The Adversarial Robustness of Neural Networks

u2001820

2021/22

Abstract

We shall introduce machine learning with the linear regression model, displaying an example of how it works in practise. We shall then define what a neural network is and how it can be trained to 'learn' a set of data, hence allowing it to classify unseen inputs. Finally, we will discuss adversarial attacks and what it means for a neural network to be stable. From this, we shall explore the trade-off between accuracy and stability of a neural network, touching on potential solutions facing contemporary neural network security.

Contents

1 What *is* Machine Learning?

Machine learning deals with training a computer to perform tasks and functions automatically [6, p.1]. A machine learning model can be trained to distinguish images and predict costs of stocks, but can also be utilised for more interesting things like developing self-driving cars and diagnosing medical diseases from a patient's prognosis. As one might imagine, such models are not perfect and are susceptible to being fooled- this relates to the topic of adversarial robustness, which our discussion shall converge to.
In the meantime, we begin with a brief explanation of how machine learning works. Suppose we have two sets of data: a training data set $\mathcal{T} = (x^1, \ldots, x^r)$ to train the Machine Learning model with and a validation data set $\mathcal{V} = (y^1, \ldots, y^s)$ to test the ML[1] model against. We say that the ML model has been trained successfully if it reasonably classifies a certain proportion of the data in $\mathcal{V}$ correctly[2]. In other words, the ML model learns from $\mathcal{T}$ and is then tasked with classifying the unforeseen $\mathcal{V}$.
A machine can 'learn' in many different ways, depending on what form the dataset takes. We begin by considering data that may be linearly correlated.

2 Linear Regression

2.1 Problem Setup

Suppose we have a set of points on the Cartesian plane $\{(x^1, y^1), \ldots, (x^m, y^m)\}$ and we wish to find the line of best fit for these points. Assuming a linear relationship exists between the x and y coordinates, we aim to optimise the vector $\theta = (\theta_0, \theta_1)$ so that the approximation $y \approx \theta_0 + \theta_1 x$ is as good as possible. From this, we define the *hypothesis function* for linear regression as $h_\theta(x) \coloneqq \theta_0 + \theta_1 x$. The hypothesis function is the function we wish to assimilate to the values of the vector y as best we can. We can quantify the accuracy of our hypothesis function by computing what is known as the *cost function*:

$$C(\theta) \coloneqq \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m} \|h_\theta(x) - y\|^2 \tag{1}$$

i.e. for each data point $(x^{(i)}, y^{(i)})$, find the square of the difference between the value of the h_θ at that $x^{(i)}$ and the actual $y^{(i)}$ value. Then, sum this up across all the points in the dataset. The $\frac{1}{2m}$ represents taking the average of the errors incurred across all the data points. In other words, the cost

[1] We shall abbreviate 'Machine Learning' to ML from now on.

[2] The meanings of 'reasonably classifies' and 'certain proportion' are not well-defined; it is up to the person training the ML model to quantify these terms.

function allows us to quantify how *wrong* our h_θ is; the smaller the value of the cost function for the given vector θ, the better the hypothesis h_θ.
Our aim is now to optimise the vector $\theta = (\theta_1, \theta_2)$ such that $C(\theta)$ is minimised. If we can do this, then we've essentially found our 'line of best fit' to solve our linear regression problem. Let us now discuss one possible way of achieving this.

2.2 Gradient Descent

Suppose we now have a cost function of n parameters, i.e. $C : \mathbb{R}^n \to \mathbb{R}$, where C takes input of the form $(\theta_1, \ldots, \theta_n)$. It is up to the user to state the initial values of each θ_i. Assuming the existence and continuity of the partial derivatives of C, we can then utilise the following algorithm.

Algorithm 1 Gradient Descent Algorithm

$\theta_j \coloneqq \theta_j - \dfrac{\partial C}{\partial \theta_j}$

UPDATE SIMULTANEOUSLY for all $j = 0, ..., n-1$

Updating $\theta_1, \ldots, \theta_{n-1}$ simultaneously is important. This can be done by storing the value of h_θ in some variable h_{temp} at the start of each new iteration of the algorithm. The new values of $\theta_0, \ldots, \theta_{n-1}$ can then be calculated using h_{temp}.
Relating this to our linear regression problem, we will apply Algorithm (1) to $C : \mathbb{R}^2 \to \mathbb{R}$ as defined in Equation (1). One easily verifies, using the chain rule, that

$$\frac{\partial C}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

and

$$\frac{\partial C}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}.$$

This gives us a closed form of the gradient descent algorithm for linear regression:

Algorithm 2 Gradient Descent Algorithm for Linear Regression

$\theta_0 \coloneqq \theta_0 - \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$

