Introduction to Python, machine learning and data handling for the physical sciences

Mattias Ermakov Thing

November 30, 2023

1 Week 50

1.1 Intro

This week the topic is neural network and how to get started using the Python package `keras` [3], which is a simply interface extending the underlying package `tensorflow` [1].to get started you can always look at the official documentation:

- `https://keras.io/getting_started/`
- `https://www.tensorflow.org/learn`

Neural networks belongs to a separate tool of machine learning and there are many ways to use neural networks and how to train them. The most classical approach to using and training neural networks is via supervised learning. In this case you have some training dataset $\{X,Y\}$ of inputs and outputs. Since an arbitrarily large deep neural network can learn any function f [4], we train a neural network to imitate any arbitrary function:

$$X \to f(X) = Y. \tag{1.1}$$

This method is straight forward if you have a dataset. The problem is just that you might not have enough data, or no data at all. In this case things get more complicated and there are other methods to try such as genetic learning methods. In some cases we can also do the training unsupervised, but these advanced training methods are beyond this course.

1.2 Neural network

As the name suggest neural networks are composed of neurons similar to our brain. Any neuron network uses the same basic building block, the single neuron. In some applications such as in robots less than 100 may be sufficient [5], whereas image classification networks may contain thousands of neurons [7]. Thus, to understand neural networks we should take our time to understand how they work. Neurons are fundamentally doing linear regression, and later we discuss how nonlinear behavior is obtained as this is key to the power of neural networks. As a simple example, one can consider

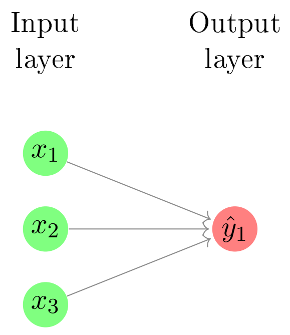


Figure 1: *A simple neural network with an input layer, and a single neuron in the output layer.*

a single neuron "network" shown in Figure 1. This "network" takes in three inputs, x_i, where i = 1,2,4, and computes a predicted value $\hat{y}$. The word network is in quotation as one would need multiple interconnected neurons to properly talk of a network.

In the case of this single neuron with three inputs shown in Figure 1 the equation becomes,

$$z(x_i) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b, \tag{1.2}$$

where each input is associated with the weight w_i which modulates the influence of the input, and then there is a bias term, b. This can be written in a compact matrix form as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b \end{bmatrix} = w_{ij} x_i + b_j \cdot \vec{1}, \tag{1.3}$$

where an additional j index is added to indicate the number of neurons, but since there is only one neuron it is redundant. The use of this index becomes apparent when considering a network like in Figure 2, but with two fully connected output nodes, $j = 2$. In this case, the equation would be,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = w_{ij} x_i + b_j \cdot \vec{1}. \tag{1.4}$$

This highlights the benefit of the notation with the weight matrix as w_{ij}, the input vector as x_i, and the bias vector as b_j. A technical note would be that, in practice, it is more efficient to skip the process of adding the bias terms explicitly. One can exploit the definition of matrix multiplication and write it as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} & b_1 \\ w_{12} & w_{22} & w_{32} & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} = w_{ij} x_i + b_j \cdot \vec{1}. \tag{1.5}$$

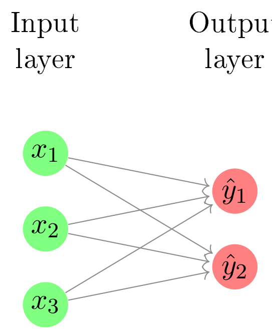


Figure 2: *A simple neural network with an input layer, and two neurons in the output layer.*

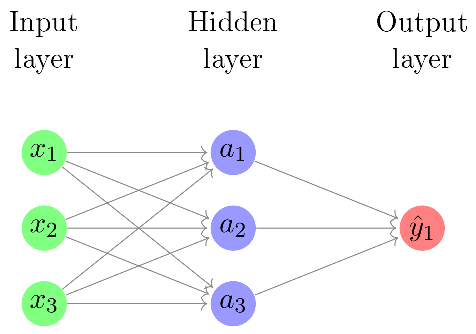


Figure 3: *A simple neural network with an input layer, a hidden layer, and a single neuron in the output layer.*

From this we can then also consider more complex networks with hidden layers as in Figure 3. In this case we would first compute the first the values at the hidden layers $z_1(x_i) = a_i$ and then we can use that result to compute that result $z_2(a_i) = \hat{y}$.

