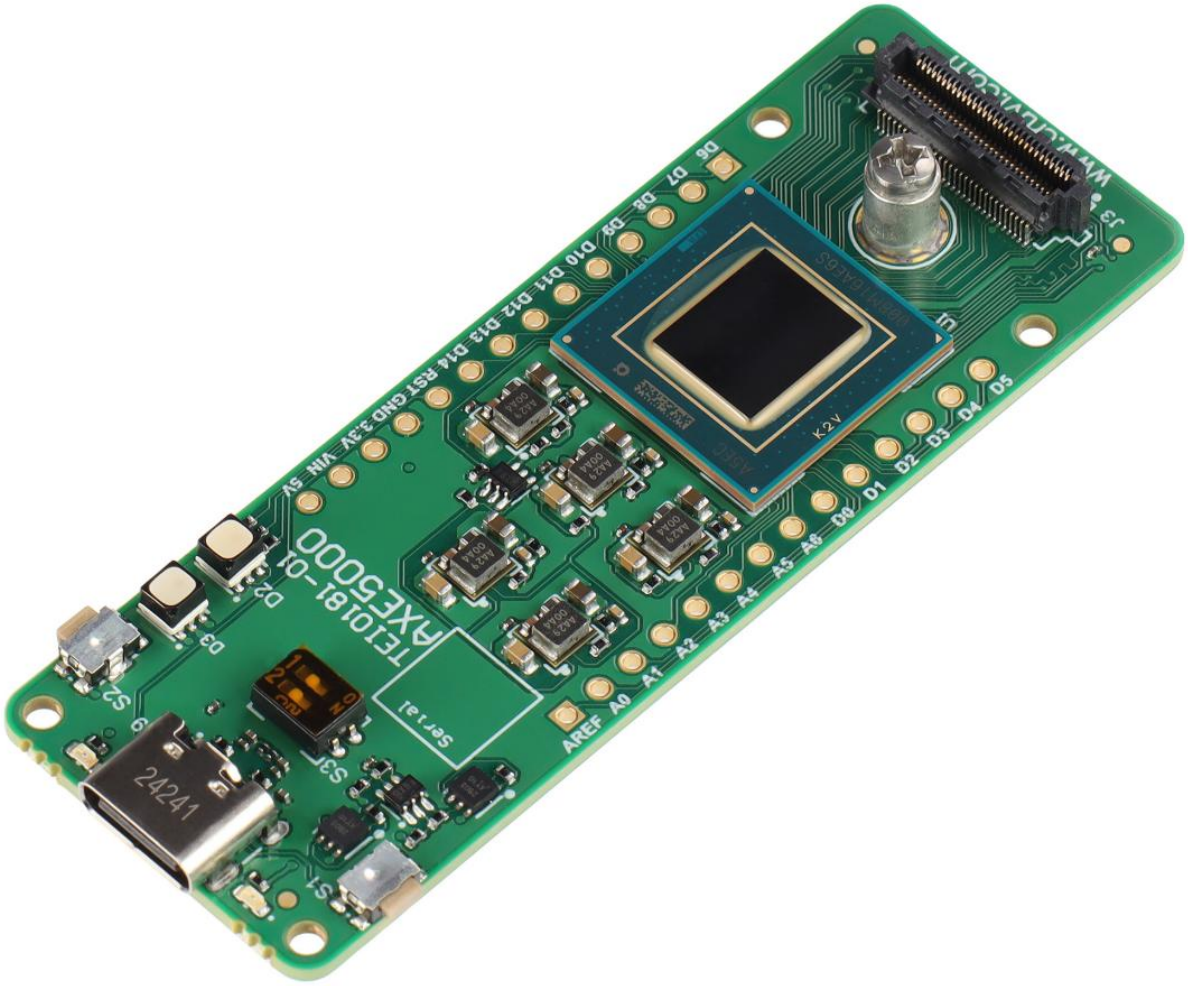
Introduction to AI



Software and hardware requirements to complete all exercises

Software Requirements: None

Hardware Requirements: None

Document Control

Document Version:	Version 1.1
Document Date:	19 / 08/2025
Document Author(s):	Steven Kravatsky, Erika Peter, Naji Naufel, ESC Team
Document Classification:	Released
Distribution of Document:	This document is still under development. All specifications, procedures, and processes described in this document are subject to change without prior notice
Prior Version History:	0.1

Table of Contents

1. AI for Altera FPGA

This background, which assumes no prior AI knowledge, covers the following:

- Background for AI
    - What is AI?
    - What is ML (machine learning) compared to other AI variations?
    - ML Learning vs Intelligence
    - Where is AI used?
    - Why look at AI?
    - How does AI work at a high level?
- Details on how CNN works
- 3D Visualization of CNN
- Details on how a model learns and modifies its own parameters
- Frameworks - what is it, and what are examples of this
- Datasets - where to find datasets
- Model errors - why do you see errors and how to resolve them?
- Why use FPGAs for AI deployment?
- Why look at Altera Agilex 3 FPGAs?
- Two Workshop streams - OneWare and FPGA AI Suite.

If you are familiar with the AI background material, you can “jump” to the hands-on workshop sections in the other documents.

2. Definitions

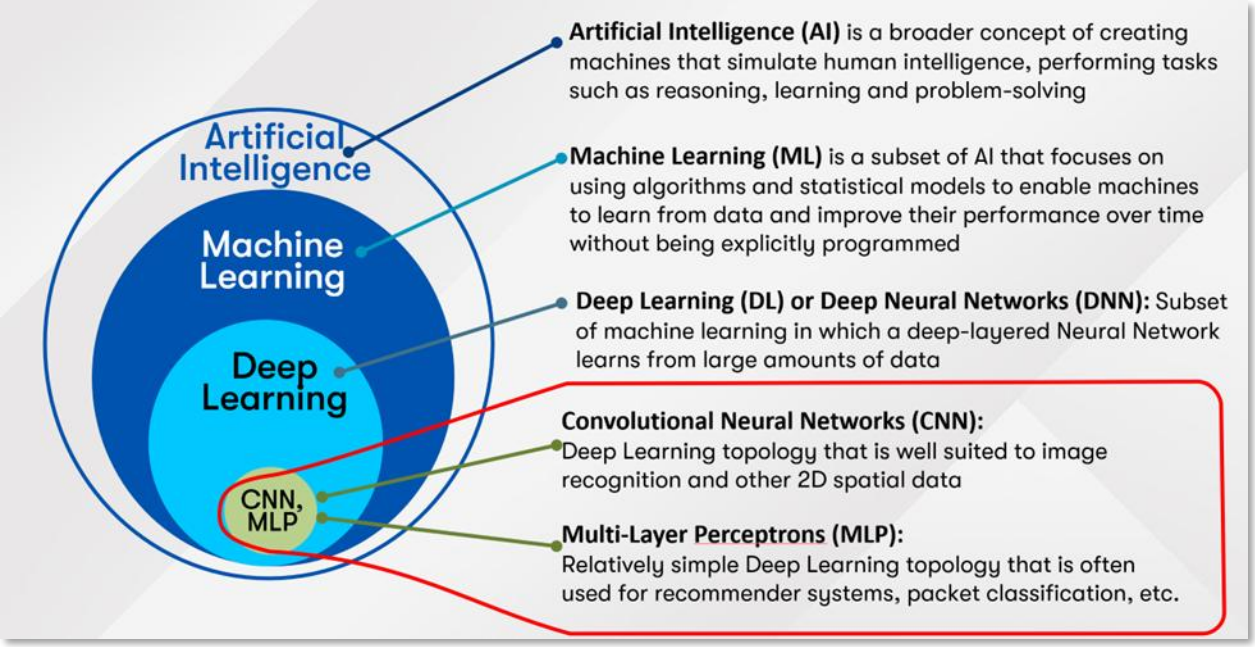
Abbreviation	Description
AI	Artificial intelligence: “any technique that computers use to mimic human intelligence -logic, decision trees, machine learning, etc.”
CNN	Convolutional neural networks. CNN is commonly used for image processing and other tasks as classification. CNN is a subcategory of feedforward neural networks
NN	Neural Networks. Building blocks for Deep Learning. Works like the neuron network within a human brain.
DL	Deep Learning. A subset of machine learning in which deep-layered Neural Networks learn from a large amount of data.
GNN	Graph Neural Networks. It uses graph data i.e., such as social networks.
LLM	Large Language Models.
LSTM	Long Short Term- Memory Models.
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology. A benchmark dataset used in machine learning and computer vision, especially for image classification tasks.
MLP	Multi-layer Preceptron: A subcategory of feedforward Neural Networks.
RNN	Recurrent Neural Networks.

3. Very high-level AI overview

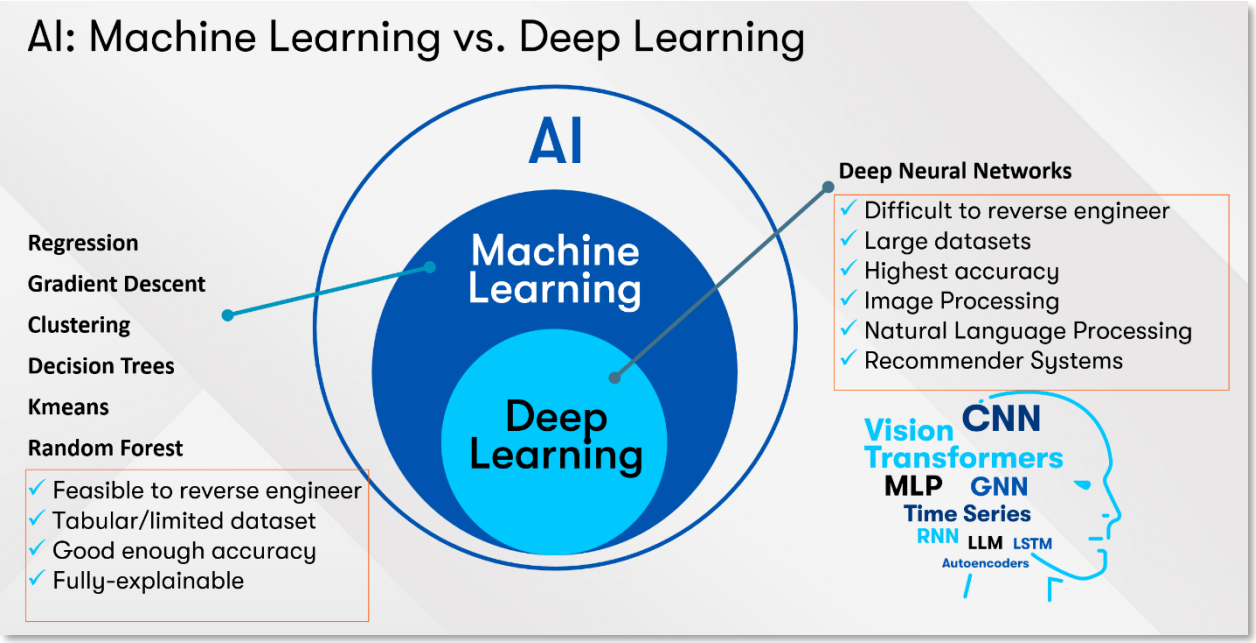
Artificial intelligence, also known as AI, is “any technique that computers use to mimic human intelligence -logic, decision trees, machine learning, etc.”

AI, which is a broad field, has been used for a long time. However, recently there have been vast AI improvements in terms of algorithms, much larger data sets and lower cost of hardware components.

Within AI, there is the sub-category of Machine Learning (ML). Machine Learning is a broad category that provides ways to automate learning. Deep learning is a sub-set of ML, where it uses neural networks to “learn” large datasets. Neural networks work similarly to how neuron networks function in a human brain. The focus of this workshop will be on CNN (convolutional neural networks) which is commonly used for image processing and other tasks as classification.



Deep Learning is very useful for doing complex processing such as image processing and natural language processing which would be very complex (and likely to be error prone) when implemented manually.



With Deep Learning, there are many sub-categories including *CNN*, *Vision Transformers* (where an image is calculated in "patches", which are then used to understand relationships), *GNN*, *RNN*, *LLM* and more. The sub-category used depends on the applications. One sub-category is CNN, which stands for convolutional neural networks. CNN will be covered in this workshop.

When an AI model reports a result, it reports probability and not exact numbers. For example, if the model states that it detects a car, it might be 90% likely to be a car, and 10% to be something else. AI can have errors. These errors will also be covered in this documentation and hands-on workshops.

Within AI, there are two major steps:

1. Training - which involves datasets, and
2. Inference - where the results of the trained model are deployed on a device.

Training and inference will be covered in this documentation and hands-on workshops.

4. Where is AI used?

AI is used in many different applications which include:

Education	Teacher Assistant	Student Study Buddy	Parent Chat Portal
Health	Drug Discovery	Doctor Co-pilot	Patient Family Chatbot
Finance	Algorithmic Trading	Customer Portfolio Assistant	Risk / Credit Assessment
Retail	Product Promotion	Customer Help Sentiment Tool	Image Shopping Aid
Government	Gov Services Chatbot	Document Search Summarization	Live Language Translation
Energy	Energy Consumption Forecasting	Operational Performance	Energy Trading Assistant
Automotive	Autonomous Car Development	Multi-language in car aid	Supply Chain Optimization
Manufacturing	Factory Automation	Predictive Maintenance	Precision Agriculture
Telco	Personalized Customer Services	Network Automation	Operational Performance

5. Why use AI?

Some reasons to use AI include:

- Increasing greater precision and accuracy for specific tasks. The most common applications are image processing and data processing tasks.
- Process and analyze data much faster than humans can.
- Efficiently and continuously do repetitive and time-consuming tasks
- Learn features from the dataset rather than the data being manually programmed.
    - Features (attributes) are characteristics of the dataset that the algorithmic model will extract and learn. Examples include gender, age, numbers, letters and more. Features found by DL can be even more lower-level than items such as shapes, edges, curves, lines, similarities and differences. As an example, for age estimation, it may find features such as wrinkle lines, grey hairs within colored hair, skin lines, etc An AI model identifies these features through mathematical processing, including the use of filters (where the filter values are called weights) before classifying or detecting the incoming data. These weights are adjusted when the model's mathematical output significantly deviates from the expected results (e.g., aiming for over 90% accuracy). By learning additional new features, increasing the dataset size, and performing more training, the model can generalize to recognize other new features.
- AI can achieve higher accuracy than humans for many tasks.
- AI can identify patterns or new connections that are difficult for humans to detect due to the sheer volume of data involved.

However, there are items to be concerned about when using AI, such as incorrect results, biased data, security concerns, privacy, and more. Some of these items will be covered in the following sections.

6. Security, Privacy and legal concerns of AI

When working with AI datasets or implementing AI functionality, there are concerns of security (i.e. what material is being used to train with), privacy, and legality. These concerns depend on what country or region that the AI is being used in.

Data security should be a concern due to the amount of data being used and where the data set(s) comes from.

Privacy should be a concern because AI uses large amounts of data from different sources. Europe, for example, has GDPR (General Data Protection Regulation) that deals with data privacy.

There can also be legal concerns when implementing AI in a design, such as potential liability issues if an accident occurs.

7. Learning vs Intelligence

Common questions about learning and intelligence in relation to ML include:

1) **Does ML "learn"?**

   Yes. Machine learning can "learn" because it improves its output results. It uses math-based algorithms to extract certain characteristics from incoming data. It modifies internal parameters as it compares the results to known correct data and makes corrections to the output.

2) **Is ML equal to intelligence?**

   No. Learning is not equal to intelligence. Intelligence has features such as understanding, reasoning, and self-awareness. Current ML systems do not understand details, do not contemplate results, and they lack self-awareness. Instead, ML excels at rapidly processing information much faster than humans can.

8. How does AI work at a very high level?

Before describing how AI works at a high level, let's focus first on how humans learn and then provide details on how machine learning works.

Let's assume that you are looking at a vehicle on the road.

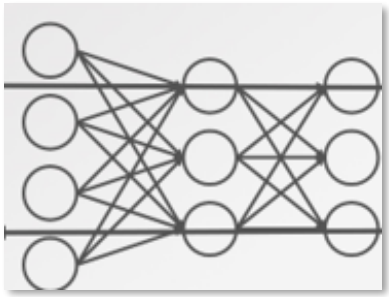
The question could be: How do you recognize that the image shown below is a car, if you have never seen this vehicle before?

There are certain characteristics that are common to a car i.e. it has wheels, doors, side windows, a license plate, and it is smaller than other vehicles, but larger than a bicycle.

You have been "trained" to automatically extract characteristics by seeing lots of items over time. These characteristics work for many objects, i.e. think of a house (has a front door, windows), zebra (has stripes), bear (think large brown or black animal), an ATM, etc. This means that even if you have never seen an object before, you would recognize the characteristics. For example, when you tell a child that an animal is a dog, you do not describe the various features of a dog (i.e. color of the fur, size of the dog, etc.). Then, when you show a child another dog (which may be larger or smaller than the first dog or even completely different) you would still state that it is a dog. If you see enough items over time, you are able to extract the features and generalize what the object is.

ML uses a similar process for learning, but it uses algorithms.

Algorithms are based on neural networks which are like how the human brain functions. A very simplified neural network diagram looks like:



The network includes inputs, processing (internal nodes) and outputs. The internal circles (nodes/neurons) are where the math calculations are done (i.e. filtering, pooling), while the black lines are where the data "travels". There are values (weights) that are associated with neurons. Weights, which are assigned during training, have probability values. The outputs are the sum of the neuron weights.

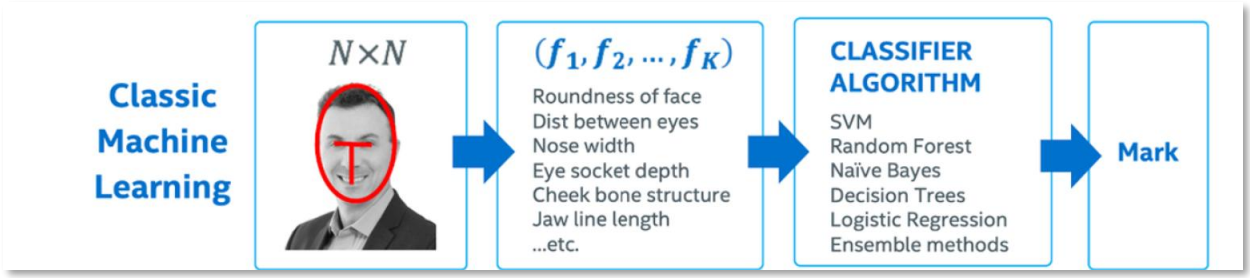
The specific importance of machine learning is that it can "learn" the math (filter) values rather than having to do detailed step by step programming. The internal nodes (which are known as hidden nodes) do the detection or classification of the object. The method of learning filter values will be described later in the document.

Detection and classification are not equal. Classification means that the object is assigned to specific categories and thus assigned a label. An example of classification is identifying a handwritten digit as the number 4, as seen below, where it is given a label of 4.

Detection is more complex because it involves classification and defining the coordinates (location) of the object shown in a bounding box. The example below shows that AI detected a parked car in a parking lot, and provided the location of it with a bounding box.

