Introduction

In recent years, machine learning, and in particular its subset deep learning, has found great success across a variety of domains across the natural sciences and has driven progress in problems previously believed to be unsolvable: deep learning powers DeepMind's MuZero, a general program able to learn and fully master the games of go, shogi, chess, and a suite of classic Atari games[12], as well as DeepMind's protein folding deep learning system AlphaFold, which was able to achieve atomic accuracy in protein structures, even if no similar structure is known[7]; Tesla and Waymo, among other groups, apply deep learning to achieve fully autonomous vehicles; and OpenAI's Codex is able to convert effectively translate natural language to code[3].

The power of machine learning and deep learning to solve complex tasks extends to one of the most pressing issues facing modern society: energy generation and management. As instantly switching from all polluting energy sources to completely renewable energy sources is infeasible, one way to mitigate the pollution produced in the process of generating energy from sources like coal or oil is to attempt to minimize the amount of power that needs to be generated from these sources. The major dilemma facing grid operators in solving this problem is that enough energy always needs to be available for consumers to use, and excess energy can't always be effectively or efficiently stored. Thus, algorithms that are able to accurately predict how much energy consumers will use, as well as how much energy will be able to be generated from sources where the amount of energy can change on the whim of the environment, like solar and wind, are crucial towards maximizing energy production efficiency and minimizing pollution. Xcel Energy, the utility firm with the highest total wind capacity in the United States, for example, was able to save consumers \$60 million and reduce annual CO_2 emissions from fossil-reserve power generation by a quarter million tonnes due to better power forecasting[11]. DeepMind has also previously conducted research in this area, and researchers were able to increase the value of wind energy produced by over 20% by applying machine learning to more accurately estimate the amount of energy that wind farms would produce in the following 36 hours[8].

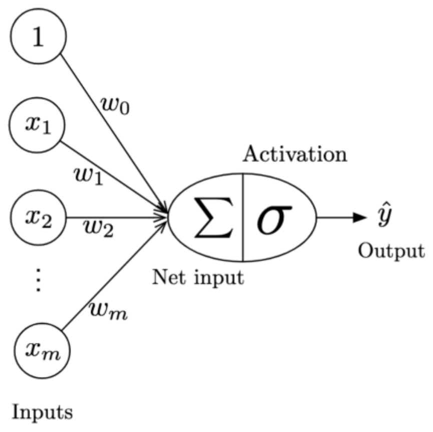
Moreover, saving energy and being more efficient with better power forecasting is not limited to power grids. Google researchers were able to develop machine learning models that could predict power usage effectiveness within a 0.4% error at Google data centers based on parameters of how the plant was run[5]. With these models, Google was able to optimize its data centers to be more power efficient, saving costs and overall energy use.

Seeing the importance of this issue, we wanted to see if machine learning could be applied to estimate the overall energy demand at a given time to assist in creating better overall efficiency in meeting power demand, as well as in reducing costs.

Background Information/Related Work

The issue of estimating the energy demand at a given moment is one of regression analysis. Various types of machine learning models can be applied to regression models, including the single-layer perceptrons, multi-layer perceptrons, and recurrent neural networks, all trained with gradient-based optimization.

All of these models fundamentally rely on one basic structure: the perceptron.



Schematic of a single-layer perceptron[14]

Single-layer perceptrons have only one node; multilayer perceptrons have multiple layers, each with one or more nodes each connected to each of the nodes in the next layer; and recurrent neural networks have multiple multilayer perceptrons that get passed to each other, as explained later.

