AN INTRODUCTION TO NEURAL NETWORKS AND DEEP LEARNING WITH APPLICATIONS IN NUMERICAL WEATHER PREDICTION AND TROPICAL CYCLONE FORECASTING

EVAN DAVID WELLMEYER

M.Sc. Atmospheric Science and Technology

Machine Learning

Professor Elio di Claudio

Professor Massimo Panella

Università Degli Studi Dell'Aquila, L'Aquila

Sapienza Università Di Roma, Rome

2023

Contents

1 Introduction

> A machine undoubtedly can be its own subject matter...These are possibilities of the near future, rather than Utopian dreams.
>
> *Alan Turing*

Traditional computational programming takes an input and applies an explicit rule defined by the programmer to generate an output. In contrast, machine learning works by applying an input and the desired output, and a machine learning model works to generate the rule that produces the desired output from the input. These learning models can be broken down into three different main types: Supervised learning, Unsupervised Learning and Semi-supervised learning.

Supervised machine learning is when an algorithm learns from example data (training data) when the associated target response to the input such as numeric values, string labels, etc., is included. There are two main problems that describe supervised learning: Classification and Regression. The classification problem comprises algorithms where the desired output is a categorical. The regression problem is that where the desired output is a real value.

Unsupervised learning is a class of machine learning techniques that aim to find patterns in data. The data given to an unsupervised algorithm are not labeled, which means only the input variables are given without corresponding output labels. Then the algorithm is left to discover structures in the data from the multi-spectral feature space. The two most common problems in unsupervised machine learning are clustering and Association. The clustering problem is where the algorithm works to discover inherent groupings in the data. The association problem is where the algorithm works to discover rules that describe large portions of data.

Reinforcement learning is another type of machine learning in which an agent learns to make decisions through trial and error, interacting with an environment to maximize a cumulative reward signal. The agent learns from experience by adjusting its actions based on positive or negative feedback from the environment, with the goal of eventually finding the optimal policy for achieving its objectives.

A neural network is a type of machine learning algorithm that is inspired by the

structure and function of biological neurons, such as those in the brain. It consists of interconnected nodes, or neurons, that process and transmit information through a series of weighted connections. Neural networks can be trained to learn patterns and relationships in data by adjusting the weights and biases of the network based on the error between the predicted and actual output.

Many variations of neural networks have been developed that modify the way information will pass from the input layer to the output layer. Some of the most common types of neural networks include convolutional neural networks (CNNs), which are commonly used for image processing or recognition, and recurrent neural networks (RNNs), which are designed for processing sequential data such as time series or language.

Neural networks have achieved state-of-the-art performance in many applications recently due to the exponential growth of data availability as well as increased computational power. The availability of these resources previously served as a huge limitation to possible applications of neural networks since training models can be computationally expensive and require large amounts of training data to achieve good performance. The development of new algorithms and architectures for "deep" neural networks have further expanded the capabilities of neural networks and enabled breakthroughs in a variety of subjects.

Deep learning is a subset of machine learning that uses artificial neural networks with many layers to analyze and learn from large and complex data sets. It involves training models by processing multiple layers of non-linear transformations of the input data, from which they are able to make accurate predictions, decisions or analyses. Deep learning algorithms use backpropagation to adjust the weights and biases of the neural network to minimize the difference between the predicted output and the actual output. Some common applications of deep learning these days include computer vision, language processing, image recognition/analysis/generation, and autonomous machines. The field of deep learning has advanced rapidly in recent years and continues to be applied to new areas.

The purpose of this paper is to give an introduction to the mathematical components commonly used in neural networks, explore some of the most common neural network architectures and understand their strengths, and finally, to explore some studies applying concepts of machine learning and neural networks to the fields of Numerical Weather

Prediction, precipitation forecasting, and tropical cyclone forecasting.

1.1 History

The history and development of machine learning, neural networks and subsequently deep learning can allegedly be traced back to the mathematical investigation of biological neuron function beginning in the 1940s. There is a bit of controversy around who is responsible for the first inspirations of such self learning algorithmic structures. However, it is generally accepted that the term "machine learning" was coined by Arthur Samuel in the 1950s. While working for IBM, he developed a number of algorithms for a computer to play the game of checkers that was able to measure the progress of the game and chances of each side winning based on positions and pieces on the board. He then began to implement a number of additional mechanisms that would allow the program to record previous positions and outcomes using a reward function thus allowing it to predict the next best move.

The perceptron was first introduced in 1957 by Frank Rosenblatt, combining the work of Arthur Samuel on machine learning with a model of brain cell interaction created by Donald Hebb in 1949. The use of multiple layer perceptrons was introduced in the 1960s when it was discovered that the use of two or more layers significantly increased processing power compared to the use of one layer perceptrons. The implementation of multi-layer perceptrons lead to the development of feedforward neural networks. However, early neural networks were extremely limited. The combined lack of computational resources and absence of large datasets limited the imagination of what would one day be possible.

One big breakthrough in the development of neural networks came in the 1970s with the invention of backpropagation, a learning algorithm that allows neural networks to learn from data by adjusting the weights and biases of the network based on the error between the predicted and actual output. This made it possible to train neural networks with multiple layers, also known as deep neural networks, which were previously difficult to train effectively.

In the 1980s and 1990s, the development of convolutional neural networks (CNNs) and recurrent neural networks (RNNs) further improved the performance of deep learning models. The field of machine learning appeared to have plateaued in the late 1990s and early 2000s due to the limited availability of large data sets and computational

resources needed to train deep neural networks. However, advances in computing power, the availability of large data sets, and the development of new algorithms for training deep neural networks, such as dropout regularization and batch normalization, all fueled the resurgence of deep learning in the 2010s.

In recent years, the development of new architectures and techniques such as generative adversarial networks (GANs) and transformers have further expanded the capabilities of deep learning models, enabling breakthroughs in many areas. Today, deep learning is a rapidly evolving field with significant potential for further advancements in the coming years.

1943	Logician Walter Pitts and neuroscientist Warren McCulloch publish a paper attempting to mathematically map out thought processes and decision making in human cognition.
1949	Donald Hebb models brain cell interaction in his book "The Organization of Behavior".
1950	Alan Turing publishes his paper introducing the famous Turing Test.
1952	Arthur Samuel develops a number of "learning" algorithms for a computer to play the game of checkers.
1957	Frank Rosenblatt introduces the perceptron, combining the work of Arthur Samuel on machine learning with a model of brain cell interaction created by Donald Hebb.
1960	Multiple layer perceptrons introduced.
1967	Nearest neighbor algorithm introduced.
1970	Finnish mathematician Seppo Linnainmaa publishes the general method for automatic differentiation of discrete connected networks of nested differentiable functions as his masters thesis, the method corresponding to the modern version of backpropagation.
1979	Kunihiko Fukushima first publishes work on a type of Artificial Neural Networks (ANN) he calls the neocognitron, which later inspire Convolutional Neural Networks (CNN). The same year autonomous obstacle navigation introduced with the "Stanford cart".
1981	Gerald Dejong introduced the concept of Explanation Based Learning (EBL).
1982	John Hopfield creates Hopfield networks, a type of Recurrent Neural Network (RNN).
1986	Rumelhart, Hinton and Williams demonstrate the modern scope of backpropagations application.
1989	Christopher Watkins develops Q-learning, which greatly improves the practicality and feasibility of reinforcement learning.
1995	Tim Kam Ho first publishes a paper describing random decision forests. Corinna Cortes and Vladimir Vapnik publish work on support-vector machines.
1997	Sepp Hochreiter and Jurgen Schmidhuber invent Long Short-Term Memory (LSTM) RNNs. The same year IBM's Deep Blue beat the world champion at chess.
2006	Geoffrey Hinton coins the term "deep learning".
2017	A team working at Google brain invent the Transformer architecture and the attention mechanism.

Table 1.1.1: Timeline of a few interesting events in the history of machine learning.

2 Fundamentals of Deep Learning

A neural network is composed of interconnected nodes or "neurons". A group of nodes comprise a layer. The number of nodes in a layer is the density of the layer, and the number of layers is the depth of the network. There are two main categories of neural

network architectures that depend on the type of interconnections that exist between nodes, these are Feed-Forward Neural Networks and Recurrent Neural Networks. While the differences between these two types of networks can have significant impacts (which will be discussed later on), they share many of the same components. Beyond the simple composition of weights and nodes, all neural networks use activation functions to achieve arbitrary learning capabilities, loss functions to quantify accuracy, most use gradient descent as an instruction on how to learn, backpropagation to distribute these learning instructions through the network and many techniques to improve the learning capabilities and improve robustness of a model. In the following section these components in their various forms will be explained in detail.

2.1 Activation Functions

Activation functions are the key to allowing neural networks take on arbitrary problem solving capabilities. These are non-linear functions that are applied to the output of each neuron in a neural network to introduce non-linearity into the model. It transforms the input signal into an output signal that is then passed on to the next layer of the network. Activation functions serve two main purposes in neural networks. The first is that they introduce non-linearity into the model, which is essential for the network to learn complex patterns and relationships in the data. Without non-linearity, a neural network would essentially be a linear model that can only learn linear relationships. The second purpose of activation functions is to help ensure that the output of each neuron is within a desired range, preventing the output from growing too large or too small.

There are several types of activation functions that are commonly used in neural networks. The following subsections will discuss a wide variety of activation functions and their extensive properties.

2.1.1 Sigmoid

The sigmoid function, also known as the logistic or soft step, maps any input to a value between 0 and 1, which can be useful for models that require a probabilistic interpretation of the output. The shape of the sigmoid function causes small changes in the input near 0 result in large changes in the output near the extremes of 0 and 1. This property makes the sigmoid function useful in classification tasks, where the output is interpreted as a

probability of belonging to a particular class.

$$\Phi(z) = \frac{1}{1 + e^{-z}} \tag{2.1}$$

$$\Phi'(z) = \Phi(z)(1 - \Phi(z)) \tag{2.2}$$

The sigmoid function is defined mathematically in Equation 2.1, and its derivative in Equation 2.2. These functions are graphed in Figure 2.1.1.

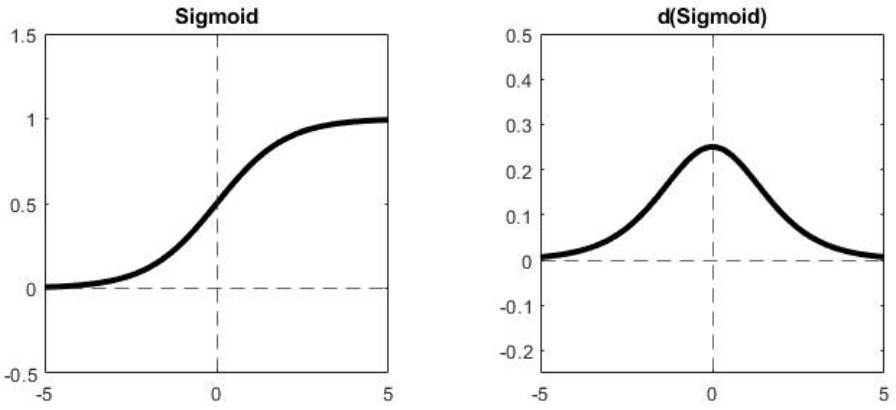


Figure 2.1.1: Graph of the Sigmoid activation function (left) and its derivative (right).

The sigmoid function can suffer from the vanishing gradient problem, where the gradients become very small as the input moves away from 0. This can make it difficult to train deep neural networks that use sigmoid activation functions, as the gradients can become too small to propagate through the network effectively. As a result, other activation functions such as ReLU and variants of ReLU are often used in modern neural network architectures.

2.1.2 Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) function outputs the input directly if it is positive, and outputs zero if it is negative. ReLU has become one of the most widely used activation functions because it is computationally very efficient and tends to work well in practice. Another advantage is that it helps to address the vanishing gradient problem, which can occur when training deep neural networks with other activation functions, such as sigmoid or tanh.

$$\Phi(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{2.3}$$

$$\Phi'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases} \tag{2.4}$$

The ReLU function is defined mathematically in Equation 2.3, and its derivative in Equation 2.4. These functions are graphed in Figure 2.1.2.

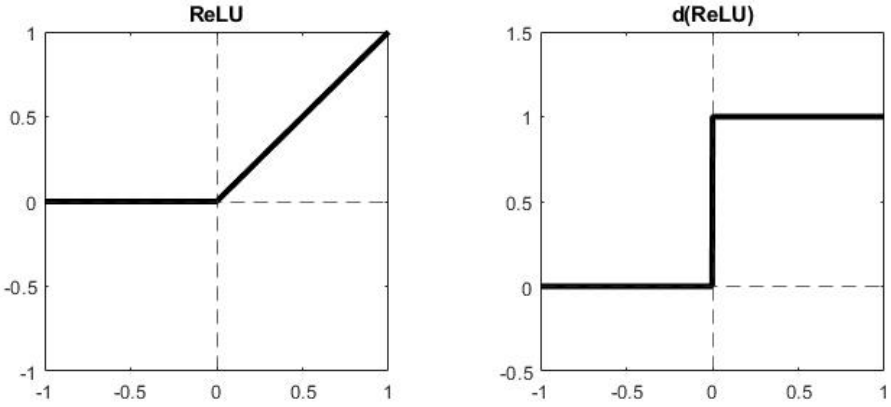


Figure 2.1.2: Graph of the Rectified Linear Unit (ReLU) activation function (left) and its derivative (right).

A potential disadvantage of ReLU is that it can result in "dead" neurons that output zero for all inputs. This can occur if the neuron's weights are such that its input is always negative, and therefore the ReLU output is always zero. To address this issue, several variations of ReLU have been proposed, such as leaky ReLU, parametric ReLU, and exponential ReLU.

2.1.3 Hyperbolic Tangent (Tanh)

The tanh function maps any input to a value between -1 and 1, which can be useful for models that require a symmetric output.

$$\Phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

$$\Phi'(z) = 1 - \Phi(z)^2 \tag{2.6}$$

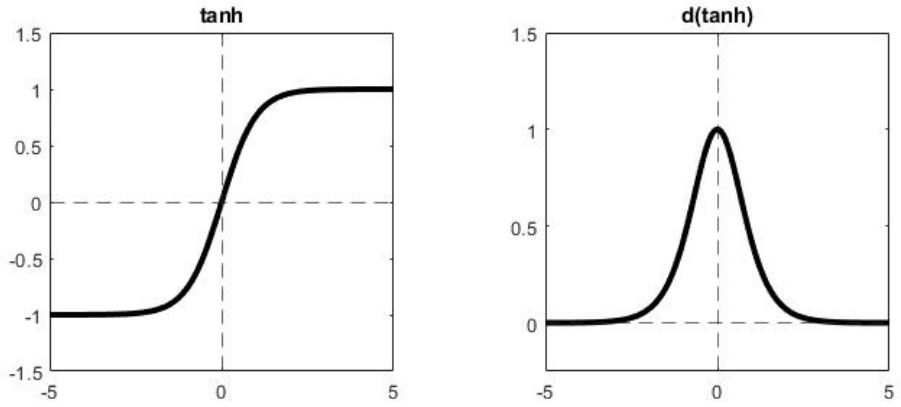


Figure 2.1.3: Graph of the hyperbolic tangent (tanh) activation function (left) and its derivative (right).

One advantage of the tanh function is that it is a smoother and shifted version of the sigmoid function. Like the sigmoid function, the tanh function is also differentiable and has a well-defined derivative, which is important for gradient-based optimization algorithms used in training deep neural networks. However, the tanh function can potentially also suffer from the vanishing gradient problem, due to the derivative of tanh approaching zero as its input becomes very large or very small. This leads the gradients to become very small during backpropagation, slowing down the learning process.