A linear equation is not sufficient to produce complex nonlinear decision boundaries, therefore it is key to modify the result $z_j(x)$ such nonlinearity is achieved. This is very simply to prove, consider our simple network with one hidden layers. The math becomes,

$$\hat{y} = z_2(z_1(x_i)) = w_{i2}\left(w_{i1}x_i + b_1 \cdot \vec{1}\right) + b_2 \cdot \vec{1}. \tag{1.6}$$

The problem here is that while we have different weights and biases, due to the lack of nonlinearity, this hidden actually achieve nothing because we can redefine this equation to be just like a network without a hidden layer,

$$\hat{y} = z_2(z_1(x_i)) = \underbrace{w_{i2}w_{i1}x_i}_{w_{i1}x_i} + \underbrace{w_{i2}b_1 \cdot \vec{1} + b_2 \cdot \vec{1}b_1 \cdot \vec{1}}_{b_1 \cdot \vec{1}}. \tag{1.7}$$

This is perfectly valid because the latter terms is nothing but a sum of trainable variable, which is nothing but a number, and then first part contains the input times two different weight, but again we can just redefine that as a single weight. Therefore, without any nonlinearity we can add an arbitrary number of hidden layers and achieve the equivalent of having no hidden layers.

This is fixed using nonlinear activation functions, $a(z)$, such as the sigmoid function, $\sigma(x)$, and the rectifier linear (ReLu) function, ReLu(x). There are many more options, and it is up to the designer of the network to determine the best activation function. The activation function, f, can be applied on a per-neuron basis, but it is commonly applied across the layers,

$$a(z_j(x_i)) = f(w_{ij}x_i + b_j \cdot \vec{1}). \tag{1.8}$$

Each layer of the neural network is generally computed in this way. One can then stack the layers, $a(z_j)^n$, where n is the number of layers. Deep neural networks then have three layers or more, $n \geq 3$. The output of the neural network is then computed by iteratively computing the layers from input to output, with the previous layer being the input to the next layer.

1.3 Supervised learning

The first step in training a network is to determine the error of the network. The goal is generally to minimize the error that the network is making. For this, one considers the loss function, $\mathcal{L}_{loss}$, which is a function of the network prediction, $\hat{Y}$, which is the result of the last layer of the neural network. In supervised learning, you have the input X and output Y, which you have labeled to be something specific. Since the goal is to mimic this result that you have determined to be correct, we are thus interested in comparing the two. There are many options comparing which can be chosen as the loss function. Popular choices include the root mean squared error (RMSE), mean squared error (MSE), and mean absolute error (MAE). As an example, one can write the loss function of the MSE as,

$$\mathcal{L}_{loss}\left(\hat{Y}, Y\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}_i - Y_i\right)^2, \tag{1.9}$$

where n is the number of entries in $\hat{Y}$ and Y. On top of the loss function, one can consider additional regularization or other custom terms, which can be included in the cost function that is used for training. An example of a regularizer is the L2 regularizer which is the sum of squares of the weight, which naturally penalize large weight and this can help against over-fitting. In this case, the total cost function with the MSE as the loss function is,

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{loss}\left(\hat{Y}_i, Y_i\right) + \frac{\lambda}{2k}\left\|\tilde{W}\right\|_2^2, \tag{1.10}$$

where θ represents the trainable parameters, m is the batch number, k is the number of regularized weights, and λ is the strength of the regularization term. Note that here $\tilde{W}$ is the vector of regularized weights.

Since the cost function is a function of the result of the network, and by extension its trainable parameters, it is possible to do gradient descent to train the network. In literature, the cost function is often called a loss function, and this term is often used interchangeably. Sometimes the loss function is considered as a single term that measures a loss, thus, one can have multiple loss functions and regularizer terms in the cost function, where the total function which the network uses for training is the cost function. For a simple training rule, where one only uses a single loss function, the loss and cost function will be the same.

