POLITECNICO DI TORINO

Corso di Laurea Magistrale

in Mechatronic Engineering

Tesi di Laurea Magistrale

Review on Optimization Methods for Artificial Intelligence Models in Edge Computing

Relatore:
Prof. Marcello Chiaberge

Candidato:
Ondina Colalongo

Anno Accademico 2020/2021

1. Introduction

1.1 Context

Artificial Intelligence (AI) is a computer technology that allows machines to "learn" from experience, adapt to new information received and perform tasks similar to those performed by man. Its use has a fundamental role in our life and it is present in most of the various production sectors. Just think of the assisted and autonomous driving systems of the latest generation cars, the voice assistants of the various smart-devices and smart manufacturing.

Most of the examples of AI, like chess programs and self-driving cars, are mainly based on deep learning and natural language processing. Using these technologies, computers can learn to perform specific tasks processing large amounts of data and recognizing patterns.

Deep learning is one of the most interesting methodologies under development in the field of Artificial Intelligence. Thanks to its characteristics, deep learning architectures have been applied in computer vision, in the automatic recognition of spoken language, in the processing of natural language, in audio recognition and bioinformatics.

Owing to its multilayer structure, deep learning is also appropriate for the edge computing environment.

End devices, such as smartphones and Internet-of-Things sensors, are generating data that need to be analysed in real time using deep learning or used to train deep learning models.

Edge computing, where a dense mesh of compute nodes is placed close to end devices, is a viable way to meet the high computation and low-latency requirements of deep learning on edge devices and also provides additional benefits in terms of privacy, bandwidth efficiency, and scalability.

1.1.1 Artificial Intelligence

Artificial intelligence (AI) is a wide-ranging branch of computer science. It is a technique that enables the machine to act like humans by replicating their behaviour and nature. Through AI, it is possible for machines to "learn" from experiences.

The goal of AI is to create systems that can work intelligently and independently.

The most understandable way to introduce the artificial intelligence topic is in the context of human behaviours (actions). For this reason, the fields of artificial intelligence will be explained through simple examples related to the daily life of humans and these will be displayed in Fig.1.

Humans can see with their eyes and process what they see, this is the field of *computer vision.*

Humans recognize the scene around them through their eyes which create images of that world, this is the field of *image processing*.

Computer vision and *image processing* fall under the symbolic way for computers to process information.

Humans can understand their environment and move around fluidly, this is the field of *robotics*.

1.1.2 Machine Learning

"Machine learning is a subset of AI technique which uses statistical methods to enable machines to improve with experience".

Machine learning (ML) came into existence in the late 80s and the early 90s with the purpose to efficiently train large complex models in the field of computer science and artificial intelligence with the idea of designing the brain functioning model. ML enables the computer to act and make data-driven decisions to carry out a certain task.

Let us see, with some examples of everyday life which are the related fields of the ML involved:

The *speech recognition* is a process by which human oral language is recognized and subsequently processed through a computer or, more specifically, through a special voice recognition system. Much of speech recognition is statistically based, hence it is called statistical learning; it is possible to notice the similarity of how speech recognition simulates the behaviour of the human being who communicates by speaking and listening through a certain language.

Natural language processing or NLP is a system of automatic processing of information written or spoken in a natural language by an electronic computer; also in this case, it is possible to notice how the NLP assumes the human behaviour of writing and reading a text in a certain language.

1.1.3 Deep Learning

In the machine learning field, an artificial neural network, ANN or NN, is a computational model composed of artificial "neurons". This field is a particular kind of machine learning that is inspired by the functionality of our brain cells, called neurons, which led to the concept of artificial neural network.

The human brain is a network of neurons and we use these to learn things. Parallelly, the field of neural networks, inspired by the biological neural networks, tries to replicate the structure and the function of the human brain. If these networks are more complex and deeper, we use the field of deep learning.

There are different types of deep learning which are essentially different techniques to replicate what the human brain does.

In particular, a convolution neural network (CNN) is used to recognize objects in a scene. Thus for example, it is possible getting the network to scan images from left to right, top to bottom.

Instead, the recurrent neural network (RNN), a class of artificial neural network that includes linked neurons among them in a loop, is used to remember a limited past.

Due to its high efficiency in studying complex data, deep learning will play a very important role in future Internet of things (IoT) services. Nowadays, DL has been introduced into many businesses related to IoT and to the mobile applications like as Edge computing.

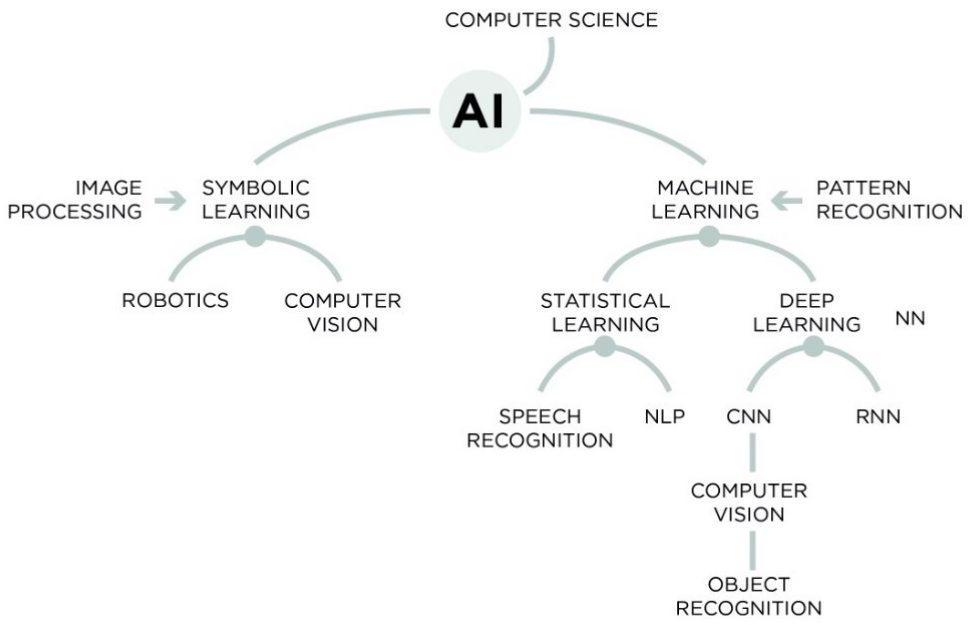

Fig.1 Representation of AI fields

1.1.4 Edge Computing vs. Cloud Computing

Edge computing is a distributed computing model in which data processing occurs as close as possible to where the data is required. As the name suggests, edge computing places networked computing resources as close as possible to where data is created.

It is associated with the Internet of Things, mesh networks, and with the application of small computing devices such as those shown in the Fig. 2, here below.

Fig.2 INTEL NCS 1, CORAL EDGE TPU, NVIDIA JETSON NANO

Cloud computing has been one of the biggest digital trends and involves the delivery of computing resources over the Internet. In the early days, most of the devices that accessed cloud services were PCs and other end-user hardware. But increasingly, devices accessing cloud services are also IoT appliances that transmit data for analysis online. Connecting cameras and other sensors to the Internet facilitates the creation of smart factories and smart homes.

However, transmitting an increasing volume of data for remote centralized processing is becoming problematic. Not least, transmitting video from online cameras to cloud-based vision recognition services can overload available network capacity and result in a slow speed of response.

This is the reason for the rise of edge computing.

Edge computing allows devices, that would have relied on the cloud, to process some of their own data. So, for example, a networked camera may perform local vision recognition. This can improve latency -- or the time taken to generate a response from a data input – as well as reducing the cost and requirement for mass data transmission.

