reliably sub-par solutions.

One thing that probably resulted in unfavourable results for ABC is how the fitness function is computed, Equation (11). For example, a parameter reliably scoring -925 only has a slight advantage over parameters scoring -920 and -900, all with fitness 926, 921, and 901. These minor numerical differences represent good, okay, and bad solutions. If only these solutions are considered, the probabilities of each attracting accountants/onlookers are 33.7%, 33.5%, and 32.8% respectively. Due to the large number of users, the accountants/onlookers get evenly split. This should have been foreseen and is probably the reason why ABC got not exploitation and why ABC-BA has so many trials in the worse regions. Also, swarm algorithms usually use a larger swarm than eight members and more iterations than twenty. The results of using pure ABC could maybe have been predicted, as it might not be meant for these computationally expensive systems. TPE and Bayesian-styled algorithms are probably the way forward, as the time it takes to evaluate the systems is so much longer than the time for computing the next parameters. Using all available information from earlier runs also resulted in low-iteration convergence and very few runs of parameters that yielded invalid solutions. TPE and ABC-BA do however give rise to many new hyperparameters.

6 Conclusions and Future work

Edge computing is an approach to ensure continued high-quality connections to service providers when the number of cloud devices greatly increases. To ensure the functionality of edge computing, the devices need to be assigned and connected to a local edge server. The edge user allocation problem consists of assigning as many devices/users to as few edge servers as possible. This can be reformulated as a constrained satisfaction problem. In this thesis, the design for a stochastic spiking neural network that solves the edge user allocation problem is shown. SNNs have been shown to decrease energy consumption for constrained satisfaction problems by a factor of a thousand when run on neuromorphic hardware [5]. The thesis also presents a proposed modification to the artificial bee colony algorithm, ABC-BA, meant to improve the algorithm's capability to tune the hyperparameters of stochastic spiking neural networks.

The network is first tuned on four different problem sizes, ranging from 6 to 10000 users. Each hyperparameter tuning is 20 iterations long with simulation time depending on the problem size. Obtained parameters are run 100 times to catalogue their consistency. Then the performance of the network with tuned parameters is tested on unseen problems. Lastly, some experiments on the developed hyperparameter tuner are made. These compared the search pattern and computed parameters for the proposed algorithm and its building blocks.

The tuned new SNN displays a large improvement to the earlier implementation [8], easily finding the true minimum for the 6 and 100 users problem-size where the earlier SNN does not. When increasing the problem size, the SNN quite consistently found solutions only slightly worse than the optimum. The optimum is computed by rewriting the edge user allocation problem as an integer linear programming problem and solving it with an established ILP solver. While the SNN is able to find suboptimal solutions for the 10000-user problem, the ILP for that problem size cannot be run given the current computational resources.

The SNN's parameters dictate both the performance and behaviour of the network. Results suggest that the best performance is obtained when the network's behaviour is aggressive, that is, rapidly changing states and often existing in states corresponding to invalid solutions. Aggressive networks seem to take time to find optimal solutions, while parameters resulting in calmer networks usually quickly and reliably find suboptimal solutions. Without careful simulations, the tuner can accidentally tune a calm network, as might have been the case with the tuning of the largest problem.

The performance of the proposed tuner, ABC-BA, is compared to the artificial bee colony algorithm and the tree-structures Parzen estimator algorithm, the two algorithms essentially creating ABC-BA. Results show that while introducing many new hyperparameters, the two algorithms with Bayesian elements display a much more intelligent search pattern. For a computationally costly network, where tuning is limited, the artificial bee colony algorithm seems no better than random guessing. Results seem favourable for ABC-BA, finding superior parameters in a more desirable search pattern. To properly investigate whether this conclusion holds, more careful, repeated, and thorough testing between the algorithms is needed on both the EUA and other problems.

For future work, continuing to scale up the network and problem will provide interesting insight into possible limitations. As the current implementation takes a very long time, problems larger than 10000 users are skipped. Designing more accurate tests will also provide good insights, as perhaps the apparent shortcomings of the network when each user has access to 30+ servers might be irrelevant. Also, just barely skipped in this thesis, is a systematic comparison of runtimes between the SNN and ILP solver. Although the SNN is currently simulated on a CPU, the SNN hints at better runtime scaling than the ILP solver (Managing to run the 1arge+ problem). Additionally, continuing to develop ABC-BA and test the algorithm on other problems will probably give rise to more tweaks and improvements. The choice of the fitness function is already a clear source for improvement. And lastly, trying to find a relationship between the good parameters

and the network's performance will bypass the need to tune the edge user allocation problem. This will massively improve the usefulness of the SNN but also require a lot of thorough testing to correctly characterize the effect of each parameter. This was left as future work.

References

[1] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. Optimal edge user allocation in edge computing with variable sized vector bin packing. In *Service-Oriented Computing*, Lecture Notes in Computer Science, pages 230–245, Cham, 2018. Springer International Publishing.

