UNIT 1
CONCISE NOTES

NOTE: PLEASE REFER TO PPTs , REFERENCE DOCUMENTS & TEXTBOOKS AS WELL APART FROM THESE NOTES FOR COMPLETE PREPARATION

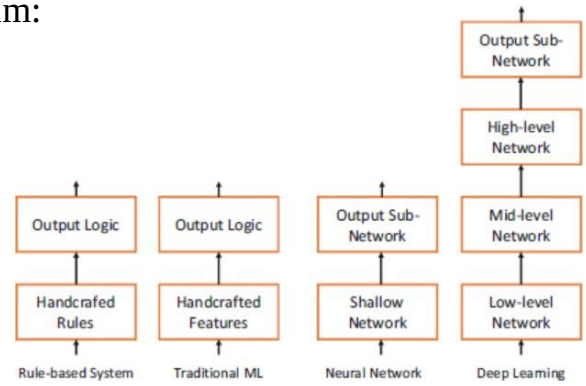
VARIOUS DEVELOPMENT STAGES OF ARTIFICIAL INTELLIGENCE:

- AI is a technology that allows machines to acquire intelligent and inferential mechanisms like humans. This concept first appeared at the Dartmouth Conference in 1956.The development of Al has mainly gone through three stages, where each stage represents the exploration of the human trying to realize Al from different angles.
- In the **first stage**, people tried to develop intelligent systems where they summarized and generalized some logical rules and implemented them in the form of computer programs. But those logical rules are too simple and were difficult to be used to express complex and abstract concepts. This stage is called the inference period.
- In the 1970s, scientists tried to implement AI through knowledge database and reasoning. They built a large and complex expert system to replicate the intelligence level of human experts. One of the biggest difficulties of inference period was that the process of human recognition of pictures and understanding of languages cannot be replicated by established rules.
- To solve these problems, a research discipline that allowed machines to automatically learn rules from data, known as machine learning, was born which became a popular subject in AI in the 1980s. This is the **second stage.**
- In machine learning, there is a direction to learn complex, abstract logic through neural networks.
- The **third revival of AI** was since 2012 when the applications of deep neural network technology have made major breakthroughs in fields like computer vision, natural language processing (NLP), and robotics.
- Some tasks have even surpassed the level of human intelligence.. Deep neural networks eventually got a new name – deep learning. The essential difference between neural networks and deep learning is not large as deep learning refers to models or algorithms based on deep neural networks.

THE ADVENT OF NEURAL NETWORKS AND DEEP LEARNING:

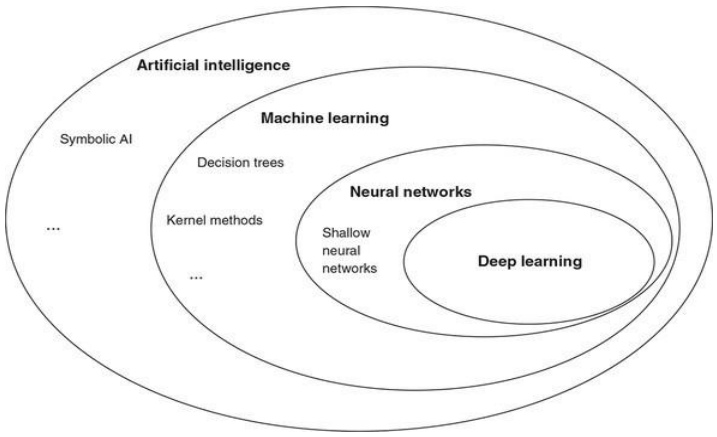
- Neural network algorithms are a class of algorithms that learn from data based on neural networks. They still belong to the category of machine learning. Due to the limitation of computing power and data volume, early neural networks were shallow.
- Therefore, the network expression ability was limited. With the improvement of computing power and the arrival of the big data and highly parallelized graphics processing units (GPUs) and make training of large-scale neural networks possible.
- In 2006, Geoffrey Hinton first proposed the concept of deep learning.

- In 2012, Alex Net, an eight-layer deep neural network, was released and achieved huge performance improvements in the image recognition competition. Since then, neural network models with thousands of layers have been developed successively, showing strong learning ability.
- Algorithms implemented using deep neural networks are generally referred to as deep learning models. In essence, neural networks and deep learning can be considered the same.
- The comparison of deep learning with other algorithms can be explained with the help of following diagram:



- The rule-based systems usually write explicit logic, which is generally designed for specific tasks and is not suitable for other tasks.
- Traditional machine learning algorithms artificially design feature detection methods with certain generality, such as SIFT and HOG features which are suitable for a certain type of tasks and have certain generality. But the performance highly depends on how those features are designed.
- The emergence of neural networks has made it possible for computers to design those features automatically through neural networks without human intervention.
- Shallow neural networks typically have limited feature extraction capability, while deep neural networks are capable of extracting high-level, abstract features and have better performance.

RELATIONSHIP BETWEEN AI , DEEP LEARNING AND MACHINE LEARNING.

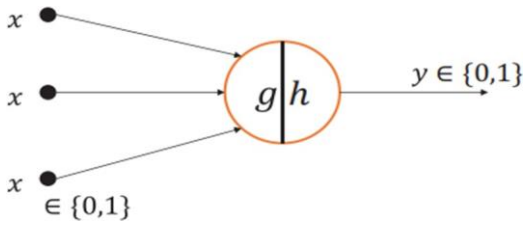


AI is a technology that allows machines to acquire intelligent and inferential mechanisms like humans.'

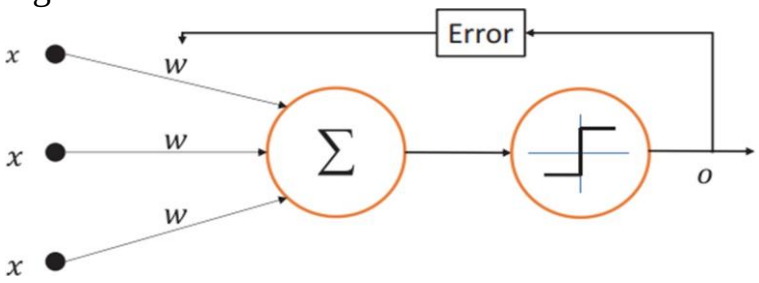
- The people tried to develop intelligent systems where they summarized and generalized some logical rules and implemented them in the form of computer programs. But those logical rules are too simple and were difficult to be used to express complex and abstract concepts.
- To solve inference period problems, a research discipline allowed machines to automatically learn rules from data, known as machine learning.
- In machine learning, there is a direction to learn complex, abstract logic through neural networks.
- Deep neural networks eventually got a new name – deep learning.

SHALLOW NEURAL NETWORKS ALONG WITH ITS DEVELOPMENT TIMELINE:

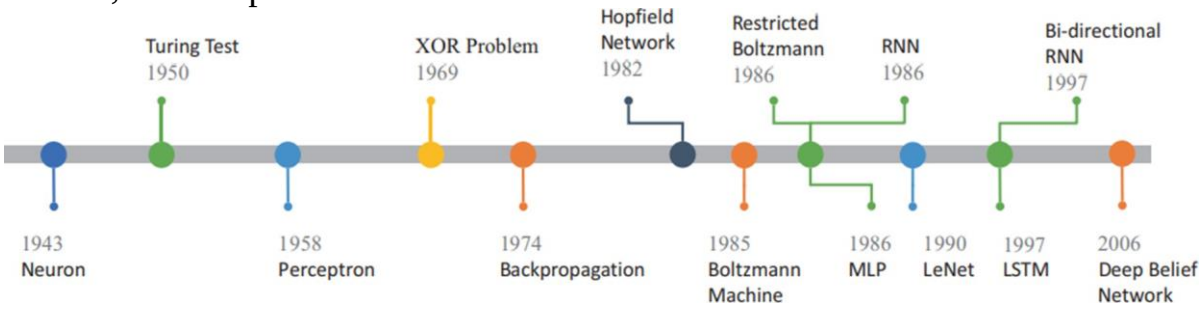
- We divide the development of neural networks into shallow neural network stages and deep learning stages.
- In 1943, psychologist Warren McCulloch and logician Walter Pitts proposed the earliest mathematical model of neurons based on the structure of biological neurons, called MP neuron models after their last name initials.
- The model f (x) = h (g(x)), where g(x) = ∑xi, xi ∈ {0, 1}, takes values from g(x) to predict output values.
- If g(x) ≥ 0, output is 1; if g(x) < 0, output is 0.
- The MP neuron models have no learning ability and can only complete fixed logic judgments.



- In 1958, American psychologist Frank Rosenblatt proposed the first neuron model that can automatically learn weights, called perceptron. The error between the output value o and the true value y is used to adjust the weights of the neurons {w1,w2,…,wn}. Frank Rosenblatt then implemented the perceptron model based on the "Mark 1 perceptron" hardware. The input is an image sensor with 400 pixels, and the output has eight nodes.it can finally identify English letters.

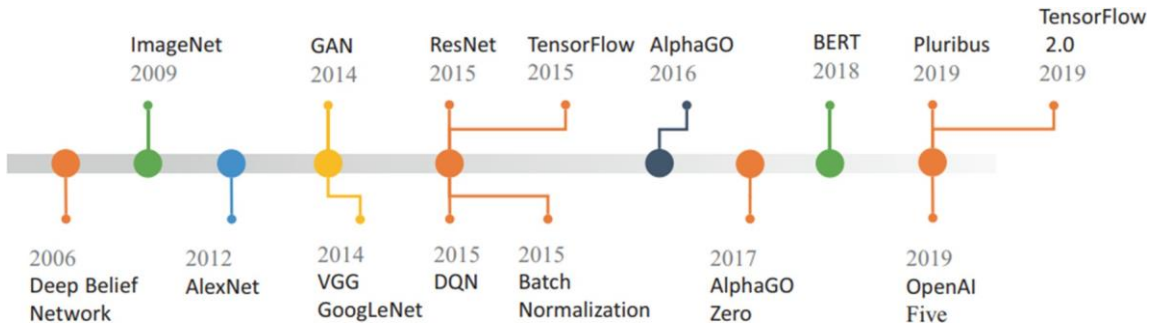


- The main flaw of linear models such as perceptrons is that perceptrons cannot handle simple linear inseparable problems such as XOR. This directly led to the tough period of perceptron-related research on neural networks. It is generally considered that 1969–1982 was the first winter of artificial intelligence.
- Although it was in the tough period of AI, there were still many significant studies published such as back propagation (BP) algorithm, which is the core foundation of modern deep learning algorithms. In fact, the mathematical idea of the BP algorithm has been derived as early as the 1960s, but it had not been applied to neural networks at that time.
- In 1974, American scientist Paul Werbos first proposed that the BP algorithm can be applied to neural networks in his doctoral dissertation. Unfortunately, this result has not received enough attention. In 1986, David Rumelhart et al. published a paper using the BP algorithm for feature learning in Nature. Since then, the BP algorithm started gaining widespread attention.
- During the second wave of artificial intelligence renaissance that started from 1982 to 1995, convolutional neural networks, recurrent neural networks, and backpropagation algorithms were developed. In 1986, David Rumelhart, Geoffrey Hinton, and others applied the BP algorithm to multilayer perceptrons.
- In 1989, Yann LeCun and others applied the BP algorithm to handwritten digital image recognition and achieved great success, which is known as LeNet. The LeNet system was successfully commercialized in zip code recognition, bank check recognition, and many other systems.
- In 1997, one of the most widely used recurrent neural network variants, Long ShortTerm Memory (LSTM), was proposed by Jürgen Schmidhuber. In the same year, a bidirectional recurrent neural network was also proposed.
- Unfortunately, the study of neural networks has entered a tough with the rise of traditional machine learning algorithms represented by support vector machines (SVMs), which is known as the second winter of artificial intelligence.
- Support vector machines have a rigorous theoretical foundation, require a small number of training samples, and also have good generalization capabilities. In contrast, neural networks lack theoretical foundation and are hard to interpret. Deep networks are difficult to train, and the performance is normal.



