UNIVERSITY OF CALIFORNIA
SANTA CRUZ

RECURRENT NEURAL NETWORK MORTALITY PREDICTION AND DEEP REINFORCEMENT LEARNING IN MANUFACTURING SCHEDULING

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

PHYSICS

by

David C. Jenkins

December 2021

The Dissertation of David C. Jenkins
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Recurrent Neural Network Mortality prediction and Deep Reinforcement Learning in Manufacturing Scheduling

by

David C. Jenkins

Neural networks are a powerful tool which have shown usefulness in a number of challenging domains. We have developed novel methods which incorporate neural networks in two important domains. The first domain considered is in mortality prediction in intensive care units. For this problem, a model involving long short term memory recurrent neural networks was developed. This model was trained and tested using the Medical Information Mart for Intensive Care (MIMIC iii) data set. Our method generates a mortality trajectory after each input health measurement consisting of a sequence of mortality predictions. Through testing we demonstrate that our model is capable of providing effective mortality predictions in this data set. The next domain considered is production scheduling in semiconductor manufacturing facilities. We developed two methods for training scheduling policies that utilize deep reinforcement learning. The first involves a Deep Q-Network, and the second is a novel method we developed known as the Predictron Deep Q-Network. A factory simulation model was developed. Our methods were demonstrated to outperform common industry methods on a set of simulated factory environments based on real world systems.

To my parents,

Peter and Robin,

for being supportive of my education.

Acknowledgments

The text of this dissertation includes a reprints of the following material which may be published: The contents of chapter 4 ”PDQN - A Deep Reinforcement Learning Method for Planning with Long Delays: Optimization of Manufacturing Dispatching” have been submitted to the International Conference on Learning Representations 2022. The co-author Ram Akella listed in this publication submission directed and supervised the research which forms the basis for the dissertation.

I David Jenkins, the PhD candidate, was the primary contributor to this work. I developed the simulation model, and majorly contributed to the development of the Deep Reinforcement Learning models, their training and testing to gather results, as well as the writing of the paper submission.

Chapter 1

Introduction

Over recent years methods involving artificial neural networks have seen a surge in popularity due to their effectiveness in a number of previously challenging domains. Neural network based methods are now at the cutting edge of image, text, and speech processing as well as a number of games such GO, chess and Shogi. Due to their remarkable effectiveness there exists a great opportunity for the development and refinement of these techniques to additional real world challenging problems. In this dissertation we focus in particular on problems which involve sequential prediction and decision making.

Two major domains are considered in this dissertation for the application and development of novel neural network based methods. The first is in mortality prediction in intensive care units. Health care related costs approach 18% of gross domestic product in the United States [17]. Medical care also has significant impact on the health and quality of life of individuals. Health care in intensive care units is of particular importance due to the relatively high severity of illness among patients in such units. To help address this issue and allow for more effective allocation of care, we have developed a method for mortality prediction in intensive care units based upon long short term memory (LSTM) recurrent neural networks.

Due to the sequential nature of health and vital measurements in intensive care units, recurrent neural networks and in particular LSTM models may be especially effective. This method generates a mortality trajectory consisting of a sequence of mortality probability estimates covering a rolling window after each electronic health record is input. Using the Medical Information Mart for Intensive Care (MIMIC iii) data set, we were able to train our model to generate these predictions based upon sequences of recorded numerical health measurements. Our method was validated on a portion of the data that was allocated for testing and demonstrated to be effective based on the recorded area under receiver operator curve (AUROC).

The other domain considered is production scheduling in semiconductor manufacturing facilities. Such facilities involve long sequences of complex operations in order to produce semiconductor devices. These operations involve expensive machinery so efficient utilization of resources is of significant importance. Such production scheduling tasks can be modelled dynamically as a sequence of dispatching decisions which determine at each time step the priority of distinct parts for processing on limited resources.

In recent years deep reinforcement learning techniques have shown cutting edge performance on a number of sequential decision making tasks such as playing GO, chess, and shogi. Due to this remarkable performance there exists a great opportunity for application and development of these techniques for this domain. Due to the complexity of this problem traditional operations research methods may be computationally intractable, therefore relatively simple dispatching policies are often used in the industry.

By working directly with the semiconductor device manufacturer Western Digital Corporation, we developed a factory simulation model. We then developed two distinct methods for production scheduling policy generation involving deep reinforcement learning. The first involves use of Google DeepMind's Deep Q-Network [13]. The next is a novel method we developed which expands upon the Deep Q-Network by combining it with another method from DeepMind, the predictron [18].

We trained and tested these methods on a set of simulated factory systems based on real world environments from Western Digital Corporation. Our methods were trained to reduce the lateness of completion of parts with respect to their due dates. Our results demonstrated that our methods outperform the commonly used dispatching policies on this task in these simulated environments.

In chapter 2, background information is introduced for neural network methods and reinforcement learning. This includes description of LSTM recurrent neural networks and deep reinforcement learning methods which are used in the subsequent chapters. In chapter 3 the work on LSTM methods for mortality prediction in intensive care units is described, along with the gathered results. In chapter 4 the work on deep reinforcement learning for semiconductor manufacturing systems is covered.

Chapter 2

Background

2.1 Classes of Machine Learning Algorithms

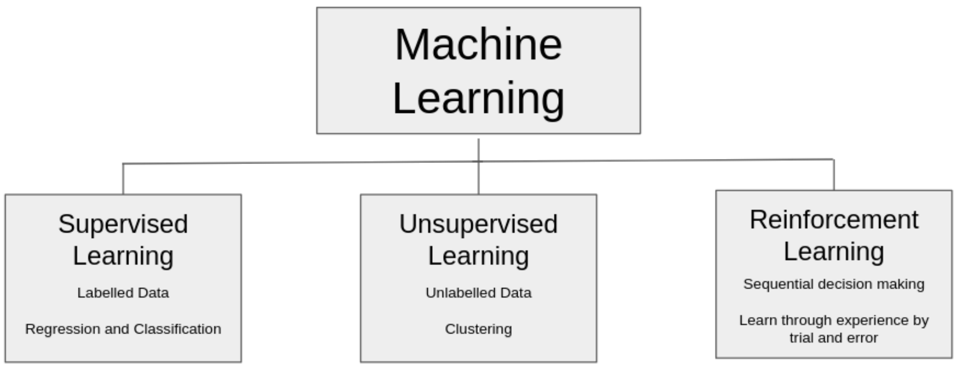

Figure 2.1: Machine Learning can be classified into three main categories. Supervised Learning uses learns from labeled data, Unsupervised Learning learns from unlabeled data, and reinforcement learning learns policies for sequential decision making tasks through trial and error.

Machine Learning can generally be classified into three main categories. These categories are Supervised Learning, Unsupervised Learning, and reinforcement learning. Supervised Learning is generally applied in situations in which there exists data sets of both the input data values as well as their corresponding correct output value labels. By applying machine learning techniques one can generate a function approximation between the inputs and output labels. Using this function approximation one could then generate predictions of the outputs corresponding to new previously unseen inputs. Supervised learning can also be subdivided into classification, where the output corresponds a discrete set of categories, and regression, in which continuous variables are estimated.

Unsupervised learning however, refers to problems in which the correct output labels are not know. In such methods, outputs are generated without the use of given output labels from the training data. Applications of unsupervised learning include clustering of samples into distinct classes data based upon relative values of their features.

Reinforcement learning is generally considered to be a distinct category from Supervised and Unsupervised Learning. In reinforcement learning the goal is generally to generate a policy for choosing actions based upon experience. In reinforcement learning one typically does not seek to mimic a correct policy given ahead of time but rather must generate a new policy through repeated trial and error. Generally sequences of actions are chosen by a reinforcement learning agent following some policy while acting in an environment. Then based on the outcome of this sequence of actions, the policy may be updated with the intent to increase the prevalence of desirable outcomes.

There also exists combinations of these three classes of Machine Learning algorithms. One such common combination is semi-supervised learning, in which both label and unlabelled data is used. In addition supervised learning can sometimes be used as a component of a reinforcement learning algorithm.

2.2 Artificial Neural Networks

Artificial Neural Networks are a type of machine learning algorithm that have grown in popularity in recent years due to their effectiveness in a number of previously challenging domains. These methods are named due to their nodal structure which is inspired by the neurons in the brain. There are a number of different classes of neural networks that are employed. These include fully connected feed forward neural networks, convolutional neural networks, and recurrent neural networks. In this dissertation fully connected feed forward, and recurrent neural networks are used extensively so they will be reviewed in this section.

2.2.1 Fully Connected Feed Forward Neural Networks

Fully connected feed forward neural networks are one of the more basic and standard architectures. They are referred to as feed forward because inputs progress through the network without recursive connections such as in recurrent neural networks, and they are referred to as fully connected because every node in each layer connects to every other node on the next layer.

Such networks may take in a vector of inputs. The networks generate the output vector by repeatedly applying a linear transformation to the input and then applying a non-linear activation function to each element of the resulting vector. Each such transformation is referred to as a layer.

An example fully connected feed forward neural network is shown in figure 2.2. Here each node represents a value of one element of the vector for that layer, and the edges represent the multiplication of a weight value. The values corresponding to the directed edges pointing into a node are summed along with the bias value to generate the value of that node.

In this example there is a single hidden layer between the input and output

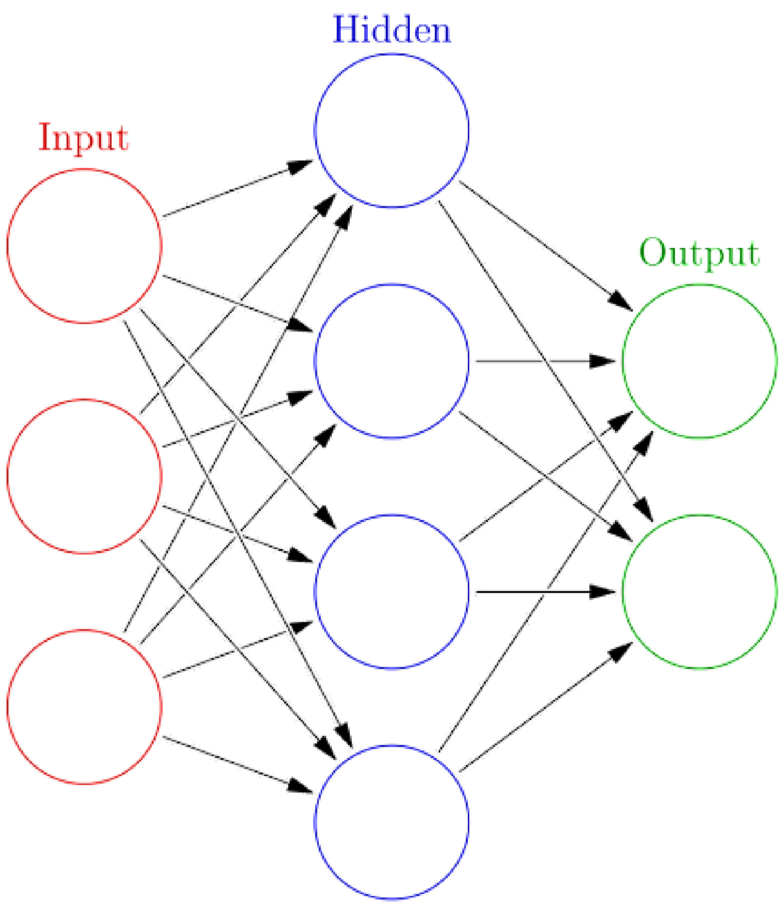

Figure 2.2: Here a fully connected feed forward neural network with one hidden layer is shown.

vectors. Let x be the input vector, h be the hidden layer, and y be the output. The mapping of the input to the output may be written as follows:

$$h = \sigma(W_1 x + b_1) \tag{2.1}$$

$$y = \sigma(W_2 h + b_2) \tag{2.2}$$

Here the matrices and vectors represented by W and b are the weights and biases that parameterize the linear transformations. σ represents the elementwise application a non-linear activation function. This structure can be extended to include any number of hidden layers which each have some number of nodes.