Having a machine "learn" the item means that it is much faster to "extract" features from data than doing manual programming.

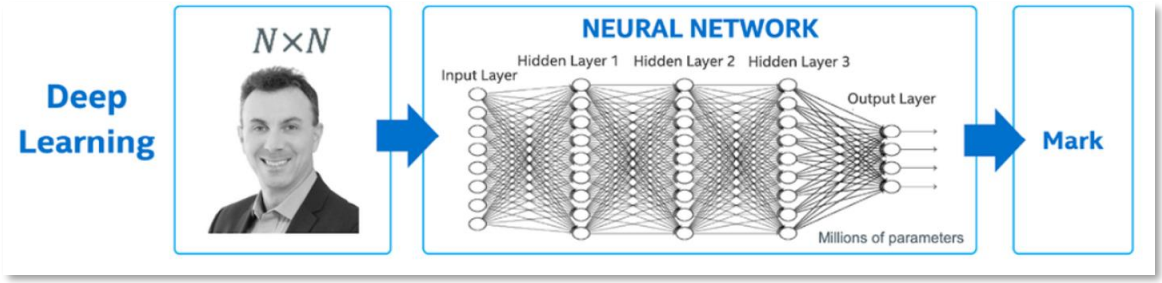
To better explain this, let's assume that you need to determine a person, Mark. If you used classic ML (the early version of ML where you need to manually program code), you would need to define carefully the facial features i.e. eye distance, nose width, etc. and then use classification algorithms such as random forest (which uses decision trees) to determine one person, Mark. Details of this can be seen as:



Some issues with this approach are:

- **Very limited in scope.** The results would also vary if the weather conditions were different, i.e. if it is cloudy, there will be shadows on the person. You would need many different photos of Mark in different settings, from different angles, with different lighting and shadows. (A NN unlike implementing this manually would learn via a dataset with variations what features are important and consistent even the face of the changes that are irrelevant)To identify another person, the same manual programming steps need to be re-done, so generalizing to another person would be difficult.
- **Time consuming.** All the parameters and algorithms need to be correct to even identify one person.
- **It can be prone to errors.** When trying to classify this for other people, this can be difficult.

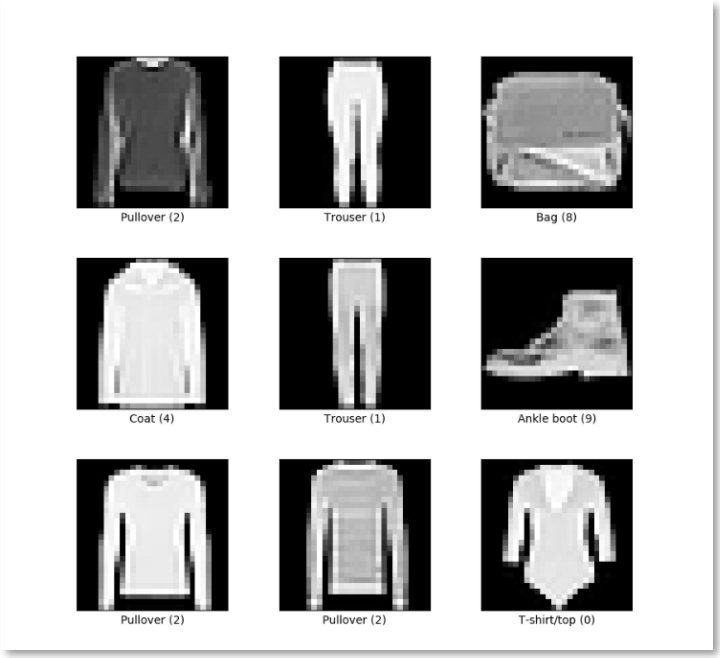
If deep learning (DL) is used instead, where the data itself can be used to determine to learn that the person is Mark. It "learns" this by analyzing large datasets. Each of the neurons in the neural network "work" on certain features of the data of the person, i.e. first extract edge features, determine facial features, and more. The hidden layers (internal nodes that are shown as circles) are where the data is worked on.



The DL could then report that it is 90% likely to be Mark, and 10% to be someone else.

A common question is: How can CNN learn features and how does it know how to correct errors?

CNN uses pre-trained datasets, where labels are assigned with the images. Let's assume, for example, that you use the Fashion MNIST (from TensorFlow software) where there are 10 categories of clothing such as 1 for Trouser, 2 for Pullover. Some of the images can be seen below. Note that each image is associated with a label, for example, Pullover (2). Labels are not used during the actual neural network calculations; they are used to compare the predicted outputs to the correct outputs. In machine learning, levels are represented as numbers of words, as numerical values make it easier to perform mathematical operations than using text.



Let's assume after the CNN processing is complete, the model incorrectly predicted that a pullover (2) was a trouser (1). Predicting means the model will use the largest value in the result as the output. The error is identified as incorrect by a function called *loss comparison*. It does loss comparison by comparing the labels to the label name of the output result. Since the result is incorrect, the model parameters will be changed somewhat to try to get the correct result, i.e. closer to the actual result. This requires an iterative approach of the model to get the image to match the correct label. Models, as you will see in the hands-on portion of this workshop will run many times through the data.

9. Overall flow for implementing AI

Overall flow for AI is as follows:

1) Design requirements: What will be implemented?
2) Select a Network Topology (category) which will affect the framework (software).
3) Train the network with considerable data to determine the parameters (weighing).
4) Use software or toolkit to convert the trained model algorithm to create inference files.
5) Inference (deployment) that runs the design locally on a device.

These steps are iterative until the correct results are achieved. Details on these steps are described below.

Step 1 Design requirements

The first step for any design is to clearly and concisely define design requirements. For example, when looking at using AI for vision applications, a very important requirement is whether the design will use object classification or object detection. This requirement will then determine in turn how the design will be implemented.

There are differences between object classification (also known as annotation) and object detection.

- **Object classification.** Identify what the item is, but not where it's located. It assigns the object to a specific class. For example, in single digit handwriting recognition for digits 0 to 9, the model can classify the input into one of 10 possible “buckets” (outputs) representing 0 through 9.
    - When the buckets are more than two, this is considered multi-class classification rather than a binary output (e.g., true/false). To determine which bucket and output belongs to, the model selects the class with the highest probability. For example, if the probability for the number 4 is 0.8, for the number 1 is 0.2, and for all other numbers (0, 2, 3, 5, 6, 7, 8, 9) is 0.0, the model will classify the output as belonging to the bucket (class) of 4.
- **Object detection.** Determines both what the object is and where it is located. For example, it can detect a person, animal, or other objects and specify its position within the image. Object detection is more complex than object classification because it must identify the object’s class and determine its precise location.

- For example, for this vehicle, the object detection detected an object, put a bounding box around it (for the location) and that it is 69.6% certain of the object.

Step 2: Network Topology (Graph)

Depending on what is being implemented, the next step is to choose a network topology.

A network topology defines

- how “neurons” are connected,
- the number of layers, which determines how many features will be extracted from the data,
- connections of the layers and more.

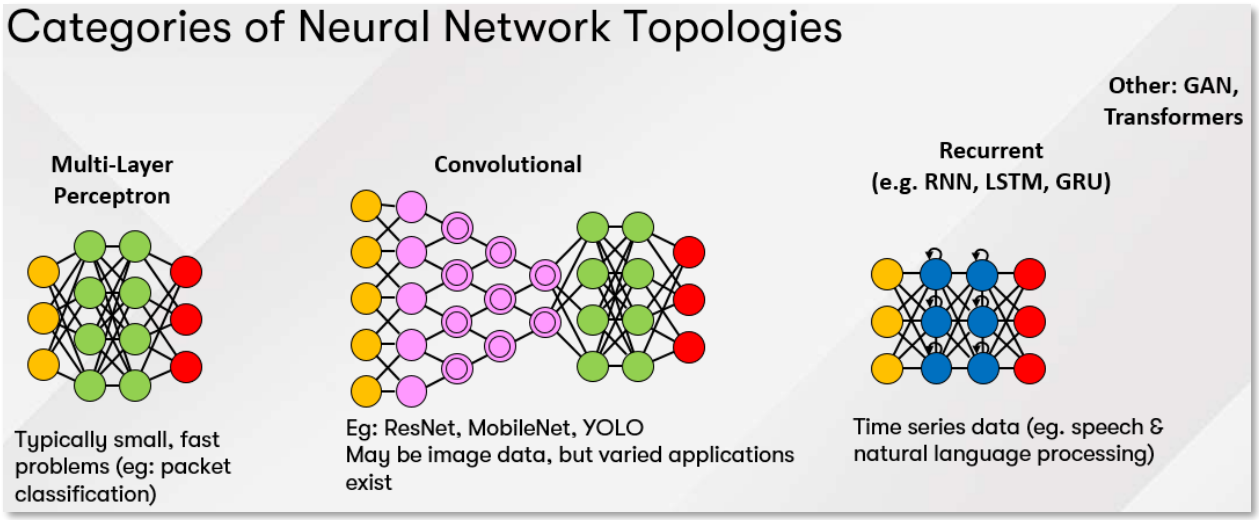
Different topologies are used for different applications. For example,

a) **Image processing and classification.** CNN is a common topology. This one will be covered in this workshop.

b) **Network packet classification.** MLP topology.

c) **Speech and natural language processing**. Recurrent topology.

Here are some more details of the topologies.

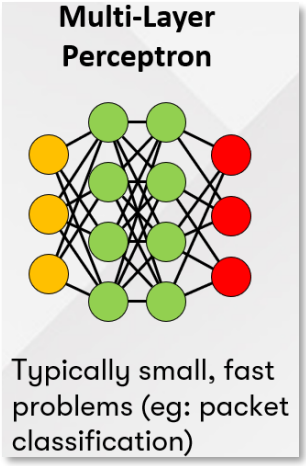


where the nodes are defined as follows:

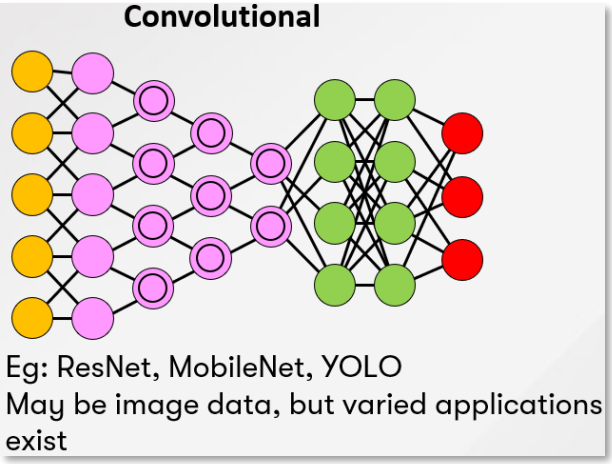
Some details for each topology are:

Multi-layer Preceptors (MLP)

MLP, which is a simple neural network, is used for such applications as packet classification (i.e. packets in network traffic). It is used for small, fast problems. Note in MLP, the neurons are fully connected, which means that yellow "neurons" connect to all the green "neurons." The yellow neurons represent the inputs (3 of them), the green neurons are the hidden layers (i.e. the "calculation layers"), and the red neurons are the 3 outputs.



Convolutional Neural Networks (CNN)



CNN, which is a commonly used neural network, is often used for image recognition and classification. CNN, which is known as feed forward, is where neural networks are connected to each other going left to right, and there is some connection to each other on the same level. More details on CNN will be explained in a later section.

In the diagram, the input layers are the yellow "neurons". The hidden layers (i.e. the "calculation layers") are the purple "neurons". Purple neurons could be convolutional and pooling layers (further sections in this workshop will describe this more). These calculation layers are where the feature extraction occurs. The hidden layer ("classification section") are the green neurons. These green neurons, which are fully connected layers, is where the classification occurs. The output layer is represented by the red neurons.

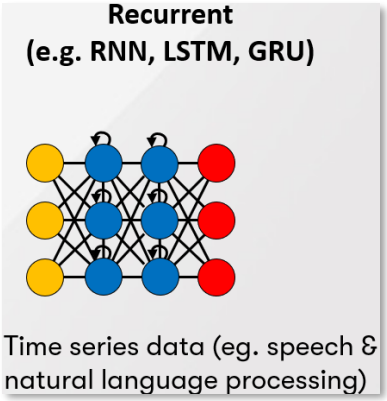
Note how the green neurons are connected to each other (which means fully connected) while the other neurons (orange, purple and red) are connected left to right.

The purple nodes and green neurons are where the math calculations (algorithms) are implemented, while the black lines represent where the "data" travels.

Within CNN, there are various network sub-categories such as Alexnet (released in 2012), GoogleNet (released in 2014), Vgg16 (16 layers), ResNet-8 (8 layers) ResNet-18 (18 layers deep), ResNet-50 (50 layers deep) , YOLO (you only look once) and more. Having more layers i.e. 50 layers vs 8 layers means that more features ("attributes") can be extracted ("pulled out") from the data. Extracting more features can be necessary for more complex tasks (such as classifying into more categories.

Recurrent neural Networks (RNN)

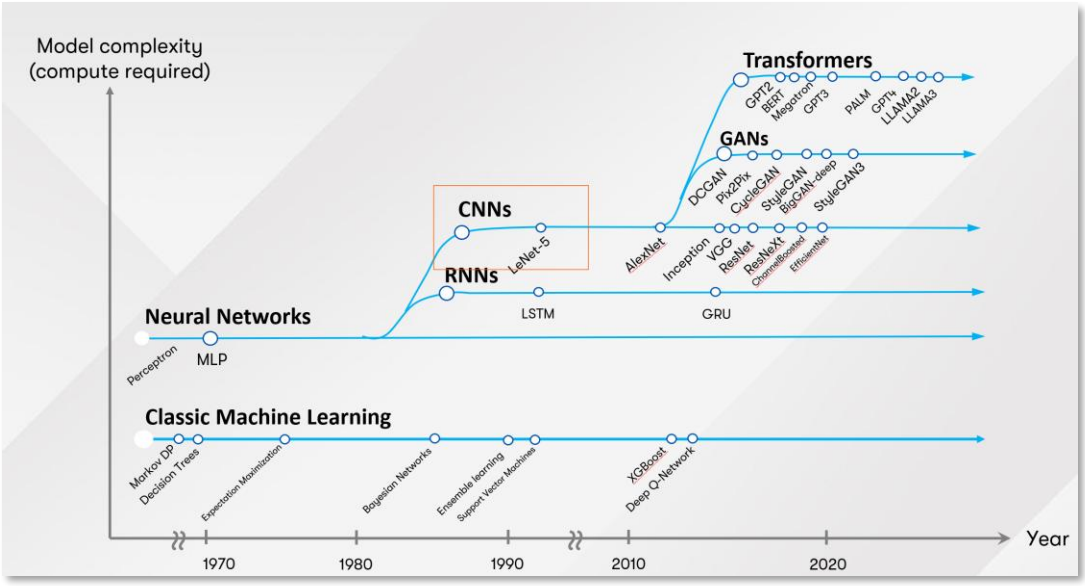
RNN is a Neural Network commonly used for speech and natural language. RNN feeds back onto itself, because it is used for temporal data (time series data). It is like CNN, but it can predict via memory what will follow next like a human does. For example, for RNN, it is memory of previous words such as "how are you" where it can predict that the next word could be "doing"



The focus of this document is on CNN, which is commonly used for imaging applications.

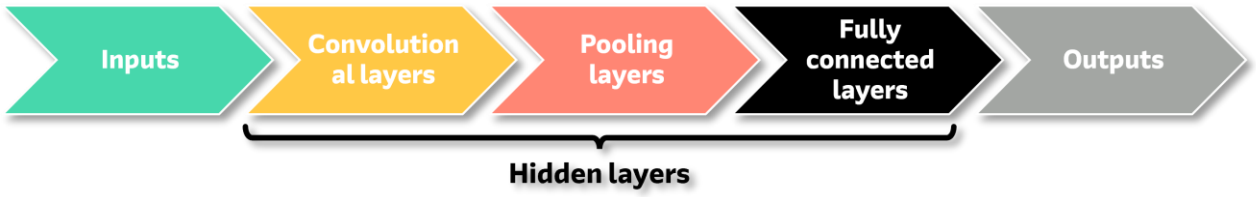
Step 2a. Details of how the CNN (Convolutional Neuron Network) topology works

There are many CNN variations that have been developed over time. One early, simpler, model variation of CNN is LeNet-5. LeNet-5 is used for basic handwriting (letters and numbers).



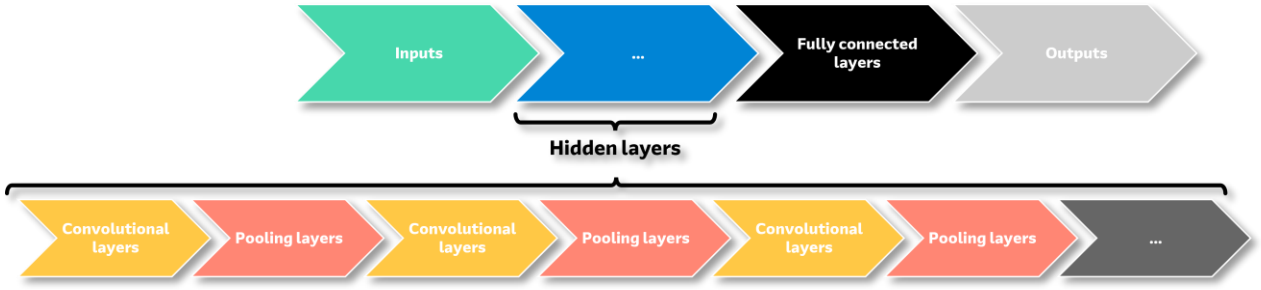
This section will provide more details on how CNN works to provide background information for the workshop steps.

At a high level, CNN (commonly used for image processing) consists of multiple sections: inputs, convolutional layers, pooling layers, fully connected layers and outputs. An easy way to visualize each of these stages is through the diagram that illustrates the flow from the input image, through the processing steps ("hidden layers"), to the final output.



Hidden layers are where the algorithms (calculations) are implemented.

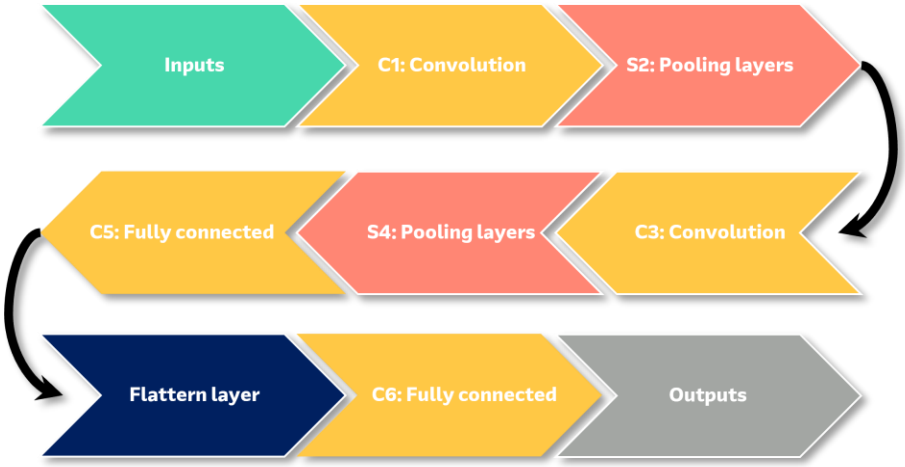
The above diagram shows a high-level block of the sections. However, in actual implementations, there are many convolutional and pooling layers that connect to various fully connected layers as shown below.



