11-785 Introduction to Deep Learning: Notes

Feiyang Chen

feiyangchen98@gmail.com

March & April 2019

Contents

1 Deep learning Theory

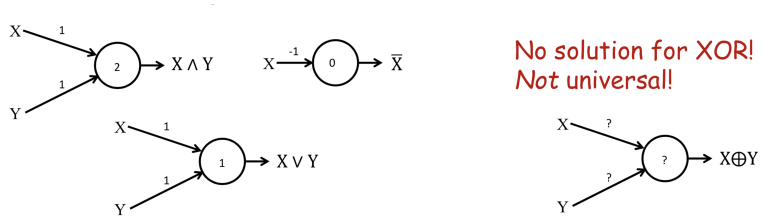
1.1 MLP: Intuitive Understanding

Question: Why can a neural network be used as a universal approximator to fit any function? — Intuitive Understanding

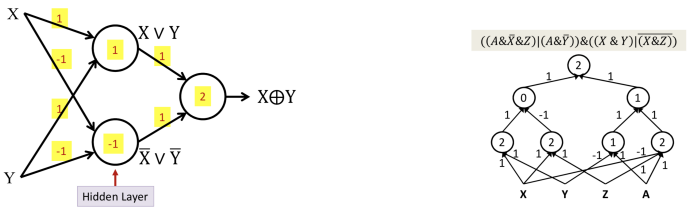
From the perspective of the intuitive understanding of the perceptron, it can be discussed from four angles:

1. (0,1) input: Boolean operation

- A single perceptual function can simulate and/or/not gate, but it cannot represent XOR operation



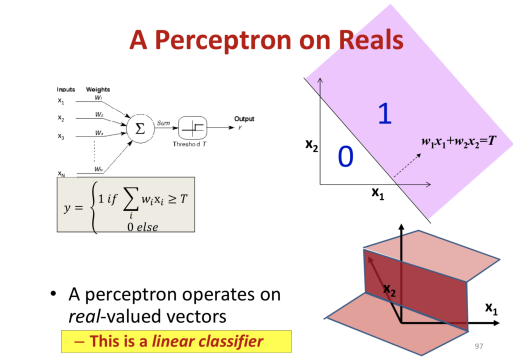
- Combine multiple perceptrons to represent XOR operations only when dealing with inputs and outputs (left figure)



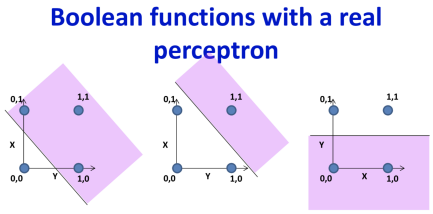
- Therefore, the perceptron has the ability to construct arbitrary functions of Boolean operations (right figure)

2. Real input + discrete output: linear classifier

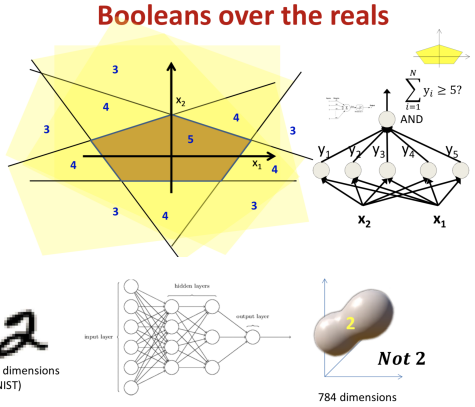
- When the input and output are real, the perceptron is activated if the weighted sum exceeds the threshold. Essentially, the perceptual machine of real input will form a linear classifier.

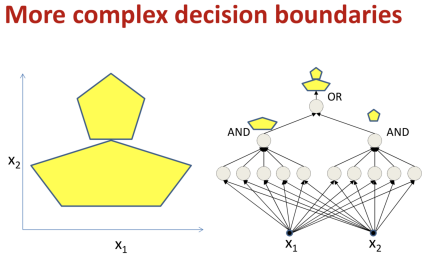


- It's easy to find 3 classifiers to fit Boolean operations (and/or/not), but XOR is impossible



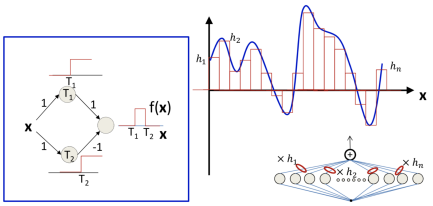
- Therefore, when any number of linear classifiers are properly combined, any decision surface can be represented





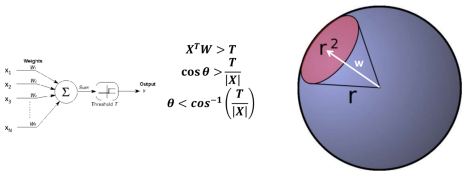
3. Real input + real output: regression problem

MLP as a continuous-valued regression

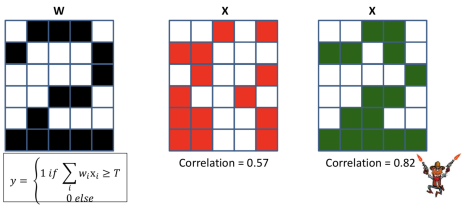


- A simple 3-unit MLP can generate a "square pulse" over an input
- **An MLP with many units can model an arbitrary function over an input**
  - To arbitrary precision
    - Simply make the individual pulses narrower
- This generalizes to functions of any number of inputs (next class)

4. Intuitive reasons for modeling capabilities: the relationship between input x and weight w

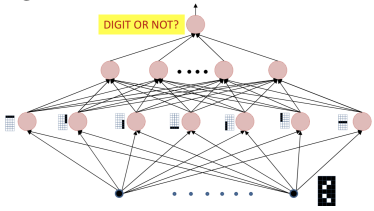


- The perceptron fires if the input is within a specified angle of the weight
- Neuron fires if the input vector is close enough to the weight vector.
  - If the input pattern matches the weight pattern closely enough



- If the *correlation* between the weight pattern and the inputs exceeds a threshold, fire
- The perceptron is a *correlation filter!*

In summary, the weight of the high-level perceptron reduces the input information of the lower layer as much as possible, and continuously retains and reorganizes the identified multiple features, thereby realizing the ability of pattern recognition.



SUMMARY

- Neural network based AI has taken over most AI tasks
- Neural networks originally began as computational models of the brain
  - Or more generally, models of cognition
- The earliest model of cognition was associationism

- The more recent model of the brain is connectionist
  – Neurons connect to neurons
  – The workings of the brain are encoded in these connections
- Current neural network models are connectionist machines – The comprise networks of neural units
- McCullough and Pitt model: Neurons as Boolean threshold units
  – Models the brain as performing propositional logic
  – But no learning rule
- Hebb's learning rule: Neurons that fire together wire together
  – Unstable
- Rosenblatt's perceptron : A variant of the McCulloch and Pitt neuron with a provably convergent learning rule
  – But individual perceptrons are limited in their capacity (Minsky and Papert)
- Multi-layer perceptrons can model arbitrarily complex Boolean functions
- MLPs are classification engines
  – They can identify classes in the data
  – Individual perceptrons are feature detectors
  – The network will fire if the combination of the detected basic features matches an "acceptable" pattern for a desired class of signal
- MLP can also model continuous valued functions
- Perceptronsarecorrelationfilters
  – They detect patterns in the input
- Interesting AI tasks are functions that can be modelled by the network

1.2 MLP: Theoretical Proof

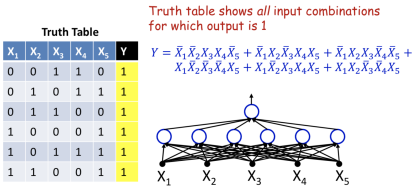
Question: Why can a neural network be used as a universal approximator to fit any function? — Theoretical Proof

Lecture1 mainly introduces the neural network as an intuitive understanding of the universal approximator. Lecture2 turns to a more in-depth theoretical proof, which is still discussed from three aspects: Boolean operation, linear classifier and regression problem.

1. Boolean operation

Multi-layer perceptrons can construct Boolean operations in any form and combination. Then, how many layers and how many neurons are perceived by a common Boolean function?

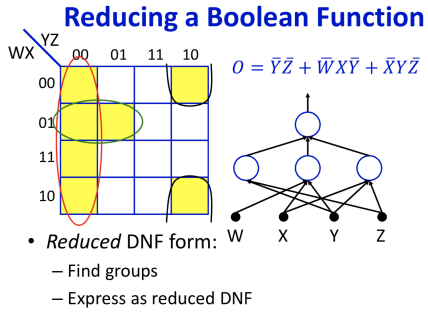
- Truth Table + DNF Formula: Exhaustive Method



Notes: Any Boolean operation can be exhausted by the truth table. Each row of the truth table corresponds to each item of the DNF formula. Each item of the DNF formula corresponds to each node of the network hidden layer.

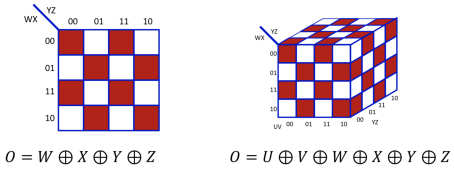
Conclusion 1.1: A one-hidden-layer MLP is a Universal Boolean Function

- Karnaugh map: simplification of truth table



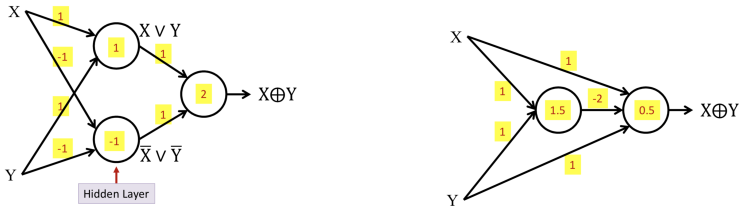
Notes: A four-variable Karnaugh map with yellow for 1 and white for 0. Combine the maximum 2^n yellow 1 adjacent to the horizontal/vertical direction to simplify the truth table DNF formula, thus simplifying the number of hidden layers.

Worst case: In the Karnaugh map, any squares with a true value of 1 cannot be merged, and the DNF formula cannot be simplified.

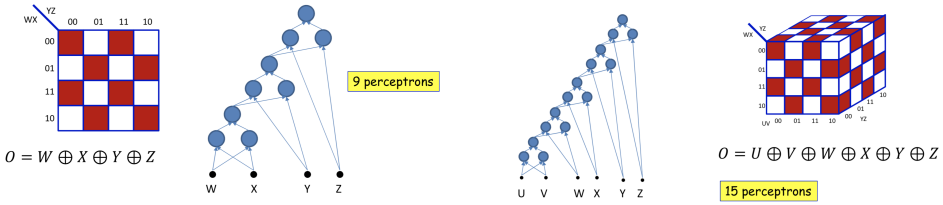


Conclusion 1.2: Under the worst Karnaugh map, the maximum number of neurons in a layer of perceptron networks is 2^{N-1}. That is, the number of white squares. The number of neurons is an exponential multiple of the number of variables.

- In the worst case, use the XOR method: increase the number of layers and reduce the number of neurons



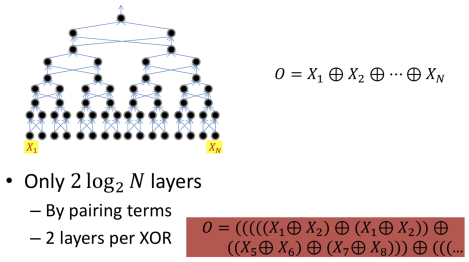
Notes: Two representations of XOR.



Notes: In the worst case four-variable and six-variable Karnaugh maps, use the XOR method to increase the number of layers and reduce the number of neurons.

Conclusion 1.3: Under the worst-case Karnaugh map, the maximum number of neurons in a multi-layer perceptron network is $3(N-1)$ or $2(N-1)$, and the number of network layers is $2(N-1)$ using the XOR method. The number of neurons and the number of network layers are constant multiples of the number of variables.

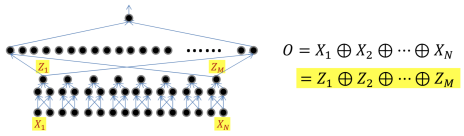
- In the worst case, the law of the XOR method: reducing the number of layers



Notes: In the worst case, the N-variable XOR method reduces the number of layers by combining the laws.

Conclusion 1.4: Under the worst-case Karnaugh map, using the XOR method, the number of network layers is reduced to $2log_2N$ in a multi-layer perceptron network.

- In the worst case, specify the K+1 layer: the trade-off between the number of layers and the number of neurons

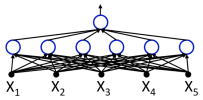


Notes: Weigh the number of layers and the number of neurons, specify the K+2 layer, which is equivalent to the pre-K layer XOR method, and the K+1 layer Karnaugh map method.

Conclusion 1.5.1: Under the specified k+2 layer, as of the kth layer, according to the XOR method, the number of neurons $= 3\sum_1^k \frac{N}{2^i}$, at the k+1th layer, according to the Karnaugh map method, the number of neurons $= 2^{(\frac{N}{2^i}-1)}$, and the last layer, the number of neurons = 1. That is, the final number of neurons = 3 parts of the sum.

Conclusion 1.5.2: If you want to limit the number of layers before the minimum number of layers, it is inevitable that you need to increase the number of neurons in the subsequent layers exponentially. Otherwise, you will not be able to fully cover all the possibilities of the truth table. Cannot fully fit the specified Boolean operation.

- Multilayer perceptron network parameter number

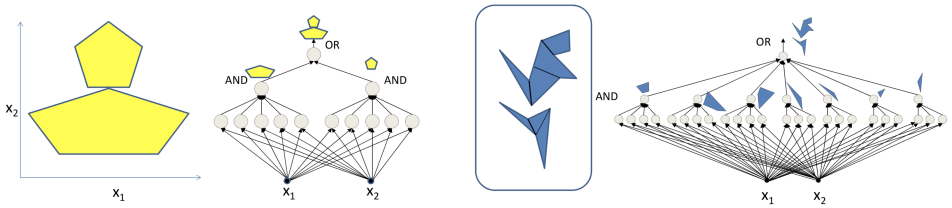


Notes: The number of parameters is the number of connections in the network. Very important in hardware and software implementation.

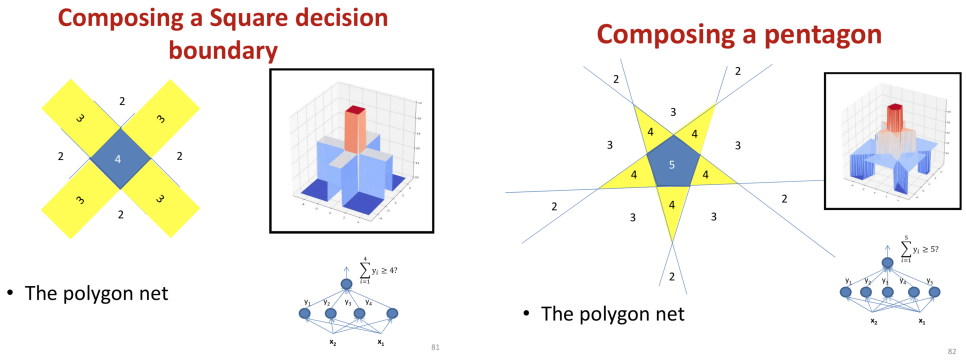
Conclusion 1.6: The number of parameters depends on the number of neurons. If the number of neurons is exponential, the number of parameters of exponential or super-exponential level is inevitably required.