Today, Amazon, Google and Microsoft all offer cloud vision recognition services that can receive a still image or video feed and return a cognitive response. These cloud AI services rely on neural networks that have been pre-trained on data center servers. When an input is received, then they perform inference -- again on a cloud data center server -- to determine what the camera is looking at.

Alternatively, in an edge computing scenario, a neural network is usually still trained on a data center server, as training requires a lot of computational power. But once training is complete, a copy of the neural network is deployed to a networked camera connected to edge computing hardware. Latency is improved and the demands on the network are decreased.

An example of edge computing used in vision recognition has just been reported above. But the same concept is equally applicable for the edge processing of audio, sensor data, and the local control of robots or other cyber physical systems.

In summary, the main difference between edge computing and cloud is that cloud computing prefers using remote data centres for storage while Edge computing makes partial use of local drives.

As a conclusion, it is possible to say that edge computing focuses on providing stable and fast performance and it can store large amounts of data but the performance is smoother.

The comparison between Edge Computing and Cloud Computing is shown in the table displayed in the Fig. 3, hereunder.

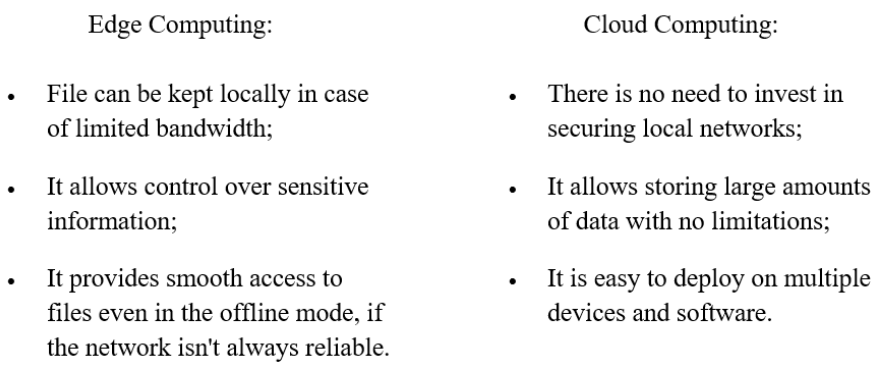

Fig.3 Comparison between Edge and Cloud Computing

Let's summarise the key benefits of Edge computing.

First of all, it's a reduced latency: it provides a faster user experience to end users.

Safety is also a major advantage: edge computing allows companies to keep some of their control over data by storing the key pieces of information locally.

Also the scalability is present: edge computing allows storing increasing amounts of data both in remote centres and on the edges of networks.

Finally, there is the versatility: edge computing finds a balance between traditional centralised cloud data storage in local storage.

Despite all these advantages, it is necessary to mention some of the problems related to edge computing. One of the them concerns power supply: if the device is cut off from the stable electricity source it won't be able to process data in the local network. This challenge can be addressed by implanting alternative energy production means and accumulators. The second problem is space: if there are enough local servers, the edge computing will not be able to accommodate a lot of data. The final challenge is security: technically Edge Computing can be a lot more secure than cloud computing because you don't have to intrust sensitive information to the third party provider. In reality, this is only possible if you invest in securing your local network.

1.2 Aim of the thesis project

The aim of this thesis project is to present some optimization methods for Artificial Intelligence models in Edge Computing. In particular, some basic deep learning models, that are most commonly used, will be presented. After that, DL models, consisting of various types of Deep Neural Networks (DNNs), will be described. Finally, in the last chapter, platforms and devices used in the implementation of Edge computing will be introduced.

2. Neural Networks in Deep Learning

Neural networks (NN) form the base of deep learning where the algorithms are inspired by the structure of the human brain. NNs are made up of layers of neurons which are the core processing units of the network.

Basically, a neural network is a sub component of an artificial intelligence system that tries to mimic how the human brain works.

Like as biological neurons, *artificial neurons* can transmit signal or information to another neuron and the *receiving neuron* can transmit signal to the next one.

The most basic neural network architecture is composed of an *input layer, hidden layers* and an *output layer*. The so-called deep neural network refers to a sufficient number of hidden layers between the input layer and the output layer.

In the Fig. 4, it is shown the functional principle that consists in receiving an input signal (x_0), processing it and sending it in output to other neurons. The connection used by the signal to reach a neuron changes its intensity, as each connection is associated with a certain weight (w_0) which is multiplied with the signal itself ($w_0 x_0$). Once all the information is fully received by the cell body, the neuron's processing will then be the sum of the inputs and a possible bias (b) that is a parameter associated with the neuron itself. The basic idea is to assign and modify the weights of connections and neurons in such a way as to obtain a specific output: in doing so, the ability to learn of neural networks is modelled.

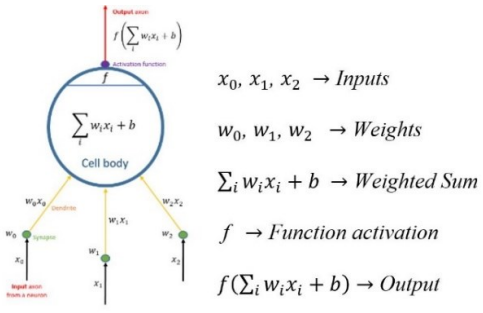

Fig.4 "Artificial" neuron

2.1 Perceptron

The Perceptron is one of the simplest NN architectures, invented in 1957 by Frank Rosenblatt and it is based on a threshold logic unit (TLU) that is shown in Fig. 5. The inputs and output are numbers and each input connection is associated with a weight.

The TLU computes a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + \ldots + w_nx_n = x^Tw$), then applies a step function to that sum and outputs the result:

$$h_w(x) = step(z),$$

where $z = x^Tw$

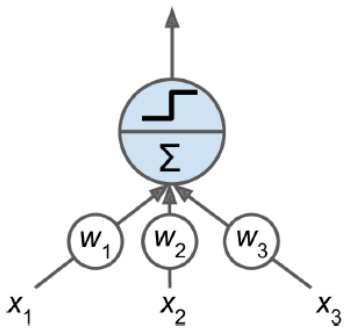

Fig.5 Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

The most common step function used in Perceptron is the Heaviside step function:

$$\text{Heaviside (z)} = \begin{cases} 0 \ \ if\ z < 0 \\ 1 \ \ if\ z \geq 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs. A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs.

A Perceptron with two inputs and three outputs is represented in Fig. 6. This Perceptron can classify instances simultaneously into three different binary classes.

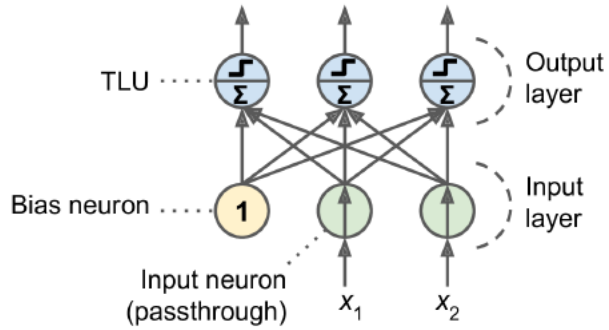

Fig.6 Architecture of Perceptron with two input neuron, one bias neuron and three output neurons

The outputs of a layer of NN for several instances at once is computed by this equation:

$$h_{w,b}(X) = \phi(XW + b)$$

Where:

- X represents the matrix of input features. It has one row per instance and one column per feature;
- W is the weight matrix which contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer;
- B is the bias vector which contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron;
- ϕ is a function called the activation function.