DEEP NEURAL NETWORKS ALONG WITH ITS DEVELOPMENT TIMELINE:

- We divide the development of neural networks into shallow neural network stages and deep learning stages, with 2006 as the dividing point.
- In 2006, Geoffrey Hinton found that multilayer neural networks can be better trained through layer-by-layer pre-training and achieved a better error rate than SVM on the MNIST handwritten digital picture data set, turning on the third artificial intelligence revival.
- In 2011, Xavier Glorot proposed a Rectified Linear Unit (ReLU) activation function, which is one of the most widely used activation functions now
- In 2012, Alex Krizhevsky proposed an eight-layer deep neural network AlexNet, which used the ReLU activation function and Dropout technology to prevent overfitting.
- Since the AlexNet model was developed, various models have been published successively, including VGG series, GoogleNet series, ResNet series, and DenseNet series.
- In 2014, Ian Goodfellow proposed generative adversarial networks (GANs), which learned the true distribution of samples through adversarial training to generate samples with higher approximation. Since then, a large number of GAN models have been proposed.
- In 2016,DeepMind applied deep neural networks to the field of reinforcement learning and proposed the DQN algorithm, which achieved a level comparable to or even higher than that of humans in 49 games in the Atari game platform.
- In the field of Go, AlphaGo and AlphaGo Zero intelligent programs from DeepMind have successively defeated human top Go players Li Shishi, Ke Jie, etc.
- In the multi-agent collaboration Dota 2 game platform, OpenAI Five intelligent programs developed by OpenAI defeated the TI8 champion team OG in a restricted game environment, showing a large number of professional high-level intelligent operations. Figure 1-9lists the major time points between 2006 and 2019 for AI development.
- Timeline for deep learning development:

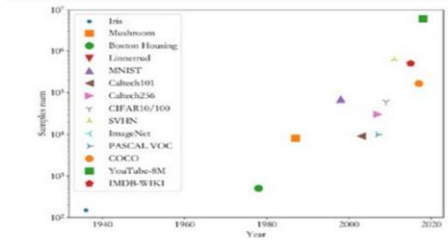


CHARACTERISTICS MODERN DEEP LEARNING ALGORITHMS:

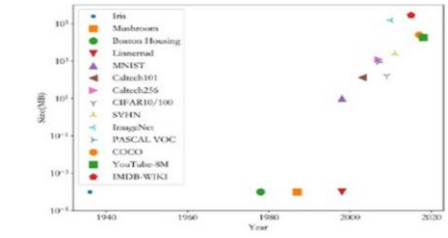
Compared with traditional machine learning algorithms and shallow neural networks, modern deep learning algorithms usually have the following characteristics.
Following are the characteristics of modern deep learning algorithms which make it better to use:

- ***DATA VOLUME*** :

- Early machine learning algorithms are relatively simple and fast to train, and the size of the required dataset is relatively small, such as the Iris flower dataset.
- With the development of computer technology, the designed algorithms are more and more complex, and the demand for data volume is also increasing. With the rise of neural networks, especially deep learning networks, the number of network layers and model parameters are large.
- To prevent over fitting, the size of the training dataset is usually huge.Although deep learning has a high demand for large datasets, collecting data, especially collecting labeled data, is often very expensive.
- The formation of a dataset usually requires manual collection, crawling of raw data and cleaning out invalid samples, and then annotating the data samples with human intelligence, so subjective bias and random errors are inevitably introduced.



Dataset sample size change over time

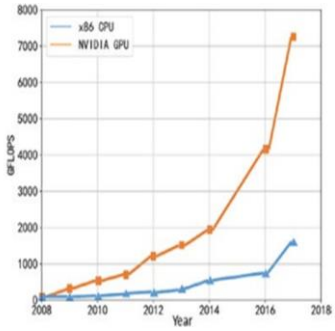


Dataset size change over time

- ***COMPUTING POWER :***

- The increase in computing power is an important factor in the third artificial intelligence renaissance.
- The real potential of deep learning was not realized until the release of AlexNet.
- Traditional machine learning algorithms do not have stringent requirements on data volume and computing power like deep learning. But deep learning relies heavily on parallel acceleration computing devices.
- Most of the current neural networks use parallel acceleration chips such as NVIDIA GPU and Google TPU to train model parameters.

- For example, the AlphaGo Zero program needs to be trained on 64 GPUs from scratch for 40 days before surpassing all AlphaGo historical versions. At present, the deep learning acceleration hardware devices that ordinary consumers can use are mainly from NVIDIA GPU graphics cards.



. NVIDIA GPU FLOPS change (data source: NVIDIA)

- This shows the variation of one billion floating-point operations per second (GFLOPS) of NVIDIA GPU and x86 CPU from 2008 to 2017. It can be seen that the curve of x86 CPU changes relatively slowly, and the floating-point computing capacity of the NVIDIA GPU grows exponentially, which is mainly driven by the increasing business of game and deep learning computing.

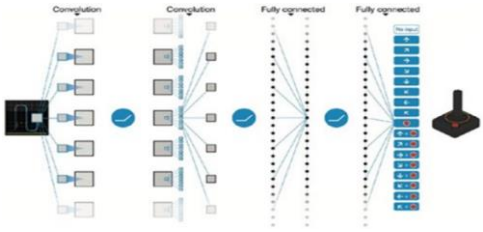
➢ ***NETWORK SCALE:***

- Early perceptron models and multilayer neural networks only have one or two to four layers, and the network parameters are also around tens of thousands.
- With the development of deep learning and the improvement of computing capabilities, models such as AlexNet (8 layers), VGG16 (16 layers), GoogleNet (22 layers), ResNet50 (50 layers), and DenseNet121 (121 layers) have been proposed successively, while the size of inputting pictures has also gradually increased from 28×28 to 224×224 to 299×299 and even larger.
- The increase of network scale enhances the capacity of the neural networks correspondingly, so that the networks can learn more complex data modalities and the model performance can be improved accordingly.
- On the other hand, the increase of the network scale also means that we need more training data and computational power to avoid over fitting.

➢ ***GENERAL INTELLIGENCE:***

- Designing a universal intelligent mechanism that can automatically learn and self-adjust like the human brain has always been the common vision of human beings.
- Deep learning is one of the algorithms closest to general intelligence. In the computer vision field, previous methods that need to design features for specific tasks and add a priori assumptions have been abandoned by deep learning algorithms.
- At present, almost all algorithms in image recognition, object detection, and semantic segmentation are based on end-to-end deep learning models, which present good performance and strong adaptability.

- On the Atari game platform, the DQN algorithm designed by DeepMind can reach human equivalent level in 49 games under the same algorithm, model structure, and hyperparameter settings, showing a certain degree of general intelligence.
- DQN network structure:



DEEP NEURAL NETWORKS VS SHALLOW NEURAL NETWORKS:

- The basic structure of a simple neural network in modern applications consists of three layers: an input layer, a hidden layer (or middle layer), and an output layer.
- The input layers consist of the number of attributes or values that an input consists of. **Example**: if the input is a 20x20 pixel image, the input layer will consist of 400 input nodes, that each represent a pixel.
- The middle layer consists of one or more hidden layers, which are responsible for the majority of the transformations on the input data into output signals, depending on their various synaptic weights and activation function.
- The last layer, the output layer, combines all the signals or outputs from the last hidden layer and performs a classification or output transformation.

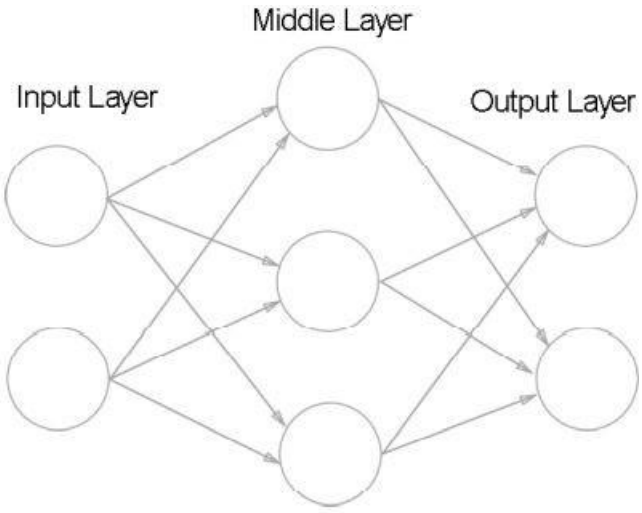


Figure 3.1: Neural network example

- A Shallow Neural Network is a simple neural network that consist of only 1 or 2 hidden layers.

- A Deep Neural Network is a simple neural network that consist of multiple hidden layers.
- Both models suffer from overfitting or poor generalization in many cases. Deep networks include more hyper-parameters than shallow ones that increase the overfitting probability.

➢ **Differences between SNN and DNN**

- In addition to the input and output layers, neural networks have an intermediate layer, also known as a hidden layer.
- shallow networks have less hidden layers. Because of which there will be a significant increase in the number of parameters when dealing with complex functions.
- Deep Neural Networks have multiple hidden layers which helps to fit complex function better with less parameters compared to a shallow network.
- Shallow neural networks process the features directly, while deep networks extract features automatically along with the training.
- The structure of shallow networks allows them to learn important features independently from other features, which is best suited for learning tasks dealing with data with low dimensionality and a relatively small number of features.
- Deep neural networks excel in very complex tasks that have large input data and high dimensionality.
- These networks have multiple hidden layers, with lower layers learning low-level features and higher layers learning higher-level features which are dependent on the patterns and knowledge derived from the lower layers. The layer dependent factor of deep neural networks means that learning is dependent from layer to layer and the hidden layers reuse the learned features from lower layers to learn more complex features and solve more complex tasks
- Deep learning works best when the training set is huge and feature set is complex.
- Shallow network works best when data has low dimensionality and relatively less important feature set.
- Deep neural networks excel in areas such as computer vision tasks, speech recognition, and signals processing
- Shallow neural networks are widely used for simple regression tasks.

DEEP LEARNING AND ITS APPLICATIONS:

- Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input.

Example: In image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

- A Deep learning process can learn which features to optimally place in which level on its own.
- Each level learns to transform its input data into a slightly more abstract and composite representation.
- This does not eliminate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.

Applications of Deep Learning

DAILY LIFE APPLICATIONS:

1. voice assistants in mobile phones.
2. intelligent assisted driving in cars.

