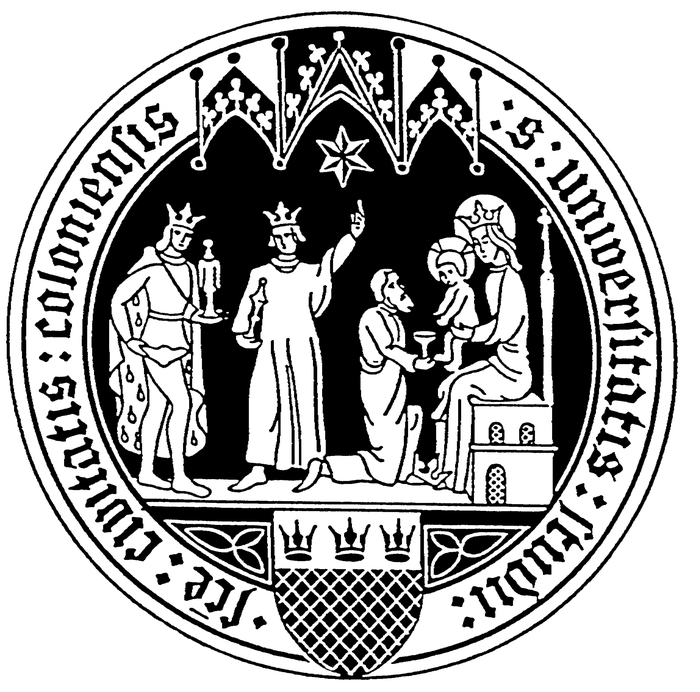
TODO end of writing



Cologne,

## Preface

This thesis was planned and discussed in the winter of 17/18. On February 1st, the work phase of six months started. Within these six months, I discovered many previously unknown or unforeseen complexities. These include the communication technologies developed to permit a complete python based broker and a large variety of API approaches within the RL agent libraries currently available. While I have invested a significant amount of effort into the development of the required components, I always intended to build something that may be reused in the future instead of being discarded after my thesis was graded. This lead me to the decision of implementing a best practice based communication instead of a quick minimal approach and led me to try and write my python code in a way that will let future broker developers reuse it as a framework for their broker implementations.

Why not just write another broker in Java? I believe PowerTAC answers an important question of our time. But I also believe there are not enough people working on this field and it doesn’t receive the attention it should. Thousands of researchers and those who want to become one are working on getting AI agents to become better at Atari games or playing Doom. While the underlying technology advancements are fantastic, the application area is of no use to humanity. I wanted to apply these new technologies to a problem that matters and do so in a way that will create artifacts that others can build upon to outperform my solutions quickly. I wanted to create a bridge between the researchers of RL implementations of recent years and their large community and the exciting field of energy markets. PowerTAC offers another "game" to play with, another environment to let agents compete in. But it is an environment which actually generates value when explored.

As of July, I was not able to complete my research question and reach the intended target of evaluating a variety of neural network architectures that let a RL learn from other agents in its environment. Because of university regulations, changing a thesis title is not permitted. And while my research question was not answered, I believe I still contributed something valuable to the PowerTAC community. With my implementation, current state-of-the-art neural network algorithms and especially reinforcement agent implementations can be used to act in the PowerTAC competition. Python developers can come and join the competition. And while I was not able to create a well performing broker in time and compete with the current participants of the competition, it is nonetheless now possible for others to work on a broker that deploys NN technologies and to focus on the core problems of RL learning problems: Environment observation filtering, NN input preprocessing, reward function definition, NN architecture experimentation etc. With the created Docker images, developers are quickly able to start a competition with multiple brokers and future participants may be encouraged to adopt the Docker based distribution of their agents to include more advanced technologies in their broker implementations without placing a burden on others to manage these dependencies. The new communication layer may be adopted by the competition maintainers to improve performance and to enable other platforms to be used for writing brokers.

When reading the thesis, please be aware that the title does not match the contents as one would expect. Adding a simple "Towards" at the beginning of the title would make it a perfect fit again. Unfortunately, I fell into the same trap that many software engineers and entire project teams fall into: Underestimating the complexity of the project which leads to either loss in quality, time overruns or budget overruns. I chose quality of the work I completed over making it work once but being useless for anyone else afterwards. I hope the thesis is still valuable to anyone who reads it and maybe upcoming graduate theses will continue where I left off.

## Abbreviations

# Introduction

In recent years, AI research saw a steady rise in publications and overall interest in the field . It has been discussed as a key future challenge for nation states and companies alike . Researchers have produced a large corpus of research focusing on visual data learning such as image recognition, audio and text based language recognition and robotics. In the field of RL, recent breakthroughs were achieved in robotics as well as common game challenges like solving Atari games or playing Go .

There are other important problem fields that can also benefit from these technologies, one being global energy markets. These are expected to shift radically in the upcoming decades, adapting to new problems related to global warming, distributed and alternative energy sources, intelligent coordination systems, cybersecurity and electric vehicles . New problem solving techniques are required to solve such *wicked problems*, because they depend on numerous factors such as economic, social, political and technical factors. .

On a local scale, and much more prominent in day-to-day life, appliance manufacturers continuously need to improve their efficiency and machines need to deliver their performance with minimal energy requirements. Cars, fridges, water heating appliances, dishwashers and entertainment systems alike have all shown improvements in their efficiency and it has become a key component of a customers purchasing choice. Similarly, large distributed IT systems as well as building management systems are adapted to more efficiently make use of the energy they require .

On a macro scale, the problem is just as complex, albeit less salient. Electricity grids were conventionally not built to contain *energy buffers*. Electricity always needed to be produced to match demand. This is expected to change over the coming years due to an increasing number of electric vehicles and smart appliances. In addition, decentralized solar energy production changes the demand curve of macro-level energy supply. California is currently suffering a large supply of energy during sunny summer days while lacking energy when wind and solar energy output less due to lack of wind or sunshine. This puts previously unseen stress on the grid systems which were constructed to deliver steady amounts of energy from few sources to many consumers instead of having many small producers distributed throughout the system. Furthermore, large conventional power plants struggle to adapt quickly to change in demand patterns .

PowerTAC, a competitive simulation of future energy markets, attempts to solve the planning dilemma of such complex systems. It allows researchers to experiment with numerous scenarios and participant designs. By adapting system parameters, robust system designs are developed that incentivize participants to behave in alignment with overall interests. The interaction of a variety of market participants using different technologies to automatically generate profit is explored in a competitive game environment. Researchers are invited to participate in this simulation by supplying usage models for appliances and developing *brokers* that participate in the game. Brokers trade energy, offer contracts and coordinate storage capacities within their own customer network as well as with the overall market. The simulation offers opportunities for several fields of research: Game design, energy demand forecasting, intelligent contract design, commodity trading and general simulation and software design questions .

Brokers can be developed by anyone and the competition has been organized for several years now. This means that some broker developers have years of experience while others have not participated in a single competition. Each simulation takes approximately two to three hours to complete and each time slot takes five seconds. Previous researchers have identified the problem as a POMDP, a common model of RL literature . Deep NN architectures have proven to be successful in solving games in a variety of instances. It is therefore intuitive to attempt and apply such architectures to the problems posed by the PowerTAC simulation. Unfortunately, most such implementations are only available in Python and PowerTAC is almost exclusively based on Java. An extension of the current communication protocols to other languages may therefore benefit the reach of the simulation and motivate newcomers to join the competition with their Python based NN architectures.

Finally, a sub field of RL research has identified a problem in the transfer of knowledge from previously trained networks to newly developed iterations. Because NN are mostly black boxes to researchers , it is difficult to extract knowledge and transfer this to another architecture. The learned weights of a NN can not easily be transferred between models, especially when architectures fundamentally differ in their hyperparameters. The field of transfer learning has shown new approaches for solving this problem. Agents with access to previously developed models may pass their observations to the *teacher agent* and initially attempt to align their decisions to those that their teacher would do . More general problem solving agents may be trained by first training several small narrow focus agent networks on sub problems and then training the general agent on the actions of the narrow focus agents . For problems where a reward function is difficult to construct, *inverse reinforcement learning* can be used to train an agent to behave similar to an observable expert. The policy function of the agent shows good performance despite lacking a specific reward function .

In summary, NN are an interesting technology to solve complex problems and energy markets stand to benefit from their usage. To ensure beneficial results, PowerTAC simulates complex markets before implementing them in the real world. PowerTAC focuses on brokers as intermediaries between end consumers and wholesale markets to reduce complexity and decentralize which also aids resilience. To allow current and future teams competing in the PowerTAC competition to easily deploy NN technologies and to allow new brokers in the PowerTAC competition to quickly catch up to previously developed competitor brokers, extending the technology scope of the competition and enabling learning transfer methods and their underlying deep architectures for the problem scope of PowerTAC may be beneficial. The research question for this work therefore goes as follows:

*Can deep reinforcement learning agents learn from actions of other agents in the PowerTAC environment? If so, how? Can imitation allow for boosted performance of reinforcement learning algorithms within a competitive simulation environment?*

To answer the question, a lot of foundation work has to be done. First, the competition needs to be able to interface with the technologies required by modern NN frameworks. Then a problem mapping needs to occur that maps the PowerTAC problems to a structure that the frameworks and libraries can work with. Finally, the current research methods for learning transfer need to be applied to the PowerTAC environment.

## Methodology

First, I will perform a literature research into the fields of AI, RL and the PowerTAC competitive simulation for energy markets. In the field of AI it’s sub fields of SL and UL will be introduced. Here I will focus on the area of NN and a way to let them learn through Backpropagation. In the field of RL I will focus on the MDP framework as well as the POMDP subclass. Next follows an introduction of the recent research in using NN in RL settings to allow for what is now called Deep Reinforcement Learning. This field has seen success in recent research, allowing for agents that successfully play Atari games, 3D games and the game Go on superhuman levels of performance . For PowerTAC , it’s concepts and how agents (called brokers in the context of PowerTAC) make decisions are analyzed. This includes an analysis of previous agents solution approaches.

Following the theoretical background, the main technologies used are briefly explained. Afterwards, the implementation of two important decision areas, wholesale trading and demand predicting, is summarized. Both implementations outline the ability to use current research results from the NN and RL research community to apply them to the PowerTAC problem set.

Finally, a conclusion is drawn and the limits and weaknesses as well as recommended further research is discussed.

# Background

This chapter will introduce the two underlying research fields, AI and the PowerTAC simulation. The broad field of AI will be separated into three sections: An AI introduction, NN and RL introduction, NN and RL. PowerTAC will be discussed by introducing it, comparing it to similar work and analyzing its components and some dominant past broker implementations.

## Artificial Intelligence

The field of AI is both old and yet quiet contemporary. Right with the advent of computers around the middle of the 20th century, research has started to aim for artificial intelligence. Generally, defining AI in a single sentence is hard. structures historical definitions along two dimensions: The grade of how *human* a system is *thinks* or *behaves* and how *rational* it thinks or behaves. These four directions are all pursued by researchers. In this thesis, the goal of *acting rationally* is most appropriate sub fields of research in the larger field of AI.

Today, some 70 years later, AI is again extensively discussed by both researchers and main-stream media . The reasons for this are diverse but it can be argued that the combination of easily available computing power through cloud computing and advances in the mathematical underpinnings have allowed for fast-paced advances in recent years. Also, the currently popular NN architectures often require large amounts of data to learn which have lately been readily available for companies and researchers through the adoption of online technologies by the majority of the population .

### Learning

According to , learning agents are those that *improve their performance on future tasks after making observations about the world* . Among living animals, learning behavior is present in many species. The general goal of AI research is to imitate these skills to dynamically adapt to unforeseen environments. To create a learning algorithm means that the creator did not have to anticipate every potential variant of an environment that the learning agent is confronted with while still creating an agent that can act successfully in such environments. Cognitive Sciences define learning as the change of state due to experiences as a necessary requirement and often limit the recognition of learning to some observable behavior . This applies to all known species and the same definition can easily be applied to a learning artificial agent. A learning agent that doesn’t change its behavior is not helpful and an agent that doesn’t change its state can hardly have learned something

In AI research, a *loss function* is commonly used as a measure of learning progress. Loss functions describe the difference between the actual utility of the right actions versus the results of the agents learned actions. The exact loss function might be a mean squared error function or an absolute loss depending on the learning algorithm that is used or whether the researcher intends to emphasize large deviations from the target .

Computational learning theory looks at different problems of learning: How to learn through a large number of examples, the effects of learning when the agent already knows something, how to learn without examples, how to learn through feedback from the environment and how to learn if the origin of the feedback is not deterministic . In this work, two of those problems are of special interest: The ability to learn from previously labeled examples and the ability to learn through feedback from the environment. The former is called and the latter is referred to as . To understand the difference, it is also important to understand algorithms that don’t have access to labels for existing data, yet are still able to derive value from the information. These belong to the class of UL. Although this class is not heavily relied upon in the implementation of the actual agent in the later practical implementation, it is crucial for tasks in machine learning such as data exploration or anomaly recognition.

The following sections will describe both and and Section [2.2](#sec:neural_networks) will introduce an architecture that can be used as the learning function in these learning problems. Finally, Section [2.2.1](#sec:Backpropagation) will explain how exactly NN learn.

#### Supervised Learning

As noted above, supervised learning uses labeled examples to learn to recognize future examples that might be of the same kind but not identical. Common examples of this form of learning include object recognition in images or time-series prediction. One of the most known examples to date is the Imagenet classification algorithm by which was one of the first NN based algorithms to break a classification high-score on a popular image classification database. The goal is to correctly classify images according to a set of defined labels. If a picture of a dog is read by the NN, it needs to be able to classify the fact that a dog is in the picture. In areas such as financial trading or electricity demand prediction, it can be helpful to be able to predict future patterns based on current and previous observations. In the space of machinery, learning to recognize sensor data that indicates faulty parts can be used to avoid down-time of machines through preemptive replacement during scheduled service intervals. In the online marketing industry, recognizing user interests to send appropriate ads benefits just as well from the approach as do spam filters that recognize ads and filter them out again.

The general problem of supervised learning is as follows:

1. Generation of a *training set* that holds a set of input-output pairs
2. Training of algorithm against training set
3. Verification of results against previously unseen *test set*

If can be any of a set of answers, the problem is a *classification* problem and if the problem requires the prediction of a potentially infinite number of alternatives (e.g. a real number between 1 and 10), it’s a *regression* problem. The outputs , or labels, are created based on an underlying true function which the algorithm tries to learn or approximate through a function , the hypothesis. The space of hypotheses is infinitely large and the general principle, called Ockhams razor, is that simpler hypotheses with equal performance as more complex ones are to be preferred. By deciding up-front about the decision space (e.g. all linear functions) the best hypothesis might not be able to perfectly match the underlying true function . On the other hand a hypothesis chosen from a expressive hypotheses space may generalize very well and is easier to understand and implement.

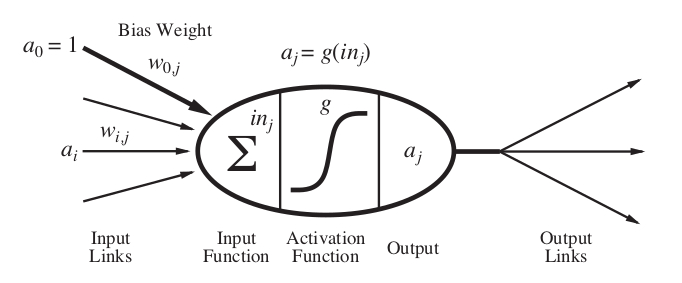
The tradeoff described above is a key factor when deciding on the *right* function to use to solve a supervised learning problem. A linear regression model is easier to understand than complex convoluted functions and NN have often been described as hard to interpret as it is not clear *what* they learn. Systems such as decision trees, which make many sequential decisions about features of the input in question to arrive at a classification, are easy to interpret and might therefore be more appropriate when not only the performance of the system is important but also the inner workings of it.

