Neural Networks

1 Introduction

Neural Networks, also known as Artificial Neural Networks, are algorithms used to create models that exhibit the property of machine learning. For example, the models may be used to predict a certain property of an image that is given to it, or describe the behaviour of a certain object present in an image.

In neural networks, the computer is not told how to solve the problem. Instead, it is given a bunch of training data and test data and figures out a solution to the problem on its own. The idea is that the model learns a large number of patterns based on the inputs it is given - the training data - and determines the right output based on these patterns. The following sections will give us a deeper understanding of how a neural networks function.

Neural Networks are used in various fields like pattern and image recognition, speech translation and natural language processing.

1.1 General Composition

The illustration below gives us a sense of how a simple neural network looks like

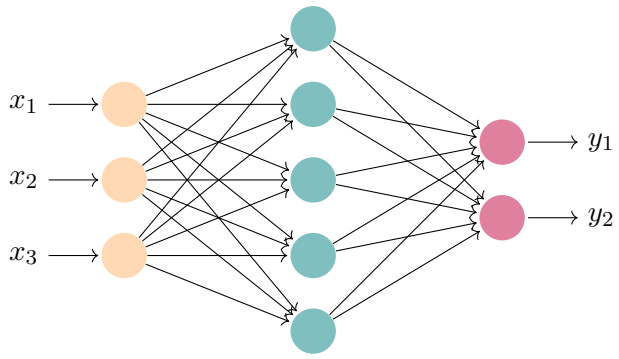

Figure 1: Artificial Neural Network with an input layer, a hidden layer and output layer

The model is essentially comprised of a number of nodes, formally known as neurons, categorised into multiple layers starting with an input layer, a hidden layer and an output layer.

Every neuron starting from the input layer is connected to one or more neurons in the next layer and the type of connections is given by certain weights attached to the connections between two neurons. Every neuron acts as an input-output box where the output it produces depends on the inputs it receives from its connections to its preceding neurons along with their weights.

Thus, a neuron, in the simplest sense, takes in an input and a gives out a single output, which is

computed using certain functions. We look into the components in more detail in the next section.

To understand a little more about how a network is designed, we can look at how the input layer is created in a generic model.

The inputs given to a network are usually images in the form of matrices, but in some cases, text is also passed as input in the form of strings or characters. Every image is composed of pixels, each having their own value based on their color. The easiest way to pass in these images into the network is to create the input layer in such a way as to contain the information of every single pixel. Due to this reason, input layers generally have as many neurons as there are pixels in the image it is receiving as input. For example, if we are passing a 3×3 matrix that stores an image, we create 9 neurons in the input layer, corresponding to each pixel in the image.

This dependency on the type of input being passed in and the desired output governs the entire structure of a neural network : what type of layers are used, how many neurons there must be, if all the neurons have to be connected, etc.

1.2 Components of an ANN

1. **Neurons**

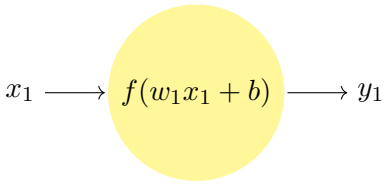

Figure 2: Sketch of a neuron receiving an input x_1, computing the activation based on its activation function, f, a weight, w_1 and a bias b and producing the output, y_1.

The above figure shows the layout of a single neuron. An input x_1 is passed along with a weight attached to that connection, say w_1. Every neuron has a specific function called its **activation function** that it uses to then calculate the output, y_1. If there are any more layers after this neuron, y_1 will be passed as input to the neurons in the next layer.

In a fully connected network, a neuron can have multiple inputs based on all the connections it has. In this case, we can compose a general outline of the processing in a neuron.

Consider a vector $\mathbf{x} = \{x_1, x_2, x_3, ..\}$ that contains all the inputs to a neuron and a vector $\mathbf{w} = \{w_1, w_2, w_3, ..\}$ that holds the respective weights.

Then the output is calculated as

$$y = f(\mathbf{w} \cdot \mathbf{x}) \tag{1}$$

This weighted sum, obtained by computing the scalar product between $\mathbf{w}$ and $\mathbf{x}$ is known as the activation and f is the activation function. In some cases, a bias term, b is added to the input

of this function whose role is to filter out only those weighted sums that are above (or below) the value of the bias.

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{2}$$

This output y then acts as an input to a successive neuron or acts as the final output.

Note 1. The bias term is also referred to as the threshold value of the neuron, where $b \equiv -\text{threshold}$

2. **Weights**

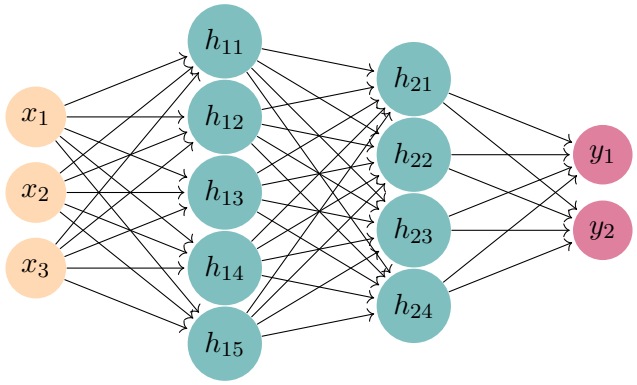

Figure 3: Dense/Fully-connected Network with two hidden layers

A weight is a number assigned to every connection between two neurons. The relative magnitude of the weight determines how important a neuron's output is to a successive neuron's input.

If we have a fully connected structure, once the output of all neurons is computed in the first pass, the weights will determine how important the output is to the next neuron.

Thus, if $w_i = 0$ for a connection, it implies that the respective connection between the neurons is nonexistent.

The weights are not predetermined and change in value based on the network's learning algorithm. The initial values of weights are randomised in order to calculate the outputs of neurons in the first pass.

From the figure above, we can compute the input to node h_{11} as the scalar product :

$$\mathbf{w_1} \cdot \mathbf{x} = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \tag{3}$$

where $\mathbf{w_1}$ represents the vector containing all the weights that connect the inputs to the first node of the first hidden layer and $\mathbf{x}$ is the vector containing the 3 inputs.

Note 2. The term **first pass** refers to the first time the neural network processes an input using all its initial values of weights and biases. The network begins learning only after this pass.

3. **Connections**

Neurons in one layer are only connected to neurons from the preceding and successive layer. There are many possible connection patterns between two layers. They can be **fully connected** (as mentioned above) or connected by pooling where a group of neurons are connected to a single neuron in the next layer, thereby reducing the number of neurons in the next layer.

Pooling neurons creates a directed acyclic graph and the neural networks that incorporate these types of connections are seen in **feedforward networks**. Feedforward networks are those where the output from one layer acts as an input to the next layer. Hence, information is only passed forward.

Fully connected neural networks are seen in **recurrent networks**. These networks incorporate feedback loops where the input function depended on the output.

Complex networks would incorporate a hybrid setup of pooling and fully connected connections. The diagrams shown in the previous sections are examples of fully connected, feed-forward networks.

