Machine Learning with Neural Networks

Professor James L. Crowley
Grenoble Institut Polytechnique
Univ. Grenoble Alpes

Outline

Glossary of Symbols

x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$
K	Number of classes
C_k	The kth class
$\vec{X} \in C_k$	Statement that an observation $\vec{X}$ is a member of class C_k
$\hat{C}_k$	The estimated class
$D(\vec{X})$	A Recognition (discriminant) function
$\hat{C}_k = D(\vec{X})$	A recognition function that predicts $\hat{C}_k$ from $\vec{X}$
	For a detection function (K=2), $C_k \in \{P,N\}$
y	The true class for an observation $\vec{X}$
$\{\vec{X}_m\}\ \{y_m\}$	Training samples of $\vec{X}$ for learning, along with ground truth $\vec{y}$
y_m	An indicator variable (or ground truth) for sample *m*
M	The number of training samples.
$\vec{y}$	A dependent variable to be estimated.
$\vec{y} = f(\vec{w}^T\vec{X}+b)$	A function (model) that predicts $\vec{y}$ from $\vec{X}$.
$\vec{w}, b$	The parameters of the model.
$C_m = \frac{1}{2}(a_m - y_m)^2$	The Loss (or cost) for the function for estimating y_m as a_m
$\vec{\nabla} C_m = \frac{\partial C_m}{\partial \vec{w}}$	The gradient (vector derivative) of the Loss (or cost).
$a_j^{(l)}$	The activation output of the *j*th neuron of the *l*th layer.
$w_{ij}^{(l)}$	The weight from unit *i* of layer *l–1* to the unit *j* of layer *l*.
b_j^l	The bias for unit *j* of layer *l*.
η	A learning rate. Typically very small (0.01). Can be variable.
L	The number of layers in the network.
$\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$	The Output Error of the network for the *m*th training sample
$\delta_{j,m}^{(l)}$	Error for the jth neuron of layer *l*, for the mth training sample.
$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)}\delta_{j,m}^{(l)}$	Update for weight from unit *i* of layer *l–1* to the unit *j* of layer *l*.
$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$	Update for bias for unit *j* of layer *l*.
ρ	The sparsity parameter
$(f*s)(n) = \sum_{m=1}^{N} f(m)s(n-m)$	The convolution equation. * is the convolution operator
$f*P(i,j) = \sum_{v=1}^{N}\sum_{u=1}^{N} f(u,v)P(i-u,j-v)$	2D convolution of a 2D filter *f(i,j)* with an image *P(i,j)*

$\vec{h}^{(t)}$	The hidden recurrent activation vector of the network. Note that in previous lectures we used $\vec{a}^{(t)}$ for activation.
$f(-)$	A process equation that computers $\vec{h}^{(t+1)}$ from $\vec{h}^{(t)}$
$\vec{X}^{(t)}$	A sequence of τ input vectors. Equivalent to $\{\vec{X}_1,...,\vec{X}_\tau\}$ in earlier lectures.
$\vec{o}^{(t)}$	The network output vector.

Machine Learning

Machine learning explores the study and construction of algorithms that can learn from and make predictions about data. Many of the foundational techniques for machine learning were originally developed for problems of detecting signals and recognizing patterns. However, as the scientific study of machine learning has matured, and as computing and data become increasingly available, it has become increasingly clear that machines can learn any computable function from data or experience with phenomena.

The term machine learning was coined in 1959 by Arthur Samuel, a pioneer in the field of computer gaming and inventor of the first computer program capable of learning to play checkers. An early landmark was a textbook by Nilsson in 1960 entitled Learning Machines, dealing mostly with machine learning for pattern classification using statistical techniques grounded in signal detection work from the early 20th century. A key landmark was the 1973 text-book by Duda and Hart named "Pattern Recognition and Scene Analysis", and the field was dominated by the scientific field of "Pattern Recognition" through the 1960's and 1970's.

We now understand that machine learning can be used to learn any computable function from data or experience, and can be used to learn techniques for pattern generation, for control of machines, for natural language interaction with humans, and for any form of intelligent behavior. Machine Learning is now seen as a core enabling technology for artificial intelligence. A modern definition would be:

Machine learning involves computers discovering how they can perform tasks without being explicitly programmed to do so.

Many of the techniques, including neural networks, have originally been developed for pattern recognition. Thus we will begin with techniques for the problems of recognizing patterns, and then show how these can be generalized to other forms of learning.

For pattern recognition, the classic approach is to use a set of "training data" training data $\{\vec{X}_m\}$ to estimate the discriminant function $\vec{g}(\vec{X})$. This can be done with a variety of techniques. A decision function $d\left(\vec{g}\left(\vec{X}\right)\right)$ is then used to select a pattern label from a set of possible target labels.

Supervised Learning

Most classical methods for machine learning, learn to estimate a function a set of labeled training data, composed of M independent examples, $\{\vec{X}_m\}$ for which we know a target value $\{y_m\}$. The set $\{\vec{X}_m\}$ is called the training data. The set $\{y_m\}$ are the indicator variables or target variables. Having target values $\{y_m\}$ makes it much easier to estimate the function $\hat{y} = f(\vec{X})$.

Semi-Supervised Learning.

A number of hybrid algorithms exist that initiate learning from a labeled training set and then extend the learning with unlabeled data, using the initial algorithm to generate synthetic labels for new data.

Unsupervised Learning

Unsupervised Learning techniques learn the function without a labeled training set. Most unsupervised learning algorithms are based on clustering techniques that associate data based on statistical properties. Examples include K-nearest neighbors, and Expectation Maximisation.

Self-Supervised Learning

Self-supervised learning learns to reconstruct data missing data and to guess associated data from examples. Two classic self-supervised techniques are masked-token completion and next sentence prediction. With Self-supervised learning, the data set is its own ground truth.

Reinforcement Learning

Reinforcement learning refers to techniques were a system learns through interaction with an environment. While originally developed for training robots to interact with the world, reinforcement learning combined with deep learning has recently produced systems that outperform humans at games such as Go or Chess. Deep reinforcement learning uses training with realistic simulators adapted through additional training with a target domain by transfer learning.

Transfer Learning

With transfer learning a system is first trained with a very large general-purpose data set or simulator, and then refined through additional training in a target domain. Transfer learning has provided a very useful method for overcoming the need for very large training data sets for most modern machine learning techniques based on Neural networks.

Perceptrons

History

The Perceptron is an incremental learning algorithm for linear classifiers invented by Frank Rosenblatt in 1956. The approach was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model. During the 1950's, Frank Rosenblatt developed the idea to provide a trainable machine for pattern recognition. The first Perceptron was a room-sized analog computer that implemented Rosenblatz's learning function for recognition. However, it was soon recognized that both the learning algorithm and the resulting recognition algorithm are easily implemented as computer programs.

The Perceptron Classifier

The perceptron is an on-line learning algorithm that learns a **linear decision boundary** (hyper-plane) for separable training data. As an "on-line" learning algorithm, new training samples can be used at any time to update the recognition algorithm. However, if the training data is non-separable, the method will not converge, and must be stopped after a certain number of iterations.

The Perceptron algorithm uses errors in classifying the training data to iteratively update the hyper-plane decision boundary. Updates may be repeated until no errors exist.

Assume a training set of M observations $\{\vec{X}_m\}$ of D features, with indicators variables, $\{y_m\}$ where

$$\vec{X}_m = \begin{pmatrix} x_{1m} \\ x_{2m} \\ \vdots \\ x_{Dm} \end{pmatrix} \text{ and } y_m = \{-1, +1\}$$

The indicator variable, $\{y_m\}$, tells the class label for each sample.
For binary pattern detection,

$y_m = +1$ for examples of the target class (class 1)
$y_m = -1$ for all others (class 2)

The Perceptron will learn the coefficients, $\vec{w}, b$, for a linear boundary

$$\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix} \text{ and } b$$

Such that for all training data, $\vec{X}_m$,

$\vec{w}^T \vec{X}_m + b \geq 0$ for Class 1 and $\vec{w}^T \vec{X}_m + b < 0$ for Class 2.

Note that $\vec{w}^T \vec{X}_m + b \geq 0$ is the same as $\vec{w}^T \vec{X}_m \geq -b$.
Thus b can be considered as a threshold on the product: $\vec{w}^T \vec{X}_m$

The decision function is the sgn() function: $\text{sgn}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$
Where $z = \vec{w}^T \vec{X}_m + b$

A training sample is correctly classified if:

$$y_m \cdot \left(\vec{w}^T \vec{X}_m + b \right) \geq 0$$

The algorithm requires a learning rate, η. Typically set to a very small number such as $\eta = 10^{-3}$

The Perceptron Learning Algorithm

The algorithm will continue to loop through the training data until it makes an entire pass without a single misclassified training sample. If the training data are not separable then it will continue to loop forever.

Algorithm:

```
$\vec{w}^{(0)} \leftarrow 0$; $b^{(i)} \leftarrow 0$, $i \leftarrow 0$; set η (for example η = $10^{-3}$)
WHILE update DO
    update ← FALSE;
    FOR $m = 1$ TO $M$ DO
        IF $y_m \cdot \left(\vec{w}^{(i)T}\vec{X}_m + b^{(i)}\right) < 0$ THEN
            update ← TRUE
            $\vec{w}^{(i+1)} \leftarrow \vec{w}^{(i)} - \eta \cdot y_m \cdot \vec{X}_m$
            $b^{(i+1)} \leftarrow b^{(i)} - \eta \cdot y_m$
            $i \leftarrow i + 1$
        END IF
    END FOR
END WHILE.
```

Notice that the weights are a linear combination of training data that were incorrectly classified.

The final classifier is: if $\vec{w}^{(i)T}\vec{X}_m + b^{(i)} \geq 0$ then P else N.

If the data is not separable, then the Perceptron will not converge, and will continue an infinite loop. Thus it is necessary to have a limit the number of iterations.

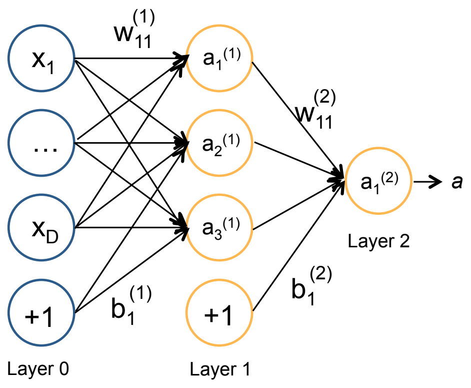
In 1969, Marvin Minsky and Seymour Papert of MIT published a book entitled "Perceptrons", that claimed to document the fundamental limitations of the perceptron approach. Notably, they claimed that a linear classifier could not be constructed to perform an "exclusive OR". While this is true for a one-layer perceptron, it is not true for multi-layer perceptrons.

The fact that the algorithm requires separable training data WAS a major weakness. This limitation was later overcome by reformulating the algorithm using a soft decision surface and Gradient descent. The result was promoted as a form of "Artificial Neural Network".

Artificial Neural Networks

In the 1970s, frustrations with the limits of Artificial Intelligence research based on Symbolic Logic led a small community of researchers to explore a perceptron based approach. In 1973, Steven Grossberg, showed that a two layered perceptron could overcome the problems raised by Minsky and Papert, and solve many problems that plagued symbolic AI. In 1975, Paul Werbos developed an algorithm referred to as "Back-Propagation" that uses gradient descent to learn the parameters for perceptrons from classification errors with training data. Back-propagation is a parallel form of Gradient descent easily implemented on a SIMD parallel computer.

Artificial Neural Networks are computational structures composed a weighted sums of "neural" units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

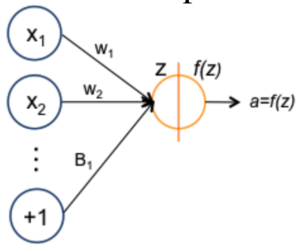


Note that the term "neural" is misleading. The computational mechanism of a neural network is only loosely inspired from neural biology. Neural networks do NOT implement the same learning and recognition algorithms as biological systems.

During the 1980's, Neural Networks went through a period of popularity with researchers showing that Networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, the resulting systems were fragile and difficult to duplicate. The popularity of Artificial Neural Networks was overtaken by more mathematically sound approaches for statistical pattern recognition based on Bayesian learning. These were, later, overtaken by techniques such support vector machines and kernel methods.

The Artificial Neuron

The simplest possible neural network is composed of a single neuron.



A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of an activation, a.

$$a = f(z)$$

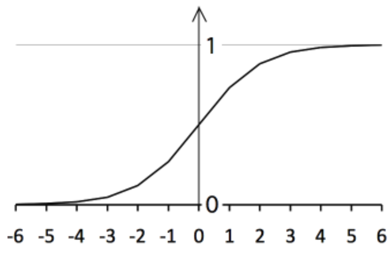
The neuron is composed of a weighted sum of input values

$$z = w_1x_1 + w_2x_2 + ... + w_Dx_D + b$$

followed by a non-linear "activation" function, $f(z)$

$$a = f(z) = f(\vec{w}^T\vec{X} + b)$$

A popular choice for activation function is the sigmoid: $\sigma(z) = \dfrac{1}{1+e^{-z}}$



The sigmoid is useful because the derivative is: $\dfrac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$

For the sigmoid, the target function is $y_m \in \{0,1\}$, enabling easy generalization to multi-class decisions. This can give a decision function:

if $f(\vec{w}^T\vec{X} + b) \geq 0.5$ the P else N

We will use Gradient descent to learn the best weights and bias for a training set of M samples $\{\vec{X}_m\}$ with indicator variables $\{y_m\}$.

Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the local minimum of a differentiable function. Gradient descent is a popular algorithm for estimating parameters for a large variety of models.

The gradient of a scalar-valued differentiable function of several variables, $f(\vec{X})$ is vector derivatives:

$$\vec{\nabla} f(\vec{X}) = \frac{\partial f(\vec{X})}{\partial \vec{X}} = \begin{pmatrix} \frac{\partial f(\vec{X})}{\partial x_1} \\ \frac{\partial f(\vec{X})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\vec{X})}{\partial x_D} \end{pmatrix}$$

The gradient of a function $f(\vec{X})$ at a point $\vec{X}$ is the direction and rate of change for the greatest slope of a surface. The direction of the gradient is the direction of greatest slope, the magnitude is the gradient is the rate of change in that direction.

To find a local minimum of a function using gradient descent, we iteratively update the function by subtracting corrections proportional to the gradient of the function at the current point. To use this to determine the parameters for a perceptron (or neural unit), we must introduce the notion of a Loss or cost for an error.

Loss (Cost) Function

The Loss (or cost) function is the cost of an error for classifying a data sample $\vec{X}_m$ with ground truth y_m using with network parameters $\vec{w}$. Assume M samples of training data $\vec{X}_m$ with indicator variables y_m. The vector, $\vec{X}_m$, has D dimensions. The indicator y_m, gives the expected result for the vector. Suppose that the neural unit uses a vector of weights, $\vec{w}$ and a bias, b, to estimate y_m from $\vec{X}_m$.

$$a_m = f(z_m) = f(\vec{w}^T \vec{X}_m + b)$$

The cost (or Loss) for using the weights and biases $\vec{w}$ to discriminate $\vec{X}_m$ is C_m

$$C_m = \frac{1}{2}(a_m - y_m)^2$$

Where we have multiplied by "1/2" to simplify the algebra.

The gradient of the cost with respect to each of the parameters tells us how much each parameter contributed to the error. We will use these to define a vector of correction factors for each parameter.

$$\vec{\nabla} C_m = \frac{\partial C_m}{\partial \vec{w}} = \begin{pmatrix} \frac{\partial C_m}{\partial w_1} \\ \vdots \\ \frac{\partial C_m}{\partial w_D} \\ \frac{\partial C_m}{\partial b} \end{pmatrix} = \begin{pmatrix} \Delta w_1 \\ \vdots \\ \Delta w_D \\ \Delta b \end{pmatrix}$$

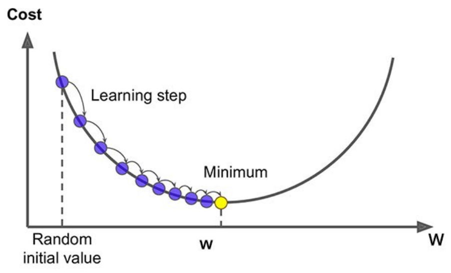
In order to evaluate these derivatives, we use the chain rule. Each gradient term can provides a correction term for the function parameters. For a single neural unit:

$$\Delta w_1 = \frac{\partial C_m}{\partial w_1} = \frac{\partial C_m}{\partial a_m} \cdot \frac{\partial a_m}{\partial z_m} \cdot \frac{\partial z_m}{\partial w_1}$$

To correct the network, we will subtract a fraction of this change from each of the network parameters. Because the training data typically contains many unmodelled phenomena (noise), the correction is weighted by a (very small) learning rate "η" to stabilize learning

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta \Delta \vec{w}_m$$

The fraction, η, is referred to as the Learning rate. Typical values for η are from η=0.01 to η=0.001.



(Drawing recovered from the internet - Source unknown)

The "optimum" coefficients are the coefficients that provide the smallest loss. To determine the optimum coefficients, we iteratively refine the model to reduce the errors, by subtracting a part of the derivative from the model parameters.

Ideally, at the optimum parameters, both the loss and the gradient are zero. For all other parameters, the loss increases. With real data, this will rarely be obtained because of noise in the training data.

Noise (un-modeled phenomena) in the training data will drive individual updates in random directions. A small learning rate is used to limit noise from driving the parameters too far from the optimum.

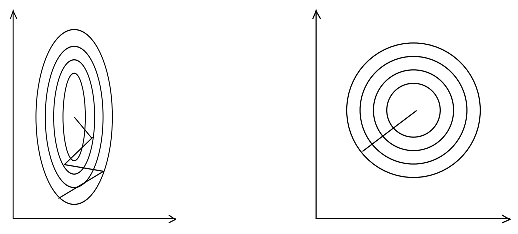
Warning: If you evaluate gradient descent by hand with real data, do not expect to easily see a path to convergence. Typically, arriving at the optimum requires a LOT of training data and MANY passes through the training data. Each pass through the training data is referred to as an "epoch". Gradient descent may require many epochs to reach an optimal (minimum loss) model.

Feature Scaling

For a training set $\{\vec{X}_m\}$ of M training samples with D values, if the individual features do not have a similar range of values, than large values will dominate the gradient. Small errors in this dimension are magnified.

One way to assure sure that features have similar ranges is to normalize the training data. A simple technique is to normalize the range of sample values.

For example, $$\forall_{m=1}^{M} : x_{dm} := \frac{x_{dm} - \min(x_d)}{\max(x_d) - \min(x_d)}$$



After estimating the model, use $\max(x_d)$ and $\min(x_d)$ to project the data back to the original space.

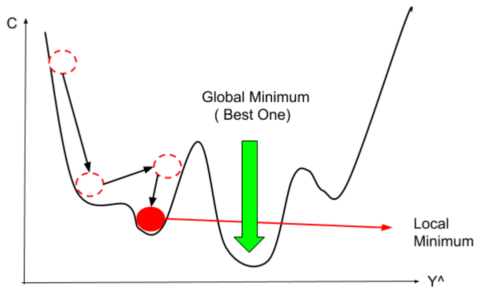
Note that the 2D surface shown here would correspond to two parameters, for example w, b for a single neural unit with a scalar input x. The actual surface is hyper-dimensional and not easy to visualize.

Local Minima

Gradient descent assumes that the loss function is convex. However, the loss function depends on real data $\vec{X}_m$ with unmodeled phenomena (noise).

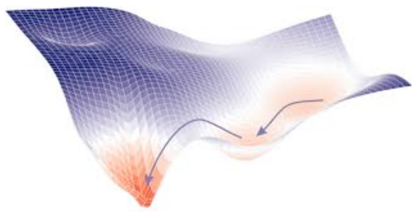
$$C_m = \frac{1}{2}\left(f(\vec{X}_m) - y_m\right)^2$$

Noise in the training samples $\{\vec{X}_m\}$ can create a non-convex loss with local minima.