#### Unsupervised Learning

suffers one key difference: The set of data that is used to learn from does not include labels or classifications to learn from. In other words, there are no examples present to know what it is that needs to be learned. Common examples of unsupervised learning are *clustering* or *principal components analysis*. The overall goal of UL is therefore not to predict but to learn information about the underlying distributions and reasons as to why the values that have been measured were measured as such. UL is also used during pre-processing data for SL problems to improve the later results of the regression or classification problems . Additional features can be constructed from results of unsupervised learning such as distances to cluster centers. These additional features may then also be fed to the learning algorithm. Doing so is also risky however, as it may introduce implicit biases of the data analyst.

## Neural Networks

are a technology that is used to approach problems from both SL and UL problems. The original concept can be dated back as far as 1943 and the mathematical description of a neuron is a linear combination of many input variables and their weights . If the linear combination of the input variables exceeds a threshold, defined by an activation function , the neuron activates or *fires*. The activation can be binary, which leads to the unit being called a *perceptron* or a real value (usually ), which is called a *sigmoid perceptron* . A visual model of this unit is given in Figure [[fig:perceptron]](#fig:perceptron).



Model of the perceptron, taken from .

A neural network is a collection of such neuron components, often layered. The properties of the neurons as well as the overall network properties are called *hyperparameters* and describe the overall architecture of the NN.

A common architecture is the *feed-forward network* which holds several layers of sets of neurons. Each set has no connection within itself but its activation output is fed into the next layers neurons. It is a directed acyclic graph. Other than the weights, this network has no internal state and can not hold information about the input in some form of memory. An alternative is a which includes loops and can hold state. The former network is often used for image classification problems while the latter is used for time-series analysis and natural language processing.

When looking at NN one important decision is the number of layers. In fact, the history of has shown three key phases of progress, the first phase which included simple single-layer networks, the second which included one *hidden layer* and the third phase, today, which uses networks that benefit of several hidden layers. A hidden layer is a number of neurons between the input layer and the output layer. This allows the network to generate complex input-output relationships. Such a multi-layer network is conceptualized in Figure [[fig:multilayernn]](#fig:multilayernn). Each layer feeds into the next until finally the output layer is reached.



Multi-layer neural network from

Neural networks can therefore represent complex non-linear and discontinuous functions even with small numbers of layers or neurons. Such *deep* networks however long suffered from a large issue: It was unclear how to train them, i.e. how to make them learn. The next section describes a solution to this problem.

### Learning Neural Networks and Backpropagation

The previous sections have described learning in respect to the goal of the learning process and the input data that is used to learn from. This section explains the forms of learning and focuses on one widely used form called backpropagation.

When looking at NN while remembering the definition of learning from earlier, it becomes clear that there are many ways a NN can change its state. It could:

1. develop new connections
2. remove existing connections
3. change the connecting weights
4. change the threshold values of the activation functions
5. change its input function
6. develop new neurons
7. remove existing neurons

Of these many actions, changing the weights is the most common way to let a NN learn. This is because many of the other changes in its state can be performed by a specific way of changing the weights. Removing connections is equivalent to setting the weight of the connection to 0 and forbidding further adaption afterwards. Equally, adding new connections is the same as setting a weight of 0 to something that is not 0. Changing the threshold values can also be achieved by modeling them as weights. Changing the input function is uncommon. The addition and removal of neurons (i.e. the growing or shrinking of the network itself) is a popular field of research but will not be discussed further .

Learning by changing the weights therefore covers a wide range of possible adaptions to the network structure. When looking at a single (sigmoid) perceptron, the changing of the weights of its input values is the same process as that of the concept of gradient descent algorithms. Because the activation function is most often *soft*, to ensure differentiability and because a hard threshold creates a non-continuous function, the process of fitting the weights to minimize loss is called logistic regression . For a detailed explanation of the gradient descent approach, I will refer to the works of as well as .

The above described concept of learning from labeled examples is intuitive for single-layer NN. The output can be directly compared to the labels provided by the training set and logistic regression applied to correct the weights of the network to reduce the loss. It becomes problematic though, when several layers are inserted between the input and the output. The weights of the hidden layers are not included in the labeled examples. This is where the concept of *backpropagation* becomes useful. For Figure [[fig:multilayernn]](#fig:multilayernn), any error of the weights of the neurons in layer influence the values of the output values of layer and (in the case of fully connected layer). For any additive loss function (such as ), the error however is simply the sum of the gradients of the losses of the outputs. For a loss it is therefore

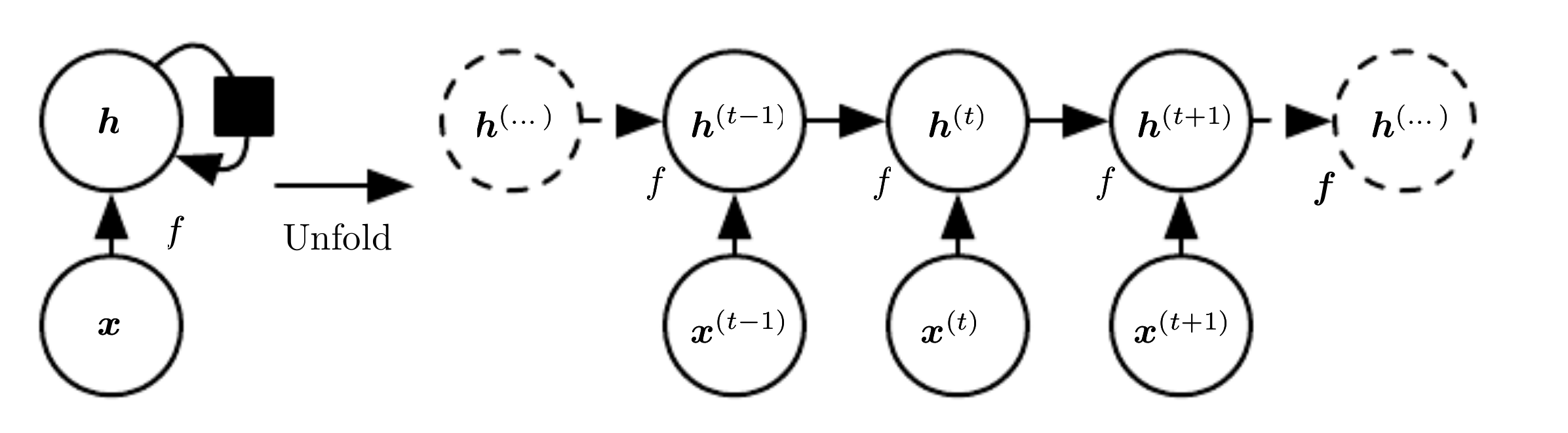
Where is the weight of the target neuron, the target value and the index of the nodes in the output layer . This however does not solve the issue that the training set doesn’t include the expected values for the hidden layers. This is solved by back-propagating the error values through the network. Each previous hidden neuron is considered to be partially responsible for a downstream error in relation to its weight in the target neuron.

### Recurrent Neural Networks

As was already noted in the previous chapter, NN can be both acyclic and cyclic graphs. The *vanilla* NN is usually considered to be an acyclic feed-forward network, as it has no internal state and is therefore more suited to describe the concepts of how the networks operate. Especially in translation and text to speech applications though, RNN are popular as they are able to act on previously seen information in a sequence of data. Generally they are suitable for many applications where the data has some kind of time-dependent embedding .

A RNN, therefore computes its output based on the weights , commonly noted as , it’s current input and it’s previous hidden units internal states .

The network generally learns to use to encode previously seen aspects relevant to the current task, although this is inherently lossy as the previous number of inputs (i.e. ) is arbitrary. Figure [[fig:rnn\_concept]](#fig:rnn_concept) shows this concept.



. *Left*: Circuit diagram where the black square represents a 1 time slot delay. *Right:* The same network unfolded where each node represents a particular time instance. Taken from .

[fig:rnn\_concept]

The network structure has two benefits: Firstly, it allows for arbitrary sequence length, as the network size is dependent on the time slot specific input and not on the number of previous time slots. Secondly, the same network with the same weights (or in mathematical terms the same transition function ) can be used during each time slot. This means: When a RNN is fed a sequence of data, the weights will stay the same throughout the sequence. They can be updated after the entire sequence has been processed.

Such recurrent systems, while theoretically able to hold information across inputs, suffer from an issue called the *vanishing gradient problem*. A network that sequentially processes 20 samples is not easily capable to hold useful information within its state from the early beginning to then act upon it later in the sequence. This is a common problem for translation: Sentences often have structures where the first word influences the meaning of the final one. The network processes each word at a time, diluting the information that is representing the first word because it is covered with noise from the other (potentially irrelevant) words. developed the LSTM model to solve this problem. Each unit in the network is actually a group of gates that act in harmony to store information in a recurrent cell. LSTM implementations differ between libraries but generally, they follow the same core concepts. Modern, tensorflow-based implementations offer GPU acceleration which significantly increases performance and allows for distributed calculation.

## Reinforcement Learning

The previous chapters have introduced concepts of SL , NN, backpropagation and RNN for time-embedded learning tasks. RL can be described as an intersection between supervised and unsupervised learning concepts and Deep RL is the usage of NN, especially those with many layers, to perform RL.

On the one hand RL does not require large amounts of labeled data to enable successful systems which is beneficial for areas where such data is either expensive to acquire or difficult to clearly label. On the other hand it requires some form of feedback. Generally, RL *agents* use feedback received from an *environment*. The general principle of RL therefore includes an agent and the environment where it performs actions. The function that determines the action taken by the agent in a given state is called its policy, usually represented by . The environment reacts to the actions of the agent by returning new states which are evaluated and a corresponding reward is given to the agent. The reward gives the agent information about how well it performed .

This section will first introduce the concepts of a MDP, then introduce different concepts of RL agents, describe approaches to encourage exploration of its options and finally describe how NN can be used to create state-of-the-art agents that can solve complex tasks. The majority of the chapter is based on chapters 17 and 21 of unless otherwise marked.

### Markovian Decision Processes

A common model describing the conceptual process of states and actions followed by new states and new actions of an agent and its environment is called a MDP. In fact, RL is an approach for solving such MDP problems optimally.

A MDP is usually defined by the following components:

* : Finite set of allowed actions
* : Finite set of states
* : Probability of transitioning from state to state when action is taken
* : Discount factor for each time slot, discounting future rewards to allow for long-term and short-term focus
* : Reward function that defines the reward received for transitioning into state

To solve such a problem, an agent needs to be equipped with a policy that allows for corresponding actions to each of the states. The type of policy can further be distinguished between *stationary* and *non stationary* policies. The former type refers to policies that recommend the same action for the same state independent of the time step. The latter describes those policies which are trying to solve non-finite state spaces and where an agent might therefore act differently once time becomes scarce. However, also infinite-horizon MDP can have terminal states which conceptually mean that the process has ended.

A more complex form of MDP is the POMDP which involves agents basing their actions on a belief of the current state. As the later practical application to PowerTAC however can be mapped to a MDP where the transition probability implicitly represents the partial observability , this will not be discussed.

### Bellman Equation

The Bellman Equation offers a way to describe the utility of each state in an MDP. For this, it defines the utility of a state as the reward for the current state plus the sum of all future rewards discounted by .

In the above equation, the *max* operation selects the optimal action in regard to all possible actions. The Bellman equation is explicitly targeting *discrete* state spaces. If the state transition graph is a cyclic graph the solution to the Bellman equation requires some equation system solving. That is because may depend on and the other way around. Further, the *max* operator creates non linearity which, for large state spaces, becomes intractable quickly which is the reason for an iterative approach called *Value Iteration*.

### Value and Policy Iteration

Value Iteration uses the Bellman equation to iteratively converge towards a correct estimation of each states utility, assuming both the transition function and the reward function are known to the agent. In the algorithm, the utility of each state is updated based on the *Bellman update* rule:

This needs to be performed for *each* state during *each* iteration. It is clear how quickly this becomes intractable as well when is reasonably close to 1, meaning that also long-term rewards are taken into consideration.

Practically, the agent however doesn’t care much about the values of various states. It cares about making the right decisions, using the value of states as a basis for doing so. It is often observed that the policy converges far sooner than the utility estimates . This is the basis for the *Policy Iteration* approach which alternates between:

1. evaluating the current policy by calculating , the value of each state if is executed and
2. improving the policy using one-step look-ahead based on

This process stops when the policy is no longer showing any significant improvements in respect to its loss value. It is generally also not necessary to always apply the above operations to *every* state. Instead, state values and policies can be updated only in respect to newly discovered knowledge regarding specific states or specific actions. This is called *asynchronous policy iteration*.

Both variants require the transition function and the reward function to be known to the agent. RL research has developed several methods that adapt the concepts of the two iteration algorithms for environments with the two unknown functions. They are explained in the next sections.

### Temporal Difference Learning

When the underlying transition function is not known, but the agent has the ability to perform many trial runs in the environment, an empirical approach can be adapted. For this, the agent performs a number of trials where it acts according to a (fixed) policy and observes the rewards it receives. Each string of alternating actions and observations is called a trial[[1]](#footnote-42). The update rule for the utility of each state is as follows:

Where is the learning rate and the utility under the execution of in state . This only updates the utilities based on the observed transitions so if the unknown transition function sometimes leads to extremely negative rewards through rare transitions, this is unlikely to be captured. However, with sufficiently many trials, these rare probabilities will be adequately represented in the utilities for the states. If is continuously reduced appropriately, this will converge to the correct value.

### Exploration

The above learning approach has one weakness: It is only based on observed utilities. If follows the pattern of always choosing the action that leads to the highest expected , i.e.

then it will never explore possible alternatives and will very quickly get stuck on a rigid action pattern mapping each state to a resulting action. To avoid this, the concept of *exploration* has been introduced. There are many approaches to encourage exploration. The simplest is to define a factor which defines the probability of choosing a random action at each step.

A more advanced variant is to add a term to the loss function that corresponds to negative entropy of the policy where measures the entropy of a series of actions. This encourages randomness in the policy but it permits the policy function to determine how this randomness gets to occur . This entropy based loss also automatically regulates itself: When the agent is not at all able of choosing rewarding actions it reduces its loss through high entropy choices, i.e. lots of exploration. Once the agent finds actions for certain states that lead to high rewards, choosing other random actions negatively outweighs following the best action. Therefore, it becomes less random and the entropy reduces. If is progressively lowered, the impact on the loss is also progressively lowered, allowing the agent to continuously improve its loss despite less exploration. Another alternative is the positive weighting of actions in states that have not been tried yet, essentially giving such actions an optimistic prior as if they promise higher rewards than the already explored regions. This is easy to implement for small, discrete state and action spaces but more complex for continuous spaces.

### Q Learning

In Section [2.3.3](#sub:policy_and_value_iteration) I have already described how to learn the values of states, given an action. This action can also be derived from a policy function. In the case of an agent that wants to learn its policy (i.e. learn what a good policy is), this becomes problematic if the transition function is not known. An alternative model is called *Q-Learning* which is a form of Temporal Difference Learning. It learns an action-utility value instead of simply the values. The relationship between this *Q-Value* and the former value of a state is simply

so the value of a state is that of the highest Q-Value. The benefit of this approach is that it does not require a model of how the world works, it therefore is called a *model-free* method. The update rule for the Q-Values is simply the Bellman equation with and replaced with and respectively.

