

Identifying Ising model phase transitions with neural networks: a motivation for convolutions

Daniel Haas*

University of Oslo, Department of Physics

(Dated: December 16, 2022)

We trained a convolutional neural network, combining classification and regression tasks to identify phase transitions in the Ising model. The employed regression method predicts the system's temperature given the input of spin configurations, while the classification model assigns the system as being below or above the temperature of phase transition. We show that a regular feed-forward network is insufficient to identify correlations between neighboring elements in the dataset, motivating the introduction of convolutions. Subsequently, we develop our own convolutional neural network from scratch and, later, a more robust one based on TensorFlow API. The regression task resulted in a general temperature prediction of up to 4% precision in test data. Defining the critical temperature as the one with the highest classification uncertainty yielded a precision of 0.8% from theoretical values.

I. INTRODUCTION

Artificial neural networks have recently been showing promising results in a myriad of diverse scientific fields and applications. As with most machine learning models, neural networks (NNs) can be considered statistical models based on iteratively learning patterns from fed-in data sets. The distinctive character of these networks, however, comes from their inspiration in the biological brain units, the neurons. In this context, the way NNs learn information from data consists of passing the information to smaller units - the artificial neurons, and updating internal parameters based on previous performances, in the case of what is called supervised learning.

This network-like learning process has provided several breakthroughs in modern technologies. A type of NNs with particular success in computer vision and image processing is that of convolutional neural networks (CNNs). The development of CNNs started with the feat of handwritten digit recognition [1], and more recently, similar networks are easily able to recognize sets of over thousands of arbitrary objects [2].

CNNs are a type of feed-forward neural network, but in contrast to more standard ones, they do not exhibit a connection between all neurons in adjacent layers. Instead, CNNs use the convolution operation to reduce the dimensionality of the parameter space by connecting only small patches of neurons while extracting the most important features from data.

In this article, we explore the advantages of CNNs over dense regular FFNNs in the ability to learn correlations between neighbor elements in bi-dimensional data sets. These data set elements can be pixel values of images or, in our case, electron spin values in a 2D

lattice following the Ising model. The Ising model is a statistical mechanics model which describes the behavior of electron spin orientations in a ferromagnetic material given the material temperature. This formalism predicts a critical temperature for which a phase transition occurs in the material and around which there is spontaneous magnetization of clusters of electrons. We developed and trained convolutional models to perform the regression task of predicting a given lattice temperature and the classification task of determining whether a lattice is above or below the critical temperature.

In section II, we present an overview of CNNs and their basic ingredients, such as padding, convolution, and pooling layers. Subsequently, we present the physical model to explain how we generated the data set via simulations. We start section III, by showing that a regular FFNN is unable to learn correlations in the lattice for a satisfactory regression. We then build our own CNN, showing that even superficial tuning yields reasonable results for predicting arbitrary lattice temperatures. To optimize further the regression task, we implement a CNN via TensorFlow API, using later the same architecture to classify a lattice as being above or below the critical temperature. Lastly, we predict the critical temperature of the model as the one for which the classification is most uncertain.

Section IV analyses the obtained results, comparing the implemented methods and their ability to extract data from the model, noting possible future improvements. In Section V, we conclude with a summary of what was learned from the several results and methods.

II. METHODS

For the following material, we will assume familiarity with some concepts of regular neural networks, such as back-propagation and cost functions. Furthermore, as

* <https://github.com/Daniel-Haas-B/FYS-STK4155>

the Ising model is limited to analyzing one grid configuration at a time, the convolution formalism will be for images of only one channel instead of the usual three-color RGB structure. We also note that during this discussion, we will be referring interchangeably to image as the grid of pixel values

A. Motivating convolutional networks

A convolutional neural network is a type of FFNN in the sense that information also only travels forward throughout the network. Nonetheless, when a regular NN deals with intrinsically bi-dimensional data sets, such as images, it unravels the data into one-dimensional vectors. This process then erases the information about neighbor elements correlations, as it does not preserve the data spatial structure.

Furthermore, a regular FFNN connects all neurons in what is called a dense structure. As a consequence, the number of trainable parameters becomes exponentially big when dealing with higher-dimensional information. To counter this, CNNs arise as a way of addressing both setbacks.

1. Convolutional layers

The convolution process is a mathematical way of combining two functions to generate a third one that measures how these functions, or signals, affect each other. For that, convolutions are widely used in signal processing analysis. This operation, mathematically defined between two functions, is given by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau.$$

When translated to discrete signals, however, this procedure becomes more easily explainable and can be seen in Figure 1. The process illustrated in this figure can be expressed in terms of the pseudo-code in 1.

Algorithm 1 Convolution (simplified for 1 filter)

```

for  $i = 1, 2, \dots, (I_h - K_h + 1)$  do
  for  $j = 1, 2, \dots, (I_w - K_w + 1)$  do
     $(I * K)_{ij} = 0$  ▷ Initialize  $(I * K)$  with 0
  for patch, i, j in patches do ▷ patches of I of size K
     $(I * K)_{ij} \leftarrow \text{patch} \odot K$  ▷ elementwise product

```

In algorithm box 1, I denotes the original image, K is the kernel, also referred to as the filter, and $I * K$ is the convolved result. The subscripts h and w denote the height and width, respectively. Note that the algorithm only shows the convolution procedure for 1 filter.