MAIN STREAM APPLICATIONS:

1. Computer vision
2. NLP
3. Reinforcement learning

Computer Vision Applications

1. Image classification
2. Object detection
3. Semantic segmentation
4. Video Understanding

Natural Language Processing Applications

1. Machine Translation
2. Chatbot

Reinforcement Learning Applications

1. Virtual Games.
2. Robotics
3. Autonomous driving

Image classification

- The input of the neural network is pictures, and the output value is the probability that the current sample belongs to each category.
- Generally, the category with the highest probability is selected as the predicted category of the sample.

Chatbot

- Mainstream task of natural language processing.
- Machines automatically learn to talk to humans, provide satisfactory automatic responses to simple human demands, and improve customer service efficiency and service quality.
- Neural network is trained to generate appropriate responses for input questions.

- Chatbot is often used in consulting systems, entertainment systems, and smart homes.

Virtual Games

- virtual game platforms can both train and test reinforcement learning algorithms
- Neural network trains and learns in a complex environment on the basis of reward function.
- avoid interference from irrelevant factors while also minimizing the cost of experiments

MAIN STREAM APPLICATIONS OF DEEP LEARNING:

Deep learning algorithms have been widely used in our daily life, such as voice assistants in mobile phones, intelligent assisted driving in cars, and face payments. We will introduce some mainstream applications of deep learning starting with computer vision, natural language processing, and reinforcement learning.

- **Fraud detection:**
  - Fraud is a growing problem in the digital world. In 2021, consumers reported 2.8 million cases of fraud to the Federal Trade Commission. Identify theft and imposter scams were the two most common fraud categories.
  - To help prevent fraud, companies like Signifyd use deep learning to detect anomalies in user transactions. Those companies deploy deep learning to collect data from a variety of sources, including the device location, length of stride and credit card purchasing patterns to create a unique user profile.
  - Mastercard has taken a similar approach, leveraging its Decision Intelligence and AI Express platforms to more accurately detect fraudulent credit card activity. And for companies that rely on e-commerce, Riskified is making consumer finance easier by reducing the number of bad orders and chargebacks for merchants. **Relevant companies:** Neurala, ZeroEyes, Motional

- **Computer Vision:**
  - Computer Vision Image classification is a common classification. The input of the neural network is pictures, and the output value is the probability that the current sample belongs to each category.
  - Generally, the category with the highest probability is selected as the predicted category of the sample.

- **Image recognition** is one of the earliest successful applications of deep learning. Classic neural network models include VGG series, Inception series, and ResNet series.
- **Object detection** refers to the automatic detection of the approximate location of common objects in a picture by an algorithm. It is usually represented by a bounding box and classifies the category information of objects in the bounding box. Common object detection algorithms are RCNN, Fast RCNN, Faster RCNN, Mask RCNN, SSD, and YOLO series.
- **Semantic segmentation** is an algorithm to automatically segment and identify the content in a picture. We can understand semantic segmentation as the classification of each pixel and analyze the category information of each pixel. Common semantic segmentation models include FCN, U-net, SegNet, and DeepLab series.
- **Video Understanding.** As deep learning achieves better results on 2D picture–related tasks, 3D video understanding tasks with temporal dimension information (the third dimension is sequence of frames) are receiving more and more attention. Common video understanding tasks include video classification, behavior detection, and video subject extraction. Common models are C3D, TSN, DOVF, and TS_LSTM.
- **Image generation** learns the distribution of real pictures and samples from the learned distribution to obtain highly realistic generated pictures. At present, common image generation models include VAE series and GAN series. Among them, the GAN series of algorithms have made great progress in recent years.

➢ Agriculture

- Agriculture will remain a key source of food production in the coming years, so people have found ways to make the process more efficient with deep learning and AI tools. AI to detect intrusive wild animals, forecast crop yields and power self-driving machinery.
- Blue River Technology has explored the possibilities of self-driven farm products by combining machine learning, computer vision and robotics. The results have been promising, leading to smart machines — like a lettuce bot that knows how to single out weeds for chemical spraying while leaving plants alone. In addition, companies like Taranis blend computer vision and deep learning to monitor fields and prevent crop loss due to weeds, insects and other causes.
- Relevant Companies: Blue River Technology, Taranis

- **Natural language processing**
  - The introduction of natural language processing technology has made it possible for robots to read messages and divine meaning from them. Still, the process can be somewhat oversimplified, failing to account for the ways that words combine together to change the meaning or intent behind a sentence.
  - Deep learning enables natural language processors to identify more complicated patterns in sentences to provide a more accurate interpretation. Companies use deep learning to power a chatbot that is able to respond to a larger volume of messages and provide more accurate responses. Grammarly also uses deep learning in combination with grammatical rules and patterns to help users identify the errors and tone of their messages.
    Relevant companies: Gamalon, Strong, Grammarly

- **Autonomous vehicles**
- Driving is all about taking in external factors like the cars around you, street signs and pedestrians and reacting to them safely to get from point A to B. While we're still a ways away from fully autonomous vehicles, deep learning has played a crucial role in helping the technology come to fruition.
- It allows autonomous vehicles to take into account where you want to go, predict what the obstacles in your environment will do and create a safe path to get you to that location.
  Relevant companies: Zoox, Tesla, Waymo

- **Climate change**
- Organizations are stepping up to help people adapt to quickly accelerating environmental change. One Concern has emerged as a climate intelligence leader, factoring environmental events such as extreme weather into property risk assessments.
- Meanwhile, NCX has expanded the carbon-offset movement to include smaller landowners by using AI technology to create an affordable carbon marketplace.
- **Entertainment**
- Streaming platforms aggregate tons of data on what content you choose to consume and what you ignore. Take Netflix as an example. The streaming platform uses machine learning to find patterns in what its viewers watch so that it can create a personalized experience for its users.
  Relevant companies: Amazon, Netflix

VARIOUS DEEP LEARNING FRAMEWORKS:

1) **Theano** is one of the earliest deep learning frameworks. It was developed by Yoshua Bengio and Ian Goodfellow. It is a Python-based computing library for positioning low-level operations. Theano supports both GPU and CPU operations. Due to Theano's low development efficiency, long model compilation time, and developers switching to TensorFlow, Theano has now stopped maintenance.
2) **Scikit-learn** is a complete computing library for machine learning algorithms. It has built-in support for common traditional machine learning algorithms, and it has rich documentation and examples. However, scikit-learn is not specifically designed for neural networks. It does not support GPU acceleration, and the implementation of neural network–related layers is also lacking.
3) **Caffe** was developed by Jia Yangqing in 2013. It is mainly used for applications using convolutional neural networks and is not suitable for other types of neural networks. Caffe's main development language is C ++, and it also provides interfaces for other languages such as Python. It also supports GPU and CPU. Due to the earlier development time and higher visibility in the industry, in 2017 Facebook launched an upgraded version of Caffe, Caffe2. Caffe2 has now been integrated into the PyTorch library.
4) **Torch** is a very good scientific computing library, developed based on the less popular programming language Lua. Torch is highly flexible, and it is easy to implement a custom network layer, which is also an excellent gene inherited by PyTorch. However, due to the small number of Lua language users, Torch has been unable to obtain mainstream applications.
5) **Keras** is a high-level framework implemented based on the underlying operations provided by frameworks such as Theano and TensorFlow. It provides a large number of high-level interfaces for rapid training and testing. For common applications, developing with Keras is very efficient. But because there is no low-level implementation, the underlying framework needs to be abstracted, so the operation efficiency is not high, and the flexibility is average.
   Keras can be understood as a set of high-level API design specifications. Keras itself has an official implementation of the specifications. The same specifications are also implemented in TensorFlow, which is called the tf.keras module, and tf.keras will be used as the unique high-level interface to avoid interface redundancy.
6) **TensorFlow** is a deep learning framework released by Google in 2015. The initial version only supported symbolic programming. Thanks to its earlier release and Google's influence in the field of deep learning, TensorFlow quickly became the most popular deep learning framework. However, due to frequent changes in the interface design, redundant functional design, and difficulty in symbolic programming development and debugging,

TensorFlow 1.x was once criticized by the industry. In 2019, Google launched the official version of TensorFlow 2, which runs in dynamic graph priority mode and can avoid many defects of the TensorFlow 1.x version. TensorFlow 2 has been widely recognized by the industry.At present, TensorFlow and PyTorch are the two most widely used deep learning frameworks in industry. TensorFlow has a complete solution and user base in the industry. Thanks to its streamlined and flexible interface design, PyTorch can quickly build and debug networks, which have received rave reviews in academia. After TensorFlow 2 was released, it makes it easier for users to learn TensorFlow and seamlessly deploy models to production.

REGRESSION : NEURON MODEL

Regression is a ML algorithm which is used to find the relationship between a dependent variable and other independent variables.

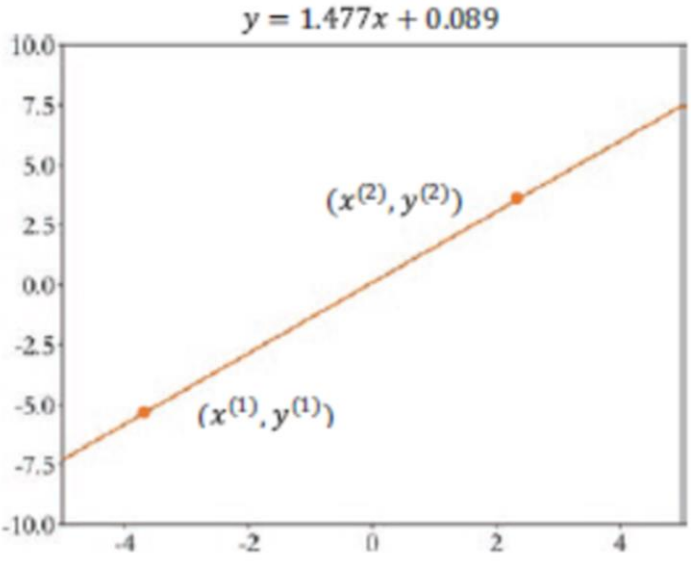
1. If we consider a Simple Linear Regression ,the number of parameters are one.
2. When the number of input nodes $n = 1$ (single input), the neuron model can be further simplified as

$$y = wx + b$$

w-slope of the straight line

b-bias of the straight line

3. Then we can plot the change of y as a function of x , which is in the form of a straight line.
4. In order to estimate the value of w and b, we only need to sample any two data points ($x(1)$, $y(1)$) and ($x(2)$, $y(2)$) from the straight line.