(Drawing recovered from the internet - Source unknown)

In fact the gradient has MANY parameters, and the Loss function is evaluated in a very high dimensional space. It is helpful to see the data as a hyper-dimensional cloud descending (flowing over) a complex hyper-dimensional surface.



(Drawing recovered from the internet - Source unknown)

Batch mode

Individual training samples will send the model in arbitrary directions. While, updating with each sample will eventually converge, this tends to be costly. A more efficient approach is to correct the model with the average of a large set of training samples. The training data is typically divided into "folds" and the model is updated with the average of each fold.

This is called "batch mode".

$$\Delta\vec{w} = \frac{1}{M}\sum_{m=1}^{M}\Delta\vec{w}_m = \frac{1}{M}\sum_{m=1}^{M}\vec{\nabla}C_m$$

The model is then updated with the average error.

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta\Delta\vec{w}$$

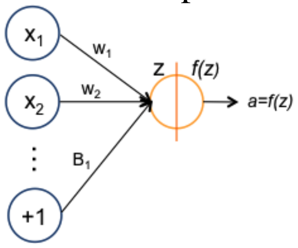
Stochastic Gradient Descent

Batch gradient descent often efficiently converges to a local minimum and becomes stuck. This can be avoided with stochastic gradient descent. With Stochastic gradient descent, a single training sample is randomly selected and used to update the model. This will send the model in random directions, that eventually flow to the global minima. While much less efficient than batch mode, this is less likely to become stuck in local minima.

Artificial Neural Networks

Artificial Neural Networks, also referred to as "Multi-layer Perceptrons", are computational structures composed a weighted sums of "neural" units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

The simplest possible neural network is composed of a single neuron.

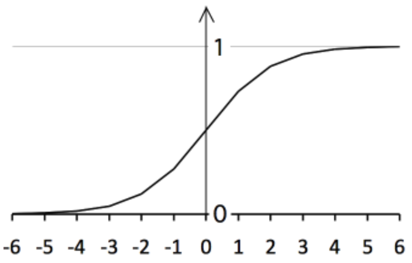


A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of an activation, a. The neuron is composed of a weighted sum of input values $z = w_1x_1 + w_2x_2 + ... + w_Dx_D + b$ followed by a non-linear "activation" function, $f(z)$

$$a = f(\vec{w}^T\vec{X} + b)$$

Many different activation functions may be used. Historically, the classic activation function is the sigmoid (or Logistic) activation function:

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

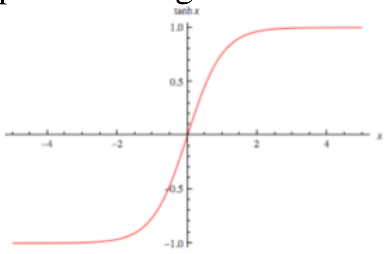


The sigmoid has long been used in biology and in economics to model processes that grow exponentially to a point of saturation. For example, the population of bacteria during fermentation, or the growth in performance of a new technology.

The sigmoid is useful because the derivative is: $\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$

Another classic decision functions is the hyperbolic tangent:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



For multiple classes, we can use the Softmax activation function.

$$f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

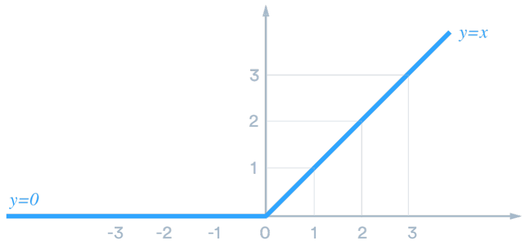
The softmax function takes as input a vector $\vec{z}$ of K real numbers, and normalizes it into a probability distribution consisting of K probabilities.

The softmax function is used to select the maximum from a vector of activations for K classes. Before applying softmax, the vector components of $\vec{z}$ will generally not sum to 1, and some of the components may be negative, or greater than one. After applying softmax, each component will be in the interval [0, 1] and the components will sum to 1. Thus the output can be interpreted as a probability distribution indicating the likelihood of each component.

Softmax is used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

The rectified linear function is popular for deep learning because of a trivial derivative:

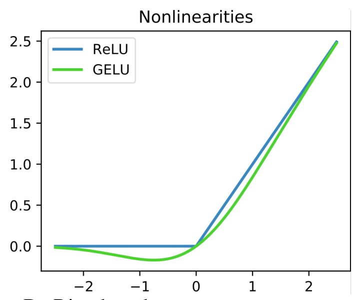
$$relu(z) = \max(0, z)$$



For $z \le 0$ $\frac{d(relu(z))}{dz} = 0$ for $z > 0$: $\frac{d(relu(z))}{dz} = 1$

Recently, a variation of RELU called GELU (Gaussian Error Linear Unit) has gained popularity.

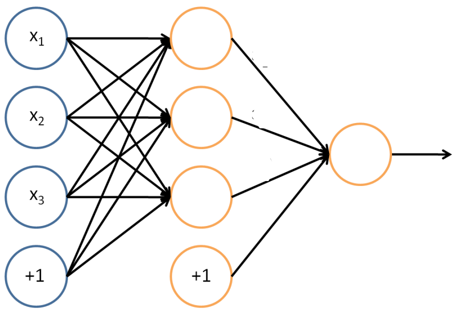
$$gelu(z) = 0.5z\left(1 + \frac{2}{\sqrt{\pi}} \int_0^{\frac{z}{\sqrt{2}}} e^{-x^2} dx\right)$$



From Wikipedia: By Ringdongdang - https://commons.wikimedia.org/w/index.php?curid=95947821

The Multilayer Neural Network model

A neural network is a multi-layer assembly of neurons. For example, this is a 2-layer network:



The circles labeled +1 are the bias terms.
The circles on the left are the input terms. Some authors, notably in the Stanford tutorials, refer to this as Level 1.

We will NOT refer to this as a level (or, if necessary, level L=0).
The rightmost circle is the output layer, also called L.
The circles in the middle are referred to as a "hidden layer". In this example there is a single hidden layer and the total number of layers is L=2.

The parameters carry a superscript, referring to their layer.
We will use the following notation:

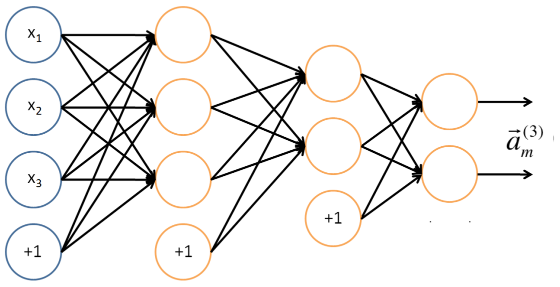
L	The number of layers (Layers of non-linear activations).
l	The layer index. l ranges from 0 (input layer) to L (output layer)
$N^{(l)}$	The number of units in layer l. $N^{(0)}=D$
$a_j^{(l)}$	The activation output of the jth neuron of the lth layer.
$w_{ij}^{(l)}$	The weight from the unit i of layer l-1 for the unit j of layer l.
$b_j^{(l)}$	The bias term for jth unit of the lth layer
$f(z)$	A non-linear activation function, such as a sigmoid, relu or tanh.

For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer.
$W_{13}^{(2)}$ is the weight for neuron 1 from the first level to neuron 3 in the second level.

The above network would be described by:

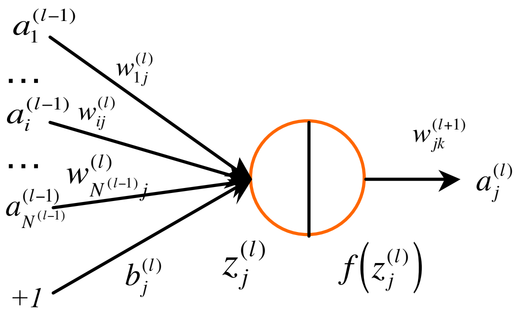
$$a_1^{(1)} = f(w_{11}^{(1)}X_1 + w_{21}^{(1)}X_2 + w_{31}^{(1)}X_3 + b_1^{(1)})$$
$$a_2^{(1)} = f(w_{12}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{32}^{(1)}X_3 + b_2^{(1)})$$
$$a_3^{(1)} = f(w_{13}^{(1)}X_1 + w_{23}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$
$$a_1^{(2)} = f(w_{11}^{(2)}a_1^{(1)} + w_{21}^{(2)}a_2^{(1)} + w_{31}^{(2)}a_3^{(1)} + b_1^{(2)})$$

This can be generalized to multiple layers. For example:



$\vec{a}_m^{(3)}$ is the vector of network outputs (one for each class) at the third layer.

Each unit is defined as follows:



The notation for a multi-layer network is

$\vec{a}^{(0)} = \vec{X}$ is the input layer. $a_i^{(0)} = X_d$

l is the current layer under discussion.

$N^{(l)}$ is the number of activation units in layer l. $N^{(0)} = D$

i,j,k Unit indices for layers l-1, l and l+1: $i \to j \to k$

$w_{ij}^{(l)}$ is the weight for the unit i of layer l-1 feeding to unit j of layer l.

$a_j^{(l)}$ is the activation output of the jth unit of the layer l

$b_j^{(l)}$ the bias term feeding to unit j of layer l.

$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$ is the weighted input to jth unit of layer l

$f(z)$ is a non-linear decision function, such as a sigmoid, tanh(), or soft-max

$a_j^{(l)} = f(z_j^{(l)})$ is the activation output for the jth unit of layer l

For layer l this gives:

$$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \qquad a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right)$$

and then for l+1 :

$$z_k^{(l+1)} = \sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \qquad a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)}\right)$$

So how to do we learn the weights W and biases B?

We could train a 2-class detector from a labeled training set $\{\vec{X}_m\},\{y_m\}$ using gradient descent. For more than two layers, we will need to use the more general "back-propagation" algorithm.

Back-propagation adjusts the network the weights $w_{ij}^{(l)}$ and biases $b_j^{(l)}$ so as to minimize an error function between the network output $\vec{a}_m^L$ and the target value $\vec{y}_m$ for the M training samples $\{\vec{X}_m\}$, $\{\vec{y}_m\}$.

This is an iterative algorithm that propagates an error term back through the hidden layers and computes a correction for the weights at each layer so as to minimize the error term.

This raises two questions:
1) How do we initialize the weights?
2) How do we compute the error term for hidden layers?

Initializing the weights
How do we initialize the weights?
The obvious answer is to initialize all the weights to 0.
However, this causes problems.

If the parameters all start with identical values, then the algorithm will end up learning the same value for all parameters. To avoid this, the parameters should be initialized with small random variables that are near 0, for example computed with a normal density with variance ε (typically 0.01).

$\forall_{i,j,l}\, w_{ji}^{(l)} = \mathcal{N}(X;0,\varepsilon)$ and $\forall_{j,l}\, b_j^{(l)} = \mathcal{N}(X;0,\varepsilon)$ where $\mathcal{N}$ is a sample from a normal density.

An even better solution is provided by Xavier GLOROT's technique.

Glorot, X. and Bengio, Y., 2010, March. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256). JMLR Workshop and Conference Proceedings.

Backpropagation

Back propagation is a distributed parallel algorithm for computing gradient descent. Back-propagation propagates the error term back through the layers, using the weights. We will present this for individual training samples. The algorithm can easily be generalized to learning from sets of training samples (Batch mode).

Given a training sample, $\vec{X}_m$, we first propagate the $\vec{X}_m$ through the L layers of the network (Forward propagation) to obtain an output activation $\vec{a}^{(L)}$.

We then compute an error term. In the case, of a multi-class network, this is a vector, with k components, one output for each hypothesis. In this case the indicator vector would be a vector, with one component for each possible class:

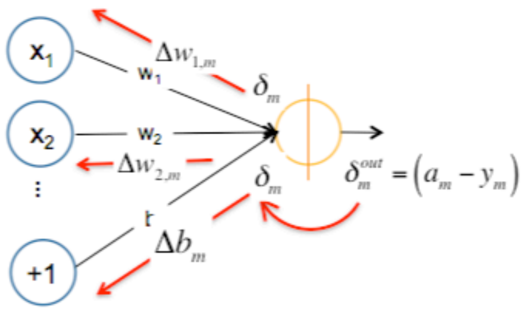
$$\vec{\delta}_m^{(out)} = \left(\vec{a}_m^{(L)} - \vec{y}_m\right) \qquad \text{or for each class k:} \quad \delta_{k,m}^{(out)} = \left(a_{k,m}^{(L)} - y_{k,m}\right)$$

To keep things simple, let us consider the case of a two class network, so that δ_m^{out}, $h(\vec{X}_m)$, $a_m^{(L)}$, and y_m are scalars. The results are easily generalized to vectors for multi-class networks.

For a single neuron, at the output layer, the "error" for each training sample is:

$$\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$$

The error term $\vec{\delta}_m^{out}$ is the total error for the whole network for sample m. This error is used to compute an error for the weights that activate the neuron:



$$\delta_m = \frac{\partial f(z)}{\partial z}\delta_m^{out}$$

This correction is then used to determine a correction term for the weights:

$$\Delta w_{d,m} = x_d \delta_m$$
$$\Delta b_m = \delta_m$$

Backpropagation can be generalized for multiple neurons at multiple layers (*l=1* to *L*). The error term for unit *k* at layer *L* is:

$$\delta_{k,m}^{(L)} = \frac{\partial f(z_k^{(L)})}{\partial z_k^{(L)}} \delta_m^{out}$$

For the hidden units in layers $l < L$ the error $\delta_j^{(l)}$ is based on a weighted average of the error terms for $\delta_k^{(l+1)}$.

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{l+1}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

We compute error terms, $\delta_j^{(l)}$ for each unit *j* in layer *l* back to layer *l–1* using the sum of errors times the corresponding weights times the derivative of the activation function. This error term tells how much the unit j was responsible for differences between the activation of the network $\vec{h}(\vec{x}_m; w_{jk}^{(l)}, b_k^{(l)})$ and the target value $\vec{y}_m$.

For the sigmoid activation function, $\sigma(z) = \frac{1}{1+e^{-z}}$ the derivative is:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

For $a_j^{(l)} = f(z_j^{(l)})$ this gives: $\delta_{j,m}^{(l)} = a_{j,m}^{(l)}(1-a_{j,m}^{(l)}) \cdot \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$

This error term can then used to correct the weights and bias terms leading from layer *j* to layer *i*.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)}$$
$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

Note that the corrections $\Delta w_{ij,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are NOT applied until after the error has propagated all the way back to layer *l=1*, and that when *l=1,* $a_i^{(0)} = x_i$.

For "batch learning", the corrections terms, $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are averaged over M samples of the training data and then only an average correction is applied to the weights.

$$\Delta w_{ij}^{(l)} = \frac{1}{M}\sum_{m=1}^{M}\Delta w_{ij,m}^{(l)} \qquad \Delta b_{j}^{(l)} = \frac{1}{M}\sum_{m=1}^{M}\Delta b_{j,m}^{(l)}$$

then

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\cdot\Delta w_{ij}^{(l)} \qquad b_{j}^{(l)} \leftarrow b_{j}^{(l)} - \eta\cdot\Delta b_{j}^{(l)}$$

where η is the learning rate.

Back-propagation is equivalent to computing the gradient of the loss function for each layer of the network. A common problem with gradient descent is that the loss function can have local minimum. This problem can be minimized by regularization. A popular regularization technique for back propagation is to use "momentum"

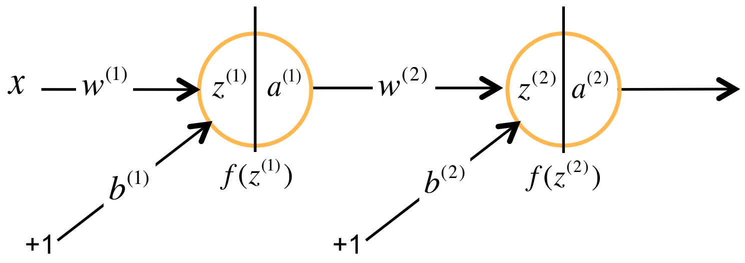
$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\cdot\Delta w_{ij}^{(l)} + \mu\cdot w_{ij}^{(l)}$$
$$b_{j}^{(l)} \leftarrow b_{j}^{(l)} - \eta\cdot\Delta b_{j}^{(l)} + \mu\cdot b_{j}^{(l)}$$

where the terms $\mu\cdot w_{j}^{(l)}$ and $\mu\cdot b_{j}^{(l)}$ serves to stabilize the estimation.

The back-propagation algorithm may be continued until all training data has been used. For batch training, the algorithm may be repeated until all error terms, $\delta_{j,m}^{(l)}$, are a less than a threshold.

Derivation of Backpropagation as gradient Descent.

To derive the backpropagation equations, consider a simple 2 layer network with 1 neuron at each level that maps a scalar feature, x, to a activation $a^{(2)}$.



The network equations are

$$\begin{aligned}
z^{(1)} &= w^{(1)}x + b^{(1)} \\
a^{(1)} &= f(z^{(1)}) = f(w^{(1)}x + b^{(1)}) \\
z^{(2)} &= w^{(2)}a^{(1)} + b^{(2)} \\
a^{(2)} &= f(z^{(2)}) = f(w^{(2)}a^{(1)} + b^{(2)})
\end{aligned}$$

The network has 4 parameters

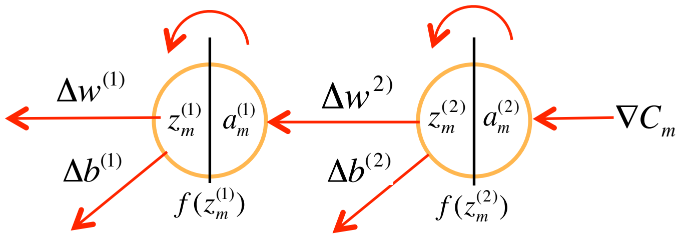
$$\vec{w} = \begin{pmatrix} w^{(1)} \\ b^{(1)} \\ w^{(2)} \\ b^{(2)} \end{pmatrix}$$

The "cost", C, of the error of the network for using the parameters to discriminate the input, X_m, with ground truth, y_m, is:

$$C_m = \frac{1}{2}\left(a_m^{(2)} - y_m\right)^2$$

Where we have multiplied by "1/2" to simplify the algebra.

The gradient of the cost with respect to each of the parameters in $\vec{w}$ tells us how much each parameter contributed to the error.



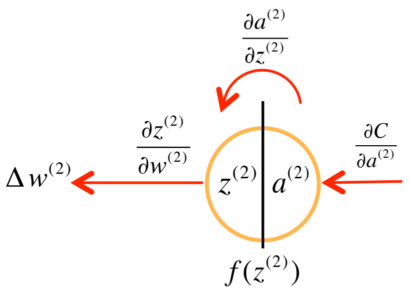
For our 2 layer network.

$$\nabla C = \frac{\partial C}{\partial \vec{w}} = \begin{pmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \frac{\partial C}{\partial w^{(2)}} \\ \frac{\partial C}{\partial b^{(2)}} \end{pmatrix} = \begin{pmatrix} \Delta w^{(1)} \\ \Delta b^{(1)} \\ \Delta w^{(2)} \\ \Delta b^{(2)} \end{pmatrix}$$

To evaluate these derivatives we use the chain rule. For example the derivative with of the cost with respect to the weight of the second neuron, $w^{(2)}$ is

$$\frac{\partial C}{\partial w^{(2)}} = \frac{\partial C}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}} = \Delta w^{(2)}$$

This can be seen graphically as:

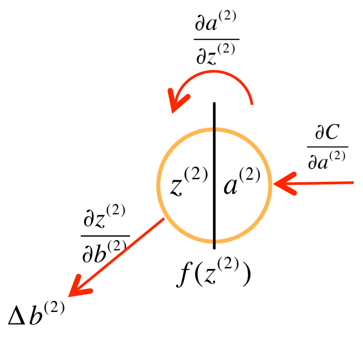


$\frac{\partial a^{(2)}}{\partial z^{(2)}}$
$\frac{\partial z^{(2)}}{\partial w^{(2)}}$
$\Delta w^{(2)}$
$z^{(2)}$
$a^{(2)}$
$f(z^{(2)})$
$\frac{\partial C}{\partial a^{(2)}}$

The derivative with respect to $b^{(2)}$ is:

$$\frac{\partial C}{\partial b^{(2)}} = \frac{\partial C}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(2)}} = \Delta b^{(2)}$$

This can be seen graphically as:



$\frac{\partial a^{(2)}}{\partial z^{(2)}}$
$z^{(2)}$
$a^{(2)}$
$\frac{\partial C}{\partial a^{(2)}}$
$\frac{\partial z^{(2)}}{\partial b^{(2)}}$
$f(z^{(2)})$
$\Delta b^{(2)}$

We can simplify the notation by defining an error term for each neuron.