4. **Activation Function**
The activation function is a predetermined function that produces the output of a neuron from its inputs. The functions can be linear or non-linear.

Note 3. There also exists a **propagation function** that just computes the weighted sum $\mathbf{w} \cdot \mathbf{x}$. A bias term is then added to the output of this function to produce an input to the activation function.

The most common example of a linear activation function is the **Identity function** :

$$f(x) = x \tag{4}$$

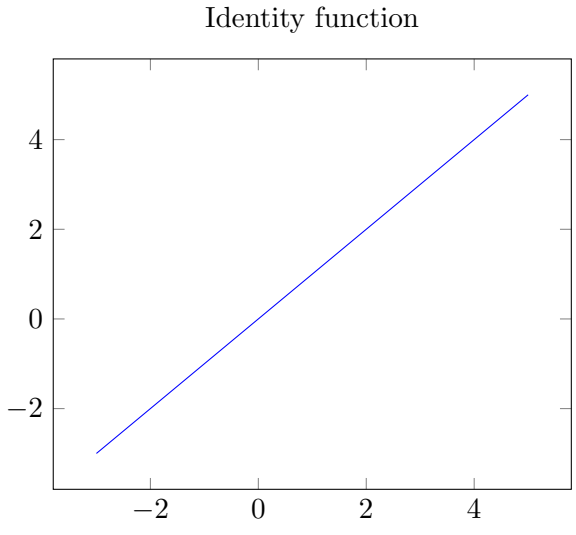

Figure 4: Graph of the identity function, $f(x) = x$

The function does not really modify any of the data being passed in and hence may not be frequently used in sophisticated networks.

However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes.

Over the years many non-linear functions have been used as activation functions. Some of the most common ones are listed below:

1. **Logistic Activation function**, or the Sigmoid function :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

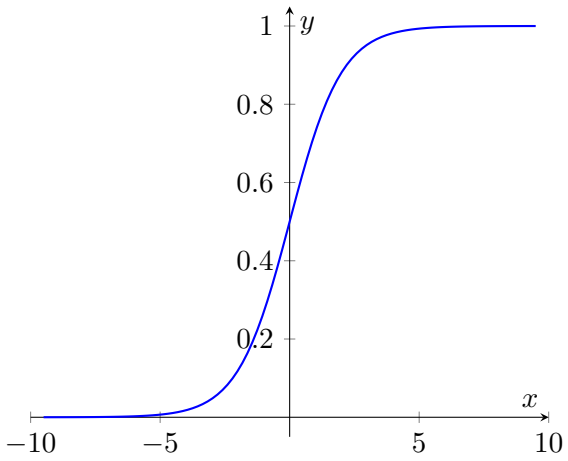

Figure 5: Graph of the Sigmoid function. The output ranges from $[0, 1]$.

Observing the graph shows us that the range of the function lies in the interval $[0, 1]$. Since the output of neurons act as inputs to successive neurons, it is feasible to make the inputs to the neurons in the first layer lie between $[0, 1]$ as well.

For example, if the inputs we pass in are images, we make sure every pixel in the image has a value between 0 and 1.

Another detail to note is that the x-axis is given as Z, which is another variable used to signify the weighted sum computation.

$$Z = \mathbf{w} \cdot \mathbf{x} + b \tag{6}$$

Sigmoid functions are especially used for models where a probability of an output needs to be predicted, since probability values also lie in the range $(0, 1)$

2. **Rectified Linear Activation function**, or ReLU :

$$\phi(x) = max(0, x) \tag{7}$$

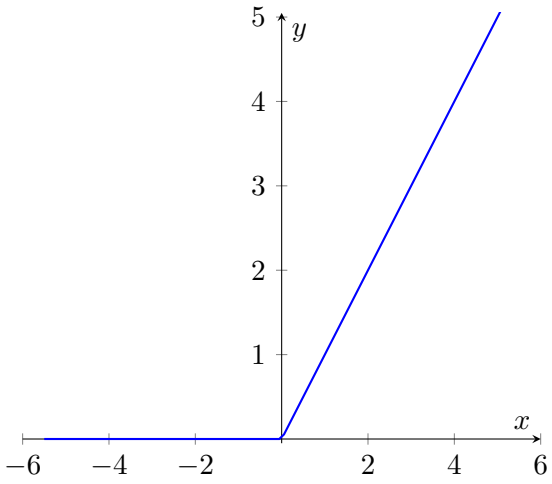

Figure 6: Graph of the ReLU function.

Note 4. We denote this function using ϕ here. Some references also use ψ.

As seen in the graph, the function cancels any input that is negative and passes in inputs that are positive unchanged. The functions range lies between $[0, \infty)$.

ReLU is the most used activation function right now and is seen quite often in Convolutional Neural Networks.

3. **Hyperbolic Tangent function**, or tanh :

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{8}$$

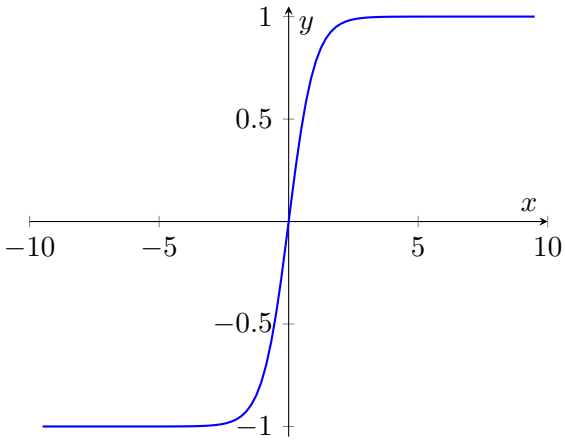

Figure 7: Graph of the Hyperbolic Tangent function

The benefit of using this function is that negative values are mapped to negative values and inputs close to zero are mapped near zero. The range of this function thus lies between $(-1, 1)$.

The tanh function is arguably a better version of the sigmoid function because of its ability to map negative values more accurately. This function is mainly used in classification models.

5. **Layers**
The first layer is called the **input layer**. This layer does not have an activation function (or you could say its activation function is the identity function) and the neurons in this layer give the values from the training data directly. Hence, their function as a layer is not the same as the rest of the layers.

The layer at the end of the network is called the **output layer** .This layer consists of neurons that contain the final values corresponding to the solution of the data provided. For example, if the network is designed to recognise a specific digit from an image, the neurons in the output layer will contain the final probabilities of all the digits with the correct digit most likely having the highest probability.

The rest of the inner layers are called **hidden layers** . These layers serve various functions and act as the core of the learning process in the network. Each layer will probably utilise a different function that narrows down the possible outputs from the given data and passes it on to the next layer.

6. **Hyperparameters**
Hyperparameters are certain variables that govern the efficiency and accuracy of the model. These parameters are continuously changed after the model has been built and tested to improve the model.

To run the model initially, the hyperparameters are given an estimated initial value. The number of neurons in a layer, the number of hidden layers, batch size, the learning rate are all examples of hyperparameters.

7. **Data**
The last component worth mentioning is the data set provided to the neural network. There are many datasets online that store a large number of images or text files pertaining to a certain theme or category.