2.1.4 Softmax

Softmax: A softmax function is commonly used in the output layer of a neural network for classification tasks. It normalizes the output of the network so that the sum of the outputs equals 1, which can be interpreted as probabilities for each class.

$$\Phi(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{2.7}$$

The choice of activation function can have a significant impact on the performance of a neural network, and different activation functions may be more suitable for different types of models or tasks.

2.2 Loss Functions

Loss (or cost) functions tell the neural network the error between the target output y and the models prediction $\hat{y}$. There are many loss functions, and the optimal loss function depends on the type of problem the neural network is aiming to solve, whether it is regression or classification. The most common loss function in regression problems is the Mean Square Error loss (MSE), which is the average of the squared differences between the target and the prediction:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2 \tag{2.8}$$

The summation form here is relevant to batch (or mini-batch) gradient descent methods with n training examples, which will be discussed further later on. The MSE loss is able to penalize large errors by squaring them; however, this can cause problems if the data is prone to outliers.

The second most common regression loss function is Mean Absolute Error (MAE), which takes the average of the absolute differences between the target and prediction:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{n}\sum_{i=1}^{n}|y - \hat{y}| \tag{2.9}$$

Once again the summation form allows the function to hold under multiple gradient descent methods. The removal of the squared difference in this loss function allows it to be more robust when data is prone to outliers.

In classification problems, the most commonly used loss function is the cross-entropy loss. If the classification is binary, the *binary* cross-entropy loss has the form:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \tag{2.10}$$

In the case of a multi-class problem, the cross-entropy loss is of the form:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) \tag{2.11}$$

The second most common loss function used in classification problems is the Hinge loss. The hinge loss works by penalizing both incorrect predictions as well as correct predictions

with low confidence. For an target output $y \in \{-1, 1\}$ and prediction $\hat{y}$, the hinge loss is:

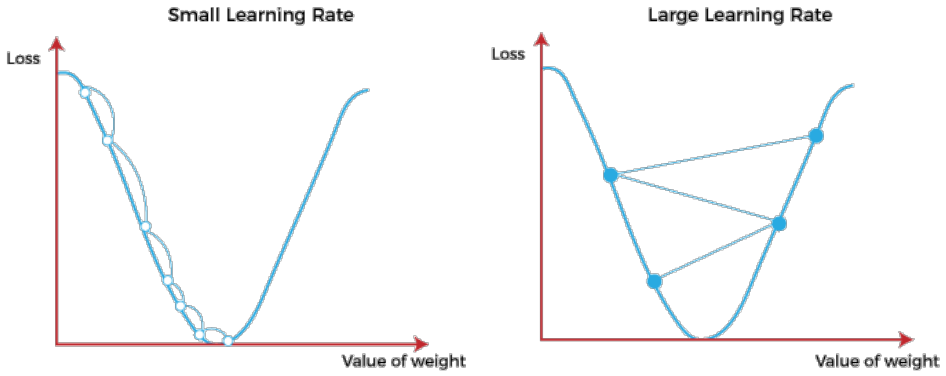
$$\mathcal{L} = \max\{0, 1 - y \cdot \hat{y}\} \tag{2.12}$$

Hinge loss is commonly used when implementing a learning method called support vector machine.

2.3 Gradient Descent

Gradient descent is the underlying process by which a neural network "learns" to produce the desired result from an arbitrary input. The goal of gradient descent is to minimize the cost (or loss) function. The generic process of gradient descent is as follows: A neural network is initialized (usually randomly), takes and input and generates the output. The loss function calculates the loss by comparing the models output with the desired output. By differentiating the loss function we can calculate the slope of the loss for the parameters that produced the output, allowing us to determine in which direction (positive or negative) the parameters can be adjusted that will result in a smaller magnitude of loss in a subsequent iteration. Once the direction in which the parameter updates should take place, the size of the adjustment to the parameters is usually determined by first the magnitude of the calculated gradient, and second the learning rate which scales the gradient to perform either larger or smaller parameter updates with each iteration. Figure 2.3.1 shows two plots of loss vs. weight value over iterations using both a small learning rate and a comparatively large learning rate. In this situation using a small learning rate can allow the networks solution to descend smoothly to a minimum in the cost function, whereas a larger learning rate, while potentially reducing the required number of iterations to reach the minimum, can result in the network oscillating around the optimal solution.

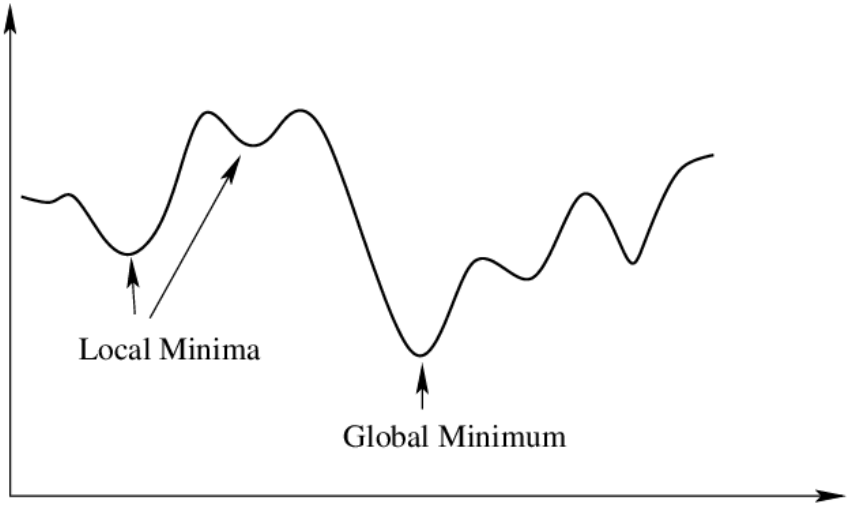
However, in a more realistic situation many minima can exist in the loss vs weight value landscape, which can complicate the choice of a learning rate. Figure 2.3.2 shows a plot of loss vs weight value in which multiple minima exist. In this situation it is easy to see how a small learning rate is subject to becoming "stuck" in a local minimum, whereas a larger learning rate could have the potential to escape a local minimum. Many optimization techniques have been developed to resolve this issue, many of which will be discussed



(https://www.javatpoint.com/gradient-descent-in-machine-learning)

Figure 2.3.1: Behavior of learning rate in a one dimensional representation of Loss vs Weight value.

later on.



(https://inverseai.com/blog/gradient-descent-in-machine-learning)

Figure 2.3.2: Representation of local and global minima in a loss landscape.

There are three main types of gradient descent that can be used in the training of a neural network: Batch gradient descent, Stochastic gradient descent and Mini-batch gradient descent, which will be explored in detail in a later section. The process of backpropagation is identical for the three cases, and they differ only in the way training data is handled during the training process.

2.4 Backward Propagation

Forward propagation and backward propagation are the two main phases of the training process in a neural network. The forward propagation phase takes an input and processes

it through a series of layers to produce an output. Each layer of the network consists of a set of neurons, and each neuron in a layer is connected to neurons in the previous layer by a set of weights. The input to the first layer is the raw input data, which is processed by each neuron in the layer to produce a set of output values. These output values are then passed on to the next layer as input, where they are processed in the same way, and so on, until the final layer of the network produces an output.

During backward propagation, or "backpropagation", the neural network calculates the gradient of a loss function with respect to the weights of the network. The loss function is a measure of networks output with respect to that of the desired output. The goal of training is to minimize the loss function by adjusting the weights of the network. Backward propagation calculates the derivative of the loss function with respect to each weight in the network using the chain rule, which involves calculating first the derivative of the loss function with respect to the output of the final layer, and then propagating this derivative backwards through the layers of the network to calculate the derivative of the loss function with respect to the weights in each layer. Once the derivative of the loss function with respect to the weights has been calculated, the weights can be updated.

To illustrate the generic process of backpropagation (through gradient descent), lets assume we have a simple neural network with one input, one hidden layer and one output. This network consists of the input value, x, the weights W_1 connecting the input to the hidden layer h with bias term b_1, the activation function for the hidden layer Φ and the weights W_2 with the second bias term b_2 connecting the hidden layer to the output y. The feedforward algorithm for this simple network is as follows:

$$\begin{aligned} Z_1 &= W_1 x + b_1 \\ h &= \Phi(Z_1) \\ Z_2 &= W_2 h + b_2 \\ y &= Z_2 \end{aligned} \tag{2.13}$$

The first step in backpropagation is to calculate the loss. The loss $\mathcal{L}$ is calculated from the network output y and the desired output y' with a cost function:

$$\mathcal{L} = (y - y')^2 \tag{2.14}$$

Here we are using the single valued mean squared error loss (MSE) version from Equation 2.8. The next step is to compute the gradient of the cost function with respect to the weights and biases. Using the chain rule, the the gradient is broken down into smaller gradients for each layer of the network. Starting from the output layer, the gradient of the cost function with respect to the output is:

$$\nabla_y \mathcal{L} = \frac{\partial \mathcal{L}}{\partial y} = 2(y - y') \tag{2.15}$$

The gradient of the cost function with respect to the weights and biases for the output layer is then:

$$\begin{aligned} \nabla_{W_2} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial Z_2} \frac{\partial Z_2}{\partial W_2} = 2(y - y')h \\ \nabla_{b_2} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial Z_2} \frac{\partial Z_2}{\partial b_2} = 2(y - y') \end{aligned} \tag{2.16}$$

The gradient of the cost function with respect to the activation function of the hidden layer is:

$$\frac{\partial h}{\partial Z_1} = \Phi'(Z_1) \tag{2.17}$$

The gradient of the cost function with respect to the weights and biases of the hidden layer is:

$$\begin{aligned} \nabla_{W_1} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial Z_2} \frac{\partial Z_2}{\partial h} \frac{\partial h}{\partial Z_1} \frac{\partial Z_1}{\partial W_1} = 2(y - y') \cdot W_2 \cdot \Phi'(Z_1) \cdot x \\ \nabla_{b_1} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial Z_2} \frac{\partial Z_2}{\partial h} \frac{\partial h}{\partial Z_1} \frac{\partial Z_1}{\partial b_1} = 2(y - y') \cdot W_2 \cdot \Phi'(Z_1) \end{aligned} \tag{2.18}$$

Now that the gradients have been calculated, the weights and biases can be updated according to the learning rate α:

$$
\begin{aligned}
W_1(t+1) &= W_1(t) - \alpha \nabla_{W_1} \mathcal{L}(t) \\
b_1(t+1) &= b_1(t) - \alpha \nabla_{b_1} \mathcal{L}(t) \\
W_2(t+1) &= W_2(t) - \alpha \nabla_{W_2} \mathcal{L}(t) \\
b_2(t+1) &= b_2(t) - \alpha \nabla_{b_2} \mathcal{L}(t)
\end{aligned}
\tag{2.19}
$$

Now that the weights have been updated, the process restarts on the next iteration, with each consecutive iteration aiming to reduce the loss.

In the previous example we have use the post-activation value of the output in order to calculate the gradient back propagation. However, in some instances it may be preferred to use the pre-activation value to perform the gradient backpropagation. Expanding both instances of this method to a generic network of arbitrary size. For a sequence of hidden layers $h_1, ...h_k$, output y, weights from h_r to h_{r+1} as $W_{(h_r, h_{r+1})}$. For the number of possible paths from h_r to y as $\mathcal{P}$, the gradient of the cost function with respect to the weights for post-activation variables is as follows:

$$
\frac{\partial \mathcal{L}}{\partial W_{(h_{r-1}, h_r)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial y} \cdot \left[\sum_{[h_r, h_{r+1}, \ldots, h_k, y] \in \mathcal{P}} \frac{\partial y}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\text{Backpropagation computes } \Delta(h_r, y) = \frac{\partial \mathcal{L}}{\partial h_r}} \cdot \frac{\partial h_r}{\partial W_{(h_{r-1}, h_r)}} \tag{2.20}
$$

The gradient with respect to the weights for pre-activation variables is then:

$$
\frac{\partial \mathcal{L}}{\partial W_{(h_{r-1}, h_r)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial y} \cdot \Phi'(Z_y) \cdot \left[\sum_{[h_r, h_{r+1}, \ldots, h_k, y] \in \mathcal{P}} \frac{\partial Z_y}{\partial Z_{h_k}} \prod_{i=r}^{k-1} \frac{\partial Z_{h_{i+1}}}{\partial Z_{h_i}} \right]}_{\text{Backpropagation computes } \delta(h_r, y) = \frac{\partial \mathcal{L}}{\partial Z_{h_r}}} \cdot h_{r-1} \tag{2.21}
$$

The process of forward propagation and backward propagation is repeated iteratively during the training process, until the network has learned to produce accurate outputs for the given input data.

2.5 Optimization Techniques

Gradient descent can be implemented using a variety of methods. The three methods following use approximately the same formulation for updating parameters, where the adjustment to a matrix of weights ΔW_{ij} is calculated based on the magnitude of the gradient and the learning rate. In order to simplify the following section, let θ be a generic parameter to be updated and $g = \nabla_\theta \mathcal{L}$ as the gradient of the loss function with respect to that parameter so that the updating equations 2.19 can be rewritten:

$$\theta_{t+1} = \theta_t - \alpha g_t \tag{2.22}$$

In batch gradient descent, the entire training dataset is run through the model simultaneously at each iteration. The model parameters are then updated based on the average of the gradients computed for the entire training dataset. For a dataset of n training examples, the batch parameter update is as follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{n} \sum_{m=1}^{n} g_{m,t} \tag{2.23}$$

Propagating and computing gradients for the entire dataset at every iteration makes batch gradient descent computationally expensive; however, it has a lower variance and noise compared to other gradient descent algorithms making it an ideal option for smaller datasets.

In stochastic gradient descent (SGD), the model runs one training example at a time (chosen at random), then calculates the gradients and updates the parameters. The parameter update per iteration is then:

$$\theta_{t+1} = \theta_t - \alpha g_{m,t} \tag{2.24}$$

with $m \in \{1, ..., n\}$ as the randomly selected training example. SGD is computationally much less expensive due to the massive reduction in parallel calculations, but the process can be noisy and oscillate around the optimal solution rather than converging.

Mini-batch gradient descent is a compromise between batch and stochastic gradient descent. In this method the training data is divided into smaller subsets or "mini-batches", which are typically organized randomly to ensure the algorithm is not biased

to any specific subset. A mini-batch is selected at each iteration to run through the model and update the parameters in the same way as in batch gradient descent. For a mini-batch $p \subseteq n$ of k training examples, the parameter update is:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{k} \sum_{[m_1,\ldots,m_k] \in p} g_{m,t} \tag{2.25}$$

This approach can converge faster than batch gradient descent and is less noisy than SGD.

2.5.1 First-order Optimization

There are also several optimization techniques that can be used to improve the performance of gradient descent methods. These techniques fall under two categories: First-order optimization and second-order optimization. First-order optimization techniques use first-order differentials to minimize the cost function. First-order techniques include Momentum, Nesterov, Adagrad, Adadelta, RProp, RMSProp, Adam, Adamax, Nadam and AMSGrad are some of the most popular optimization techniques used in deep learning. It should be noted that while writing this section, a number of sources interchange some of the names of certain optimization techniques and the corresponding algorithms. What is is reported here is therefore based on what is reported at a higher frequency as well as what seemed more appropriate considering development timeline and respective algorithm naming.

Momentum is a popular optimization technique utilized with SGD that adds an additional "momentum" term to the weight updating equations based on the accumulated gradients from previous iterations. With the addition of the momentum term, Equation 2.22 becomes:

$$\theta_{t+1} = \theta_t - \alpha g_t + \mu g_{t-1} \tag{2.26}$$

where μ is a hyper-parameter that scales contribution from past gradient adjustments.