2.2 Multilayer Perceptron

As suggest the name, the multiplayer perceptron (MPL) is a neural network made up of different layers. These multiple layers and non-linear activation distinguish MLP from a linear perceptron. As it is shown in the Fig. 7, it is composed of one (passthrough) input layer, one or more layers of TLUs, called hidden layers, and one final layer of TLUs called the output layer. The layers close to the input layer are usually called the lower

layers, and the ones close to the outputs are usually called the upper layers. Every layer except the output layer includes a bias neuron and is fully connected to the next layer. MPLs can be used for feature extraction and function approximation with high complexity, modest performance and slow convergence.

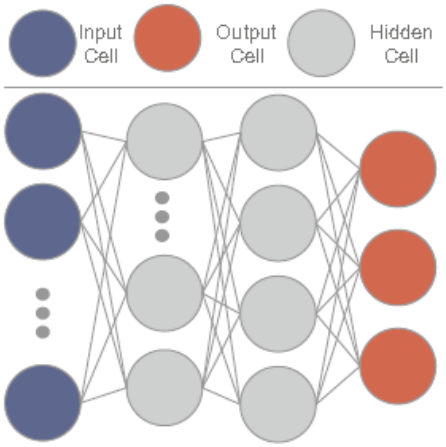

Fig.7 Architecture of a MLP with n inputs, two hidden layer and three output neurons (the bias neurons are implicit)

MLPs can be used for *regression tasks*. In particular, if you want to predict a single value, then you just need a single output neuron: its output is the predicted value. While for multivariate regression, you need one output neuron per output dimension.

MLPs can also be used for *classification tasks*. For a *binary* classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.

MLPs can also easily handle *multilabel* binary classification tasks. More generally, you dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1.

In addition, MLPs is used for the *multiclass* classification. It is the problem of classifying instances into one of three or more classes. In this case, the softmax activation function is used for the whole output layer because it ensures that all the estimated probabilities are between 0 and 1 and that they add up to 1.

2.2.1 Backpropagation

The training process of the MPL occurs by continuous adjustment of the weight of the connections after each processing. In particular, the adjustment is based on the error in output. To train MLPs, backpropagation algorithm is used.

The backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter. It can find out how each connection weight and each bias term should be tweaked in order to reduce the error.

The basic idea of backpropagation is to guess what the hidden units should look like, based on what the input looks like and what the output should look like.

The backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).

The backpropagation algorithm is described in detail below:

- Firstly, it handles one mini-batch at a time which is passed to the network's input layer and sent to the first hidden layer. The algorithm then computes the output of all the neurons in this layer and the result is passed on to the next layer. Its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer.

- Next, the algorithm measures the network's output error.

- Then it computes how much each output connection contributed to the error.

- The algorithm then measures how much of these error contributions came from each connection in the layer below working backward until the algorithm reaches the input layer.

- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

In order for this algorithm to work properly, the step function is replaced by the *logistic* (*sigmoid*) *function* with output value ranges from 0 to 1:

$$\sigma(z) = 1 / (1 + e^{-z})$$

The reason of this replacement is due to the properties of these functions. The step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step.

Two other popular choices to replace step function are *hyperbolic tangent function* and *Rectified Linear Unit function*. The first one is *S*-shaped, continuous and differentiable and its output value ranges from –1 to 1. The last one is continuous but not differentiable at $z = 0$ and its derivative is 0 for $z < 0$.

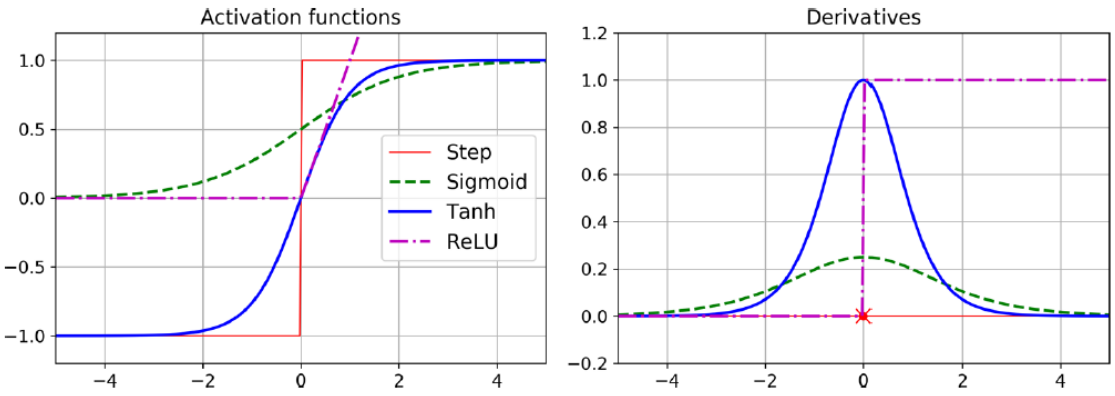

Fig.8 Different Activation functions and their Derivatives

2.3 Convolutional Neural Network (CNN)

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In recent years, thanks to the improvements that have taken place, CNNs have begun to achieve superhuman performances on some complex visual tasks like. In addition, CNNs are successfully used in many other tasks such as voice recognition and natural language processing.

Anyhow, CNNs are explicitly conceived to process images among neural networks. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper, i.e. with more feature maps, due to the convolutional layers. At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers, and the final layer outputs the prediction. The characteristic of CNNs, that distinguishes them from MPLs, is the alternation of convolutional layers and pooling layers, as in Fig. 9:

- Convolutional layer: it applies the mathematical operation of the convolution to the data it receives and it deals with extracting the features, i.e. any relevant information, from each image in input. Convolutional layers are similar to what we call "filters" on computer image processing. The key feature of the convolution layer is to learn local patterns in their input feature space.

- Pooling layer: it aggressively downsamples feature maps. It performs a downsampling of the input image, reducing the computational load, the memory usage and the number of parameters, while gaining information on the features. On the one hand, the feature map is made smaller, simplifying the computational complexity of the network; on the other hand, feature extraction and compression can be performed to extract the main features. So it can extract features while reducing the model complexity, which mitigates the risk of overfitting.

Thanks to its characteristics, once CNN has learned to recognize a pattern in a location, it is able to recognize it in any other location. Differently, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

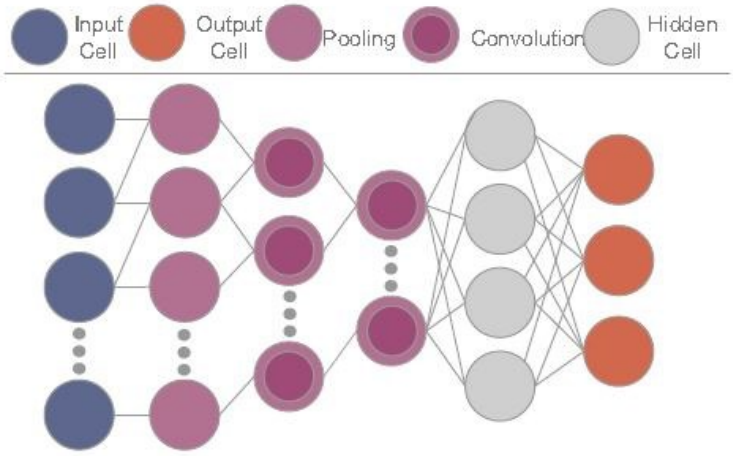

Fig.9 Convolutional Neural Network

Typical CNN architectures are made up of a stack composed by a succession of convolutional layers and pooling layers. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper, i.e. with more feature maps. At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers.