While the general concept of training involves some form of gradient descent, there are many optimizers available such as RMSProp, Adam, AdaGrad etc. which are modified gradient descent algorithms that might include effects such as momentum which is where the effect of the previous step is also part of the update. The network is then trained across a certain number of epochs, which is the number of times the network is training on the full data set. One may use batch sizes to evaluate a subset of input and output values before making the next gradient and decent update. This can improve training performance.

Another key training hyperparameter is the learning rate, α. Note that hyperparameters are model parameters that are not trained, but it is up to the designer to choose them such as the learning rate, batch size, epochs, number of layers, etc. The learning rate decides how large steps the learning should take. A too-large learning rate might prevent the learning algorithm to find the desired minimum, and a too-low learning rate may cause the network to train for an unnecessarily long amount of time, but could also cause the network to get stuck in small local minima.

1.4 Iris classification

To show a simple use-case of a simple neural network with just one hidden layers, we can try to classify iris flowers as we have a nice and simple dataset for that. In this dataset there are four input features and three different categories. There are many things we need to consider:

- Loading the data
- Covert the category to a number
- Consider normalization of the input
- Splitting the data into training, validation and test datasets
- Constructing the neural network
- Setting hyperparameters
- Training the network
- Evaluation of network performance

For this purpose you can find the example code for the iris network at: `https://github.com/CP3-Origins/FY555/blob/main/python/main.py`

When loading the data you should first understand the format of your data. Consider the file extension and find out how you load it in python. Then you can inspect it using commands like `print()`, `type()`, and `.shape` to learn about the structure of the data. You want to end up with a `numpy` array or a `tensorflow` tensor as your input and output.

Consider if any of you input and outputs are not numbers. Since neural networks are mathematical equations we need everything to be numbers. If you have categories that are not number, you need to encode them to a number. In the case of the iris dataset, we have three different types described by a name. So we need to encode these three categories to numbers.

Normalization is another thing to consider. Generally, this is not necessary as networks are sort of scale invariant. But since weights are generally initialized around one, we would also like to have our input to be around one, so it is often a good idea to normalize your input unless especially if your input values are far from $\mathcal{O}(1)$. Now there might be reasons not to normalize, but in many cases it might help the network perform better.

When training a network in a supervised manner we need to split of out dataset. We need data for the network to train on. This should be the bulk of the dataset. Now, we need some data that the network isn't trained on, because we want the network to learn an arbitrary underlying function and not just to fit to our training data. That's why we take away some of the data as validation data. This data is **not** used for training, but to check how our model is performing on unseen data. For good measure we can have another dataset called the test data. This is another small dataset which is unseen and can be used in the end to evaluate the trained model.

You might ask why do we need both a validation and test dataset? The validation data is used to fine tune hyperparameters and thus our model is biased based on both the training and validation data. The latter is key for us to ensure the selection of hyperparameters such that we are not over-fitting, but getting reliable result of unseen data. Therefore, in a true experiment you would want to save a test dataset for the end when you might have several different trained models, and then based on this final evaluation using the test data, we can choose the best model. How you split your dataset is up to you and it depends on how much data you have. In this example of the iris model start by taking away 20% for testing. Then we have 80% of which another 20% (16% of the total) are set aside for validation, with the remaining (64% of the total) being used for training.

Now you have your data and a strategy of how to use the dataset, you can construct your network. Here you need to consider the shape of the input and output. If your network doesn't match, then you will get an error. If you have four inputs, then you need your first layer to take four inputs, and likewise if your output has three categories, you need 3 nodes in your final layer. Now what you decide is how many nodes there are in the hidden layers, the number of hidden layers, the activation functions and other parameters beyond this course. These things are called hyperparameters

Hyperparameters is where practice, knowledge, and experience comes into play because you have to figure out these parameters. There is no go-to method of finding the right parameters. One strategy is to change one and see how that affects the result. Another strategy is to set hyperparamters at random, within some limits, and then simply log what random parameters worked best and use them in the end. Or one could use a combination of the two, or just try some other strategy. An experienced person might have an idea what could be a reasonable starting point, but there is no guarantee.

