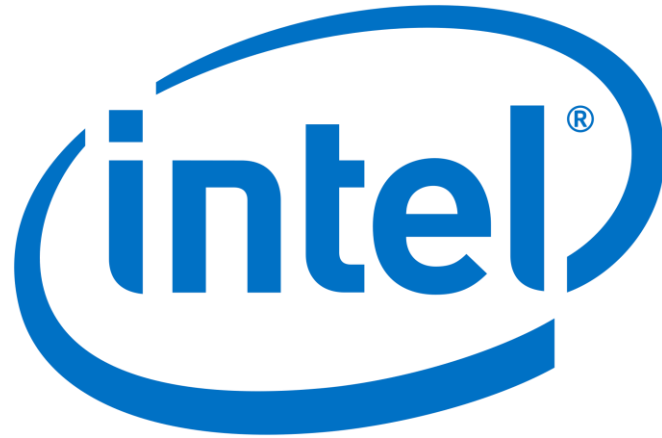


An Introduction to Deep Learning

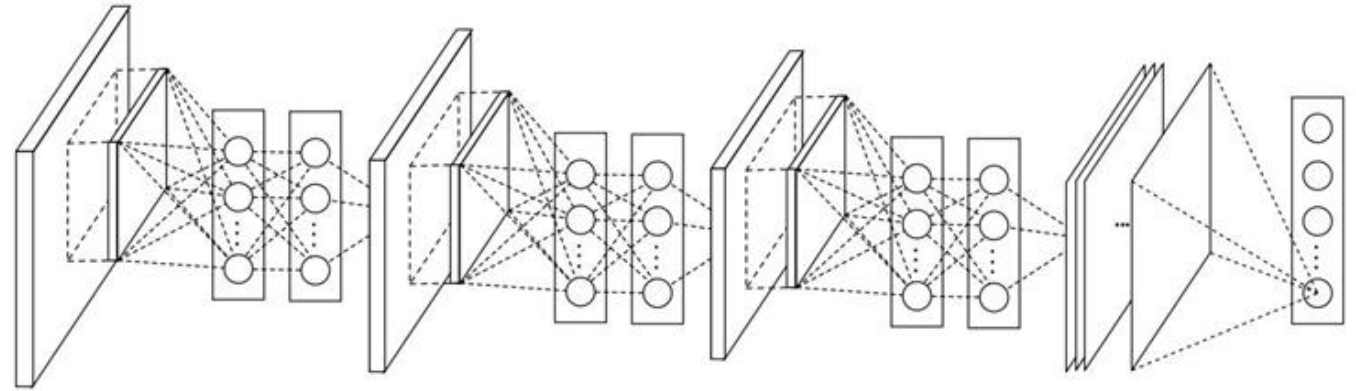
Anthony D. Rhodes

7/2018



Contents

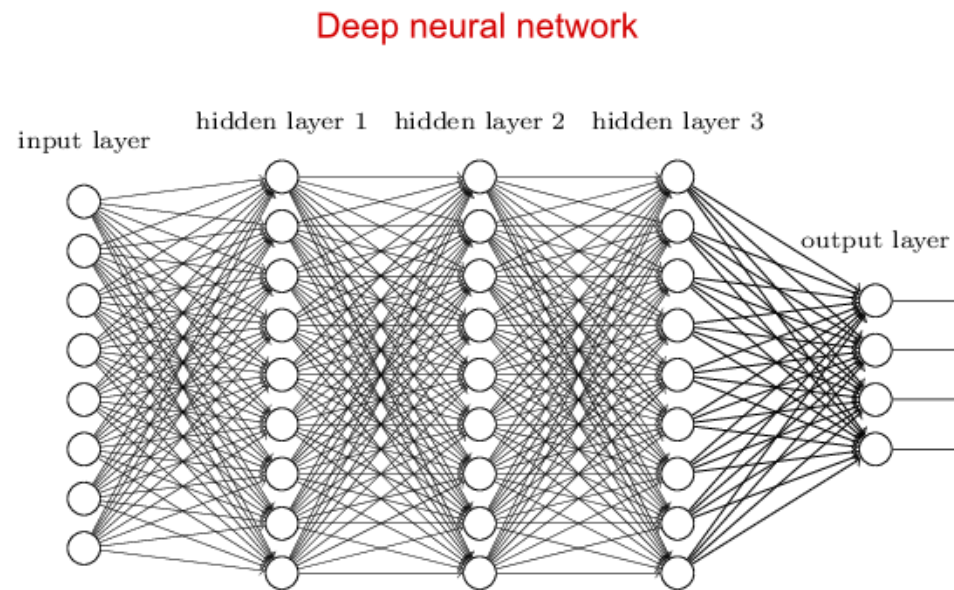
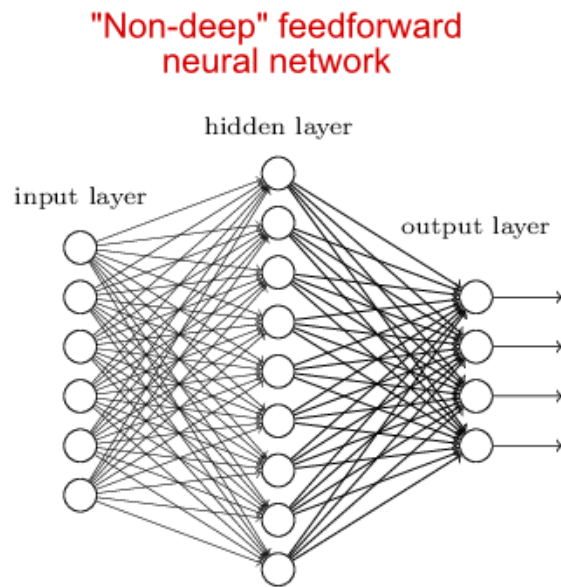
- Overview: Neural Nets
- Activation functions, Regularization
- Deep Learning Challenges
- SGD, Momentum, Parameter Initialization, adaptive learning rate algorithms
- CNNs
- Siamese Networks and One-Shot Learning, Similarity Learning
- Deep Learning: Practical Considerations



Neural Networks

(*) A Neural Network (NN) consists of a network of McCulloch/Pitts computational neurons (a single layer was known historically as a “perceptron.”)

(*) NNs are *universal function approximators* – meaning that they can learn any arbitrarily complex mapping between inputs and outputs. While this fact speaks to the broad utility of these models, NNs are nevertheless prone to **overfitting**. The core issue in most ML/AI models can be reduced to the question of **generalizability**.



(*) A “deep” net has many hidden layers.

Neural Networks

(*) Each neuron receives some inputs, performs a dot product and optionally follow it with a non-linearity (e.g. sigmoid/tanh).

Training a NN entails tuning the weights of the network. The weights are commonly updated incrementally during training, so as to minimize a “loss function” (e.g. classification error). The goal of training is to produce a model with low generalization error.

Put simply, denote the **loss function** for an algorithm by:

$$L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

where f is the model output given input \mathbf{x} , $\boldsymbol{\theta}$ are the model parameters, and \mathbf{y} is the true output associated with \mathbf{x} . In general the loss function outputs 0 when the $f(\mathbf{x})=\mathbf{y}$ (i.e. the prediction was correct), and otherwise the loss is non-zero (a larger error yields a larger loss).

In ML literature, the **objective function** is the function we which to optimize (either maximize or minimize) that defines a learning algorithm.

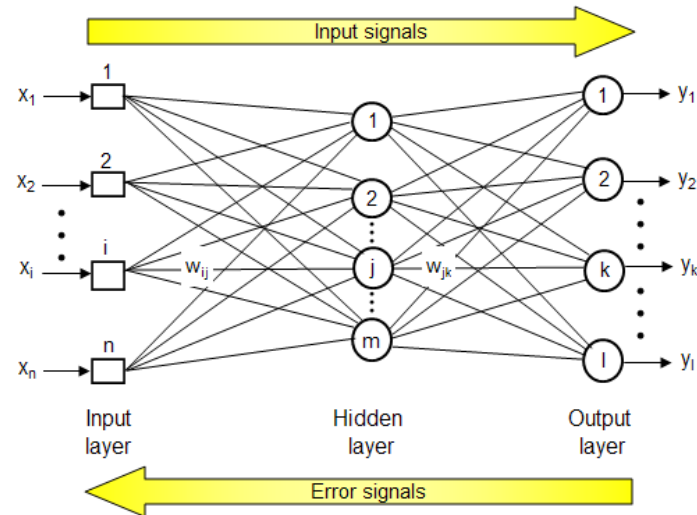
Most commonly, the objective function is denoted $J(\boldsymbol{\theta})$, as the mean of the training errors:

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$$

Neural Networks

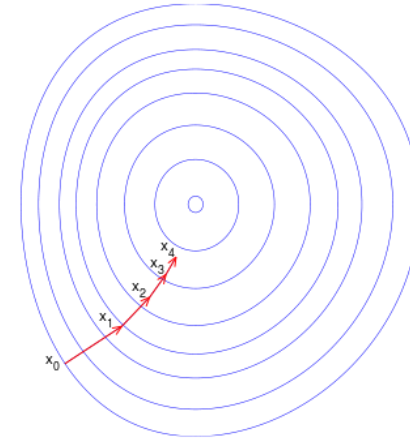
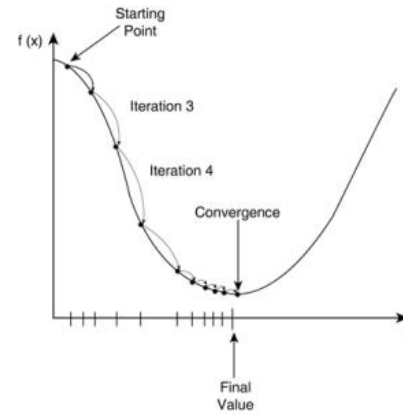
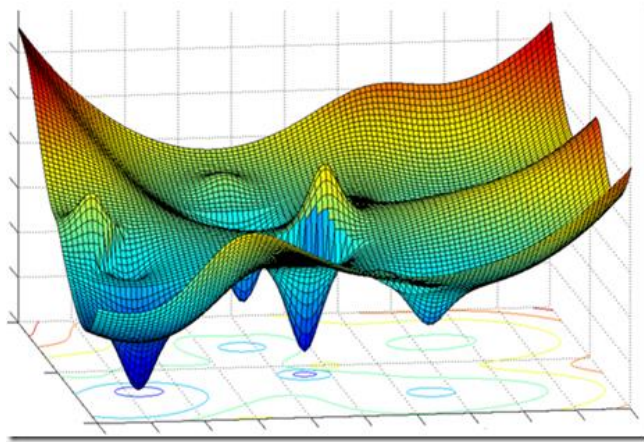
(*) Each neuron receives some inputs, performs a dot product and optionally follow it with a non-linearity (*e.g.* sigmoid/tanh).

Training a NN entails tuning the weights of the network. The weights are commonly updated incrementally during training, so as to minimize a “loss function” (*e.g.* classification error). The goal of training is to produce a model with low generalization error.



(*) NNs are typically trained using **backpropagation**. This method calculates the gradient of a loss function (*e.g.* squared-loss) with respect to all the weights (W) in the network. More specifically, we use the chain rule to compute the ‘delta’ for the weight updates (one can think of this delta as assigning a degree of ‘blame’ for misclassifications).