Single-input linear neuron model

$$y^{(1)} = wx^{(1)} + b$$

$$y^{(2)} = wx^{(2)} + b$$

- If, $(x^{(1)}, y^{(1)}) \neq (x^{(2)}, y^{(2)})$ we can solve the preceding equations to get the value of *w* and *b*. Let's consider a specific example:

$$1.567 = w \cdot 1 + b$$

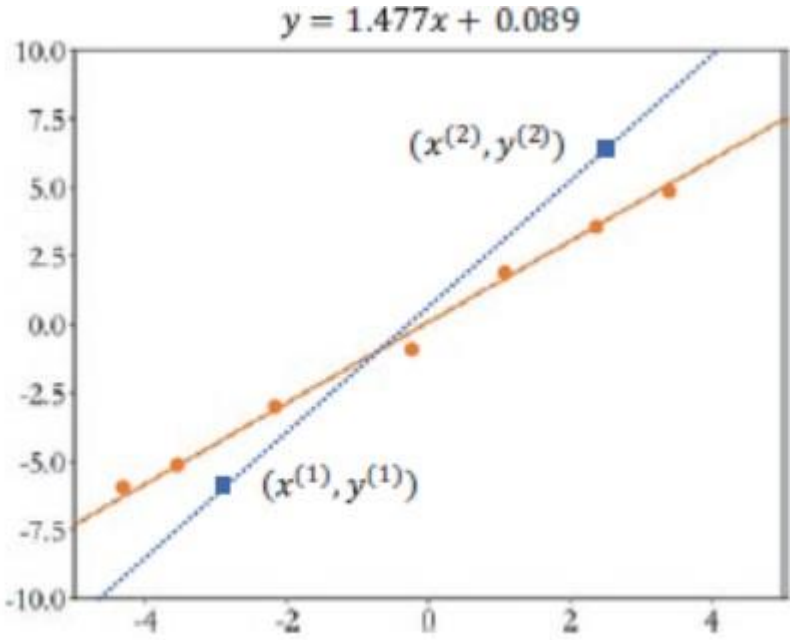
$$3.043 = w \cdot 2 + b$$

- Solving these 2 linear equations we obtain the values of w=1.477 and b=0.089
- If we consider other than simple linear regression model, the number of data points may or may not be two.
- For linear neuron models with *N* input, we only need to sample *N* + 1 different data points.
- Practically, there may be **observation errors** for any sampling point, we assume that the observation error variable ϵ follows a normal distribution $\mathcal{N}(\mu, \sigma^2)$
- where μ is mean and $\sigma 2$ is variance. Then the samples follow:

$$y = wx + b + \epsilon, \epsilon \sim \mathcal{N}(\mu, \sigma^2)$$

Effect of Estimation Bias:

- Once the observation error is introduced, even if it is as simple as a linear model, if only two data points are sampled, it may bring a large estimation bias



. *Model with observation errors*

If the estimation is based on the two blue rectangular data points, the estimated blue dotted line would have a larger deviation from the true orange straight line.

- In order to reduce the estimation bias introduced by observation errors, we can sample multiple data points.

$$\mathbb{D} = \left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$$

- We should find the best straight line, so that it minimizes the sum of errors between all sampling points and the straight line.
- Due to the existence of observation errors, there may not be a straight line that perfectly passes through all the sampling points **D**.
- Therefore, we should be able to find a good straight line cloe to all sampling points with the help of **Mean Squared Error(MSE)**

- Mean squared error (MSE) is calculated between the predicted value $wx(i) + b$ and the true value $y(i)$ at all sampling points as the total error, that is

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2$$

- Then search a set of parameters $w*$ and $b*$ to minimize the total error 'L'.
- The straight line corresponding to the minimal total error is the optimal straight line we are looking for, that is

$$w^*, b^* = \arg\min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \left(wx^{(i)} + b - y^{(i)} \right)^2$$

- Therefore,Optimal straight line equation is :

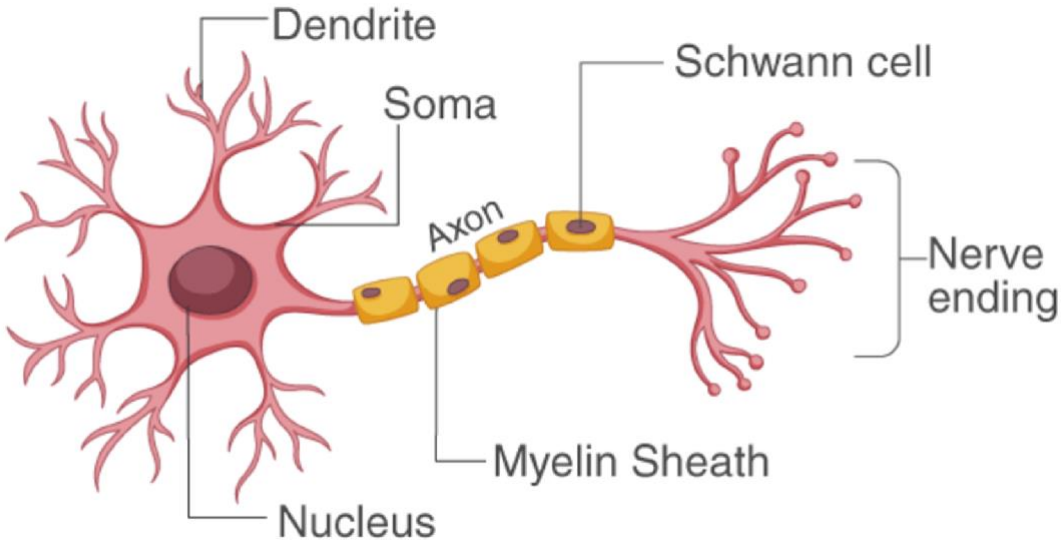
$$w^*, b^* = \arg\min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \left(wx^{(i)} + b - y^{(i)} \right)^2$$

- Where n= number of samples

BIOLOGICAL NEURON VS BASIC NEURON MODEL:

Neurons are the building blocks of the nervous system. They receive and transmit signals to different parts of the body. This is carried out in both physical and electrical forms.

- **Structure of a biological Neuron:**



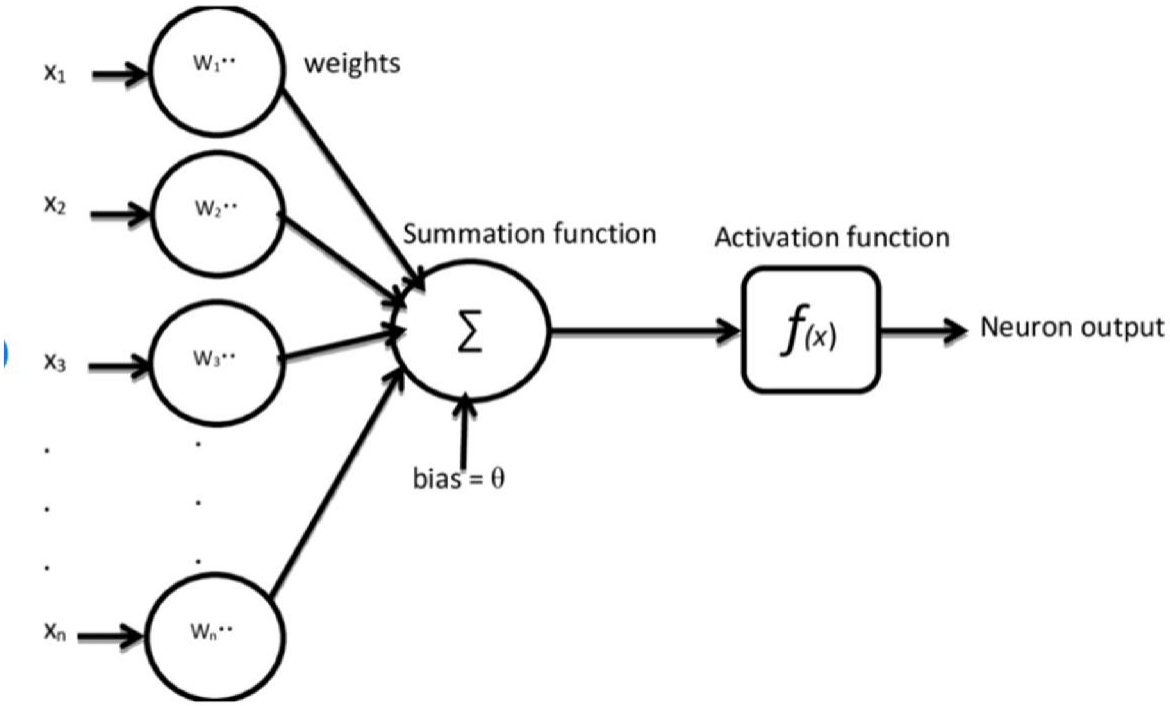
- **Biological Neural Network :**
  Biological Neural Network (BNN) is a structure that consists of Synapse, dendrites, cell body, and axon. In this neural network, the processing is carried out by neurons. Dendrites receive signals from other neurons, Soma sums all the incoming signals and axon transmits the signals to other cells.

Artificial Neuron:

An artificial neuron is a connection point in an artificial neural network. Artificial neural networks, like the human body's biological neural

network, have a layered architecture and each network node (connection point) has the capability to process input and forward output to other nodes in the network.

- **Structure of an Artificial Nueron:**



- **Artificial Neural Network:**
  Artificial Neural Network (ANN) is a type of neural network which is based on a Feed-Forward strategy. It is called this because they pass information through the nodes continuously till it reaches the output node. This is also known as the simplest type of neural network.

CONCEPT OF GRADIENT ,GRADIENT VECTOR & GRADIENT DESCENT:

Gradient / derivative is the function that tells you the slope or rate of change of the line that is tangent to the curve at any given point. **The gradient points to the direction of greatest increase; keep following the gradient, and you will reach the local maximum.** The gradient of a function is defined as a vector of partial derivatives of the function on each independent variable.

Considering a three dimensional function z = f (x, y), the partial derivative of the function with respect to the independent variable x is $\partial z/\partial x$, the partial derivative of the function with respect to the independent variable y is recorded as $\partial z/\partial y$, and the gradient ∇f is a vector $(\partial z/\partial x , \partial z/\partial y)$. Let's look at a specific function f (x, y) = − (cos2 x + cos2 y)2 . As shown in Figure 2-6, the length of the red arrow in the plane represents the modulus of

the gradient vector, and the direction of the arrow represents the direction of the gradient vector.
Mathematical representation of gradient vector:

$$\nabla f(x)=(\partial f/\partial x1\ (x)\ ,\ \partial f/\partial x2\ (x)\ ,\ \cdots\ ,\ \partial f/\partial xn\ (x))$$

It can be seen that the direction of the arrow always points to the function value increasing direction. The steeper the function surface, the longer the length of the arrow, and the larger the modulus of the gradient.

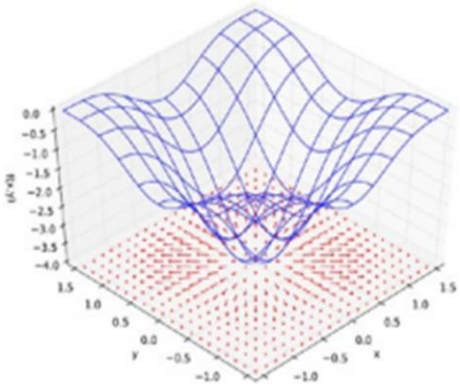


Figure 2-6. A function and its gradient

Generally the gradient direction of the function always points to the direction in which the function value increases. Then the opposite direction of the gradient should point to the direction in which the function value decreases.

$$\boldsymbol{x}' = \boldsymbol{x} - \eta \cdot \nabla f \qquad (2.1)$$

To take advantage of this property, we just need to follow the preceding equation to iteratively update x′ . Then we can get smaller and smaller function values. η is used to scale the gradient vector, which is known as learning rate and generally set to a smaller value, such as 0.01 or 0.001. In particular, for one-dimensional functions, the preceding vector form can be written into a scalar form:

$$x' = x - \eta \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

By iterating and updating x′ several times through the preceding formula, the function value y′ at x′ is always more likely to be smaller than the function value at x. The method of optimizing parameters by the formula (2.1) is called the gradient descent algorithm.