Convolutional layers and pooling layers do the feature extraction, while the fully connected layers do the classification.

The number of hidden layers varies depending on the features that need to be extracted. The more layers, the more features can be extracted. 50 layers will extract more features than 8 layers, and more extracted features can result in better accuracy. With multiple layers, the first layers can extract general features such as edge detection, horizontal lines, vertical lines, while later layers can extract more specific features

For the simple CNN model, LeNet-5, which can be used for handwriting digits from 0 to 9, the structure is:



There are two convolution layers, two pooling layers, and two fully connected layers. The following sections will describe the blocks one by one.

Inputs

Inputs can be a dataset of still images or live images. It could be images of cars, people, animals, buildings, vehicles and much more. Examples of an image could be:

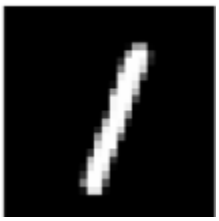
a snake

a car

4 in black and white

For images that are in color, it is very common to use RGB (red, green and blue), while for numbers in black and white, the greyscale image is used.

In this step, the inputs are converted to digital values that the model can use. As an example, let's assume that the number 1 from the MNIST dataset is used for an input.



A dataset, which contains this number 1, needs to be preprocessed before it can be used. Preprocessing of the input data involves converting it into "better" (easier to use) format for ML calculations. Preprocessing includes such features as re-shaping (where numbers are put into a 4D array that is used in CNN array), normalizing values (dividing by the number 255 so that the numbers in further mathematical steps will be smaller) and more. In this step, for example, let's focus on normalization.

The value of 1 has pixel values that range between 0 (white) and 255 (black). Grey scale means that there is no red, green or blue but rather the color ranges between white to black. Note some pixel numbers are less than 255, which means that the pixels are not completely black.

$$\begin{pmatrix} 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 230 & 0 & 255 \\ 255 & 225 & 0 & 255 & 255 \\ 255 & 0 & 230 & 255 & 255 \\ 255 & 220 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 \end{pmatrix}$$

After dividing each number by 255, this results in values that range from 0 to 1 including fractions.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0.9 & 0 & 1 \\ 1 & 0.88 & 0 & 1 & 1 \\ 1 & 0 & 0.9 & 1 & 1 \\ 1 & 0.86 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Convolutional Layers

Convolution is where the filtering is used to extract features from the image. This is the learning stage of the model. Convolution reduces the amount of data that needs to be processed but still retains important features.

Mathematically, convolution refers to multiplying two different matrix sizes (along with addition) creating a new resultant matrix. The first matrix is the data, while the second matrix is a filter (e.g., a 3×3 or 5×5 matrix).

For fine details of image processing, a 3×3 filter may have advantages over 5 x 5 filters, as it is easier to extract more precision details and faster convolutional calculations because there are less values. The 3 x 3 filter contains 9 values while a 5 x 5 filter has 25 values. Larger filter sizes as 5 x5 (or 7 x7) , on the other hand, can provide better overall coverage of large images and they are often used for the first layer of many CNN models. Larger kernels can help reduce the size of image faster than with smaller kernels.

The numbers within the filters (also called kernels) are the *weights*. Weights, also known as *learnable parameters,* are values that the model adjusts during training to minimize the difference between predicted and actual results. By tuning these weights, the model strengthens the connection between specific neurons, improving its ability to recognize patterns and make accurate predictions.

An example of 3 x3 matrix filter could be the following:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

There are different kernel values for horizontal edge detection, vertical edge detection, sharpening of the image, etc. What is important in CNN is that the filter values (weights) are **learned** during the machine learning process rather than having **pre-determined** filter values. Having ML learn these kernel values enables the model to be flexible.

The filters move across the image, i.e. left to right and up to down, to extract features such as edge detection (look at the edge of an image), sharpen (focus) the image, etc.

Edge detection is important, because it helps determine the object outlines, space between object and even the background. Once the edge detection is complete, other features such as patterns and lines can be extracted.

It is very common to have multiple convolutional layers in a model. More layers means that it can extract more features in the images.

After the convolution step is complete, an activation function can be applied to the matrix in a bitwise manner. Activation introduces non-linearity, enabling the model to handle complex, nonlinear patterns in the data. Without activation, the model would behave only linearly. Since real-world data is often non-linear, activation functions are essential. One common example is *ReLU* (Rectified Linear Unit), which outputs zero for any negative input. When the output is zero, the corresponding neurons are inactive, making it easier for the model to focus on detecting relevant features.

For example, let's assume that the output of the convolution is

$$\begin{pmatrix} -1 & 2 & -3 \\ 4 & -5 & -6 \\ -7 & -8 & 9 \end{pmatrix}$$

When ReLU is applied, all negative values are set to zero, while the other values remain positive to help with non-linearity, as shown below:

$$\begin{pmatrix} 0 & 2 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

The following section provides more details on the convolution mathematics.

For example, let us assume the image represents a snake, and its pixel dimensions are 244×224:

CNN would work with each color channel (layer) separately. It would be 3 channels like:

- one channel would be 224×224 pixels for the red component,
- second channel/layer would be 224×224 pixels for the green component, and
- last channel/layer would be 224×224 pixels for the blue component.

The total number of 224×244 pixels x 3 color channels = 150,528, which is a large number. A large number means that a considerable amount of processing time is needed before the image features can be extracted and then classified before the object can be determined.

Convolution is a technique used to reduce computing time, while extracting features from data. To illustrate this, let's take an example and apply convolution using a 3×3 data filter.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Simplified data Data filter

Let's focus first on the items highlighted in the blue from the simplified data and then convoluted them with the kernel (data filter).

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Simplified data Data filter

Convolution calculations multiply the data in the blue color going left to right and top to bottom with the values in the data filter. The calculations are:

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 = 5$$

The first number in each multiplication step is the items highlighted in blue, while the second number in each multiplication is from the data filter.

Note that the result of the previous operation, which is equal to **5**, will replace the value at the center of the 3×3 blue pixel square periods, where the resulting matrix will appear as follows:

$$\begin{pmatrix} & 5 & & 1 & 1 \\ & & & 0 & 1 \\ & & & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The data filter will then move across other pixels in the data, sweeping from left to right and up and down, continuing the convolution process (as shown in the blue section) below.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Simplified data Data filter

Convolutional math would again follow the same process of multiplication and addition of the highlighted blue numbers convoluted with the kernel (data filter) to provide the value of 5 as seen:

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 1 + 1 \times 1 + 0 \times 0 + 1 \times 1 = 5$$

$$\begin{pmatrix} & 5 & 5 & 1 \\ & & & 1 \\ & & & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

NOTE: For reference, reaching a corner where there is no complete 3×3 block to multiply with the data filter, you can use padding (adding "bits" such as zeros around the data) so the

multiplication can proceed correctly with the same vector size. This process continues until all values are filled out.

The next step is to apply activation. If *ReLU* is used, negative numbers are replaced with zeros, while positive numbers remain unchanged.

In this example, there are no negative numbers, since all values in both the data and the kernel are positive, so the numbers remain as positive.

Pooling Layer

Pooling layers (down-sampling) provide a method to decrease the size of the convolution feature map (to reduce the computation needed) while still retaining its most critical features. There is no learning at this stage.

Pooling, which extracts the strongest features, removes the less important features including noise. Pooling makes the model more robust due to it helping with generalization.

During pooling a matrix (i.e. a 2×2 filter) is moved over the convolutional layer matrix data. There are different ways to do pooling. One pooling method is looking for the maximum value to extract the strongest features.

Let's look at an example to illustrate this.

If pooling is applied using a 2×2 filter (with a stride of 1) and the data values are as follows [A stride, for reference, refers to the number of pixels the filter moves each time, in this case 1 pixel]

$$\begin{pmatrix} 6 & 5 & 4 & 3 \\ 5 & 2 & 5 & 1 \\ 3 & 4 & 2 & 5 \\ 4 & 1 & 4 & 6 \end{pmatrix}$$

When pooling is complete, the highest value (features) in each colored section would be

$$\begin{pmatrix} 6 & 5 \\ 4 & 6 \end{pmatrix}$$

Note how the 4 x 4 matrix (16 values) are reduced to a much smaller 2 x 2 matrix, which enables faster mathematical computation.

Flatten Layer

This layer acts as a converter, where it concatenates the multi-dimensional arrays from the CNN (and pooling) stages and converts it to one-dimensional array that the fully connected layers require as an input.

For example, if the array is:

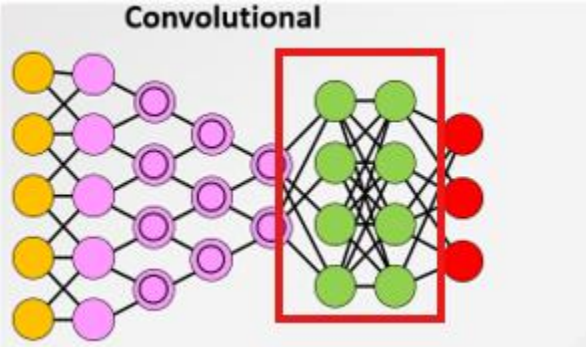
$$\begin{pmatrix} 6 & 5 & 3 \\ 4 & 6 & 0 \\ 1 & 2 & 3 \end{pmatrix}$$

the 1D matrix would be changed as:

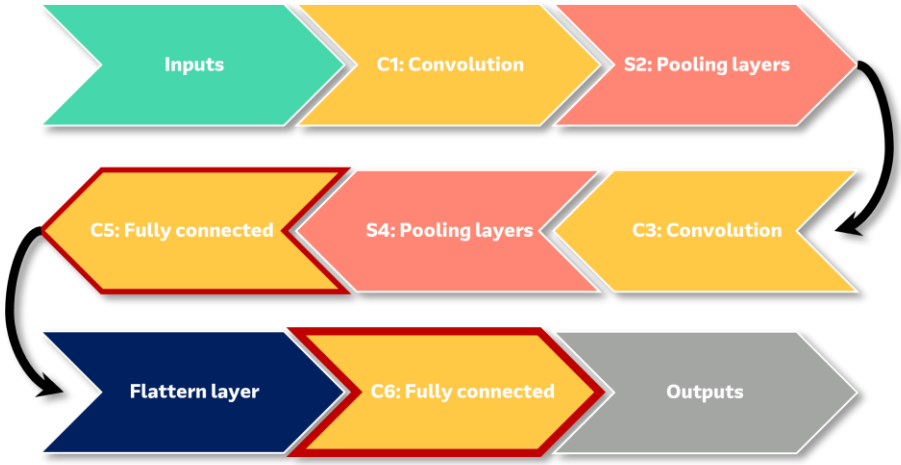
$$\begin{pmatrix} 6 \\ 5 \\ 3 \\ 4 \\ 6 \\ 0 \\ 1 \\ 2 \\ 3 \end{pmatrix}$$

Classification: Fully Connected Layers

Fully Connected (FC) layers, also known as dense layers, are layers in which every neuron is connected to every neuron in the previous layer. This stage is responsible for classification or object detection, ensuring that the outputs are generated accurately. This stage looks like:



For a model, there can be one or more fully connected layers. In the LeNeT-5 model, for example, there are 2 fully connected layers as



The first FC layer has more neurons and more connections than the second one. However, the math for FC layers remains the same.

For an FC layer for LeNeT-5, the 1D output of the first pooling layer contains 400 values that will connect to 120 NNs. Each of the connections between the 400 values will have weights and biases.

What is bias?

Bias is similar to a weight, because its value changes during the learning process to produce more accurate results. In addition, bias is used so that even when the input value is zero, the model can still learn features, improving accuracy for complex patterns. For example, in the equation for a line $\boldsymbol{y = mx + b}$, the term b represents bias. This means that y will still have the value b even if the $m \times x$ equals zero.

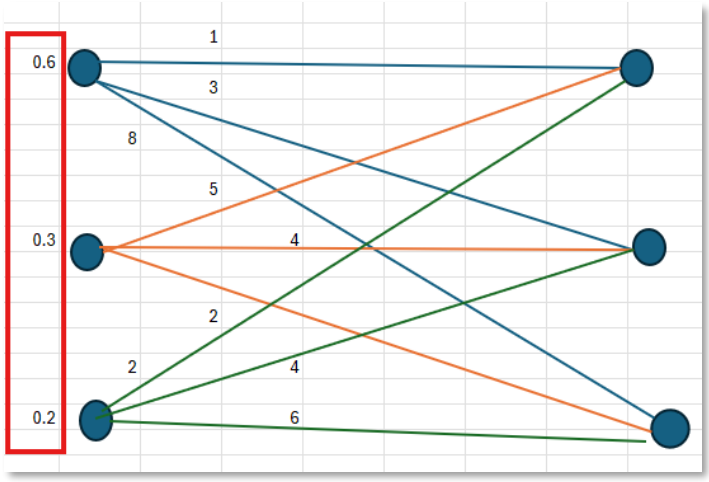
If we use the equation of $y = mx + b$ and use m for the number of values in the 1D array, x for the number of NNs and $b =$ bias, the number of parameters (connections) will be $(400 \times 120) + 120 = 48{,}120$.

Since the number 48,120 is quite large, let's use a much smaller example to illustrate how math in a fully connected layer works. For example, if the values from the 1D input array from the pooling layer are:

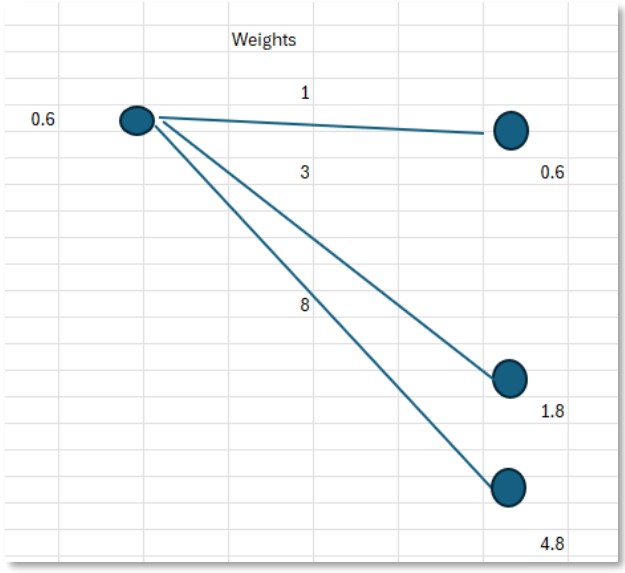
$$\begin{pmatrix} 0.6 \\ 0.3 \\ 0.2 \end{pmatrix}$$

The fully connected layers can be seen below.

The values on the left-hand side (in the red box) are from the pooling layer. The blue, orange and green lines are used to visually represent the different connections. Each neural network on the left-hand side is connected to each neural network on the right-hand side, i.e. the blue line from the top left NN connects to each of the 3 NN on the right-hand side.



Now, let's examine each connection separately, starting with the blue connection to more clearly understand the math involved.

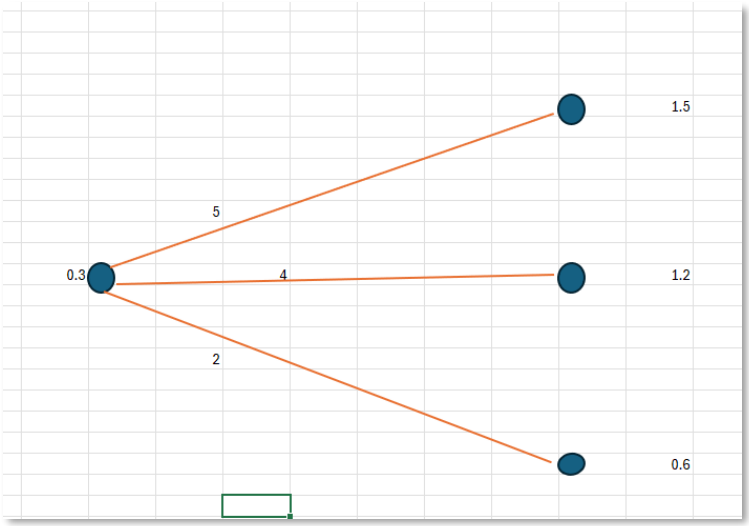


The output nodes multiply the input pooling value NN multiplied by weight:

- $0.6 \times 1 = 0.6$
- $0.6 \times 3 = 1.8$

- $0.6 \times 8 = 4.8$.

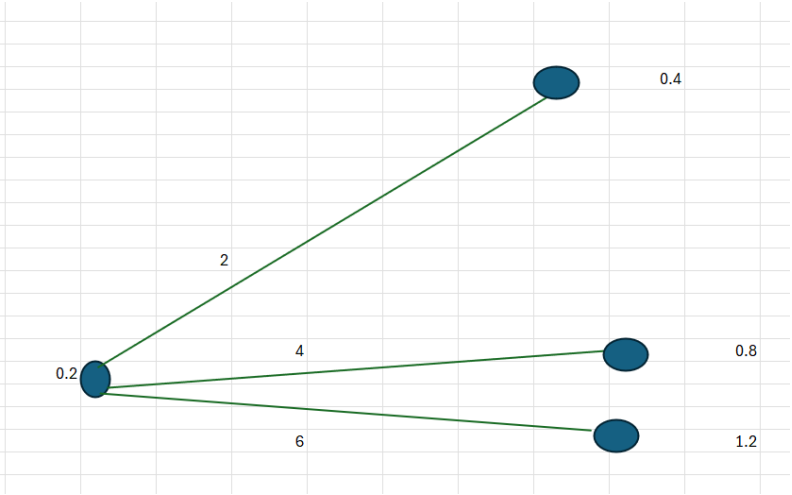
The math for the orange connection is similar:



Where it is also NN multiplied by weight:

- $0.3 \times 5 = 1.5$,
- $0.3 \times 1.4 = 1.2$
- $0.3 \times 2 = 0.6$

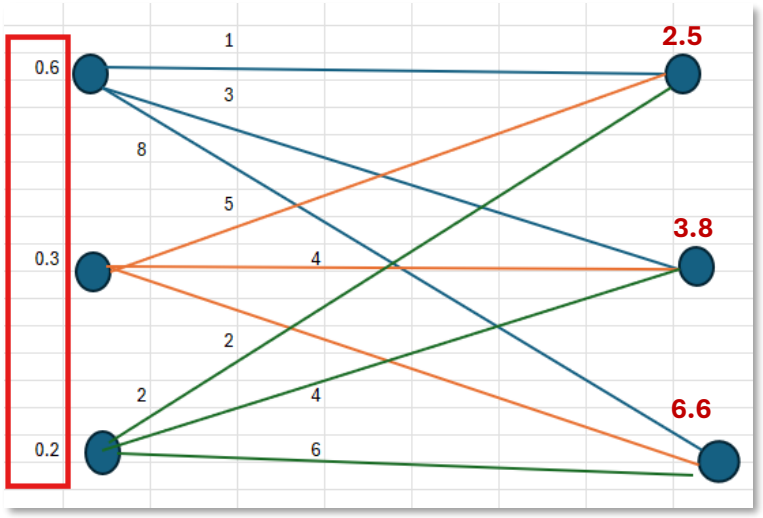
For the third connection, the math process is similar:



Remember, each NN value is multiplied by its corresponding weight:

- $0.2 \times 2 = 0.4$
- $0.2 \times 4 = 0.8$
- $0.2 \times 0.6 = 1.2.$

Now, the value for each NN output needs to be summed together

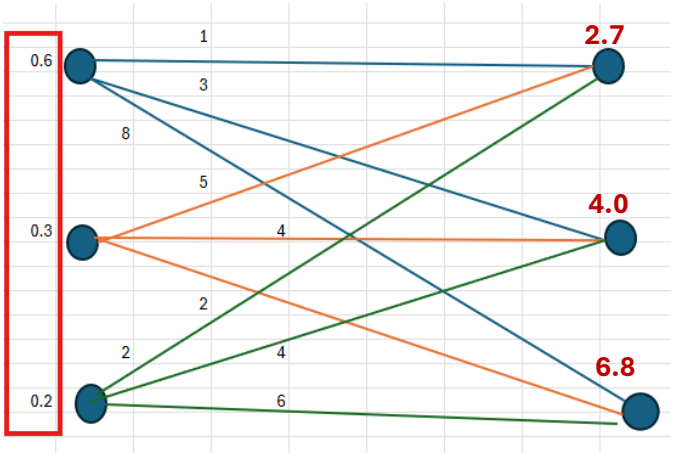


For example, the value for the top output NN equals 2.2 is broken down as 0.6 (total from the blue connection) + 1.5 (from the orange connection) + 0.4 (from the green connection). The value for 2.2 is then the total.