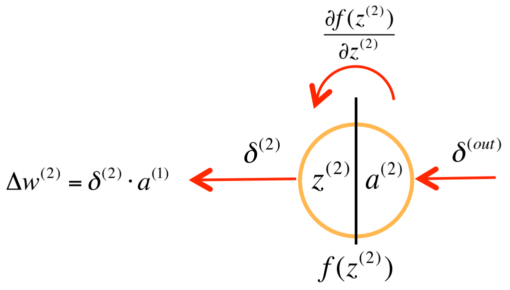
Let $\delta_m^{(out)} = \left(a_m^{(2)} - y_m\right) = \frac{\partial C_m}{\partial a_m^{(2)}}$ be the error for the error for the network for training sample X_m with ground truth indicator y_m.

The error term for 2nd neural unit is $\delta_m^{(2)} = \left(\frac{\partial a_m^{(2)}}{\partial z_m^{(2)}} \cdot \delta_m^{(out)}\right) = \left(\frac{\partial f(z_m^{(2)})}{\partial z_m^{(2)}} \cdot \delta_m^{(out)}\right)$

with this notation $\Delta w_m^{(2)} = \left(\frac{\partial C_m}{\partial a_m^{(2)}} \cdot \frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right) \cdot \frac{\partial z_m^{(2)}}{\partial w^{(2)}} = \frac{\partial z_m^{(2)}}{\partial w^{(2)}} \cdot \delta_m^{(2)}$

Reordering the terms and noting that $\frac{\partial z_m^{(2)}}{\partial w^{(2)}} = \frac{\partial(w^{(2)} a_m^{(1)} + b^{(2)})}{\partial w^{(2)}} = a_m^{(1)}$

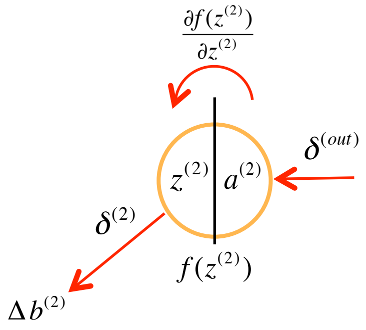
gives: $\Delta w_m^{(2)} = a_m^{(1)} \cdot \delta_m^{(2)}$


$\frac{\partial f(z^{(2)})}{\partial z^{(2)}}$
$\Delta w^{(2)} = \delta^{(2)} \cdot a^{(1)}$
$\delta^{(2)}$
$z^{(2)}$
$a^{(2)}$
$\delta^{(out)}$
$f(z^{(2)})$

Similarly for the bias term for the 2nd neural unit: $\Delta b_m^{(2)} = \left(\frac{\partial C_m}{\partial a_m^{(2)}} \cdot \frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right) \cdot \frac{\partial z_m^{(2)}}{\partial b^{(2)}} = \delta_m^{(2)} \cdot \frac{\partial z_m^{(2)}}{\partial b^{(2)}}$

Noting that $\frac{\partial z^{(2)}}{\partial b^{(2)}} = \frac{\partial(w^{(2)} a^{(1)} + b^{(2)})}{\partial b^{(2)}} = 1$

We obtain: $\Delta b_m^{(2)} = \delta_m^{(2)}$



For the next layer we continue the same process recursively

The derivative of the cost with respect to $w^{(1)}$ is:

$$\Delta w_m^{(1)} = \frac{\partial C_m}{\partial w^{(1)}} = \left(\frac{\partial C}{\partial a_m^{(2)}} \cdot \frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right) \cdot \left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right) \cdot \left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right) \cdot \frac{\partial z_m^{(1)}}{\partial w^{(1)}}$$

Substituting $\delta_m^{(2)} = \left(\frac{\partial C}{\partial a_m^{(2)}}\cdot\frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right)$ gives $\Delta w_m^{(1)} = \delta_m^{(2)}\cdot\left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right)\cdot\left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right)\cdot\frac{\partial z_m^{(1)}}{\partial w^{(1)}}$

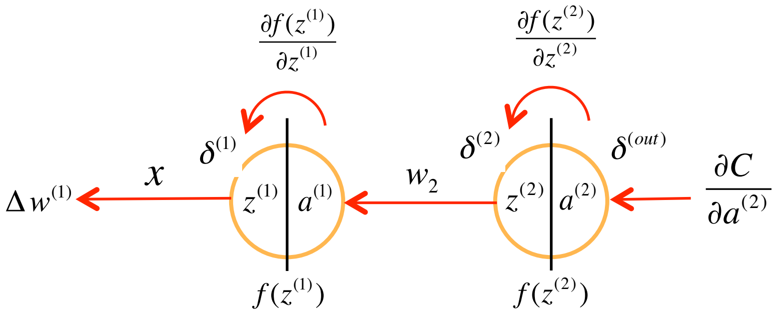
Substituting $w_m^{(1)} = \frac{\partial(w^{(2)}a_m^{(1)}+b^{(2)})}{\partial a_m^{(1)}} = \left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right)$ gives $\Delta w_m^{(1)} = \delta_m^{(2)}\cdot w^{(2)}\cdot\left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right)\cdot\frac{\partial z_m^{(1)}}{\partial w^{(1)}}$

Substituting $\frac{\partial f(z^{(1)})}{\partial z^{(1)}} = \left(\frac{\partial a^{(1)}}{\partial z^{(1)}}\right)$ gives $\Delta w^{(1)} = \left(\delta^{(2)}\cdot w^{(2)}\cdot\frac{\partial f(z^{(1)})}{\partial z^{(1)}}\right)\cdot\left(\frac{\partial z^{(1)}}{\partial w^{(1)}}\right)$

Substituting $x_m = \frac{\partial(w^{(1)}x_m+b^{(1)})}{\partial w^{(1)}} = \left(\frac{\partial z_m^{(1)}}{\partial w^{(1)}}\right)$ gives $\Delta w_m^{(1)} = \left(\delta_m^{(2)}\cdot w^{(2)}\cdot\frac{\partial f(z_m^{(1)})}{\partial z_m^{(1)}}\right)\cdot x_m$

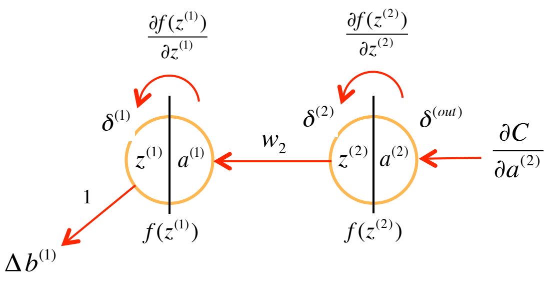
We define the error term for level 1 as $\delta_m^{(1)} = \left(\delta_m^{(2)}\cdot w^{(2)}\cdot\frac{\partial f(z_m^{(1)})}{\partial z_m^{(1)}}\right)$

Rearranging the terms gives: $\Delta w_m^{(1)} = x_m\cdot\delta_m^{(1)}$


$\frac{\partial f(z^{(1)})}{\partial z^{(1)}}$
$\frac{\partial f(z^{(2)})}{\partial z^{(2)}}$
$\delta^{(1)}$
$\delta^{(2)}$
$\delta^{(out)}$
$\Delta w^{(1)}$
x
$z^{(1)}$
$a^{(1)}$
w_2
$z^{(2)}$
$a^{(2)}$
$\frac{\partial C}{\partial a^{(2)}}$
$f(z^{(1)})$
$f(z^{(2)})$

Similarly for the correction factor of $b^{(1)}$

$$\Delta b_m^{(1)} = \frac{\partial C_m}{\partial b^{(1)}} = \left(\frac{\partial C_m}{\partial a_m^{(2)}}\cdot\frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right)\cdot\left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right)\cdot\left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right)\cdot\frac{\partial z_m^{(1)}}{\partial b^{(1)}}$$


$\frac{\partial f(z^{(1)})}{\partial z^{(1)}}$
$\frac{\partial f(z^{(2)})}{\partial z^{(2)}}$
$\delta^{(1)}$
$\delta^{(2)}$
$\delta^{(out)}$
$z^{(1)}$
$a^{(1)}$
w_2
$z^{(2)}$
$a^{(2)}$
$\frac{\partial C}{\partial a^{(2)}}$
1
$\Delta b^{(1)}$
$f(z^{(1)})$
$f(z^{(2)})$

Substituting $\delta_m^{(2)} = \left(\frac{\partial C_m}{\partial a_m^{(2)}}\cdot\frac{\partial a_m^{(2)}}{\partial z_m^{(2)}}\right)$ gives $\Delta b_m^{(1)} = \delta_m^{(2)}\cdot\left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right)\cdot\left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right)\cdot\frac{\partial z_m^{(1)}}{\partial b^{(1)}}$

Substituting $w_m^{(2)} = \frac{\partial(w^{(2)}a_m^{(1)}+b^{(2)})}{\partial a_m^{(1)}} = \left(\frac{\partial z_m^{(2)}}{\partial a_m^{(1)}}\right)$ gives $\Delta b_m^{(1)} = \delta_m^{(2)}\cdot w^{(2)}\cdot\left(\frac{\partial a_m^{(1)}}{\partial z_m^{(1)}}\right)\cdot\frac{\partial z_m^{(1)}}{\partial b^{(1)}}$

Substituting $\left(\frac{\partial a^{(1)}}{\partial z^{(1)}}\right) = \frac{\partial f(z^{(1)})}{\partial z^{(1)}}$ gives $\Delta b_m^{(1)} = \left(\delta_m^{(2)} \cdot w^{(2)} \cdot \frac{\partial f(z_m^{(1)})}{\partial z_m^{(1)}}\right) \cdot \left(\frac{\partial z_m^{(1)}}{\partial b^{(1)}}\right)$

noting $\left(\frac{\partial z_m^{(1)}}{\partial b_m^{(1)}}\right) = \frac{\partial(w^{(1)}x_m + b^{(1)})}{\partial w_m^{(1)}} = 1$ and substituting $\delta_m^{(1)} = \left(\delta_m^{(2)} \cdot w^{(2)} \cdot \frac{\partial f(z_m^{(1)})}{\partial z_m^{(1)}}\right)$

Gives: $\Delta b_m^{(1)} = \delta_m^{(1)}$

General formula for the error term

In general, the chain rule $\frac{\partial C}{\partial w^{(l)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \ldots \cdot \frac{\partial z^{(l+1)}}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial w^{(l)}}$

Provides a recursive formula for each neural unit:

$$\delta^{(l)} = \left(\frac{\partial f(z^{(l)})}{\partial z^{(l)}} \cdot w^{(l+1)} \cdot \left(\frac{\partial f(z^{(l+1)})}{\partial z^{(l+1)}} \cdot w^{(l+2)} \cdot \left(\cdots \left(\frac{\partial f(z^{(L)})}{\partial z^{(L)}} \cdot \delta^{(out)}\right)\right)\right)\right)$$

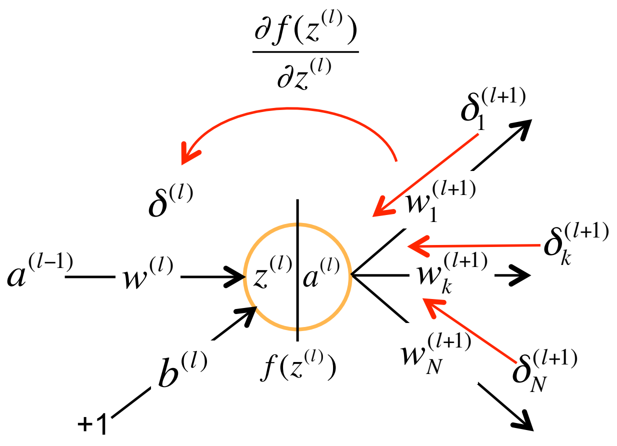
Giving a simple formula for adjusting the values of weights and biases

$\Delta w^{(l)} = a^{(l-1)}\delta^{(l)}$ and $\Delta b^{(l)} = \delta^{(l)}$

Formula for multiple activations

In the case where there are N neural units at level *l+1*,
the error at level *l* is the weighted sum of the errors at level *l+1*.

$$\delta^{(l)} = \left(\frac{\partial f(z^{(l)})}{\partial z^{(l)}} \cdot \sum_{k=1}^{N} w_k^{l+1} \cdot \delta_k^{(l+1)}\right)$$



Summary of Backpropagation

The Back-propagation algorithm can be summarized as:

1) Initialize the network and a set of correction vectors:

$$\forall_{i,j,l}\, w_{ji}^{(l)} = \mathcal{N}(X;0,\varepsilon)$$
$$\forall_{i,l}\, b_{j}^{(l)} = \mathcal{N}(X;0,\varepsilon)$$
$$\forall_{i,j,l}\, \Delta w_{ji}^{(l)} = 0$$
$$\forall_{i,l}\, \Delta b_{j}^{(l)} = 0$$

where $\mathcal{N}$ is a sample from a normal density, and ε is a small value.

2) For each training sample, $\vec{x}_m$, propagate $\vec{x}_m$ through the network (forward propagation) to obtain a network activation $a_m^{(L)}$. Compute the error and propagate this back through the network:

a) Compute the network error term: $\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$

b) Compute the error term at Layer L: $\delta_m^{(L)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \delta_m^{out}$

c) Propagate the error back from *l=L-1* to *l=1*: $\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$

d) Use the error at each layer to set a vector of correction weights.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)} \qquad \Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

3) For all layers, *l=1 to L*, update the weights and bias using a learning rate, η

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij,m}^{(l)} + \mu \cdot w_{ij}^{(l)}$$
$$b_{j}^{(l)} \leftarrow b_{j}^{(l)} - \eta \cdot \Delta b_{j,m}^{(l)} + \mu \cdot b_{j}^{(l)}$$

Note that this last step can be done with an average correction matrix obtained from many training samples (Batch mode), providing a more efficient algorithm.

Generative Networks

Deep learning was originally invented for recognition. The same technology can be used for generation. Up to now we have looked at what are called "discriminative" techniques. These are techniques that attempt to discriminate a class label, y from a feature vector, $\vec{X}$.

$$\begin{tikzcd}\vec{X} \arrow[r] & \boxed{D(\vec{X})} \arrow[r] & \hat{y}\end{tikzcd}$$

The same process can be used to learn a network that generates $\vec{X}$ given a code y. This is called a "generative" process.

$$\begin{tikzcd}y \arrow[r] & \boxed{G(y)} \arrow[r] & \vec{X}\end{tikzcd}$$

Given an observable random variable $\vec{X}$, and a target variable, gradient descent allows us to learn a joint probability distribution, $P(\vec{X},\vec{Y})$, where $\vec{X}$, is generally composed of continuous variables, and $\vec{Y}$ is generally a discrete set of classes represented by a binary vector.

A discriminative model gives a conditional probability distribution $P(\vec{Y} \mid \vec{X})$.
A generative model gives a conditional probability $P(\vec{X} \mid \vec{Y})$

We can combine a discriminative process for one data set with a generative process from another and use these to make synthetic outputs.

$$\begin{tikzcd}\vec{X} \arrow[r] & \boxed{D(\vec{X})} \arrow[r] & \hat{y} & y \arrow[r] & \boxed{G(y)} \arrow[r] & \vec{X}\end{tikzcd}$$

A classic example is an autoencoder. However, to learn the Autoencoder we need to change use a new form of loss function based on entropy.

Entropy

The entropy of a random variable is the average level of "information", "surprise", or "uncertainty" inherent in the variable's possible outcomes. Consider a set of M random scalars $\{X_m\}$ with N possible values, [1,N].

Formally: $\forall m = 1, M : \; h(X_m) \leftarrow h(X_m) + 1$

From this training set we can compute a probability distribution $P(X_m = x)$ more commonly written as $P(x)$

$$P(X_m = x) = \frac{1}{M} h(x)$$

The information in any one observation is

$$I(X_m = x) = -\log_2 \left(P(x) \right)$$

Using a log of base 2 gives us information measured in binary digits (bits). The negative sign assures that the number is always positive or zero, because the log of a number less than 1 is negative.

Information expresses the number of bits needed to encode and transmit the value for an event.

Low probability events are surprising and convey more information.
High probability events are unsurprising and convey less information.

For example, consider x to have N=2 values, say 1, or 2. Then the P(X=x)=0.5 and the information is I(X) is 1 bit. If X had 8 possible values then, all equally likely, then P(X=x) = 1/8 =1/(2^3) and the information is –3 bits.

Computing Entropy

For a set of M observations, the entropy is the expected value from the information from the observations. The entropy of the distribution measures the surprise (or information) obtained from an observation of a sample in the distribution. .

For a distribution *P(x)* of feature values X with N possible values, the entropy is

$$H(X) = -\sum_{x=1}^{N} P(x)\log_2\left(P(x)\right)$$

For example, for tossing a coin, there are two possible outcomes (N=2).
The probability of each outcome is P(X=x)=1/2.
This is the situation of maximum entropy

$$H(X) = -\sum_{x=1}^{2} P(x)\log_2(P(x)) = -\sum_{x=1}^{2}\frac{1}{2}\log_2\left(\frac{1}{2}\right) = -\sum_{x=1}^{2}\frac{1}{2}(-1) = 1$$

This is the most uncertain case. Similarly, in the case where there are N possible values for X, and all values are equally likely, then $P(X_m = x) = \frac{1}{N}$ and

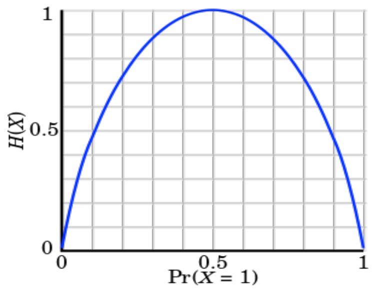
$$H(X) = -\sum_{x=1}^{N}\frac{1}{N}\log_2\left(\frac{1}{N}\right) = -\left(\frac{N}{N}\right)\log_2\left(\frac{1}{N}\right) = -\log_2\left(\frac{1}{N}\right)$$

For example, for 4 values, the entropy is 2 bits. It would require 2 bits to communicate an observation. On the other hand, consider when the distribution is a Dirac function, where X_m is always the same value of x_o,

$$P(x) = \delta(x - x_o) = \begin{cases} 1 & \text{if } x = x_o \\ 0 & \text{otherwise} \end{cases}$$

In this case, the value of X_m will always be x_o. Thus there is no information in an observation and the entropy will be zero. There is no surprise in an observation.

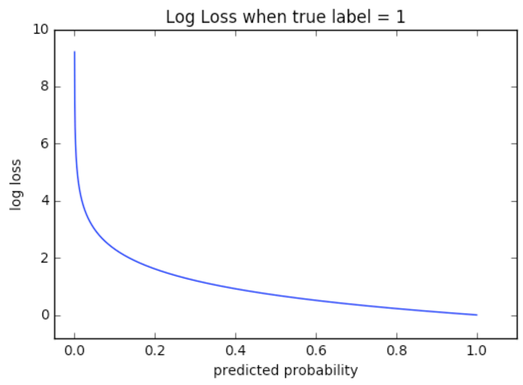
For any other distribution, Entropy measures the non-uniformity of the distribution.



Copied from https://en.wikipedia.org/wiki/Entropy_(information_theory)

Cross entropy

Cross-entropy is a measure of the difference between two probability distributions for a given set of events. Cross-entropy can be thought of as the total entropy between the distributions.