Gradient Descent



(*) Backpropagation is one particular instance of a larger paradigm of optimization algorithms known 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>

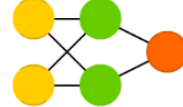
A Neural Network “Zoo”

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

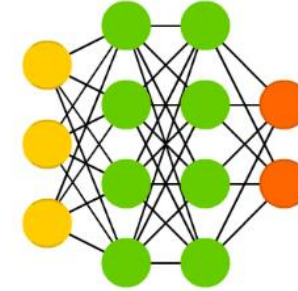
Perceptron (P)



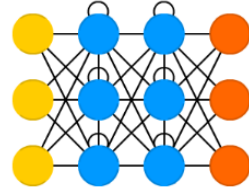
Feed Forward (FF)



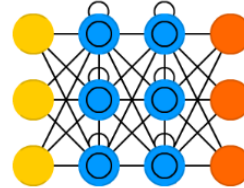
Deep Feed Forward (DFF)



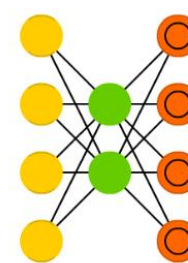
Recurrent Neural Network (RNN)



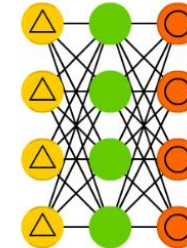
Long / Short Term Memory (LSTM)



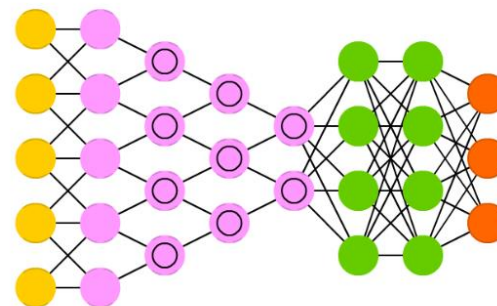
Auto Encoder (AE)



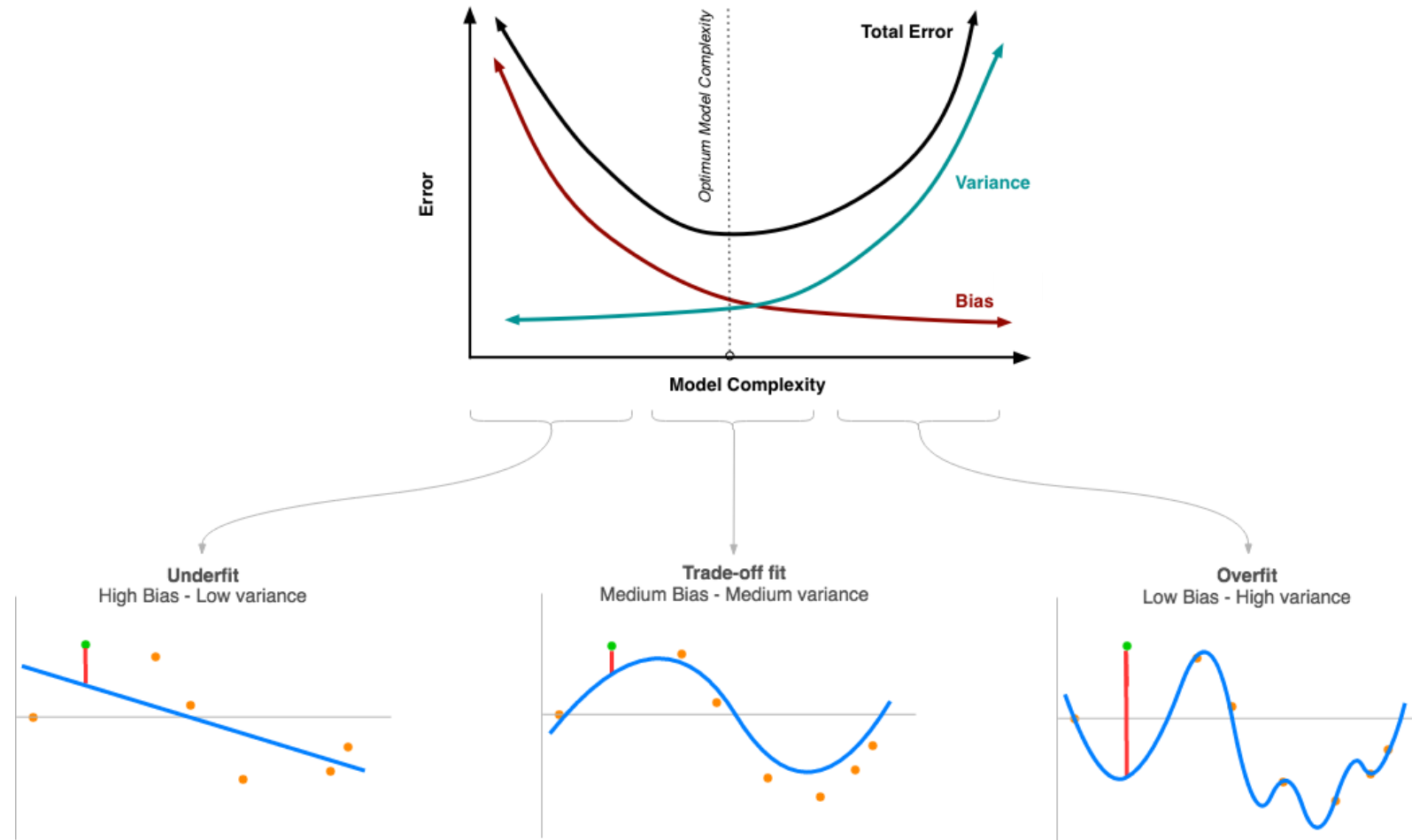
Denoising AE (DAE)



Deep Convolutional Network (DCN)



Overfitting, Underfitting and the Bias-Variance tradeoff



(*) Because it can accommodate very complex data representations, a deep neural network (DNN) is severely prone to overfitting (and thus poor generalization error); common remedies to overfitting include data augmentation and regularization, among other techniques.

Deep Learning

(*) **Fundamental idea for Deep Learning**: automate the process of learning a hierarchy of concepts -
- this approach obviates the need for human operators to formally specify the knowledge/parameters that a computer needs.

(*) A “deep” network enables learning a more complex/“expressive” model that can successfully discern useful patterns in data. In particular, deep learning enables the computer to build complex concepts from simpler concepts.

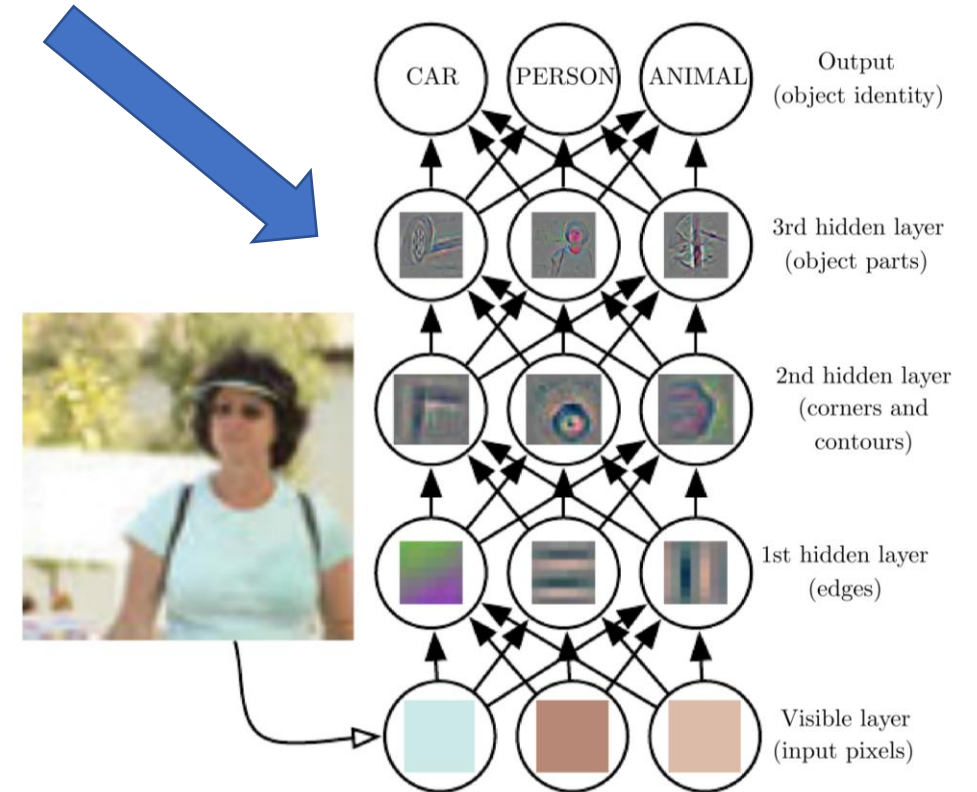
Deep Learning

(*) **Fundamental idea for Deep Learning**: automate the process of learning a hierarchy of concepts -
- this approach obviates the need for human operators to formally specify the knowledge/parameters that a computer needs.

(*) A “deep” network enables learning a more complex/“expressive” model that can successfully discern useful patterns in data. In particular, deep learning enables the computer to build complex concepts from simpler concepts.

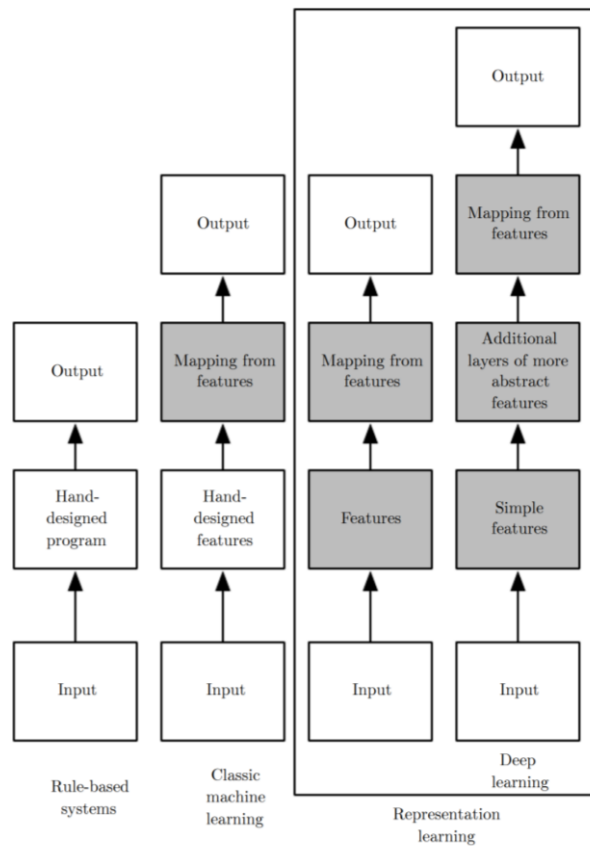
(*) Deep learning resolves the difficulty of learning a complicated mapping into a series of nested, simple mappings.

Credo of deep learning & data science: **more data**
/ quality data tends to trump specific design and model choices.

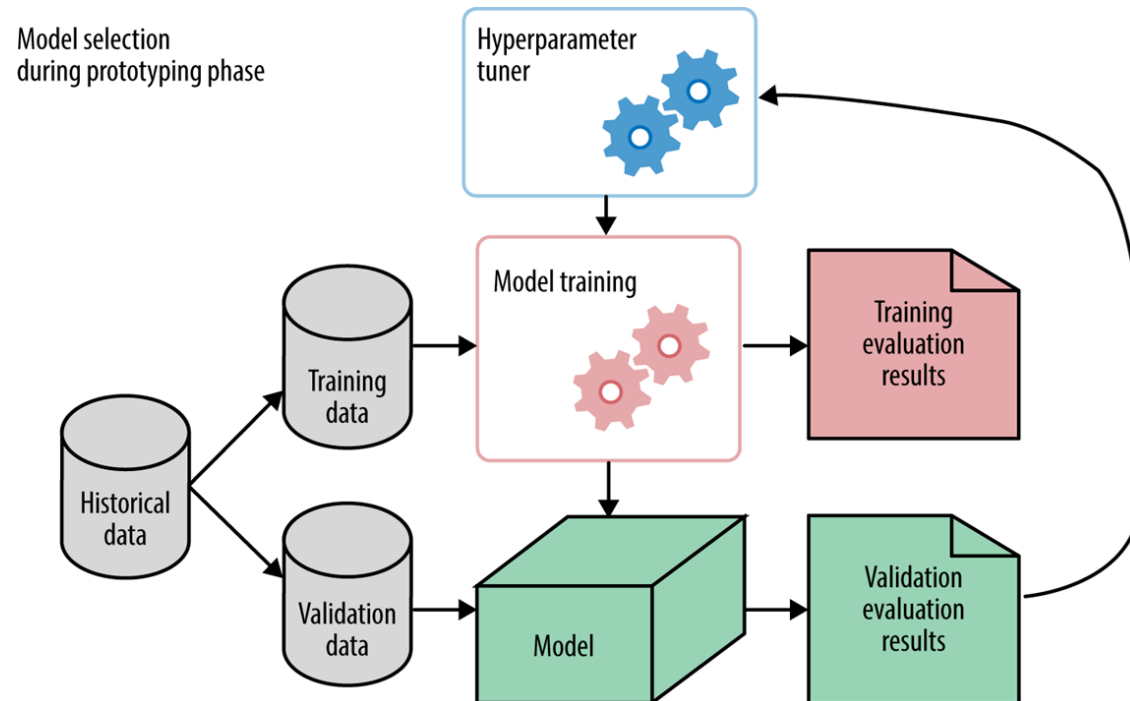


Deep Learning

(*)ML is perhaps the most viable approach known today for building AI systems that can solve complex problems for real-world environments; deep learning is a particular ML paradigm that assumes a representation of the world as a nested hierarchy of concepts.