The calculations for the other neurons are

- $1.8 + 1.2 + 0.8 = 3.8$
- $4.8 + 0.6 + 1.2 = 6.6$

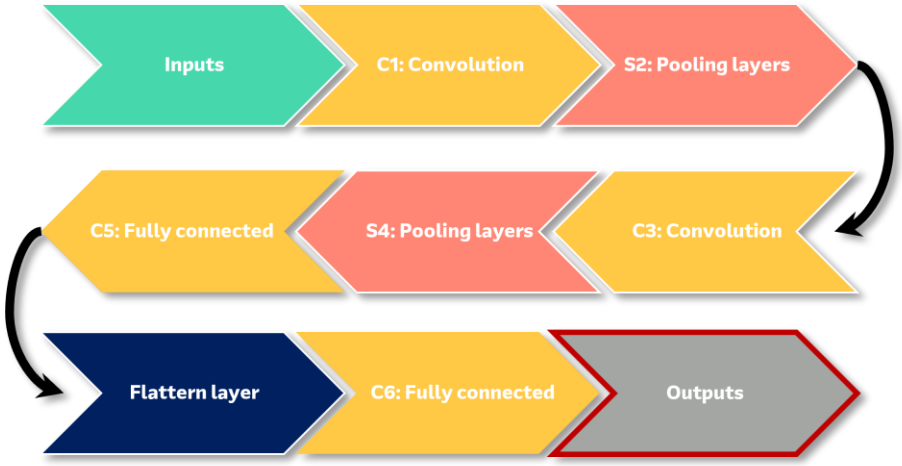
The value that is now missing is bias (to avoid having a calculation of zero). Let's assume that the bias is 0.2, which means that every neural network (output) has an addition of 0.2 more.



These output values are then passed to the output layer, which would identify the highest value to determine the percentage likelihood of the item.

Output

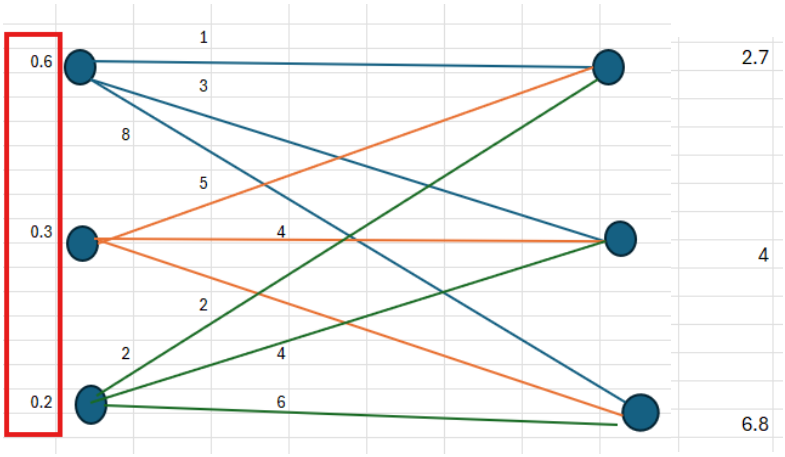
The last stage for the model is the output stage, as seen for the LeNet-5 model.



At this stage, an activation function is used to produce the final output. One common option is SoftMax, which converts the results into probabilities.

SoftMax works by summing the total of all node values in the last fully connected layer, then dividing each node's value by this total to determine its probability.

As an example, the output of the FC layer was 2.7, 4.0 and 6.8. Therefore, the total is $2.7 + 4.0 + 6.8 = 13.5$



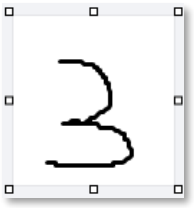
This means that

- $2.7 \div 13.5 = 0.2$,
- $4.0 \div 13.5 = 0.29$,
- $6.8 \div 13.5 = 0.50$.

The total number of SoftMax is one (ie adding 0.2+0.29+0.50 (there is some rounding), so 6.8 would be 50% likely to be the output.

Now for LeNet-5 model, where there are 10 classes ("output buckets") for the digits of 0 to 9, the output will show the percentage likelihood of what number (s) that it could be.

Let's assume that you entered the input number as:



The model would do similar calculations as per the above to determine the totals.

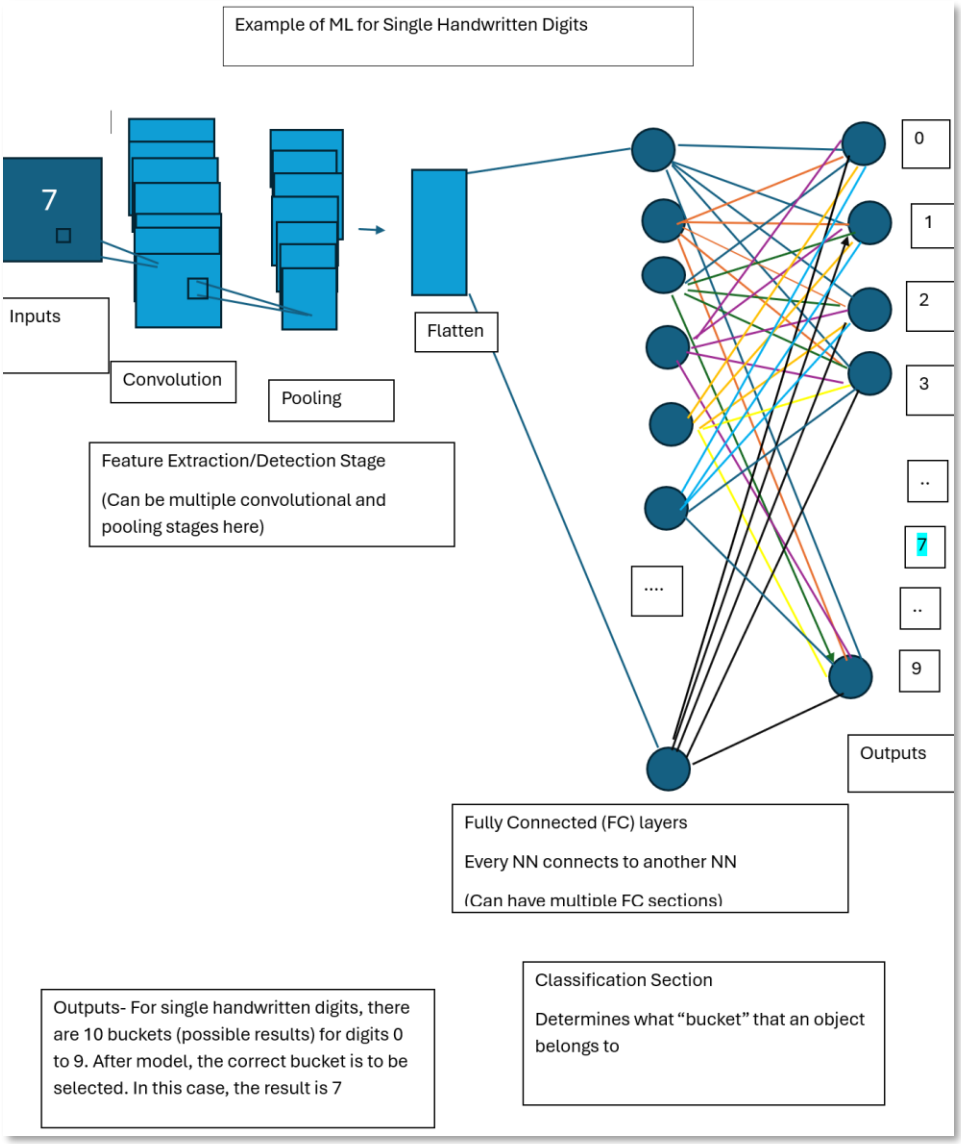
For example, the SoftMax outputs are:

Output "buckets" for the digits 0 to 9	0	1	2	3	4	5	6	7	8	9
SoftMax	0	0	0.1	0.7	0	0	0	0	0.2	0

It shows that there is 0.7 (70%) chance that it is a number 3, while it is 0.2 (20%) chance that it is 8 and 0.1 (10%) that it is a 2. The model might interpret the shape of the number 3 as the number 8.

Example of a Single Handwritten Digit CNN Model

As an example, the connections of the above various building blocks using a handwritten digit recognition can be seen below as a full diagram. Note the flow from the inputs → convolution → pooling → flatten → fully connected (dense) → outputs. Convolution and Pooling stages do the feature extraction/detection. Fully connected layers do the classification i.e. select the correct output.



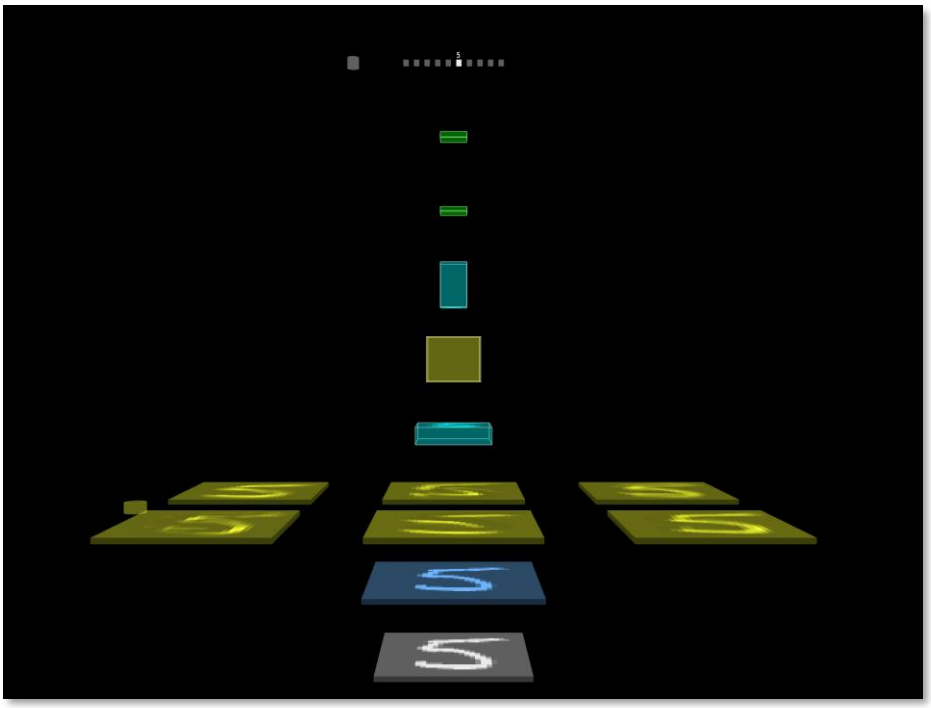
3D Visualization of a LeNet-5 model

Since it may not be easy to visualize the LeNet-5 building blocks, the next step will be to interactively experiment with LeNet-5 model in a 3D visualizer.

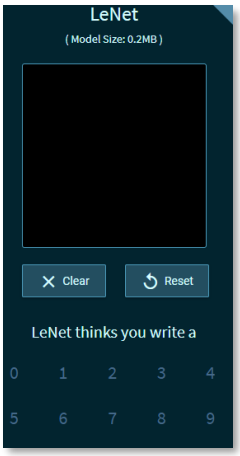
Go to the website of TensorSpace Playground - LeNet. This website does not need a log in.

NOTE: If you are working remotely on CloudLabs, and you do not have access to the internet, in CloudLabs which can be the case, you just **open another browser on your local computer,** and you can go to this website.

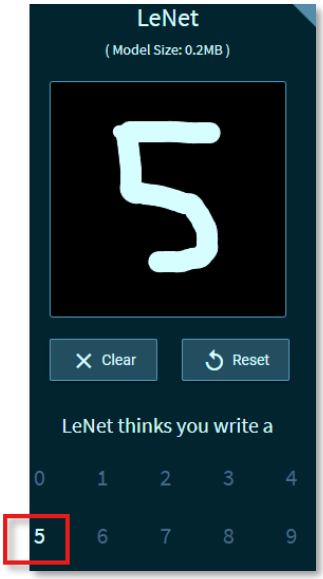
You will see the default view as:



There will be a window to the top right that shows

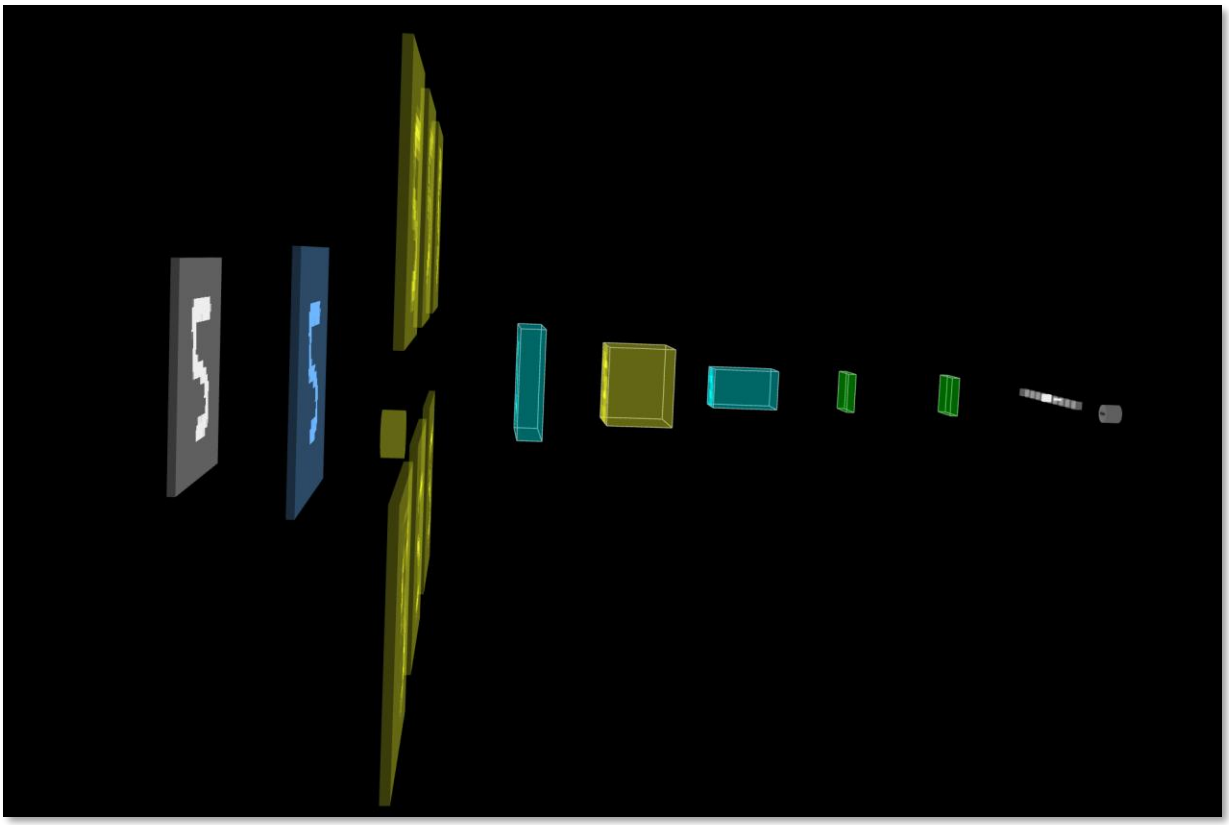


Enter a value such the number of 5 by drawing the number 5 with your mouse as follows:



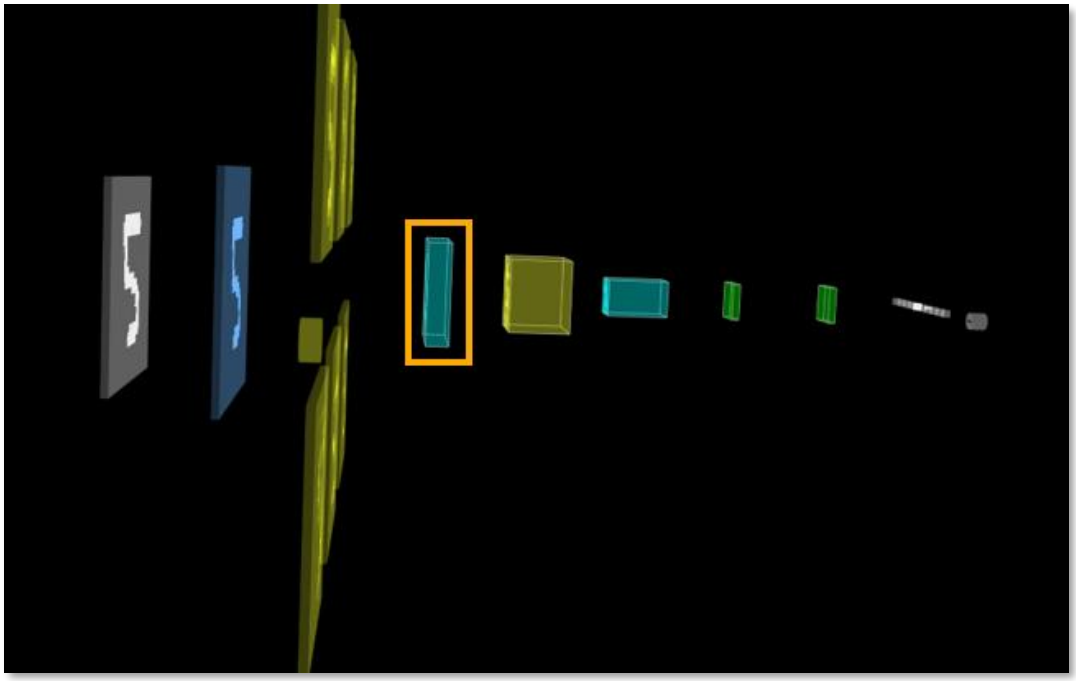
It should show that it figured that it was 5 (red box).

