Classification - Neural Network

Unit 3: Classification
Artificial Neural Network

–

School of Mathematical Sciences
Institute of Science and Technology (IoST), TU

Feb, 2025

Table of contents

Neural Networks

What is neural network?

A neural network is a machine learning model inspired by the human brain, designed to recognize patterns and make predictions.

Two Views

- A way to **simulate biological learning** by simulating the nervous system.
- A way to increase the power of known models in machine learning by stacking them in careful ways as **computational graphs**
  - The number of nodes in the computational graph controls learning capacity with increasing data
  - The specific architecture of the computational graph incorporates domain-specific insights (e.g., images, speech).
  - The success of deep computational graphs has led to the coining of the term **"deep learning"**.

History of ANN

Groundwork

1943: *Warren McCulloch* and *Walter Pitts* proposed the first mathematical model of a neuron, the **"McCulloch-Pitts neuron"** which could perform simple logical operations.

Publication: A Logical Calculus of the ideas Imminent in Nervous Activity [1]

This laid the theoretical groundwork for artificial neural networks (ANNs).

Reference

- https://www.cse.chalmers.se/ coquand/AUTOMATA/mcp.pdf
- McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron

History of ANN (cont.)

1958: *Frank Rosenblatt* introduced the **Perceptron**, a single-layer neural network capable of binary classification.

It was the first model implemented on a computer and sparked interest in machine learning. However, it could only solve linearly separable problems.

- Was implemented using a large piece of hardware.

- Generated great excitement but failed to live up to inflated expectation

AI Winter 1960s-1970s

- Minsky and Papert's Book "Perceptrons" (1969) showed that the perceptron only had limited expressive power
- Minsky is often blamed for setting back the field (fairly or unfairly)

Were Minsky/Papert justified in their pessimism?

History of ANN (cont.)

Did We Really Not Know How to Train Multiple Units?

It depends on who you ask

- AI researchers didn't know (and didn't believe it possible).
- Training computational graphs with dynamic programming had already been done in control theory (1960s)

History of ANN (cont.)

First attempt of backpropagation

Paul Werbos proposed backpropagation in his 1974 thesis (and was promptly ignored - formal publication was delayed)

Werbos (2006): "In the early 1970s, I did in fact visit Minsky at MIT. I proposed that we do a joint paper showing that MLPs can in fact overcome the earlier problems if (1) the neuron model is slightly modified to be differentiable; and (2) the training is done in a way that uses the reverse method, which we now call backpropagation in the ANN field. But Minsky was not interested. In fact, no one at MIT or Harvard or any place I could find was interested at the time."

History of ANN (cont.)

Seventies/Eighties - Era of logic

It was the era or work on logic and reasoning (discrete mathematics)

Work on continuous optimization had few believers

- Researchers like Hinton were certainly not from the main-stream
- This view has been completely reversed today
- The early favorites have little to show in spite of the effort

History of ANN (cont.)

Backpropagation: The Second Coming

Rumelhart, Hinton, and Williams wrote two papers on backpropagation in 1986 (independent from prior work)

1986: *David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams* popularized **backpropagation**, an algorithm for training multi-layer neural networks.

- This breakthrough allowed networks to learn complex, non-linear patterns, overcoming earlier limitations
- Networks with hidden layers (multilayer perceptrons, or MLPs) emerged, revitalizing interest in the field

History of ANN (cont.)

The Nineties

Acceptance of backpropagation encouraged more research in multilayer networks.

By the year 2000, most of the modern architectures had already been set up in some form.

- They just didn't work very well!
- The winter continued after a brief period of excitement

It was the era of the support vector machine.

The new view: SVM was the panacea (at least for supervised learning)

History of ANN (cont.)

The Deep Learning Revolution (2010s-Present)

2012: AlexNet, a deep convolutional neural network (CNN) developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, won the ImageNet competition by a wide margin. This demonstrated the power of deep learning, fueled by large datasets and GPU acceleration.