2. Universal classifier

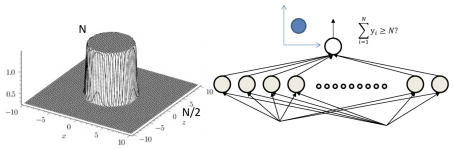
Intuitively, a multi-layer perceptron can construct a classification hyperplane combination of any shape by combining multiple decision boundaries. So, is it really only a multi-layer perceptron that implements a universal classifier? Can I use only one layer?



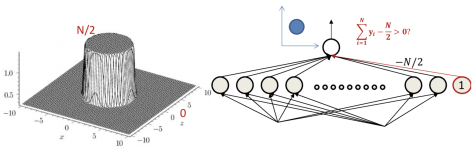
- A layer of perceptron network: an infinite number of neurons, depending on accuracy



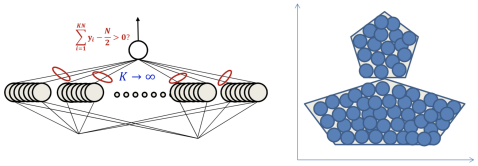
Notes: Each neuron corresponds to a decision line, and the value of the central decision surface is the largest, as shown by the black box in the three-dimensional space.



Notes: It is always possible to make the center close to the cylinder by continuously increasing the decision line. The center value is N and quickly drops to N/2. Note that the cylinder itself can be located anywhere, depending on the decision line position of the combination.



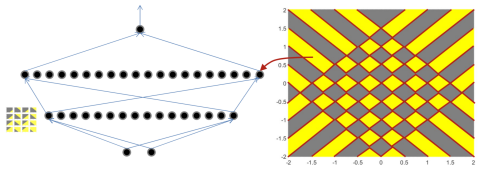
Notes: Change the center of the cylinder by adding a neuron 1.



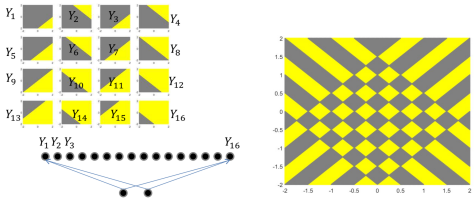
Notes:Infinitely filled, fits the decision surface of any shape. Accuracy can be specified

Conclusion 1.1: The perceptron network of a single layer of infinitely many neurons can fit any decision surface, but a deep network requires fewer neurons to achieve its goal.

- Multilayer perceptron network, choose the optimal depth: the infinity of single-layer network neurons, requires the use of multi-layer perceptron to solve the problem



Notes: Karnaugh map: two hidden layer networks: the first layer represents 16 decision lines (2 original inputs, which can be drawn freely in the plane), and the 2nd layer represents 40 decision surfaces (black squares).



- But this is just $Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{16}$

Notes: XOR law: Several hidden layer networks: Layer 1 represents 16 decision lines, and subsequent layers represent two or two XOR (16 inputs, 15 XOR operations, totaling $15 * 3$ neurons).

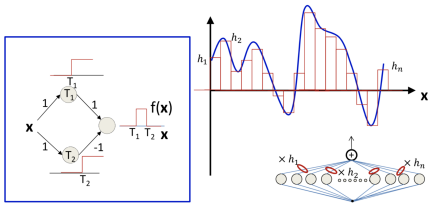
Conclusion 2.1: In the two hidden layer networks, the second layer requires $(N+2)^2/8$ neurons. Although from the first hidden layer, the number of neurons is not exponential. But compared to the input layer (two-dimensional variable), the number of neurons is exponentially increasing.

Therefore, as the input layer dimensions increase, the number of shallow network neurons constructed in a Karnaugh map manner increases exponentially. However, the network constructed by the XOR law has little effect, because the number of neurons in its subsequent layer depends only on the number of decision lines in the first layer.

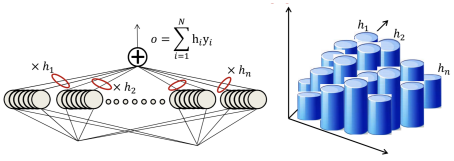
3. Universal continuous functionr

Lecture1 has discussed the univariate function fitting. For multivariate high-dimensional input, the fitting idea of the general classifier can be used to fit the cylinder.

- Single variable input, pulse function



- Multivariate input, cylinder partitioning: similar to the discussion of general classifiers

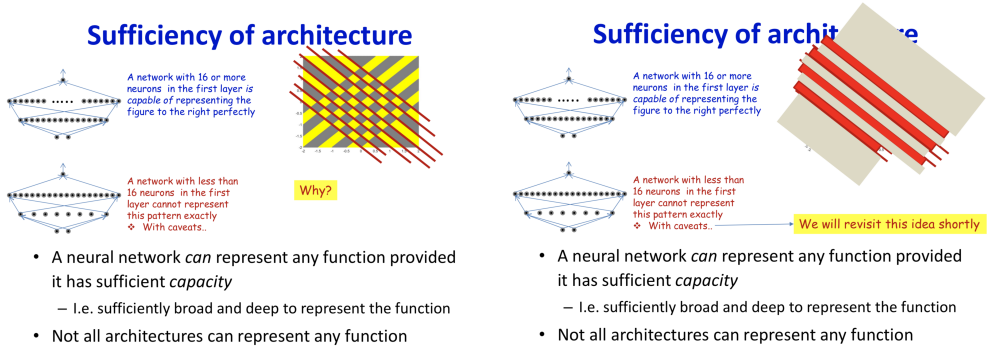


3. Adequacy of neural network architecture: Neural networks adapted to generic functions must have sufficient capacity

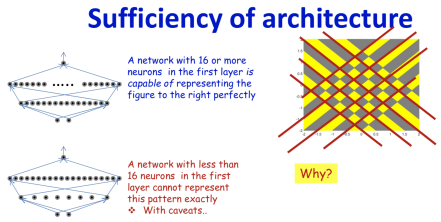
- Use threshold activation function

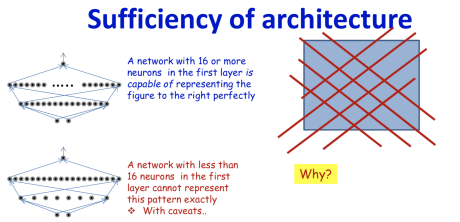
In the discussion of the general classifier, if the first hidden layer replaces 16 neurons with 8 neurons, it will result in insufficient pattern capture. The second hidden layer could not be lost in the first layer anyway.

Similarly, if the second hidden layer replaces 40 neurons with 16 neurons, it will also miss the decision surface of the square or stripe, losing the pattern.



Notes: Lost the decision line from the top right-bottom decision line

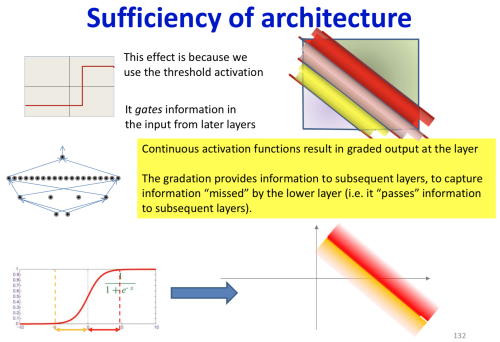




Notes: Lost resolution information for squares and strip decision faces

- Use other activation functions: sigmoid, tanh, relu, etc.

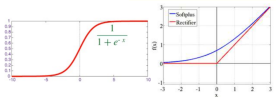
Compared to the non-black or white of the threshold function, other activation functions provide more resolution information in the capture mode. These resolution information will be passed from the lower layer to the upper layer in the form of a gradient, and even compensate for other information lost by the lower layer.



Notes: Threshold activation vs sigmoid

The more sufficient the gradient information, the more information can be delivered. For example, relu is better than a more saturated sigmoid. But it also requires a deeper network layer.

A good activation function requires the ability to continuously capture information (relu ¿ tanh ¿ sigmoid), as well as the ability to capture boundaries (relu ¿ linear activation). Therefore, a valid activation function usually requires a turning point and cannot be a purely linear activation.



Notes: sigmoid vs relu

- Width, depth and activation function trade-offs

In theory, any activation function, even a threshold function, captures all pattern information as long as there is sufficient depth and width.

If threshold activation is used, once the depth and width are not enough, the lost pattern will not be compensated. However, the choice of activation function can alleviate the problem. Choose a good activation function whose gradient will compensate the higher level information lost due to insufficient low-level capacity.

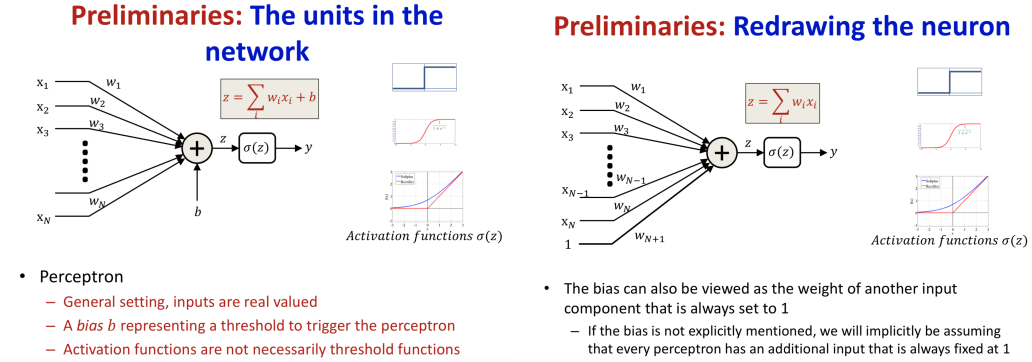
The more you continue to capture the activation function of the mode, the more you will be able to compensate for the missing information. Therefore, this is why people generally use relu to replace sigmoid and threshold functions, because it can capture more information at the same width and depth.

2 Neural Networks: Part 1

2.1 Basic Concepts And Theoretical Foundations

It mainly introduces the basic concepts and theoretical foundations of training neural networks, including perceptron network training rules and empirical risk minimization theory.

1. Preliminaries: The units in the network



- Perceptron
  - General setting, inputs are real valued
  - A *bias* b representing a threshold to trigger the perceptron
  - Activation functions are not necessarily threshold functions

- The bias can also be viewed as the weight of another input component that is always set to 1
  - If the bias is not explicitly mentioned, we will implicitly be assuming that every perceptron has an additional input that is always fixed at 1

2. Training network: known network structure, fitting function g(X)

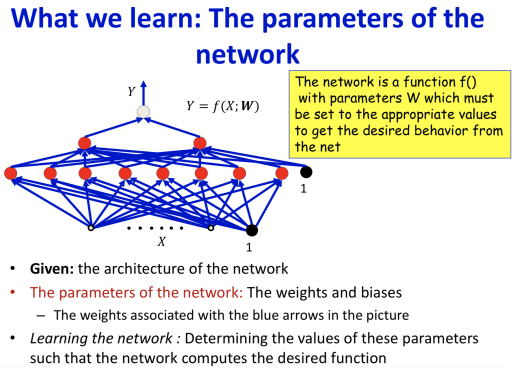
- Definition

Assume the network,

(1) feedforward network (the network is loop-free, the output does not affect the input)

(2) has the ability to fit all functions

Therefore, the training network, that is, under the given network structure, learns the weight parameters of the network.



Notes: The blue arrow represents the parameter, and the training network determines the value of the parameter to completely fit the function.

- Objective

Assume that the objective function $= g(X)$, the function of the multi-layer perceptron network training $= f(X; W)$. Therefore, the ultimate goal of training is to minimize the difference between $g(X)$ and $f(X; W)$. In the two-dimensional plane, the training network parameter W, the area enclosed by the function curves of $g(X)$ and $f(X; W)$ is minimized.

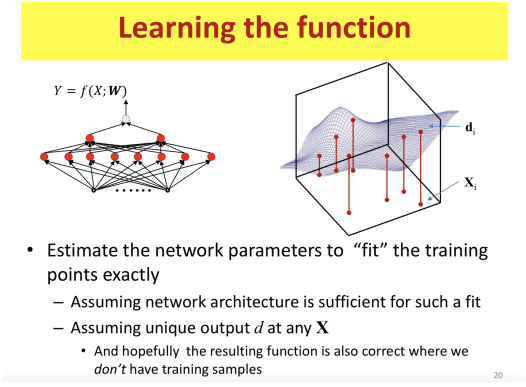
The formula is expressed as follows:

$$\hat{W} = \underset{W}{argmin} \int_{x}^{\infty} div(f(X; W), g(X))dx$$

Notes: $div()$ represents the divergence (gap) between functions; the area enclosed by product dispersion.

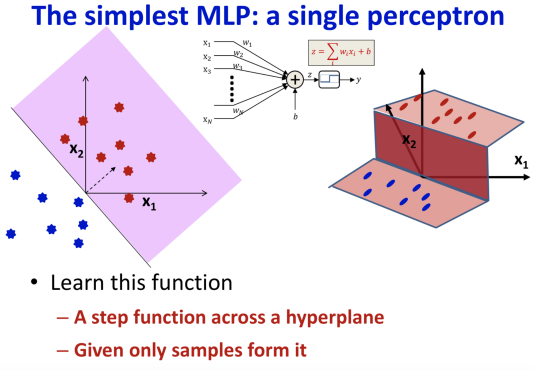
- Problem

But in practice, $g(X)$ is unknown. Mathematically, $g(X)$ is estimated by sampling statistics. Therefore, collecting training samples (input-¿output pairs) is the process of sampling. This is also why the quality of training samples (coverage, uniformity, etc.) will directly affect the accuracy of network training.



Notes: Perceptron network estimates $g(X)$ by accurately fitting the training samples.

3. Start training: taking the two-class problem of a single neuron as an example

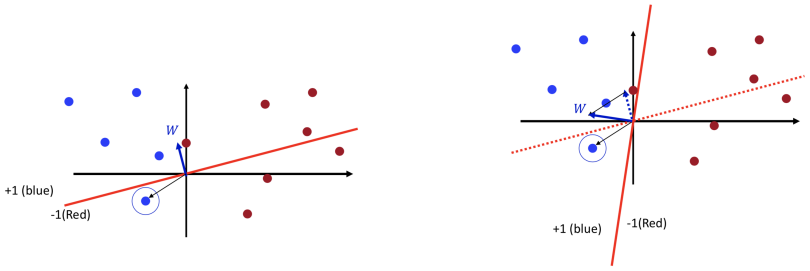


- Algorithm

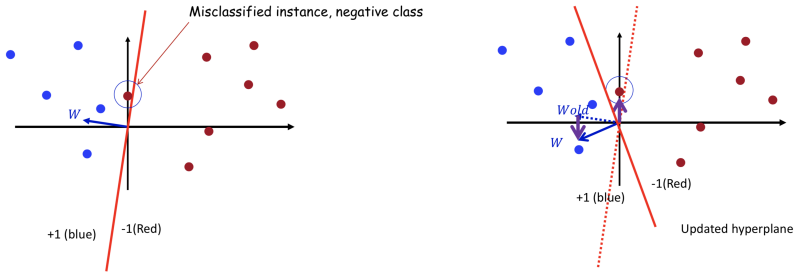
Find a hyperplane $\sum_1^{N+1} w_i \cdot X_i = 0$ so that the right and left sides of the hyperplane are separated from the positive and negative classes respectively. From the mathematical expression, it can be regarded as the weight vector W and The inner product between the feature vectors X. Therefore, the inner product is 0, that is, the weight vector W and all points on the hyperplane (feature vector X) are orthogonal.

