university of groningen / faculty of science and engineering / artificial intelligence

Memory Consolidation by Deep-Q Forward-Forward Learning in Games

Bachelor's Project Thesis*

Floris de Kam, s4748050, f.p.j.de.kam@student.rug.nl,
Supervisor: J.J.M.A. (Jordi) Timmermans

Abstract: Neural networks have been pivotal in transforming various fields through machine learning techniques. Training of these networks relies heavily on the backpropagation algorithm, which, despite its success, presents several limitations. These include large memory requirements and limited biological plausibility. This thesis implements the novel Forward-Forward (FF) algorithm, which locally optimizes a neural network by performing two forward passes. FF is tested on blackjack to explore its performance in simple games. This research extends the FF algorithm with Deep-Q Forward-Forward Learning (DQFFL), which combines FF with reinforcement learning to enable an FF neural network to learn on the fly. The results show performance for FF comparable to traditional backpropagation while reducing memory capacity and improving biological plausibility. The performance of DQFFL evaluated on two simple game environments indicates a promising subject for future research. This study contributes to the field of neuromorphic computing by presenting an alternative local learning rule for neural networks in reinforcement learning.

1 Introduction

Deep neural networks play a crucial role as Machine Learning (ML) models that currently shape and transform our society (LeCun et al., 2015). Essential to this success has been the training of these neural networks using stochastic gradient descent, where the gradients are often calculated iteratively using BackPropagation (BP) (Rumelhart et al., 1986). The BP algorithm is effective but has some drawbacks.

Firstly, BP requires the storage of all activations in memory, since they are necessary for the backward pass through multiple layers. This makes BP computationally more memory-expensive than local learning rules.

A second limitation is that a backward pass of the BP algorithm can only be performed when the computation of the forward pass is completely known and differentiable.

Lastly, BP is not considered a biologically plausible model of how the brain learns (Crick, 1989; Gardner, 1993; Stork & Hall, 1989; Lillicrap et al., 2020). "There is no convincing evidence that the cortex explicitly propagates error derivatives or stores neural activities for use in a subsequent backward pass"(G. Hinton, 2022). The brain continually processes temporal sensory inputs and utilizes a learning method that processes and learns from this data simultaneously. The biological implausibility is not a problem in itself, but nature might provide us with insights on how to design a more efficient algorithm.

To reduce memory capacity issues and make learning more biologically plausible, the recently published paper by G. Hinton (2022) introduces the Forward-Forward (FF) algorithm. It is a training algorithm for multi-layer perceptrons inspired by Boltzmann machines (G. E. Hinton et al., 1986) and noise-contrastive estimation (Gutmann & Hyvärinen, 2010). The forward pass in the FF algorithm is comparable to its BP counterpart. The backward pass, however, is replaced by another forward pass. Distinctive are the objective function and inputs, different for the two otherwise similar forward passes. Inputs consist of a training sample and cor-

*Code available at: github.com/Floris93100/Blackjack-FFNN

responding label. The first pass contains positive training samples and updates weights to increase the "goodness". A second pass then uses generated negative training data with false labels to decrease goodness for these samples. The network is greedily optimized layer by layer using gradients provided by a local goodness function.

Forward-Forward Neural Networks (FFNN) have comparable performance to BP networks on the MNIST (G. Hinton, 2022) and CIFAR-10 (Krizhevsky et al., 2009) datasets. Additionally, they offer the benefit of learning while processing sequential data through the network. They do not generalize as well as BP networks do, but can still be a considerable alternative as a more biologically plausible model or when power is limited.

Several variations and implementations of the algorithm have since been proposed. FF is extended to sentiment analysis and evaluated using varying training parameters (Gandhi et al., 2023), combined with a convolutional neural network for better image classification (Scodellaro et al., 2023), with recurrent neural systems to build a predictive model (Ororbia & Mali, 2023) or with self-supervised representation learning (Brenig & Timofte, 2023). The FF algorithm has been adapted and implemented in microcontrollers (De Vita et al., 2023). Further variations include a hybrid network with local backpropagation (Giampaolo et al., 2023) and adjusting the loss function (Lee & Song, 2023).

So far, there has been no research on the performance of the FF algorithm in (video) games. The ability of ML models to learn to play games has been extensively investigated (Skinner & Walmsley, 2019). Video games provide a controlled and cost-effective environment for training and developing algorithms. The problem-solving capabilities useful in games can also be applied to real-world problems requiring similar decision-making processes.

This thesis contributes by applying and investigating the performance of the FF algorithm on games. We first train an FFNN to play blackjack, a simple casino card game that has an optimal strategy, to try and find out whether an FFNN can learn to play simple games given known optimal actions. The results of training an FFNN are subsequently compared to a BP baseline. We also introduce a new adaptation of the FF algorithm, coined Deep-Q Forward-Forward Learning (DQFFL). This algorithm uses an FFNN as a function approximator in reinforcement learning.

2 Theoretical Background

2.1 Environments

2.1.1 Blackjack

This thesis investigates the performance of the FF algorithm on the simple casino card game of blackjack, also known as 21. Blackjack was chosen as a toy problem since it is a stochastic game with known optimal actions. There is a statistically best action for each combination of the player's hand and the dealer's up-card. The strategy in which a player picks the statistical best action is called the basic or optimum strategy (Baldwin et al., 1956; Epstein, 2012). Knowing optimal actions allows us to initially create training data to test the effectiveness of an FFNN on a supervised learning problem.

In blackjack, a player tries to get as close to 21 without going over it, while competing against the dealer's cards. Numbered cards are worth their face value, face cards are worth 10 and an ace is either 1 or 11. The player and dealer start with two cards each. Only their own, and one of the dealer's cards, called the "up-card", are visible to the player. A player can keep hitting, i.e., receiving cards, until choosing to stand. The dealer then takes cards from the deck until their hand totals 17 or more. A player can also choose to double down -doubling their bet and receiving only one more card- or split if they have a pair. In case of a split, the game continues as if played with two regular hands. The player wins if their total is more than the dealer's. A player loses if they "bust" (their hand value is over 21) or their hand total is lower than the dealer's. Splitting twice or after doubling down is not allowed. If an agent's action selection policy selects a disallowed action, the agent hits. The agent always stands if the player's hand totals 21, also known as blackjack.

Blackjack's discrete observation space consists of five integer values. One observation contains the player's hand total, the dealer's up card, whether the player has a usable ace, whether doubling down is allowed, and whether splitting is permitted. The player's hand total can have values ranging from 4 to 21, and the dealer's hand ranges from 2 to 11. The last three observations are boolean values. An

Table 2.1: Rewards for the Blackjack Environment

Situation	Reward
Win with double down	+2.0
Win game	+1.0
Draw game	0
Lose game	-1.0
Lose with double down	-2.0
Split hands	Combination
Natural blackjack	+1.5

ace counts as eleven when it is 'usable' and one when it is not. An ace is called usable when it can be used as one or eleven and this eleven does not cause the player's total to exceed 21. The game is not deterministic and the environment is not fully observable.

For this thesis, blackjack is implemented by extending the Farama Gymnasium environment (Towers et al., 2023) with the double down and split actions. The total reward is returned at the end of an episode (see Table 2.1). All the rewards have been chosen according to payouts by blackjack in real-life casinos. For example: a player doubles their bet when winning, so the net payout (reward) is +1.0. We differ from Sutton & Barto (2018) in choosing natural blackjack to pay 1.5, as is common in casinos.

2.1.2 Frozen Lake

The second environment is a reinforcement learning toy problem by Farama's Gymnasium (Towers et al., 2023). In the frozen lake environment, the agent's goal is to cross a frozen lake from start to finish without falling into a hole in the ice. The start is at the top left and the finish is at the bottom right of a 4x4 map (see Figure 4.6). The layout of the map is the same for every game. The discrete observation space encompasses the one-hot encoded location, which can be one of the sixteen places on the map. There are four discrete actions corresponding to the movements: left, down, right and up. An agent receives a reward of +1 when reaching the goal. An episode terminates if the agent falls into a hole, reaches the goal tile, or if the episode takes more than 100 timesteps. The environment is deterministic and not fully observable.

Figure 2.1: The Frozen Lake Gymnasium environment

2.2 The Forward-Forward Algorithm

The Forward-Forward (FF) algorithm by G. Hinton (2022) is a "greedy multi-layer learning procedure" that can be viewed as an application of the predictive coding brain model in machine learning. The predictive coding framework is based on the idea that the brain constantly makes predictions about sensory inputs and subsequently adjusts these predictions according to the difference between expected and actual information (Rao & Ballard, 1999; Spratling, 2017).