The update rules for the Q-Value approach are related to the Temporal Difference Learning rules but include a operator

An alternative version is the reduction of the above equation by removing the operator. This results in the *actual* action being considered instead of the one that the policy believes to be the best. Q-Learning is *off-policy* while the latter version, called SARSA, is *on-policy*. The distinction has a significant consequence: While Q-Learning may be used to learn the Q-Values from recorded state-action pairs, SARSA requires the action taken to be derived from the current policy function.

### Policy Search and Policy Gradient Methods

These two approaches are possibly the simplest of the RL algorithms. In its simplest form, policy search requires the algorithm to start with an initial policy and then adapt this policy until no further gains can be made. While the concepts is simple, it may lead to significant performances, if the *choices regarding what to change* are made wisely. If the policy is just randomly changed, the results will be equally random. If the policy is changed depending on a good interpretation of the environments responses however, this method can offer good performance without the need to have a model of how the world works. Such an agent simply takes the current state as input and uses its policy to determine an output action . The value of a policy is noted as .

For simplicity, I will assume actions derived from a policy to be continuous as both the later application relies on such actions and because the analysis of policy search algorithms becomes more complex in discrete action spaces. When both the policy and the environment are deterministic and without noise, policy search algorithms are quiet effective. The agent can repeat actions in the equivalent states several times, adapting its policy parameters by small values and determine the empirical gradient values which allow the agent to perform hill-climbing in the policy function. This will converge to a local optimum, hence simply trying different actions allows the agent to improve its performance as long as the local optimum has not been reached.

In real world scenarios however, environments (and also policies) are commonly stochastic. Changing the policy parameters by a very small value and comparing results of two instances of executing the policy may lead to strong variations in the reward due to the stochasticity of the environment and therefore the noise in the reward signal. This is a common problem of statistics and the typical answer is to increase the number of trials until statistical significance can be reached. But this is often impractical for real world problems and also not the best approach.

The general idea of modern policy gradient methods therefore follows an approach of using a different function as an estimator for the gradient of the policy in a given configuration. A common approach is to use an advantage function to create an estimator for the policy gradient:

Where describes the advantage of taking one action over another in a given state. It can therefore be described as an *actor-critic architecture*, because , meaning that the advantage value is equivalent to the difference in the estimated value of the state itself and the value of performing a specific action (derived from the policy) in that state

### Deep Learning in Reinforcement Settings

The previous sections have outlined the conceptual approaches for designing learning agents based on various approaches for what is essentially a system that tries to act intelligently in respect to its environment. Especially in recent years, many breakthroughs have been driven by using NN in RL settings. NN have proven effective as parameterized Q-Value estimators, state-value estimators and as policy functions. Most approaches suffer similar problems: Data efficiency in respect to the trial number required to learn a desired skill, scalability and robustness . The reasons for these challenges are obvious: The agents receives minimal feedback and it has a hard time mapping its received reward to specific alterations in its behavior (aka credit assignment problem) .

The research has shown many approaches to alleviate these shortcomings: Inverse Reinforcement Learning allows the learning of a policy by imitating a trusted expert, allowing faster learning rates through clear signals . have created the *gym* which allows for coherent benchmarking of various approaches against a common set of challenges. have created a setup that allows for massive parallel processing of several environments which all contribute to the improvement of a central policy function. The implemented algorithm uses NN and an advanced form of the advantage based policy methods introduced earlier. By this time, the research groups were usually referring to their agents learning progress in the range of millions of time slots . One common argument for the benefit of AI is the ability to transfer knowledge learned by one agent to many agents. The structure of NN however makes it difficult to transfer knowledge between agents with varying hyperparameters. The weights for the neurons cannot simply be copied between networks with different structures and due to the complexity of the systems, learning them again from scratch is resource intensive.

have introduced a concept that helps agents learn faster by breaking complex challenges down into several simpler sub tasks, similar to how humans are taught in educational institutes. This allows researchers to quickly get new generations of agents up to speed with their predecessors.

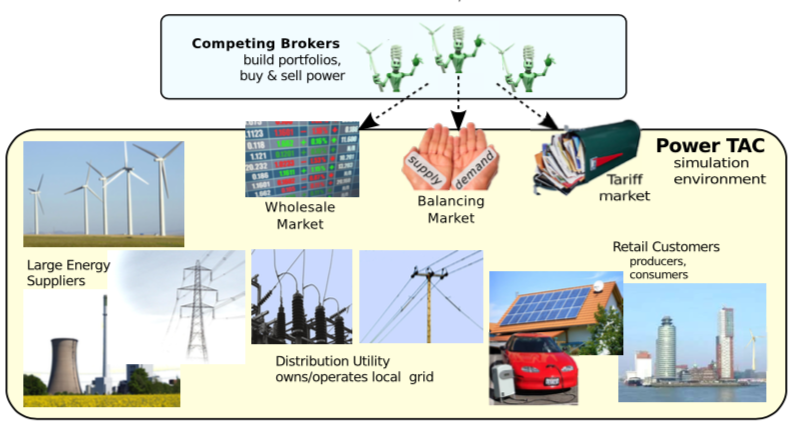
Another approach to solve this problem of repetitive learning has been introduced by . In their setup, the newly created agent transitions from trying to act similar as its *teacher agent* towards trying to improve its performance independently. To achieve this, the student agent includes a term that describes the difference between its action and the action its teacher would have taken.

In summary, many tweaks to the core concepts allow for improvements in the challenges outlined before. Faster learning given limited resources through bootstrapping, improving wall time by leveraging large-scale architectures through parallelization, transferring knowledge from (human) experts through inverse RL etc. A rich landscape of tools is in rapid development and to construct able agents, it is beneficial to leverage both the specific problem domain structure and the available resources.

## PowerTAC: A Competitive Simulation

In the following section, I will introduce the as well as summarize some similarities to comparable research. At the end of the section, some competitor agents are compared and where possible, their underlying functioning analyzed.

PowerTAC simulates a liberalized retail electrical energy market where multiple autonomous agents compete in different markets. Firstly, a tariff market where agents, or *brokers*, compete for numerous end-users through the offering of tariff contracts. Secondly, a wholesale market in which brokers buy and sell large amounts of electric energy to match their customers demands. This market allows brokers to place bids up to 24 hours in advance and each hour the broker has the ability to place new bids to correct for changes in their forecast models. Lastly, the balancing market which places relatively high costs on any broker that causes an imbalance in the system, giving incentives to the brokers to balance their own portfolios prior to the balancing operations. Figure [[fig:powertacoverview]](#fig:powertacoverview) summarizes this ecosystem.



PowerTAC overview of markets

The broker to be developed has to contest in a number of markets and handle a variety of customer types. While PowerTAC generates a fairly complex landscape, it mostly aims at economic complexity rather than modeling the technical underpinnings of the system. It therefore doesn’t simulate any hardware but rather focuses on the different agents involved in the market.

Its goal is the exploration of numerous market designs to find designs that give the right incentives to market participants, allowing for future energy grids to be distributed, failure tolerant and adaptable. Future grids need to handle the changing landscape of energy production, delivery and consume patterns. Consumers need to be incentivized to behave in accordance to energy availability.

### Similar research

PowerTAC is part of a larger body of research based on agent based simulations. The current landscape of generic agent based simulation frameworks is summarized by . PowerTAC falls into a subcategory of simulations concerning the energy markets. surveyed a number of tools in 2009, before the inception of PowerTAC. They define six categories to be used to compare a number of existing platforms and frameworks for creating simulations. In this work, I will just discuss the components PowerTAC does or does not exhibit without describing the other platforms. PowerTAC mostly focuses on the intermediaries between the end consumers and the producers of energy, simulating both ends of the market through automated models and not by defining them as agents with goals and intelligent behavior. It also does not simulate the transmission infrastructures and its capacity, nor does it assume hierarchical structures of local and inter-regional grid interaction. PowerTAC offers, in the form of the central server instance, a strong "Independent System Operator", i.e. an instance that manages the grid, the market and the communication between all agents in the simulation. The wholesale market deploys mostly bidding approaches, in contrast to other simulations that also support bilateral mid- and long-term contracting options. It does however emphasize the concept of offering balancing capacity through energy storage devices and curtailment of energy consumption which was not noted in the survey by .

PowerTAC follows a distributed approach as a technical but as research approach. Several teams can create their own agents and compete with each other. This creates a rich landscape of solution approaches from researchers based in a number of countries and with diverse backgrounds . One drawback: Few teams have opened their agents implementations to others which increases the entry barrier and may lead to duplicate efforts that could have been reused.

### Components

PowerTAC is both technically and logically separated into several components to aid both comprehensibility of the system and yet allow complex simulations of more realistic scenarios. In the following pages, those logical components will be explained. Most of these components are easily mapable into the technical implementation. The technical structure will not be explained in detail but can be found under the GitHub PowerTAC organization.

##### Distribution Utility

The DU represents an entity that regulates the real-time electric usage and corrects for any imbalances in brokers portfolios by correcting the overall net-balance of the system. Any broker who did not balance it’s electric supply and demand incurs costs and is therefore incentivized to always balance its portfolios as good as possible. It also owns the distribution grid and every broker must pay fees for the use of the grid in proportion to the number of the customers it serves . Fees for the grid are constructed in a way to incentivize brokers to not only balance their portfolio but also to avoid high peak demand. It further offers tariffs and is therefore the equivalent of a *baseline broker* whose tariffs create an upper bound on broker profitability.

##### Accounting

All accounting is managed by the central simulation server as to avoid adversarial brokers from tampering with the games rules. Negative balances are usually punished with a 10% p.a. interest rate while positive balances receive a 5% p.a. interest rate. This component tracks every brokers financial balance as well as all brokers customer subscriptions and wholesale market positions .

##### Wholesale Market

Every broker needs to purchase energy before it can sell it to the customers unless the customers of the broker itself generate sufficient energy to balance its own portfolio. For this, PowerTAC offers a wholesale market that operates a *periodic double auction* which represents traditional energy exchanges like those existing in the United States and European markets. Participants in this wholesale market are all brokers as well as a large general entity representing a number of generating facilities, a grid buyer who simulates large-scale demands based on real-world data adjusted based on weather-forecasts and a wholesale buyer who regularly places high-volume, low-price bids. During each time slot, 24 future slots are open for placing bids by the brokers. After the bids have been collected, a clearing price gets calculated which is the intersection between the supply and demand curves. Orders without limit prices are always served first. After the clearing, all uncleared bids and asks are distributed to the brokers to indicate the direction of the markets’ demand and supply curves.

##### Balancing Market

The Balancing market is the last and final trading opportunity for agents and in the sense of the game is at meaning that it occurs virtually in parallel to the consume of electricity. Any imbalance during this phase gets corrected for by the DU who imposes forced balancing of brokers with an imbalanced portfolio. Brokers with too much supply in their portfolio therefore receive very little reimbursement for it and those whose customers’ usage is higher than the estimated amount pay high prices for the additionally supplied energy. The DU also distributes the cost for the grid infrastructure according to the peak demand distribution among all brokers. This is based on the assumption that the grid infrastructure has a static capacity that is required to support the scale it does due to the peak transmission demands. Brokers are therefore incentivized to create portfolios that don’t exhibit large deltas between different hours of the day or days of the week.

Brokers who have tariffs with economic control abilities can pass this capacity along to the DU who utilizes these capacities to correct the markets imbalances, charging customers’ storage devices if an oversupply is present or depleting their devices in the case of an under-supply. It is therefore economically beneficial for brokers to attract customers with such balancing capabilities since it offers a buffer capacity against the balancing costs otherwise incurred through the actions of the DU .

##### Customer Market

The foundation for any brokers ability to generate profit is a sufficient amount of customers being subscribed to its tariffs. For this to occur, the broker must publish tariffs that are competitive to attract customers. On the other hand, if the broker offers tariffs that lead to net losses, long term profit will not be possible[[2]](#footnote-56).

The broker has a wide variety of actions at its disposal to create a rich portfolio. The simulation offers the creation of a variety of tariff types that have variables which are adaptable by the broker. The types include:

Flat rate

Customers pay a flat rate per kWh and they always receive their demanded amount.

Tariff with fixed fee

Customers pay a definable fixed fee every day to receive the service.

Tiered rates

Customers pay a certain price per kWh until a limit is reached after which the kWh price changes. Arbitrarily many such tiers can be added.

Time-of-use

Customers pay different prices depending on the time of the day or the day of the week.

Dyanmic Pricing

Allows the broker to dynamically adapt the price per kWh in real-time to incentives customers to reduce their usage during high demand times. A minimum, maximum and mean price per kWh as well as a notification interval needs to be specified.

Curtailable

Customers can opt in to a tariff that allows the broker to reduce the delivered amount of electricity per time slot up to a certain percentage. This means the customer is exposed to a risk of not receiving the entire electrical supply demanded, usually for a discounted unit cost per kWh.

Storage

Customers can offer their storage capacity to brokers to allow the broker to balance his portfolio. Customers receive payment from the broker if their storage devices are being depleted and pay a (reduced) fee for charging events .

Signup fees and withdrawl fees

Customers can receive bonuses or pay fees for signing up or canceling a subscription.

Some of the above types can also be combined to create complex tariff landscapes for customers to choose from.

##### Customer models

The final part of the simulation environment is made up by the customer models which simulate real-world customers. Each customer can both produce and consume electricity. Consumers are modeled both by factored and elemental models , allowing for small numbers but detailed patterns and large number averaged patterns respectively. The customers evaluate the offered tariffs based on a number of deterministic functions including the various costs and variants of the offered tariffs multiplied by a *irrationality factor* that allows for a more realistic limited rationality of the actors. Additional assessments such as broker reputation evaluation and energy source preferences are also included in the utility function.

Customers do not evaluate every new tariff but only do so irregularly based on an *inertia factor* that limits their attention to new tariffs. Customers are not inherently loyal to their brokers but the inertia factor indirectly causes customers to not immediately switch if there is a more rational tariff available.

As previously noted, customers can both consume and produce electricity. While most production is non deterministic and non controllable (i.e. in the case of solar and wind electricity), some are controllable such as CHP or bio-gas units . Devices such as electric vehicles or water heaters can also offer regulation actions to brokers to balance their portfolios. A *smart* water heater could refill only minimally after heavy use if usage patterns show that the owners will most likely not use it again for several hours. This way, an additional capacity for energy consumption is created that can be profitable for the customer, as the broker usually charges less for electricity delivered under capacity regulation terms .

### Applying observation learning to PowerTAC

In this section I describe some concepts that may be pursued in the development of more performant brokers for PowerTAC. They all adhere to the idea to learn from either previously recorded actions of other brokers or by observing other brokers in the environment.

Generally, a neural network based policy function or value function requires a significant amount of training. Similarly, supervised learning problems require a large training dataset to converge onto their potential performance. Running a simulation takes about 3 hours and delivers some 1600 training steps. This is far below what supervised learning algorithms can train on in a given time span and also RL agent algorithms can perform several hundred steps a second on contemporary hardware. When accelerating the training of the network using modern GPU, this discrepancy becomes even more significant. For the RL based wholesale trading component, some techniques can be applied to boost its learning performance prior to having it interact with a live environment. These techniques are described below.