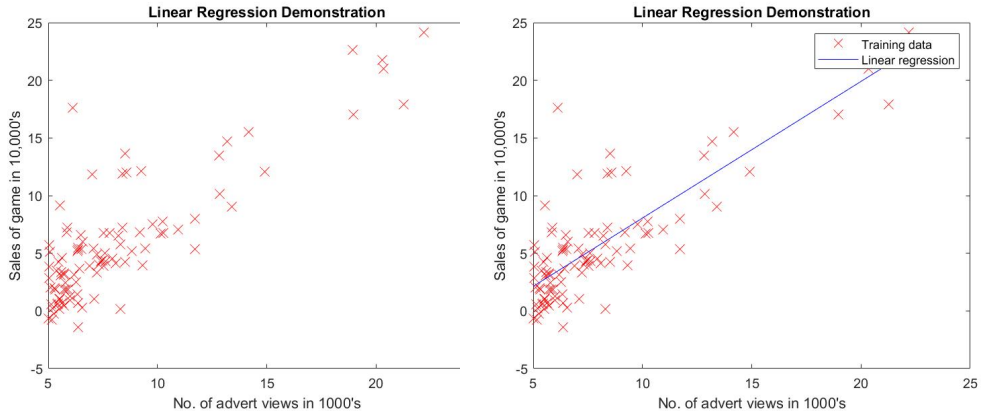
$\theta_1 \coloneqq \theta_1 - \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$

UPDATE SIMULTANEOUSLY for θ_0, θ_1.

One can plot a graph of $C(\theta_1, \theta_2)$ against the number of iterations of the algorithm. If this is a strictly decreasing function, then the error incurred by the approximation is reducing for each iteration, meaning the algorithm is working correctly.

2.3 A Tangible Example

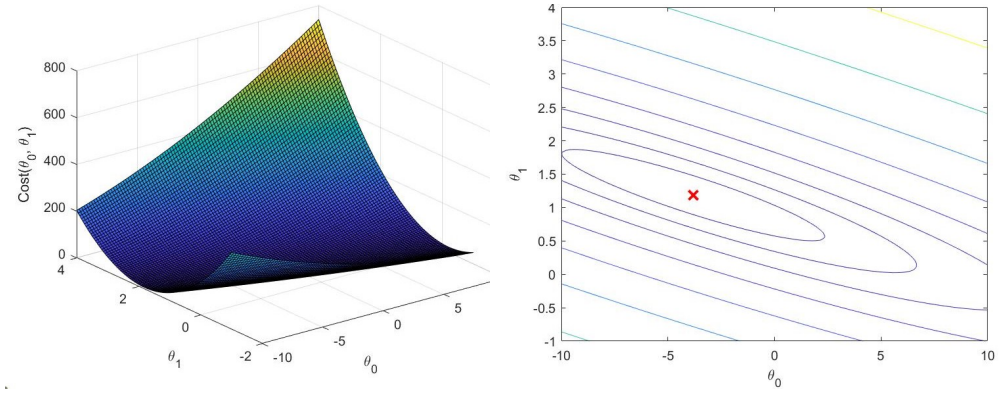
The following example features code provided in the Coursera: Machine Learning online course [5].
Suppose a video game store wishes to identify the correlation (if any) between the number of times an advert for a game is viewed and how many sales are made for that game. The scatter plot for this data may look like that of Figure (1a). The plotted points can be thought of as the training dataset $\mathcal{T}$.



(a) Original scatter plot (b) Scatter plot with line of best fit

Figure 1: Scatter plots showcasing linear regression.

We begin our gradient descent algorithm with $(\theta_0, \theta_1) = (0, 0)$, generating a rather large cost of 32.030658. We choose to carry out 3000 iterations, leading us to $(\theta_0, \theta_1) = (-3.795429, 1.184909)$. (Note that the x-axis in these plots begins from $x = 5$.) This is a pretty good estimation, given how Figure (1b) looks relative to the data.
The surface plot of C with respect to its parameters and a contour plot of this surface have also been provided, displayed in Figures (2a) and (2b) respectively. The red marker on Figure (2b) represents the minimum point suggested by our model.



(a) Surface Plot (b) Contour Plot

Figure 2: Further visualisations of the linear regression solution.

3 Neural Networks

A neural network can be used to learn a dataset and carry out identification tasks on unforeseen datasets far larger (and potentially more complex) for humans to deal with manually. The neural network can 'learn' the dataset in a similar way to the linear regression model: the neural network itself has parameters which we aim to optimise by minimising a certain cost function. We abbreviate 'neural network' to NN from now on.