2010s-2020s: Innovations like Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), and Transformers (e.g., BERT, GPT) expanded neural networks into natural language processing (**NLP**), image generation, and beyond.

Today, neural networks underpin technologies like self-driving cars, virtual assistants, and advanced scientific research (e.g., protein folding with AlphaFold).

History of ANN (cont.)

2024 Nobel Prizes

The 2024 Nobel Prizes marked a significant moment for artificial intelligence (AI)

- Recognition of Foundational AI Contributions: The Nobel Prize in Physics was awarded to **John Hopfield and Geoffrey Hinton** for their pioneering work on artificial neural networks, which are the backbone of modern machine learning.
- AI as a Tool for Scientific Breakthroughs: The Nobel Prize in Chemistry went to **David Baker, Demis Hassabis, and John Jumper**, with **Hassabis and Jumper** recognized for developing AlphaFold, an AI system that solved the 50-year-old problem of protein structure prediction.

By awarding Nobels in Physics and Chemistry to AI-related work, the scientific community signaled that AI is **no longer just a computational tool** but a fundamental driver of progress across disciplines

How it All Started

The Biological Inspiration

Neural networks were originally designed to simulate the learning process in biological organisms.

The human nervous system contains cells called **neurons**.

The neurons are connected to one another with the use of synapses.

- The **strengths** of synaptic connections often **change in response to external stimuli**.
- This change causes learning in living organisms

Neural Network - The Biological Inspiration

Figure: (left) Biological Neuron, (right) ANN

- Neural networks contain computation units ⇒ Neurons
- The computational units are connected to one anotherthrough weights ⇒ Strengths of synaptic connections in biological organisms

Neural Network - Learning

Living organisms

- Synaptic weights change in response to external stimuli
- An unpleasant experience will change the synaptic weights of an organism, which will train the organism to behave differently

Artificial neural networks

- The weights are learned with the use of **training data**, which are input-output pairs (e.g., images and their labels)
- An error made in predicting the label of an image is the unpleasant "stimulus" that changes the weights of the neural network
- When trained over many images, the network learns to classify images correctly

Is ANN/DNN represent brain?

No one exactly knows how the brain works

The functions computed in a neural network are very different from those in the brain

Nevertheless, there are several examples, where the principles of neuroscience have been successfully applied in designing neural networks.

Convolutional neural networks are based on architectural principles drawn from the cat's visual cortex

Machine Learning versus Deep Learning

Machine Learning

For smaller data sets, traditional machine learning methods often provide slightly better performance
Traditional models often provide more choices, interpretable insights, and ways to handcraft features

Deep Learning

For larger data sets, deep learning methods tend to dominate

Perceptron

A Perceptron is one of the simplest types of artificial neural networks and serves as the foundational building block for more complex neural architectures.

A perceptron is a single-layer neural network that takes multiple binary or real-valued inputs, applies weights to them, sums them up, and passes the result through an activation function to produce an output.

Perceptron (cont.)

How do perceptron works?

A perceptron takes several binary inputs, $x_1, x_2, \ldots,$ and produces a single binary output:

In the example shown the perceptron has three **inputs**, x_1, x_2, x_3.
In general it could have more or fewer inputs.

Rosenblatt proposed a simple rule to compute the output.

He introduced **weights**, $w_1, w_2, \ldots$, real numbers R expressing the importance of the respective inputs to the output.

Perceptron (cont.)

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value.

Just like the weights, the threshold is a real number which is a parameter of the neuron.

To put it in more precise algebraic terms:

$$output = \begin{cases} 1, & if \sum_j w_j x_j \leq threshold \\ 0, & if \sum_j w_j x_j > threshold \end{cases}$$

That's all there is to how a perceptron works!

Perceptron (cont.)

A way you can think about the perceptron is that it's a **device that makes decisions by weighing up evidence**.

Example

Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

Perceptron (cont.)

