Introduction to deep-learning

Antonin Verdier

June 2024

1 Introduction

1.1 Principles and learning paradigms

Artificial neural networks (ANNs) are a type of machine learning model that takes inspiration from brain architecture (Krogh, 2008). They are referred to as deep because they contain a large number of layers; hence, the data travel 'deeply' into the network. Mathematically, they can be seen as a collection of non-linear functions. A network is composed of a series of layers. Each layer performs a specific computation on the data and has unique properties. Layers can be further divided into neurons, which are the fundamental units of a deep neural network. One of the simplest neurons we can find is one that will perform a weighted sum of its inputs and pass the results through an activation function, ultimately outputting a 1 or a 0. This neuron alone forms what is called a perceptron (Fig. 1), a fundamental machine learning algorithm and one of the first conceptualized (Rosenblatt, 1958).

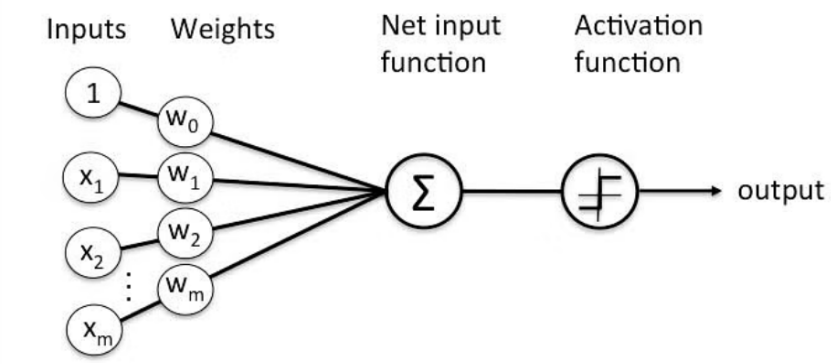

Figure 1: A straightforward schematic of a perceptron, with its inputs, weights and activation function

Nowadays, a basic ANN is composed of an input layer, where the data is fed by the user (Krizhevsky et al., 2012)(LeCun et al., 2015). This data is then transmitted to hidden layers, which computations are usually not shown to the user. Each hidden layer transforms the information from the previous layer and feeds it to the next (Figure 2). The first hidden layer receives data from the input layer. Finally, an output layer collects the information from the last hidden layer and outputs the predicted value. An ANN should be seen as a large mathematical function that transforms the input data into something else. The technique is powerful because the transformation is learned rather than defined in advance. This learning classically relies on a massive dataset that the network will train on.

There are three main categories of learning: supervised, unsupervised and reinforcement learning (Sarker, 2021).

Supervised learning is the most classical type of learning where the network learns to associate an input dataset to a target dataset (Singh et al., 2016). In other words, the network knows the answers to the given problem for many examples and tries to infer the underlying rule to ultimately apply it to new, unseen data. A classic application would be the classification of pictures, where the user possesses a large library of images of cars or planes, each image being correctly labelled. Using this data, a supervised network will be able to learn to classify cars from planes through a trial and error scheme.

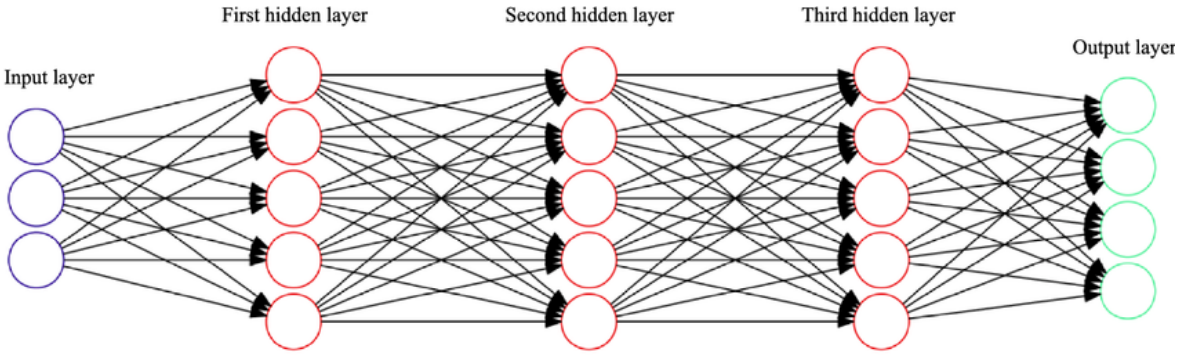

Figure 2: A classic example of a contemporary artificial neural network, Nedjah & al 2019