The concept of FF is to increase the goodness, or, the likelihood that neural activity suggests the input sample originates from the target training data distribution, for real data. In contrast, it aims to decrease the goodness for augmented negative data. Weights are updated with gradients of a local goodness function. Training is done layer by layer, removing the need to propagate errors backwards. Inputs to the network consist of a training sample combined with the label. This label is randomly changed to another label different from the original to generate negative training data. This entails that there are possibly many negative datasets for only one positive dataset.

2.2.1 General Network Layout

The architecture of a multi-layer perceptron $\mathcal{N}$ (and in this case our FFNN) consists of several layers and can be described as:

$$[L^0, L^1, ..., L^l]. \tag{2.1}$$

Here, L^k denotes the number of neurons in the layer with the number k. $k = 0$ denotes the input layer, l is the output layer, and the rest are hidden layers. The units of two successive layers are fully connected, meaning that each neuron in one layer is connected to every neuron in the subsequent layer. The pre-activation value of unit z_i in layer k is computed with the formula

$$z_i^k = \sum_{j=1}^{L^{k-1}} w_{ij}^k a_j^{k-1} + w_{i0}^k, \tag{2.2}$$

where w_{ij}^k is the weight from neuron j to i, a_j the activation of neuron j in layer $k-1$ and w_{i0}^k the weight of the bias for i from bias unit 0. This value is then passed through a nonlinear activation function λ:

$$a_i^k = \lambda(z_i^k), \tag{2.3}$$

which gives the activation a_i^k for neuron i in layer k. The activation function λ is usually the same for each neuron (except the output layer, where it is chosen based on the specific task) and can be for example a sigmoid, hyperbolic tangent or rectified linear unit.

2.2.2 Goodness

A local goodness function is needed to separate positive samples from negative samples. Hinton chooses the summed squared activities of the neurons in a layer after a Rectifier Linear Unit (ReLU) has been applied, as the goodness function. He gives two main reasons for selecting the summed squared activities. The first reason is that it has a simple derivative, making gradient calculation easier. The second reason is that layer normalization can be used to remove all signs of goodness. This thesis later explains why layer normalization is necessary. Another mentioned option for the goodness function is the negative summed squared activities. In line with Hinton, we choose the summed squared activities as goodness function.

Separation of real data and negative data is done by comparing the goodness to a certain threshold. The goal of learning is therefore to change the weights of the network such that the goodness is enhanced above this threshold for positive data, and decreased below this threshold for data that does not belong to the target class.

2.2.3 Updating the Weights

All the activations in a layer can be used to calculate the goodness for that specific layer given a normalized input vector from the previous layer. More explicitly, this goodness is used to precisely classify the input vector as positive or negative data, where the probability that this vector is labelled as being positive is approximated by the function

$$g^k = \sigma\left(\sum_{i=1}^{L^k} (a_i^k)^2 - \theta\right), \tag{2.4}$$

where g is the goodness for layer k, a_i^k the activity of neuron i and θ the threshold.

This goodness function can be transformed into an objective function for supervised learning tasks. The loss is separately but similarly calculated for the positive

$$\mathcal{L}_k^+ = \log\left(1 + \exp\left(-\sum_{i=1}^{L^k} (a_i^k)^2 + \theta\right)\right), \tag{2.5}$$

and negative forward pass

$$\mathcal{L}_k^- = \log\left(1 + \exp\left(\sum_{i=1}^{L^k} (a_i^k)^2 - \theta\right)\right), \tag{2.6}$$

where $\mathcal{L}_k^+$ ($\mathcal{L}_k^-$) is the positive (negative) pass loss, $\sum_{i=1}^{L^k} (a_i^k)^2$ the goodness and θ the threshold. Intuitively, positive data should produce high neural activation, while false data should induce low activity.

The local loss for layer k is then obtained by combining loss from the positive and negative samples

$$\mathcal{L}_k = \mathcal{L}_k^+ + \mathcal{L}_k^-. \tag{2.7}$$

The weights from layer j to i are updated by performing gradient descent on $\mathcal{L}$ with respect to the weights, that is

$$\Delta w_{ij}^k = -\mu \frac{\partial \mathcal{L}_k}{\partial w_{ij}^k}, \tag{2.8}$$

where μ is the learning rate. It is crucial to mention that the goodness calculation and gradient descent are performed locally, removing the requirement to use the chain rule and propagating errors backwards.

The FF procedure of two forward passes introduces a new hyperparameter that requires tuning, the threshold θ. Gandhi et al. (2023) analysed variations of this threshold. They found that a threshold equal to 0.3-0.5 times the number of neurons in a layer provided the best results. As elaborated on later, we perform hyperparameter tuning of this threshold, to discover for which value our FFNN performs most optimally on the task. It is important to note that this threshold causes the goodness of positive samples to increase continuously towards infinity, while negative samples converge to zero. This creates asymmetric gradients. We do not eliminate this asymmetry as done by Lee & Song (2023).

2.2.4 Network Architecture

Our initial network consists of one input layer with nine units, of which five correspond to the blackjack observation input sample and the other four to the one hot encoded action. The network further consists of four fully connected layers of 2000 neurons each (Figure 2.2).

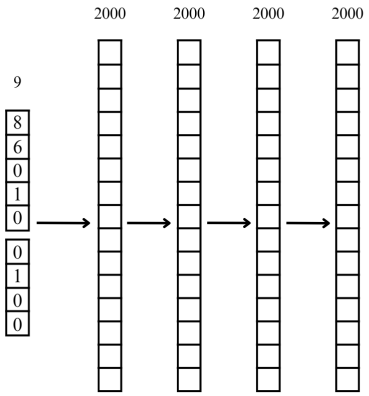


Figure 2.2: Forward-Forward Neural Network Model Architecture. In this Example, the Input Consists of Five Integers Representing the Blackjack Observation with Underneath Four Nodes Representing the One-Hot Encoded Label. The Numbers Above the Arrays Correspond to the Number of Neurons in that Layer.

The length of the hidden activity vector is normalized at each layer before being used as input. This is an important and necessary step. If this layer normalization (Ba et al., 2016) is omitted, it is simple for a hidden layer to separate positive from negative data based on the length of the activity vector from the previous layer, and it would not be necessary to extract new features. Normalization eliminates all the information that was utilized to assess the goodness in the previous layer and ensures it uses the orientation, i.e., relative activities, of the input vector to determine the goodness. Layer normalization is done by dividing the activity of a neuron by the summed squares of all activities:

$$a_{i,norm} = \frac{a_i}{\sqrt{\sum_{i=1}^{N} a_i^2}}, \tag{2.9}$$

where a_i is the activity of neuron i and N the total number of neurons in that layer. This normalization corresponds to L2-normalization. The network uses the Rectified Linear Unit (ReLU) activation function.

2.2.5 Training

The goal of training the network is to increase the layer activation for real data while reducing the layer activation for generated false data. Training is done layer by layer, each for a specified number of epochs.

Training is done in batches. At every epoch, a forward pass is performed for a batch of positive samples and the corresponding set of negative samples. The forward pass entails first normalizing the input vector, then multiplying it by the weight matrix and finally adding the bias vector. Lastly, the ReLU activation function is applied. Thereafter the sum of squared activations is calculated to obtain the goodness. The goodness is used to calculate the positive and negative loss as clarified in Section 2.2.2. The derivative of the combined loss functions with respect to the weights is used to perform gradient descent and an optimization step. This is repeated for every batch and epoch. After the last epoch, a final forward pass is done with all the input data through the layer. The layer's output vector resulting from this final forward pass is then used to train the next layer. This process is repeated for every layer.

2.2.6 Inference

There are two ways in which inference can be done by an FFNN, as described by G. Hinton (2022).

The first technique is the "accumulated goodness" method. After an FFNN has been trained, a sample desired to be classified is combined with the first label and passed as input y_0. The goodness for every hidden layer except the first one is then collected and summed. This is repeated with the same sample for all possible labels. The correct label $\hat{y}$ is given by

$$\hat{y} = \underset{y_j}{\operatorname{argmax}} \sum_{k=2}^{l} \sum_{i=1}^{L^k} (a_i^{k,y_j})^2, \tag{2.10}$$

where $a_i^{k,y}$ is the activation of neuron i in layer k for input y_j, L^k denotes the number of neurons in layer k, and l denotes the number of layers in the model. This thesis uses the accumulated goodness method by default.