Example

We can represent these three factors by corresponding binary variables x_1, x_2 and x_3. For instance, we'd have

- $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad
- $x_2 = 1$ if your boyfriend or girlfriend wants to go
- $x_2 = 0$ if not. And similarly again for x_3 and public transit

How you decide?

Now, suppose you absolutely adore cheese so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really want check the weather, and there's no way you'd go to the festival if the weather is bad.

Perceptron (cont.)

You can use perceptrons to model this kind of decision-making.

- One way to do this is to choose a weight $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions.
- The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit.

Finally, suppose you choose a **threshold of 5** for the perceptron.

With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad.

It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

Perceptron (cont.)

Threshold = 3

The perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your boyfriend or girlfriend was willing to join you.

In other words, it'd be a different model of decision-making.

$$output = \begin{cases} 1, & if \sum_j w_j x_j \leq threshold \\ 0, & if \sum_j w_j x_j > threshold \end{cases}$$

Dropping the threshold means you're more willing to go to the festival.

Multilayer Perceptron

The perceptron isn't a complete model of human decision-making!

But what the example illustrates is how a perceptron can weight up different kinds of evidence in order to make decisions.

And it should seem plausible that a complex network of perceptrons could make quite subtle decisions.

Multilayer Perceptron (cont.)

In this network, the first column of perceptrons – what we'll call the *first layer* of perceptrons – is making three very simple decisions, by weighing the input evidence.

What about the perceptrons in the second layer?
Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making.In this way a perceptron in the **second layer** can make a decision at a more complex and more **abstract level than perceptrons in the first layer**.

And **even more complex decisions can be made by the perceptron in the third layer**. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

Multilayer Perceptron (cont.)

Does perceptron has multiple output?

By defination, the perceptron has only one output. In must of the figures they look like they have multiple outputs.

In fact they are still single output.

The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as input to several other perceptrons.

Generalizing the perceptron

Let's observe the output of the perceptron. Try to identify the mathematical limitation ...

$$output = \begin{cases} 1, & if \sum_j w_j x_j \leq threshold \\ 0, & if \sum_j w_j x_j > threshold \end{cases}$$

Can we improve this?

Generalizing the perceptron (cont.)

Let's simplify the way we describe perceptrons.
The condition $\sum_j w_j x_j > threshold$ threshold is difficult to use, and we can make two *notational changes* to simplify it.

- The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x = \sum_j w_j x_j$, where w and x are vectors whose components are the weights and inputs, respectively.
- The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's *bias*, $b \equiv -threshold$.

Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} \tag{2}$$

Generalizing the perceptron (cont.)

Bias

We can think of the bias as a measure of how easy it is to get the perceptron to output 1.

Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to *fire*.

- For a perceptron with a really **big bias**, it's extremely easy for the perceptron to output 1.
- But if the bias is **very negative**, then it's difficult for the perceptron to output 1.

Computational universality of perceptrons

Another way perceptrons can be used is to **compute the elementary logical functions** we usually think of as underlying computation, functions such as AND, OR, and NAND.

For example, suppose we have a perceptron with two inputs, each with weight -2, and an overall bias of 3. Here's our perceptron:

Then we see that input 00 produces output 1, since $(-2) \times +(-2) \times +3 = 3$ is positive.

Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2) \times +(-2) \times +3 = -1$ is negative.

And so our perceptron implements a NAND gate!

Computational universality of perceptrons (cont.)

The computational universality of perceptrons is simultaneously reassuring and disappointing.

- It's **reassuring** because it tells us that networks of perceptrons can be as powerful as any other computing device.
- But it's also **disappointing**, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

Computational universality of perceptrons (cont.)

What make neural network different to conventional logic gates?

We can devise **learning algorithms** which can **automatically tune the weights and biases** of a network of artificial neurons.

This tuning happens in response to external stimuli, without direct intervention by a programmer.

These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates.

Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

What is Learning?

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network?

For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit.
To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible.