- Procedure

If the current hyperplane has a wrong class, adjust the hyperplane. The adjustment is done by changing the normal vector W such that if the positive case is misclassified, W is brought closer to the positive case that is misclassified, ie $W' = W + X_i$; if the negative case is misclassified, W is further away from being A negative example of misclassification, ie $W' = W - X_i$.



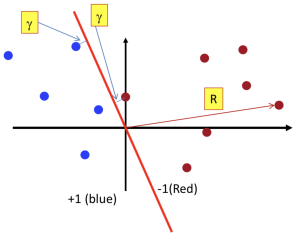
Notes: The positive example is misclassified, so that W is close to the positive case of being misclassified.



Notes: The negative case is misclassified, keeping W away from the negative case of being misclassified.

- Theory

What should I do? The theory proves that if the training sample itself is linearly separable, the algorithm must converge, and the number of bad cases does not exceed $(\frac{R}{\gamma})^2$.

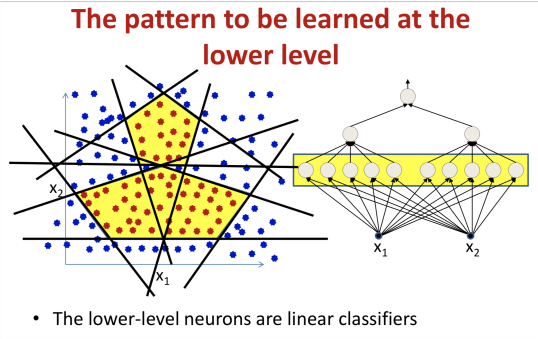


Notes: R: the length of the farthest sample point. r: the closest distance of the correct sample to the classifier.

4. Advanced training: taking a more complex perceptron network as an example

- Premise

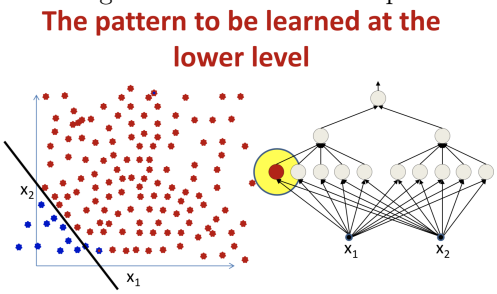
Given training data, perceptron training rules and adequate network structure, how to train a complex network structure, how difficult is the training process? (Training: Get all the parameters in the network)



Notes: Positive class: red dot, negative class: blue dot (2 variables, 10 decision lines, 2 pentagons for summation).

- Thought

Each decision line is a two-class linear classifier of a single perceptron, but needs to change the label of some samples.



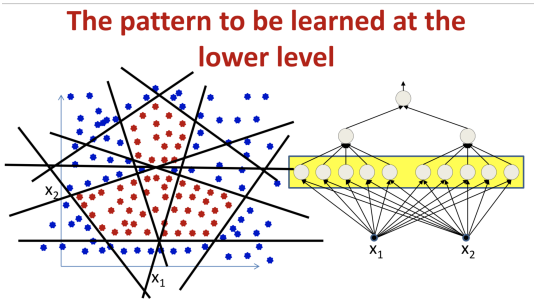
Notes: Turn the blue sample on the right side of the decision line into red and start training.

- Question

How do I change the label? This also requires training!

In theory, for each decision line, you need to try each method of changing the blue point to red, get different decision faces, and finally choose a correct one. Therefore, in the process of training, it is also necessary to train at the same time, how to modify the rules of the label color for each decision line.

Computationally, this is an exponential search operation, NP problem, it is impossible to obtain the optimal solution on the calculation. But there are 2 greedy algorithms that can try to get suboptimal solutions, namely Adaline and Madaline.

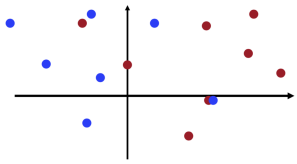


Notes: Each neuron must be trained to output all samples, including the output of the modified sample.

5. Another way of thinking: Why is it so difficult to train the network according to the perceptron rules above?

- Reason

For the threshold function, on any x, non-zero or 1 y, the small change of the weight w will not be immediately fed back to the network, and only if w has changed enough, we know w Whether the change really optimizes the fitting result. That is, the threshold function is a step function, which is the problem!

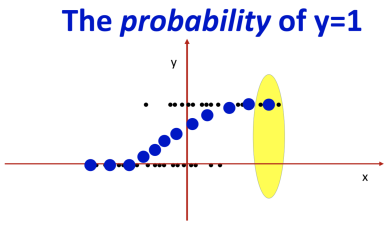


Notes: Step function, unable to act on linearly inseparable training samples.

- Optimization

Change the activation function so that the neurons are guided everywhere. That is, any small change in weight w is immediately fed back to the output in a smooth manner.

For linearly inseparable sample points, taking a one-dimensional variable as an example, a suitable threshold function cannot be found to separate the sample points of the lower graph y=0 and y=1. But P(Y=1—X) has a natural probability explanation for this.



Notes: Each blue point represents the probability value of y=1 on the corresponding x, ie P(Y=1—X).

For multivariate variables, construct a weighted mean Z between the multivariate variables, and find that P(Y=1—Z) satisfies the probability change of the above graph, which is essentially the expression of logistic regression, the famous sigmoid function.

The chain rule of the derivation is also applicable to the sigmoid function, which makes the whole perceptron network can guide all w of any layer, that is, any w change, the perceptron network output will change.

6. Minimize the empirical error: When the objective function cannot be perfectly fitted, make the network as close as possible to it

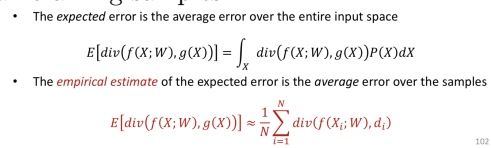
- Principle

When the range of the variable x is limited, it is meaningful to discuss the minimum expected error. When x occurs more frequently in certain intervals, more attention needs to be given to the region. Therefore, more accurately, the training network minimizes the expected error weighted for the probability of occurrence of x.

$$\hat{W} = \underset{W}{argmin} \int_{x}^{\infty} div(f(X;W), g(X))P(X)dx = \underset{W}{argmin} E[div(f(X;W), g(X))]$$

- Concept

The expected error (left of red) is the average error for the entire input space, and the empirical error of the expected error (right of red) is the average error for all training samples.



Notes: The empirical error (red to the right) is what is faced in actual training: for training samples.

- Objective

Find a set of ws to minimize empirical error and complete the final training of the entire perceptron network (significantly different from the threshold function-based perceptron training rules).

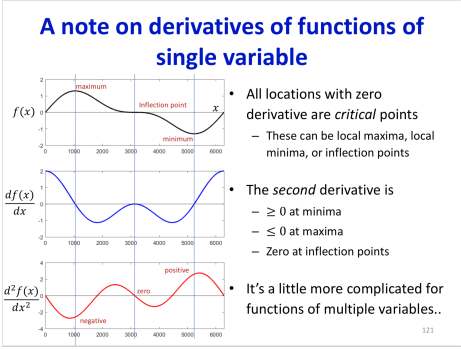
2.2 Gradient Descent Theory and Objective Function

It mainly introduces the definition of gradient descent theory and neural network objective function as a preliminary knowledge of backward propagation.

1. Gradient descent

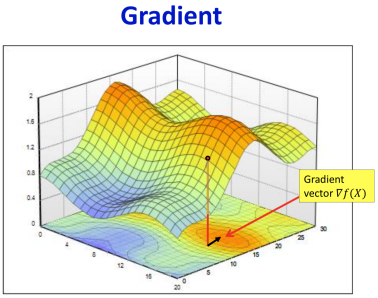
- Extreme Value

Usually at the stagnation point, that is, the derivative is 0. At the same time, if and only if the second derivative of the stagnation point is > 0, it is a minimum value; when the second derivative is < 0, it is a maximum value.



- Gradient

If a gradient is defined, it is a row vector consisting of each partial derivative, indicating to what extent $\Delta\bar{X}$ affects $\Delta\bar{Y}$, with $\Delta\bar{Y} = Grade\Delta\Delta\bar{X} = |Gradient|\Delta|\Delta\bar{X}|\Delta cos\theta$ if and only if $\theta = 0$, ie gradient $\Delta\bar{X}$ When the direction is the same, $\Delta\bar{Y}$ is the largest and Y is the fastest. Similarly, if $\theta = 180$, the direction is reversed, Y decreases the fastest.



Notes: The gradient direction is perpendicular to the contour direction.

- Optimization

Although the extreme points of the function have analytical solutions, because of the complexity of the calculation, iterative methods are usually used to achieve the goal. The core of the iterative method is the gradient descent,

so that X always changes along the opposite direction of the gradient (partial guide vector) until Y is no longer reduced.

So, the question is, given a training set, how to use the empirical error minimization theory, define the objective function f(X), and use the gradient minimization to train the neural network model?

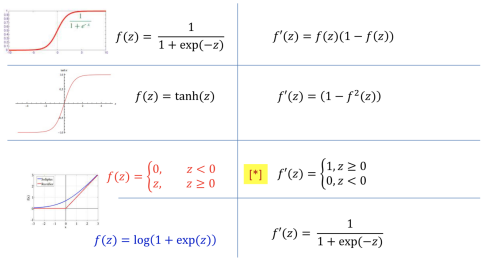
2. Objective function

- The definition of the premise

Because the gradient is to be used, the objective function must be continuous, and each parameter is measurable everywhere. Therefore, the neural network must use the activation function and redefine the error function instead of the number of misclassifications as the objective function.

- Activation function

In theory, any differentiable function can be used as an activation function. But usually, there are four activation functions as follows:



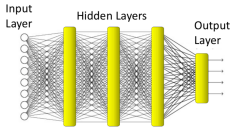
Notes: Sigmoid, Tanh, ReLU, Softplus

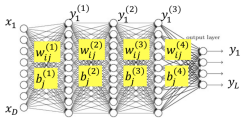
- Typical structure and parameter definitions

Input layer: neural network layer 0, input is output, no neuron, $y_i^0 = x_i$.

Output layer: the Nth layer of the neural network, $y_i = y_i^N$. Usually a single real or real vector.

Hidden layer: the middle layer of the neural network, containing w_{ij} and $b_j 2$ parameters. Where w_{ij}^k = the ith neuron of the $k-1$th layer points to the weight of the jth neuron of the kth layer; b_j^k = the jth neuron of the kth layer Offset.





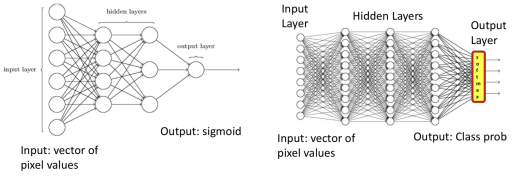
- The input/output layer represents

Input layer: usually a real value vector such as a pixel value, a speech feature or an embedded representation of text.

Output layer:

when the output is real (regression), no special treatment is required.

when the output is a binary representation (classification), first use 0/1 (two classification) or one-hot (multi-class) to indicate yes/no or true label, use sigmoid=P(Y =1—X) or Softmax=$P(Y_i = 1|X)$ represents the output probability of the neural network.



- Objective function definition: (Y = neural network output, d = real label)

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

What is the divergence div()?

Notes: The core of the objective function is to define Div(Y,d).

Real value output (regression): Euclidean distance

$$Div(Y, d) = \frac{1}{2}||Y - d||^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

Two classification problem: cross entropy

$$Div(Y, d) = -dlogY - (1 - d)log(1 - Y)$$

Multi-classification problem: cross entropy

$$Div(Y, d) = -\sum_i d_i logy_i$$

3. Training algorithm

- Initialize all weights $\{w_{ij}^{(k)}\}$
- Do:
  - For all i, j, k, initialize $\frac{dErr}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1:T$
    - For every layer k for all i, j:
      - Compute $\frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}$
      - $\frac{dErr}{dw_{i,j}^{(k)}} += \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}$
  - For every layer k for all i, j:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T}\frac{dErr}{dw_{i,j}^{(k)}}$$

- Until Err has converged

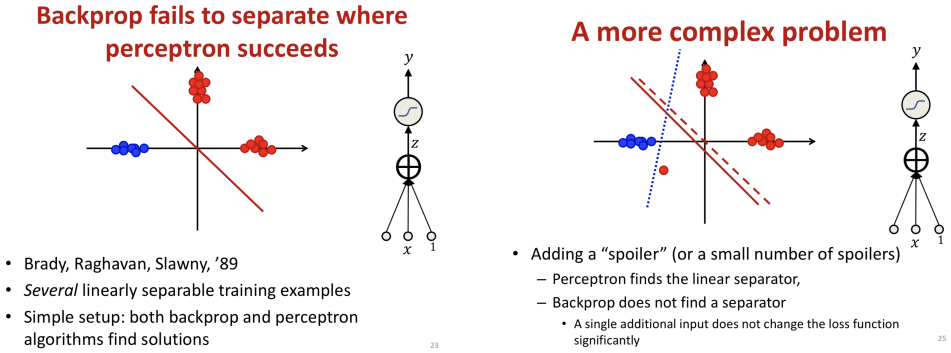
3 Neural Networks: Part 2

3.1 Backward Propagation Theory And Convergence

It mainly introduces the two parts of the neural network's "backward propagation theory" and "convergence".

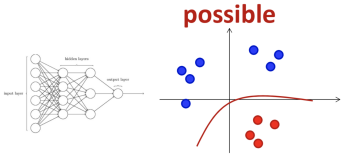
1. Backward propagation, is it effective? That is, can it help the neural network find the global best?

In the classification problem, because of the discrete output, the classification error function is not a microscopic function in nature. Therefore, in practice, only one softmax layer can be added to the neural network of the classification problem, and then a new one can be added. A micro-objective function (for example, crossover), as a "proxy function", evaluates the classification problem. However, please note that minimizing this "proxy function" may not be the optimal solution to the classification problem. As shown in the figure below, for a perceptron, in the left picture, the backward propagation can find the global best, but in the right picture, an abnormal deviation point is added, and the backward propagation does not track each point. Therefore, only a small reaction to the new point (red) is obtained, and the global optimal solution (blue) cannot be obtained. This is a feature of backward propagation because its "low variance" may lead to better generalization effects.



Even in the data with a few abnormal points, the neural network constructed by the multi-layer perceptron can not find the global optimal solution. This is mainly because the answer to the optimal separation is not the optimal solution that the "agent function" can achieve. There are many authoritative research papers that discuss the neural network error surfaces with a large number of saddle points.