Deep Learning Framework



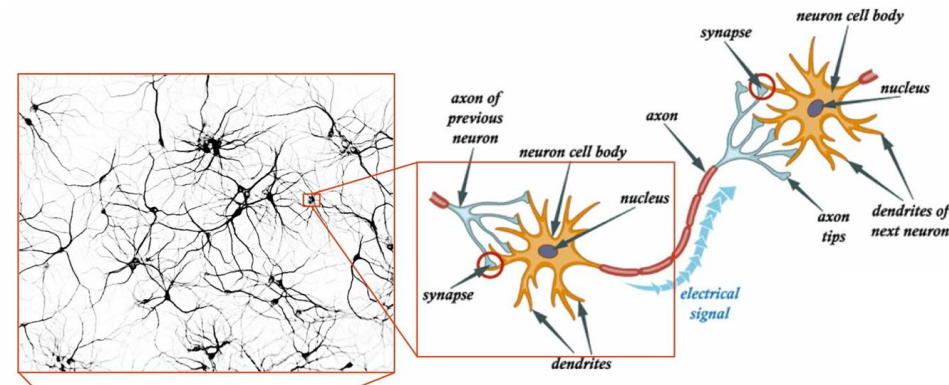
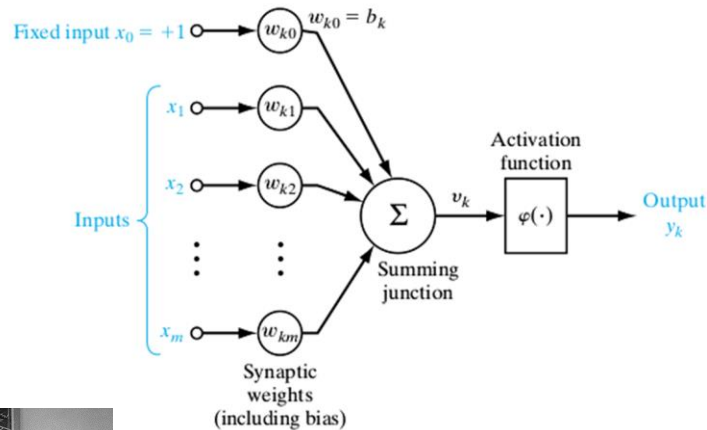
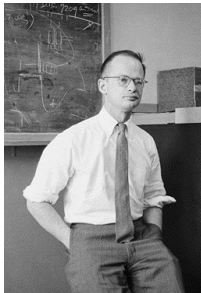
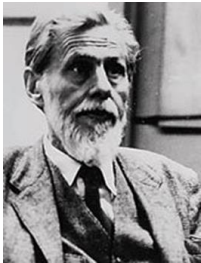
Standard ML Framework

A Very Brief History of NNs

(1) **First wave** (1940s/50s):

(*) Simple neural-computational model, inspired by early research in neuroscience: hope is that a single algorithm / architecture can solve a great variety of problems.

McCulloch & Pitts Neuron Model

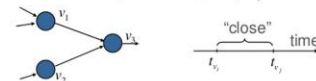


Hebb's Postulate

“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.

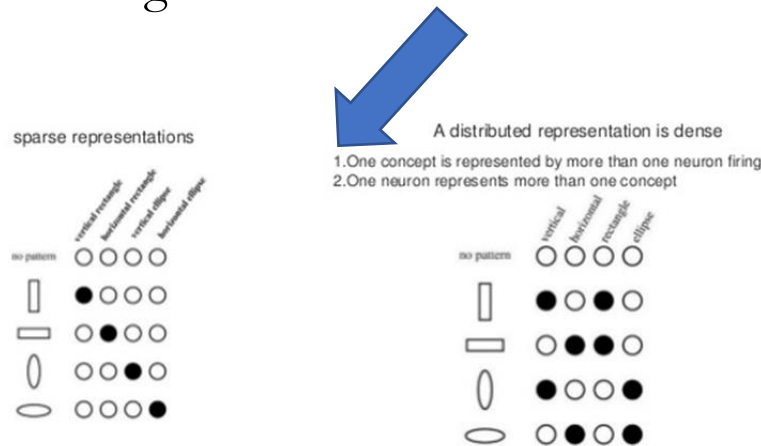


A Very Brief History of NNs

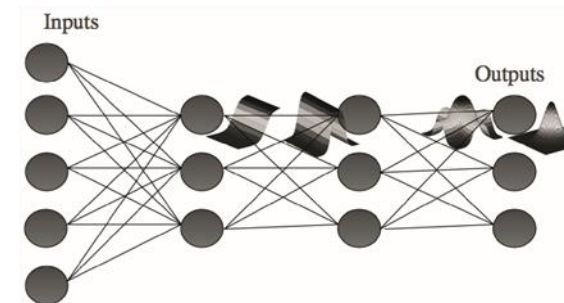
(2) **Second wave** (1980s-90s):

(*) Inspired, by cognitive science, **connectionism** / parallel distributed processing emerged as a dominant principle in NN research.

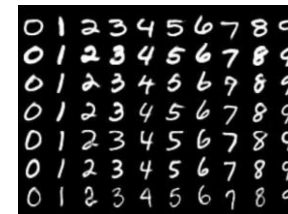
Central idea: a large number of simple computational units can achieve intelligent behavior when networked together.



Hinton et al. “rediscover” backprop algorithm (1986)



Universal Approximation Theorem (1989)

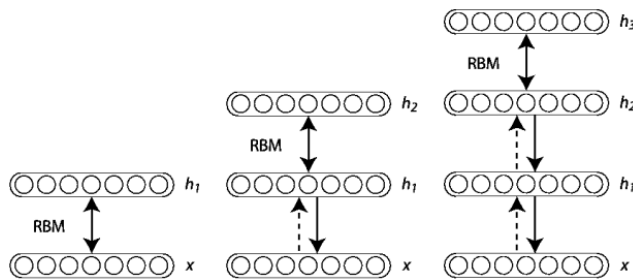


LeCun et al., handwritten digit recognition with CNNs (1995)

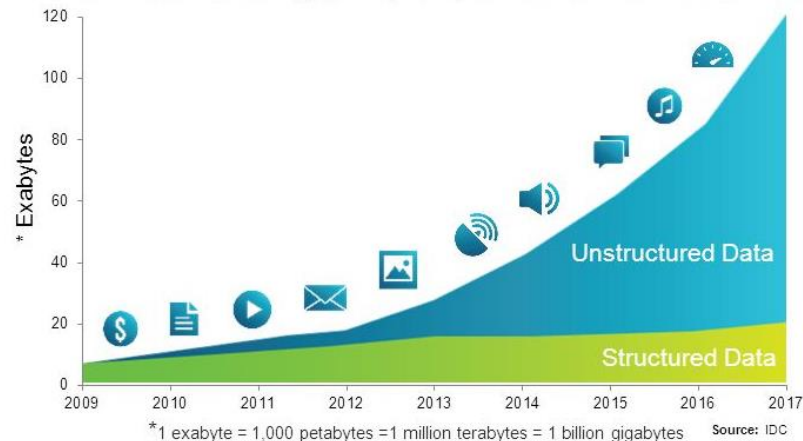
A Very Brief History of NNs

(3) **Third wave** (2000s-present):

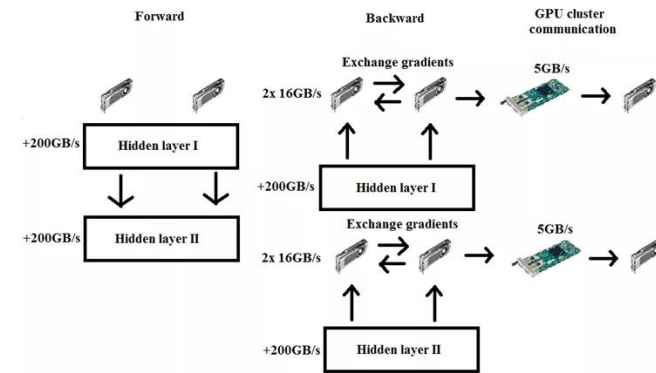
Following the success of backprop, NN research gained popularity and reached a peak in the early 1990s. Afterwards, other ML techniques became more popular until the modern deep learning renaissance that began in 2006.



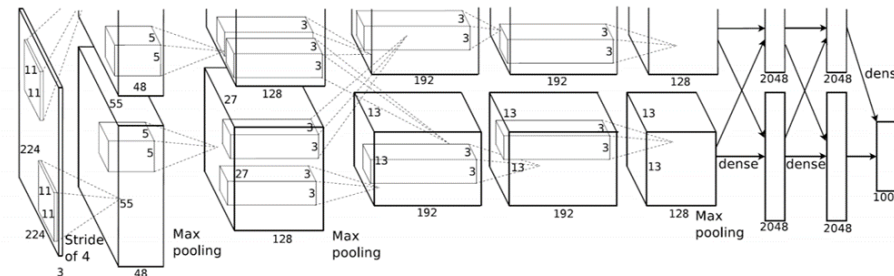
Hinton et al. “deep belief networks” (2006)



Big Data / Dataset size explosion



NN Parallelization with GPUs



AlexNet (2012)

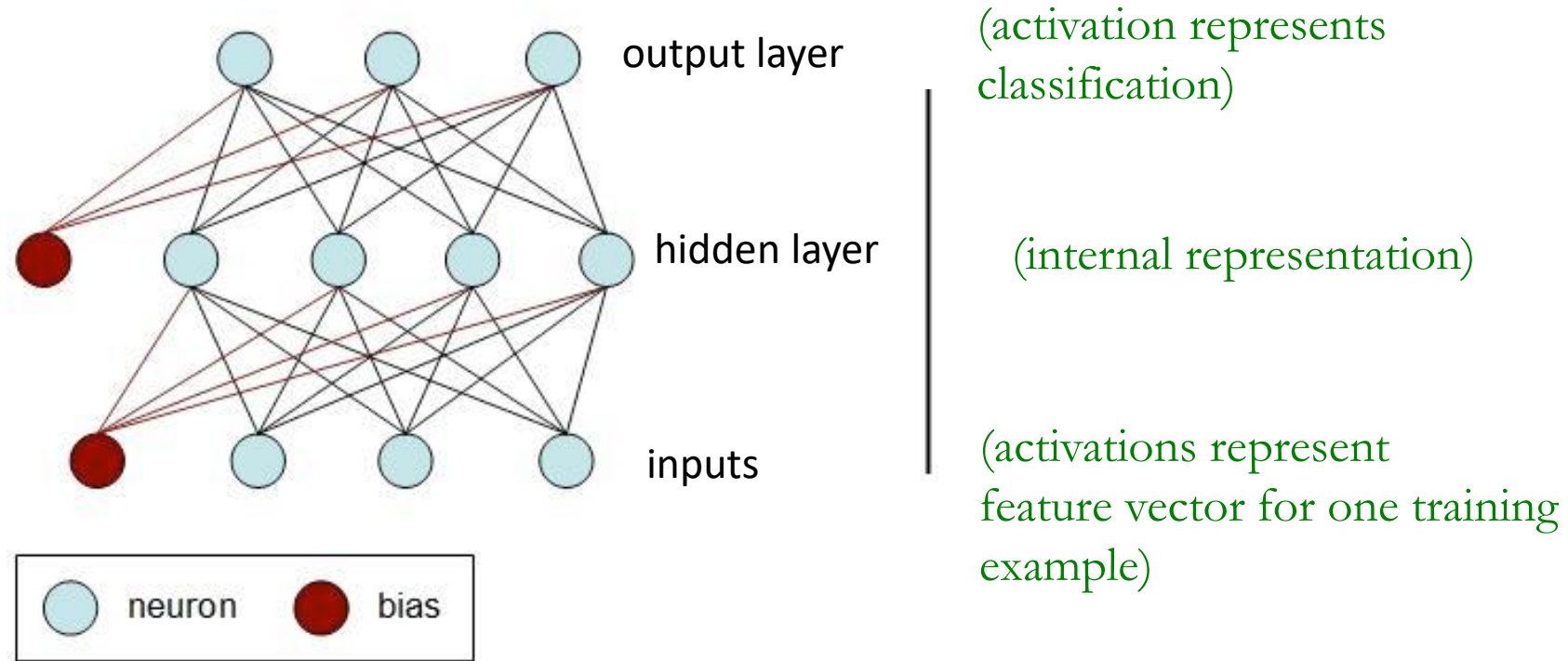
A Very Brief History of NNs

- 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.

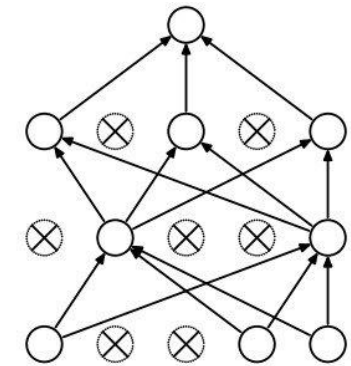
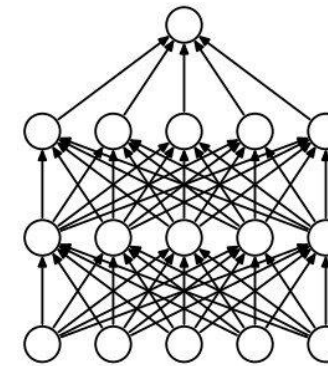
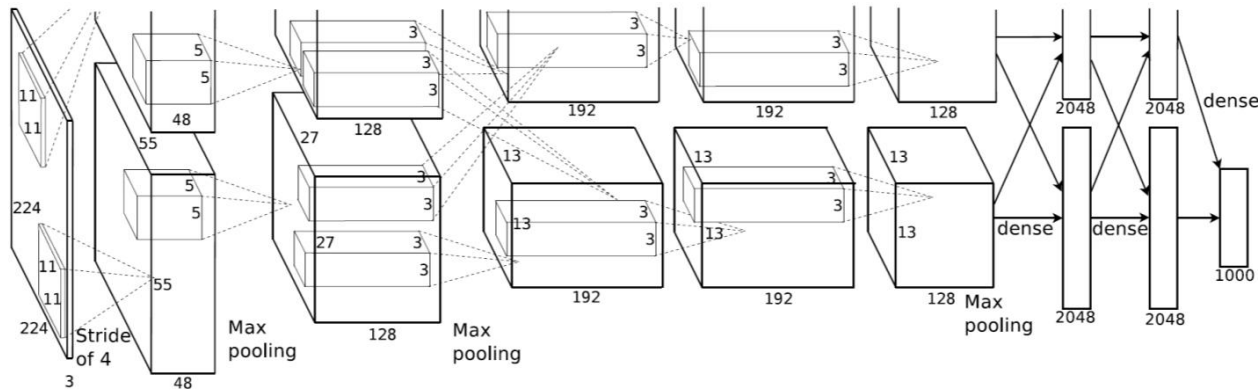
(2) NNs have come much larger because of more powerful computer (including the use of GPUs) and better software infrastructure (NNs are on pace to have the same number of neurons as the human brain by 2050).