Common activation functions include the sigmoid function shown in figure 2.3 and equation 2.3, as well as the rectified linear function shown in figure 2.4 and equation 2.4.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

$$ReLU(x) = max(0, x) \tag{2.4}$$

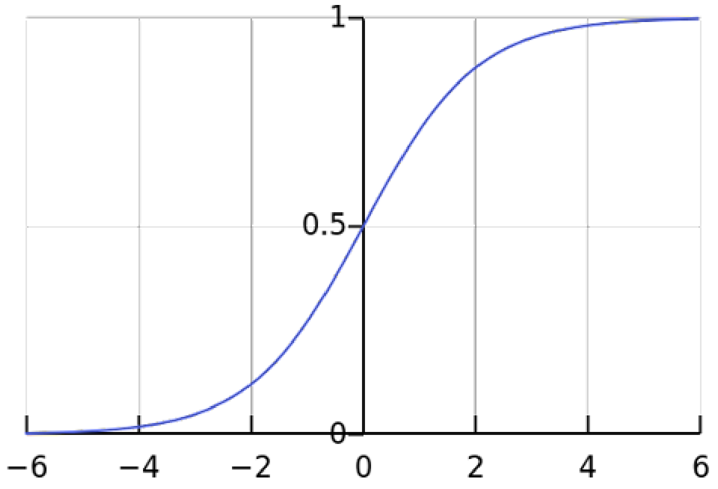

Figure 2.3: Here the sigmoid information function is shown.

Sigmoid was common originally but rectified linear activation functions have

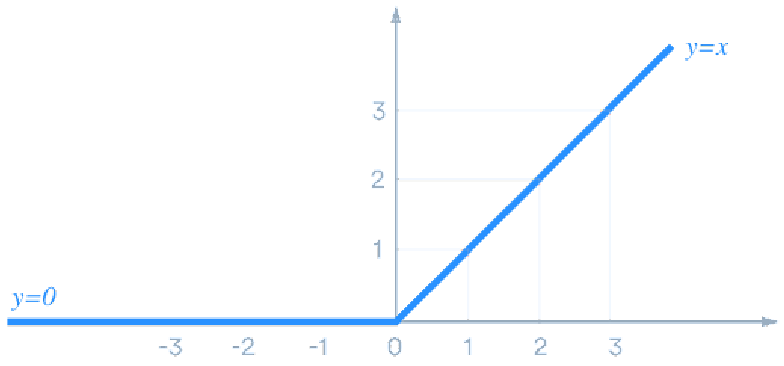

Figure 2.4: Here the rectified linear unit activation function is shown.

become more popular do to increased performance in many applications [14].

Neural networks are generally trained in a supervised manner where weights and biases are updated through stochastic gradient descent to minimize a chosen loss function. A common loss function is the mean squared error between the outputs and training labels.

$$l(x, y, W) = (y - f(x, W))^2 \tag{2.5}$$

By taking the gradient of the loss with respect to weight and bias parameters, these may be updated to reduce the loss

$$w_{t+1} = w_t - \alpha \frac{\partial l}{\partial w} \tag{2.6}$$

Here α is a learning rate parameter which is used to adjust the magnitude of each weight update.

2.2.2 Recurrent Neural Networks

Recurrent neural networks are a class of artificial neural network that specialize in handling sequences of input data $x^{(1)}, ..., x^{(T)}$. Recurrent neural networks maintain an internal state in their hidden layers. At each time step and additional input $x^{(t)}$ is fed into the network, combined with the internal state, and then processed to form a new internal state. Recurrent neural networks allow for this by use of parameter sharing. This parameter sharing allows the model to be applied to sequences of different lengths. If there were different parameters at each time index, then the model could not generalize to sequence lengths not seen during training [7].

One such basic form of a recurrent neural network is shown in equation 2.7

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{2.7}$$

Here each hidden layer is a function of both the previous hidden layer as well as the input $x^{(t)}$ for that time step. θ represents the parameters of the function which may be used repeated at each each time step for any number of steps.

2.2.2.1 Long Short Term Memory Recurrent Neural Networks

While basic recurrent neural networks can be effective on some tasks for short sequences of inputs, there are challenges when training on longer sequences. Weight parameters in the networks are trained via stochastic gradient descent. In order to calculate the gradient of a weight with respect to a loss function, the gradient must be propagated back through each layer of the network between that use of the weight and the output from which the loss is calculated. When this number of layers is very large it makes training difficult as the gradient has a tendency to either blow up to large values or shrink down to small values [7].

Some models of recurrent neural networks, expand upon the basic structure to help address this issue. One such model is the long short term memory recurrent neural network (LSTM). A core component of the LSTM is a central cell state that is updated additively at each time step. Due to this additive nature, changes to the cell state at a given time step will have a strong effect on the loss calculated many steps in the future. The calculated gradient of such a cell state will no longer need to include repeated applications of the chain rule as one passes through the numerous layers between the weights and losses. This will reduce the problem of vanishing gradients that is prevalent is basic recurrent neural networks and allow for increased performance on long input sequences [7]. The equations representing the LSTM are shown in 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6.

$$f_t = \sigma(W^f * [h_{t-1}, x_t] + b_f) \tag{2.8}$$

$$i_t = \sigma(W^i * [h_{t-1}, x_t] + b_i) \tag{2.9}$$

$$o_t = \sigma(W^o * [h_{t-1}, x_t] + b_o) \tag{2.10}$$

$$C'_t = \tanh(W^c * [h_{t-1}, x_t] + b_c) \tag{2.11}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot C'_t \tag{2.12}$$

$$h_t = o_t \odot \tanh(C_t) \tag{2.13}$$

As mentioned above a main component of the LSTM is the cell state. This cell state is updated as shown in equation 3.5. Here the previous cell state C_{t-1} is updated by adding on the candidate cell update C'_t. C_{t-1} and C'_t are also modified by elementwise multiplication of the forget gate and input gate values which are calculated from the input x_t and the previous hidden state h_{t-1} as shown in equations 3.1, 3.2. The elements of the forget gate and input gate are bounded between 0 and 1 and serve to reduce the magnitude of components of the cell state and candidate cell update that may not be necessary to retain. After the cell state is updated the hidden state is generated as shown in equation 3.6 which includes an output gate that functions similarly to the input and forget gates.

The LSTM architecture has proven to be very effective on a number of applications including handwriting recognition, speech recognition, handwriting generation, machine translation, and image captioning [7].

2.3 Reinforcement Learning

2.3.1 Foundations of Reinforcement Learning

Reinforcement learning algorithms are typically employed when making sequences of decisions in an environment. These environments are generally modelled as Markov Decision Processes. A Markov Decision Process consists of a set of states which describe the current state of the environment, a set of actions which represent the possible choices that can be made, and the reward signal which is a scalar value that indicates how well the control agent is doing at it's desired task at each time step.

For each time step, the control agent for the reinforcement learning algorithm receives some observation variables related to the state of the environment. Based upon these observed values, the agent then chooses an action. Conditional upon this action the environment then progresses to the next time step and a reward signal is generated. The goal of the control agent is to maximize the the sum of all future rewards. This sum is referred to as the return.

There are a number of challenges that make this task difficult. For one, actions may have effects on rewards many time steps into the future. This delay between actions and rewards makes it challenging to determine which actions have caused the rewards at each time. Additionally, there can be stochastic components in the environment which cause the return to vary even when the same sequence of actions is taken.

2.3.2 Value functions and Q-functions

Many approaches to reinforcement learning involve considering the expected return. When calculating the return, there is often a discount factor included. This discount factor reduces each rewards contribution to the return based on how many time steps into the future that reward is. Rewards are discounted by a factor of γ^k where k is the number of time steps from the current time step. Let $V^\pi(s)$ be the expected return for an environment in state s that follows policy π.

$$V^\pi(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, \pi] \tag{2.14}$$

Here r_{t+k+1} is the reward value received after the transition from state s_{t+k} to state s_{t+k+1}. Here the optimal expected return could be written as:

$$V^*(s) = max_\pi V^\pi(s) \tag{2.15}$$

This represents the expected return when following the optimal policy, where likewise, the optimal policy is that which maximizes the expected return. Often the goal of reinforcement learning is to find an optimal policy.

In addition to the value function, the Q-function can also be considered. The Q-function represents the expected return conditional upon being in a state s, taking an action a in that state and then following the policy π from there on.

$$Q^{\pi}(s,a) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a, \pi] \tag{2.16}$$

Similarly to the value function, the optimal Q-function is that which maximizes the expected return.

$$Q^*(s,a) = max_{\pi} Q^{\pi}(s,a) \tag{2.17}$$

Properly estimating the Q-function can be of great use in reinforcement learning tasks. If one has access to the correct Q-function for an environment then the optimal policy can be found simply by selecting the action for each state which maximizes the value of the Q-function. Here the optimal policy could be represented as:

$$\pi^*(s) = argmax_a Q^*(s,a) \tag{2.18}$$

Many reinforcement learning algorithms function by first estimating the Value or Q-functions and then using these estimates to determine the optimal policy. These types of algorithms are referred to as Value-based methods.

The relationship between the Q-function and the value function can be represented by the Advantage function, which is the difference between the two.

$$A(s,a) = Q(s,a) - V(s) \tag{2.19}$$

2.3.3 Categories of Reinforcement Learning Algorithms

There exists a variety of distinct types of reinforcement learning algorithms. Each type has it's own advantages and disadvantages. Which type of algorithm is most appropriate for a given task depends upon the nature of the problem being addressed. Some of the main types of algorithms are summarized in this subsection.

2.3.3.1 Policy based vs Value based

One way to categorize reinforcement learning algorithms is by Value-based and Policy-based methods. Value-based methods operate by first learning to estimate the expected return conditional upon states and actions. Once the expected return can be estimated the optimal policy can be determined by selecting the action with the highest expected return at each time step.

One popular Value-based method is Temporal Difference (TD) learning [22]. Temporal difference methods function by running the policy in the environment for some number of steps to sample the rewards. Using these observed rewards an estimate of the return can be made. This estimate of the return is then used to update the Value function's estimate of the expected return.

One method for doing this is simply to run the policy all the way until the end of the episode. This allows one to directly sample the actual return for that sample path. This is referred to as Monte Carlo sampling. Let G_t be this observed return which is the sum of all rewards observed along that sample path. Using G_t the value function can then be updated as:

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha(G_t - V^{\pi}(s_t)) \tag{2.20}$$

Here α is a learning rate parameter which controls the magnitude of each update. One disadvantage of this kind of Monte Carlo update is that G_t will have a high variance. Especially in cases where there are a large number of time steps per episode, G_t will vary significantly. This may prevent the Value function from converging. In addition, it may be computationally slow to run the episode to its conclusion for each update, and in environments which may go on indefinitely, it is not possible to sample G_t in this manner.

Conversely TD learning may also be employed by only sampling rewards for one time step and then estimating the remainder of the return using the value function. This may be referred to as 1-step TD learning or TD(0). In TD(0) the G_t component of equation 2.20 is replaced by the sum of the one step return r_{t+1} and the discounted estimated value for the next state $\gamma V^{\pi}(s_{t+1})$. In this method the update is heavily biased by the current value function. Therefore it may provide poor updates early in in training when the value function is not yet very accurate. This may lead to slow training.