There are also other hyperparameters to consider such as learning rate, batch size, epoch, loss function, metrics etc.

Having build your network and set some initial values for the hyperparameters, it is time to train the network using training and validation data. This is how you test network performance, and you would want to train it every time you changed any hyperparamter to check how well this affects the performance. For classification we can often use accuracy as a metric to measure how often we classified correctly. This is the benchmark

One thing to consider is the issue of over-fitting, which is when your network begins learning your training data and not the underlying trend. You can detect such behavior if the accuracy or loss using the training dataset improves while it gets worse for the validation dataset. To help this you could train less or make the network small. There are other ways to deal with this like regularization which is beyond this course. In the end you can evaluate on your test data to determine how good your network performs.

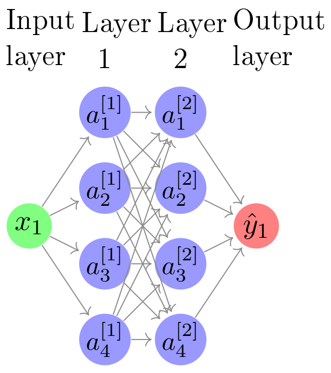
1.5 Beyond the basics

Neural networks can be immensely powerful, and fully connected layers, also called dense layers, as described in the previous section are the backbone of that. There are however downsides of dense layers, these standard neurons. One issue is that the number of parameters can simply be too large. Another problem is that these neurons don't store any memory, in the sense they are oblivious to any previous information. There are generally many downsides of the classical feed-forward artificial neural network. In this section details regarding the architecture of different networks is described.

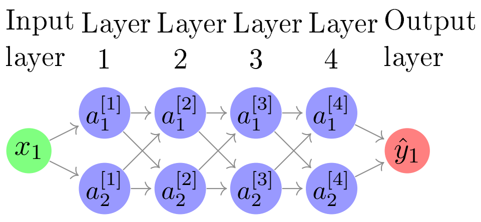
1.6 Deep neural networks

A subcategory of ML methods is that of deep learning (DL) methods. As the naming suggests, it has to do with learning more deep details about some system or problem. This requires more than just simple single or two-layer networks. Deep learning involves networks with thousands of parameters if not millions, like that of ChatGPT [2]. In the context of DL methods one considers the use of deep networks or the combination of different types of networks and ML methods.

When a classical artificial neural network (ANN) has three or more layers one can consider it as a deep neural network (DNN). This, arguably, is not enough to qualify the network as a DL method, but it is a step in that direction. DNNs can contain tens of layers if not even more in order to have enough parameters to model complex behavior. Having a very wide network with many neurons in each layer is one way to add degrees of freedom, but one may reduce the number of neurons per layer and have a longer network with many smaller layers. Since the layer parameters scale as the product of the number of nodes in the previous layer times the number in the current layer, one can quickly end up with a lot of parameters to train for wide networks. This fact is illustrated in Figure 4, where the wide network in Figure 4a as 33 parameters compared to the narrow network in Figure



(a) *A simple neural network with wide layers. This network 33 trainable parameters.*



(b) *A simple neural network with many narrow layers. This network 25 trainable parameters.*

Figure 4: *Two architectures with the same number of nodes, but with a different number of parameters.*

4b with 25 parameters. This includes the weight parameters, each weight parameter corresponds to a connection, plus one bias parameter from each node. Both networks have the same number of neurons, but not the same number of parameters. This is something the architect of the network has to consider in terms of what effects the width has on the output and the trainability of the network.

This example demonstrates the concept that wide layers cost more to train, as you have more parameters to train. For many applications, a DNN with a sufficient number of layers can prove very successful, but the hidden layers are nothing but a black box, a common problem with neural networks. The larger the network, the harder it becomes to interpret the logic of the network, and in practice, it is generally not possible to understand the logic the network has learned.