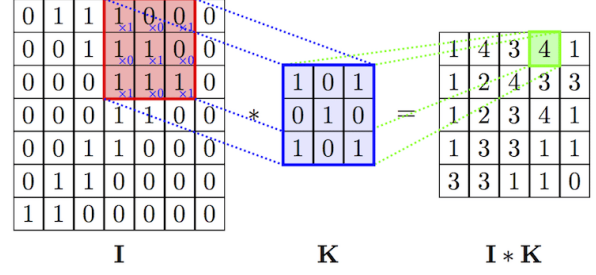


Figure 1: Illustration of a 2D convolution process taken from [3]. Here an image I gets convolved with a kernel or filter K , producing an image $I * K$.

Nonetheless, we could have n filters K_n initialized with different values. In this case, each filter would convolve independently with the image I , and the result of the convolution would be a cube $(I * K)_{ijk}$ with $(I * K)_{**k}$ the result of each filter's convolution.

Notice the convolved lattice is of a smaller dimension than the original one. This downscale is responsible for the mentioned dimensionality reduction in the learnable parameters. This downscaling ratio is defined by the filter sizes used and also the stride (s). The stride defines how much the applied filter moves to the right (or down when it reaches the rightmost edge of the image) after applying the convolution operation in different patches of the image; it is the step the filter takes between patches. For all our implementations, however, we used a stride of 1, meaning we only slide the filter by one element in the convolution.

Starting with an image of initial height h_1 and width w_1 , and similarly, a filter of dimensions K_w and K_h , the output width and heights are given,

$$w_2 = \left\lfloor \frac{w_1 - K_w + 2P}{s} \right\rfloor + 1,$$

$$h_2 = \left\lfloor \frac{h_1 - K_h + 2P}{s} \right\rfloor + 1,$$

where P denotes the amount of padding applied to the image, which will be discussed in the following subsection. Figure 1 shows a filter K with a specific configuration, but, in reality, the values for the filter's elements are randomly initialized and normalized by the filter size as,

$$K_{ij} \sim \mathcal{N}(0, 1)/(K_h * K_w),$$

where $\mathcal{N}(0, 1)$ is the normal distribution centered around zero and with a standard deviation of 1. Subsequently, the values are updated in the back-propagation process, being part of the parameter space to be learned by the network.

In the back-propagation phase, the network learns to optimize parameters which, given the reduction defined by the filter sizes, is able to maintain the most relevant information. Notice that the network learning process is what determines what is defined as "relevant". This will be the information to be retained despite the down-scaling of the image, being determined by how the cost function will change according to the updates in the back-propagation step.

2. Pooling Layers

The pooling layer is another way of significantly reducing the size of the input while maintaining its most significant information. There are several different types of pooling strategies, the most common ones being applying max pooling and average pooling filters to the network. Despite the difference in the mathematical operation they perform, the operational processes of all pooling filters are the same. In the following example, we will use the case of max pooling.

Similarly to the convolution process, we slide a filter, now F , along the input image. Instead of performing the convolution operation, we take the average of the input over the region of $F_w \times F_h$ elements covered by the filter. We then slide s elements to the right (or down if on the edge of the image), which is again characterized by the stride. The resulting reduction of the dimensions of the image is also given by II A 1, albeit with K_w and K_h replaced by F_w and F_h respectively. A pseudo-code for the pooling process is given in 2.

Algorithm 2 Max pooling (simplified for 1 filter)

```

for  $i = 1, 2, \dots, (I_h - F_h + 1)$  do
  for  $j = 1, 2, \dots, (I_w - F_w + 1)$  do
     $pool(I)_{ij} = 0$  ▷ Initialize  $pool(I)_{ij}$  with 0
  for patch,  $i, j$  in patches do ▷ patches of  $I$  of size  $P$ 
     $pool(I)_{ij} \leftarrow avg(patch)$  ▷ average of elements

```

For the case of max pooling, the only difference is that the mathematical operation performed by the filter is that of selecting the largest element in the patch.

3. Padding

Padding an image before convolution simply means that we fill the edges of the lattice grid with a specific value. The most common type of padding is zero padding, in which the edges are filled with the value of zero. When the values of the matrix elements in which we are performing convolution represent the RGB scale, this is equivalent to painting the edge of the image as

black. The thickness of the padding is also a customizable parameter and refers to the number of pixel rows and columns that are being filled with the specific value to the edge.

We will not explore the use of padding throughout the report. Nonetheless, we mention the twofold advantage of applying it to CNNs. Firstly, it guarantees that the central pixels in the original image do not appear with a much larger frequency than the ones in the corner of the image. This unfairness towards corner elements is a consequence of how the central elements appear more in whichever filter operation is slid through the original image. Secondly, by adjusting the number of padding elements, we are able to arbitrarily control the output size from the convolution or pooling process, which can sometimes be useful depending on the network's architecture.

