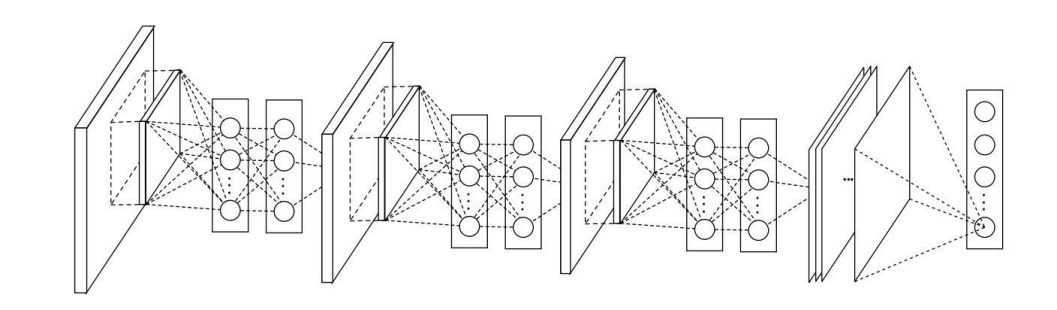
Deep Learning
CS 446/546

Outline

- Historical Notes
- Activation Functions
- Regularization Techniques



- Deep Learning Challenges
- SGD, Momentum, Parameter Initialization, AdaGrad, Adam, Newton's Method
- CNNs
- Siamese Networks: One-Shot Learning, Similarity Learning; GANs

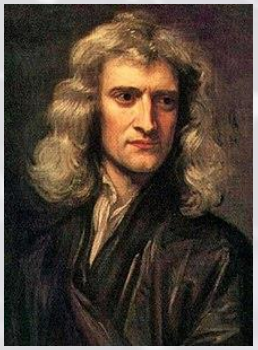
Historical Notes

• Feedforward networks can be seen as efficient non-linear function approximators based on using gradient descent to minimize the error in a function approximation.

•As such, the modern feedforward NN is the culmination of centuries of progress on **the general function approximation task**.

• The *chain rule* underlying backprop was invented by Leibniz (1796), and due naturally to foundations also laid by Newton.

• Calculus and algebra have been used to solve optimization problems in closed form since their inception, but *gradient descent* was not introduced as a technique for iteratively approximating the solution to optimization problems until 19C (Cauchy, 1847).



Newton

Leibniz

Cauchy

Al-Khwarizmi

Galois

$$\frac{\partial E}{\partial w_{ij}} = \delta_j * x_i$$

$$o_j' = o_j * (1 - o_j)$$

$$\varphi_j' = (1 - \varphi_j) * (1 + \varphi_j)$$

$$\varphi_j' = \varphi_j * (1 - \varphi_j)$$

$$e_j = (o_j - t_j)$$

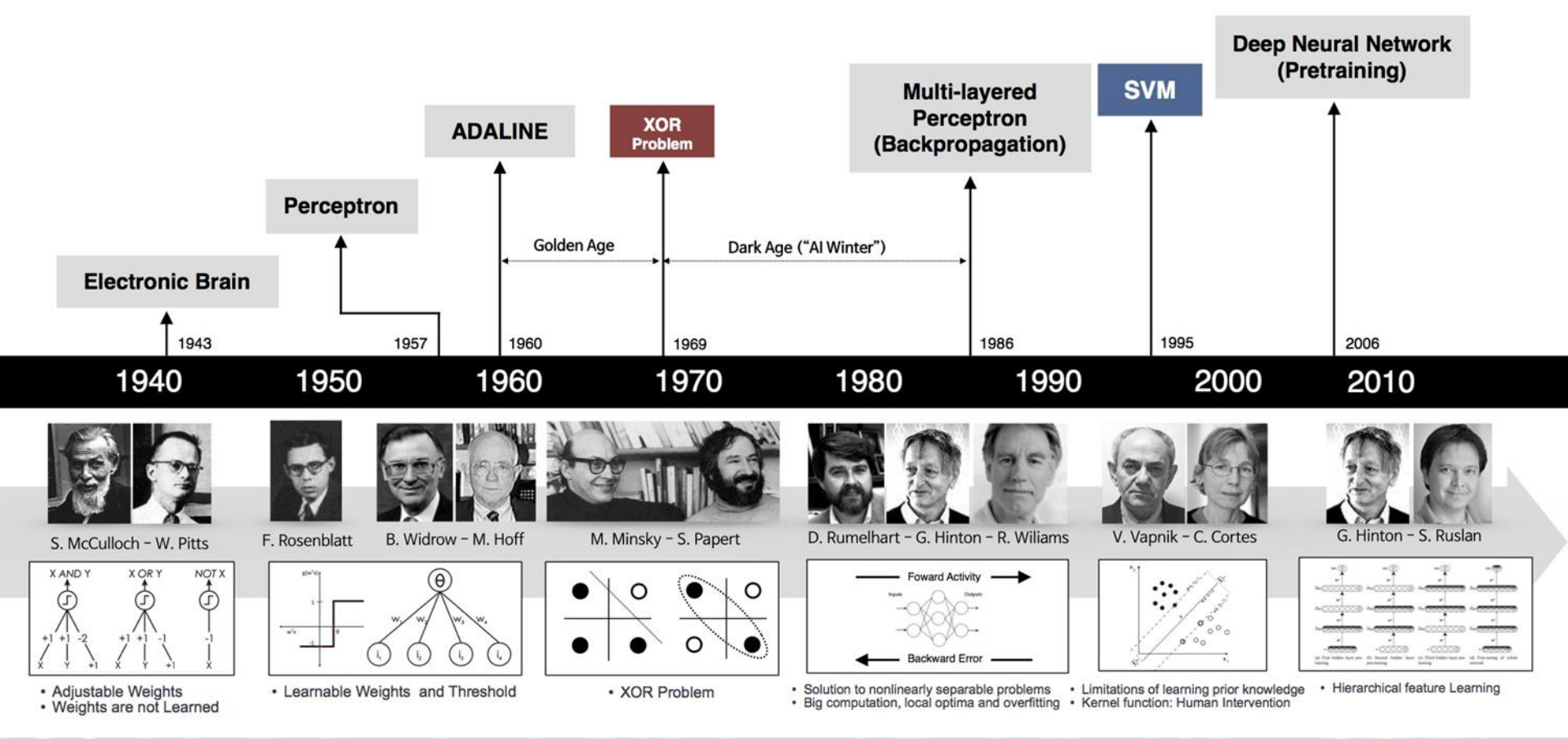
$$\delta_j = e_j * o_j'$$

$$\delta_j = (\sum \delta_j w_j) * \varphi_j'$$

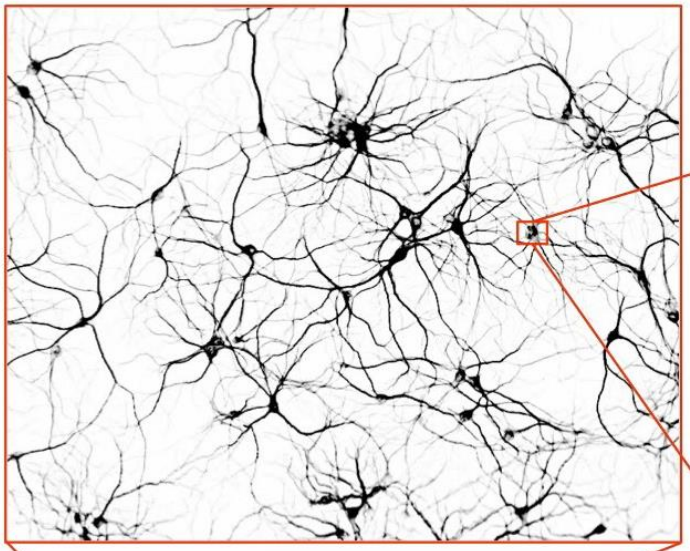
$$\Delta w_{ij} = \alpha * \frac{\partial E}{\partial w_{ij}}$$

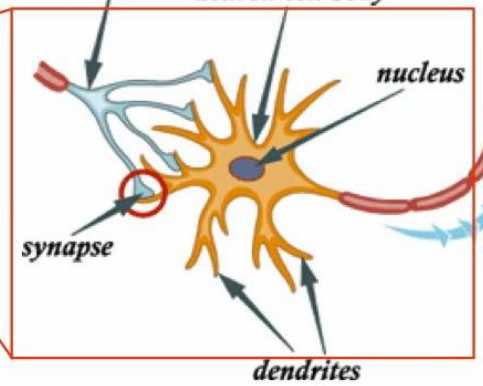
$$w_{ij}' = w_{ij} + \Delta w_{ij}$$

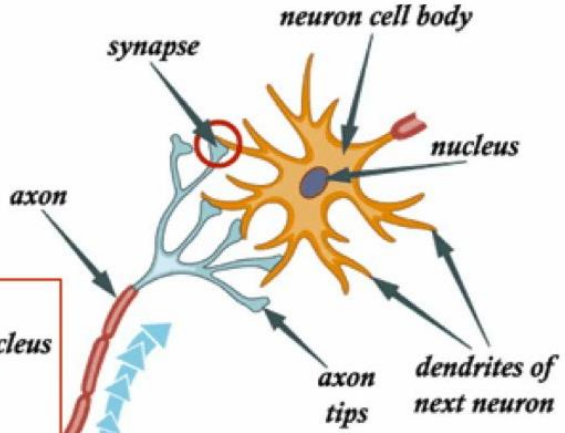
Historical Notes



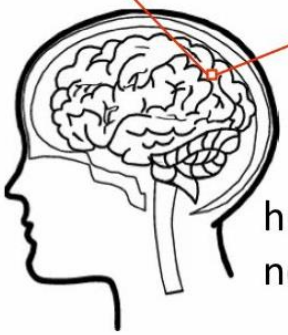
Neurons & the Brain



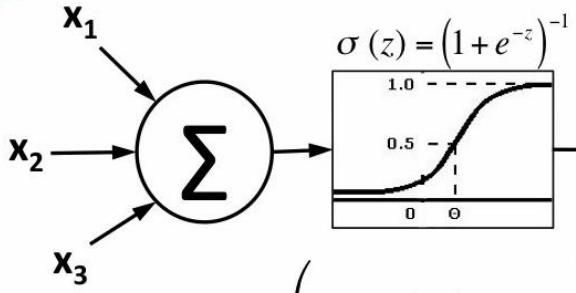




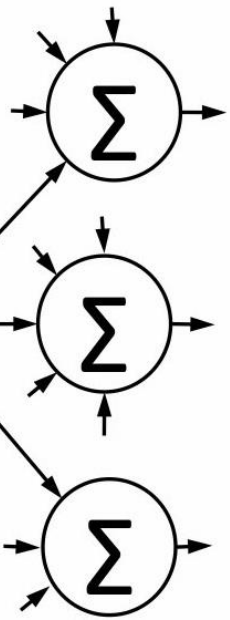
electrical signal



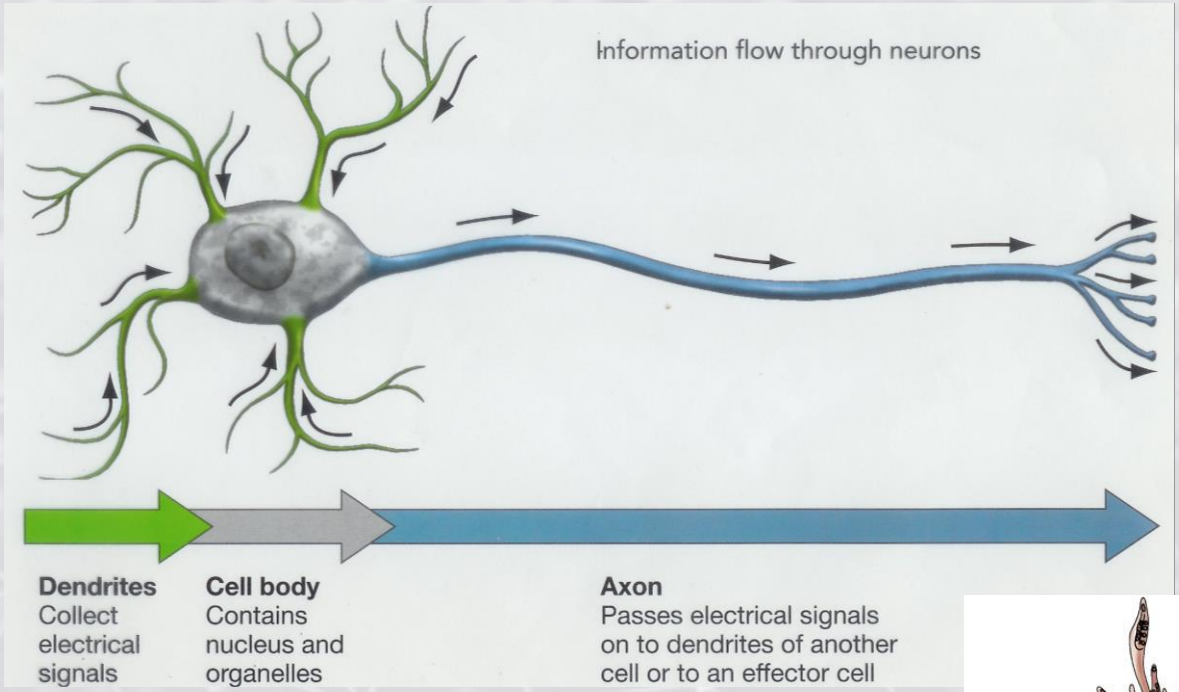
humans don't need features

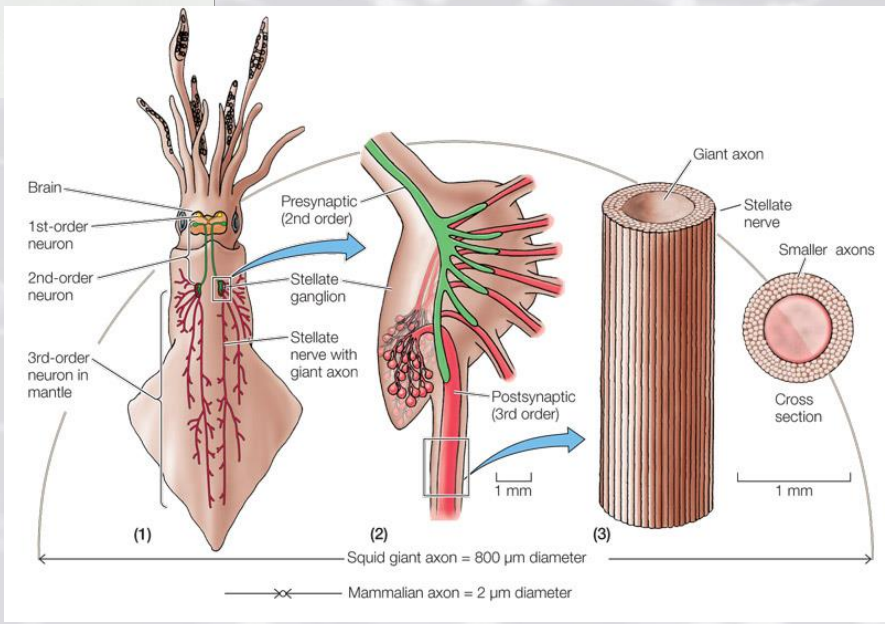


$$y_j = \sigma\left(b_j + \sum_i w_{ij} x_i\right)$$



Neurons & the Brain



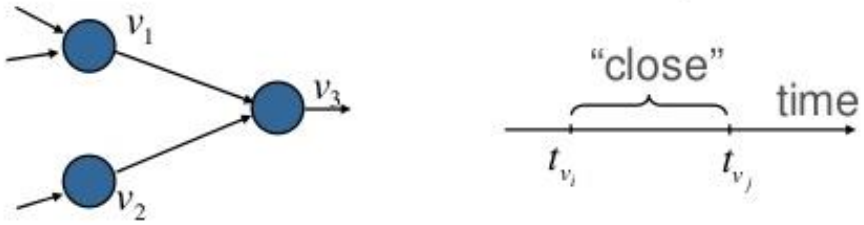


Hebb's Postulate

"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

- In other words: if two neurons fire "close in time" then strength of synaptic connection between them increases.