Reinforcement learning, on the other hand, tries to learn the rule without first knowing the outcome. The model will, during training, learn the set of rules called the policy (Glorennec, 2000). It will apply this policy to an entity called the agent. The policy will drive the agent's behaviour depending on the state of the environment. A good example of such a paradigm would be one that learns to play a video game. At each time, the network must decide if Mario (the agent), given the observation of the environment, should react with action A or B. If Mario is close to a hole, then the policy must dictate for him to jump. On the contrary, if he needs to fit into a tight space in the next frame, then the policy must dictate for him to crouch. The amount of time taken to complete the level can be used to measure the network's performance so it progressively adapts its behaviour.

Lastly, unsupervised learning regroups learning algorithms with access to only the input data and not the answers, unlike supervised learning (El Bouchefry & de Souza, 2020). Thus, the ANN needs to learn on its own (unsupervised) some relevant structures in the data. A classical application of such a network would be the clustering of unlabeled datasets or compression of data. In this scenario, the network must find relevant features that define the input stimulus and remove any redundancy to gain memory space. Importantly, the network must learn which input features are of interest and which can be discarded. In our attempt to create a comprehensive sensory encoding model for cortical stimulation, we use this unsupervised approach for the compression of environmental sounds.

1.2 Generalization

The concept of generalization is central to artificial neural networks. For most use cases, the goal of training an artificial neural network is to be able to use this prediction capabilities with unlabeled data. For the network to perform on previously unseen data, it must learn generalities in the training data and not details, as the details may not be present in the unseen data. To ensure that a network does not learn details, a phenomenon referred to as over-fitting, the labeled data is often split into two subgroups, a training set and a validation or testing set. The training set usually encompasses the majority (around 80%) of the data is use for direct training of the network and updating of the parameters. The validation set, however, is used to measure the performance of the network during training but must never be used to update the weights of the networks. This clear separation ensure that if a network performs well at predicting the validation set, it means that it has learn clear generalities while looking at the training set.

1.3 Architectures

Besides their learning type, artificial neural networks' main versatility comes from their rich diversities in network architectures. In fact, many more neuron arrangements exist than the perceptron, and a lot are still being discovered for various applications (Hussain et al., 2022). The hidden layers inside a neural network can be of numerous types, depending on the desired application. There exist hundreds of network architectures, but for the sake of simplicity, we will focus on the main ones used in neuroscience research.

1.3.1 Fully connected layers

Fully connected layers or linear layers are the most classical type of neural network layer. Given an input, they usually perform a multiplication (which factors are referred to as weights) and an addition (which terms are referred to as bias). Both the weights and the bias will be learned during training. In this scheme, every neuron of the layer is connected to all neurons of the preceding layer (visible in figure 2). This dense architecture allows the layer to learn very complex relations but comes with a high computational cost. In fact, the number of parameters to optimize in a fully connected layer is one the highest of all layer types.

1.3.2 CNN

Many neuroscience models, especially vision neuroscience, are formed with convolution networks (Celeghin et al., 2023). A convolutional network relies on convolutional hidden layers to perform computations. This type of network is known to be excellent at analyzing image data and, therefore, is commonly used to model the visual system. Conceptually, CNNs will find invariant variable in the data, meaning line, curves, corner, independently of their size or orientation. The detection of these invariants allow for characterization of an object as a collection of these invariants.

A convolutional layer comprises a set of kernels, which, in this case, can be considered as the neurons of the layer (Alzubaidi et al., 2021). A kernel is a small matrix. In a CNN, we want to optimize this set of kernels (set of neurons) to extract the relevant features from the data. A kernel is a small matrix that can be seen as a filter that will be applied to the input data.