#### Offline wholesale environment approximation

PowerTAC allows developers to download large amounts of historical game records. Several hundred games are available for 2017 alone, all with different broker participants and broker counts. The powertac-tools repository makes it convenient to download all of them and analyze them for specific data, providing csv files for further analysis. I created records using the powertac-tools project for all games downloadable for 2017 to let the broker train on the datasets. The customer usage analysis[[3]](#footnote-60) provides a historical dataset to create a hypothetical portfolio for the learning RL agent. To design a RL environment, the broker needs a realistic portfolio of required energy. Therefore, a subset of the customers may be chosen to pose as the brokers portfolio. While in a real simulation setting, the customers constantly join and leave brokers tariffs, this offline environment approximation would assume a static portfolio. Furthermore, the market prices analysis[[4]](#footnote-61) gives a historical record of all market closings for each game. In a historical data based environment approximation, the market prices don’t get influenced by the brokers placement of ask or bid orders. This is unrealistic if the broker represents any significant percentage of the overall market but may be a good approximation if the portfolio of the broker is only covering a small percentage of the market. Ultimately, this environment allows for rapid training of a RL agent in the PowerTAC environment by approximating its wholesale market. The disadvantage is the fact that it’s an approximation of the later simulation environment. The learning speed improvement is due to the agent not having to wait for the server to inform it about a new open time slot. Instead, the time slot gets artificially stepped whenever the wholesale trader has completed its trades.

#### Learning from recorded teacher agent actions

The RL agent may in addition to a fixed portfolio be taught to imitate the recorded behavior of a teacher broker such as a former winner of the competition. It may be given the recorded demand data of a competing broker and the reward function would be modeled in a way that incentivizes the agent to behave similar to its teacher broker. This is in accordance to the concepts of inverse reinforcement learning. If the broker may also act in a live competition, it could implement the kickstarting concept of , feeding its observations to a competitor teacher broker and initially attempt to align its actions to those of the teaching broker. Unfortunately, this concept is difficult to implement in the PowerTAC environment. Brokers are black boxes and it is not possible to assume that they will behave correctly if their submitted actions are not the actions that are actually submitted to the server. This would be required, because the learning agent is the one that actually determines the policy, only giving its observations to the teacher agent to *ask for its opinion*. model lets the agent consider the question of what its teacher would do if it were in its situation. Due to the inaccessibility of the teaching brokers inner workings, an alternative model could only ask *what would I do if I were in my teachers situation*. To allow for such analysis, the next technique is required.

#### Counterfactual analysis

Many real-world problems are approachable with RL agent research. What makes PowerTAC and other simulations interesting is the ability to perform counterfactual steps. A counterfactual event is one that is not aligned with what is actually true. In a real scenario, the phrase *"Alan Turing would have not solved the Enigma encryption if he had been fed one apple a day by his mother every morning"* is against what is actually true and therefore cannot possibly be verified. In the PowerTAC simulation, this is very different. Because the entire state of the server is recorded in its state files, it can be reproduced exactly. Unfortunately, the brokers do not offer such ability to reproduce their state. A level of randomness is inherent in their decision making. If a statement were to be: *"Had the broker offered tariff X in time slot 1200, it would have won the competition"*, it is not possible to reproduce the state of the server from the state files alone to verify this hypothesis.

With a technology that allows for *snapshotting* of memory space in Linux, it is possible to create a snapshot of the server state and all its participants running on the same machine. A broker developer may therefore include the ability to create a snapshot of the entire environment state to evaluate a number of alternative actions at any point within the game instead of having to rerun an entire simulation. This approach does not require all broker developers to support this feature. Instead, the mentioned technology allows the recreation of *any* process in the operating system to be recreated and to be in the exact same state as the original. This means, a RL agent can learn not simply through performing full episodes within a learning session, slightly altering its behavior randomly within each episode. If the network can determine points within a MDP where a number of actions are available and any one may lead to an increase in future rewards, the agent may decide to try all or a subset of the possible actions to determine which of the alternative actions leads to the highest rewards. The concept would therefore be a bit different from usual MDP models. It would allow the agent to submit a range of actions and ask the environment to give back a range of alternative scenarios and rewards. While this is still susceptible to random behavior *after* the snapshot occurred, it is guaranteed to be the exact same state at the point where multiple actions are considered. Remaining uncertainty may now be compensated by running a significant amount of trials ceteris paribus.

### Existing broker implementations

Before designing my own agent, it is helpful to investigate previously developed agents and their design to understand the current state of research. For this, I have analyzed the papers of the brokers AgentUDE, TacTex and COLDPower, as they performed well in previous tournaments and because their creators have published their concepts. I also analyzed the paper by which was the first paper to describe a RL agent acting in a predecessor of the PowerTAC environment. This broker, although technically not competing in the PowerTAC competition, is referred to as SELF. Their architectures, models and performances are summarized in the following sections. These are based on publications that describe the TacTex, COLDPower and AgentUDE agents of 2015, as these are the last publications of these brokers that are available on the PowerTAC website. Unfortunately, the source code of these agents has not been made available, which does not allow inspection of the exact inner mechanics.

From what is visible by their shared binaries, all agents are based on java and do not employ any other technologies to perform their actions during competitions.

#### Tariff market strategies

AgentUDE deploys an aggressive but rigid tariff market strategy, offering cheap tariffs at the beginning of the game to trigger competing agents to react. It also places high transaction costs on the tariffs, by making use of early withdrawal penalties and bonus payments . While this may be beneficial for the success in the competition, it doesn’t translate into real-world scenarios as energy markets are not a round based, finite game.

TacTex does not target tariff fees such as early withdrawal fees to make a profit. It also doesn’t publish tariffs for production of energy . TacTex has modeled the entire competition as a MDP and included the tariff market actions in this model. It selects a tariff from a set of predefined fixed-rate consumption tariffs to reduce the action space complexity of the agent. Ultimately though, it uses RL to decide on its tariff market actions, reducing the possible actions based on domain knowledge.

COLDPower also deploys RL approaches with a Q-Learning based agent choosing from a range of predefined changes to its existing tariff portfolio. It can perform the following actions: *maintain, lower, raise, inline, minmax, wide, bottom*. These actions describe fixed action strategies that have been constructed based on domain knowledge. The agent is not *learning* how to behave in the market on a low level but rather on a more abstract level. It can be compared to an RL agent that doesn’t learn how to perform locomotion to move a controllable body through space but rather one that may choose the direction of the walking, without the need to understand *how* to walk. While this leads to quick results, it may significantly reduce the possible performance as the solution space is greatly reduced.

SELF also defines the tariff market as a MDP and uses feature selection and regularization to reduce the state space of their learning SARSA agent. The action space has been defined with discrete pre-defined actions that are similar to that of the COLDPower agent . As COLDPower, the discrete action space by itself introduces assumptions about the problem domain that the agent cannot overcome. As an example, the two actions *LowMargin* (10% margin) and *HighMargin* (20% margin) restrict the profitablity of the agent to two points in the overall action space. Maybe the optimum is at 14.25% or maybe it is even higher than 20%. A discrete action agent cannot discover nor act upon these possible improvements. NN may help overcome this limitation because they can both handle large state spaces and act successfully in continuous state spaces.

#### Wholesale market strategies

AgentUDE considers the wholesale market to include both demand and price prediction. For the demand prediction, AgentUDE uses a simple weighted estimation based on the previous time-step and the demand of 24 hours before the target time-step . Their price prediction is more complex and involves a dynamic programming model based on to find *similar hours* in recent history and determine current prices using Q-Learning . Their MDP is constructed in a way that the agent needs to determine the limit price that minimizes costs. It only has one action dimension which describes the limit price and its environment observation is represented by a belief function which makes it a POMDP. The agent uses value iteration to solve the Bellman equations, determining the expected price. The ultimate limit prices are then determined based on a heuristic that works by offering higher prices for "short-term" purchases and adjusting this to also offer higher prices in the case of an expected higher overall trading volume .

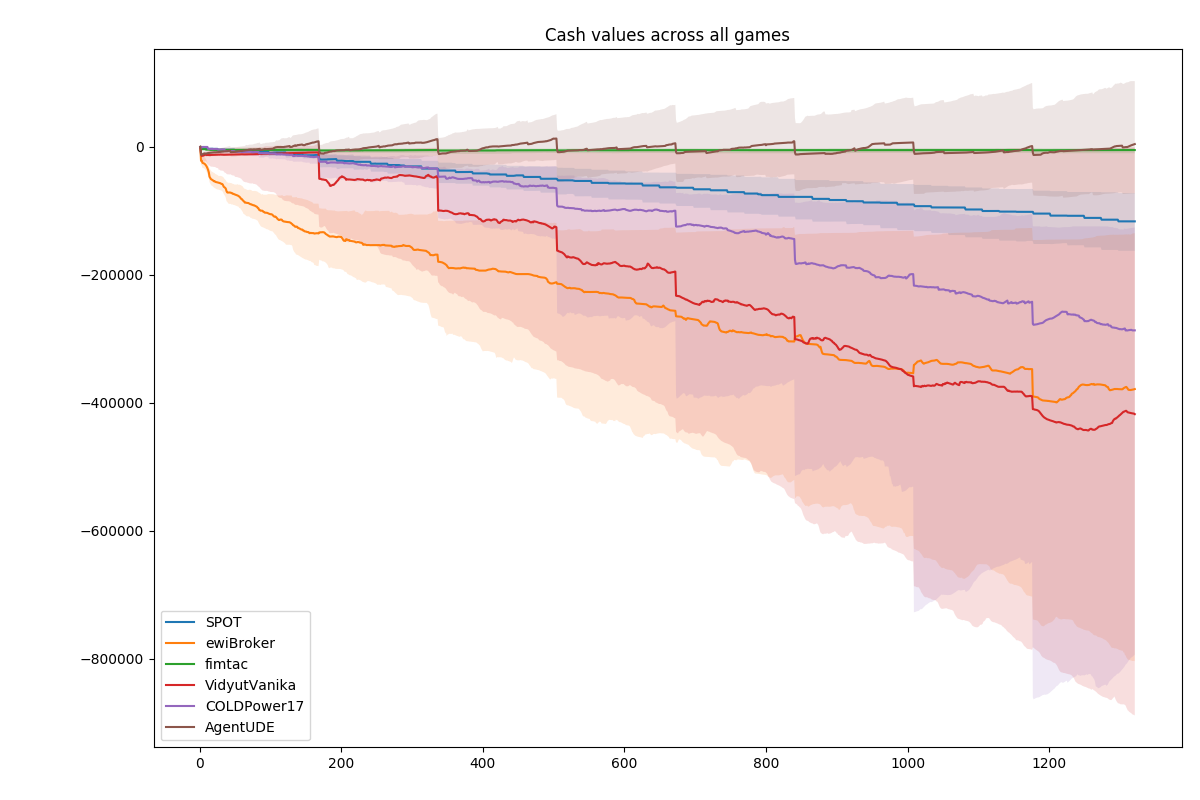
TacTex considers the wholesale market actions to be part of the overall complexity reduced MDP. It uses a demand predictor to determine the mWh amount to order and sets this amount as the amount that is placed in the order. The predictor is based on the actual customer models of the simulation server itself. While this surely leads to good performance, it can be argued whether this is something that actually benefits the research goal. The price predictor is a linear regression model based on the bootstrap period, corrected by a bias correction based on the prediction error of the last 24 hours .

COLDPower deploys a linear regression model to predict prices and determines the demand by "using the energy demand historical information" . The order is placed accordingly.

The authors of SELF don’t describe its actions in the wholesale market. Probably, the early variant of the simulation probably did not contain this component yet and instead, simply calculated the market price for the electricity and submitted it to the agent.

#### Past performances

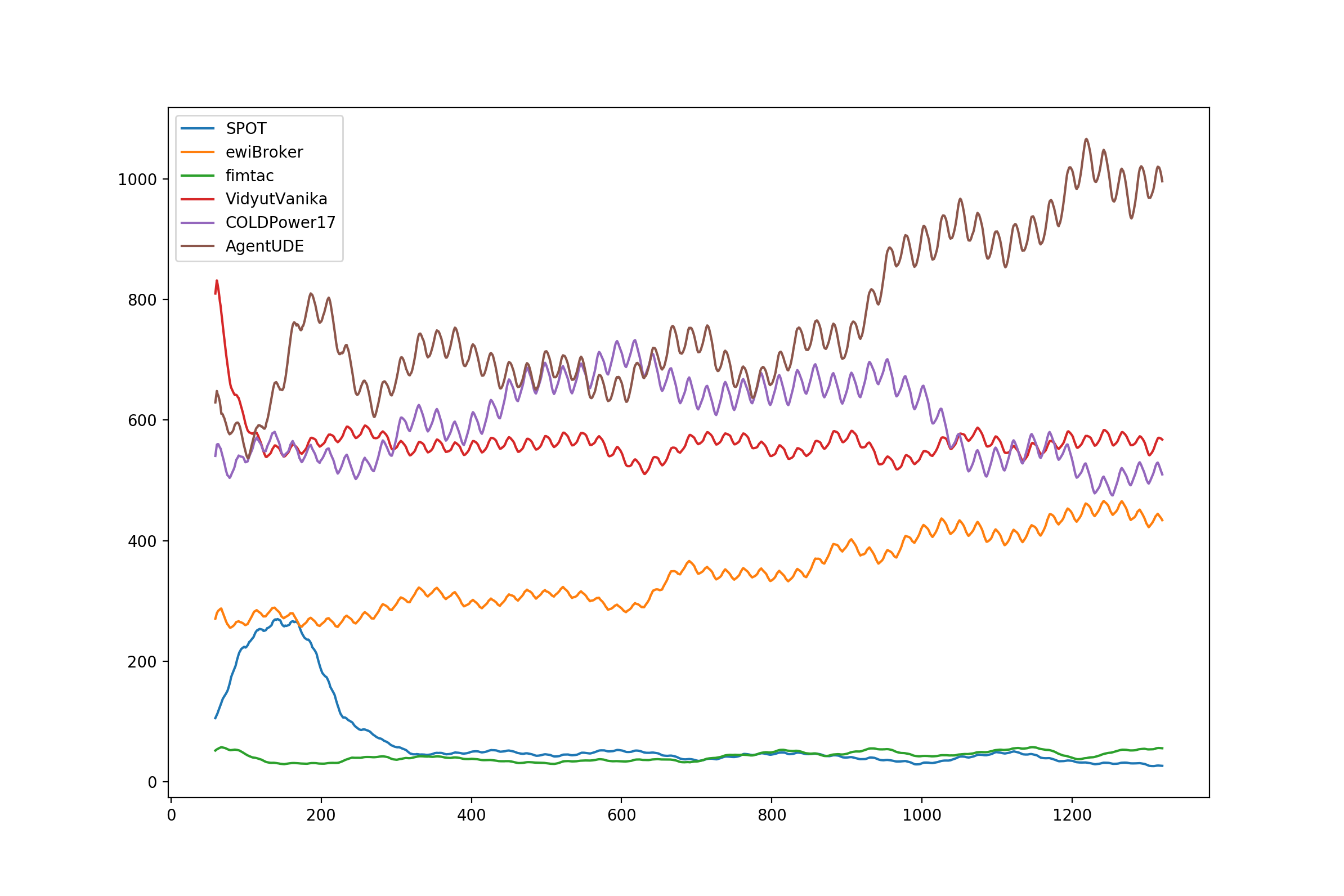
To analyze the performance, all cash statistics of the final rounds of year 2017 were analyzed. TacTex did not participate in the 2017 competition and is therefore excluded in this analysis. Their last participation was in 2015 where they ended up in second place. The improvements made to the previously mentioned agents between their latest publications and their current performances are Unfortunately not determinable.



