Deep Neural Networks For Astrostatisticians

David Banks

SAMSI

Learning Objectives

- survey of what deep learning (DL) does
- introduction to generic deep learning
- convolutional neural networks
- recurrent neural networks

We shall focus upon Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs work well for images, and RNNs work well for video or audio streams.

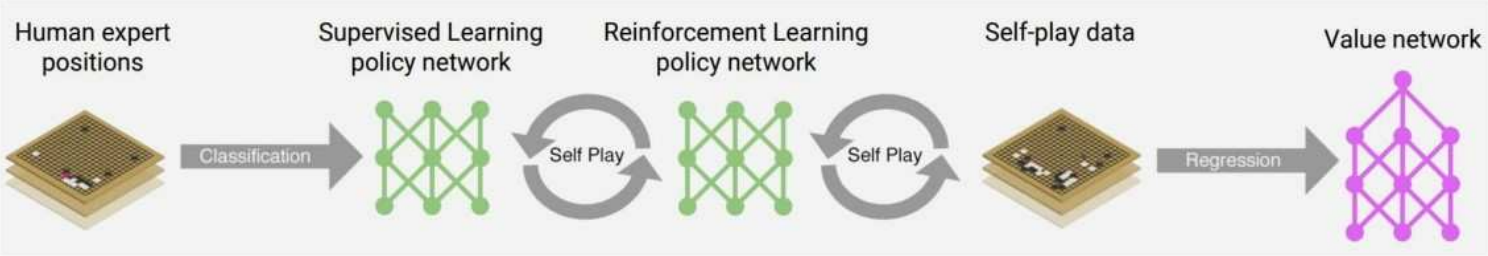
The Bible of DL is Deep Learning by Goodfellow, Benjio and Courville. It is freely available on-line at `https://www.deeplearningbook.org/`.

1. Why Do We Care?

Deep neural networks (DNNs) have achieved spectacular success in many problems, of which classification and prediction are just two examples. Others are:

- autonomous vehicles
- Google translate
- recommender systems
- image synthesis
- Alpha Go
- control of complex systems

Reinforcement learning on a DNN enabled Google's Alpha Zero to learn chess through self-play in nine hours.



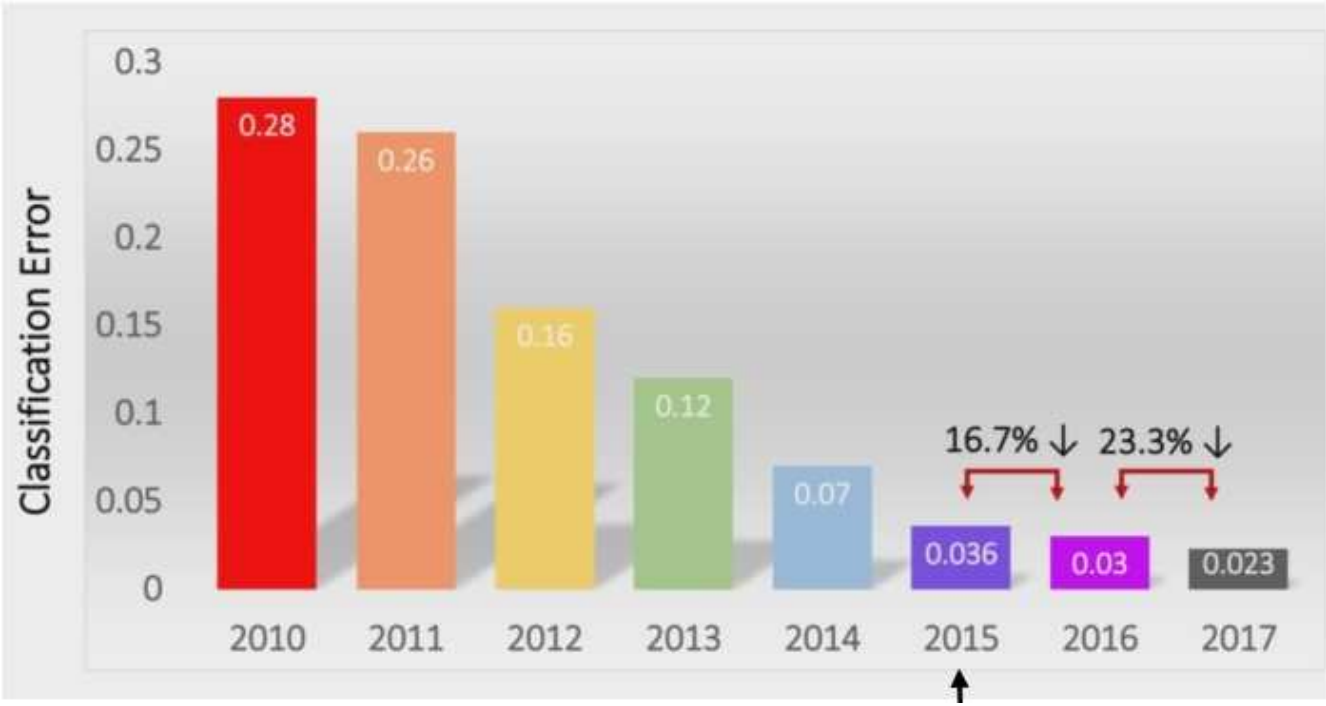
Its opponent was Stockfish, which had undergone continual development from 2008 to 2017.

In 2017, AlphaZero beat Stockfish as white 25 times, as black 3 times, and drew the other 72 games in the set.

Style Transfer Example: Picasso/Van Gogh/Seurat

Inpainting Example

Image classification is a natural application for astrostatistics. The ImageNet Challenge is an annual competition on classification.



AlexNet (2012) was a Convolutional Neural Network with 8 layers and 61 $*10^6$ parameters. ResNet (2015) had 152 layers. CUImage (2016) was an ensemble of six models.

Problems with Deep Learning

- Deep Learning (DL) needs a lot of data to train a NN.
- Training can be slow and may get caught in a bad local minimum (it will get caught in a local minimum).
- There is a large carbon footprint.
- There is not much mathematical/statistical theory.
- There are known pathologies.
- They can be used for evil.

1.1 Software

The popular programming platforms are Keras, PyTorch and TensorFlow. All three are Python friendly. TensorFlow is explictly tuned for Google's Tensor Processing Units (TPUs).

When choosing a platform, consider the learning curve, the speed of development, the commitment of the user community, the likelihood of long-term growth and stability, and the available tools.

Two tools: AdaNet is TensorFlow software that supports deep learning for ensembles. AutoAugment is a tool that distorts images so that the CNN focuses on salient features.

1.2 Architecture

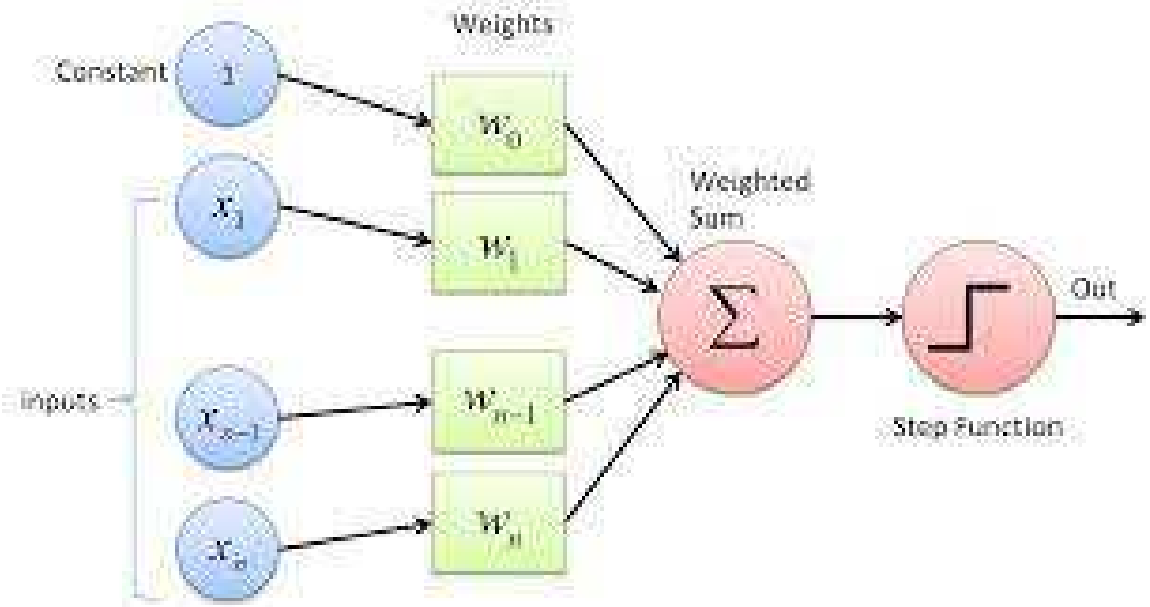
DNN architecture is the connectivity structure among the nodes. Different architectures are favored in different applications, and one of the challenges in DL is to find a good architecture.

The architecture links the input data through a series of layers consisting of perceptron nodes, which then finally produce the output. Determining the number of layers and numbers of nodes in each layer is an art informed by lots of experience but little theory.

The **depth** of a DNN is the number of hidden layers. The **width** of a DNN is the number of perceptrons in a layer, but this may change from layer to layer, so it need not be a fixed value.

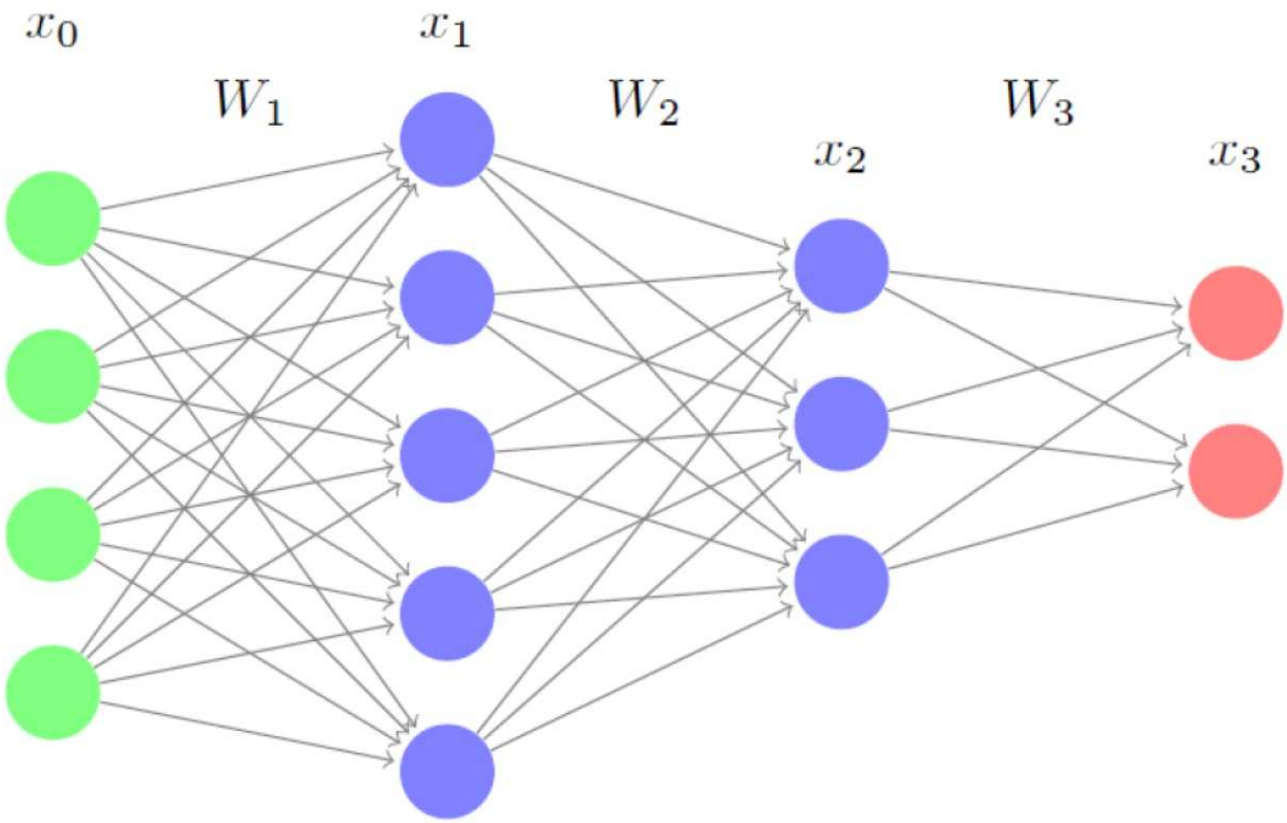
CNNs and RNNs have very different architectures.

The perceptron was invented by Frank Rosenblatt in 1957. It is the building block on DNNs.



The blue circles denote the vector of inputs, including a constant. The green squares weight the inputs which are then summed and sent to an **activation function**.

Architecture shows how the perceptrons are arranged in the DNN. Green nodes are the inputs, blue are in the hidden layers, and pink are the outputs.



This shows the DNN composition calculation $f(W_3 * f(W_2 * f(W_1 * x_0)))$.

The $x_1, \ldots, x_n$ are are covariates that are input (e.g., pixel values for images, or words coded as numbers). The w_0 is called the **bias** or **offset**.

The perceptron multiplies each covariate by a corresponding weight, $w_1, \ldots, w_n$, and adds the constant w_0 to produce the outputs. Our notation takes $W = (w_0, w_1, \ldots, w_n)$.

The activation function $f(z)$ is a monotone function of z. The one shown in the figure is a step function, but other popular choices include the sigmoidal function, the tanh function, and the ReLU function. Ideally, the activation function is fast and differentiable.

Step Function $$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ a & \text{if } z \geq 0 \end{cases}$$

Sigmoidal Function $$f(z) = \frac{\exp(z)}{1+\exp(z)}.$$

Tanh Function $$\mathrm{f}(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

ReLU Function $(a > 0)$ $$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ az & \text{if } z \geq 0 \end{cases}$$

To start, for most applications, use the ReLU function. It is fast and almost everywhere differentiable.

Activation Functions

Sigmoid

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

Leaky ReLU

$\max(0.1x, x)$

tanh

$\tanh(x)$

Maxout

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

ReLU

$\max(0, x)$

ELU

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

There are three main architectures we shall consider, each of which can be varied in many ways:

- The feedforward DNN, in which outputs cannot be inserted as inputs to previous layers and all nodes in one layer are connected to all nodes in the next layer.
- The CNN, which is like the feedforward DNN except that not all nodes in one layer are connected to all nodes in the next layer.
- The RNN, where the graph representation has cycles (good for time data, such as speech recognition).

There other architectures, such as recursive (not recurrent) NNs, which re-use weights.

1.3 Theory

Cybenko (1989) proved the universal approximation theorem: a feedforward NN with a single hidden layer having a finite number of perceptrons can approximate any continuous function on a compact subset of $\mathbb{R}^n$. He used a sigmoidal activation function, but a similar result was found by Hanin (2017) for ReLU activation functions.

This is related to earlier work by Kolmogorov and Arnold, who partially solved Hilbert's 13^{th} problem: Doea a solution exists for all 7th-degree equations using algebraic (variant: continuous) functions of two arguments?

The intuition for the Universal Approximation Theorem is simple.

- A continuous function on a compact set can be approximated by a piecewise constant function.
- To represent a piecewise constant function as a NN, for each region where the function is constant, use a NN as an indicator function for that region.
- Then build a final layer with a single node whose input is a linear combination that sums all the indicators, with weights equal to the constant value of the corresponding region.

Single hidden layer feedforward networks require exponential width. Universal approximability results exist for DNNs in terms of both depth and width.

2. Training a Deep Neural Network

Training a DNN is an optimization problem. One seeks to find the weights $\boldsymbol{W}$ and the biases (or offsets) $\boldsymbol{b}$ that minimimize a loss function between the output $\hat{Y}_i$ and the training sample Y_i.

A common loss function in regression is quadratic loss, $L(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = \sum (Y_i - \hat{Y}_i)^2$, but we shall see that it is advantageous to penalize this.

Assume the input vector $\vec{x}_i$ is d-dimensional, and that there are K nodes in the network. Thus $\boldsymbol{W}$ is a $K \times d$ matrix of weights and $\vec{b}$ is length K vector of biases.

The architecture may be whatever we sensibly choose, and the activation functions are also quite flexible (but ReLU is the standard choice: it is almost linear, almost continuously differentiable, and it is relatively quick to train; leaky ReLU is also good).

At each node, the activation function $\sigma(\cdot)$ takes its inputs (initially a vector $\vec{x}_i$ for the ith observation, but in subsequent layers these are inputs from previous layers) and outputs $h_{k+1}\sigma(\vec{w}_k^\top \vec{h}_k + b_k)$ for $k = 1, \ldots, K-1$.

The activation function should be monotone. During training, back-propagation tells each node how much it should influence nodes in the next layer. If the activation function is not monotone, then increasing a node's weight can cause it to have less influence, which is the opposite of what is wanted. The result is chaotic training behavior, with the network being unlikely to converge.

A key concept in effective training is regularization, which is closely related to parsimony and sparsity. One wants to have the simplest possible model that is adequate to one's purpose. This often implies that the model is parsimonious (containing only a few terms) and this may be achieved by regularization (e.g., by forcing nodes with small weights to have zero weight).

Sparsity is essentially Ockham's Razor, and is a key idea in all the inferential paradigms. It takes many forms.

- most of the explanatory variables in $p >> n$ regressions are irrelevant,
- a nonnegative matrix can be approximately factored as a product of two matrices, each with low rank,
- the weights in a neural network take only a small number of distinct values (this is **quantization**).

2.1 Back-Propagation and Training

The idea for training a deep NN is to optimize its set of hyperparameters $\boldsymbol{\theta} = (\boldsymbol{W}, \vec{b})$ in order to approximate a function of interest. The quality of the approximation is evaluated using a cost function $J(\vec{x}, y; \boldsymbol{\theta})$ such as mean squared error (for regression) or KL Divergence (for classification).