It calculates the gradient ∇f of the function f and iteratively updates the parameters θ to obtain the optimal numerical solution of the parameters θ when

the function f reaches its minimum value. It should be noted that model input in deep learning is generally represented as x and the parameters to be optimized are generally represented by θ, w, and b.

Now we will apply the gradient descent algorithm to calculate the optimal parameters w∗ and b∗ in the beginning of this session. Here the mean squared error function is minimized:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2$$

The model parameters that need to be optimized are w and b, so we update them iteratively using the following equations:

$$w' = w - \eta\frac{\partial \mathcal{L}}{\partial w}$$

$$b' = b - \eta\frac{\partial \mathcal{L}}{\partial b}$$

FEATURES OF VANILLA,STOCHASTIC AND MINI BATCH GRADIENT DECENT ALGORITHMS:

Vanilla Gradient Descent	**Stochastic Gradient Descent**	**Mini Batch Gradient Decent**
Computes gradient using the whole Training sample	Computes gradient using a single Training sample	Computes gradient using the Subset of Training sample
Slow and computationally expensive algorithm	Faster and less computationally expensive than Vanilla GD	Computation time is lesser than SGD Computation cost is lesser than Vanilla Gradient Descent
Not suggested for huge training samples.	Can be used for large training samples. But it maybe slow when datasets are huge.	Can be used for large training samples and it is also faster than SGD.
Cost Function reduces smoothly	Lot of variations in cost function	Smoother cost function as compared to SGD

Vanilla Gradient Descent	**Stochastic Gradient Descent**	**Mini Batch Gradient Decent**
Gives optimal solution given sufficient time to converge.	Gives good solution but not optimal.	Gives optimal solution in less time compared to SGD
No random shuffling of points are required.	The data sample should be in a random order, and this is why we want to shuffle the training set for every epoch.	The Data sample is Shuffled in a random order and then divided into batches.
Can't escape shallow local minima easily.	SGD can escape shallow local minima more easily.	Mini Batch can escape local minima easily compared to vanilla Gradient Descent
Convergence is slow.	Reaches the convergence much faster.	At times mini batch can reach convergence faster than SGD.
Vanilla Gradient descent is generally used for Small databases that fit into computer memory	SGD is a basis of more advanced stochastic algorithms used in training artificial neural networks	Mini batch gradient descent is most commonly used in practical applications

- Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.
- Batch Gradient descent can prevent the noisiness of the gradient, but we can get stuck in local minima and saddle points
- With stochastic gradient descent we have difficulties to settle on a global minimum, but usually, don't get stuck in local minima

- The mini-batch approach is the default method to implement the gradient descent algorithm in Deep Learning. It combines all the advantages of other methods, while not having their disadvantages.

Advantages of Mini-Batch Gradient Descent :

1. **Computational Efficiency**: In terms of computational efficiency, this technique lies between the two previously introduced techniques.
2. **Stable Convergenc**e: Another advantage is the more stable converge towards the global minimum since we calculate an average gradient over n samples that results in less noise.
3. **Faster Learning**: As we perform weight updates more often than with stochastic gradient descent, in this case, we achieve a much faster learning process.

- Analogous to the batch gradient descent we compute and average the gradients across the data instance in a mini-batch. The gradient descent step is performed after each mini-batch of samples has been processed.
- Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch.
- So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**
    1. Pick a mini-batch
    2. Feed it to Neural Network
    3. Calculate the mean gradient of the mini-batch
    4. Use the mean gradient we calculated in step 3 to update the weights
    5. Repeat steps 1–4 for the mini-batches we created.

Vanilla Gradient Descent

- This is the simplest form of gradient descent technique. Here, vanilla means pure / without any modification.
- Its main feature is that we take small steps in the direction of the minima by taking gradient of the cost function.
- There are high chances of getting struck in local minima. Therefore,Learning rate needs to be chosen very carefully.

Adam optimization algorithm

- The **Adam optimization algorithm** is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications.

- Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems
- Instead of adapting the parameter learning rates based on the average first moment (the mean). Adam also makes use of the average of the second moments of the gradients (the uncentered variance).

Main Advantage:

- Adam optimization's main advantage over Vanilla Gradient Descent is its capability of using both momentum and adaptive gradient.
- momentum takes Adam beyond the local minimum where vanilla gradient decent can get struck and then, adjustment from the sum of gradient can give Adam proper direction to explore and eventually finding the global minima.

S.No.	ANN	BNN
1.	It is short for Artificial Neural Network.	It is short for Biological Neural Network.
2.	Processing speed is fast as compared to Biological Neural Network.	They are slow in processing information.
3.	Allocation for Storage to a new process is strictly irreplaceable as the old location is saved for the previous process.	Allocation for storage to a new process is easy as it is added just by adjusting the interconnection strengths.
4.	Processes operate in sequential mode.	The process can operate in massive parallel operations.
5.	If any information gets corrupted in the memory it cannot be retrieved.	Information is distributed into the network throughout into sub-nodes, even if it gets corrupted it can be retrieved.
6.	The activities are continuously monitored by a control unit.	There is no control unit to monitor the information being processed into the network.

SIMPLE PYTHON PROGRAM TO ACHIEVE A MODEL OUTPUT Y=1.477*X+0.089+EPS (WITH RANDOM ERRORS)

```
import numpy as np
data = [] # A list to save data samples
for i in range(100): # repeat 100 times
 # Randomly sample x from a uniform distribution
```

```
  x = np.random.uniform(-10., 10.)
 # Randomly sample from Gaussian distribution
  eps = np.random.normal(0., 0.01)
 # Calculate model output with random errors
  y = 1.477 * x + 0.089 + eps
  data.append([x, y]) # save to data list
data = np.array(data) # convert to 2D Numpy array

def mse(b, w, points):
  # Calculate MSE based on current w and b
  totalError = 0
  # Loop through all points
  for i in range(0, len(points)):
    x = points[i, 0] # Get ith input
    y = points[i, 1] # Get ith output
    # Calculate the total squared error
    totalError += (y - (w * x + b)) ** 2
  # Calculate the mean of the total squared error
  return totalError / float(len(points))

def step_gradient(b_current, w_current, points, lr):
  # Calculate gradient and update w and b.
  b_gradient = 0
  w_gradient = 0
  M = float(len(points)) # total number of samples
  for i in range(0, len(points)):
      x = points[i, 0]
      y = points[i, 1]
   # dL/db:grad_b = 2(wx+b-y) from equation (2.3)
      b_gradient += (2/M) * ((w_current * x + b_current) - y)
   # dL/dw:grad_w = 2(wx+b-y)*x from equation (2.2)
      w_gradient += (2/M) * x * ((w_current * x + b_current) - y)
  # Update w',b' according to gradient descent algorithm
 # lr is learning rate
  new_b = b_current - (lr * b_gradient)
  new_w = w_current - (lr * w_gradient)
  return [new_b, new_w]

def gradient_descent(points, starting_b, starting_w, lr,num_iterations):
 # Update w, b multiple times
  b = starting_b # initial value for b
 w = starting_w # initial value for w
 # Iterate num_iterations time
```

```
  for step in range(num_iterations):
    # Update w, b once
    b, w = step_gradient(b, w, np.array(points), lr)
    # Calculate current loss
    loss = mse(b, w, points)
    if step%50 == 0: # print loss and w, b
     print(f"iteration:{step}, loss:{loss}, w:{w}, b:{b}")

  return [b, w] # return the final value of w and b
def main():
  # Load training dataset
  data = []
  for i in range(100):
    x = np.random.uniform(3., 12.)
# mean=0, std=0.1
    eps = np.random.normal(0., 0.1)
    y = 1.477 * x + 0.089 + eps
    data.append([x, y])
data = np.array(data)
lr = 0.01 # learning rate
initial_b = 0 # initialize b
initial_w = 0 # initialize w
num_iterations = 150
# Train 150 times and return optimal w*,b* and corresponding loss
[b, w]= gradient_descent(data, initial_b, initial_w, lr,num_iterations)
loss = mse(b, w, data) # Calculate MSE
print(f'Final loss:{loss}, w:{w}, b:{b}')
```

NUMERIC TENSOR

A numeric tensor is the main data format of TensorFlow. According to the dimension, it can be divided into:

• Scalar: A single real number, such as 1.2 and 3.4, has a dimension of 0 and a shape of [].

• Vector: An ordered set of real numbers, wrapped by square brackets.

• Matrix: An ordered set of real numbers in n rows and m columns.

• Tensor: An array with dimension greater than 2. Each dimension of the tensor is also known as the axis. Generally, each dimension represents specific physical meaning.

In TensorFlow, scalars, vectors, and matrices are also collectively referred to as tensors without distinction. You need to make your own judgement based on the dimension or shape of tensors.

*Creating a Tensor x of the value [0,-2,-1],[0,1,2]:

#Importing required libraries:

```
import tensorflow as tf
import numpy as np
In[44]:
x=tf.constant([[0,-2,-1],[0,1,2]])
print(x)

Out[44]:
tf.Tensor( [[ 0 -2 -1] [ 0 1 2]], shape=(2, 3), dtype=int32)
```

***Creating a Tensor y as a tensor of zeroes with the same shape as that of tensor x.**

```
In[34]:
Y=tf.fill([2,3], 0)
print(Y)

Out[34]:
tf.Tensor( [[0 0 0] [0 0 0]], shape=(2, 3), dtype=int32)
```

***Return a Boolean tensor that yields True if x equals y element wise.**

```
In[46]:
tf.math.equal(x, Y)
Out[46]:
<tf.Tensor: shape=(2, 3), dtype=bool, numpy=
array([[ True, False, False],
 [ True, False, False]])>
```

In TensorFlow, you can create tensors in a variety of ways, such as from a Python list, from a Numpy array, or from a known distribution.

(i) Create Tensors from Arrays and Lists:

Numpy array and Python list are very important data containers in Python. Many data are loaded into arrays or lists before being converted to tensors. The output data of TensorFlow are also usually exported to arrays or lists, which makes them easy to use for other modules.

The tf.convert_to_tensor function can be used to create a new tensor from a Python list or Numpy array.

For example:

```
 In [22]: # Create a tensor from a Python list
 tf.convert_to_tensor([1,2.])
Out[22]:<tf.Tensor: id=86, shape=(2,), dtype=float32, numpy=array([1.,2.],
dtype=float32)>
 In [23]: # Create a tensor from a Numpy array
tf.convert_to_tensor(np.array([[1,2.],[3,4]]))
Out[23]:
```

<tf.Tensor: id=88, shape=(2, 2), dtype=float64, numpy=
array([[1., 2.], [3., 4.]])>
Note that Numpy floating-point arrays store data with 64-bit precision by default. When converting to a tensor type, the precision is tf.float64. You can convert it to tf.float32 when needed.