$$\Delta w_{ij}(t) = \eta v_i v_j g(t_{v_i}, t_{v_j})$$



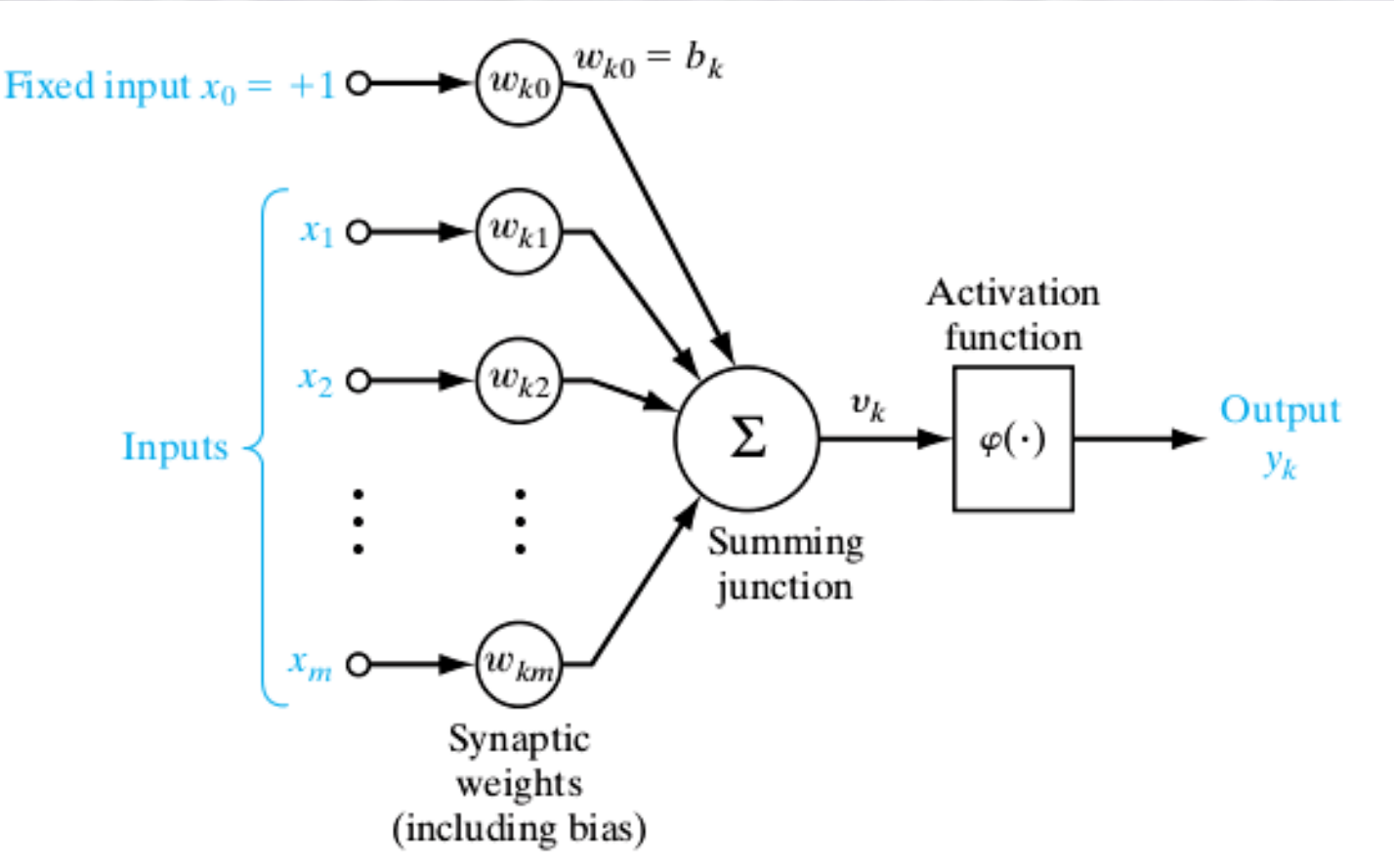
- Weights reflect correlation between firing events.

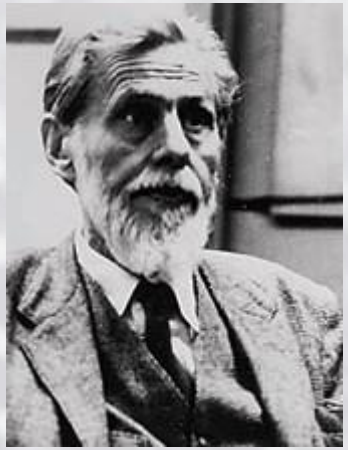
Neurons & the Brain

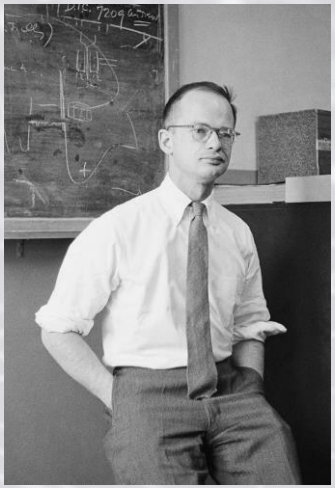
- Human brain contains ~10^{11} neurons
- Each individiaul neuron connects to ~10^{4} neuron
- ~10^{14} total synapses!

	Brain	Computer
Number of Processing Units	$\approx 10^{11}$	$\approx 10^{9}$
Type of Processing Units	Neurons	Transistors
Form of Calculation	Massively Parallel	Generally Serial
Data Storage	Associative	Address-based
Response Time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Processing Speed	Very Variable	Fixed
Potential Processing Speed	$\approx 10^{13}$ FLOPS [14]	$\approx 10^{18}$ FLOPS
Real Processing Speed	$\approx 10^{12}$ FLOPS	$\approx 10^{10}$ FLOPS
Resilience	Very High	Almost None
Power Consumption per Day	20W	300W [15]

McCulloch & Pitts Neuron Model (1943)







(3) Components:

(1) Set of weighted inputs $\{w_i\}$ that correspond to synapses

(2) An "adder" that sums the input signals (equivalent to membrane of the cell that collects the electrical charge)

(3) An activation function (initially a threshold function) that decides whether the neuron fires ("spikes") for the current inputs.

McCulloch & Pitts Neuron Model (1943)

Limitations & Deviations of the M-P Neuron Model:

- Summing is linear.
- No explicit model of "spike trains" (sequence of pulses that encodes information in biological neuron).
- Threshold value is usually fixed.
- Sequential updating implicit (biological neurons usually update themselves asynchronously)
- Weights can be positive (excitatory) or negative (inhibitory); biological neurons do not change in this way.
- Real neurons can have synapses that link back to themselves (e.g. feedback loop) – see RNNs (recurrent neural networks).
- Other biological aspects ignored: chemical concentrations, refractory periods, etc.

Historical Notes

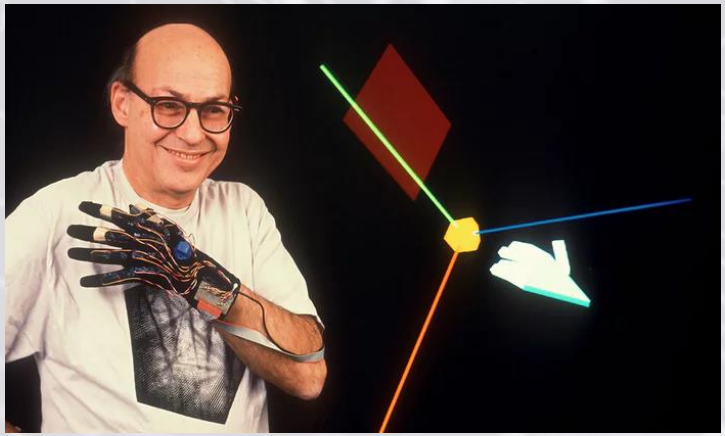
• Beginning in the 1940s, these function approximation techniques were used to motivate ML models such as the **percepton**. However, the earliest models were based on linear models.

• In the 1960s **Rosenblatt** proved that the perceptron learning rule converges to correct weights in a finite number of steps, provided the training examples are linearly separable.

•Critics including Marvin Minsky point out several of the flaws of the linear model family, such as its inability to learn the XOR function, which led to a backlash against the entire NN approach.

• Learning non-linear functions required the development of a MLP (multi-layer perceptron) and a means of computing the gradient through such a model. Efficient applications of the chain rule based on DP (dynamic programming) began to appear in the 1960s and 1970s.

Rosenblatt



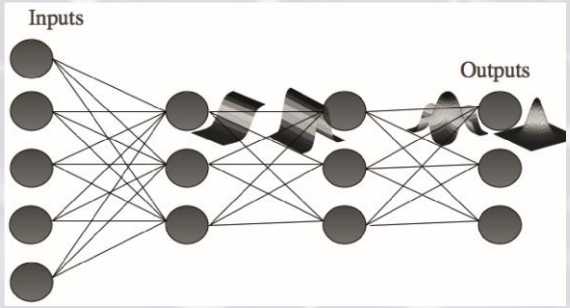
Minsky

Historical Notes

- 1969: **Minsky and Papert** proved that perceptrons cannot represent non-linearly separable target functions.

- However, they showed that adding a fully connected hidden layer makes the network more powerful.
  - i.e., Multi-layer neural networks can represent non-linear decision surfaces.
- Later it was shown that by using continuous activation functions (rather than thresholds), a fully connected network with a single hidden layer can in principle represent any function.
- 1986: "rediscovery" of backprop algorithm: **Hinton** et al.

- The **Universal Approximation Theorem (1989)** states that <u>one hidden layer is sufficient to approximate any function to arbitrary accuracy</u> with a NN. (we say: "NNs are universal function approximators"); RNNs are *Turing Complete*.



Universal Approximation Properties

- A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models.

- The **universal approximation theorem** (UAT) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any *Borel measurable* (e.g. a continuous function on a closed and bounded subset of R^n) function from one finite-dimensional space to another with any desired non-zero amount of error, provided the network is given enough hidden units.

- The UAT states that regardless of what function we are trying to learn, we know that a sufficiently large MLP will be able to represent this function. We are not guaranteed, however, that the training algorithm will be able to learn the function.

Universal Approximation Properties

- Even if the MLP is able to represent the function. Learning can fail for (2) different reasons:

(1) The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.

(2) The training algorithm might choose the wrong function as a result of overfitting.

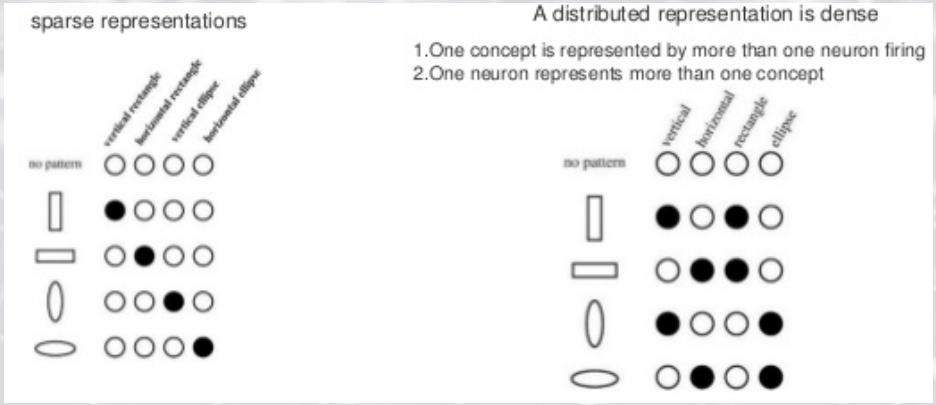
- Feedforward networks provide a universal system for representing functions in the sense that, given a function, there exists a feedforward network that approximations the function; there is **no universal procedure** for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

*Note also that the theorem does not prescribe the size of the network (some bounds can be approximated); unfortunately, in the worst case, an exponential number of hidden units may be required.

*Recall that any time we choose a specific ML algorithm, we are implicitly state some set of prior beliefs we have about what kind of function the algorithm should learn; choosing a deep model generally indicates that we want to learn a composition of several simpler functions.

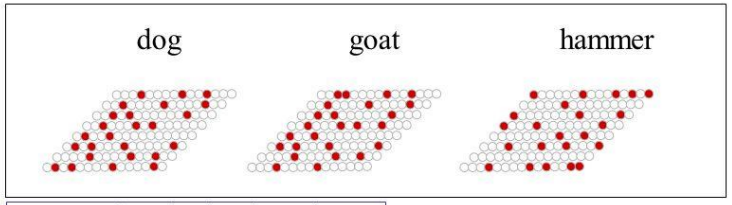
Historical Notes

•The "rediscovery" of the backpropagation algorithm (Hinton & Rumelhardt) initialed a very active period of research for MLPs. In particular, "**connectionism**" took root in the ML community, which placed emphasis on connections between neurons as the locus of learning and memory (cf. **distributed representation**: each concept is represented by many neurons, each neuron participates in the representation of many concepts.

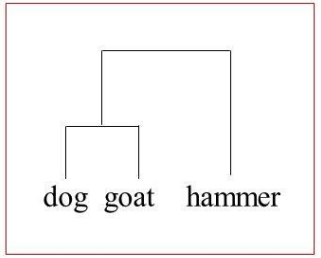


http://www.cs.toronto.edu/~bonner/courses/2014s/csc321/lectures/lec5.pdf

Distributed Representations in the Brain: Overlapping Patterns for Related Concepts (Kiani et al, 2007)



- Many hundreds of single neurons recorded in monkey IT.
- 1000 different photographs were presented twice each to each neuron.
- Hierarchical clustering based on the distributed representation of each picture:
  - The pattern of activation over all the neurons



http://www.jneurosci.org/content/35/13/5180

Historical Notes

• Following the success of backprop, NN research gained popularity and reached a peak in the early 1990s.Afterwards, other ML techniques became more popular until the modern deep learning renaissance that began in 2006.

• The core ideas behind modern feedforward nets have not changed substantially since the 1980s. The same backprop algorithm and the same approaches to gradient descent are still in use. Most of the improvement in NN performance from 1986-2018 **can be attributed to two factors**:

(1) Larger datasets have reduced the degree to which statistical generalization is a challenge for NNs.

(2) NNs have come much larger because of more powerful computer (including the use of GPUs) and better software infrastructure.

Historical Notes

•A small number of algorithmic changes have also improved the performance of NNs.

One of these algorithmic changes was the replacement of mean squared error (MSE) with the **cross-entropy family of loss functions**. MSE was popular in the 1980s and 1990s but was gradually replaced by cross-entropy losses and the principles of MLE as ideas spread between the statistics community and ML community.

• The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using MSE.

• The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with **piecewise linear hidden units**, such as *rectified linear units* (RELUs). Rectification using the $\max\{0,z\}$ function was introduced in early NN models.

As of the early 2000s, rectified linear units were avoided due to the belief that activation functions with non-differentiable points must be avoided.

For small datasets, Jarrett et al. (2009) observed that using rectifying non-linearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, enabling the classifier layer at the top to learn

Historical Notes

• When more data are available, learning begins to extract enough useful knowledge to exceed the performance of randomly chosen parameters.

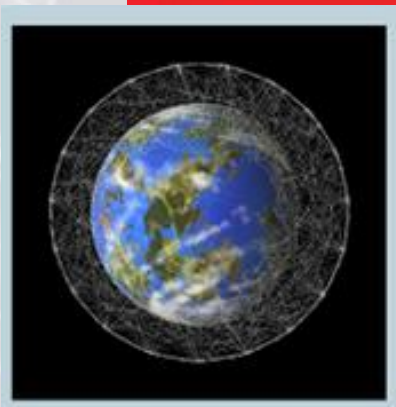
RELUs are also of historical interest because they show that neuroscience has continued to have an influence on the development of deep learning algorithms. Glorot et al. (2011) motivated RELUs from biological considerations. The half-rectifying non-linearity was intended to captured these properties of biological neurons:

(1) For some inputs, biological neurons are completely inactive

(2) For some inputs, a biological neuron's output is proportional to its inputs

(3) Most of the time, biological neurson operate in the regime where they are inactive (i.e. they should have sparse activations).