3.1 The Sigmoid Function

Before delving into the maths behind neural networks, we must introduce a very important non-linear function used abundantly in NN computations.

Definition 3.1. The *Sigmoid Function* [7, p. 33] is defined as $\sigma : (-\infty, \infty) \to (0, 1)$, where $\sigma(z) \coloneqq \frac{1}{1+e^{-z}}$.

The sigmoid function allows us to 'squish' any real value to a real value between 0 and 1, a property which will soon prove invaluable. The function has another appealing property:

Lemma 3.1 (Derivative of the Sigmoid Function)**.** *Taking* $\sigma(z) \coloneqq \frac{1}{1+e^{-z}}$ *on* $\mathbb{R}$*, we have that*

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

We will later observe a recurrence relation that utilises the derivative of $\sigma(z)$, so this closed form will significantly boost computational efficiency. In reality there are many different sigmoid functions we can choose to use [7, p. 35], such as $\tanh : (-\infty, \infty) \to (-1, 1)$, so long as the function applies the desired scaling to the input.

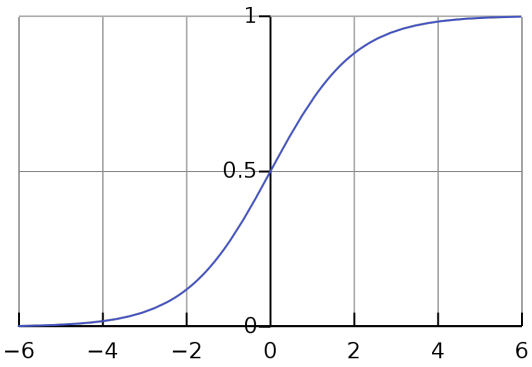


Figure 3: The Sigmoid Function.

3.2 Examples of Neural Networks

All neural networks are comprised of an input layer and an output layer. Each layer is comprised of nodes called neurons. Any other layers between the input and output layers are called hidden layers. A neural network with any number of hidden layers is called a *deep neural network*. We attribute a value in the interval (0,1) to every neuron in each layer. This is called the **activation value** of the neuron.

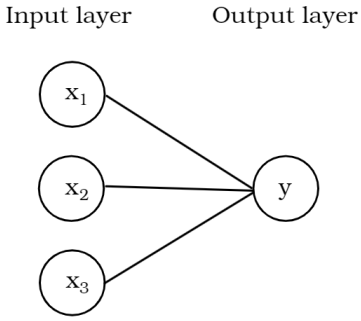


Figure 4: A Neural Network with 2 layers

Example 3.1. The neural network detailed in Figure 4 is comprised of an input layer and an output layer and $x_1, x_2, x_3, y \in \mathbb{R}$. This NN uses the values x_1, x_2, x_3 to compute the value of y. Each x_i is attributed a scalar-valued 'weight', denoted w_1, w_2, w_3 respectively. We also include another scalar value b, called the 'bias'. So the values x_i, w_i, b for $i \in \{1, 2, 3\}$ are used to compute the activation y. The formula for this is as follows:

$$y = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + b),$$

yielding a y value in the range (0,1).

We can say that this neural network is *parametrised* by w_1, w_2, w_3, b. Assuming we have a cost function C for the neural network, we can evaluate the cost function at a starting point $(\tilde{w}_1, \tilde{w}_2, \tilde{w}_3, \tilde{b})$ and apply the gradient descent algorithm in the hopes that we converge to parameters that minimise $C(w_1, w_2, w_3, b)$. We will specify our chosen cost function later on, but in reality we can define the cost function however we like, so long as it quantifies how inaccurate the neural network is with respect to the parameters w_1, w_2, w_3, b.

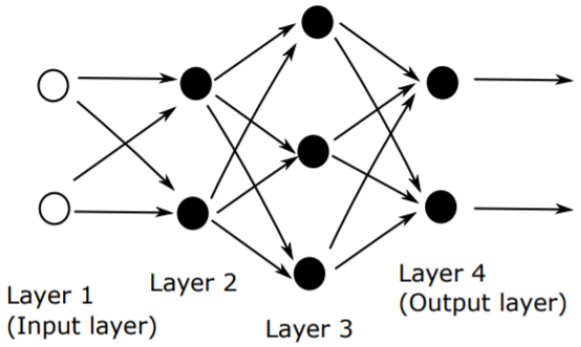


Figure 5: A Neural Network with four layers [4, p. 5].