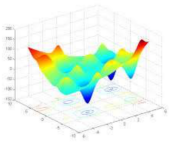
Backprop fails to separate even when possible



- This is not restricted to single perceptrons
- In an MLP the lower layers "learn a representation" that enables linear separation by higher layers
  - More on this later
- Adding a few "spoilers" will not change their behavior

The Error Surface

- The example (and statements) earlier assumed the loss objective had a single global optimum that could be found
  - Statement about variance is assuming global optimum



- What about local optima

2. Backward propagation, can it converge, and how fast does it converge?

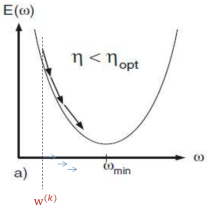
First, we simplify the problem by using the gradient descent method to solve a univariate quadratic objective function with a global minimum, the starting position of the $w^{(k)}$ iteration:

Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

Gradient descent with fixed step size η to estimate *scalar* parameter w



- Gradient descent to find the optimum of a quadratic, starting from $w^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

How to choose the learning rate of gradient drop η ? The best solution, I hope to achieve the best in one step. Because the objective function is so simple, the w_{min} at the minimum can be obtained directly by Newton's method, so the optimal learning rate η_{opt} is also in the limelight.

- Minimizing w.r.t w, we get (Newton's method)

$$w_{min} = w^{(k)} - E''(w^{(k)})^{-1} E'(w^{(k)})$$

- Note:

$$\frac{dE(w^{(k)})}{dw} = E'(w^{(k)})$$

- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size

$$\eta_{opt} = E''(w^{(k)})^{-1} = a^{-1}$$

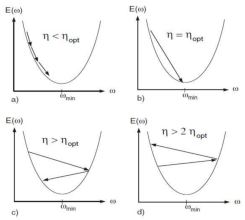
44

When η is not the optimal learning rate, the iteration of the gradient descent is shown in the figure below.

Note the choice of η when converging, oscillating, and diverging.

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w

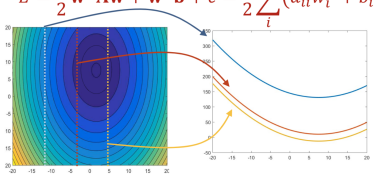


- For $\eta < \eta_{opt}$ the algorithm will converge monotonically
- For $2\eta_{opt} > \eta > \eta_{opt}$ we have oscillating convergence
- For $\eta > 2\eta_{opt}$ we get divergence

Next, we discuss a more general case, using the gradient descent method to solve a multi-objective quadratic objective function with global minimum: the function value E is a scalar with $E^T = E$, so A must be a symmetric positive definite matrix. All eigenvalues are also positive numbers. Furthermore, assuming that A is a diagonal matrix consisting of eigenvalues, the objective function can be simplified such that the arbitrary variables w_i are decoupled from each other and parallel to the function space axis. At the same time, under the premise of ensuring that other variables are unchanged, arbitrarily select two variables (w_1, w_2) and function value E to form an elliptical contour, and select one of the variables (w_1, w_2) and the function value respectively. E, projected to the plane. Therefore, the function expression and optimal learning rate η_{opt} in the two cases, as shown in the figure, at this time, $\eta_{opt} < 2 * min(\eta_{i,opt})$ is the optimal choice.

Multivariate Quadratic with Diagonal A

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i (a_{ii}w_i^2 + b_i w_i) + c$$



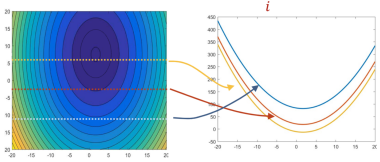
- Equal-value contours will be parallel to the axis
  - All "slices" parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$

49

Multivariate Quadratic with Diagonal A

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i (a_{ii}w_i^2 + b_i w_i) + c$$

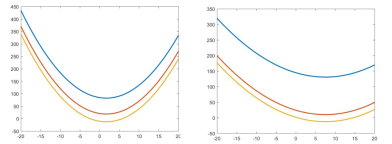


- Equal-value contours will be parallel to the axis
  - All "slices" parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$

50

"Descents" are uncoupled



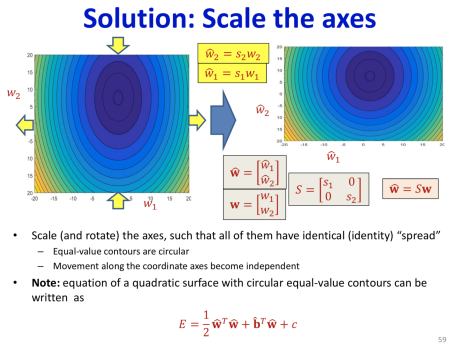
$$E = \frac{1}{2}a_{11}w_1^2 + b_1 w_1 + c + C(\neg w_1) \qquad E = \frac{1}{2}a_{22}w_2^2 + b_2 w_2 + c + C(\neg w_2)$$

$$\eta_{1,opt} = a_{11}^{-1} \qquad \eta_{2,opt} = a_{22}^{-1}$$

- The optimum of each coordinate is not affected by the other coordinates
  - I.e. we could optimize each coordinate independently
- **Note: Optimal learning rate is different for the different coordinates**

However, in a function, $\eta_{i,opt}$ in different directions is varied and the gap is

very large, so the learning rate selected according to the above principle usually converges very slowly. Therefore, scaling the target function axis, normalizing the objective function, making $\eta_{i,opt}$ equal in all directions, will make it easier to find an optimal learning rate.



Eventually, along a diagonal matrix, the axis is scaled and the contours are changed from ellipse to circular. According to the previous assumption, the matrix A itself is a diagonal matrix composed of one eigenvalue, and therefore, the new transformation matrix S will be the square root of the matrix A. At this time, the comprehensive optimal learning rate is equal to one.

- We have

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

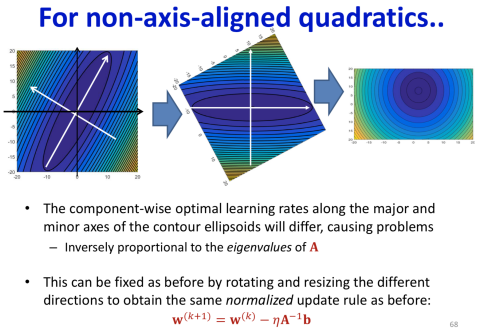
$$\hat{\mathbf{w}} = S\mathbf{w}$$

$$E = \frac{1}{2}\hat{\mathbf{w}}^T\hat{\mathbf{w}} + \hat{\mathbf{b}}^T\hat{\mathbf{w}} + c$$

- Solving for S we get

$$\hat{\mathbf{w}} = \mathbf{A}^{0.5}\mathbf{w}, \qquad \hat{\mathbf{b}} = \mathbf{A}^{-0.5}\mathbf{b}$$

For the more general matrix A, the variable w_i is not completely decoupled. That is, if the matrix A is not a diagonal matrix, the contours will no longer be parallel to the coordinate axes. But after the axis is first "rotated" and then "zoomed", the discussion is still similar.



But a new problem is that when the second derivative is obtained, a Hessian

matrix is added. Therefore, the gradient descent also needs to calculate the inverse matrix of the Hessian matrix.

- Taylor expansion

$$E(\mathbf{w}) \approx E(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^T H_E(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \cdots$$

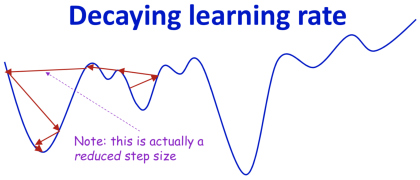
- Note that this has the form $\frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$
- Using the same logic as before, we get the normalized update rule

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E(\mathbf{w}^{(k)})^{-1} \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})^T$$

Note that this is still not the whole problem. For the ordinary symmetric matrix A, it is very likely to be semi-definite, that is, the eigenvalue may have a negative value and cannot solve its square root. At the same time, even if the matrix A is positive, this is almost impossible to achieve for a neural network with a large number of parameters. Therefore, it is best not to use the Hessian matrix directly. Some optimization algorithms achieve the same goal by approximating the Hessian matrix, such as the BFGS and Levenberg algorithms, but they have recently appeared less because we have better algorithms. It will be introduced in the next lesson.

Finally, the function that the neural network needs to fit is not a convex function. Therefore, a relatively large learning rate may be exactly what it needs, because a large learning rate helps to jump out of the local optimal solution. However, please note that after reaching the search interval of the global optimal solution, the large learning rate will not find the optimal solution. Therefore, in order to balance the relationship between the two, the more common practice is to gradually attenuate the learning rate in the process of training, and there are many ways to achieve it, as shown in the following figure:

Decaying learning rate



- Start with a large learning rate
  - Greater than 2 (assuming Hessian normalization)
  - Gradually reduce it with iterations

Decaying learning rate

- Typical decay schedules
  - Linear decay: $\eta_k = \frac{\eta_0}{k+1}$
  - Quadratic decay: $\eta_k = \frac{\eta_0}{(k+1)^2}$
  - Exponential decay: $\eta_k = \eta_0 e^{-\beta k}$, where $\beta > 0$

- A common approach (for nnets):
  1. Train with a fixed learning rate η until loss (or performance on a held-out data set) stagnates
  2. $\eta \leftarrow \alpha\eta$, where $\alpha < 1$ (typically 0.1)
  3. Return to step 1 and continue training from where we left off

4 Neural Network Training

It mainly introduces the most important four parts of neural network training: three methods of gradient descent (Batch, Stochastic, mini-Batch), and several methods of optimization algorithm (Momentum, Nestorov, Adagrad, ADAM, RMS Prop, AdaDelta), Generalization strategy (L2 regularization, Dropout) and neural network training techniques (difference evaluation function, Batch Normalization, gradient cutting, data expansion).

4.1 Concept combing

- In traditional data mining tasks, use traditional machine learning

Assuming that the form of the function/model H(x) is given, the purpose of learning is to find several features as the independent variable x_i of H(x), and select different parameters under the criterion of minimizing divergence. The estimation method, while defining the loss function J(x), iteratively updates the parameter w_i of H(x) such that the divergence of J(x) on the training set is minimal. The method of iteratively updating parameters usually uses gradient descent or Newton's method.

Taking logistic regression as an example, it uses the linear weighted + sigmoid function as the given model H(x), and explores the different features x_i in the training data by means of feature engineering, based on the premise of minimizing the evaluation criteria of divergence. The maximum likelihood estimate, as J(x), uses gradient descent to iteratively update the model parameter w_i that is most likely to produce this batch of training data.

Therefore, the most important place in the traditional machine learning method is 3 points.

(1) Find the adaptive hypothesis function H(x) according to the specific scene;

(2) Feature engineering, find the adaptation function in the training data. The independent variable x_i;

(3) selects the implementation of the minimum divergence evaluation criterion, that is, the loss function J(x).

- In the face of very large training data sets and extremely complex application scenarios, use deep learning

Combining specific activation functions, constructing different neural networks (CNN, RNN), and using neural networks to fit the conclusions of arbitrary functions, expecting it to automatically adapt to the hypothesis function H(x) to train the data itself (End-to-end or feature engineering (other scenarios), etc., define the x_i of the model input. Under the criterion of minimizing the divergence, use the method of minimizing the empirical error to define the L2 divergence or KL divergence as the loss function J. (x), using the gradient descent, iteratively updating the network parameter w_i such that the L2 divergence or KL divergence is minimized.

In the gradient of the neural network, because the network level is too much or too deep, the algorithm framework of backward propagation (ie, a dynamic programming idea) is born to speed up the training; because of the huge training data, the memory The effects of limitation and convergence have resulted in gradient descent methods such as SGD and mini-Batch. In each iterative update of parameters, different numbers of training data are used to achieve time, space and effect trade-offs; The non-convex characteristics, the probability that the neural network converges to local optimum or simply diverges, the optimal methods such as Momentum and ADAM are born, and the optimal convergence effect is obtained as much as possible.

4.2 Three ways to reduce the gradient

1. Batch gradient descent

Usually the loss function minimizes the divergence of the predicted and actual values on the entire training sample, so every iteration update of the normal gradient drop requires the use of the entire training set data.

Taking the linear regression of L2 divergence as an example:

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})^2$$

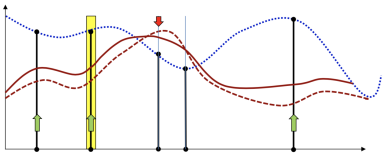
$$\min_\theta J_\theta$$

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m}\left(y^{(i)}-h_\theta(x^{(i)})\right)x_j^{(i)} \quad \text{(for every } j\text{).}$$

}

Take the visual understanding of the image as an example:



Standard Batch training steps:

- Given (X_1, d_1), (X_2, d_2),..., (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$
- Do:
  - For all $t = 1{:}T$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
      - Update

        $W_k = W_k - \eta \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
- Until Err has converged

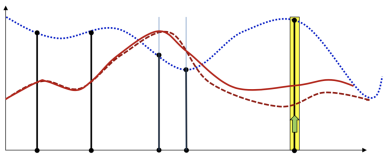
2. Stochastic gradient descent

The batch gradient is degraded, and in all iterations, all training data is traversed, so this method is not very useful when the amount of data is too large or the memory is limited and cannot be loaded to calculate all the data. Therefore, instead of m=1, each iteration, only one training sample is randomly selected to update the parameters. Although each iteration does not advance toward the overall optimal direction, the end result is always close to the overall optimal. Usually, a round of training data is traversed in its entirety, called an epoch.

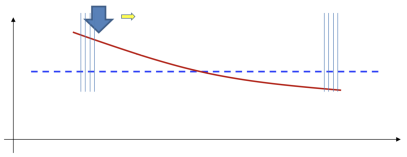
Take the linear regression of L2 divergence as an example:

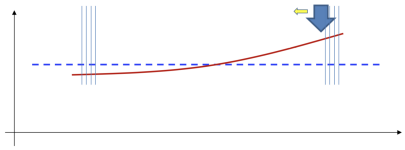
$$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)})-y^{(i)})\cdot x_j^{(i)}$$

Take the visual understanding of the image as an example: (Obviously, the ordinary method of batch is closer to the global optimal solution)



Random selection: If each iteration of the updated loop uses the same sample order, SGD is prone to cyckic behavior.





Learning rate strategy: Even the global optimal fitting direction may be penalized on a single or small sample. Therefore, SGD needs to gradually reduce the "learning rate" during the iterative process.

Considering the following, the training steps of SGD:

- Given (X_1, d_1), (X_2, d_2),..., (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),..., (X_T, d_T)
  - For all $t = 1:T$
    - $j = j + 1$
    - For every layer k:
      - Compute $\nabla_{W_k} Div(Y_t, d_t)$
      - Update

        $W_k = W_k - \eta_j \nabla_{W_k} Div(Y_t, d_t)$
- Until Err has converged

Randomize input order

Learning rate reduces with j

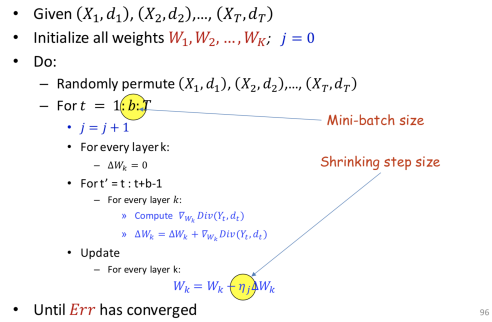
3. mini-Batch gradient descent

The stochastic gradient is reduced, and only one sample is used for each iteration. It is not the optimal direction of the loss function itself, and it is easy to fall into the local optimum. As a compromise, let m=b, each iteration, randomly select b samples to calculate the gradient and update the parameters.