4. Fully Connected Layer

It is common to add fully connected layers to the end of the convolutional networks in order to better extract whichever patterns were selected and preserved along the feature extractions of the convolutions and pooling layers. Indeed, adding fully connected layers is an effective way to learn the non-linearity of the extracted features, although not strictly necessary.

Finally, by connecting the output of the dense layers to one or more than one output neuron, we are able to perform the usual classification or regression tasks according to the output layer activation functions.

5. Our implementation

In this work, we have implemented two CNNs. The first one, with a very simple architecture, was built from scratch, while the second one was developed with TensorFlow API and was inspired by the classic LeNet-5 [1].

The first network's base architecture, without specific information about the layers, can be seen in Figure 2. The CNN was composed of a 2D convolutional layer of original filter size 4x4 followed by a max pool layer of size 2x2. The filter sizes and max pool sizes were customizable, and different values were tested. The output of the max pooling layer is then flattened and fed to two dense layers of 10 neurons each. For the dense layers, we employed ReLU activation functions, while the convolutional and max pooling layers were not followed by activations. A more detailed description of the inputs and outputs of each layer can be verified in Appendix VI, Figure 16.

The LeNet-5 inspired network presents two pairs of intercalated convolution and pooling layers, followed by

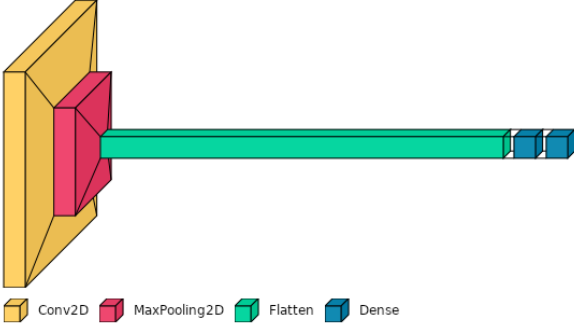


Figure 2: Architecture of the CNN implemented from scratch. Dense layers are followed by ReLU activation functions.

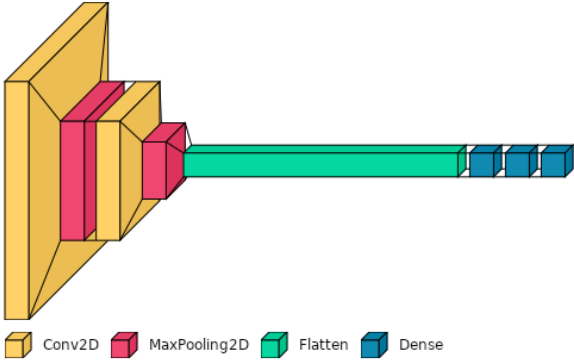


Figure 3: TensorFlow-implemented CNN architecture. Conv2D layers are followed by leaky ReLU activation, while dense layers are followed by ReLU activation.

three final fully connected layers. The general setup architecture can be seen in Figure 3, while a detailed description is in Appendix VI, Figure 17. The convolution layers had a filter size of 5×5 and pooling layers of size 2×2 . After every Conv2D layer, there is leaky ReLU activation, and every dense layer is followed by ReLU activation function. After every layer, there is also 12 regularization of $\lambda = 0.01$. Finally, the dense layers had arbitrary 120, 84, and 1 neurons, respectively.

B. Data Set

1. Brief physics overview: generating the data set

To generate the 2D grid configurations, we implemented a Markov Chain Monte Carlo algorithm known as Metropolis [4]. We initialize the spin configuration of the lattice as being randomly selected values $s \in \{-1, 1\}$. The algorithm then consists of iterating throughout all $L \times L$ lattice electrons, giving the possibility of each to change its current spin value. This

procedure is repeated N times, where N is called the number of Monte Carlo Cycles of the simulation. The acceptance rate of this spin change is modeled by a probability that depends on the system's temperature. More specifically, given the vector representation of all spins in the lattice, \mathbf{s} , the probability of this system's state is given by

$$p(\mathbf{s}; T) \propto \exp\left(-\frac{E(\mathbf{s})}{k_B T}\right),$$

where E is the energy of the system, which depends on the current state, $k_B \approx 1.38 \times 10^{-23} J \cdot K^{-1}$ and T is the system's temperature. We ran the simulations considering unitless spins, which give temperature units of J/k_B , where J is the coupling constant and the energy units. Our data set consists of 1265 grid configurations with temperatures between $T = 1.0 J/k_B$ and $T = 3.52 J/k_B$. To ensure a balanced data set, and given the stochastic character of the model, every temperature value appears 4 times, each time from a probably different lattice configuration. Figure 4 shows 3 of the 1265 configurations and their respective temperatures.

All spin values are initialized randomly between -1 and 1 at the beginning of the simulation, but the grids in the data set consist of the last configuration after 1×10^6 Monte Carlo cycles. This gives the system enough iterations to reach a statistically stable configuration. It is important to notice that no scaling or centering of the data is needed as any point in the 2D lattice can only assume values 1 or -1. All train and test splits were made using an 80 : 20 proportion.