Historical Notes



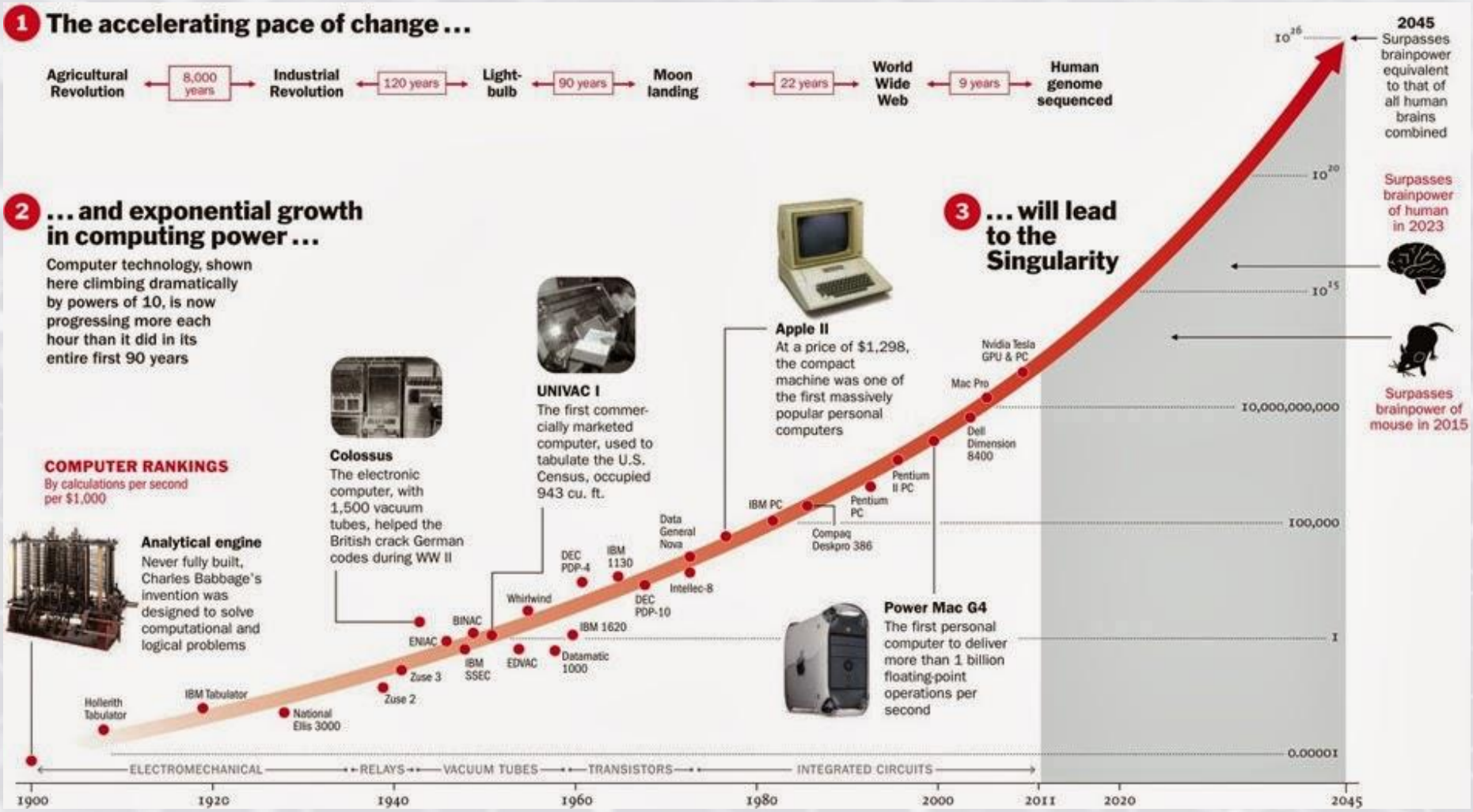
TYPE I CIVILIZATION harnesses all the resources of a planet. Carl Sagan estimated that Earth rates about 0.7 on the scale.

TYPE II CIVILIZATION harnesses all the radiation of a star. Humans might reach Type II in a few thousand years.

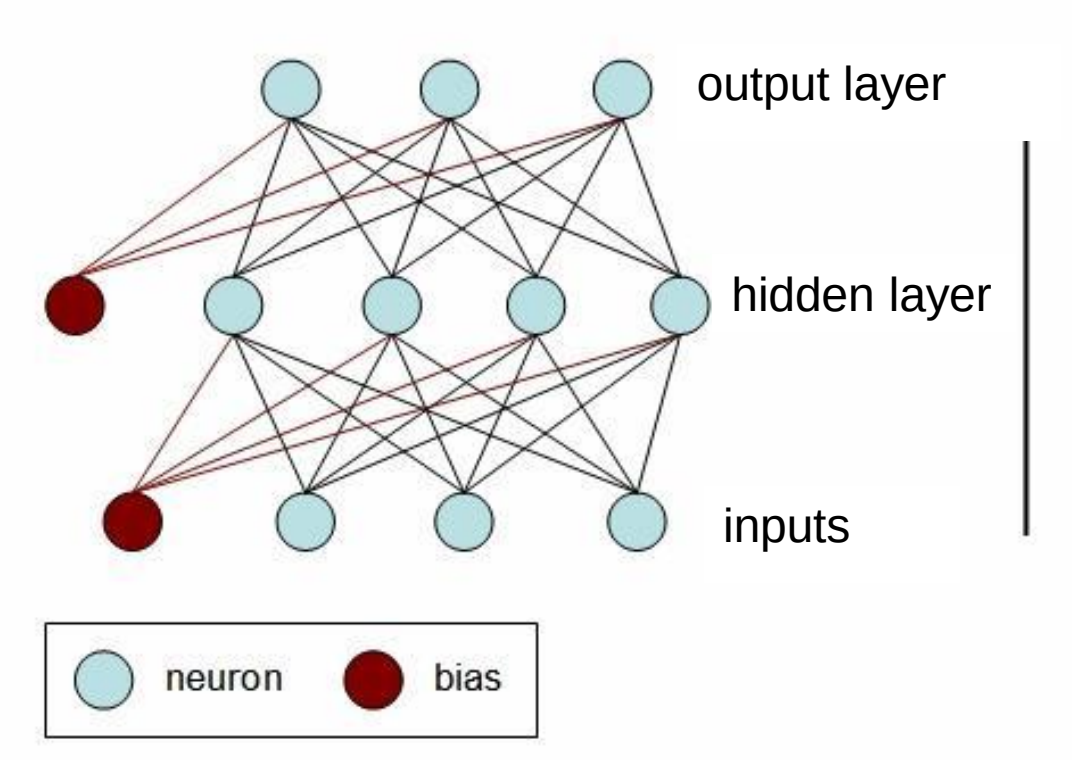
TYPE III CIVILIZATION harnesses all the resources of a galaxy. Humans might reach Type III in a few hundred thousand to a million years.

The Kardashev Scale

Historical Notes



A "two"-layer neural network



(activation represents classification)

(internal representation)

(activations represent feature vector for one training example)

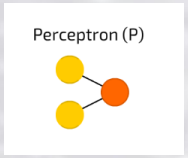
•**Input layer**—It contains those units (artificial neurons) which receive input from the outside world on which network will learn, recognize about or otherwise process.

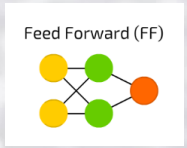
•**Output layer**—It contains units that respond to the information about how it's learned any task.

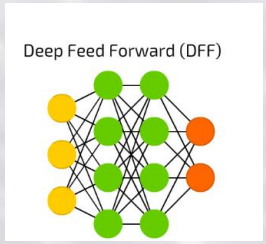
•**Hidden layer**—These units are in between input and output layers. The job of hidden layer is to transform the input into something that output unit can use in some way.

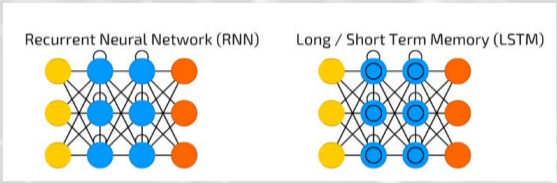
Most neural networks are fully connected that means to say each hidden neuron is fully connected to the every neuron in its previous layer(input) and to the next layer (output) layer.

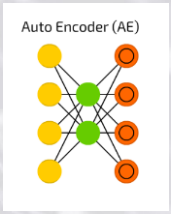
A Neural Network "Zoo"

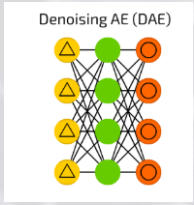


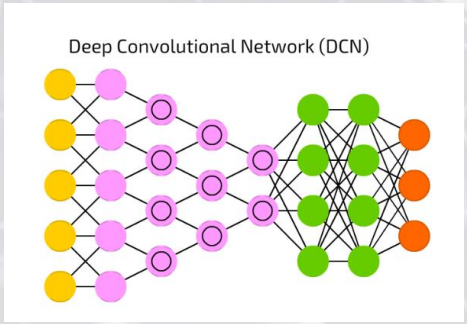




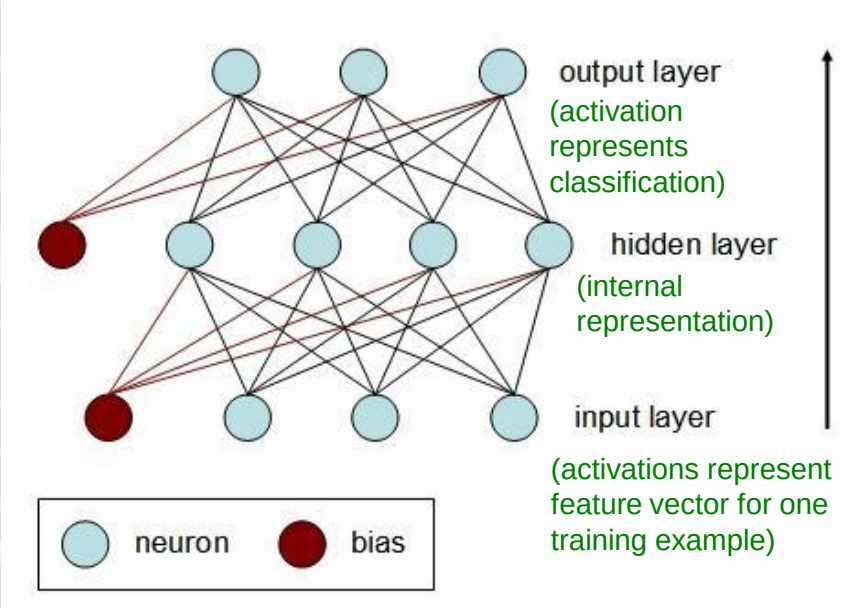








Neural network notation



x_i : activation of **input** node *i*.

h_j : activation of **hidden** node *j*.

o_k : activation of **output** node *k*.

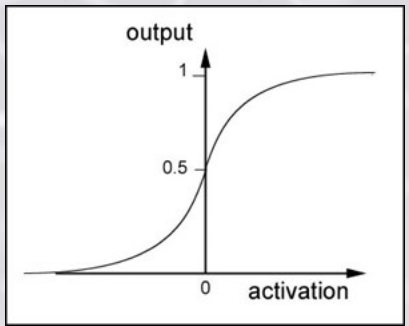
w_{ji} : weight from node *i* to node *j*.

σ : "sigmoid function".

For each node *j* in hidden layer,

$$h_j = s\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0}\right)$$

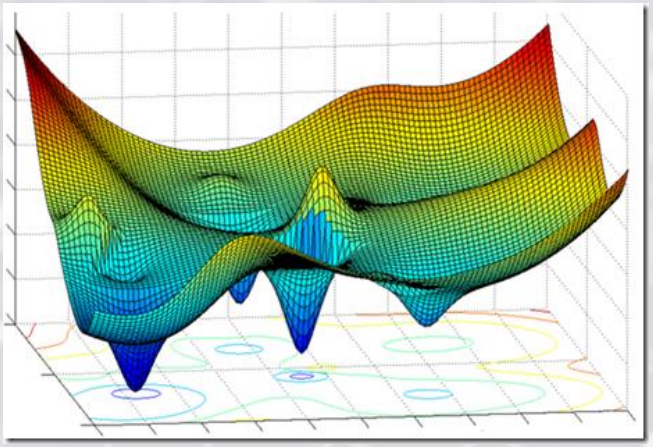
Sigmoid function:

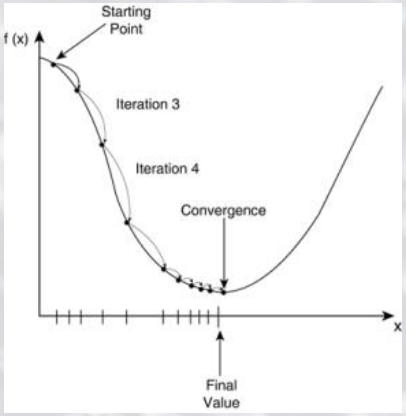


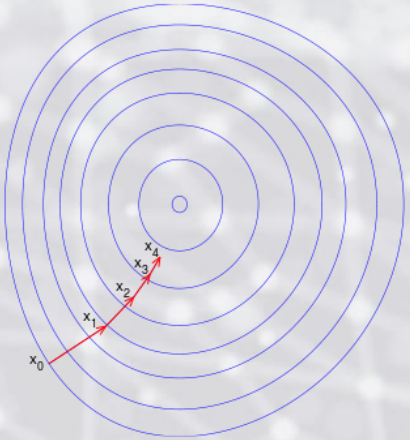
For each node *k* in output layer,

$$o_k = s\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0}\right)$$

Gradient Descent





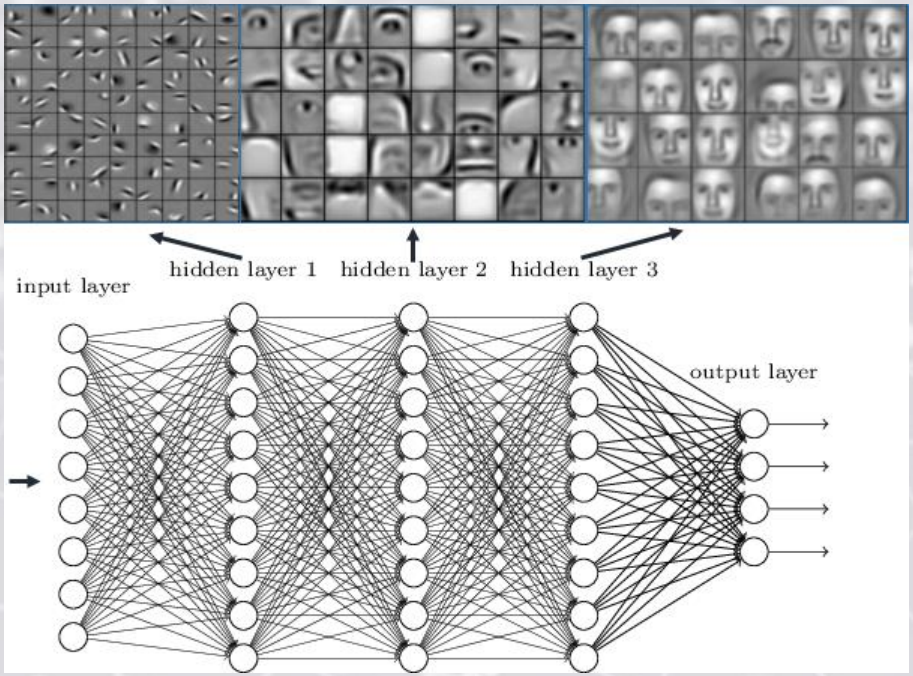


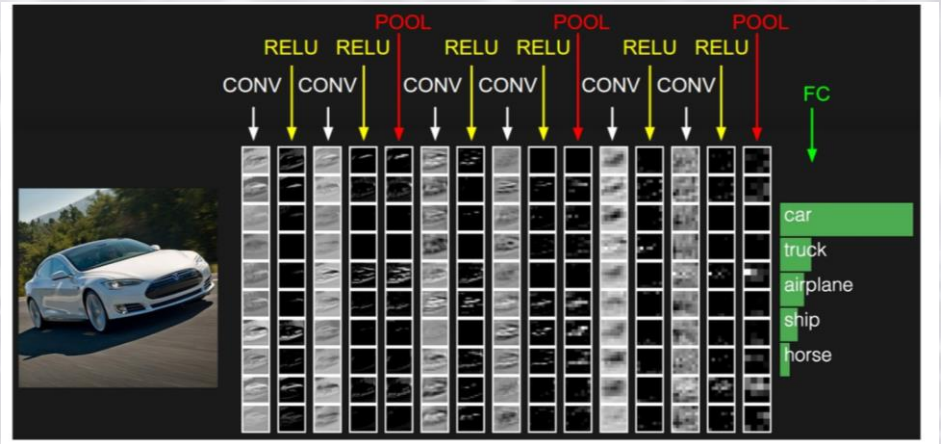
(*) Backpropagation is one particular instance of a larger paradigm of optimization algorithms know as **Gradient Descent** (also called "hill climbing").