Copied from (https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html)

Cross-entropy loss can be used to measure the performance of a classification model whose output is a probability value between 0 and 1, as with the sigmoid or soft-max. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log-loss of 0.

Binary Cross-entropy loss is useful for training binary classifiers with the sigmoid activation function. Categorical Cross-Entropy is used to train a a multi-class network where softmax activation is used to output a probability distribution, $\vec{a}^{(out)}$, over the K classes .

Binary cross entropy

For a network with a single activation output, $a^{(out)}$

$$a^{(out)} = f(z^{(L)}) = \frac{1}{1+e^{-z^{(L)}}} = \frac{e^{z^{(L)}}}{e^{z^{(L)}}+1}$$

The Binary cross entropy is

$$C(a_m, y_m) = y_k \log(a_m) + (1-y_m)\log(1-a_m)$$

Categorical Cross Entropy Loss

For a network with a vector of K activation outputs, $\vec{a}^{(out)}$ with indicator vector $\vec{y}$ we calculate a separate loss for each target class.

The output activation is the softmax is

$$a_k = f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

and the Categorical cross entropy is

$$C(\vec{a}^{(out)}, \vec{y}) = -\sum_{k=1}^{K} y_k \log(a_k^{(out)})$$

When the indicators variables are encoded with one-hot encoding (Binary encoding with one variable for each output class), only the positive class where $y_k = 1$ is included in the loss. All other K-1 activations are multiplied by 0. In this case .

$$C(\vec{a}^{(out)}, \vec{y}) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

Where z$_k$ is the linear input for the positive case. The derivative for the positive activations is

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial f(z_k)}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-\log\left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)\right) = \left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} - 1\right)$$

The derivative for the negative class activations.

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial f(z_k)}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-\log\left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)\right) = \left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)$$

The Kullback-Leibler Divergence

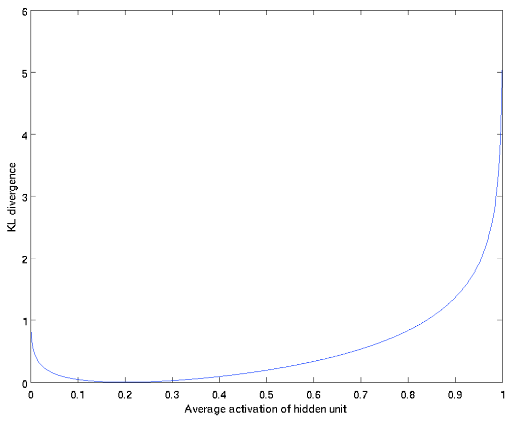
The Kullback-Leibler divergence, $D_{KL}(P \| Q)$ also known as the relative entropy of Q with respect to P measures the divergence between two distributions, P(X) and Q(X).

This can be used to define cross entropy as

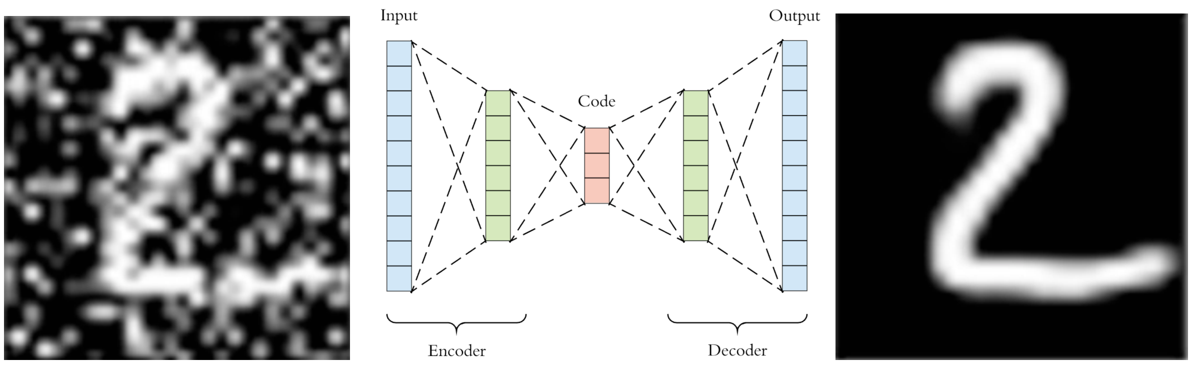
$$H(P,Q) = H(P) + D_{KL}(P \| Q)$$

We can use the Kullback-Leibler divergence to measure the divergence between a constant target activation, a, and an average observed activation for each unit, a_j.
The KL divergence between the desired and average activation is:

$$\sum_{j=1}^{N^{(1)}} KL(a \| a_j) = \sum_{j=1}^{N^{(1)}} \left(a \log \frac{a}{a_j} + (1-a) \log \frac{1-a}{1-a_j} \right)$$



AutoEncoders



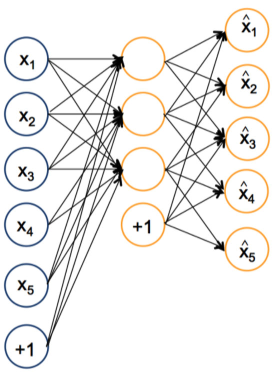
An auto-encoder is an unsupervised learning algorithm that uses back-propagation to learning a sparse set of features for describing the training data. Rather than try to learn a target variable, y_m, the auto-encoder tries to learn to reconstruct the input X using a minimum set of features (latent variables).

The autoencoder was initially invented as means to use back-propagation to perform Principal Components Analysis (PCA). For PCA, the loss (or cost) is the reconstruction error for a signal.

Let $\hat{X} = f(\vec{X})$ be the reconstructed version of a pattern $\vec{X}$. For PCA, cost (or loss function) of using $\hat{X} = f(\vec{X})$ to reconstruct $\vec{X}$ is the means square error of the reconstruction.

$$C(\vec{X}, f(\vec{X})) = \frac{1}{2}(\hat{X} - \vec{X})^2$$

An Autocoder learns to reconstruct (generate) clean copies of data without noise. The Key concepts are:
1) The training data is the target. The error is the difference between input and output
2) Training is with standard back-propagation (or gradient descent).



Using the notation from our 2 layer network, given an input feature vector $\vec{X}_m$ the auto-encoder learns $\{w_{ij}^{(1)}, b_j^{(1)}\}$ and $\{w_{jk}^{(2)}, b_k^{(2)}\}$ such that for each training sample, $\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ using as few hidden units as possible.

Note that $N^{(2)} = D$ and that $N^{(1)} << N^{(2)}$

When the number of hidden units $N^{(2)}$ is less than the number of input units, D, $\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ is necessarily an approximation. The hidden units provide a "lossy" encoding for $\vec{X}_m$. This encoding can be used to suppress noise!

The error for back-propagation for each unit is a vector $\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$ with a component $\delta_{i,m}$ for component $x_{i,m}$ of the training sample $\vec{X}_m$

The hidden code is composed of independent "features" that capture some component of the input vector. Each cell of the code vector is driven by a receptive field whose sum of products with the receptive fields of other code cells is almost zero.

With PCA, the code vectors are required to be orthogonal. For pattern recognition, it is sufficient that that the code vectors use a minimum number of independent hidden units (Code vectors). This is done with an information theoretic term referred to as "sparsity". Sparsity forces learning to generate the smallest set of code vectors that can reconstruct the training data without noise. The code vectors may have some slight overlap.

The average degree of independence is captured by a "sparsity parameter", $\hat{\rho}$.

The Sparsity Parameter

The sparsity $\hat{\rho}_j$ is the average activation for each of the hidden units $j=1$ to $N^{(1)}$.

The auto-encoder will learn weights subject to a sparseness constraints specified by a target sparsity parameter ρ, typically set close to zero.

The simple, 2-layer auto-encoder is described by:

Level 0: $\vec{X}_m = \begin{pmatrix} x_{1,m} \\ \vdots \\ x_{D,m} \end{pmatrix}$ an input vector level 1: $\vec{Y}_m = a_{j,m}^{(1)} = f(\sum_{i=1}^{D} w_{ij}^{(1)} x_{i,m} + b_j^{(1)})$ the code vector

level 2: $\hat{X}_m = a_{k,m}^{(2)} = f(\sum_{j=1}^{N^{(1)}} w_{jk}^{(2)} a_{j,m}^{(1)} + b_k^{(2)})$ the reconstruction of the input.

The output should approximate the input.

$$\vec{a}_m^{(2)} = \begin{pmatrix} a_1^{(2)} \\ \vdots \\ a_D^{(2)} \end{pmatrix} = \hat{X}_m \approx \vec{X}_m, \quad \text{with error } \vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

The sparsity $\hat{\rho}_j$ for each hidden unit (code component) is computed as the average activation for the M training samples:

$$\hat{\rho}_j = \frac{1}{M} \sum_{m=1}^{M} a_{j,m}^{(1)}$$

The auto-encoder is trained to minimize the average sparsity. This is accomplished using back propagation, with a simple tweak to the cost function.

Standard back propagation tries to minimize a loss based on the sum of squared errors. The loss for each sample is.

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - y_m)^2$$

For an auto-encoder, the target output is the input vector, and the loss is squared difference from the input vector:

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2$$

To impose “sparsity” we add an additional term to the loss.

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2 + \beta \sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$$

where $\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$ is the Kullback-Leibler Divergence of the vector of hidden unit activations and β controls the importance of the sparsity parameter.

The average activation $\hat{\rho}_j$ is used to compute the correction. Thus you need to compute a forward pass on a batch of training data, before computing the back-propagation. Thus learning is necessarily batch mode.

The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by the target sparsity, ρ. Thus the hidden units act as a form of basis space for the input vectors. The values of the hidden code layer are referred to as latent variables. The latent variables provide a compressed representation that reduces dimensionality and eliminates random noise.

To incorporate the KL divergence into back propagation, we replace

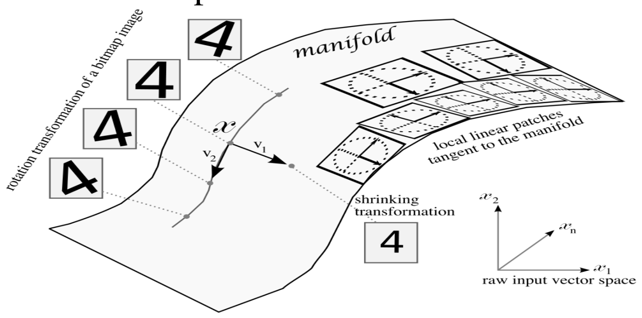
$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)}$$

with

$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \left(\sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)} + \beta \left(-\frac{a}{a_j} + \frac{1-a}{1-a_j} \right) \right)$$

where $N^{(2)} = D$, the size of the size of the input and output vectors. (The network output has the same number of components as the input).

AutoEncoders project the data onto a non-linear manifold that (should) provide a better representation of the latent space.


rotation transformation of a bitmap image
manifold
local linear patches tangent to the manifold
shrinking transformation
raw input vector space
x
v_1
v_2
x_1
x_2
x_n

Affine Transformations of a Bitmap Image

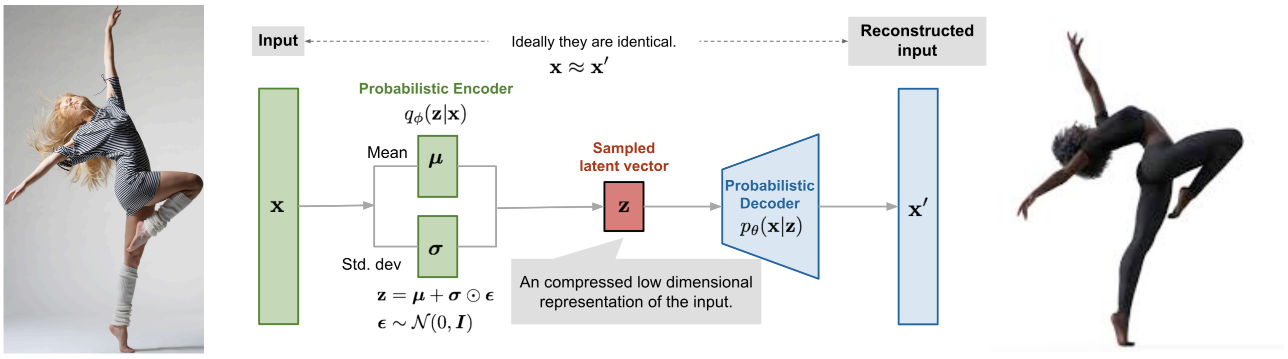
(Illustration from the NAACL 2013 lecture from R. Socher and C. Manning)

Positions on this manifold are expressed as vectors of latent variables. Noise is not encoded in the latent variables. Thus the latent variable can be used to reconstruct any signal on the manifold, without the presence of any signal (noise) that was not part of the manifold.

Variational Autoencoders

The output of an auto-encoder can be used to drive a decoder to produce a filtered version of the encoded data or of another training set. However, the output from an auto-encoder is discrete.

We can adapt an auto-encoder to generate a *nearly* continuous output by replacing the code with a probabilistic code represented by a mean and variance.


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{x}$
Mean
$\boldsymbol{\mu}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

This is called a Variational Autoencoder (VAE). VAEs combine a discriminative network with a generative network. VAEs can be used to generate "deep fake" videos sequences.

For a fully connected network, decoding is fairly obvious. The network input is a binary vector $\vec{Y}$ with k binary values y_k, with one for each target class. This is a code. The output for a training sample $\vec{Y}_m$ is an approximation of a feature vector belonging to the code class, $\hat{\vec{X}}_m$

$$\vec{a}_m^{(2)} = \hat{\vec{X}}_m \approx \vec{X}_m$$

and the error is the difference between a output and the actual members of the class.

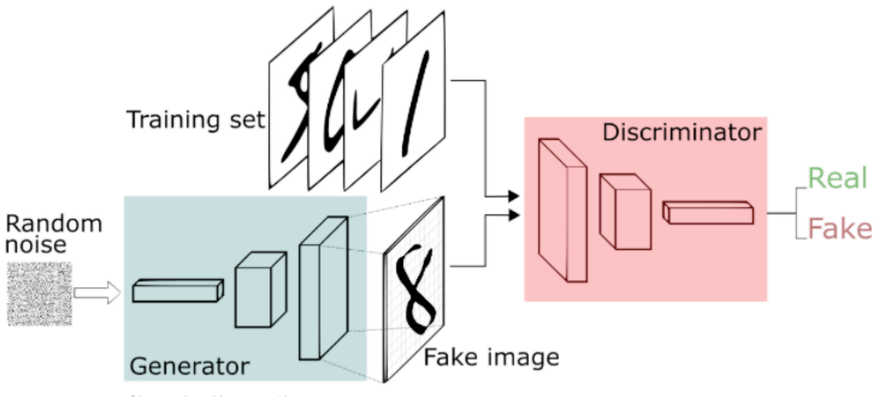
$$\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

The average error for at training set $\{\vec{Y}_m\}$, $\{\vec{X}_m\}$ can be used to drive back-propagation.

Generative Adversarial Networks

It is possible to put a discriminative network together with a generative network and have them train each other. This is called a Generative Adversarial Network (GAN).

A Generative Adversarial Network places a generative network in competition with a Discriminative network.



The two networks compete in a zero-sum game, where each network attempts to fool the other network. The generative network generates examples of an image and the discriminative network attempts to recognize whether the generated image is realistic or not. Each network provides feedback to the other, and together they train each other. The result is a technique for unsupervised learning that can learn to create realistic patterns. Applications include synthesis of images, video, speech or coordinated actions for robots.

Generally, the discriminator is first trained on real data. The discriminator is then frozen and used to train the generator. The generator is trained by using random inputs to generate fake outputs. Feedback from the discriminator drives gradient ascent by back propagation. When the generator is sufficiently trained, the two networks are put in competition.

Convolutional Neural Networks.

Fully-Connected Networks

Towards the end of the first wave of popularity of Neural Networks in the late 80s, several researchers began experimenting with networks composed of more than 3 layers. Most experiments explored fully connected networks, where each unit at layer $l+1$ receives activations from all units at layer l. The result is a very rapid growth in the number of parameters to learn, even for simple problems.

If there are $N^{(l)}$ units at layer l and $N^{(l+1)}$ units are layer $l+1$ then a fully connected network requires learning $N^{(l)}\cdot N^{(l+1)}$ parameters for layer l. Reliable learning requires that the number of data samples exceed the number of parameters. While this may be tractable for small examples, it quickly becomes excessive for practical problems, as found in computer vision or speech recognition.

For example, a typical image may have 1024 x 2048 = 2^{21} pixels. If we assume, say a 512 x 512 =2^{18} hidden units we have 2^{39} parameters to learn for a single class of image pattern, requiring more than 2^{39} training images. Clearly this is not practical (and, in any case not necessary).

Early Convolutional Neural Networks: LeNet5

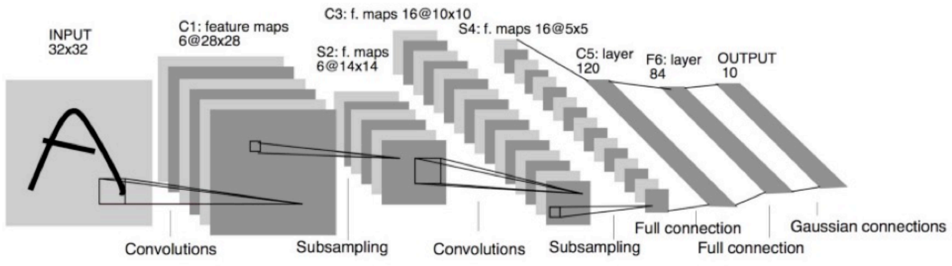
From 1988, Yann LeCunn began experimenting with a series of multi-layer architectures, referred to as LeNet, for the task of recognizing handwritten characters.

LeCunn's first insight was to limit each neural unit to a connection to small window of units in the previous level, and to learn the same weights for all units. This leads to a technique where all possible, overlapping, image windows of size NxN provide training data to train a small number of parameters for a receptive fields network. The network then uses the same learned weights with every hidden cell. Recall that, generally, the amount of training required for a network depends on the number of parameters to be trained. Thus any technique that gives equivalent performance with fewer parameters will scale to larger networks.

The resulting operation is equivalent to a “convolution” of the learned weights with the input signal and the learned weights are referred to as “receptive fields” in the neural network literature.

A second insight was to use several convolutional units in parallel to describe each window. This lead to a map of features for each pixel with the number of units referred to as "depth".

A third insight was to reduce the resolution of the image by resampling while increasing the number of parallel receptive fields (depth) at each level. This can be illustrated with the LeNet5 architecture shown here:



The LeNet5 architecture (1994)

In 1994 Yann LeCunn showed that LeNet5 provided the best performance for written character recognition. Because processing power, memory and training data were very limited at that time, many of the innovations in LeNet5 concerned methods to reduce parameters and computing without degrading performance.

LeNet5 is composed of multiple repetitions of 3 operations: Convolution, Pooling, Non-linearity. Convolution windows were of size 5x5 with a stride of 1, no zero padding and a depth of 6. That is 6 receptive fields are learned for each pixel in the first layer. Using 5x5 filters without zero padding reduced the input window of 32 x 32 pixels to a layer of composed of 6 sets of 28 x 28 units. A Sigmoid was used for the activation function. Pooling was performed as a spatial averaging over 2x2 windows giving a second layer of 6 x 14 x 14. The output was then convolved with 16 5x5 receptive fields, yielding a layer with 16 x 10x10 units. Average pooling over 2x2 windows reduced this to a layer of 16x5x5 units. These were then fed to two fully connected layers and then smoothed with a Gaussian filter to produce 10 output units, one for each possible digit.

Despite the experimental success, LeCun found it very difficult to publish his results in the computer vision and machine learning literatures, which were more concerned with multi-camera geometry and Bayesian approaches to recognition. The situation began to change around 2010, driven by the availability of GPUs, and planetary scale data (continued exponential growth of the World Wide Web) and the emergence of challenged based research in computer vision. During this period, computer vision and machine learning were increasingly organized around open competitions for Performance Evaluation on benchmark data sets.