Cash values across all games in the 2017 finals (median, 0.25 percentile, 0.75 percentile)

[fig:cash\_vals\_across\_games]

When looking at the overall performance profiles (see Figure [[fig:cash\_vals\_across\_games]](#fig:cash_vals_across_games)) of the top 6 brokers of the 2017 finals, it becomes obvious that most brokers are performing rather bad most of the time. Only SPOT, fimtac and AgentUDE managed to consistently stay close to zero or in the case of AgentUDE even above 0 cash balance. When inspecting the tariff transactions closer (see Figure [[fig:allttxucline]](#fig:allttxucline)), it becomes clear that only AgentUDE achieves this through actually being successful in the market. SPOT only acts in the market initially and then quickly looses many of its customers. Fimtac keeps a small continuous customer base throughout most games. AgentUDE on the other hand trades actively in the market, having a solid number of customers subscribed to it. COLDPower also trades actively but its financial results are not as satisfying, loosing significant amounts of money each week and also not being able to sustain its continuous income towards the end of the games.



Tariff TX credit values across all games in the 2017 finals (rolling average)

Generally, AgentUDE can be seen as the peer with the most consistent and stable performance. Their broker acts in all parts of the simulation and makes use of various strategies, including tariff optimization and balancing capacity.

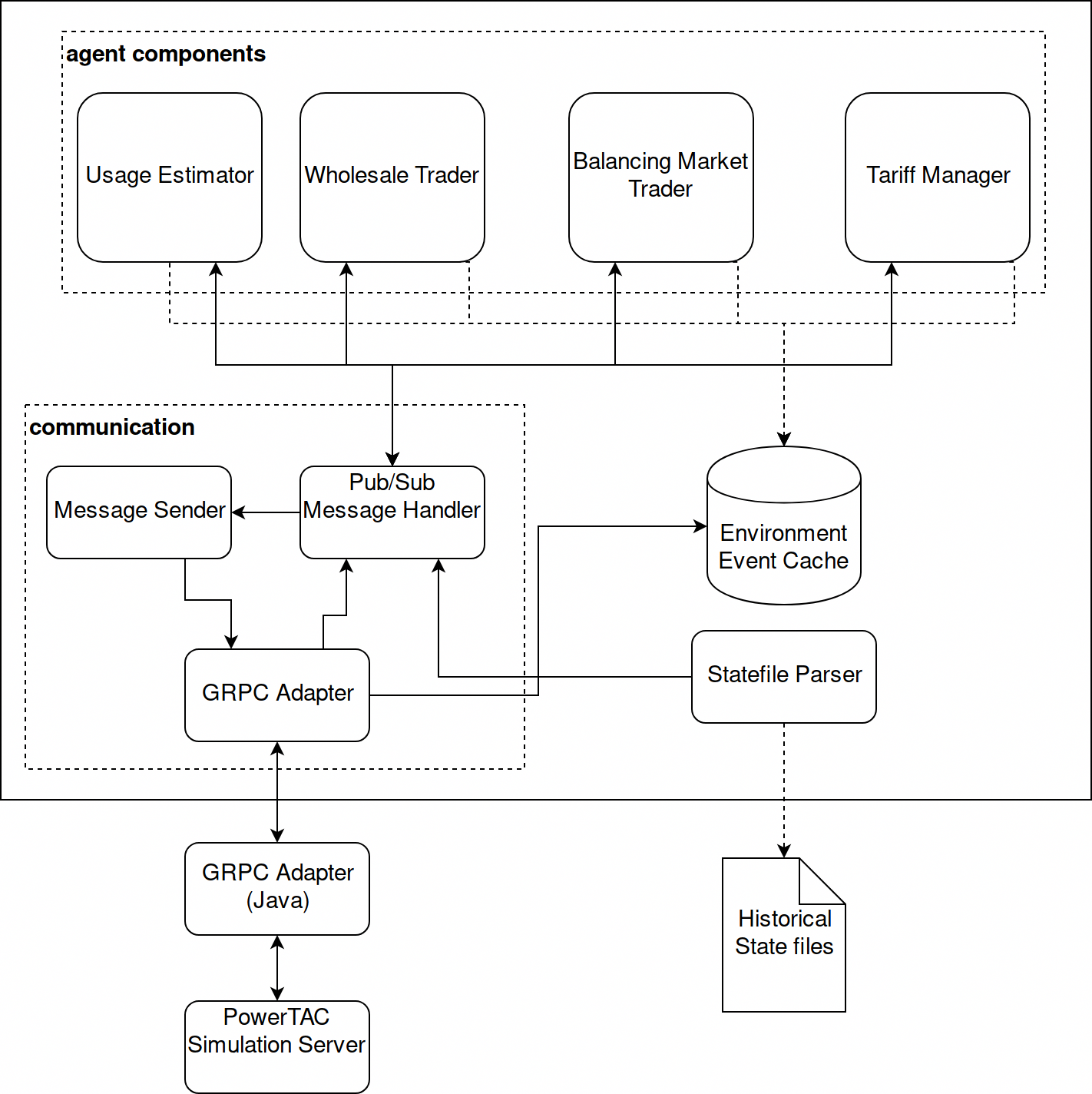
# Implementation

The following chapter will describe the concepts and reasons behind various components needed to allow a broker to leverage modern reinforcement learning tools and algorithm libraries in the PowerTAC environment. Current state-of-the-art algorithms for RL, available mostly in Python , are used. These leverage both the TensorFlow and Keras high-level abstraction library .

The overall architecture of the broker is composed of 5 main components: The communication abstraction, wholesale market agent, balancing market agent, tariff market agent and demand predictor. In this implementation, only the wholesale market and demand predictor are actively making decisions. Future researchers can make use of the component structure however. The architecture is visualized in Figure [[fig:agentframework]](#fig:agentframework).

While have defined the entire simulation as a POMDP (although they interpret it as a MDP for ease of implementation) with all three markets integrated into one problem, I believe breaking the problem into distinct sub-problems is a better approach as each of them can be looked at in separation and a learning algorithm can be applied to improve performance without needing to consider potentially other areas of decision making. A subsequent algorithm could then be trained to perform the same actions as one unified decision making system according to the concepts of *Curriculum Learning* and *Transfer Learning* . Such steps require more advanced forms of machine learning architectures and should therefore be approached in future work. To justify this separation of concerns, I refer to the estimation of fitness for a given tariff in a given environment. A tariffs’ competitiveness in a given environment is independent of the wholesale or balancing trading strategy of the agent since the customers do not care about the profitability of the agent or how often it receives balancing penalties. While the broker might incur large losses if a tariff is too competitive (by offering prices that are below the profitability line of the broker), such a tariff would theoretically be quiet competitive and should therefore be rated as such. The question which of the tariffs to actually offer on the market is a separate problem, that balances competitiveness against profitability. Similar arguments can be made for the other components.

I will first describe a number of tools used in the implementation as well as the preprocessing of the existing data using both new and existing code. Afterwards I will describe the new communication architecture for non-java clients. Finally I will explain the code behind the two implemented learning components, the demand prediction and the wholesale trading agent.



Python Broker framework

## Tools

To develop the functionality of the agent, which is supposed to be mainly driven by deep learning technologies, a number of state-of-the-art tools and frameworks were used. These include Keras and TensorFlow to allow for easy creation and adaption of the learning models, GRPC to communicate with the Java components of the competition and *Click* to create a CLI interface that allows the triggering of various components of the broker.

### TensorFlow and Keras

TensorFlow is a library developed by Google to facilitate machine learning algorithms. It can leverage both CPU and GPU computing power which can significantly increase performance. It is Open Source, used in various technologies and serves as a base technology for many higher level frameworks .

Keras is one of these higher level frameworks that focuses on NN. It offers a intuitive API, oriented towards NN terminology, to quickly develop and iterate on various NN architectures. It integrates TensorFlow and its accompanying UI Tensorboard, which visualizes training, network structure and activation patterns. It also supports other base technologies beside TensorFlow, but these will not be discussed. A simple example for a 2 layer Dense NN written in Keras is shown in Listing [[lst:kerasbasic]](#lst:kerasbasic).

from keras.layers import Dense  
  
model.add(Dense(units=64, activation='relu', input\_dim=100))  
model.add(Dense(units=10, activation='softmax'))  
model.compile(loss='categorical\_crossentropy',  
 optimizer='sgd',  
 metrics=['accuracy'])  
# x\_train and y\_train are Numpy arrays -- just like Scikit  
model.fit(x\_train, y\_train, epochs=5, batch\_size=32)  
loss\_and\_metrics = model.evaluate(x\_test, y\_test, batch\_size=128)

### Tensorforce and kerasl-rl

Tensorforce and kerasl-rl are both relatively young libraries that intend to offer a high-level API for building RL agents. Keras-rl, as the name suggests, is based on the keras library and includes a number of RL agent implementations such as Deep Q-Learning (discrete and continuous) and SARSA . Over the last few months, the progress of the project has been rather slow though and Tensorforce offers a rich alternative. Although not based on the high-level keras library, it offers configuration of architectures and hyperparameters via JSON files. Due to large changes in the TensorFlow library between versions 1.6 and 1.8, it is scheduled to be replaced by a new framework "YARL", but the API will be similar[[5]](#footnote-76). While I initially implemented my RL trials with keral-rl, I quickly switched to tensorforce due to its higher flexibility, better documentation, stronger developer activity and most importantly, because it allows for a reversal of process flow control as is needed for the PowerTAC environment.

### Click

Click allows the creation of CLI interfaces in Python. Programs can be customized with parameters and options as well as structured into sub commands and groups . This allows for patterns such as agent compete –continuous or agent learn demand –model dense –tag v2. An annotated function is shown in Listing [[lst:click\_sample]](#lst:click_sample).

@cli.command()  
@click.argument('component', type=click.Choice(AGENT\_COMPONENTS))  
@click.option('--model', help="omitted in paper")  
@click.option('--tag', help="omitted in paper")  
def learn(component, model, tag):  
 """Triggers the learning of various components  
 off of state files"""  
 if component in cfg.AGENT\_COMPONENTS:  
 component\_configurator = get\_learner\_config(component)  
 component\_configurator.configure(model, tag, True)  
 instance = component\_configurator.get\_instance()  
 instance.learn()

[lst:click\_sample]

### Docker

Docker creates isolated, transferable images that include everything an application requires to run. A container can be based on various distributions and many containers can run on a single server without much overhead. VM technologies are often compared to containers, but VMs abstract on a different layer. A VM simulates an entire operating system on top of a layer called the hypervisor. Docker on the other hand only abstracts the application layer, letting all containers run in the same kernel and therefore makes use of the existing resources in a more efficient way. Nonetheless, it allows the creation of portable infrastructure components. This may be helpful, if brokers become more complex, requiring more technologies, or simply to allow new developers to quickly get started with a competition environment.

### GRPC

GRPC is a remote procedure call framework developed by Google Inc. It allows various languages and technologies to communicate with each other through a common binary format called *protocol buffers* or short *protobuf*. All communication can be encrypted via SSL, offering security and authentication. Over-the-wire data representation can either be binary or JSON . The benefits over the current implementation are described in Section [3.2.1.2](#sub:grpc_based_communication). It is used by many machine learning frameworks to allow distributed learning on many computing nodes .

### MapStruct

MapStruct transfers data between Java objects of different classes. This problem is very common in large software projects where domain objects may be outside the control of the developing team or based on external libraries. If several components need to be integrated, translation is often necessary to adhere to the object structure required by the library. MapStruct offers to generate otherwise manually created code based on best practices and naming conventions. It is compile-time based, generating all code during compile time. This offers better error avoidance and performance compared to alternatives that are reflection based . An example is given in Section [3.2.2](#sub:implementing_the_communication_with_ac_grpc_and_mapstruct).

## Connecting Python agents to PowerTAC

To connect an agent based on Python to the PowerTAC systems, a new adapter was developed. In early 2018, a simple bridge was provided by John Collins, a member of the PowerTAC team. It allowed external processes to communicate with the system through a bridge via the provided sample-broker. All messages received by the broker were written to a First in First Out pipe on the local file system and a second pipe was created to read messages from the external process. This was the first approach towards opening up the simulation to other languages and development environments. Another alternative approach would have been the creation of a function-specific adapter that only calls an external python program to perform specific decisions[[6]](#footnote-84).

Due to my interest in writing my Agent using certain frameworks which are mainly developed and maintained in Python, the adapter was necessary. Creating a complete communication adapter opens the doors for possible later migration of the technology to the server. Using a highly performant technology instead of using XML may also enable future competitions to scale to many more competitors.Generally, the following problems needed to be solved:

* Java model classes, or some central model class, should be reused if possible, automatically generating target language model definitions from the Java source code to avoid duplication of semantically identical information
* Permit future developers using even more languages (such as C, R or Go) with little effort
* Possibly lay the basis for a change of the communication technology of the entire simulation which is more language agnostic and performant

### Evaluating communication alternatives

After researching the current implementation and based on previous development experiences and current best practices, the following three alternatives have been investigated in detail.

#### XML via GRPC

The first approach is quiet similar to the original bridge but instead of writing the XML strings to the local file system, they are passed to the final environment via GRPC by simple messages that just serve as a wrapper for the XML string. While this is not elegant from a engineering perspective (GRPC should be used on a method level and messages should not contain other message formats as strings), it is simple and leads to quick results. A problem is that the resulting XML will then have to be parsed in the Python broker. Before the introduction of other languages, the communication was basically an internal API and broker developers only needed to concern themselves with the handling of the Java handleMessage method. Therefore, no formal descriptions for the structure of the XML messages exist. All XML parsing would therefore be based on observable structures of the XML which can be extracted from the sample-broker logs and all model classes need to be rewritten. Furthermore, agents wanting to use other programming languages would have to reimplement all of this again, with no reuse possible.

#### True GRPC

A better but more complicated approach is based on GRPC to transmit the messages between the Java sample-broker and the final client, hooking into the handleMessage methods in the sample broker. While previous developers have handled these messages in the Java environment, I pass these messages to the ultimate environment by converting them into protobuf messages which are then sent to a connected broker who implements corresponding handler methods in the target language.

The advantage of this approach is that this theoretically allows the maintainers of the project to also adapt this approach for the Java clients in general, massively reducing the communication overhead of XML messages. The over-the-wire protocol is much more efficient (as the data is sent in a binary format) and the message structure is clearly documented in the grpc\_messages.proto file. When serializing a Competition object, XML requires 48 kByte while the protobuf message is 14 kByte large, 70% smaller[[7]](#footnote-89). When looking at the serialization and deserialization performance of XML vs protobufs, a comparison of 1000 iterations of each operation for each variant also shows a significant improvement. While the deserialization of protobuf messages performs about 5x less well (7444ms protobufs, 1366ms XML), the serialization is 44x times faster (1619ms XML, 37ms protobufs)[[8]](#footnote-91). This can be explained by the amount of string handling that XML requires and on the other hand the fact that the deserialization of protobuf messages includes a mapping of the binary format into the proper Java object via MapStruct instead of using reflection. Generally, the server sends more messages than it receives, having to answer most messages and redistribute information to all participants for any public information.

The disadvantage is the need to translate each POJO into a protobuf message and vice versa. This is however not different from the current XStream implementation which also requires the annotation of class files in Java to declare which properties are serialized and included in the XML strings. If the project should adopt the GRPC based communication, the GRPC architecture will then allow the server to be addressed by any of the supported languages[[9]](#footnote-93). Using MapStruct as a mapping tool also makes the mapping structured and by performing round trip tests of the transformed elements, it can be assured that the transformations between protobuf messages and POJO perform as expected[[10]](#footnote-94).