Nesterov Accelerated Gradient (NAG) is a optimization technique closely related to the Momentum method. NAG adds a interim parameter ϕ to represent the classical gradient descent update ($\theta_{t+1} \rightarrow \phi_{t+1}$) so that Equation 2.22 becomes:

$$\begin{aligned} \phi_{t+1} &= \theta_t - \alpha g_t \\ \theta_{t+1} &= \phi_{t+1} + \mu(\phi_{t+1} - \phi_t) \end{aligned} \tag{2.27}$$

This method can be tedious in terms of finding the optimal values of the learning rate α and the momentum μ. However, correct implementation of the method can result in reduced oscillation about a minimum. There are many versions of this method that will not be considered here.

Adagrad (Adaptive Gradient Algorithm) is an optimization technique that adapts the learning rate to the parameters. It performs larger updates for infrequent parameters and smaller updates for frequent parameters making it well-suited for sparse data. This is done by modifying the general learning rate α at each iteration t for every parameter θ based on the past gradients that have been computed for θ:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \varepsilon}} g_t \tag{2.28}$$

where ε is a quantity to avoid division by zero and G is the sum of the squares of the gradients with respect to θ:

$$G_t = \sum_{i=0}^{t} g_i^2 \tag{2.29}$$

This method allows more frequently updated features to achieve a decaying learning rate and sparse features to maintain a higher learning rate.

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Adadelta instead uses an average of past squared gradients to update the learning rate, however in this case the adaptation is based on a moving window of gradients instead of the accumulation of all past gradients. In addition to this, Adadelta implements an "adaptive delta" term D. For a moving window of length w, the adaptive delta term and Equation 2.29 are:

$$\begin{aligned} D_t &= \gamma D_{t-1} + (1-\gamma)(\alpha g_t)^2 \\ G_t &= \gamma G_{t-1} + (1-\gamma) g_t^2 \end{aligned} \tag{2.30}$$

where γ is a decay constant similar to that used in the Momentum method. Using the calculations from Equation 2.30, the parameter update for the Adadelta method is then:

$$\theta_{t+1} = \theta_t - \frac{\sqrt{D_{t-1} + \varepsilon}}{\sqrt{G_t + \varepsilon}} \cdot g_t \tag{2.31}$$

This windowed accumulation method keeps the accumulation from becoming too large, ensuring learning continues to make progress after many iterations.

Resilient Back Propagation (RProp) is an optimization method for batch gradient descent that attempts to resolve the problem that gradients (in the batch) may vary widely in magnitudes, making it difficult to determine a global learning rate for the algorithm. In order to combat this problem, the RProp method takes into account only the sign of the partial derivative. This reduction in computational load also makes it one of the fastest weight updating algorithms.

RMSprop (Root Mean Square Propagation) is an optimization technique that is very similar to the Adadelta method. Instead of the using the delta term D in the numerator, RMSprop uses only the learning rate α so the RMSprop parameter update equation is the same as in Equation 2.28; however with G as is defined in Equation 2.30.

Adam (Adaptive Moment Estimation) is an optimization technique that computes adaptive learning rates for each parameter. It combines the advantages of both AdaGrad and RMSProp optimization techniques using the exponential moving average of gradients m and the adaptive learning rate via the exponential moving average of squared gradients. We define m and G as:

$$\begin{aligned} m_t &= \gamma_1 m_{t-1} + (1 - \gamma_1) g_t \\ G_t &= \gamma_2 G_{t-1} + (1 - \gamma_2) g_t^2 \end{aligned} \tag{2.32}$$

we then compute the interim parameters $\hat{m}$ and $\hat{G}$:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \gamma_1^t} \\ \hat{G}_t &= \frac{G_t}{1 - \gamma_2^t} \end{aligned} \tag{2.33}$$

the parameter update equation for Adam optimization can then be defined:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{G}_t} + \varepsilon} \cdot \hat{m}_t \tag{2.34}$$

Adam is proposed as the most computationally efficient stochastic optimization method and has been shown to work well for a wide range of deep learning models (generalized performance).

2.5.2 Second-order Optimization

Second-order optimization is a class of optimization algorithms that uses the second derivative of the loss function with respect to the weights of the neural network to update the weights during training. These algorithms are also known as Newton-based methods. The idea behind second-order optimization is to use the curvature information of the loss function to make smarter updates to the weights. Specifically, the second derivative of the loss function tells us how the gradient changes as we move away from the current weight position. By taking into account this curvature information, second-order optimization algorithms can converge faster and more accurately than first-order methods. A popular second-order optimization algorithm is called Newton's method, which updates the weights as follows:

$$\theta_{t+1} = \theta_t - H_t^{-1} g \tag{2.35}$$

where H is the Hessian matrix, which is the matrix of second derivatives of the loss function with respect to the weights. The inverse of the Hessian matrix is known as the curvature matrix, and it describes the curvature of the loss function. Computing the Hessian matrix is extremely expensive computationally for large neural networks, so other second-order optimization algorithms like the Hessian-free optimization and the Conjugate Gradient method have been developed to approximate the Hessian matrix more efficiently.

2.6 Regularization Techniques

Neural network regularization techniques aim to prevent the overfitting of a model to the training data. Overfitting occurs when a model becomes too complex and begins to incorporate noise and irrelevant features in the data to the solution, leading to poor

generalization performance on new data. Ways of implementing regularization include modifying the loss function, data augmentation and training algorithm modification. Methods modifying the loss function include L1 and L2 regularization and Entropy regularization. Methods modifying the sampling method with K-fold Cross-Validation and data augmentation. Methods involving modification of the training algorithm include dropout and injecting noise. Here we will discuss only L1 and L2 and dropout, as they seem to be the most common methods. In L2 regularization, the method involves using a penalty term in the loss function that encourages the model to retain smaller weights:

$$\mathcal{L}_{L2} = \mathcal{L} + \frac{\lambda}{2} \sum_{i=1}^{n} W_i^2 \tag{2.36}$$

Where $\mathcal{L}$ is the original loss function and λ as the parameter controlling the strength of the regularization. When we add the L2 regularization penalty term to the cost function, we also take the derivative of that term with respect to each weight. This derivative is proportional to the weight itself, so when we adjust the weight to reduce the cost, we also adjust it to be smaller due to the L2 regularization penalty.

The L1 regularization method is very similar to that of the L2 method; however, in the case of L1 the absolute values of the weights are penalized:

$$\mathcal{L}_{L1} = \mathcal{L} + \frac{\lambda}{2} \sum_{i=1}^{n} |W_i| \tag{2.37}$$

The dropout technique randomly sets some node activations to zero during the training, forcing the network to learn more robust features that are not dependent on the presence of specific activations. This is done by randomly selecting a set of neurons to "drop out" with probability p. The output of the remaining neurons is then scaled by a factor of $1/(1-p)$ to maintain the expected value of the output. Figure 2.6.1 shows a simple neural network with dropout probability $p = 0.5$.

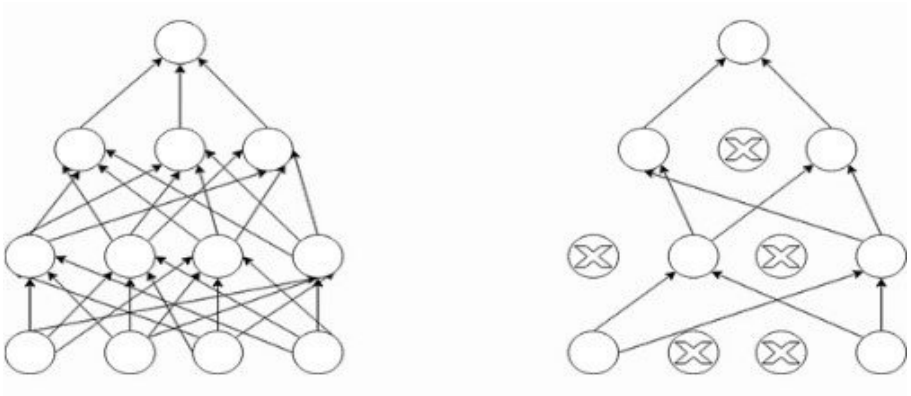


Figure 2.6.1: Neural network with 50% dropout. Source: Rahman et al.,2018.

Physically, dropout can be thought of as a form of ensemble learning, where we are training multiple sub-networks simultaneously. Each sub-network is trained to perform well on a subset of the training data, and by combining the predictions of all the sub-networks, we get a more robust and accurate model.

3 Deep Learning Algorithms

The basic building block of a neural network is the perceptron, which takes in input values, multiplies them by corresponding weights, sums the results, and applies an activation function to produce an output. From the two main categories of network architectures presented earlier—feed-forward and recurrent, the latter of which will be discussed later—this concept is more in line with the former category. Most types of neural networks fall into the category of feed-forward networks. To begin, lets first consider a single layer perceptron that can be represented mathematically in the form

$$\hat{y} = \Phi\{\overline{W} \cdot \overline{X} + b\} = \Phi\left\{\sum_{j=1}^{d} w_j x_j + b\right\} \tag{3.1}$$

where $\overline{X} = \{x_1, x_2, ..., x_d\}$ represents the input; $\overline{W} = \{w_1, w_2, ..., w_d\}$ are the weights with $d \in \mathbb{N}_1$ as the density and b as the bias term; $\Phi\{\}$ represents the activation function and $\hat{y}$ is the output. Perceptrons are commonly represented in the form of diagrams (computational graphs). For example, the above algebraic formula can be represented as shown in Figure 3.0.1.

The computational graph allows easy visualization of the processes involved in an architecture. To further illustrate this with respect to both the computational graph as well

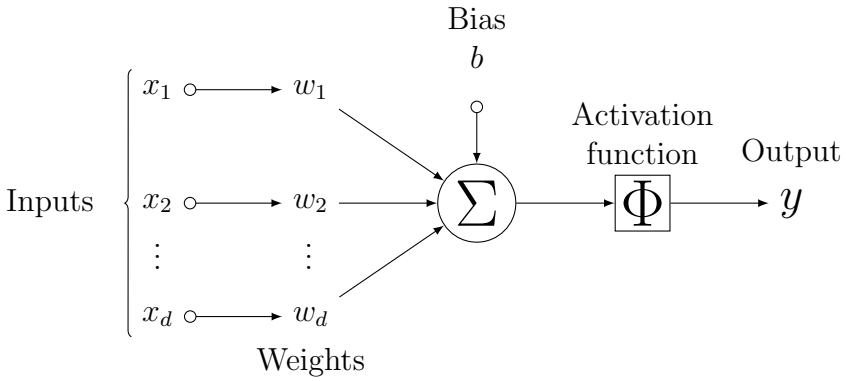


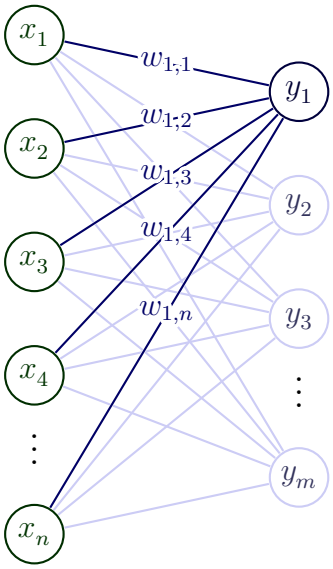
Figure 3.0.1: Diagram of a simple one layer perceptron with bias term and without hidden layers.

as the linear transformation, Figure 3.0.2 shows the computation from an input layer x to the output layer y in a single layer fully-connected neural network.

A multi-layer fully connected perceptron in a feedforward neural network consists of multiple layers of perceptrons process the input data through a series of non-linear transformations to produce the final output. Representing these algebraically then requires a series of equations to describe the process. For a neural network with k hidden layers h of arbitrary density, we have:

$$\begin{aligned} \overline{h_1} &= \Phi(W_1^\top \overline{X} + b_1) \\ \overline{h}_{p+1} &= \Phi(W_{p+1}^\top \overline{h}_p + b_p) \quad \forall p \in \{1, ..., k-1\} \\ \overline{y} &= \Phi(W_{k+1}^\top \overline{h}_k + b_k) \end{aligned} \tag{3.2}$$

where the input X has propagated though the hidden layers to produce the output y. In the following section, a number of different deep learning algorithms will be considered.



$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \Phi \left[\begin{pmatrix} w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \right]$$

$$y^{(1)} = \Phi\left(\mathbf{W}x + \mathbf{b}\right)$$

Figure 3.0.2: Single processing unit and its components. The activation function is denoted by Φ and applied on the actual input x of the unit to form its output $y = \Phi(x)$. $x_1, \dots, x_n$ represent input from other units within the network.

3.1 Bayesian Neural Networks

A Bayesian Neural Network (BNN) combines the architecture of a fully connected feed-forward neural network with Bayesian inference. In standard neural networks (SNN), the weights and biases are fixed values that are adjusted through optimization techniques. In the case of a BNN, the fixed weights and biases are replaced by probability distributions. This modification allows the uncertainty introduced by the model to be quantified; it addresses the overfitting problem and enables the network to learn from small datasets.

Overfitting occurs in SNNs due their high flexibility in adjusting parameters to fit data. BNNs address this problem by modeling uncertainty into the weights of the network. For example, in SNNs we say the training data $\mathcal{D}$ is used to minimize the loss function $\mathcal{L}$ by adjusting the model's weights w through a gradient-based optimizer such as gradient descent. The objective of Bayesian inference is to compute the conditional probability of the weights given the data such as the posterior distribution $p(w|\mathcal{D})$. This is achieved by applying Bayes' theorem:

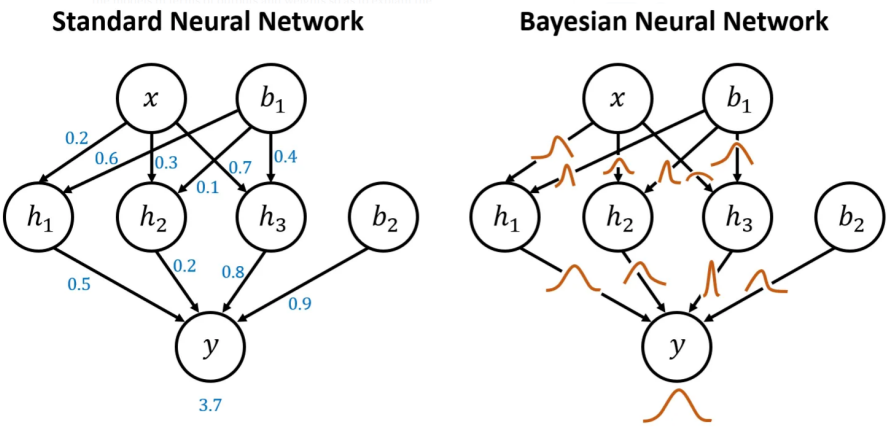
$$p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)p(w)}{p(\mathcal{D})} \tag{3.3}$$

where $p(\mathcal{D}|w)$ is the likelihood of the data given the weights, $p(w)$ is the prior distribution over the weights, and $p(\mathcal{D})$ is the marginal likelihood of the data. By computing the

posterior distribution, we can then obtain a probabilistic prediction for a new given input x by computing the predictive distribution:

$$p(\hat{y}(x)|\mathcal{D}) = \int_w p(\hat{y}(x)|w)p(w|\mathcal{D})dw = \mathbb{E}_{p(w|\mathcal{D})}[p(\hat{y}(x)|w)] \tag{3.4}$$

where $p(\hat{y}(x)|w)$ is the likelihood of the output given the weights, and $\mathbb{E}_{p(w|\mathcal{D})}$ is the expected value of the likelihood under the posterior distribution. The predictive distribution is obtained by averaging the likelihood over all possible weight configurations weighted by their posterior probability. The ability of BNNs to capture the uncertainty in the model's parameters with predictive distribution allows them to provide a more robust estimate of a models output compared to a traditional neural network.