To adjust these hyperparameters, their gradients are calculated with respect to the cost function. Since any value in $\boldsymbol{W}$ or $\vec{b}$ is related to the cost function $J(\vec{x}, y; \boldsymbol{\theta})$ through function composition, the **chain rule** is used to calculate partial derivatives. Due to the structure of the network, partial derivatives must be calculated starting at the output layer and working recursively backwards in a process known as **back-propagation**.

Consider the very simple neural network with a single scalar input x, two hidden layers, a single node at each layer, and differentiable activation functions:

$$x \longrightarrow h_1 \longrightarrow h_2 \longrightarrow y$$

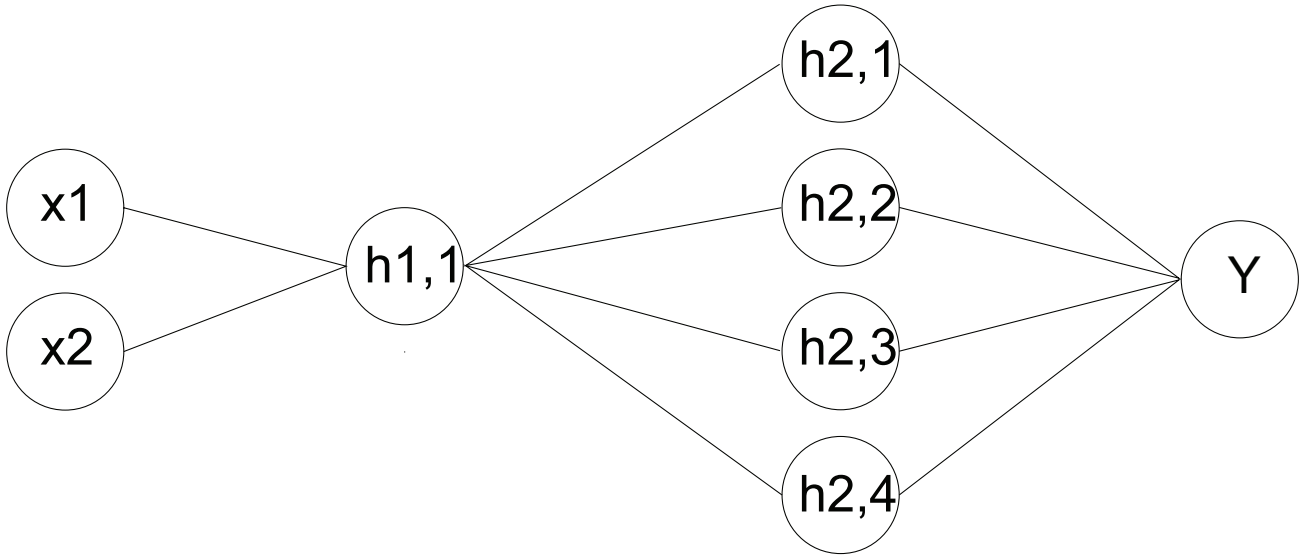
$$x \longrightarrow \sigma_1\big(w_1 x + b_1\big) \longrightarrow \sigma_2\big(w_2 h_1 + b_2\big) \longrightarrow \sigma_{out}\big(w_3 h_2 + b_3\big)$$

The $\frac{\partial J(\vec{x},y;\,\boldsymbol{\theta})}{\partial y}$ is easily found (since we directly observe the output layer), but the chain rule must be used to find the partial derivatives of each $\frac{\partial J(\vec{x},y;\,\boldsymbol{\theta})}{\partial w_i}$ or $\frac{\partial J(\vec{x},y;\,\boldsymbol{\theta})}{\partial b_i}$. In the example above,

$$\frac{\partial J(\vec{x}, y;\, \boldsymbol{\theta})}{\partial w_1} = \frac{\partial J(\vec{x}, y;\, \boldsymbol{\theta})}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial w_1}$$

Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

An Example



If $h_{1,1} = \sigma(w_{1,1}x_1 + w_{1,2}x_2 + b_1)$, then $\frac{\partial J(\vec{x},y;\,\theta)}{\partial h_{1,1}}$ can be calculated as

$$\frac{\partial J(\vec{x}, y; \boldsymbol{\theta})}{\partial h_{1,1}} = \frac{\partial J(\vec{x}, y; \boldsymbol{\theta})}{\partial y} \sum_{i=1}^{4} \frac{\partial y}{\partial h_{2,i}} \frac{\partial h_{2,i}}{\partial h_{1,1}}.$$

Once this has been found, the gradient of the cost function with respect to $\vec{w}$ can be calculated using back-propagation through the directional derivative

$$\bigtriangledown_{\vec{w}} J(\vec{x}, y; \boldsymbol{\theta}) = \left(\bigtriangledown_{\vec{w}} h_{1,1} \right) \frac{\partial J(\vec{x}, y; \boldsymbol{\theta})}{\partial h_{1,1}}.$$

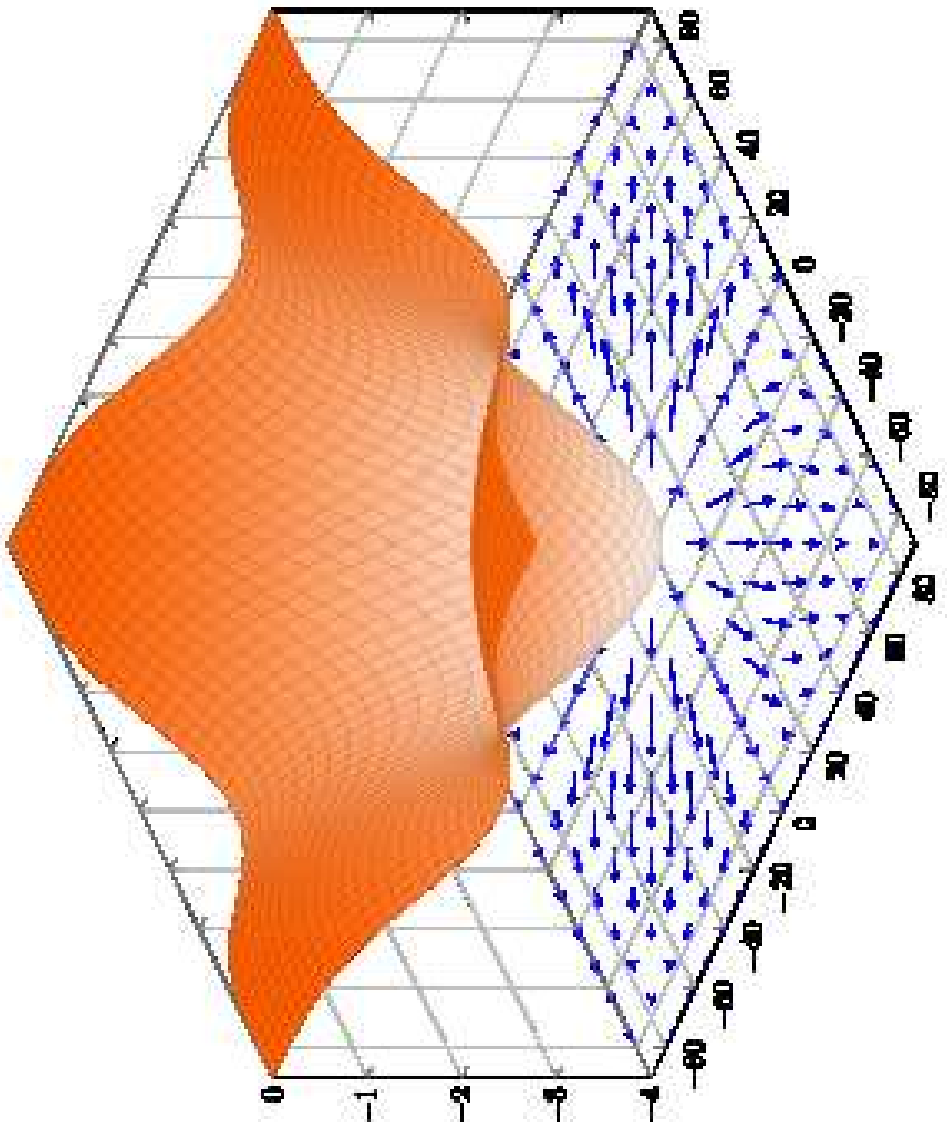
Recall that the directional derivative is the rate of change in a given direction:

$$\bigtriangledown_{\vec{w}} f(\vec{x}) = \lim_{h \to 0} \frac{f(\vec{x} + h\vec{w}) - f(\vec{x})}{h}.$$

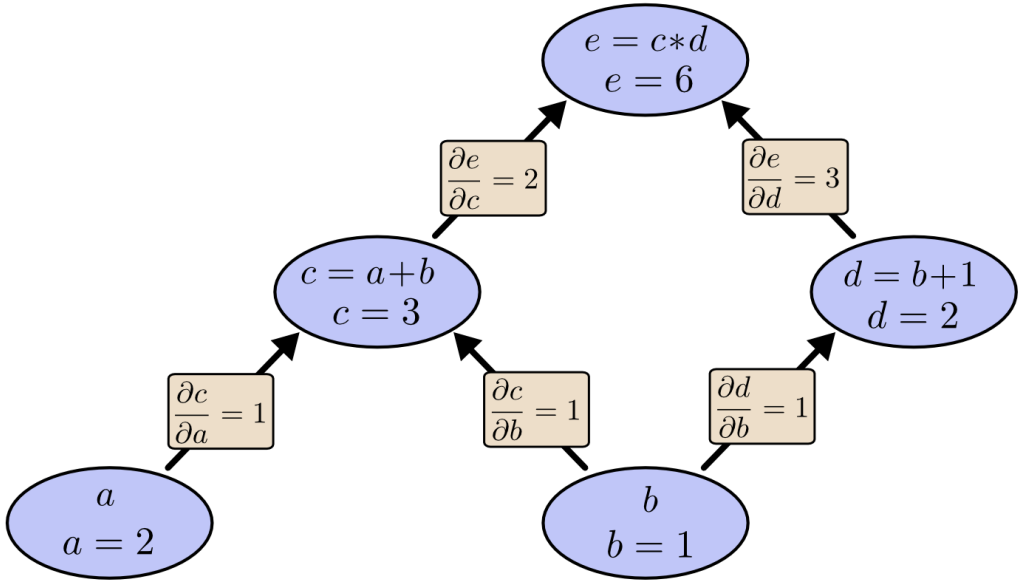
For a three-dimensional system, the gradient is

$$\bigtriangledown f(x, y, z) = \frac{\partial f}{\partial x}(1, 0, 0) + \frac{\partial f}{\partial y}(0, 1, 0) + \frac{\partial f}{\partial z}(0, 0, 1).$$

The gradient at a point gives the direction of steepest change in the function.



A **computation graph** shows the steps in a calculation. Each node represents a variable (scalar, vector, matrix, or tensor). When a variable y is found by applying an operation to variable x, one draws a directed edge from x to y. At each node, one can compute its derivative w.r.t. all of its inputs.



The General Back-Propagation Algorithm treats each node in the computational graph of a neural network not as scalars, vectors, or matrices, but as tensors of appropriate dimensions.

Tensor calculus is exactly the same as vector calculus. We denote the gradient of output y in the "direction" of $\mathbf{W}$ as $\bigtriangledown_{\mathbf{W}} y$. In a sense, we have flattened out the tensor $\mathbf{W}$ as a vector. Just as a vector has integer indices and a matrix entry is indexed by its row and column, a 3-dimensional tensor has indices that are triples of integers.

If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then the chain rule for a tensor is:

$$\bigtriangledown_{\mathbf{X}} z = \sum_{j \in \{\textbf{index set of } \mathbf{Y}\}} \left(\bigtriangledown_{\mathbf{X}} \mathbf{Y}_j \times \frac{\partial z}{\partial \mathbf{Y}_j} \right)$$

If $\mathbf{X} = \mathbf{AB}$ and $z = f(\mathbf{X})$ and $\bigtriangledown_{\mathbf{X}} z = \mathbf{G}$ then

$$\bigtriangledown_{\mathbf{A}} z = \mathbf{G}\mathbf{B}^T$$

$$\bigtriangledown_{\mathbf{B}} z = \mathbf{A}^T\mathbf{G}$$

The intuition behind back-propagation is not about the rules of differentiation. Let u_1 be the input, u_n be the output, and $u_{2 \leq i \leq n-1}$ be the intermediate nodes in a computation graph. Let D=$\{\bigtriangledown_{u_i} u_n \mid i = 1, \ldots, n\}$. The goal is to find the weights and intercepts by working backwards from u_n all the way to u_1, moving (sort of) in the direction of most improvement. The gradients of all parameters are needed in doing the optimization.

The "sort of" reflects the fact that stochastic gradient descent takes steps in the best direction at a given point. It does not move continuously.

2.2 Some Back-Propagation Details

There are many other issues and technicalities that arise in training a deep network.

One common practice is to scale all the inputs to lie between 0 and 1. Unscaled inputs can slow or destabilize the training. The transformation is $(x - \min)/(\max - \min)$.

It is also good to scale the outputs to lie between 0 and 1. Unscaled outputs in regression problems can cause exploding gradients, which also cause the training to fail.

Scaling the data has the useful property of putting all the weights and biases in the nodes on a common footing, so that one can better assess the important variables that contribute to the output.

A difficulty with scaling arises when one applies the trained network to new data. The new data, when given the same scaling as was used with the training data, may not lie between 0 and 1. If the overage or underage is small, that is not a serious problem. If it is large, there can be an issue.

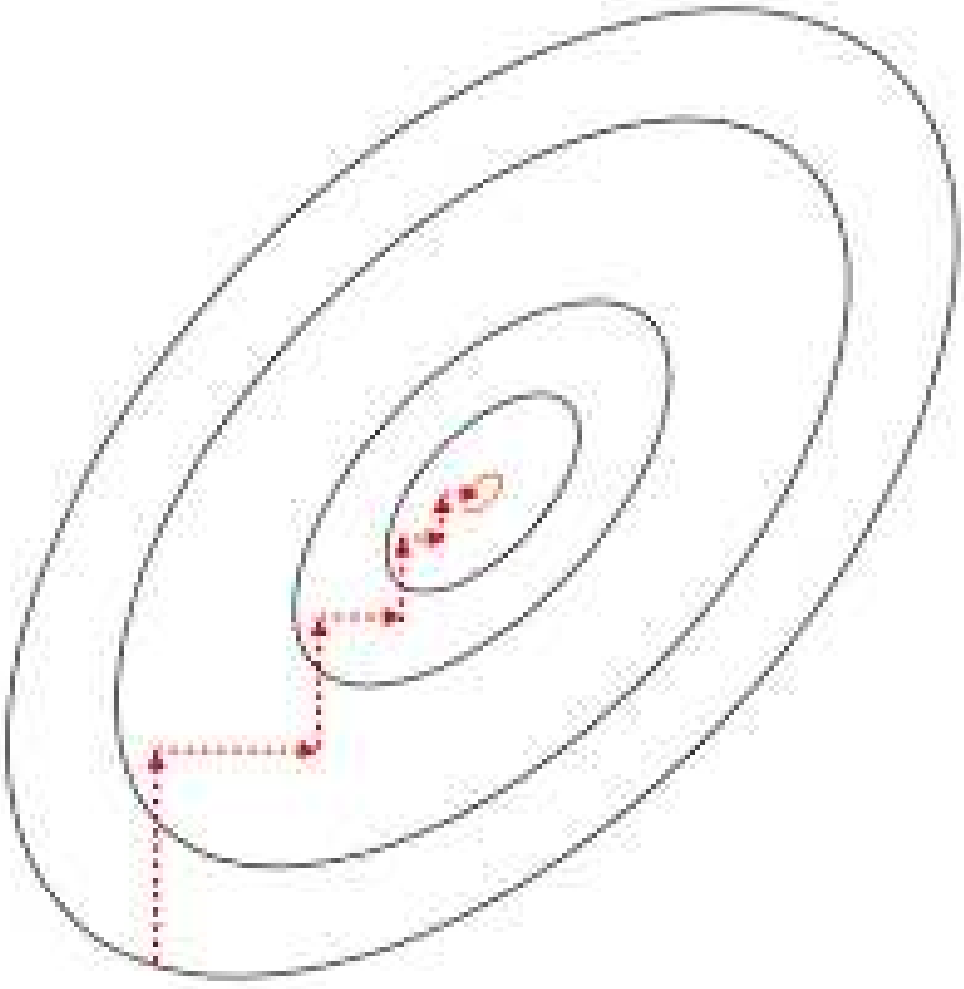
With some BIG data sets, it may be hard to find the maximum and the minimum.

Some people normalize the input and output data by subtracting the mean and dividing by the standard deviation, but this makes the strong assumption that the inputs and outputs have a Gaussian distribution.

Our previous discussion of back-propagation implicitly assumed that all of the data are used in training the DNN. In practice, that is time consuming.

In optimization by coordinate gradient descent, one cycles through the components of the input vectors, taking the derivative with respect to each coordinate, and moving to the location on the cost-function surface that minimizes the cost function along that coordinate.

With DNNs, there are millions of coordinates and often millions of data points. Gradient descent is too slow. Instead, one uses **stochastic gradient descent** (SGD).



In SGD, one trains with subsamples of the data, called **minibatches**. (Batch methods train with all the data, on-line methods train with one observation at a time, and minibatch methods are the compromise used in essentially all DNN training.)

The size of the minibatch is determined by several factors:

- Large batches more accurately estimate the gradient.
- Small batches underuse multicore processors.
- For parallel processing, memory demands scale with minibatch size.
- With GPUs, batch sizes that are powers of 2 are efficient (32 to 256).
- Wilson and Martinez (2003) report that small minibatches regularize.

The standard approach is to choose the data in the minibatches at random from the training data. But as a statistician, I wonder if it is smarter to choose training data that are very dissimilar (at first) with respect to an appropriate metric, and then becoming more similar as training proceeds.