#### JSON schema based communication

A final approach is the generation of schema definitions from the Java model classes that are transmitted between the brokers and the server. This formalizes the currently informal XML API. Generally, two human readable over-the-wire structures are reasonable: XML and JSON. XML messages can be formally defined using XML Schemas and the JAXB project[[11]](#footnote-97) offers to generate such schemas from Java class definitions. This however did not succeed for the PowerTAC model definitions which lead me to create a question on StackOverflow, a discussion platform for programming questions. The resulting answer lead to the ultimate alternative which is the generation of JSON schemas which can then be converted into Python class files[[12]](#footnote-99). The choice of JSON as the base communication protocol might also be intelligent as a future choice two reasons: Firstly, it seems to be the more popular serialization protocol in comparison to XML due to its easy readability and because it is more data efficient. Secondly, GRPC can also transmit data in JSON form and protobuf messages can easily be printed as JSON, making both alternatives more interoperable[[13]](#footnote-101).

### Communicating with GRPC and MapStruct

After adapting the projects scope in response to the mid-thesis coordination with my supervisor, I chose the second approach, the pure GRPC solution. Because the focus will now be on the wholesale market, only a subset of messages need to be mapped. This permits the implementation of a subset of message mappers between the protobuf and PowerTAC entities, reducing the scope while retaining the benefits of a GRPC implementation. Since GRPC is JSON compatible, it appears to be the best choice.

Using MapStruct, all messages required for the wholesale learning component are mapped from the simulation core entities to the protobuf messages. To map classes, a mapper interface is created for each type. Most simple types can automatically be mapped and don’t require any adaption. All properties have been named the exact same way as the properties of the data holding entities in the PowerTAC environment, allowing MapStruct to deduce the corresponding properties to map to. Some properties require custom initiation, more specifically those where the PowerTAC entities don’t follow the bean specification for getters and setters or where getters and setters are simply not available. An example is given in Listing [[lst:mapperexample]](#lst:mapperexample). Mappings are defined with the @Mappings({}) annotation. Complex compositing objects require the other needed Mappers to be defined in the @Mapper(uses = {…}) annotation. Support for protocol buffers in MapStruct is still new and many currently required lines of code may soon be redundant. Because the PowerTAC classes often require a generated ID which cannot be set via any setters, any such object can be forced to adopt the ID provided from the GRPC message via the builderSetId method in the extendable abstract class AbstractPbPtacMapper. This method uses reflection to determine if the target object or any of its parent classes has a private id property and if so, sets it accordingly. This is necessary due to the restrictive property write permissions of most PowerTAC domain objects which is again influenced by Java best practices.

@Mapper(uses = {  
 InstantMapper.class,  
 TimeslotMapper.class,  
 OrderbookOrderMapper.class  
  
})  
@Service  
public abstract class OrderbookMapper{  
  
 @Mappings({})  
 public abstract PBOrderbook.Builder map(Orderbook in);  
  
 @Mappings({})  
 abstract Orderbook map(PBOrderbook in,  
 @MappingTarget Orderbook out);  
  
 PBOrderbook.Builder builder() {  
 return PBOrderbook.newBuilder();  
 }  
  
 @ObjectFactory  
 Orderbook build(PBOrderbook in) {  
 Orderbook out = new Orderbook(in.getTimeslot(),  
 in.getClearingPrice(), new Instant(in.getDateExecuted()));  
 return builderSetId(in, out);  
 }  
}

To ensure the mapping works as expected, the tests for the mapper classes perform a *round trip test*. This takes a Java class as commonly found in the simulation, converts it into XML using the current XStream systems, then performs a translation into protobuf and back. Finally, this resulting object is serialized into XML again and both XML strings are asserted to be equal. By doing this several things are tested at once: Is the translation working as expected, i.e. does it retain all information of the original objects? Is the mapping of IDs to objects still working as expected? Are any values such as dates or time values misrepresented? Are any values missing? The round trip test allows for a generic testing of all object types that covers a large number of possible errors. It also avoids having to rewrite test code for every type conversion.

With an ability to translate Java objects into protobuf messages, those messages now need to be transferred. GRPC offers the ability to transfer protocol buffer objects both as streams and as unary operations. The entire communication overhead between the server and the client is abstracted away from the developer. The messages can therefore simply be sent to the connected python broker code through the GRPC adapter. The integration with the existing code is shown in Listing [[lst:handlemessageexample]](#lst:handlemessageexample).

//...  
@Autowired  
private MarketBootstrapDataMapper marketbootstrapDataMapper;  
  
public synchronized void handleMessage(MarketBootstrapData data)  
{  
 comm.marketStub.handlePBMarketBootstrapData(  
 marketbootstrapDataMapper.map(data).build()  
 );  
}

On the Python side, the messages are now accepted and applied to the brokers knowledge base. This is encapsulated in the env module of the broker as described before. Messages can be considered as action triggers and are therefore shared with all subscribed components through the publish-subscribe event system. A message signaling a completed time slot for example may trigger the broker to learn on the newly observed usage patterns, improve its predictions on the expected usages of its customers and evaluate its next steps in the wholesale trading market.

## Creating Containers from competition components

To run a competition on a local machine, one must install several components: Maven, Java 8 and all of the brokers and their dependencies as well as ones own technology stack. If the scale of this set of components exceeds the local computation power available, the stack needs to be moved to a machine in a server with sufficient computation power or distributed across several machines. While tools like Vagrant allow the configuration and setup of environments to quickly allow new developers to start working with a set of tools in a given project , it requires virtual machines which have significant overhead in comparison to container technologies. Furthermore, these virtual machines do not support GPU accelerated computing, the main platform of modern machine learning. If the competition and its components are abstracted into docker images, tools like Kubernetes or Docker Compose can quickly instantiate a competition on any machine or cluster, given it has enough resources and a docker runtime installed . It also allows broker developers to increase their broker complexity without loosing the ability to easily share it with other developers. One broker already depends on R, a common statistics tool and programming language. A container can abstract this dependency (and any other such dependency) by including it in the distributed image which is self-contained and can be run on any docker host.

Generally, containerized application infrastructures have become quiet popular. Amazon, Google and Microsoft are offering services specifically tailored to host containerized applications and it is easy to share created docker images through the docker hub platform.

To create a Docker image for the server, the Dockerfile listed in Listing [[lst:servertodocker]](#lst:servertodocker) can be used[[14]](#footnote-105). It is also a good practice to run the build in one container and move the created executable artifact into another container which only holds the runtime and the artifact. The *alpine* image type is a light-weight Linux base that only requires about 5Mb of storage. As can be seen, container images and processes are therefore light-weight in comparison to virtual machines.

FROM openjdk:alpine  
LABEL maintainer=pascalwhoop  
LABEL name=powertac-server  
  
WORKDIR /powertac  
RUN mkdir data  
  
COPY bootstrap-data.xml ./  
COPY init.sh ./  
COPY server.properties ./  
#assumes a built server jar is present in the target folder  
COPY target/server-jar-1.5.1-SNAPSHOT.jar server.jar  
  
EXPOSE 8080 61616  
#and start it up  
CMD "/powertac/init.sh"

This offers another advantage that may become increasingly attractive in the long-term: Tools like Kubernetes or Docker Swarm, both being open source enterprise level container management software, seamlessly allow for the creation of 1, 10 or 1000 instances. OpenAI, a deep learning research company, has successfully scaled Kubernetes to 2500 nodes to run their deep RL learning systems . Such scalability can greatly improve the experiment opportunities based on the simulation.

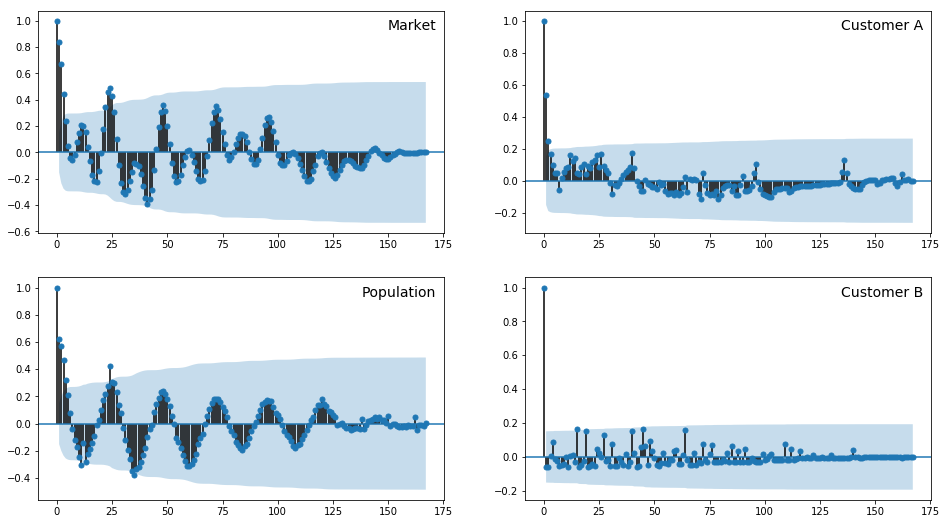
## Inner Python communication

Once the competition environment is running and messages start streaming into the python environment, the various components need to be coordinated. Event driven architectures are light-weight and offer enough flexibility to coordinate the various components. The server may send a number of TariffTransaction messages followed by a TimeslotComplete message that signals the demand estimator it may now calculate new forecasts for any customer subscribed to the broker. Once the component has completed the task, instead of directly calling another component such as the wholesale trader, it sends a signal via the event system so that any subscribed component may react to the event. Because the simulation only accepts messages within a certain time-frame, tardy messages are ignored. Components need to observe the message topics of interest to them and start their processing as soon as they have collected all the necessary information.

To enable possible later extensions to also access events retrospectively, all events are cached in an in-memory event store. For later analysis, all messages can be logged to the local file system either as JSON or in binary format.

## Usage Estimator

The broker needs to predict the amount of energy the customers in its portfolio will require to make good decisions in the wholesale market. When predicting demand, it is helpful to first perform a preliminary analysis of the structure of the demand patterns. This has been done using Jupyter Notebooks and the work can be seen in the notebooks folder in the broker-python code repository[[15]](#footnote-109). All data was generated using the powertac-tools project. Generally, the customer patterns vary widely and it is difficult to predict individual customer usage. While population level demands are rather systematic, customer level look sporadic, random and noisy. When the number of customers increases though, the individual errors partially cancel each other out and the overall predictability increases. Figure [[fig:demandtimelag]](#fig:demandtimelag) shows a clear correlation between the current demand of the market and historical demands with a delay of hours. This correlation degrades and slowly converges towards no correlation. The population models exhibit similarly predictable patterns while the individual customers behave unpredictable.



Lag Plot showing the correlation of the population usage data in relation to the time lag

The analysis also showed large differences in the demand profile of different customers. Some consume several thousand kWh per time slot while most normal consumers only consume small amounts. This however is not directly translatable into tariff market actions because some customers are actually just the population model that simulates many thousand individuals. Usage is not reported as individuals but rather as a sum. These models may create contracts with a number of brokers, breaking the demand of these large models down into several small transactions spread across tariffs.

This section describes the development of the demand estimator using NN technologies. It also describes the issues I ran into when designing this model, which are common issues when designing prediction models using NN.

### Preprocessing existing data

To learn from the large amount of data already available from previous simulations, parsing the state files provided by the simulation is a reasonable approach to boost the ability of several parts of the agent to learn faster. It is especially ideal for the predictor, as it is a supervised problem.

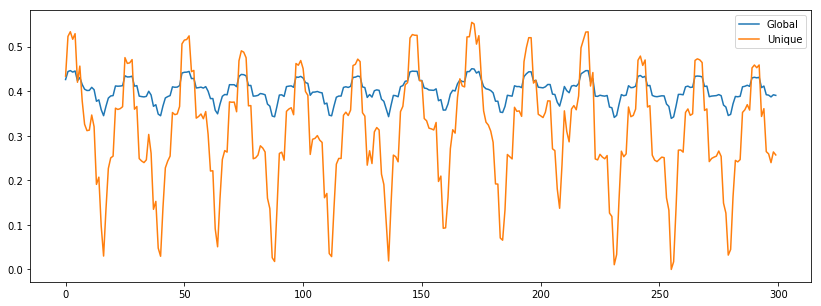
A first approach was to manually parse these files and reconstruct the server state in python. While this was successful for the demand component, I then discovered the powertac-tools repository which holds similar tools based on a combination of python and java components. This repository allows the creation of customer production and consumption information in a comma separated file format. Each tool creates a csv file with a specific focus instead of parsing all information in one central loop. The initial approach to parse the files was therefore scrapped and replaced with this prebuilt variant that makes use of the powertac-server source code.

While the current demand prediction is solely based on historical demand, this can easily extended (as it has been in the python only approach previously mentioned) with weather data, time information and up-to-date tariff information[[16]](#footnote-113).

### Model design phase

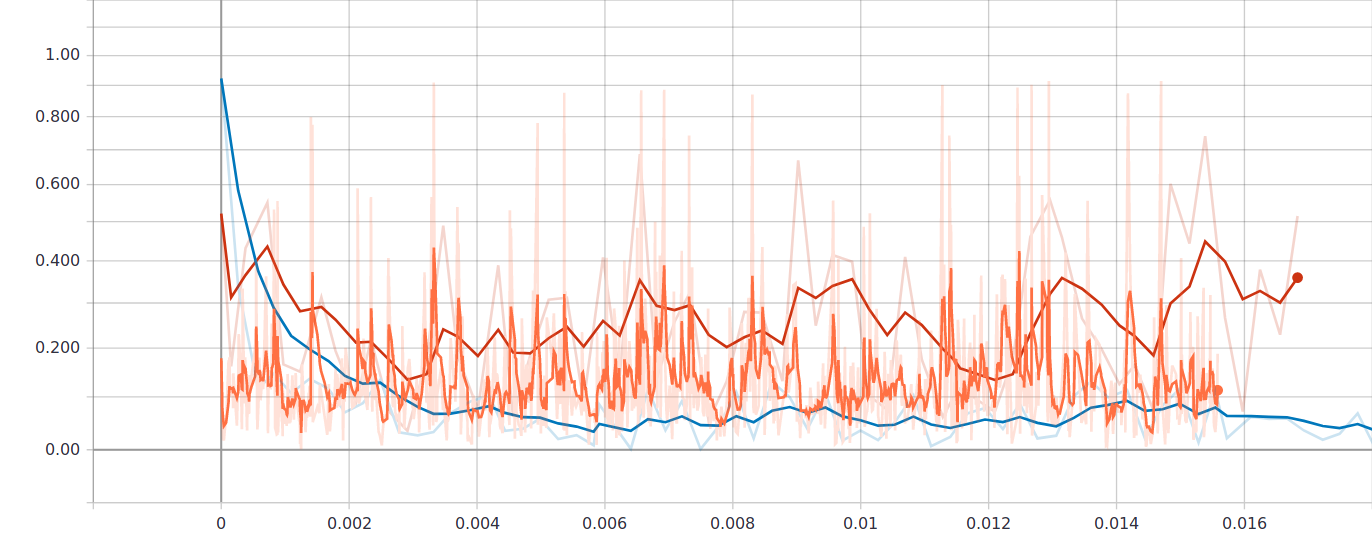
From a dependency perspective, this component has no dependencies onto the other learning components and can easily be trained using historical data. It is therefore a supervised learning algorithm, matching known information in time slot to a prediction for the expected energy usage at time slot . Because there are several games on record, the historical realized usages are the labels for the supervised learning problem. Known information includes: Weather forecasts, historical usages, time, tariff and customer metadata. A simplest approach and therefore a baseline to measure against is to predict the customer to consume the same amount as in the previous time slot. All demand prediction loss is measured in mean squared error.