Example 3.2. Figure 5 shows a neural network with four layers and 9 neurons in total. We will denote the activation of the k-th neuron in layer l as $a_k^{(l)}$. We can ascertain the value of $a_k^{(l)}$ using the same steps as before, except this time we have more than one hidden neuron and, as such, more dense notation.
Let's find the value of $a_1^{(2)}$, the activation of the first neuron in the second layer. We have that

$$a_1^{(2)} = \sigma(w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)})$$

where $w_{jk}^{(l)}$ denotes the weight of activation $a_k^{(l-1)}$ on the activation $a_j^{(l)}$ and $b_j^{(l)}$ denotes the bias acting on $a_j^{(l)}$. This notation will only get more cumbersome, so let's involve some matrix notation.

Suppose we have a neural network with L layers. Suppose that layer l contains n_l neurons, for $l \in \{1, \ldots, L\}$. We can define $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ to be the matrix containing the weights from layer $l-1$ to layer l, i.e.

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & & \vdots \\ \vdots & & \ddots & \\ w_{n_l 1}^{(l)} & \cdots & & w_{n_l n_{l-1}}^{(l)} \end{pmatrix},$$

and $b^{(l)} \in \mathbb{R}^{n_l}$ to be the vector of biases from layer $l-1$ to layer l:

$$\left[b_1^{(l)}, b_2^{(l)}, \ldots, b_{n_l}^{(l)}\right]^T.$$

If we set $a^{(l)}$ to be the vector containing all the activations in layer l, then we have the following recurrence relation for $l \in \{2, \ldots, L\}$:

$$a^{(l)} = \sigma(W^{(l)} a^{(l-1)} + b^{(l)})$$

which can simplified further by setting $z^{(l)} \coloneqq W^{(l)} a^{(l-1)} + b^{(l)} \in \mathbb{R}^{n_l}$. Hence, for $l \in \{2, \ldots, L\}$,

$$a^{(l)} = \sigma(z^{(l)}).$$

A few remarks: each application of the above recurrence relation is known as a *forward pass*. This is because we are passing information from layer $l-1$ to layer l and updating all subsequent activation values. Also, notice that σ as above is now implicitly a vector-valued function, with our σ as in Definition (3.1) simply applied elementwise. Finally, how many weight and bias parameters are involved in this example? It turns out that there are 23 parameters to optimise over[3]! Notice how rapidly the number of unknowns increases after adding only slightly more complexity to the network.

3.3 Backpropagation

How do we evaluate and minimise the cost function for a neural network? Recall Example (3.1); a NN with 3 neurons in its input layer and 1 neuron in its output layer. Recall that $a = \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3 + b) = \sigma(z)$ where $z \coloneqq w_1 x_1 + w_2 x_2 + w_3 x_3 + b$. In this example, our cost function is just

$$C(w_1, w_2, w_3, b) = \frac{1}{2}(a - y)^2$$

with the $\frac{1}{2}$ as an arbitrary constant to simplify the following algebra a bit. Recall that we can define the cost function however we wish, so long as it quantifies how 'badly' our model has performed. We now require all the partial derivatives encapsulated in the vector

$$\nabla C = \left[\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial w_3}, \frac{\partial C}{\partial b}\right]^T$$

and so we get

[3] [4, p. 6] $W^{(2)} \in \mathbb{R}^{2\times 2}, b^{(2)} \in \mathbb{R}^2, W^{(3)} \in \mathbb{R}^{3\times 2}, b^{(3)} \in \mathbb{R}^3, W^{(4)} \in \mathbb{R}^{2\times 3}, b^{(4)} \in \mathbb{R}^2$, i.e. $2 \times 2 + 2 + 3 \times 2 + 3 + 2 \times 3 + 2 = 23$.

$$\begin{aligned}\frac{\partial C}{\partial w_i} &= \frac{\partial C}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial w_i} && \text{(Chain rule)}\\ &= (a-y)\cdot\sigma'(z)\cdot x_i\\ &= (a-y)\cdot\sigma(z)(1-\sigma(z))\cdot x_i && \text{(By Lemma 3.1).}\end{aligned}$$

Similarly, $\dfrac{\partial C}{\partial b} = (a-y)\cdot\sigma(z)(1-\sigma(z))$. We now have all our partial derivatives in terms of our cost function parameters, so we can plug everything into our gradient descent algorithm and iterate as much as needed. That's great, but does this generalise at all for more complex neural networks?