For the sake of simplicity, let's consider a simple task where we need to classify images of dogs and cats. The network is provided with a set of labelled images, meaning that each image is associated with the true answer (dog or cat). To identify whether the image contains a dog or a cat, the network must analyze large features of the image simultaneously: the shape of the face, the size of the ears or even the spacing of the eyes. Unfortunately, fully connected networks usually perform poorly at these broad-range evaluations and will be too computationally expensive.

The first hidden layer of the model, receives data from the input layer, will convolve the image with its kernels. Usually, the filter is a square of a few pixels wide. Each pixel of the image will get convolved by each kernel of the layer (Fig. 3). The collection of convoluted images will be sent to the new layer up until the output layer.

One could imagine the filter being a 3x3 matrix with all values at 0 except for the middle row, which is set to one. When multiplied with the input image, such matrix, or filter will output a high value only when applied on top of a line that follows the middle row. Otherwise, its output would be close to zero. Therefore, we can quickly see how a collection of such filters, each coding for lines, corners, and circles, can extract meaningful features from the input images. Also, by stacking such layers, we enable deeper layers to apply the same logic to the previously computed kernel set. Thus, the second layer will try to output high values when it encounters a set of lines, circles and corners that form a dog's ear, for example.

Theoretically, as we go deeper into the network, the more "high-level" the features encoded are (Molnar, 2024). Traditionally, these convolutional layers end with a fully connected layer (see Section 1.3.1). Finally, this linear layer will itself pour into the output layer, which reports the final prediction. In our case, the output layer will be two neurons long, one for the dog and one for the cat. For each input passed through the network, the output neurons will contain the probability of the input image being a dog or a cat.

Convolutional neural networks are, therefore, excellent tools for image processing. In machine learning, it is often easier to convert, if possible, the input data into an image so we can use all the tools from the convolutional networks.

Notably, there is a simpler version of CNN layers that are one-dimensional convolutional layers that apply the same principle to signal data using a one-dimensional filter. Such filters can be helpful in ordinary situations, such as designing a bandpass filter, but also in very advanced solutions for sound compression (Oord et al., 2016).

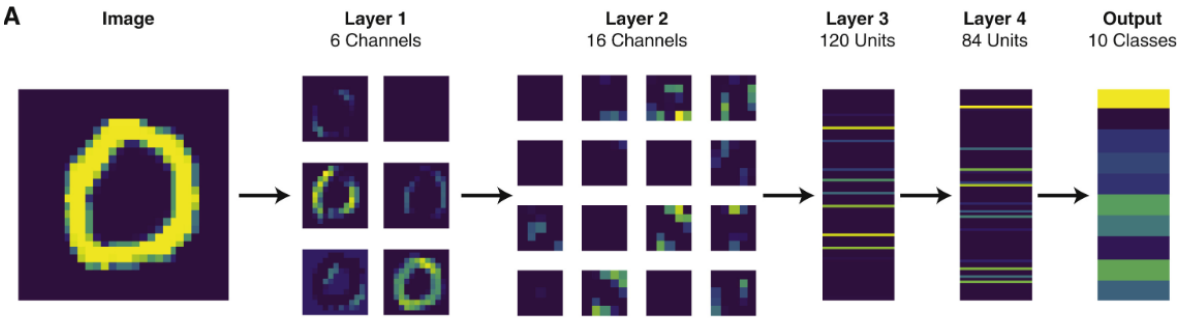

Figure 3: CNN filters, adapted from (Yang & Wang, 2020)

1.3.3 RNN

Recurrent Neural Networks are often referred to as networks with memory. They are usually seen in applications when causality is of strong matter (i.e. time-series) (Yang & Molano-Mazón, 2021) and to mimic behavioural task (Barak, 2017). In a classical RNN, the activity of a neuron at time t is defined by the information from the previous layer and by a new term called connectivity that encapsulates some relevant information from earlier results in time ((Fig. 4). Such networks are historically difficult to train (Pascanu et al., 2013) because of the exploding or vanishing gradient problems (Indrajitbarat, 2023).