Using machine learning compatible libraries like Tensorflow on Python allows these datasets to be easily accessed and modified based on the model's needs.

Once the data set is loaded into a program, to fit the model it needs to undergo some **pre-processing**.

The entire data set is usually divided into two groups: training data and test data.

- The **training data**, which would be a larger portion of the entire data set, is provided for the model to learn and develop a functional model.
- The **test data** is then used to check the accuracy of the model once it has learnt.

Most models work by checking how accurate their output is based on some reference output. These

reference outputs are usually provided by the datasets as well and are called **labels**.

For example, say a model is given an image from the **training data** showing the number 5. The model does some initial computation that is most likely inaccurate, compares its final output to the target output stored in the **training labels** (which should be 5) and sees that the error between the two outputs is not negligible. The model will then perform learning algorithms on itself to improve its answer.

1.3 Learning

Neural Networks employ a learning algorithm to change the values of the weights and biases so that the final output from the network can approximate the expected output for any input it receives. The most common method used is known as **Gradient Descent**.

To apply this method, we need to define a function dependent on the weights and biases, called the cost function. There are many forms of cost functions that are used. The most common one is the **Mean square Loss function :**

$$C(w,b) = \frac{1}{2n}\sum_{i=1}^{n} \|y(x_i) - \tilde{y}(x_i)\|^2 \tag{9}$$

where n represents the size of the training data, y represents the expected output, from the training labels and $\tilde{y}$ represents the approximate output, from the network. Also, w and b refer tothe collection of all weights and biases in the network.

Notice that if $C(w,b)$ comes close to zero, then the difference between y and $\tilde{y}$ also comes close to 0, thereby minimising the error.

This is the main idea of the learning process : **to minimise the cost function as a function of weights and biases.**

Note 5. This function is also known as a **quadratic cost function**. In general, The cost functions that are chosen for the learning process depend on the application of the model.

Gradient Descent Explanation

We want to find the change in C and find the points where $\Delta C < 0$: Let $\Delta\mathbf{u}$ be defined as the vector containing the change in variables, w and b such that

$$\Delta\mathbf{u} = (\Delta w, \Delta b)^T$$

$$\Delta C \approx \nabla C \cdot \Delta\mathbf{u}$$

We want this to be negative, and hence if we take

$$\Delta\mathbf{u} = -\eta\nabla C$$

$$\implies \Delta C \approx -\eta \|\nabla C\|^2$$

which guarantees that $\Delta C < 0$ as long as $\eta > 0$. η is a hyperparameter called the **learning rate** .

Note 6. Since gradient descent needs calculations of partial derivatives, when the cost function depends on multiple variables, this turns out to be computationally costly. This turns a few heads away from using this method for learning when a large number of variables are involved.

When encountering large number of inputs, like in our image example, finding the gradient with respect to each input and then averaging them to find the general gradient vector is very tedious. Instead, a method called **Stochastic gradient descent** is used to determine the gradient much faster.

Once the gradient is computed, we change the values of the weights and biases based on the step we took to minimize our cost function as

$$w \to w' = w - \eta \frac{\partial C}{\partial w} \tag{10}$$

$$b \to b' = b - \eta \frac{\partial C}{\partial b} \tag{11}$$

Note 7. These expressions will look slightly different under the stochastic process. Extra hyper-parameters like the **batch size** will also be present.

We calculate this for each weight and bias and once the values from every training input is computed, we redo the process to reduce the error even more. Each time this computation is done for the entire set is called an **epoch** of training.

1.4 Back propagation

We will begin analysing the idea of backpropagation intuitively and then look at the quantitative steps taken.

The idea of backpropagation is to analyse every single input from the training data, look at what output is required, and change the values of the weights and biases of the network a little in order to get the correct output.

For example, if we use a classification model where the output layer in the network contains 10 neurons, each indicating the probability of the input image being a corresponding number from 0 to 9.

Let an image of the digit 4 be an element from the training data be passed in as input. The output we expect is such that the neuron in the output layer representing the digit 4 has an output close to 1 and the other neurons would have output values close to 0. We start the process from this layer and look at all the connections each neuron has with the previous layer and work backwards.

Taking an example of the neuron representing the digit 4, we look at the connections it has to the neurons in the previous layer and observe which ones impacts the output of this neuron the most and which ones do not have a huge impact. Note that the output of the digit 4 neuron is computed in the form of a weighted sum of the outputs of the previous neurons added to a bias and passed into an activation function. Thus, there are 3 ways we can shift the values : by changing

the weights in proportion to the outputs, by changing the outputs in proportion to the weights or by changing the biases.

In this way, we find the amount we have to change the weights and biases relative to each separate neuron in the output layer and add up all these changes for each weight and bias. Then we can move on to the preceding layer and look at the connections for each neuron in this layer with its preceding layer.

Quantitative analysis : In order to know how much change needs to be applied on the weights and biases, we utilise the cost function.

Let us look at how the cost function is computed at the output layer L with respect to its previous layer, $L-1$: Let a_j^L denote the output of the j^{th} neuron in the layer L and y_j represents the target output we want it to be. Then, assuming we use the mean squared loss function, C_o computes the error in these values as :

$$C_o = \sum_{j=0}^{m-1} \left(a_j^L - y_j\right)^2 \tag{12}$$

where m represent the number of neurons in that layer.

Then we take its derivative with respect to the weight and its derivative with respect to the bias.

First, lets define some notations.

- Let w_{jk}^L to be the weight accompanying the connection between the k^{th} neuron in layer $L-1$ to the j^{th} neuron in layer L.
- Let b_j^L represent the bias accompanying the j^{th} neuron in layer L.
- To make it convenient, we can also define another variable, z_j^L that equals the weighted sum of the values and acts as an input to the j^{th} neuron's activation function. This variable is called the weighted input to the neuron.

$$z_j^L = \sum_{k=0}^{m} \left(w_{jk}^L a_k^{L-1} + b_j\right) \tag{13}$$

Then the output for the j^{th} neuron in layer L is

$$a_j^L = \sigma(z_j^L) \tag{14}$$

The derivative of the cost function with respect to each weight now becomes,

$$\frac{\partial C_o}{\partial w_{jk}^L} = (\frac{\partial C_o}{\partial a_j^L})(\frac{\partial a_j^L}{\partial z_j^L})(\frac{\partial z_j^L}{\partial w_{jk}^L})$$

We can similarly look at the impact of the outputs of the previous layer on the cost function by computing the derivative with respect to each output and summing all the values up :

$$\frac{\partial C_o}{\partial a_k^{L-1}} = \sum_{j=0}^{n-1} (\frac{\partial C_o}{\partial a_j^L})(\frac{\partial a_j^L}{\partial z_j^L})(\frac{\partial z_j^L}{\partial a_k^{L-1}})$$

Knowing these values we can finally apply gradient descent and effectively push the values of the weights, biases to produce an effective model.

1.5 Example Model

We can understand the working of a neural network by explaining the digit recognition example in more detail. But first, we will need to make ourselves familiar with libaries that can help us code the model, such as Tensorflow.