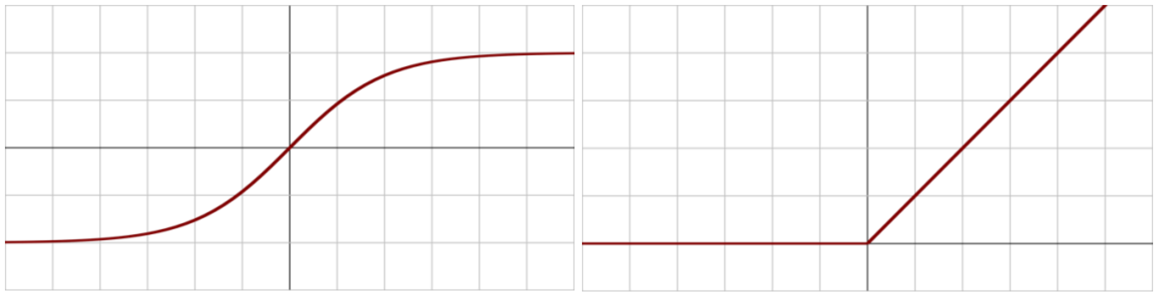
The node functions very similarly to a linear transformation. Each node is connected to each of the outputs in the previous layer or to the initial inputs to the overall machine learning model, and each connection has a weight assigned to it. The values in the previous layers are multiplied by their respective weights, and then summed together to form a single number, the net input. This net input is then added to a variable constant called the bias, and then fed through a so-called “activation” function to produce the final output[14].

The job of the machine, then, is to find the particular set of weights and biases for each of the nodes that produce the most accurate output of the model.

Note that this process modeled by the node follows the linear transformation because it is simply a version of the equation $y = mx + b$, only with more terminology and with a final activation function added.

Activation functions are necessary to the functioning of complex machine learning models because note that if the model only had the linear components of $y = mx + b$, multiple nodes connected in multiple layers to each other would essentially function the same as a single linear node: for example, if there was a model composed of only two nodes in two layers, and node 1 in the first layer applied the transformation $y_1 = m_1x + b_1$, and node 2 in following layer applied the transformation $y_2 = m_2x + b_2$, then the model could easily be simplified to $y_2 = m_2(m_1x + b_1) + b_2 = m_1m_2x + (m_2b_1 + b_2)$, which could have been trained on just a single linear node. The activation functions provide a nonlinearity to the model which allows it to learn more complicated algorithms beyond that of a linear model.

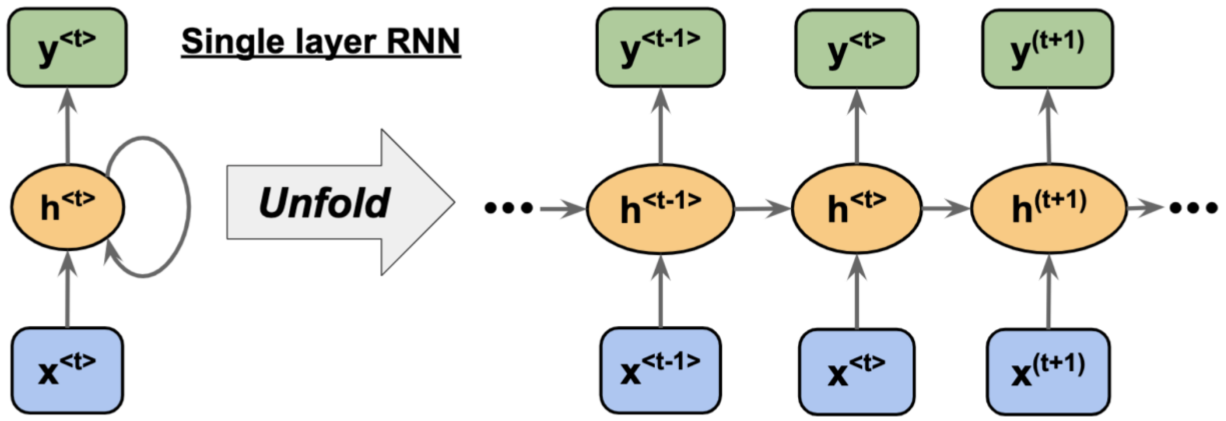
Typical examples of activation functions include the rectifier, also called the rectified linear unit (ReLU), and the hyperbolic tangent function.