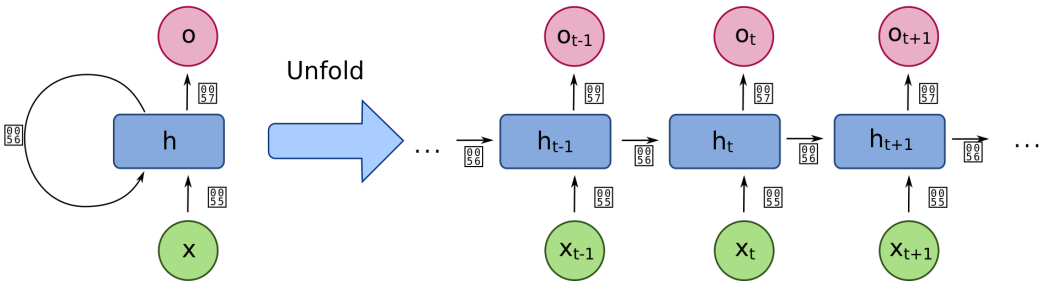

Figure 4: Cartoon of a recurrent neural network unit.

There are countless other architectures of neural networks, such as reservoir networks, Long Short Term Memory (LSTM) networks, transformers, residual networks, generative adversarial networks, etc. Each type of neural network aims to answer a specific question or address a particular limit of other architecture. It is important to mention the very recent transformer networks published in 2017 (Vaswani et al., 2023), whose usage and recognition exploded since their use in public products such as DALL.E 2 or ChatGPT.

1.4 Objectives functions

Neural networks are not only composed of layers of neurons but can also comprise objective functions that are instrumental in the learning process and help the networks reach the required performances.

1.4.1 Activation

Most neurons in an ANN have an activation function, which basically describes the computation applied to the output of a neuron. It helps homogenize the response of all neurons and drastically increases learning speed. Usually, this activation function is non-linear (Szandała, 2021), the most

common one would be the Rectified Linear Unit function (ReLU) (Bai, 2022):

$$f(x) = max(x, 0) \tag{1}$$

While the derivative at 0 is undefined, it is set at 0 in practice to allow for backpropagation (see Section 2.1).

Due to the vanishing gradient problem (see Section 2.2), the hyperbolic tangent is more commonly utilized in recurrent neural networks. These activation functions can be either saturating or non-saturating (Ding et al., 2018; Xu et al., 2016).

1.4.2 Normalization

In deep neural networks, normalization or regularisation functions are essential to keep the overall parameters of the network in a manageable range. There exists a variety of normalization methods that either normalize the weights to keep them inside a given range or normalize the data to make the training set more homogenous. Proper data normalisation has been shown to reduce training time and accuracy but does not constitute an imperative for efficient training (Shao et al., 2020).

1.4.3 Losses

Undoubtedly, the most critical objective function in a neural network is the loss function, which, given the network's current predictions, can compute the amount of error. Loss functions are primordial in training because they are the primary drive for the network's weight adjustments. Various loss functions exist depending on the type of data and goal to achieve (Terven et al., 2023).

The most common loss function in use for regression tasks is the Mean Squared Error function, defined as such:

$$L(x, \theta) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \hat{x}_i)^2 \tag{2}$$

Where θ represents the parameters of the model, N the number of samples, x the input data and $\hat{x}$ the predicted data. $x_i \in x_1, ..., x_N$. x_i represents a training example and N is the total number of training examples available.

During training, the MSE will be computed for each batch of input data, and the error will be backpropagated in the network to adjust the weights.

The binary cross entropy log function is preferred for a model in which the output is a probability between 0 and 1. It is the most popular for classification tasks and is defined as such:

$$L(y, \theta) = -\frac{1}{m} \sum_{n=1}^{m} y_i * log(p_i)) \tag{3}$$

Here, m is the number of different classes (or categories in which to classify). y is the target label and p_i the predicted probability of the input data to belong to class i.

A powerful feature of ANNs is that the loss function can be customised to fit the exact needs of the task. By adding other terms to the loss function, it is possible to encourage the network to learn to minimize multiple errors simultaneously. Still, adding terms to the loss function increases its complexity and make learning difficult. When designing such loss functions, it is necessary to carefully weigh each term and ensure that each term minimization does not compete with another to avoid network instability.