However, in many datasets, the initial ordering of the data can show significant dependence (e.g., successive frames in a movie, digits written by the same person). Such dependence introduces bias when estimating the gradient.

The standard approach is to randomly rearrange the initial data, and then work with the rearrangement going forward.

The SGD algorithm is:

1. Set a learning rate ϵ_k.
2. Pick an initial parameter value of weights and biases $\boldsymbol{\theta}_1$.
3. For $k = 1, \ldots,$ stopping criterion
   - sample a minibatch of size m, $(\vec{x}^{(1)}, \ldots, \vec{x}^{(m)})$ and corresponding $(y^{(1)}, \ldots, y^{(m)})$
   - estimate the gradient $\hat{g} = \frac{1}{m}\big(\bigtriangledown_{\boldsymbol{\theta}} \sum_{i=1}^{m} J(\vec{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}_k)\big)$
   - update so $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \epsilon_k \hat{g}$.

The learning rate determines how much the parameter changes in the direction of the estimated gradient. Jordan (2018) used symplectic geometry to get bounds on how quickly NNs can train.

Usually, for k between 1 and τ, one sets

$$\epsilon_k = (1 - \alpha_k)\epsilon_0 + \alpha_k \epsilon_\tau$$

for $\alpha_k = k/\tau$. After the τth iteration, it is constant. But there are reasons to let the step size go to zero.

To choose ϵ_0, ϵ_τ and τ, one can monitor a plot of the approximate empirical cost function against time. Commonly, ϵ_τ is 1% of ϵ_0 and τ is perhaps 300.

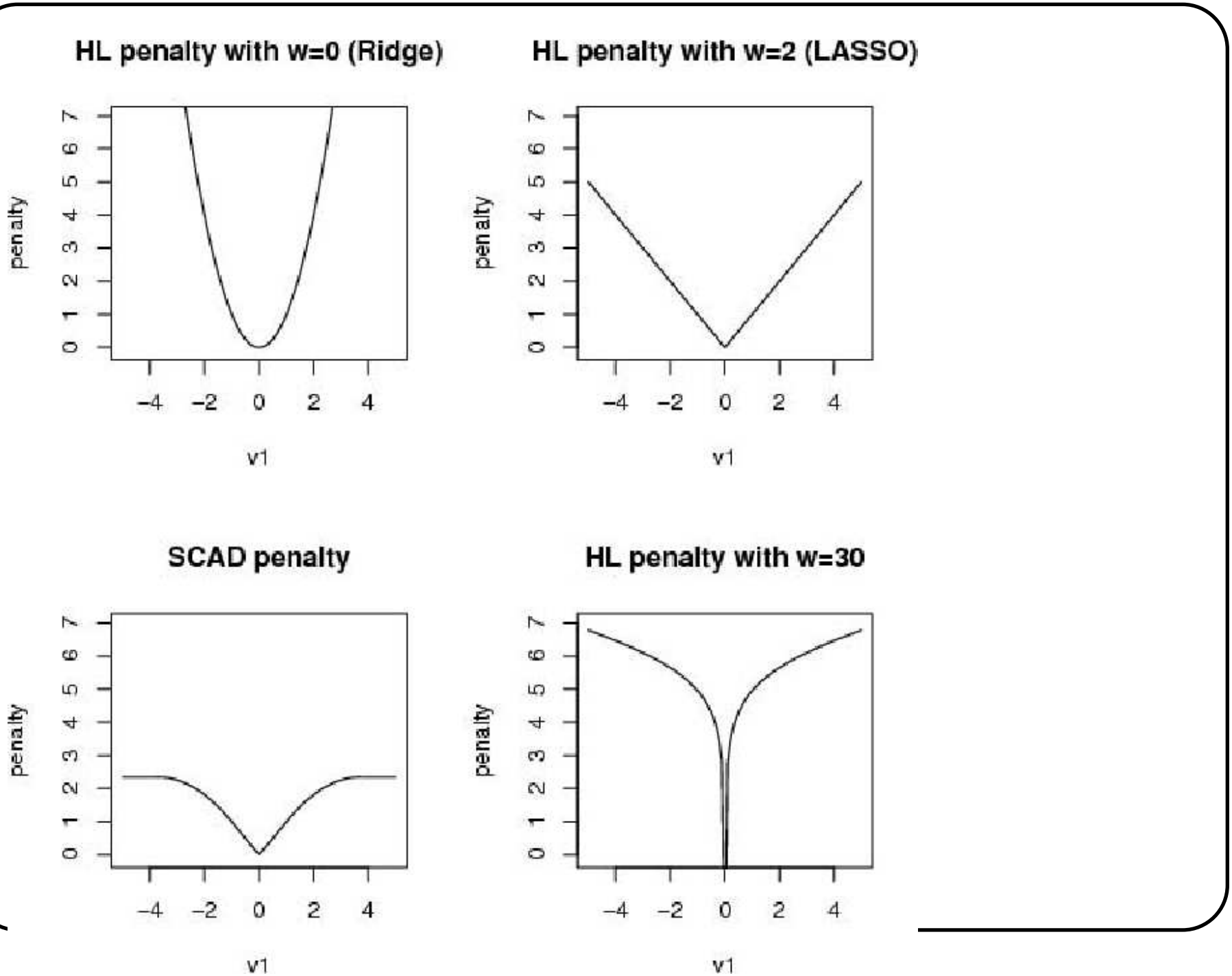
If ϵ_0 is too large, then SGD is unstable and wildly overshoots the true minimizer and then overcorrects in the next iteration.

One full pass through the training data is called an **epoch**.

The previous work assumed that the cost function was differentiable (which requires differentiable activation functions). But ReLU and many other activation functions are not differentiable.

The optimization world has many tricks for handling non-differentiable cost functions. Subgradients are one tool. If $f : \mathbb{R}^d \to \mathbb{R}$ is a real-valued convex function, then a vector $\vec{v} \in \mathbb{R}^d$ is a subgradient at a point x_0 if for all $\vec{x} \in D$ one has $f(\vec{x}) - f(\vec{x}_0) \geq \vec{v}^\top (\vec{x} - \vec{x}_0)$.

When the cost function is not convex, other tricks are available, especially when the cost function is a difference of two convex functions (SCAD, LASSO, and many others). The **Difference of Convex Functions Algorithm** applies.


HL penalty with w=0 (Ridge)
HL penalty with w=2 (LASSO)
SCAD penalty
HL penalty with w=30
penalty
v1
0 1 2 3 4 5 6 7
−4 −2 0 2 4

Another trick is to use weight decay. This performs regularization that improves generalization and thus predictive accuracy on the test data.

In weight decay, after each iteration, one multiplies the weights by a number slightly less than 1. This is very much like the ridge regression penalty on the sum of the squares of the coefficients in the regression. It is equivalent to including a ridge regression penalty on the cost function.

Other complications arise from use of 32-bit versus 64-bit floating point data types. These types of problems are handled almost automatically by TensorFlow and PyTorch.

Cost functions in DNNs must be written as averages, and they depend only on the weights, biases, and training data. Quadratic cost is often used for regression, and is

$$\sum_j (\hat{y}_j - y_j)^2.$$

Sometimes there is an additional penalty on the sum of the squares or absolute values of the weights and biases, to shrink them towards zero.

Similarly, one can have a cost function that reflects mean absolute error. This does not weight large discrepancies as strongly, and it may also have a penalty term. Other options are Huber loss and SCAD.

If one is doing classification with M categories, where the empirical probability in the training sample for the mth category is $\hat{p}_m$, one commonly uses cross entropy:

$$-\sum_{1}^{M}\sum_{j} I(\hat{y}_j \textbf{ correctly labelled}) \ln(\hat{p}_m).$$

Another standard choice is Kullback-Leibler divergence, or

$$\sum_{m=1}^{M} \hat{p}_m \ln \hat{p}_m / q_m$$

where q_m is the proportion of the minibatch or training sample that is correctly classified. (This can be written as an average.)

When classifying with M categories, assume the output layer has produced a vector $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$ with M components. Each component is an unnormalized log probability, so

$$z_i = \ln \mathbb{P}[y \textbf{ is category } m \,|\, \boldsymbol{x}]$$

for $m = 1, \ldots M$.

To find the probability of category m use the **softmax** function:

$$\textbf{softmax}(\boldsymbol{z})_m = \frac{\exp(z_m)}{\sum_{i=1}^{M} \exp(z_i)}.$$

For a classification problem, consider a NN with a single hidden layer. We train it with minibatch SGD. Each minibatch is represented by $\boldsymbol{X}$ with labels $\boldsymbol{y}$.

To simplify things, assume all biases are 0. We then compute the hidden features as $\max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$ (i.e., we use ReLU activation), obtaining the output $\boldsymbol{H}$.

The predictions of the unnormalized log-probabilities are $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We use a cross-entropy operation to compute the cost between the targets $\boldsymbol{y}$ and the probability distribution determined by the unnormalized log-probabilities.

The cross-entropy defines the cost, and minimizing it performs maximum likelihood estimation for the classifier. But it is good to add a regularization term with penalty λ, so the total cost is

$$J = J_{MLE} + \lambda\Big(\sum_{i,j}(\mathbf{W}^{(1)}_{(i,j)})^2 + (\mathbf{W}^{(2)}_{(i,j)})^2\Big)$$

$$\frac{\partial J}{\partial \mathbf{W^{(2)}}} = 2\lambda\mathbf{W}^{(2)} + \mathbf{H^T}\frac{\partial \mathbf{J}}{\partial \mathbf{U}^{(2)}}$$

$$\frac{\partial J}{\partial \mathbf{W^{(1)}}} = 2\lambda\mathbf{W}^{(1)} + \mathbf{X^T}\frac{\partial\Big(\frac{\partial \mathbf{J}}{\partial \mathbf{U}^{(2)}}(\mathbf{W}^{(2)})^{\mathbf{T}}\Big)}{\partial \mathbf{U}^{(1)}}$$

where $\boldsymbol{U}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)}$ and $\boldsymbol{U}^{(2)} = \boldsymbol{H}\boldsymbol{W}^{(2)}$.

This training is represented in the following computation graph.

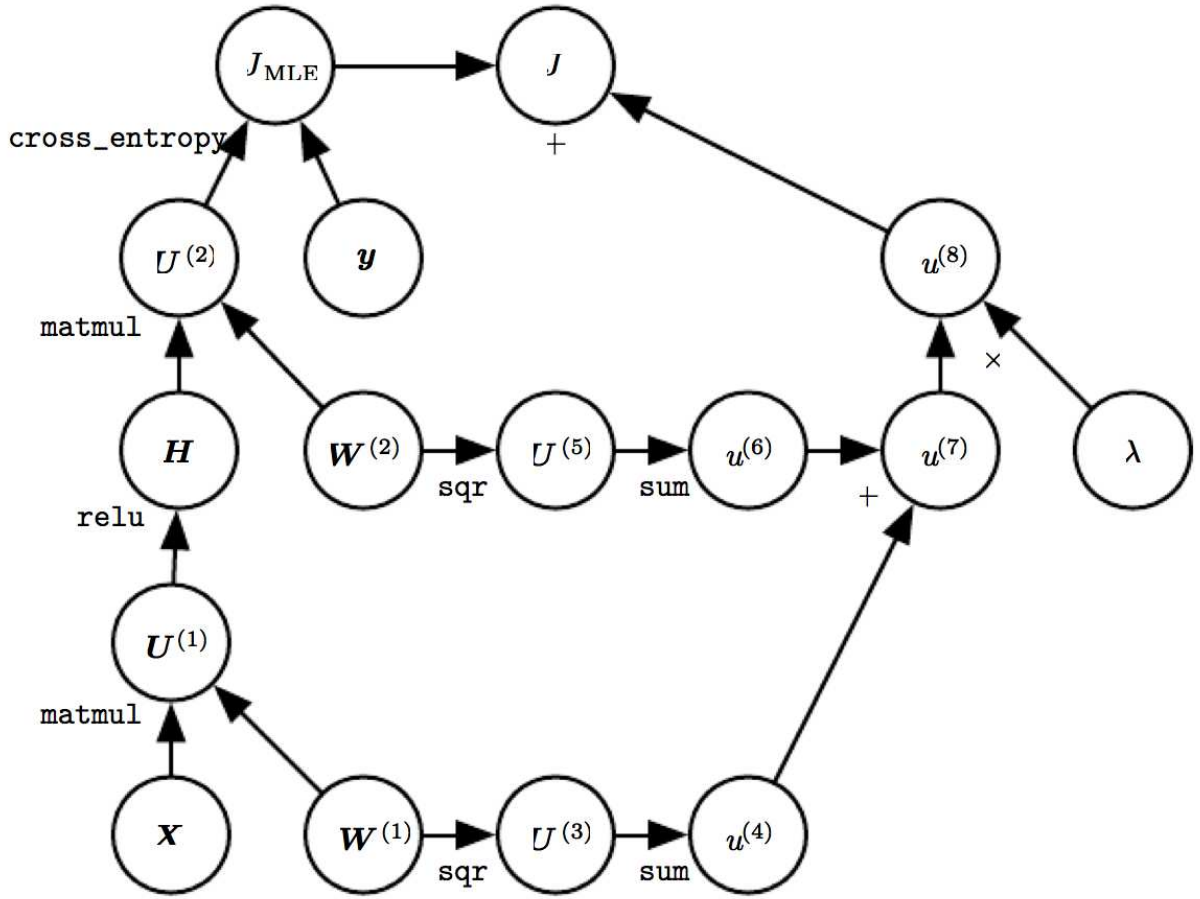


Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

2.3 Network Architecture

Deeper networks with the same number of nodes but more layers have empirically been found to **generalize** better. This means that they perform better on test sets.

This is particularly the case for problems with many abstractions (such as image recognition) and this occurs because the deeper network can construct more complex functions through larger numbers of compositions.

Often the most impactful parameter to tune when training a network is its depth.

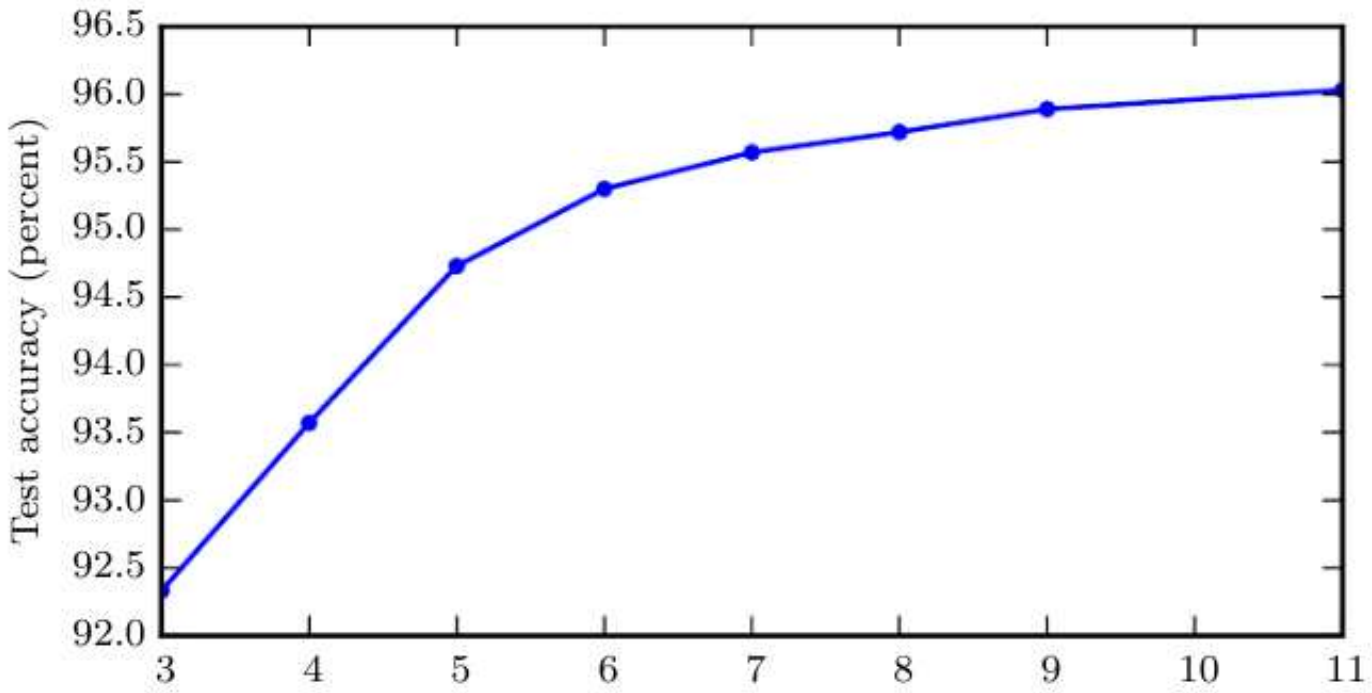


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

NB: To train a deeper or larger network sufficient amounts of data must exist to prevent overfitting.