For the second method, training of the FFNN is the same but inference is done by training a softmax linear classifier after the network has finished training. Every neuron in the hidden layers except the first hidden layer is connected to an output layer where the number of neurons equals the number of labels y. The first hidden layer is unused since it increases performance, according to Hinton. This results in $(L^2 + ... + L^l) * y$ trainable weights. First, a normal forward pass is performed with the same training observations given as input to the FFNN, together with a neutral label appended to the input data consisting of y nodes with a value of $1/y$. Second, the activations of the hidden layers as described above are passed as input to a softmax linear classifier that is subsequently trained for the same number of epochs and using the same batch size as the FFNN. The loss is calculated using the cross-entropy loss. Updating is done by performing a gradient descent optimization step using the Adam algorithm with a learning rate identical to the FFNN.

Inference is then accomplished by combining input with the neutral label, collecting the hidden layer activations, and obtaining the class for which the softmax activation is the highest. This activation can be viewed as the probability that a sample belongs to that class.

2.3 Reinforcement Learning

Our FFNN can be used on supervised learning tasks, i.e., problems with labelled input/output pairs. This is not favourable when trying to learn to play games in real-time, which are predominantly problems where the optimal action is unknown. To alleviate this issue, a combination of an FFNN and Reinforcement Learning (RL) is introduced, which removes the need for labelled input.

RL is a machine learning paradigm in which the goal is to have an agent learn an optimal policy for a specific task. A policy π is learned by performing actions a in an environment that represents the task the agent is trying to learn. The emphasis is on striking a balance between exploration and exploitation, maximizing long-term reward even when feedback is partial or delayed.

An RL environment is usually stated as a Markov Decision Process (MDP) (Howard, 1960). An MDP is a mathematical model for decision-making in states where the transition to the next state is solely decided by the present state, the action taken, and occasionally certain elements of randomness. It can be modeled as

$$(\mathcal{S}, \mathcal{A}, P_a(s, s'), R_a(s, s')), \tag{2.11}$$

were $\mathcal{S}$ is the set of states named the state space, $\mathcal{A}$ the set of actions named the action space, $P_a(s, s')$ the probability that action a in state s leads to state s', and $R_a(s, s')$ the reward immediately received for transitioning from state s to s'.

A popular RL algorithm is Q-learning (Watkins, 1989), it is a model-free algorithm that learns the action-value function: the value of a particular action in any state. It makes it possible to find the optimal action-selection policy for any finite MDP.

A problem arises when the state or action space is considerably large, making it infeasible to calculate the exact values of state-action combinations. In the field of deep reinforcement learning, neural networks can be used as function approximators to estimate these state-action values, i.e., Q-values. The goal is then to learn a set of parameters θ such that the neural network can assign the correct goodness to a state-action pair that maximises the expected reward, that is

$$Q(s, a; \theta) \approx Q^*(s, a). \tag{2.12}$$

Here, $Q(s, a; \theta)$ gives the state-action value for state s and action a with model parameters θ. $Q^*(s, a)$ is defined as

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right), \tag{2.13}$$

where π is the policy with the maximal expectation for all possible trajectories from state s with action a, t the timestep, γ the discount rate, and r_t the reward at time t. State-action values in the Q-learning algorithm are adapted based on the current belief, immediate reward, and expected future reward. This Q-learning update rule satisfies a Bellman equation and iteratively adjusts the action-value function (Dietterich, 2000). It is defined as

$$\begin{aligned} &Q(s_t, a_t) := Q(s_t, a_t) \\ &+ \mu\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t) - Q(s_t, a_t)\right), \end{aligned} \tag{2.14}$$

where $Q(s_t, a_t)$ is the state-action value of action a in state s at timestep t, μ the learning rate, r_t the received reward at t, γ the discount factor, and s_{t+1} the next state. $\gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t)$ is the expected future reward when the agent chooses the best action in the next state.

2.4 Deep-Q Forward-Forward Learning

For an FFNN to solve RL tasks without pre-generated labelled training data, we can combine deep reinforcement learning, specifically Q-learning, with an FFNN. The FFNN's ability to learn from positive as well as negative samples can be utilized and seamlessly combined with the positive and negative rewards an RL environment provides. These (expected) rewards allow the network to assess the goodness of possible actions in certain states. We name this value-based algorithm Deep Q-Forward-Forward Learning (DQFFL). The algorithm learns the value of the optimal policy independently of the agent's actions (off-policy) and does not rely on a model of the environment (model-free).

It is necessary to adapt the Q-learning update rule for DQFFL to work. It essentially involves changing the rule such that an FFNN can be used as a function approximator. The key of DQFFL is to decide the polarity of an action, i.e. whether an action is a good (positive) or bad (negative) one. This outcome can be used to train the FFNN. One significant obstacle is that we cannot simply use immediate rewards to decide whether an action is a good one. A positive reward indicates a good action, but this action may lead to a suboptimal outcome later. Fortunately, we can use the network's expected future goodness of an action for comparison. The comparison used to determine the polarity of an action takes shape as follows:

$$p(s_t, a_t, r_t, s_{t+1}) = \begin{cases} +, & \text{if } g(s_t, a_t, r_t, s_{t+1}) >= 0 \\ -, & \text{if } g(s_t, a_t, r_t, s_{t+1}) < 0 \end{cases}, \tag{2.15}$$

where

$$\begin{aligned} g(s_t, a_t, r_t, s_{t+1}) = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) \\ - Q(s_t, a_t, ; \theta), \end{aligned} \tag{2.16}$$

where s_t is the current state, a_t the action taken in that state, r_t the reward received for that action, γ the discount factor, θ the FFNN model parameters, θ^- the FFNN target model parameters, and $Q(s, a; \theta)$ the accumulated goodness given by an FFNN with parameters θ for state s and action a.

This function is essentially comparing the difference between the expected future goodness $r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$ and the current goodness $Q(s_t, a_t, ; \theta)$. if the expected goodness is higher than the current goodness, the action is better than expected and the FFNN is "underestimating" the goodness. If the inverse is true, the action is worse than expected and the network is "overestimating" the goodness.

A target model with a different parameter set alleviates the moving-target problem, which occurs when both the network's weights are updated every timestep, leading to oscillations. The target model's parameters are set to the model's parameters every x timesteps. This update frequency is another tunable hyperparameter.

2.4.1 DQFFL Training Procedure

During the exploration of the environment, an agent collects trajectories and saves them in

an Experience Replay (ER) buffer. A trajectory $(s_t, a_t, r_t, s_{t+1}, d)$ consists of the state, the chosen action, the received reward, the next state, and whether that state was terminal. For each update iteration, a random set of trajectories $\langle T \rangle$ is uniformly sampled from the ER buffer D. This allows off-policy learning for the algorithm and turns the RL problem into a supervised learning problem. This technique reduces the correlation between samples and thereby reduces the variance of updates. Furthermore, trajectories are possibly used in multiple weight updates, improving data efficiency. The size of the ER buffer is another hyperparameter requiring careful tuning.

The TD target, i.e., how good an action is, is calculated by summing the reward and the maximum expected goodness for the possible actions in the next state, multiplied by the discount factor. This TD target is equal to merely the immediate reward when the next state is terminal since no more rewards should be expected. As described in section 2.4, the polarity of an action is decided by subtracting the estimated goodness.

Finally, the FFNN is updated by generating training data based on whether an action was positive or negative. In either scenario, positive data is created by appending the action label to the state. If the polarity for a sample is positive (the action is considered good), the input for the corresponding negative forward pass is created by appending a random label that is not the "correct" action to the state. If the action is considered to be negative, the same sample from the positive pass is used for the negative pass. This cancels out the forward pass that mistakenly learned the wrong action. In summary, the FFNN learns from both good and bad actions (see Algorithm 2.1).