Collectively, layers with activation, normalization and loss functions form a network. The final organization of these components will define the overall architecture of the network. One particular organization, the autoencoder, popular for data compression, is of interest in this manuscript, as we aim to compress and transform incoming sounds for auditory restoration.

```python
# Initialization of the training loop
optimizer = torch.nn.MSELoss()
model.train()

# Start of the training loop
for batch_idx, (data, target) in enumerate(track(training_loader)):

    optimizer.zero_grad() # Initializing the optimizer

    output = mdoel(data) # Prediction by the network
    loss = reconstruction_loss(output, target) # Error computation

    loss.backward() # Backpropagating the loss
    optimizer.step() # Update the weights
```

Listing 1: A short model training loop in PyTorch

2 Building a neural network: A survival guide

If the theory behind neural networks is often extensively described and discussed, only a few resources correctly detail how to build a neural network from available resources. Moreover, some abstractions operated by contemporary programming libraries can be opaque. This section aims to describe the essential steps for building a neural network and the challenges that come with it.

2.1 Practical run

Two main libraries are available nowadays in Python for building neural networks: Keras (Tensorflow) and PyTorch. Keras is often regarded as more user-friendly, high-level and straightforward to use. It is possible to train a complete model with a few lines of code. While simple to use, PyTorch requires a more detailed minimal structure to run but can be more flexible for complex models. It also provides insights into individual steps when training the neural network. In the backend, the main difference is that PyTorch uses a dynamical computational graph, meaning that the structure of the model is computed on the fly, while the architecture in Keras must be fixed before training. Here is a hands-on example of a training loop of an autoencoder (See Code 1) using the PyTorch package.

This code example assumes that a model called *model* is available and trainable.

For an ANN to accomplish the required transformation of the input data into the output data, a training paradigm must be implemented. Considering that we have at our disposal a vast dataset of n samples (here *data*) and the correct answer for each one (here *target*), we can use them to train our neural network. In some cases, for memory considerations, training examples and target examples are provided by a unique object (here *training_loader*).

At the beginning, all internal parameters of all layers are initialised at random (this step is usually performed at the creation of the model before entering the training loop for the first time). Then, samples are sent in small packages called batches, and a prediction is made by the network (here, a batch would be represented by the variable *data*, which encapsulates at each loop a different subset of the *training_loader* dataset). A prediction (here *output*) is made for any input sample provided to the network. The prediction is compared to the correct answer, and an error (or loss) is calculated using a loss function (here, a mean squared error function). This entire process is often referred to as the forward pass.

Following a forward pass, the backward pass (here simply illustrated by the *loss.backward()* method) sends back the error into the network to compute the error gradients through a process called backpropagation (Rumelhart et al., 1986). Knowing the magnitude of the error given by the loss function, we can infer the contribution of each layer to this error and, ultimately, in which direction to tweak the parameters. Here, the gradients are stored in the optimizer object, which will further adjust how the backpropagation is made following hyperparameters such as learning rate, momentum, etc. After computing the gradients with the *loss.backward()* method, the optimizer applies them on the last line. By tweaking the parameters, we expect to produce a better prediction on the next forward pass. This optimization is performed for each batch of data.

Looping through the entire dataset in batches and performing a forward and backward pass each

time constitutes an epoch. Depending on the task, repeating these epochs hundreds of times will eventually make the network converge to a stable solution.

2.2 Challenges

Building a neural network presents many challenges, and solutions are often empirical. Depending on the distribution of the input data, outliers, activation functions, and layer size, different errors and limitations can arise. I will here describe two problems encountered when building classical encoding models as well as essential parameters.

2.2.1 Gradient vanishing

Gradient vanishing occurs when the network optimises its weights to be zero (Haroon-Sulyman et al., 2024). In that case, no matter the input, the output data will be null, and training will end. This usually happens for very sparse data, where predicting only zeros still results in a decent loss minimization score. Also, it is known that some activation functions are prone to gradient vanishing. ReLU, for example, as it outputs a hard 0 for every negative input, will increase the tendency of the network to fall into this vanishing regime. One may use a different activation function with a soft 0, such as the hyperbolic tangent as a counterbalance measure. If possible, the sparsity of the data should be reduced.