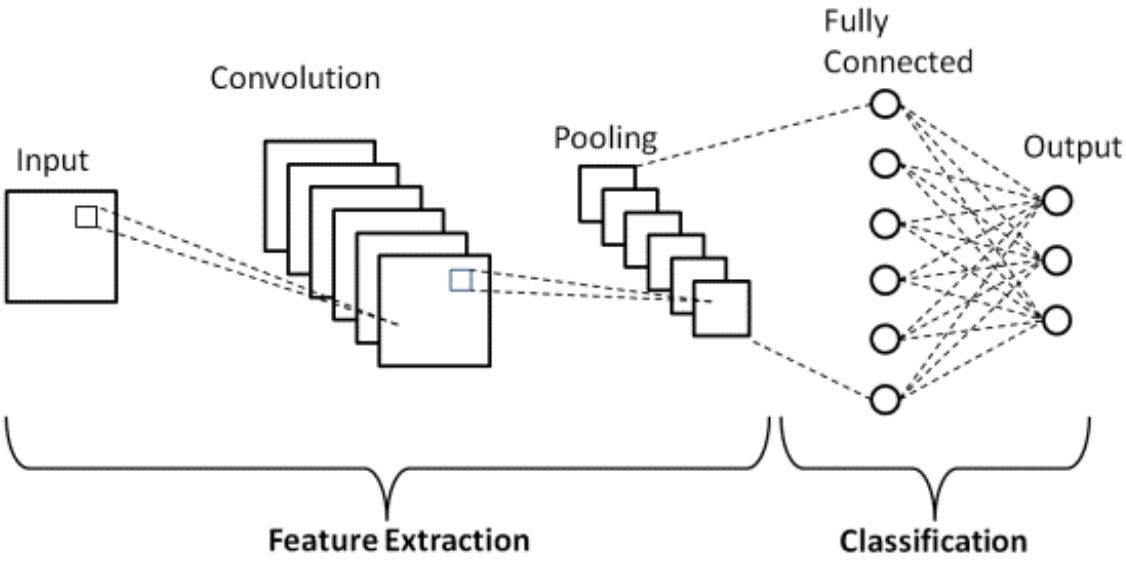
(https://towardsdatascience.com/why-you-should-use-bayesian-neural-network-aaf76732c150)

Figure 3.1.1: Representation of a Standard Neural Network versus a Bayesian Neural Network with probability distributions in place of weights.

3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural network that use convolutional layers to extract features from input data. CNNs are popular for applications in image analysis, but have shown to be highly effective in other areas where input data is multidimensional and analysis can benefit from the feature mapping technique. The basic functionality of a convolutional neural network can be broken down into four key parts: The input layer, the convolutional layer, the pooling layer and fully-connected layer(s). A schematic of the operations in a CNN is shown in Figure 3.2.2.

The convolutional layers are based on shared-weight convolutional kernels or filters

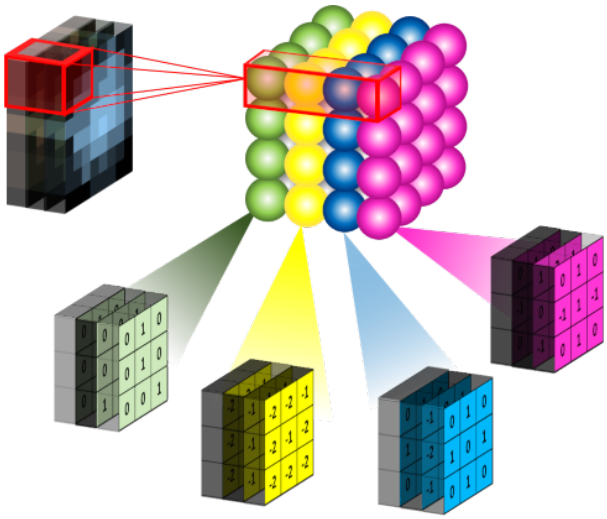


(https://www.analyticsvidhya.com/blog/2022/03/basic-introduction-to-convolutional-neural-network-in-deep-learning/)

Figure 3.2.1: Overview of the operations in a Convolutional Neural Network for image classification.

that slide along the input features produce feature maps. The objective of this operation is to extract high level features from the input such as edges. To represent this mathematically we can consider a CNN where the input at the qth layer is of size $L_q \times B_q \times d_q$, where L_q is the height, B_q is the width and d_q is the depth. These values at the input layer are defined by the dimensions of the input; i. e., a color image 28 x 28 pixels would have the input values 28 x 28 x 3, later layers will have the same three dimensional organization, however the value of d_q at the hidden layers is much larger than 3. For $q > 1$, the grids of values are referred to as feature maps or activation maps. In a CNN, the parameters are organized into sets of 3-dimensional structural units called filters. The height and width of the filter is usually much smaller than that of the input, however the depth of the filter is always the same as that of the layer to which is is applied. The dimensions of the filter at the qth layer can be defined as $F_q \times F_q \times d_q$. The convolutional operation places the filter at each possible position in the input so that the filter fully overlaps the input and performs a dot product between the parameters in the filter and the matching grid in input volume, which treats the entries in the region of input and the filter as vectors of the same size. The number of positions for placing the filter on the input defines the feature for the next layer, so the number of alignments between the filter and input defines the spatial height and width of the next (hidden) layer. When

performing convolutions in the qth layer, the filter can be aligned at $L_{q+1} = (L_q - F_q + 1)$ positions along the height and $B_{q+1} = (B_q - F_q + 1)$ along the width, so that the filter fully overlaps the layer input without going out of the "input boundary", which results in a total of $L_{q+1} \times B_{q+1}$ possible dot products. The depth of a subsequent layer is defined by the number of filters with independent sets of parameters, where each set of spatially arranged features from the output of each filter is is called a feature map. The number of filters used in each layer controls the capacity of the model since it directly controls the number of parameters.



(https://leovan.me/cn/2018/08/cnn/)

Figure 3.2.2: Visual representation of filtering kernels in a CNN acting on multilayer input.

The pth filter in the qth layer has parameters denoted by the three dimensional tensor $W^{(p,q)} = [w_{ijk}^{(p,q)}]$. The indices i, j, k indicate the positions along the height, width and depth of the filter. The feature maps in the qth layer are represented by the 3-dimensional tensor $H^{(q)} = [h_{ijk}^{q}]$. When the value of q is 1, the special case corresponding to the notation $H^{(1)}$ simply represents the input layer (not hidden). Then the convolutional operations from the qth layer to the $(q+1)$th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(k)} \quad \forall \begin{cases} i \in \{1, ..., L_q - F_q + 1\} \\ j \in \{1, ..., B_q - F_q + 1\} \\ k \in \{1, ..., d_{q+1}\} \end{cases} \tag{3.5}$$

Where the process holds for an arbitrary number of convolutional layers. Conventionally the first convolutional layer is responsible for capturing low-level features such as edges and orientation of gradients. With the addition of subsequent convolutional layers network is able to adapt to higher level features.

Similar to the convolution layer, the pooling layer has the task of dimensionally reducing the convolved feature. In addition to reducing the size of the features, pooling is also useful for extracting dominant features showing rotational and positional invariance. The pooling layer operates over each activation map from the convolution and scales its dimensionality according to the type of pooling desired. This is commonly done using a "max" function that returns the maximum value from the portion of the input covered by the kernel. An advantage to max pooling is that it discards noisy activations. Another option is *average* pooling, which takes the average of the values covered by the kernel.

Pooling layers typically operate over a small rectangular window, or "pooling region", and produce a single output value for each region. There are several types of pooling layers commonly used in CNNs, including max pooling, average pooling, and L2 pooling. Max pooling is the most commonly used type of pooling, and works by selecting the maximum value within each pooling region. This helps to preserve the most important features of the input image while discarding irrelevant details. Average pooling, on the other hand, computes the average value within each pooling region, which can be useful for reducing the impact of noisy or irrelevant features. L2 pooling is a variant of average pooling that computes the L2-norm of the values within each pooling region. This can be useful for preserving the magnitude of the feature vectors while also reducing the impact of outliers. Pooling layers are typically inserted between convolutional layers in a CNN architecture, and are used to gradually reduce the spatial dimensionality of the feature maps. This helps to reduce the number of parameters in the network and improve its computational efficiency, while also allowing the network to capture higher-level features in the input data.

The fully-connected layer in a CNN takes as input the flatten output of the pooling layers and performs the linear operations with activation as described in the beginning of this section (and previous sections).

3.3 Graph Neural Networks

Graph neural networks (GNNs) are a class of neural network that have been designed to operate on "graph-structured" data. The idea behind these types of NNs is that real world data and relationships that exist between things can be represented by graphs. The graph in a GNN consists of two parts: nodes and edges. The nodes in the graph represent entities, and edges represent relationships between these entities. The goal of a GNN is to learn a function that maps the input graph to some output prediction.

The formulation of GNNs involves node features, edge features, message passing, and node update functions. Each node in the graph is associated with a feature vector x_i that captures information about the node. Each edge in the graph is associated with a weight w_{ij} that determines how much information is passed between the connected nodes. The message passing step involves passing information between the nodes and edges of the graph where each node receives messages from its neighboring nodes and combines these messages with its own features to update its representation. The messages themselves are calculated as a function of the features of the sender node, the receiver node, and the edge connecting them. The message from node j to node i at time step t can be represented as:

$$h_{j\to i}^{t} = f(x_j^t, x_i^{t-1}, w_{ij}) \tag{3.6}$$

where f is a message function that takes in the feature vectors for nodes j and i and the edge weight between them. After receiving messages from its neighbors, each node updates its representation based on these messages and its own features. The update function is a learned function that takes as input the current node representation and the aggregated messages from neighboring nodes. The update function for node i at time step t can be represented as:

$$x_i^t = g(x_i^{t-1}, \sum_{j\in N(i)} h_{j\to i}^t) \tag{3.7}$$

where g is a node update function that takes in the current node features and the aggregated messages from neighboring nodes. The message passing and node update steps are repeated multiple times until the node representations have converged to a stable state. The final node representations can be used for downstream tasks like node classification,

link prediction, or graph classification.

There are many variations of GNNs, each with their own specific message passing and update functions. Examples include Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and GraphSAGE. Despite their differences, all GNNs share the same core idea of learning node representations that capture both node-specific and neighborhood information, making them a powerful tool for modeling graph-structured data.

3.4 Recurrent Neural Networks

Traditional neural networks such as convolutional and fully-connected networks treat input data as independent examples, therefore having no "memory" of the past inputs. Recurrent Neural Networks (RNNs) introduce the concept of a "memory" by including interdependence between data using a feedback loop. This is achieved by taking the output of a layer and recycling it to be fed back into the input. The feedback loop allows information to pass laterally between nodes of a layer as opposed to only forward to subsequent layers. Another perspective on this is allowing the neural network to incorporate temporal evolution.

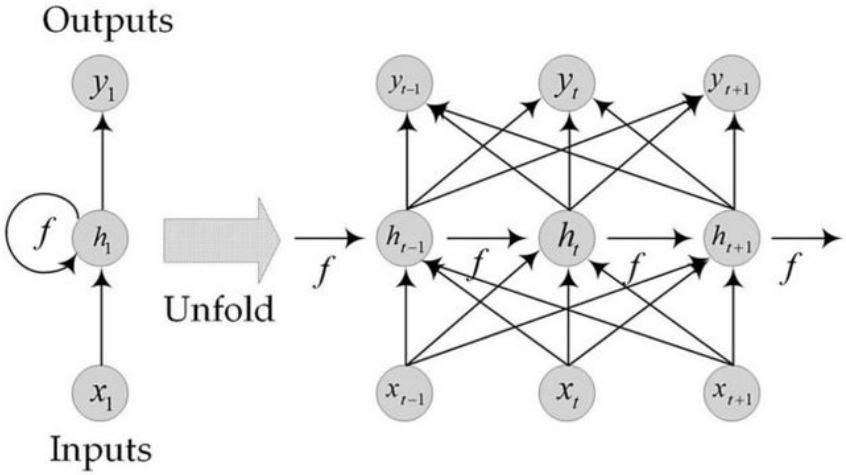


Figure 3.4.1: Recurrent Neural Network architecture. Source: Tadjer et al., 2021

For a calculation at a network layer h_t with input x_t, the calculation is as follows:

$$h_t = \Phi(W_x x_t + W_h h_{t-1} + b) \tag{3.8}$$

where the calculation at the layer h_t uses both the input data x_t and the calculation

for the same layer from a previous input, allowing RNNs to handle sequential data and identify patterns in historical data. RNNs tend to suffer from vanishing gradient descent. During the network training, the gradients of the loss function with respect to the weights are calculated using back propagation. Due to the repeated multiplication of the weight matrices during this process, the gradients can become very small as they propagate backwards in time thus creating the "vanishing gradient problem." When the gradients become very small, the network has trouble learning any further as the updates to the weights become very small (proportional to the gradient). Several techniques have been developed to mitigate the vanishing gradient problem in RNNs, including using gradient clipping to limit the size of the gradients, and using specialized RNN architectures such as long short-term memory (LSTM) or gated recurrent unit (GRU) networks, which are designed to better handle long-term dependencies.

3.5 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a type of deep neural network architecture designed for generating new data that resembles a given dataset (real data). GANs consist of at least two neural networks: a generator and a discriminator. The generator takes in a random noise vector as input and generates synthetic data, while the discriminator takes in the synthetic data from the generator plus a real data example and then attempts to classify them as either real or fake.

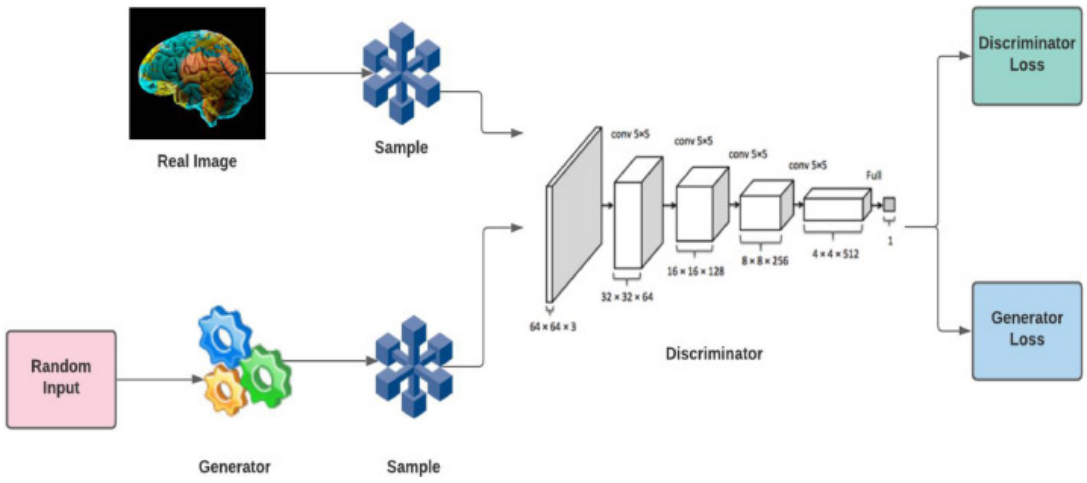


Figure 3.5.1: Simple schematic representation of the GAN architecture. Source: Aggarwal et al., 2021

The goal of the GAN is to train the generator to produce a synthetic output that is

indistinguishable from the real data according to the discriminator. This is done through an adversarial training process, where the two networks play a min-max game with each other. The generator tries to generate data that the discriminator will classify as real, while the discriminator tries to correctly classify the real and synthetic data.

The actual generator and discriminator parts of the GAN can have many different architectures depending on the task. For example, if the goal of a GAN is to produce a realistic data grid such as a precipitation map, this can be treated like a photo input/output. In this case it is likely a CNN is the best architecture for the generator and discriminator.

In addition to taking random noise vectors as input, the generator could take in other data as input if the goal is to improve or in some way modify that data. This implementation of GANs is common for atmospheric research, as will be explored later on.