Tensorflow

Tensorflow is an open source library used in python to code machine learning applications easily. Tensorflow makes it easy to develop neural networks in just a few lines of code. There are many useful guides that can be used to understand the functions and variables used in this library on their website (5).

The most important parts can be described through the example below.

1. We first import the tensorflow library and load in the dataset of images (MNIST dataset) that store all the digits. The **Keras** library under tensorflow has a large number of datasets that can easily be accessed and modified.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

   Notice, the last two lines above segregate the images into training data and test data as well as make sure the pixel values of each element in the image lies between 0 and 1.

   This step is known as **preprocessing** and is crucial to make sure the data being input into the network is valid.

2. Next, we build the neural network with three layers : the input layer, one hidden layer and the output layer. This can be done using the **models.Sequential( )** function as our network is a sequence of layers, where data moves from one layer to the next.

   The input layer will contain as many neurons as there are pixels in the image it reads. Since inputs and outputs can be any real number between 0 and 1, we can use the gray scale (brightness) values of each pixel as the inputs, 1 signifying black (dark) and 0 signifying white (light).

In the case of the MNIST data, we have images of 28x28 dimensions. If we want each pixel to correlate to its own neuron we need to have a total of 784 neurons in the input layer.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

This is how we would define a rudimentary neural network.

- The first layer is a **Flatten** layer, used to decompose all the pixels in a 28x28 image as it is passed in. This makes our input layer.
- The next layer contains lesser neurons and is fully-connected to its previous and succeeding layer, giving it the name **Dense**. Notice that we are using a ReLU Activation function here.
- We can also see a **Dropout** function which sets a fraction of the input neurons to 0. This is used to prevent a phenomenon called **overfitting**.
- Finally, we have our 10 neurons in the final dense layer that store the probabilities of the image depicting a certain digit. This is our output layer.

Note 8. Overfitting is the phenomenon where the network has become extremely knowledgable in classifying or predicting on the data it was trained on, but is not useful on completely new, untrained data.

Most of the parameters here are **hyperparameters** that can be modified to make the predictions of the model more accurate.

3. We now have to train the model by giving it loss function and fitting the training data.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

- **model.compile( )** essentially provides the network a loss function and instructs it to perform a learning algorithm, such as backpropagation.The **optimizer** is another hyperparameter that is interlinked with the loss function (6).
- **model.fit( )** is the function that passes in the training data to learn from. The number of times it runs depends on the hyperparameter **epochs**.
- Finally, **model.evaluate( )** checks the accuracy of the model using the test data.

Another example of a classification neural network using the Fashion MNIST data is present on the Tensorflow website (7).

Algorithm

Python also allows you to manually create a neural network model including the definitions of its layers and loss function (8). Whether it be manually or using Tensorflow and keras, in general, the algorithm to define a neural network can be outlined as follows :

1. Define a neural network with n layers and initial sizes for the input, hidden and output layers.

2. Randomly initialise all the weights and biases and store them in the matrices, w and b.

3. Define the expression to compute the outputs according to the initial values as

$$\mathbf{y}' = \sigma(w\mathbf{y} + b) \tag{15}$$

where y represents the output values from the previous layer and y' is the outputs for the next layer.

4.1 Define the cost function.
4.2 Apply the learning process on the training data by setting up hyperparameters (mini-batches, epoch value, η value) to employ SGD. These values can be changed.

5. Provide some test data to check the model's accuracy.

6. If model needs to be improved, go to 4.2.

2 Recurrent Neural Networks

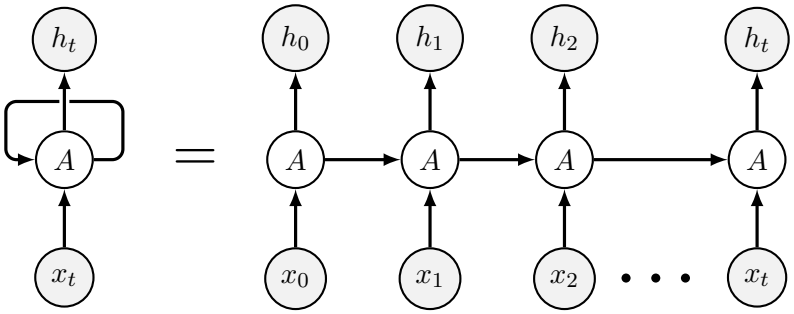

Figure 8: Depicts the compact (left) and unrolled (right) form of a recurrent neural network for t time-steps or t number of inputs in each input sequence.

RNNs are a derived form of artificial neural networks. The goal of an RNN is to implement **sequential modeling**, where either the inputs or the outputs are sequences of a system like time-steps in a simulation by containing an important state that holds memory of past steps, called the **hidden state**.

The figure depicts the general structure of an RNN where the left side is the unrolled form of the network for any time-step, t. The right side just unwraps/unrolls the model to show the network at every time-step. This structure is analogous to a regular network, where each time-step corresponds to a set of layers in the network : the input layer, hidden layer and output layer.

RNNs are mainly used in temporal problems, like natural language processing, speech recognition, etc.

2.1 General Properties

Sequential Data

1. The first important idea behind using RNNs is sequence data.

   Sequence data is a collection of information in a particular order that tells a story of a certain object or about a certain event. Taking multiple snapshots of a ball rolling on the ground is one such example.

   Using sequential data, it is possible to make predictions on the future of a certain object or an event based on the past information it stores. The snapshots of the rolling ball indicate the direction the ball is moving and hence, we can make predictions on the future position and velocity of this ball.

   In general, audio clips and even texts are forms of sequences used in RNNs.

2. The second important idea is that the sequential data we receive is encoded using an RNN and then eventually passed into a regular feed-forward neural network that will perform

classification or any other operation. Therefore, RNNs are usually paired up with ANNs or even CNNs (detailed in the next section).

Hidden States

The main objective of the RNN is to retain information that it processes as it continually receives input data. This means that as each input is passed in, the network stores some information regarding the input that it then passes to itself in when the next input is sent. This information is stored in what is called the **hidden state**.

To better understand this, take the example of a sentence : "*Is it snowing in Chicago ?*". This is a form of sequential data, and we pass this sentence, word-by-word, into the RNN as input.

Note 9. The term **time-step** is used to refer to the sequential passing in of data as input to an RNN. Therefore, for a sentence, each word being passed in as input corresponds to a time-step.

Now, "*Is*" is the first input to be passed in. The network computes certain weights and biases and creates an output. In order to retain the memory of this word for the second input, it also stores some data in its hidden state which acts as an input in the next **time-step**.

This process continues until it gets to the last word which takes in data from all the previous time-steps, thanks to the hidden state, and produces an optimal output. Since this final output is created from the rest of the sequence, it can just be passed into a feed-forward NN for further use.

An example of the implementation in Python is as shown :

```
#Functions containing the neural networks and their attributes
rnn = RNN()
ff = FeedForwardNN()

#Initial value of hidden state for the first input
hidden_state = [0.0,0.0,0.0,0.0]

for word in input:
    output, hidden_state = rnn(word, hidden_state)

#The final output is sent into the feedforward network for further use.
prediction = ff(output)
```

Some other key points to note in RNNs are :

- The model updates temporally, which means parameters are shared across every layer/time-step in the network. Thus, the learning process is a lot more convenient as the number of parameters to keep track of is quite small.
- An output can be produced after every output but depending on the problem, it is not necessary. The same thing applies for inputs as well. An input may not be needed at each time-step. The important component of the network is the hidden state.