Algorithm 2.1 DQFFL Update Iteration, Let D be the Replay Buffer Containing Trajectory Samples T

Require: LENGTH(D) >= $batchsize$
  $\langle T \rangle \Leftarrow$ SAMPLE(D)
  for T_i in $\langle T \rangle$ **do**
    $TDtarget \Leftarrow reward_i$
    if $done_i \neq true$ **then**
      $TDtarget \Leftarrow TDtarget + \gamma$ MAXQ($nextstate_i$)
    end if
    $x_{pos,i} \Leftarrow (state_i, action_i)$
    if $TDtarget >=$ Q($state_i, action_i$) **then**
      while $action_i = sample(\mathcal{A})$ **do**
        $action_i \Leftarrow sample(\mathcal{A})$
      end while
    end if
    $x_{neg,i} \Leftarrow (state_i, action_i)$
  end for
  TRAIN(x_{pos}, x_{neg})

2.4.2 Action Selection

One issue still present in DQFFL is the exploration-exploitation dilemma. An RL agent should not simply pick the action it believes is best every time, since it may leave more optimal trajectories to the goal undiscovered. Only exploring by choosing random actions prevents the agent from converging to a certain strategy. The epsilon-greedy action selection strategy is chosen to strike a balance between exploration and exploitation.

Epsilon-greedy works by generating a random number between 0 and 1. If this number is lower than the current epsilon value ϵ, a random action a is picked from the action space $\mathcal{A}$. In the opposing case, the current best action according to the model is picked by doing a forward pass through the network. The epsilon value decreases over time, forcing the agent to pick more valuable actions towards the end of training, that is

$$\epsilon = \text{Max}\left(\epsilon_{min}, \epsilon - \frac{\epsilon_{max}}{n/2} * episode\right), \tag{2.17}$$

where ϵ is the current epsilon value, ϵ_{max} and ϵ_{min} are the start and end epsilon values respectively, n is the total number of episodes and *episode* is the current episode. The action is then decided by the policy π for a state s at timestep t as follows:

$$\pi(s_t) = \begin{cases} \underset{a}{\text{argmax}} Q(s_t, a; \theta) \text{ with probability } 1 - \epsilon \\ a \sim U(\mathcal{A}) \text{ with probability } \epsilon \end{cases}, \tag{2.18}$$

where Q returns the accumulated goodness of the FFNN with parameters θ, a is the action, and $\sim U(\mathcal{A})$ means sampled uniformly from the action space $\mathcal{A}$.

3 Method

3.1 Dataset

A *balanced dataset*, i.e., a dataset containing an equal number of observations for each action, is created to evaluate the performance of the FF algorithm. Firstly, every combination of observations is generated. One observation includes the player's hand value, dealer up-card, whether the player has a usable ace, whether doubling down is allowed, and whether splitting is allowed. Secondly, impossible combinations are removed and the optimal action, according to the basic strategy, is appended. This resulted in 1368 unique data points. 30.000 samples are subsequently sampled for each action, creating a dataset with 120.000 samples. The balanced data task tests whether an FFNN can learn all possible blackjack combinations. All possible generated combinations are present in the final dataset.

Simulated training data is generated by letting an agent play 75,000 episodes according to the basic strategy defined in Baldwin et al. (1956). This created 113,009 input samples each consisting of the observation and taken action. The corresponding negative dataset, necessary for the training of a forward-forward network, is created by substituting the action with a random incorrect action. An equal number of positive and negative samples is used for training.

3.2 Experiment Setup

3.2.1 Preprocessing

Training data for the balanced data task, consisting of samples with the environment's five observations and the best action, is divided using a random train/test split of 80/20.

Positive samples are then generated by appending one-hot-encoded labels of the correct action to their corresponding observations. The negative dataset is generated by randomly picking incorrect actions for each observation, and appending a one-hot-encoded label of these actions to the observation. This procedure is repeated for the simulated dataset.

3.2.2 Backpropagation Baseline

A neural network that is trained using the classic BackPropagation (BP) method is used to establish a baseline such that we can compare the performance of an FFNN to this. Hyperparameters are hand-picked and were only minimally adjusted using trial and error. The model is trained on the simulated data set in batches of size 64 for 100 epochs. The model's architecture comprises five input nodes, four hidden layers of 30 neurons each and a five-node output layer. It uses the ReLU activation function, and the cross-entropy loss is calculated for the network to be optimized with the Adam algorithm (Kingma & Ba, 2014). A learning rate μ of 0.001 is chosen.

The BP neural network's test loss is calculated on the test set. It is further evaluated by predicting optimal actions for the test set and calculating the accuracy. Lastly, a BP neural network is implemented into a blackjack agent which played for 75.000 episodes to asses obtained rewards on the game.

3.2.3 Balanced Data FFNN

Our initial model for the balanced dataset task consists of four fully connected layers of 2000 neurons each (as described above and consistent with Hinton's implementation). The Adam algorithm with a learning rate μ of 0.001 is used to optimize the network. The threshold θ is arbitrarily set at 0.5 and the batch size is 64. The network is trained without learning rate decay on the complete training set for 240 epochs.

3.2.4 Hyperparameter Tuning for FFNN

A machine learning algorithm's success does not solely depend on the training of the weights of the network, but also on hyperparameters that control the behaviour of this training. These hyperparameters are not learned during training but need to be tuned manually (Claesen & Moor, 2015). Examples of common hyperparameters for neural networks include the number of layers and neurons, the learning rate, and the batch size. FF includes another hyperparameter, the threshold. This threshold decides whether a data point is classified as positive or negative, thus crucial to the algorithm's performance. Larger differences between the threshold

and goodness result in different losses and gradient descends.

We use k-fold Cross-Validation (CV) to find the optimal model hyperparameters. This method has several advantages. Firstly, it reduces the variance of the accuracy estimate in comparison with a single train-test split. Additionally, it increases the reliability of the generalisation performance indication.

A grid search is performed on the model's hyperparameters. Each combination's performance is estimated using k-fold CV. Combinations are made from a pre-specified range of candidates. For each combination, the training data is divided into k parts of approximately equal size, hence the name k-fold. One fold is left out and designated as the validation set. The remaining $k-1$ folds are used to train the model. Predictions are made and a validation risk (the test error between the predictions and validation set) is calculated using the validation set. This process is repeated for each fold. The average risk of each validation set is the model's validation risk. The best model is the one with the minimal validation risk and therefore, the best generalisation capability.

It is important to mention that CV in our particular experiment is not necessarily used as intended, i.e., to prevent overfitting, but to find the combination of hyperparameters that has the potential to reach the highest accuracy. Blackjack only features a limited number of observable combinations with pre-determined best actions. The goal is therefore to train an FFNN with the right combination of parameters that enable the model to learn every combination encountered by the basic strategy agent during gameplay.

All hyperparameters manipulated with CV are found in Table 3.1. Trial-and-error and literature reviews of papers studying FF were initially used to find suitable ranges for hyperparameters. The explored values can be found in Appendix A. This subsequently resulted in 80 different evaluated models.

The model with 4 hidden layers of 2000 neurons each, threshold $\theta = 10$, learning rate $\mu = 0.01$, a batch size B of 64, and without learning rate decay, had the lowest validation risk and therefore best hyperparameters for generalization. This model was subsequently trained on the complete training set for 240 epochs. Additionally, it is adapted to an FFNN agent which returns the best action for an observation according to the accumulated goodness method. This agent is used to evaluate blackjack gameplay performance for 75.000 episodes.

Table 3.1: Hyperparameters of the FF Algorithm that are Optimized with Cross-Validation

Hyperparameter	Symbol
Model Architecture	$\mathcal{N}$
Threshold	θ
Learning Rate	μ
Batch Size	B
LR Decay	$\mu(e)$

3.2.5 FFNN with Linear Classifier

An FFNN with the synaptic weights, architecture, and hyperparameters found with cross-validation is used to test the linear classifier inference method. The softmax linear classifier is trained for 240 epochs in batches of equal size to the best batch size hyperparameter discovered in cross-validation. The network's weights are optimized using the Adam optimizer with a learning rate identical to the learning rate found for the FFNN in the CV scheme. This learning rate was chosen since the trained FFNN appeared to perform well on this task.

3.2.6 DQFFL

A DQFFL agent is trained in two toy game reinforcement learning environments to test the algorithm's performance. The first is the blackjack environment also used to generate FFNN training data, allowing for comparison between the supervised and reinforcement learning methods.

An episode of the experiment starts with an observation of the environment. The subsequent action selected by the agent's action selection policy is used to take a step in the discrete blackjack and frozen lake environments. This results in a new observation, a reward and a boolean value indicating whether the episode has terminated. This trajectory T is thereafter appended to the Experience Replay (ER) buffer. The DQFFL algorithm then utilizes the trajectories in the ER buffer to update the agent (see Section 2.4.1). If an episode ends, the agent's epsilon is updated according to the decay rule.