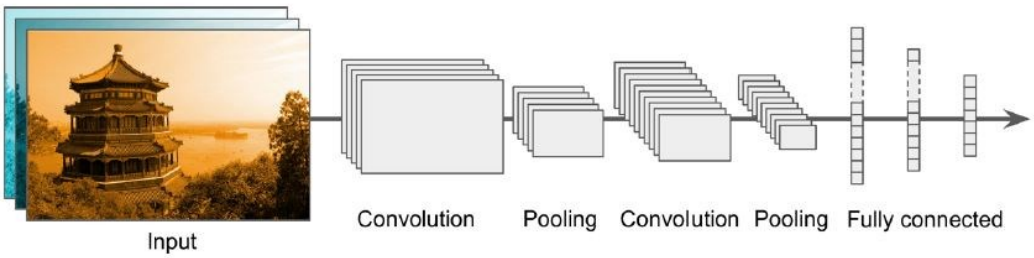

Fig. 10 Typical CNN architecture

Over the years, variants of CNN architectures have been developed. Starting from 1998, the LeNet-5 architecture has been creating; later in the 2012, 2014 and 2015 respectively AlexNet, GoogLeNet and ResNet have been developed.

2.4 Recurrent Neural Network (RNN)

RNNs are designed for handling sequential data. As depicted in Fig.11, each neuron in RNNs not only receives information from the upper layer, but also receives information from the previous channel of its own. Thus, an RNN can simultaneously take a sequence of inputs and produce a sequence of outputs. In general, RNNs are used for predicting future information or restoring missing parts of sequential data.

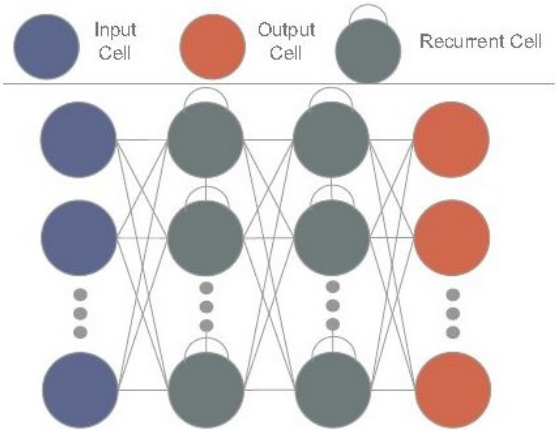

Fig. 11 Recurrent neural network

As is shown in the Fig. 12, the simplest possible RNN is composed of one neuron receiving inputs, producing an output and sending that output back to itself. At each time step t, recurrent neuron receives the inputs $x_{(t)}$ as well as its own output from the previous time step, $y_{(t-1)}$.

Moreover, you can easily create a layer of recurrent neurons. In fact, as shown in Fig. 13, at each time step t, every neuron receives both the input vector $x_{(t)}$ and the output vector from the previous time step $y_{(t-1)}$.

Fig.12 Recurrent neuron

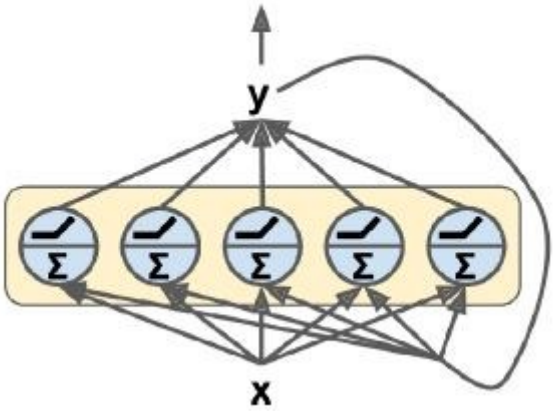

Fig.13 Layer of Recurrent neuron

The output vector of the whole recurrent layer can then be computed as:

$$y_{(t)} = \phi(W_x^T x_{(t)} + W_y^T y_{(t-1)} + b)$$

Where:

- W_x, W_y are the weight matrices in which all the weight vectors are placed;
- $x_{(t)}$ is the input;
- $y_{(t-1)}$ is the output of the previous time step t;
- b is the bias vector;
- ϕ is the activation function

The output vector of a layer of recurrent layer for all instances in a mini-batch can be computed as:

$$Y_{(t)} = \phi(X_{(t)} W_x + Y_{(t-1)} W_y + b)$$

$$= \phi([X_{(t)} \quad Y_{(t-1)}]W + b)\ ,$$

$$W = \begin{bmatrix} W_x \\ W_y \end{bmatrix}$$

Where:

- $Y_{(t)}$ is an $m \times n_{neurons}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch;
- $X_{(t)}$ is an $m \times n_{inputs}$ matrix containing the inputs for all instances;
- W_x is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights for the inputs of the current time step;
- W_y is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step;
- b is a vector of size $n_{neurons}$ containing each neuron's bias term.

An improving of RNN is Long Short-term Memory, LSTM, as in Fig. 14, by adding a gate structure and a well-defined memory cell. It controls (prohibiting or allowing) the flow of Information. A common LSTM is composed of three gates: input gate, forget gate and output gate. The input gate is used to select the content update of the memory cell, the forget gate determines which information needs to be discarded, and the output gate controls which part of the memory cell will be output at that moment. According to these characteristics, LSTM is widely used in NLP fields such as speech recognition and machine translation.

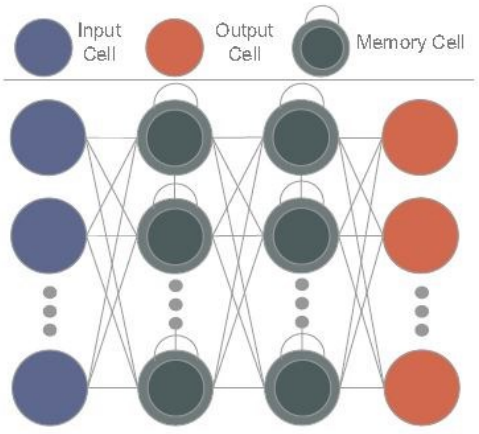

Fig. 14 Long short-term memory

2.5 Auto-Encoder (AE)

Auto-encoders are artificial neural networks capable of learning dense representations of the input, called coding. Owing to the lower dimensionality of these coding, auto-encoders are utilized for dimensionality reduction especially for visualization purposes. They are also used for unsupervised pre-training of deep neural networks because they are capable of randomly generating new data that looks very similar to the training data.

Auto-Encoder is actually a stack of two NNs that replicate input to its output in an unsupervised learning style, as it is shown in Fig. 15. The first NN learns the representative characteristics of the input (encoding). The second NN takes these features as input and restores the approximation of the original input at the match input output cell, used to converge on the identity function from input to output, as the final output (decoding). Thus, an auto-encoder typically has the same architecture as a Multi-Layer Perceptron except that the number of neurons in the output layer must be equal to the number of inputs.

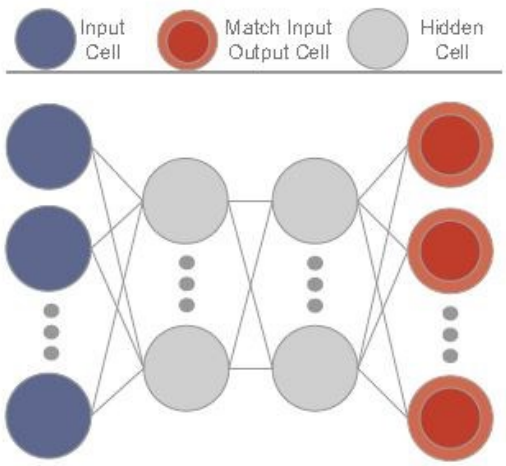

Fig. 15 Auto-Encoder

2.6 Generative Adversarial Network (GAN)

GANs were proposed in 2014 by Ian Goodfellow et al. and originates from game theory. As illustrated in Fig. 16, GAN is composed of generator and discriminator. The generator takes a random distribution as input and outputs some data. The goal of the generator is to learn about the true data distribution as much as possible by deliberately introducing feedback at the *backfed* input cell. The discriminator takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real. Thus the discriminator's goal is to correctly determine whether the input data is coming from the true data or from the generator. According to the features learned from the real information, a well-trained generator can thus fabricate indistinguishable information.