A Two-Layer Neural Network



- **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.

DNNs: AlexNet (2012)

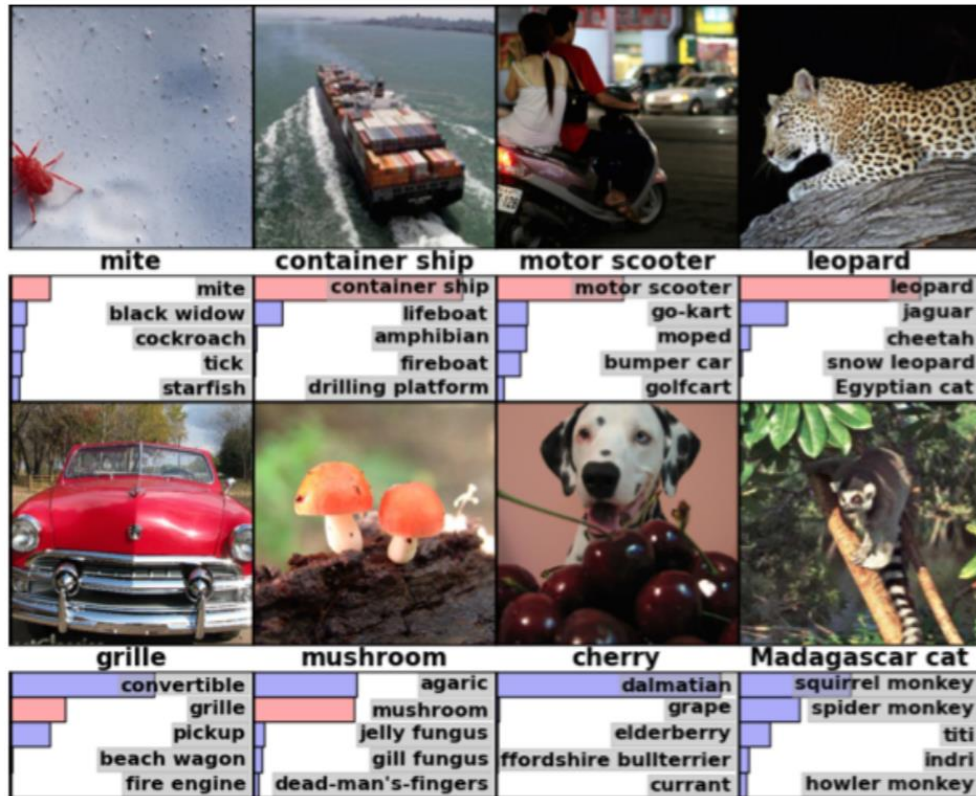


AlexNet was developed by Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever; it uses CNNs with GPU support. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points ahead of the runner up.

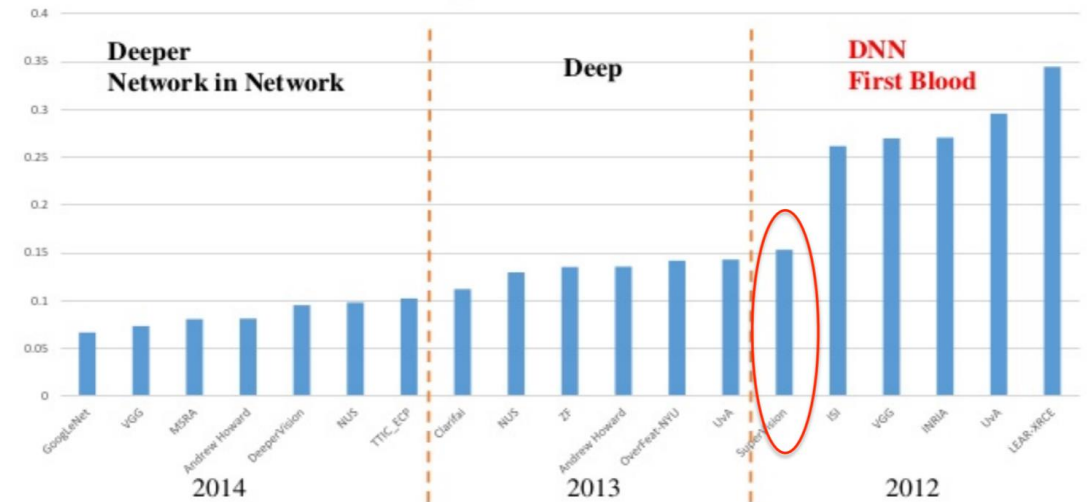
Among other innovations: AlexNet used GPUs, utilized RELU (rectified linear units) for activations, and “dropout” for training.

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

DNNs: AlexNet (2012)

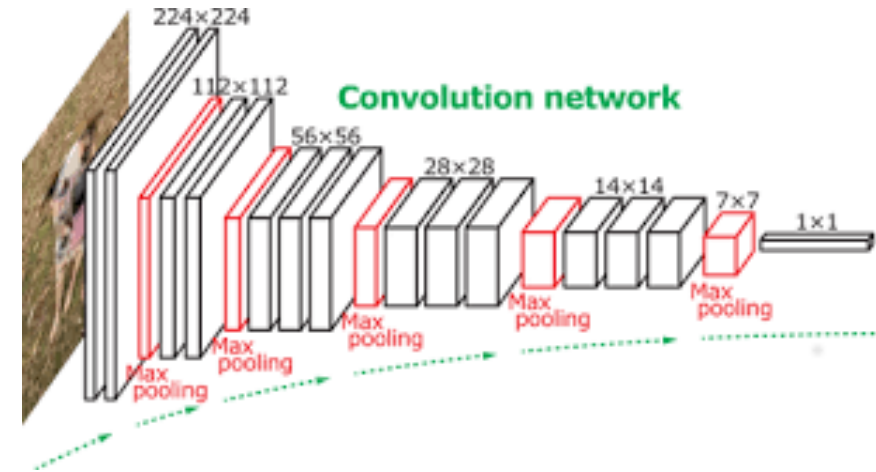


ImageNet Classification error throughout years and groups



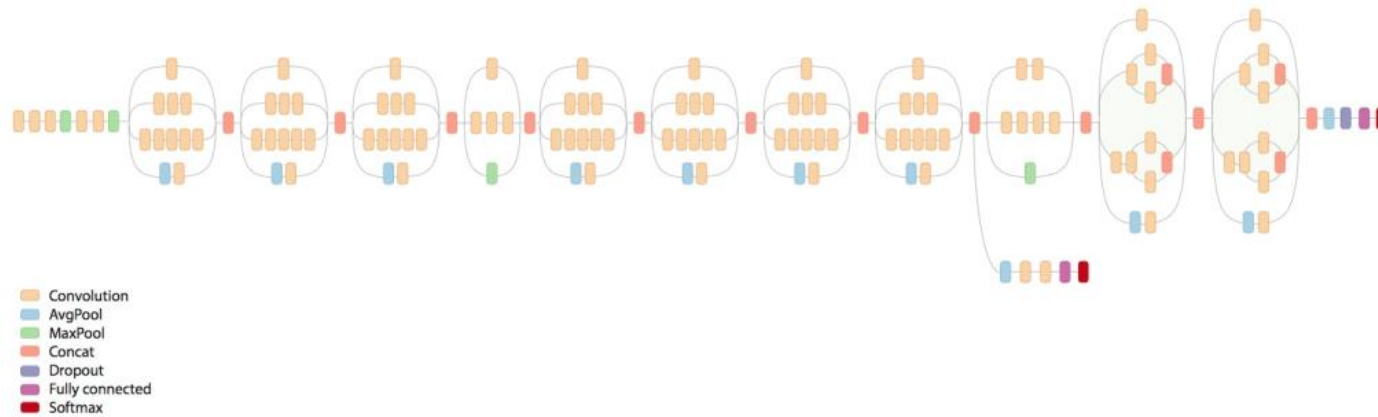
Li Fei-Fei: ImageNet Large Scale Visual Recognition Challenge, 2014 <http://image-net.org/>

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.

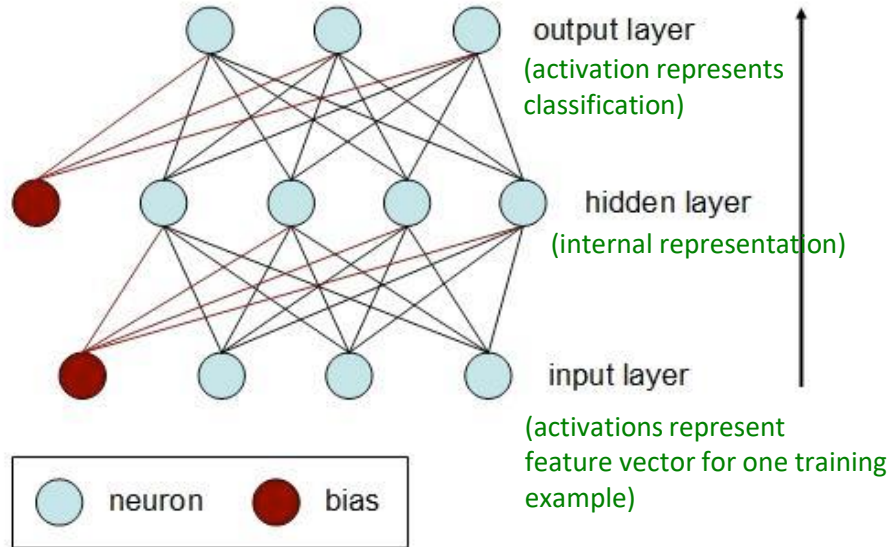
DNNs: Inception (2015, Google)



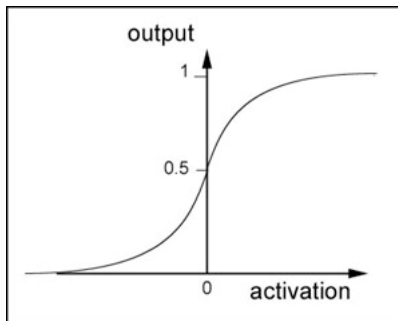
<https://arxiv.org/pdf/1409.1556.pdf>

- 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).

Neural Network Notation



Sigmoid function:



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,

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

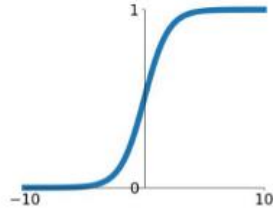
For each node k in output layer,

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

Common Activation Functions

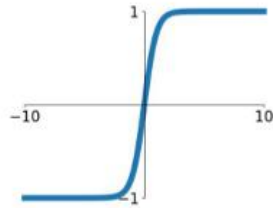
Sigmoid

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



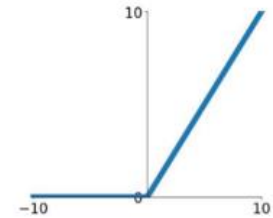
tanh

$$\tanh(x)$$



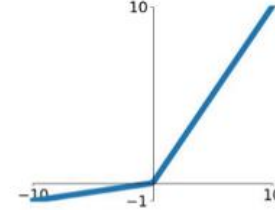
ReLU

$$\max(0, x)$$



Leaky ReLU

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

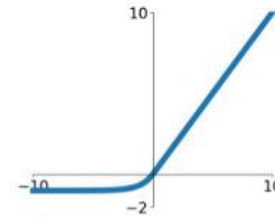


Maxout

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

ELU

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



RELU & Their Generalizations

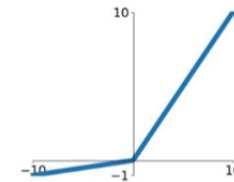
(3) Generalizations of RELUs are based on using a non-zero slope α_i when $z_i < 0$:

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

(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 polarity reversal of the input illumination.