Left: hyperbolic tangent function ($\tanh x$)[2]; right: rectified linear unit (ReLU)[1]

Recurrent Neural Networks

Recurrent neural networks (RNNs) are a special class of neural networks specifically designed to handle datasets where the data is continuous, and the previous samples of the data can have an influence on the next. RNNs have applications in natural language processing, stock data, or in our situation, possibly processing energy data[13].



Schematic of a single-layer RNN[13]

RNNs are able to make informed decisions based on previous data in a continuous dataset because they take in not only the inputs for one particular point, but also an embedded version of the data accumulated from all of the previous samples.

A sample is fed forward through a neural network to produce an intermediate embedded representation of the point, which is then added to the intermediate embedded representation of the previous point, which is then fed through another neural network that produces the final output. In the case of the first sample, the embedded representation is usually set to be all zeroes.

With RNNs, contextual information from previous points can be fed through this embedded representation to the current point to make an informed decision about the final output. However, there still are some flaws with this structure: over time, more and more information from samples from further back in the data will be lost in the representation, so if the information at the beginning of the data can strongly influence the output from samples much later on, this proposed RNN model may not be a good fit for it.

In the case of the energy dataset this study investigates, an RNN may be helpful, but realistically speaking, an RNN may be excessive because the most important samples necessary to predict the current output will always be the samples directly before it, so a simple multilayer perceptron is all that is necessary.

Datasets

In order for the model to be useful for any sort of purpose, the model needs to be able to accurately predict a piece of data, given some other set of data.

A dataset is composed of many different samples, each with their own values for a set of features. For example, a dog dataset might have many different samples with the features of the dog's breed, age, weight, and height.

The objective of a supervised machine learning model is to predict a specific feature based on another set of features. The feature to be predicted by the model is called the target, also called the label, and the features used to predict the target are simply called the features. In this way, the target can be thought of as the dependent variable to be predicted based on the independent variable of the features[6].

Training & Gradient Descent

Making a good machine learning model is not as simple as finding the correct set of hyperparameters, in other words the correct structure of the model, however.

Once the hyperparameters of the model are set, the correct set of weights and biases for the model, collectively called the parameters of the model, need to be found as well. This process of finding the correct parameters is known as training the model.

In supervised learning, the objective of training is to find the set of parameters that produce predictions closest to the actual target, given the features. The “closeness,” or more accurately the lack of “closeness,” of the prediction to the target is measured quantitatively with a loss function. The more accurate the model, the smaller the loss function, and so, in other words, training seeks to optimize the model by minimizing the loss function. Typical loss functions include mean squared error for regression problems; in that case, the loss would be calculated by taking the average of the squared errors of each predicted value of the model from the correct target[9].

Once a value for the loss function is calculated for a set of samples, the method of gradient descent is usually used to optimize the model. Although the precise mathematical details of gradient descent are highly involved, the high-level overview of the process is to find the gradient of the loss function of the model with respect to the parameters of the model for every sample in the dataset, and then take the average of all of the computed gradients.

The model then moves in the opposite direction of the average gradient to optimize the function because moving the parameters in the direction of the derivative increases the loss of the function and moving the parameters in the opposite direction of the derivative decreases the loss. The certain amount that the model moves is called the learning rate of the model, and if this process of calculating the gradients and then adjusting the weights accordingly is repeated over and over in iterations called epochs, the process will eventually find the set of weights and biases that produce a local minimum for the loss function[9].