Regarding the issue of training parameters, one can consider a problem in which the input is very large. In a DNN each input is densely connected to the first layer, thus for each neuron in the first layer, one can multiply the number of weights required to process the input. Furthermore, large inputs usually require a rather large first layer to retain the information from the input layer, thus the larger the input the larger the first layer should be leading to the problem of too many parameters. This is one of the limiting factors of DNNs, together with the fact that the neurons have no memory to store information about the previous value it processed.

1.7 Convolutional Neural Networks

A solution to the large input problem is the convolutional neural network (CNN). The problem is that dense connections can result in an exploding number of parameters, which become computationally unfeasible to train. Luckily, it isn't necessary to always use dense layers and a way to reduce the number of parameters is using CNN which uses convolution and pooling to reduce the problem, while retaining key information, before handing it over

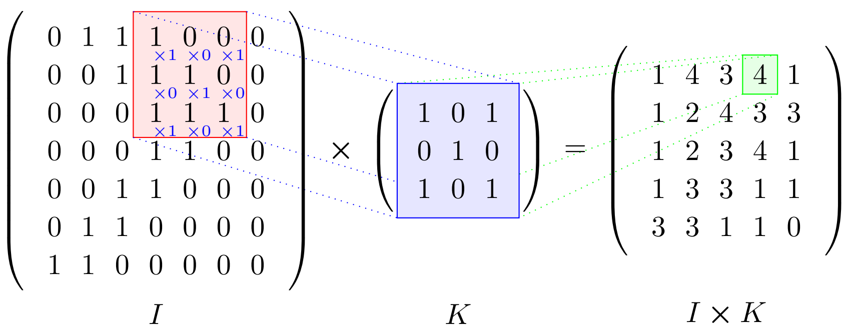


Figure 5: *Example of convolution of input, I, with the kernel, K. The kernel is the trainable weights.*

to a DNN network to produce the desired output. In the case of networks such as autoencoders, one might upscale the dimension of the output to match that of the input by doing upsampling instead of pooling.

In this section, the focus is the convolution and pooling used in CNNs. These networks are often used in image processing, thus, it makes sense to consider the case of a 2D input. Instead of connecting every input to every single neuron in the first layer, one can use a kernel. This means each neuron in the layer is only connected to a subspace of the full input. The size of the kernel is thus the size of the weights of each neuron, and one can still have a bias term. By having for example a three by three kernel, as shown in Figure 5, one would end up with only 9 weights per node plus the bias instead of e.g. 49 weights if the full input is a 7x7 two dimensional input. The reduction here is limited because the input as shown in Figure 5 is not very large due to practical reasons, but consider a modern 43.2 MP picture with dimensions of 6000x7200. In this case, the convolution neuron would still have 9 weights and not 43.2 million weights as the case would be for a fully connected neuron.

The values of the kernel acting on the input are then summed and processed with some activation function yielding the final value of the convolution neuron. The procedure is done across the outer input, but one may note that the kernel requires input from surrounding values of the input, one can not simply apply the kernel to edge values. Depending on the size of the kernel one would thus naturally get a size reduction in the convolution layer. It is however possible to apply some padding like zero-padding to pad around the edge such that the kernel can be applied around the edge such that the convolution layer retains the same size as the input.

Another option in case one seeks to reduce the size of the convolution layer is to change the stride. The striding regulates the stepping which the kernel should move across the input. One can thus apply a stride of two to only compute a value for every second step. This will then quadratically reduce the size of the convolution layer.

The final important thing to consider with convolution layers is the number of filters. Since you can generally reduce the number of parameters by orders of magnitudes with convolution layers, one might face an issue with

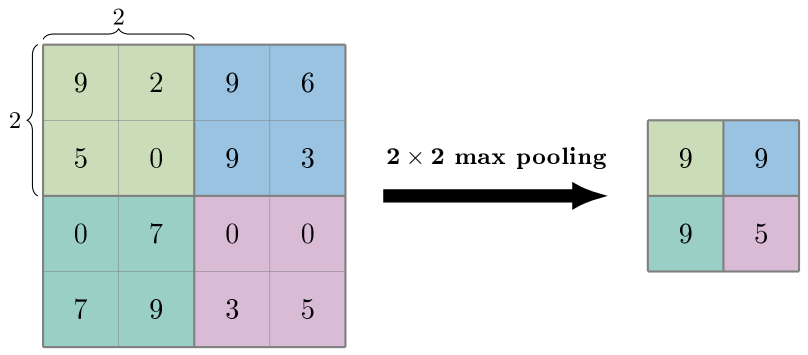


Figure 6: *Example of using max pooling with a pool size of two and a stride value of two.*