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

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

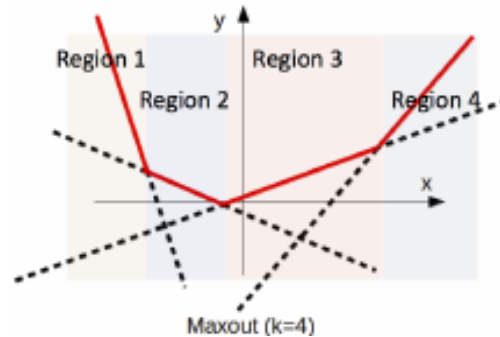


RELU & Their Generalizations

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

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

(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”; as mentioned, regularization is frequently used to combat overfitting.

Some form of regularization should almost always been applied to a DNN model (with very few exceptions).

Regularization

- 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 thought of as soft constraints applied to the parameter values. If chosen correctly, these extra constraints and penalties can lead to a significant performance improvement.
- 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

- Two common regularization approaches are L2 and L1-regularization, respectively.

For **L2-regularization**, the loss function is appended with an L2 “penalty”:

$$L_{new} = L + \frac{\lambda}{2} W^2$$

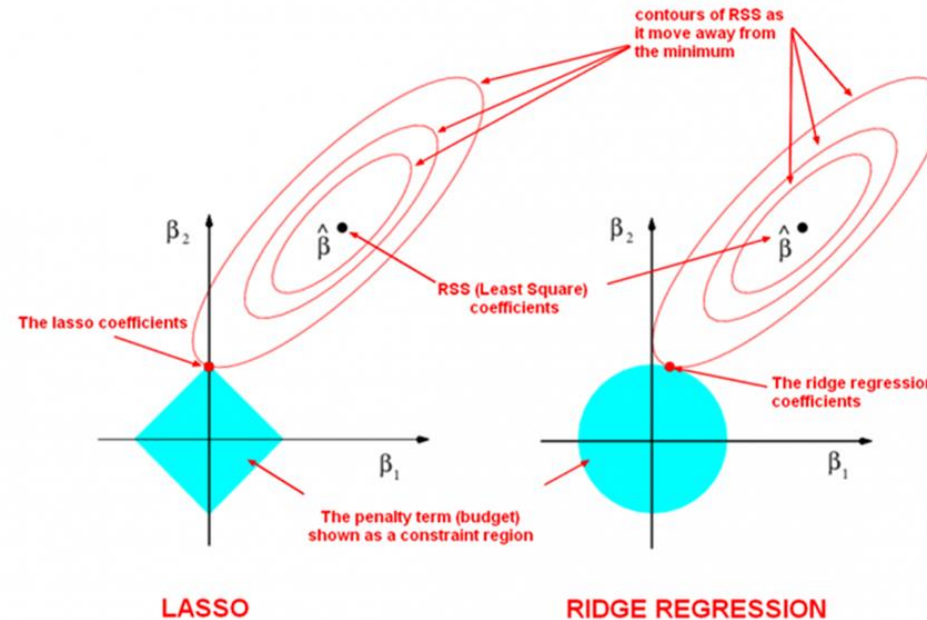
Where λ is a hyperparameter that determines the degree to which we “value” regularization; applying L2-regularization results in a model with small weight values, which safeguards against overfitting.

L1-regularization applies an L1 penalty term:

$$L_{new} = L + \frac{\lambda}{2} |W|$$

Regularization

- In comparison to L^2 regularization, L^1 regularization results in a **sparse model**. 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.

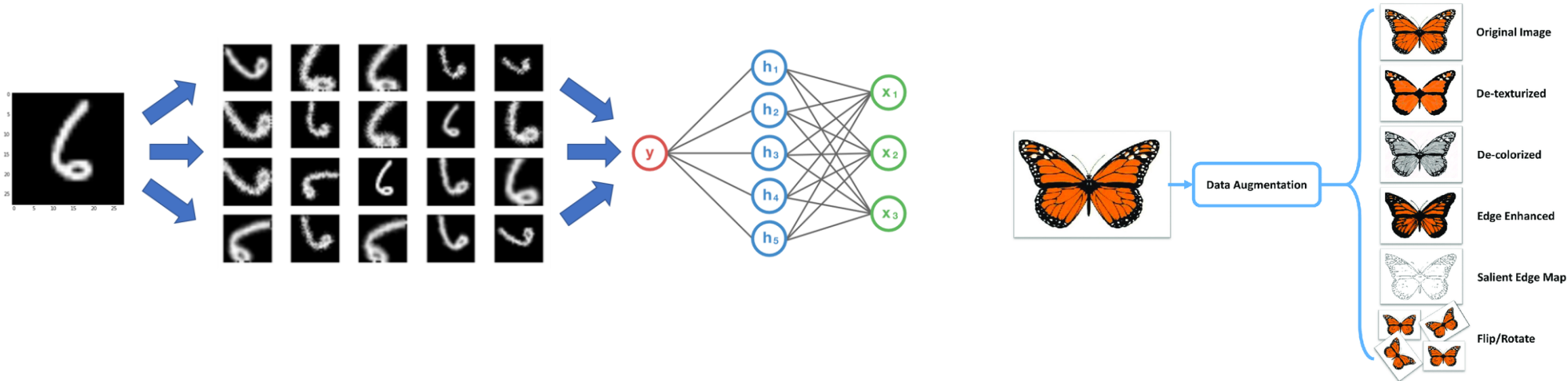


Data Augmentation

- 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 \mathbf{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 (\mathbf{x}, y) pairs easily by transforming the \mathbf{x} inputs in our training set.

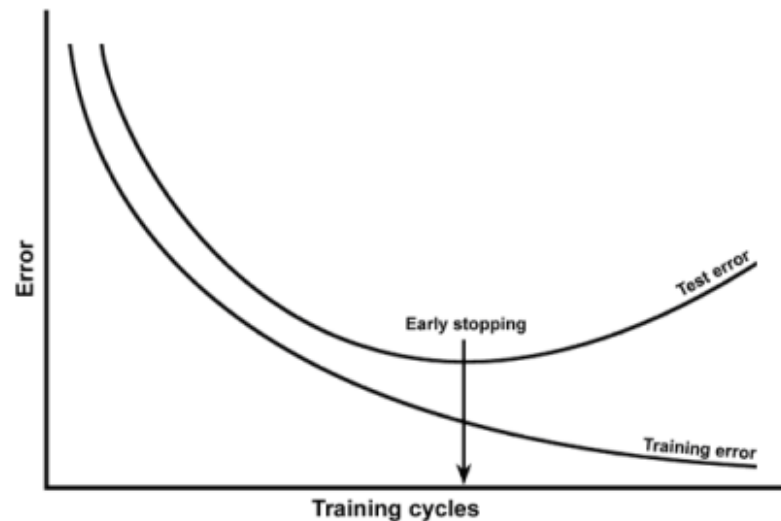
Data Augmentation

- Another form of regularization, **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.



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).

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



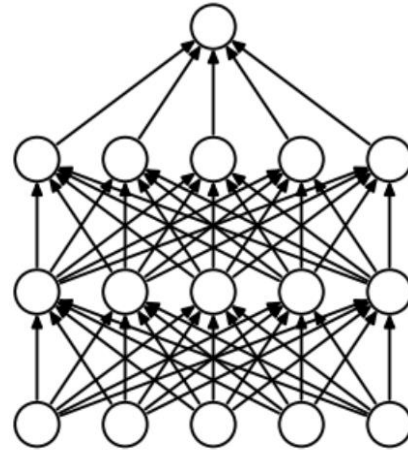
Sparsely parameterized linear regression model ($y=Ax$)

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

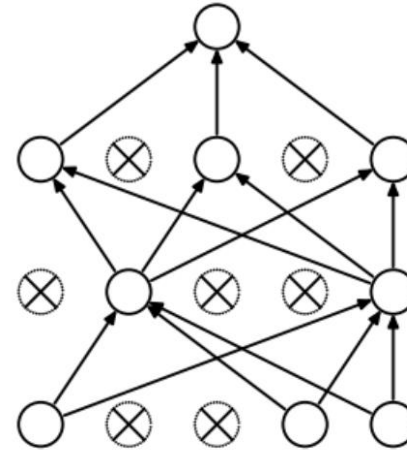


Linear regression with a sparse representation h of the data x ; ($y=Bh$)

Dropout



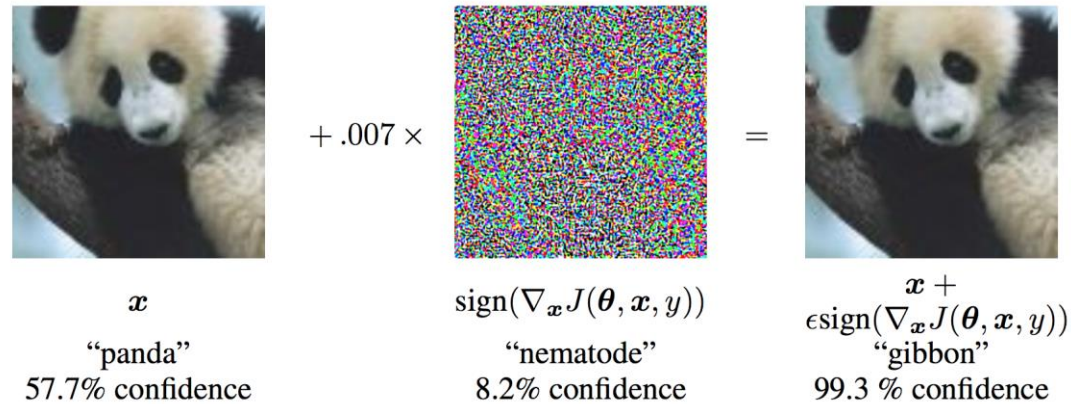
(a) Standard Neural Net



(b) After applying 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.

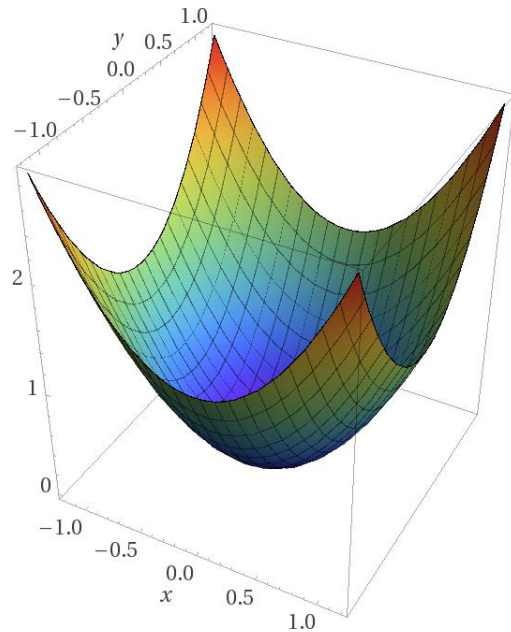
Adversarial Training


$$\begin{array}{ccc} \text{[Panda Image]} & + .007 \times & \text{[Noise Image]} & = & \text{[Perturbed Panda Image]} \\ x & & \text{sign}(\nabla_x J(\theta, x, y)) & & x + \epsilon \text{sign}(\nabla_x J(\theta, x, y)) \\ \text{"panda"} & & \text{"nematode"} & & \text{"gibbon"} \\ 57.7\% \text{ confidence} & & 8.2\% \text{ confidence} & & 99.3\% \text{ confidence} \end{array}$$

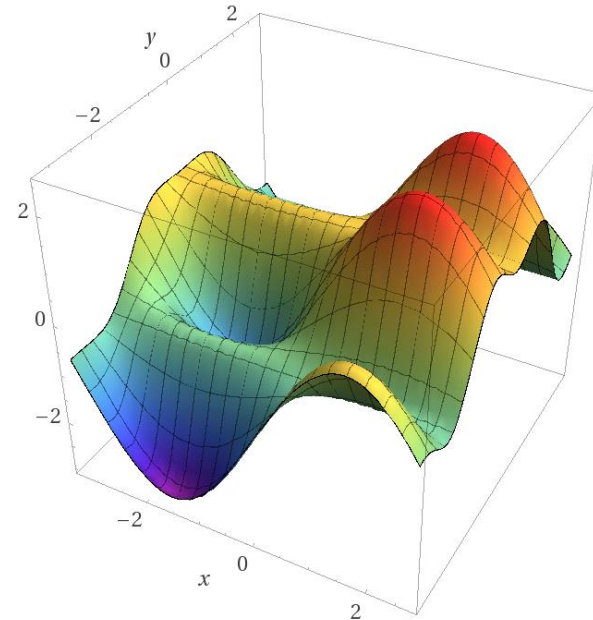
- Szegedy et al. (2014) found that even NNs that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally constructed by using an optimization procedure to search for an input x' near a data point x such that the model output is very different from x' (oftentimes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