(*) There exists a large array of nuanced methodologies for efficiently training NNs (particularly DNNs), including the use of **regularization**, **momentum**, **dropout**, **batch normalization**, pre-training regimes, initialization processes, etc.

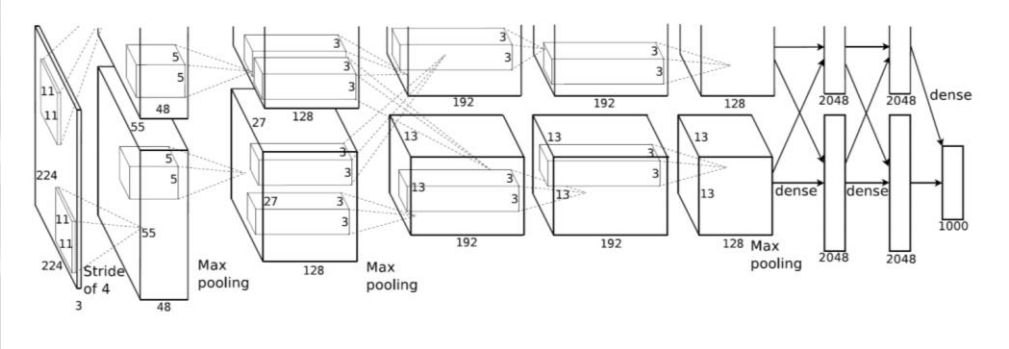
(*) Traditionally, the backpropagation algorithm has been used to efficiently train a NN; more recently the **Adam stochastic optimization method** (2014) has eclipsed backpropagation in practice: https://arxiv.org/abs/1412.6980

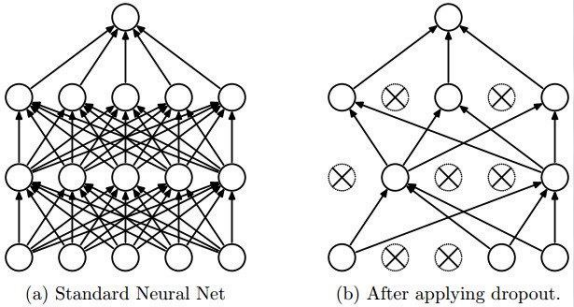
DNNs Learn Hierarchical Feature Representations





DNNs: AlexNet (2012)



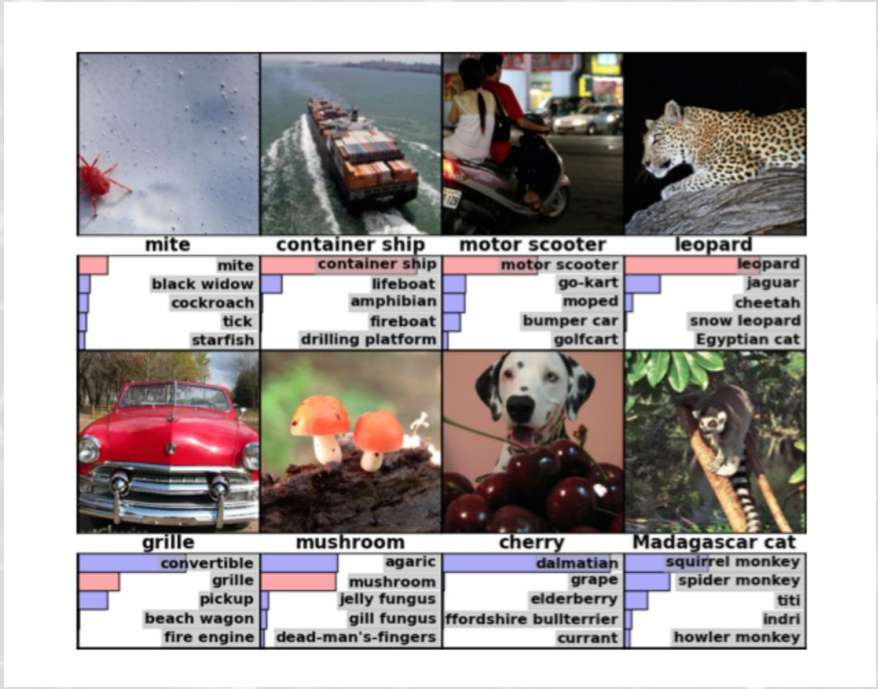


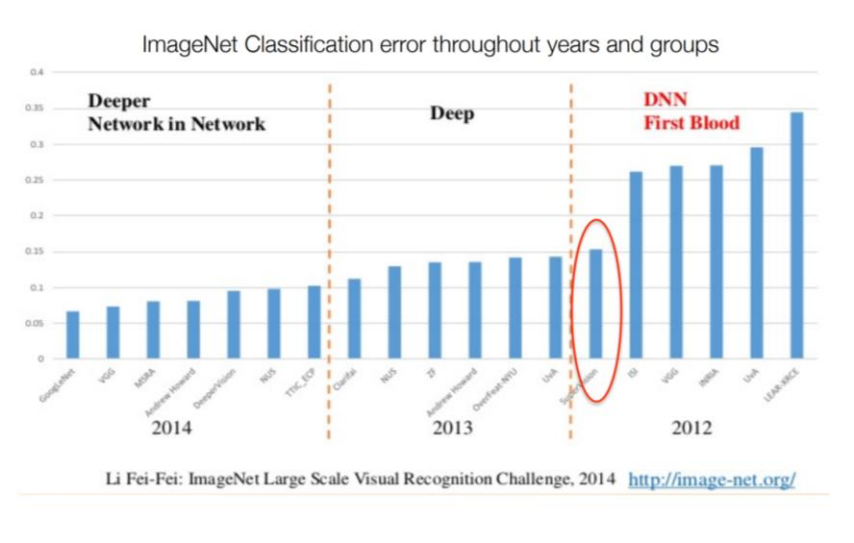
AlexNet was developed by Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever; it uses CNNs with GPU support. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points ahead of the runner up.

Among other innovations: AlexNet used GPUs, utilized RELU (rectified linear units) for activations, and "dropout" for training.

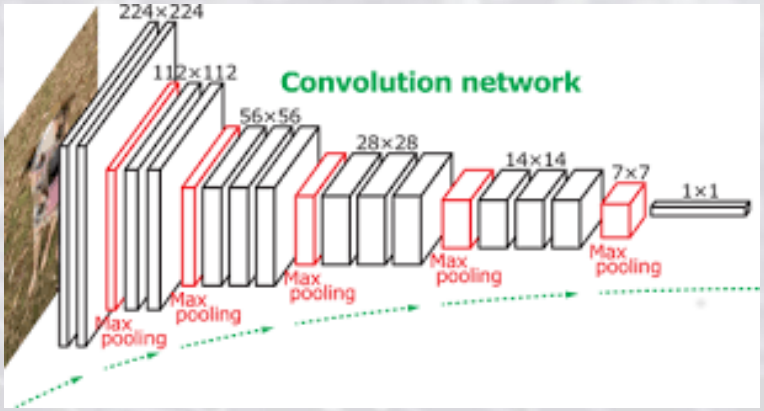
https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

DNNs: AlexNet (2012)





DNNs: VGG (2014)



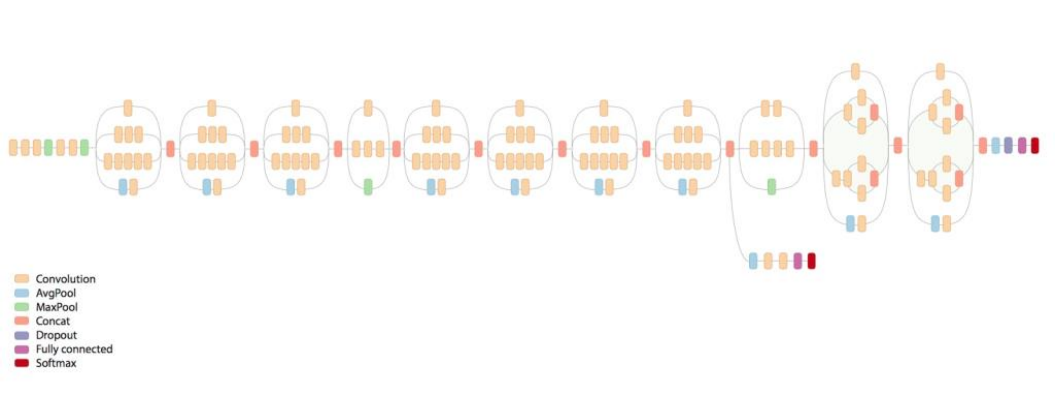
- Team at Oxford produced influential DNN architecture (VGG).Using very small convolutional filters (3x3), they achieved a significant improvement on the prior-art configurations by pushing the depth to 16–19 weight layers.

- Team achieved first and second place on the ImageNet Challenge 2015 for both localization and classification tasks, respectively.

- Using pre-trained VGG is very common practice in research.

https://arxiv.org/pdf/1409.1556.pdf

DNNs: Inception (2015, Google)



- Team at Google (Szegedy et al.) produced an even deeper DNN (22 layers). No need to pick filter sizes explicitly, as network learns combinations of filter sizes/pooling steps; upside: newfound flexibility for architecture design (architecture parameters themselves can be learned); downside: ostensibly requires a large amount of computation – this can be reduced by using 1x1 convolutions for dimensionality reduction (prior to expensive convolutional operations).

- Team achieved new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14; 6% top-5 error rate for classification.

https://arxiv.org/pdf/1409.1556.pdf

Cross-Entropy Loss

• As mentioned, **cross-entropy loss** is generally preferred to MSE, particularly for classification problems with DNNs (it can also be used in non-classification settings).

Cross-entropy loss is defined:

$$E = -\sum c_i \log(p_i) + (1 - c_i) \log(1 - p_i)$$

Where *c* refers to one hot encoded classes (or labels), whereas *p* refers to softmax applied probabilities

(2) Properties make cross-entropy a natural loss function:

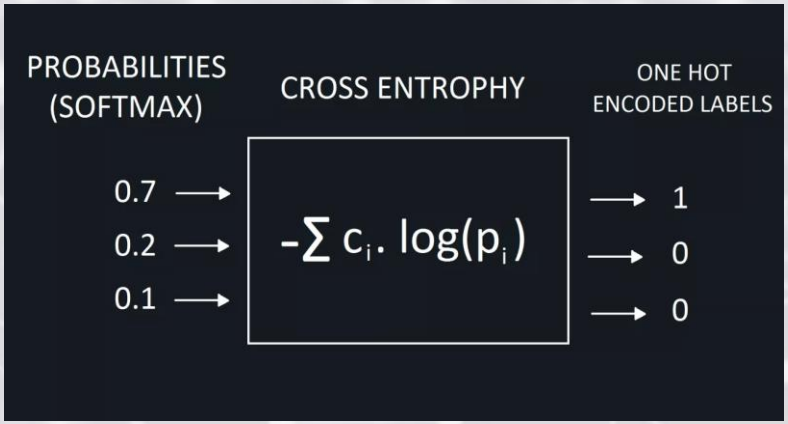
(1) $E \geq 0$; all individual terms are negative and there is a minus outside.

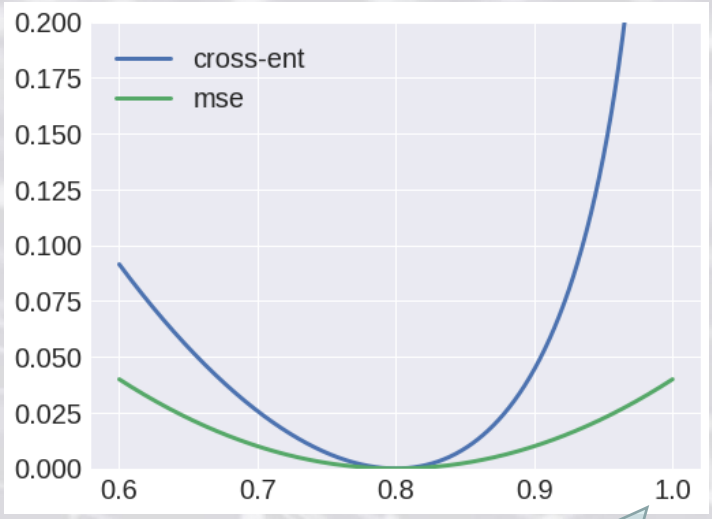
(2) If the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero. To demonstrate this, we assume (WLOG) that the desired outputs c are all either 0 or 1. Suppose for example that $c = 0$ and $p \approx 0$, for some input x (so the neuron has done well on this input). The first term in E vanishes, while the second term is close to zero; a similar analysis holds when $c = 1$ and $p \approx 1$.

Cross-Entropy Loss

• Cross-entropy loss is defined:

$$E = -\sum c_i \log(p_i) + (1 - c_i) \log(1 - p_i)$$





One can show that, for example, that the partial derivative of the cross-entropy loss function is:

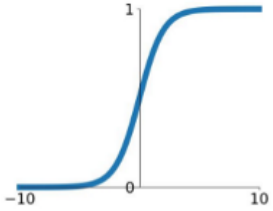
$$\frac{\partial E}{\partial w_j} = \sum_x x_j \left(\sigma(z) - y\right)$$

*(σ denotes the sigmoid function) Which indicates that the gradient is larger (i.e. learning is faster) the larger the error; in addition, the cross-entropy loss function does not in general "bottom out" like the MSE loss.

Activation Functions

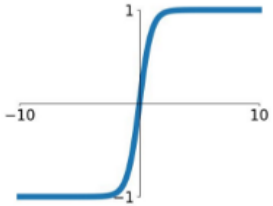
Sigmoid

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



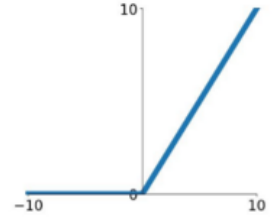
tanh

$$\tanh(x)$$



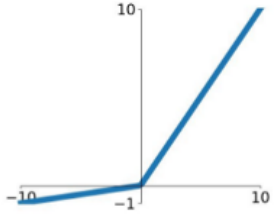
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

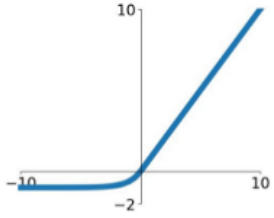


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



RELU & Their Generalizations

• Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

• These units are easy to optimize because they are so similar to linear units; the only difference being the RELU is zero across half of its domain. This makes the derivatives through a RELU remain large whenever the unit is active.

• The gradients are therefore not only large but consistent.

RELUs are typically used on top of an *affine transformation*:

$$\mathbf{h} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{b}\right)$$

(in practice one can set all elements of **b** to a small positive value such as 0.1; doing so makes it very likely that the RELUs will be initially active for most of the inputs in the training set and allow derivatives to pass through).

•One drawback of RELU: is that they cannot learn via gradient-based methods on examples for which their activation is zero; various generalizations of RELUs guarantee they receive gradient everywhere.

*affine transformations preserve points, straight lines, planes, and parallelism.

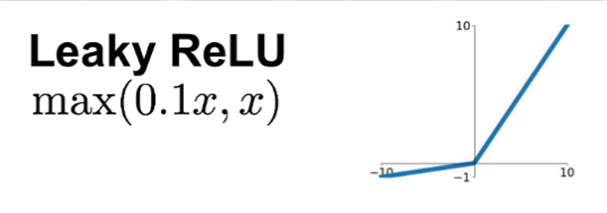
RELU & Their Generalizations

(3) Generalizations of RELUs are based on using a non-zero slope α_i when $z_i < 0$:

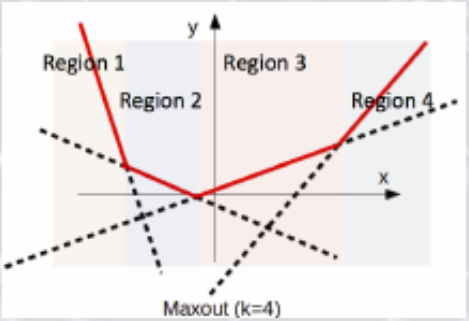
$$h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

(1) **Absolute value rectification** fixes $\alpha_i = -1$, to obtain $g(z) = |z|$; this method has been used for object recognition from images, where it makes sense to seek features that are invariant under poliartiy reversal of the input illumination.