Batch-size: m=b In the mini-Batch training scheme, b is an optimizable hyperparameter.

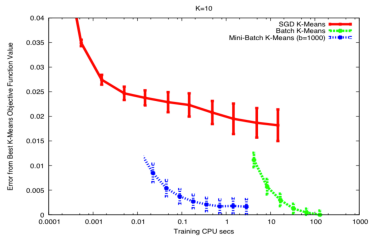
Learning rate strategy: a simple solution, using a fixed learning rate until the error oscillates, and then using a fixed ratio to reduce the learning rate. Advanced program, adaptive learning rate, as part of the training estimate.

Comprehensive consideration, mini-Batch training steps:

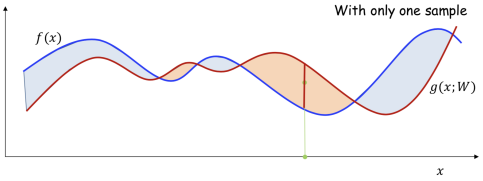


4. Intuitive comparison in K-Means

Mini-Batch as a compromise between standard Batch and SGD, while taking into account the advantages of both sides, can alleviate the potential shortcomings of the two. Therefore, in practical applications, mini-Batch is usually used as a training scheme.



Vertical vertical line of the error curve: indicates the variance of the training error. Intuitively, using a sample update parameter alone will inevitably increase the variance of the error sequence during training.



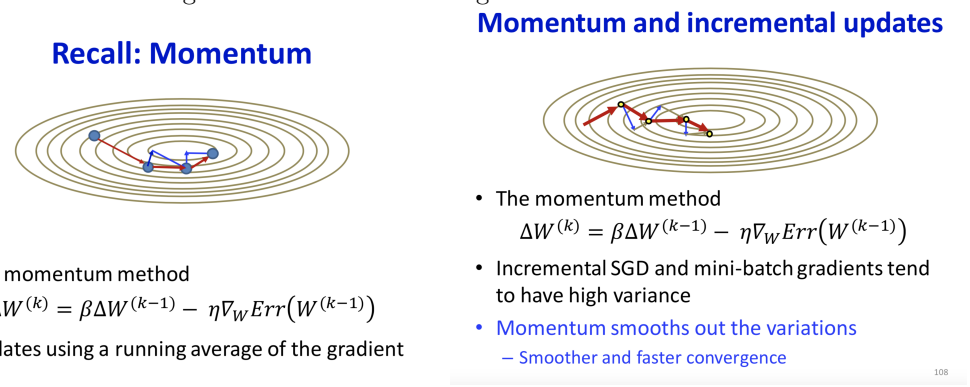
4.3 Several methods of optimizing the algorithm:

First of all, it is clear that the non-convex characteristics of the neural network to be fitted to the large probability of the function are converged to local optimum or the probability of divergence is large, so the optimization methods such as Momentum and Nestorov's acceleration gradient are born, and the training error is smoothed. When the variance is used, the convergence speed can be accelerated, and the optimal convergence effect can be obtained as much as possible.

1. Momentum

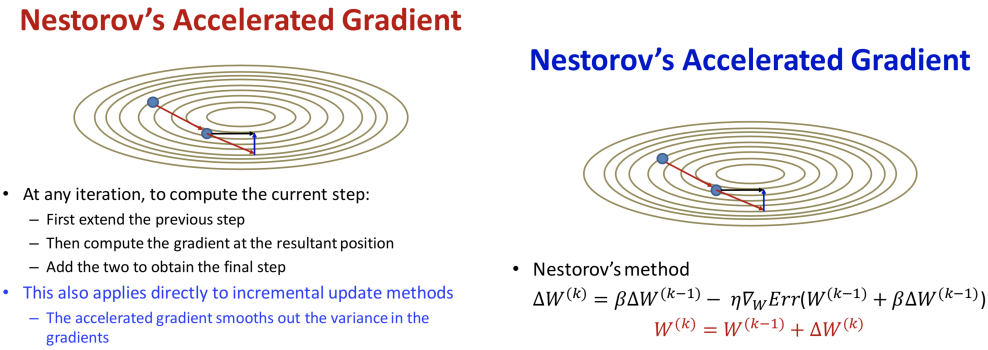
Momentum believes that for each iteration, the gradient descent no longer uses the gradient of the point itself, but uses the upper point gradient and the weight of the point gradient. When the upper point gradient and the current

gradient are in the same direction, the iteration is accelerated, otherwise, the iteration is slowed down. The final iterative gradient is physically similar to the resultant force of the last gradient and the current gradient.



2. Nestorov's Accelerated Gradient

Nestorov's acceleration gradient is the optimization scheme of Momentum. Before each iteration, first use the gradient of the previous point to descend at this point. The position to fall is used as the transit temporary point, and then the gradient of the transit temporary point is calculated. Finally, the gradient of the point is equal to the previous point. The weighted sum of the gradient and the transition temporary point gradient.



3. Other Optimization Algorithms

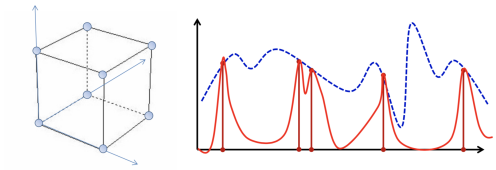
In recent years, some new optimization algorithms have been proposed, their core purpose is to enhance the long-term trend of the gradient to smooth the variance of training errors in the iterative update, mini-Batch gradient, such as RMS Prop, ADAM, Adagrad and AdaDelta et al. You are welcome to say that in practice, they have roughly the same effect, but ADAM is used more.

4.4 Generalization strategy

1. over-fitting phenomenon

Consider a scenario: for a neural network, each input is represented by a 100-dimensional binary vector (one-hot), then there may be $2^100 = 10^30$ possible

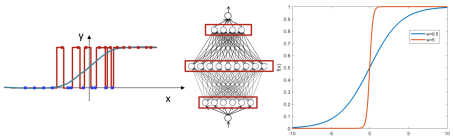
training data, at least 10^30 It is obviously unrealistic for the sample to accurately fit the neural network. Therefore, when training a neural network, additional constraints are needed to "fill" the missing areas within the sample space. That is, regularization.



2. over-fitting reasons

Taking the simple two classification as an example, in the activation function of the neural network, the larger the modulus $|w|$ of the weight, the more the ability to respond to steep changes. And the more powerful the network is in responding to steep variables, the easier it is to overfit. Therefore, limiting the size of the weight $|w|$ forces the network to adapt to a smoother output response. (Blue line, is the correct objective function, activation function and w, red line, is over-fitting)

Smoothness through weight manipulation



- Steep changes that enable overfitted responses are facilitated by perceptrons with large w
- Constraining the weights w to be low will force slower perceptrons and smoother output response

3. Relieve overfitting:

Adding a regularization term of L1 or L2 to the loss function, penalizing the small error produced by the larger $|w|$, so that the weight $|w|$ is minimized while minimizing the loss function.

Regularizing the weights

$$L(W_1, W_2, \ldots, W_K) = \frac{1}{T}\sum_t Div(Y_t, d_t) + \frac{1}{2}\lambda \sum_k \|W_k\|_2^2$$

- Batch mode:

$$\Delta W_k = \frac{1}{T}\sum_t \nabla_{W_k} Div(Y_t, d_t)^T + \lambda W_k$$

- SGD:

$$\Delta W_k = \nabla_{W_k} Div(Y_t, d_t)^T + \lambda W_k$$

- Minibatch:

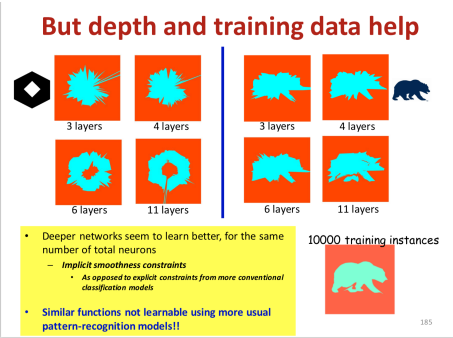
$$\Delta W_k = \frac{1}{b}\sum_{\tau=t}^{t+b-1} \nabla_{W_k} Div(Y_\tau, d_\tau)^T + \lambda W_k$$

- Update rule:

$$W_k \leftarrow W_k - \eta \Delta W_k$$

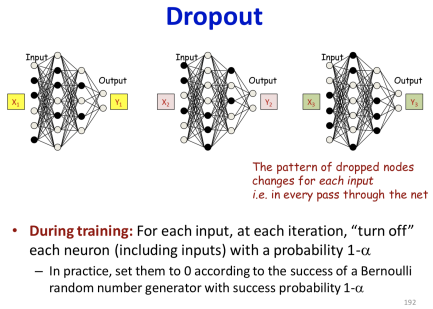
Given the total number of neurons, the deeper the network, the more training data, the stronger the generalization ability, the more responsive to smooth output. Because each layer of processing is based on the previous layer has

been processed, the more generalized smooth boundaries.



Dropout: The most influential over-fitting solution in deep learning.

Statistical Interpretation: Similar to Bagging, the strategy of using multiple machine learning models to output the results of voting. A network of N neurons, which contains 2^N sub-networks in total, samples different network structures through Dropout, and finally learns an average network that integrates all sampling structures.



Implementation strategy: For each iteration of each input, use the Bernoulli sampling method to turn off any neurons (including the input layer) of each layer with a probability of $1-\alpha$. At the same time, during the training process, forward propagation and backward propagation are only performed on the currently visible network, and the closed neurons have an output and a gradient of zero. Therefore, the network results for effective prediction for different inputs are also different.

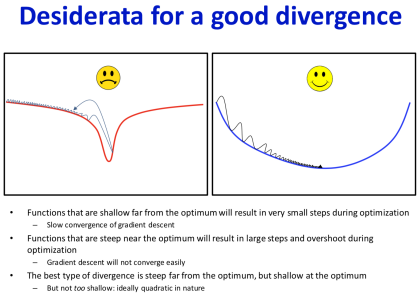
In practice, Dropout can be implemented in a variety of ways during the training and testing phases, and the details are handled differently.

4.5 Training skills

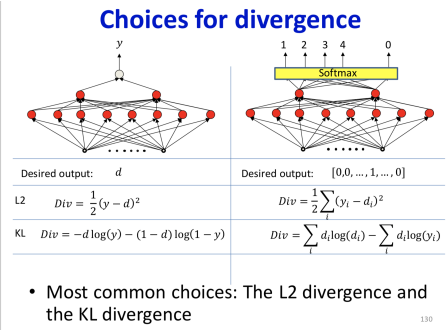
4.5.1 Divergence Evaluation Function

The divergence evaluation function in deep learning, that is, the loss function, is preferably a relatively smooth convex function as with traditional machine learning. The convex function is well understood, the only global minimum.

But how to understand relatively smooth? A smooth convex function, each time the gradient drops relatively slowly, and is always updated toward the optimal solution. However, the steep convex function has a large gradient and easily breaks down the optimal solution.



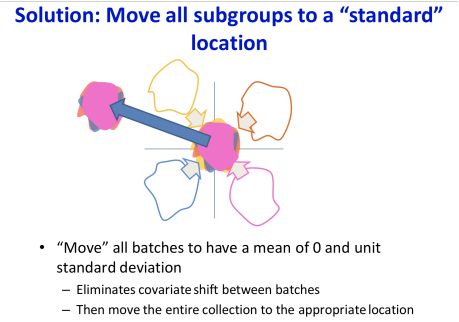
Therefore, the ideal divergence evaluation function is steep when it is far from the optimal solution, and can be quickly degraded and updated; near the optimal solution, it needs to be very smooth and ensure convergence. In practical applications, the most commonly used divergence evaluation functions are L2 divergence and KL divergence. The former applies to the regression problem of numerical prediction, and the latter applies to the classification problem after softmax.

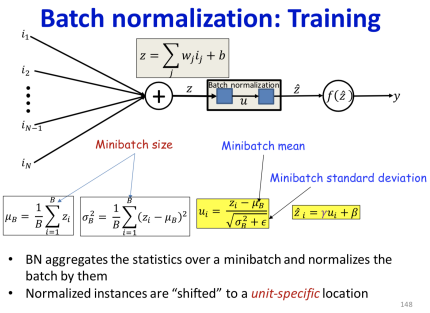


4.5.2 Batch Normalization

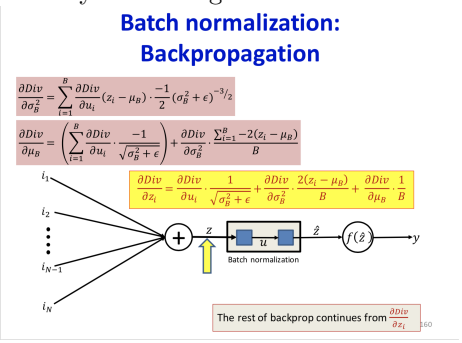
When the neural network is trained using mini-Batch, it is assumed that each of the selected b samples has the same distribution. But this is not the case. The distribution of b samples in different batches is usually different. Therefore, this may lead to "covariate shift" in the propagation of each layer of the network. After the variance offset is accumulated, the training effect will be greatly impaired.

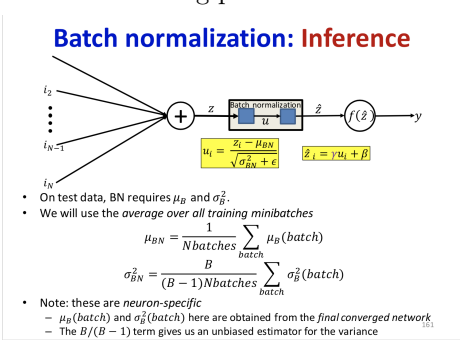
Therefore, in neural network training, it is necessary to eliminate the covariance offset between different batches, and at the same time adapt the batch distribution to the appropriate distribution. The solution is Batch normalization. The implementation step is to first count the mean and variance of each batch, then normalize the weighted z_i (0 mean and unit variance), and finally transfer z_i to the appropriate position.





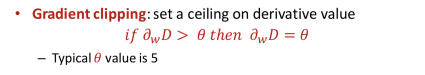
After the normalization of the neural network, the backward propagation during training and the test without the prediction need to be accurately adapted. The former should accurately calculate the new gradient, and the latter can be replaced by the average of the batch statistics in the training phase.