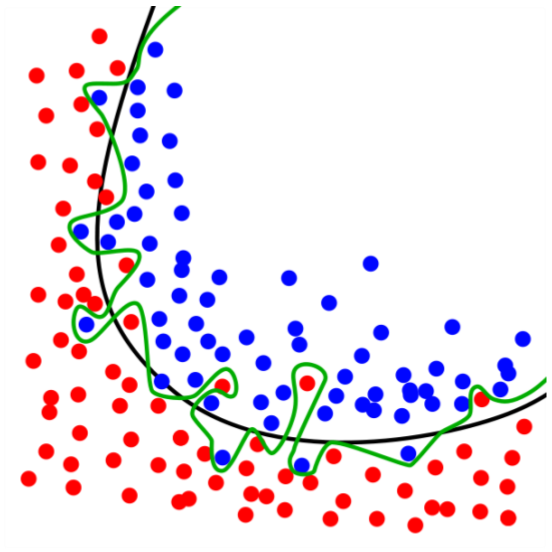
Because calculating the gradient of the loss function for every single sample in the dataset is computationally intensive, especially for large datasets, modern machine learning models take advantage of optimization algorithms that use stochastic gradient descent. As opposed to the method outlined above which computes the gradient of the loss function with respect to the entire dataset, also called batch gradient descent, stochastic gradient descent estimates the true gradient by taking random samples from the dataset and computing the gradients of those. Modern optimization algorithms use stochastic gradient descent in addition to other mathematical functions to improve the accuracy of the estimation[9].

Validating Models and Linearization

Ideally, by minimizing the loss of the model, the model should then be able to be applied to samples of data that it has never seen before and be able to accurately predict their labels. To test this hypothesis, datasets are frequently split into two separate sets: the training set and the testing set.

The training set is quite self-explanatory and is used to train the model. After training the model, the testing dataset is used to test the performance of the final model against data that has never been used to train or improve the model in any way in order to provide an unbiased evaluation of the model.

Occasionally, a third dataset called the validation dataset is used to address the most frequent problem with machine learning models: overfitting. Because machine learning models are trained on a finite dataset, absolutely minimizing the loss of the function is not always beneficial. It could mean that the model, instead of learning the general pattern of the data, learns only how to predict the labels for the specific dataset used to train it very precisely[6].



The green line represents the overfitted model; instead of fitting the overarching pattern of the data, it learns to fit the particular data in this training set.[4]

Testing the model after each epoch of training with a separate validation dataset can help indicate where the model begins to overfit to the training dataset because if the model is learning the general pattern of the data, theoretically both the loss of the training and the validation sets should decrease, but if the loss of the training set is decreasing while the loss of the validation set is increasing, the model is likely overfitting to the training data.

Other strategies of preventing overfitting, also called regularization, include L1 and L2 parameter regularization, dropout, and bootstrap aggregating.

L2 parameter regularization is one of the most common kinds of regularization, both in machine learning and other fields, and other names for it include weight decay, ridge regression, and Tikhonov regularization. L2 regularization in a neural network simply adds a term to the loss function called the regularization term[6, 9].

$$\frac{\lambda}{2n}\sum_{w} w^2$$

The L2 regularization term is the sum of the squares of all of the weights in the network, scaled by a factor of $\frac{\lambda}{2n}$, where $\lambda > 0$ is called the regularization parameter and n is the size of the training set.

Intuitively speaking, L2 regularization makes it so that the network prefers to learn small weights, and only learns large weights if it considerably improves the model's performance, but it's not entirely obvious as to why minimizing the weights can help prevent overfitting. In short, minimizing weights regularizes the model by preventing the model from giving too much weight to specific data points or nodes in the model; in other words, it prevents the model from changing too drastically given small changes in the sample, which is usually a symptom of overfitting to the data. The only challenges with L2 regularization are determining whether it is a good fit for the problem and finding the correct balance between the weights of the model and the performance of the model can be difficult through adjusting the regularization parameter λ.

L1 regularization is very similar to L2 regularization, but instead of adding the squares of the weights to the loss function, L1 regularization simply adds the absolute value of them[6, 9].

$$\frac{\lambda}{2n}\sum_{w} |w|$$

Both L1 and L2 regularization penalize large weights, but the way in which they shrink the weights is different. In L1 regularization, because the penalty is simply the absolute value of the weights, adjusting one weight by a certain amount p is penalized the same as adjusting many weights by an amount that sum up to p. Assuming then that certain weights are more important to the model than others, L1 regularization will tend to more heavily weight the important weights in the model while reducing all others to zero because it has no real preference to reduce weight size or increase weight count, only to reduce the overall weighting of the model. L2 regularization is different because reducing a single weight by p induces a higher penalty than reducing a large number of weights by an amount that sums to p would. Thus, L2 regularization incentivizes the model to remove large outliers and distribute smaller weights over a larger number of weights compared to L1 regularization.