3.4 The general case

[4, pp. 12–14] We begin this subsection by introducing

$$\delta_j^{(l)} \coloneqq \frac{\partial C}{\partial z_j^{(l)}}$$

which represents the *sensitivity* of the cost function to the weighted input for neuron j at layer l. Hence, $\delta^{(l)}$ is just the vector containing all the $\delta_j^{(l)}$ for $j \in \{1,\dots,n_l\}$, i.e.

$$\delta^{(l)} = \left[\frac{\partial C}{\partial z_1^{(l)}}, \frac{\partial C}{\partial z_2^{(l)}}, \dots, \frac{\partial C}{\partial z_{n_l}^{(l)}}\right]^T.$$

Remember that $z_j^{(l)}$ is just the value of $a_j^{(l)}$ before we apply the sigmoid squishing function. The following propositions will allow us to write ∇C in terms of just the weights and biases:

Proposition 3.2 (Closed form for the sensitivities of a Neural Network)**.** *[4, p. 12] For a neural network with L layers and n_l neurons in layer l:*

$$\delta^{(L)} = \sigma'(z^{(L)}) \circ (a^L - y) \tag{2}$$

$$\delta^{(l)} = \sigma'(z^{(l)}) \circ (W^{(l+1)})^T \delta^{(l+1)} \qquad \text{for } 2 \le l \le L \tag{3}$$

where $\circ$ denotes component-wise multiplication and W denotes the matrix of weights associated with the neural network.

Proposition 3.3 (Computing ∇C for a Neural Network)**.** *[4, p. 12] The partial derivatives of the cost function with respect to its parameters are given by*

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)} \qquad \text{for } 2 \le l \le L, \tag{4}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \qquad \text{for } 2 \le l \le L. \tag{5}$$

The full proofs of both propositions are provided in [4], on pages $12 - 14$. We now have all the components of ∇C in terms of just weights and biases as required. Now we can choose a starting value for each weight and bias, encapsulated in the below string of weight matrices and bias vectors

$$\left(W^{(2)}, b^{(2)}, \ldots, W^{(n_L)}, b^{(n_L)}\right),$$

and apply the gradient descent algorithm to minimise C, thus training the neural network!
Potential applications of neural networks are detailed below.

3.5 Decision Boundaries

A neural network can be trained to learn a non-linear pattern in data. Suppose we plot a set of points on the Cartesian Plane as shown in Figure (6).

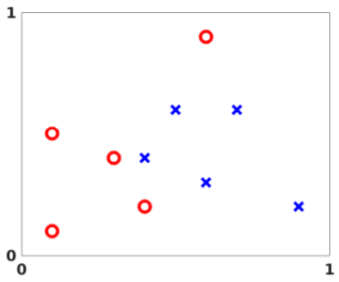
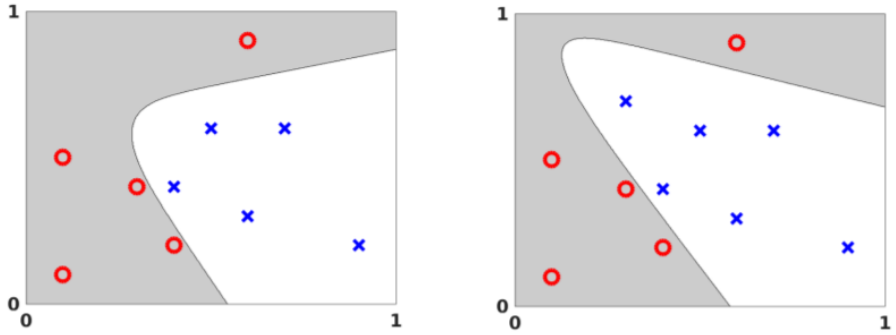


Figure 6: A set of plotted points. Points in Category A are marked as red noughts while points in Category B are marked as blue crosses [4, p. 3]

We can train a NN to return a *decision boundary* that the computer uses to determine how to mark newly plotted points. The number of training examples will affect the accuracy of this decision boundary too. Contrast the plots in figure (7) to gain an appreciation for both how to apply a NN and how to improve its accuracy.

3.6 The MNIST Database

The MNIST database [`http://yann.lecun.com/exdb/mnist/`] consists of $70,000$ images of the grayscale handwritten digits $0 - 9$. We can construct and train a neural network to recognise these digits: for example, we could train the network on $60,000$ of the images and then test it against the remaining $10,000$. An image in the dataset could consist of 28×28 pixels and so can be represented by a vector $x \in \mathbb{R}^{784}$. Figure 8 showcases a neural network architecture that could be trained on the MNIST dataset. The input



(a) Decision boundary for 10 points (b) Decision boundary for 11 points

Figure 7: Visualisations of the decision boundary produced by a NN for points plotted in $\mathbb{R}^2$ [4, pp. 6–7].

layer could consist of 784 neurons with each neuron's value representing the colour of a pixel in the input image. The output layer could consist of 10 neurons and the neuron with the highest activation could be interpreted as the output of the neural network. We can choose to include any number of hidden layers, but the question of making the most optimal choice is well beyond the scope of this essay. If we take the highest output activation to be the network's perceived output, then the 0.8 activation in Figure 8 could suggest recognition of the digit '1'.