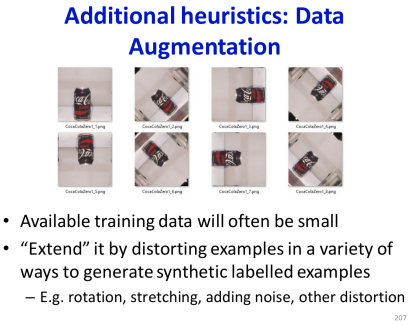


4.5.3 Other Tips

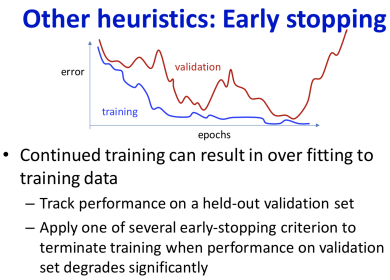
- Parameter initialization: random initialization is a good method.
- Gradient Clipping: When the divergence evaluation function is steep, the gradient is large, which will lead to instability of the training results. Therefore, during training, when the gradient is found to exceed the set threshold, it is immediately truncated.



- Data Augmentation: In reality, the sample size of deep learning is often insufficient. Therefore, you can try to transform the training samples by various methods to increase the sample size. In the CV field, it is often achieved by means of rotation, stretching, and salting (increasing noise).



- Early stopping: Excessive training time may lead to over-fitting. Therefore, in the training process, it is necessary to track the effect of the verification set in real time, and when it is found that the error of the verification set is soaring, the training is ended.



5 Convolutional Nerual Networks (CNN)

5.1 Different layers in CNN

5.1.1 Convolution Layer

Convolution layer: uses filters as parameters, convolve filters with the image by multiplying its values element-wise with the original matrix.

In forward pass, we take many filters and convolve them on the input, each convolution gives a 2D matrix output, then stack them into a 3D volume. Usually every convolution layer is followed by a relu layer.

In backward pass, we calculate the derivations of each variables based on the following parameters.

This is the formula for computing dA.

$$dA \mathrel{+}= \sum_{h=0}^{nH} \sum_{w=0}^{nW} W_c * dZ_{hw}$$

Where W_c is a filter and dZ_{hw} is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down).

This is the formula for computing dW_c with respect to the loss:

$$dW_c \mathrel{+}= \sum_{h=0}^{nH} \sum_{w=0}^{nW} a_{slice} * dZ_{hw}$$

This is the formula for computing db with respect to the cost for a certain filter.

$$db = \sum_h \sum_w dZ_{hw}$$

5.1.2 Pooling Layer

The pooling layer reduces the height and width of input volume/tensor. It helps reduce the computation, as well as helps make feature detectors more irvariant to its position in the input. There are two types of pooling layer: max-pooling and average-pooling.

The pooling layer's implementation is some what similiar to convolution layer, it uses a filter slicing over the input volume. Usually a pooling layer doestn't contain zero-padding and it doesn't contain parameters. It will need two hyper-parameters: filter size and stride. If filter size is 2 and stride is 2, then after pooling the height and width of volume will be half of before.

5.2 Multi-class learning

For multi-class classification, we use Softmax function to output classification.

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

5.2.1 Derivation

If we use cross-entropy loss function, such as following:

$$L(\boldsymbol{W}) = -\frac{1}{N} \sum_{n=1}^{N} [y_n \log \hat{y} + (1 - y_n) \log(1 - \hat{y})]$$

Then the derivation should be:

$$\frac{\partial L}{\partial \boldsymbol{z}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

5.2.2 Notes

When used in back propagation in practical programs, espicially in mini-batch, don't forget to divide the derivation by the batch size.

$$\frac{\partial L}{\partial \boldsymbol{z}} = (\hat{\boldsymbol{y}} - \boldsymbol{y}) / batch - size$$

5.3 Computation Improvment

As it is mentioned before, if we compute convolution and pooling explicitly by single steps, it will be very time-consuming as it will need four nested four-loops to compute a 4D volume/tensor, and this type of computing contains a lot duplicates computation. In pracitical code, using explicit for-loop is usually avoided as it will take a lot time for computation.

Usually in most deep learning frameworks, unrolling convolution layers are used to speeding up the computation process of convolution and pooling layers. The central idea is an unfolding and duplication of the input and rearrangement of the kernel parameters that produces a CPU/GPU friendly ordering.

Using this approach each convolutional layer is converted to a matrix product during forward propagation. A nice byproduct of this is that we can simply write down the back-propagation step as another matrix product.

Simple unfolding of convolution is a well known technique. It is commonly implemented in signal processing and communications applications. For example, Matlab® has two functions, convmtx and convmtx2 (signal processing toolbox) which create "convolution matrices" in order to transform convolution into a matrix multiplication.

6 CNN Case studies

6.1 Classic Networks

6.1.1 LeNet-5

Two Convolution layers followed by avg-pool layers, then two fully-connected layer, use softmax as the output layer. As the network goes deep, the height and width of the volume go down, and the number of filters (or the depth of the volumn) go up.

6.1.2 AlexNet

Same padding is used, $pad = (f-1)/2$. AlexNet is similar to LeNet, but much bigger. And uses Relu as activate function.

6.1.3 VGG-16

All convolution layers are 3*3 filters, stride = 1, same pooling.
All max-pooling layers are 2*2 filters, stride = 2.
VGG simplied neural network architectures. The architecture of VGG-16 is uniform. Use max-pooling with 2*2 filters, stride = 2 makes the height and width go half every time.

6.2 ResNet

6.2.1 Introduction

Use resudual blocks allow to build deep networks. Number of training error goes down with ResNet layer numbers goes up.

6.2.2 Intuition

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$
$$a^{[l+2]} = g(w^{[l+2]} * a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

Use same padding convolve to ensure the demisons of the matrix. Or using pooling layers to adjust the demisons.

6.3 1*1 convolution

Filters are 3D volume, and size is $1 * 1 * N_{c_prev}$.Works as fully connected network with Relu. Network in Network. Useful to shrink the number of channels.

Inception layer uses 1*1 convolutions, other convolutions and pooling layers, and stack up all the outputs. Using 1*1 convolution reduces the computational cost without harming the performance.

6.4 Inception Network

Inception module concatenates convolution and maxpooling layers output.
Inception network puts inception modules together.
Inception module makes network go deeper.

6.5 Pratical Advices

- Use Open-Sourece Implementation
- Transfer Learning
- Data Augmentation

6.5.1 Data Augmentation

- Mirroring
- Random Cropping
- Rotation
- (PCA) Color shifting

6.5.2 Deep Learning for Computer Vision

Two sources of knowledge: Labeled data and hand engineering/network architecture.

Tips for doing well on benchmarks or competetions:

- Train several networks independently and average their outputs.
- Multi-crop at test time and average results.
- Use architectures of networks published in the literature.
- Use open sourece implementations if possible.
- Use pretrained models and fine-tune on your dataset.

7 Detection Algorithm

7.1 Object Location

For an image, the coordinate is set like this: the left up corner is set to $(0, 0)$, and the right bottom corner is set to $(1, 1)$. The center point of an bounding box is then (b_x, b_y), in the coordinate of image. The bounding box height and width is (b_h, b_w).

7.2 Convolutional Implementation of Sliding Windows

Running sliding windows explicitly slide over the image contains a lot dublicate computation. Instead, treating the whole computation process of a window sliding as a convolutional layer, and compute the image convolutional, each part of output volume corresponds to a sliding window output.

7.3 YOLO Algorithm

YOLO Algorithm sets grid on the image, and for each grid cell, it will feed the data into convolutional network, output the predict labels, and then use back propagation to train the convolutional network.

The center of a bounding box (b_x, b_y) is relative position in a grid cell, so b_x, b_y are between 0 and 1. So the output function of b_x, b_y can be sigmoid function The heigth and width of a bounding box (b_h, b_w) are relative length in a grid cell, but b_h, b_w can be larger than 1. The output function of b_h, b_w can be exponential function.

7.4 Intersection Over Union

Intersection Over Union is a way to evaluating object localization. It calculate the intersection part of output and ground truth over the union part of output and ground truth.
Usually the output is considerred as "correct", if $IoU \geq 0.5$.
More generally, IoU is a measure of the overlap between two bounding boxes.

7.5 Non-max Suppression

Non-max Suppression is a way to make sure the algorithm detects the object only once.

Basically, Non-max Suppression will discard all bounding boxes with $P_c \leq 0.6$. Then it will do the following steps iteratively:

1. Pick the bounding with highest P_c, output that as prediction.
2. Suppress those bounding boxes that $IoU \geq 0.5$ with previous bounding box.

7.6 Anchor Boxes

Anchor boxes are pre-difined to avoid object overlapping.
Each object in the training image is assigned to grid cell that contains objects midpoint and anchor box for the grid cell with the highest IoU.

8 Special Applications

8.1 Face Recognition

8.1.1 Face verification

One shot learning is the challenge. Instead of traing softmax CNN, it learns a similarity function, which indicates the degree of difference between images. Face verification is a single step in face recognition.

8.1.2 Nerual Network For Degree Difference - Siamese Network

Instead of softmax, output full-connected output result as encodings of input. Parameters are learning so that if inputs are same people, output should be small, else the output should be large.

8.1.3 Triplet Loss

Looking at three images at a time, anchor, positive, negative images, which means $d(A, P) + \alpha \leq d(A, N)$. The definition is :

$$L(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$$
$$Cost = \sum_{i=1}^{M} L(A^i, P^i, N^i)$$

Choose triplets that are "hard" to train on.

8.1.4 Binary Classification

Use a sigmoid function to output result of element-wise difference encoding of input.

8.2 Neural Style Transfer

9 Recurrent Neural Networks (RNN)

A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition.

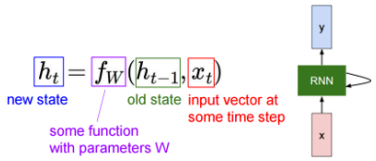


Figure 1: RNN

Notice that the same function and the same set of parameters are used at very time step! We can not have RNNs of enormous length because we have to store in memory the hidden state of each time step to be able to back-propagate. Normally we will have RNNs of max 25 length. To process bigger sequences we will divide the sequence in chunks of 25 and the last hidden state of a chunk is the initial hidden state of the next chunk.

Let's see some wire examples (fig 2):

1. Vanilla Neural Networks
2. Image Captioning (image to sequence of words)
3. Sentiment Classification (sequence of words to sentiment)
4. Machine Translation (seq of words to seq of words)
5. Video classification on frame level

We can stack RNNs together to produce deeper RNN. In figure 4 case we have stack 3 RNNs. It still works the same as before but now we have 3 weights.

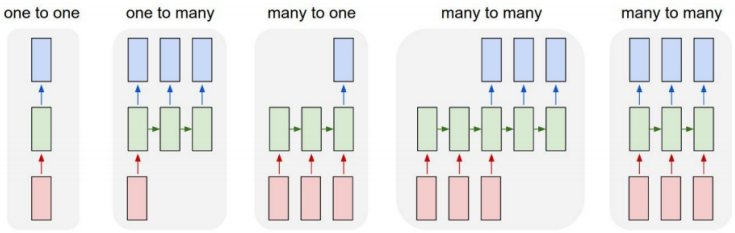


Figure 2: RNN examples

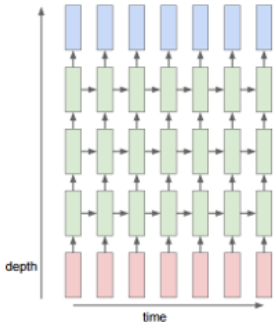


Figure 3: Stacked RNNs

9.1 Practical Example

Lets create a RNN that given a sequence of characters predicts the next one. The output is always the prob of each letter in the vocabulary to be the next character:

- Vocabulary: [h,l,e,o]
- Example training sequence: "hello"

For this example we will use a Vanilla RNN:

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

$$y_t = W_{hy}h_t \tag{2}$$

$$h_0 = 0 \tag{3}$$

A 100 lines of code implementation of a character recognition in python is implemented here: https://gist.github.com/karpathy/d4dee566867f8291f086

9.2 Image Captioning

Example using RNN for image captioning

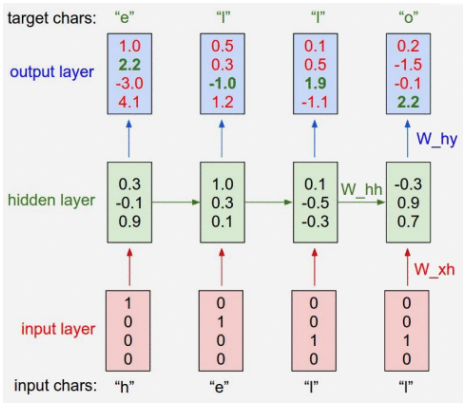


Figure 4: Practical example. You can see this blue boxes being softmax classifier, in other words, at each time step there is a softmax classifier.

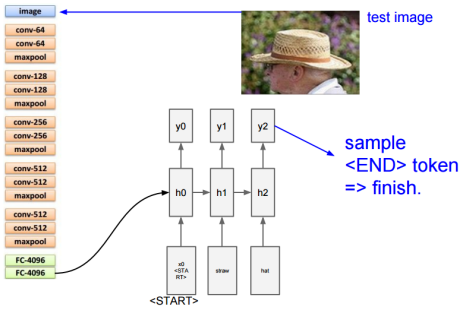


Figure 5: "Deep Visual-Semantic Alignments for Generating Image Descriptions", Karpathy and Fei-Fei

9.3 LSTM

Normally we will not use Vanilla RNNs, instead, all papers use LSTM. It is very similar, it stills take into account the input and the last state but now the combination of both is more complex and works better. With RNN we have one vector h at each time step. But with LSTM we have two vectors at each time step h_t and c_t. Moreover in RNNs the repeating module only has one layer, *tanh*. Instead, LSTM has four.

How do they work?

where:

- c cells. Are best though as counters
- h hidden states

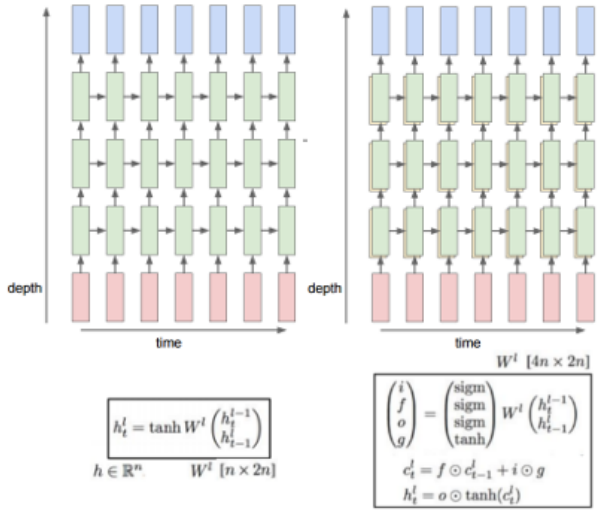


Figure 6: **Left**: RNN with h(green), **Right**: LSTM with h(hidden vector, green) and c (cell state vector, yellow)

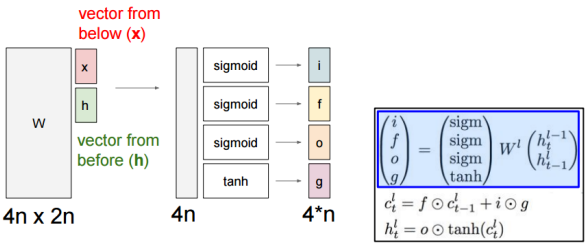


Figure 7: LSTM first equation diagram

- h_t^{l-1} vector from below of size $n \times 1$ (x in the figure)
- h_{t-1}^l vector from before of size $n \times 1$ (h in the figure)
- $i \in [0, 1]$ to chose if we want to add, or not, to a cell
- $f \in [0, 1]$ forget gate to reset cells to 0
- $o \in [0, 1]$ to choose which cells are used to produce
- $g \in [-1, 1]$ to add -1 or 1 to a cell

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM does have the ability

to remove or add information to the cell state, carefully regulated by structures called gates.