Learning (cont.)

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9".
We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9".
And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

Is this possible with perceptron?

Learning (cont.)

A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1.

That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way.

What is the solution?
We can overcome this problem by introducing a new type of artificial neuron called a **sigmoid neuron**. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. **That's the crucial fact which will allow a network of sigmoid neurons to learn**.

Sigmoid neurons

We'll depict sigmoid neurons in the same way we depicted perceptrons:

The sigmoid neuron has inputs, $x_1, x_2, \ldots$.
But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1.
Also just like a perceptron, the sigmoid neuron has weights for each input, $w_1, w_2, \ldots$, and an overall bias, b. But the output is not 0 or 1. Instead, it's $\sigma(wx + b)$, where σ is called the sigmoid function

Sigmoid neuron (cont.)

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs x_1,x_2,..., weights $w_1, w_2, \ldots$, and bias b is

$$\frac{1}{1+\exp\left(-\sum_j w_j x_j - b\right)}. \tag{4}$$

At first sight, sigmoid neurons appear very different to perceptrons.

In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

Sigmoid neuron(cont.)

When sigmoid act like perceptron?

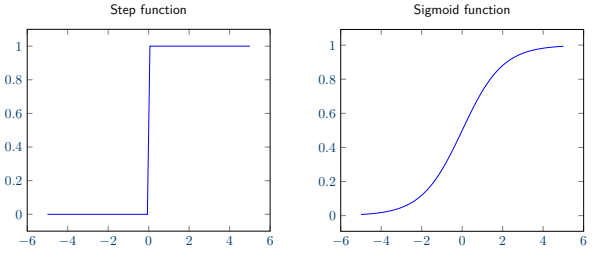
Suppose $z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$.

Suppose on the other hand that $z = w \cdot x + b$ is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$. So when $z = w \cdot x + b$ is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron.

It's only when $w \cdot x + b$ is of modest size that there's much deviation from the perceptron model.

Sigmoid vs. Perceptron

This shape is a smoothed out version of a step function:



Learning on Sigmoid

Why the smoothness of σ matters?

The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron.

How can we compute Δoutput?
Solution: Use calculus $\Rightarrow \Delta \text{output}$ is well approximated by

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b \tag{5}$$

Where, the sum is over all the weights, w_j, and $\partial \text{output}/\partial w_j$ and $\partial \text{output}/\partial b$ denote partial derivatives of the output with respect to w_j and b, respectively.

Learning on Sigmoid

How should we interpret the output from a sigmoid neuron?
Suppose we want the output from the network to indicate either **"the input image is a 9"** or **"the input image is not a 9"**.
Obviously, it'd be easiest to do this if the output was a 0 or a 1, as in a perceptron.

But in practice we can **set up a convention** to deal with this
For example, by deciding to interpret any output of at least 0.5 as indicating a "9", and any output less than 0.5 as indicating "not a 9".

Multilayer Neural Network

Suppose we have the network:

The leftmost layer in this network is called the input layer, and the neurons within the layer are called *input neurons*.

The rightmost or *output* layer contains the *output neurons*, or, as in this case, a single output neuron.

The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs.

The network above has just a single hidden layer, but some networks have multiple hidden layers.

Multilayer Neural Network

For example, the following four-layer network has two hidden layers:

Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or *MLPs*, despite being made up of sigmoid neurons, not perceptrons.

Designing input and output layer

How to design input and output layer?
The design of the input and output layers in a network is often straightforward.

For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not.

A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we'd have $4,096 = 64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1.

The output layer will contain just a single neuron, with output values of less than 0.5 indicating "input image is not a 9", and values greater than 0.5 indicating "input image is a 9".

Design of Hidden Layers

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers.

In particular, **it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb**.

Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets.

For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network.

Feedforware Neural Network

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called ***feedforward* neural networks**.

This means there are no loops in the network – information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

Recurrent Neural Network

There are models of artificial neural networks in which feedback loops are possible. These models are called **recurrent neural networks**.