Vanishing Gradient

One main drawback observed in a simple RNN is the vanishing gradient problem. In the example above, the input sentence only had a few words. What if we were to use an entire text file, comprising of multiple sentences and thousands of words. It is easy to realize that by the time the network reaches the 1000th time-step, it would have forgotten information from the first few time-steps.

To understand this in more detail, we can look at how backpropagation works once again. A regular neural network makes an initial prediction, compares it to the ground truth using a loss function and shifts the weights and biases of each layer based on the error from the loss. Therefore, the bigger the loss, the greater the shifting of the values in each layer. Adding to that, each neuron in a layer calculates its gradient based on the gradients of the neurons in the preceding layer. Hence, small shifts in the previous layer will lead to smaller shifts in the current layer. This causes an exponential shrinkage as the backpropagation continues down the network. The first few layers do not change much during the learning process and this can create inaccurate models.

With regards to an RNN, if we consider each time-step to be a layer, backpropagation essentially moves through time and the gradient values keep decreasing as we propagate backwards to the first time-step (9). In conclusion, RNNs have a hard time retaining information for long-periods of time.

Neurons with Recurrence

In the neural network structure, let us consider each set of inputs $\{x_1, x_2, ..., x_m\}$ be from a single time-step and the same for the outputs $\{y_1, y_2, ...y_n\}$. Thus, lets condense this information as vectors where at each time step we have an input vector, $x_t \in \mathbb{R}^m$ and output vector, $y_t \in \mathbb{R}^n$. Now we can look at multiple time-steps where the model is trained at each time-step for each set of input and output vectors. Overall, the output vector at any time-step is just a function of the input vector at that time-step : $y_t = f(x_t)$

However, it could be that a label (output) at a future time-step could be dependent on a feature/features (input) from a previous time-step. Thus, we incorporate an additional term, the **hidden state** that acts as a memory storage variable which is used to pass down information obtained from previous time-steps to the future ones. Thus, our output function becomes : $y_t = f(x_t, h_{t-1})$

Note 10. The hidden state goes by many names including the memory state or the self-state, to name a few.

In order to update this hidden-state at each time step, we use a recurrence relation (We can just take the initial hidden state to have 0 as its value) :

$$h_t = f_W(x_t, h_{t-1})$$

$$h_t = \tanh(\mathbf{W_{hh}} \cdot h_{t-1} + \mathbf{W_{xh}} \cdot x_t)$$

$$y_t = \mathbf{W_{hy}} \cdot h_t$$

where f_W denotes a function with W as its weights. This function can be applied just like the activation function used for the outputs, keeping in mind that there are two variables, x_t and h_{t-1} each having their own weights that have to be multiplied.

2.2 The LSTM Model

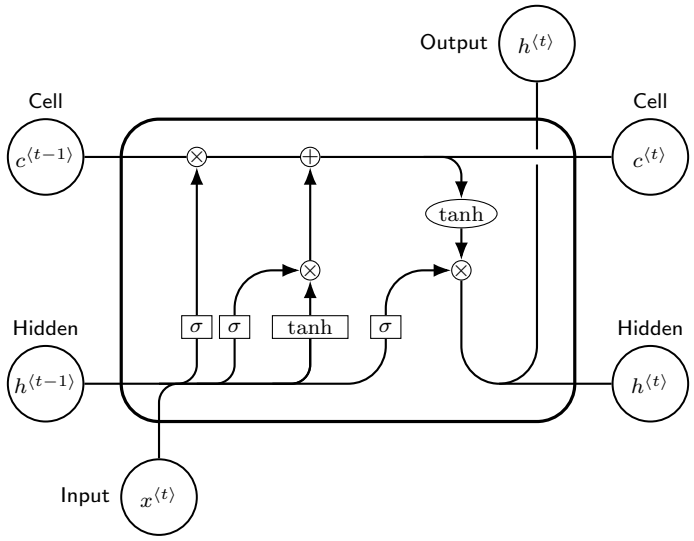

Figure 9: Internal structure of an LSTM cell

The major drawback of a recurrent neural network is that the hidden states cannot retain long term information due to the vanishing gradient problem.

The Long Short-Term Memory model (LSTM) was developed to tackle this drawback. It is a type of RNN that can handle long-term dependencies.

In general RNN structures, the repeating module will be composed of a single activation function that works on all the parameters and states. However, LSTM uses multiple functions and gates to retain information for a long time.

The most important feature in the LSTM model is the **cell state**, (the top line in the diagram). The cell state, denoted by C_t stores information based on the gates attached to it below. The gates determine whether information is passed to the cell state or not.

1. The first process is handled by the **forget gate**. This branch determines how much of the input information needs to be kept and how much needs to be discarded. The input information is the combination of the regular input data as well as the hidden state from the previous time-step. As seen in the LSTM figure above, the forget gate is usually controlled using a sigmoid function which will give an output between 0 and 1.

   We can analyse the forget gate layer numerically by considering an input, x_t, a hidden state, h_t and the corresponding weights associated with the connections of x_t and h_t to the node in the gate. The activation function is the sigmoid function and hence, the output is calculated as

   $$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f) = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{16}$$

   where f_t describes the output from the forget gate and the three parameters that are tracked are the two weight matrices and the bias.

2. The second step is to figure out what new information needs to be stored from this time-step on to the cell state.

   - The **input gate** is responsible for determining what values in our information need to be updated from the inputs.This gate is also controlled by a sigmoid function just like the forget gate.

     Thus, we get a similar equation

     $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{17}$$

   - Another layer controlled by a tanh activation function produces a new set of values, called as Candidate values that could be added to the cell state.

     $$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{18}$$

     where $\tilde{C}_t$ denotes the set of candidate values.

   - The candidate values are then multiplied with the output from the input gate to determine the final set of updated information that needs to be added to the cell state.

3. After all the information has been computed, it is simply added to the cell state. The output from the forget gate acts on the information of the cell state from the previous time-step and the input gate with the candidate values adds new information onto the cell state.

   $$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{19}$$

4. The last step is to compute the output and the hidden state for the given time-step. The output as well as the hidden state is basically determined from the information from the cell state.

   In most cases, since only the output of the final time step is required, there is no need for an extra gate or layer to separate the output we receive from the cell state into a hidden state and an output for that time-step. As seen in the figure, the h_t line branches into two, one that acts as an output for that time-step, the other acting as the hidden-state being passed on, both containing the same information, nonetheless.

   - An **output gate** is present that has a similar role as the input gate. It determines what part of the information from the cell state should be sent as output.

     Hence, our equation to compute the output from this gate is

     $$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{20}$$

   - The cell state is then passed through a tanh activation layer. This modifies the state's values to lie in the range $[-1, 1]$, after which it is multiplied with the computation from the output gate to get the final set of information for the hidden state.