3.6 Deep Reinforcement Learning

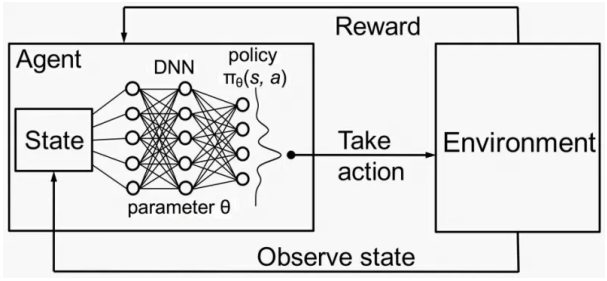
Reinforcement learning (RL) is a type of machine learning that involves an agent choosing an action to interact with its environment. The environment in a reinforcement learning algorithm is typically stated in the form of a Markov decision process (MDP). The reinforcement agent interacts in the environment in discrete timesteps, where at each timestep t the agent receives the current state s_t and a reward r_t. The agent then chooses an action a_t from a set of available actions which is subsequently sent to the environment. The environment then moves to a new state s_{t+1} according to a transition function or transition probability and the reward r_{t+1} associated with the transition is determined. The agents action selection is modeled as a map called a policy π.

(https://medium.com)

Figure 3.6.1: Process of reinforcement learning.

In practical situations the states of the MDP can have high dimensionality and can-

not be solved by these normal RL agents. Deep reinforcement learning (DRL) is the process of combining a "traditional" neural network architecture inside the reinforcement learning environment, where the entire neural network acts as the agent, and the policy is represented by the neural network. The addition of the neural network into the algorithm allows it to take in a very large input.



(https://medium.com)

Figure 3.6.2: Schematic of Deep Reinforcement Learning architecture.

3.7 Transformers

Transformers are a type of neural network architecture designed to transform an input sequence into an output sequence. The design of these models aimed to simultaneously solve the problem of input parallelization as well as the long term dependency issues with RNNs. The transformer model is a combination encoder-decoder together with an attention (or self-attention) mechanism. The attention mechanism is designed to better capture the long range dependencies in sequential data by focusing on specific parts of subset of the sequences of information given.

The transformer model can contain a number of encoders and decoders. Each encoder consists of two parts: there is the attention mechanism and a feedforward neural network. Inputs to the encoder first enter the self-attention layer, allowing the encoder to observe the entire input sequence as it encodes a specific part of the data. The decoder consists of both the self-attention mechanism and feedforward neural network with an additional encoder-decoder attention mechanism between the two layers.

Each "packet" of data input, or *token*, is turned into a vector using an embedding algorithm. The self-attention mechanism works by creating three vectors from each of the encoders input vector. For each embedding vector, a query vector, a key vector and a value vector are created. These vectors are then used to calculate the output of the

self-attention layer for that specific token. Transformer models make use of multihead attention, where a sequence of input data is embedded and processed sequentially.

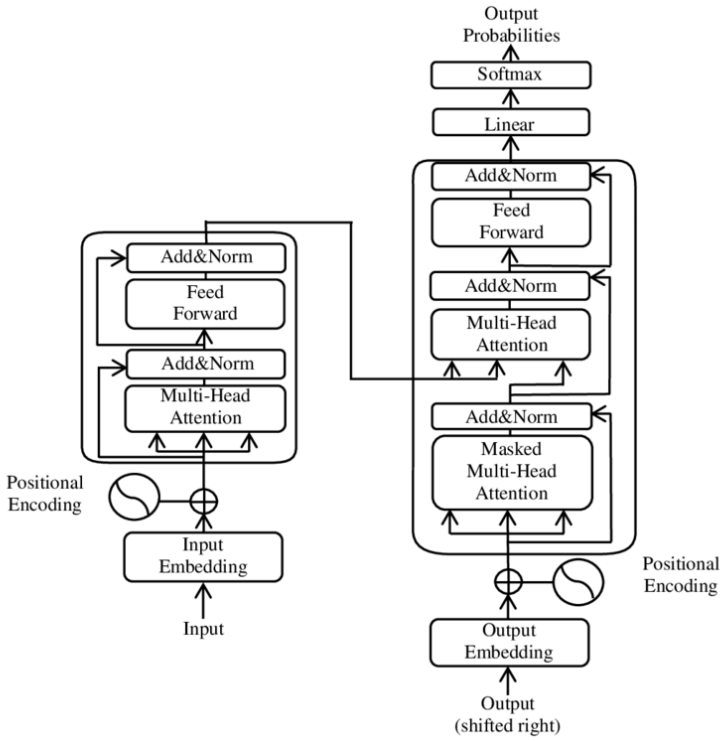


Figure 3.7.1: Transformer model architecture. Source: Jia, 2019

In addition to the self-attention mechanism, the transformer architecture also uses feedforward neural networks that operate on each tokens embedding vector. The transformer is trained using supervised learning, where the network learns to predict the correct output token given the input sequence, and the parameters are optimized using backpropagation.

4 Physics Informed Neural Networks

Physics-informed neural networks (PINNs), or physics-constrained, are a type of modified neural network architecture that uses physical laws to incorporate constraints into the loss functions. By adding a regularization term to the loss function, the network can be encouraged to satisfy certain aspects of governing equations of the physical system being modeled. Consider a physical system described by a set of governing equations:

$$\mathcal{L}_{PDE} = f(u(x), \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, ..., x) \tag{4.1}$$

where $u(x)$ is the solution to the system at the point x, and $\mathcal{L}_{PDE}$ is the governing equation. The goal is to train a neural network to approximate the solution $u(x)$ to this equation. The loss function of the PINN is defined as a linear combination:

$$\mathcal{L}_{\text{PINN}} = \mathcal{L}_{\text{data}} + \lambda \mathcal{L}_{\text{phys}} \tag{4.2}$$

where $\mathcal{L}_{\text{data}}$ is the loss considering neural network prediction and the true data, and $\mathcal{L}_{\text{phys}}$ is the "physics-informed" loss that enforces the constraints of the governing equations and λ is a hyperparameter that controls the relative importance of the two terms. The *data* loss function typically uses the mean squared error between the neural networks prediction and the true data:

$$\mathcal{L}_{\text{data}} = \frac{1}{N} \sum_{i=1}^{N} ||u(x_i) - \hat{u}(x_i; \theta)||^2 \tag{4.3}$$

where $u(x_i)$ is the true data value at location x_i, and $\hat{u}(x_i; \theta)$ is the model prediction. The physics-informed loss term is based on the governing equations:

$$\mathcal{L}_{\text{phys}} = \frac{1}{M} \sum_{j=1}^{M} ||f(u(x_j), \frac{\partial u}{\partial x}(x_j), \frac{\partial^2 u}{\partial x^2}(x_j), ..., x_j)||^2 \tag{4.4}$$

where M is the number of points chosen to enforce the governing equations, x_j is the location of the j-th point, and $f(u(x_j), \frac{\partial u}{\partial x}(x_j), \frac{\partial^2 u}{\partial x^2}(x_j), ..., x_j)$ is the residual of the governing equation at location x_j. The loss function is then minimized using standard gradient-based optimization algorithms such as those discussed previously. The resulting model is both accurate in predicting the solution as well as satisfying the governing equations of the physical system.

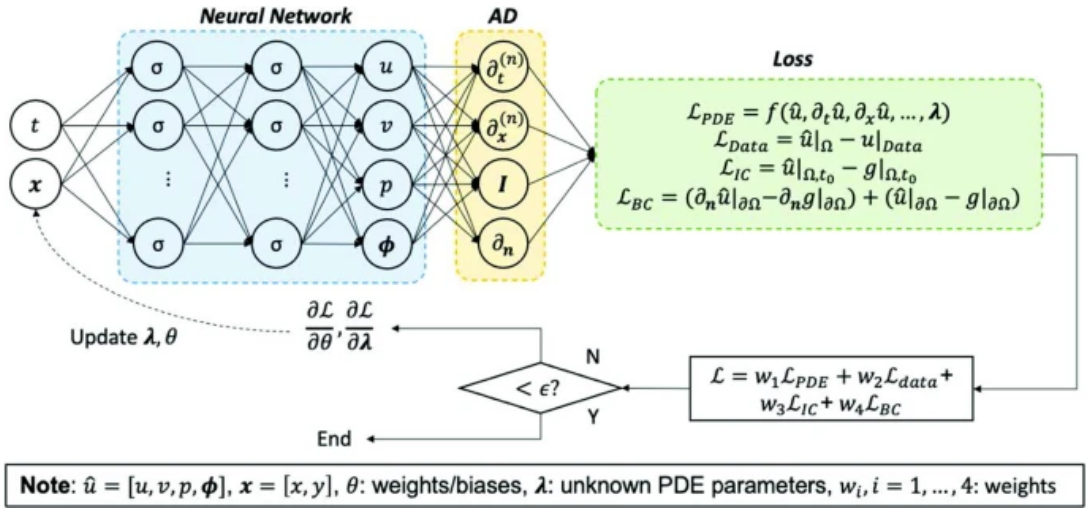


Figure 4.0.1: Schematic of a Physics-informed Neural Network. Source: Cai et al., 2021

5 Applications of Deep Learning for Numerical Weather Prediction

In this section, a number of studies will be discussed which have applied various different deep learning techniques to Numerical Weather Prediction (NWP) and forecasting. Due to the novelty of this field, the discussion will be limited to a number of recent studies.

5.1 Downscaling with GANs, Leinonen et al., 2020

A study published in 2020 titled "Stochastic Super-Resolution for Downscaling Time-Evolving Atmospheric Fields with a Generative Adversarial Network" by Leinonen et al. aimed to combined a number of deep learning techniques in order to produce an ensemble of high-resolution atmospheric fields from a low-resolution input.

The study uses a conditional GAN, in which both the generator (G) and the discriminator (D) are given a low resolution image, and the discriminator is trained to distinguish between real high-resolution images from the training dataset and artificial high-resolution images produced by the generator, conditionally to the corresponding low-resolution images. In this network both the G and D are deep CNNs with recurrent layers in the form of convolutional gated recurrent units. The convolutional layers are able to learn the deep features present in the input image while the recurrent layers permit the network to learn the temporal evolution. The input was a sequence of low-resolution time-evolving atmospheric fields and the output is an ensemble of high-resolution sequences of images of the same field. The low resolution image sequences were obtained by downsampling high-resolution images, allowing a "real" Using a training dataset of

180000 image sequences of 8 images each, results from the generator were saved after 361600 training sequences. An example of the GAN output for a single time instance compared with low-resolution and real high-resolution images is shown in Figure 5.1.1.

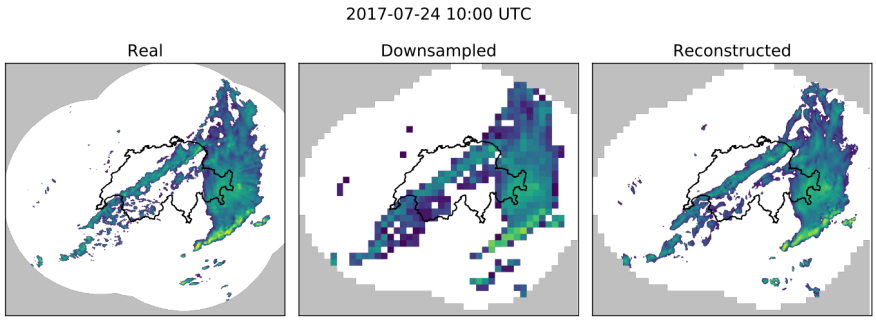


Figure 5.1.1: Example of High-resolution real data compared with downsampled data and reconstructed NN output. Source: Leinonen et al., 2020

The quality of image reconstruction was measured using an array of error analysis techniques such as root mean square error (RMSE), multi-scale structural similarity index (MS-SSIM) and log spectral distance (LSD). The calculations for these metrics, along with the continuous ranked probability score (CRPS), are shown in Figure 5.1.2.

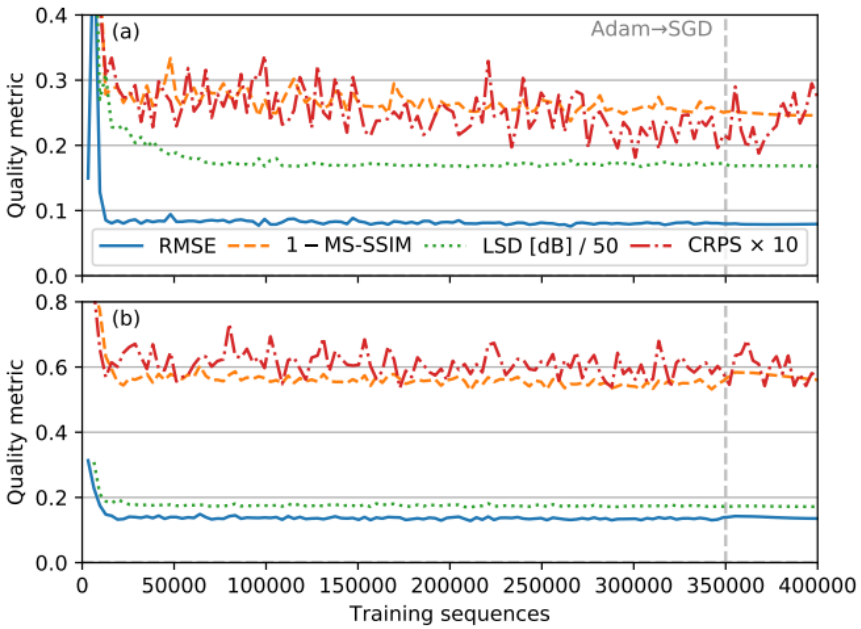


Figure 5.1.2: Calculations for error metrics with respect to number of training sequences. Source: Leinonen et al., 2020

5.2 Downscaling Precipitation Forecasts with GANs, Harris et al., 2022

As an appropriate follow up to the study of Leinonen at al., the study "A Generative Deep Learning Approach to Stochastic Downscaling of Precipitation Forecasts" by Harris et al. published in 2022 uses GANs for to post-process global weather forecast model output in order to produce more realistic precipitation forecasts than the input forecast data. The model was trained to take input from ECMWF IFS operational forecast data set at approximately 10 km resolution that included nine variables: Total precipitation, convective precipitation, surface pressure, TOA incident solar radiation, CAPE, total column cloud liquid water, total column water vapor and horizontal component wind velocities at 700 hPa. The corresponding "truth" data set for comparison with outputs was 1km resolution UK composite rainfall data from the Met Office NIMROD System.

This study used two variations of GANs, both with the same goal of post-processing the low resolution forecast data and producing ensembles of high-resolution precipitation forecasts. Schematics for the two models are shown in Figure 5.2.1. The first model was a conditional GAN, where both the generator and discriminator were conditioned with lower-resolution atmospheric fields and full resolution orography and land-sea mask data. The generator has an explicit noise input, allowing the multiple samples to be generated for a given forecast. The discriminator is the trained to distinguish between high-resolution predictions produced by the generator and the corresponding "true" high-resolution precipitation data.

The second model was a hybrid variational auto-encoder (VAE) GAN, where the generator of the GAN is substituted by the VAE, which consists of an encoder which maps the input to a latent space representation of the input data encoded with the means and log variances of normal random variables, and a decoder which samples the normal distributions via an external noise input. Table 5.2.1 shows numerical results for the two GAN models along a number of other statistical post-processing models including ecPoint, RainFARM and Lanczos. ecPoint is an ECMWF statistical post-processing technique that gives a prediction for rainfall intensity at a point, RainFARM (Rainfall Filtered Auto Regressive Modeling) is a downscaling method developed specifically for rain, and Lanczos is an interpolation based image scaling method. In terms of the continuous ranked probability score (CRPS), the GAN and VAE-GAN by Harris et al. outperformed all the other models, with the VAE-GAN model slightly outperforming the GAN.