Several standard architectures were tried, however none significantly outperformed a simple heuristic approach such as guessing the demand to be equivalent to that of 24 hours ago. A later analysis[[17]](#footnote-116) revealed why this occurs. The neural networks architectures are having trouble handling both very large and very small customer patterns. When training a neural network for each customer individually, the performance is much higher. This intuitively leads to the idea that creating a model for each customer may lead to a performance boost. This however creates a range of other complications. Most frameworks are written based on the assumption that one neural network is trained per machine. Most neural networks are only limited by the amount of hardware that is available as well as the data that it may learn from. Having some 200+ neural networks learning from data and predicting in parallel, all in under 5 seconds per time slot is hard to achieve on conventional hardware. A test run showed that the system can only process one customer every 3 seconds. While this does include the creation, compilation and fitting of the model, it is near impossible to reduce this number enough to be able to handle all customers within the initial 5 seconds of the game. It was therefore necessary to either add significant amount of processing power to the broker or somehow get one model to predict all kinds of customer classes with decent precision. One improvement was the creation of a separate preprocessing scaler for each customer. While the previous approach used one MinMaxScaler, a scaler that scales all values between two limits, usually 0 and 1, the second approach was the creation of a scaler for each customer, to allow for a individual scaling across customers. The difference is visualized in Figure [[fig:imgfrosty]](#fig:imgfrosty) and it allows for clearer signals to be received by the network.



Scaled with unique scaler (yellow) or global (blue)

Ultimately, the predictor model that seemed most successful in terms of average error and robustness against repetitive training against various types of usage patterns was a dense vanilla feed-forward NN with 168,100,100,50,50,24 units, using stochastic gradient descent as its optimization function and mostly using ReLu activation function except for the last layer which is a linear activation. To overcome the problem of catastrophic forgetting, a common phenomenon observed when networks have been trained on several tasks in sequence , all input data is shuffled instead of processed in sequence per customer. While this does not keep the network from forgetting, it makes it forget learned knowledge about patterns uniformly, replacing the previous weights with newly learned weights from newly observed patterns uniformly. Results for this predictor are shown in Figure [[fig:imgcombined\_model]](#fig:imgcombined_model), a comparison with the baseline and more examples of various customer types of -24h are given in the appendix. It is visible from the figure that the model has learned to predict regular spikes. It doesn’t seem to understand that there is a natural maximum to the usage pattern, which is understandable for a continuous model. It also doesn’t capture the reduced usage on every 7th day as can be seen via the flat hump in the brown realized curve. An LSTM model is usually considered to be successful with these kinds if problems but my experiments did not succeed. A comparison between the baseline, a vanilla feed-forward and an LSTM model is sown in Figure [[fig:baseline\_dense]](#fig:baseline_dense)



Demand baselines and models, -24h baseline: orange, lstm: red, dense: blue

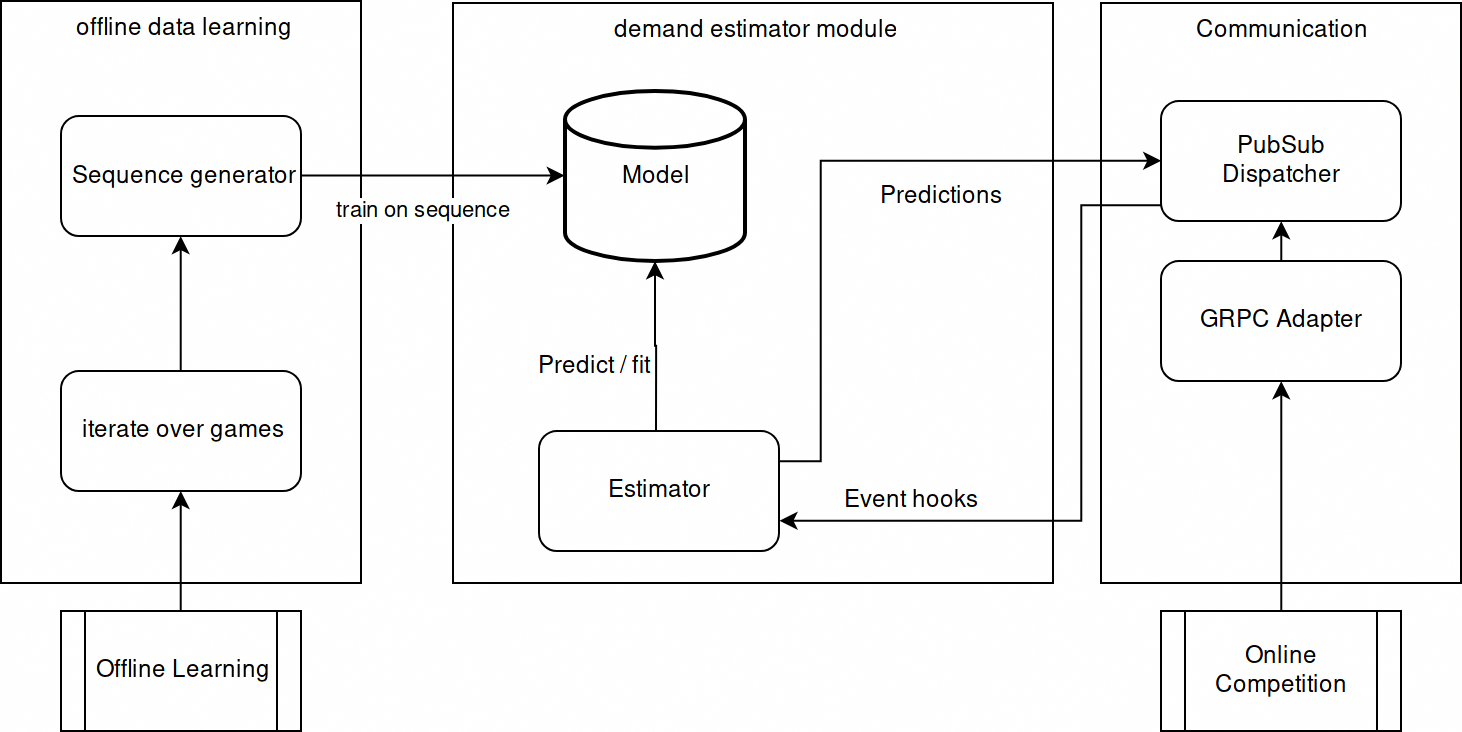
[fig:baseline\_dense]

The timing of this unified model is also acceptable with 3 to 4 ms per 24h forecast required, adding up to a 800 ms delay for predicting 200 customers, which would almost cover the entire games customer base.



Plotting of forecasts and realized usage

[fig:imgcombined\_model]



Demand Estimator structure

### Integrating the model into the python broker

Once the general concept of the learning model was decent, the model had to be integrated into the broker framework and a surrounding architecture had to be built to allow this network to both learn and predict live in a game setting. The overall structure of the demand estimator component is shown in Figure [[fig:DemandEstimator]](#fig:DemandEstimator). The model can be both trained offline based on the state files as well as online during the competition. This is possible because in both situations, the environment model of the agent is a continuous representation of the agents knowledge about the world. In fact, during the state file parsing, the environment may even hold information that the agent usually cannot observe in a competition environment. This is also the case for the demand learning, as the state files hold the demand realizations of all customers while the server during the competition only transmits the usage realizations of the customers that are subscribed to the agents tariffs. Regardless, this does not affect the ability to learn from the customers usage patterns in either setting.

During a competition, the agent may learn from the realized usage of customers after each time slot is completed. The server transmits TariffTransaction objects for each time slot that hold the energy usage of the subscribed subset of all customers of each customer model. To avoid mismatching predictions, these subset usages are scaled up to the whole population count for the prediction step. Afterwards, the values are scaled back down to the actual subscription partition of the customer model.

Because the process of learning from newly observed data may require some resources, it is advantageous to first perform the prediction of the subscribed customers demands for the current time slot to pass this information to the wholesale component before training the model on the received meter readings. While the broker is waiting for the server to process a step in the game, it can perform any learning on newly received information[[18]](#footnote-124). A sketch of the core loop is shown in Listing [[lst:estimatorpseudo]](#lst:estimatorpseudo).

handle\_timeslot():  
 X,Y,X\_PRED = prep\_data()  
 if not X or Y or X\_PRED:  
 return  
 pred = model.predict(X\_PRED)  
 pred = scaler.inverse\_transform(pred)  
 pred = correct\_customer\_count(pred)  
 # dispatch to pubsub  
 dispatcher.send(pred)  
 model.fit(X,Y)

## Wholesale Market

To approach the wholesale trading problem, a subset of the definition of the trading problem developed by is assumed. More specifically, the agent only concerns itself with the activities in the wholesale market and does not act or evaluate tariff market or balancing market activities. This is due to the separation of concern approach described earlier. It is therefore a MDP that can be solved with RL techniques. The goal was the ability to apply current and future NN implementations to the PowerTAC problem set. For this, many of the previously described implementations were necessary. Now that a Python based broker is possible, application of PPO, DQN and other modern RL agent implementations seems reasonable. All required messages can be subscribed to via the publish-subscribe pattern. What is missing are the following components which are explained in detail in this section:

* A mapping from the 24 parallel environments to a single MDP environment
* A correction of the common paradigm where the agent is in control of the program flow
* A solution to the problem that one agent is supposed to be in control of and learn from several environments in parallel
* A way to learn quickly from offline data
* Suitable reward functions
* Input preprocessing
* An initial implementation of a RL agent using modern deep NN frameworks

### MDP design

#### MDP design comparison

There are two possible ways of modelling the MDP: Per time slot or per game. Per time slot is aligned to the definition by . Per game considers each game a unified MDP where the agent acts in all time slots and therefore has an action space of 48 values per time slot.

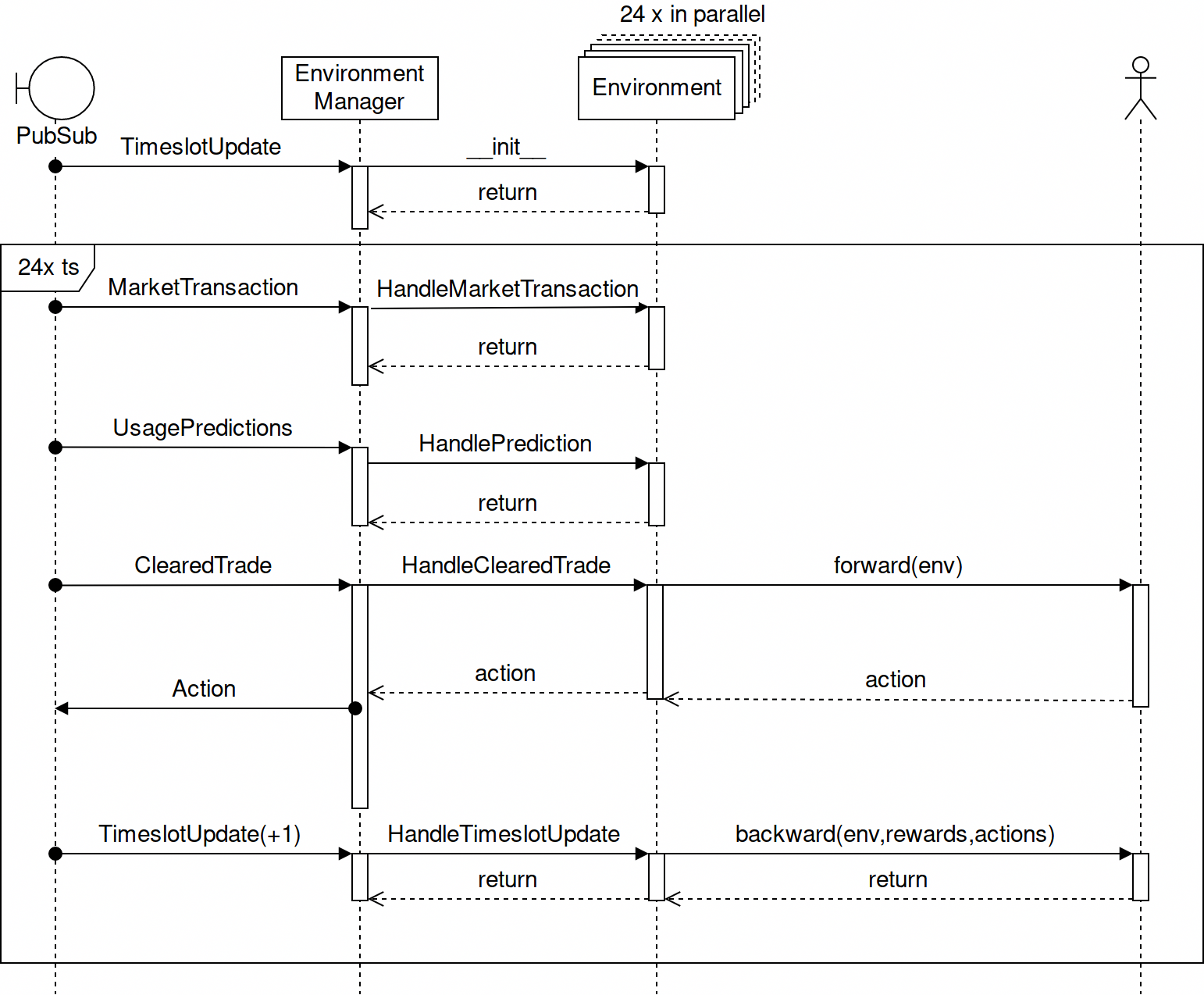
Both approaches have advantages and disadvantages. The former creates short, fixed-length episodes that more closely match the concepts of contemporary RL problems such as the locomotion examples described in Chapter [2](#cha:background). However, because PowerTAC allows for trading up to 24 hours into the future, 24 environments would have to be stepped in parallel. Approaches for parallel asynchronous stepping of multiple environments with a NN based policy function approximator exist but require more complex architectures that update a central policy function based on experiences from all environments. The latter avoids this, allowing a fairly simple off-the-shelf algorithm to be applied to the problem. Problems appear with the compatibility to the action spaces this agent requires as well as the increased signal noise. Common algorithms such as DQN, SARSA or A3C are not easily applied to such large action spaces. They are written to be applied to discrete action spaces . PowerTAC trading is in its purest form a continuous action space, allowing the agent to define both amount and price for a target time slot. Furthermore, the agent would observe 24 environments in parallel and generate 24 largely independent trading decisions. The network would have to learn to match each input block to an output action, as the input for time slot 370 has little effect on the action that should be taken in time slot 380. In a separated MDP, each environment observation would only hold the data needed for the specific time slot rather than information about earlier and later slots as well.

#### MDP implementation

After experimenting with the concept of considering the entire simulation as a single MDP [[19]](#footnote-130) but not seeing any successful learning, the separated approach was chosen.

To separate the messaging from the MDP logic, as well as to separate the 24 environments parallelism complexity from the individual MDP, several layers of abstraction were introduced. First, all relevant messages are subscribed to in the WholesaleEnvironmentManager using the publish subscribe pattern. Individual messages are then passed along to the corresponding active MDP and new environments are created for every newly activated time slot. The WholesaleEnvironmentManager therefore abstracts the multiplicity complexity from the individual PowerTacEnv.