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha(r_{t+1} + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)) \tag{2.21}$$

One may also employ n-step TD learning methods in which multiple step returns are used without going all the way to the end of the episode.

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V^\pi(s_{t+n}) \tag{2.22}$$

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(G_t^{(n)} - V^\pi(s_t)) \tag{2.23}$$

N-step TD learning allows for a balance between the high variance of Monte Carlo and the high bias of 1-step TD learning.

Conversely to Value-based methods, Policy-based methods develop a control policy without first estimating the value. These may function by denoting a policy distribution π which represents the probability of taking an action a conditional upon being in state s with policy parameters θ.

$$\pi(a|s, \theta) = P(a_t = a|s_t = s, \theta_t = \theta) \tag{2.24}$$

The policy distribution may be updating by modifying the policy parameters. Policy-based methods may be preferable in cases where a stochastic policy is desirable and in environments where estimating the Value is challenging.

2.3.3.2 Model-based vs Model-free methods

Reinforcement learning algorithms can also be categorized based on whether they are Model-based or Model-free. In this context "model" refers to a component of the algorithm that allows for prediction of future states and/or rewards. This model may take the form of a transition function which takes in as input the current state and is able to output the next state or some distribution over possible next states [24].

Model-based reinforcement learning algorithms incorporate models into their methods either as part of the learning or as part of action selection. Those which utilize the model as part of action selection are employing online planning. If the model is used only as part of the learning process, it is know as background planning.

Model-based methods are generally more sample efficient than model-free methods as they may use the states and rewards generated by the model to further refine the policy without having to collect them directly from the environment. However, the creation of the model itself can be challenging and in some environments it does not provide additional benefit. Often only some parts of state are required for action selection, so some Model-based methods will only model part of the state or create lower dimensional embeddings of the state which focus on the most relevant aspects.

Model-free methods learn their policies from data collected through experience without explicitly being able to predict future states or rewards. These methods are often simpler as they do not include the additional steps of developing and utilizing a model. However they often require more samples from the environment in order to refine the policy. Due to their relative simplicity and effectiveness, Model-free methods are commonly used.

2.3.3.3 On-policy vs Off-policy

Reinforcement learning algorithms can be either On-policy or Off-policy. On-policy algorithms update the policy using data that was sampled from the environment using the same policy that is being updated. One such On-policy method is SARSA. SARSA operates by sampling the state, action, and reward at the current time step as well as the state and action at the following time step. These values are then used to update the Q-function.

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t)] \tag{2.25}$$

This is an On-policy method because the policy π which is being updated is the same as the policy which is being used to sample the states, actions, and rewards.

Conversely, Off-policy methods sample from the environment using policies which do not always match the policy the samples are used to update. One such Off-policy method is Q-learning. In this method the update is made using the maximum Q-value for the next state over the set of actions.

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q^{\pi}(s_{t+1}, a) - Q^{\pi}(s_t, a_t)] \tag{2.26}$$

Here, the action corresponding to the maximum Q-value is not necessarily the same as the action which would have been chosen by the policy π. Therefore this would be an Off-policy method.

2.3.3.4 Deep Q-Network

An important advancement in reinforcement learning algorithms is the inclusion of artificial neural network deep learning methods. One such popular method is the Deep Q-Network [13]. The Deep Q-Network seeks to estimate the Q-function by doing a form of Q-learning where a Neural Network function approximator is trained to estimate the state-action value.

In this method, instead of individually updating the Q-value for each state-action, all state-action values are generated by a neural network which is trained through stochastic gradient descent. Generally, the input to this network will be the set of state observations and the output will be the set of estimated expected returns for each action. The loss function for the Deep Q-Network is shown in equation 2.27.

$$L(\theta_i) = E[(r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a_{t+1}; \theta^-) - Q(s_t, a_t; \theta))^2] \tag{2.27}$$

Similarly to updates done in Q-Learning, the target for the updates is the sum of the one step reward and the discounted estimate of the expected return from the subsequent state. Using a neural network to estimate values allows for generalization to previously unseen state-action pairs, and makes it feasible to apply to environments with large numbers of distinct states and actions.

There are a number of important advancements which were introduced in the original Deep Q-Network paper which have proven to be useful in deep reinforcement learning. One such important advancement is the use of Experience Replay [32]. This method functions by storing samples of state, actions, and rewards in a replay buffer. Then these can be randomly sampled from to perform batch training on the neural network. This aids in learning by allowing samples from different times to mixed together in batches which breaks temporal correlations that could impede training.

Another contribution is the practice of temporarily freezing the weight parameters of the target network used to calculate the loss. As shown in equation 2.27 the loss is dependant upon the estimated max Q-value for the state s_{t+1}. These estimates are made using the target network weights θ^{-} which are kept temporarily fixed during training to make learning more stable. The target weight values are then updated after some defined interval to more closely match the new weights θ.

Chapter 3

Long Short Term Memory Recurrent Neural Network for Mortality prediction in Intensive Care Units

3.1 Introduction

Analysis of data with time dependency is a topic that is challenging yet relevant to many tasks. The basic machine learning techniques that are often applied to data analysis do not account for the time dependency of the data. This dependency however is frequently an important and informative aspect of the data. Therefore it is important for techniques to be developed that can account for, and make use of this dependency when generating predictions.

One area in which time series techniques may be useful is health care data analysis in intensive care units. In this domain not only are the values of the measurements taken relevant, but also the time at which they were taken. Some techniques that have been developed in recent years to tackle such dynamic or temporal prediction problems are based on recurrent neural networks, and Bayesian Kalman filters.

In recent years neural networks have shown remarkable effectiveness in a range of machine learning tasks. Recurrent neural networks are a type of neural network architecture specifically designed to be appropriate for time series data analysis. The structure of recurrent neural networks allows them to deal with time series data of variable lengths and keep track of time dependencies. In particular long short term memory (LSTM) recurrent neural network architectures are more effective at modeling long term dependencies and provide state of the art results on many time series tasks.

For these reasons we consider recurrent neural networks to be a promising pathway for enhanced capability in time series analysis. One task of importance which neural networks may be useful for is the analysis of electronic medical records in intensive care units (ICU). Doctors and Nurses have at their disposal a large number of measurements with time dependencies which can be used to help influence estimations of the patients health state and make treatment decisions. However with so many different measurements available it may be difficult for a medical worker to quickly and effectively consolidate this information into semantic meaning that can influence treatment. By application of neural networks it may be possible to process these types of data in order to help inform health care workers of the onset of potentially important and adverse healthcare events, with a view to intervention.

We have developed a novel technique using recurrent neural networks to perform mortality prediction for patient data in the Medical Information Mart for Intensive Care (MIMIC) III datset. The MIMIC III dataset is an open source data set containing intensive care unit data. This data set includes a variety of vitals, lab measurements, doctors' and nurses' notes as well as information on when patients were admitted, the times measurements were taken and, if the patients deceased, the time of death.

Our method generates a mortality trajectory after each recorded health measurement. This trajectory consists of a sequence of mortality predictions over a rolling window. We feel this rolling window prediction may be particularly useful in intensive care units as it provides predictions for the near term mortality as opposed to total in hospital mortality. This focus on the near term may be relevant when considering imminent threats to patient health.

Additionally our approach helps to address the issue of missing values. Unfortunately in health care applications of data analysis, often only a small subset of the input features are measured at each time. Therefore often values must be imputed for each of the feature types that are not measured. This is a serious obstacle that must be addressed when applying these techniques. We have developed a new approach to modelling the input values that avoids the need for imputing missing values.

3.2 Related Work

One risk assessment tool commonly considered in the literature is the new Simplified Acute Physiology Score, (SAPS II) described in the paper ”A New Simplified Acute Physiology Score (SAPS II) Based on a European/North American Multicenter Study” [12]. The SAPS II was designed to measure the severity of disease for patients admitted to the intensive care unit (ICU) aged 18 or older. The score is calculated based on 12 physiological variables, and three disease related variables during the first 24 hours. It also includes information about previous health status such as chronic diseases as well as the type of admission to the ICU. Since it is based upon data for only the first 24 hours after admission, it may be lacking important data for mortality prediction in longer ICU stays, and it may be inapplicable to stays shorter than 24 hours. They also provide a probability of in hospital mortality estimate based on the SAPS II score and report an AUROC of 0.86 for their validation set.

In the paper “Recurrent Neural Networks For Multivariate Time Series With Missing Values” [5] a recurrent neural network is applied to mortality prediction on intensive care unit data from the MIMIC III data set. In this paper they use a gated recurrent unit (GRU) which is a type of recurrent neural network. One distinctive aspect of the approach used in this paper is the way the inputs are formatted for the gated recurrent unit. In their approach input features are assembled into vectors for each time step where each element in the vector is used to record a specific type of feature (ie heart rate, blood pressure, etc). These input vectors are then fed into the gated recurrent unit chronologically. In this paper they predict in hospital mortality based on the first 48 hours of patient stays.

Using this approach they reported an AUC score of 0.8527 on in hospital mortality prediction [5]. This approach however is only applicable to patient stays of at least 48 hours and can does not predict near term mortality after each recorded health measurement.

In the paper "Dynamically Modeling Patients Health State from Electronic Medical Records A Time Series Approach" a method is proposed based on Generalized Linear Dynamic Models that models the probability of mortality as a latent state that evolves over time. This paper uses the MIMIC II data set, which is the predecessor to MIMIC III. In this method input features are again assembled into vectors for each time period. As a result there are again missing values that must be filled. In this paper the missing values are filled by means of a Regularized Expectation Maximization method. By using this approach an AUROC of 0.7606 is reported for in hospital mortality prediction from numerical features for the first 24 hours of stay [1].

3.3 Methods

Our method involves using an LSTM to predict patient mortality rates from the MIMIC III data set. The goal of this method is to create an alert system for healthcare workers that will indicate increases in chance of mortality of a patient after each time a new measurement is input for that patient. Mortality predictions are made using the values of numerical measurements taken on patients while in the ICU. For this study we chose to utilize the top 50 most frequently occurring measurement types. The measurement types used are shown in table 2.1 along with their labels which describe what that measurement type is recording.

Each measurement is fed into the LSTM in the order in which they were recorded. For each measurement input, a mortality trajectory is output by the LSTM for the next 60 minutes following the time at which the measurement was recorded. This mortality trajectory includes 60 values, where each value corresponds to a minute, and represents the probability that the patient will have died by the time that minute has elapsed. In this manner the model is able to generate mortality estimates in a rolling window for the hour following each recorded measurement.