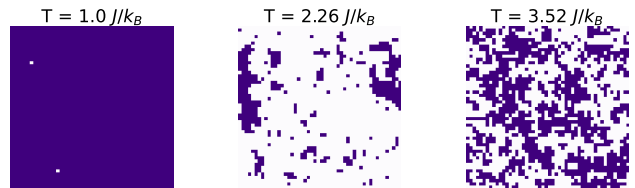


Figure 4: Configurations of spins after 1×10^6 MC cycles for an $L = 50$ lattice at temperatures $T = 1, 2.26$ and $3.52 J/k_B$. Dark purple indicates $s = +1$, light purple $s = -1$.

III. RESULTS

A. Regression

Given a 50 by 50 lattice configuration of spins assuming values of -1 or 1, the regression task employed consisted of predicting the grid's temperature for the

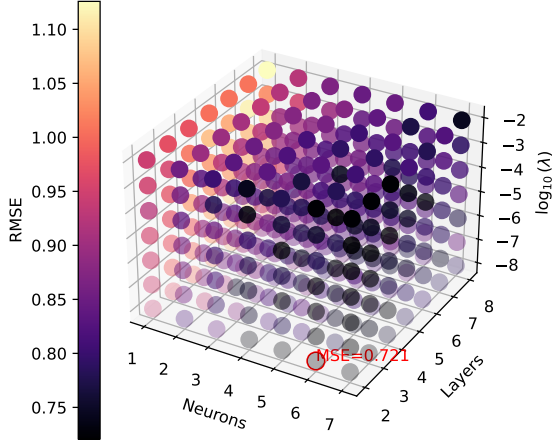


Figure 5: Grid search over FFNN architecture and regularization parameter (λ). The search is done over the number of neurons, the number of layers, and λ yielding a minimum RMSE score of 0.721 over 5-fold cross-validation.

Monte Carlo simulation. This is done first with a regular FFNN and later with two CNNs. While the first CNN was built from zero, the second one was implemented using TensorFlow API.

1. Regular FFNN

We started by training the neural network developed in [5] for the regression task, borrowing also intuition over some of the hyperparameter choices. Figure 5 shows a tri-dimensional grid search in the network’s architecture as well as l_2 regularization parameter λ . This search was done with 5-fold cross-validation over the RMSE score, with a minimum value of 0.721. Table I summarizes the optimal model, together with other hyperparameter information such as batch size, number of epochs for training, activation functions, and momentum coefficients.

After fine-tuning the network, we predicted test-set lattice temperatures in Figure 6, displaying the predicted values against the true values for reference. Here and in all following regression plots, the temperature values were sorted in order to give better visualization.

2. Introducing convolutions: building a CNN from scratch

Subsequently, we implemented our own simple convolutional neural network. The network’s base archi-

Table I: Overview of hyperparameters in the optimal regular FFNN.

Hyperparameter	Value
Learning rate	1×10^{-3}
Regularization (λ)	1×10^{-8}
Epochs	500
Activation	Sigmoid
Momentum (γ)	0.9
Batch Size	20
Neurons	6
Layers	2

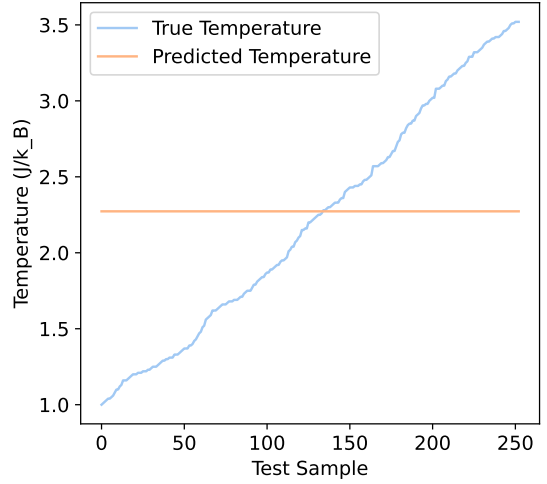


Figure 6: Prediction over the test set of the lattice temperature given lattice configurations. The predictions were made for the fine-tuned FFNN obtained from the grid search of Figure 6

tecture, without specific information about the layers, can be seen in Figure 2. A more detailed description of the inputs and outputs of each layer can be verified in Appendix VI, Figure 16.

To better understand the convolution and pooling process effects in the lattice configurations, we generate Figures 7, 8 and 9. These figures show what independently changing the number of filters, filter size, and pool size does to a randomly selected initial configuration, respectively. Note that since the possible spin values are only -1 or 1, the intermediate values from the operations have no physical interpretation.