4 Adversarial Robustness

4.1 Image perturbation

Let us extend the concept discussed in the previous section. Suppose we now wish to train a neural network to recognise full-colour pictures of animals. Suppopse the neural network has been trained and can now classify pictures of animals with high accuracy. Is it now possible to now 'trick' the neural network into misclassifying an image it ought to recognise relatively easily? Figure 9 showcases an example of this: suppose we overlay the image of a panda with a grid of pixels with seemingly random attributed colours. (The network in Figure 9 recognises this to be a worm: this is complete nonsense, but image classifiers have to output something!). It may be possible to engineer this perturbed input to change the colours of the standard image just enough so that the neural network incorrectly classifies the image without drastically modifying how the standard input looks. This can cause major issues; suppose a neural network is used for a user authentication system, restricting access to a location or utility only to authorised personnel. Adversarial attacks could make it possible for hackers to infiltrate such systems,

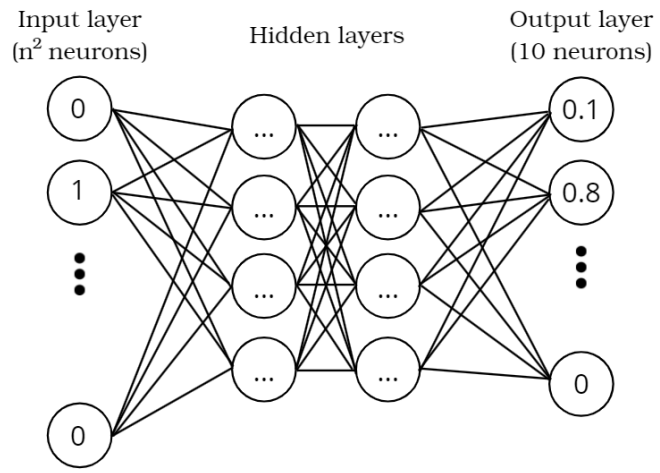


Figure 8: A Neural Network that could be trained to learn the MNIST database. Each of the neurons in the hidden layer has an activation value in the range $[0, 1]$. (For the sake of demonstration, we allow for activation values equal to 0 and 1.)

rendering all security obsolete.
This possibility motivates the following definition.

Definition 4.1. An *adversarial input* to a NN is a perturbation Δx to a standard input x that is misclassified by the NN.

4.2 A set-theoretical approach

Before exploring the concept of adversarial robustness, we must first define neural networks and their features using set theory.

Definition 4.2 (Set-theoretical definition of a neural network)**.** [2, p. 3] Suppose we have a dataset of input $\mathcal{X} = \mathbb{R}^d$ and output $\mathcal{Y} = \{1, \ldots, m\}$ that can be represented by some classification function $f(\mathcal{X}) = \mathcal{Y}$. Our objective is to train a neural network to 'learn' the relationship f as accurately as possible. We define $\mathcal{NN}_{\mathbf{N},L}$ to be the set of all L-layer neural networks with N_l neurons in the l^{th} layer, where $\mathbf{N} \coloneqq (N_1, \ldots, N_L)$ denotes the dimensions of the neural network, with $N_1 = d, N_L = m$.

- Example (3.1) can be represented as $\phi \in \mathcal{NN}_{\mathbf{N},2}$ with dimensions $\mathbf{N} = (N_1 = 3, N_2 = 1)$. In this case, $\mathcal{X} = \{x_1, x_2, x_3\}$ and $\mathcal{Y} = a$. We attempt to train ϕ to 'learn' the classification function f for which $f(\mathcal{X}) = \mathcal{Y}$.
- Example (3.2) can be represented as $\phi \in \mathcal{NN}_{\mathbf{N},4}$ with dimensions $\mathbf{N} = (N_1 = 2, N_2 = 2, N_3 = 3, N_4 = 2)$.

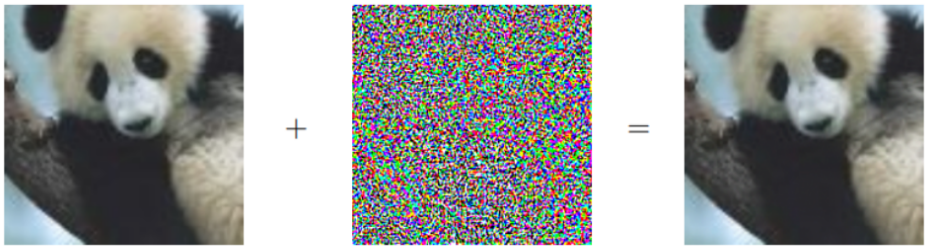


Figure 9: A hypothetical example of adversarial attack through image perturbation [3, p. 3].