Each x_t is a vector of length 51 which includes the data for a single measurement for a single patient. Each x_t consists of two things. The first is a vector of length 50 which indicates which of 50 feature types the x_t is. I will refer to this as the indication vector. The second is a single value which is the value of that measurement after it has been normalized to mean zero and standard deviation one across all values of that

ITEMID	LABEL
581	Previous WeightF
618	Respiratory Rate
51	Arterial BP [Systolic]
52	Arterial BP Mean
113	CVP
184	Eye Opening
198	GCS Total
211	Heart Rate
454	Motor Response
455	NBP [Systolic]
456	NBP Mean
492	PAP [Systolic]
646	SpO2
677	Temperature C (calc)
678	Temperature F
723	Verbal Response
742	calprevflg
5813	ABP Alarm [Low]
5814	CVP Alarm [Low]
5815	HR Alarm [Low]
5817	NBP Alarm [Low]
5819	Resp Alarm [Low]
5820	SpO2 Alarm [Low]
8368	Arterial BP [Diastolic]
8441	NBP [Diastolic]

Table 3.1: This table shows the first half of the measurement types used for mortality prediction in this study. Included are the numerical item IDs as well as their corresponding labels which describe what this measurement type is recording

ITEMID	LABEL
8448	PAP [Diastolic]
8547	ABP Alarm [High]
8548	CVP Alarm [High]
8549	HR Alarm [High]
8551	NBP Alarm [High]
8553	Resp Alarm [High]
8554	SpO2 Alarm [High]
224168	Parameters Checked
224054	Braden Sensory Perception
224641	Alarms On
220739	GCS - Eye Opening
220045	Heart Rate
220050	Arterial Blood Pressure systolic
220051	Arterial Blood Pressure diastolic
220052	Arterial Blood Pressure mean
220074	Central Venous Pressure
220179	Non Invasive Blood Pressure systolic
220180	Non Invasive Blood Pressure diastolic
220181	Non Invasive Blood Pressure mean
220210	Respiratory Rate
223900	GCS - Verbal Response
223901	GCS - Motor Response
220277	O2 saturation pulseoxymetry
223753	Riker-SAS Scale
223761	Temperature Fahrenheit

Table 3.2: This table shows the second half of the measurement types used for mortality prediction in this study. Included are the numerical item IDs as well as their corresponding labels which describe what this measurement type is recording

type for all patients in the dataset. These two are concatenated together to form each x_t of length 51.

The indication vector is constructed as follows. Each space in the vector represents a type of feature. To indicate a certain type of feature, a value of 1 is placed in the space corresponding to that feature type, and a value of 0 is placed in each other space. In this manner each feature type has a unique length 50 vector associated with it that has a 1 in the space for that feature type and zeros in each other spot.

This type of feature representation is commonly referred to as One-Hot encoding in deep learning literature. Representing categorical variables in this manner is often used when encoding for input to Neural Networks. The main advantages of this method of data representation include the ease of implementation and the efficiency of running time [9].

For the purpose of clarifying this idea assume that there were only three distinct feature types included in the data used as opposed to the 50 distinct types. Assume that these are heart rate, blood pressure and temperature and that they are encoded in that order in the vector. Then for example if the first measurement for a patient was a measurement of blood pressure and that measurement was 0.5 standard deviations above the mean then x_1 would be.

$$x_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0.5 \end{pmatrix}$$

These values are then fed sequentially into an LSTM which generates a new mortality trajectory each time an input is entered. The LSTM is defined by the following equations:

$$f_t = \sigma(W^f * [h_{t-1}, x_t] + b_f) \tag{3.1}$$

$$i_t = \sigma(W^i * [h_{t-1}, x_t] + b_i) \tag{3.2}$$

$$o_t = \sigma(W^o * [h_{t-1}, x_t] + b_o) \tag{3.3}$$

$$C'_t = \tanh(W^c * [h_{t-1}, x_t] + b_c) \tag{3.4}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot C'_t \tag{3.5}$$

$$h_t = o_t \odot \tanh(C_t) \tag{3.6}$$

Here h_t (the hidden state at time t) and C_t (the cell state at time t) are vectors of 80 values each which are initialized to be all zeros at the start of each patient's time series (ie C_0 and h_0 are all zeros) and then are recursively updated according to the equations above. $[h_{t-1}, x_t]$ denotes a concatenation of the hidden state at the t-1 step and the input vector at the t step (the resulting length is 131 = 80 + 51).

The weight matrices (W^f, W^i, W^o, W^c) and the bias vectors (b_f, b_i, b_o, b_c) are parameters which are initialized randomly with mean zero at the beginning of training and then updated using stochastic gradient descent during the training process. The shapes of the weight matrices and bias vectors are (80, 131) and (80) respectively. Here σ and tanh represent elementwise application of the sigmoid function and the tanh function respectively. $\odot$ represents elementwise multiplication of vectors.

The mortality prediction $\hat{y_t}$ is then calculated at each time step as follows:

$$\hat{y_t} = \sigma(W^y * h_t + b^y)$$

Where W^y and b^y are again parameters updated by stochastic gradient descent but this time with shapes of (60, 80) and (60) respectively in order to end up with the 60 mortality prediction values for each of the 60 minutes following the time at which the estimate is made.

The loss function used was mean weighted cross entropy where the weight ratio was set to the ratio of survival cases to death cases in order to adjust for class imbalance. An initial bias of 1.0 was used for the biases in the forget gate (b_f). The weights of the LSTM are optimized using the Adam Optimizer for stochastic gradient descent. Training was done by batching together data into groups of 60 patients each. These were then segmented in time into time segments of 100 measurements each. The weights were updated after each batch of 100 time steps and 60 patients based upon the loss calculated on the final $\hat{y_t}$ values and the corresponding mortality labels for each patient who had measurement values in that time segment.

The labels were represented by vectors of length 60 which represented the state of the mortality of the patient for each minute. These were generated such that each value in the label was one if the patient died before the corresponding minute represented by the index, and zero otherwise.

One aspect which sets this method apart from previous approaches is the way the input features are formatted. In our approach, as described above each x_t contains information about a single measurement. These measurements can be of different feature types at each time step, but only a single one is input at a time. By formatting the inputs in this manner we are able to put in each feature value one at a time while allowing for multiple types of features. This approach lends itself nicely to cases in which one is measuring various types of features over time but only getting data on a single feature or a small subset of those measured at each time.

Conversely in our approach only a single measurement value is input at each time step. This means that we can input the value of one feature type at a specific time with no obligation to put in values for the other feature types at that time. In this approach there is no necessity to put in a value for every feature type at each step so we are able to put in only the values which actually exist in the data. By structuring the inputs in this manner we can help deal with the missing data problem by avoiding the need to impute missing values.

Another aspect in which this method differs from other approaches is the way in which mortality prediction trajectories are generated in a rolling window from each measurement time. In methods such as SAPS II. The severity score is calculated based upon measurements taken within the first 24 hours. Therefore it is unable to provide a score before the first 24 hours have elapsed and it would not include data taken after this initial period. By excluding data taken after the initial 24 hours it may not provide a good estimate for longer ICU hospital stays.

Conversely, in our approach the predictions can be made for any length of stay in the ICU. This allows for greater flexibility in terms of when it can be applied. In addition this would allow it to include more data in predictions than the SAPS II method when assessing hospital stays that are longer than 24 hours.

The approach presented here is also novel because it focuses on generating mortality prediction for the near future. With this method, predictions are made for each minute for the next 60 minutes following the most recently recorded data point. Therefore it can serve as an alert system to identify patients who are at high risk within the next hour. This can help allow medical workers to focus on patients who are in imminent risk of death by alerting them that a patient has a higher risk of mortality in the near future.

3.4 Results

The model was trained on 80% of the data, tested on the remaining 20%. We tested the model by generating the final $\hat{y}_t$ values and corresponding labels for each patient in each batch for the entirety of the test dataset and then calculating the area under receiver operator curve (AUC) for each of the 60 predictions in these sets. We report the mean over these resulting 60 AUC values as well. Using our approach we were able to achieve a mean AUC of 0.857. With further inclusion of more data types and more advanced regularization we feel this value could get even higher.

The AUROC value for the paper "Dynamically Modeling Patients Health State from Electronic Medical Records A Time Series Approach" [1] was 0.7606 for in hospital mortality predictions on the first 24 hours based on numerical measurements. The paper "Recurrent Neural Networks For Multivariate Time Series With Missing Values" [5] reported their AUROC value of 0.8527 for in hospital mortality. The AUROC reported for the validation set in "A New Simplified Acute Physiology Score (SAPS II) Based on a European/North American Multicenter Study" was 0.86.

The mean AUROC for our method was either comparable or exceeded these AUROC values reported in each of the papers listed here. Our method however, is novel in that it can be applied more flexibly to incorporate data from different lengths of stay, it provides near term mortality predictions to indicate imminent mortality risks, and provides a mortality trajectory consisting of a set of predictions over a rolling window. Figure 3.1 shows the measured AUROC values for each minute in the rolling window.

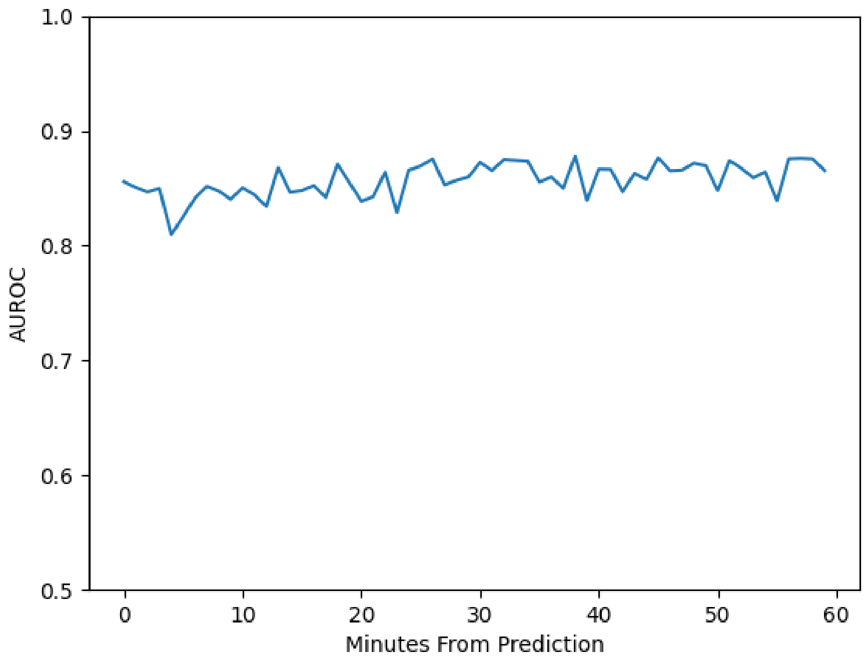

Figure 3.1: Here, the AUROC for each of the 60 predicted mortality probabilities are shown.

3.5 Conclusion

Neural networks have risen in recent years as the cutting edge in many machine learning tasks. Among neural networks, recurrent neural networks and in particular LSTMs have proven to be effective at time series data analysis. The issue of predicting mortality in intensive care units is important for properly allocating care and and involves sequential data.

In this work we have developed a novel technique for generating mortality predictions using LSTMs. Our method makes predictions by generating a mortality prediction trajectory after each measurement consisting of a sequence of predictions over a rolling window. This type of prediction may be useful for medical workers as it can alert them to patients who are at high risk in the near future by combining a variety of different health measurements to predict patient mortality rates.

The current work includes only numerical health records. However, included in the data are also text based records. In future work these methods could potentially be extended to include text based data which may allow for additional predictive ability. Text based data could first be encoded in a manner which matches the dimensions of the numerical values. This would allow this text data to input into the LSTM alongside the numerical values.

Our method is novel for a number of reasons. This method avoids the issue of imputing missing values by formatting the inputs in a way that does not necessitate it, it can be applied flexibly to different lengths of hospital stays, it generates a series of predictions to form a mortality prediction trajectory, and it focuses on near term mortality to alert to imminent mortality risk.

Our method has been demonstrated to be effective by the reported AUROC on our testing data. This work serves to progress the analysis of medical time series data analysis for mortality prediction.