The next step is to use your mouse in the middle section of the window to **rotate the image** so that it appears in the traditional LeNet-5 flow, displayed from left to right as follows:

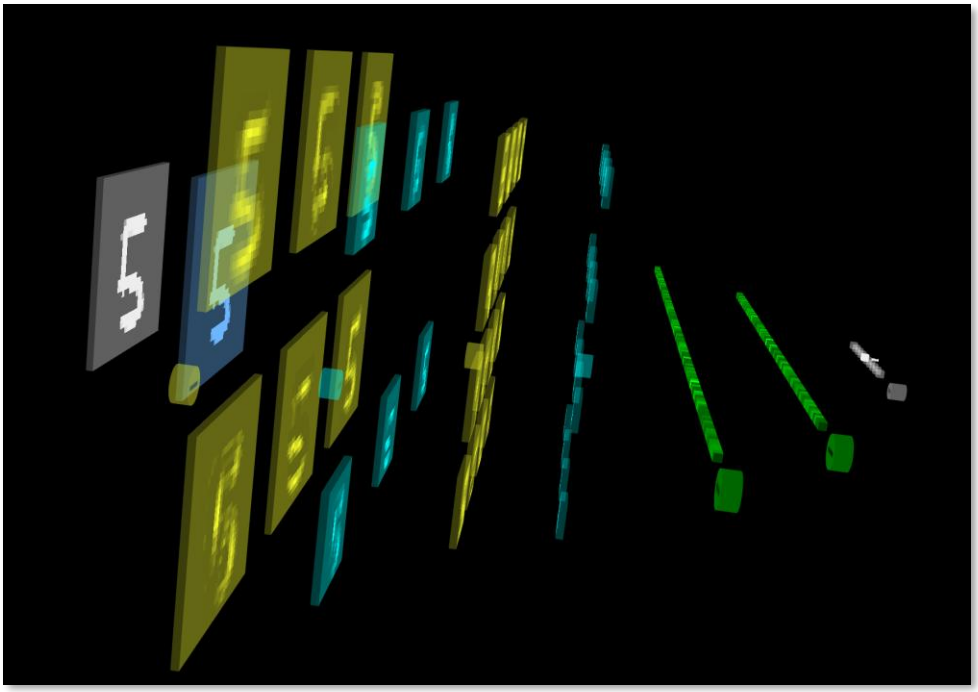


You will want the number (5) right side up, to make it easier to recognize.

Click on each of the unexpanded blocks, such as the yellow block to view its contents.



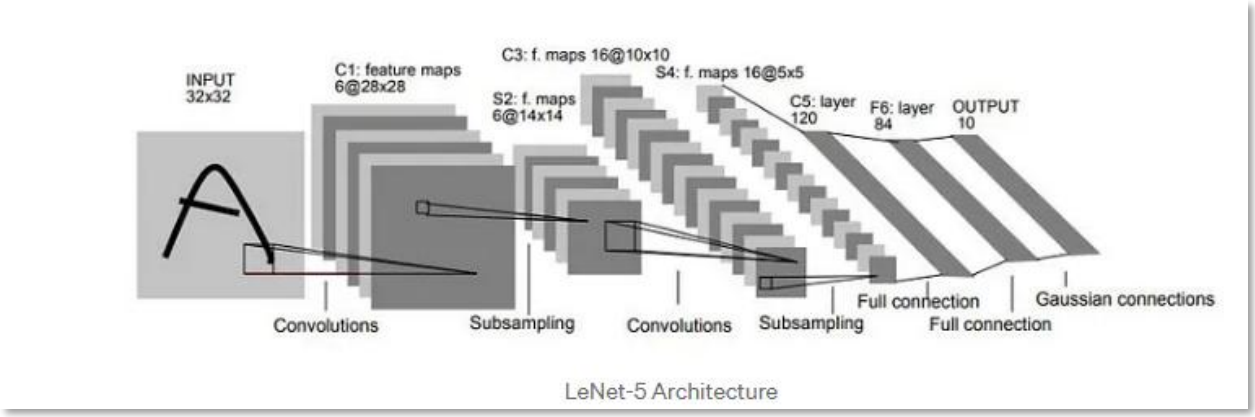
The expanded block view will now appear as follows:



Now let's examine the building blocks and compare them to a LeNeT-5 model.

There LeNet-5 model appears as follows:

[LeNet 5 Architecture Explained. In the 1990s, Yann LeCun, Leon Bottou... | by Siddhesh Bangar | Medium]

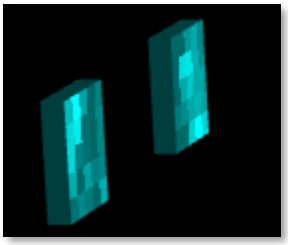


LeNet-5 Architecture

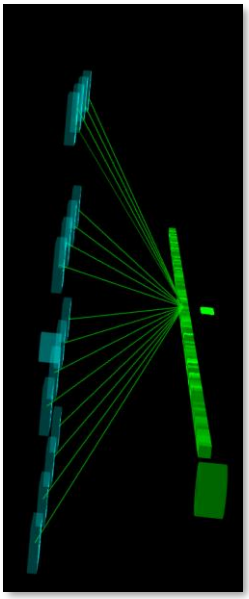
Note the basic building blocks that are present in this LeNet-5 model is what is expected. There is convolution → pooling → convolution → pooling and then 2 fully connected layers before the output. The convolution and pooling sections are the feature extraction section, while the fully connected layers are the classification section.

Now if you examine the 3D image, let's match them to the LeNet-5 model in order as:

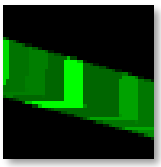
- First grey block contains what you entered. It is in 28×28 pixels. If you put the mouse over the grey block, it will show 28×28 pixels. 28×28 is a standard pixel interface for NMIST.
- Blue block is your same entered number, but the pixel size is now 32×32, which is larger. Within a model, the pixel size can be enlarged.
- Two yellow box sections are convolutional sections. For LeNet-5, it is common to use for the first convolution section- 6 filters (6@28x28), which results in 6 images. The filters extract features like horizontal lines, vertical lines and curves. Note in the first convolution section, you can recognize the number. However, in the second convolution layer, the number is less recognizable. This size reduction is due to the down sampling. In the second convolution stage, there are 16 filters (16@10x10). 16 filters extract more features than 6 filters, which means the model is better able to "learn" the digits. 16 filters are used rather than other filters such as 32 because of the tradeoff of performance vs efficiency found during empirical testing. One way to imagine the 2 convolutional stages is to look at numbers 3 and 8. During the first convolution stage, the model can detect curves which both these numbers have. However, in the second stage, the model needs to extract more features and thus learn that the number 8 has more curves than the number 3.
- Two light blue sections are the pooling (down sampling). Note how the size of the images is reduced, which is expected in CNN because the model extracts the max features (i.e. remove noise and other features that are not as prevalent). If you select the second pooling section, the number is hard to recognize, but it is "recognized" by the model.



- Two bright green sections are fully connected layers. If you select the first green section, you can see many connections from the pooling layer to one node in the fully connected layer such as below.



If you examine the fully connected layer, you will see shapes that you cannot recognize such as:



However, the model has "learnt" via training what this will be connected to. There are different ways to connect to the output results such as Gaussian connections and SoftMax. Gaussian means that it follows the Gaussian curve and uses the mean values. SoftMax does the conversion into probability.

- The last grey column is the output (i.e. the 10 numbers) that you need. The 10 numbers correspond to the 0 to 9 digits.

You can experiment with other numbers as you wish before you close the website tab.

How does the LeNet-5 model even know that the outputs are correct?

The previous sections describe the model itself. However, now there may be further questions such as

- How does the LeNet-5 model even "know" that the outputs are even correct?
- How are output errors reduced?

Let's discuss the details of these questions individually.

How does the LeNet-5 model even "know" that the outputs are even correct?

The LeNet-5 model uses the mathematical function called *loss functions*. There are different implementations of loss function but what is normally used for LeNet-5 models is called *categorical cross-entropy*.

Categorical cross-entropy means that the function will check if the incorrect class ("output bucket") was selected.

For LeNet-5, there are 10 different output classes that range from 0 to 9. The error function looks if the number selected is put into the correct or wrong "output bucket". For example, if the output ("bucket") 3 was selected instead of the output ("bucket") of 8, this would be incorrect. The Categorical cross-entropy is applicable because one digit can only belong to one output "bucket" and not 2 or more "buckets".

The loss function mathematically uses multiplication and log functions with SoftMax results and one-hot encoded value of the label. The goal of this function is to reduce this loss by making changes in the model to make the results more accurate.

Now, let's explain this with an example.

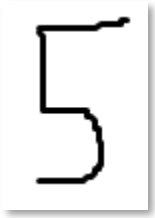
LeNet-5 uses a labeled dataset. A labeled dataset means that each data has an associated label with it.

Let's assume that the data is a 5 (as below) and the label associated with this data is a five. A model does not use words for labels but rather, it uses a number representation because it works better for calculations.

Label = five

Both the data and the label will be used for the loss function calculation.

The first step is to work with the data portion itself.



The data is input into the model. The model transverse the model forward by doing calculations using convolution, pooling, flattening, Fully Connect layers and then providing an output using the SoftMax function to arrive at the result.

Let's assume that SoftMax shows probability of

Class "Bucket" of Digital Output	0	1	2	3	4	5	6	7	8	9
SoftMax	0	0	0.1	0	0	0.8	0	0.1	0	0

This means that number 5 is 0.80 or 80% likely to be the correct result. The model cannot determine from the SoftMax value whether the results are correct or incorrect, nor can it use this value alone to adjust its parameters. However, using the loss function will help determine some of the values.

For the loss function, it compares the SoftMax result to the label (label = five in this case). First, the label for five needs to be converted to one-hot encoding. One-hot encoding means that all the numbers are zero unless it corresponds to the number in question (5) and then it is 1. The one-hot values would look like:

One-hot encoded of Five	0	0	0	0	0	1	0	0	0	0

Note in this above table that the number in the 6th position (related to number 5) is 1.

The next step is to use an equation to use the one-hot value and the SoftMax calculations. The equation of this is

$$\sum True\ label \times \log\left(Predicted\ label\right)$$ or

[Sum of the (true label * log (the predicted label)]

Which means that every value from the true label (one-hot encoded) is multiplied with log of the predicted label.

True label

One-hot encoded of Five	0	0	0	0	0	1	0	0	0	0

Predicted label

One-hot encoded of Five	0	0	0.1	0	0	0.8	0	0.1	0	0

Going through each number of the predicted label with the equation, the math becomes

$$0 \times \log(0) + 0 \times \log(0) + 0 \times \log(0.1) + 0 \times \log(0) + 0 \times \log(0) + 1 \times \log(0.8) + 0 \times \log(0) + 0 \times \log(0.1) + 0 \times \log(0) + 0 \times \log(0)$$

Due to the many zeros in the one-encoded value, this means that equation gets simplified to be $1\times \log(0.8)$

NOTE: There are different bases for log. There is natural log, which is based on Euler number ($e \approx 2.71$), log base 10 (which is used by default in Excel), log base 2 and others.

Natural log is commonly used for ML, where this equation becomes on

$$-\ln(0.8) = 0.22$$

Had the SoftMax value been 0.9 instead of 0.8, the calculation would be $-1 \times log(0.9)$ which in Excel as $-\ln(0.9)$ equals 0.10, an even smaller value.

Backpropagation and Gradient Descent

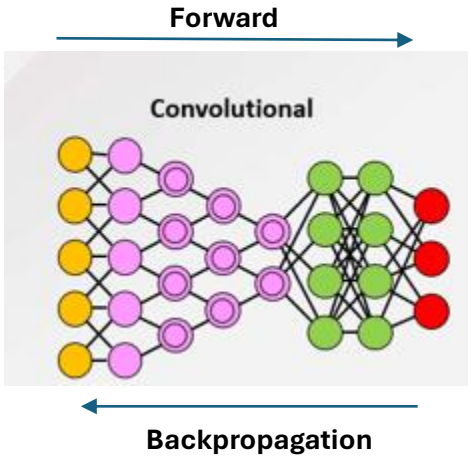
The goal for ML is to minimize the function (error) loss.

How are the errors even reduced?

During the training stage, the algorithm uses the calculated loss function to see the value of the difference between the actual and predicted result.

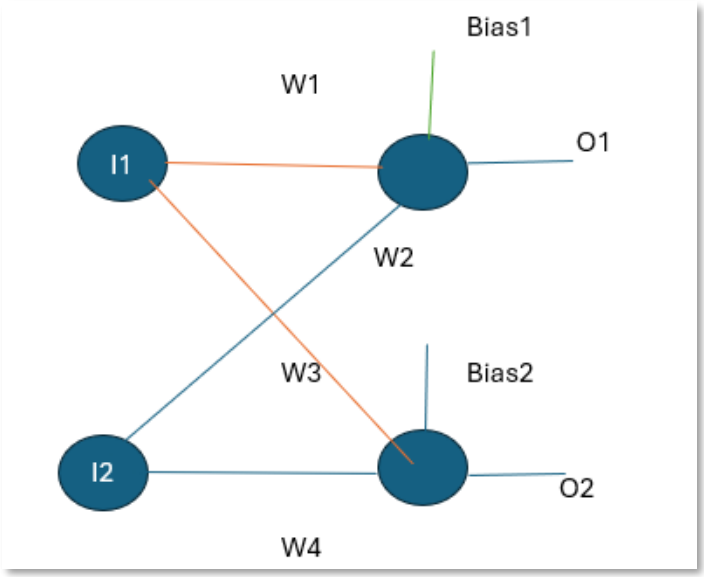
The goal is to reduce the errors to be as small as possible. The errors are reduced by using back propagation. Backpropagation means that the algorithm now goes backwards through the model i.e. from the outputs of the model and goes to FC layers and other layers towards the inputs. Along the process, it changes the weight values of the NNs.

The algorithm determines which weights contribute the most or least to the output errors. Weights that contribute more to the error will be adjusted more significantly than those that contribute less.



The model will go back and forth over epochs (an epoch means one pass through the dataset) to reduce this error. To better explain backpropagation, here is an example.

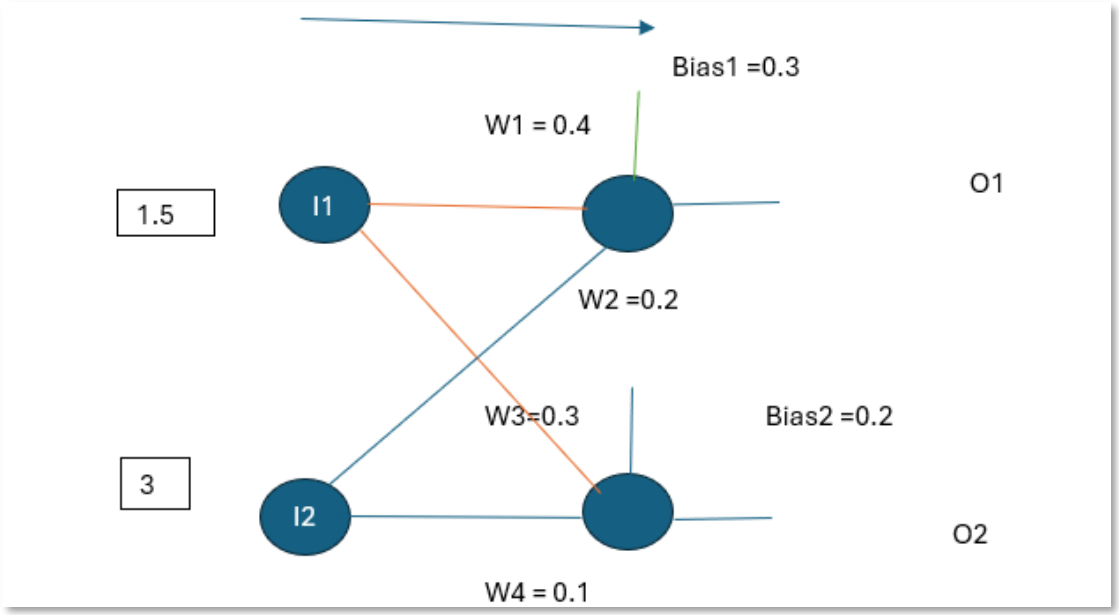
Let's assume part of NN with the inputs **(i1, i2)**, 1 hidden layer (with 2 NNs) and 2 outputs **(O1, O2)** as it is illustrated in the following image:



Forward Pass

The first step is to go through the neural networks forward, which means the model will go from the inputs (i1, i2)-> hidden layer -> outputs (O1, O2).

Since the calculations can be easier to understand with numbers, let's assume the following values for weights (W1, W2, W3 and W4) and bias (Bias1 and Bias2). The model uses the parameters of weights and biases to determine the output.



The first calculation is a linear calculation which is $\boldsymbol{output} = \boldsymbol{input} \times \boldsymbol{weights} + \boldsymbol{bias}$. For reference, this is the same formula as for a line equation $\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}$.

For Output1 this will be

$$\begin{aligned} O1 &= I1 \times W1 + I2 \times W2 + Bias1 \\ &= 1.5 \times 0.4 + 3 \times 0.2 + 0.3 \\ &= 1.5 \end{aligned}$$

For Output2 this will be

$$\begin{aligned} O2 &= I1 \times W2 + I2 \times W2 + Bias2 \\ &= 1.5 \times 0.3 + 3 \times 0.1 + 0.2 \\ &= 0.95 \end{aligned}$$

Since NN needs to handle nonlinear data, reflecting the nature of real-world information, an activation factor is required. There are various activation functions, with ReLu being one of the most common.

ReLu means:

- if the output value is > 0, the value remains as it is,
- if the output value is <0, the value becomes 0.

One way to understand ReLu is to view it as multiplying the output value by 1 for positive numbers and by 0 for negative numbers.

Since the values for *O1* and *O2* are greater than 0, applying the Relu activation leaves them unchanged at *1.5* and *0.95*.