information being lost as the convolution might condense the problem with information loss. To compensate one can have multiple filters in convolution layers, in this case, one essentially repeat the convolution process as described in Figure 5. In this way, each point of the input is compiled multiple times with different weights that can be trained to capture different information about each point. For, example the filters might each capture some color value.

Overall, the convolution layer helps to massively reduce the number of parameters, but one should consider additional hyperparameters such as filter number, kernel size, padding and stride.

Another operation relevant to CNNs is that of pooling. In practice, this is a layer, but it doesn't add any new trainable parameters. It provides a systematical method to merge output values of the previous layer to decrease the number of parameters passed forward in the network as shown in Figure 6 with max pooling.

There are a few different ways to do the pooling such as min, max and average pooling. In the case of max pooling one reduces the pool size to a single value by selecting the maximum value in the given pool as shown in Figure 6. In this example the pool size is two, thus a 2×2 pool is considered and the highest value is selected. Since the stride is of value two, then one skips one ahead before creating another pool. Had the stride been of value one, then there would be overlap between the pools, but it might be desirable to choose such a value. As with the convolution layer one can also add padding to a pooling layer. The pooling layers generally have pool size, stride and padding as hyperparameters.

1.8 Recurrent neural networks

In certain applications such as speech recognition, and other cases with time dependence, the previous input value may be correlated with the subsequent value. This is something a usual DNN cannot take into account at the neuron level. This requires the neuron to have some memory. To achieve this one can make use of recurrent neurons to make recurrent neural networks (RNN) [6].

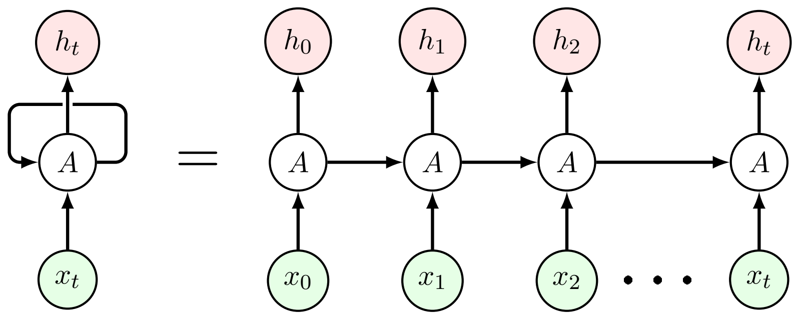


Figure 7: *Illustration of recurrent neurons using the previous output as input for the current node computation.*

The simplest way to incorporate memory is to modify the neuron to store the previous output and include it in the current input. In this way, the historical output is used for the current evaluation in the neuron. This concept of recurrent neurons is illustrated in Figure 7.

One of the limitations of the simple recurrent neuron is that it only takes into account the previous output. This means we are limited to short-term memory. Using more complex logic it is possible to achieve long-term memory, and for this one can use long short-term memory (LSTM) layers. These layers a significantly more complex and compute-intensive, but the long term cells in these layers are capable of storing information long-term. Such models are useful in the case where there are long-term memory is needed to obtain the correct prediction.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Aram Bahrini, Mohammadsadra Khamoshifar, Hossein Abbasimehr, Robert J. Riggs, Maryam Esmaeili, Rastin Mastali Majdabadkohne, and Morteza Pasehvar. Chatgpt: Applications, opportunities, and threats, 2023.

[3] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[4] Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, Jul 2017.

[5] Jan Nagler, Anna Levina, and Marc Timme. Impact of single links in competitive percolation. *Nature Physics*, 7(3):265–270, January 2011.

[6] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview, 2019.

[7] Òscar Lorente, Ian Riera, and Aditya Rana. Image classification with classic and deep learning techniques, 2021.