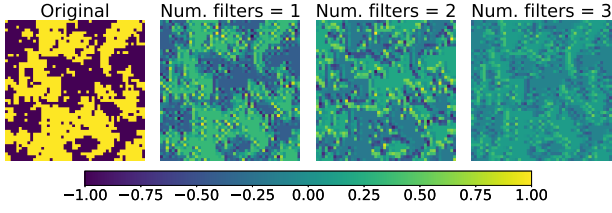


Figure 7: The effects of different convolution filter numbers for the Ising model lattice with a randomly selected temperature.

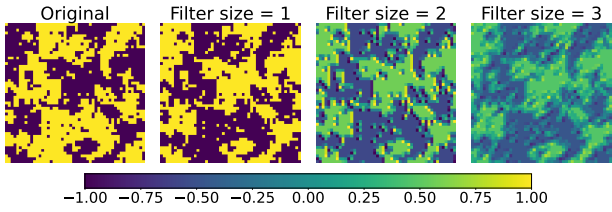


Figure 8: The effects of different convolution filter sizes in the Ising model lattice given a random temperature.

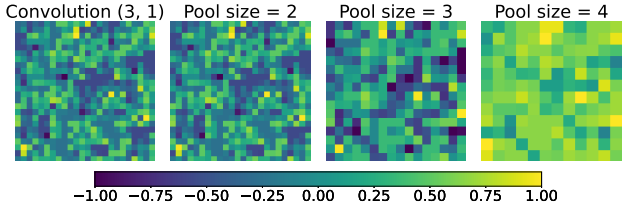


Figure 9: The effects of different max pooling layer sizes in the Ising model lattice for a randomly selected temperature. We start with an already convoluted grid over 1 filter of size 3 by 3.

After having explored the functionalities of different layers of the CNN, we proceeded with a grid search over the CNN architecture, aiming at optimizing the RMSE score over test data. For the grid search, the best RMSE obtained was 0.144 over 3-fold cross-validation. The optimal parameters, together with the activation function, learning rate, and number of epochs for training, are available in table II.

After defining optimal parameters, we proceed to perform the regression task on test data, as usual. The temperature predictions and the actual temperature values for the given test lattices are available in Figure 11.

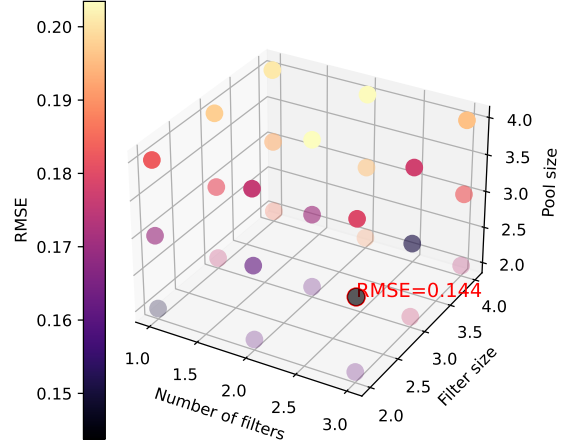


Figure 10: Grid search over the architecture of our CNN built from scratch. We optimize the number of filters, filter size, and pooling layer size over the RMSE score. The minimum RMSE obtained was 0.144 over 3-fold cross-validation.

Table II: Overview of hyperparameters in the optimal built CNN.

Hyperparameter	Value
Learning rate	1×10^{-3}
Epochs	5
Filter size	2
Number of filters	3
MaxPool size	3
Dense layers	2
Dense layers activation	ReLU
Dense layers neurons	10

3. Building a CNN with TensorFlow

With the objective of implementing a more robust and easily modifiable convolutional neural network, we make use of the TensorFlow API. This implementation was arbitrary, with no grid search analysis on the parameters made beforehand. Nonetheless, some parameters can be seen in table III. Given the complexity and variability of the layers, this table brings only general information about the CNN. The general setup architecture can be seen in Figure 3, while a detailed description is in Appendix VI, Figure 17.

Differently from the CNN we implemented in III A 2, we now added an l_2 regularization parameter of $\lambda = 0.01$ to every layer, dense or not. To better understand the effects of the added l_2 term, we show, in Figure

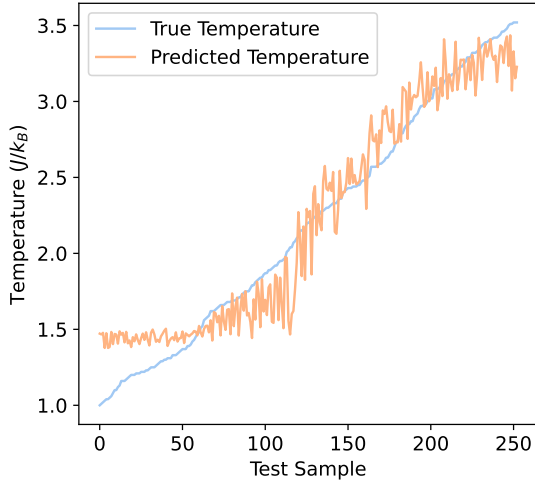


Figure 11: Prediction over the test set of the lattice temperature given lattice configurations. The predictions were made for the fine-tuned CNN we built from scratch. The optimal parameters were obtained via the grid search of Figure 10

Table III: Overview of hyperparameters in the TensorFlow CNN.