DQFFL agents in both environments have multitudinous hyperparameters requiring manual tuning. DQFFL, like all deep reinforcement learning algorithms, is highly sensitive to changes in these hyperparameters (Eimer et al., 2023). As such, the tuning of the hyperparameters took numerous runs and manual adjustments.

One blackjack run provides 4000 episodes for the DQFFL agent to learn the action-value function and maximize reward. An FFNN with three hidden layers of 1000 neurons each, a threshold θ of 0.5 and a learning rate μ of 0.001 is used as the agent's model and TD target. All hyperparameters used in DQFFL are summarized in Appendix B.

The agent used in the frozen lake environment utilized a three hidden-layer 100-neuron network with $\theta = 0.5$ and $\mu = 0.001$, one run contained 2000 episodes. Both experiments were repeated five times to achieve precise and reliable results.

3.2.7 Evaluation

The models are evaluated on the test sets split at the beginning and kept apart during training to provide the most accurate representation of generalization power. The classification capability of the FFNN model on balanced and simulated data (accumulated goodness method and linear classifier method), the BP-trained network, and the final DQFFL blackjack agent's model are reported by accuracy. The optimal actions according to the basic strategy feature a class imbalance. Confusion matrices are examined to better analyze and report model performance.

Training accuracy is measured to receive insights on model convergence compared to training duration. This is done for the last layer of each FFNN by summing true negatives and true positives, i.e. the number of positive samples with a goodness above the threshold and negative samples with a goodness below the threshold, divided by the total number of samples.

Performance of the basic strategy, FFNN and BP blackjack agents is measured by the average reward over the played 75.000 episodes and compared to a random action policy agent. DQFFL learning capability is analysed by plotting the mean and standard deviation over five runs against the episodes.

Different seeds are used for testing and training to reduce the possibility of overfitting by manual hyperparameter tuning of algorithms.

4 Results

4.1 FFNN

Our results show that FFNNs can accurately learn and predict actions for the supervised learning blackjack task. Our FFNN trained and tested on the balanced data task achieved an accuracy of 99.97% after training for 240 epochs (see Figure C.1a). Accuracy is calculated employing the accumulated goodness method described in Section 2.2.6. The last layer's training accuracy shows convergence after about 50 epochs (see Figure 4.1). Note the scale on the accuracy plot, accuracy at epoch 1 already starts at a significant level. This can be attributed to the three layers previously trained in this run.

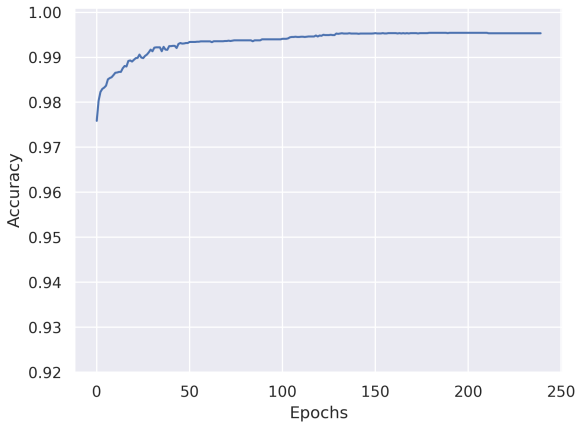


Figure 4.1: Last Layer Training Accuracy of the FFNN on the Balanced Data Task

Five-fold Cross-Validation (CV) with the parameter combinations disclosed in Appendix A revealed the lowest validation risk for a network of four layers of 2000 neurons each, a threshold θ of 10 and a learning rate μ of 0.001, with a final validation risk of 0.00094 ± 0.00078 (SE). This hyperparameter combination therefore had the best generalisation potential and achieved a test accuracy of 100% on the complete training set. Despite the class imbalance of the simulated dataset, results show no difference in predictions based on class (see Figure C.1b). The last layer's model weights converged af-

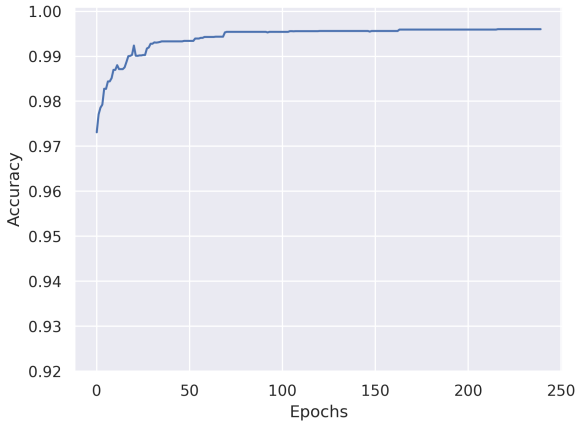


Figure 4.2: Last Layer Training Accuracy of the FFNN on the Simulated Data Task

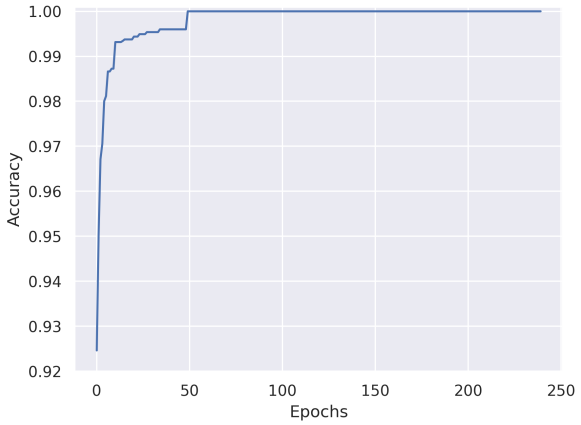


Figure 4.3: Training Accuracy for the Softmax Linear Classifier

ter approximately 70 epochs (see Figure 4.2). Remarkable are the top ten models from CV, which all have a threshold of either 5 or 10 and similarly low validation risks but differ in architecture and learning rate (see Table A.2). The second and third-best models of 500 and 100 neurons per layer have a validation risk of 0.0017 ± 0.0014 (SE) and 0.0020 ± 0.00041 (SE) respectively. The tested learning rate of 0.03 implemented by G. Hinton (2022) tends to be too high for our models. The best-performing small network with 30 neurons per layer, a threshold of 0.5 and a 0.01 learning rate, performed relatively well with a final test accuracy of 83.01%. It, however, had insufficient capacity to correctly predict the less frequently occurring actions 2 and 3 (see Figure C.2).

The softmax linear classifier that is subsequently trained on all but the first hidden layer activations of the optimal model achieved an accuracy of 100%. It converged after only 50 epochs (see Figure 4.3).

Our backpropagation-trained multi-layer perceptron attained a 3.18e−4 cross-entropy test loss and was perfectly able to predict actions based on the test data samples, corresponding to 100% accuracy (see Figure C.3).

Playing a perfect strategy on blackjack for 75,000 episodes yields an average reward of -0.0069 ± 0.0041 (SE). The backpropagation network achieved a reward of -0.012 ± 0.0041 (SE). It must be mentioned that these rewards were obtained on two different runs (see Figure C.4). FFNN performed comparably, acquiring an average reward of -0.013 ± 0.0041 (SE) (see Figure C.5). All the results are visible in Table 4.1.