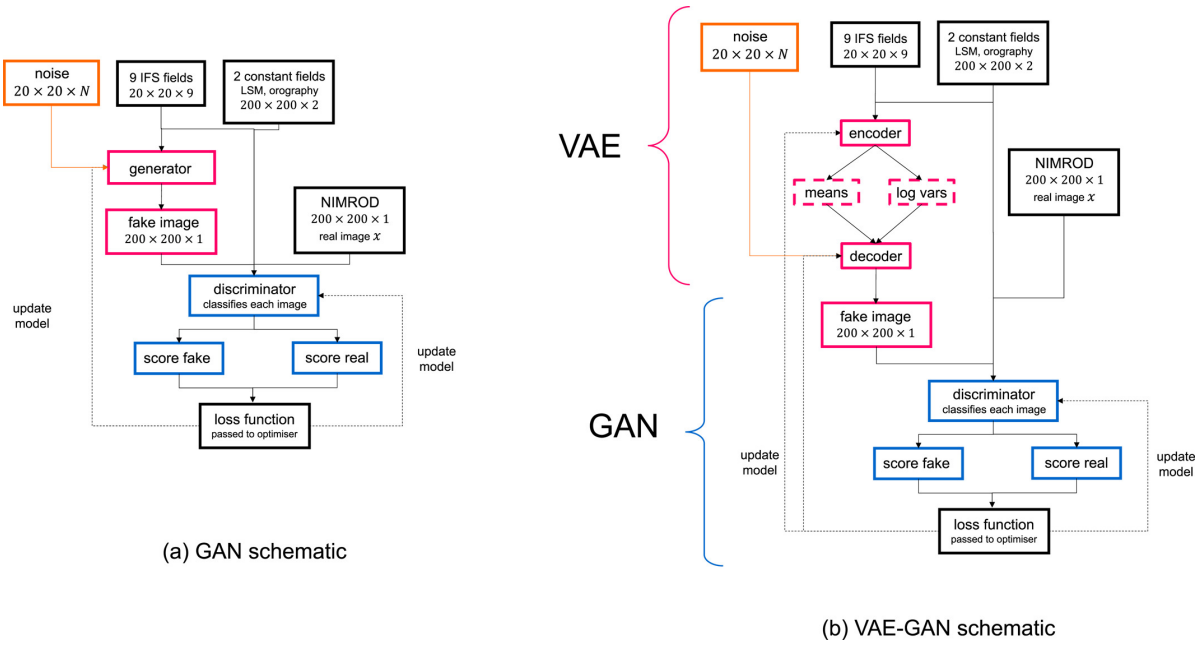


Figure 5.2.1: Schematic of the GAN and VAE-GAN used in Harris et al., 2022.

Figure 5.2.2 shows plots from the different models for four example cases. Comparing the results from the different models, the GAN appears to produce the most detailed and visually realistic than all the other models. The GAN is also able to reproduce the more intense rainfall which is not well localized in other models.

Model	Evaluation metric							
	CRPS (mm/hr)					RALSD (dB)	RMSE (mm/hr)	
	Pixelwise [a]	Avg 4 [a]	Max 4 [a]	Avg 16 [a]	Max 16 [a]		Ens-mean	Individual
GAN	0.0856	0.0844	0.1151	0.0806	0.2117	**4.88**	**0.404**	0.528
VAE-GAN	**0.0852**	**0.0840**	**0.1147**	**0.0802**	**0.2104**	5.34	0.405	0.499
ecPoint no-corr [b]	0.0895	0.1075	0.3987	0.1195	1.5948	16.35	0.423	0.644
ecPoint part-corr [b]	0.0895	0.0889	0.1255	0.0883	0.2485	9.78	0.423	0.644
RainFARM	0.1331	0.1332	0.1697	0.1286	0.2888	9.95	0.442	**0.444**
Lanczos [c]	0.1412	0.1392	0.1731	0.1309	0.2923	15.38	0.447	
Det CNN [c]	0.1347	0.1325	0.1644	0.1250	0.2817	16.74	0.404	

Table 5.2.1: Evaluation metrics for the GAN and VAE-GAN in comparison with other models from Harris et al., 2022.

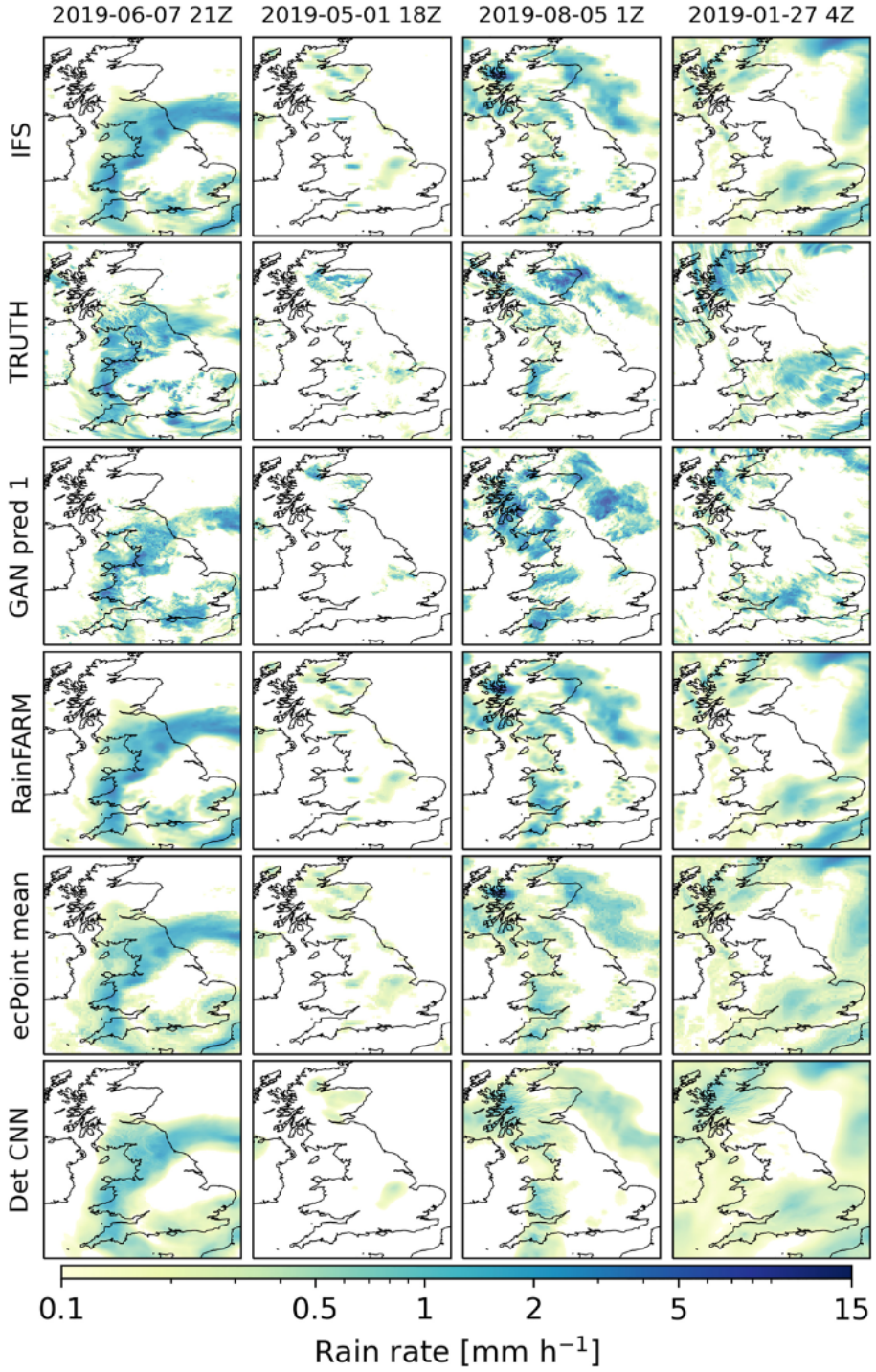


Figure 5.2.2: Comparison of true data with outputs of the GAN model and other models. Source: Harris et al., 2022.

5.3 CNNs for Improving WRF Simulations, Sayeed et al., 2021

Despite recent advancements in numerical weather prediction models that feed a greater understanding of weather phenomena, the models themselves contain inherent biases due to the parameterization of many subscale processes. The study "A Deep Convolutional Neural Network Model for Improving WRF Simulations" by Sayeed et al. aims to use CNNs as a postprocessing technique to improve mesoscale WRF simulation outputs. The WRF simulations were run at 27 km resolution using NCEP FNL data. Training of the model involved the use of ASOS meteorological station data. The parameters obtained from the stations were wind speed, wind direction, precipitation, relative humidity, temperature, dewpoint temperature, and surface pressure.

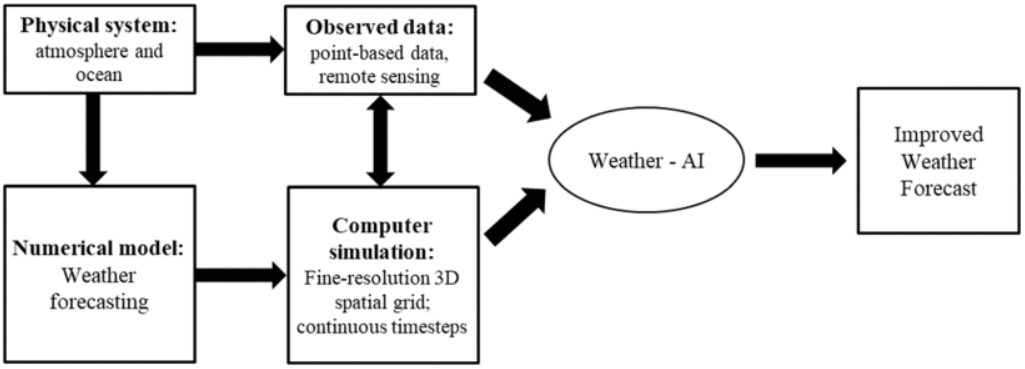
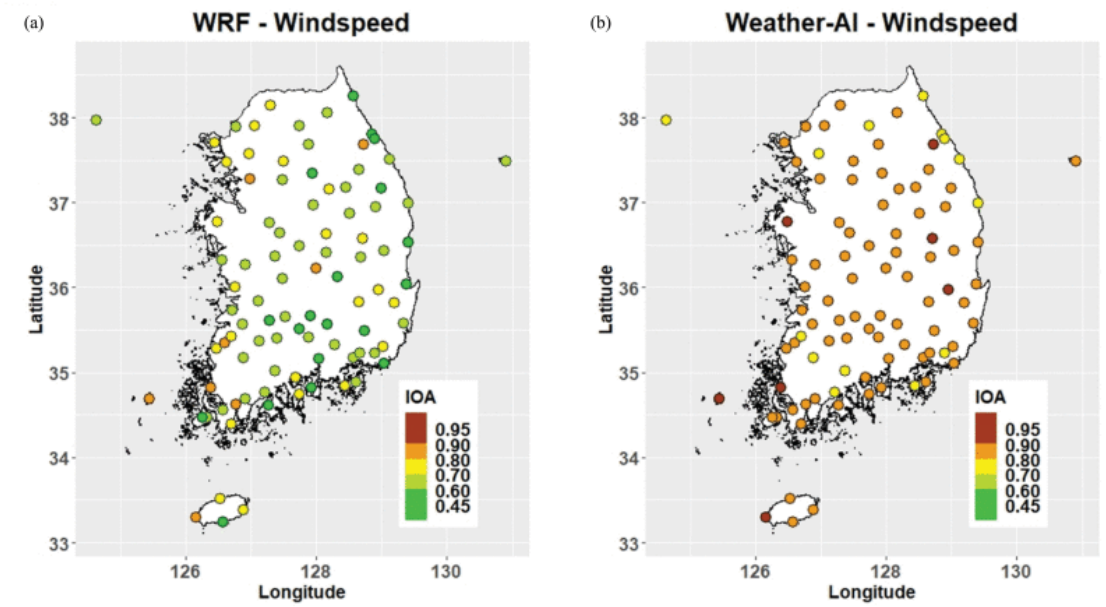


Figure 5.3.1: Flow chart for the Weather-AI system from Sayeed et al., 2021.

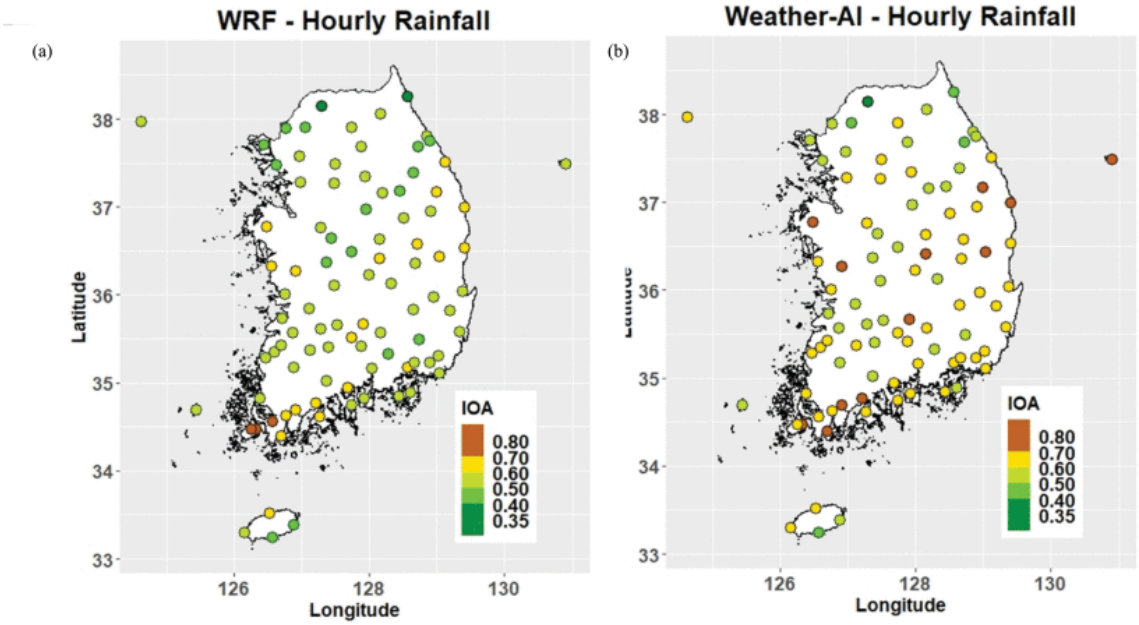
Upon completion of the WRF simulation, the grid point closest to each station was identified and assigned to extract and compare meteorological data during the training phase. The ML model was trained using four years of WRF simulations and meteorological data from 2014 to 2017, with 2018 as the test data. Results of the model output were evaluated against the WRF outputs, performance was based on index of agreement (IOA) with the observations. Figure 5.3.2 shows the results for the WRF and ML model output with IOA values for each station for two of the output variables: wind speed and hourly rainfall. For wind speed, the models performance is significantly better than the WRF forecast, showing improvement at all 93 stations in the range of 2.3-39.3%. The performance of the model for hourly rainfall is less obvious, however the model showed a slight overall improvement 6% over WRF alone.

The deep CNN used is a relatively simple model and was able to reduce bias increase the accuracy of predictions with respect to the WRF model alone when trained with

observation station data. While the result is impressive, the method does not allow for WRF domains over the sea due to the lack of observational stations for training.



(a) WRF vs. WRF+NN for Windspeed



(b) WRF vs. WRF+NN for hourly rainfall

Figure 5.3.2: Results from the Weather-AI model by Sayeed et al. for windspeed and hourly rainfall index of agreement of models with weather stations.

6 Deep Learning for Tropical Cyclone Meteorology

Numerical modelling of tropical cyclones is a challenging problem in meteorological forecasting. Due to intensity of such storms, the temporal resolution required to resolve the

phenomena without instabilities is extremely high with respect to most other phenomena. The combination of these temporal limitations with the fact these storms can reach near synoptic scale, creates a unique problem in computational requirements to simulate such phenomena. Machine learning presents many potentially useful applications to the field of modelling and forecasting tropical cyclones, in order to increase the reliability and reduce the computational cost of numerical simulations, or to replace the numerical simulations all together in some aspects. Here following will be analysed a number of early and recent studies applying a number of machine learning approaches to tropical cyclone meteorology.