     This is given by

     $$h_t = o_t * \tanh(C_t) \tag{21}$$

To summarise :

- All processing is done in the memory cell, governed by a cell state, represented by c_t
- The components h_{t-1} and h_t represent the outputs of the previous cell into the current cell and the outputs of the current cell to the next cell, respectively.
- There are three major gates that carry out the important processes in a time-step, namely, the forget gate, the input gate and the output gate.

This is how a general LSTM model works. However, there are many variants of the model have been implemented to solve different types of problems (11).

The major advantage of the LSTM, as mentioned before, is that it solves the vanishing gradient problem. During the computations in the cell, the only operations that are encountered during gradient descent are addition and multiplication. Since, these operations are linear, the LSTM model can ensure that between any two cells from two time-steps, the gradient is always 1. This prevents it from vanishing, allowing the model to retain information for much longer.

2.3 Learning

Since RNNs share the same weights and biases across the layers the backpropagation algorithm and gradient descent methods used are slightly different from the norm. RNNs employ Backpropagation Through Time algorithms (BPTT) to determine the gradients.

At each time-step we obtain a loss value based on the predicted output and the target output. After some t time-steps we sum up all the losses which gives us the Total Loss function that we will use to train the RNN.

2.4 Example Model

Let us look at an example of a recurrent neural network that uses the LSTM cell structure using Tensorflow.

The main outline of the network will be the same as the example from the first section :

- Load in a dataset and perform preprocessing, if required. We will use the IMDB dataset of movie reviews in this example.
- Creating the model, by adding multiple layers.
- Compiling the model, training the model and finally evaluating the model.

In this example, we will use the IMDB dataset that contain movie reviews in text format. The reviews have labels paired with them that depict a certain sentiment from the review : whether the review was positive or negative.

Our goal is to train a model such that if we pass in a review, it can correctly predict the sentiment of the review. To convert the idea of sentiments into numbers, we can just have one neuron in our output layer whose output will give us a probability value between 0 and 1. Any value greater than 0.5 will be considered a positive review and below 0.5 will be considered a negative review.

We start by importing all the necessary libraries.

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.layers import Embedding, Dense, LSTM
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.sequence import pad_sequences pad_sequences
```

The last line imports a padding function that is used to clip the input data, described later.

Next, we define some constants that determine our initial parameters like input size and number of distinct words. We want to make sure that all of our inputs have the same size to prevent any errors while training our model. We also want to make sure our training dataset is large enough to accurately train the model for any possible sentence. Thus, we make sure there are 5000 distinct words in this example.

We can then load in the dataset and split it into training and test data.

```
# Model configuration
num_dist_words = 5000 #Number of distinct words
seq_length_MAX = 200 #Max number of words in one sequence

#Loading the dataset
(x_train, y_train) , (x_test, y_test) = imdb.load_data(num_words = num_dist_words)

#Pad sequences to max length
x_train_padded = pad_sequences(x_train, maxlen = seq_length_MAX, value = 0.0)
x_test_padded = pad_sequence(x_tet, maxlen = seq_length_MAX, value = 0.0)
```

The padding section in the end is required to make sure our inputs have a fixed size. If the size is greater, we convert all values past the size to 0.

We can now create our model

```
#Defining the model
model = Sequential()
model.add(Embedding(num_dist_words, 15, input_length = seq_length_MAX))
model.add(LSTM(10))
model.add(Dense(1, activation = 'sigmoid'))
```

- Most machine learning models take in vectors of numbers as inputs. Thus, if the inputs we have are in text//string format it is important to come up with a method to convert them to numbers or vectorize the strings. This is done by the first layer, the **Embedding layer**. This layer essentially acts as the input layer. The embedding layer creates a fixed size vector corresponding to each word passed in as input. In the above example, each word will be

mapped to a vector of size 15. There are many ways of creating this embedding for the inputs and a few of these methods are detailed on the Tensorflow website (13).

- The second layer is our familiar LSTM layer. The number 10 in the code is the output dimensions from the LSTM layer which essentially reduces our dimensionality from $200 * 15$ corresponding to each input sequence down to 10. Remember that one input sequence gives us one entire moview review.
- The last layer takes the final output vector from the LSTM cell and passes it through one neuron with a sigmoid activation function that will output the right sentiment, a value ranging from $[0, 1]$.

The last layer is essentially its own feed-forward neural network that is being given a certain input and classifying the data. The recurrent processes in the model only occur in the LSTM cell.

The last few steps left are to compile, train and evaluate the model.

```
# Compile the model
model.compile(optimizer= Adam(), loss=BinaryCrossentropy(), metrics=['accuracy'])

#Summary of the model
model.summary()

#Train the model
history = model.fit(x_train_padded, y_train, batch_size = 128, epochs = 5, verbose
    = 1, validation_split = 0.20)