Compared to the traditional statistical methods of L1 and L2 regularization, two modern methods of regularization for machine learning include bootstrap aggregating, also called bagging, and dropout[6].

Unlike L1 and L2, dropout and bagging do not involve modifying the cost function. Dropout modifies a neural network by randomly deactivating certain nodes of the network; bagging involves constructing k datasets by sampling with replacement from the original dataset, and then training a model on each of the new datasets. The k different models, each with the same hyperparameters but each likely with different weights, then work together to produce a single label on testing data by taking the average of the responses from each of the k models.

Both of these methods prevent an overall model from developing a reliance on specific weights. More specifically, dropout forces the model to learn robust features unreliant on any specific piece of data by temporarily disabling some nodes, and the datasets formed by random sampling with replacement for bagging creates models that learn different details to accurately identify samples, which over many models will usually average to produce a single more accurate label.

Dataset

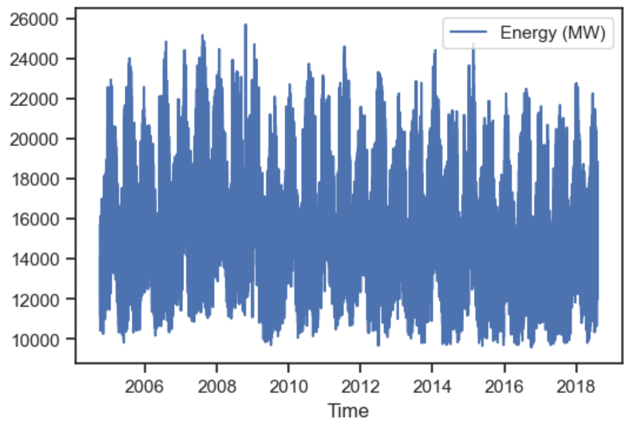
The dataset investigated here is the hourly energy consumption statistics from PJC Interconnection, a regional transmission organization that serves all or parts of Delaware, Illinois, Indiana, Kentucky, Maryland, Michigan, New Jersey, North Carolina, Ohio, Pennsylvania, Tennessee, Virginia, West Virginia and the District of Columbia[10].

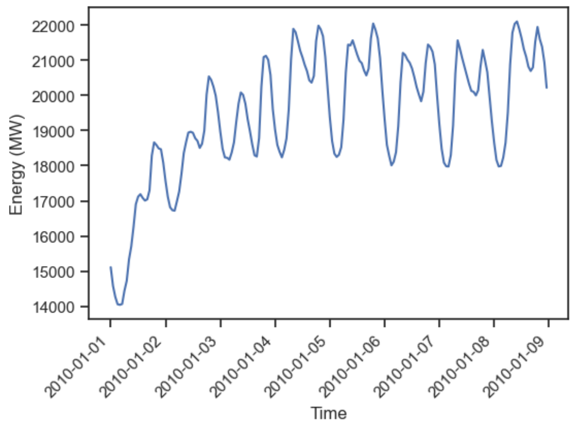
The data is composed of two columns: the date and time, and the power consumption at that date and time in megawatts. Data is collected individually for each of the transmission zones.

We seek to be able to accurately predict the power consumption at a given time, possibly given some information about power consumption for a certain period of time before.

The date and time data of the dataset was split into the year, month, the day, the day of the week (represented from 0 to 6), and the hour of the day.