Chapter 4

PDQN - A Deep Reinforcement Learning Method for Planning with Long Delays: Optimization of Manufacturing Dispatching

4.1 Introduction

Scheduling is an important component in Semiconductor Manufacturing systems, where decisions must be made as to how to prioritize the use of finite machine resources to complete operations on parts in a timely manner. Traditionally, Operations Research methods have been used for simple, less complex systems. However, due to the complexity of this scheduling problem, simple dispatching rules such as Critical Ratio, and First-In-First-Out, are often used in practice in the industry for these more complex factories. This paper proposes a novel method based on deep reinforcement learning for developing dynamic scheduling policies through interaction with simulated stochastic manufacturing systems. We experiment with simulated systems based on a complex Western Digital semiconductor plant. Our method builds upon DeepMind's Deep Q-network, and predictron methods to create a novel algorithm, Predictron Deep Q-network, which utilizes a predictron model as a trained planning model to create training targets for a Deep Q-Network based policy. In recent years, deep reinforcement learning methods have shown state of the art performance on sequential decision-making processes in complex games such as Go. Semiconductor manufacturing systems, however, provide significant additional challenges due to complex dynamics, stochastic transitions, and long time horizons with the associated delayed rewards. In addition, dynamic decision policies need to account for uncertainties such as machine downtimes. Experimental results demonstrate that, in our simulated environments, the Predictron Deep Q-network outperforms the Deep Q-network, Critical Ratio, and

First-In-First-Out dispatching policies on the task of minimizing lateness of parts.

This paper proposes a method based on deep reinforcement learning for automated production scheduling in semiconductor manufacturing systems. In such systems, scheduling decisions must be made for processing operations. These systems involve machines that each perform operations to process a variety of different semiconductor devices. Each type of device requires a specific set of operations to be performed which varies depending on the type of device. These systems also include re-entrant connections, and the machines may be subject to machine failures, in which machines break down during processing and need to be repaired before resuming operations. In the industry, static dispatching policies such as Critical Ratio (CR) or First-In-First-Out (FIFO) are often used, together with manual adjustments at failing machines. Obtaining efficient scheduling and dispatch policies at every machine, especially in a system with machine failures, is a challenging and complex task.

In each factory system modeled in this paper, there is a set of machines that are used to process boxes of semiconductor devices on sheets of silicon wafers. These boxes are referred to as parts. At each point in time that a machine becomes available, a dynamic scheduling decision (dispatching decision) must be made as to what that machine should do next. In this paper, this decision is modeled as a choice of which part to process next from the queue at that machine group. Here, this process is modeled as a Markov Decision Process (MDP). An MDP includes a state, action, and reward where the state is the information describing the system, the action is a choice of which decision to make, and the reward is a signal returned after an action is taken which indicates the quality of that action. In the factory systems being modeled, the semiconductor devices produced are hard drive head chips which are used to read from and write to hard drives. There is a range of different head types produced in the facility. In our experiments, we simulate the first 20 processing steps for each head type to model the front of line in the facility.

In recent years deep reinforcement learning (RL) techniques have been demonstrated to show remarkable performance in a number of previously challenging domains such as complex games [13, 19, 20, 2, 25, 16]. In addition, some work has been done in applying such techniques to manufacturing systems, e.g. [28]. One of the main difficulties in this domain is delayed rewards with particularly long delays; other domains with this feature include medical interventions in healthcare. Consequently, our focus in this paper is to develop a planning method that can account for extra long delays in rewards, together with highly stochastic dynamics.

So, this paper presents the Predictron Deep Q-Network (PDQN), a novel deep RL technique that combines the Deep Q-Network (DQN) [13] and predictron [18] methods to learn a policy for dispatching of parts in a simulated system modeled after a semiconductor manufacturing facility. The DQN is a model-free RL optimization algorithm that trains through experience to estimate Q-values which can be used to form a policy. The predictron is a model-based policy evaluation algorithm that can be rolled forward multiple "imagined" planning steps to predict future rewards and values. The PDQN uses the predictron as a trained background planning model to generate value estimates for use in fine-tuning a pre-trained DQN. By doing this, it is possible to incorporate background planning as part of the training process. This combination helps the algorithm account for the highly delayed rewards encountered in these factory systems. Inspiration for this setup comes in part from Dyna [23] in which a model is used to train a policy, the main difference is that the PDQN uses an abstract model to perform its planning. The PDQN is also closely related to methods such as Value Prediction Network (VPN) [15] and MuZero [16], but is different from these papers and approaches by using background planning instead of decision time planning and by using an arbitrary number of steps in between each abstract state representation. Background planning has recently been shown to be the largest contributor to policy improvement when using planning in model-based RL [8]. We compare the PDQN with both DQN and two standard factory dispatching policies, CR and FIFO.

4.2 Related Work

4.2.1 Deep reinforcement learning in Production Scheduling

In recent years, work has been done in applying deep reinforcement learning to production scheduling tasks [3]. Stricker et al (2018) [21] presents a Q-learning with artificial neural network function approximation dispatching method. This method was demonstrated to outperform a First-In-First-Out on the task of maximizing utilization and minimizing lead time on a small simulated semiconductor manufacturing system.

In Zhang et al (2020) [31] a method is similarly proposed to automatically generate priority dispatch rules using a deep reinforcement learning agent. Here, a

Graph Neural Network-based scheme is used to embed the states. This work, however, only applies to job shop problems in which there are a fixed number of jobs to be completed. Therefore this approach would be inapplicable to the more realistic production scheduling problem encountered in semi-conductor manufacturing where new jobs are repeatedly being added into the system and production can continue indefinitely. In addition, the work in [31] only considers deterministic systems which don't account for uncertainties such as machine failures.

In Katsikopoulos et al (2003) [11] an approach to addressing Markov decision processes with delays and asynchronous cost collection is considered. However that work assumes fixed or deterministic delays or stochastic delays which are independent of the state. In the systems considered in this paper the delays are dependent upon the policy so this approach may not be applicable.

In Derman et al (2021) [6] they consider MDPs in which there are action delays such that actions are executed some number of steps after they are chosen. This is a separate issue than the one we are addressing with PDQN. In our case actions are executed immediately, the challenge in our case is that actions affect the return over many time steps.

In Campbell et al (2016) [4] they consider the problem of applying Q-learning with stochastic time delays in the reward signal. This is again different from the issue we are addressing as in our problem, it is not that specific reward values are not received immediately it is that the effects of each action impact the return over long horizons which makes learning and credit assignment difficult. Delayed feedback is also considered in Walsh et al (2008) [26]. However it is limited to fixed, constant delays, which is not applicable for our case.

4.2.2 Deep Q-Network

DQN [13] is a Q-learning method, which works by estimating the expected discounted future return for a given state and action pair. For DQNs this is accomplished by using a Neural Network function approximator. DQN uses mini-batch stochastic gradient descent to update the weights of the neural network based on the gradient of a loss function to minimize the expected value of the loss. For DQNs the Mean Squared Error (MSE) loss is used as seen in Equation 4.1.

$$L(\theta) = E[(r + \gamma \max_{a'} Q(s', a'; \bar{\theta}) - Q(s, a; \theta))^2] \tag{4.1}$$

where θ represents the weight parameters of the neural network. $\bar{\theta}$ represents an earlier copy of θ, which is used to form a target network. The purpose of the target network is to estimate the expected value of the next state. The training is stabilized by only updating $\bar{\theta}$ after a number of training iterations, thereby giving the online network a stable target. An alternative to this is to use a soft update technique where the target is gradually updated towards the online network.

Q-learning is considered an off-policy method, as it can train on data collected by a different policy. To get the best result, however, exploration and exploitation should be tuned according to the problem. A common solution is to use the ϵ-greedy policy on the Q-function, where ϵ is slowly decayed over time. Experienced data is stored as a set of {state, action, reward, next state} in an experience replay buffer and sampled according to some distribution, originally uniformly.

4.2.3 Predictron

The predictron [18] is an architecture for model-based value estimation and policy evaluation. It consists of a fully abstract model which works by "imagining" a sequence of waypoints, each simultaneously describing an arbitrary number of steps into the future and an estimation of the value from the abstract waypoint state. It is strongly related to methods such as n-step TD-learning [29] and eligibility traces [24]. The predictron learns a representation function $\mathbf{f}$ which outputs the first abstract state. Furthermore, it learns K sets of functions, where each set includes a value function $\mathbf{v}^k$, a next abstract state function $\mathbf{s}^k$, a reward function for the transition to the next abstract state $\mathbf{r}^k$, a discount value function γ^k and an eligibility trace function λ^k. Figure 4.1 shows how this architecture is implemented in this paper.

The predictron has two outputs describing the predicted returns (preturns), $\mathbf{g}^{0:K}$ and $\mathbf{g}^\lambda$. Here $\mathbf{g}^{0:K}$ is the set of K preturns, with one k-preturn $\mathbf{g}^k$ for each abstract step k, as seen in Equation 4.2. The λ-preturn $\mathbf{g}^\lambda$, seen in Equation 4.3, is the weighted average of the k-preturns, where the λ-weights are determined using the learned eligibility trace parameters.

$$\mathbf{g}^k = \mathbf{r}^0 + \gamma^0(\mathbf{r}^1 + \gamma^1(\ldots + \gamma^{k-2}(\mathbf{r}^{k-1} + \gamma^{k-1}\mathbf{v}^k)\ldots)) \tag{4.2}$$

$$\mathbf{g}^{\lambda} = \sum_{k=0}^{K} w^k \mathbf{g}^k \tag{4.3}$$

where

$$w^k = \begin{cases} (1-\lambda^k)\prod_{j=0}^{k-1}\lambda^j & \text{if } k < K \\ \\ \prod_{j=0}^{K-1}\lambda^j & \text{otherwise} \end{cases} \tag{4.4}$$

The predictron is trained by minimizing the MSE loss for both $\mathbf{g}^{0:K}$ and $\mathbf{g}^{\lambda}$, as defined in Equations 4.5 and 4.6.

$$L^{0:K} = \frac{1}{2K}\sum_{k=0}^{K}\left\| E_p[\mathbf{g}|s] - E_m[\mathbf{g}^k|s] \right\|^2 \tag{4.5}$$

$$L^{\lambda} = \frac{1}{2}\left\| E_p[\mathbf{g}|s] - E_m[\mathbf{g}^{\lambda}|s] \right\|^2 \tag{4.6}$$

where E_p is the sampled sum of discounted rewards gained in the episode, and E_m is the predicted value from the model.

Thirdly, an optional loss is minimized, the consistency loss, improving the consistency between the k-preturns and the λ-preturn, as seen in Equation 4.7.

$$L = \frac{1}{2}\sum_{k=0}^{K}\left\| E_m[\mathbf{g}^{\lambda}|s] - E_m[\mathbf{g}^k|s] \right\|^2 \tag{4.7}$$

The predictron is the inspiration for both the VPN [15] and MuZero [16], in the sense that they all use abstract state-space representations. The main difference is that VPN and MuZero also train a policy for control, whereas the predictron is purely policy evaluation. VPN and MuZero both use tree searches to conduct their planning, whereas the predictron uses an estimation of eligibility traces to different depths of its abstract version of the expected future.

4.3 Methods

This chapter describes the general setup of the methods used in this paper. Section 4.3.1 describes how the simulated factory is modeled as an MDP. Section 4.3.2 describes the CR and FIFO dispatching policies. Section 4.3.3 describes the proposed new PDQN method while the Neural Network architecture and hyperparameter setup for both the DQN and PDQN are described in the Appendix section 4.6.1.

4.3.1 MDP Modelling

4.3.1.1 State Space Representation

The state-space representation consists of the set of variables $S = S_{hs} \cup S_{hd} \cup S_{hp} \cup S_m$ representing the number of parts of each type at each sequence step, the number of parts due for each head type, the number of parts that are past due, and the machine that the dispatch decision corresponds to.