Many of the insights of LeNet5 continued to be relevant as more training data, and additional computing power enabled larger and deeper networks, because they allowed more effective performance for a given amount of training data and parameters.

The Convolution Equation

For a digital signal, *s(n)*, the equation for convolution of a Finite Impulse Response (FIR) digital filter, *w(n)* composed of *N* coefficients is:

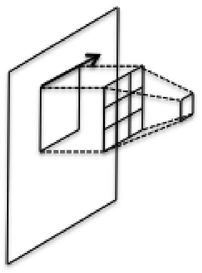
$$(w * s)(n) = \sum_{m=1}^{N} w(m)s(n-m)$$

For image processing, the signal and filter are generally 2D: To avoid overloading the symbols x and y, we will refer to the image columns and rows as i and j. Thus the image is *P(i, j)*. The formula for 2D convolution of an *NxN* filter *w(i,j)* with an image is:

$$w * P(i,j) = \sum_{v=1}^{N}\sum_{u=1}^{N} w(u,v)P(i-u, j-v)$$

The value at each position *i, j* is the sum of the product of a filter (kernel, or receptive field) *w(u,v)* with a neighborhood of the image placed at *i,j*. Note that a 2D convolution can easily be re-expressed as a 1D convolution by mapping successive rows of the *NxN* filter *w(u,v)* into 1 long column with N^2 coefficients, *f(n)*, using: $n = (v-1)\cdot N + u$

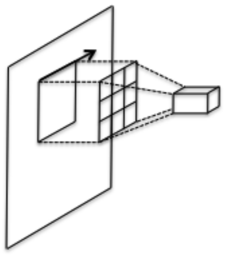
The use of *i–u* and *j–v* is rather than *i+u* and *j+v* is purely to assure equivalence with the classical signal processing operation of convolution. In convolution, the filter is “flipped” around the center pixel. In reality, many implementations simply use *i+u* and *j+v*. Technically, in signal processing, this would be called a “cross-correlation”.



$$a(i,j) = f(z(i,j)) = f\left(\sum_{u,v}^{N} w(u,v)P(i-u, j-v) + b_k\right)$$

Multiple Receptive Fields at each Layer

A second innovation was to learn multiple *NxN* receptive fields at each layer, as was observed by David Hubel and Torsten Wiesel and used in Computer Vision. The number of receptive fields is called the "depth" at that layer. We will use the symbol d from 1 to D as an index for the depth (number of receptive fields) at each level.



$$a_d(i,j) = f(z_d(i,j)) = f\left(\sum_{u,v} W_d(u,v)P(i-u, j-v) + b_d\right)$$

For each *NxN* window, the CNN will compute the product with a vector of *K* receptive fields, $W_k(u,v)$ with a bias b_k.

$$z_d = \sum_{u,v} W_d(u,v)X_{i,j}(u,v) + b_d = \sum_{u,v} W_d(u,v)P(i-u, j-v) + b_d$$

The weighted sum is then processed with a non-linear activation function, *f()*, typically a relu or sigmoid of the sum of the product.

$$a_k = f(z_k) = f\left(\sum_{u,v} W_k(u,v)X_{i,j}(u,v) + b_k\right)$$

Because a vector of activations $\vec{a}_d = \begin{pmatrix} a_1 \\ \vdots \\ a_D \end{pmatrix}$ is computed for each image position, this should properly be written as $a_d(i,j) = f(z_d) = f\left(\sum_{u,v} W_d(u,v)X_{i,j}(u,v) + b_d\right)$

The result is a "feature map" of d features at each position $a_d(i,j)$, with d values at each image position *(i,j)*.

The receptive fields, $W_k(u,v)$ can be learned using back-propagation, from a training set where each window is labeled with a target class, using an "indicator" image *y(i,j)*. For multiple target classes, the indicator image can be represented as a vector image, $\vec{y}(i,j)$. More classically, *y(i, j)* is a binary image with 1 at each location that contains the target class and 0 elsewhere.

CNN Hyper-parameters

CNNs are typically configured with a number of "hyper-parameters":

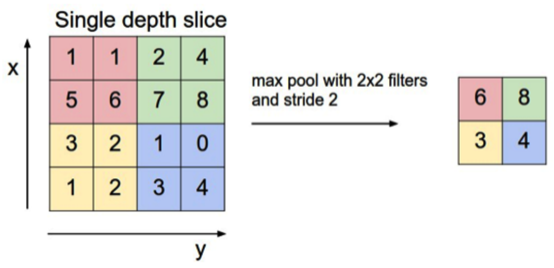
Spatial Extent: This is the size of the filter, NxN. Early networks followed computer vision theory and used 11x11 or 9x9 filters. Experimentation has shown that 3x3 filters can work well with multi-layer networks.

Depth: This is the number D of receptive fields for each position in the feature map. For a color image, the first layer depth at layer 0 would be D=3. If described with 32 image descriptors, the depth would be D=32 at layer 1. Some networks will use NxNxD receptive fields, including 1x1xD.

Stride: Stride is the step size, S, between window positions. By default it generally 1, but for larger windows, it is possible define larger step sizes.

Zero-Padding: Size of region at the border of the feature map that is filled with zeros in order to preserve the image size (typically N).

Pooling



Pooling is a form of down-sampling that partitions the image into non-overlapping regions and computes a representative value for each region. The feature map is partitioned into small non-overlapping rectangles, typically of size 2x2 or 4x4, and a single value it determined for each rectangle. The most common pooling operators are average and max. Median is also sometimes used. The earliest architectures used average, creating a form of multi-resolution pyramid. Max pooling was soon shown to work better.

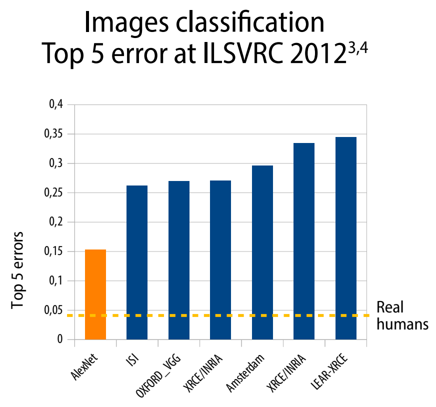
Classic CNN Architectures

The emergence of the internet and the world-wide web made it possible to assemble massively large data sets of training data, and to issue global challenges for computer vision techniques to compete on these challenges. Many of the most famous CNN architectures have been established by winning large scale image classification challenges. The parameters of the challenge often explain the choice of parameters for the network, such as the size of the input image and the number of output categories.

Several key data sets that have influenced the evolution of the domain. Many of the popular architectures were designed specifically to address research challenges based on these data sets. Most state-of-the-art object detection networks pre-train on ImageNet and then rely on transfer learning to adapt the learned recognition system to a specific domain.

ImageNet
ImageNet is an image database organized according to the nouns in the WordNet hierarchy. Each node of the WordNet hierarchy is depicted by hundreds of images in ImageNet. In 2006, Fei-Fei Li began working on the idea for ImageNet based on the word-database of WordNet, eventually using Amazon Mechanical Turk to help with the classification of images. The database was first presented as a poster at the 2009 Conference on Computer Vision and Pattern Recognition (CVPR) in Florida.

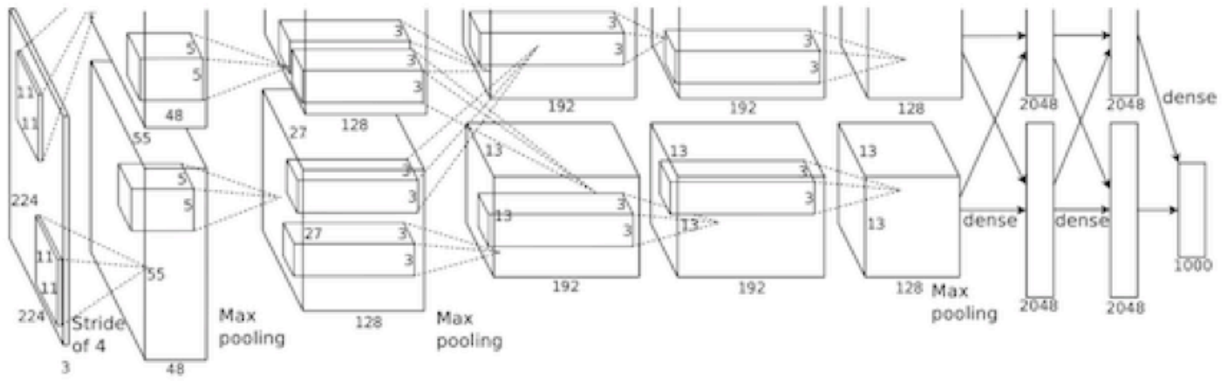
In 2010 Fei-Fei Li joined with the PASCAL VOC team to create a joint research challenge where research teams compete to achieve higher accuracy on several visual recognition tasks. The resulting annual competition is known as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The ILSVRC uses a "trimmed" list of only 1000 image categories or "classes", including 90 of the 120 dog breeds classified by the full ImageNet schema. In 2010 and 2011, a good score for the ILSVRC top-5 classification error rate was 25%.



Initial champions were statistical recognition techniques using techniques such as SIFT and HoG. However, in 2012, Alex Krizhevsky won the competition with a deep convolutional neural net based on LeNet5 called AlexNet. AlexNet achieved and error rate of 16% (accuracy of 84%). This dramatic quantitative improvement marked the start of the rapid shift to techniques based on Deep Learning using Neural Networks. By 2014, more than fifty institutions participated in the ILSVRC, almost exclusively with different forms of Network Architectures. In 2017, 29 of 38 competing teams demonstrated error rates less than 5% (better than 95% accuracy).

AlexNet

AlexNet, is a deeper and larger variation of LeNet5.



AlexNet Architecture (2010)

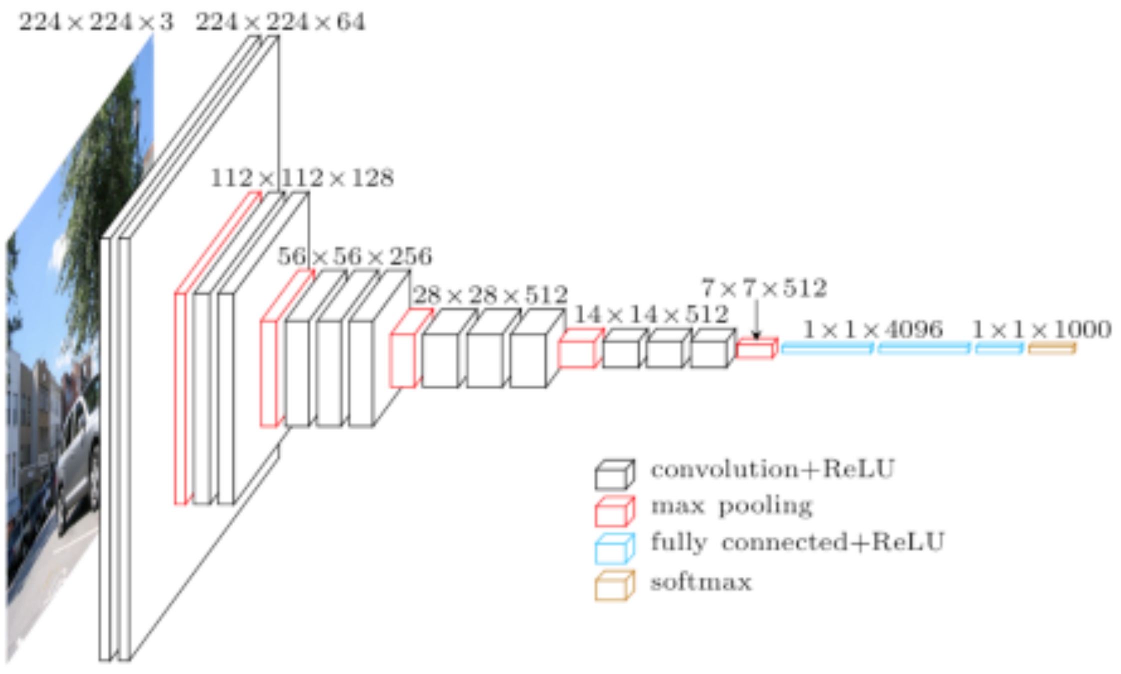
Innovations in AlexNet include:

1. The use of relu instead of sigmoid or tanh. Relus provided a 6 times speed up with the same accuracy, allowing more training.
2. A technique called "dropout" in which randomly chosen units are temporarily removed during learning. This regularizes the network preventing over-fitting to training data.
3. Overlap pooling, in which pooling is performed on overlapping windows.

The architecture is composed of 5 convolutional layers followed by 3 fully connected layers. Relu is used after each convolution and in each fully connected layer. The input image size of 224 x 224 is dictated by the number of layers in the architecture. Larger images are generally texture mapped to this size.

A good implementation can be found in PyTorch. The network has 62.3 million parameters, and needs 1.1 billion computations in a forward pass. The convolution layers account for 6% of all the parameters, and consume 95% of the computation. The network is commonly trained in 90 epochs, with a learning rate 0.01, momentum 0.9 and weight decay 0.0005. The learning rate is divided by 10 once the accuracy reaches a plateau.

VGG - Visual Geometry Group



The VGG Architecture (2014)

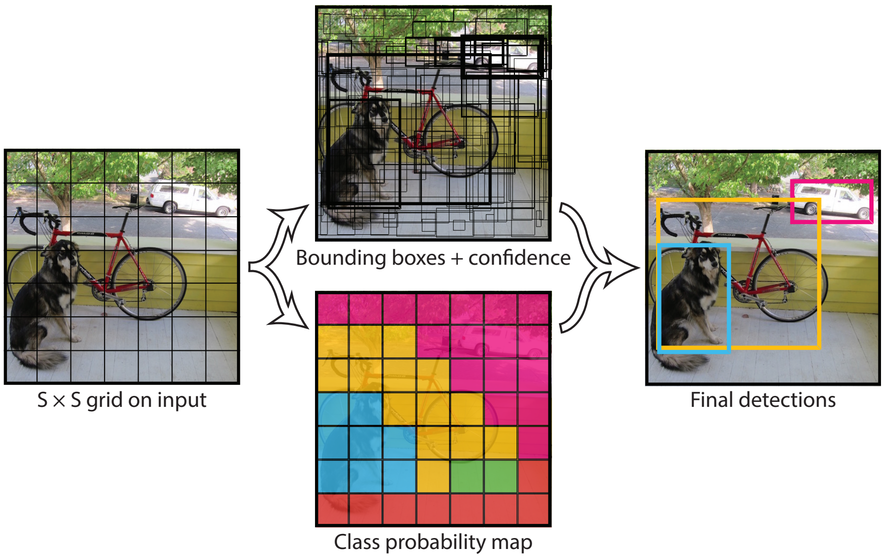
In 2014, Karen Simonyan and Andrew Zisserman of the Visual Geometry Group at the Univ of Oxford demonstrated a series of networks referred to as VGG. An important innovation was the use of very many small (3x3) convolutional receptive fields. The also introduced the idea of a 1x1 convolutional filter.

For a layer with a depth of D receptive fields, a 1x1 convolution performs a weighted sum of the D features, followed by non-linear activation. The weights can be learned with back-propagation.

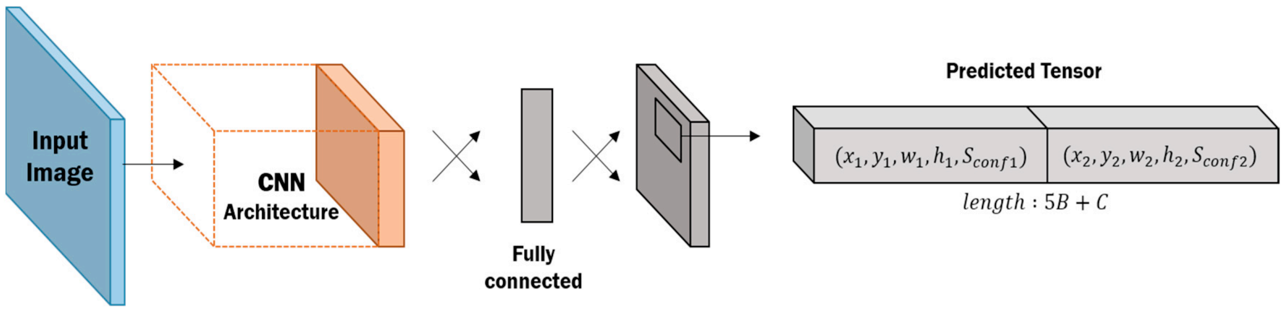
A stack of convolutional layers is followed by three Fully-Connected layers: the first two have 4096 channels each, the third performs classification and thus contains one channel for each class (1000 channels for ILSVRC). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks. All layers use Relu activation.

YOLO: You Only Look Once

YOLO poses object detection as a single regression problem that estimates bounding box coordinates and class probabilities at the same time directly from image pixels. A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for each box in a single evaluation. The result is a unified architecture for detection and classification that is very fast.



The input image is divided into an S x S grid of cells. Each grid cell predicts B bounding boxes as well as C class probabilities. The bounding box prediction has 5 components: (x, y, w, h, confidence).



(From Kim, J. and Cho, J. Exploring a Multimodal Mixture-Of-YOLOs Framework for Advanced Real-Time Object Detection. Applied Sciences, 2020, vol. 10, no 2, p. 612.)

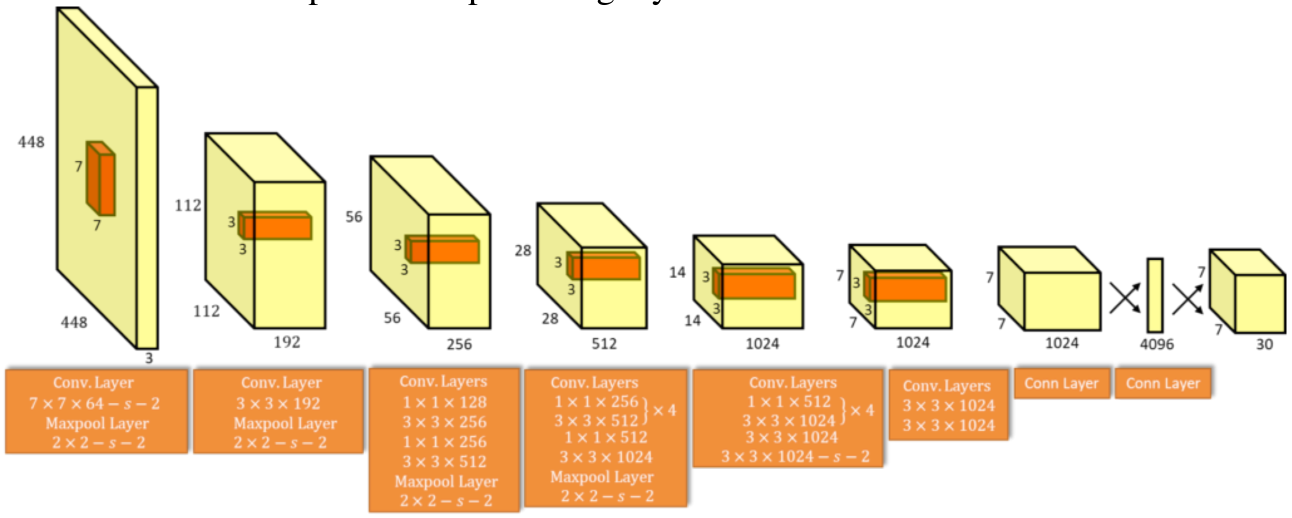
The (x, y) coordinates represent the center of the predicted bounding box, relative to the grid cell location. Width and height (w, h) are predicted relative to the entire image.