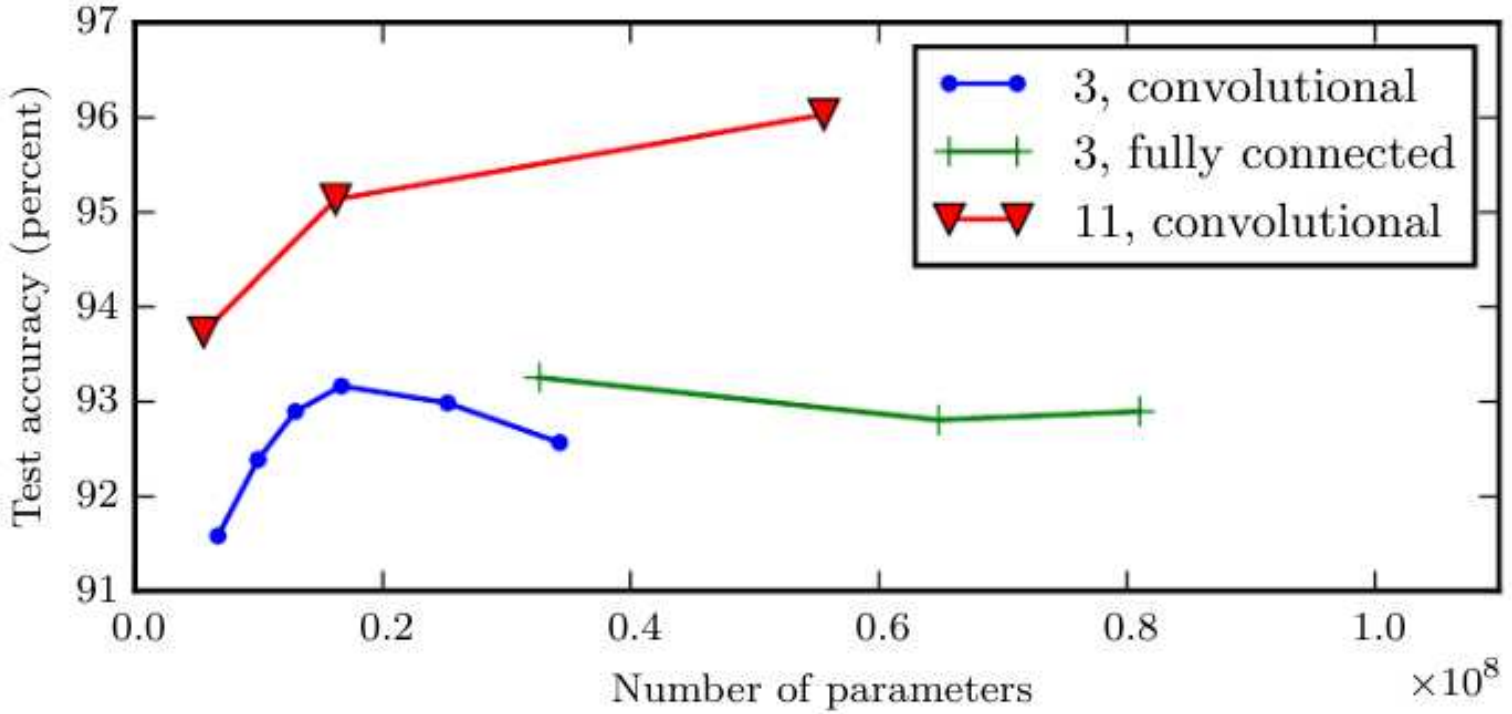


Fig. 6.7 from (Goodfellow et.al 2015) showing that the depth of layers (represented by the different colored lines) has a much larger effect than increasing the number of units.

However, while there are gains to generalization, deeper networks are harder to train for two reasons:

- The problem is more non-convex, because of a larger number of connections between the layers.
- A deeper network is more likely to experience the **vanishing/exploding gradient problem** when trained with back-propagation.

The capacity of a classifier or a regression refers to its flexibility in describing a wide range of situations. Too much flexibility is bad—it introduces high variance but low bias (e.g., 1-nearest neighbor classifiers). And too little flexibility is also bad—one has small variance but large bias (e.g., linear discriminant analysis).

2.4 Initialization

Successful DNNs require good initialization of the weights and biases. A bad initialization can:

- prevent convergence
- cause convergence to a minimum with high cost (loss)
- lead to very slow convergence.

Also, minima with equivalent loss can have very different performances on the test data.

Initialization is largely heuristic and understudied. But it can have a large impact on the quality of the results.

The only real dictum is that the initialization must break symmetry between different nodes. If two hidden nodes have the same initialization and the same activation function and the same inputs, then they will train to the same result.

People initialize the weights randomly and the biases by judgement. The weights are typically drawn from a Gaussian or uniform distribution. The variance of those distributions has a large effect on both the local minimum that is discovered and the generalization error on test data.

Large initial weights do help in symmetry breaking. They also preserve signal during training, since large values in the weight matrix lead to large values in the multiplication (conversely, setting weights to zero kills any initial signal). But if weights are too large, then their values can explode during training (strong signal with large weights is self-reinforcing).

In recurrent NNs (not discussed), large weights can cause sensitivity to small perturbations of the input data, so the calculation of the output in the forward propagation appears random.

One heuristic for a NN with n inputs and r outputs is to draw weights independently from the uniform distribution on $(-n^{-0.5}, n^{-0.5})$. But Glorot and Bengio (2010) propose drawing weights independently as

$$W \sim U\left(-\sqrt{\frac{6}{n+r}}, \sqrt{\frac{6}{n+r}}\right).$$

This is a compromise between initializing all layers to have the same variance in the gradient and having all layers have the same activation variance. The derivation assumes linearity in the NN, which is false, but often linear heuristics generalize to nonlinear situations.

As a statistician, I wonder whether there is value from initializing the weights as antithetic random variables, so that the negative correlations break the symmetry.

I also suspect that initialization on a coarse integer grid in $\mathbb{R}^p$ for p-dimensional inputs could do well. If the grid contains zeroes, that might facilitate variable selection.

For many activation functions, the biases are set to zero. However, with ReLU it can avoid **saturation** to make the biases slightly positive, say 0.1. Saturation occurs when the node usually produces values from the extremes of its activation function's left or right limits, and this can reduce the rate of learning.

If a hidden node is a gate that controls whether other nodes participate in a function evaluation, you should initialize so that most of the time, at the beginning, the nodes participate (so the DNN can learn whether they should participate, rather than excluding them a priori). For example, suppose one node outputs z in $\{0, 1\}$ and another node outputs u, and the function takes the product zu. If z is mostly 0, then the NN cannot learn quickly about the value of u.

The output layer is special. It turns out to be helpful if the initialization preserves the marginal distribution of the outputs in the training sample. For the output layer, encourage (but do not enforce) that the weights at that layer are zeroes. Then pick the intercepts to match the marginals.

2.5 Training

Back-propagation has variants and details that deserve attention.

Stochastic gradient descent is often slow. Polyak (1964) proposed **momentum** as a way to accelerate learning when when there is large curvature, or small but consistent gradients, or noisy gradients.

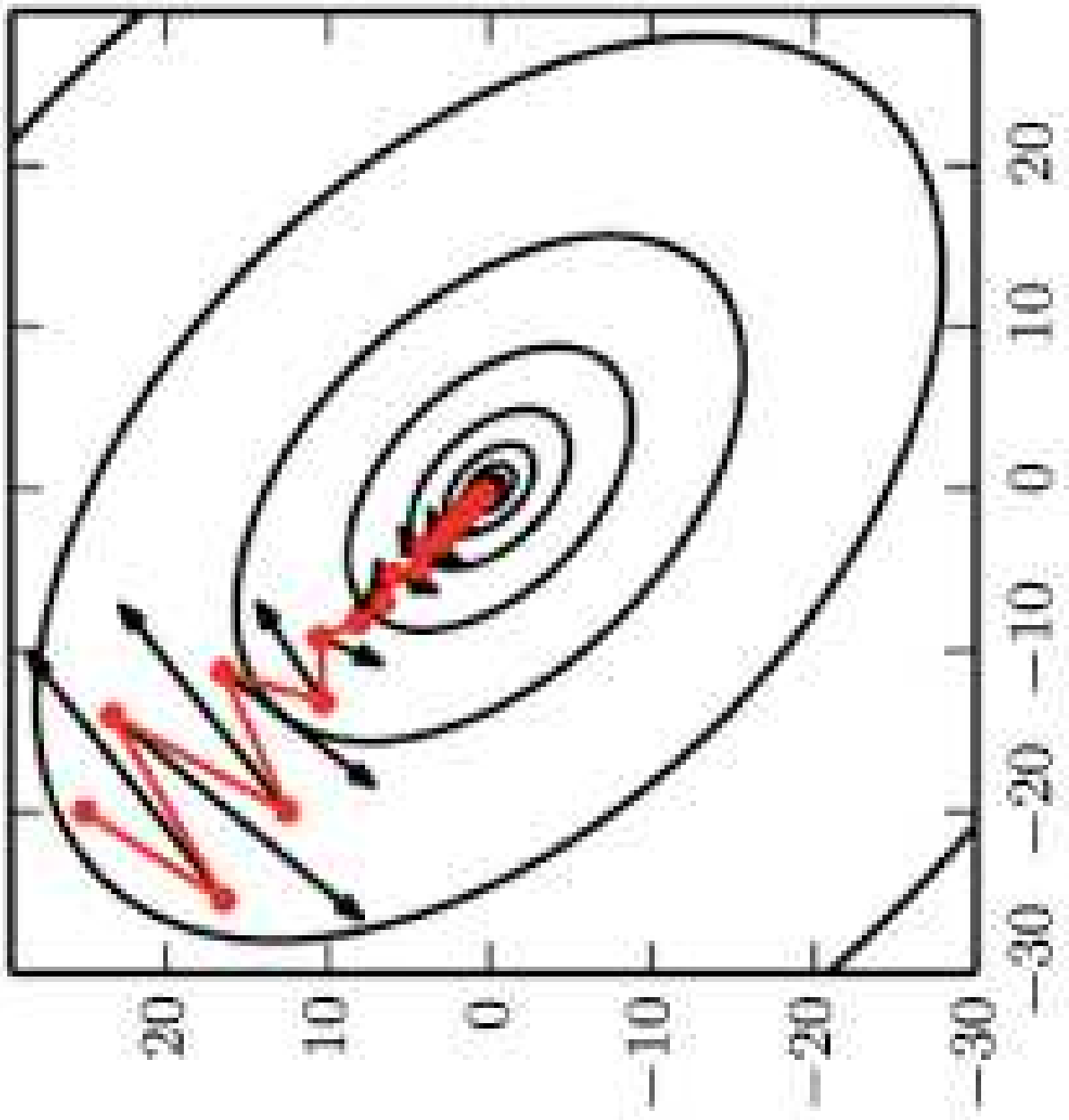
The term momentum derives from an analogy with physics, implying that the direction of the descent path tends to change slowly, instead of instantaneously responding to the gradient.

Momentum uses a vector $\boldsymbol{v}$ (similar to velocity) and a hyperparameter $\alpha \in [0, 1]$ that controls the rate at which contributions from previous gradients exponentially decay.

For $\boldsymbol{\theta}_t$ the vector of weights and biases at time t, the update rules are:

$$\begin{aligned} \boldsymbol{v}_{t+1} &= \alpha \boldsymbol{v}_t - \epsilon \bigtriangledown_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \right) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta} + \boldsymbol{v}_{t+1}. \end{aligned}$$

The following figure gives a visual representation for how momentum works in a two-dimensional quadratic loss minimization.


20
10
0
−10
−20
−30
20
10
0
−10
−20
−30

The contours indicate quadratic loss with a poorly conditioned Hessian. The red path is that taken by momentum learning. The black arrows indicate the steps that would have been taken by unadjusted stochastic gradient descent.

The velocity vector $\boldsymbol{v}$ is a weighted sum of previous gradients. If α is large relative to ϵ, the previous gradients contribute largely to the current direction.

Before, the size of the step was the norm of the gradient multiplied by the learning rate. With momentum, the step size depends on how large and how aligned a sequence of gradients are.

If the momentum algorithm repeatedly sees the same gradient $\boldsymbol{g}$, it will accelerate in the direction $-\boldsymbol{g}$ to a maximum velocity of $\epsilon\|\boldsymbol{g}\|/(1-\alpha)$.

Common choices of α are 0.5, 0.9, and 0.99. As with the learning rate, α may change over time, starting small and then increasing.

Nesterov momentum is a variation introduced by Sutskever et al. (2013) based on previous work by Nesterov (a world leader in numerical optimization). Here the update rules are:

$$\begin{aligned}\boldsymbol{v}_{t+1} &= \alpha\boldsymbol{v}_t - \epsilon \bigtriangledown_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t + \alpha\boldsymbol{v}_t), \boldsymbol{y}_i) \right) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta} + \boldsymbol{v}_{t+1}.\end{aligned}$$

Nesterov momentum increases the asymptotic rate of convergence, but this seems to be a theoretical result that does not add value in practice.

Duchi et al. (2011) proposed **AdaGrad**, which separately adapts the learning rate for each parameter by scaling them to be inversely proportional to the square root of the sum of all the previous squared values of the gradient. So parameters with large partial derivatives of the loss function have a rapid decrease in the learning rate, while those with small partials have a slower decrease in their learning rate.

These leads to greater progress in the less steep directions of the parameter space.

AdaGrad has some good theoretical properties, but has failed on some datasets. So it is a tool to use cautiously.

Set $\boldsymbol{r}_0 = \mathbf{0}$. Given a global learning rate ϵ, an initial parameter $\boldsymbol{\theta}$, and a small constant δ (about 10^{-7}) for stability, the AdaGrad algorithm works as follows.

For $t = 1, \ldots,$ stopping criterion,

- Sample a minibatch $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ and corresponding $(\boldsymbol{y}_1, \ldots \boldsymbol{y}_m)$.
- Compute the gradient $\boldsymbol{g}_t = \bigtriangledown_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \right)$.
- Accumulate the squared gradient: $\boldsymbol{r}_{t+1} = \boldsymbol{r}_t + \boldsymbol{g}_t \odot \boldsymbol{g}_t$ (the Hadamard product).
- Calculate $\Delta \boldsymbol{\theta}_t = \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}_{t+1}}} \odot \boldsymbol{g}_t$ (division and square root are applied element-wise).
- Set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta \theta_t$.

Another common training algorithm is **RMSProp** (Hinton, 2012). It adapts AdaGrad to nonconvex settings. The key idea is to forget the distant past when summing the historical squared values of the gradient.

Set $\boldsymbol{r}_0 = \mathbf{0}$. Given a global learning rate ϵ, an initial parameter $\boldsymbol{\theta}$, a decay rate ρ and a small constant δ (about 10^{-6}, the RMSProp algorithm works as follows:

For $t = 1, \ldots,$ stopping criterion,

- sample a minibatch $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ and corresponding $(\boldsymbol{y}_1, \ldots \boldsymbol{y}_m)$.
- Compute the gradient $\boldsymbol{g}_t = \bigtriangledown_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \right)$.
- Accumulate the squared gradient: $\boldsymbol{r}_{t+1} = \rho \boldsymbol{r}_t + (1 - \rho) \boldsymbol{g}_t \odot \boldsymbol{g}_t$ (the Hadamard product).
- Calculate $\Delta \boldsymbol{\theta}_t = \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}_{t+1}}} \odot \boldsymbol{g}_t$ (division and square root are applied element-wise).
- Set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta \theta_t$.

There are other algorithms that some favor, such as RMSProp with Nesterov momentum and Adam (from "adaptive moments"). These are not on our critical path for learning, but be aware that alternatives exist.

A different class of algorithms uses second-order gradient methods (which requires tractable loss functions). Newton's method gives the flavor, but as with first-order gradient methods, there are many variants.

Recall the vector Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \bigtriangledown_{\vec{\theta}} J(\boldsymbol{\theta}_0) + \tfrac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$.

The Hessian is the matrix with i, j entry

$$\frac{\partial^2}{\partial \theta_i \partial \theta_j} J(\boldsymbol{\theta}).$$

It is the Jacobian of the gradient.

Solving for the critical point of this function gives the update rule

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}^{-1} \bigtriangledown_{\vec{\theta}} J(\boldsymbol{\theta})_t.$$

Let $\boldsymbol{\theta}_0$ be the initialization. Then the algorithm for Newton's method is as follows:

For $t = 1, \ldots,$ stopping criterion,

- Sample a minibatch $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ and corresponding $(\boldsymbol{y}_1, \ldots \boldsymbol{y}_m)$.
- Compute the gradient $\boldsymbol{g}_t = \bigtriangledown_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \right)$.
- Compute the Hessian: $\boldsymbol{H} = \bigtriangledown^2_{\vec{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \right)$.
- Compute the inverse Hessian.
- Compute the update $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}^{-1}\boldsymbol{g}$.

In general, I don't recommend second-order methods. In this case, the Hessian must be positive definite, which is not guaranteed. The loss function is (almost always) nonconvex and has saddlepoints, which are a problem.

Inverting the Hessian is computationally expensive. There are numerical tricks to avoid the inversion, but they are still expensive.

If the eigenvalues of the Hessian are not all positive, which occurs at a saddlepoint, then Newton's method can push the path in the wrong direction.

One partial fix is to mimic the regularization from ridge regression:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - [\boldsymbol{H}(\boldsymbol{J}(\boldsymbol{\theta}_t)) + \boldsymbol{\alpha}\boldsymbol{I}]^{-1} \bigtriangledown_{\vec{\theta}} J(\boldsymbol{\theta})_t$$

for some $\alpha > 0$.

A way to avoid calculation of the inverse Hessian is **conjugate gradient descent**. It arises from an analysis of the problems associated with steepest descent, which, when applied in a quadratic bowl, zig-zags inefficiently.

Here the next search direction is conjugate to the previous search direction, in the sense that it does not undo progress made in that direction.