Hyperparameter	Value
Learning rate	1×10^{-4}
Epochs	1000
Optimisation	Adam
Optimization ρ_1	0.9
Optimization ρ_2	0.999
Number of filters	3
MaxPool size	3
Dense layers	2
Dense layers activation	ReLU

12, the logarithm over the RMSE score over up to 400 epochs of training. This plot was obtained with and without regularization and over the training and testing samples.

We thereafter trained the network over 1000 epochs, with the regularization term, and performed the regression predictions that can be checked in Figure 13, similarly to what was done for the previous 2 networks.

4. Comparing the network's RMSE

We finally compare the three implemented neural networks in the regression task by summarizing their best RMSE score over testing data in table IV. Those RMSE scores are the ones from the predictions of figures 6, 11,

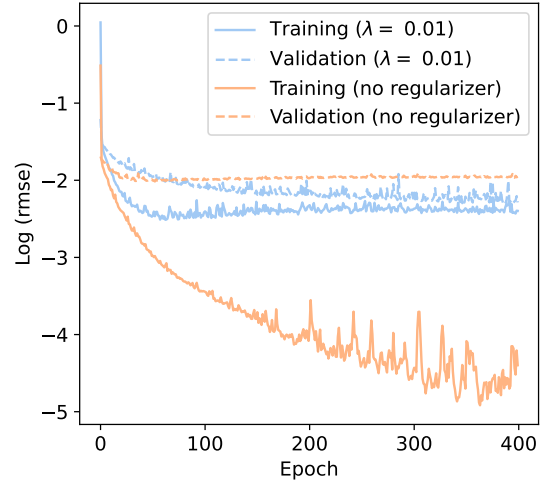


Figure 12: Logarithm of the RMSE score for training and testing sets over the CNN implemented via Tensorflow over multiple epochs. We show the effect of adding an l_2 regularization parameter of $\lambda = 0.01$ over training and testing scores.

Table IV: RMSE obtained on test data. Optimal hyperparameters for the FFNN and our CNN were applied.

Model	Test RMSE	Test error %
Regular FFNN	0.981	44.5
Our CNN	0.197	8.93
TF CNN	0.091	4.12

and 13. We also display the percentage the RMSE represents over the average test data.

B. Classification

By changing the last layer's activation function of the TF CNN to the sigmoid function and setting the loss function as cross-entropy, we then move to the classification task. All other parameters for the TensorFlow-implemented network are maintained the same.

The classification task is that of assigning a given lattice configuration to be above or below the critical temperature. We start by analyzing the confusion matrix of Figure 14, for which the accuracy score was 0.992 over test data.

By analyzing which probabilities are given for each lattice of the test set, we are able to plot the probability for a given temperature of being above or below critical temperature in Figure 15. With that, we infer T_c as

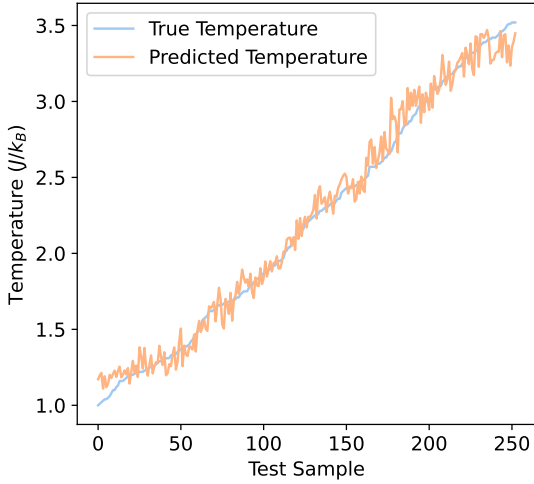


Figure 13: Prediction over the test set of the lattice temperature given lattice configurations. The predictions were made with the CNN implemented via TensorFlow API, trained over 1000 epochs, and l_2 regularization parameter of $\lambda = 0.01$.

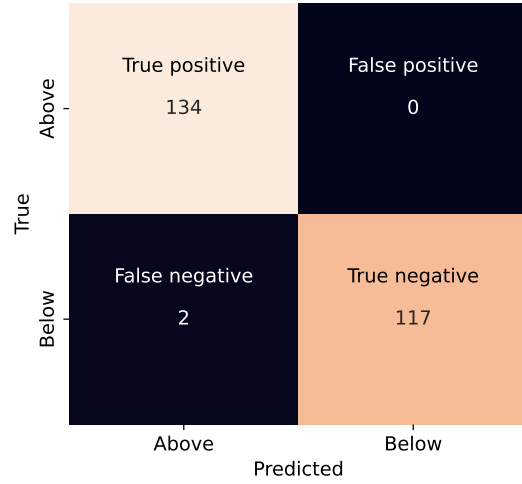


Figure 14: Confusion matrix over the classification of test lattices being above or below the critical temperatures. The obtained accuracy score over the test data was 0.992. The CNN used for the classification had the same architecture as the one in section III A 3, now with a sigmoid output layer.

being the one that shows the largest uncertainty in the classification task. In this case, we predicted $\bar{T}_c = 2.25$

IV. DISCUSSION

A. Regression

1. Regular FFNN

We start by noticing from Figure 6 that even after considerable fine-tuning of the standard FFNN, the model is unable to perform better than a simple average prediction of the temperatures. Indeed, since our data set was generated as extremely balanced and with a good number of samples, the average over training data should be approximately the average over test data. This average is then always predicted independently of the model input.