[2] Edge computing - a must for 5g success. `https://www.ericsson.com/en/edge-computing`. Accessed: 2023-03-10.

[3] If the human brain were so simple that we could understand it, we would be so simple that we couldn't. `https://quoteinvestigator.com/2016/03/05/brain/`. Accessed: 2023-05-10.

[4] Zeno Jonke, Stefan Habenschuss, and Wolfgang Maass. Solving constraint satisfaction problems with networks of spiking neurons. *Frontiers in neuroscience*, 10:118–118, 2016.

[5] Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.

[6] Gabriel A Fonseca Guerra and Steve B Furber. Using stochastic spiking neural networks on spinnaker to solve constraint satisfaction problems. *Frontiers in neuroscience*, 11:714–714, 2017.

[7] Quadratic unconstrained binary optimization (qubo) with lava. `https://github.com/lava-nc/lava-optimization/blob/release/v0.2.0/tutorials/tutorial_02_solving_qubos.ipynb`. Accessed: 2023-05-30.

[8] Kim Petersson Steenari. A neuromorphic approach for edge user allocation. 2022.

[9] Daniel Haşegan, Matt Deible, Christopher Earl, David D'Onofrio, Hananel Hazan, Haroon Anwar, and Samuel A. Neymotin. Training spiking neuronal networks to perform motor control using reinforcement and evolutionary learning. *Frontiers in computational neuroscience*, 16:1017284–1017284, 2022.

[10] Catherine D. Schuman, J. Parker Mitchell, Robert M. Patton, Thomas E. Potok, and James S. Plank. Evolutionary optimization for neuromorphic systems. In *Proceedings of the Neuro-Inspired Computational Elements Workshop*, NICE '20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] N.G. Pavlidis, O.K. Tasoulis, V.P. Plagianakos, G. Nikiforidis, and M.N. Vrahatis. Spiking neural network training using evolutionary algorithms. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005*, volume 4, pages 2190–2194 vol. 4. IEEE, 2005.

[12] José Altamirano-Flores, Manuel Ornelas, Andres Espinal, Raul Santiago-Montero, Héctor Soberanes, Martín Carpio, and Sergio Tostado. Comparing evolutionary strategy algorithms for training spiking neural networks. *Research in Computing Science*, 96:9–17, 12 2015.

[13] Training spiking neural models using artificial bee colony. *Computational intelligence and neuroscience*, 2015:947098–14, 2015.

[14] Roberto A. Vazquez. Training spiking neural models using cuckoo search algorithm. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 679–686, 2011.

[15] Dervis Karaboga. An idea based on honey bee swarm for numerical optimization, technical report - tr06. *Technical Report, Erciyes University*, 01 2005.

[16] Chunfeng Wang, Pengpeng Shang, and Peiping Shen. An improved artificial bee colony algorithm based on bayesian estimation. *Complex intelligent systems*, 8(6):4971–4991, 2022.

[17] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[18] Lava software framework. `https://lava-nc.org/`. Accessed: 2023-05-08.

[19] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[20] Joao D. Nunes, Marcelo Carvalho, Diogo Carneiro, and Jaime S. Cardoso. Spiking neural networks: A survey. *IEEE access*, 10:1–1, 2022.

[21] Moses Agebure, Paula Wumnaya, and Edward Baagyere. A survey of supervised learning models for spiking neural network. *Asian Journal of Research in Computer Science*, pages 35–49, 06 2021.

[22] Neuromorphic constrained optimization library. `https://github.com/lava-nc/lava-optimization`. Accessed: 2023-05-09.

[23] John J. Hopfield and David W. Tank. Computing with neural circuits: A model. *Science (American Association for the Advancement of Science)*, 233(4764):625–633, 1986.

[24] Sander M. Bohté, Joost N. Kok, and Han La Poutré. Spikeprop: backpropagation for networks of spiking neurons. In *The European Symposium on Artificial Neural Networks*, 2000.

[25] Spike-timing dependent plasticity. `http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity`. Accessed: 2023-03-10.

[26] Ashraf Mohamed Hemeida, Somaia Awad Hassan, Al-Attar Ali Mohamed, Salem Alkhalaf, Mountasser Mohamed Mahmoud, Tomonobu Senjyu, and Ayman Bahaa El-Din. Nature-inspired algorithms for feed-forward neural network classifiers: A survey of one decade of research. *Ain Shams Engineering Journal*, 11(3):659–675, 2020.

[27] Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, and Masaki Onishi. Multiobjective tree-structured parzen estimator for computationally expensive optimization problems. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 533–541, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Zeno Jonke, Stefan Habenschuss, and Wolfgang Maass. Supplementary material: Solving difficult computational tasks with spiking neurons. 10, 2016.

[29] Scipy documentation: optimize.linprog. `https://docs.scipy.org/doc/scipy/reference/optimize.linprog-highs-ds.html#optimize-linprog-highs-ds`. Accessed: 2023-05-07.

[30] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.