Think of i, f, o as boolean variables, we want them to have an interpretation like a gate. They a result of a sigmoid to make them differentiable. They allow us to reset and add to counters, as well as to choose what cells should be used to update h_t^l. We can do two operations to cells (counters):

- reset them with $f \odot c_{t-1}^l$
- add -1 or 1 with $i \odot g$
- choose what cells should be used with o to update h_t^l: $o \odot \tanh(c_t^l)$

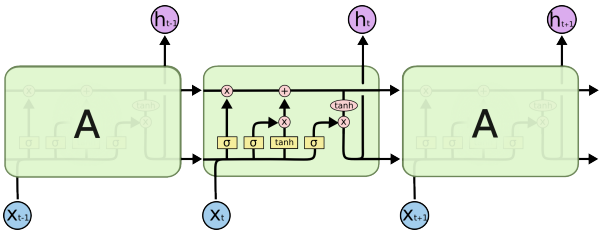
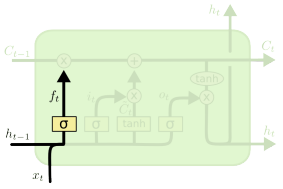


Figure 8: LSTM diagram

Step by step

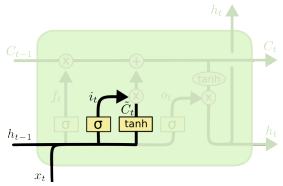
First Decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this." For example, in a language model trying to predict the next word based on all the previous ones the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 9: First, decide what information we're going to throw away from the cell state

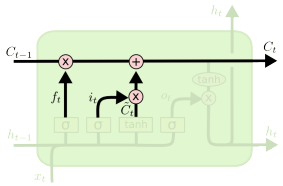
Second Decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a *tanh* layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state. For example, in the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 10: Second, decide what information we're going to store in the cell state

Third Update the old cell state, C_{t-1}, into the new cell state CtCt. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t\widetilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

Figure 11: Third, update the old cell state

Fourth Finally, we decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through *tanh* (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to. For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

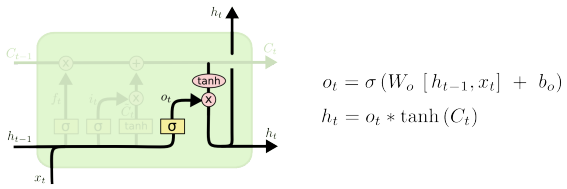


Figure 12: Fourth, decide what we're going to output

9.4 Variants on LSTMs

But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it's worth mentioning some of them. Greff, et al. (2015) do a nice comparison of popular variants, finding that they're all about the same.

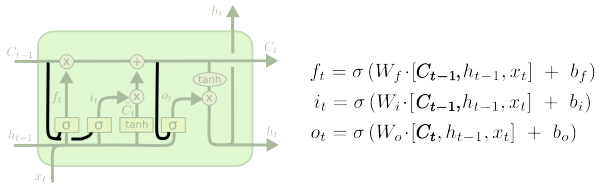


Figure 13: One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections." This means that we let the gate layers look at the cell state. The diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

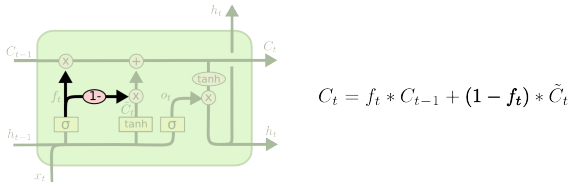
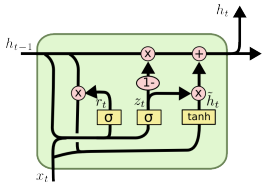


Figure 14: Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.

These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by Yao, et al. (2015). There's also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014).



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 15: A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

9.5 Difference between RNN and LSTM

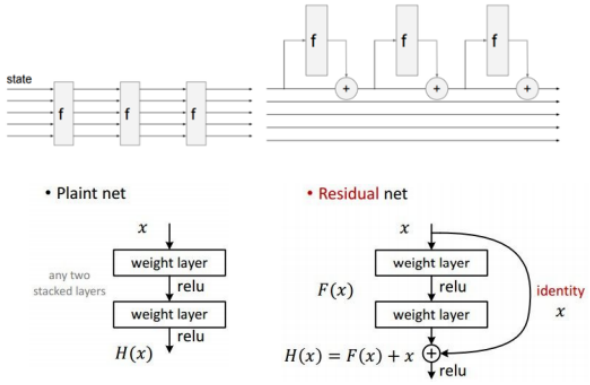


Figure 16: Difference between RNN (**left**) and LSTM (**right**)

- RNN transformative interaction of the state, LSTM additive interaction of the state.
- In RNN you are operating and transforming the state vector. So you are changing your hidden state from time step to time step. Instead, LSTM has these cells states flowing through. A subset of cells (or all of them) to compute the hidden state. Then, based on hidden state, we decide how to operate over the cell. We can reset it and/or adding interaction.
- RNNs look identical to plain nets, LSTMs to ResNets.
- RNNs has vanishing gradients, so you can not learn dependencies between distant time steps.LSTMs do not have the problem of vanishing gradients. In a video in evernote they introduce gradient noise to a layer and show how it evolves. We can see that RNN automatically kills it while LSTM

maintains it more time. This means that RNN can not learn long term relationships.

9.6 Bidirectional RNN

A Bidirectional Recurrent Neural Network is a type of Neural Network that contains two RNNs going into different directions. The forward RNN reads the input sequence from start to end, while the backward RNN reads it from end to start. The two RNNs are stacked on top of each others and their states are typically combined by appending the two vectors. Bidirectional RNNs are often used in Natural Language problems, where we want to take the context from both before and after a word into account before making a prediction.

9.7 Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.

10 Attention Models

First recall RNN for Captioning. It would be better if it could look the image more than once and focus its attention to specific parts of the image.

10.1 Soft Attention for Captioning

1. Compute the features with a CNN. Extract not the last ones but from an early layer. This is way it is a grid of features and not a single vector. In this way we have information of the localization of the input image features
2. Use this features to initialize the h0 hidden state
3. Now things get different with respect RNN for captioning. Instead of using h0 to compute distribution over words, we use it to compute a prob distribution over L locations. This would we implemented with a couple of affine layers (= FC) and softmax to give a distribution.

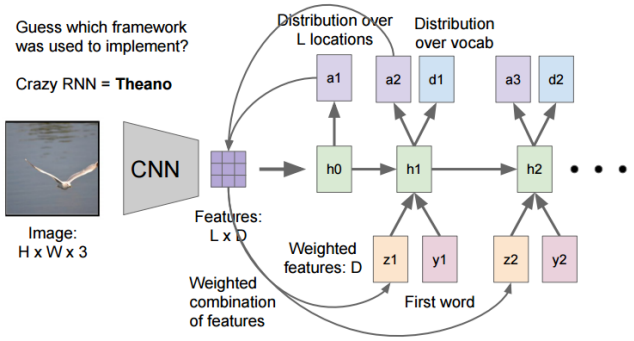


Figure 17: Soft Attention for Captioning - Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

4. We produce a weighted sum of features using the prob distribution over locations. This can be seen as taking the feature vector and summarizing it to a vector. This gives as a weighted features that is used to decide where to focus.
5. The next hidden state has as inputs the past hidden state and a word (like CNN) but now we also add the weighted features.
6. The hidden state is used to produce a prob distribution over words and also a new distribution over locations. These are implemented with a couple of FC layers on top of the hidden state.
7. Go to 4

10.2 Soft vs Hard Attention

Lets see how this summarization vectors z are produced

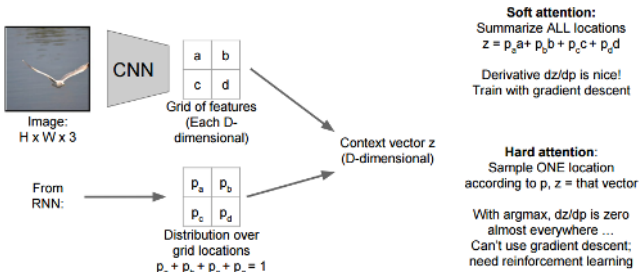


Figure 18: Soft Attention vs Hard Attention

Both soft and hard attention model produce the same output. Notice that the soft attention is more diffuse because it is averaging prob from the image. And Hard attention it is only focusing in one element.

Hard attention is normally faster at test time because is only focusing on an specific thing at every step instead of looking at big regions of the image.

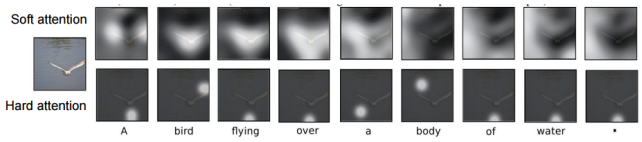


Figure 19: Example Soft Attention vs Hard Attention

Both have the same problem, they are constrained by a grid over the input image which makes them more blurry. z is not continuous. To work in a continuous way it exists the so called "Spatial Transformed Networks"

10.3 Recap

Soft attention:

- Easy to implement: produce distribution over input locations, reweight features and feed as input
- Attend to arbitrary input locations using spatial transformer networks

Hard attention:

- Attend to a single input location
- Can't use gradient descent!
- Need reinforcement learning (because they are not differentiable)

11 Generative Adversarial Nets (GAN)

A strategy to generate images with less math than Variational Autoencoders.

The idea is to start with a random noise that is normally drawn from a unit Gaussian (or something similar) and we will pass it through a Generator network. The Generator network looks very much like the decoder on the autoencoder and spits fake images. Then, a discriminator network is responsible for deciding/estimating if the input image is fake or a real.

How to we train them?

We train this network all together. The Generator will receive minibatches of random noise, at it will spit fake images. The Discriminator will receive batches partially fake and real; and it will try to decide which ones are real and which ones are fake.

The generator will improve its fake images by trying to trick the discriminator.

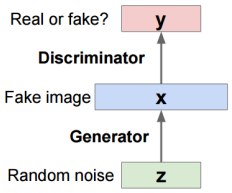


Figure 20: GAN

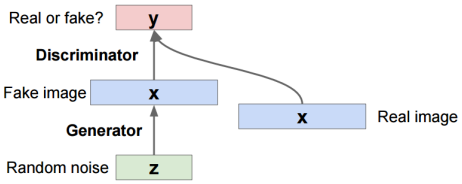


Figure 21: GAN

11.1 Examples

Interpolating random noise vectors. In figure 22 the images in the left and right extremes are obtained by sampling two random points (vectors) of the z space and passing them through the generative net. Then the interpolation between pairs of extreme images is performed by interpolating the z vectors and passing them through the generative net.

Figure 22: Interpolating random noise vectors

Another experiment shows that the generator is learning a useful representation of the data. In this experiment they randomly generated multiple images of faces and then manually organized in the classes "smiling woman", "neutral woman", "neutral man". Then, they produce the mean z average representation of each image and perform the arithmetic operation of z_smiling woman-z_neutral woman+z_neutral man produces a smiling man.

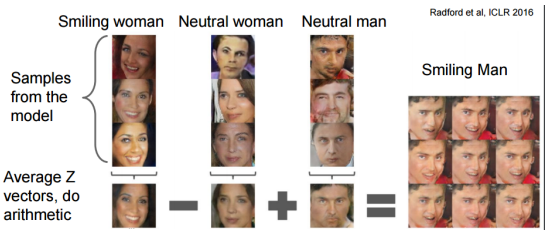


Figure 23: Generator learning a useful representation of the data

12 Reinforcement Learning Basic Model

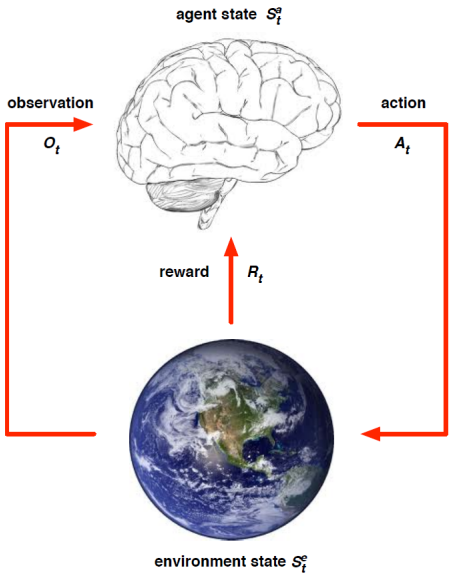


Figure 24: Basic Model of RL

The goal gere is to select a set of **actions** $\{A_1, A_2, A_3 ... A_t\}$ to maximize total future reward R_t based on the observations O_t. Since actions have long-term consequences and reward may be delayed, we cannot approach this problem greedily.

Definition **History** is the sequence of observation, actions and rewards. i.e.

$$H_t = A_1, O_1, R_1, A_2, O_2, R_2, ..., A_t, O_t, R_t$$

Definition A **state** contains the information that determines what happens next (i.e. $A_{t+1}, O_{t+1}, R_{t+1}$), and is defined as a function of the history.

$$S_t = f(H_t)$$

Definition The **environmental state**, S_t^e , is the information that determines the next observation O_{t+1} and reward R_{t+1} produce by the environment. Usually not visible to agent.

Defintition The **agent state**, S_t^a, is the information that determines the next action A_{t+1} produce by the agent, which is often a function of history. i.e. $S_t^a = f(H_t)$

12.1 Information State

Definition The **information state** (a.k.a **Markov state**) contains all useful information from the history. Given the present, the future is independent of the past. The Markove state is a sufficient statistic of the future. More formally, the state S_t is **Markov** i.f.f

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, S_2, S_3, ..., S_{t-1}, S_t] \tag{4}$$

The environmental state is Markov (by definition).

Proposition The history is also Markov.

Proof

$$\begin{aligned}\mathbb{P}[H_{t+1}|H_1, ..., H_t] =& \mathbb{P}[H_{t+1}|A_1, O_1, R_1, A_1, O_1, R_1, A_2, O_2, R_2, ..., \\ & A_1, O_1, R_1, A_2, O_2, R_2, ..., A_t, O_t, R_t] \\ =& \mathbb{P}[H_{t+1}|A_1, O_1, R_1, A_2, O_2, R_2, A_3, O_3, R_3..., A_t, O_t, R_t] \\ =& \mathbb{P}[H_{t+1}|H_t]\end{aligned}$$

12.2 Observability

An environment is **fully observable** if agent can directly observe the environment state. i.e. $S_t^a = O_t = S_t^e$. Formally known as Markov Decision Process (MDP).
An environment is **partially observable** agent indirectly observe the environment. $S_t^a \neq S_t^e$. Formally known as Partially Observable Markov Decision Process (POMDP). We can construct the agent state S_t^a by using:

- Complete History: $S_t^a = H_t$
- Beliefs on environment state: $S_t^a = (\mathbb{P}[S_t^e = s_1], ..., \mathbb{P}[S_t^e = s_n])$
- Recurrent Neural Network $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$

12.3 Components of RL agent

Definition Policy is the mapping of **state** to **action**

- Deterministic Policy: $a = \pi(s)$
- Probabilisitc Policy $\pi(\mathbf{a}|\mathbf{s}) = \mathbb{P}[A_t|S_t]$

Definition A **Value Function** is a function **associate with each policy** used to evalulate how good the states are, therefore use to select actions. Formally,

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s]$$

Definition A model predicts what the **environment** will do next.