(ii) Create All-0 or All-1 Tensors:
Creating tensors with all 0s or 1s is a very common tensor initialization method. Consider linear transformation y = Wx + b. The weight matrix W can be initialised with a matrix of all 1s, and b can be initialised with a vector of all 0s. So the linear transformation changes to y = x. We can use tf.zeros() or tf.ones() to create all-zero or all-one tensors with arbitrary shapes.
*Create a vector of all 0s and all 1s:
In [24]:
tf.zeros([]),tf.ones([])
Out[24]:
(<tf.Tensor: id=90, shape=(), dtype=float32, numpy=0.0>,<tf.Tensor: id=91, shape=(), dtype=float32, numpy=1.0>)
*Create a matrix of all 0's:
In [26]: tf.zeros([2,2])
Out[26]:
<tf.Tensor: id=104, shape=(2, 2), dtype=float32, numpy=
array([[0., 0.],
[0., 0.]], dtype=float32)>
*Create a matrix of all 1's:
In [26]: tf.ones([2,2])
Out[26]:
<tf.Tensor: id=108, shape=(2, 2), dtype=float32, numpy=
array([[1., 1.],
[1., 1.]], dtype=float32)>
With tf.zeros_like and tf.ones_like, you can easily create a tensor with all 0s or 1s that is consistent with the shape of another tensor.

(iii) Create a Customized Numeric Tensor:
In addition to initializing a tensor with all 0s or 1s, sometimes it is also necessary to initialize the tensor with a specific value, such as –1. With tf.fill(shape, value), we can create a tensor with a specific numeric value, where the dimension is specified by the shape parameter. For example, here's how to create a scalar with element –1:
In [30]:
tf.fill([], -1)
Out[30]:
<tf.Tensor: id=124, shape=(), dtype=int32, numpy=-1>

Create a vector with all elements –1:

```
In [31]:
        tf.fill([1], -1)
Out[31]:
        <tf.Tensor: id=128,  shape=(1,), dtype=int32, numpy=array([-1])>
```

(iv) Create a Tensor from a Known Distribution:

Sometimes, it is very useful to create tensors sampled from common distributions such as normal (or Gaussian) and uniform distributions. For example, in convolutional neural networks, the convolution kernel W is usually initialized from a normal distribution to facilitate the training process. In adversarial networks, hidden variables z are generally sampled from a uniform distribution

With tf.random.normal(shape, mean=0.0, stddev=1.0), we can create a tensor with dimension defined by the shape parameter and values sampled from a normal distribution N(mean,stddev^2). For example, here's how to create a tensor from a normal distribution with mean 0 and standard deviation of 1:

```
In [33]: tf.random.normal([2,2]) # Create a 2x2 tensor from a
normal distribution
Out[33]:
<tf.Tensor: id=143, shape=(2, 2), dtype=float32, numpy=
array([[-0.4307344 , 0.44147003], [-0.6563149 , -0.30100572]],
dtype=float32)>
```

*Create a tensor from a normal distribution with mean of 1 and standard deviation of 2:

```
In [34]: tf.random.normal([2,2], mean=1,stddev=2)
 Out[34]:
<tf.Tensor: id=150, shape=(2, 2), dtype=float32, numpy=
array([[-2.2687864, -0.7248812],
        [ 1.2752185, 2.8625617]], dtype=float32)>
```

With tf.random.uniform(shape, minval=0, maxval=None, dtype=tf. float32), we can create a uniformly distributed tensor sampled from the interval [minval,maxval). For example, here's how to create a matrix uniformly sampled from the interval [0, 1) with shape of [2, 2]:

```
In [35]: tf.random.uniform([2,2])
Out[35]:
<tf.Tensor: id=158, shape=(2, 2), dtype=float32, numpy=array([[0.65483284,
0.63064325], [0.008816 , 0.81437767]], dtype=float32)>
```

Create a matrix uniformly sampled from an interval [0, 10) with shape of [2, 2]:

In [36]: tf.random.uniform([2,2],maxval=10)
Out[36]:
<tf.Tensor: id=166, shape=(2, 2), dtype=float32, numpy=
array([[4.541913 , 0.26521802], [2.578913 , 5.126876]], dtype=float32)>

If we need to uniformly sample integers, we must specify the maxval parameter and set the data type as tf.int*.
Notice that these outputs from all random functions may be distinct. However, it does not affect the usage of these functions.

LET US NOW , Create a tensor x of the value
(29.05088806627.61298943 31.19073486, 29.35532951
0.97266006 26.67541885. 38.08450317, 20.74983215,
4.94445419. 34.45999146 29.06485367, 36.01657104
27.88236427. 20.56035233. 30.20379066, 29.51215172.
33.71149445, 28.59134293, 36.05556488, 28.66994858]
Get the indices of elements in x whose values are greater than 30 and then extract those elements.
In[1]:

```
import tensorflow as tf
array=tf.constant([29.05088806,27.61298943 ,31.19073486, 29.35532951,
0.97266006
        ,26.67541 20.74983215,4.94445419 ,34.45999146, 29.06485367,
        36.01657104,27.882364 29.51215172 ,33.71149445, 28.59134293,
        36.05556488, 28.66994858])  # Create array
```

In[2]:

```
print(array)
```

Out[2]:

```
tf.Tensor( [29.050888 27.61299 31.190735 29.35533 0.97266006 26.675419
38.084503
 20.749832 4.944454 34.45999 29.064854 36.01657 27.882364 20.560352
30.20379 29.512152 33.711494 28.591343 36.055565 28.669949 ],
shape=(20,), dtype=float32)
```

In[3]:

```
for idx, elem in enumerate(array):
        if(elem>30.0):
                tf.print(idx, elem)
```

Out[3]:

```
2 31.1907349
6 38.0845032
9 34.4599915
```

11 36.016571
14 30.2037907
16 33.7114944
18 36.0555649

BASIC OPERATIONS ON TENSORS:

Tensors

A Tensor is a multi-dimensional array. Similar to NumPy ndarray objects, tf.Tensor objects have a data type and a shape. Additionally, tf.Tensors can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations (for example, tf.math.add, tf.linalg.matmul, and tf.linalg.inv) that consume and produce tf.Tensors. These operations automatically convert built-in Python types.

In[1]:
```
import tensorflow as tf
a = tf.constant([1, 2, 3, 4])  # Create tensor a
b = tf.constant([10, 21, 35, 46])  # Create tensor b
```

Operations on Tensors

1)Addition

Function:
tf.add(x, y, name=None) Adds two tensors

In[2]:
```
print("Shape of a:",a.shape) #shape of tensor a
print("Shape of b:",b.shape) #shape of tensor b
```

Out[2]:
Shape of a: (4,)
Shape of b: (4,)

Int[3]:
```
 x=tf.add(a,b) #Addition of tensors a and b
tf.print("Addition of two tensors: ",x)
```

Out[3]:
Addition of two tensors: [11 23 38 50]

2) Subtraction

Function:
subtract(x, y, name=None) Subtracts two tensors

In[4]:

```
 x=tf.subtract(a,b)#Subtraction
 y=tf.subtract(b,a)
 tf.print("Subtraction of two tensors: ",x)
tf.print("Subtraction of two tensors: ",y)
```

Out[4]:
Subtraction of two tensors: [-9 -19 -32 -42]
Subtraction of two tensors: [9 19 32 42]

3) Multiplication

Function:
multiply(x, y, name=None) Multiplies two tensors

In[5]:

```
x=tf.multiply(a,b) #Multiplication of tensors
tf.print("Multiplication of two tensors: ",x)
```

Out[5]:
Multiplication of two tensors: [10 42 105 184]

4) Division

Function:
divide(x, y, name=None) Divides the elements of two tensors
(or)
div(x, y, name=None) Divides the elements of two tensors

In[6]:

```
 x=tf.divide(b,a)#Division of two tensors
tf.print("Division of two tensors: ",x)
```

Out[6]:
Division of two tensors: [10 10.5 11.666666666666666 11.5]

When operating on floating-point values, div and divide produce the same result. But for integer division, divide returns a floating-point result, and div returns an integer result. The following code demonstrates the difference between them:

```
a = tf.constant([3, 3, 3])
b = tf.constant([2, 2, 2])

div1 = tf.divide(a, b)          # [ 1.5 1.5 1.5 ]

div2 = a / b                    # [ 1.5 1.5 1.5 ]
```

```
div3 = tf.div(a, b)              # [ 1 1 1 ]

div4 = a // b                    # [ 1 1 1 ]
```

The div function and the / operator both perform element-wise division. In contrast, the divide function performs Python-style division.

5) Logarithm of a tensor(tf.log(x))

Computes natural logarithm of x element-wise.
Tensor x must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

In[7]:

```
a=tf.constant([0,0.5,1.2,2.5]) #dtype=float
x=tf.math.log(a)
tf.print(x)
```

Out[7]:
[-inf -0.693147182 0.182321593 0.91629076]

6) Exponent of a tensor (tf.exp(x))

Computes exponential of x element-wise.
Tensor x Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

In[8]:

```
a=tf.constant([0,0.5,1.2,2.5])
x=tf.math.exp(a)
tf.print(x)
```

Out[8]:
[1 1.64872122 3.320117 12.1824942] Double-click (or enter) to edit

Applications can perform identical operations by using regular Python operators, such as +, -, *, /, and //.

MERGING AND SPLITTING:

Merging means combining multiple tensors into one tensor in a certain dimension. Taking the data of a school's gradebooks as an example, tensor A is used to save the gradebooks of classes 1–4. There are 35 students in each class with a total of eight subjects. The shape of tensor A is [4,35,8]. Similarly, tensor B keeps the gradebooks of the other six classes, with a shape of [6,35,8]. By merging these two gradebooks, you can get the gradebooks of all the classes in the school, recorded as tensor C, and the corresponding shape should be

[10,35,8], where 10 represents ten classes, 35 represents 35 students, and 8 represents eight subjects.

Tensors can be merged using concatenate and stack operations. The concatenate operation does not generate new dimensions. It only merges along existing dimensions. But the stack operation creates new dimensions. Whether to use the concatenate or stack operation to merge tensors depends on whether a new dimension needs to be created for a specific scene. We will discuss both of them in the following session.