First, information about the work in process (WIP) is included in the state space. Let $N_{ht,j}$ be the number of parts in the factory of head type ht at step j of production. Included in the state space are the values of $N_{ht,j}$ for all combinations of head type $ht \in H$, where H is the set of all head types and sequence step $j \in J_{ht}$, where J_{ht} is the set of sequence steps of head type ht, $J_{ht} = \{1, 2, ..., M_{ht}\}$, where M_{ht} is the number of sequence steps for head type ht. Let this set of state variables be S_{hs}:

$$S_{hs} = \{N_{ht,j} | (ht \in H) \wedge (j \in J_{ht})\} \tag{4.8}$$

Second, information about the number of parts that are due for each head type is included in the state space. Let D_{ht} be the number of parts due for head type ht. Let the set of due part state variables be S_{hd}:

$$S_{hd} = \{D_{ht} | ht \in H\} \tag{4.9}$$

Third, information about the parts which are past due are included as well. As such the values P_{ht} representing the number of parts past due for each head type ht are included as well. Let this set of state variables be S_{hp}:

$$S_{hp} = \{P_{ht} | ht \in H\} \tag{4.10}$$

Lastly, the state space includes the machine variables where S_m is a one-hot vector indicating the machine that the dispatching decision for that time step corresponded to.

4.3.1.2 Action Space

An action is made every time a machine is ready to process a new part and a part is in its queue. The action determines which part to process next, and consists of selecting a head type and a sequence step. Let H_m be the head types present in the queue for the affected machine m.

$$A = (ht \in H_m, j \in J_{ht}) \tag{4.11}$$

4.3.1.3 Reward Signal

The reward signal is designed to penalize the agent for being late on parts with respect to their due dates. To accomplish this the reward at each time step is made to be negative and proportional to the amount of time that has elapsed during the time step and the number of parts that are past due. This can be represented as an integral over the time between two time steps. Let t_i be the time at which time step i occurs in the MDP representing the factory. Let R_i be the reward for time step i.

$$R_i = -\int_{t_{i-1}}^{t_i} N_d(t)\,\mathrm{d}t \tag{4.12}$$

where $N_d(t)$ is the number of parts that are past due at time t. This way the agent will be repeatedly penalized each time step at a rate which is proportional to the number of parts that are past due. This serves as a heuristic that will encourage the agent to complete parts before their due dates.

4.3.2 Critical Ratio and First-In-First-Out dispatching policies

CR dispatching is a part priority rule based on the ratio between remaining time until due and the remaining time needed to complete processing of a part. The CR dispatching policy proceeds by selecting the part from the queue with the smallest critical ratio value. This will prioritize the parts which are most late and should therefore reduce the lateness of parts.

FIFO dispatching is done by selecting the part to process which was added to the queue first. When considering a single queue, FIFO may reduce max waiting time at that queue by always processing the part that has been waiting for the longest. However, this policy may not be optimal for the whole system as it does not take into account the state of other queues.

4.3.3 PDQN

Presented here is the PDQN algorithm. PDQN addresses the need for learning from highly delayed rewards and dynamic uncertainty due to machine downtimes by including abstract planning for a large number of steps. PDQN consists of two parts, an optimal decision policy determining component based on DQN, and an abstract planning trajectory-based value estimation component based on the predictron; this latter value estimate function of the predictron is fed to the DQN during training. In a traditional DQN, the model trains by minimizing the difference between its Q value estimate and a target value, as seen in Equation 4.1. This target is formed by sampling the reward for one step and then estimating the discounted return from the subsequent state using the same DQN architecture but with older weights, referred to as the target model. In the PDQN, we instead train the predictron part to estimate the value of states under the DQN policy, and use this value estimate as the target for the DQN, effectively substituting $\max_{a'} Q(s', a'; \bar{\theta})$ with $E_m[\mathbf{g}^\lambda|s']$, as seen in Equation 4.13.

$$L(\theta) = E[(r + \gamma E_m[\mathbf{g}^\lambda|s'] - Q(s, a; \theta))^2] \tag{4.13}$$

where $E_m[\mathbf{g}^\lambda|s']$ is the predictron estimate for the discounted return from the subsequent state s' and r is the reward given by the environment.

By using the predictron as target, the policy is trained using background planning. This is because the predictron part is trained to estimate the actual return from running the policy. By incorporating background planning, the policy can be trained towards better targets, which can take much more delayed rewards into account, and by using learned eligibility trace weights, it can learn to better assign the right weight to late rewards. As the policy converges towards a better policy, the predictron will have to be updated to fit the policy again. Where, in traditional DQN, the target model is updated by simply copying the weights of the online model, we here update the target from the predictron by training it on new samples collected by the policy. This is done by fixing the policy for a number of steps and collecting samples of states along with the following h rewards and the value of the h$^{\text{th}}$ state as estimated by the policy. It is important to note, that the predictron is trained in a supervised manner, where the data is collected using the policy. With i as the time step, the target for the predictron

$E_p[\mathbf{g}_i|s_i]$ is defined as

$$E_p[\mathbf{g}_i|s_i] = \sum_{n=0}^{h-1}(\gamma^n * R_{i+n}) + \gamma^h * max(Q(s_{i+h}, \mathbf{a}; \theta_i)) \tag{4.14}$$

where R is the actual return from the following h steps and $max(Q(s_{i+h}, \mathbf{a}; \theta_i))$ is the expected value of the h$^{\text{th}}$ state, estimated by the policy. Therefore the target for the predictron is biased by the estimate of the policy. When using a long horizon the policy will have a small effect on the total discounted return target, while the actual return from within the horizon will increase the variance of the target. Shorter horizon lengths, however, would be more biased by the policy and have lower variance. Consequently, there is a trade-off between these that may lead to different horizon lengths being optimal for different scenarios. The tested PDQN here uses a horizon higher than the estimated number of steps needed to complete on average more than two batches of parts at any given time of the environment.

The loss for the predictron is then calculated using Equations 4.15 and 4.16 and the consistency update loss from Equation 4.7.

$$L^{0:K} = \frac{1}{2K}\sum_{k=0}^{K}\left\| E_p[\mathbf{g}_i|s_i] - E_m[\mathbf{g}^k|s] \right\|^2 \tag{4.15}$$

$$L^{\lambda} = \frac{1}{2}\left\| E_p[\mathbf{g}_i|s_i] - E_m[\mathbf{g}^{\lambda}|s] \right\|^2 \tag{4.16}$$

As a policy evaluation method, the expectation is that the predictron model can provide better estimates of long-term returns than the DQN based Q-value estimator itself. Better return estimates will then create stronger targets for learning when used to train the policy. We expect the target to be better as the predictron architecture can create an abstract planning model, including abstract states which are rolled forward to predict future returns weighed by a learned eligibility trace. The desirable features of the algorithm derive from the predictron and Eligibility Traces properties in incorporating: 1. The true delayed rewards, rather than inaccurate surrogates and 2. The most effective bias-variance trade-off, with the associated dimensional reduction. We hypothesize that these characterizes would enable it to firstly perform very well. Secondly, it would likely dominate stand-alone Q-learning approaches such as DQN and variants.

The choice of using the predictron for planning instead of using decision time planning methods, such as tree search, was based on the nature of the environment. Due to the large delays between when actions are taken and the completion of the parts those actions relate to, actions generally have very little effect on the expected return over the next few states. Therefore if a tree search should be used, it would need to be traversed to a high depth before seeing the outcome of the immediate action taken. By using the predictron with an external control policy, effectively using background planning, the policy can get an estimate of the long-term state value directly on the state. It is then used as the target for the policy that is trained using background training, and through this, the policy is indirectly trained through background training as well. The PDQN training algorithm is summarized in Algorithm 1 along with a detailed hyperparameter setup in the appendix section 4.6.1. The predictron architecture used is shown in Figure

4.1. It is an alternation of the original architecture from [18]. The main change is that all convolutional layers are replaced with fully connected layers. This alternation is made as the state representation in our environment is not spatially related in the same manner as the environments used in the original paper, making local convolutional filters less meaningful.

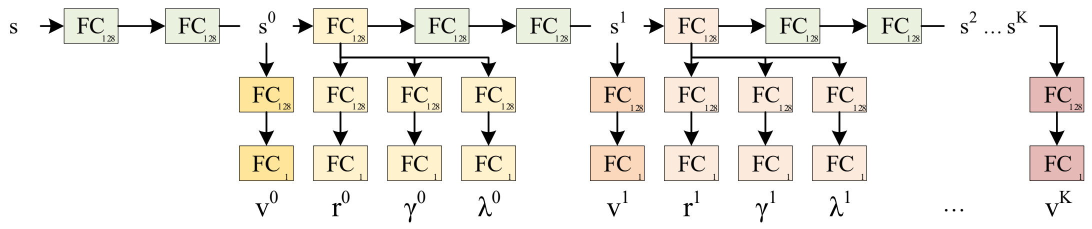

Figure 4.1: The fully connected version of the original predictron implemented with 16 depth layers with individual weights. Each fully connected layer has 128 neurons, except for the output layers which have 1. s is the input state while $s^{0:K}$ are the abstract states. $V^{0:K}$ are the estimated values for each abstract state. $r^{0:K-1}$ are the expected abstract rewards received for transitioning from one abstract state to the next. $\gamma^{0:K-1}$ is the expected abstract discount factors to apply for each abstract step. $\lambda^{0:K-1}$ is the expected eligibility assigned to each abstract step. As the abstract steps is arbitrarily long, γ and λ can vary as well.

4.4 Experiments

To evaluate the PDQN method, two different factory systems are used; a balanced factory system, which was a balanced version of a real factory system, and a randomly generated factory system, which was made publicly available. We compared the mean lateness and the sum of lateness, which was the return received by the agent during an episode. We compared the performance against the CR, FIFO, and DQN policies[1]. For the evaluation, the DQN and PDQN policies were fixed, meaning that they were not allowed to train on the data from the test set.

4.4.1 Setup

The balanced 20 sequence steps (B20) factory system was based on a subset of a real semiconductor manufacturing facility owned by Western Digital Corporation. In this system, the first 20 processing steps for each head type were simulated. The machine groups, process routings, and processing times in the simulated system were set to match the real system. Other factory parameters were set based on simulation results under the CR dispatch policy. Some machines were added to the simulated system to reduce bottle-necking at stations with high utilization. The system was tested with two machine failure rates to experiment with the robustness to changes in this parameter. The Mean Time To Failure (MTTF) was therefore set to either 100,000 minutes and 10,000 minutes, while the Mean Time To Repair (MTTR) was set to 120 minutes, and the WIP level was set to 30 release batches where each batch contained one part of each head type. The due date lead times and WIP levels were set to ensure a mix of on-time and past due part completions. For each sample path in validation and testing, the environment was executed for 100,000 simulation minutes.

The generated 20 sequence steps (G20) factory systems were based upon data shared by Western Digital for a real factory system. These systems were designed to

[1]Initial experiments did not show an improved performance of the DQN by applying the extensions proposed in Hessel et al (2018) [10]. Furthermore, MuZero was considered, as according to Hamrick et al (2021) [8] this method incorporates planning. However, the results from the implementation did not converge.

resemble real systems while being partially randomly generated to allow for sharing of the factory settings without divulging proprietary data. The G20 systems have the same (MTTF) and WIP levels as the B20 systems. The generation of the G20 system factory files is described in the appendix. In all setups, the release of parts into the system was controlled by a CONWIP based policy to maintain relatively constant levels of WIP in the system by releasing each batch of parts when another completes. Due dates were set using a set due date lead time for each head type, which specifies the time between when a part was released and when it was due. Machine failure and repair times were sampled from exponential distributions with specified mean times based on data from Western Digital.