- Transition Probability. The probaility of going from a state to another.

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

- Emission Probability. The expected reward obtained from the current state.

$$\mathcal{R}^a_s = \mathbb{E}[R_t|S_t = s, A_t = a]$$

12.4 Learning and Planning

If the environment is initially unknown, then we need **learning** by interact with the environment to improve the policy. A.k.a Exploration.
Conversely, if the environment is known, all wee need is to perform computations (query) the model to improve the policy. This is known as **planning** (a.k.a exploitation).

1. Query the emulator.
2. If action a is taken from state s, what is the score and what is the next state?
3. Do tree search to find the optimal policy.

13 Markov Decision Process

13.1 Introduction to MDP

Recall that a Markvo Decision Process (MDP) is characterized by its fully observable environment. That being said, any RL problems can be transformed into MDP.
The Markov Decision Process inherits from Markov Reward Process (MRP), which extends from the Markov Process (Markov Chain).

Definition Markov Process (MP) is the tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where

- $\mathcal{S}$ is the finite set of states
- $\mathcal{P}$ is the state transitional probability matrix. where an entry $\mathcal{P}^a_{ss'}$ is defined

$$\mathcal{P}_{ss'} := \mathbb{P}[S_{t+1} = s' | S_t = s]$$

Definition Markov Reward Process (MRP) is the tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

- $\mathcal{R}$ is the reward function

$$\mathcal{R}_s = \mathbb{E}[R_t | S_t = s]$$

- γ is the discount factor, $\gamma \in [0, 1]$

Definition The return G_t (not to be confused with reward R_t) is the total discounted reward from step t

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$

Definition The state value function $v(s)$ of an MRP gives the expected return.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] \\ &= \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \end{aligned}$$

Notice the equation for the value function casted in matrix form is known as **Bellman Equation**. Concretely,

$$\begin{aligned} v &= \mathcal{R} + \gamma \mathcal{P} v \\ (\mathrm{I} - \gamma \mathcal{P}) v &= \mathcal{R} \\ v &= (\mathrm{I} - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned}$$

13.2 Markov Decision Process

Finally, let's dig into the topic of today - the Markov Decision Process (MDP). MDP is an MRP with decisions, and states are Markov.

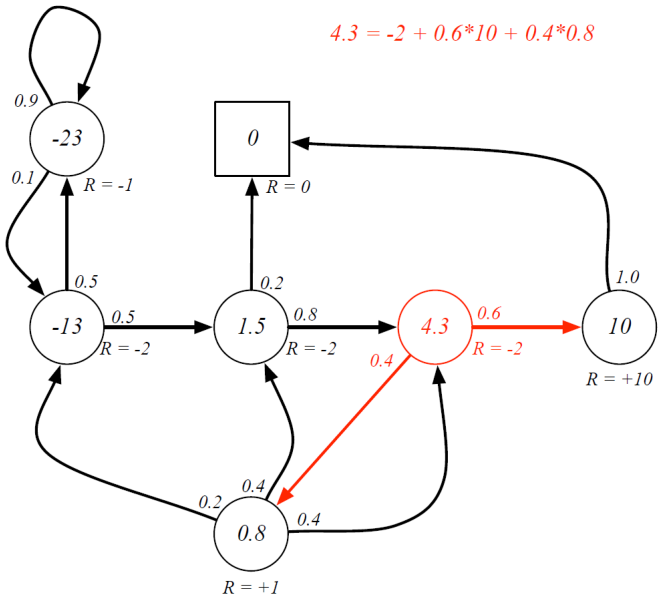


Figure 25: Value function of a Markov Reward Process

Definition The Markov Decision Process (MDP) is the tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{A} \rangle$ where

- $\mathcal{S}$ is the finite set of states
- $\mathcal{A}$ is the set of finite actions
- $\mathcal{P}$ is the state transitional probability matrix. where an entry $\mathcal{P}^a_{ss'}$ is defined
$$\mathcal{P}^a_{ss'} := \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$
- $\mathcal{R}$ is the reward function
$$\mathcal{R}^a_s = \mathbb{E}[R_t | S_t = s, A_t = a]$$
- γ is the discount factor, $\gamma \in [0, 1]$

Definition A policy (probabilistic) given by
$$\pi(\mathbf{a}|\mathbf{s}) = \mathbb{P}[A_t | S_t]$$
defines the behavior of the agent, is stationary and dependent of the current state (not history).

Note that the MDP $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{A} \rangle$ can be reduced to the MRP $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

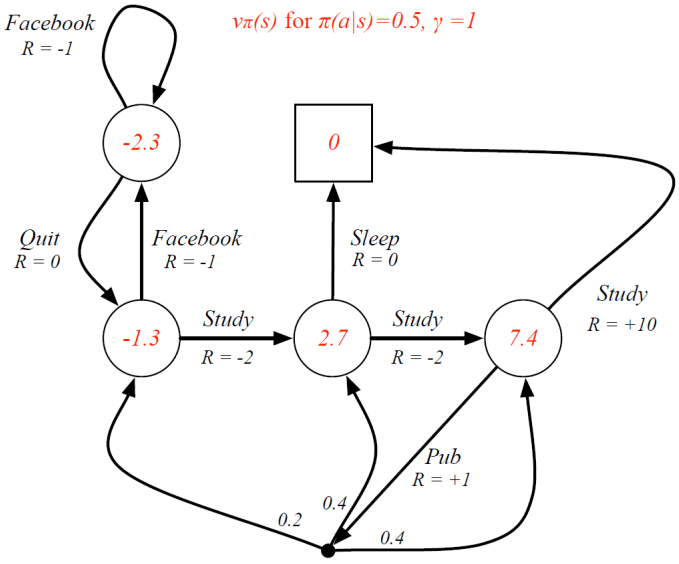


Figure 26: A Markov Decision Process

- $\mathcal{P}^{\pi}_{ss'} = \sum_{a \in \mathcal{A}} \pi((a|s))\mathcal{P}^{a}_{ss'}$
- $\mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \pi((a|s))\mathcal{R}^{a}_{ss'}$

Definition The *state-value* function is the expected return from state s following policy π

$$v_{\pi}(s) = \mathbb{E}[G_t|S_t = s]$$

Definition The *action-value* function is the expected return from state s, taking action a, and following policy π

$$q_{\pi}(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]$$

The inter-dependent relationship of The action-value and state-value functions are captured by the following equation.

$$v_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(\mathbf{a}|\mathbf{s}) q_{\pi}(s, a)$$

$$q_{\pi}(s, a) = \mathcal{R}^{a}_{s} + \gamma \sum_{s' \in \mathcal{S}} P^{a}_{ss'} v(s')$$

Expressing the equations above in a recursive manner, we have the **Bellman**

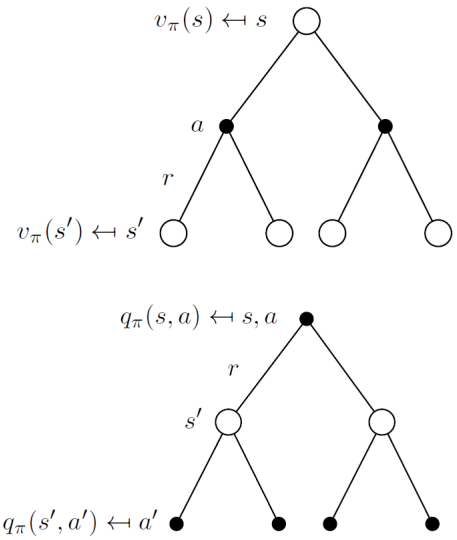


Figure 27: Recursive relation of value functions

Expectation Equation

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(\mathbf{a}|\mathbf{s})(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} P_{ss'}^a v(s'))$$
$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} P_{ss'}^a (\sum_{a'\in\mathcal{A}} \pi(\mathbf{a}'|\mathbf{s}')q_\pi(s',a'))$$

Definition The **state-value function** $v_*(s)$ is said to be **optimal** if

$$v_*(s) = \max_\pi v_\pi(s)$$

Similarly, The **action-value function** $q_*(s,a)$ is said to be **optimal** if

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Theorem Define $\pi \geq \pi'$ iff $v_\pi(s) \geq v_\pi|(s)$, for any Markov Process,

- $\exists\, \pi_*$ such that $\pi_* \geq \pi$
- All optimal policies achieve the same optimal value function $v_{\pi_*}(s) = v_*(s)$
- All optimal policies achieve the same optimal action-value function $q_{\pi_*}(s,a) = q_*(s,a)$

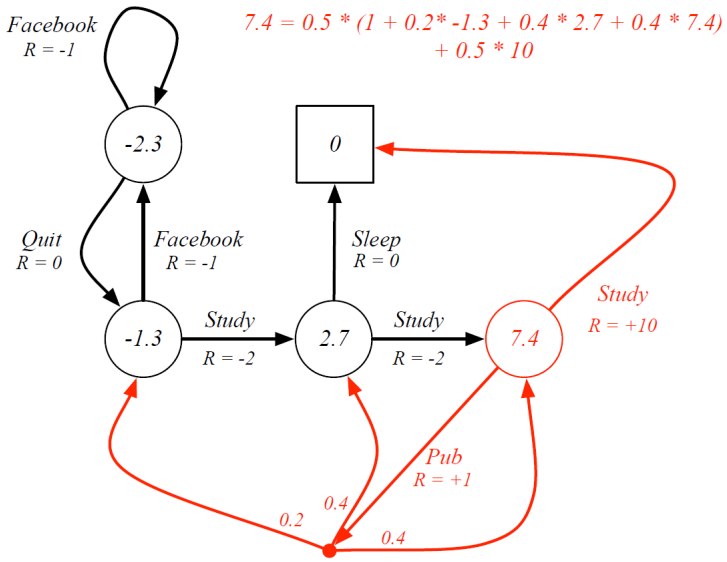


Figure 28: Computing state-value function using the Bellman Expectation Equation

Theorem An optimal policy can be found by maximizing over $q_*(s, a)$

$$\pi_*(\mathbf{a}|\mathbf{s}) = \begin{cases} 1, if a = \text{argmax}_{a \in A} q_*(s, a) \\ 0, otherwise \end{cases}$$

Finally, we summarize this chapter by introducing the **Bellman optimality Equation**

Definition The recursive definition of the optimal value function is given as the following:

$$v_*(s) = \max_a q_*(s, a) = \max_a (\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s'))$$

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a')$$

13.3 Infinite MDPs

The 3 cases of infinite MDPS are listed as follows.

1. Countable infinite

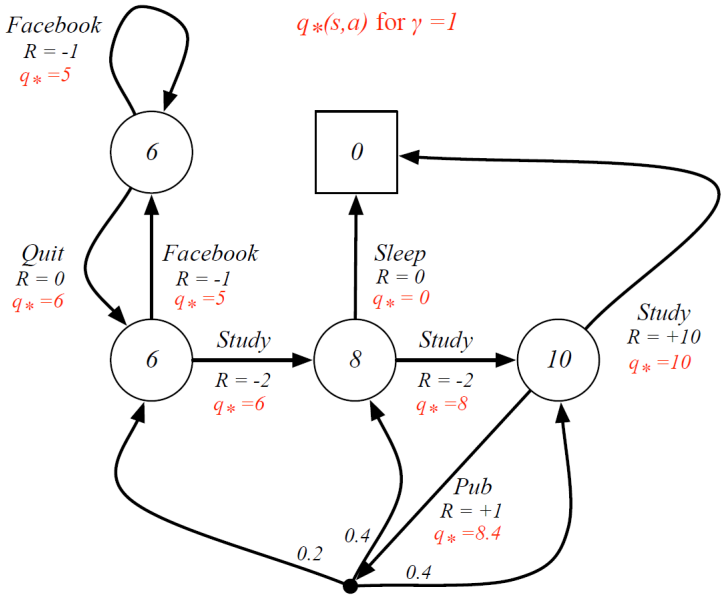


Figure 29: Computing optimal action-value function using the Bellman Optimality Equation

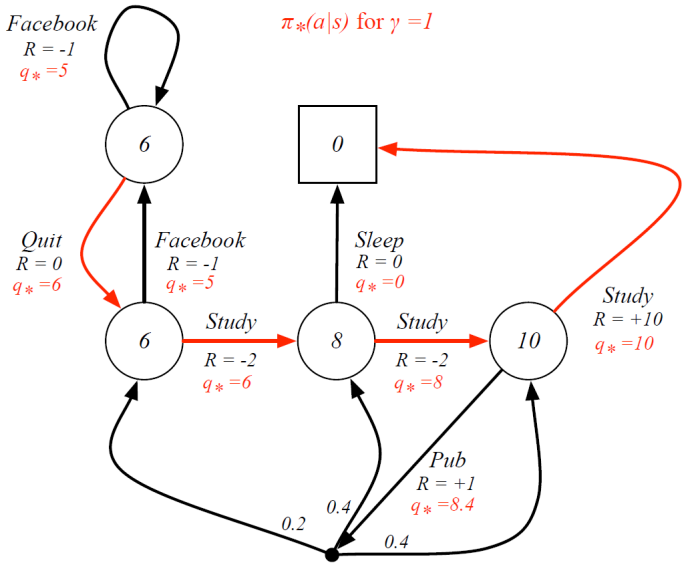


Figure 30: Finding the Optimal Policy

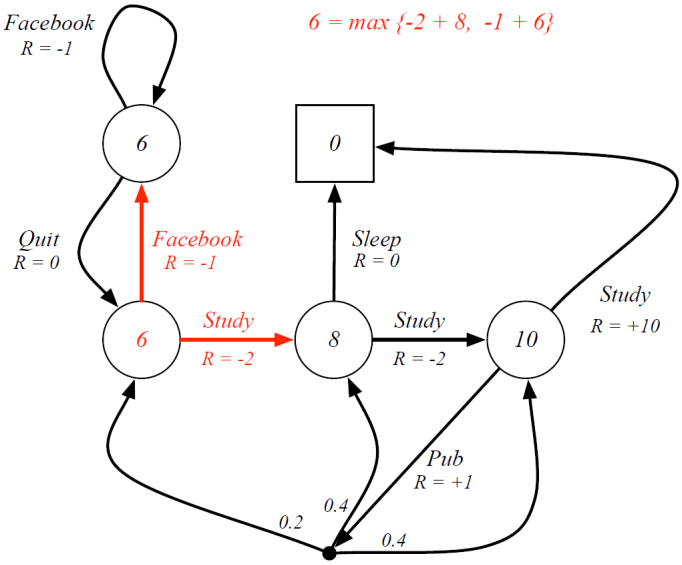


Figure 31: Finding Optimal value function $v_*(s)$