Formally, two directions $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ are conjugate if $\boldsymbol{g}_1^\top \boldsymbol{H} \boldsymbol{g}_2 = 0$, where $\boldsymbol{H}$ is the Hessian. For conjugate $\boldsymbol{g}_t$ and $\boldsymbol{g}_{t+1}$, the next search direction has the form:

$$\boldsymbol{g}_{t+1} = \nabla_{\vec{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{g}_t$$

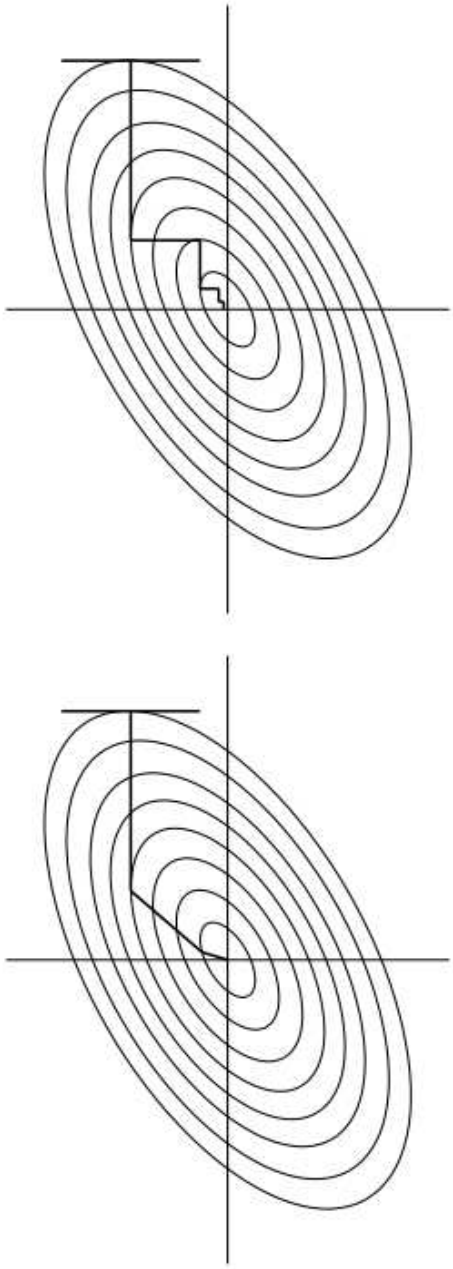


Figure 6.14: Steepest descent vs. conjugate gradient.

The direct approach to conjugacy is to find the eigenvectors of $\boldsymbol{H}$ in order to find β_t. But that is hard. Two popular alternatives for calculating β_t are:

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\vec{\theta}} J(\boldsymbol{\theta})_t^\top \nabla_{\vec{\theta}} J(\boldsymbol{\theta})_t}{\nabla_{\vec{\theta}} J(\boldsymbol{\theta})_{t-1}^\top \nabla_{\vec{\theta}} J(\boldsymbol{\theta})_{t-1}}.$$

2. Polak-Ribière:

$$\beta_t = \frac{\left(\nabla_{\vec{\theta}} J(\boldsymbol{\theta})_t - \nabla_{\vec{\theta}} J(\boldsymbol{\theta})_{t-1}\right)^\top \nabla_{\vec{\theta}} J(\boldsymbol{\theta})_t}{\nabla_{\vec{\theta}} J(\boldsymbol{\theta})_{t-1}^\top \nabla_{\vec{\theta}} J(\boldsymbol{\theta})_{t-1}}.$$

For a quadratic surface in $\mathbb{R}^k$, the conjugate gradient method requires at most k line searches to find the minimum. But DNNs are only locally approximately quadratic.

With any of these algorithms, one may use **block coordinate descent**. Instead of finding the gradient with respect to one coordinate at a time (coordinate descent) or with respect to all variables (gradient descent), one descends by taking derivatives for a subset of the variables.

This can help when there are natural groups of variables that influence the outputs relatively independently of each other. But in deep NN, where the parameters are weights and intercepts, there may be little insight into such groups.

This is analogous to minibatching, and may achieve faster convergence.

2.6 Batch Renormalization

Ioffe and Szegedy (2015) invented **batch renormalization**. It adaptively reparameterizes the objective function, and leads to faster training.

The gradient tells us how to update each parameter, but it implicitly assumes that all the other parameters are held constant. The Hessian allows us to take account of pairwise interactions in learned parameters, but that is not sufficient. In deep NNs, the compositions create complex dependencies.

Batch renormalization reduces the problem of coordinating change across layers. It may be applied to any input or hidden layer. Let $\boldsymbol{B}$ be the the minibatch of inputs to a layer, so that entry b_{ij} corresponds to the jth input to node i in the layer.

To normalize $\boldsymbol{B}$, for each entry, subtract the row mean (i.e., for node i) and divide by (almost) the row standard deviation. This creates a new matrix $\boldsymbol{B}^*$. For training, the network operates on $\boldsymbol{B}^*$ exactly the way that it previously acted on $\boldsymbol{B}$.

The slight departure from the standard deviation is that we set

$$s_i = \sqrt{\delta + \frac{1}{m}\sum_j (b_{ij} - \bar{b}_i)^2}$$

where δ is a small constant, say 10^{-8}, to avoid the chance of division by zero.

One back-propagates through the operations of computing the node mean and standard deviation. This ensures that the gradient never proposes a change that only acts to increase the mean or the standard deviation at node i.

When applied to the test sample, the means and standard deviations are replaced by running averages collected on the minibatches during training.

Normalizing the mean and standard deviation of each node can reduce the range of expression that node has. To preserve the expressive power of the NN, one uses $\boldsymbol{\gamma}_1 \boldsymbol{B}^* + \boldsymbol{\gamma}_0$ instead of $\boldsymbol{B}^*$.

This looks like it has no effect, but it does. The new parameterization can represent the same family of functions of the inputs as the old, but it has better learning dynamics. In the old parameterization, the mean of $\boldsymbol{B}$ had complex dependence on interactions in lower layers. In this new parameterization, the mean depends solely on $\boldsymbol{\gamma}_0$, which is learned as part of the training.

Networks with the same form of activation functions within the same layer have the form $\phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{\delta})$. One could apply batch normalization to either the inputs $\boldsymbol{X}$ or to the $\boldsymbol{X}\boldsymbol{W} + \boldsymbol{\delta}$. Ioffe and Szegedy recommend the latter.

More precisely, $\boldsymbol{X}\boldsymbol{W} + \boldsymbol{\delta}$ should be replaced by a normalized version of $\boldsymbol{X}\boldsymbol{W}$. The intercept term is redundant since we shall add in the $\boldsymbol{\gamma}_0$.

Since both inputs and outputs are normalized, I think this advice applies when the gradient is being calculated. Otherwise, it doesn't seem to make sense.

2.7 Polyak Averaging

Polyak Averaging is the practice of using the mean of recent parameter estimates. Specifically, take

$$\boldsymbol{\theta}_{t+1} = \frac{1}{t}\sum_{i=1}^{t} a_t \boldsymbol{\theta}_i.$$

This has good theoretical properties for convex optimization, but its justification in nonconvex applications is heuristic.

The motivation is that sometimes the estimates ricochet back and forth across a valley. In that case, the average of the estimates will place you in the middle of the valley, where you want to be.

You will need to forget the distant past.

2.8 The Robbins-Munro Algorithm

The basis for stochastic optimization is the **Robbins-Munro algorithm** from 1951. It was originally developed to solve for the root of an equation with noisy function evaluations. Specifically, one cannot observe the function $f(\theta)$, which $f(\theta) = 0$ has a unique root θ^*. But one can observe a random variable $X(\theta)$ where $\mathbb{E}[X(\theta)] = f(\theta)$.

That work laid the foundation for modern stochastic optimization.

They proved that the estimator

$$\theta_{n+1} = \theta_n - a_n X(\theta_n)$$

converges in probability to θ^* provided that

- $X(\theta)$ is uniformly bounded
- $f(\theta)$ is nondecreasing
- $f(\theta)$ has positive derivative
- $\sum a_n = \infty$
- $\sum a_n^2 < \infty$.

One such sequence is $a_n = a/n$ for $a > 0$.

Later it was shown that convergence occurs with probability 1.

In stochastic optimization we seek

$$\boldsymbol{\theta}^* = \mathbf{argmin}\ \mathbb{E}[J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y})].$$

In our problems, there is no unique root, the problem is not convex, and the cost function may not be differentiable. Nonetheless, the ideas in the Robbins-Munro algorithm carry things forward.

Polyak averaging was first proposed as a way to accelerate the Robbins-Munro solution. Under strong assumptions, it achieves the optimal convergence rate of $\mathcal{O}(n^{-1/2})$.

The Kiefer-Wolfowitz algorithm was developed to improve the Robbins-Munro algorithm, but it does not apply to DNNs.

2.9 Other Practical Comments

Why do we like ReLU? If the function is inactive then its gradient is zero and it is not participating in the network.

In contrast, for, say, the sigmoid function, it only produces a gradient of zero when it is outputting very large (negative or positive) values.

This means the sigmoid function is having a significant impact on the deep NN while also being unable to learn and update its weights.

Training deep NNs is often an art. Each step of the program seems correct, but the network does not train. Here are some tips to help.

When building a deep NN, try simple things first. Don't have too many hidden layers (just one or two), don't use regularization or other slick optimization methods.

Track the loss function. Regularly output intermediate accuracy data, and check that the loss function is decreasing. Plot it to see if it is flat (the vanishing gradient problem).

Often a pretrained DNN enables a warm start. Use a previous DNN that addresses a similar task to the one you are addressing.

If the loss function is decreasing, but decreasing very slowly, change the learning rate to something larger.

If the loss function is not changing, output the predictions from the last layer to see if there are obvious errors (e.g., all cases are assigned to the same class). The problem may be due to inappropriate initialization. You should change the starting weights and biases in the activation functions.

If the loss function still doesn't change, and if your DNN uses functions such as the logarithm or softmax, then the input to those functions may be too large or too small. For example, if the input to the ln function is too small, change $\ln x$ to $\ln x + 10^{-8}$.

Review your choice of activation function and loss function.

With output from convolutional layers, ReLU or leaky ReLU helps to avoid vanishing gradients. But in fully connected layers of CNNs, the sigmoid function can be better.

For classification problems, cross-entropy is generally a good choice for the loss function.

TensorFlow and PyTorch and Keras have default choices that are generally good. But I would like to see a large-scale experimental design that examines these issues statistically.

3. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural networks in which at least one layer's traditional matrix multiplication is replaced by a convolution operation. The convolution of two functions f and g is defined as

$$s(i) = \int_{-\infty}^{\infty} f(m)g(i-m)dm$$

The operation is often notated as $s(i) = (f * g)(i)$

When discussing CNNs, the first argument, f, is generally the input, and the second, g, is the kernel, or filter. The output s is referred to as the **feature map**.

When one has discrete data, the convolution is

$$s(i) = (f * g)(i) = \sum_m f(m)g(i-m)$$

.

In two dimensions, for a two-dimensional input image I and with a two-dimensional kernel K, we have

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$
$$= \sum_m \sum_n I(i-m, j-n)K(m,n)$$

because convolution is commutative.

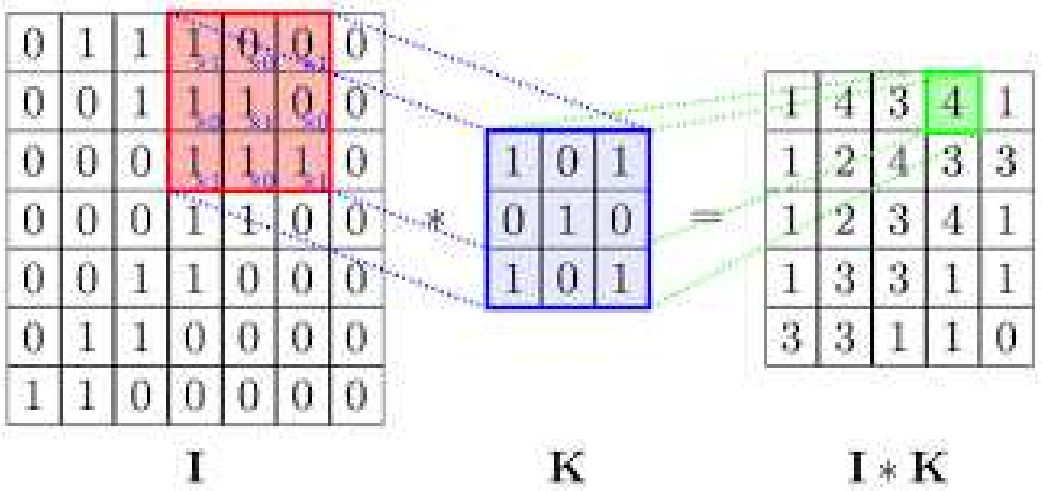


Figure 1: This illustrates the convolution operation on a $7x7$ image with a $3x3$ filter

Credit: Prakesh, Arun. Machine Learning - Convolution for image processing. Francium Tech.

Three of the most important motivations for CNNs are

1. Sparse Interactions
   - Traditional neural networks are completely connected between layers, so every input is ultimately connected to every output (via matrix multiplication and the activation functions). However, for many problems this is computationally wasteful, because often a large proportion of the information and structure is spatially local (Figure 1). So CNNs can be much more efficient compared to networks with dense connections.
   - Deeper units in these networks can still indirectly interact with larger sections of inputs (Figure 2).

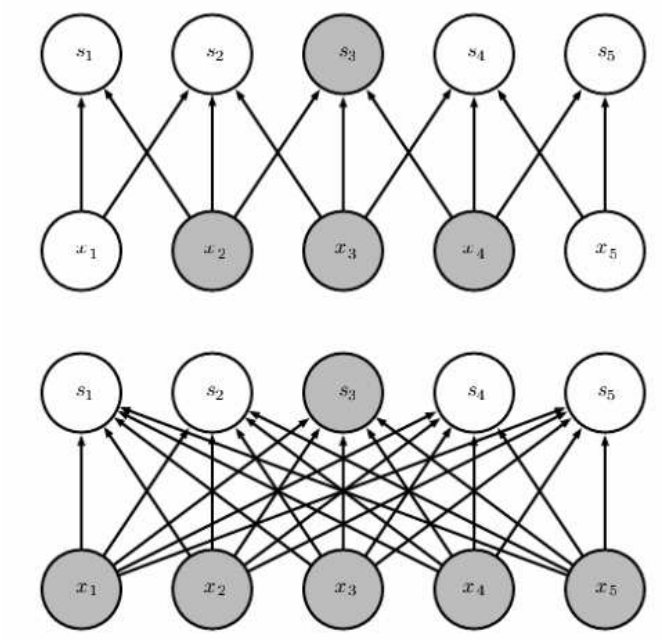


Figure 2: Each node in the top layer is influenced only by local inputs in the top, while the bottom dense network's nodes are influenced by all inputs.

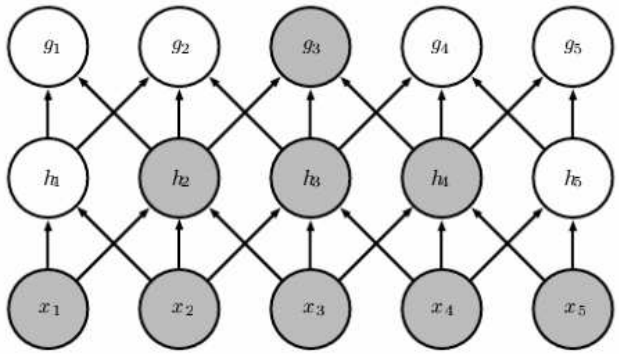


Figure 3: This illustrates how deeper nodes can be influenced by distant information indirectly.

2. Parameter Sharing

   - Parameter sharing refers to using the same parameters multiple times in the function (Figure 3). This is more efficient than dense matrix multiplication both in memory and time costs.

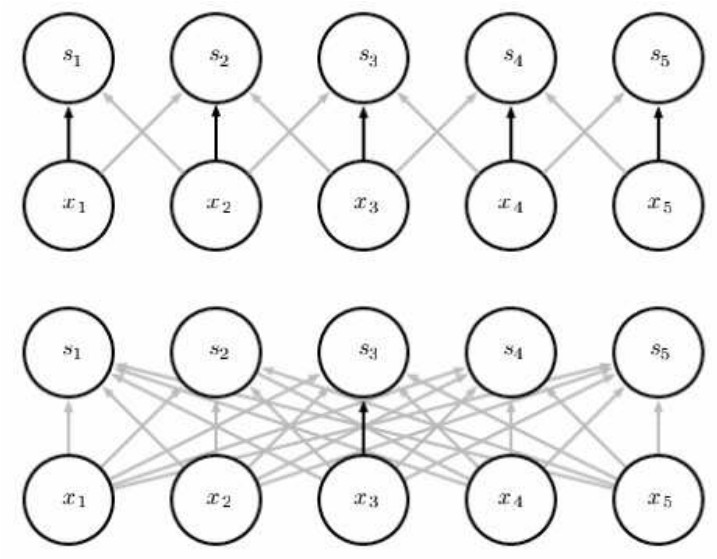


Figure 4: The black line is the same parameter in the top convolutional layer; it is used for every input. The bottom dense layer shows that the parameter is used only once for that specific pixel and output node.

3. Equivariant Representations

- This type of parameter sharing gives a useful property called **translation equivariance**. Formally, f is equivariant to g if $f(g(x)) = g(f(x))$. If g is a function that translates (shifts) then input, then the convolution is equivariant to translations, so the output of a convolution step gives a "map" of the kernel's feature(s) on the input. This is helpful because the convolution is then able to recognize objects independently of where they are in the image. For example, a clock that exists in the left part of an image is still a clock if it appears on the right side of a different image.

3.1 Pooling

Each convolutional layer of a CNN typically is comprised of three stages:

1. A convolutional stage
2. A linear activation stage
3. A pooling stage

The convolutional stage layers do several convolutions to produce a set of linear activations. This can be thought of as a kind of multiple linear regression.

The detector stage runs the linear activations through non-linear activation functions (e.g., ReLU).