The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration.

That causes still more neurons to fire, and so over time we get a cascade of neurons firing.

Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Recurrent Neural Network

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful.

But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks.

A ANN for classification

How OCR works?

We can split the problem of recognizing handwritten digits into two sub-problems. - **segmentation, recognition**

For example, we'd like to break the image into six separate images.

$\Rightarrow$

We humans solve this *segmentation problem* with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit.

We'll focus on writing a program to solve the second problem, that is, classifying individual digits.

ANN for Digit Recognition

To recognize individual digits we will use a three-layer neural network:

ANN for Digit Recognition (cont.)

The **input layer** of the network contains neurons encoding the values of the input pixels.

Suppose, our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons.

For simplicity We've omitted most of the 784 input neurons in the diagram.

The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey.

ANN for Digit Recognition (cont.)

The second layer of the network is a **hidden layer**.

We denote the number of neurons in this hidden layer by n, and we can experiment with different values for n. The example shown illustrates a small hidden layer, containing just $n = 15$ neurons.

ANN for Digit Recognition (cont.)

The **output layer** of the network contains 10 neurons.

If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0.

If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on.

A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

ANN for Digit Recognition (cont.)

After all, the goal of the network is to tell us which digit $(0, 1, 2, \ldots, 9)$ corresponds to the input image. **Four neurons are enough to encode the answer**

A seemingly natural way of doing that is to use just 4 output neurons, treating each neuron as taking on a binary value, depending on whether the neuron's output is closer to 0 or to 1.

Four neurons are enough to encode the answer, since $2^4 = 16$ is more than the 10 possible values for the input digit.

Why should our network use 10 neurons instead? Isn't that inefficient?

Dataset

How can it learn to recognize digits?

The first thing we'll need is a data set to learn from – a so-called **training data set**.

We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology.

These are 28 by 28 greyscale images.

Dataset (cont.)

We'll use the notation x to denote a training input.

It'll be convenient to regard each training input x as a $28 \times 28 = 784$-dimensional vector. Each entry in the vector represents the grey value for a single pixel in the image.

We'll denote the corresponding desired output by $y = y(x)$, where y is a 10-dimensional vector.

For example, if a particular training image, x, depicts a 6, then $y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$ is the desired output from the network.

Learning - Cost/Loss/Objective function

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x.To quantify how well we're achieving this goal we define a cost function

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{1}$$

Where, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum($\sum$) is over all training inputs x.C the *quadratic* cost function. It's also known as the *mean squared error*(MSE).

The output a depends on x, w and b, but to keep the notation simple we haven't explicitly indicated this dependence.

Learning - Cost function (cont.)

$C(w, b)$ is non-negative, since every term in the sum is non-negative.

$C(w, b)$ becomes small, i.e., $C(w, b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x.

Generalizing, our training algorithm has done a good job if it can find weights and biases so that $C(w, b) \approx 0$.

By contrast, it's not doing so well when $C(w, b)$ is large – that would mean that $y(x)$ is not close to the output a for a large number of inputs.

Learning - Cost function (cont.)

Objective

So the aim of our training algorithm will be to minimize the cost $C(w, b)$ as a function of the weights and biases.

In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as **gradient descent**.

Gradient Descent

Why introduce the quadratic cost? After all, aren't we primarily interested in the number of images correctly classified by the network? Why not try to maximize that number directly, rather than minimizing a proxy measure like the quadratic cost?

The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network. For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly.

That makes it difficult to figure out how to change the weights and biases to get improved performance.

If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost. That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

Gradient Descent (cont.)

Let's suppose we're trying to minimize $C(v)$. Where v could be any real-valued function of many variables, $v = v_1, v_2, \ldots$.

Note that we've replaced the w and b notation by v to emphasize that this could be any function – we're not specifically thinking in the neural networks context any more.

To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2. Our objective is to find where C achieves its global minimum.