In all cases, the data is normalized, and for the date and time data, the month, day, day of week, and hour are all in sinusoidal form. For example, instead of the month being a raw integer value m from 1 to 12, it was transformed by $\sin(\frac{\pi}{12}m)$, so that, similar to reality, the dates are cyclical. The year remains in its original normalized form.





AEP transmission zone data

Methods

The methodology of finding the best model was to simply test many different hyperparameters for a model. All models were trained with the Adam optimization algorithm, a learning rate of 10^{-3}, a weight decay of 10^{-1}, tanh activation functions, and an 80%/20% split of the data between the training and testing datasets, respectively. Because the data is collected for individual transmission zones, the models were only trained and tested on data from the AEP transmission zone.

Model Type	Features	Structure	Performance
Linear Model	5 features (current time)	5/1	50 epochs Loss: 1.235871 Test Loss: 1.731337
Multilayer Perceptron	5 features (current time)	5/32/16/8/1	50 epochs Loss: 0.810270 Test Loss: 1.150672
Multilayer Perceptron	29 features (24 hours historical energy usage data + current time)	29/32/8/1	200 epochs Loss: 0.058796 Test Loss: 0.042105
Multilayer Perceptron	29 features (24 hours historical energy usage data + current time)	29/64/32/8/1	200 epochs Loss: 0.075893 Test Loss: 0.043177
Multilayer Perceptron	29 features (24 hours historical energy usage data + current time)	29/16/16/16/1	200 epochs Loss: 0.071260 Test Loss: 0.045340
Multilayer Perceptron	29 features (24 hours historical energy usage data + current time)	29/8/8/8/8/8/1	200 epochs Loss: 0.102726 Test Loss: 0.066970
Multilayer Perceptron	29 features (24 hours historical energy usage data + current time)	29/32/16/8/4/1	200 epochs Loss: 0.104727 Test Loss: 0.057469

Multilayer Perceptron	8 features (3 hours historical energy usage data + current time)	8/8/8/1	200 epochs Loss: 0.083434 Test Loss: 0.081859
Multilayer Perceptron	8 features (3 hours historical energy usage data + current time)	8/16/8/4/1	200 epochs Loss: 0.133429 Test Loss: 0.131387
Multilayer Perceptron	53 features (48 hours historical energy usage data + current time)	53/64/32/16/1	200 epochs Loss: 0.042830 Test Loss: 0.062439
Multilayer Perceptron	53 features (48 hours historical energy usage data + current time)	53/128/64/32/16/1	200 epochs Loss: 0.045835 Test Loss: 0.068054
Multilayer Perceptron	173 features (168 hours historical energy usage data + current time)	173/256/128/64/32/16/1	200 epochs Loss: 0.047896 Test Loss: 0.090699
Multilayer Perceptron	173 features (168 hours historical energy usage data + current time)	173/512/256/64/16/1	200 epochs Loss: 0.046181 Test Loss: 0.090918

Table 1: Machine Learning Model Tests

Results

From the results of these tests, we draw a few important conclusions. First, the data is complex and cannot be expressed simply as a linear model, and a multilayer perceptron or other complex model must be used. Moreover, as we can see from the MLP without recent data as a feature, the date is not simply enough to be able to predict the amount of power used at a given time accurately.

Specifically with regards to the MLP, we can observe that when the historical energy usage beyond 24 hours is included as features, they don't have a statistically significant effect on the performance of the model, but in the model that only had access to data from the past 8 hours, the performance was slightly worse.