(2) **Leaky RELU** fixes α_i to a small value like 0.01.



(3) **Maxout** units (Goodfellow, 2013); instead of applying an element-wise function g(z), maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of those groups.



This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space. Each maxout unit can learn a piecewise linear, convex function with up to *k* pieces; maxout units can thus be seen as learning the activation function itself rather than just the relationship between units; with enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity.

Regularization

• Regularization can be defined as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."

• There are many different regularization strategies; some put extra constraints on an ML model; some add extra terms to the objective function that can be though of as soft constraints applied to the parameter values. If chosen correctly, these extra constraints and penalties can lead to improve performance.

• Sometimes these constraints and penalties encode prior beliefs. Conversely, they are designed to express a generic preference for a simpler model class in order to promote generalization; sometimes these penalties are necessary to make an underdetermined problem determined or soluble; *ensemble methods* can also be considered a general form of regularization.

Regularization

• Regularization has been used for decades prior to the advent of deep learning. Linear models such as linear regression and logistic regression allow simple and effective regularization strategies.

• Many regularization approaches are based on limiting the capacity of models, such as NNs, linear regression, or logistic regression, by adding a parameter norm penalty: $\Omega(\boldsymbol{\theta})$ to the objective function J. Denote the *regularized objective function*:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta})$$

Where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard object function J. Setting $\alpha = 0$ results in no regularization. Larger values of α correspond to more regularization.

* In practice it is common to choose a parameter norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized (regularizing the bias parameters can introduce a significant amount of underfitting).

L^2 Regularization

• L^2 regularization (also called: weight decay, ridge regression) rives the weights closer to the original by adding a regularization term: $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|w\|_2^2$ to the objective function.

We can gain insight into the behavior of weight decay regularization by studying the gradient of the regularization objective function. To simplify the presentation, we assume no bias parameter, so $\boldsymbol{\theta}$ is just $\mathbf{w}$. Such a model has the following *total objective function*:

$$\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + J(\mathbf{w};\mathbf{X},\mathbf{y})$$

With the corresponding parameter gradient:

$$\nabla_w \tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \alpha\mathbf{w} + \nabla_w J(\mathbf{w};\mathbf{X},\mathbf{y})$$

To take a single gradient step to update the weights, we perform this update:

$$\mathbf{w} \leftarrow \mathbf{w} - \varepsilon\left(\alpha\mathbf{w} + \nabla_w J(\mathbf{w};\mathbf{X},\mathbf{y})\right)$$

Written another way, this update is:

$$\mathbf{w} \leftarrow (1-\varepsilon\alpha)\mathbf{w} - \varepsilon\nabla_w J(\mathbf{w};\mathbf{X},\mathbf{y})$$

L^2 Regularization

$$\mathbf{w} \leftarrow (1 - \varepsilon\alpha)\mathbf{w} - \varepsilon\nabla_w J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

• We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update.

• It is also useful to study the effect of L^2 regularization on linear regression to ascertain a sense of how L^2 regularization operates in the ML framework more generally.

• For linear regression, the cost function is the *sum of squared errors* (SSE):

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})$$

• When we add L^2 regularization, the objective function changes to:

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha\mathbf{w}^T\mathbf{w}$$

• This changes the normal equations for the solution from:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \rightarrow \mathbf{w} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

L^2 Regularization

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \rightarrow \mathbf{w} = \left(\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

• Using L^2 regularization replaces the matrix X^TX with the matrix $(X^TX+\alpha I)^{-1}$; the new matrix is the same as the original, but with the addition of α to the diagonal.

• The diagonal entries of this matrix correspond to the variance of each input features. We see that L^2 regularization causes the learning algorithm to "perceive" the input X as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added noise.

L^1 Regularization

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \rightarrow \mathbf{w} = \left(\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

- L^1 regularization on the model parameter $\mathbf{w}$ is defined as:

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$$

Thus, the regularized objective function is given by:

$$\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \alpha\|\mathbf{w}\|_1 + J(\mathbf{w};\mathbf{X},\mathbf{y})$$

with the corresponding gradient:

$$\nabla_w \tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \alpha sign(\mathbf{w}) + \nabla_w J(\mathbf{w};\mathbf{X},\mathbf{y})$$

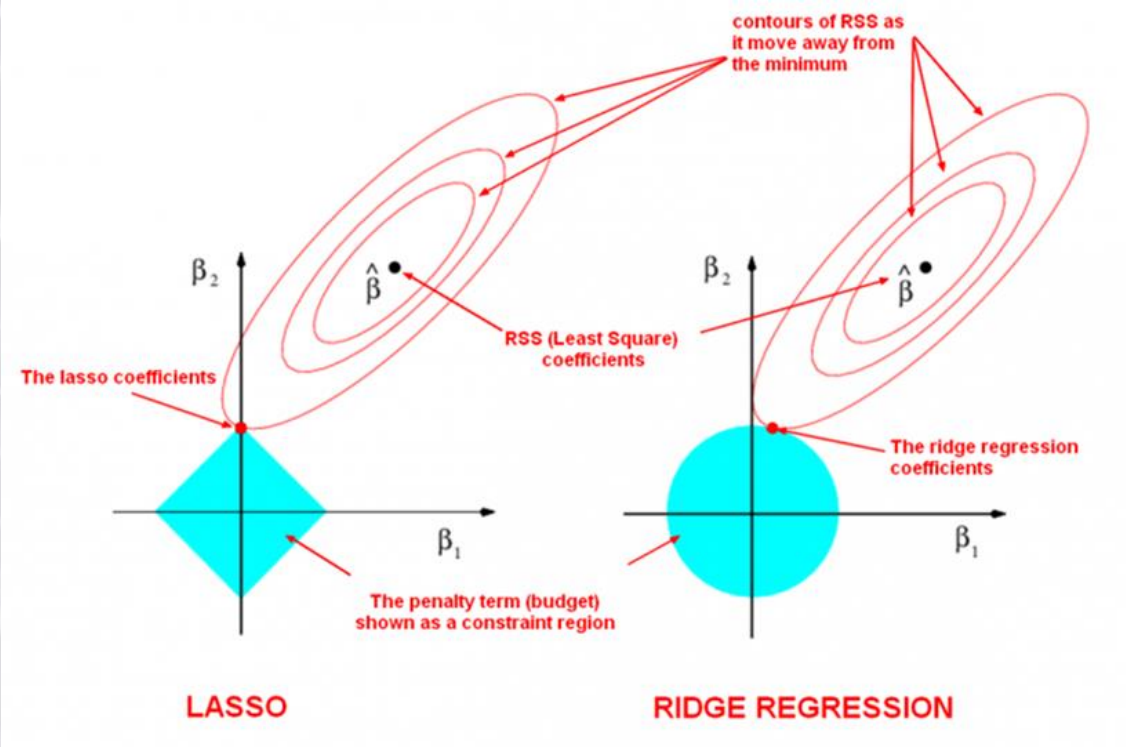
Where sign($\mathbf{w}$) is simply the sign of $\mathbf{w}$ applied element-wise.

* By inspection, we can see immediately that the effect of L^1 regularization is quite different from L^2 regularization. Specifically, the regularization contribution to the gradient no longer scales linearly with each w_i; instead, it is a constant factor with a sign equation to sign(w_i).

L^1 Regularization

• In comparison to L^2 regularization, L^1 regularization results in a solution is more sparse. Sparsity in this context refers to the fact that some parameters have an optimal value of zero. The sparsity of L^1 regularization is a qualitatively different behavior than arises with L^2 regularization.

• The sparsity property induced by L^1 regularization has been used extensively as a **feature selection mechanism**; feature selection simplifies an ML problem by choosing which subset of the available features should be used. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded.



Regularization and Under-Constrained Problems

• In some cases, regularization is necessary for ML problems to be properly defined.

• Many linear models, including regression and PCA, depend on inverting the matrix X^TX; this is not possible when X^TX is singular. This matrix can be singular whenever the data-generating distribution truly has no variance in one direction, or when no variance is observed because they are fewer examples (i.e. rows of X) than input features (columns of X).

• In this case, many forms of regularization correspond to inverting $X^TX+\alpha I$ instead; this regularized matrix is guaranteed to be invertible (*diagonally dominant matrices* are non-singular).

One can solve underdetermined linear equations using the **Moore-Penrose pseudoinverse**; it is defined as X^+ of the matrix X:

$$\mathbf{X}^{+} = \lim_{\alpha \to 0^{+}} \left(\mathbf{X}^{T}\mathbf{X} + \alpha \mathbf{I} \right)^{-1} \mathbf{X}^{T}$$

* This equation can be seen as performing linear regression with weight decay; specifically, the equation is the limit as the regularization coefficient shrinks to zero. Thus we can interpret the pseudoinverse as stabilizing underdetermined problems using regularization.

Data Augmentation

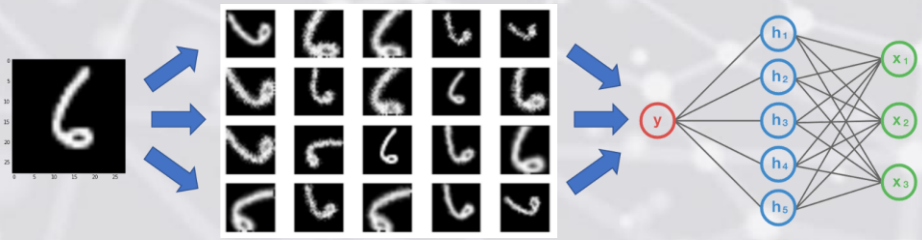
• The best way to make an ML model generalize better is to train it on more data. Of course, data are limited/expensive.

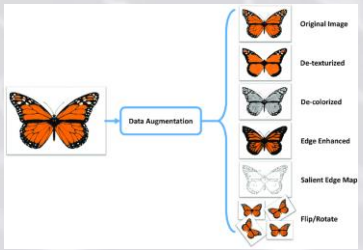
• One way to get around this problem is to generate synthetic data and add it to the training set.

• This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input **x** and summarize it with a single category identity *y*. This means that the main task facing a classifier is to be invariant to a wide variety of transformations; we can generate new (**x**, *y*) pairs easily by transforming the **x** inputs in our training set.

• Dataset augmentation has been particularly effective for **object recognition**; operations like translating the training images a few pixels in each direction can often greatly improve generalization; many operations such as rotating the image or scaling the image are also quite effective (one needs to be careful that the transformation does not alter the correct image class).

• **Injecting noise** in the input to a NN can also be seen as a form of data augmentation; one way to improve the robustness of a NN is to simply train them with random noise applied to their inputs.

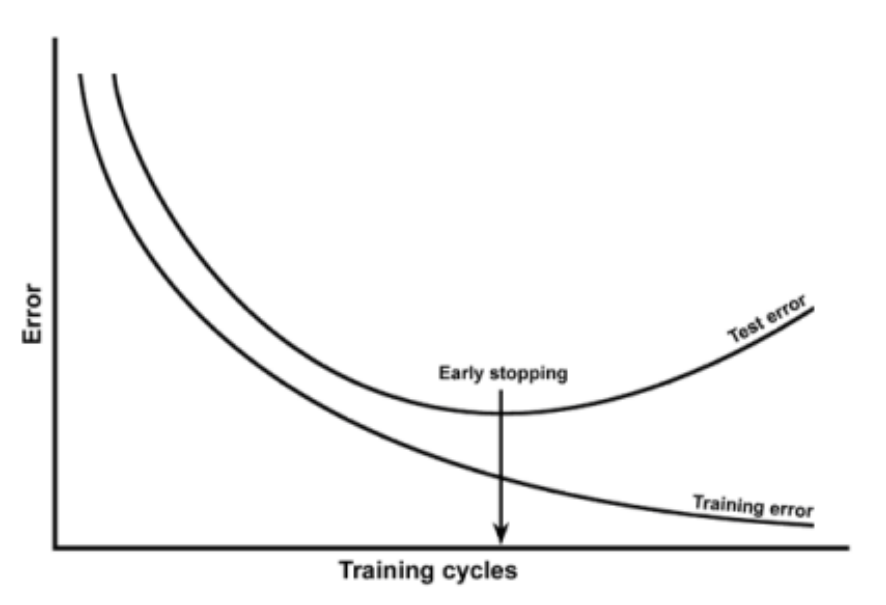




Early Stopping

• When training large models with sufficient representation capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.

• This means we can obtain a model with better validation set error (and hopefully better test error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters; when the training terminates, we return these parameters, rather than the latest parameters.



* This strategy is known as **early stopping**; it is one of the most common forms of regularization used in deep learning.

Early Stopping

• The only significant cost to choosing the training time "hyperparameter" is running the validation set evaluation periodically during training.

• An additional cost to early stopping is the need to maintain a copy of the best parameters; this cost is usually negligible, because it is acceptable to store these parameters in a slower and larger form of memory.

• Early stopping is an "unobtrusive" form of regularization – it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values (this is in contrast to weight decay).

There are (2) conventional schema for early stopping:

(1) Initialize the model again and retrain on all the data; however, there is not a good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset.

(2) Another strategy is to keep the parameters obtained from the first round of training and then continue training, but now using all the data; this strategy avoids the high cost of training the model from scratch.

Sparse Representations

• Weight decay acts by placing a penalty directly on the model parameters; another strategy is to place a penalty on the activations of the units in a NN, encouraging their activations to be sparse. This indirectly imposes a complexity penalty on the model parameters.

• Recall that L^1 regularization induces a *sparse parameterization* – meaning that many of the parameters become zero (or close to zero). **Representational sparsity** on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero).

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ 9 \end{bmatrix} = \begin{bmatrix} 7 & 0 & 0 & -2 \\ 0 & 0 & -1 & 0 \\ 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ 1 \end{bmatrix}$$

Sparsely parameterized linear regression model (y=Ax)

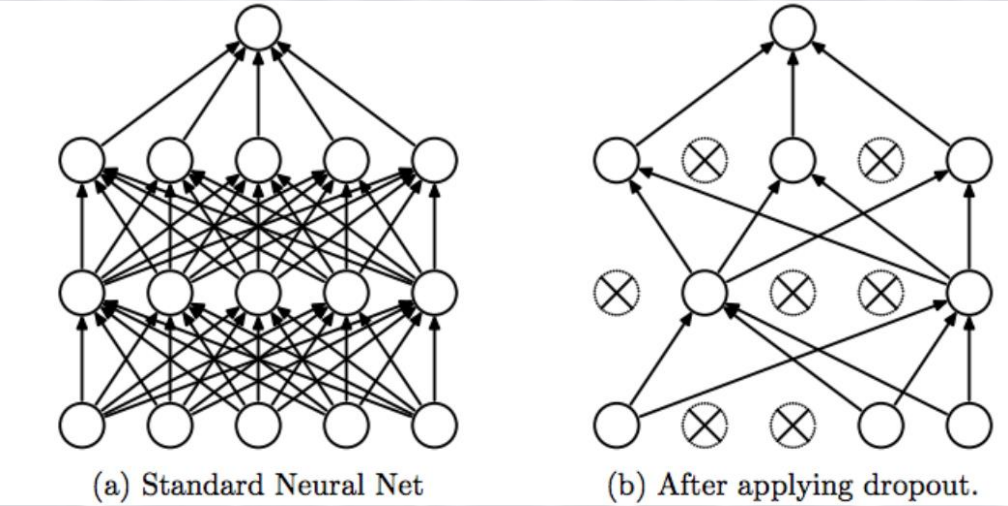
$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ 9 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 \\ 4 & 2 & 3 & -1 \\ 3 & 1 & 2 & -3 \\ -5 & 4 & 2 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

Linear regression with a sparse representation h of the data x; (y=Bh)

- One can achieve representational sparsity with a norm penalty by setting: $\Omega(\mathbf{h}) = \|\boldsymbol{h}\|_1$; yet another method is to directly formulate a constrained optimization problem:

$$\arg\min_{\mathbf{h}, \|\mathbf{h}\|_0 < k} \|\mathbf{x} - \mathbf{W}\mathbf{h}\|^2$$

Dropout



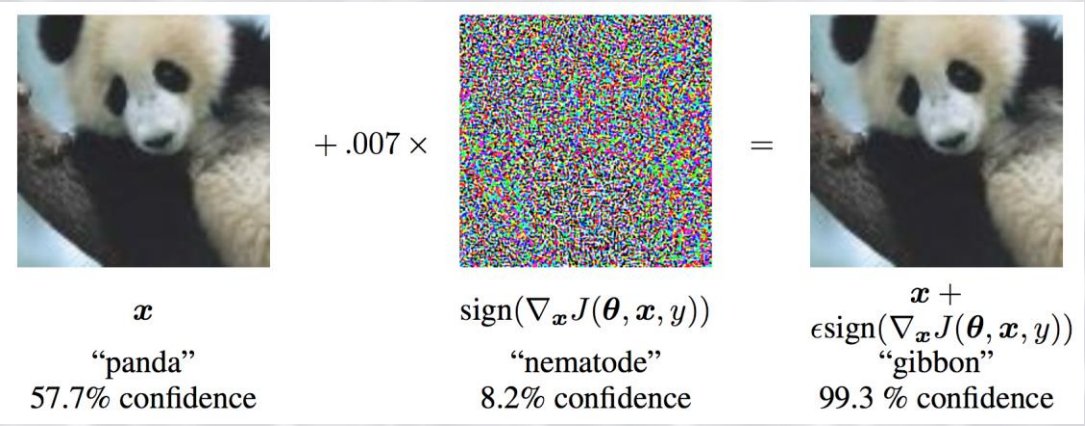
• Dropout (Srivastava et al., 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models (it is akin to *bagging*).

• Dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network. Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of NNs.

• In practice, each time we load an example into a minibatch for training, we randomly sample a different binary mask to apply to all input and hidden units in the network; the mask is sampled independently for each unit (e.g. 0.8 probability for including an input unit and 0.5 for hidden units).

• In the case of bagging, the models are all independent; for dropout, the models share parameters.

Adversarial Training



• Szegedy et al. (2014) found that even NNs that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally construction by using an optimization procedure to search for an input x' near a data point x such that the model output is very different from x' (oftentimes such **adversarial examples** are indiscernible to humans).

• In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed examples from the training set.

• Goodfellow et al. (2014), showed that one of the primary cause of these adversarial examples is excessive linearity. NNs are primarily built out of linear parts, and so the overall function that they implement proves to be highly linear as a result.

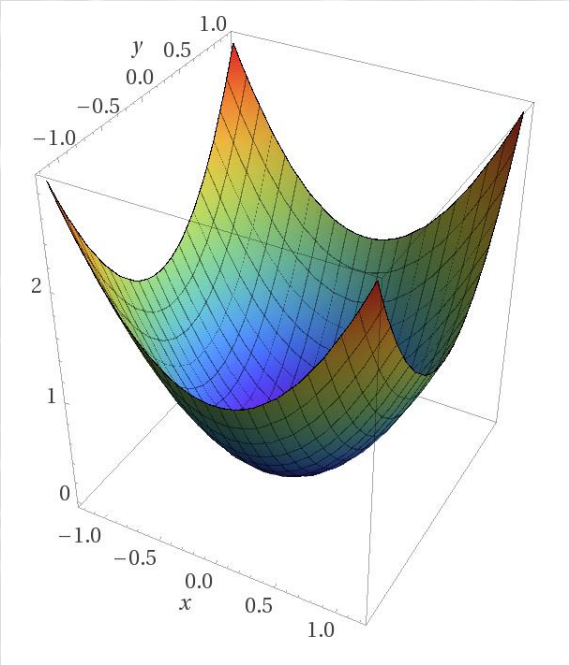
• These linear functions are easily optimized; unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data.

• Adversarial training help to illustrate the power of using a large function family in combination with aggressive regularization – a major theme in contemporary deep learning.

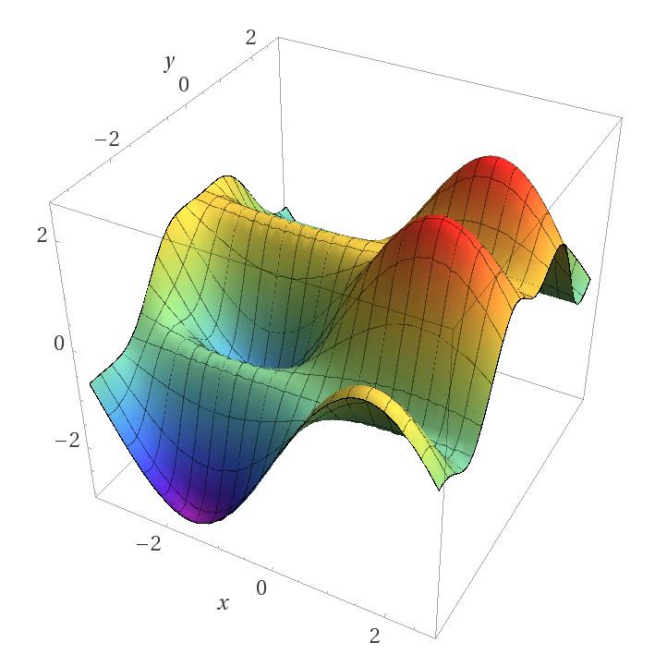
Challenges for DNN Optimization

• Traditionally, ML implementations avoid the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex.

• When training NNs, however, we must confront **the general non-convex case**.



Convex Function



Non-Convex Function

Challenges for DNN Optimization: Ill-Conditioning

• A mathematical problem is ill-conditioned if a small change in the independent variable (input) leads to a large change in the dependent variable (output). This can lead to numerical and related computational problems. If a system of equations is ill-conditioned, the solution exists, but it if very difficult to find.

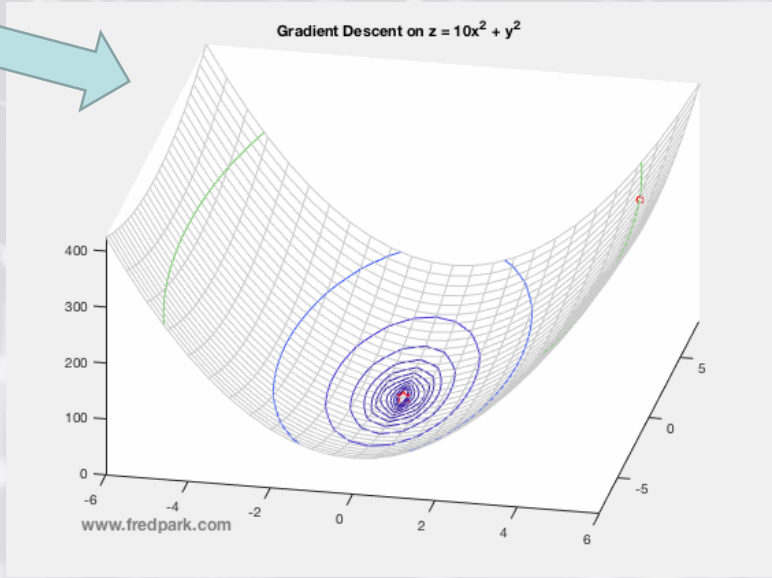
• More specifically, as it related to ML, if the Hessian matrix (wrt the model inputs and loss function) is ill-conditioned*, it means that the basins of the loss functions form elongated ellipsoids, rather than being close to "spherical."

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

* Optimization methods such as gradient descent will be slow to converge in this case, as they will render a protracted, zig-zagging path.

• The **condition number** of a matrix quantifies the degree to which a system is ill-conditioned:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$



Challenges for DNN Optimization: Local Minima

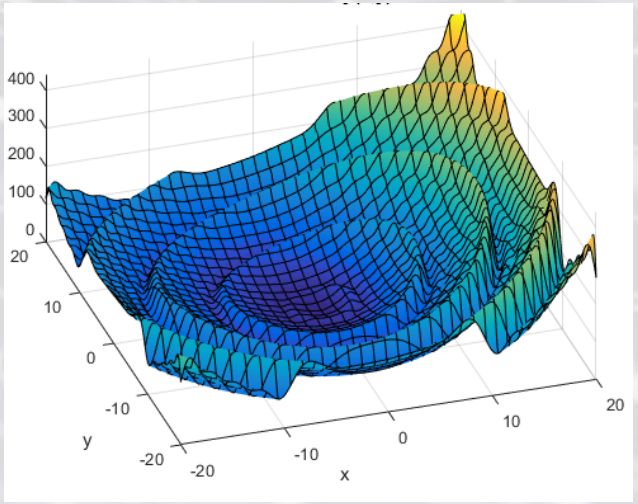
• For a convex function, any local minimum is guaranteed to be a global minimum.

• With non-convex functions, such as NNs, it is possible to have many local minima. Moreover, nearly any DNN is essentially guaranteed to have a very large number of local minimal (even uncountably many).

• One of the chief reasons for the presence of many local minima for NNs, is due to the problem of model identifiability. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the mode's parameters.

• Models with latent variables (e.g. hidden neurons) are not in general identifiable because we can obtain equivalent models by exchanging latent variables with one another. In addition, in maxout and RELU networks, for instance, one can arbitrarily scale the incoming/outgoing weights and biases to achieve non-identifiability.

• Local minima are problematic if they correspond with high cost (vis-à-vis the global minimum).



Challenges for DNN Optimization: Plateaus, Saddle Points

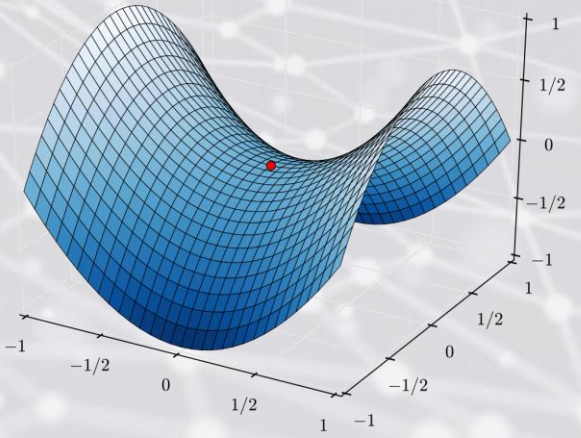
• For many high-dimensional, non-convex functions, local minima (and maxima) are in fact rare compared to **saddle points**.

• Some points around a saddle point have greater cost than the saddle point, while others have lowers cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues.

• Why are saddle points more common than local extrema in high dimensions? The basic intuition is this: in order to render a local extreme value, all of the eigenvalues must be of the same sign (naturally, this is very unlikely – all things being equal – in high dimensions).

• In fact, eigenvalues of the Hessian are more likely to be positive as we reach regions of lower cost; this means that local minima are much more likely to have low cost than high cost. For first-order optimization, saddle points are not necessarily a significant problem (Goodfellow); however, for second-order methods, they clearly constitute a problem.

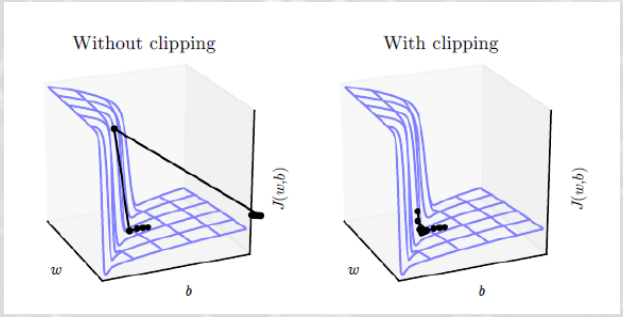
• Degenerate locations such as *plateaus* can pose major problems for all numerical algorithms.

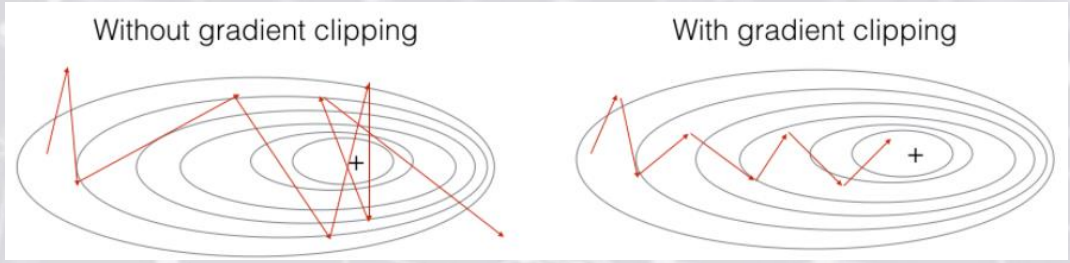


Challenges for DNN Optimization: Cliffs, Exploding and Vanishing Gradients

• NNs with many layers often have extremely steep regions resembling cliffs. This is due to the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can alter the parameters drastically.

• **Gradient clipping**, a heuristic technique, can help avoid this issue. When the traditional gradient descent algorithm proposes making a large step, the gradient clipping heuristic intervenes to reduce the step size, thereby making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.





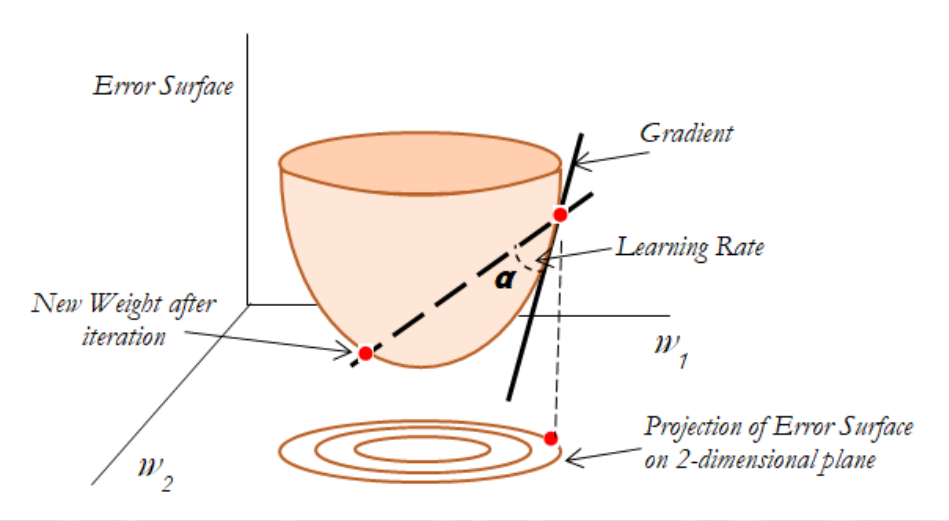
• When the computational graph for a NN becomes very large (e.g. RNNs), the issue of exploding/vanishing gradients can arise. Vanishing gradients make it difficult to known which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

*LSTMs, RELU, and ResNet (Microsoft) have been applied to solve the vanishing gradient problem.

Basic Algorithms: SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while



Basic Algorithms: SGD

• **Stochastic Gradient Descent** (SGD) and its variants are some of the most frequently used optimization algorithms in ML. Using a minibatch of i.i.d. samples, one can obtain an unbiased estimate of the gradient (where examples are drawn from the data-generating distribution).

•A crucial parameter for the SGD algorithm is the **learning rate**, ε. In practice, it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum.

A sufficient condition to guarantee convergence of SGD is that:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

In practice, it is common to decay the learning rate linearly until iteration τ:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

* Note that for SGD, the computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large.

Momentum

• The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

• The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

• Formally, the momentum algorithm introduces a variable **v** that plays the role of *velocity* – it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

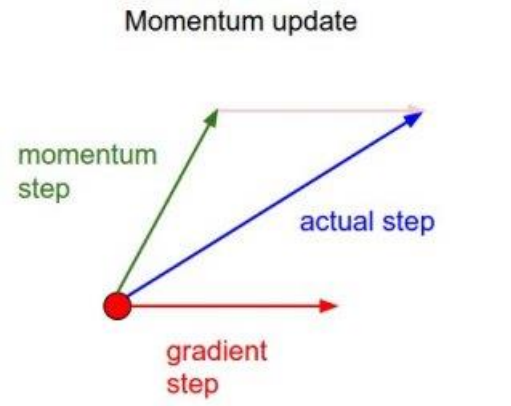
• The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. If the only force is the gradient of the cost function, then the particle might never come to rest. To resolve this problem, we add one other force, proportional to v(t); in physics terminology this force corresponds to *viscous drag*, as the if the particle must push through a resistant medium such as syrup.