4.4.2 Training and Model Selection

Initial experiments showed that when comparing the performance of the trained models of the same type, large variations were seen between the models, but a low variation was seen on the individual models. This was expected to come from the random initialization of the policies, as well as from the randomness used for exploration. To account for the variation seen between models, multiple models were trained for both DQN and PDQN. For each factory system, 10 different DQNs were trained. Each of these DQN models was then used to initialize a PDQN model training session. The best models, in terms of lateness, were selected through validation simulations. Further training details, architecture, and hyperparameter setup are explained in detail in Section 4.6.1.

4.4.3 Main test results

The best performing DQN and PDQN model from each environment was tested on 50 simulations with different sample paths. Each sample path was initialized with a seed to ensure a reproducible behavior from the environment when used to compare the performance of different policies. For comparison, the mean sum of lateness for the completed parts[2] is used.

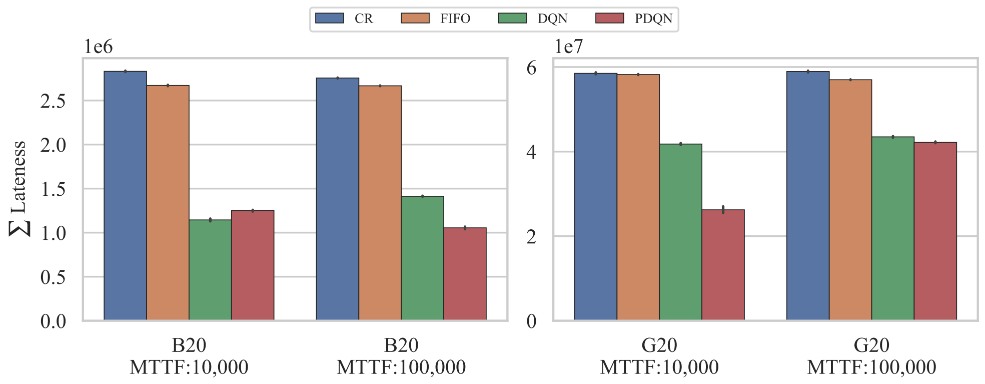

Figure 4.2: Here, the results from the objective function used to train the RL based policies are compared. The results are based on 50 test runs for each setup of each tested environment.

Figure 4.2 shows the results from the 50 test runs on all the tested environments. It is seen that for both factory systems, DQN and PDQN outperform CR and FIFO in terms of the mean sum of the lateness, which is the objective function used. Furthermore, PDQN outperform DQN on the same metric in all systems except the B20 system with MTTF of 10,000. The results seem to indicate that either DQN and PDQN perform similarly well, with very small difference in performance in the one case that

[2]The initial 2$\times$ the WIP $\times$ the release batchsize number of parts are discarded from the results to only consider the steady-state system after a burn in period.

DQN is superior, or the PDQN typically significantly outperforms DQN; we observe this in B20 with MTTF of 100,000 and G20 with MTTF of 10,000.

Overall, the results indicate that PDQN succeeds in improving the performance over the DQN policy, which again improves upon the standard factory dispatching policies, on the given lateness objective function in the tested environment. From a smart factory perspective, choosing PDQN with the correct parameters from the validation step appears to provide a robust close-to-optimal performance for the given objective function.

This performance improvement validates our initial hypothesis that incorporating the long delay reward data from the planning simulation, and the bias-variance features of the predictron, should together provide a powerful approach to addressing very long delays in highly dynamic, stochastic, and large-dimensional systems.

4.4.4 Additional observations

An interesting observation is that for both DQN and PDQN, the sum of lateness often increases when the MTTF is increased for the two systems. By measuring the mean lateness and the number of completed parts from the 50 test runs, as seen in Figure 4.3, some additional insights to this increase, are made. The results can be seen in Tables 4.3 and 4.4 in the appendix section 4.6.1.

PDQN outperforms the DQN in terms of mean lateness except in the B20 system with MTTF of 10,000. However, it is interesting to see how the number of completed parts, in general, is lower for the RL methods compared to the CR and

FIFO policies. This result indicates that the used objective function of reducing the lateness of parts might result in unwanted behavior in terms of throughput, which is not part of the current objective function. Future work might need to consider an objective combining lateness and throughput. However, we do note that in the G20 system with an MTTF of 10,000, the PDQN throughput outperformed CR.

The G20 system has also been tested with WIP level of 15. The results from this test is shown in Table 4.4 in the appendix. The PDQN and DQN performance on those systems indicated that the learned policies favors higher WIP levels. This is possibly because higher WIP levels allow the policy to have greater flexibility, as more parts will line up at the machine queues, effectively increasing the number of allowed actions.

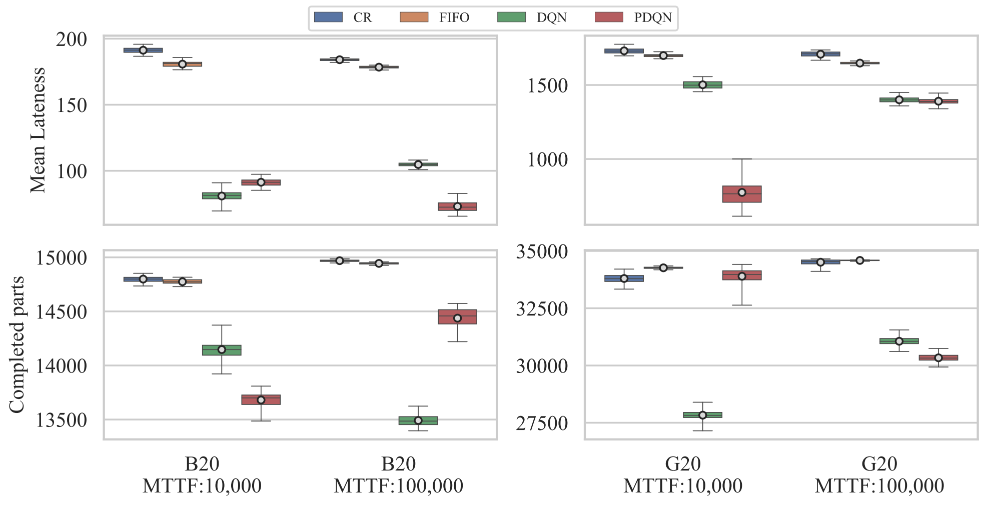

Figure 4.3: Additional observations regarding mean lateness (Top) and the number of completed parts (Bottom) are shown here. The data is shown with the inner quartiles (colored boxes), the median (black lines), the mean (white dots), and the min and max values (whiskers). Both the mean lateness and the number of completed parts is reported for all parts completed after the initial two full system runs, which are removed to align the observation with the objective function.

We observe that the validation step is fairly effective in the choice of hyperparameters including model parameters and iterations. Future research should experiment with hyperparameter optimization of the PDQN method to avoid this excess amount of training.

4.5 Conclusions

In this paper, we approach the problem of dispatching in simulated Semiconductor Manufacturing systems by using deep RL techniques. We present the PDQN, a novel deep RL approach combining DQN with value estimates from the predictron. We evaluate the use of both DQN and PDQN on two factory systems. The deep RL methods are compared against CR and FIFO dispatching policies. The results show that PDQN outperforms CR, FIFO, and DQN in terms of lateness of part in the systems and that both deep RL methods outperform CR and FIFO on this task.

From these results, we see that our hypothesis holds true, in that using the predictron architecture to better predict target values with very large delays, and provide powerful bias-variance trade-offs, can indeed increase the performance of a DQN based policy. That is, by incorporating this trained abstract planning model, the policy seems to better learn from delayed rewards in systems such as the dynamic manufacturing systems described here. Currently, however, there is a large variance in the performance of models after training, so carefully choosing a trained model by validating its performance is recommended.

Scheduling in semiconductor manufacturing facilities is a complex task that has a significant effect on the efficiency of production. Estimation of long-term values in this domain is especially difficult due to the nature of the factory systems. By including predictron methods, which have improved predictive ability, we were able to train models that outperform the DQN, CR, and FIFO on our chosen objective of reducing lateness of parts. The methods presented in this paper progress the application of deep RL to scheduling algorithms in this domain.

4.6 Appendix

4.6.1 Architecture and hyperparameter setup

Here we present the environment-dependent DQN and PDQN hyperparameter settings for the experiments. We used the same hyperparameter settings for all experiments. The hyperparameters were tuned based on experiments on the B20 system.

The architecture of the DQN was implemented as a 4 layer fully connected neural network, where the first three layers had 400, 250, and 125 neurons respectively, and used the ReLU activation function. The last layer was the output layer, which had N_a number of neurons, where N_a was the number of actions in the environment. The DQN was trained using the MSE loss with the Adam optimizer. The DQN was trained with a batch size of 32 and a learning rate of 0.005 using a discount factor γ of 0.99. The target network was softly updated with $\tau = 0.125$. Actions were chosen using the ϵ-greedy policy on the Q-function with ϵ starting at 1.0 and decaying with 0.999 at each step. A summary of the DQN hyperparameters is shown in Table 4.1.

The architecture of the predictron part, seen in Figure 4.1, is a fully connected version of the original predictron architecture [18], i.e., all convolution layers have been replaced with fully connected layers. All layers in the architecture, except for the output layers, have 128 neurons and use ReLU activations. The number of neurons for each abstract state-space representation serves as an encoding mechanism, where fewer neurons will result in a more compressed representation. Initial results show that using 128 neurons had a small positive effect on the performance when comparing to using 8, 16, 32, 64, and 256 neurons.

The depth of the predictron was set to 16, where each depth layer used its own weights. This configuration was chosen, as it was the best performing configuration in [18]. L2 regularization was used to counter overfitting. The training was conducted using the MSE loss with the Adam optimizer. The batch size of the predictron was set to 128, and the number of training batches per iteration was 128. The PDQN had the same discount rate as the DQN, $\gamma = 0.99$, and a horizon of 500, meaning that the target for the predictron was the sum of the actual discounted return of the following 500 steps and the discounted DQN value estimate for state s_{i+500}. With this setting we weighed the actual discounted return by 0.993, while weighing the remaining estimated value by 0.007.

$$\frac{\int_0^h \gamma^x dx}{\int_0^{\text{inf}} \gamma^x dx} = \frac{\int_0^{500} 0.99^x dx}{\int_0^{\text{inf}} 0.99^x dx} = 0.993$$

The horizon of 500 was found as 2.5 times the number of steps required to complete one batch on an empty system.

The policy part of the PDQN was initialized to be a copy of a pre-trained DQN. It was trained in the same manner as the DQN, with the only difference being that the target was estimated by the predictron part and that it was trained for 50,000 steps per iteration. A summary of the PDQN hyperparameters is shown in Table 4.2. The environment is set to run for an initial burn-in period using the greedy policy on the Q-function to skip the initial part of the environment before initiating the training.

First the predictron is trained for an initial number of steps, to allow it to learn a good estimate for the value of the policy. Then the policy is updated with the predictron value estimate as the target for an initial number of steps. Then the number of steps for training the predictron and the policy is set to a fixed value, and the alternation between training the two parts is continued for a number of iterations. During training, the Q function is updated after every step using the Adam optimizer on a small batch of data. The actions for the policy are sampled using epsilon greedy with a fixed epsilon. When training the predictron, the actions are sampled using the greedy policy on the Q function. The predictron is updated after all samples in that sequence have been collected in a supervised learning manner.