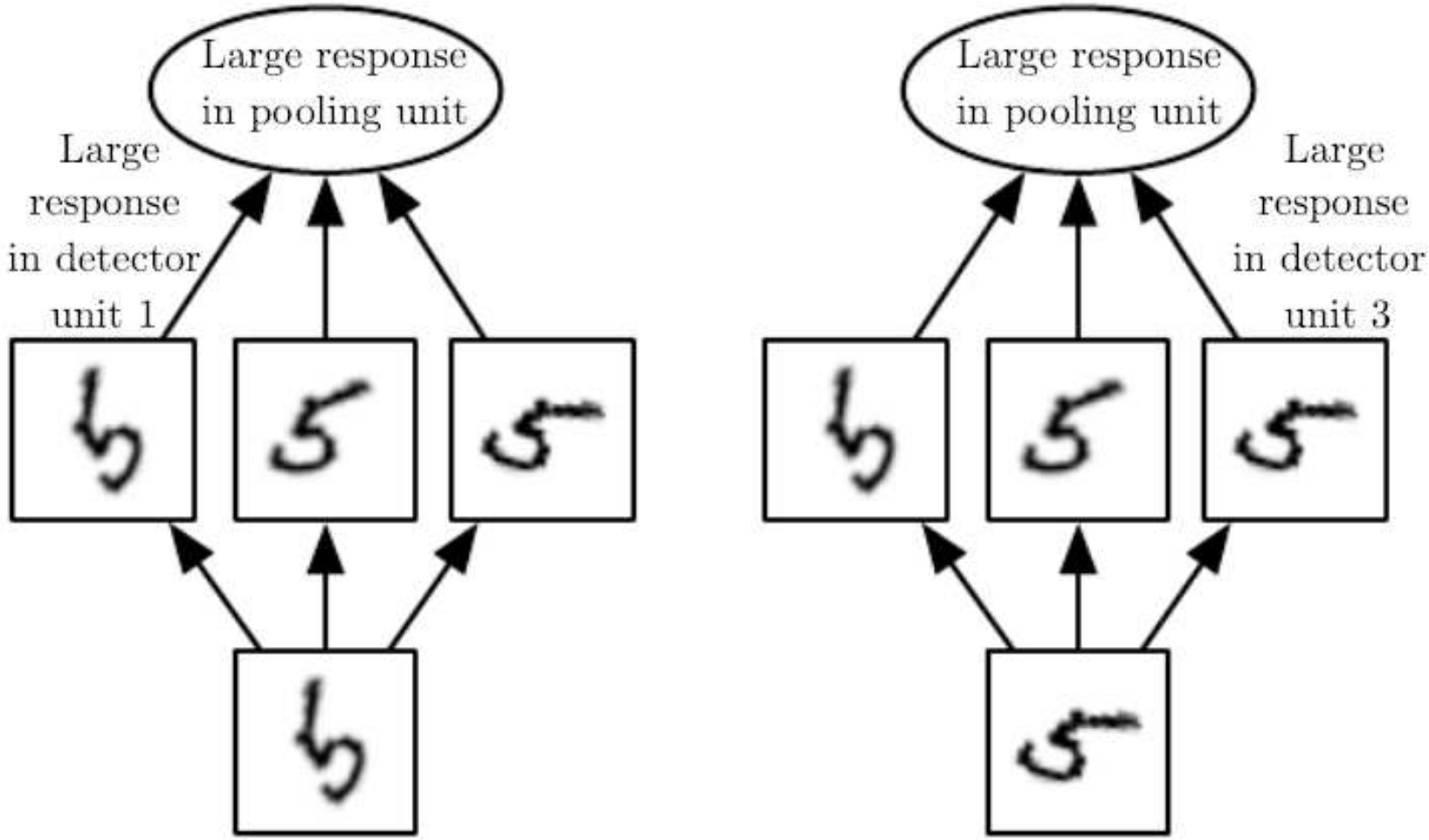
The pooling stage takes the output from each node of the current layer and replaces it with a summary statistic describing the area around that node. For the case of image analysis, this "area" may be comprised of the 8 pixels surrounding the current pixel.

Max pooling replaces the output of the network at a location with the maximum output within a rectangular neighborhood. Other pooling methods replace it with the average over that neighborhood, or an L2 norm, or a weighted average based on distance within the neighborhood.

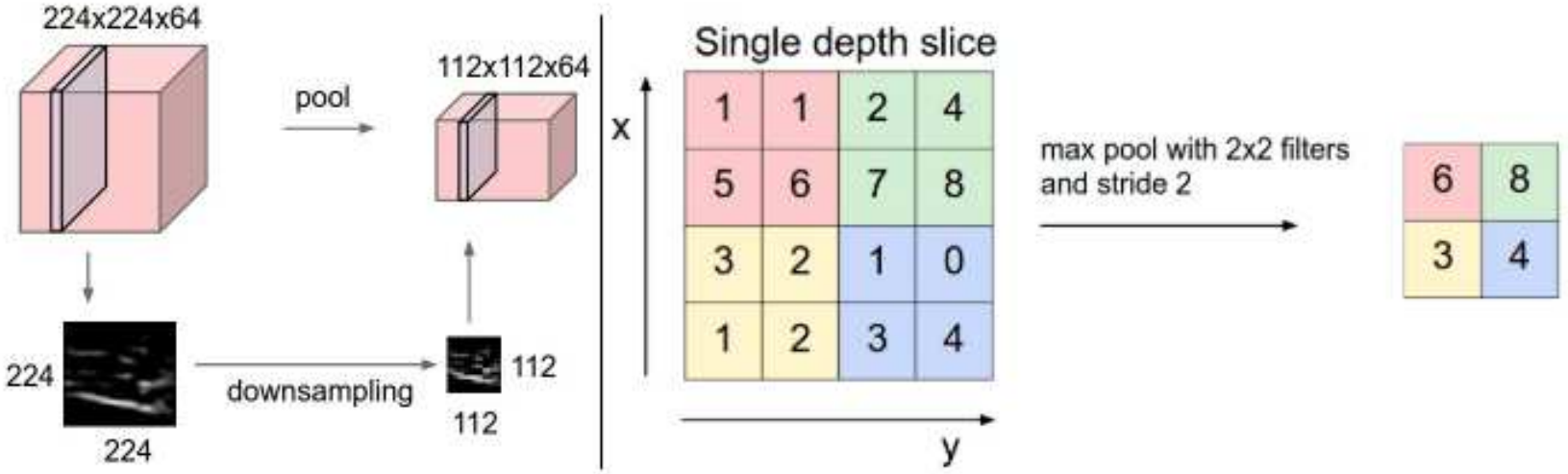
CNNs use pooling to force models to be invariant to small translations of the input. This is important when one is more concerned about whether a cat is in the image than one is about the exact location of the cat.

Pooling is also used to decrease the width of one layer to the next by compressing the information from several nodes into one.

If one pools over the outputs of separately parameterized convolutions, the features can learn the transformations to which the CNN should be approximately invariant.



Credit: Goodfellow, Ian, et al. Deep Learning. MIT Press (2017), p338 figure 9.9



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).

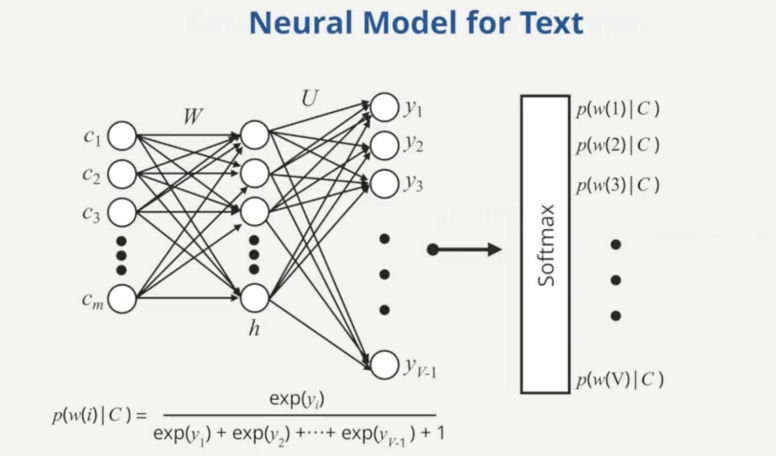
Credit: Pooling Example. CS231n Convolutional Neural Networks for Visual Recognition, cs231n.github.io/convolutional-networks/.

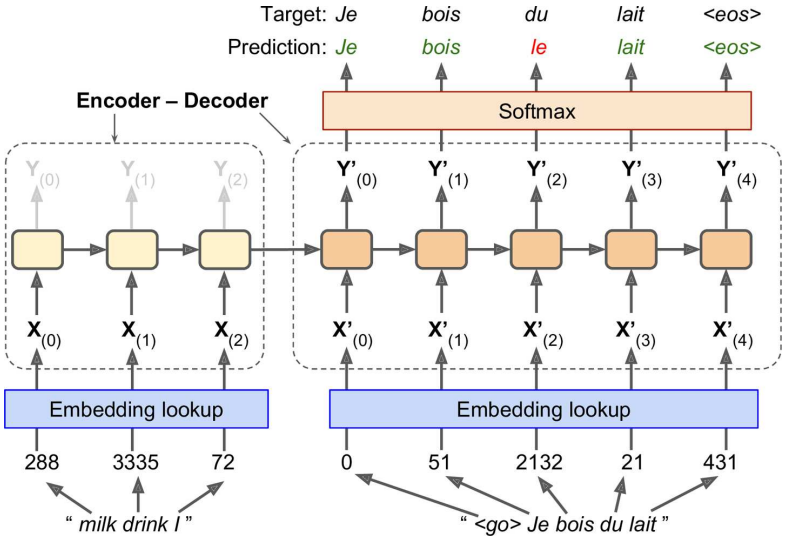
3.2 Variants of the Convolution Function

Often the convolution operation used in CNN is different from mathematical convolution. Mathematical convolution requires the kernel to be flipped along the sub-diagonal before being applied to element-wise multiplication. But in most CNNs, the kernel is not flipped, so it is actually another operation called cross-correlation. This operation has fewer pleasing properties, but it turns out to be good in implementation.

Other variations of the convolution function include "convolution with stride," different versions of "zero padding convolution," "locally connected convolution," and "tiled convolution," which we shall describe.

We may want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting features as finely). One can think of this as downsampling the output of the full convolution function. This is called **convolution with stride**. For example, the full convolution has stride=1; but if stride=100, the kernel applies to the first position, the 101th position, and so on.



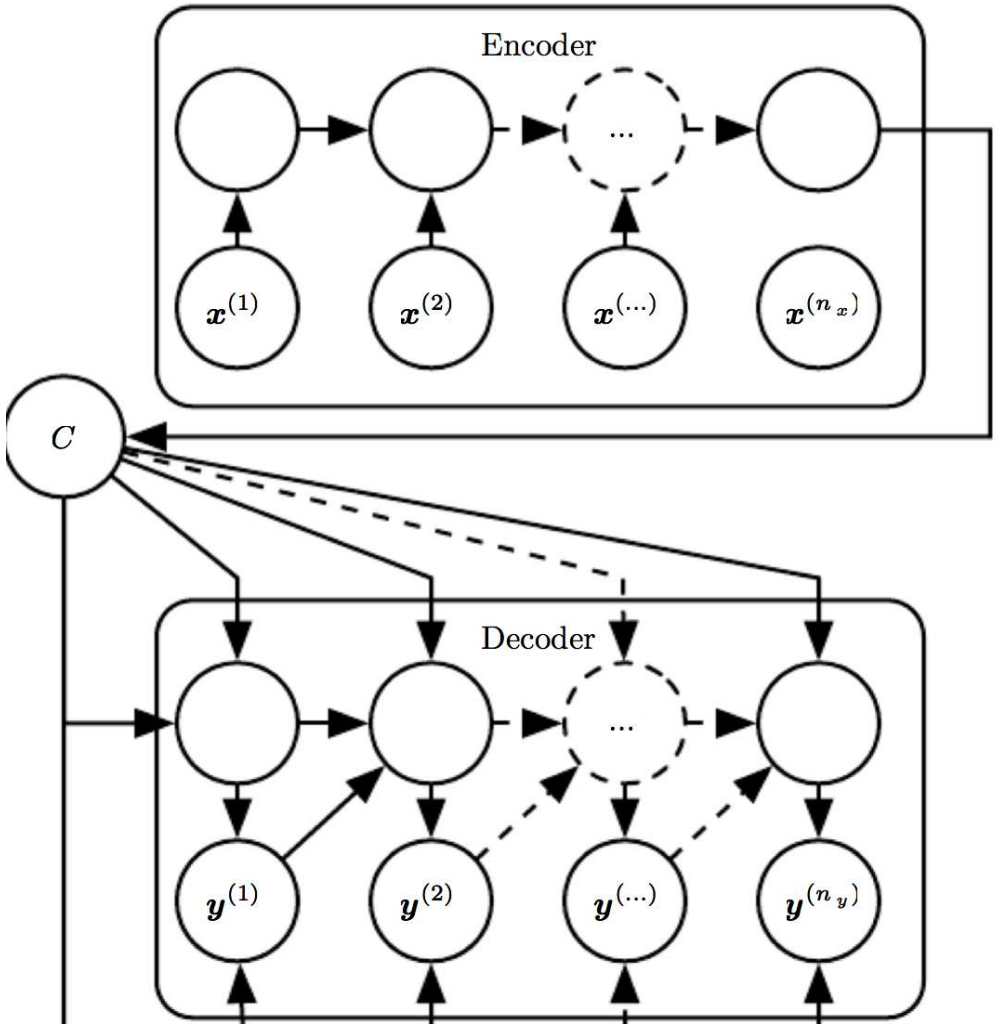


Left figure depicts stride=2, right figure depicts full convolution.

Convolution results in output representations that are smaller than the inputs by one less than the width of the kernel at each layer.

Also, input data near the edge are not receiving the same attention as their counterparts in the middle. One pads with zeros at the sides of the input image. The figure shows a form of zero-padding which ensures the input size does not shrink. The black dots are the zero paddings.

Using zeroes (as opposed to other constants) to pad avoids introduction of spurious signal. But I advise duplicating the boundary values as introducing signal more like that seen at the edges of the image.



Credit: Goodfellow, Ian, et al. Deep Learning. MIT Press (2017), p351 figure 9.13

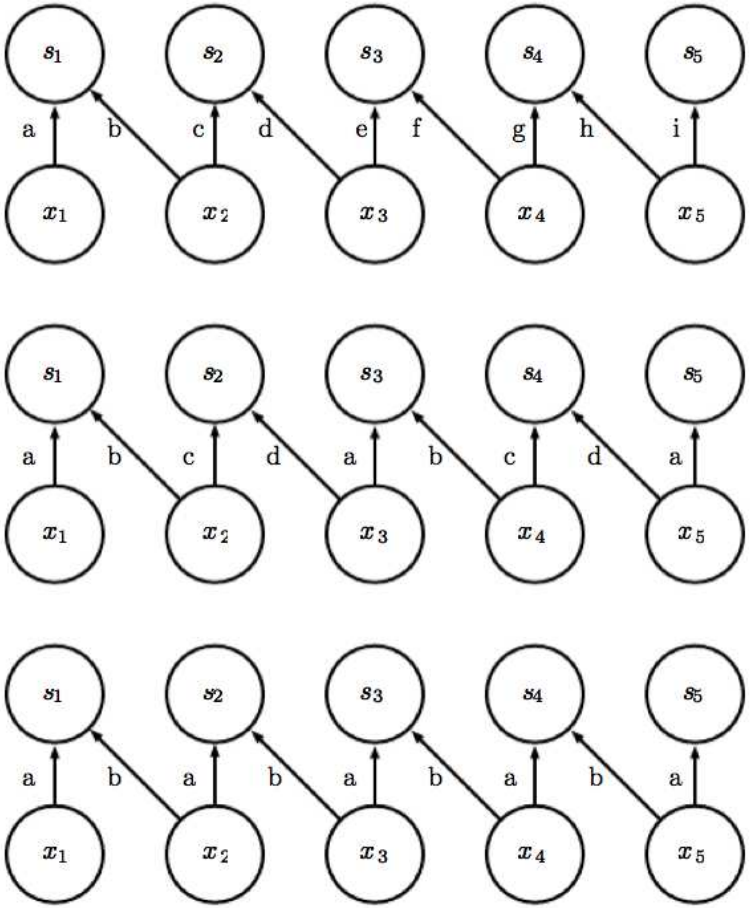
A **locally connected layer** is sometimes referred to as "unshared convolution." The idea is to focus on CNN's method of extracting local features, but instead of doing parameter sharing (e.g., a 3×3 kernel matrix convolved through a 100×100 input matrix), it will use a different set of parameters for convolution at different positions (e.g., instead of one 3×3 kernel, there are 98^2 3×3 kernels convolving at each location).

Locally connected layers are useful when each feature is a function of a small part of space, but the same features do not occur across all of space. For example, if we want to tell if an image is a picture of a face, one only needs to look for the mouth in the bottom half of the image.

With **tiled convolution** one compromises between CNNs and a regular feedforward DNN. Instead of learning a separate set of weights at every location in the image, one learns a set of kernels that one rotates through as one moves through the image.

Neighboring pixels will have different weights, as in NNs, but the memory requirements are much reduced, and are determined by the number of kernels one learns.

Tiling is one of the many cases in which tensor representations are useful in training DNNs. Each kernel is a matrix, and they may be viewed as slices in a three-dimensional tensor array.

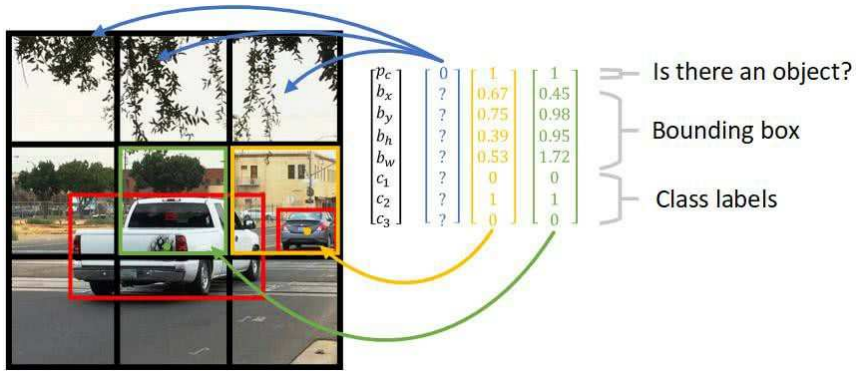


(Top) Locally Connected Layer, (Middle) Tiled Convolution, (Bottom) Full Convolution

In general, the most expensive part of CNN training is learning the features. The output layer is relatively inexpensive since only the small number of features that have passed through the layers of pooling are provided as input. While performing supervised training on the entire network, each gradient step requires a complete run of forward and backward propagation. There are ways overcome this by obtaining convolution kernels without supervised training.