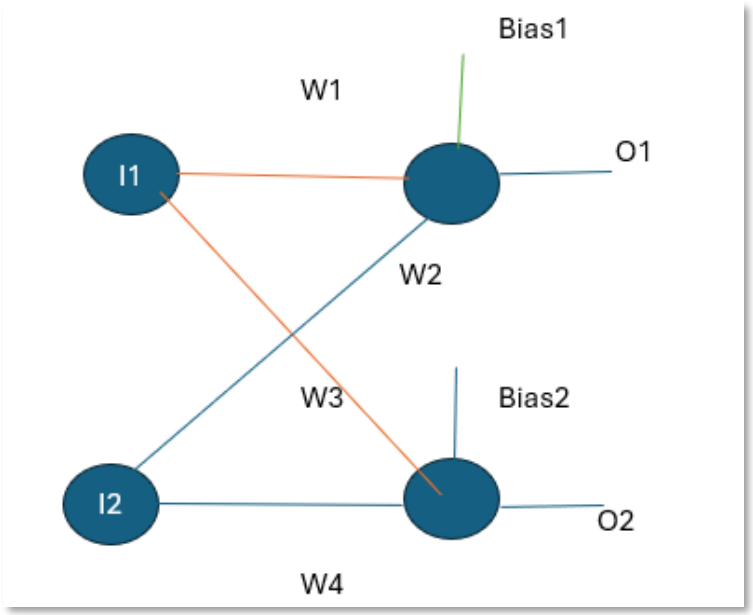
Next, the loss function calculated. A loss function measures the difference between the model's prediction and the actual value. The objective is to minimize this loss.

First, the step to calculate SoftMax:

- Add the numbers of $1.5 + 0.95 = 2.45$ total.
- SoftMax for *O1* is given by $1.5 \div 2.45 = 0.61$, while the SoftMax for *O2* is given by $1.5 \div 0.95 \div 2.45 = 0.38$.
- The loss function is calculated by $-\ln(0.61) = 0.49$, that means the loss function needs to be greatly reduced.

Backward pass

The next step is to do a backward pass (back propagation) as:



Back propagation means that the model, which is traversed backward, uses a gradient descent algorithm and the math (gradient of errors) associated with this algorithm to change values such as weights and biases to reduce the loss function.

What is gradient descent?

Gradient descent is an algorithm used to determine the minimum loss function. The minimum loss function is the smallest variation between actual vs predicted value.

Here is one way to visualize a gradient descent. Let's imagine that you are at the top of a steep mountain, and you would like to get down to a small tent on flat land at the base of the mountain. However, here is the catch. You cannot see the route down at all, because there is a very bad storm. You can only feel the way down as you carefully walk down. The beginning of the descent will be very steep, which means that you are getting down the mountain quickly. However, as you get closer to the base camp, the landscape is no longer steep, which means that you are getting to the minimum (lowest) level.

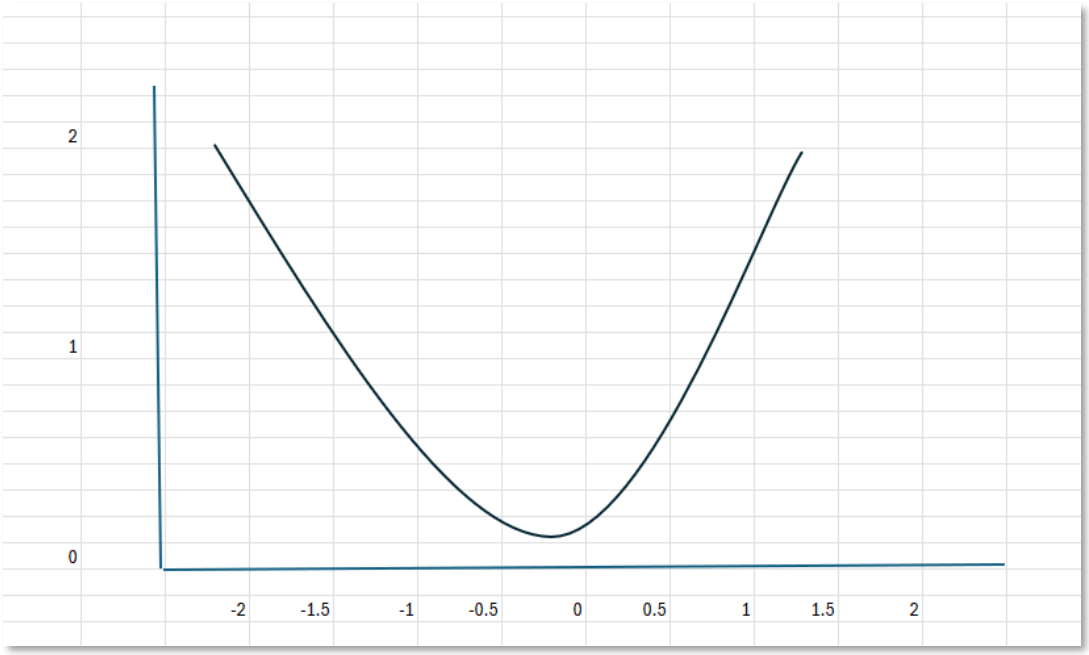
Now let's relate this mountain trekking example to ML.

- Height of the mountain is equivalent to the loss function.
- Slope of the mountain corresponds to the slope of gradient. The slope of the mountain is steepest at the top and not as steep at the base.
- Derivative of the slope of the line provides the direction of travel.
- Size of the steps that you take down the mountain are equivalent to the learning rate. The larger the steps, the higher the learning rate, whereas the smaller the steps, the slower the learning. If you take small steps rather than large steps down the

mountain, you will need much longer to get to the base camp. However, if you took large steps, you might pass by the tent.

- Base camp is the minimum function, because it is the bottom of the mountain. The minimum function is where the parameters provide the best predicted to actual values.

An example of looking at gradient and gradient descent is to look at the parabola as:



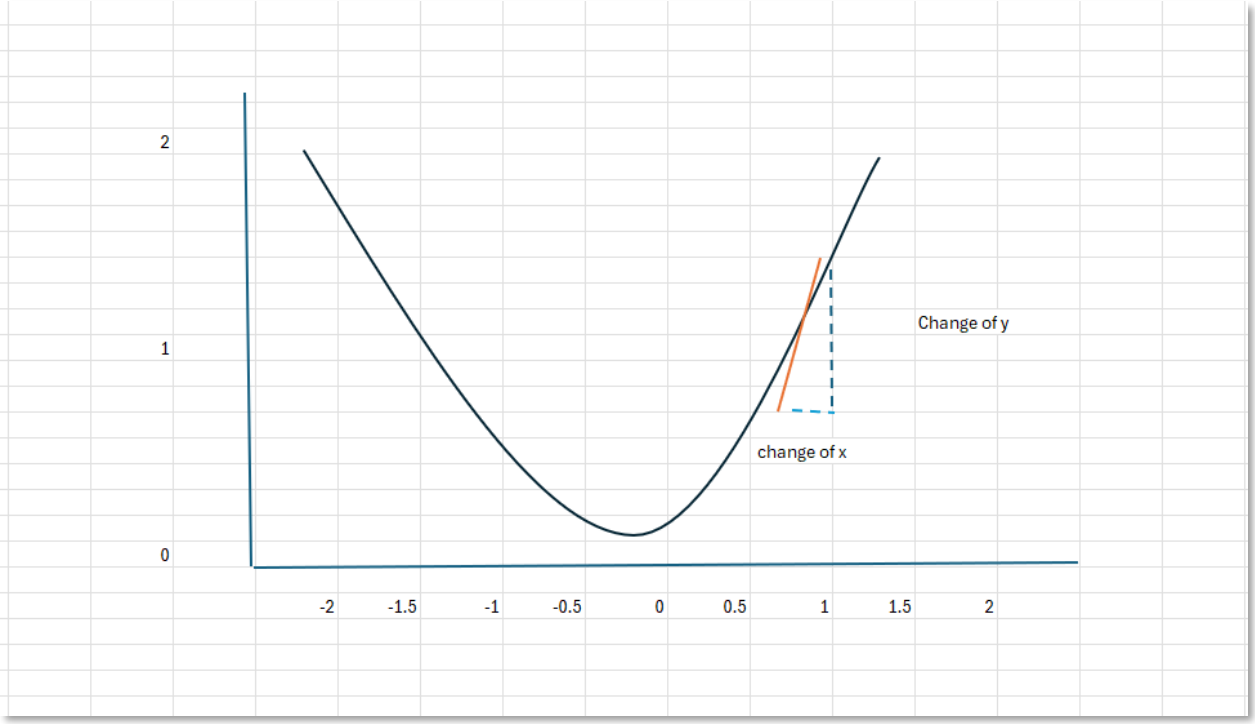
The sides of the parabola are steep (just like the side of the mountain) and the bottom of parabola is flat (just like the flat land where the tent is located on).

The objective for ML is to move down the slope (i.e. move down the mountain) until you get to the flat section, which is the minimum). The minimum section is the lowest point of the function, where the parameters provide the best prediction.

The question might be: **How is this done?**

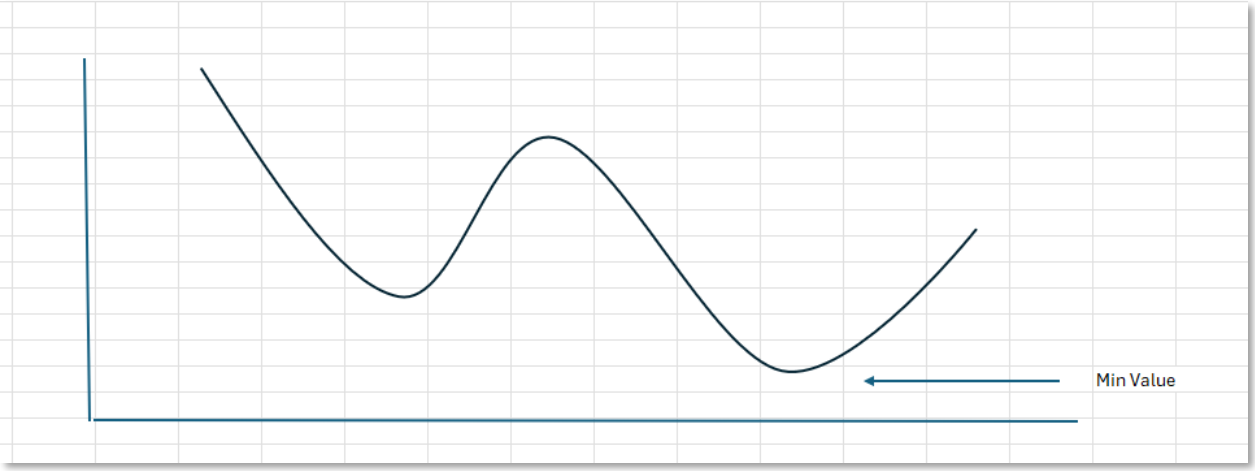
One way to start doing this calculation is by using algebra. In this approach, calculating the slope is to put a line on the parabola (shown in orange) that connects to one point on the parabola and then do the slope of the line calculations as

Slope of the line = change of y and/or change of x



Algebra works well for 2D shapes; however, ML uses more than 2D. When ML uses vectors, this is greater than 2D.

Another item to be concerned with calculating the slope in machine learning is that the results are not the ideal parabola shape, but the result could be "curvy" as below (or even other variations).

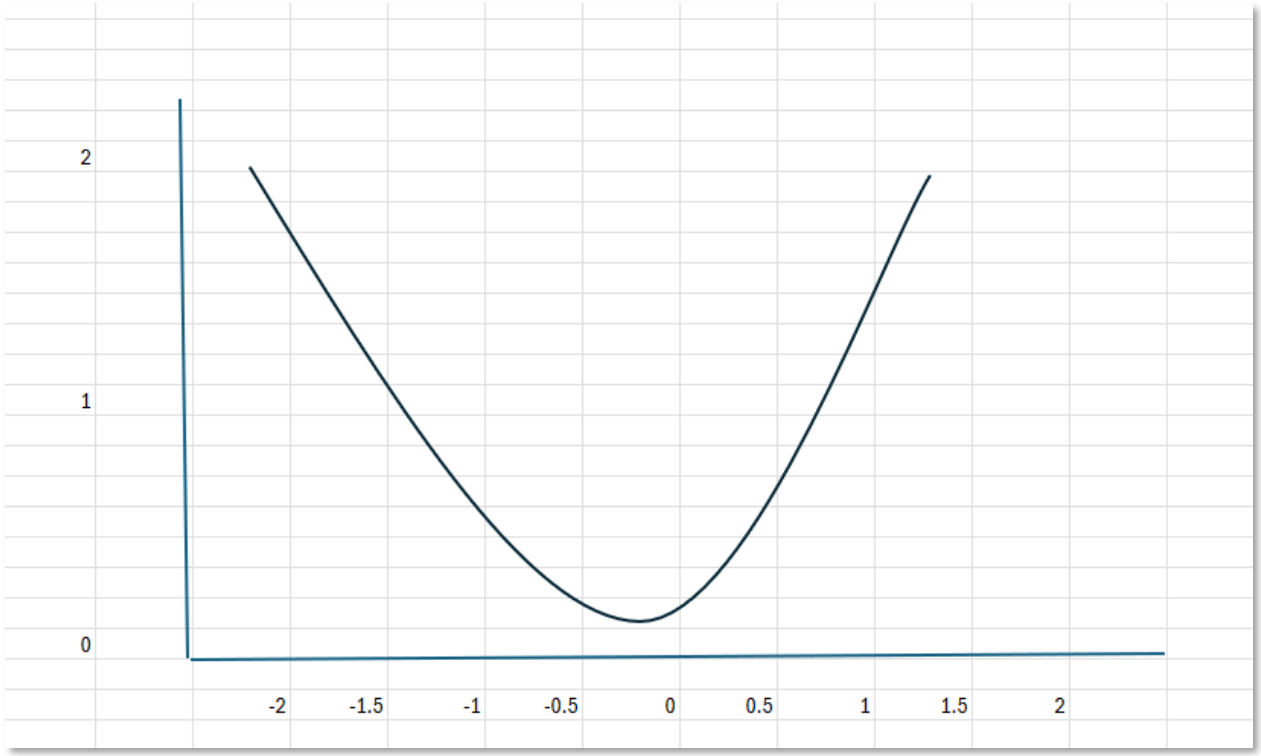


This means that the minimum value is not easy to calculate.

For example: Let's think back to climbing down the mountain during the snowstorm. If you end up on a slight plateau of the mountain side, you might think that you are at the lowest point, but you need to walk much further down so that you get to the base tent.

Now for ML, derivatives are used instead of algebra to figure out the slope. A derivative can work with multiple dimensional inputs, complex results and more.

Let's go back to the parabola curve for an example of how this works:



The parabola formula is $y = x^2$. This in turn means that the derivative is $y = 2x$.

A derivative is a rate of change (i.e. the slope), which means at every point for the parabola, the derivative equation is used to figure out the rate of change (slope).

If the slope is the line is positive, you would move the parameters to the left. If the slope is negative, you would move to the right to try to locate the minimum location.

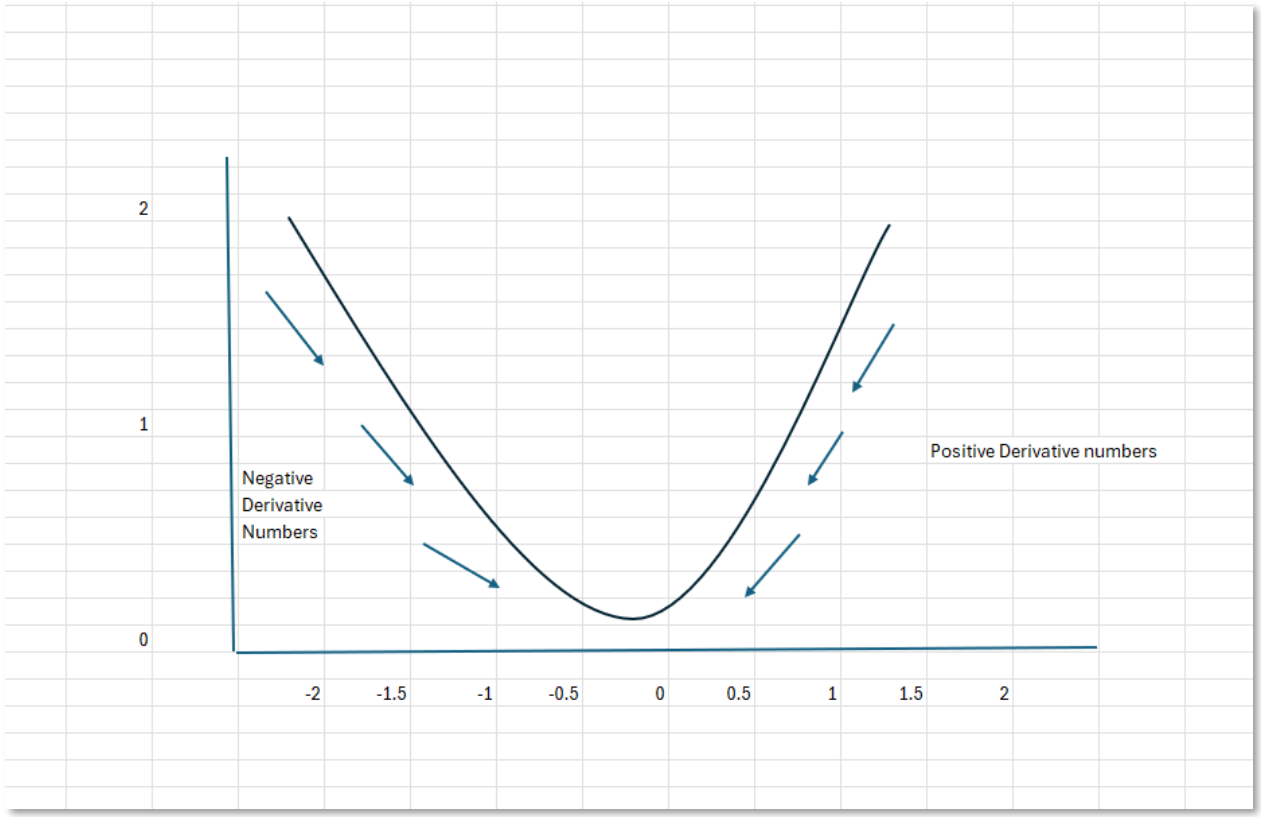
For example, look at some values to illustrate this:

x	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2
Derivate result of 2x	4	3	2	1	0	-1	-2	-3	-4

Note the positive derivative (slope) result numbers of *1* to *4*, this means that you would adjust the parameters (i.e. weights) to the left. Note the derivative result of *4* is a much steeper slope than derivative result *2*.

For the negative derivative (slope) result numbers of *-1* to *-4*, this means that you would adjust the parameters (i.e. weights) to the right.

The number *0* means that you have hit the min point.