Summarizing, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator. Moreover, GANs cannot be trained like other neural networks since generator and discriminator perform different goals. In fact, two phases of training are present: the discriminator is trained in the first phase while the generator is trained in the second one.

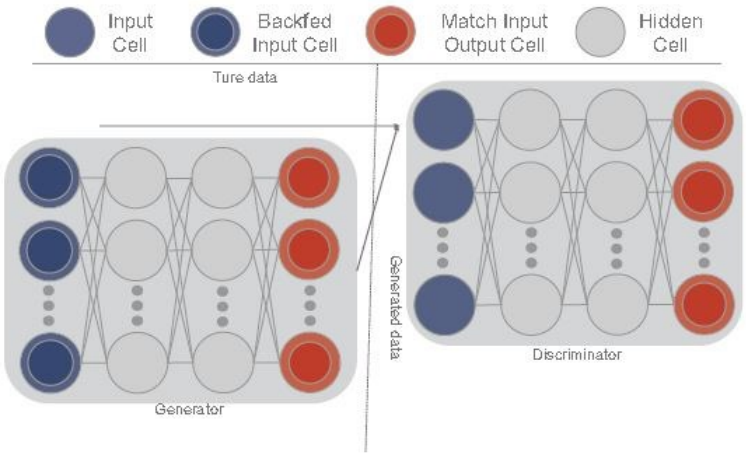

Fig. 16 Generative Adversarial Network

2.7 Transfer Learning (TL)

It is possible to improve trained networks to solve problems reducing development costs with Transfer Learning (TL). In fact, TL can transfer knowledge, as shown in Fig. 17, from the source domain to the target domain so as to achieve better learning performance in the target domain. By using TL, existing knowledge learned by a large number of computation resources can be transferred to a new scenario, and thus accelerating the training process and reducing model development costs. However, there is a problem with transfer learning. It is often applied to some small and stable datasets, making it difficult to get a wider application.

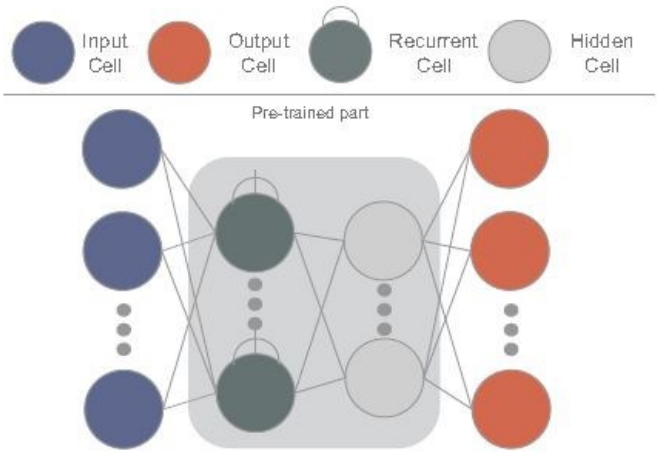

Fig. 17 Transfer Learning

Recently, Knowledge Distillation (KD), a process of transfer learning, has appeared. As indicated in Fig.18, KD can extract implicit knowledge from a well-trained model (*teacher*), the inference of which possess excellent performances but requires high overhead. Then, by designing the structure and objective function of the target DL model, the knowledge is “transferred” to a smaller DL model (*student*), so that the significantly reduced target DL model achieves high performance.

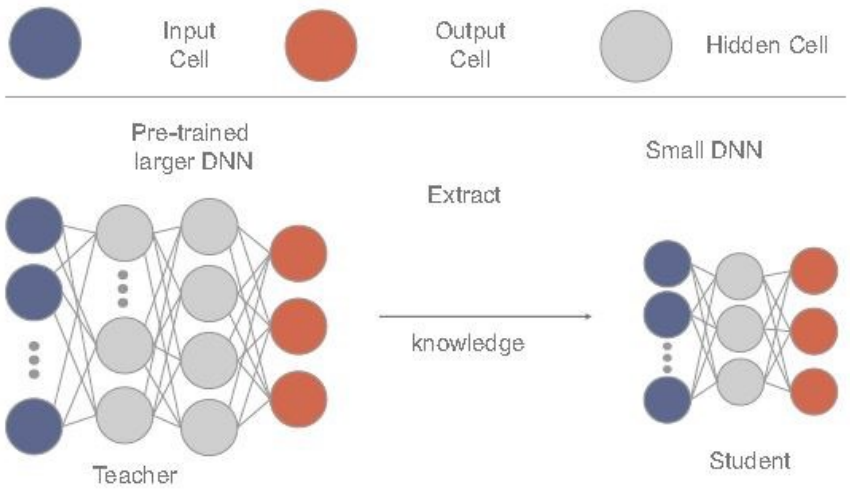

Fig. 18 Knowledge Distillation

3. Optimization Methods

Generally, when it comes to optimization in machine learning, you have to deal with reduced memory, computing and power consumption. In addition, a wide variety of hardware exists and therefore you need to make sure that models run efficiently on all these different types of hardware.

Some neural networks require billions and billions of multiplication and addition operations to perform a single calculation and the time to complete it could be quite significant. Because of this, costs increase significantly; in particular, these costs are due to operating costs required for the use of cloud services and for the installation of faster processors into edge devices. Further, some neural networks will have connection weights consisting of millions or tens of millions of neurons and storing all of these data requires a massive amount of memory.

There are several main ways for model optimization that can help with application development.

- Some forms of optimization can be used to reduce the size of a model. In fact, smaller models acquire some benefits: firstly, (1) smaller models occupy less storage space on users' devices; (2) they also require less time and bandwidth to download on user's devices and (3) they use less RAM when they are run (which frees up memory for other parts of the application) and finally (4) smaller models can achieve better performance and stability. Q*uantization* is used to reduce the size of models but at the expense of some accuracy. *Pruning* and *clustering* can reduce the size of models for download operations by making the models more compressible.

- Some forms of optimization can decrease *latency* (the amount of time it takes to run a single inference with a given model) by reducing the amount of computation required to run inference.

- Some hardware accelerators, such as the Edge TPU, can run inference extremely fast with models that have been correctly optimized.

3.1 Model Optimization

It is possible to divide the types of model optimization into two categories as shown in the Fig. 19: the first one is pertinent to the *model architecture optimizations* and the second part is related to the *model deployment optimizations*.