4.6.2 Additional observations

Here, more results from running the dispatching methods on the two factory systems introduced in section 4.4, are presented. The mean lateness and the number of completed parts are reported for the entire length of the simulation, except for the initial

Table 4.1: Hyperparameter setup for DQN

Hyperparameter	**Value**
Learning rate	0.005
Batch size	32
Discount factor γ	0.99
Exploration rate ϵ	$1.0 \rightarrow 0.02$
ϵ decay rate	0.999
Soft update coefficient τ	0.125
Training steps	$500,000$
Replay buffer size	$10,000$

Table 4.2: Hyperparameter setup for PDQN

Predictron part		**Policy part**	
Learning rate	0.01	Learning rate	0.005
L2 weight	0.01	Batch size	32
Batch size	128	Discount factor γ	0.99
Batches per iteration	128	Exploration rate ϵ	0.1
Horizon h	500	Steps per iteration	$50,000$
Depth k	16	Replay buffer size	$10,000$

$2\times$ WIP $\times$ the release batchsize number of parts, from which the results are discarded. The general trend is that the RL-based methods outperform CR and FIFO in terms of mean lateness, where PDQN outperforms DQN in 4 of 6 setups. CR and FIFO have the highest number of parts completed in the 15 and 30 WIP level experiments. Note that the number of completed parts is low while the mean lateness is also low for both the DQN and PDQN methods. The reason for this might be that the objective function used penalizes the agent from completing parts later than their due time. As the CONWIP release policy is used, the agents might learn to delay the completion of batches as much as possible to delay the introduction of new parts for as long as possible. This way, the mean lateness can be low if the majority of parts get completed fast, and the number of completed parts will be low, due to the delayed completion of the last part of the batch, which would match the results seen.

4.6.3 G20 Factory Settings

Each G20 system included 10 head types. The process routings for each head type were sampled with replacement from a set of 24 stations. The processing times were sampled from a gamma distribution which was set to match the distribution of the balanced factory system. The number of machines in each station was selected to match the level of demand at each station. The MTTF was again set to 100,000 minutes and 10,000 minutes with an MTTR of 102 minutes. The level of WIP was set to 30 release batches, where each released batch contained one part of each head type. For each sample path in validation and testing, the environment was executed for 500,000 simulation minutes. The longer execution times for these setups were set to account for a lower throughput compared to the B20 system.

Table 4.3: Test results for balanced 20 step factory, B20, on 100,000 simulation minutes. The results are averaged over 50 different test runs and shown with ± standard deviation of the sample mean over the different test runs.

	CR	**FIFO**	**DQN**	**PDQN**
		30 WIP BATCHES		
MTTF		**10,000**		
Sum of Lateness	2.83e6±3.92e3	2.67e6±3.68e3	**1.14e6±8.00e3**	1.25e6±4.61e3
Mean Lateness	191.3±0.30	180.7±0.28	**80.9±0.62**	91.3±0.39
Completed parts	**14798.7±3.90**	14774.2±3.18	14146.8±12.52	13680.6±10.78
MTTF		**100,000**		
Sum of Lateness	2.76e6±1.97e3	2.67e6±1.74e3	1.41e6±2.68e3	**1.05e6±7.92e3**
Mean Lateness	184.1±0.13	178.4±0.12	104.7±0.28	**73.1±0.39**
Completed parts	**14968.9±1.26**	14943.7±1.12	13491.3±7.16	14437.5±12.97

Algorithm 1: PDQN training

```
Q = Load pretrained DQN;
Start and run environment for an initial burn-in period, using the greedy
  policy on the Q function;
$predictron\_train\_steps = predictron\_batch\_size * batches\_per\_iteration$;
$TrainPolicy = False$;
$i = 0$;
while training do
    Update state: $\hat{s} = \hat{s}'$;
    Update allowed actions: $\hat{c} = \hat{c}'$;
    Use greedy policy on Q function to find action: $\hat{a} = argmax(Q(\hat{s}, \epsilon)|\hat{c})$;
    Take step: $\hat{s}', \hat{r}, \hat{c}' = step(\hat{a})$;
    Save to replay buffer: $ReplayBuffer.add(\hat{s}, \hat{a}, \hat{r}, \hat{s}', \hat{c}')$;
    Increase step counter: $i += 1$;
    if $TrainPolicy$ then
        Sample batch: $s, a, r, s', c' = sample_{batch}(ReplayBuffer)$;
        Calculate loss from Equation 4.13;
        Use Adam optimizer on batch loss;
        if $i >= policy\_steps\_per\_iteration$ then
            $TrainPolicy = False$;
            $i = 0$;
        end
    else
        $S_i = \hat{s}$;
        $R_i = \hat{r}$;
        if $step >= h$ then
            $E_p[\mathbf{g}|S_{i-h}] = \sum_{n=0}^{h}(\gamma^n * R_{i-h+n}) + \gamma^h * max(Q(\hat{s}')|\hat{c}')$;
            Append $S_{i-h}, E_p[\mathbf{g}|S_{i-h}]$ to $train\_data$;
            if $step >= predictron\_train\_steps$ then
                while $train\_data$ is not empty do
                    $s, E_p[\mathbf{g}|s] = get_{batch}(train\_data_{predictron})$;
                    Calculate k loss from Equation 4.5;
                    Calculate $\lambda$ loss from Equation 4.6;
                    Use Adam optimizer on k loss and $\lambda$ loss;
                    To use consistency updates do:
                    Calculate cu_loss from Equation 4.7;
                    Use Adam optimizer on cu_loss;
                end
                $TrainPolicy = True$;
                $i = 0$;
            end
        end
    end
end
```

Table 4.4: Test results for generated 20 step factory, G20, on 500,000 simulation minutes. The results are averaged over 50 different test runs and shown with ± standard deviation of the sample mean over the different test runs. 2 different failure rates and 2 different WIP levels are compared.

	CR	**FIFO**	**DQN**	**PDQN**
		15 WIP BATCHES		
MTTF		**10,000**		
Sum of Lateness	19.3e6±27.1e3	18.4e6±16.5e3	**13.2e6±24.8e3**	14.9e6±24.2e3
Mean Lateness	561.4±0.74	533.4±0.47	489.9±1.29	**464.6±0.96**
Completed parts	34417.2±9.20	**34540.5±7.01**	26919.6±28.99	32060.0±38.26
MTTF		**100,000**		
Sum of Lateness	18.6e6±16.7e3	17.9e6±10.5e3	13.4e6±22.5e3	**13.0e6±20.4e3**
Mean Lateness	532.7±0.45	512.4±0.30	**428.5±0.68**	449.3±0.82
Completed parts	**34926.1±3.72**	34902.6±2.07	31311.7±18.01	28828.8±24.01
		30 WIP BATCHES		
MTTF		**10,000**		
Sum of Lateness	58.5e6±115.2e3	58.2e6±48.0e3	41.8e6±99.2e3	**26.2e6±399.2e3**
Mean Lateness	1730.8±2.55	1698.9±1.45	1501.6±3.68	**774.6±12.65**
Completed parts	33798.6±25.51	**34265.2±5.32**	27831.6±27.91	33895.5±44.15
MTTF		**100,000**		
Sum of Lateness	58.9e6±102.5e3	57.0e6±35.8e3	43.5e6±87.7e3	**42.2e6±94.4e3**
Mean Lateness	1707.9±2.45	1647.9±1.05	1400.2±2.64	**1390.7±2.95**
Completed parts	34508.6±16.75	**34587.8±2.22**	31057.7±26.64	30336.1±25.41

Chapter 5

Conclusion

The central goal of this dissertation has been the development of neural network based methods for application to problems with sequential prediction and decision making tasks. In chapter 3, I describe a method we developed using LSTM recurrent neural networks for mortality prediction. this method is novel in how it models the inputs to the algorithm in order to address the issue of missing values. It is also novel because it generates a mortality prediction trajectory consisting of a sequence of predictions over a rolling window. This rolling window may be more useful for allocating care as it focuses on the pertinent near term as opposed to overall hospital mortality as has been often done in other approaches.

In chapter 4, I present our methods of deep reinforcement learning for production scheduling in semiconductor manufacturing facilities. We developed two deep reinforcement learning methods. One based on the Deep Q-Network, as well as a novel method known as Predictron Deep Q-Network (PDQN) which incorporates the predictron architecture into the algorithm to allow for better target training of the policy network. We also developed a simulation model for training and testing of our methods and we created a number of simulated factory systems based on real systems at Western Digital Corporation using this simulation model. The presented deep reinforcement learning methods outperformed common industry benchmarks on the task of reducing mean lateness of parts in these simulated environments. Additionally our novel PDQN algorithm outperformed the Deep Q-Network on this task in a majority of the factory systems tested.

This dissertation work contributes to the progression of neural network based methods for sequence based prediction and decision making tasks. The methods developed are applicable to the two important domains of mortality prediction and production scheduling. Here in this dissertation I present these methods, as well as relevant background information, and demonstrate their effectiveness through test results using available data sets and our own developed simulation environments.

Bibliography

[1] Karla L. Caballero Barajas and Ram Akella. Dynamically Modeling Patient's Health State from Electronic Medical Records: A Time Series Approach. In *Proceedings of the 21st International Conference on Knowledge Discovery and Data Mining*, pages 69–78. ACM, 2015.

[2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[3] Juan Pablo Usuga Cadavid, Samir Lamouri, Bernard Grabot, and Arnaud Fortin. Machine learning in production planning and control: A review of empirical literature. *IFAC-PapersOnLine*, 52(13):385–390, January 2019.

[4] Jeffrey S Campbell, Sidney N Givigi, and Howard M Schwartz. Multiple model Q-

Learning for stochastic asynchronous rewards. *J. Intell. Rob. Syst.*, 81(3):407–422, March 2016.

[5] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David A. Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. *CoRR*, abs/1606.01865, 2016.

[6] Esther Derman, Gal Dalal, and Shie Mannor. Acting in delayed environments with non-stationary markov policies. In *International Conference on Learning Representations*, 2021.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Jessica B Hamrick, Abram L. Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Holger Buesing, Petar Veličković, and Theophane Weber. On the role of planning in model-based deep reinforcement learning. In *International Conference on Learning Representations*, 2021.

[9] John Hancock and Taghi Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7, 04 2020.

[10] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, April 2018.

[11] Konstantinos Katsikopoulos and S.E. Engelbrecht. Engelbrecht, s.e.: Markov decision processes with delays and asynchronous cost collection. ieee trans. autom. control 48(4), 568-574. *Automatic Control, IEEE Transactions on*, 48:568 – 574, 05 2003.

[12] Jean-Roger Le Gall, Stanley Lemeshow, and Fabienne Saulnier. A New Simplified Acute Physiology Score (SAPS II) Based on a European/North American Multicenter Study. *JAMA*, 270(24):2957–2963, 12 1993.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[14] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[15] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6118–6128. Curran Associates, Inc., 2017.

[16] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[17] William H. Shrank, Teresa L. Rogstad, and Natasha Parekh. Waste in the US Health Care System: Estimated Costs and Potential for Savings. *JAMA*, 322(15):1501–1509, 10 2019.

[18] David Silver, Hado Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3191–3199, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[19] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George

van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[21] Nicole Stricker, Andreas Kuhnle, Roland Sturm, and Simon Friess. Reinforcement learning for adaptive order dispatching in the semiconductor industry. *CIRP Annals*, 67(1):511–514, 2018.

[22] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[23] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[25] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P Agapiou, Max Jaderberg, Alexander S Vezh-

nevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019.

[26] Thomas Walsh, Lihong Li, and Michael Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18:83–105, 02 2008.

[27] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[28] Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. *Procedia CIRP*, 72:1264–1269, 2018. 51st CIRP Conference on Manufacturing Systems.

[29] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

[30] Lawrence M Wein and Philippe B Chevalier. A broader view of the job-shop scheduling problem. *Manage. Sci.*, 38(7):1018–1033, July 1992.

[31] Cong Zhang, Wen Song, Zhiguang Cao, J Zhang, Puay Siew Tan, and Chi Xu. Learning to dispatch for job shop scheduling via deep reinforcement learning. *NeurIPS*, 2020.

[32] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017.