The next question might be: **How do the above gradient calculations relate to the earlier NN and change the parameters?**

There are three items to consider:

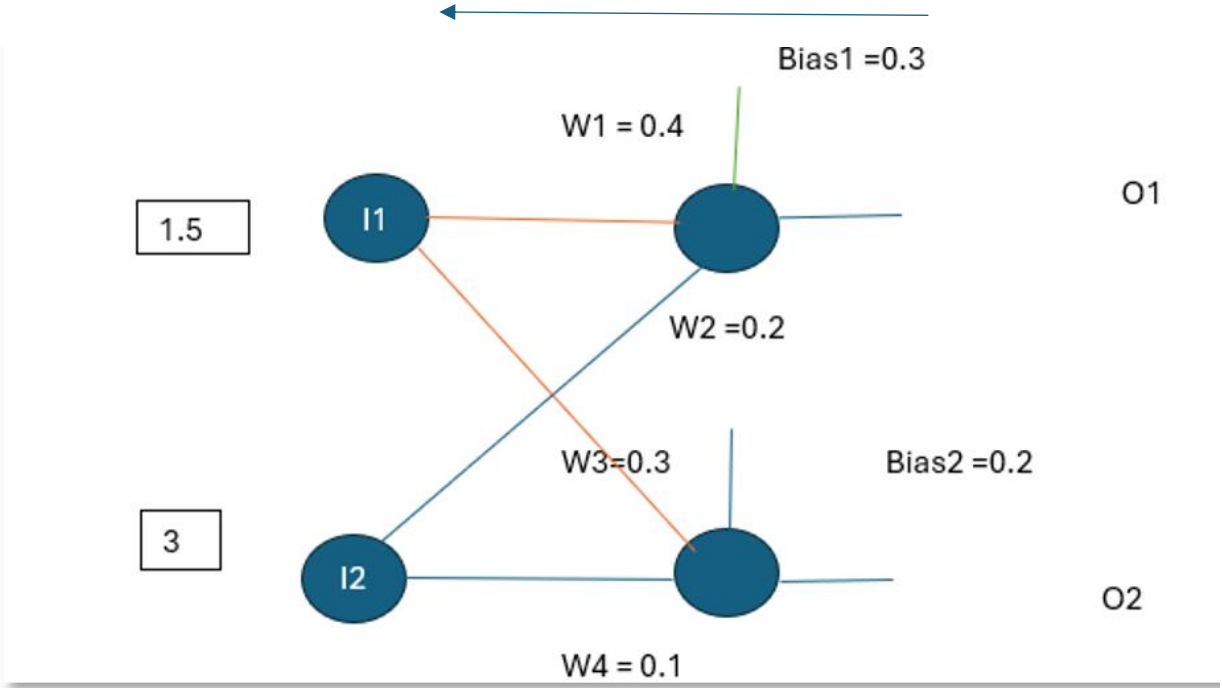
- **Loss function.** This is a number that is the difference between the model prediction and the actual value. The goal of ML is to minimize this loss function.
- **Gradient descent.** This is an algorithm that is used to find the minimum value. It uses the values of the gradient of errors to help guide it to the minimum function.

- **Gradient of error.** This does the math (calculations) for the gradient descent. It provides the numbers/ of the error (loss) that the parameters should be changed to.

The gradient of errors is then feedback though the model. These errors affect all the functions, i.e. the activation functions, outputs, weights, bias, fully connected layers, pooling, convolution, and inputs.

Gradient of errors are the variations of gradient- errors caused by various sections of the model. During backpropagation, the parameters (weights and biases) are modified so that when the model goes through the forward path again, it will use the modified values to try to predict even more accurately. The forward and back passes will be iteratively done to reduce the loss function.

If we examine the model going backwards, the gradient of errors will affect the activation (not shown), output, weights, bias, and input. The following will show the calculations.



Each of these losses can be figured out as:

- ***Gradient of Loss for the Activation Function (ReLu)***

Derivative of the ReLu is *1* when the output is greater than *1*, but if value is negative, the value becomes zero. The ReLU calculation in the forward path was *1.5* and *0.95* but with the derivative of ReLu, the value of *1.5* will change to be *1* (da/dz_1) and the value of *0.95* will also change to be *1* (da/dz_2)

- da/dz_1 stands for derivative of activation per dz

- $da/dz_1 = 1$ (first output) and $da/dz_2 = 1$ (second output)

- ***Gradient of Loss for the outputs (O1 and O2)***

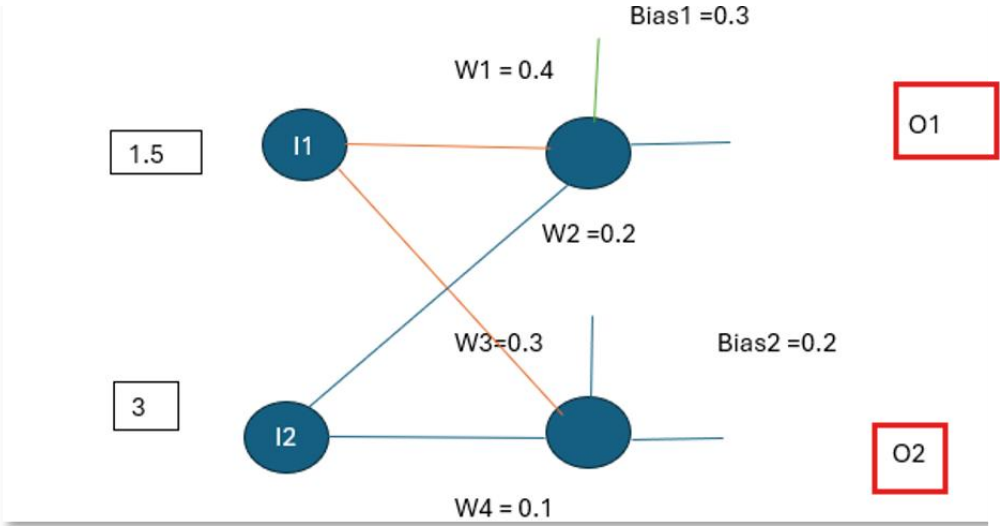
Let's assume that the gradient of the loss for the output was known where it is *0.2* and *0.1*, respectively. The gradient of loss for the outputs is given as

$$dl/dz_i\ (gradient\ of\ loss\ for\ outputs) = derivative\ of\ activation\ function\ (\frac{da}{dz}) \times output,$$

where derivative of activation function is dl/dz_x, and dl derivative of gradient of loss for corresponding output. Therefore, for this example $da/dz_1 = 0.2$ and $da/dz_2 = 0.1$.

- $dl/dz_1 = da/dz_1 \times output_1 = 1\ x\ 0.2 = 0.2$
- $dl/dz_2 = da/dz_2 \times output_2 = 1\ x\ 0.1 = 0.1$

Graphically, this can be seen where the 2 outputs (01 with the value of 0.2 and 02 with the value of 0.1) are multiplied by the derivative of the activation function (value of 1).



- ***Gradient of loss for the weights***

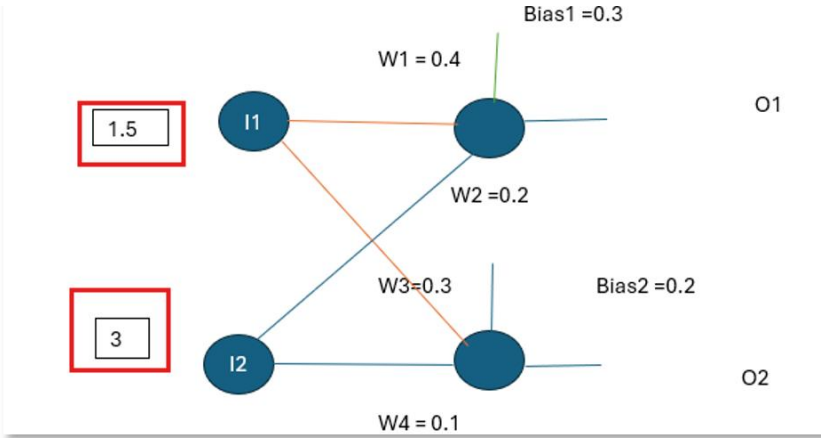
Every weight will be modified a little. The formula is given by

$$dl/dw_i\ (gradient\ of\ loss\ for\ the\ weights) = Gradient\ of\ the\ loss\ for\ the\ outputs \times inputs$$

The input values are $I_1 = 1.5$ and $I_2 = 3$.

- For weight 1 → $dl/dw_1 = dl/dz_1 \times I_1 = 0.2 \times 1.5 = 0.3$
- For weight 2 → $dl/dw_2 = dl/dz_1 \times I_2 = 0.2 \times 3 = 0.6$
- For weight 3 → $dl/dw_3 = dl/dz_2 \times I_1 = 0.1 \times 1.5 = 0.15$
- For weight 4 → $dl/dw_4 = dl/dz_2 \times I_2 = 0.1 \times 3 = 0.3$

Or graphically, from the diagram, this calculation takes the 2 inputs (1.5 and 3) and multiplies them by the calculated gradient of the output loss (0.2 and 0.1) to see the change of the loss for the weights.



- ***Gradient of loss for Bias***

The bias will be the same in this case. In more complex NN, this would be a variable, and it would change in value.

- ***Gradient of loss for the input***

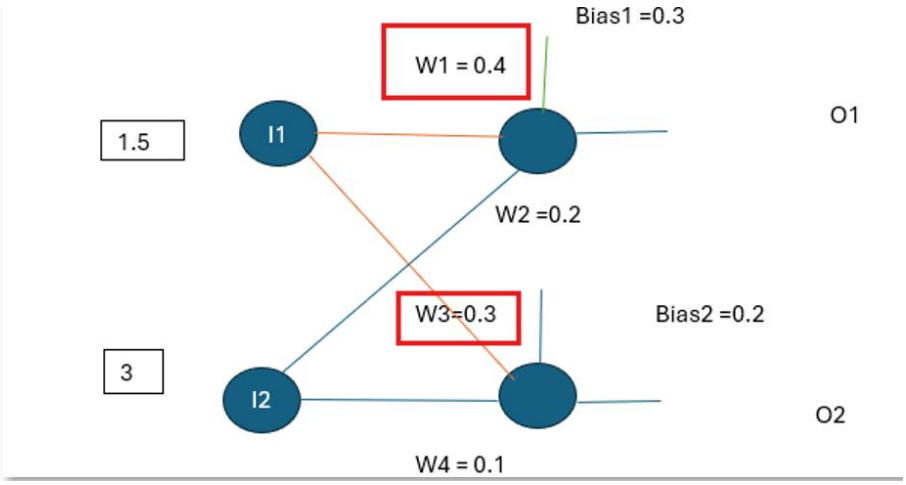
The reason for calculating gradient of loss for the input means that if the inputs are changed somewhat, it would check if the outputs could be correct or not. This can help with corner cases. The formula is given by

$$dl/dx_i(gradient\ of\ loss\ for\ the\ input) = \ weights \times gradient\ of\ loss\ for\ the\ output.$$

By replacing the corresponding values, we obtain the following:

- $dl/dx_1 = w_1 \times da/dz_1 + w_3 \times da/dz_2 = 0.4 \times 0.2 + 0.3 \times 0.1 = 0.11$
- $dl/dx_2 = w_2 \times da/dz_1 + w_4 \times da/dz_2 = 0.2 \times 0.2 + 0.1 \times 0.1 = 0.05$

Another way to see this graphically, for the dl/dx1 equation (input 1), there are 2 weights connected to input 1 as W1 and W3. The weight values are multiplied by the activation numbers (0.2 and 0.1) respectively.



Gradient of Loss, Weight Values and Learning Rate

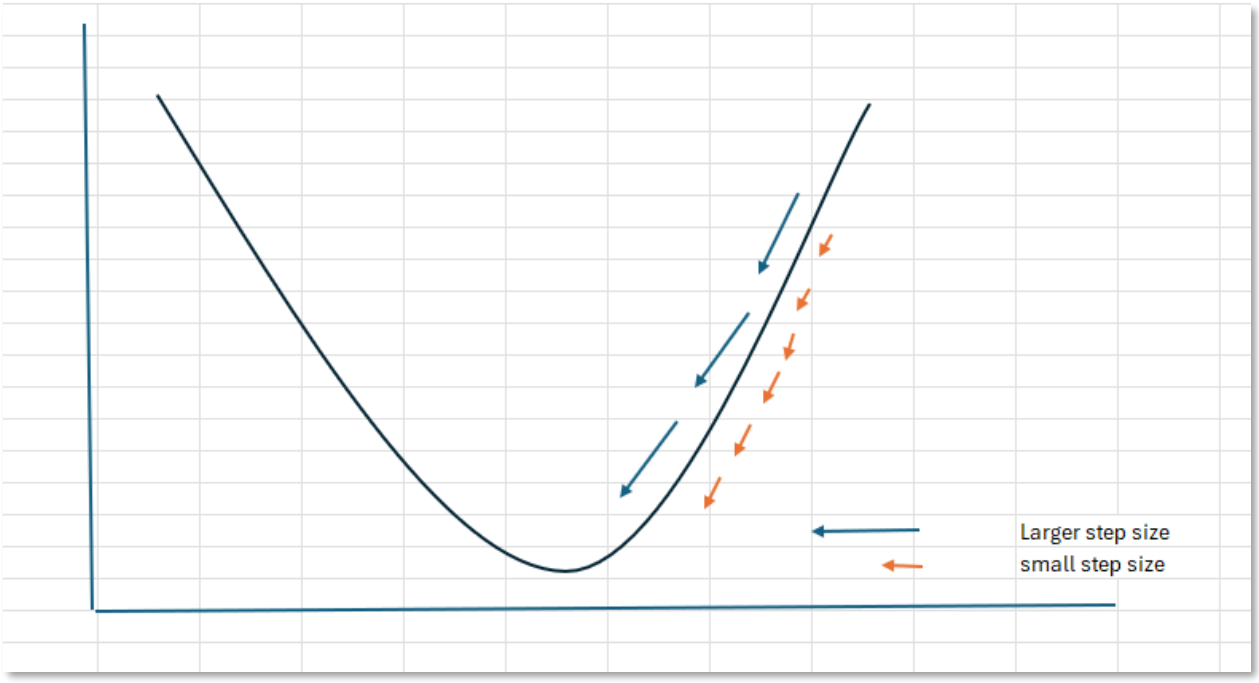
The next question is: **How does gradient of loss values affect the weight values?**

The formula for this is

$$New\ weight\ =\ Original\ weight\ -\ (Learning\ rate \times gradient\ of\ loss\ for\ the\ weight)$$

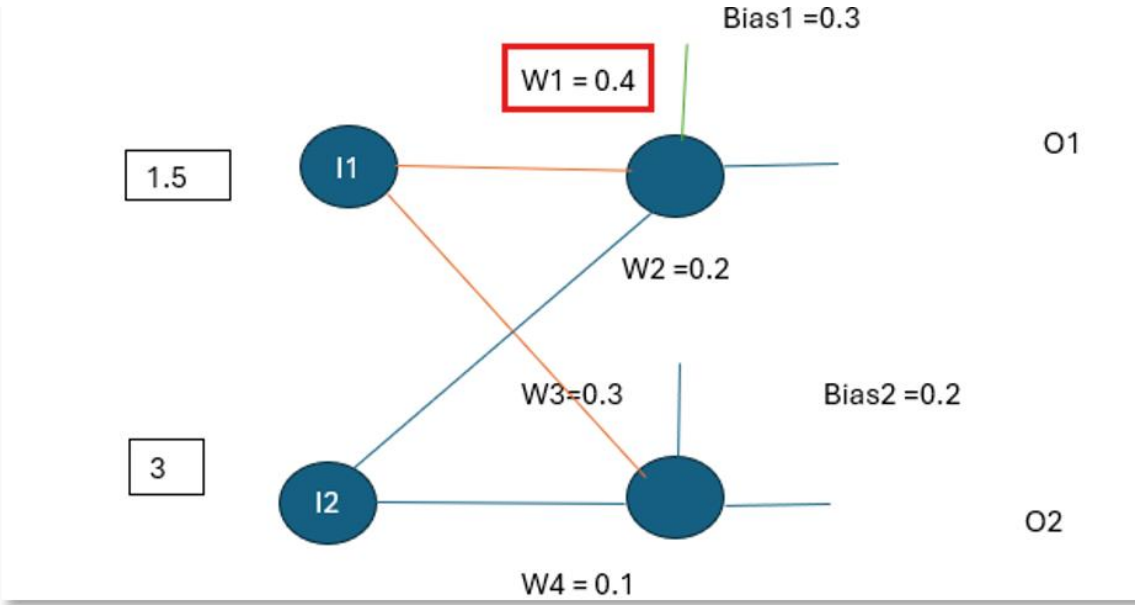
Learning rate, which is a small number up to the value *0.1*, controls how much the weights are updated. Learning rates are step sizes that range from *0.0001* (very small step size) to *0.1* (large step size). The closer the number is to *0.1* the more the weights will be changed in each step. However, there can be overfitting if the step size is too large. A lower learning rate means that it will need much more time to reach the min. function.

For example, in this diagram, note relatively many more step sizes are needed for the small step size vs the larger step size.



Now let's examine a numerical example.

Use the original $w_1 = 0.4$ as



The learning rate of 0.1 (which is a fast-learning rate)

Gradient of loss for the weight 1 as 0.3 (dl1/dw1) as calculated earlier with weight 1 → $dl/dw_1 = dl/dz_1 \times I_1 = 0.2 \times 1.5 = \mathbf{0.3}$

in the formula of

$$New\ weight\ =\ Original\ weight\ -\ (Learning\ rate \times gradient\ of\ loss\ for\ the\ weight)$$

to obtain new *weight* as

$$new\ weight_1 = 0.4 - (0.1 \times 0.3) = 0.37$$

The new weight will be *0.37*.

If the learning rate had been *0.01* (slower learning rate) instead, the new weight would be *0.397*, which is very close to the original weight of *0.4*.

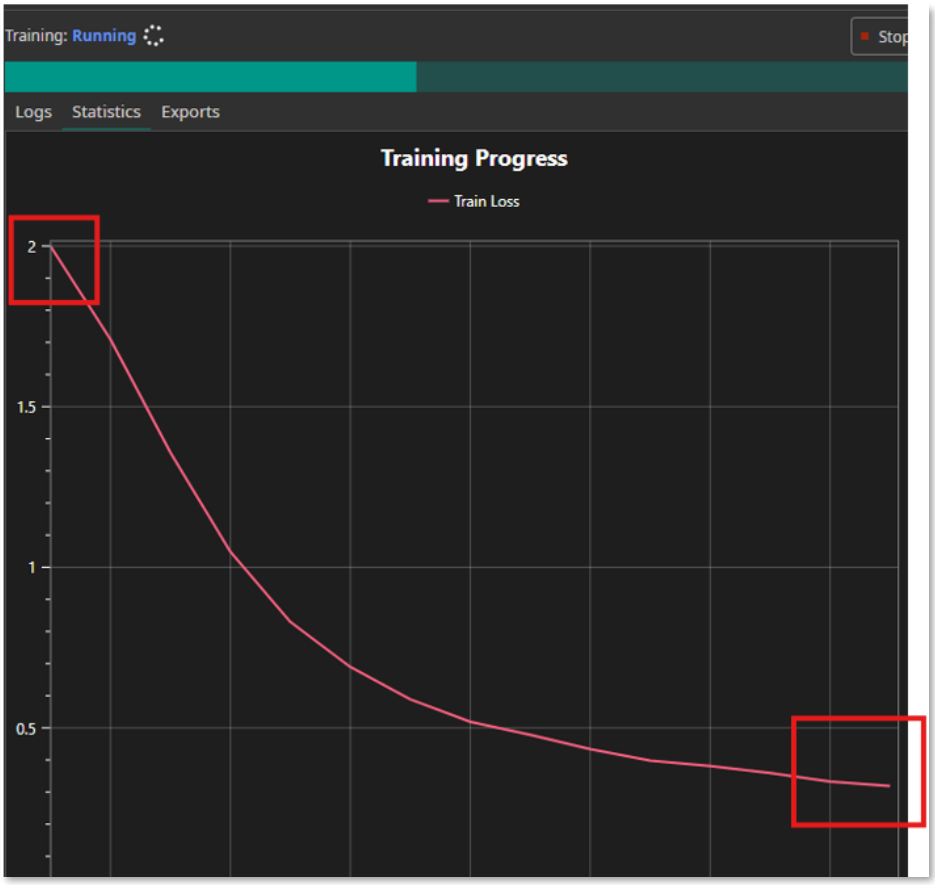
The other weight values and the bias values would also change similarly. This means that when the model goes through the forward path in the next epoch, the new weights will be used instead.