Concatenate: In TensorFlow, tensors can be concatenated using the tf.concat(tensors, axis) function, where the first parameter holds a list of tensors that need to be merged and the second parameter specifies the dimensional index on which to merge. Back to the preceding example, we merge the gradebooks in the class dimension. Here, the index number of the class dimension is 0, that is, axis = 0. The code for merging A and B is as follows:

In[0]:
```
import tensorflow as tf
 a = tf.random.normal([4,35,8]) # Create gradebook A
b = tf.random.normal([6,35,8]) # Create gradebook B
tf.concat([a,b],axis=0) #Merging A and B along axis 0
```

Out[0]:
```
[[ 1.0221779 , -1.6211283 , 1.2535826 , ..., 0.8566991 , -0.72349036, -
1.1157284 ], [-1.2414012 , -1.8060911 , 0.75162935, ..., 0.3864265 , 1.7944626
, -0.22267966], [ 0.6155348 , 0.29014412, -1.2273517 , ..., -1.2903832 , -
0.18366472, 1.1815189 ], ..., [ 0.24838325, -0.73552257, 0.1266093 , ...,
1.0176892 , -1.2254806 , 0.30242777], [-0.31192377, -2.3388796 , -0.645616 ,
..., 1.4724126 , 0.40083158, -0.4169775 ], [-0.46578857, 1.282099 ,
0.40268826, ..., 1.1217413 , 1.2570121 , -0.36919504]], ..., [[ 0.04562129, -
1.7338026 , -0.47456488, ..., 1.5377275 , 0.4686792 , 0.49222776], [
0.87741333, -1.8917205 , 1.1468201 , ..., 0.1872781 , -1.9547023 , -1.3492054
], [-1.2246968 , -0.77289253, -1.6376779 , ..., 0.5537161 , 0.4848342 ,
0.5912451 ], ..., [ 1.192786 , -0.13226394, 0.4575644 , ..., -1.5059961 , 2.73987
, -0.7606196 ], [-0.8590228 , -0.1044452 , -0.9715867 , ..., -0.9215654 ,
1.5691227 , -0.5617567 ], [ 0.606453 , 0.49316478, -0.6383934 , ..., -1.3212531
, -0.62109023, 0.1684509 ]], [[-0.02599278, -0.67362314, 0.48570326, ...,
0.22644693, 0.9934903 , -2.5620646 ], [ 0.1159848 , -0.47915196, -0.6811871 ,
..., 0.09434994, 0.3029398 , 0.42174685], [ 0.10794386, 0.46578902, -
2.4382775 , ..., 0.770606 , -0.76425624, -1.3866335 ], ..., [ 0.5851059 , -
```

0.06696272, 1.4475126 , ..., 0.34095022, -0.13524076, -1.0530555], [-1.4368769 , -0.82656825, -0.6933661], ..., [0.5489716 , 0.54459846, -0.8586318 , ..., -0.5403502 , -0.61418784, 0.23939401], [-0.72123253, 0.7499539 , -0.63613904, ..., 0.78031933, -0.98876697, -0.56911653], [1.1187263 , 0.5863637 , -0.04054144, ..., 0.31619748, 0.47991168, 0.5674796]]], dtype=float32)>0.73706305, -1.3118304 , ..., 0.32239807, 1.0951602 , -1.5393324], [-0.07207929, 0.00580749, 0.22784917, ..., 0.81736535, -0.89541644, 0.7397061]], [[0.6770822 , -0.74941754, 0.8807446 , ..., -0.5339431 , 1.5411109 , 0.15334469], [1.0478606 , 1.9049921 , 0.48189902, ..., 0.35219425, 1.9818558 , 0.2982525], [-2.1503694 , 0.05650509, -1.7292447 , ..., 0.37240797,]]

In addition to the class dimension, we can also merge tensors in other dimensions. Consider that tensor A saves the first four subjects' scores of all students in all classes, with shape [10,35,4] and tensor B saves the remaining 4 subjects' scores, with shape [10,35,4]. We can get the total gradebook tensor by merging A and B as in the following:

In [2]:
a = tf.random.normal([10,35,4])
b = tf.random.normal([10,35,4])
tf.concat([a,b],axis=2) # Merge along the last dimension

Out[2]:
<tf.Tensor: id=28, shape=(10, 35, 8), dtype=float32, numpy=
array([[[-5.13509691e-01, -1.79707789e+00, 6.50747120e-01, ...,
2.58447856e-01, 8.47878829e-02, 4.13468748e-01],
[-1.17108583e+00, 1.93961406e+00, 1.27830813e-02, ...,]

Syntactically, the concatenate operation can be performed on any dimension. The only constraint is that the length of the non-merging dimension must be the same. For example, the tensors with shape [4,32,8] and shape [6,35,8] cannot be directly merged in the class dimension, because the length of the number of students' dimension is not the same – one is 32 and the other is 35, for example:
In [3]:
a = tf.random.normal([4,32,8])
b = tf.random.normal([6,35,8])
tf.concat([a,b],axis=0) # Illegal merge.
Second dimension is different.
Out[3]: InvalidArgumentError: ConcatOp : Dimensions of inputs should match: shape[0] = [4,32,8] vs. shape[1] = [6,35,8] [Op:ConcatV2] name: concat
Suppose We get the gradebook tensor of the entire school with the shape of [10,35,8]. Now we need to cut the data into ten tensors in

he class dimension, and each tensor holds the gradebook data of the corresponding class using tensor split operation.Explain the procedure.

Ans:
Splitting:The inverse process of the merge operation is split, which splits a tensor into multiple tensors.

Let's continue the gradebook example. We get the gradebook tensor of the entire school with shape of [10,35,8]. Now we need to cut the data into ten tensors in the class dimension, and each tensor holds the gradebook data of the corresponding class.

tf.split(x, num_or_ size_splits, axis) can be used to complete the tensor split operation. The meaning of the parameters in the function is as follows:
• x: The tensor to be split.
• num_or_size_splits: Cutting scheme. When num_or_ size_splits is a single value, such as 10, it means that the tensor x is cut into ten parts with equal length. When num_or_size_splits is a list, each element of the list represents the length of each part. For example, num_or_size_splits=[2, 4,2,2] means that the tensor is cut into four parts, with the length of each part as 2, 4, 2, and 2.
• axis: Specifies the dimension index of the split. Now we cut the total gradebook tensor into ten pieces as follows:

In [8]:
```
x = tf.random.normal([10,35,8])
# Cut into 10 pieces with equal length
result = tf.split(x, num_or_size_splits=10, axis=0)
len(result) # Return a list with 10 tensors of equal length
```

Out[8]: 10

We can view the shape of a tensor after cutting, and it should be all gradebook data of one class with shape of [1, 35, 8]:

In [9]:
```
result[0] # Check the first class gradebook
```

Out[9]:
```
<tf.Tensor: id=136, shape=(1, 35, 8),
dtype=float32, numpy=
array([[[-1.7786729 , 0.2970506 , 0.02983334, 1.3970423 ,
 1.315918 , -0.79110134, -0.8501629 , -1.5549672 ],
```

[0.5398711 , 0.21478991, -0.08685189, 0.7730989 ,...

It can be seen that the shape of the first class tensor is [1,35,8], which still has the class dimension.

Let's perform unequal length cutting.
For example, split the data into four parts with each length as [4, 2, 2, 2] for each part:

```
In [10]: x = tf.random.normal([10,35,8])
# Split tensor into 4 parts
result = tf.split(x, num_or_size_splits=[4,2,2,2] ,axis=0)
len(result)
```

Out[10]: 4

Check the shape of the first split tensor. According to our splitting scheme, it should contain the gradebooks of four classes. The shape should be [4,35,8]:

```
In [10]: result[0]
Out[10]: <tf.Tensor: id=155, shape=(4, 35, 8),
dtype=float32, numpy=
array([[[-6.95693314e-01, 3.01393479e-01, 1.33964568e-01, ...,
```

In particular, if we want to divide one certain dimension by a length of 1, we can use the **tf.unstack(x, axis)** function. This method is a special case of tf.split. The splitting length is fixed as 1. We only need to specify the index number of the splitting dimension.
For example, unstack the total gradebook tensor in the class dimension:

```
In [11]: x = tf.random.normal([10,35,8])
result = tf.unstack(x,axis=0)
len(result) # Return a list with 10 tensors
```

Out[11]: 10

View the shape of the split tensor:

```
In [12]: result[0] # The first class tensor
```

```
Out[12]: <tf.Tensor: id=166, shape=(35, 8),
dtype=float32, numpy=
```

array([[-0.2034383 , 1.1851563 , 0.25327438,
-0.10160723, 2.094969 ,
-0.8571669 , -0.48985648, 0.55798006],...

It can be seen that after splitting through tf.unstack, the split tensor shape becomes [35, 8], that is, the class dimension disappears, which is different from tf.split

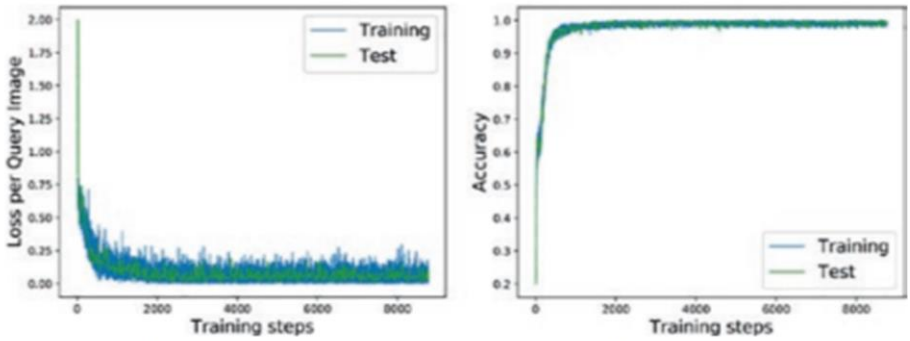
APPLICATIONS OF TENSORS:

Tensors have a vast application in physics and mathematical geometry. The mathematical explanation of electromagnetism is also defined by tensors. The vector analysis acts as a primer in tensor analysis and relativity. Elasticity, quantum theory, machine learning, mechanics, relativity are all affected by tensors.

(i) SCALAR :

Scalar is a simple number with 0 dimension and a shape of []. Typical uses of scalars are the representation of error values and various metrics, such as accuracy ,precision and recall.

Consider the training curve of a model. The x-axis is the number of training steps, and the y-axis is Loss per Query. Image error change (Figure 4-1 (a)) and accuracy change (Figure 4-1 (b)), where the loss value and accuracy are scalars generated by tensor calculation.