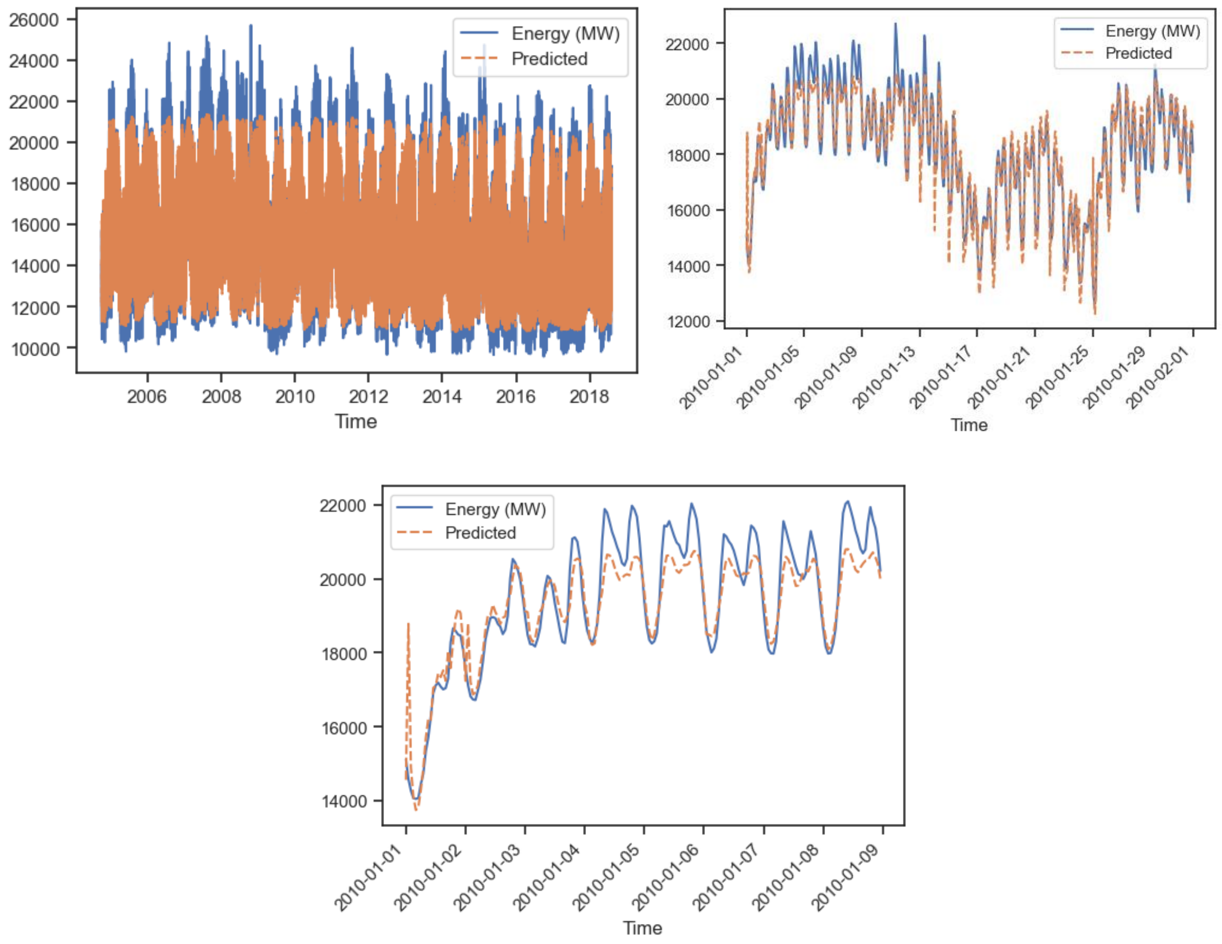
Additionally, less complicated models had performances that were similar to more complicated models with more nodes, suggesting that the depth of the model beyond even fairly simple MLPs was insignificant towards the performance.

Because the models predict the normalized power usage, in order to get a more realistic portrayal of the model, the prediction from the model must be converted back to the unnormalized form by multiplying by the standard deviation and adding the mean.

The loss can also be transformed in such a fashion; because the loss is the mean squared error of the model, the mean error in unnormalized MW can be found by taking the square root and multiplying by the standard deviation.

We can use the example of one of the best performing sets of hyperparameters to estimate the efficacy of this machine learning approach to predicting energy usage as a whole. In the discussion below, we use a model with the structure of a MLP with layers 64/32/8/1, which has 29 features in total with the date and 24 hours of historical hourly energy consumption data.