6.1 Track and Intensity

Tropical cyclone track and intensity prediction is a challenging task in meteorology, as it involves the stochastic nature of the atmosphere, as well as complex interactions between the atmosphere and ocean. However, recent studies have shown that demonstrated the potential of numerous deep learning methods to improve both tropical cyclone track and intensity prediction.

6.1.1 Hurricane Tracking with NNs, Johnson and Lin, 1995

One of the earliest examples (if not the first) of applying a neural network approach to the forecasting of tropical cyclones is in the study "Hurricane tracking via backpropagation neural network" by Johnson and Lin in 1995. The purpose of the study was to investigate the ability of a Neural Network to predict the future tracks of hurricanes in the North Atlantic basin six hours in advance when trained using historical data. The model used was a simple feedforward neural network with one hidden layer. The input was very simple, including only time, location, speed, direction, maximum wind, minimum pressure and cyclone stage. All in all the network was extremely humble by modern standards. The output of the network is then the same set of variables as the input but forecast six hours. The results from this model were as to be expected from such an analogue type model, with approximate average errors of 1.63°latitude and 5.75°longitude. Johnson and Lin state that the model provides the performed surprisingly well (assuming their low expectations) and list a number of meteorological variables to include in future studies with the similar purpose.

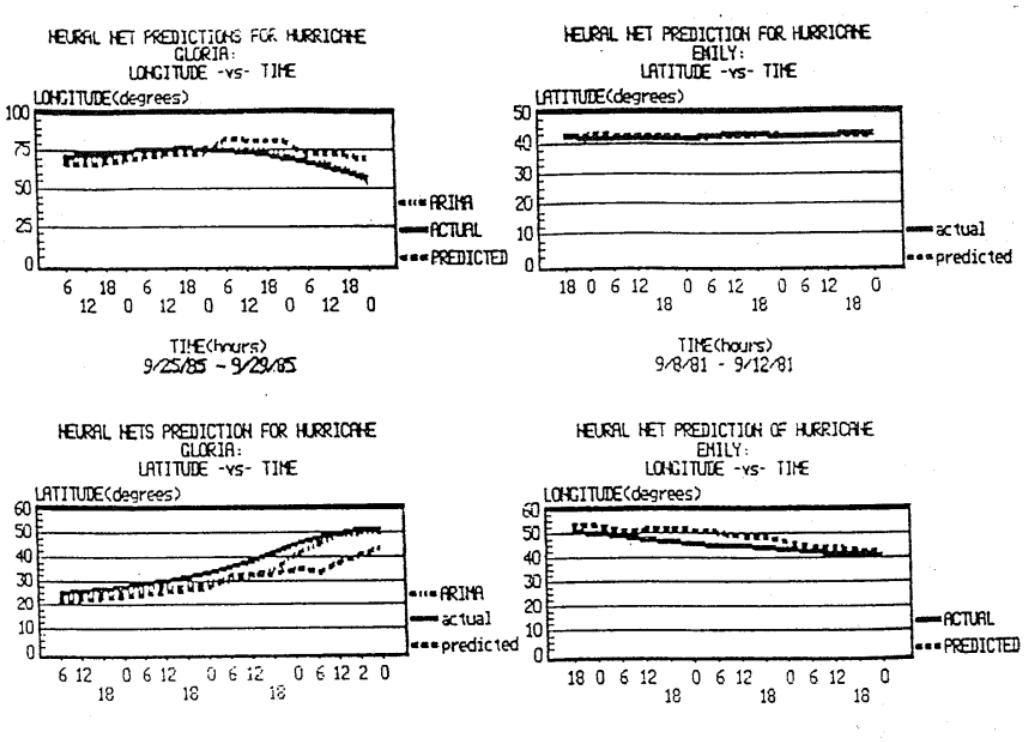


Figure 6.1.1: Results for a few a tropical cyclone track predictions from the study by Johnson and Lin, 1995.

6.1.2 Hurricane Track prediction using Reanalysis, Giffard-Roisin et al., 2018

A particularly interesting study from Giffard-Roisin et al. in 2018 titled "Fused Deep Learning For Hurricane Track Forecast From Reanalysis Data", create a relatively simple model from the philosophical standpoint to forecast hurricane trajectories using the input of reanalysis wind fields and the previous hurricane track. The model consisted of two parts in what they call a "Fusion Network". One part is a simple feedforward network for taking as input the hurricane track from 12 hour prior. The second part is a CNN that takes as input the wind fields at 700/500/225hPa from reanalysis at times t and t-6 hours. A simple schematic of the model used is shown in Figure 6.1.2.

Each part of the network was trained by itself initially so that each part could be analysed standalone for its contribution to the forecast accuracy. The last two layers of the the individual networks were then fused and retrained. The model was trained using the NOAA database IBTrACS, containing over 3000 extratropical and tropical cyclones. The output of the model is the 6 hour lead time forecast of the cyclones position. Results from the model are very promising, with mean Forecast error for the Fusion model at

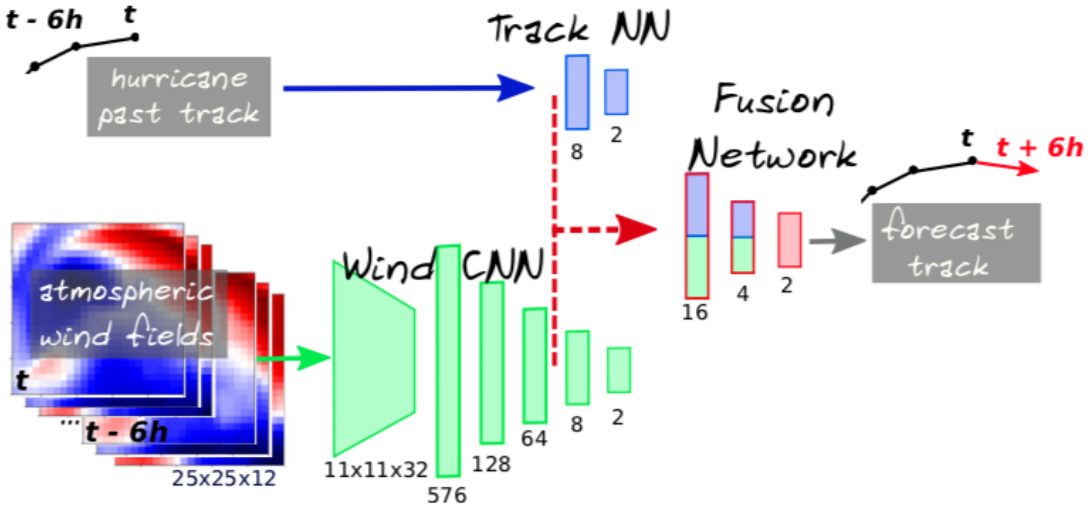


Figure 6.1.2: Simple schematic of the model used in Giffard-Roisin et al., 2018.

32.9 km. Results from the the individual network components are shown in Figure 6.1.3.

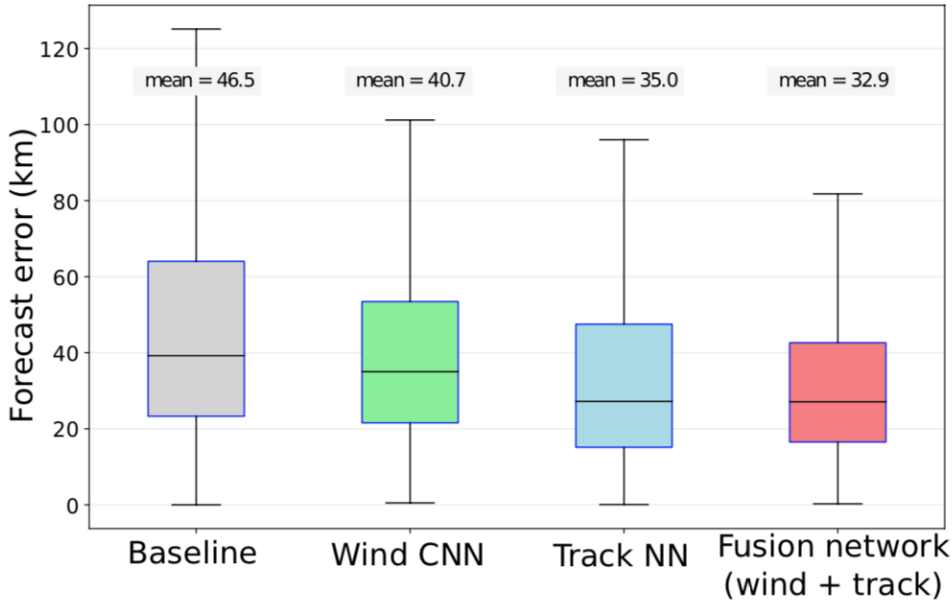
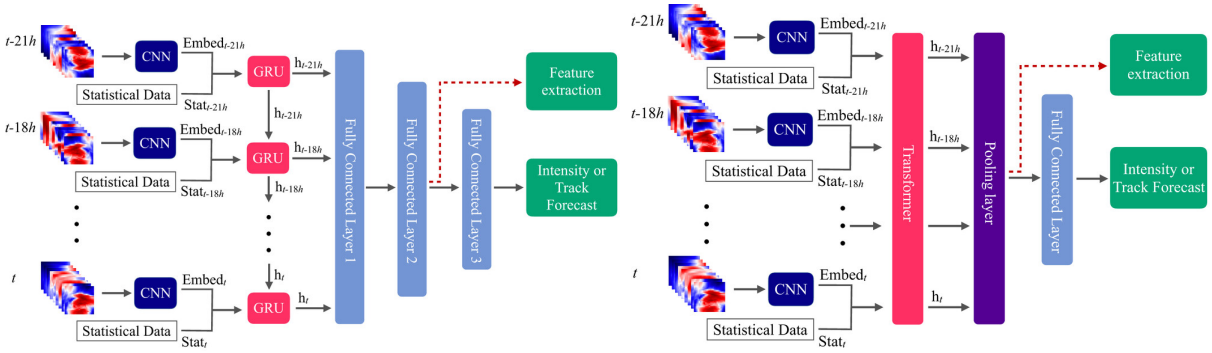


Figure 6.1.3: Accuracy of the model used in Giffard-Roisin et al., 2018.

The results from this study are very promising, and show that simple network configurations, with a sufficient input, can forecast short term cyclone tracks with relatively small error considering the computation time needed to perform such calculation once the network is already trained.

6.1.3 Hurricane Forecasting ML Framework, Boussioux et al., 2022

A recent study by Boussioux et al. titled "Hurricane Forecasting: A Novel Multimodal Machine Learning Framework" uses a deep learning model to improve hurricane track and intensity forecasts up to 24 hours. The model used an encoder-decoder architecture, where the encoder was comprised of a CNN. The decoder had two versions, one was comprised of a Gated Recurrent Unit (GRU) and the second used a transformer architecture. The schematics for these two network models are shown in Figure 6.1.4. Inputs to the convolutional layers include ERA5 Reanalysis maps at three pressure levels (225, 500 and 700 hPa) containing three variable features: 2-component wind (meridional and zonal) and geopotential height.



(a) CNN + GRU (b) CNN + Transformer

Figure 6.1.4: Model architectures used in Boussioux et al., 2022. a) Convolutional Neural Network (CNN) with Gated Recurrent Units (GRU) and b) CNN with transformer.

Reanalysis input maps are input from 3 hourly intervals from 21 hours before the time of forecast (t-21h) to the time of forecast (t). Input into the GRU and Transformer layers include the feature extraction output from the convolutional layers and the historical data. The historical data includes the position, wind intensity and minimum pressure of the storm at 3 hourly intervals from t to t-21h, coinciding with the reanalysis map inputs. Historical storm training data for the model was obtained from the NOAA IBTrACS dataset. Training was performed by splitting the data by year ranges. Training and validation was performed using storms from all basins from 1980-2015. The test set then comprised storms from 2016 to 2019 only from the North Atlantic and Eastern Pacific basins. Results from the study show the stand-alone machine learning models performance is comparable to that of the operational models. Table 6.1.1 shows the 24h lead time error for storm track and wind intensity considering four versions of the

stand alone machine learning model compared with four operational models for storm track and compared with three operational models for wind intensity. In the case of track forecasting, three of the four operational models slightly outperformed the machine learning model; however the ML and operational models had an approximate difference of only 20km. In the case of intensity forecasting, the ML and operational models best cases were comparable in error, with a difference in forecast wind speed of less than one knot in both Eastern Pacific and North Atlantic basins.

		Eastern Pacific basin			North Atlantic basin		
		Comparison on 837 cases			Comparison on 899 cases		
Model type	Model name	MAE (km)	Skill (%)	Error sd (km)	MAE (km)	Skill (%)	Error sd (km)
Hurricast (HUML) methods	HUML-(stat, xgb)	81	33	47	144	28	108
	HUML-(stat/viz, xgb/td)	81	33	47	140	30	108
	HUML-(stat/viz, xgb/cnn/gru)	**72**	**40**	**43**	111	45	79
	HUML-(stat/viz, xgb/cnn/transfo)	**72**	**40**	**43**	**109**	**46**	**71**
Stand-alone operational forecasts	CLP5	121	0	67	201	0	149
	HWRF	67	45	42	75	63	**49**
	GFSO	65	46	45	**71**	**65**	54
	AEMN	**60**	**50**	**37**	73	64	55

(a) C

		Eastern Pacific basin			North Atlantic basin		
		Comparison on 877 cases			Comparison on 899 cases		
Model type	Model name	MAE (kt)	Skill (%)	Error sd (kt)	MAE (kt)	Skill (%)	Error sd (kt)
Hurricast (HUML) methods	HUML-(stat, xgb)	10.6	9.4	10.5	10.7	−4.9	9.3
	HUML-(stat/viz, xgb/td)	10.6	9.4	10.4	10.6	−3.9	9.2
	HUML-(stat/viz, xgb/cnn/gru)	**10.3**	**12.0**	10.0	10.8	−5.9	9.2
	HUML-(stat/viz, xgb/cnn/transfo)	**10.3**	**12.0**	**9.8**	**10.4**	**−2.0**	**8.8**
Stand-alone operational forecasts	GFSO	15.7	−34.2	14.7	14.2	−39.2	14.1
	Decay-SHIPS	11.7	0.0	**10.4**	10.2	0.0	9.3
	HWRF	**10.6**	**9.4**	11.0	**9.7**	**4.9**	**9.0**

(b)

Table 6.1.1: Results of the Model in comparison with standard numerical models for cyclone track prediction and intensity error from Boussioux et al., 2022.

Results from this study from Boussioux et al. are very significant in that they indicate certain combinations of modern neural network architectures, when exploiting the right data, are able to forecast with comparable performance to operational models. In addition to the comparable accuracy, machine learning models such as the one considered in this study can perform forecasts in a matter of seconds once they have been trained, compared to the high computational resources needed to run operational models.

6.2 Satellite Imagery Analysis

A popular application of deep learning to tropical cyclone analysis is in the use of satellite imagery to cyclone intensity. Due to the availability of a massive quantity of satellite imagery, in combination with the ability to make multiple training and test cases from the same storm at different times, this is an attractive area to investigate the potential of deep learning.