Both the (x, y) coordinates and the window size (w, h) are normalized to a range of [0,1]. Predictions for bounding boxes centered outside the range [0,1] are ignored. If the predicted object center (x, y) coordinates are not within the grid cell, then object is ignored by that cell. Each grid cell also predicts C class conditional probabilities $P(Class_i \mid Object)$

These are conditioned on the grid cell containing an object. Only one set of class probabilities are predicted per grid cell, regardless of the number of boxes.
These predictions are encoded as an S x S x (5B+C) tensor. Where SxS is the number of grid cells, B is the number of Bounding Boxes predicted and C is the number of image classes. For the Pascal visual Object Classification challenge, S = 7, B = 2 and C=20 yielding a 7x7x30 tensor.

These scores encode the probability of a member of class *i* appearing in a box, and how well the box fits the object. If no object exists in a cell, the confidence score should be zero. Otherwise the confidence score should equal the intersection over union (IOU) between the predicted box and the ground truth.

Yolo-1 was inspired by GoogleLeNet. The detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1 by 1 convolutional layers reduce the features space from preceding layers.

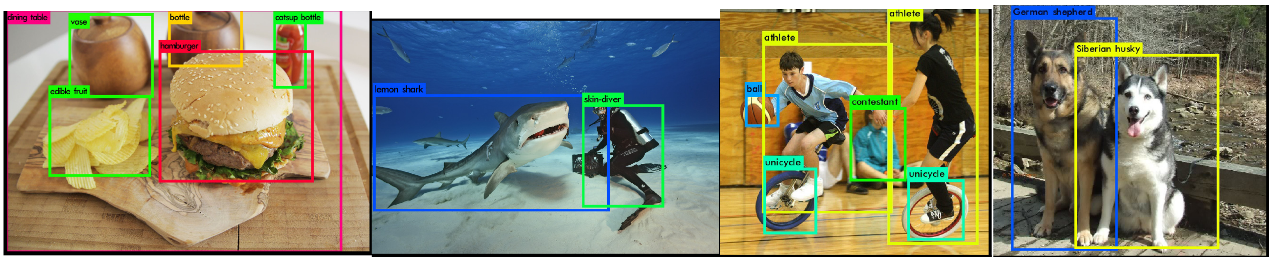

448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7×7×64−s−2 Maxpool Layer 2×2−s−2
Conv. Layer 3×3×192 Maxpool Layer 2×2−s−2
Conv. Layers 1×1×128 3×3×256 1×1×256 3×3×512 Maxpool Layer 2×2−s−2
Conv. Layers 1×1×256 3×3×512 ×4 1×1×512 3×3×1024 Maxpool Layer 2×2−s−2
Conv. Layers 1×1×512 3×3×1024 ×4 3×3×1024 3×3×1024−s−2
Conv. Layers 3×3×1024 3×3×1024
Conn Layer
Conn Layer

(from: http://datahacker.rs/how-to-peform-yolo-object-detection-using-keras/)

The convolutional layers were pretrained on the ImageNet data-set at half the resolution (224 by 224 input image). Image resolution was then doubled to (448 x 448) for detection.

Layer Name	Filters	Stride	Output Dimension
Conv 1	7 x 7 x 64	2	224 x 224 x 64
Max Pool 1	2 x 2	2	112 x 112 x 64
Conv 2	3 x 3 x 192	1	112 x 112 x 192
Max Pool 2	2 x 2	2	56 x 56 x 192
Conv 3	1 x 1 x 128	1	56 x 56 x 128
Conv 4	3 x 3 x 256	1	56 x 56 x 256
Conv 5	1 x 1 x 256	1	56 x 56 x 256
Conv 6	1 x 1 x 512	1	56 x 56 x 512
Max Pool 3	2 x 2	2	28 x 28 x 512
Conv 7	1 x 1 x 256	1	28 x 28 x 256
Conv 8	3 x 3 x 512	1	28 x 28 x 512
Conv 9	1 x 1 x 256	1	28 x 28 x 256
Conv 10	3 x 3 x 512	1	28 x 28 x 512
Conv 11	1 x 1 x 256	1	28 x 28 x 256
Conv 12	3 x 3 x 512	1	28 x 28 x 512
Conv 13	1 x 1 x 256	1	28 x 28 x 256
Conv 14	3 x 3 x 512	1	28 x 28 x 512
Conv 15	1 x 1 x 512	1	28 x 28 x 512
Conv 16	3 x 3 x 1024	1	28 x 28 x 1024
Max Pool 4	2 x 2	2	14 x 14 x 1024
Conv 17	1 x 1 x 512	1	14 x 14 x 512
Conv 18	3 x 3 x 1024	1	14 x 14 x 1024
Conv 19	1 x 1 x 512	1	14 x 14 x 512
Conv 20	3 x 3 x 1024	1	14 x 14 x 1024
Conv 21	3 x 3 x 1024	1	14 x 14 x 1024
Conv 22	3 x 3 x 1024	2	7 x 7 x 1024
Conv 23	3 x 3 x 1024	1	7 x 7 x 1024
Conv 24	3 x 3 x 1024	1	7 x 7 x 1024
Fully-Connected 1	-	-	4096
Fully-Connected 2	-	-	7 x 7 x 30 (1470)

YOLO-9000 (YOLOv2)



In 2016, the YOLO team published performance evaluation results and source code for a new version of YOLO referred to as Yolo-9000. Yolo-9000 employed a number of innovations, including ideas that had emerged in the machine learning literature the previous year. These included:

- Batch Normalization
- Higher Resolution Classifie
- Convolutional With Anchor Boxes.
- Dimension Clusters.
- Bounding boxes with dimension priors and location prediction.
- Fine-Grained Features
- Multi-Scale Training

At low resolutions YOLOv2 operates as a cheap, fairly accurate detector. At 288x288 it runs at more than 90 FPS. This makes it ideal for smaller GPUs, high framerate video, or multiple video streams. At high resolution the network is competitive with the state of the art giving 78.6 mAP on VOC 2007 while still operating above real-time speeds

Code and pre-trained models for Yolo-9000 are available on-line at http://pjreddie.com/yolo9000/. Additional incremental improvements have been provided for YOLOv3 and YOLOv4.

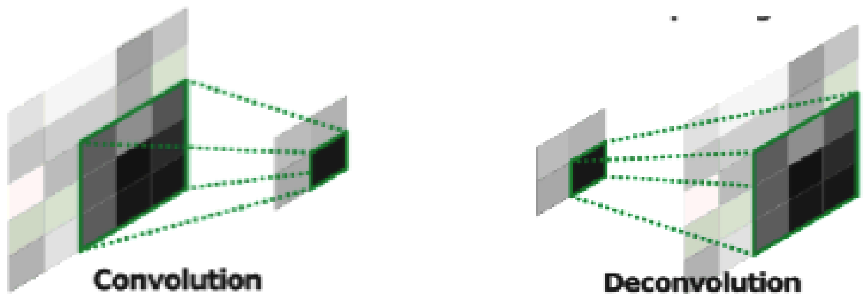
Generative Convolutional Networks

Generating images with deconvolution.

Just as it is possible to generate signals from codes using fully connected generative networks, it is possible to construct Generative Convolutional Networks for CNNs using an operation known as deconvolution.

Deconvolution is often used with convolutional networks to determine the location of a detected pattern in an image. Deconvolution provides a coarse pixel-wise label map that segments the image into regions corresponding to recognized classes and can be used for semantic segmentation.

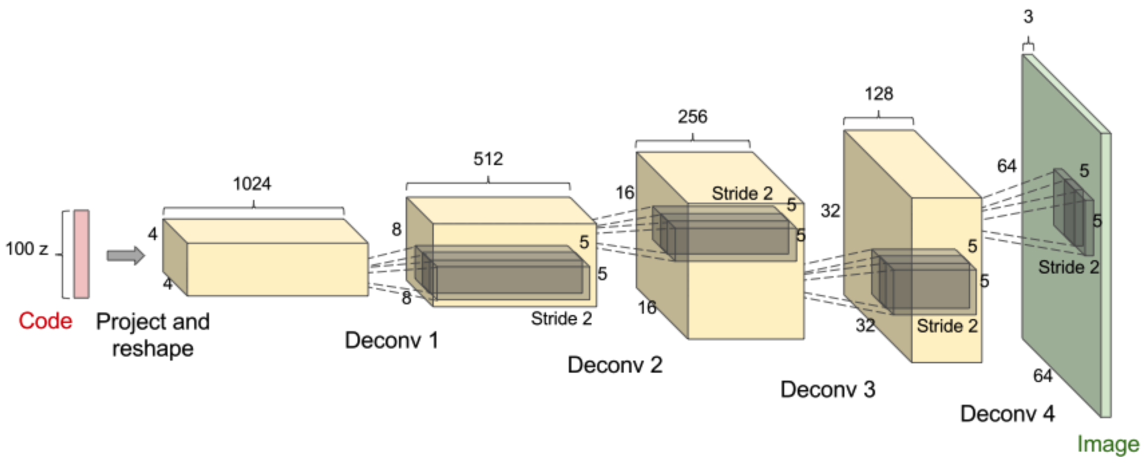
De-convolution treats the learned receptive fields as basis functions, and uses the activation at level *l* to create a weighted sum of bases at level *l+1*. The learned receptive fields are multiplied by the map of activation at level *l* to generate overlapping projections of receptive fields. These are then summed to create an image at level *l+1*. In some cases, the boundary is cropped to obtain an image at the target window size.



A stride greater than 1 can be used to create a larger image. The stride acts as the opposite of pooling. For 2x2 average pooling, de-convolution simply projects 4 displaced copies of the receptive field onto a 2 x 2 grid of overlapping receptive fields. These are then summed to give an image. An example of such a network is DCGAN architecture.

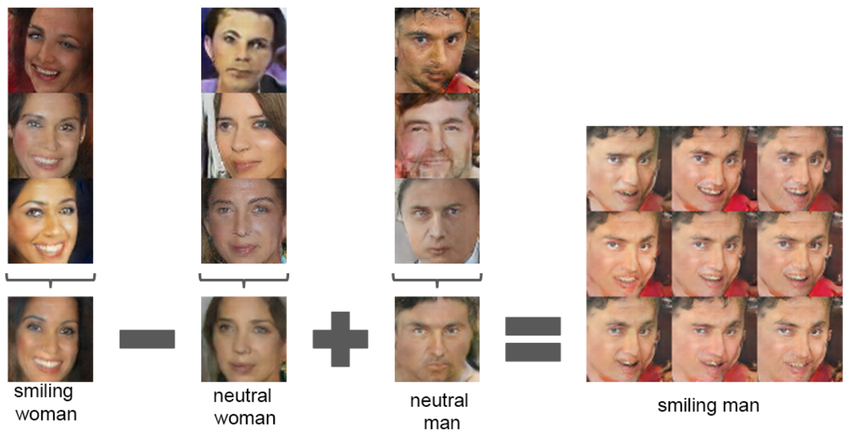
DCGAN

A DCGAN *(deep convolutional generative adversarial network)* takes 100 random numbers as an input (or code) and outputs an color image of size 64x64x3



The first fully connected layer is a 4 x 4 array of 1024 cells (Depth = 1024). Total number of cells is 16 K. This layer has 160 K weights and 16 K biases to train. This first layer is deconvolved into an 8 x 8 by 512 second layer, where deconvolution projects each of the cells in the 4x4 layer onto an overlapping set of 5x5 receptive field with a stride of 2. The process is repeated to create a 3rd layer that is 16x16x256 and then a 4th layer that is 32 x 32 by 128. The final output is a 5th layer with 64 x64 pixels of 3 colors.

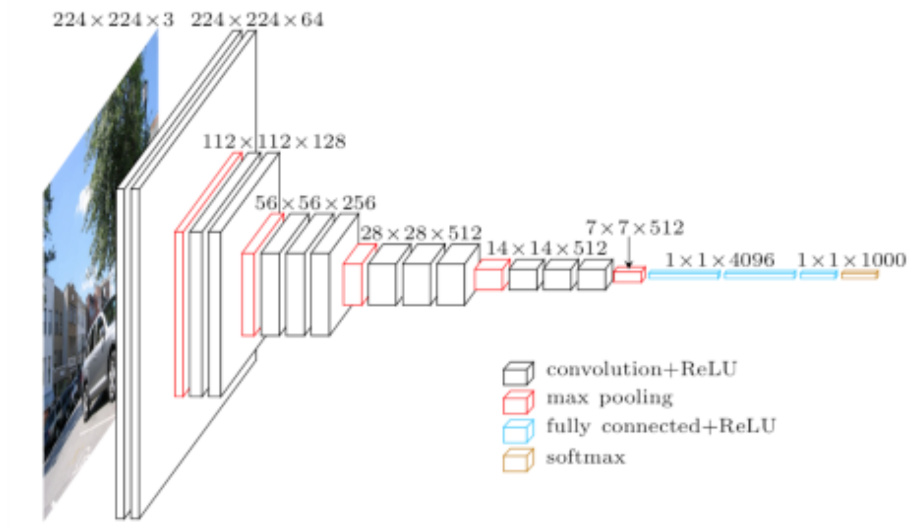
The following are some examples of images generated using DCGAN:



Example smiling man images generated from smiling woman images.

From:
Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks, ICLR 2016.

Deconvolution with VGG16



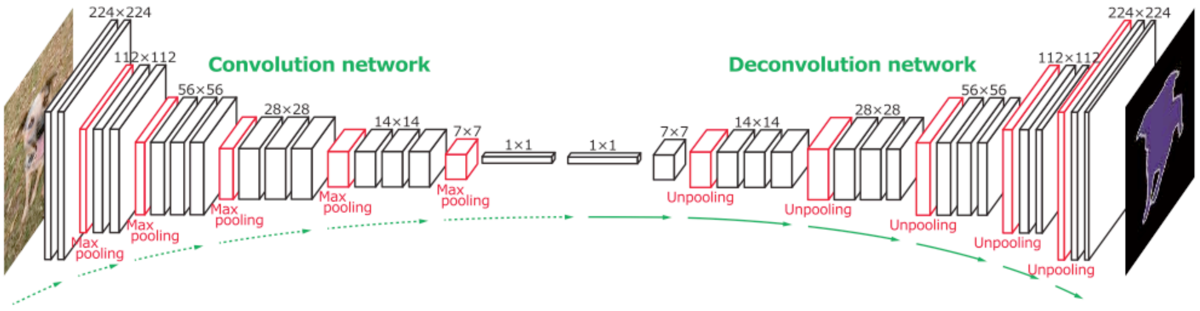
VGG16 is a convolutional neural network architecture proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. VGG16 scored 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes.

VGG16 improves on AlexNet by replacing large kernel-sized filters (11 x 11 and 5 x 5) with a cascade of 3×3 kernel-sized filter. VGG16 was trained for weeks and using NVIDIA Titan Black GPU’s.

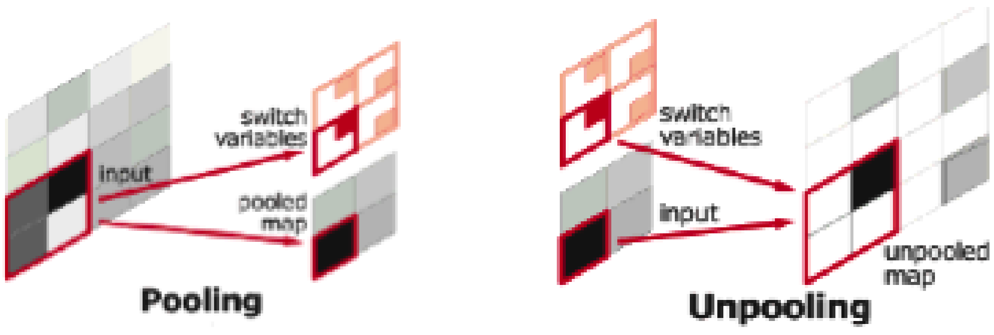
VGG accepts a 224 x 224 RGB image as input. The first 17 layers use 3x3 convolutions, relu and 2x2 max pooling with a stride of 2 after layers 2, 4, 7, 10 and 13. The depths are D=64 (layers 1, 2), D=128 (layers 3, 4), D=256 (layers 5, 6, 7). D=512 (layers 8 to 13). Layers 14 and 15 are a 1 x 1 convolution with depth 4096. Layer 16 is 1 x 1 x 1000 likelihood score for 1000 pretrained classes using softmax activation.

Three Fully-Connected (FC) layers follow a stack of 1x1 convolutional layers. The first two full-connected layers have 4096 channels each. The third layer has 1000 channels corresponding to the 1000 image classes corresponding to the 1000 image-net classes used in the ILSVRC (Image-net Large Scale Visual Recognition Classification) challenge for which it was designed. The final layer uses soft-max activation to determine the most likely classes in the 224 x 224 input image.

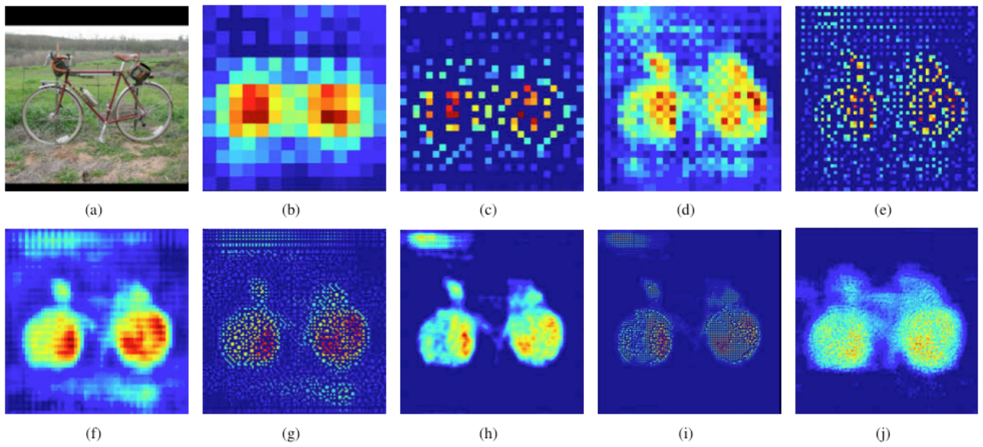
Normally VGG16 is used by scaling (texture mapping) the input image into a 224 by 224 window, without regard for the scale of the input, and produces only a probability for 1000 trained classes in the image. However, VGG16 can be adapted as a multiple object detector using deconvolution. The deconvolution network is a mirror image, replacing pooling with "un-pooling" and convolution with "deconvolution". This is often referred to as a U-net encoder-decoder.



VGG uses max pooling. With Max pooling, unpooling requires remembering which unit was selected for each pooling operation. This is done with a "switch Variable" that records the selected unit. The output is a larger sparse layer in which 3/4 of the activations are zero.

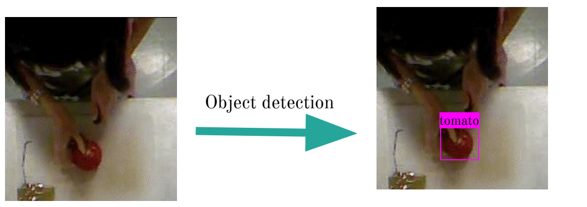


The following shows an example with deconvolution of the VGG net of a bicycle. (a) is the original image. The other images show the results of max-pooling for the 14x14, 28x28, 56x56, 112x112, and 224x224 layers

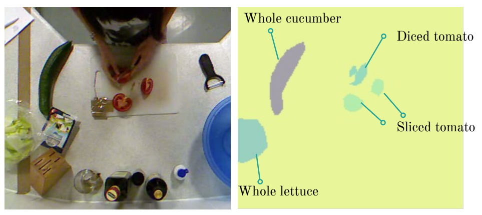


The output pixels can be used to provide scores for semantic segmentation for each pixel. Alternatively bounding boxes can be estimated by computing the 1st and 2nd moments (center of gravity and covariance), with a likelihood provided by the zeroth moment (sum of pixel class likelihoods) for each class.

For example, the following are multi-class object detection and semantic segmentation images obtained from deconvolution with VGG taken from Nachwa Aboubakr's thesis on observation of cooking activities. Her experiments use the 50 Salads data set.