With this definition of neural networks, each $\phi \in \mathcal{NN}_{\mathbf{N},L}$ is uniquely determined by its weights and biases.

Definition 4.3 (Set-theoretical definition of the cost function)**.** [2, p. 3] The *cost function* $\mathcal{R}$ of a neural network trained on $\mathcal{T} = (x^1, \ldots, x^r)$ is an element of the set

$$\mathcal{CF}_r \coloneqq \{\mathcal{R} : \mathbb{R}^r \times \mathbb{R}^r \to [0, \infty) \mid \mathcal{R}(v, w) = 0 \text{ if and only if } v = w\}. \quad (6)$$

In the context of Example (3.1), the cost of the neural network $\phi \in \mathcal{NN}_{\mathbf{N},L}$ may look like

$$\mathcal{R}\Big(\{\phi(x^j)\}_{j=1}^r, \{f(x^j)\}_{j=1}^r\Big) \in [0, \infty) \quad (7)$$

where we have r examples in our training dataset. Indeed, $\mathcal{R} = 0$ for $\phi = f$. So minimising $\mathcal{R}$ is simply a matter of finding a neural network $\phi \in \mathcal{NN}_{\mathbf{N},L}$ such that

$$\phi \in \underset{\varphi \in \mathcal{NN}_{\mathbf{N},L}}{\arg\min}\, \mathcal{R}\Big(\{\varphi(x^j)\}_{j=1}^r, \{f(x^j)\}_{j=1}^r\Big)$$

Definition 4.4. Consider a neural network $\phi \in \mathcal{NN}_{\mathbf{N},L}$ that is trained to learn a function f using the training data set $\mathcal{T} = \{x^1, \ldots, x^r\}$ and is tested against a validation data set $\mathcal{V} = \{y^1, \ldots, y^s\}$.
The neural network ϕ is said to be *accurate* if $\phi(y) \approx f(y)$ for a sufficient proportion[4] of $y \in \mathcal{V}$.
Moreover, the neural network ϕ is said to be *stable* if, for any perturbation Δx, we have that

$$\phi(x + \Delta x) = \phi(x).$$

[4]We leave this part of the definition intentionally ambiguous; it is up to the NN engineer to quantify their own version of a 'sufficient proportion'.

We shall now discuss a theorem taken from [2] that concerns simple classification functions of the form

$$f : [0,1]^d \to \{0,1\} \tag{8}$$

that could, for example, take an image of d pixels [5] as input and return 'cat' if $f(x) = 0$ and 'dog' if $f(x) = 1$. For the below theorem (which we shall not prove), we will assume that all images arise from a distribution $\mathcal{D}$ over $[0,1]^d$.

Theorem 4.1 (Stability of Neural Networks for simple classification functions)**.** *[2, pp. 4]*
Fix some instability parameter $\varepsilon > 0$. Then there exists a function f as defined in (8) with $d \geq 2$, a distribution $\mathcal{D}$ over $[0,1]^d$, a training set $\mathcal{T} = \{x^1, \dots, x^r\}$ and a validation set $\mathcal{V} = \{y^1, \dots, y^s\}$ such that the following occur simultaneously with high probability, provided $r + s$ is sufficiently large:

(i) For all cost functions $\mathcal{R} \in \mathcal{CF}_r$, there exists a neural network ϕ such that

$$\phi \in \underset{\varphi \in \mathcal{NN}_{\mathbf{N},L}}{\arg\min}\, \mathcal{R}\Big(\{\varphi(x^j)\}_{j=1}^r, \{f(x^j)\}_{j=1}^r\Big)$$

and $\phi(x) = f(x)$ for all $x \in \mathcal{T} \cup \mathcal{V}$.

(ii) For all $\hat{\phi} \in \mathcal{NN}_{\mathbf{N},L}$ (and, in particular, $\hat{\phi} = \phi$), there exists a dataset $\tilde{\mathcal{T}} \subset \mathcal{T} \cup \mathcal{V} \setminus \{\varnothing\}$ for which there exist uncountably many perturbations $\Delta x \in \mathbb{R}^d$ such that, for each $x \in \tilde{\mathcal{T}}$, we have that

$$|\hat{\phi}(x + \Delta x) - f(x + \Delta x)| \geq \frac{1}{2}$$

for $\|\Delta x\|_\infty \leq \varepsilon$.