2.2.2 Gradient exploding

Gradient exploding is the opposite of the vanishing problem, where to minimize a loss, the network weights go to extremely high values (Philipp et al., 2018). This is often caused when the data is not normalized, and the activation functions are not bounded. Again, the ReLU function allows for infinite values above zero. Since many computations in the networks are multiplicative, weights above one increase the tendency of exploding gradients. Changing the activation function for a bounded alternative, such as a sigmoid or softmax function, may help.

2.2.3 Hyperparameters

Learning rate (LR) and batch size are two parameters crucial for learning a network. As its name indicates, the learning rate is an arbitrary value that guides how fast the network should learn from each optimization. Setting an LR too low might significantly increase the time necessary for converging. On the other hand, a too-strong learning rate will make the network learn too fast, and each input particularity will strongly influence the network weights. This translates to an unstable network, jumping from a good loss score to a bad loss score, depending on the input. Often, the network will learn too much from the training data and become overfitted. This means that when asked to predict new, unseen data, the network will perform poorly because it is perfectly tuned to the previously seen data and will not generalize. Batch size corresponds to how many individual input examples will be concatenated for each training loop. In fact, it is too computationally expensive and difficult to train a network by looping over every input data example. To go around this problem, the input data is grouped into batches of often 32 to 128 examples, depending on the size of the dataset, and the entire batch goes through the network at once. Choosing the right batch size and learning rate combination can often be tricky when dealing with small datasets.

With this chapter, our objective is to understand the solutions available to us for building a sensory encoding model of the auditory tract. Now that we have seen how artificial neural networks function and how we can implement them, we can dive further into their similarities with the human brain. Importantly, we should describe the limitations of such systems to better grasp future challenges when using them in a cortical stimulation paradigm.

3 Example model

```
# Minimal implemantation of an autoencoder
class Model(nn.Module):
    def __init__(self, size):
        super(Model, self).__init__()

        # Declaration of 3 Convolutional layers of size 16, 32 and 64 kernels
        self.conv1 = nn.Conv2d(1, 16, kernel_size=4, padding=(0, 0), stride=(1, 1))
        self.conv2 = nn.Conv2d(16, 32, kernel_size=4, padding=(0, 0), stride=(1, 1))
        self.conv3 = nn.Conv2d(32, 64, kernel_size=4, padding=(0, 0), stride=(1, 1))

        # Declaration of a flattening layer
        self.flatten = nn.Flatten(-1, 100)

        # Declaration of the ReLU function
        self.relu = nn.ReLU()

    # Method for the forward pass. The input data x goes through each layer sequentially
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)

        x = self.conv2(x)
        x = self.relu(x)

        x = self.conv3(x)
        x = self.relu(x)

        x = self.flatten(x)

        return x
```

References

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), 53. https://doi.org/10.1186/s40537-021-00444-8

Bai, Y. (2022). RELU-function and derived function review (A. Luqman, Q. Zhang, & W. Liu, Eds.). *SHS Web of Conferences*, *144*, 02006. https://doi.org/10.1051/shsconf/202214402006

Barak, O. (2017). Recurrent neural networks as versatile tools of neuroscience research. *Current Opinion in Neurobiology*, *46*, 1–6. https://doi.org/10.1016/j.conb.2017.06.003

Celeghin, A., Borriero, A., Orsenigo, D., Diano, M., Méndez Guerrero, C. A., Perotti, A., Petri, G., & Tamietto, M. (2023). Convolutional neural networks for vision neuroscience: Significance, developments, and outstanding issues [Publisher: Frontiers]. *Frontiers in Computational Neuroscience*, *17*. https://doi.org/10.3389/fncom.2023.1153572

Ding, B., Qian, H., & Zhou, J. (2018). Activation functions and their characteristics in deep neural networks [ISSN: 1948-9447]. *2018 Chinese Control And Decision Conference (CCDC)*, 1836–1841. https://doi.org/10.1109/CCDC.2018.8407425

El Bouchefry, K., & de Souza, R. S. (2020, January 1). Chapter 12 - learning in big data: Introduction to machine learning. In P. Škoda & F. Adam (Eds.), *Knowledge discovery in big data from astronomy and earth observation* (pp. 225–249). Elsevier. https://doi.org/10.1016/B978-0-12-819154-5.00023-0

Glorennec, P. Y. (2000). Reinforcement learning: An overview.