This mathematical process of forward path and backward propagation is what the model uses during training to reduce the loss and become more accurate.

Training Loss vs Accuracy

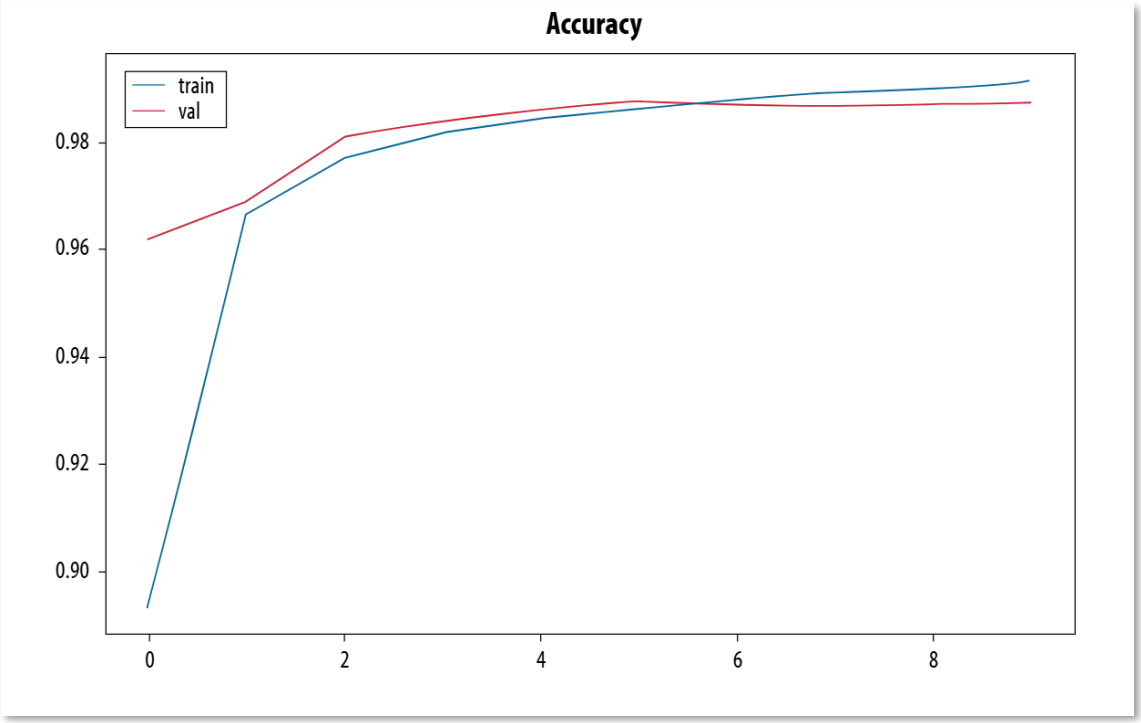
The question might be: **What does a training loss look like graphically?**

Here is an example from the OneWare training workshop. Note at the beginning of the training, the loss was high, while at the end, the training loss is much lower, because it ran through more training cycles (epochs). This curve is desired because the model adjusts the parameter values and converges near the number zero.



Just for reference, loss tends to be commonly known as the inverse of accuracy. For example, as the loss gets close to zero, the accuracy gets closer to 1.

As an example for accuracy, from the **AN 1011**: TinyML Applications in Altera FPGAs Using LiteRT for Microcontrollers, a plot of accuracy shows that it approaches *1*, as it runs through the flow.



Step 3 Framework (software)

A framework refers to the software tools required to work with neural networks.

Frameworks are general-purpose, high-level software packages such as TensorFlow Pytorch, Keras, OneWare Studio, FPGA AI Suite, and many others. Most of these are open source. The choice of framework can depend on factors such as existing tools used within a company, familiarity with the software, and specific project requirements.

When targeting TinyML, TensorFlow is a free framework option. To use TensorFlow, you would work in Python, a widely used, general-purpose programming language, where TensorFlow functions as a machine learning library. Python is like C programming language but is higher-level and thus more simplified.

For the OneWare workshop, the framework used is their software. After selecting parameters, the OneWare software will generate a custom AI model.

Step 4: Datasets-training vs pre-trained models

There are different choices of data sets for ML. There are public (pre-trained and non-pre-trained models) and proprietary models. Using a public dataset means that these models could be used without doing the iterative approach of training and thus save time and cost. Custom datasets can be created later, if needed.

When starting out working with AI, it is easier to start with a pre-trained dataset. A pre-trained dataset contains data and the associated label.

Public, pre-trained models come from many different sources such as academia, industry, government, and more where these models can be downloaded. As an example, OpenVino ModelZoo, which are Intel's pre-trained models. It can be found at the following link:

GitHub - openvinotoolkit/open_model_zoo: Pre-trained Deep Learning models and demos (high quality and extremely fast).

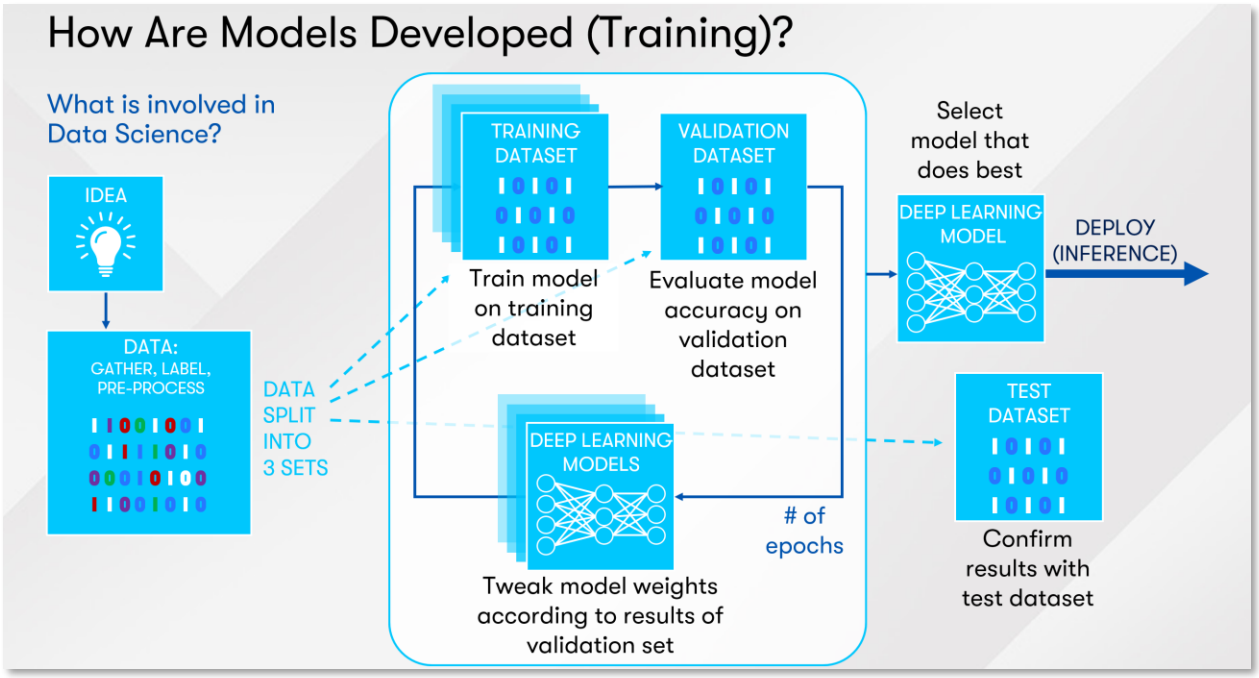
ModelZoo provides models for age, common sign language, face detection, and more. There are free datasets within the TensorFlow and OneWare frameworks.

Datasets can range from small data sets of 3,000 to 60,000 to even datasets in the millions or even hundreds of millions. The larger the dataset, the longer that it is needed to train but the more accurate that the results can become.

Should a custom proprietary model be needed, the dataset will first need to be created, preprocessed and then trained.

Training is an important step for AI. This is the step where a software model is run (trained) with a huge amount of data to determine the parameters (weighing). The more data that is used for training, the more accurate the results can become. Training in extremely large model datasets needs to be done on high-end, fast devices (i.e. GPUs or servers). The training can take days, weeks or months. During the training, there is feedback and learning, as the models also need to be tested.

Graphically the flow for developing models for training are:



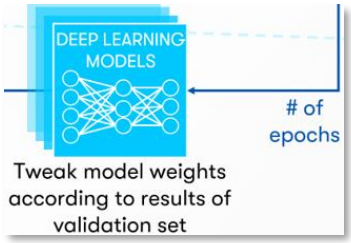
Here are more details to help clarify this flow

The first step is to define what AI should do. For example, whether it should detect people, text, vehicles, etc. These requirements will influence all subsequent steps in the design flow.

All datasets need to be split into smaller different sub-datasets.

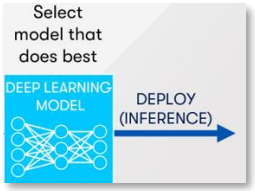
- The training dataset portion, which will be most of the dataset, is used to train the model. As an example, in the MNIST dataset for digit handwriting, there are 60,000 images in the training dataset, and 10,000 images for the test (validation) dataset.

- A validation dataset is a smaller portion of the total dataset. This set, which is not part of the training set, is used to test the model. Since the model would not have seen this data before, it is used to check mathematically what the model predicts vs the actual result.



The datasets—first the training dataset, followed by the validation dataset—are run through the model. As they pass through, the filters (which contain numerical values called *weights*) are adjusted so that the predicted outputs are as close as possible, in percentage terms, to the expected outputs.

The term *epoch refers* to the number of times the datasets are run through the model to achieve sufficiently accurate results, such as 95% accuracy. In the OneWare workshop, the number of epochs will be about 25. For early testing, epochs can be increased to around 100 to observe whether the model is converging, meaning it is getting closer to the correct values.



The software-based model then needs to be changed to a model that can be run (deployed) on a device. Training of a model is run only on a computer, i.e. Windows or Linux or a server; but the model needs to be changed so that it can be deployed (run) on a stand-alone hardware device.

As an example, when targeting the low power, Tiny ML AI flow, the TensorFlow model is changed to a TensorFlow Lite model before it is implemented along with the Nios V into a FPGA. For OneWare, the trained model is exported to VHDL files and an IP core that is used with Platform Designer so that the design is used within a FPGA.

Overall, the objective for AI is the process of iteratively converging on a set of model weights such that the model can perform whatever its intended task is, with some acceptable level of accuracy. In this workshop, there will be model training and accuracy testing.

Step 4a. Errors occurring during training/use pre-trained models

AI can only be as good as the information that is provided. Various types of errors or inaccuracies can occur during the training step or even with pre-trained models where in turn, these errors affect the inference (deployment) results.

Some errors can be caused by:

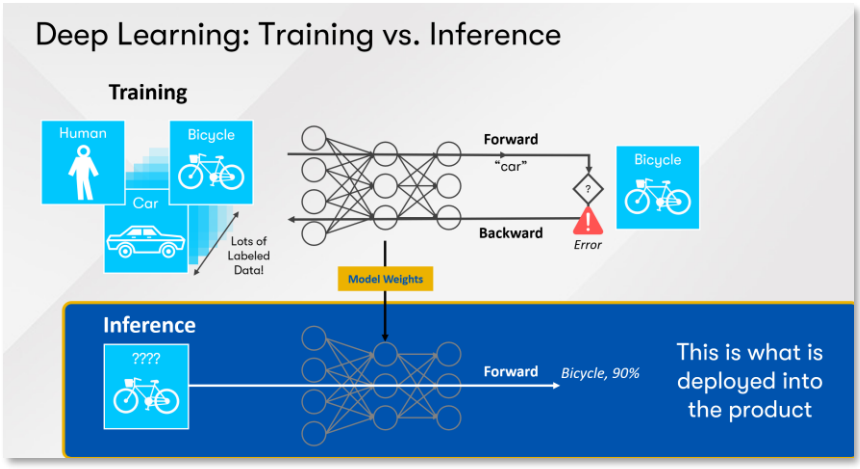
- not using large enough datasets (larger datasets can provide more accurate model results than smaller datasets because the models will be able to generalize more),
- not enough variety in the datasets (i.e. datasets are biased including not looking for corner cases), and
- using only one source of data and more.

There are different ways to determine if there are errors during the training phase. One way is to check the results visually (i.e. did the model determine that it was a bike, a specific animal, etc.). Another way is to do error analysis on the data. Error analysis uses mathematical methods including calculating error rates, data coverage, diagnostic analysis, and more to determine the error. This hands-on workshop will use error analysis on the data.

You might hear the words "AI Hallucination", which refers to the model providing incorrect results i.e. stating items are in an image, but they are not. AI hallucinations can be reduced by implementing larger and varied datasets during training.

Step 5: Inference

Once the model is checked for accuracy and it has the model weights "frozen", it will need to be deployed in a device. In the inference stage, the model weights are placed in the device.



There are different ways to implement model weights into an Altera FPGA such as:

- **Hard-code Classic ML.** This would be "painful" as it is time consuming and prone to errors.
- **Implementing TinyML**. By targeting a low power microcontroller, Nios V or ARM processors.
- **One Ware Studio.**
- **FPGA AI Suite.**

Step 6: Why Use FPGAs for ML?

FPGAs offer many advantages for ML usage that include

FPGAs ideal for custom AI solutions

- Build high performance custom AI network from ground-up
  - Customizable memory hierarchy
  - Configurable Compute capability
  - Programmable data path
- Multiple IO standard support for data ingest/egest

No need to create new heterogenous architecture

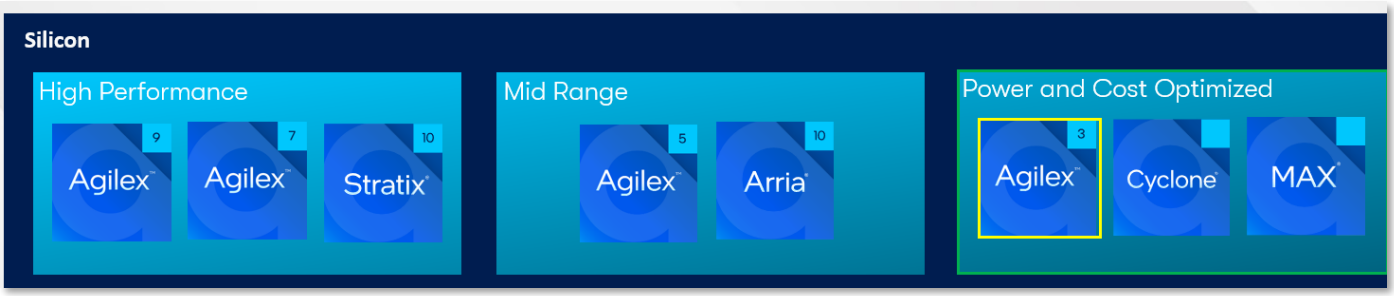
- Just add AI to FPGA
- Real-time with low deterministic latency
- Ideal for edge deployment with low power, & form factors
- Lower TCO and TTM benefits

Future proofing

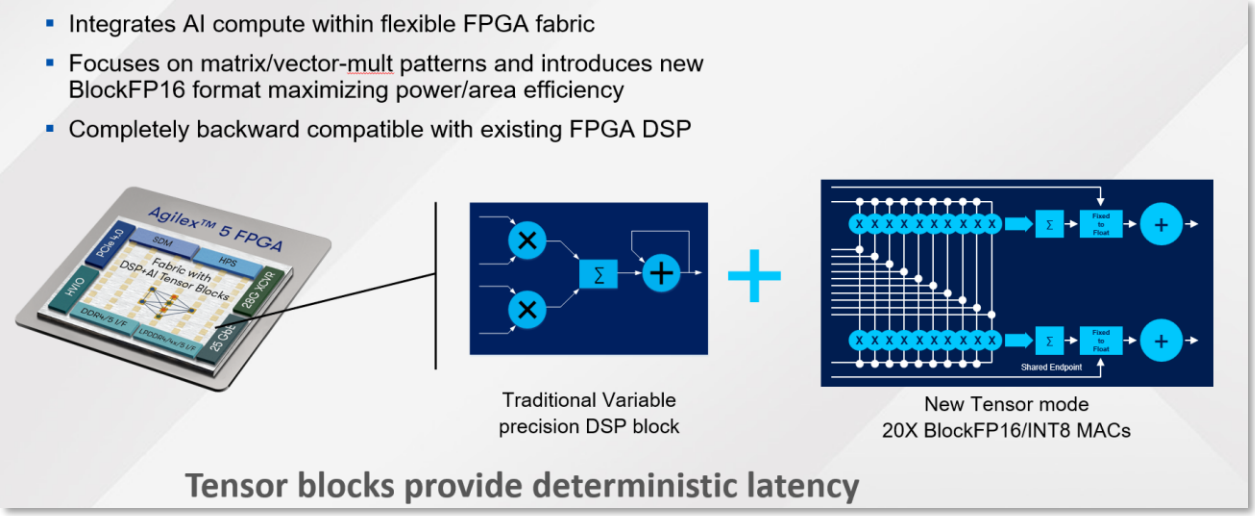
- Faster Time-to-Market with evolving AI Networks
- FPGA reconfigurability provides flexibility for customization as the use case evolves
- Adapt to new networks, data types
- Long Product Life Cycles

Step 7: Why target Agilex 3 FPGAs for ML?

Agilex 3 parts are Altera's newest power and cost optimized FPGAs.



Agilex 3 devices have DSP with AI Tensor Blocks to increase compute density.



Step 8: Workshop Streams: One Ware and FPGA AI Suite

For the workshops, there are two streams that you can select to do hands-on ML work. The details for either stream are in separate files.

Stream One: Agilex 3 and OneWare Suite

- This flow is recommended if you are new to AI and/or you would like to see the entire AI flow (from start to finish), i.e. starting with requirements, obtaining and modifying the datasets, configuring a model, training the model (and seeing the accuracy), testing the model, exporting the flow, and testing with the FPGA.

- -You will interact with the AI model with your own handwriting to see if you get the expected results.

Stream Two: Agilex 3 and FPGA AI Suite

This flow will be for those who are more experienced with AI.

This workshop starts a simple model, does training, and where it uses command line to optimize the model before implementing it. FPGA AI suite is targeted for pre-trained models.