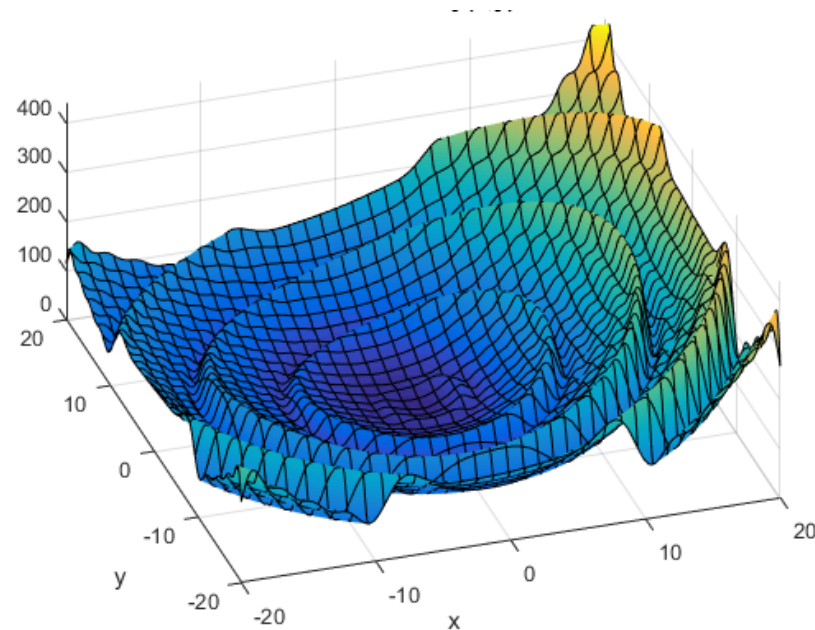
Convex Function



Non-Convex Function

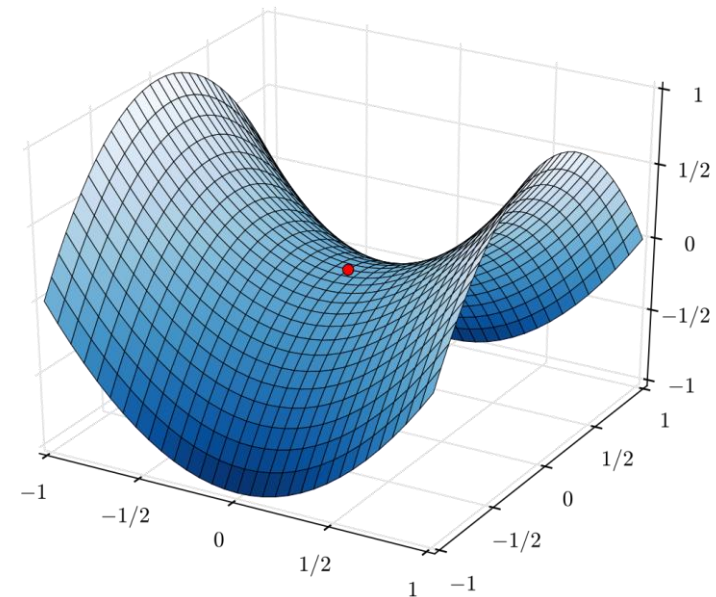
Challenges for DNN Optimization: Local Minima

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



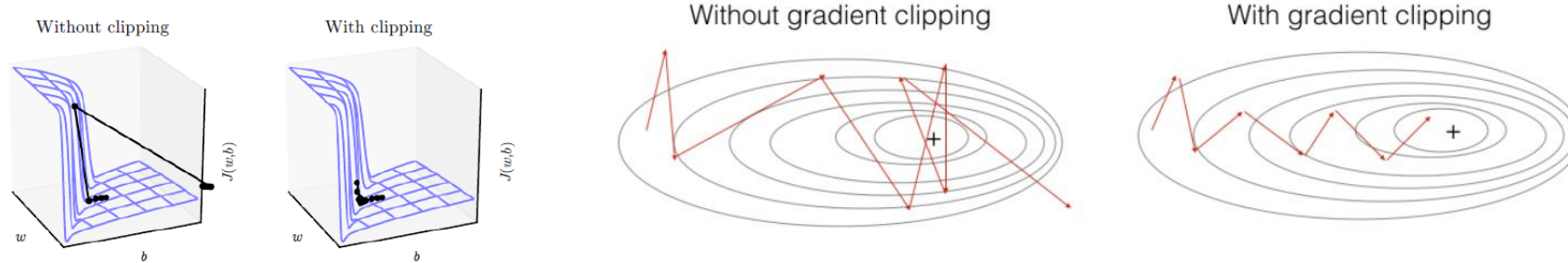
Challenges for DNN Optimization: Plateaus, Saddle Points

- 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 lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues.
- 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 know 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

- **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.

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

$$\epsilon_k = (1 - \alpha) \epsilon_0 + \alpha \epsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

* 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.

Basic Algorithms: SGD

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

Require: Learning rate ϵ_k .

Require: Initial parameter θ

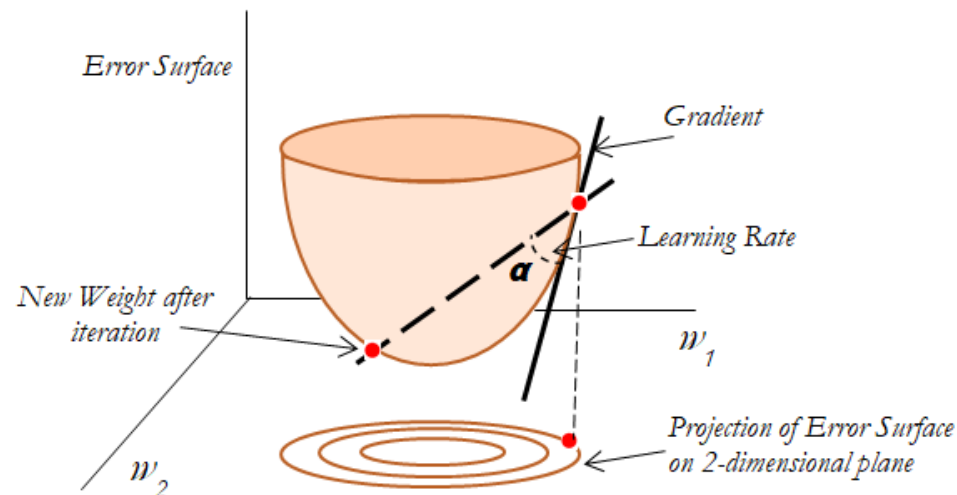
while stopping criterion not met **do**

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

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

 Apply update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

end while



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 \mathbf{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.

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

- The velocity \mathbf{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 θ , initial velocity v .

while stopping criterion not met **do**

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

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

 Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$

 Apply update: $\theta \leftarrow \theta + \mathbf{v}$

end while

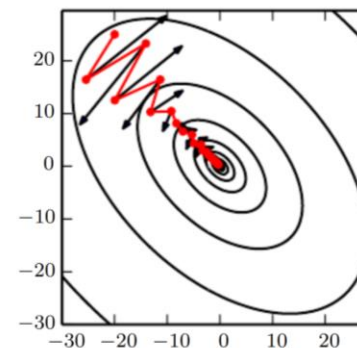
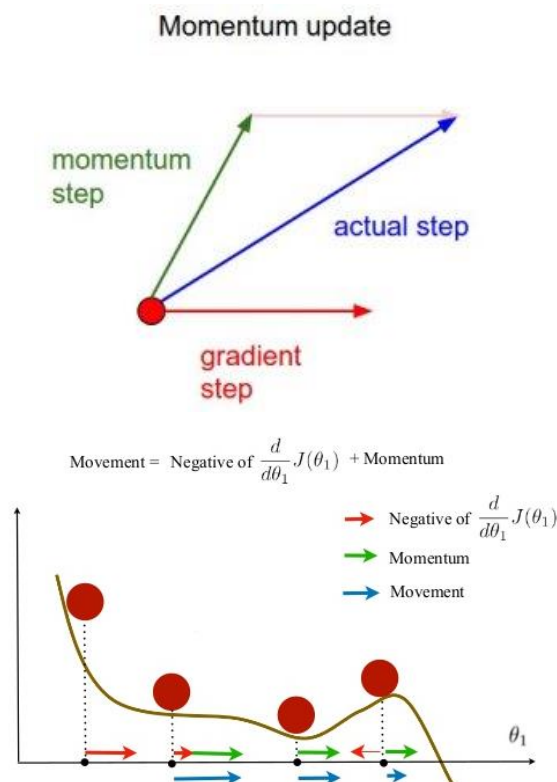


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian 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 use 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 difficulties 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 improved initialization strategies is a difficult task because NN optimization is not yet well understood.

Weight Initialization

- 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.
- 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.
- 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*.

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 sensitive 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 derivatives 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 θ

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

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

while stopping criterion not met **do**

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

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

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

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

 Apply update: $\theta \leftarrow \theta + \Delta \theta$

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 θ
Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.
 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
 $t \leftarrow t + 1$
 Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$
 Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$
 Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
 Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
 Compute update: $\Delta \theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$ (operations applied element-wise)
 Apply update: $\theta \leftarrow \theta + \Delta \theta$
end while

Second-Order Methods: Newton's Method

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

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

Require: Training set of m examples

while stopping criterion not met **do**

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

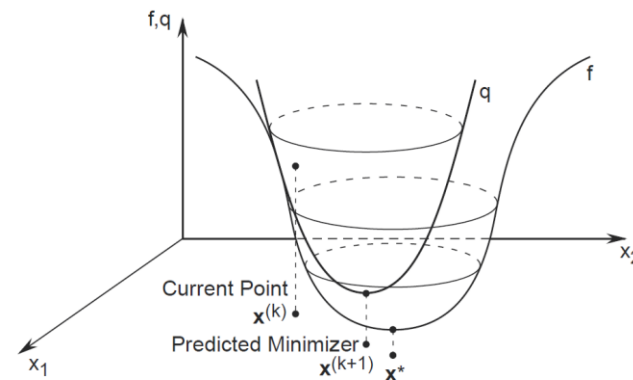
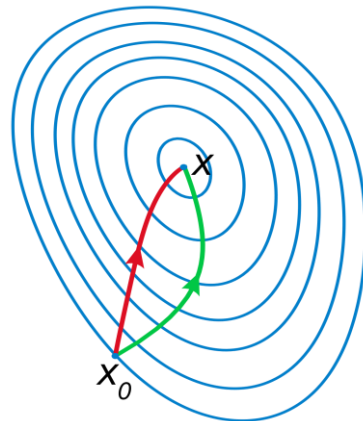
 Compute Hessian: $\mathbf{H} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}}^2 \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$

 Compute Hessian inverse: \mathbf{H}^{-1}

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

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

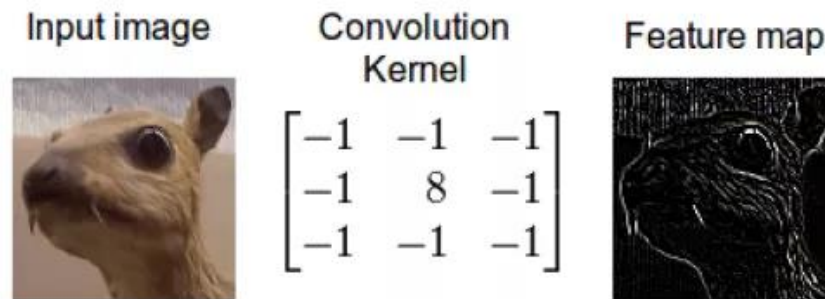
end while



Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized kind of NN for processing data that has a known grid-like topology (particularly: image data).

CNNs are simple NNs with a specialized convolution operation in place of general matrix multiplication.



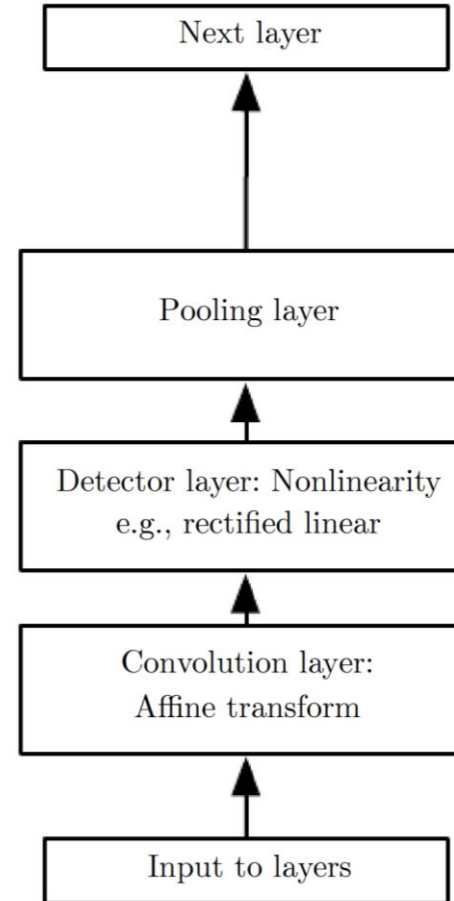
Convolution leverages (3) important ideas to help improve an ML system:

- (1) sparse interactions (i.e. the kernel is smaller than the input image)
- (2) parameter sharing: instead of learning a separate set of parameters at each location for a given kernel, we learn only one set.
- (3) equivariant representations: parameter sharing causes the layer to be equivariant to translation.