Object **label** + object **location**

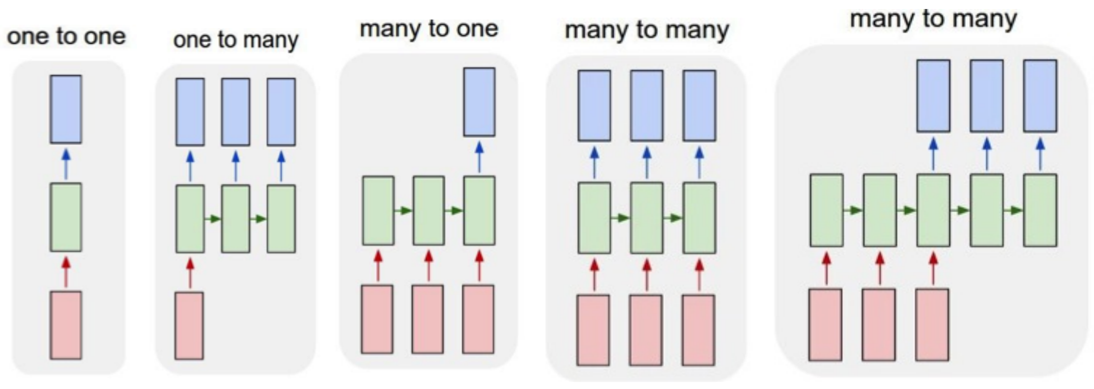


Example of pixel-wise labeling of a scene

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are used to discriminate and generate data that have an intrinsic order relation (sequences). Examples of sequences that may be discriminated and generated with RNNs include Speech, Music, Text, and Time Series data. RNNs can be combined with convolutional networks to recognize and generate video sequences of actions. RNNs have been traditionally used for natural language processing including for understanding written text and machine translation, although they are rapidly being replaced with Transformer using Self-Attention.

Recurrent Networks are Turing Universal, which means that any function that can be computed by a Turing machine can be computed by a recurrent network.



Copied from Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks", http://karpathy.github.io/2015/05/21/rnn-effectiveness/

History

In the early days of neural networks (1980's), a frequent criticism was that networks have no memory, other than the parameter learning. It was said that because networks did not maintain temporal state, they could not be suitable for tasks involving temporal or spatial sequences.

In the late 1980s, Rumelhart addressed this question by building on a class of completely connected networks proposed by Hopfield, leading to the idea of "unfolding" the network over time. Such networks are now called recurrent neural networks.

A recurrent neural network (RNN) is a neural network where connections between nodes form a directed graph along a temporal sequence. This enables the network to exhibit temporal dynamic behavior. RNNs can use internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as handwriting recognition or speech recognition.

Finite vs Infinite impulse networks

The term “recurrent neural network” refers to two broad classes of networks finite impulse and infinite impulse. Both classes exhibit temporal dynamic behavior.

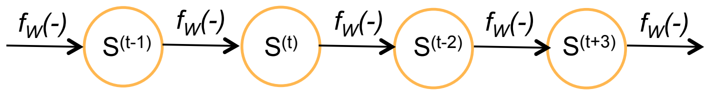
Finite Impulse: A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feed-forward neural network. The temporal dynamics are similar to a Finite Impulse Response (FIR) digital filter. In digital signal processing, FIR filters are known to be easy to design, stable, but limited in the duration of their response. With convolutional networks these are called 3D, and are a natural extension of convolutional networks. However, extending the temporal scale by more than a few frames makes learning impossible because Gradients become too small. This is called the Vanishing Gradient Problem.

Infinite impulse: An infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled because of internal feedback. These have similar temporal dynamics to Infinite Impulse Response (IIR) digital filters. In digital signal processing, IIR filters are known to be difficult to design, unstable, but very powerful and efficient. The classic Infinite Impulse Recurrent network is the LSTM (Long-Short-Term Memory) architecture.

Both finite impulse and infinite impulse recurrent networks can have additional states, and storage can be under direct control of the network. The storage can also be replaced by another network or graph. Such controlled states are referred to as gated states or gated memory, and are a key part of gated recurrent units including long short-term memory (LSTMs) networks.

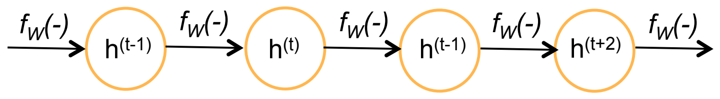
Recurrent Networks

The classic model for a dynamic process is a function, $f(-)$, that predicts the state, *s(t)* of a system at time *t*, from the state *s(t-1)* at time *t-1*, using parameters $\vec{w}$. Such as process is known as a "markov" process.



$$s^{(t)} = f_{\vec{w}}(s^{(t-1)})$$

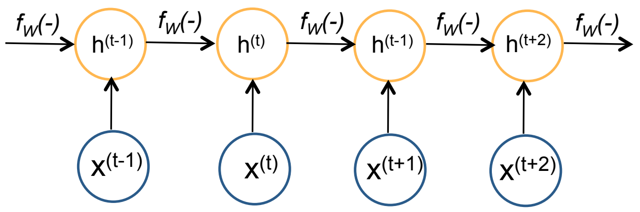
In the case of a recurrent network, the "state" is the activation (or vector of activations) of one or more "hidden" units. In previous lectures we represented the activation state of a cell with the symbol *a*. In the recurrent network literature, activation is generally represented with a state variable $h^{(t)}$



$$h^{(t)} = f_{\vec{w}}(h^{(t-1)})$$

The time variable is traditionally represented with a superscript, to keep it apart from the unit indices at each level.

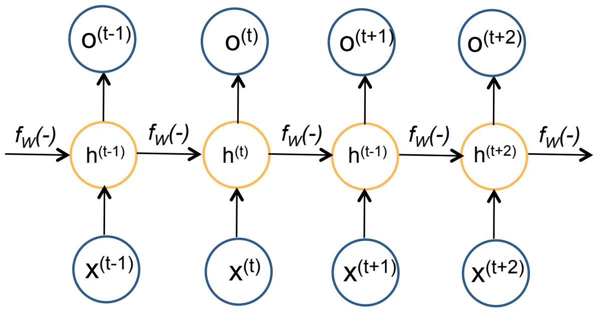
We can model the effects of an external input by adding an additional term, $x^{(t)}$, to the temporal transition function.



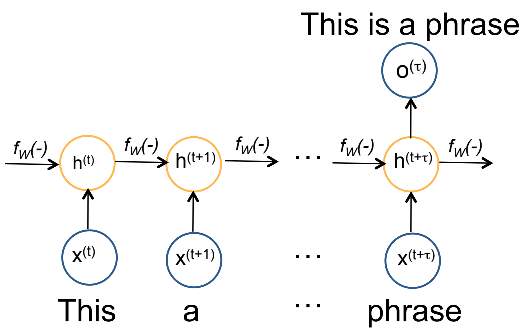
$$h^{(t)} = f_{\vec{w}}(h^{(t-1)}, x^{(t)})$$

The temporal duration of the network is typically represented the variable τ, so that the network is said to operate on a temporal sequence $x^{(t)}$ from $t=1$ to τ.

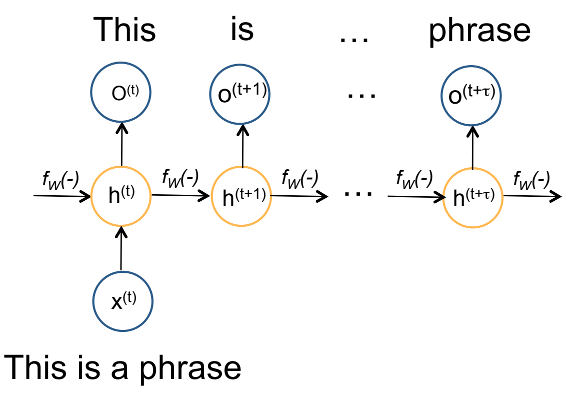
Normally, the network generates an output represented by an output variable, $o^{(t)}$.



For example, in a many-to-one network, the network would produce a single output after τ time steps. For example, the following network assembles the words "This", "is", "a", and "phrase", into a single output "This is a phrase". In this case, t is the number of words in the phrase, 4.

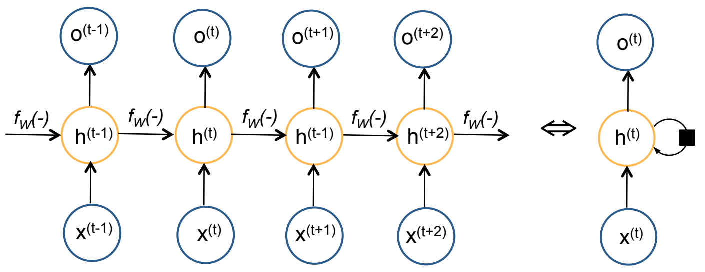


A one-to-many network would produce a sequence of τ outputs from a single input. For example, a single symbol for "This is a phrase" can be expanded into a sequence of outputs, where $\tau = 4$.

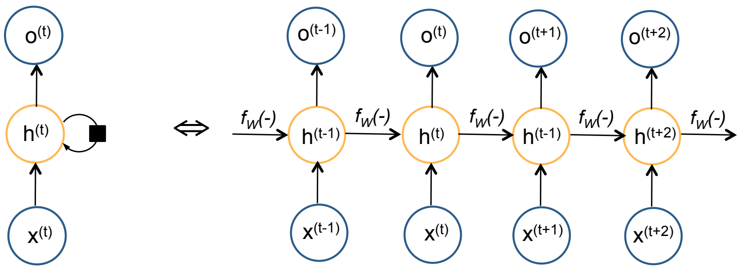


Folding and Unfolding

Recurrent networks are classically "folded" into a recurrent structure:



Where the black square represents a time delay of 1 time unit. The recurrent structure can be unfolded to see the network as a 2-D structure.



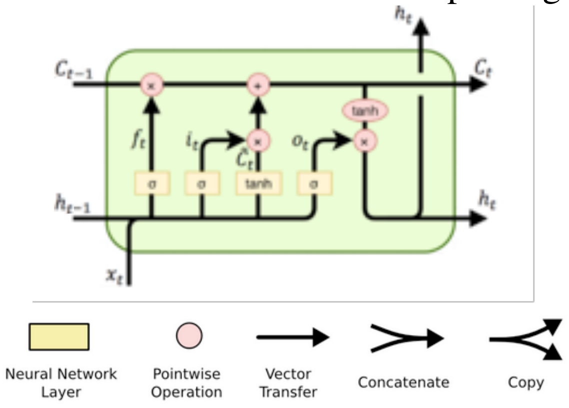
Long Short-Term Memory (LSTM)

In theory, RNNs can keep track of arbitrary long-term dependencies in an input sequences. However, this generally proves impractical because of a problem known as the "vanishing gradient" problem. When training a normal RNN using back-propagation, the gradients which are back-propagated can tend to zero (vanish) or diverge to infinity (explode), because of the accumulation of errors resulting from computation with finite-precision numbers. Long short-term memory (LSTM) provide a solution to this problem.

A long short-term memory (LSTM) is a form of RNN with a recursive memory structure. LSTM are appropriate for long temporal sequences of such as speech or video, and have been used to build systems for unsegmented, connected handwriting recognition, speech recognition and anomaly detection in network traffic.

LSTMs use feedback connections to enable design of a compact, powerful structure that can represent an arbitrarily long temporal duration, but can easily result in instability. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. LSTM partially solve the vanishing gradient problem, because LSTM units allow gradients to also flow unchanged. However, LSTM networks can still suffer from the exploding gradient problem.



LSTM with a forget gate Copied from: Understanding LSTM Networks - Christopher Olah (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

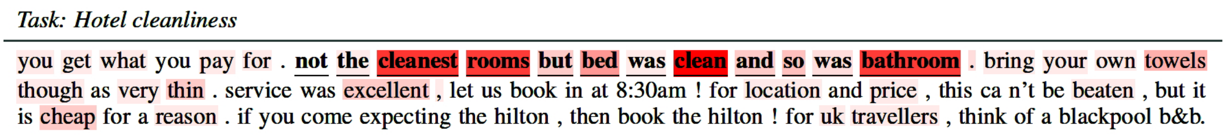
In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent point-wise operations, like vector addition, while the yellow boxes are learned neural network layers. Merging lines denote concatenation, while a forking line denotes copies of the vector going to different locations.

Attention is All You Need: Transformers

Attention is a form of filter that suppresses unnecessary tokens allowing the system to associate relevant tokens. Attention has long been studied in Computer vision as mechanism to focus processing on the relevant parts of scene. Human vision system is known to make extensive use of top-down attention processes to suppress irrelevant part of the visual environment and limit recognition to the most salient or the most relevant phenomena. Over the years, many researchers have proposed ideas for using Salience and a-priori knowledge to highlight important phenomena.

In 2010, Hinton proposed that attention could be used to a mechanism to explain the processing of deep networks. The idea was to reconstruct the parts of the input signal that contribute to the output of a recognition network. We saw an example of this with generative convolutional networks using VGG.

This idea was rapidly adopted in Natural Language processing in order to associate relevant words in a sentence through word highlighting.



Task: Hotel cleanliness

you get what you pay for . **not the cleanest rooms but bed was clean and so was bathroom** . bring your own towels though as very thin . service was excellent , let us book in at 8:30am ! for location and price , this ca n't be beaten , but it is cheap for a reason . if you come expecting the hilton , then book the hilton ! for uk travellers , think of a blackpool b&b.

Taken from from Galassi, A., Lippi, M., and Torroni, P. Attention in natural language processing. IEEE Transactions on Neural Networks and Learning Systems., 2020.

In 2017, a revolutionary paper by Vaswani et al from Google showed that the deep convolutional and recurrent networks using layers of could be completely replaced with attention.

Attention allows a network to individually focus on specific elements of a complex input. The goal is to break down complicated tasks into smaller areas of attention that are processed sequentially. Attention enhances the important parts of the input data and fades out the rest, allowing the network to devote more power on a small but important part of the data. Which part of the data is more important depends on the context and is learned through training data by gradient descent.

Attention is typically implemented as a function that maps a query and a set of key value pairs to an output. The attention function compares a query to a set of keys for possible targets. The keys provide a form of address (hash code) for targets. When a **query** matches a **key**, a **value** is generated to indicate the relevance of each target to the query.

The Query, Q, is encoded as a form of code, similar to a hash code. Each target record is identified with a key, K, in the same code. A dot product of Q and K indicates the degree that the record corresponds to the Query. This product is fed to a Softmax, to obtain a probability distribution for the set of keys.

$$V_n = soft\max(Q^T K_n) = \frac{e^{Q^T K}}{\sum_{n=1}^{N} e^{Q^T K_n}}$$

The resulting attention score is a form of filter that suppresses unnecessary tokens allowing the system to associate relevant tokens for subsequent encoding.

Attention can be implemented by adding an attention index indicating the relevance of each word token to a query (additive attention), or by multiply the amplitude of a each component (dot-product or multiplicative attention) according to relevance to a query.

Additive Attention

Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. Here is an example taken where the relevance of the components of from an input are modified by addition of an attention score and the sum is normalized to a probability distribution by soft-max.

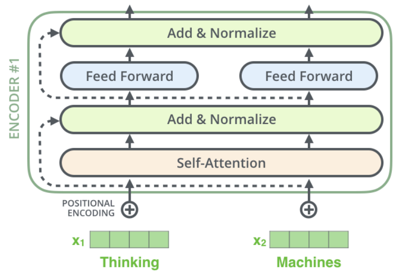
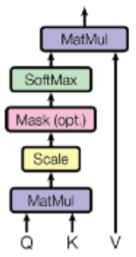


Image from Jay Alammar, The Illustrated Transformer (http://jalammar.github.io/illustrated-transformer/)

Dot Product Attention

Dot product attention is used to determine the components of a vector, V, that are most relevant to a query vector, Q using a key vector, K, for each element of the value. Queries and keys of dimension d_k, and values of dimension d_v. A key vector is computed for each token of the input vector. The product is then normalized and used to modulate a value vector to emphasize relevant components. The weight assigned to each value expressed by a compatibility function of the query with the corresponding key value. The result is used to select the most relevant parts of an input for recognition.



From: Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L and Polosukhin, I. (2017). Attention is all you need.

$$h_i = attention(Q_i, K_j, V_j) = \sum_{j=1}^{N} a_{ij} V_j \quad \text{where} \quad a_{ij} = soft\max\left(\frac{Q_i^T K_j}{\sqrt{d_K}}\right) = \frac{e^{\frac{Q_i^T K_j}{\sqrt{d_K}}}}{\sum_{n=1}^{N} e^{\frac{Q_i^T K_n}{\sqrt{d_K}}}}$$

Note that the product of Q and K are scaled by the square root of $d_{k.}$ Dividing the produce of Q and K by the square root of d_k compensates for the extremely small gradients that can arise from large vector lengths for Q and K.

While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

Transformers

A Transformer is a transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution. Most modern sequence transduction models have an encoder-decoder structure, in which the encoder maps an input sequence of symbol representations to a sequence of continuous representations. The Transformer follows a similar approach, using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder to replace recurrent and convolutional layers used in early architectures.

Self-attention is an attention mechanism relating different positions of a single sequence to other positions in order to compute a representation of the same sequence. Self-attention model is an auto-regressive model, consuming the previously generated symbols as additional input when generating the next. Self-attention allows each word in a sentence or paragraph to look at other words to better know which word contributes to the current word. Words have different meaning. Self-attention captures the meaning by encoding context words that establish the meaning.

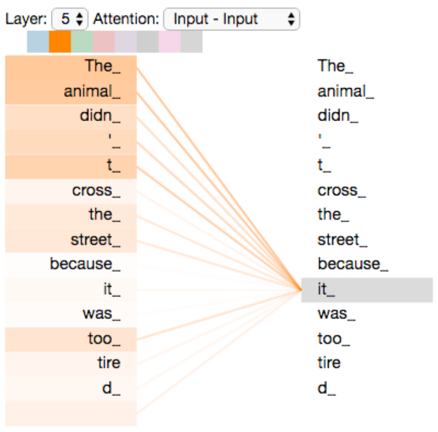


Image from Jay Alammar, The Illustrated Transformer
(http://jalammar.github.io/illustrated-transformer/)

Transformers consist of multiple layers where each layer contains multiple attention heads.

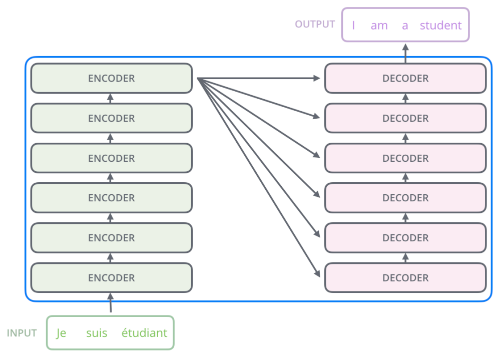


Image from Jay Alammar, The Illustrated Transformer
(http://jalammar.github.io/illustrated-transformer/)

Each encoder is composed of a parallel set of attention heads for self-attention, followed by a linear transformation that maps the selected tokens to a set of latent variables.

The Encoder

The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position- wise fully connected feed-forward network.

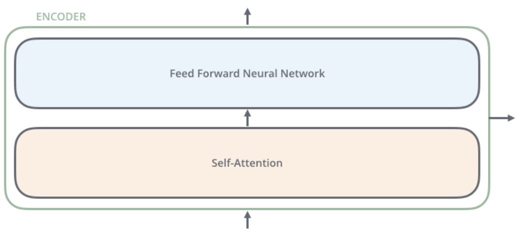


Image from Jay Alammar, The Illustrated Transformer
(http://jalammar.github.io/illustrated-transformer/)

The Decoder

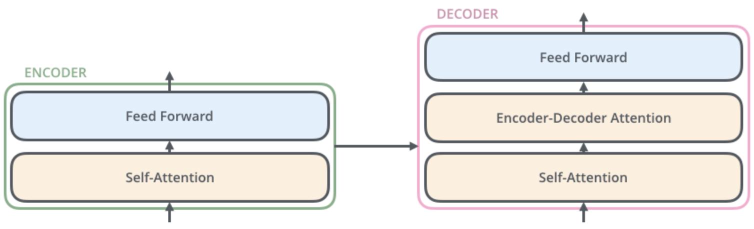


Image from Jay Alammar, The Illustrated Transformer
(http://jalammar.github.io/illustrated-transformer/)

In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections are made around each of the sub-layers followed by layer normalization.