Table 4.1: Mean Achieved Reward by Evaluated Blackjack Agents Across 75,000 Episodes

Agent	Average Reward ($\bar{r}$) ± SE
Random	-0.5 ± 0.005
Basic Strat.	-0.007 ± 0.004
FFNN	-0.01 ± 0.004
Backprop.	-0.01 ± 0.004

4.2 DQFFL

We reviewed the rewards obtained over five runs for our DQFFL agents to see if an FFNN can be combined with reinforcement learning to learn to play games without pre-generated training data or known optimal actions. The mean reward over these five runs for 4000 episodes has been plotted in Figure 4.4 using a rolling mean of 100 episodes. It is visible that an agent starts at an average reward of -0.5 and gradually moves up to an average reward fluctuating around -0.1 after 2000 episodes. Although learning is unstable, DQFFL performs better than a random agent in the blackjack environment (see Figure C.6). The FFNN model with the final weights from the last run is evaluated on the simulated data test set, achieving an accuracy of 68.42%. A (normalized) confusion matrix reveals the difficulties the model has with predicting the, least occurring in the blackjack game, actions 2 and

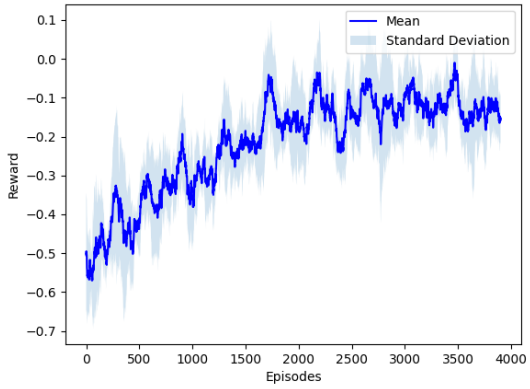


Figure 4.4: Five Run Mean Reward for the DQFFL Algorithm on Blackjack, Plotted with a 100-episode Rolling Mean

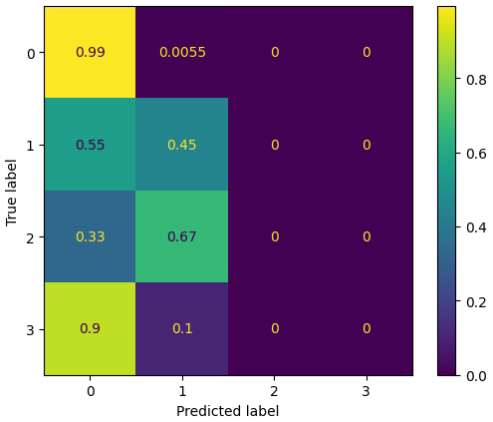


Figure 4.5: Normalized Confusion Matrix of DQFFL Trained FFNN on Simulated Test Data

3 (see Figure 4.5). It is, however, a snapshot of the model at one timestep in one run and does not fully reflect its capabilities.

More successful is the DQFFL algorithm's performance in the frozen lake environment. An average reward of 1.0 per episode, the maximum possible reward, is reached after 1000 episodes, meaning the DQFFL agent prosperously learned the path to the goal (see Figure 4.6).

5 Discussion

5.1 Interpretation of Results

The commencing supervised learning experiment on our balanced blackjack basic strategy data confirms the memory consolidation ability of FFNNs as initially discovered by G. Hinton (2022). The accuracy is notable and the model learns within a reasonable number of epochs. Cross-validation of Forward-Forward (FF) hyperparameters reveals that the threshold and learning rate are important factors regarding network performance, while model architecture, for this task at least, is of less significance. Models with a higher threshold seemed to perform better. The achieved accuracy on blackjack gameplay acquired training data reveals that our FFNN can learn and accurately classify samples even when a sizeable class imbalance is present. These experiments achieved a satisfactory result

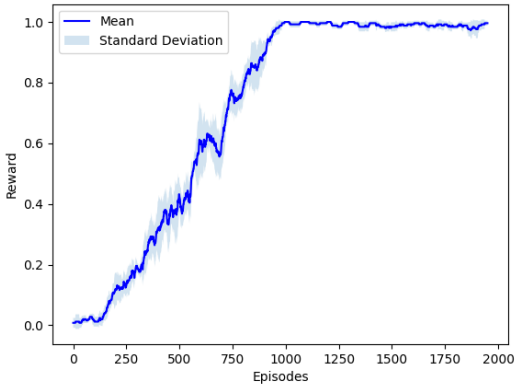


Figure 4.6: 50-episode Rolling Mean of the Five Run Average Reward for DQFFL on the Frozen Lake Environment

even without optimizations like learning rate decay (G. Hinton, 2022) or a symmetric loss function (Lee & Song, 2023). Therefore, the experiments on the balanced and simulated data tasks show support for our hypothesis that an FFNN can learn to play simple games, given known optimal actions.

The lowest validation risk on the simulated data task was achieved by our large network of 2000 neurons per layer. However, the difference with a four-layer 100-neuron network was minimal. This shows that even small FFNNs can have sufficient non-linear separability. More importantly, this means that an FFNN can be useful in environments where memory capacity is severely limited.

Both the accumulated goodness and softmax linear classifier inference methods realize equal accuracy. This is contrary to earlier research that found worse performance for the classification method. (G. Hinton, 2022; Brenig & Timofte, 2023; Scodellaro et al., 2023). However, earlier trials where the optimal model did not achieve 100% accuracy had lower test accuracy for the linear classification method in comparison with accumulated goodness. It is worth mentioning that linear classifier inference is four times as fast because it only requires one forward pass for prediction instead of one for each of the four labels. Furthermore, while the extra output layer that is required to construct a linear classifier needs additional training, convergence is reached relatively quickly.

FFNN memory consolidation ability is confirmed by the performance of the FFNN agent on blackjack, whose run yielded a comparable reward to basic strategy and in line with the mathematical expectation of the reward calculated by Baldwin et al. (1956).

The performance of our introduced Deep-Q Forward-Forward Learning (DQFFL) algorithm shows promising results on the blackjack reinforcement learning task. The agent achieves a better average reward than a purely random agent. Still, performance is not comparable to gameplay according to basic strategy and learning is unstable. The highly fluctuating mean reward across episodes can be partly explained by the stochastic nature of blackjack. Even optimal actions often lead to negative rewards when for example the dealer has a higher total or an agent receives unlucky cards. Would this happen for a longer consecutive sequence, it is possible that a DQFFL agent associated certain optimal actions with a negative reward. The algorithm would then update the FFNN with believed-to-be negative samples that were positive. A second reason for learning instability could be the class imbalance of the basic strategy optimal actions. The DQFFL algorithm was not able to successfully predict the double-down and split actions, even though they are necessary to obtain the best possible expected reward (Baldwin et al., 1956).

DQFFL's performance in the Frozen Lake environment is more impressive, reaching the goal each episode after about 1000 episodes. The epsilon-greedy strategy's epsilon value also reaches its minimum after half the total number of episodes (1000). This means the algorithm potentially learns the path to the goal faster. Improved hyperparameter tuning could speed up convergence even more. The mean that is not exactly one every episode after half the episodes can be attributed to the fact that there still is a small chance of performing a random action, due to the epsilon-greedy strategy.

In summary, both the DQFFL experiments and especially the frozen lake tests show strong support in favour of the hypothesis that an FFNN can be used as a function approximator in reinforcement learning to learn to play simple games. The DQFFL algorithm demonstrates the intriguing ability of such an agent to learn on the fly from both positive and negative experiences.

5.2 Limitations

Training on and solving learning tasks in this paper is limited to simple toy problems that lack complexity compared to other games and are relatively easily solved by established techniques. This is demonstrated by our backpropagation-trained network that achieves perfect classification accuracy after fewer epochs and uses a smaller network compared to our FF implementation. Nevertheless, comparison based on network size and convergence speed is difficult since the energy and memory needed to train these networks differ. On top of that, the intent of FF is not to replace backpropagation neural networks but to provide an efficient alternative that overcomes limitations like the biological implausibility and memory requirements of backpropagation. This thesis shows that this is indeed possible.

Inference by accumulated goodness is a simple

and effective method, but it becomes computationally more expensive when the number of labels increases. Every extra classification label requires one more inference calculation. A trained softmax linear classifier provides an elegant solution in this case.

DQFFL is a new method with only a few preliminary trials completed. Learning is unstable, this is especially visible in the blackjack environment. DQFFL also suffers from high sensitivity to hyperparameter changes. Extensive hyperparameter investigation and tuning could improve the convergence and stability of the agents. However, this thesis aims to test whether an FFNN can learn to play simple games given unknown optimal actions, something that is demonstrated in the two reinforcement learning environments.

5.3 Future Research

Potential improvements to our FFNN that could speed up convergence include differently encoding network inputs (Jia & Chua, 1993), a different goodness function or generating negative data labels based on the network's goodness instead of random (G. Hinton, 2022). These were left out since accuracy was sufficiently high and not considered to be in the scope of this thesis.

The question remains whether FF will replace the widely used backpropagation algorithm (Rojas, 1996). Regardless, the local adaptation rule of FF provides valuable insights and research possibilities in the field of neuromorphic computing, the field dealing with biologically plausible learning algorithms (Schuman et al., 2017).