#Testing the model using evaluate
results = model.evaluate(x_test_padded, y_test)
```

The loss parameter is our cost function, used to make the model learn. Our chosen optimizer in this case is Adam (6). This section is largely similar to our first example in artificial neural networks.

3 Convolutional Neural Networks

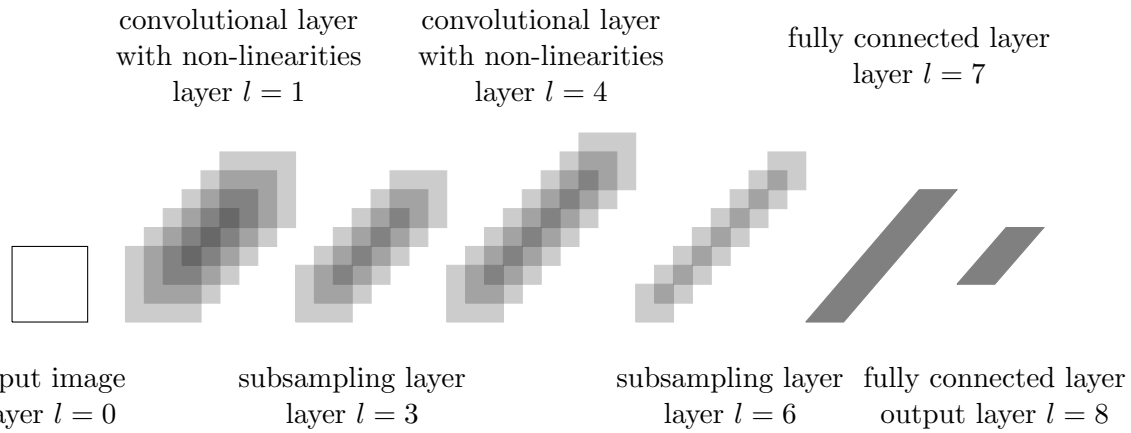

Figure 10: CNN showcasing alternating convolution and pooling operations and passing the result into a dense feed-forward network

CNNs are neural nets used to perform image classification or visualisation of images. The first thing to look at is why CNNs were developed in the first place.

When image classification is done using a dense neural network, it learns the features of the image in a global sense. It marks specific patterns that it finds in the image to certain locations of the image itself. For example, if we take an image of a cat, the network would identify features like the eyes and ears and mark them to be found in certain parts of the image. However, if the image were to be flipped and passed in as input, the network would have to relearn those features in the new areas. CNNs employ a different strategy. Rather than matching a certain feature to a certain area of the image, it learns what the feature itself is so that it can identify that feature in any part of any image. This makes a large difference when movies or multiple frames of images are provided as input, where certain patterns exist in each image but at different pixels or locations.

Using a fully connected neural network, the pixels of an image are flattened and their values are sent as inputs. Thus, many significant parts of the image cannot be identified. Thus, the idea of multiple convolutional layers present before the dense layer network is being used to figure out all the important features.

Note 11. Every image is a 2D-array of pixels with each element storing its grey scale value. If we are looking at a color image, it consists of three 2D-arrays of pixels, corresponding to each color - R,G,B. In some sense, color adds a depth dimension to each image.

The 3 main operations in a CNN are :
1. Convolution : Applying filters to generate feature maps
2. Using activation functions (RELU, Sigmoid)
3. Pooling : A downsampling operation on each feature map

3.1 Convolution and Filters

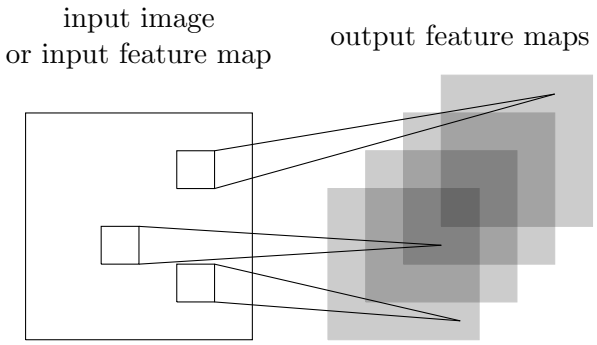

Figure 11: Sketch of the convolutional operation being used to sinle out certain features in an image and send it into a feature map.

The first step in a convolutional network is to create a bunch of patches on the images of fixed size that are passed in as input.

The reason we create these patches is to then apply filters onto these patches in the image. A filter is a matrix the same size as these patches that could depict a certain feature/pattern in an image, such as an "X" pattern or a diagonal line.

Then after performing an element wise multiplication between the image patches and the filter, we obtain a feature map which is also a 2D matrix.

A filter is just a small pattern/feature. For example, say we have a 5×5 image of an X. Let us take our filters to be 3×3 images that can narrow down certain parts of the image that seem to be a pattern, like straight lines. In this way, we pick a filter size and number of filters (usually 32 or 64) and use them to create a feature map corresponding to each filter. Another important parameter is the stride, which determines the amount of overlap between the patches during feature mapping.

The idea of stacking convolutional layers is to pick out more advanced features from the image. In this case, once we find our 32/64 feature maps from the original set of filters. We use another set of filters on these new feature maps which could identify better features like curves or rounded shapes. Similarly, we perform multiple filter operations on each layer of feature maps to find the best possible features of an image.

In most cases, after performing convolutional operation of the elements, we using an activation function in order to obtain the final output into our feature map for each of our patches.

Note 12. By default, filters in a convolutional layer in TensorFlow are given random weights.

3.2 Pooling

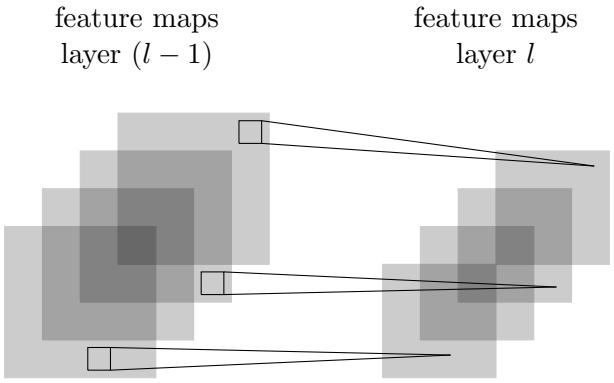

Figure 12: Depiction of pooling or downsampling to reduce the dimensionality of the image.

Clearly, the number of feature maps we obtain will be quite large and we need to reduce its dimension to make the computation more efficient.

Pooling also employs a similar technique to feature mapping. In this case, we use a smaller matrix size (usually 2×2 with stride of 2) to get a smaller output, but the operation used depends on the type of pooling. Min pooling takes the minimum value of all the elements in a patch, max pooling does the opposite and average pooling computes the average value of all pixels.

The final set of maps we get will have a smaller spatial dimension but larger depth, indicating a large number of features but of small size meaning they are well-defined. Overall, this gives us our convolutional base. We still need to classify images or do some form of processing to meet our end goal, and to do that we use a dense layer network on the final set of feature maps.

3.3 Example Model

We will now look at an example of a Convolutional Neural Network being used to perform image classification/ object detection using Tensorflow.

We start with the same outline as the previous examples and import all the necessary libraries and load in our dataset. In this example, we are using the CIFAR10 dataset that contains 60,000 color images of different objects such as planes, cars, deer, horses, etc. Our data is again split into the inputs and labels, where our labels will describe what object the input image represents.

```
import tensorflow as tf
from tensorflow.keras import datasets, models, layers
import matplotlib.pyplot as plt

#Load in dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.
    load_data()

# Normalize pixel values to be between 0 and 1
```

```
train_images, test_images = train_images / 255.0, test_images / 255.0

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer','dog', 'frog',
               'horse', 'ship', 'truck']
```

The most common layout to build our model is to stack a bunch of Convolutional and Pooling layers that can downsample our image to store the most important information. These layers together form the **convolutional base** of the network.

After this, we can add a few densely connected layers that carry out similar operations as a regular neural network.

```
#Convolutional base
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

- Each image in our training data will be of size 32×32 with 3 dimensions for each pixel to account for the RGB colors.
- To apply the convolution operation, we use 32 filters of size 3×3 with an activation function.
- For the downsampling operation, we use a 2×2 matrix that will takes 2 steps for its stride, by default.
- We repeat this process two more times, but with 64 filters instead of 32. This gives us a large set of strong features that can be used for classification.

Finally, we perform the familiar steps of compiling, training and evaluating the model.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    ,
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=4,
                    validation_data=(test_images, test_labels))