6.2.1 Tropical Cyclone Intensity from MW Satellite Imagery, Wimmers et al., 2019

The 2019 study "Using Deep Learning to Estimate Tropical Cyclone Intensity from Satellite Passive Microwave Imagery" by Wimmers et al. utilized a deep CNN to estimate the maximum sustained wind speed of a Tropical Cyclone from microwave satellite imagery. Microwave imagery used in the study was sourced from the Naval Research Lab TC MINT database which consists of 37 GHz and 89 GHz imagery from 1987-2012. Data used for the authentication of storm intensity was obtained from the HURDAT2 database from the National Hurricane Center. Figure 6.2.1 shows a histogram of the number of tropical cyclone image samples used in the in the model training relative to the corresponding maximum sustained wind speed. In order to avoid the model assigning lesser importance to imagery corresponding to storms of greater strength, data samples were duplicated according to their frequency to obtain an even distribution.

The model was trained in three ways: the first way incorporated imagery only from the 37 GHz band, the second used only 89 GHz, and the third used imagery from both 37 and 89 GHz bands. Figure 6.2.2 shows the RMSE and MSE of the three model cases with respect to the tropical cyclone intensity. These results from the training show the 89 GHz only model performs significantly better than the 37 GHz only model, and the 2-channel model outperforming the 89 GHz only model.

Figure 6.2.3 shows a number of input and output cases for the model: four cases of a Category 2 Hurricane and four cases of a Category 5 Hurricane. On the left side of each subfigure there are the input images, the 37 GHz channel on the left and the 89 GHz channel on the right. It is clear from the images that the 89 GHz channel contains much greater detail for the model to assess. The output of the model is shown on the right side of each subfigure in the for of a probability density function. From a thorough inspection

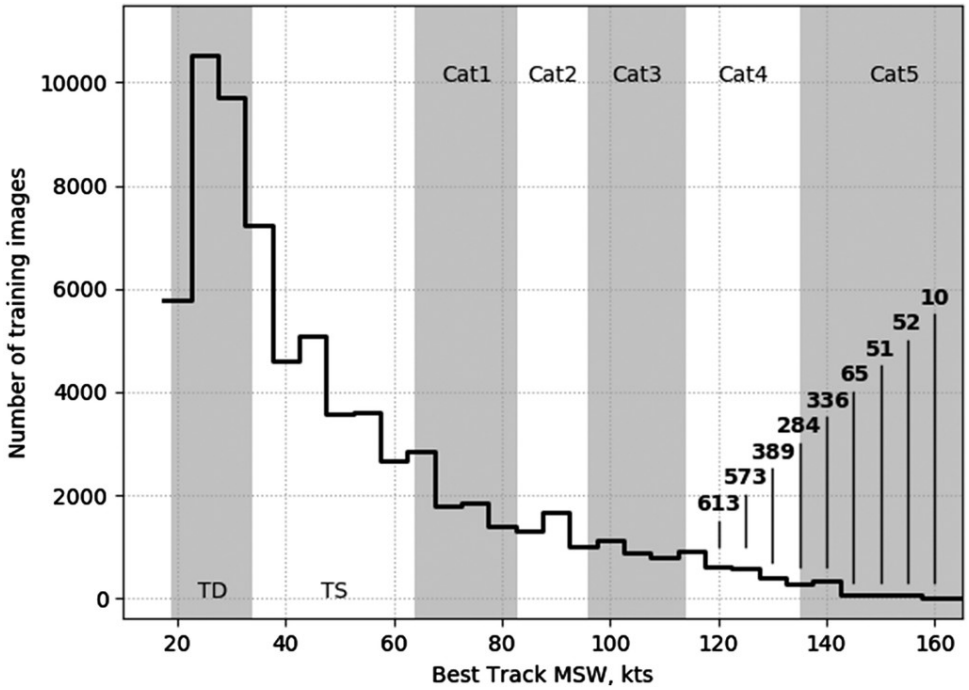


Figure 6.2.1: Histogram of available satellite imagery relative to storm intensity from Wimmers et al., 2019.

of the model input and output, Wimmers et al. state "the model predominantly responds to the characteristics of organization in the images, including eyewall definition, banding and the extent of cyclonic signatures.", they also note that departure in model accuracy in many cases appears that the image is either more organized or less organized than the maximum sustain wind speed typically indicates–which explains well the error curve seen in Figure 6.2.2.

The use of a relatively simple CNN model, the study was able to produce operational-quality intensity estimates for tropical cyclones that have traditionally not been used for qualitative techniques. The level of accuracy obtained from the implementation of a simple CNN model such as in this study is promising for the future of deep learning aided analysis.

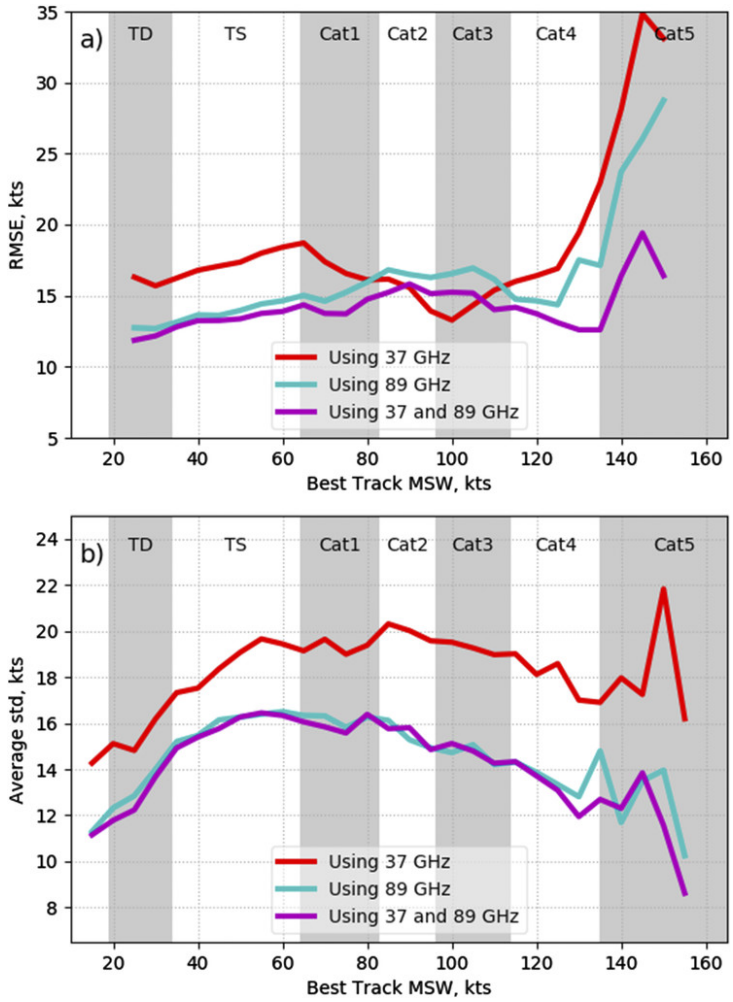
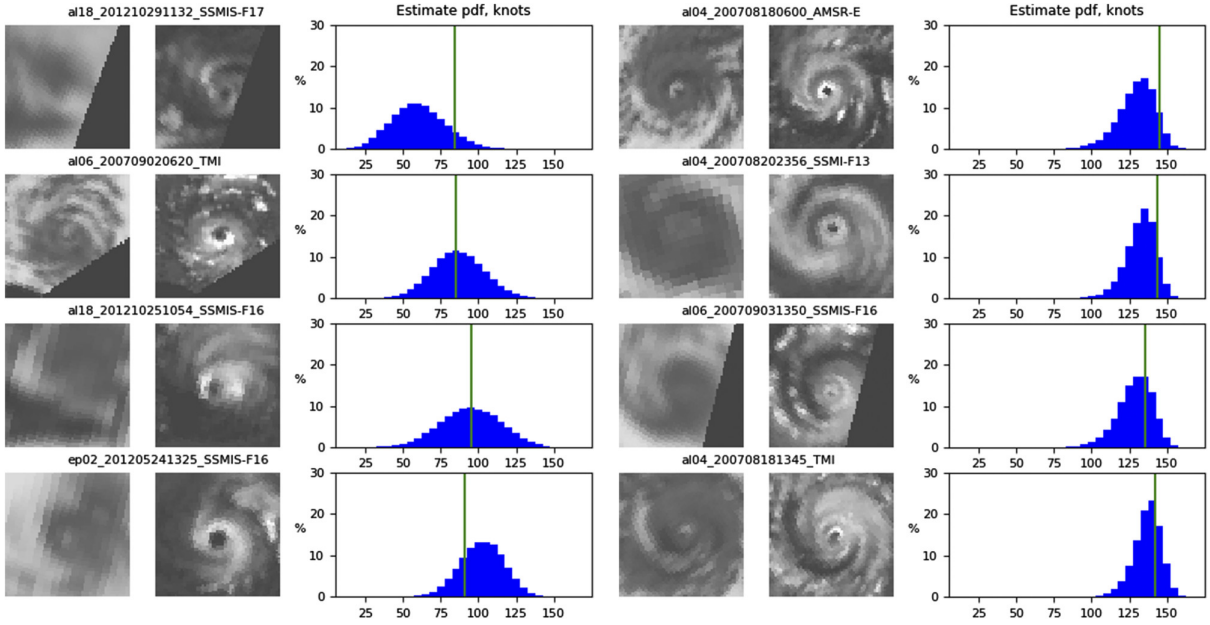


Figure 6.2.2: Cyclone intensity estimation error relative to observation frequency from Wimmers et al., 2019.



(a) Category 2 (b) Category 5

Figure 6.2.3: Examples of input images at 37 and 89 GHz ad corresponding windspeed probability distribution output from Wimmers et al., 2019.

6.2.2 Hurricane Intensity from Infrared Satellite Imagery, Dawood et al., 2020

In the 2020 study by Dawood et al. titled "Deep-PHURIE: deep learning based hurricane intensity estimation from infrared satellite imagery", infrared satellite images of tropical cyclones from 2001-2005 were used in training a deep learning model, called Deep-PHURIE, to predict the tropical cyclone intensity. The model consisted of a deep convolutional architecture with a 32 layer depth. Input data consisted of more than 170,000 IR satellite images, which were compared with historical IBTrACS data for storm intensity validation. Output of the model was the predicted maximum wind speed of the cyclone. Results show the simple model is an effective method of estimating tropical cyclone wind speed with a RMSE of approximately 9 knots for the Atlantic, Pacific and Indian basins.

7 Conclusion

The field of deep learning has exploded in the last decade, emerging as a powerful tool in a variety of applications. This paper has included a relatively thorough introduction to the fundamental components of neural networks and many of the popular architectures in use today. The constant improvement of optimization and regularization techniques as well as the recent development of new architectures such as transformers continue to push the limits of applications. Through a review of a handful of recent studies, we see how the application of numerous deep learning techniques have successfully been used to improve weather forecasts, including tropical cyclone track and intensity.

It is clear that this is the era of deep learning, and continuous advancements in deep learning architectures will continue to improve the reliability and precision of weather forecasting systems. The combination of deep learning with traditional meteorological models and the incorporation of physical constraints are already proving to yield operational quality results in many aspects, and are likely to show even further improved predictions in the very near future.

In conclusion, this paper serves as a glimpse into the vast potential of deep learning for both general meteorological forecasting as well as tropical cyclone prediction and research. As the field continues to grow and advance, it is vital for researchers to embrace the applicability of deep learning and its potential to numerous areas of science.

References

[1] Aggarwal, A., Mittal, M., and Battineni, G. Generative adversarial network: An overview of theory and applications. *International Journal of Information Management Data Insights 1*, 1 (2021), 100004.

[2] Aggarwal, C. C. *Neural Networks and Deep Learning*. Springer, Cham, 2018.

[3] Cai, S., Mao, Z., Wang, Z., Yin, M., and Karniadakis, G. E. Physics-informed neural networks (pinns) for fluid mechanics: A review. *Acta Mechanica Sinica 37*, 12 (2021), 1727–1738.

[4] Dawood, M., Asif, A., and Minhas, F. u. A. A. Deep-phurie: deep learning based hurricane intensity estimation from infrared satellite imagery. *Neural Computing and Applications 32* (2020), 9009–9017.

[5] Ding, B., Qian, H., and Zhou, J. Activation functions and their characteristics in deep neural networks. In *2018 Chinese control and decision conference (CCDC)* (2018), IEEE, pp. 1836–1841.

[6] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, 61 (2011), 2121–2159.

[7] Giffard-Roisin, S., Yang, M., Charpiat, G., Kégl, B., and Monteleoni, C. Fused Deep Learning for Hurricane Track Forecast from Reanalysis Data. In *Climate Informatics Workshop Proceedings 2018* (Boulder, United States, Sept. 2018), C. Chen, D. Cooley, J. Runge, , and E. Szekely, Eds., NCAR, pp. 69–72.

[8] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Griewank, A. Who invented the reverse mode of differentiation. *Documenta Mathematica, Extra Volume ISMP 389400* (2012).

[10] Harris, L., McRae, A. T. T., Chantry, M., Dueben, P. D., and Palmer, T. N. A generative deep learning approach to stochastic downscaling of precipitation forecasts. *Journal of Advances in Modeling Earth Systems 14*, 10 (2022), e2022MS003120. e2022MS003120 2022MS003120.

[11] Haykin, S. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.

[12] Jia, Y. Attention mechanism in machine translation. In *Journal of physics: conference series* (2019), vol. 1314, IOP Publishing, p. 012186.

[13] Johnson, G., and Lin, F. Hurricane tracking via backpropagation neural network. In *Proceedings of ICNN'95 - International Conference on Neural Networks* (1995), vol. 2, pp. 1103–1106 vol.2.

[14] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization, 2017.

[15] Leinonen, J., Nerini, D., and Berne, A. Stochastic super-resolution for downscaling time-evolving atmospheric fields with a generative adversarial network. *IEEE Transactions on Geoscience and Remote Sensing 59*, 9 (sep 2021), 7211–7223.

[16] Linnainmaa, S. *Alogritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä*. PhD thesis, Master's thesis, University of Helsinki, 1970.

[17] Lydia, A., and Francis, S. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci 6*, 5 (2019), 566–568.

[18] Nielsen, M. A. Neural networks and deep learning, 2018.

[19] O'Shea, K., and Nash, R. An introduction to convolutional neural networks, 2015.

[20] Polyak, B. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics 4* (1964), 1–17.

[21] Rahman, M. J.-U., Sultan, R. I., Mahmud, F., Al Ahsan, S., and Matin, A. Automatic system for detecting invasive ductal carcinoma using convolutional neural networks.

[22] Ruder, S. An overview of gradient descent optimization algorithms., 2016. arxiv:1609.04747.

[23] Sanchez-Lengeling, B., Reif, E., Pearce, A., and Wiltschko, A. B. A gentle introduction to graph neural networks. *Distill* (2021). https://distill.pub/2021/gnn-intro.

[24] Sayeed, A., Choi, Y., Jung, J., Lops, Y., Eslami, E., and Salman, A. K. A deep convolutional neural network model for improving wrf simulations. *IEEE Transactions on Neural Networks and Learning Systems 34*, 2 (2023), 750–760.

[25] Sazli, M. H. A brief review of feed-forward neural networks, 2006.

[26] Sharma, S., Sharma, S., and Athaiya, A. Activation functions in neural networks. *Towards Data Sci 6*, 12 (2017), 310–316.

[27] Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning* (2013).

[28] Svozil, D., Kvasnicka, V., and Pospichal, J. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems 39*, 1 (1997), 43–62.

[29] Tadjer, A., Hong, A., and Bratvold, R. B. Machine learning based decline curve analysis for short-term oil production forecast. *Energy Exploration & Exploitation 39*, 5 (2021), 1747–1769.

[30] Theodoridis, S. *Machine Learning: A Bayesian and Optimization Perspective*. Academic Press, 05 2015.

[31] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

[32] Wimmers, A., Velden, C., and Cossuth, J. H. Using deep learning to estimate tropical cyclone intensity from satellite passive microwave imagery. *Monthly Weather Review 147*, 6 (2019), 2261–2282.

[33] Zeiler, M. D. Adadelta: An adaptive learning rate method, 2012.