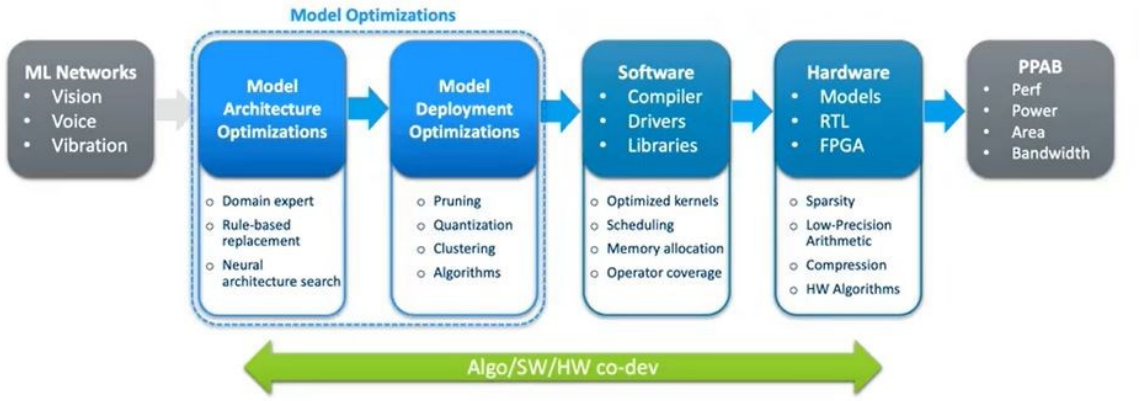

Fig.19 Optimization Process

Typically, the models that are designed for different scenarios have substructures in their architecture; these will become performance bottlenecks when deployed onto the edge devices. So, in *model architecture optimizations,* the aim is to actually change those bottlenecks replacing them with something that is more performant. In this section, there are few typical approaches; one is to have a *domain expert*. The expert, not only needs to know very well the problem domain such as vision or voice but he also needs to know the downstream software and hardware. Then, with some good dose of creativity, the expert can come up with handcrafted model that has optimized architecture for the given target. Once the experts develop patterns or rules or knowledge to do such architecture optimization, you can distill the knowledge (the process of transferring knowledge from a large model to a smaller one) into a *rule-based replacement* system, which can automate a large portion of crafting. Finally, the *neural architecture search* is promising and also powerful in terms of the generating optimized architectures for a given data set and a given target.

Regarding *model deployment optimization,* techniques such as *pruning*, *quantization*, *clustering* and *algorithms* are included.

For *software*, there are compilers, drives and libraries where you can see different levels and different coverage of optimized kernels, for a very clever way of scheduling work and memory allocation.

Therefore, as far as *hardware* is concerned, it is critical that, at the model optimization stage, the good attributes must be properly introduced into the original models because otherwise the un-optimized models will not be able to trigger to validate any of the features we have available.

3.1.1 General Methods for Model Deployment Optimization

The increase in depth and width of AI models is a direct and effective optimization method. However, increasing the depth and width of AI models will not only improve performance but also make AI models larger and more complex. So, this optimization method will make AI models more difficult to train, which may lead to the consumption of more hardware resources and cause additional training delays. Particularly on edge devices, such as mobile and Internet of Things (IoT), resources are further constrained, and model size and efficiency of computation become a major concern.

The main idea is to craft engineered models that are more efficient in terms of speed, memory and storage. Some of the widely used techniques in the industry for creating optimized deep learning models are listed here under:

1. Pruning

2. Quantization

3. Clustering

4. Regularization

3.1.1.1 Pruning

Based on the techniques used for pruning, the pruning methods can be classified as Penalty Term Methods, Cross Validation Methods, Magnitude Based Methods, Mutual Information (MI) Based Methods, Evolutionary Pruning Methods, Sensitivity Analysis (SA) Based Methods and Significance Based Pruning Methods. [RIF]

In general, pruning is defined as a network trimming within the assumed initial architecture. Magnitude-based weight pruning gradually zeroes out model weights during the training process to achieve model sparsity. Sparse models are easier to compress, and we can skip the zeroes during inference for latency improvements. This can be accomplished by estimating the sensitivity of the total error to the exclusion of each weight in the network. The weights or neurons which are insensitive to the error changes can be discarded after each step of training.

The pruned network is of a smaller size and it is likely to give higher accuracy than before its trimming.

Thus, pruning algorithms are used to remove the redundant connections while maintaining the networks performance. Several pruning algorithms are available in the literature to determine irrelevant neurons.

As far as edge intelligence is concerned, pruning describes a set of techniques to trim network size, by nodes not layers, to improve computational performance and sometimes resolution performance. The aim of these techniques is removing nodes from the network during training by identifying those nodes which, if removed from the network, would not noticeably affect network performance.

The method for finding "unimportant" nodes to be removed, is to look at the weight matrix after training; the nodes which have weights very close to the zero are often removed during pruning.

3.1.1.2 Quantization

In general, the process of quantization is associated with the reduction of the matrix dimension via limited precision of the numbers. However, the training and inference stages of deep learning neural network are limited by the space of the memory and by a variety of factors including programming complexity and even reliability of the system.

Quantization is the process of transforming an ML program into an approximated representation with available lower precision operations. It represents the easy way to use trained models to make inferences quickly, reducing operations costs, calculation loads and memory consumption.

Quantization is a process to represent a continuous value expressed as a real number to an integer multiple, i.e. you can take a continuous value expressed as a floating-point value and represent this as an integer value using as few bits as possible. This means that calculations are performed using integer calculations instead of floating-point calculations.

Let's see a brief introduction of floating/fixed-point representation. Floating point uses a *mantissa* and an *exponent* to represent real values: the *exponent* allows to represent a wide range of numbers, and the *mantissa* gives the precision. You can use integers to represent the value of a number relative to the exponent, replacing it by a fixed scaling factor.

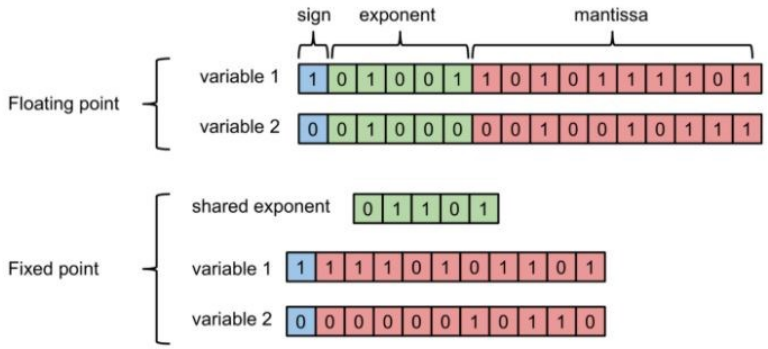

Fig. 20 Floating/Fixed-point representation

Quantization works by reducing the precision of the numbers used to represent model's parameters, which by default are 32-bit floating point numbers. This results in a smaller

model size and faster computation. For example, going from the 32-bit floats to 8-bit integers will reduce the memory making models 4x smaller. Then, integer operations are typically faster to execute and they also consume less power.

There are three different quantization types:

- *Reduced float quatization*: it basically reduces float32 in float16 parameters, so that's pretty straightforward.
- *Hybrid quantization*: it makes use of 8-bit for the parameters leaving 32-bit floats biases and activations.
- *Integer quantization*: all the parameters and operations are integers. It converts 32-bit floating-point numbers (such as weights and activation outputs) to 8-bit fixed-point numbers increasing inferencing speed.

Moreover, it is possible to distinguish two types of quantization tools: one works post training and it is called *post-training quantization* and the other works during training, called *quantization aware training*:

- The process in post training is very simple: you basically assume that you just train a model in high precision. Then currently, via the TensorFlow Lite converter, you just convert this model to TensorFlow Lite and quantize it. Then you just have a model that you can execute on whatever hardware is supported in that quantization type.
- The process during training consists in build training model in Keras, apply quantization, train model with added quantization operations, quantize via TensorFlow Lite converter producing a quantized model that finally you can execute in different hardware.

Quantization techniques are particularly effective when applied during training and can improve inference speed by reducing the number of bits used for model weights and activations.

Neural networks are typically trained in a floating point but such high precision is not necessarily needed for inference. In many applications it has been shown that in 8-bit integer arithmetic is sufficient to get good accuracy and good performance. Typically, a floating point data can be quantized to intake by using a simple equation. This equation converts FP32 CNNs into INT8 without significant accuracy loss, because INT8 math has higher throughput, and lower memory requirements.