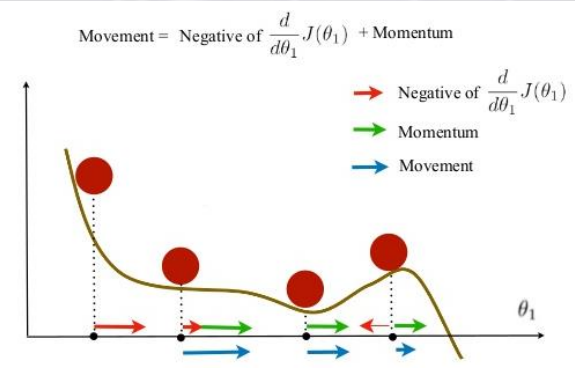
• The velocity **v** accumulates the gradient elements; the larger alpha is relative to epsilon, the more previous gradients affect the current direction.

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while





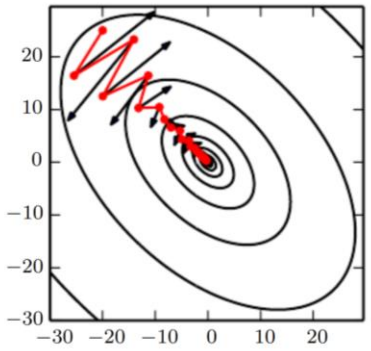


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

Weight Initialization

• Training algorithms for DNN models are usually iterative and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.

• The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficultiess and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost.

• Modern initialization strategies are usually simple and heuristic; designing improves initialization strategies is a difficult task because NN optimization is not yet well understood.

• The most general guideline agreed upon by most practitioners is known as "**symmetry-breaking**." If two hidden units with the same activation function are connected to the same inputs, then these units have different initial parameters. If the training is deterministic, "symmetric" units will update identically (and hence be useless); even if the training is stochastic, it is usually best to initialize each unit to compute a different function from all the other units.

Weight Initialization

• The goal of having each unit compute a different function motivates random initialization of the parameters. Moreover, random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper than explicitly searching for, say a large set of basis functions that are all mutually different from one another.

• Typically, the biases for each unit are set to heuristically chosen constants, and we only initialize the weights randomly. It is common practice to initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution.

• Note that the scale of the initial distribution does have a large effect on both the outcome of the optimization procedure and the ability of the network to generalize.

• Larger initial weights will yield a strong symmetry-breaking effect, helping to avoid redundant units; in addition, they will also potentially help avoid the problem of vanishing gradients. Nevertheless, they may conversely exacerbate the exploding gradient problem; in RNNs, large initial weights can manifest *chaotic behavior*.

* **Sparse initialization** (Martens, 2010) fixes the number of non-zero weights for initialization; **Xavier initialization** draws random initial values from a distribution with zero mean and variance inversely proportional to the size of the previous layer in the network.

Algorithms with Adaptive Learning Rates

• It is well known that the learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance. The cost function is often highly sensive to some directions in parameters space and insensitive to others.

• While the momentum algorithm mitigates these issues somewhat, it does so at the expense of introducing another hyperparameters.

• Recently, a number of incremental methods have been introduced that adapt the learning rates of model parameters.

AdaGrad

• The **AdaGrad** algorithm (Duchi et al, 2011) individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient.

• The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivates have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

*Note: empirically, for training DNNs, the accumulation of squared gradients *from the beginning of training* can result in premature and excessive decrease in the effective learning rate.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSProp

• The **RMSProp** algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially-weighted moving average. Where AdaGRad shrinks the learning rate according to the entire history of the squared gradient, RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl.

• Empirically, RMSProp has been to shown to be an effective and practical optimization algorithm for DNNs.

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

Adam

• **Adam** (Kingman and Ba, 2014) is another adaptive learning rate optimization algorithm ("adaptive moments"). It can be seen as a variant on the combination of RMSProp and momentum with several distinctions.

• First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

• RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. *Adam is generally regarded as being fairly robust to the choice of hyperparameters.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Second-Order Methods: Newton's Method

• Newton's method is a classical second-order iterative approximation method. In contrast to first-order methods, second-order methods make use of second derivatives (i.e. the curvature of the loss function) to improve optimization.

• Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate J(theta) near some point θ_0, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Where **H** is the Hessian of J wrt θ evaluated at θ_0. If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - H^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Second-Order Methods: Newton's Method

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \qquad \boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - H^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

• If the objective function is convex but not quadratic, this update can be iterated, yielding a training algorithm. For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step procedure: (1) update or compute the inverse Hessian; (2) update the parameters according to the equation above.

* In deep learning, the surface of the objective function is usually non-convex; with many features and potential saddle points, this is a potential problem for Newton's Method.

• Commonly, researchers apply a regularization strategy, for which the update becomes (this regularization is used in approximations to Newton's Method including the *Levenberg-Marquardt algorithm*):

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0) \right) + \alpha \mathbf{I} \right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$$

• Beyond the challenges of saddle points, the application of Newton's method for training large NNs is limited by its significant computational requirements; ostensibly, Newton's method requires the inversion of a matrix ($O(n^3)$); as a consequence, only networks with a very small number of parameters can be practically trained via Newton's method.

Second-Order Methods: Newton's Method

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

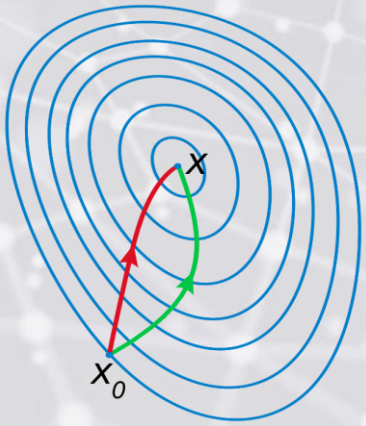
Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

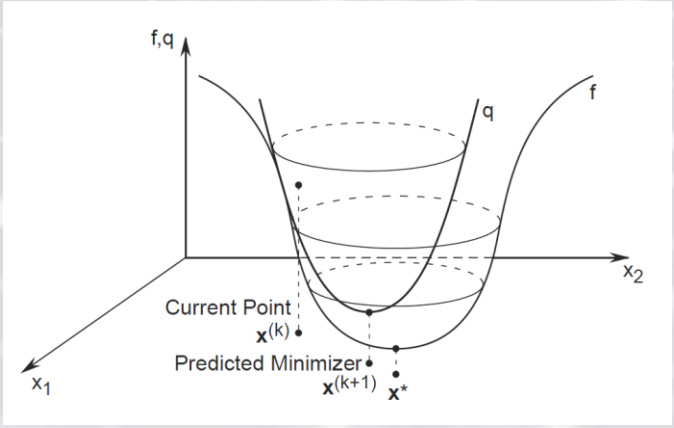
Compute Hessian inverse: $\boldsymbol{H}^{-1}$

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

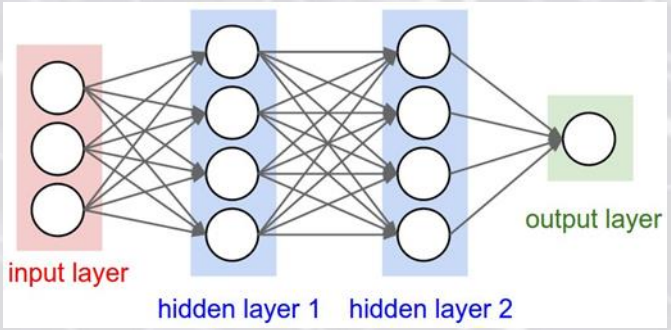
end while

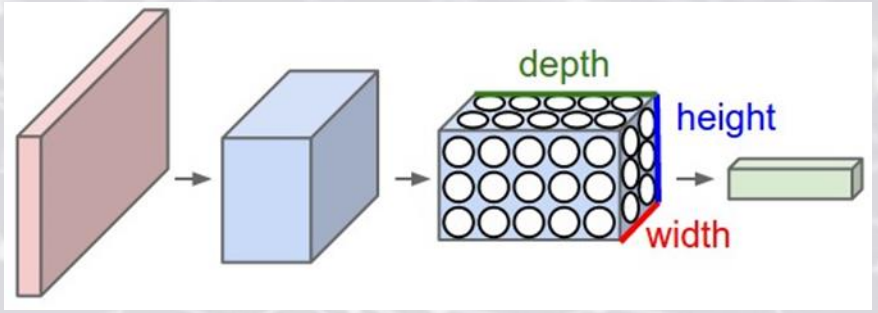




Convolutional Neural Networks

- So what is the fundamental difference between a generic NN and a CNN?
- CNNs conventionally take images as inputs. Recall that NNs (a fortiori: fully-connected NNs) do not scale well for high dimensional data!
- CNNs take advantage of the fact that the input consists of images and the architecture is constrained accordingly.
- In particular, unlike the layers of a conventional NN, CNNs have neurons arranged in 3 dimensions, including depth (usually corresponding with color channels). As such, neurons in a CNN are only connected to a small region of the layer before it, instead of in a fully-connected manner.





- **Left**: A regular 3-layer Neural Network. **Right**: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Convolutional Neural Networks

- CNNs consist of a sequence of (3) types of layers, in general: Convolutional Layers, Pooling Layers and Fully-Connected Layers. These layers are stacked to form a full CNN architecture.

Example Architecture:

INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.

RELU layer will apply an elementwise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]). (can help with 'vanishing gradient' problem)

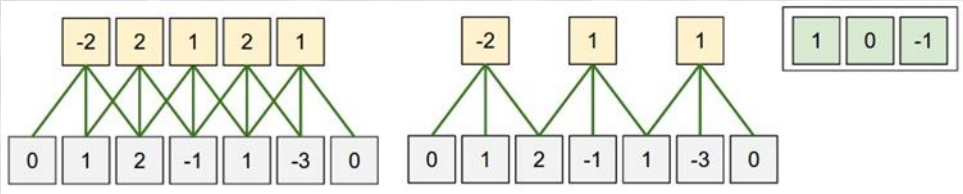
POOL layer will perform a *downsampling* operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].

FC (*i.e.* fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories for images. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

- Note that for training, the parameters in the CONV/FC layers are trained with gradient descent so that the class scores that the CNN computes are consistent with the labels in the training set for each image.

Convolutional Neural Networks

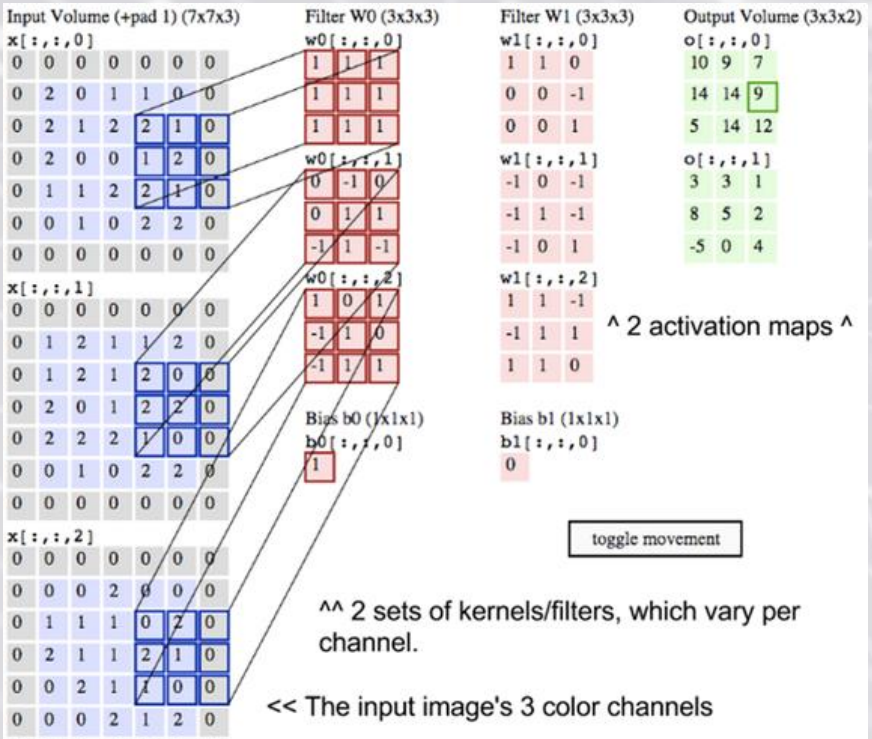
- Schematics of *spatial arrangement*.



- In the leftmost example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. **Left (top):** The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. **Right (top):** The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3.

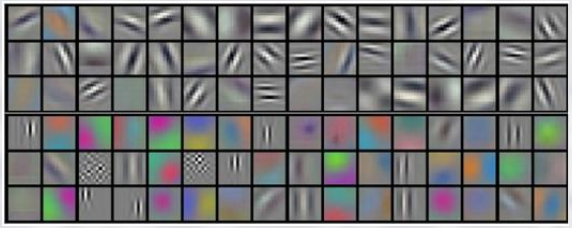
- The neuron weights are in this example [1,0,-1] (shown on very right), and its bias is zero. These weights are shared across all yellow neurons.

- Weight/parameter sharing is a common (and key) feature of CNNs, and is used to control the number of parameters.



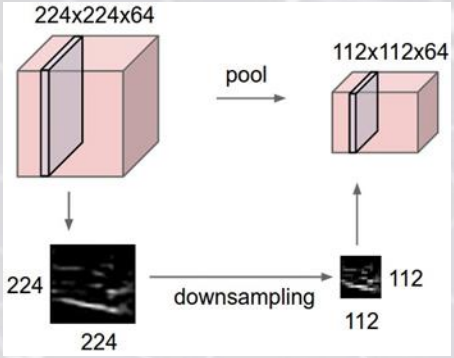
Convolutional Neural Networks

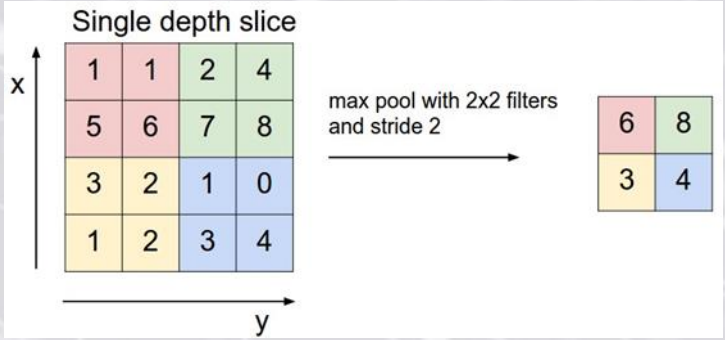
- *Learned filters*



- Example filters learned by Krizhevsky *et al.* Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

- Schematic of *pooling*

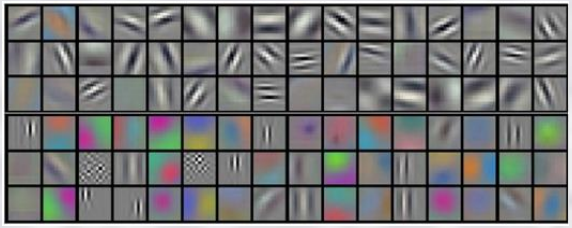




- Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

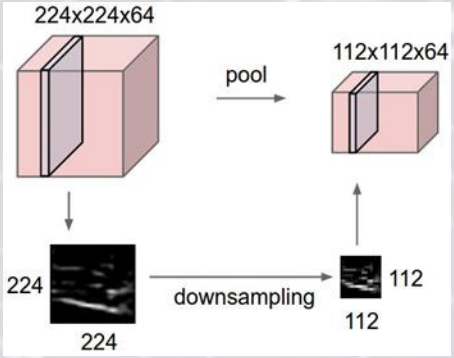
Convolutional Neural Networks

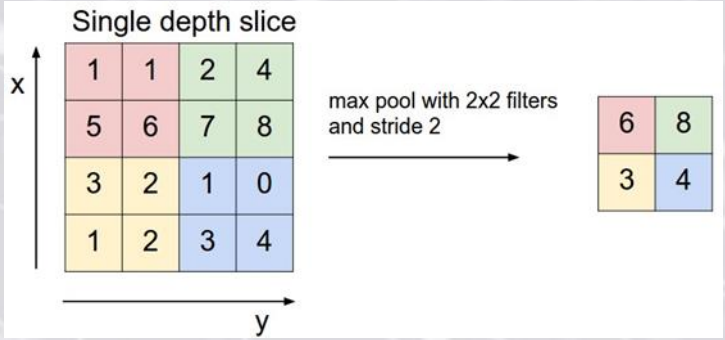
- *Learned filters*



- Example filters learned by Krizhevsky *et al.* Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

- Schematic of *pooling*





- Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

One-Shot Learning: Siamese Networks

• Typically, with deep learning, we require a large amount of data, and the quality of our results generally scales with the size (and quality) of our dataset.