The introduction of DQFFL uncovers a wide variety of potentially interesting research subjects. A fascinating characteristic of DQFFL is its ability to learn offline. An adaptation of DQFFL can be investigated that directly learns from encountered experiences and stores negative experiences for a later offline pass. How much this process resembles learning during sleep may provide new insights into the human brain. Furthermore, research suggests that people also learn from positive and negative experiences (Cox et al., 2015; Namburi et al., 2015; Paton et al., 2006). Whether this learning resembles the process of DQFFL is worth further investigating. Lastly, like FF, DQFFL could be useful in applications where memory or power is greatly limited, like microcontrollers (De Vita et al., 2023) or mortal computation (G. Hinton, 2022).

The stability of DQFFL is a main issue that requires more experimentation and research to be solved. A learning rate or weight decay scheme is a potential solution. Another possible solution is to update the network less frequently as the model converges. This prevents positive actions from being classified as bad when the model goodness estimation gets close to and sometimes overshoots the immediate reward plus the expected future reward.

Many enhancements of DQFFL are possible, e.g., refining the function that estimates whether an action is good or bad, or learning directly from encountered trajectories instead of storing them in the experience replay buffer. In general, DQFFL requires validation on a wide variety of tasks and more complex (video) games to prove its practical value in the field of machine learning and neuromorphic computing.

6 Conclusion

In this thesis, an FFNN was trained to predict optimal actions in the game of blackjack. We showed satisfactory results on a supervised learning task with a balanced dataset and a dataset created by gameplay based on the optimal blackjack strategy. Two methods of inference, namely the accumulated goodness and linear classifier methods, were tested and achieved high accuracy, positively answering the research question of whether an FFNN can be used to learn to play simple games, given known optimal actions. The FFNN performs comparably to our backpropagation baseline without the need for the storage of all activations, a differentiable forward pass or the biologically implausible backward propagation of error derivatives.

This research further contributes to the field of machine learning by introducing and testing a novel brain-inspired algorithm, coined Deep-Q Forward-Forward Learning (DQFFL), that combines an FFNN with reinforcement learning. Although there are substantial improvements possible, DQFFL shows, based on two different environments, that it is possible to use an FFNN to learn to play simple games without known optimal actions.

Acknowledgements

We thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Hábrók high-performance computing cluster.

References

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Baldwin, R. R., Cantey, W. E., Maisel, H., & McDermott, J. P. (1956). The optimum strategy in blackjack. *Journal of the American Statistical Association*, *51*(275), 429–439.

Brenig, J., & Timofte, R. (2023). A study of forward-forward algorithm for self-supervised learning. *arXiv preprint arXiv:2309.11955*.

Claesen, M., & Moor, B. (2015). Hyperparameter search in machine learning. *ArXiv*, *abs/1502.02127*.

Cox, S. M. L., Frank, M., Larcher, K., Fellows, L., Clark, C., Leyton, M., & Dagher, A. (2015). Striatal d1 and d2 signaling differentially predict learning from positive and negative outcomes. *NeuroImage*, *109*, 95-101. doi: 10.1016/j.neuroimage.2014.12.070

Crick, F. (1989). The recent excitement about neural networks. *Nature*, *337*(6203), 129–132.

De Vita, F., Nawaiseh, R. M., Bruneo, D., Tomaselli, V., Lattuada, M., & Falchetto, M. (2023). μ-ff: On-device forward-forward training algorithm for microcontrollers. In *2023 ieee international conference on smart computing (smartcomp)* (pp. 49–56).

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, *13*, 227–303.

Eimer, T., Lindauer, M., & Raileanu, R. (2023). Hyperparameters in reinforcement learning and how to tune them. , 9104-9149. doi: 10.48550/arXiv.2306.01324

Epstein, R. A. (2012). *The theory of gambling and statistical logic*. Academic Press.

Gandhi, S., Gala, R., Kornberg, J., & Sridhar, A. (2023). Extending the forward forward algorithm. *arXiv preprint arXiv:2307.04205*.

Gardner, D. (1993). *The neurobiology of neural networks*. MIT Press.

Giampaolo, F., Izzo, S., Prezioso, E., & Piccialli, F. (2023). Investigating random variations of the forward-forward algorithm for training neural networks. In *2023 international joint conference on neural networks (ijcnn)* (pp. 1–7).

Gutmann, M., & Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 297–304).

Hinton, G. (2022). The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345*.

Hinton, G. E., Sejnowski, T. J., et al. (1986). Learning and relearning in boltzmann machines. *Parallel distributed processing: Explorations in the microstructure of cognition*, *1*(282-317), 2.

Howard, R. A. (1960). Dynamic programming and markov processes.

Jia, J., & Chua, H. (1993). Neural network encoding approach comparison: an empirical study. *Proceedings 1993 The First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*, 38-41. doi: 10.1109/ANNES.1993.323087

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Lee, H.-C., & Song, J. (2023). Symba: Symmetric backpropagation-free contrastive learning with forward-forward algorithm for optimizing convergence. *arXiv preprint arXiv:2303.08418*.

Lillicrap, T., Santoro, A., Marris, L., Akerman, C., & Hinton, G. (2020). Backpropagation and the brain. *Nature Reviews Neuroscience*, *21*, 335 - 346. doi: 10.1038/s41583-020-0277-3

Namburi, P., Beyeler, A., Yorozu, S., Calhoon, G. G., Halbert, S., Wichmann, R., ... Tye, K. (2015). A circuit mechanism for differentiating positive and negative associations. *Nature*, *520*, 675 - 678. doi: 10.1038/nature14366

Ororbia, A., & Mali, A. (2023). The predictive forward-forward algorithm. *arXiv preprint arXiv:2301.01452*.

Paton, J. J., Belova, M. A., Morrison, S. E., & Salzman, C. (2006). The primate amygdala represents the positive and negative value of visual stimuli during learning. *Nature*, *439*, 865-870. doi: 10.1038/nature04490

Rao, R. P., & Ballard, D. H. (1999). Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, *2*(1), 79–87.

Rojas, R. (1996). The backpropagation algorithm. , 149-182. doi: 10.1007/978-3-642-61068-4_7

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Schuman, C. D., Potok, T., Patton, R., Birdwell, J., Dean, M. E., Rose, G., & Plank, J. (2017). A survey of neuromorphic computing and neural networks in hardware. *ArXiv*, *abs/1705.06963*.

Scodellaro, R., Kulkarni, A., Alves, F., & Schröter, M. (2023). Training convolutional neural networks with the forward-forward algorithm. *arXiv preprint arXiv:2312.14924*.

Skinner, G., & Walmsley, T. (2019). Artificial intelligence and deep learning in video games a brief review. In *2019 ieee 4th international conference on computer and communication systems (icccs)* (pp. 404–408).

Spratling, M. W. (2017). A review of predictive coding algorithms. *Brain and cognition*, *112*, 92–97.

Stork, D., & Hall, J. (1989). Is backpropagation biologically plausible? *International 1989 Joint Conference on Neural Networks*, 241-246 vol.2. doi: 10.1109/IJCNN.1989.118705

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G. d., Deleu, T., ... Younis, O. G. (2023, March). *Gymnasium.* Zenodo. Retrieved 2023-07-08, from `https://zenodo.org/record/8127025` doi: 10.5281/zenodo.8127026

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

A FFNN Hyperparameters

Table A.1: Evaluated Hyperparameter Values in Cross-Validation of the FFNN on the Simulated Data Task

Hyperparameter	Symbol	Evaluated Values
Model Architecture	$\mathcal{N}$	$[9, 30, 30, 30, 30]$, $[9, 100, 100, 100, 100]$, $[9, 500, 500, 500, 500]$, $[9, 2000, 2000, 2000, 2000]$,
Threshold	θ	$0.1, 0.5, 1, 5, 10$
Learning Rate	μ	$0.0001, 0.001, 0.01, 0.03$
Batch Size	B	64
LR decay	$\mu(e)$	False
No. of Epochs	E	240
No. of folds	k	5