- Kernels may be initialized randomly.
- Kernels may be designed by hand.
- Kernels may be learned with an unsupervised criterion.

CNNs are capable of not only outputting classifications; e.g., there is a cat or no cat. But also can make **structured outputs**. For example, they can output bounding boxes around objects and also classify each object separately. This is done by labelling each pixel and then iteratively using each prediction made from a previous pixel to predict and label further pixels. This strategy follows that used by Recurrent Neural Networks.



3.3 CNN Applications

CNNs can be used for a wide variety of different data types, not just images. For example:

- Audio waveforms, convolving over time. Also audio waveforms that have been Fourier transformed into a 2D tensor with time and the different frequencies. This makes the model equivariant to both time and frequency octaves.
- Text data convolving over a 1D sentence or a 2D alignment of many similar sentences. This is used in computational biology with multiple sequence alignments.
- Volumetric data such as CT scans or videos where the z-axis corresponds to different frames over time.
- fMRI data, where there are time and voxels.

Even in cases where computational cost and overfitting are not major issues, one advantage to a CNN is that it can process differently-sized inputs. If the input data is a different size, it can dynamically change the stride and size of its convolution and pooling layers to adapt in such a way that the same sized output is produced.

Alternatively, in the case of, say, producing bounding boxes, the original image dimensions can remain the same.

Note Bene: This only works when the change of input size is because it is more or less data of the same form. This does not work if the change in length, e.g., has extra dimensions in an input vector, corresponding to very different features or data types.

3.4 Other CNN Comments

According to Goodfellow et al. (2016), "Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing pointwise multiplication, and converting back to the time domain using an inverse Fourier transform."

When a kernel is separable, the use of the Fourier transform as described above is more efficient than performing the discrete convolutional operations. The runtime can range from $\mathcal{O}(w^d)$ to $\mathcal{O}(wd)$ where d is the dimension of the input and w is the width of the convolution kernel.

CNNs are perhaps the greatest success story of biologically inspired artificial intelligence. Though not perfectly analogous, many key features of CNNs have direct neuroscientific counterparts.

- The primary visual cortex is arranged in a spatial map with a two-dimensional structure that mirrors the structure of the image in the retina. Convolutional neural networks mimic this, as their features are also defined in terms of two-dimensional maps.
- Olshausen and Field (1996) showed that simple unsupervised learning algorithms have receptive fields similar to simple cells in the primary visual cortex. Like CNN detection units, these simple cells detect basic features in a small, spatially localized area of the image.
- The pooling units of CNNs were inspired by the complex cells that make up the rest of the primary visual cortex. These cells seem to be invariant to small shifts in the position and lighting of the feature.

Nevertheless, back-propagation is biologically impossible and the function of the brain remains more complex than that of a CNN. The relationship between CNNs and human neurobiology is important only insofar as it provided early inspiration for deep learning models.

One could make a calculation to determine how much time it would take evolution to achieve back-propagation through natural selection. It is much longer than the age of the Earth.

Successful convolutional neural networks, drawing from understood biological principles, paved the way for the acceptance of "neural networks" as a whole.

4.0 Recurrent Neural Networks

Recurrent Neural Networks (RNN) have been used in tasks that involve sequential data, such as audio data, signal data, and text data. This contrasts with CNNs, which are tuned to images and arrays. Tasks with sequential data can have new objectives than and that lead to different network structure. Andrej Karpathy (2014) proposes a taxonomy of RNN tasks:

- One-to-Many: sequential output, for image captioning.
- Many-to-One: sequential input, for sentiment classification.
- Many-to-Many: sequential input and output, for machine translation.

RNNs are trained with observed sequences of data. One usually uses minibatches of such sequences, and these sequences usually are unequal in length.

One of the most prominent uses of RNN is in natural language processing, such as text synthesis (e.g., translating Chinese to English, machine-generated Shakespearean sonnets). Therefore we present RNN within the context of **Natural Language Processing** (NLP) tasks.

In order to transform text data into mathematically manageable data, the idea of word embedding is used. If one can access to a large corpus of text data, such as all the Shakesperean works, then one can form a one-to-one map all words that appear in the text with different vectors.

One goal of this map is that the vector representation of nearby words in a sentence should be predictive of each other using some model. Another goal is that similar words, such as "cat" and "kitten," should have similar vector embeddings. And if one subtracts the vector for "queen" from the vector for "monarch" then we get a vector near the vector for "man."

Word2vec is a set of linguistic models that produce word embeddings. These models are two-hidden-layer NNs that are trained to provide contexts of words. Word2vec takes a large corpus of text and finds a vector space (of several hundred dimensions) such that each word in the corpus has a corresponding vector in the space. Word vectors are located so that words with common contexts in the corpus are close in the space.

Word2vec seems to do better than previous text models, such as latent semantic indexing and latent Dirichlet allocation.

Its idea is akin to multidimensional scaling.

Word2vec uses one of two architectures to represent words: continuous bag-of-words (CBOW) or continuous skip-gram. In the CBOW model, one predicts the current word from a window of surrounding context words. The order of context words does not influence prediction (i.e., the bag-of-words assumption).

The continuous skip-gram model uses the middle word to predict the surrounding window of context words. Skip-gram architecture weights context words that are near more heavily than distant words. CBOW is faster, but skip-gram does a better job with rare words.

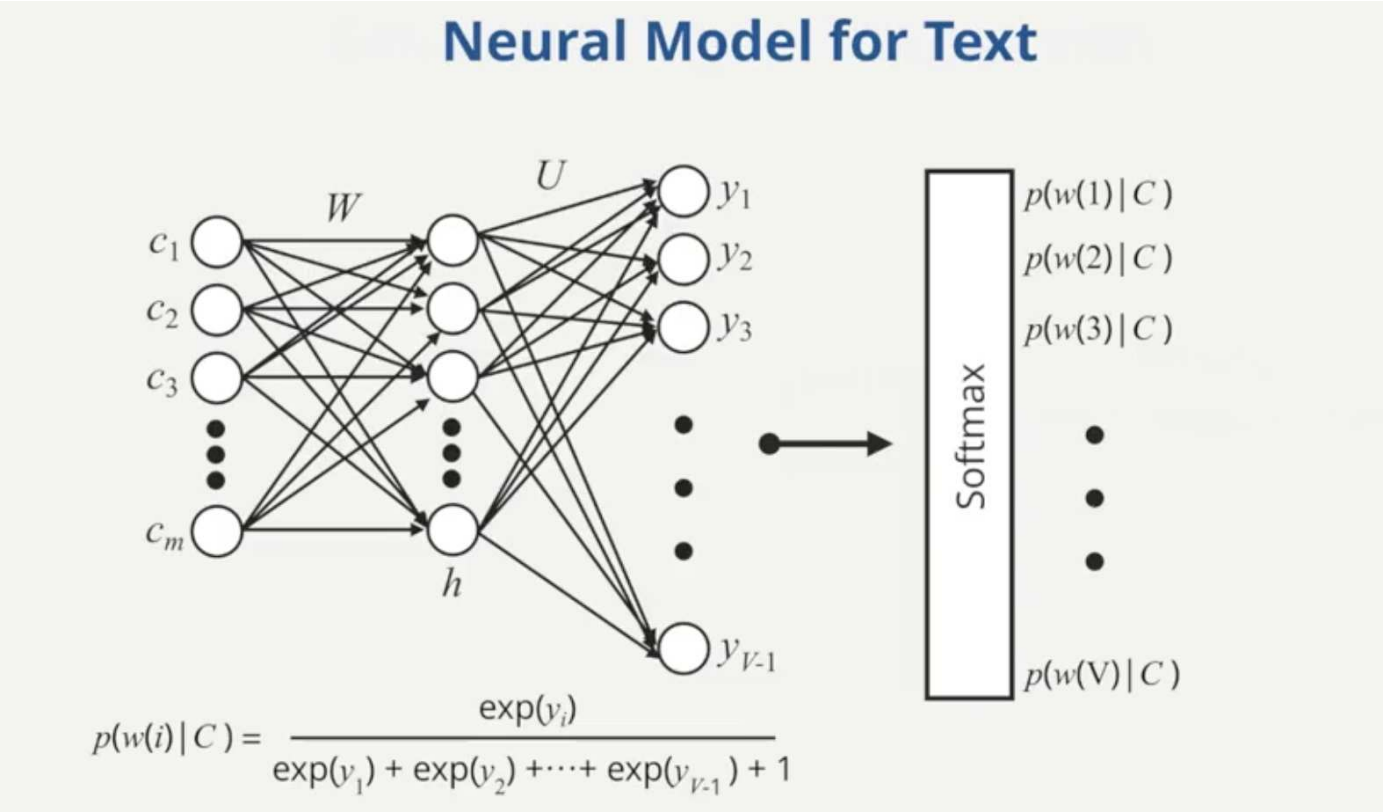
For CBOW, the usual width of the context window is 5. For skip-gram, it is 10.

Word2vec can be trained with softmax. To approximate the conditional log-likelihood that must be maximized, the hierarchical softmax method uses a Huffman tree to reduce calculation.

Alternatively, the negative sampling method maximizes the conditional likelihood by minimizing the log-likelihood of sampled negative cases.

The folk wisdom is that hierarchical softmax works better for infrequent words while negative sampling works better for frequent words and performas better with lower dimensional vector embeddings.

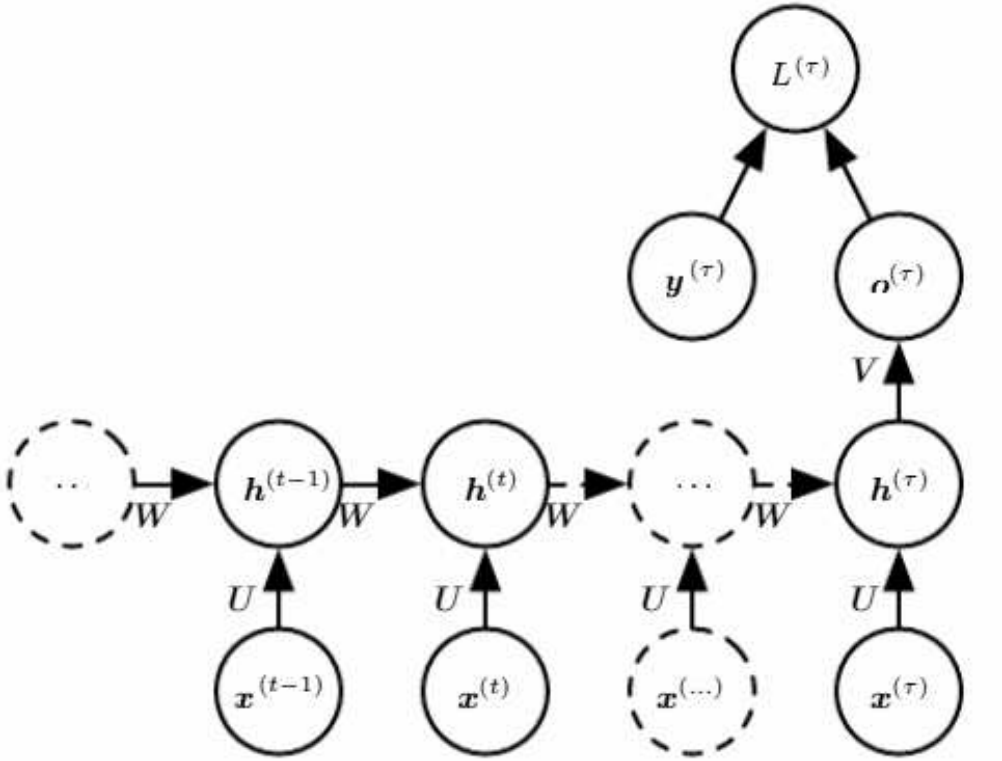
One seeks a model (typically an NLP model) such that when fed the average of vectors of the i^{th},$(i+1)^{th}$,$(i+3)^{th}$,$(i+4)^{th}$ words, the model can predict the $(i+2)^{th}$ word with high accuracy.



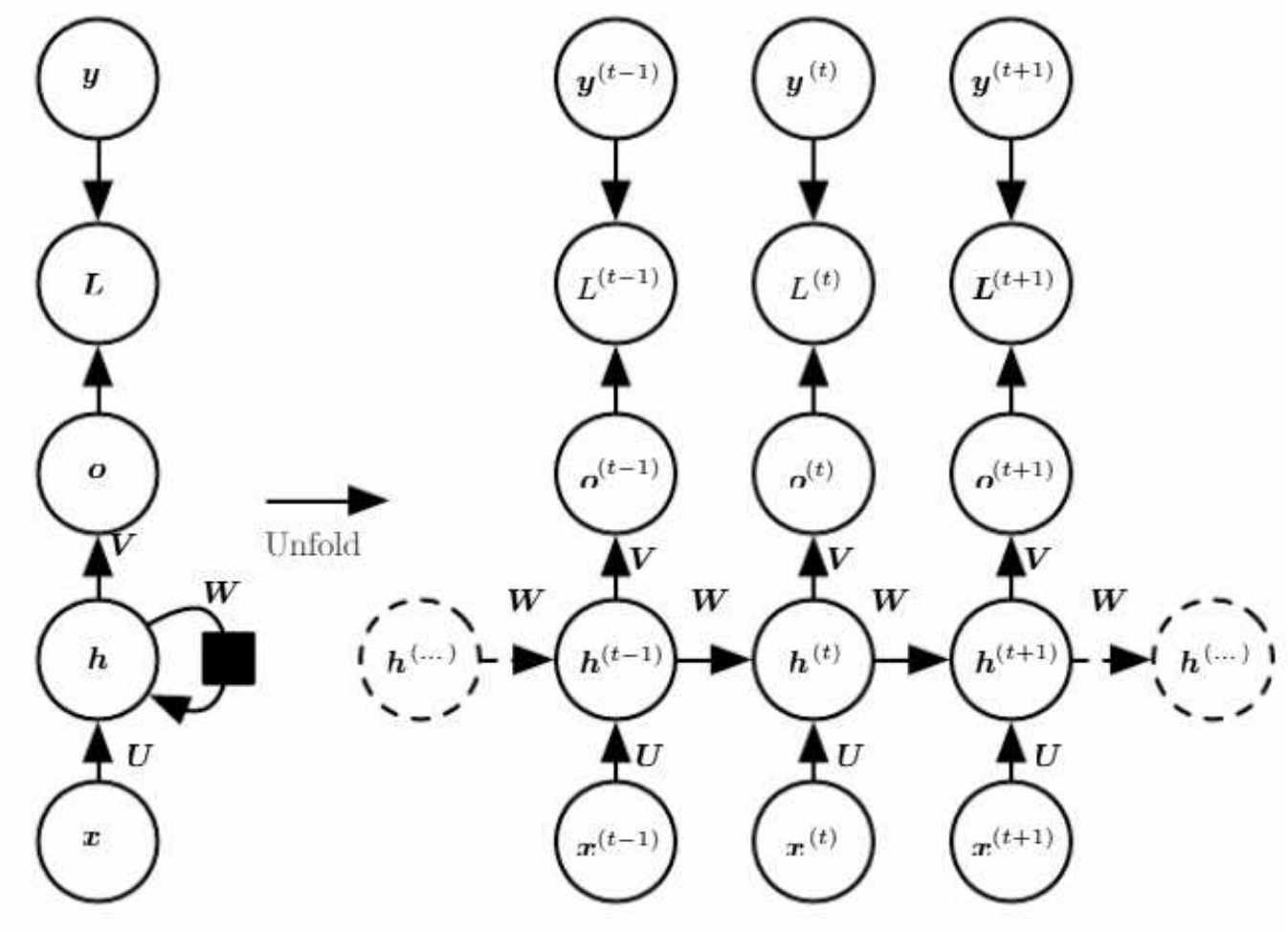
Credit: Duke Coursera "Introduction to Machine Learning"

4.1 Architectures

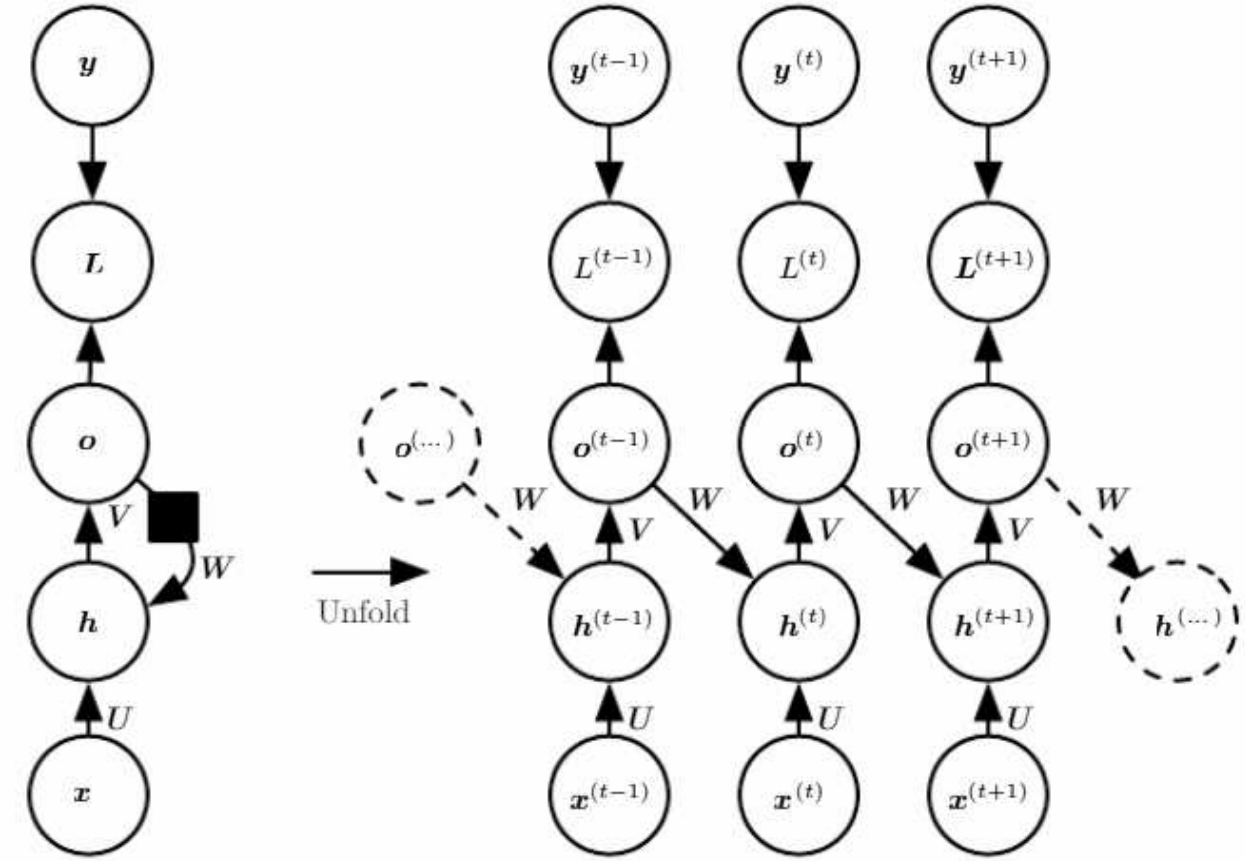
We start with three basic RNN architectures. In the first, we have a single output node, with connections between hidden units through time, as shown below. CNNs generally share parameters over time.