Theorem 4.1 provides information regarding any simple classification function f with sufficiently large training and validation datasets $\mathcal{T}$ and $\mathcal{V}$. On the one hand, condition (i) of Theorem 4.1 tells us that we can train neural networks to learn f with great accuracy. In fact, there exists a neural network that learns the training data set and recognises the validation data set perfectly! On the other hand, condition (ii) tells us that the existence of an adversarial attack on a neural network trained on f (such that the output of the neural network is perturbed) is guaranteed. This lends itself to the idea of a trade-off between accuracy and stability for deep neural networks.

[5] The interval $[0, 255]$ for each pixel could be normalised by applying the sigmoid function to the colour value of each pixel to compensate for the restrictive input $[0, 1]$ of f.

5 Conclusion: One Potential Solution to Instability

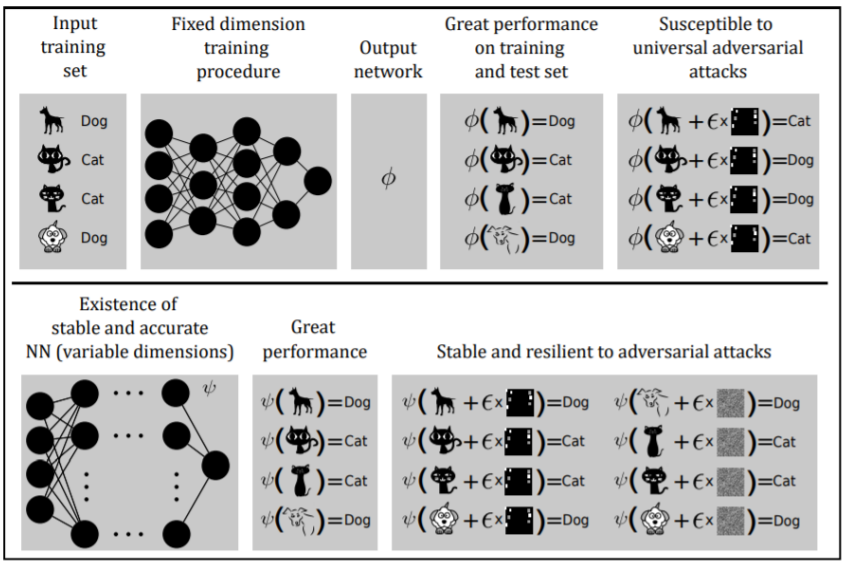


Figure 10: Comparison between a NN of fixed dimensions (which is susceptible to adversarial attack) and a NN with adaptive dimension, proficient in both accuracy and stability [2, p. 5]

Neural networks, however accurately they perform, will always be susceptible to adversarial attack when trained to learn simple classification functions. Is there anything to be done about this? Fortunately, contemporary research lends itself to a potential solution.
[1, p. 6] Thus far, we have concerned ourselves with neural networks $\phi \in \mathcal{NN}_{\mathbf{N},L}$ for fixed L. Is it possible to allow ϕ an adaptable layer number? It is suggested in [1, p. 6] that such a network could be constructed, at least in theory. Indeed, this would significantly limit the efficacy of adversarial attacks, because the perturbations needed to fool $\phi_1 \in NN_{\mathbf{N},L_1}$ would be vastly different to those needed to fool $\phi_2 \in \mathcal{NN}_{\mathbf{N},L_2}$, where $L_1 \neq L_2$. Figure 10 provides an illustrative comparison between some $\phi \in \mathcal{NN}_{\mathbf{N},L}$ and $\psi \in \mathcal{NN}_{\mathbf{N},\tilde{L}}$ for variable $\tilde{L} \in \mathbb{N}^+$. The mathematics of such ADNN's is well beyond the scope of this essay, but nonetheless provides food for thought regarding solutions to neural network instability.

References

[1] Joubine Aghili and Olga Mula. Depth-Adaptive Neural Networks from the Optimal Control viewpoint. *arXiv*, 2020.

[2] Alexander Bastounis, Anders C Hansen, and Verner Vlačić. The Mathematics of Adversarial Attacks in AI. *arXiv*, 2021.

[3] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv*, 2014.

[4] Catherine F Higham and Desmond J Higham. Deep Learning: An Introduction for Applied Mathematicians. *Siam review*, 61(4), 2018.

[5] Andrew Ng. Coursera: Machine Learning. `https://www.coursera.org/learn/machine-learning`, 2021.

[6] S.K. Panda, Mishra Vaibhav, R. Balamurali, and Ahmed A. Elngar. *Artificial Intelligence and Machine Learning in Business Management: Concepts, Challenges, and Case Studies*. CRC Press, 2021.

[7] Murray Smith. *Neural Networks for Statistical Modelling*. Cambridge University Press, 1993.