test_loss, test_acc = model.evaluate(test_images,  test_labels, ver
```

Overall, we obtain an accuracy of around 70%. However, a more accurate model can be obtained by tweaking the hyperparameters a bit more.

4 Neural Networks in Grain Growth

Microstructure evolution is an important aspect of studying grain growth and the relations between several properties of grains in materials. These evolutions can be modeled using coarse variable simulations of grains whose principles are usually described by partial differential equations. Recently, Neural networks are being used to replace the original methods of simulating the PDEs governing microstructure evolution.

Neural nets have the ability to learn the governing physical rules of the structure. Using convolutional recurrent networks, it is possible to train the network with image sequences containing simulations of common processes and accurately predict short term dynamics as well as long term statistical properties of microstructures. Using machine learning is a much more efficient method as the method it uses to learn the physical rules and laws do not need to be determined by us.

Throughout this section, we will be using the research done by Ming Tang and his group in the paper, "Self-supervised Learning and Prediction of Microstructure Evolution with Recurrent Neural Networks", as an example of how neural networks are currently being used in the study of grain growth and microstructure evolution.

4.1 What are Microstructures?

Microstructures are features of a material that act as a crucial link between the microscopic components and macroscopic properties. They have a direct impact on the processing-structure-property relationship of materials. Thus, the ability to understand and predict the evolution of microstructures is an important field of research in material science currently.

Microstructure evolution is often simulated using coarse variables that are modelled using partial differential equations. This method gets significantly harder to execute for materials that have not been studied enough. Identifying and validating the underlying PDEs takes a lot of effort as well. Using the black-box interface of neural networks, we can easily manipulate simulations of microstructure evolution to produce predictions of how the microstructure evolves in the next few instances of time as well as its properties at those instances, without any knowledge of the physical mechanisms required for the respective evolution.

4.2 Using Recurrent Neural Networks

Tang focuses on using Recurrent Neural Networks as the foundation of his machine learning model because of its ability to predict temporal data using its characteristic hidden states.

We have already seen how the LSTM variant of RNNs are used to store long term memory, but in recent years, there have been developments of hybrid forms of the LSTM model to include convolutional neural nets within itself. These models were designed specifically for predictive learning of spatiotemporal sequences. The advantage of incorporating CNNs is to extract important spatial features in the images of the simulations being passed as input while the LSTM segment

can focus on extracting features through the different time-steps. Thus, the two units collectively extract the most prominent features spatiall and temporally.

The specific model employed by Tang and his group is the Eidetic 3D LSTM model (16).

Once built, the network is rigorously tested to ascertain the accuracy of the model's predictions with that of the ground truth as well as evaluate the properties of the microstructure correctly.

The hardest task is to accumulate the required training data for the model. This would be in the form of simulations or explicit mathematical functions whose behavior is well known.

We will now look at two examples to describe the use of this network in evolution.

4.3 Examples

1. **Plane-wave propagation**

   This is the simplest example to test the RNN model. We use a scalar field, c, that is described by the following equation :

   $$c(x, y, t) = \frac{1}{2} \sin\left(k_x x + k_y y + \omega t + \theta_0\right) \exp\left(-\beta t\right) + \frac{1}{2} \tag{22}$$

   where (k_x, k_y) represent the wave vector $\vec{k}$, θ_0 is the phase of the wave (can be random) and β represents the decay constant.

   - Using this equation, we can generate a bunch of image sequences. The can be set up so that each input sequence consists of 200 images of the wave with a 0.005 second time interval between each image.
   - The other parameters are randomised based on the following intervals: $2\pi/|\vec{k}| \in [0.3, 0.6]$, $2\pi/\omega \in [0.03, 0.06]$, $2\pi/\beta \in [1.5, 6]$, $\theta_0 \in [0, 2\pi]$.
   - The entire dataset can then be split by half as training data and test data.
   - Finally, the output predicted by the RNN would be the next 50 images in a sequence based on an input of 10 consecutive images.
   - The accuracy of the model is clearly seen by comparing the predictions with the ground truth images. The mean squared error difference through pixel wise comparison of the two sets of images stays below 0.5%.
   - The structural properties such as the parameters in the wave equation are also predicted and the predictions of most of the parameters yield results that have an error of less than 2% from the ground truth.

   Overall, the model exhibits exemplary performance when tested using a simple plane wave propagation system.

2. **Grain Growth**

Grain growth refers to the increase in the average grain size of a polycrystal while the additional energy present at the boundaries is minimized.

- The most important property in grain growth is that, because of conservation of volume/area, if some grains increase in size, other grains have to decrease. This leads to a decrease in the total number of grains in a system over a period of time.
- The **von Neumann-Mullins** equation, also known as the 'N-6' rule is an equation that governs the growth or shrinkage rate of a grain in 2D polycrystals. It is given by :

$$\frac{dA}{dt} = M\gamma\frac{\pi}{3}(N-6) \tag{23}$$

  Here, A refers to the area of space taken by a single grain, M and γ are certain constants representing the grain boundary mobility and boundary energy respectively, and N is the number of sides present for the grain, a description of its shape.

  The main inference from this equation is that any grain that has lesser than 6 neighbours will end up shrinking and those with more than 6 will grow, both at a rate proportional to $(N-6)$.
- We can now use the power of the eidetic 3D LSTM model to simulate and predict certain properties of grains. Training data is generated by performing 2D grain growth simulations using a process called phase-field. This process creates images of 256×256 which are then reduced to images of 64×64 pixels by avergaing and downsampling. In total 2400 clips are used for training data and 600 are used to test. Each clip is the same as an image sequence consisting of 20 images/frames that are passed in during successive time-steps.
- The power of the neural network model is tested much more with this example as it is required to predict almost 200 images based on only one input image after finishing the learning process.
- The model doesn't fail to deliver with this example as well. When comparing the predictions to the ground truth, for small periods of time the two sets of images are difficult to distinguish but a divergence does show at later periods. The mean squared error between the two sets stabilizes to around 20% after 200 frames/images which is quite remarkable.
- Our next test is to put the eidetic model into spatial tests by making it predict grain growth with a much larger set than the training data. Because of the feature-extraction abilities present in the convolutional units in the model, the rules governing the evolution of grain boundaries are learned by the model even in larger sets without any additional training, since the physical laws stay the same despite the size of the data.
- As before a 256×256 grid of images is used that is downsampled to 64×64 images and produces similar results in accuracy and mean squared errors.

Overall, the two tests on the model showcase the ability of neural networks to generalize its learning into spatial and temporal domains. This is a strong indicator that the model itself learns the underlying laws of physics in its own way to predict the future of the grain.

4.4 Summary

With the two above examples, we have seen the power of recurrent and convolutional neural networks combined to solve spatiotemporal problems. The ability to store information for long terms through the LSTM variant of RNNs using its hidden states and the spatial extrapolation ability of convolutional layers allows complex features to be stored and identified by the model. These features would essentially contain the underlying physical laws and equations of the structure in hand and hence, the prediction of its evolution is possible to a high degree of accuracy. The biggest advantage of the whole process is in the fact that the parameters of the equations do not have to be determined at any point in time except, perhaps, initially as the model's learning process takes care of that.

References

1. Neural Networks and Deep Learning by Michael Nielsen.

2. Wikipedia : Artificial Neural Networks.

3. MIT 6.S191 : Introduction to Deep Learning

4. TensorFlow 2.0 Guide

5. Tensorflow Website

6. Optimizers

7. Image Classification Example Tensorflow

8. Image Classification Example Python

9. Illustrated Guide to Recurrent Neural Networks

10. Recurrent Neural Networks by IBM

11. Understanding LSTMs

12. LSTM Example Model

13. Word Embedding

14. https://github.com/davidstutz/latex-resources/blob/master/tikz-cnn/cnn.tex

15. CNN Image Classification Example

16. Eidetic 3D LSTM Model

17. Plane Wave propagation : Image generator

18. Self-supervised Learning and Prediction of Microstructure Evolution with Recurrent Neural Networks, Ming Tang, 2020.