Haroon-Sulyman, S. O., Taiye, M. A., Kamaruddin, S. S., & Ahmad, F. K. (2024). Systematic literature review and bibliometric analysis on addressing the vanishing gradient issue in deep neural networks for text data. In N. H. Zakaria, N. S. Mansor, H. Husni, & F. Mohammed (Eds.), *Computing and informatics* (pp. 168–181). Springer Nature. https://doi.org/10.1007/978-981-99-9589-9_13

Hussain, H., Tamizharasan, P. S., & Rahul, C. S. (2022). Design possibilities and challenges of DNN models: A review on the perspective of end devices. *Artificial Intelligence Review*, *55*(7), 5109–5167. https://doi.org/10.1007/s10462-022-10138-z

Indrajitbarat. (2023, August 13). *Recurrent neural networks (RNNs): Challenges and limitations* [Medium]. Retrieved March 20, 2024, from https://medium.com/@indrajitbarat9/recurrent-neural-networks-rnns-challenges-and-limitations-4534b25a394c

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*. Retrieved July 5, 2024, from https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Krogh, A. (2008). What are artificial neural networks? [Publisher: Nature Publishing Group]. *Nature Biotechnology*, *26*(2), 195–197. https://doi.org/10.1038/nbt1386

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning [Publisher: Nature Publishing Group]. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Molnar, C. (2024). *10.1 learned features — interpretable machine learning*. Retrieved March 20, 2024, from https://christophm.github.io/interpretable-ml-book/cnn-features.html

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., & Kavukcuoglu, K. (2016, September 19). WaveNet: A generative model for raw audio. https://doi.org/10.48550/arXiv.1609.03499

Pascanu, R., Mikolov, T., & Bengio, Y. (2013, February 15). On the difficulty of training recurrent neural networks. https://doi.org/10.48550/arXiv.1211.5063

Philipp, G., Song, D., & Carbonell, J. G. (2018, April 6). The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions. https://doi.org/10.48550/arXiv.1712.05577

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain [Place: US Publisher: American Psychological Association]. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors [Number: 6088 Publisher: Nature Publishing Group]. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Sarker, I. H. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Computer Science*, *2*(3), 160. https://doi.org/10.1007/s42979-021-00592-x

Shao, J., Hu, K., Wang, C., Xue, X., & Raj, B. (2020). Is normalization indispensable for training deep neural network? *Advances in Neural Information Processing Systems*, *33*, 13434–13444. Retrieved March 4, 2024, from https://proceedings.neurips.cc/paper/2020/hash/9b8619251a19057cff70779273e95aa6-Abstract.html

Singh, A., Thakur, N., & Sharma, A. (2016). A review of supervised machine learning algorithms. *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, 1310–1315. Retrieved March 20, 2024, from https://ieeexplore.ieee.org/abstract/document/7724478

Szandała, T. (2021). *Review and comparison of commonly used activation functions for deep neural networks* (Vol. 903). https://doi.org/10.1007/978-981-15-5495-7

Terven, J., Cordova-Esparza, D.-M., Ramirez-Pedraza, A., & Chavez-Urbiola, E. (2023, July 5). *Loss functions and metrics in deep learning. a review.*

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023, August 1). Attention is all you need. https://doi.org/10.48550/arXiv.1706.03762

Xu, B., Huang, R., & Li, M. (2016, May 2). Revise saturated activation functions. https://doi.org/10.48550/arXiv.1602.05980

Yang, G. R., & Molano-Mazón, M. (2021). Towards the next generation of recurrent network models for cognitive neuroscience. *Current Opinion in Neurobiology*, *70*, 182–192. https://doi.org/10.1016/j.conb.2021.10.015

Yang, G. R., & Wang, X.-J. (2020). Artificial neural networks for neuroscientists: A primer. *Neuron*, *107*(6), 1048–1070. https://doi.org/10.1016/j.neuron.2020.09.005