The test loss of this model was trained to be 0.046874 with a training loss of 0.013337, which translates to an average test error of 561.05 MW.



Comparison of predicted values and actual values on data from the AEP transmission zone in three time frames

From the figures above, it's clear that, as the small test error suggested, the model is at least fairly accurate. However, the model does extremely poorly at responding to extremes in energy usage, both on large and small time frames and in terms of high and low energy usage. The implications of this in terms of its real-world usage is that the machine learning models that were tested in this paper are able to predict to a high degree of accuracy the energy usage during regular circumstances but were incapable of responding to situations where more or less energy than normal needed to be used, for example immediately after a power outage or after a hurricane.

Conclusion

In summary, through the trials of the different hyperparameters for the machine learning models, we found that any multilayer perceptron with significant depth, meaning beyond 30 to 40 nodes and two layers, provided with historical energy data from at least the past day, was able to predict the energy usage during standard conditions to a high degree of accuracy. However, the models failed in being able to predict how extreme energy demand peaks were.

Overall, machine learning models like these would be helpful in predicting overall energy demand, but human supervision and existing methods are still necessary in order to provide reliable electricity during times of peak demand.

We acknowledge that this paper provides an extremely limited view on the different models that could be applied to solving this issue, as well as more definite conclusions on the influence of certain parameters towards the performance of the models. To create a more complete picture of how machine learning could be applied to accurately predicting energy demand, future research could test models trained for more epochs, average model performance over multiple runs, and train a greater diversity of hyperparameters. In addition, the hyperparameters could be studied in a more systematic way by graphing the performance of the models according to ranges of hyperparameters, such as the time frame of historical data able to be accessed. The accuracy of the final models with only access to their own predictions over a period of time could also be studied.

Bibliography

1. *A Plot of the Rectified Linear Activation Function.* 10 Nov. 2015. *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Activation_rectified_linear.svg.
2. *A Plot of the Tanh Activation Function.* 10 Nov. 2015. *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Activation_tanh.svg.
3. Chen, Mark, et al. “Evaluating Large Language Models Trained on Code.” *ArXiv:2107.03374 [Cs]*, July 2021. *arXiv.org*, http://arxiv.org/abs/2107.03374.
4. *Diagram Showing Overfitting of a Classifier*. Feb. 2008. *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Overfitting.svg.
5. Gao, Jim. "Machine learning applications for data center optimization." (2014).
6. Goodfellow, Ian, et al. *Deep Learning*. The MIT Press, 2016.
7. Jumper, John, et al. “Highly Accurate Protein Structure Prediction with AlphaFold.” *Nature*, vol. 596, no. 7873, Aug. 2021, pp. 583–89. *www.nature.com*, https://doi.org/10.1038/s41586-021-03819-2.
8. “Machine learning can boost the value of wind energy.” *Deepmind*, https://deepmind.com/blog/article/machine-learning-can-boost-value-wind-energy. Accessed 30 Oct. 2021.
9. Nielsen, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015.
10. *PJM - Who We Are*. https://www.pjm.com/about-pjm/who-we-are. Accessed 30 Oct. 2021.
11. Schiermeier, Quirin. “Germany’s Renewable Revolution Awaits Energy Forecast.” *Nature*, vol. 535, no. 7611, July 2016, pp. 212–13. *www.nature.com*, https://doi.org/10.1038/535212a.
12. Schrittwieser, Julian, et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.” *Nature*, vol. 588, no. 7839, Dec. 2020, pp. 604–09. *www.nature.com*, https://doi.org/10.1038/s41586-020-03051-4.
13. Sebastian, Raschka. *Introduction to Recurrent Neural Networks*. https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L14-rnns/L14_intro-rnn__slides.pdf.
14. Sebastian, Raschka. *The Perceptron and Introduction to Single-Layer Neural Networks*. https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L03-perceptron/L03_perceptron_slides.pdf.