Table A.2: Top 10 Hyperparameter Configurations with Lowest Validation Risk

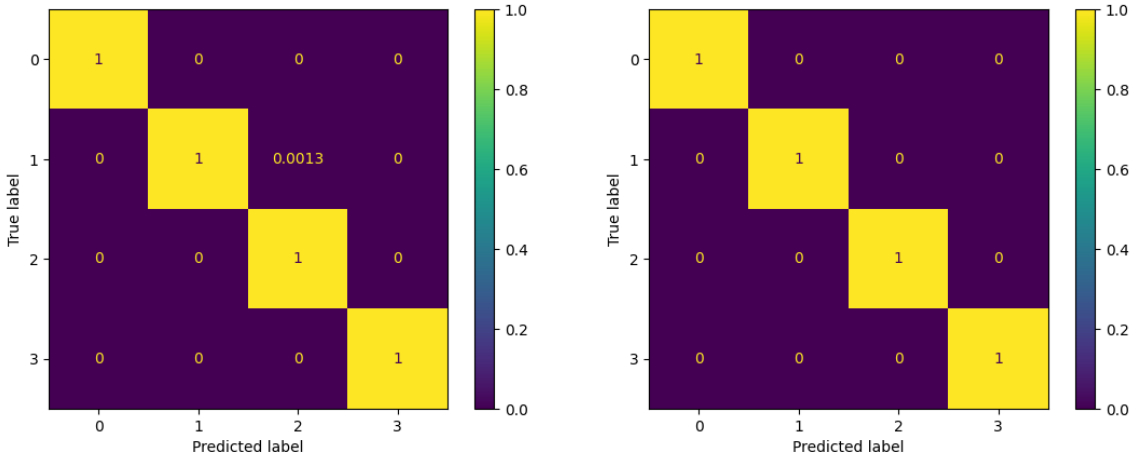
	Model Architecture ($\mathcal{N}$)	Threshold (θ)	Learning Rate (μ)	Validation Risk $\pm$ SE
1	[9, 2000 * 4]	10	0.001	0.0009 $\pm$ 0.0008
2	[9, 500 * 4]	5	0.001	0.002 $\pm$ 0.001
3	[9, 100 * 4]	10	0.01	0.002 $\pm$ 0.0004
4	[9, 500 * 4]	10	0.001	0.002 $\pm$ 0.0008
5	[9, 2000 * 4]	5	0.001	0.003 $\pm$ 0.002
6	[9, 2000 * 4]	10	0.0001	0.004 $\pm$ 0.0008
7	[9, 500 * 4]	5	0.001	0.004 $\pm$ 0.0010
8	[9, 2000 * 4]	5	0.0001	0.005 $\pm$ 0.0005
9	[9, 500 * 4]	5	0.01	0.006 $\pm$ 0.002
10	[9, 100 * 4]	5	0.01	0.007 $\pm$ 0.002

B DQFFL Hyperparameters

Table B.1: Hyperparameters for the DQFFL Reinforcement Learning Agents on the two Experimented Environments

Hyperparameter	Symbol	Blackjack	Frozen Lake
FFNN Architecture	$\mathcal{N}$	[9,1000,1000,1000]	[20,100,100,100]
FFNN Threshold	θ	0.5	0.1
FFNN Learning Rate	μ	0.001	0.001
Episodes	E	4000	2000
Epsilon Max	ϵ_{max}	1.0	0.99
Discount Factor	γ	0.99	0.9
Batch Size	B	32	32
TD Target Update	τ	50	10
Buffer Size	N	10000	1000

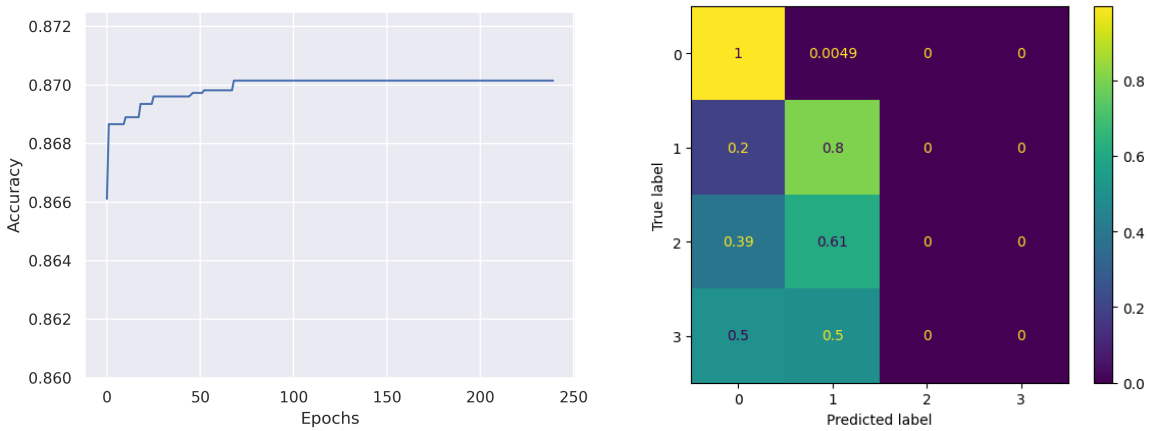
C Results



(a) Balanced Data Task

(b) Simulated Data Task (FFNN After CV)

Figure C.1: Normalized Confusion Matrices for FFNN Model predictions on the Balanced and Simulated Data Tasks



(a) Last Layer Training Accuracy

(b) Model Predictions

Figure C.2: Last Layer Training Accuracy and Confusion Matrix for the Best Performing Small Model on the Simulated Data Task

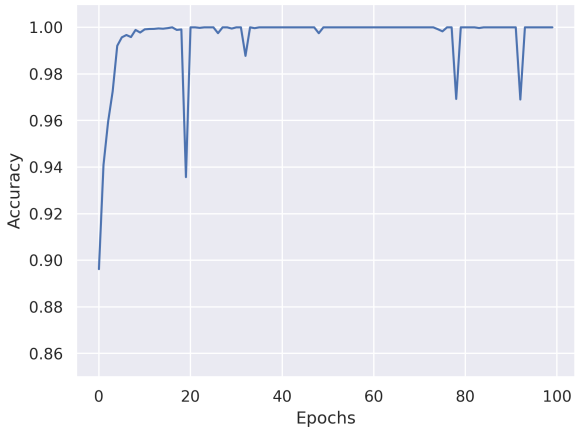


Figure C.3: Training Accuracy for the Backpropagation Network on the Simulated Data Task

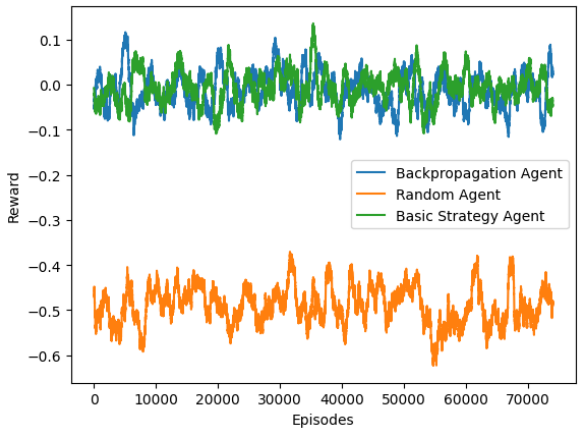


Figure C.4: 1000-episode Rolling Mean Reward for Backpropagation and Random Agents on Blackjack

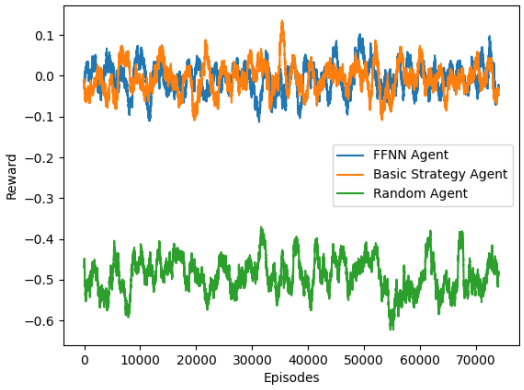


Figure C.5: 1000-episode Rolling Mean Reward for FFNN, Basic Strategy and Random Agents on Blackjack

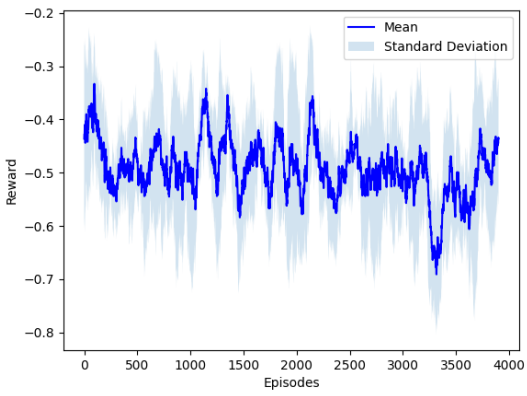


Figure C.6: Five Run Mean Reward for a Random Agent in the Blackjack Environment, Plotted with a 100-episode Rolling Mean