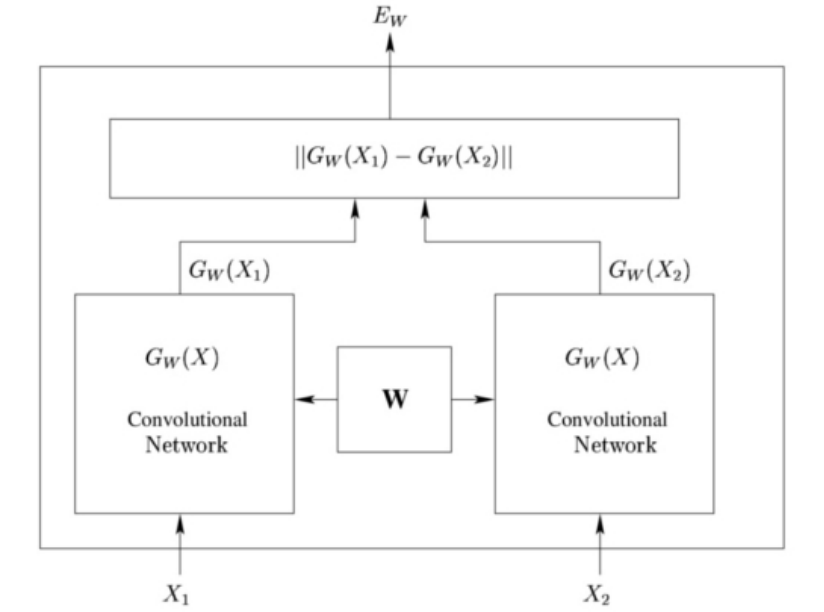
• An alternative to this "big data" paradigm, however, is **one-shot learning**. In this paradigm we learn from only a few (even just one) example. One can plausibly argue that a great deal of real-world, "biological" learning also occurs in a "low data" regime.

• Consider the problem of facial recognition. We would like to determine whether an individual is a member of a database, based on only a single instance/photo (e.g. security applications).

• One conventional approach to this problem is to train a CNN for the image processing task. However, CNNs cannot be trained effectively with very small datasets; in addition, it would be highly cumbersome to retrain the model every time we encounter a new individual.

• A Siamese network will, by contrast, allow us to solve this problem.

One-Shot Learning: Siamese Networks



• A Siamese neural network uses two identical sub-networks (e.g. pretrained CNNs) in tandem, with the overall objective to determine how similar two comparable things are (e.g. signature verification, face recognition.). The sub-networks have the same parameters and weights.

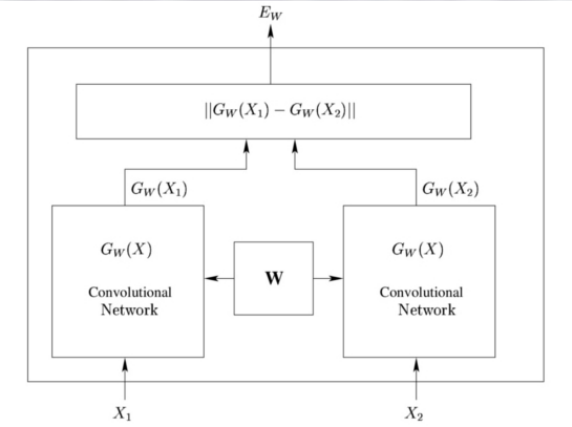
• Each sub-network is fed an input (e.g. an image of a face), producing the respective outputs. If the distance between the two encodings:

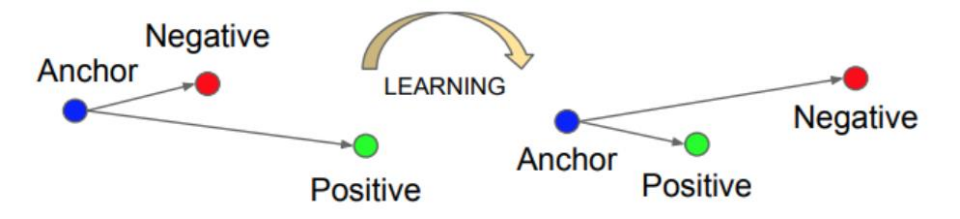
$$\left\| G_W(X_1) - G_W(X_2) \right\|$$

is less than some threshold (i.e. a hyperparameter), we consider the images to be the same, otherwise they are different.

https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf

One-Shot Learning: Siamese Networks





• To train a Siamese network we can apply gradient descent on a **triplet loss function** which is simply a loss function using three images: an anchor image **A**, a positive image **P**(same person as the anchor), as well as a negative image **N** (different person than the anchor). So, we want the distance d(A, P) between the encoding of the anchor and the encoding of the positive example to be less than or equal to the distance d(A, N) between the encoding of the anchor and the encoding of the negative example. In other words, we want pictures of the same person to be close to each other, and pictures of different persons to be far from each other.

• The problem here is that the model can learn to make the same encoding for different images, which means that distances will be zero, and unfortunately, it will satisfy the triplet loss function. For this reason, we add a margin α (a hyperparameter), to prevent this from happening, and to always have a gap between A and P versus A and N.

$$d(A,P) + \alpha \leq d(A,N)$$

$$\|f(A)\text{-}f(P)\|^2 + \alpha \leq \|f(A)\text{-}f(N)\|^2$$

$$\boxed{\|f(A)\text{-}f(P)\|^2 \text{-} \|f(A)\text{-}f(N)\|^2 + \alpha \leq 0}$$

One-Shot Learning: Siamese Networks

Define the triplet loss function:

$$L(A, P, N) = \mathbf{max}\left(\|f(A)\text{-}f(P)\|^2 \text{-} \|f(A)\text{-}f(N)\|^2 + \alpha,\ 0\right)$$

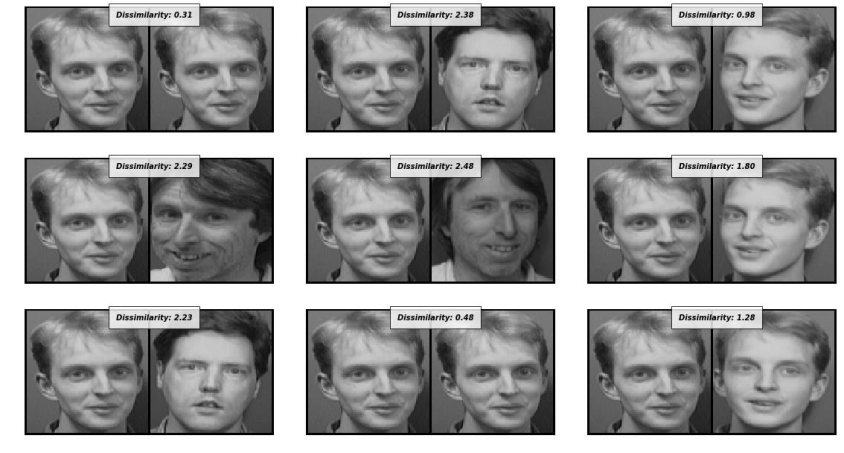
The max means as long as d(A, P)—d(A, N)+ alpha is less than or equal to zero, the loss L(A, P, N) is zero, but if it is greater than zero, the loss will be positive, and the function will try to minimize it to zero or less than zero.

The cost function is the sum of all individual losses on different triplets from all the training set:

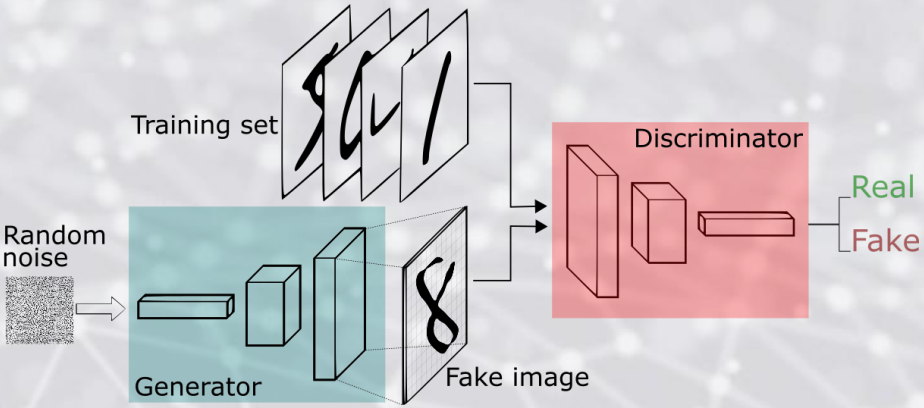
$$\text{Cost function: } J = \sum_{i=1}^{n} L(A^{(i)}, P^{(i)}, N^{(i)})$$

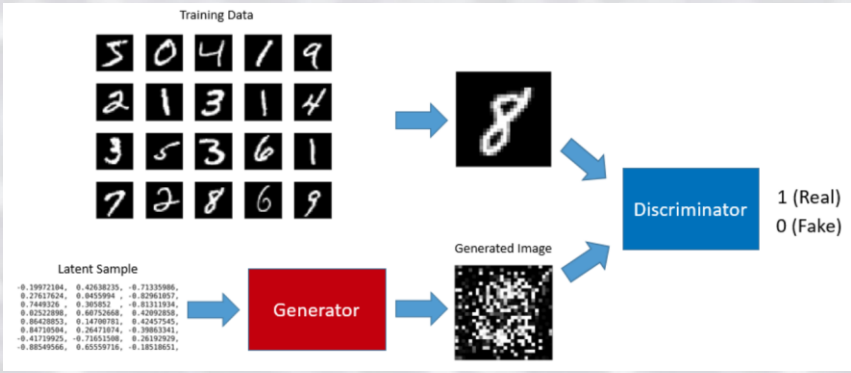
The training set should contain multiple pictures of the same person to have the pairs A and P, then once the model is trained, we'll be able to recognize a person with only one picture.

If we choose the triplets for training at random, it will be easy to satisfy the constraint of the loss function because the distance is going to be generally large; in this case gradient descent will not learn much from the training set. For this reason, we need to find A, P, and N so that A and P are so close to N. Our objective is to make it harder to train the model to push the gradient descent to learn more.



GANs: Generative Adversarial Networks





for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

https://arxiv.org/abs/1406.2661

GANs: Generative Adversarial Networks

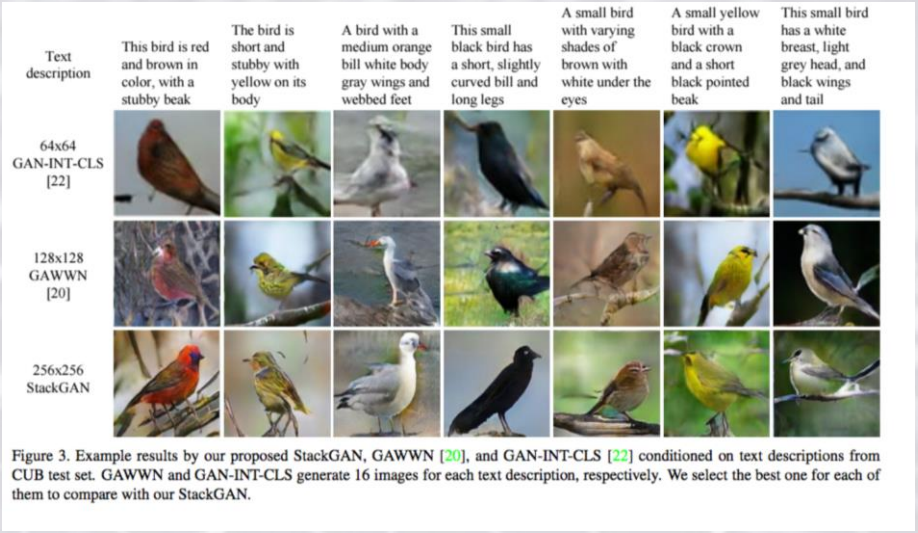
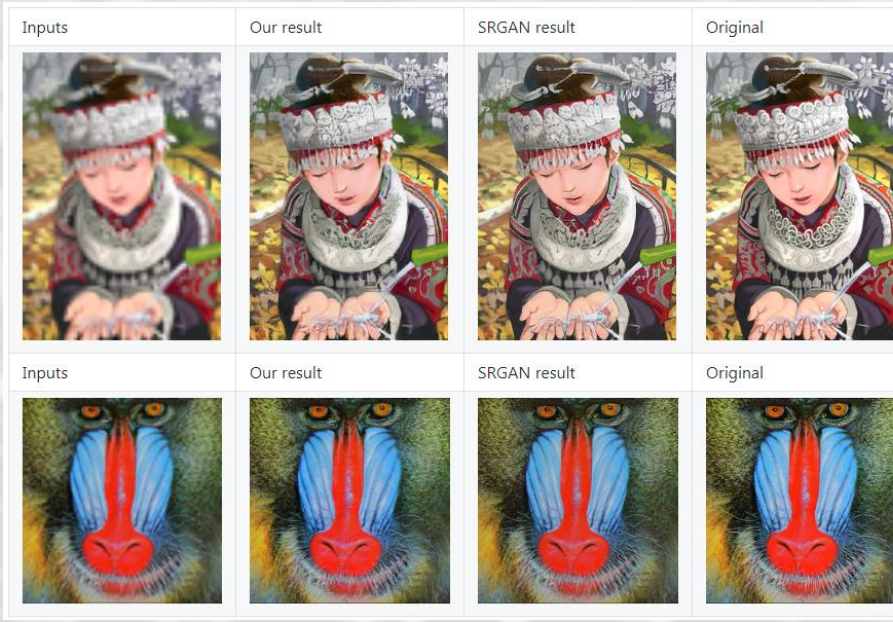
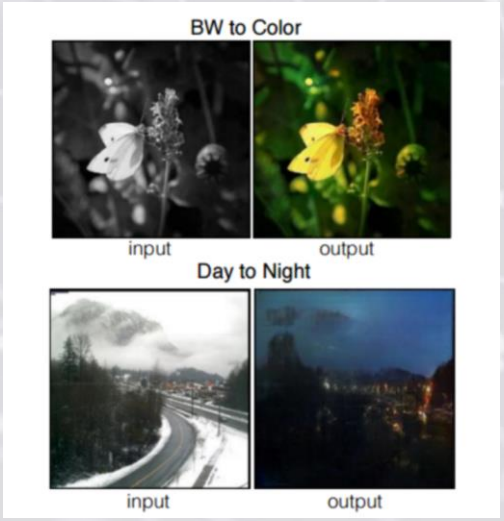


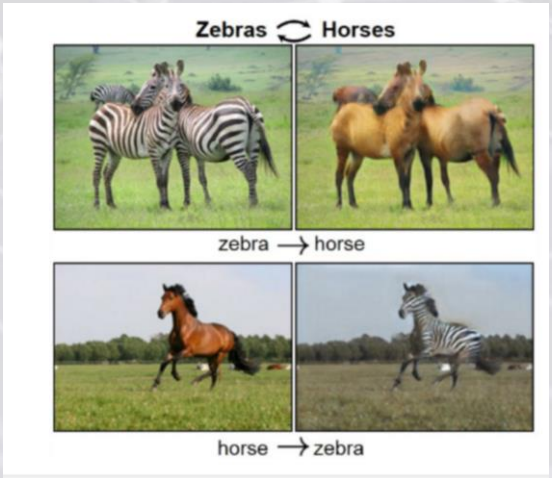
Figure 3. Example results by our proposed StackGAN, GAWWN [20], and GAN-INT-CLS [22] conditioned on text descriptions from CUB test set. GAWWN and GAN-INT-CLS generate 16 images for each text description, respectively. We select the best one for each of them to compare with our StackGAN.

https://arxiv.org/abs/1605.05396



https://arxiv.org/pdf/1609.04802.pdf





https://arxiv.org/pdf/1612.00005.pdf

https://junyanz.github.io/CycleGAN/

Fin