Convolutional Neural Networks

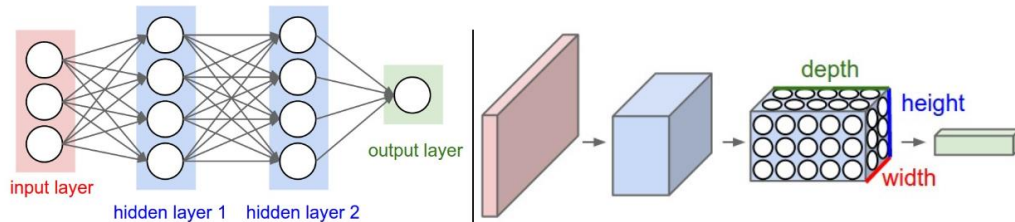
CNNs typically consist of (3) stages:

- (1) In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations
- (2) In the second stage, each linear activation is run through a non-linear activation function (e.g. RELU).
- (3) In the third stage, pooling is used to modify the output further.

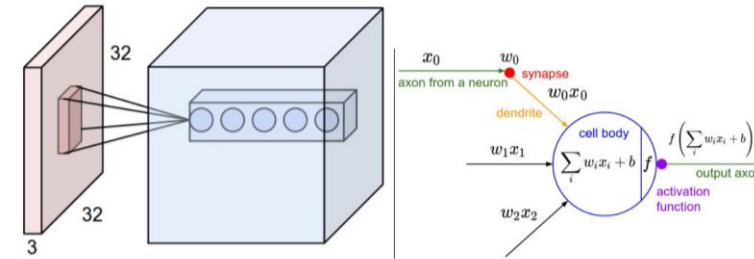


Convolutional Neural Networks

CNNs are very similar to ordinary NNs: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet 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).

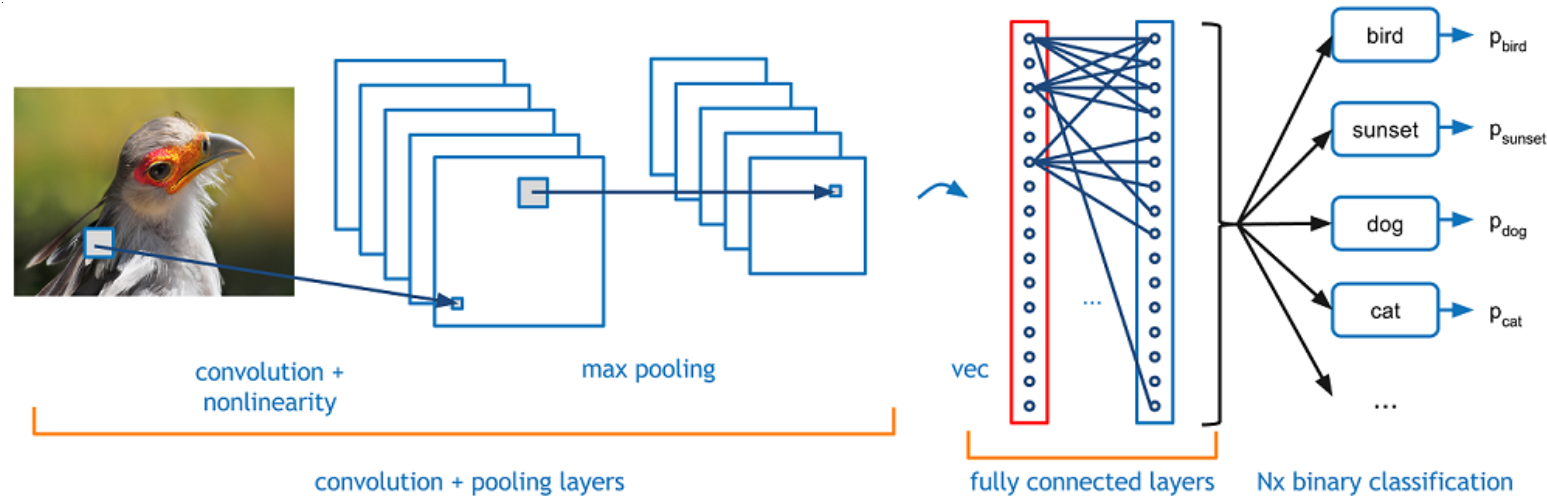


Left: An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below. Right: The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

The key difference with CNNs is that neurons/activations are represented as 3D volumes. CNNs additionally employ **weight-sharing** for computational efficiency; they are most commonly applied to image data, in which case image feature activations are trained to be translation-invariant (convolution + max pooling achieves this).

Convolutional Neural Networks

Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color. Now, we will have an entire set of filters in each CONV layer, and each of them will produce a separate 2-dimensional activations; these features are stacked along the depth dimension in the CNN and thus produce the output volume.



A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. The main types of layers to build CNN architectures are: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). These layers are stacked to form a full CNN **architecture**.

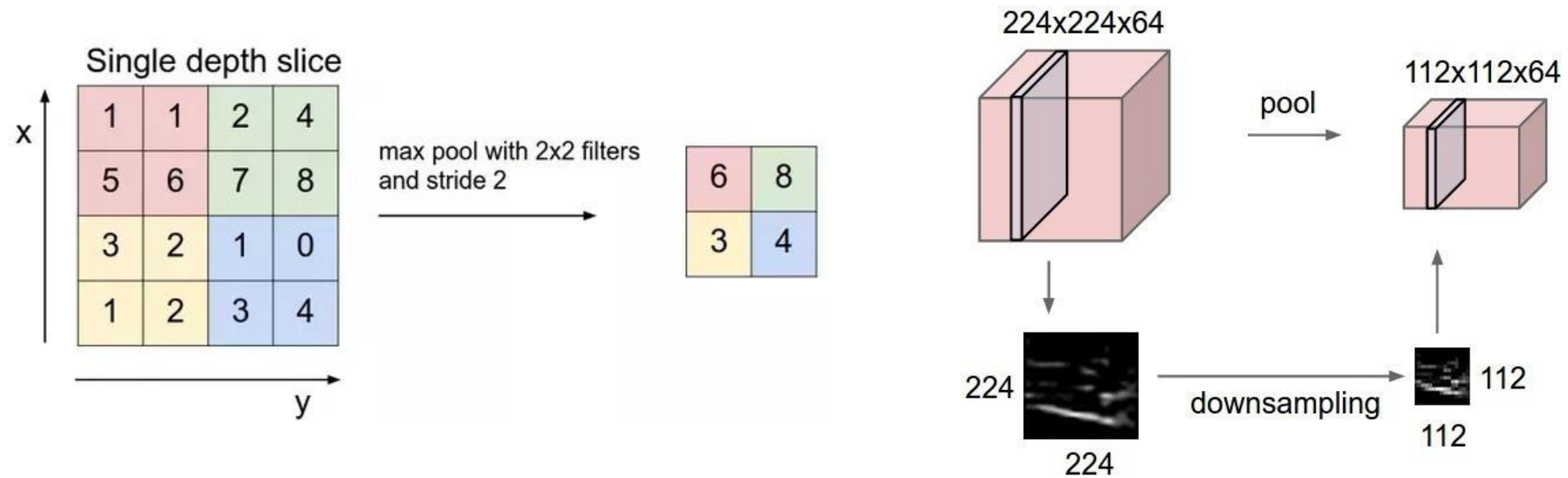
The convolution layer determines the activations of various filters over the original image; pooling is used for downsampling the images for computational savings; the fully-connected layers are used to compute class scores for classification tasks.

Pooling helps to make the representation approximately invariant to small translations of the input.

Convolutional Neural Networks

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than activation units, by reporting summary statistics for pooling regions spaced k pixels apart (k here is known as the stride). This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process.

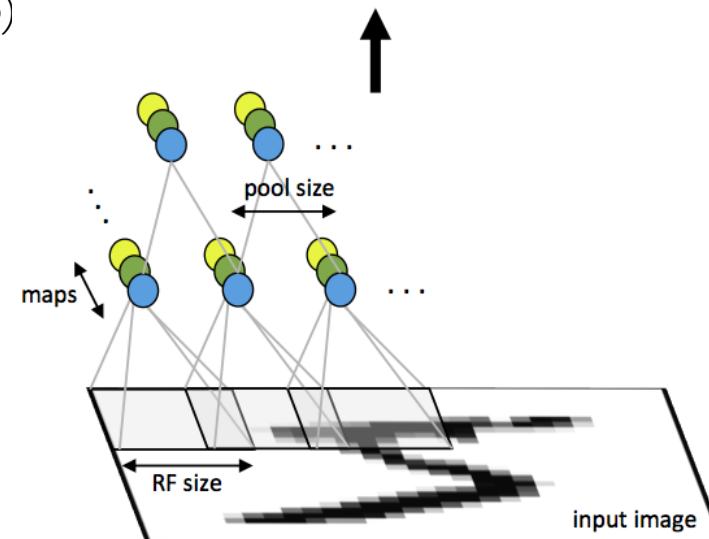
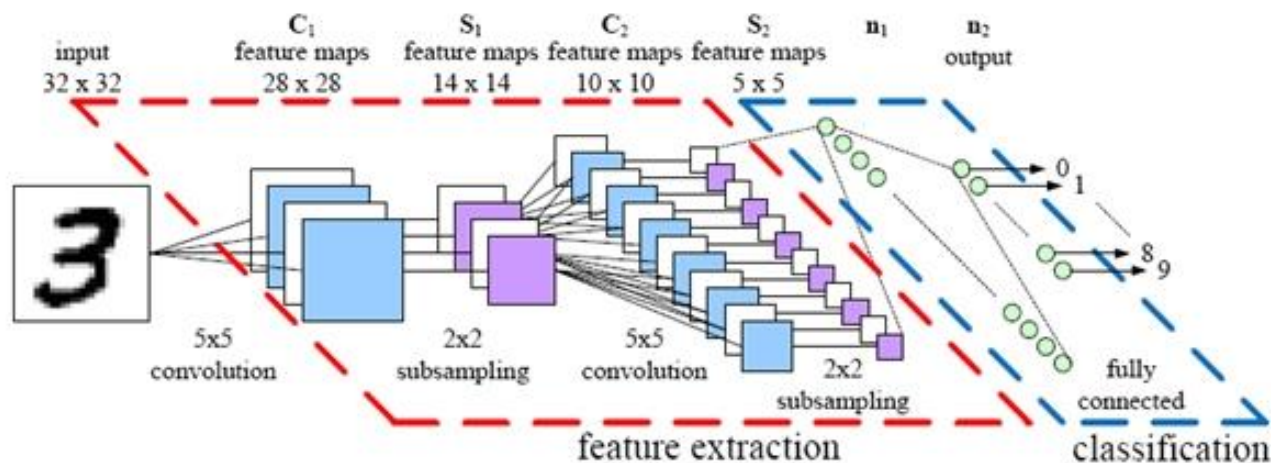
Pooling can also be essential for handling images of variable size.



Convolutional Neural Networks

A nice way to interpret CNNs via a brain analogy is to consider each entry in the 3D output volume as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since the same filter is used).