The behavior of Figure 6 is usually a sign of underfitting, despite the grid search for optimal parameters being done over 5-fold cross-validation and the training being performed with 500 epochs, which is usually sufficient. This illustrates that the FFNN is unable to identify correlations in the neighbor elements in the bi-dimensional grid, as would be expected from the non-convolutional character of the model. This insufficient presentation is again confirmed by the test error of around 44% in test samples.

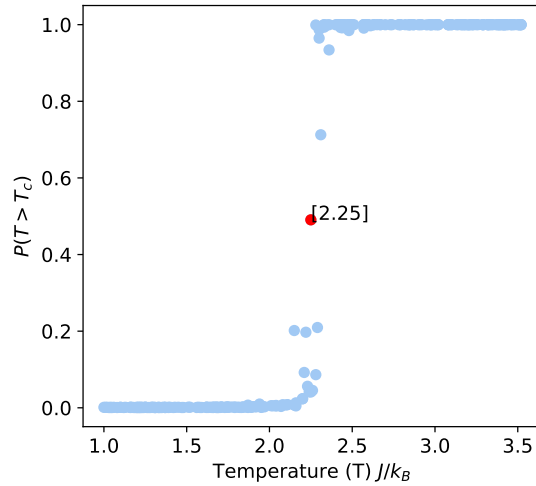


Figure 15: Classification probability of a given lattice with temperature T of being above the critical temperature. Marked in red is the inferred critical temperature value. The CNN used had the same architecture as the one in section III A 3, now with a sigmoid output layer.

2. Our own CNN

It is convenient to notice that we can obtain intuition over the optimal selected parameters of Figure 10 grid search from Figures 7, 8 and 9, where we analyze the effects of the filter number, filter size, and max pooling layer size:

While a filter size of 1 would not yield effective dimension reduction, a filter size of 3 seems to make the boundaries of the clusters of spin values too unclear. This loosely elucidates the optimal selection of filter size 2 in the grid search. Similarly, the max pool layer of size 3 is able to reduce the number of pixels in the grid while also reinforcing the boundaries. In contrast, the optimal grid search value of 3 for the number of filters could not be visually explained.

The grid search of our CNN architecture in Figure 10, despite not being so extensive, provides an RMSE score that largely outperforms the one from the regular FFNN. The obtained error over test data was, in this case, nearly 5 times smaller. This improvement demonstrates that adding convolutions to the network translates to the ability to learn patterns amongst the data set, which were uncorrelatable for regular dense architectures.

3. Tensorflow CNN

It must be mentioned that the effortlessness to implement the CNN via the TensorFlow API made the iterative process of parameter tuning much more efficient, yielding a reasonable RMSE score even without a proper grid search. It also allowed for the easy addition of l_2 regularization, the effects of which are visible in Figure 13.

Adding a regularization parameter to the CNN resulted in the same expected result as adding it to other machine-learning models. For a larger number of epochs, it reduced over-fitting significantly, allowing for test set RMSE scores lower than those obtained for the training set without regularization. Indeed, this figure shows that while training performance got arbitrarily better without regularization, this is prevented with the added parameter.

We can compare the regression plots over the test set for the TensorFlow network and the one implemented from scratch, namely Figure 11 and 13. This comparison contrasts the superior results of the former. Both models present a similar behavior around the critical temperature in the interval between $T = 2J/k_B$ and $T = 3J/k_B$, but the TF one is more robust around the extremes. This pattern is expected because the changes in grid configurations for temperatures far from T_c are more subtle. For supercritical temperatures, the spin

configurations are extremely random, and it is unreasonable to expect learnable patterns in this stochastic behavior. Similarly, sub-critical temperatures far from T_c differ very slightly, as they rapidly all align to stable spin configurations.

Consequently, The TF CNN is better able to learn more nuanced changes in the extreme intervals of the data set, which accounts for a big difference in the overall RMSE. Table IV shows that it is able to achieve a test error that is less than half the one from the previously implemented network.

B. Classification

By the confusion matrix of Figure 14, it is clear that the model is capable of classifying whether a lattice configuration had a temperature above or below that of phase transition. With only 2 out of 253 test samples wrongly predicted (99.2% accuracy), the same CNN architecture applied for the regression task was suitable for the classification of temperatures. That is expected, as being able to accurately predict arbitrary temperatures is a more complex feat, which would naturally extend to being able to separate configurations into two classes according to their temperatures.

Using the classification task to infer the critical temperature as the one which displayed the highest classification uncertainty yielded excellent results, especially considering the size of the grid. Note that despite the training labels describing the grid as above or below T_c , no information about the critical temperature per se is given.