(a)Training/validation error curves (b)Training/validation accuracy curves

Figure 4-1. *Loss and accuracy curves*

Take the mean squared error function as an example. After tf.keras. losses.mse (or tf.keras.losses.MSE, the same function) returns the error value on each sample and finally takes the average value of the error as the error of the current batch, it automatically becomes a scalar:

In [41]:

out = tf.random.uniform([4,10])

Create a model output example

y = tf.constant([2,3,2,0])
# Create a real observation
y = tf.one_hot(y, depth=10)
# one-hot encoding loss = tf.keras.losses.mse(y, out)
# Calculate MSE for each sample loss = tf.reduce_mean(loss)
# Calculate the mean of MSE print(loss)
Out[41]: tf.Tensor(0.19950335, shape=(), dtype=float32)

(ii) VECTOR:
Vectors are very common in neural networks. For example, in fully connected networks and convolutional neural networks, bias tensors b are represented by vectors. As shown in Figure 4-2, a bias value is added to the output nodes of each fully connected layer, and the bias of all output nodes is represented as a vector form b = [b1,b2]^T:

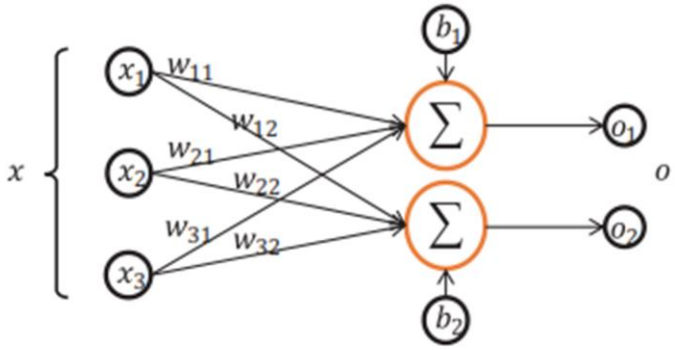


Figure 4-2. *Application of bias vectors*

Considering a network layer of two output nodes, we create a bias vector of length 2 and add back on each output node:
In [42]:
# Suppose z is the output of an activation function
z = tf.random.normal([4,2])
b = tf.zeros([2])
# Create a bias vector
z = z + b
Out[42]:
<tf.Tensor: id=245, shape=(4, 2), dtype=float32, numpy=
array([[0.6941646 , 0.4764454],
[-0.34862405, -0.26460952],
[1.5081744 , -0.6493869],
[-0.26224667, -0.78742725]], dtype=float32)>

For a network layer created through the high-level interface class Dense(), the tensors W and b are automatically created and managed by the class internally. The bias variable b can be accessed through the bias member of the fully connected layer. For example, if a linear network layer with four input nodes and three output nodes is created, then its bias vector b should have length of 3 as follows:

In [43]:

```
fc = layers.Dense(3)
 # Create a dense layer with output length of 3
# Create W and b through build function with input nodes of 4
fc.build(input_shape=(2,4))
 fc.bias # Print bias vector
```

Out[43]:

```
<tf.Variable 'bias:0' shape=(3,) dtype=float32,
numpy=array([0., 0., 0.], dtype=float32)>
```

It can be seen that the bias member of the class is a vector of length 3 and is initialised to all 0s. This is also the default initialization scheme of the bias b. Besides, the type of the bias vector is Variable, because gradient information is needed for both W and b.

(iii) MATRIX:

A matrix is also a very common type of tensor. For example, the shape of a batch input tensor X of a fully connected layer is [b,din], where b represents the number of input samples, that is, batch size, and din represents the length of the input feature. For example, the feature length 4 and the input containing a total of two samples can be expressed as a matrix:

```
x = tf.random.normal([2,4]) # A tensor with 2 samples and 4 features
```

Let the number of output nodes of the fully connected layer be three and then the shape of its weight tensor W [4,3]. We can directly implement a network layer using the tensors X, W and vector b. The code is as follows:

In [44]:

```
w = tf.ones([4,3])
b = tf.zeros([3])
o = x@w+b # @ means matrix multiplication
```

Out[44]: <tf.Tensor: id=291, shape=(2, 3), dtype=float32, numpy=
array([[2.3506963, 2.3506963, 2.3506963], [-1.1724043, -1.1724043, -1.1724043]], dtype=float32)

In the preceding code, both X and W are matrices. The preceding code implements a linear transformation network layer, and the activation function is

empty. In general, the network layer σ(X @ W + b) is called a fully connected layer, which can be directly implemented by the Dense() class in TensorFlow.

In particular, when the activation function σ is empty, the fully connected layer is also called a linear layer. We can create a network layer with four input nodes and three output nodes through the Dense() class and view its weight matrix W through the kernel member of the fully connected layer:
In [45]:

```
 fc = layers.Dense(3)
# Create fully-connected layer with 3 output nodes
fc.build(input_shape=(2,4))
# Define the input nodes to be 4
 fc.kernel # Check kernel matrix W
```

Out[45]:

```
<tf.Variable 'kernel:0' shape=(4, 3) dtype=float32, numpy=array([[ 0.06468129,
-0.5146048 , -0.12036425], [ 0.71618867, -0.01442951, -0.5891943 ], [-
0.03011459, 0.578704 , 0.7245046 ], [ 0.73894167, -0.21171576, 0.4820758 ]],
dtype=float32)>
```

(iv) Three-Dimensional Tensor:
A typical application of a three-dimensional tensor is to represent a sequence signal. Its format is:

X=[b, sequence length, feature length]

where the number of sequence signals is b, sequence length represents the number of sampling points or steps in the time dimension, and feature length represents the feature length of each point.

In order to facilitate the processing of strings by neural networks, words are generally encoded into vectors of fixed length through the embedding layer. For example, "a" is encoded as a vector of length 3. Then two sentences with equal length (each sentence has five words) can be expressed as a three-dimensional tensor with shape of [2,5,3], where 2 represents the number of sentences, 5 represents the number of words, and 3 represents the length of the encoded word vector. We demonstrate how to represent sentences through the IMDB dataset as follows:
In [46]:

```
# Load IMDB dataset
 from tensorflow import keras
(x_train,y_train),(x_test,y_test)=keras.datasets.imdb.load_
data(num_words=10000)
 # Convert each sentence to length of 80 words
 x_train = keras.preprocessing.sequence.pad_sequences(x_ train,maxlen=80)
 x_train.shape
```

Out [46]: (25000, 80)

We can see that the shape of the x_train is [25000,80], where 25000 represents the number of sentences, 80 represents a total of 80 words in each sentence, and each word is represented by a numeric encoding method. Next, we use the layers.Embedding function to convert each numeric encoded word into a vector of length 100:

In [47]: # Create Embedding layer with 100 output length
embedding=layers.Embedding(10000, 100)
# Convert numeric encoded words to word vectors
out = embedding(x_train) out.shape
Out[47]: TensorShape([25000, 80, 100])

Through the embedding layer, the shape of the sentence tensor becomes [25000,80,100], where 100 represents that each word is encoded as a vector of length 100.

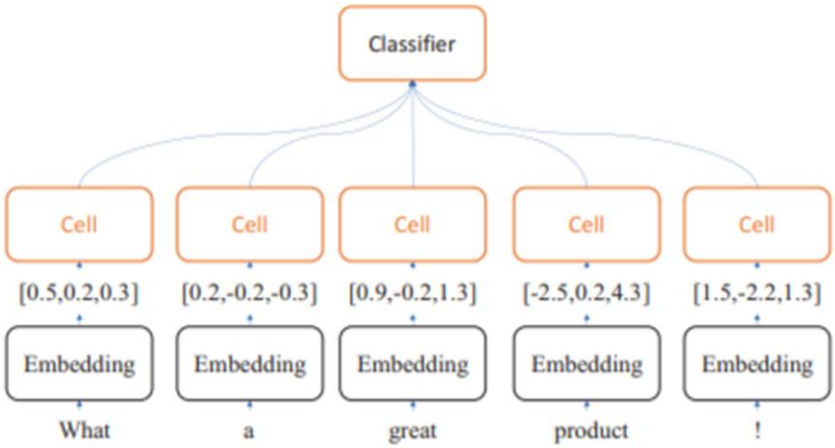


Figure 4-3. *Sentiment classification network*

For a sequence signal with one feature, such as the price of a product within 60 days, only one scalar is required to represent the product price, so the price change of two products can be expressed using a tensor of shape [2,60]. In order to facilitate the uniform format, the price change can also be expressed as a tensor of shape [2,60,1], where 1 represents the feature length of 1.

(iv)Four-Dimensional Tensor:

Most times we only use tensors with dimension less than five. For larger dimension tensors, such as five-dimensional tensor representation in meta learning, a similar principle can be applied. Four-dimensional tensors are widely used in convolutional neural networks. They are used to save feature maps. The format is generally defined as :

[b,h,w,c]

where b indicates the number of input samples; h and w represent the height and width of the feature map, respectively; and c is the number of channels. Some deep learning frameworks also use the format of [b, c,h,w], such as PyTorch. Image data is a type of feature map. A color image with three channels of RGB contains h rows and w columns of pixels. Each point requires three values to represent the color intensity of the RGB channel, so a picture can be expressed using a tensor of shape [h,w, 3]. As shown in Figure 4-4, the top picture represents the original image, which contains the intensity information of the three lower channels

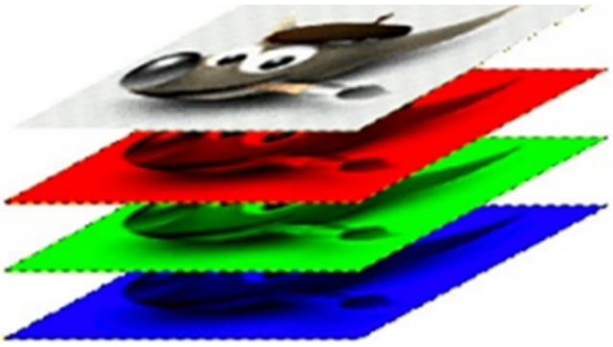


Figure 4-4. *Feature maps of RGB images*

In neural networks, multiple inputs are generally calculated in parallel to improve the computation efficiency, so the tensor of b pictures can be expressed as [b,h,w, 3]:

```
In [48]: # Create 4 32x32 color images
 x = tf.random.normal([4,32,32,3])
# Create convolutional layer
 layer = layers.Conv2D(16,kernel_size=3)
out = layer(x) out.shape
Out[48]: TensorShape([4, 30, 30, 16])
```

The convolution kernel tensor is also a four-dimensional tensor, which can be accessed through the kernel member variable: In [49]: layer.kernel.shape
Out[49]: TensorShape([3, 3, 3, 16])

BROADCASTING IN TENSORFLOW:

Broadcasting is a lightweight tensor copying method, which logically expands the shape of the tensor data, but only performs the actual storage copy operation when needed.The core idea of the broadcasting mechanism is universality. That is, the same data can be generally suitable for other locations.

For all dimensions of length 1, broadcasting has the same effect as tf.tile. The difference is that tf.tile creates a new tensor by performing the copy IO

operation. Broadcasting does not immediately copy the data; instead, it will logically change the shape of the tensor, so that the view becomes the copied shape. Broadcasting will use the optimization methods of the deep learning framework to avoid the actual copying of data and complete the logical operations.
The broadcasting mechanism saves a lot of computational resources. It is recommended to use broadcasting as much as possible in the calculation process to improve efficiency.

Continuing to consider the preceding example Y = X @ W + b, the shape of X @ W is [2, 3], and the shape of b is [3]. We can manually complete the copy data operation by combining tf.expand_dims and tf.tile, that is, transform b to shape [2, 3] and then add it to X @ W. But in fact, it is also correct to add X @ W directly to b with shape [3], for example:

```
x = tf.random.normal([2,4])
w = tf.random.normal([4,3])
b = tf.random.normal([3])
y = x@w+b # Add tensors with different shapes directly
```

The preceding addition does not throw a logical error. This is because it automatically calls the broadcasting function tf.broadcast_to(x, new_ shape), expanding the shape of b to [2,3]. The preceding operation is equivalent to:

```
y = x@w + tf.broadcast_to(b,[2,3])
```

In other words, when the operator + encounters two tensors with inconsistent shapes, it will automatically consider expanding the two tensors to a consistent shape and then call tf.add to complete the tensor addition operation. By automatically calling tf.broadcast_to(b, [2,3]), it not only achieves the purpose of increasing dimension but also avoids the expensive computational cost of actually copying the data.

Before verifying universality, we need to align the tensor shape to the right first and then perform universality check: for a dimension of length 1, by default this data is generally suitable for other positions in the current dimension; for dimensions that do not exist, after adding a new dimension, the default current data is also universally applicable to the new dimension, so that it can be expanded into a tensor shape of any number of dimensions.