Each neuron is accordingly connected to only a local region of the input volume; the spatial extent of this connectivity is a hyperparameter called the receptive field (i.e. the filter size, such as: 5x5)

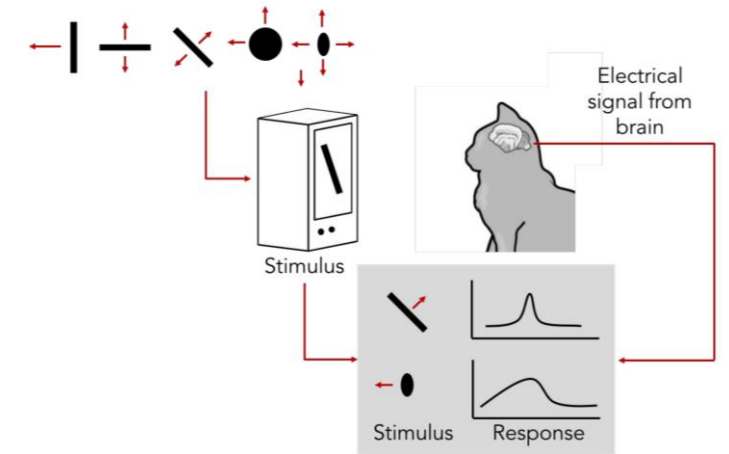


(Image from the LeCun MNIST paper, 1998)

Convolutional Neural Networks



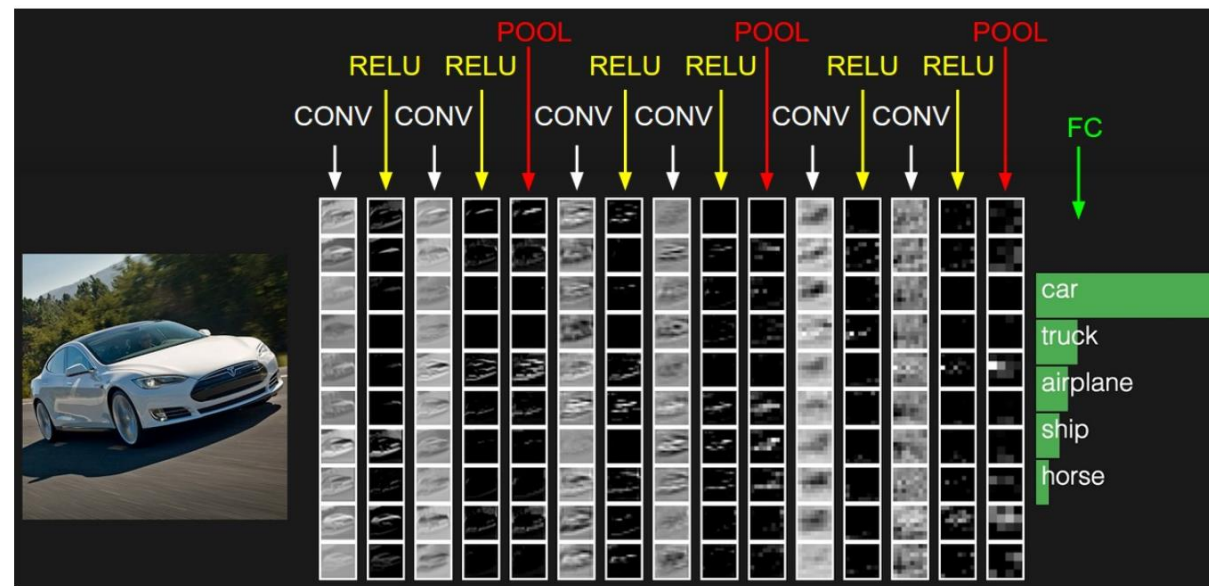
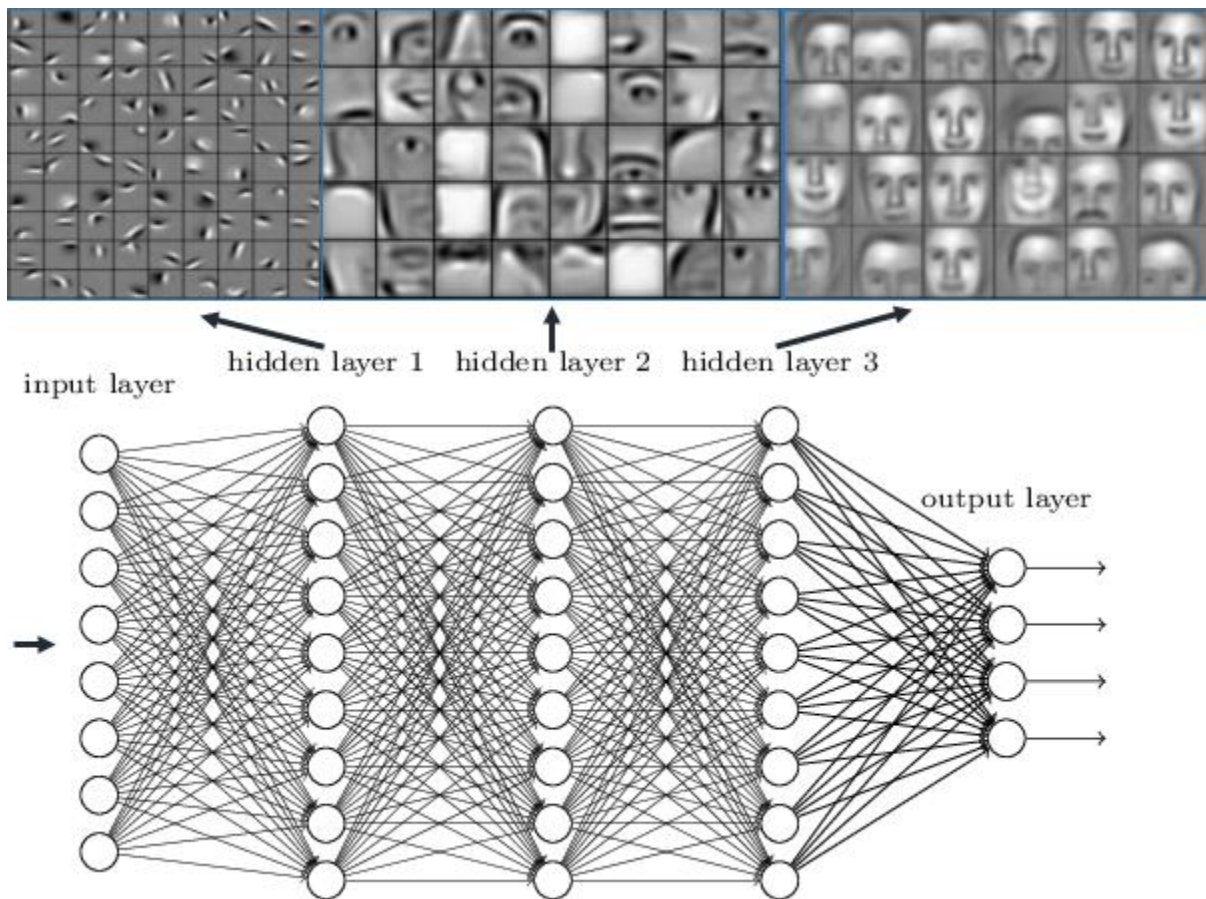
Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size $[11 \times 11 \times 3]$, and each one is shared by the 55×55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some 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.



Hubel Wiesel (1959), study of mammalian primary visual cortex

Of note, some researchers believe that the first stage of visual processing in the brain (called V1) serve as edge detectors that fire when an edge is present at a certain location and orientation in the visual receptive field.

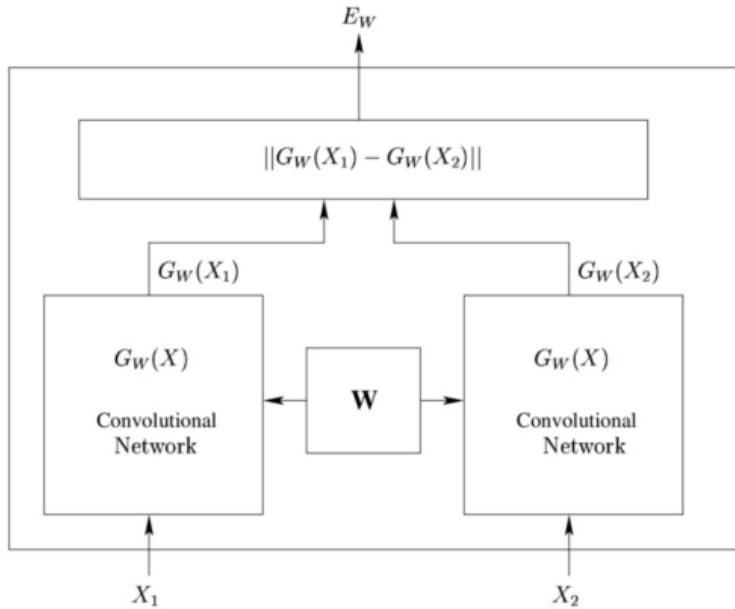
DNNs Learn Hierarchical Feature Representations



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 veritable, 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

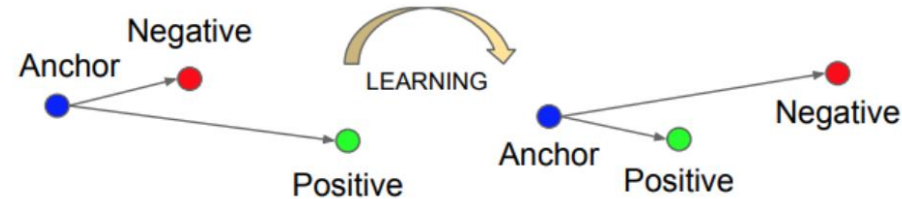
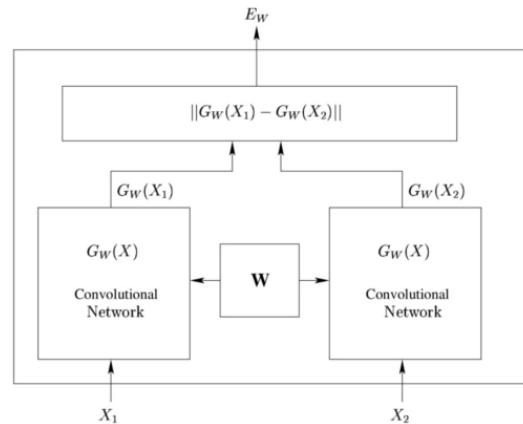


- A Siamese neural network uses two identical sub-networks (e.g. pretrained CNNs) in tandem, with the overall objective to determine how similar two comparable things are (e.g. signature verification, face recognition.). The sub-networks have the same parameters and weights.
- Each sub-network is fed an input (e.g. an image of a face), producing the respective outputs. If the distance between the two encodings:

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

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

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.

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

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

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

One-Shot Learning: Siamese Networks

Define the triplet loss function:

$$L(A, P, N) = \max (\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha, 0)$$

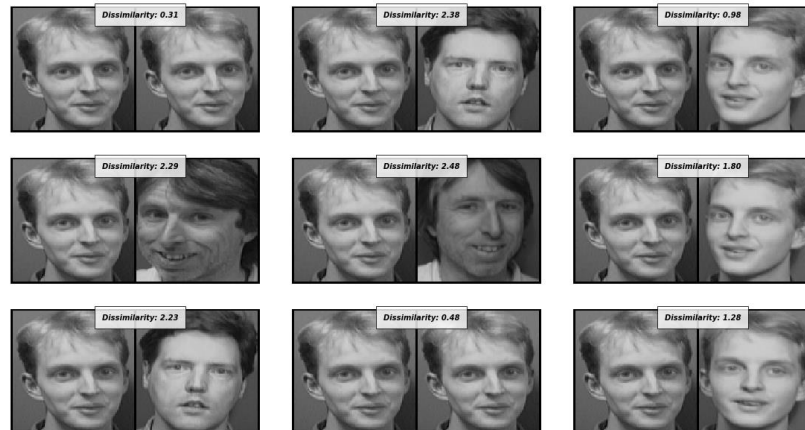
The max means as long as $d(A, P) - d(A, N) + \alpha$ is less than or equal to zero, the loss $L(A, P, N)$ is zero, but if it is greater than zero, the loss will be positive, and the function will try to minimize it to zero or less than zero.

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, 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.



Deep Learning: Practical Considerations

In general, need to know how to choose an appropriate algorithm and how to monitor and respond to feedback from experiments.

Decide what to do next: gather more data, improve optimization model, add/remove regularization features, improve optimization of model, debug, etc.

A practical design process:

- (1) Determine goals (error metric to use, target value for error metric)
- (2) Establish working end-to-end pipeline early, including estimation of performance metrics.
- (3) Determine computational bottlenecks; diagnose which components are performing worse than expected and whether performance is due to overfitting, underfitting, etc.
- (4) Repeatedly make incremental changes, e.g., gathering new data, adjusting hyperparameters, or changing algorithms.

(Andrew Ng, 2015)

Deep Learning: Practical Considerations

Performance Metrics

(*) Determine goals in terms of which error metrics to use, and reasonable level of performance to expect.

Default Baseline Models

(*) Establish a reasonable end-to-end system as soon as possible; consider beginning without using deep learning at all. If you know beforehand that your problem falls into an “AI-complete” domain (e.g. image classification), etc., you should incorporate some DL methods.

Apply some form of regularization; optimize with a variable learning rate if possible; pre-process data appropriately.

Gathering More Data

Many ML practitioners are tempted to improve model results by trying many different algorithms; in fact, it is often much better to gather more data to improve the learning algorithm (keep in mind that you will need a “large” batch of new data to see substantial improvements).

If test data performance is significantly worse than the training data performance, gathering new data may present an effective solution.

In order to know how much data one needs to add, it is possible to use error bounds or to carefully interpolate between training data size and generalization error.

Deep Learning: Practical Considerations

Selecting Hyperparameters

(*) There are two basic approaches: manually choose the hyperparameters values or automate this process.

(*) Can follow U-shaped curve of generalization error for tuning; automated hyperparameter tuning can be handled with hyperparameter optimization algorithms (e.g. Gaussian Processes); random search can also be effective for hyperparameter tuning.

(*) The learning rate is perhaps the most important hyperparameter for a DL algorithm.

Some Debugging Strategies:

(*) Visualize the model in action

Visualize the worst mistakes (e.g. using a confidence measure)

(*) Fit a tiny dataset; if you can't train a classifier to correctly label a single example, for instance, you have a bug.

(*) Monitor histograms of activations and gradient: this can reveal problems with exploding/vanishing gradient, "dead" neurons, poor choice of learning rate, etc.