Jakobsen et al. [6] show that a bigger bi-dimensional grid tends to better approximate the theoretical critical temperature of phase transition, which is $T_c \approx 2.27J/k_B$ [7]. The grids used for our model's training can be considered small ($L = 50$) and, despite that, provided a predicted T_c with an error of around 0.8% of the correct value.

C. Future work

The grid searches from Figures 5 and 10 differ not only in the number of points searched but in the number of cross-validation folds and epochs. The reason for that discrepancy was the sub-optimal time performance of our CNN. Since the implemented convolutional network was considerably slow, performing the same extensive parameter search and training was infeasible. It would therefore be desirable to optimize the model in future works in terms of computation time. This would allow us to train the network in a larger number of epochs with more parameter experimentation.

Furthermore, as mentioned in III A 3, no proper parameter search was done for the TensorFlow-implemented CNN, as the obtained performance was already satisfactory. It would be useful to first make a statistical study of the variance of the data set to understand how much better our models could perform. Secondly, depending on the results of this statistical analysis, a rigorous grid search over the network’s parameters could be justifiable.

V. CONCLUSION

We successfully built and implemented convolutional neural networks both for the task of regression and classification for a bi-dimensional Ising model dataset of lattice configurations and temperatures. We motivated the use of convolutions as a dimensionality reduction feature but, most importantly, as a distinct way to learn correlations in elements of the dataset.

We showed that a regular feed-forward network was unable to yield satisfactory results in the regression task

due to the nature of the data. We subsequently implemented a convolutional network and performed grid searches in the network’s architecture to get a better understanding of the different model possibilities that come from convolutional layers. We demonstrated how the l_2 regularization method naturally extends to convolutional networks, enabling over-fitting reduction. The implemented CNNs vastly outperformed the regular FFNN in the regression task, obtaining RMSE scores with errors of 4% over test data.

Lastly, we classified Ising model lattices as above or below temperatures of phase transition with an accuracy of 99% without prior knowledge of the temperature value. With that, we were able to infer the critical temperature with an error of 0.8% of what is predicted by theory.

Altogether the results show that the type of neural network to apply when solving regression and classification tasks depends greatly on the format of data one deals with. Furthermore, we were able to apply convolutional methods to a very specific scientific topic, displaying the versatility of CNNs beyond the usual fields of computer vision and image recognition.

-
- [1] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, *Advances in neural information processing systems* **2** (1989).
 - [2] K. Simonyan and A. Zisserman, *arXiv preprint arXiv:1409.1556* (2014).
 - [3] I. S. Mohamed, *Detection and tracking of pallets using a laser rangefinder and machine learning techniques*, Ph.D. thesis, European Master on Advanced Robotics+(EMARO+), University of Genova, Italy (2017).
 - [4] M. Hjorth-Jensen, *Computational Physics* (University of Oslo, 2015).
 - [5] N. T. A. Jakobsen, D. Haas and V. Ung, “Exploring the transition from regression models to neural networks,” (2022).
 - [6] L. H. A. Jakobsen, D. Haas and N. Abbasova, “A numerical study of phase transitions in ferromagnetic systems using a 2d ising model,” (2022).
 - [7] L. Onsager, *Physical Review* **65**, 117 (1944).

VI. APPENDIX

A. Full description of CNN architectures

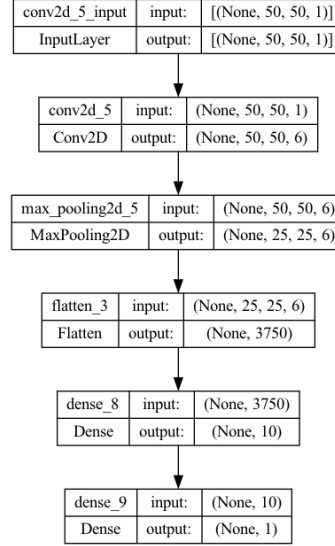


Figure 16: Detailed information about inputs and outputs of each layer of our own implemented CNN.

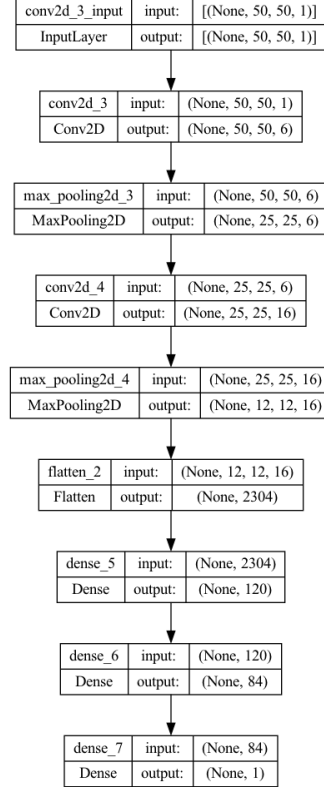


Figure 17: Detailed information about inputs and outputs of each layer of the CNN implemented via TensorFlow.