For few variables the solution can be computed using derivtives. But it'll turn into a nightmare when we have many more variables.

Gradient Descent (cont.)

We'd randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley.

We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C – those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.

Its more like physics simulation of gravity....

Gradient Descent (cont.)

Let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \tag{2}$$

We're going to find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative; i.e., we'll choose them so the ball is rolling down into the valley.

Gradient Descent (cont.)

To figure out how to make such a choice it helps to define Δv to be the vector of changes in v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, where T is again the transpose operation, turning row vectors into column vectors.

We'll also define the *gradient* of C to be the vector of partial derivatives, $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$. We denote the gradient vector by ∇C, i.e.:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{3}$$

∇C is a gradient vector.

Gradient Descent (cont.)

With these definitions, the expression (2) for ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{4}$$

This equation helps explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C, just as we'd expect something called a gradient to do.

Gradient Descent (cont.)

But what's really exciting about the equation is that it lets us see how to choose Δv so as to make ΔC negative. In particular, suppose we choose

$$\Delta v = -\eta \nabla C, \tag{5}$$

where η is a small, positive parameter (known as the *learning rate*).

Then Equation (4) tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$.

Because $\|\nabla C\|^2 \geq 0$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase, if we change v according to the prescription in (5).

Gradient Descent (cont.)

This is exactly the property we wanted! And so we'll take Equation (5) to define the "law of motion" for the ball in our gradient descent algorithm. That is, we'll use Equation (5) to compute a value for Δv, then move the ball's position v by that amount.

$$v \to v' = v - \eta \nabla C. \tag{6}$$

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until – we hope – we reach a global minimum.

Gradient Descent (cont.)

To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation (4) is a good approximation.

If we don't, we might end up with $\Delta C > 0$, which obviously would not be good!

At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly.

In practical implementations, η is often varied so that Equation (4) remains a good approximation, but the algorithm isn't too slow.

Gradient Descent (cont.)

We've explained gradient descent when C is a function of just two variables. But, in fact, everything works just as well even when C is a function of many more variables.

Suppose in particular that C is a function of m variables, $v_1, \ldots, v_m$. Then the change ΔC in C produced by a small change $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v \tag{7}$$

where the gradient ∇C is the vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T \tag{8}$$

Gradient Descent (cont.)

Just as for the two variable case, we can choose

$$\Delta v = -\eta \nabla C \tag{9}$$

and we're guaranteed that our (approximate) expression (7) for ΔC will be negative.

This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \to v' = v - \eta \nabla C \tag{10}$$

Learning with Gradient Descent

How can we apply gradient descent to learn in a neural network?

The idea is to use gradient descent to find the weights w_k and biases b_l which minimize the cost in Equation (1).

To see how this works, let's restate the gradient descent update rule, with the weights and biases replacing the variables v_j. In other words, our "position" now has components w_k and b_l, and the gradient vector ∇C has corresponding components $\partial C/\partial w_k$ and $\partial C/\partial b_l$.

Learning with Gradient Descent

Writing out the gradient descent update rule in terms of components, we have

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

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l} \tag{12}$$

Where, $C = \frac{1}{n}\sum_x C_x$ and $C_x \equiv \frac{\|y(x)-a\|^2}{2}$

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.

Limitation of Gradient Descent

Update Rules $b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$, $C = \frac{1}{n}\sum_x C_x$, and $C_x \equiv \frac{\|y(x)-a\|^2}{2}$

Can we see any issue with this update rule?

In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$.

Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

Stochastic Gradient Descent

An idea called **stochastic gradient descent** can be used to speed up learning.

The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs.

By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

Stochastic Gradient Descent (cont.)

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network.

Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \to w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{13}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{14}$$

Where the sums are over all the training examples X_j in the current mini-batch.

History

Learning representations by back-propagating errors:
https://www.cs.toronto.edu/ hinton/absps/naturebp.pdf