The individual MDP environments receive a reference to the RL agent during creation so that they can pass their observations to it and request actions as well as trigger learning cycles on received rewards. This means that each individual MDP is not aware of other instances. While this reduces complexity, it also hinders the ability of the learning agent to consider its impact of trading in time slot on any future time slots. The message flow is depicted in Figure [[fig:ws\_msg\_flow]](#fig:ws_msg_flow).



Wholesale component message flow

[fig:ws\_msg\_flow]

### Reversal of program flow control

The environment expects the agent to expose an API that includes two calls: forward and backward. This pattern has been adopted from the keras-rl and tensorforce libraries. The reason is simple: While most libraries put the agent control of the program flow, the PowerTAC broker will be stepped by the server and therefore the RL agent itself has no control of the flow. The forward and backward methods are directly aligned with the keras-rl framework and easily applicable to the tensorforce act() and atomic\_observe() methods of their agent implementations. The abstract PowerTacWholesaleAgent class just defines a few methods that need to be implemented by a developer to create a new algorithm for the wholesale trading scenario. In my case, I created the TensorforceAgent class which holds several configurations for a number of architectures. I also created a BaselineAgent which simply trades the prediction energy amount for generous market prices. This is useful to compare performance of a learned algorithm with a very intuitive trading scheme and to serve, as the name suggests, as a baseline.

The reversal of flow control has another benefit: While other approaches had to create specific designs that allow for several agents to act in several environments with one *main agent* to adopt those network changes, this is not required because the environments are calling the agent. The environments are all using one agent instance to learn and act and the environment manager is triggering the environments in sequence. discuss the benefits and drawbacks of experience replay based learning and their asynchronous parallel stepping approach which allows on-policy learning algorithms. In my implementation, all environments first request an action and then trigger the backward function that triggers the agents learning process. While this is currently not batched and therefore leads to a changed policy after the first update has been completed, it could theoretically be batched into one learning policy update, enabling on-policy algorithms to be used. The problem of correlated states and non-stationarity that was first solved by experience replay in is solved similarly to what describe, i.e. by having several environments updating the agent, decorrelating the sequence of observation-action inputs.

### Learning from historical data

Learning quickly from historical data may be facilitated for off-policy algorithms that can learn from historical records of other agents or, if the approach introduced in Section [2.4.3.1](#ssub:offline_record_based_wholesale_environment_approximation) is applied, also for on-policy algorithms. To allow reuse of existing algorithm implementations, I wrote the LogEnvManagerAdapter.py class which parses historical files and sends events via the event dispatcher as if they originated from the server. This way, the same code may be used to train online in a competition or offline from historical data. It iterates over the existing game logs and generates both forecasts for the RL agent as well as all necessary events that trigger the RL agent. The forecasts can optionally be based on the actual demand estimator or they can be the arbitrarily noisy real values. This means, the agent can be trained with perfect predictions all the way to very bad predictions.

### Reward functions

Well crafted reward functions are elementary for any non-trivial RL environment. While the Atari agents often receive their reward directly from the game as many games include a game point counter , PowerTAC technically simulates a real-world energy market which means the score equals the brokers profit. But the profit is dependent on a number of factors and therefore hardly a good choice for a reward proxy. Using the purchase prices of the energy purchased is also noisy, as it depends on the supply and demand of the entire market. Generally, the broker attempts to purchase energy at a good price and a good price can be defined as one that is better than that of other participants in the market. A reward function based on the relation between the average price paid by the broker and the average price paid by the overall market therefore describes how well the agent did in comparison to the others and therefore removes the market price fluctuation noise from the reward values.

To calculate this reward, all the purchases of the agent as well as all market clearings are averaged for a given target time slot.

Defines the reward, where is determined by

for both the market averages and the broker averages. This encourages the agent to buy for low prices and sell for high prices where possible. is the net purchasing amount after the 24 trading opportunities are completed, i.e. did the broker end up with a positive or negative net flow of energy in the wholesale market. This reward function has one one immediate drawback: It can only be calculated once the market for the target time slot is closed. The agent therefore doesn’t get any feedback during any step except the terminal state.

While RL research has stated sparse reward as a core part of RL, many of the recent algorithms do not deal well with such sparse rewards. Experience replay partially works so well in the Atari domain due to the dense reward structure of the domain, allowing randomly selected transitions from the replay buffer to hold information for the agent at any stage of the learning phase. To improve information density in the powertac environment it may be beneficial to provide further feedback to the trader agent. The wholesale trader gets a prediction for a target time slot at every of the 24 slots prior to the target. These predictions come from a specialized demand predictor component and the wholesale trader would do well to trust this prediction to some degree. It may therefore be rational to argue that a good wholesale trader does well in buying sufficient energy for the target time slot to ensure its portfolio is balanced. The reward function may therefore be extended by a term that punishes large deviations from the predicted required amounts.

The final reward is now a combination of those two reward terms. The first reward function puts emphasis on purchasing energy for a good price, no matter how the agent purchases it (e.g. by buying early and selling later for higher prices) while the second puts emphasis on purchasing energy in accordance with the portfolio predictions. A final factor may be introduced that can be changed throughout the course of the learning that decides the weight of these two terms.

This function has another benefit that became obvious during the experiments: If the offline trading approximation assumes the broker has no influence on the market price and if the reward function does not punish large orders, the broker quickly starts ordering energy several orders of magnitude larger than the overall market size. It basically learned that there is virtually free leverage to use to make use of the market price fluctuations. By adding the prediction as a limiting factor, the agent is encouraged to not try and trade absurdly large amounts of energy but to simply trade amounts that match its demand. This flaw is due to the way the offline data based environment approximation determines the closing prices which don’t depend on the agents orders. This is different from the real wholesale market where the price is influenced by any market participant.

Other reward functions are present in the reward\_functions.py file such as an automatically adjusting one that punishes balancing strongly at first and disregards the price but shifts towards the price based reward using a factor similar to above once the balancing amounts are reduced.

### Input preprocessing

### Tensorforce agent

### Agent design experimentation

The RL agent implementation is responsible for preprocessing the observation data. This enables developers to act on more or less information according to their chosen technology without having to remodify the MDP code or the WholesaleEnvironmentManager. The agents forward and backward functions both take the entire known data of the MDP as parameters. In the agent, the observation data gets reduced and normalized to avoid common issues with NN such as slow learning or gradient explosions.

The tensorforce implementation (which is my main attempt to implement a successful NN based RL trading agent) expects the previous 168 time slot price averages, all prior forecasts as well as all prior purchases for the target time slot. In a first attempt, these 216 values were flattened into a one-dimensional input array and fed to the agent as an observation. Without any preprocessing, this implementation was not able to converge towards a good reward that indicated learning progress.

#### SOME NOTES

- large part is both the choosing of the input and how to normalize it so the agent can work with it. - framework allows passing agent anything (the entire env) and then the individual agent can select and preprocess as it sees fit - utility functions hold cross-agent-impl preprocessing tools - started with offline learning to increase development turnaround rate. Simulation assumes the agent doesn’t influence the prices of the market, clearing is just dependent on the action of the agent and the market price that is recorded. - tried intuitive agent impl but didnt work: some environment data input and output is the direct action - next tried output action being mapped: relative to forecast / current market prices > didn’t work - next tried simple "two armed bandit": random action or on spot like the prediction with generous money offering. - >> worked after a few hundred steps - TODO: try bandit style: agent just learns what price to bid, not how much. - TODO: "walk backwards" from bandit to continuous action space - TODO: try with more input types / preprocess better - TODO: draw.io graphic on wholesale components

- need to solve the multiple MDP for one agent problem - no off the shelf algorithms that do continuous multi agent mdp stuff - applying NAF / DDPG to problem possible. But may need to rewrite NAF agent myself. Take stuff from Keras though. - doing a simple "always order demand prediction" baseline should be helpful

# Conclusion

In the beginning of this work, I have described the research progress in AI and how new NN based systems are able to solve problems, previously only solvable by humans. I have also indicated the obvious incentive to apply these new techniques to important contemporary problems, focusing on future energy markets as one such problem. PowerTAC, a simulation of these markets was introduced as a core research initiative that attempts to explore this problem field by letting many researchers compete in a competitive setting and simulate profit-oriented market participants. Because future market participants will not shy away from using AI technologies to improve their competitiveness, PowerTAC contenders must also be enabled to pursue these technologies to realistically represent future markets. To enable new competition participants to quickly catch up and to generally enable the use of NN technologies, I decided to pursue the question whether imitation based RL may be deployable in the PowerTAC setting.

When reviewing my work, it is obvious that I have not fully answered my original research question. Several researchers have shown that learning from other agents is possible with NN in RL environments as described in Chapter [2](#cha:background). During the course of my thesis, new research by has also shown how significant learning performance can be improved when allowing new agent implementations to learn from existing agents. To adapt the PowerTAC environment to a state where these research results can be applied however, required large amounts of software engineering work. This caused me to not reach the ultimate goal, the application of these SOTA algorithms to the PowerTAC wholesale trading environment. While I have not answered the original question, I have contributed a large amount of the work required to make answering it possible.

PowerTAC uses different base technologies than contemporary RL research problems and I have extended this to Python and other languages through the GRPC message adapter. This new communication layer also offers some significant performance improvements, reducing the over-the-wire size by 70% and making serialization of objects 44x faster.

RL agents require many trials to converge towards a useful policy and through the historical data MDP approximation as well as the containerization of the PowerTAC components, I have made it easier and more efficient to quickly train an RL agent for several thousand steps. The container abstraction allows for an easy instantiating of several competitions at once with different or equal configuration parameters. It also allows for easy portability of competing brokers.

Participants can add a number of technologies to their docker image as well as binary files such as NN weights and configurations. This allows the entire competition to expand beyond the realms of Java without placing a burden on other teams to manage not only their own dependencies but also those of the other brokers. It also permits the simulation organizers to easily host the entire competition runtime on a central server cluster instead of every team connecting to this server remotely which may avoid many of the current complexity sources such as connectivity issues and mandatory time synchronization.

The concept of counterfactual analysis, which has been described conceptually but not further described in the thesis has been shown to work with the PowerTAC environment[[20]](#footnote-141) and an adapted PowerTAC server[[21]](#footnote-143) allows a broker to control this behavior to simulate such a counterfactual scenario.

Finally, my Python based broker implementation may be used as a base for future developers that wish to also use these new technologies in their brokers. It serves as base implementation that may be extended and improved by others to build better and more sophisticated brokers that make use of all the AI technologies that are continuously created by researchers. I have shown the ability to build a broker using TensorFlow, Keras and TensorForce technologies. The broker, albeit not exhibiting outstanding performance yet, acts based on decisions derived from NN based RL policies and usage predictors. Clearly, a lot of work remains to be done to see if these technologies can exceed current performances.

Future research may now look at using the NN technologies of recent years to compete in the PowerTAC competition. The PowerTAC competition server itself may be adapted to incorporate the GRPC based communication as a secondary protocol available to brokers. This would eliminate the need for the intermediate adapter. Because NN are able to incorporate large input data into their functions, all components of a broker may now make use of a larger number of input dimensions to improve their performance. The initial demand predictor already showed promising results, completely ignoring weather, customer metadata, market data etc.

In summary, this work offers a large contribution to bringing together the socially and economically important field of energy markets and recent developments in AI research. While a breakthrough has not been achieved, nothing suggests that future work won’t be able to show great success with the path laid out. NN keep succeeding in a variety of contexts and smart energy markets will not succeed without smart participants and components, carefully embedded in a market model that incentivizes everyone to cooperate in a way that benefits the population as a whole.

# Digital resources

Attached to the thesis is a data medium that holds all cited sources, developed and used source code, graphics and analyses using e.g. Jupyter Notebooks. Below is an overview of each folder present on the disk with a short description of its contents.

README.html

is a document holding information about the additional information contained on the DVD as well as links to further information

analysis

Holds all Jupyter Notebooks. See them also directly on <https://github.com/pascalwhoop/broker-python/tree/master/notebooks>.

code\_and\_data

contains a 2.4GB zip file that contains all source code used and developed. It’s a collection of PowerTAC project folders as well as my own projects. It’s a direct zip of my file system and therefore contains .git directories and links to the upstream GitHub repositories. It also contains all data used to train the demand predictor and the wholesale agent offline. This zip expands to over 16GB.

graphics

contains a number of generated graphics that was used to better understand competitor agents, the overall dynamics of the game and other information that can be best grasped when visualized.

sources

contains all papers, books and websites that were used to write the thesis

thesis

contains all sources and the rendered file of the main thesis document

others

contains anything that didn’t match the aforementioned categories

1. in newer RL literature this is also called a *trajectory* [↑](#footnote-ref-42)
2. While the 2017 competition technically allowed for brokers to remain in the game despite offering highly under-priced tariffs that corrupted the simulation results, a proper broker must not pursue such strategies simply due to economic reasoning. [↑](#footnote-ref-56)
3. CustomerProductionConsumption.java [↑](#footnote-ref-60)
4. MktPriceStats.java [↑](#footnote-ref-61)
5. Information received during private correspondence with the authors, see also <https://github.com/pascalwhoop/broker-python/issues/5> [↑](#footnote-ref-76)
6. A comparing discussion can be found at <https://github.com/powertac/powertac-server/issues/974> [↑](#footnote-ref-84)
7. <https://github.com/pascalwhoop/grpc-adapter/blob/master/adapter/src/test/java/org/powertac/grpc/mappers/CompetitionMapperTest.java#L64> [↑](#footnote-ref-89)
8. <https://github.com/pascalwhoop/grpc-adapter/blob/master/adapter/src/test/java/org/powertac/grpc/mappers/CompetitionMapperTest.java#L90> [↑](#footnote-ref-91)
9. Which as of today are: C++, Java, Python, Go, Ruby, C#, Node.js, PHP and Dart [↑](#footnote-ref-93)
10. <https://github.com/pascalwhoop/grpc-adapter/blob/master/adapter/src/test/java/org/powertac/grpc/mappers/AbstractMapperTest.java#L54> [↑](#footnote-ref-94)
11. <https://github.com/javaee/jaxb-v2> [↑](#footnote-ref-97)
12. <https://stackoverflow.com/questions/49630662/convert-java-class-structures-to-python-classes/49777613#49777613> [↑](#footnote-ref-99)
13. <https://github.com/powertac/broker-adapter-grpc> [↑](#footnote-ref-101)
14. All resources regarding the container technologies can be found under <https://github.com/pascalwhoop/powertac-kubernetes> [↑](#footnote-ref-105)
15. <https://github.com/pascalwhoop/broker-python/blob/master/notebooks> [↑](#footnote-ref-109)
16. All preprocessing code has been deleted in commit [c54ee7c](https://github.com/pascalwhoop/broker-python/commit/c54ee7c05585d15462f40e2be6850343e8aea27a) in the broker-python repository. [↑](#footnote-ref-113)
17. see Jupyter notebooks for Demand Estimator [↑](#footnote-ref-116)
18. The component code can be found under <https://github.com/pascalwhoop/broker-python/tree/master/agent_components/demand> [↑](#footnote-ref-124)
19. <https://github.com/pascalwhoop/broker-python/blob/5876c2d5044102d3fbff4bde48b5febfdb15a84f/agent_components/wholesale/mdp.py> [↑](#footnote-ref-130)
20. <https://www.youtube.com/watch?v=10iqP9Zdi4U> [↑](#footnote-ref-141)
21. <https://github.com/pascalwhoop/powertac-server/commit/1f0455929e7062faf8617cd5ca6e6a138f250382> [↑](#footnote-ref-143)