BERT - Bidirectional Transformers

Bidirectional Encoder Representations from Transformers (BERT) is a Transformer-based machine learning technique for natural language processing (NLP) pre-training developed by Google. BERT was created and published in 2018 by Jacob Devlin and his colleagues from Google. BERT was trained by self-supervised learning using unlabeled data extracted from English Wikipedia with 2,500M words the BooksCorpus with 800M words. The source code for a trained version of BERT may be found at https://github.com/google-research/bert

BERT is pre-trained on 3.3 billion tokens of English text to perform two tasks: Mask Language Model (MLM) and Next Sentence Prediction (NSP). In the MLM task, the model predicts the identities of words that have been masked-out of the input text. In the NSP task, the model predicts whether the second half of the input follows the first half of the input in the corpus, or is a random paragraph. Further training the model on supervised data results in impressive performance across a variety of tasks ranging from sentiment analysis to question answering.

The BERT architecture is easily extended to multimodal perception and interaction by simple concatenation of encodings of different modalities. Each layer uses multiple Self-Attention Heads to associate multiple mutually relevant entities to be interpreted at that next level. Thus BERT can be trained to complete missing data with multiple modalities, or to predict an appropriate reaction in all modalities for a stimulus in one or more modalities.

Programming Environments For Machine Learning

Research in Machine Learning is essentially Empirical. Most research in Machine Learning is performed in the interactive Python environment in response to public research challenges using publically available data sets published along with the research challenge. Researchers are expected to publish their code so that others can compare results. This style of research has enabled extremely rapid progress at the expense of ever-growing requirements for computing power and data. Currently, the computing power required for state of the art research is doubling every 3 to 4 months (5 times faster than Moore's law!).

Python

Python is an interpreted, high-level programming language that is widely used in machine learning research. Python was created in the late 1980s by Guido van Rossum at the CWI research center in the Netherlands as a language that emphasizes code readability. Its language constructs and object-oriented approach are intended to help programmers write clear, logical code for small and large-scale projects. Python is ideal for rapid protyping of software. Python 3.0 was released in 2008 and was a major revision of the language that is not completely backward-compatible with Python 2.

Python uses whitespace indentation, rather than curly brackets or keywords, to delimit blocks. An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block. Thus, the program's visual structure accurately represents the program's semantic structure. This feature is sometimes termed the off-side rule, which some other languages share, but in most languages indentation does not have semantic meaning. You can find many on-line tutorials and MOOCs on the web.
For example, https://www.python.org/about/gettingstarted/

Conda Python

Conda is an open source environment and package management system that runs on Windows, Apple macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda can easily be used to create, save, load and switch between environments on your a computer. It was created for Python programs, but can package and distribute software for any language including C and HTML.

As a package manager, conda makes it easy to find and install packages. If you need a package that requires a different version of Python, you do not need to switch to a different environment manager, because conda is also an environment manager. With just a few commands, you can set up a totally separate environment to run a different version of Python, while continuing to run your usual version of Python in your normal environment.

In its default configuration, conda can install and manage the thousands of packages available at repo.anaconda.com that are built, reviewed and maintained by Anaconda. Anaconda install packages are available for Linux, Apple MacOS, or MS Windows. Installer packages for full anaconda can be found at https://www.anaconda.com/download/
We recommend a simple minimal version referred to as miniconda. The installer packages for miniconda are at (https://conda.io/miniconda.html)

Jupyter Notebooks.
Jupyter notebooks (http://jupyter.org/) are widely used for collaborative machine learning. A Jupyter Notebook is an open-source web application that allows creation and sharing of documents that contain live code, equations, visualizations, HTML markups and narrative text. Jupyter notebooks provide a browser-based tool for interactive authoring of documents that may combine explanatory text, mathematics, computations and their rich media output.

Jupyter notebooks provide:
- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

To install Jupyter Notebooks with miniconda, type:

```
$ conda install jupyter notebook
```

Keras Example of a network to recognize handwritten digits

The MNIST (Modified National Institute of Standards and Technology) database is a large collection of handwritten digits. The MNIST database contains 60,000 training images and 10,000 testing images. The database was created by "re-mixing" samples of digits from NIST's original datasets taken from American Census Bureau employees and American high school students. The black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced gray-scale levels.

The following is an example of a 2-layer fully connected neural network to classify MNIST digits written using Keras and Pytorch. The first layer has 784 units using RELU activation and the second layer is composed of 10 units using Softmax activation. We train this network using the MNIST training data with Categorical Cross Entropy and an Adam Optimizer. We then print the accuracy and loss for the resulting network using the MINST test data.

```
# An example of a 2 layer network for MNIST digits
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

# Define a 3 layer fully connected model with 2 layers of 784 units using relu
# and a final layer of 10 units using softmax

model = Sequential([
   Dense(28*28, input_shape=(28*28,), activation='relu'),
   Dense(10, activation='softmax')
])

# Compile the model with categorical_crossentropy, adam optimizer using accuracy
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# load MNIST Training and Test sets
(trainingSet, trainingLabel), (testSet, testLabel) = mnist.load_data()

# Keras digits are 28 by 28 pixels with 8 bits per pixel
# Flatten the training and test data arrays to 1-D
# and normalize grayscale pixels to values of 0 to 1.

trainingSet = trainingSet.reshape((60000, 28 * 28))
trainingSet = trainingSet.astype('float32') / 255
testSet = testSet.reshape((10000, 28 * 28))
testSet = testSet.astype('float32') / 255
```

```
# map the training and test labels from integers to one-hot coding
trainingLabel = to_categorical(trainingLabel)
testLabel = to_categorical(testLabel)

# Train for 5 epochs using a batch size of 128
model.fit(trainingSet, trainingLabel, epochs=5, batch_size=128)

#evaluate the model with the test data and print the results
test_loss, test_acc = model.evaluate(testSet, testLabel)
print('test_loss', "%.4f" % test_loss, ' - test_accuracy:', "%.4f" % test_acc)
```

```
Epoch 1/5
60000/60000 [====================] - 6s 97us/step - loss: 0.2475 - accuracy: 0.9305
Epoch 2/5
60000/60000 [====================] - 6s 95us/step - loss: 0.0973 - accuracy: 0.9709
Epoch 3/5
60000/60000 [====================] - 6s 101us/step - loss: 0.0621 - accuracy: 0.9811
Epoch 4/5
60000/60000 [====================] - 6s 103us/step - loss: 0.0438 - accuracy: 0.9868
Epoch 5/5
60000/60000 [====================] - 6s 104us/step - loss: 0.0313 - accuracy: 0.9908
10000/10000 [====================] - 1s 83us/step
test_loss 0.0602 -  test_accuracy: 0.9803
```

A Keras example of a simple CNN

The following is a simple Keras example of to detect MNIST digits provided by Frank Cholet of Google. This example processes 28x28 pixel imagettes with a convolutional layer of 32 3x3 filters using relu, followed by 2x2 max pooling, a convolutional layer of 64 3x3 filters, using relu, followed by 2x2 max pooling, a flatten layer, dropout of 0.5 and a fully connected layer.

```
model = keras.Sequential(
   [
      keras.Input(shape=input_shape),
      layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
      layers.MaxPooling2D(pool_size=(2, 2)),
      layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
      layers.MaxPooling2D(pool_size=(2, 2)),
      layers.Flatten(),
      layers.Dropout(0.5),
      layers.Dense(num_classes, activation="softmax"),
   ]
)
```

```
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
```

```
_________________________________________________________________
dropout (Dropout)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 10)                16010
```

Performance Evaluation for Pattern Classification

Pattern Recognition is the process of assigning observations to categories.
An observation is sometimes called an "entity" or "data" depending on the context and domain.

Our problem is to build a function, called a recognizer or classifier, $D(\vec{X})$, that maps the observation, $\vec{X}$ into a statement that the observation belongs to a class $\hat{C}_k$ from a set of K possible classes. $D(\vec{X}) \rightarrow \hat{C}_k$

$$\begin{tikzcd} \vec{X} \arrow[r] & \boxed{D(\vec{X})} \arrow[r] & \hat{C}_k \end{tikzcd}$$

In most classic techniques, the class $\hat{C}_k$ is from a set of K known classes $\{C_k\}$.
For a 2-class detection function, there are K=2 classes : *k=1* is a positive detection P, *k=2* is a negative detection, N. Thus $C_k \in \{P,N\}$

Almost all current classification techniques require the number of classes, K, to be fixed. ($\{C_k\}$ is a closed set). An interesting research problem is how to design classification algorithms that allow $\{C_k\}$ to be an open set that grows with experience.

In most learning algorithms, we partition the training data in distinct sets in order to estimate the recognizer and to evaluate the results.
A FUNDAMENTAL RULE in machine learning:

NEVER LEARN AND TEST WITH THE SAME DATA !

A typical approach is to use cross validation (also known as rotation estimation) in learning. Cross validation partitions the training data into N folds (or complementary subsets). A subset of the folds are used to train the classifier, and the result is tested on the other folds. A taxonomy of common techniques include:

- Exhaustive cross-validation
  - Leave p-out cross-validation
  - Leave one-out cross-validation
- Non-exhaustive cross-validation
  - k-fold cross-validation
  - 2-fold cross-validation
  - Repeated sub-sampling validation

When training neural networks, we will generally divide the data into three sets:

Training Set - used to train the discriminant functions
Evaluation Set - Used to monitor learning and avoid overfitting to the training set
Test Set - used to evaluate the final result and compare different techniques and architectures.

Two-Class Pattern Detectors

A pattern detector is a classifier or recognizer with K=2.
Class k=1: The target pattern, also known as P or positive
Class k=2: Everything else, also known as N or negative.

Pattern detectors are used in computer vision, for example to detect faces, road signs, publicity logos, or other patterns of interest. They are also used in signal communications, data mining and many other domains.

The pattern detector is learned as a discriminant function $g(\vec{X})$ followed by a decision rule, *d()*. For K=2 this can be reduced to a single function, as

$g_1(\vec{X}) \geq g_2(\vec{X})$ is equivalent to $g(\vec{X}) = g_1(\vec{X}) - g_2(\vec{X}) \geq 0$

A "threshold" value, B, can be used to bias the detector.

The detection function is learned from a set of training data composed of *M* sample observations $\{\vec{X}_m\}$ where each sample observation is labeled with an indicator variable $\{y_m\}$
y_m = P or Positive for examples of the target pattern (class k=1)
y_m = N or Negative for all other examples (class k=2)

Observations for which $g(\vec{X}) + B \geq 0$ are estimated to be members of the target class. This will be called POSITIVE or P.

Observations for which $g(\vec{X}) + B < 0$ are estimated to be members of the background. This will be called NEGATIVE or N.

We can encode this as a decision function to define our detection function $R(\vec{X}_m)$

$$D(\vec{X}) = d(g(\vec{X})) = \begin{cases} P & \text{if } g(\vec{X}) + B \geq 0 \\ N & \text{if } g(\vec{X}) + B < 0 \end{cases}$$

For training we need ground truth (annotation). For each training sample the annotation or ground truth tells us the real class y_m

$$y_m = \begin{cases} P & \vec{X}_m \in \text{Target - Class} \\ \text{N} & \text{otherwise} \end{cases}$$

The Classification can be TRUE or FALSE.

if $D(\vec{X}_m) = y_m$ then T else F

This gives
$D(\vec{X}_m) = y_m$ AND $D(\vec{X}_m) = P$ is a TRUE POSITIVE or TP
$D(\vec{X}_m) \neq y_m$ AND $D(\vec{X}_m) = P$ is a FALSE POSITIVE or FP
$D(\vec{X}_m) \neq y_m$ AND $D(\vec{X}_m) = N$ is a FALSE NEGATIVE or FN
$D(\vec{X}_m) = y_m$ AND $D(\vec{X}_m) = N$ is a TRUE NEGATIVE or TN

To better understand the detector we need a tool to explore the trade-off between making false detections (false positives) and missed detections (false negatives). The Receiver Operating Characteristic (ROC) provides such a tool

Performance Metrics for 2 Class Detectors

A number of performance metrics are commonly used to compare 2-class classifiers. These can be extended to multi-class detectors by using "one vs many".
That is, the detector for each class, C_k, is evaluated individually by labeling C_k as the target or Positive (P) class and all other classes as the non-target or Negative (N) class.

ROC Curves

Two-class classifiers have long been used for signal detection problems in communications and have been used to demonstrate optimality for signal detection methods. The quality metric that is used is the Receiver Operating Characteristic (ROC) curve. This curve can be used to describe or compare any method for signal or pattern detection.

The ROC curve is generated by adding a variable Bias term to a discriminant function.

$$D(\vec{X}) = d(g(\vec{X}) + B)$$

and plotting the rate of true positive detection vs false positive detection.

As the bias term, B, is swept through a range of values, it changes the ratio of true positive detection to false positives.

When $B << 0$ all detections will be Negative.
When $B >> 0$ all detections will be Positive.
For some range of values of B, $D(\vec{X})$ will give a mix of TP, TN, FP and FN.

The bias term, B, can act as an adjustable gain that sets the sensitivity of the detector. The bias term allows us to trade False Positives for False Negatives.

The resulting curve is called a Receiver Operating Characteristics (ROC) curve.
The ROC plots True Positive Rate (TPR) against False Positive Rate (FPR) as a function of B for the training data $\{\vec{X}_m\}$, $\{y_m\}$.

True Positives and False Positives
For each training sample, the detection as either Positive (P) or Negative (N)

IF $g(\vec{X}_m)+B \geq 0$ THEN P else N

The detection can be TRUE (T) or FALSE (F) depending on the indicator variable y_m

IF $y_m = D(\vec{X}_m)$ THEN T else F

Combining these two values, any detection can be a True Positive (TP), False Positive (FP), True Negative (TN) or False Negative (FN).

For the M samples of the training data $\{\vec{X}_m\}$, $\{y_m\}$ we can define:

#P as the number of Positives in the training data.
#N as the number of Negatives in the training data.
#T as the number of training samples correctly labeled by the detector.
#F as the number of training samples incorrectly labeled by the detector.

From this we can define:

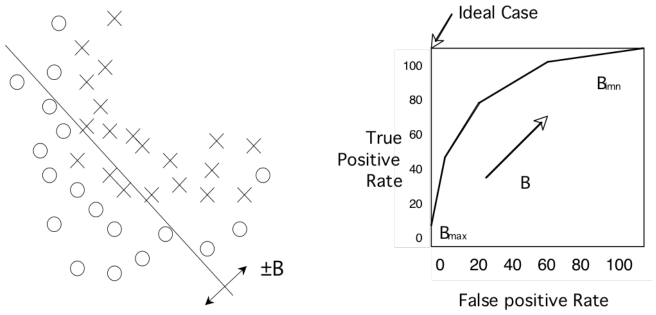
#TP as the number of training samples correctly labeled as Positive
#FP as the number of training samples incorrectly labeled as Positive
#TN as the number of training samples correctly labeled as Negative
#FN as the number of training samples incorrectly labeled as Negative

Note that #P = #TP + #FN (positives in the training data)
And #N = #FP+ #TN (negatives in the training data)

The True Positive Rate (TPR) is $TPR = \frac{\#TP}{\#P} = \frac{\#TP}{\#TP + \#FN}$

The False Positive Rate (FPR) is $FPR = \frac{\#FP}{\#N} = \frac{\#FP}{\#FP + \#TN}$

The ROC plots the TPR against the FPR as a bias B is swept through a range of values.



When B is at its minimum, all the samples are detected as N, and both the TPR and FPR are 0. As B increases both the TPR and FPR increase. Normally TPR should rise monotonically with FPR. If TPR and FPR are equal, then the detector is no better than chance.

The closer the curve approaches the upper left corner, the better the detector.

	$y_m = D(\vec{X}_m)$	$y_m \neq D(\vec{X}_m)$
$d(g(\vec{X}_m)+B \geq 0)$	True Positive (TP)	False Positive (FP)
$d(g(\vec{X}_m)+B < 0)$	True Negative (TN)	False Negative (FN)

Precision and Recall

Precision, also called Positive Predictive Value (PPV), is the fraction of retrieved instances that are relevant to the problem.

$$PP = \frac{TP}{TP + FP}$$

A perfect precision score (PPV=1.0) means that every result retrieved by a search was relevant, but says nothing about whether all relevant documents were retrieved.

Recall, also known as sensitivity (S), hit rate, and True Positive Rate (TPR) is the fraction of relevant instances that are retrieved.

$$S = TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

A perfect recall score (TPR=1.0) means that all relevant documents were retrieved by the search, but says nothing about how many irrelevant documents were also retrieved.

Both precision and recall are therefore based on an understanding and measure of relevance. In our case, “relevance” corresponds to “True”.
Precision answers the question “How many of the Positive Elements are True ?”
Recall answers the question “How many of the True elements are Positive”?

In many domains, there is an inverse relationship between precision and recall. It is possible to increase one at the cost of reducing the other.

F-Measure
The F-measures combine precision and recall into a single value. The F measures measure the effectiveness of retrieval. The best value is 1 when Precision and Recall are perfect. The worst value is at Zero.

The F_1 score weights recall higher than precision.

F_1 Score:

$$F_1 = \frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}} = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is the harmonic mean of precision and recall.

Accuracy

Accuracy is the fraction of test cases that are correctly classified (T).

$$ACC = \frac{T}{M} = \frac{TP + TN}{M}$$

where M is the quantity of test data.

Note that the terms Accuracy and Precision have a very different meaning in Measurement theory. In measurement theory, accuracy is the average distance from a true value, while precision is a measure of the reproducibility for the measurement.

Benchmark Data Sets Visual Task Challenges

As we saw in lesson 11, many of the popular architectures were designed specifically to address research challenges based on image data sets. Classically, these data sets were for challenges related to object detection. More recently the challenges increasingly address other visual tasks.

The ImageNet Challenge for Object Detection

ImageNet was originally concerned with Image Classification: Does an image (or imagette) contain an instance of a class? Most state-of-the-art object detection networks pre-train on ImageNet and then rely on transfer learning to adapt the learned recognition system to a specific domain. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) uses a "trimmed" list of only 1000 image categories or "classes", including 90 of the 120 dog breeds classified by the full ImageNet schema.

ImageNet crowdsources its annotation process. In 2018 there were more than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. Image-layer annotations indicate the presence or absence of an object class in an image. Object-layer annotations provide a bounding box around the (visible part of the) indicated object.

In 2014, more than fifty institutions participated in the ILSVRC, almost exclusively with different forms of Network Architectures. In 2017, 29 of 38 competing teams in the ILSVRC demonstrated error rates less than 5% (better than 95% accuracy).

However, the ILVSRC task is to identify images as belonging to one of a thousand categories; humans can recognize a larger number of categories, and also (unlike the programs) can judge the context of an image. More importantly, humans are capable of MANY other visual tasks involving Spatio-temporal interaction with 3D. In cognitive psychology, these are referred to as visual competences.

COCO - Common Objects in Context
Microsoft COCO is a large-scale object detection, segmentation, and captioning dataset created in 2015. Images in the COCO data set display are everyday objects captured from everyday scenes. This adds some "context" to the objects captured in the scenes

COCO contains more than 2.5M instances in 91 object categories, with 5 captions per image 330K images (200K+ annotated) with 250,000 people with key points.

Data sets for other visual tasks
An extensive (very large) list of publically available benchmark data sets and research challenges for visual tasks may be found at.
https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research
This list continues to grow rapidly.

There is a growing interest in developing techniques for recognizing actions and activities from video sequences and multimodal data. The recent emergence of generative techniques, combined with rapid advances in Robotics and Autonomous Systems appear likely greatly expand this set of tasks. In particular the recent progress in Transformers and Attention-based techniques in Natural Language processing appear likely to enable many new competences for computer vision.

The following are some techniques for multiple object detection and semantic detection.