The Fig. 21 shows the quantization workflow that is used in the industry.

Starting with a pre-trained floating point model, the first step is to quantize the weights: using divergence operator you find the optimal quantization ranges and then you evaluate the model to make sure that accuracy is preserved. However, finding the quantization ranges for activations is not easy because you don't have access to the activations. You have to run the model through some calibration data to capture the histograms or the statistics of the activations. Typically, a few hundreds of calibration data are sufficient to capture the statistics of activations. From these statistics you can find the activation quantization ranges and after that you can evaluate the model with the weights and activations quantized.

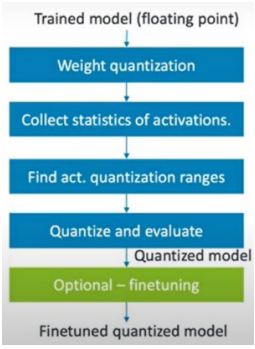

Fig.21 Quantization Workflow

Optionally, if training data is available, you can do quantization away training or basically fine tuning with quantization nodes or quantization constraints in the graph, improving accuracy.

3.1.1.3 Clustering

Clustering, or weight sharing, reduces the number of unique weight values in a model, leading to benefits for deployment. It first groups the weights of each layer into N clusters, then shares the cluster's centroid value (value that represents the middle of a cluster) for all the weights belonging to the cluster.

This technique brings improvements via model compression. Future framework support can unlock memory footprint improvements that can make a crucial difference for deploying deep learning models on embedded systems with limited resources.

Please note that clustering will provide reduced benefits for convolution and dense layers that precede a batch normalization layer.

In particular, if weights are represented numerically, it is possible to apply clustering techniques to them in order to identify groups of similar weights. This is precisely how weight clustering for model optimization works. By applying a clustering technique, it is possible to reduce the number of unique weights that are present in a machine learning model.

3.1.1.4 Regularization

A good way to reduce overfitting is to regularize the model: the fewer degrees of freedom it has, the harder it will be for it to overfit the data. Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more *regular*. This is called *weight regularization*, and it’s done by adding to the loss function of the network a *cost* associated with large weights. There are two types of *cost*:

- L1 regularization, the cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights);

- L2 regularization, the cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks.

3.1.2 Model Optimization for edge devices

AI models should be rationally modified and optimized according to the hardware and software characteristics of a specific edge device, in order to be deployed on it. In this section, efforts for running AI on edge devices are differentiated and presented.

Model Input: Each application scenario has specific optimization spaces. Narrowing down the classifier's searching space and dynamic Region-of-Interest (RoI), i.e. encoding to focus on target objects in video frames, can reduce the bandwidth consumption and data transmission delay. Though this kind of methods can significantly compress the size of model inputs and hence reduce the computation overhead without altering the structure of AI models, it requires a deep understanding of the potential optimization space.

Model Structure: Pointwise group convolution and channel shuffle, paralleled convolution and pooling computation and depth-wise separable convolution can greatly reduce computation cost while maintaining accuracy.

Model Selection: With various AI models, choosing the best one from available AI models in the edge requires weighing both precision and inference time. The model selection can be determined by a predictor along with a set of automatically tuned features of the model input. Besides, combining different compression techniques (such as model pruning), multiple compressed AI models (with different trade-offs between the performance and the resource requirement) can be derived.

Model Framework: Given the high memory footprint and computational demands of DL, running them on edge devices requires expert-tailored software and hardware

frameworks. A software framework is valuable if it (1) provides a library of optimized software kernels to enable deployment of DL; (2) automatically compresses AI models into smaller dense matrices by finding the minimum number of non-redundant hidden elements; (3) performs quantization and coding on all commonly used DL structures; (4) specializes AI models to contexts and shares resources across multiple simultaneously executing AI models. With respect to the hardware, running AI models on Static Random Access Memory (SRAM) achieves better energy savings compared to Dynamic RAM (DRAM). Hence, DL performance can be benefited if hardware directly supports the running optimized AI models on the on-chip SRAM.

3.2 Segmentation of AI models

Most of the intelligent applications only perform in the cloud while the edge devices play a role of collecting and uploading the data. This brings a heavy burden to the cloud.

Common deep learning model is composed of multilayer neural networks. Owing to this, it was thought to push part or all of the computing tasks in the edge, by the segmentation of deep learning model. In this way, a large number of computing tasks can be decomposed to different parts. Different device can solve the problem collaboratively.

The delay and power consumption of the most advanced AI models are evaluated on the cloud and edge devices, finding out that uploading data to the cloud is the bottleneck of current AI servicing. Dividing the AI model and performing distributed computation can achieve better end-to-end delay performance and energy efficiency. In addition, by pushing part of DL tasks from the cloud to the edge, the throughput of the cloud can be improved. Therefore, the AI model can be segmented into multiple partitions and then allocated to (1) heterogeneous local processors (e.g., GPUs, CPUs) on the end device, (2) distributed edge nodes, or (3) collaborative “end–edge–cloud” architecture. Partitioning the AI model horizontally, i.e., along the end, edge, and cloud, is the most common segmentation method. The process of data analysis is usually divided into two parts. One part is processed in the edge and the other part is processed in the cloud. Since the

uploading reduces intermediate data instead of input data, the network traffic between the edge and the cloud is reduced and the risk of security and privacy leakage in data transmission is avoided.

Another kind of model segmentation is vertically partitioning particularly for CNNs. In contrast to horizontal partition, vertical partition fuses layers and partitions them vertically in a grid fashion, and thus divides CNN layers into independently distributable computation tasks.

3.3 Early Exit of Inference (EEoI)

Model compression and model segmentation can facilitate deploying AI in the edge computing network. However, both of them have disadvantages: the first one may jeopardize the model accuracy irreversibly, while the second one may cause large communication overhead between segmented models. To reach the best trade-off between model accuracy and processing delay, multiple AI models with different model performance and resource cost can be maintained for each AI service. This idea can be improved by EEoI.

EEoI is an application that provides an efficient method for running Neural Network model inference on end devices.

A typical neural network uses a series of computational layers to sequentially transform input data. The output of one layer feeds into the next, with the output of the last layer producing the final model output. Models with more layers have the capacity to achieve higher inference accuracy over more complex data, but also have a larger memory footprint and have slower inference speed.

However, most data sources contain samples of varying levels of complexity. Within a data set, some samples are more well behaved, strictly adhering to a primary distribution. On the other hand, other samples will contain more unique elements or noise. To design a neural network architecture that only uses as many layers as it needs for inferring any given sample Early Exit on Inference is implemented.

EEoI incorporates multiple output layers, at various layer depths within a neural network. Each output layer is learning to accurately produce classification predictions based only upon the information that has been processed within the layers that have come before it. Output layers that are deeper in the network have the opportunity to learn more complex distributions, leading to better inference accuracy than the preceding output layers. During inference, each successive output layer is observed. Inference continues through the proceeding layers until an output layer makes a prediction with a sufficiently high probability, or until the final output layer is reached. The prediction probability threshold for an EEoI is a pre-set hyper-parameter, with a trade-off between overall inference accuracy and inference speed. A higher probability threshold produces more accurate, but slower inference, on average.

4. Edge hardware for AI

The support of edge computing is required for the huge deployment of AI services, especially for mobile AI. Customized edge hardware and corresponding optimized software frameworks and libraries can help AI execution more efficiently. In particular, the edge computing architecture can enable the offloading of AI computation and a well-designed edge computing frameworks can better maintain AI services running on the edge.

AI applications are more valuable if directly enabled on lightweight edge devices, such as mobile phones, wearable devices and surveillance cameras that are nearby to the location of events. Low-power IoT edge devices can be used to undertake lightweight AI computation hence avoiding communication with the cloud but, still face several challenges: (1) limited computation resources; (2) limited memory footprint; (3) limited energy consumption bottleneck.

The principal requirements of Edge computing hardware are multiple. First of all, it must be rugged and compact, it has to have sufficient storage, have rich connectivity options, have a wide power range, and finally it has to meet the performance requirements for the tasks they will perform. Edge computers must satisfy these requirements as they are often deployed in harsh environments where they must operate reliably and optimally.

Edge AI hardware can be classified into three categories according to their technical architecture: (1) Graphics Processing Unit (GPU) based-hardware, (2) Field Programmable Gate Array (FPGA) based-hardware and (3) Application-Specific Integrated Circuit (ASIC) based-hardware.

4.1 Mobile CPUs and GPUs

GPU was originally designed to process a large number of concurrent computing tasks. Therefore, GPU has many arithmetic and logic units (ALU) and very few caches purpose of which is not to save the data that needs to be accessed later but to improve services for threads. Today, Graphics Processing Unit (GPU), hardware based architecture, is the main architecture in the field of artificial intelligence hardware because of its outstanding computing power. It tends to have good compatibility and performance, but generally consumes more energy, e.g., NVIDIA' GPUs based on Turing architecture.

4.2 FPGA-based solutions

Most of the existing edge servers are structured based on CPUs with tightly coupled co-processors (e.g., GPUs). However, CPUs and GPUs are power-hungry and have limited energy efficiency, creating enormous difficulties for deploying them in energy or thermal constrained application scenarios. Therefore, future edge servers call for a new general-purpose computing system stack at low power consumption and high energy efficiency. The FPGAs, field-programmable gate array, are highly energy-efficient and adaptive to a variety of workloads. FPGA is an integrated circuit which can be "field" programmed to work as per the intended design. It means it can work as a microprocessor, or as an encryption unit, or graphics card, or even all these three at once. As implied by the name itself, the FPGA is field programmable. So, an FPGA working as a microprocessor can be reprogrammed to function like the graphics card in the field. The designs running on FPGAs are generally created using hardware description languages such as VHDL and Verilog. Due to its flexibility in hardware, FPGA has many advantages in the field of latency, connectivity and energy efficiency compared with GPU.

4.3 TPU-based solutions

The most obvious feature of ASIC (Application-Specific Integrated Circuit) is that it is developed for specific needs, so it has incomparable advantages respect to other hardware in specific scenarios. Therefore, during the development of edge intelligence, ASIC-based hardware architecture can help edge intelligence achieve better results which could be more stable in terms of performance and power consumption, such as Google's TPU. Edge TPU is Google's specially designed ASIC to perform AI at the edge. With high performance, a small footprint and low power consumption, it enables the implementation of high-precision AI services at the edge. Edge TPU complements the Cloud TPU by performing inferencing of trained models at the edge. It delivers high performance in a small physical and power footprint, enabling the deployment of high-accuracy AI at the edge. These purpose-built chips can be used for emerging use cases such as predictive maintenance, anomaly detection, machine vision, robotics, voice recognition, and many more. Developers can optimize TensorFlow models for Edge TPU by converting them to TensorFlow Lite models compatible with the edge. Unlike NVIDIA and Intel edge platforms, Google's Edge TPU cannot run models other than TensorFlow at the edge.

With the continuous upgrade of hardware, more and more AI applications will be able to run in the Edge and make our life better.

References

Hands-On Machine Learning with Scikit-Lear - Aurelien Geron

François Chollet - Deep Learning with Python-Manning (2018)

Z. Fadlullah et al., "State-of-the-Art Deep Learning: Evolving Machine Intelligence Toward Tomorrow's Intelligent Network Traffic Control Systems," IEEE Commun. Surveys & Tutorials, DOI: 10.1109/COMST.2017.2707140.

Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing, He Li, Kaoru Ota, and Mianxiong Dong

H. Li, K. Ota, M. Dong, Learning IoT in edge: deep learning for the Internet of Things with edge computing. IEEE Netw. ***32****(1), 96–101 (2018)*

S.S. Haykin, K. Elektroingenieur, Neural Networks and Learning Machines (Pearson Prentice Hall, Englewood Cliffs, 2009)

R. Collobert, S. Bengio, Links between perceptrons, MLPs and SVMs, in Proceeding of the Twenty-first International Conference on Machine Learning (ICML 2004) (2004), p. 23

C.D. Manning, C.D. Manning, H. Schütze, Foundations of Statistical Natural Language Processing (MIT Press, New York, 1999)

M.D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, in 2014 European Conference on Computer Vision (ECCV 2014) (2014), pp. 818–833

J. Schmidhuber, Deep learning in neural networks: an overview. Neural Netw. ***61****, 85–117 (2015)*

S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. ***9****(8), 1735–1780 (1997)*

S.J. Pan, Q. Yang, A survey on transfer learning. IEEE Trans. Knowl. Data Eng. ***22****(10), 1345–1359 (2010)*

G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network (2015). arXiv preprint:1503.02531

Y. Jiang, S. Wang, B.J. Ko, W.-H. Lee, L. Tassiulas, Model pruning enables efficient federated learning on edge devices (2019). Preprint. arXiv:1909.12326

L. Du et al., A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things. IEEE Trans. Circuits Syst. Regul. Pap. ***65****(1), 198–208 (2018)*

S. Jiang, D. He, C. Yang et al., Accelerating mobile applications at the network edge with software-programmable FPGAs, in 2018 IEEE Conference on Computer Communications (INFOCOM 2018) (2018), pp. 55–62

www.tensorflow.org/model_optimization/guide

www.tensorflow.org/model_optimization/guide/pruning

www.tensorflow.org/model_optimization/guide/quantization/training

www.tensorflow.org/model_optimization/guide/clustering

https://heartbeat.fritz.ai/8-bit-quantization-and-tensorflow-lite-speeding-up-mobile-inference-with-low-precision-a882dfcafbbd

https://medium.com/swlh/machine-learning-model-optimization-for-intelligent-edge-d0f400111002

INTEGER QUANTIZATION FOR DEEP LEARNING INFERENCE: PRINCIPLES AND EMPIRICAL EVALUATION-HaoWu1 Patrick Judd1 Xiaojie Zhang1 Mikhail Isaev2_ Paulius Micikevicius1 arXiv:2004.09602v1 [cs.LG] 20 Apr 2020

Efficient adaptive inference for deep convolutional neural networks using hierarchical early exits-Nikolaos Passalis, Jenni Raitoharju, Anastasios Tefas, Moncef Gabbouj- Pattern Recognition 105 (2020) 107346

Sample-weighted clustering methods-Jian Yua, Miin-Shen Yangb,∗, E. Stanley Lee

Pruning algorithms of neural networks - a comparative study-M. Gethsiyal Augasta1_, T. Kathirvalavakumar2†

A review of edge computing reference architectures and a new global edge proposal - Inés Sittón-Candanedo a,∗, Ricardo S. Alonso a, Juan M. Corchado Sara Rodríguez-González a,∗, Roberto Casado-Vara Comparison of Edge Computing Implementations: Fog Computing, Cloudlet and Mobile Edge Computing-Koustabh Dolui Soumya Kanti Datta, 978-1-5090-5284739-03/17/$31.00 ©2017 IEEE

A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing- Mário P. Véstias- Algorithms 2019, 12, 154; doi:10.3390/a12080154

Are FPGAs Suitable for Edge Computing?- Saman Biookaghazadeh, Ming Zhao, Fengbo Ren