We could also have outputs at every time step.

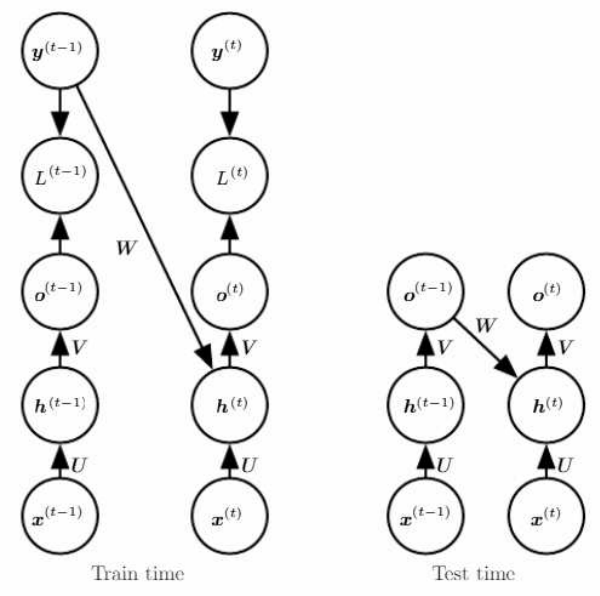


What about training in parallel?



But this still relies on previous sequence information...

Teacher Forcing trains by using the ground truth output y_t as input for time $t + 1$.



However, the ground truth and model outputs might be very different, so the model's testing performance might be low. Bengio et al. (2015) suggest during training to randomly use ground truth and model outputs, with a "curriculum learning" process.

Teacher forcing is a training method useful for RNNs that have connections from the outputs to their hidden layers at the next time step.

During training, one feeds the correct output y_t from the training sample as input to the hidden activation function h_{t+1}. But when the model is deployed, often (not always; e.g., weather forecasting) the previous truth is not known. In that case, one uses the prediction for time t as input to h_{t+1}.

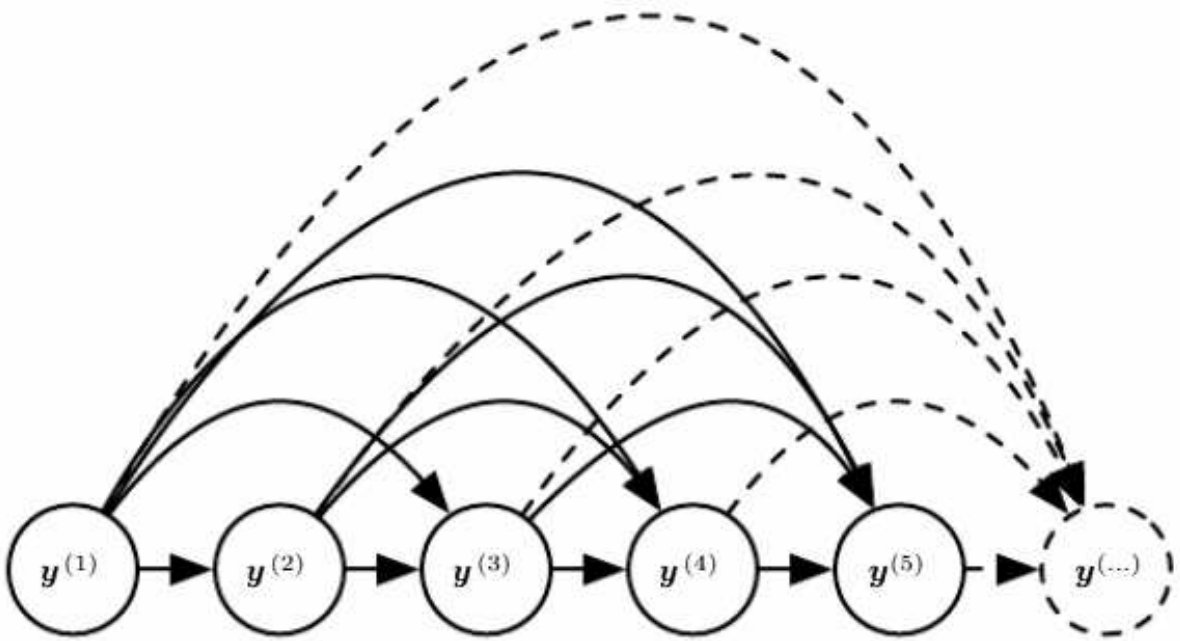
In RNNs, the output from from the present time can influence its value in the future. This corresponds to cycles in the computational graph.

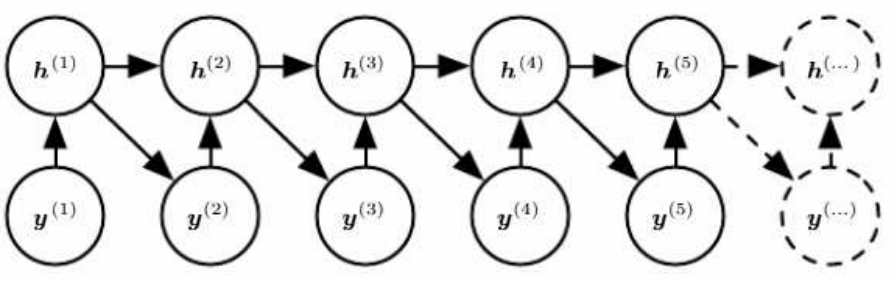
One can represent the model as a directed acyclic graph, and then apply **backpropagation through time** (BPTT) through the graph. The only difference is that one is sharing parameters through time. However, if one has very long sequences, the gradients may either explode or go to zero. Even if they don't, it is a very expensive backward pass on every training step.

Truncated BPTT (Sutskever PhD Thesis, 2013) addresses this by running BPTT every k_1 timesteps for k_2 timesteps backwards. Pseudo-code:

1: **for** t **from** 1 **to** T **do**
2: Run the RNN for one step, computing h_t and z_t
3: **if** t **divides** k_1 **then**
4: Run BPTT (as described in sec. 2.5), from t down to $t - k_2$
5: **end if**
6: **end for**

When thinking about previous ground truth values' affects on the future, we can describe a model such as:

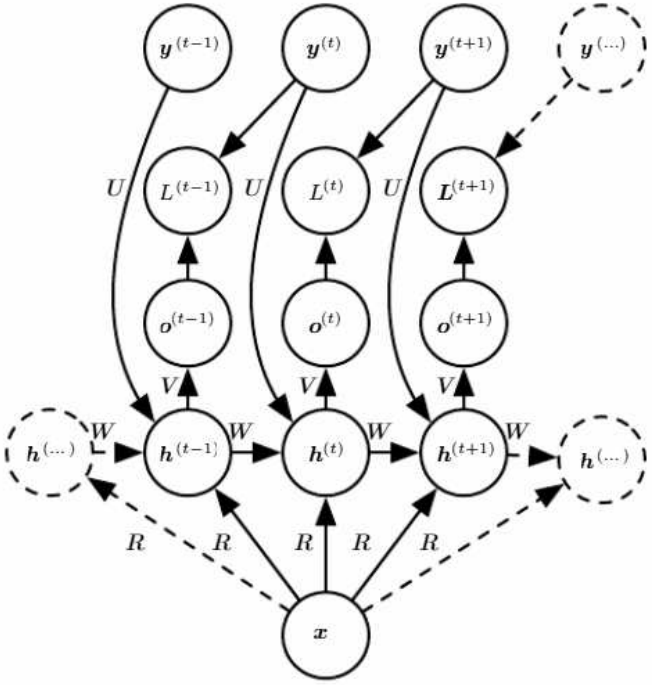




RNNs can be thought of as efficient parameterizations of the joint distribution over the observations. This independence of observations and sequence length is computationally efficient.

A Markov model cannot capture distant dependence, but an RNN can through the state variables $\boldsymbol{h}$. However, this parameter sharing assumes that the conditional probability distribution over the variables at time $t+1$ given time t is stationary. To get our output, we simply sample from the conditional distribution at each step. We include a "stop" token so the sequence can end.

If we have a fixed size input that we want to use, we can feed it into the model at each timestep. This would arise, e.g., in image captioning, where the same image $\boldsymbol{x}$ is input to generate a string of text. The $L^{(t)}$ terms are the loss function values.



In many applications we want to output a prediction of $y^{(t)}$ which may depend on not only the past but also the whole input sequence. One such task is handwriting recognition, where the flow of cursively written letters depends on its adjacent letters from both sides (from the past and future).

To address such need, bidirectional RNN was invented, and has been successful when the need to look both ahead and backwards arises. Bidirectional RNNs combine an RNN that moves forward through time beginning from the start of the sequence with another RNN that moves backward through time beginning from the end of the sequence.

This idea can be naturally extended to 2-dimensional input, such as images, by having four RNNs, each one going in one of the four directions: up, down, left, right.

Recall that in the vanilla implementation of RNN, if one wants to map an input of sequential data to an output of sequential data, the lengths of input and output data have to be the same. This is a problem for tasks such as machine translation. Therefore, the **encoder-decoder unit** was invented.

The encoder RNN can be a simple network with a self-loop on the hidden state. It produces the context vector $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$.

The decoder RNN is conditioned on the context to generate an output sequence $(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)})$.

This makes sure that the final state of the encoder RNN contains information about the entire past. This final state (after simple transformation) is the fixed-length context vector that feeds into each time step of the decoder RNN.

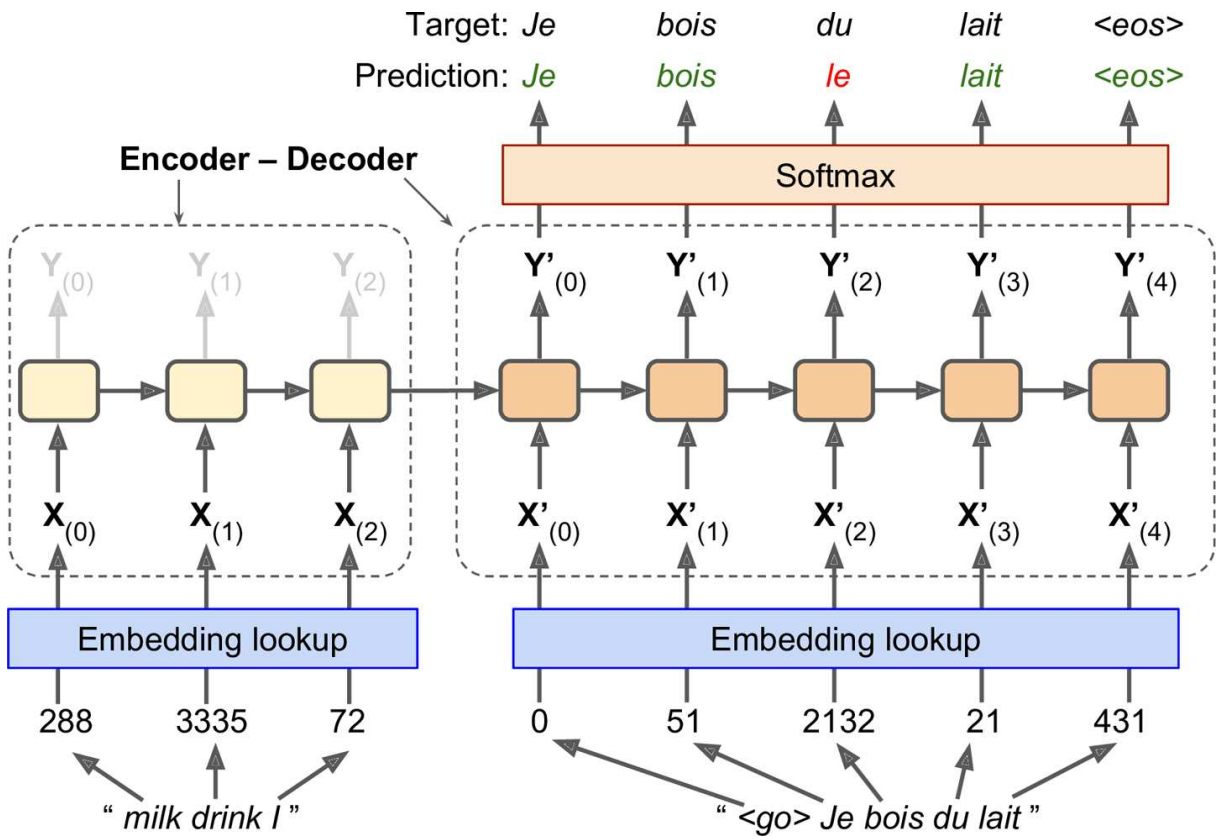
Both RNNs are trained jointly by minimizing the objective function

$$-\sum_i \ln(P(y_i^{(1)}, \ldots, y_i^{(t)}, \ldots y_i^{(n_y)} | x_i^{(1)}, \ldots, x_i^{(t)}, \ldots x_i^{(n_x)}))$$

Note that n_y and n_x need not be the same.

When the context is a vector, then the decoder RNN is just a vector-to-sequence RNN. The input can be added initially, or at each of the hidden layers, or some combination.

The two RNNs need not have the same architecture, and the encoder is usually simpler than the decoder.



The English sentence is reversed since "I" is the first thing the encoder predicts. The target French sentence is fed into the decoder during training, The final state of the encoder is input for all hidden states of the decoder.

4.3 Long-Term Dependencies

Any DNN may have exploding (rare) or vanishing (common) gradients. This problem is both numerical, in that very large or small values can result in buffer over/underflow, but also intrinsic to calculating the gradient between long term connections from the first layer to the loss function at the end.

In a normal feedforward network there are more opportunities for the layers in between to adapt and prevent over- or underflow. But in a RNN, the distant past generally gets downweighted compared to the recent past, and many successive downweightings forces the gradient to zero.

To put this mathematically, where we make the simplifying assumption of a linear activation function:

$$\boldsymbol{h}_t = \boldsymbol{W}^\top \boldsymbol{h}_{t-1}$$

which can be rewritten as

$$\boldsymbol{h}_t = (\boldsymbol{W}^\top)^t \boldsymbol{h}_0.$$

If $\boldsymbol{W}$ has an eigendecomposition of the form $\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$ for orthogonal $\boldsymbol{Q}$ and diagonal $\boldsymbol{\Lambda}$, then

$$\boldsymbol{h}_t = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}_0.$$

The eigenvalues are raised to the tth power. Those larger than 1 go to infinity, and those less than 1 go to zero. Any component of $\boldsymbol{h}_0$ that does not align with the largest eigenvalue gets discarded.

Bengio et al. (1994) found that the probability of successfully training an RNN via stocastic gradient descent fell to 0 for sequences of only length 10 or 20. One of the reasons that the hidden layers of an RNN often use the tanh activation function is to help avoid the gradient problem.

If the weights have eigenvalues near 1 so that they are robust to perturbations in the data input, this means that they lose the ability to capture long-term dependence and can only propagate short-term relationships.

Learning long-term dependence is one of the main challenges in DL. Some attempts to better capture long term dependencies are:

- Skip Connections
- Leaky Units
- Removing Connections

Skip connections have a time delay of d between connections, so the gradients now diminish/explode exponentially as a function of τ/d rather than τ, the length of the input sequence.

Leaky Units have a self-connection at each hidden layer which acts similarly to an exponential moving average calculation. The hidden layer activation is calculated and then combined with the previous activation value parameterized by α so that:

$$\boldsymbol{h}_t = \alpha \boldsymbol{h}_{t-1} + (1 - \alpha)\boldsymbol{h}_t.$$

A final solution is to in fact remove connections between $\boldsymbol{h}_{t-1}$ and $\boldsymbol{h}_t$ and only have longer-range connections. This forces the network to only learn longer-term connections but this is at the cost of learning short term connections.

There are lots of ad hoc approaches. Time series thinking seems relevant; e.g., it is well-known how to handle seasonal effects.

4.4 LSTMs and Other Gated RNNs

Long Short-Term Memory (LSTM) and **Gated RNNs** are strategies for avoiding vanishing or exploding gradients by forgetting the past. They are now industry standards, and are used in speech recognition (Siri and Goodle Home), handwriting identification, and Google Translate.

One of the greatest strengths of RNNs is the ability to "remember" information as a sequence of inputs is processed. One of the greatest challenges of using RNNs is enabling the gradient to flow backward through time during training. A class of RNNs which are particularly good at accumulating information over time and creating paths for the gradient to flow backward through time are gated RNNs.

Gated RNNs are engineered with mechanisms that store and forget information over time and which can learn when to perform either of these actions. Gated RNNs are by far the most effective and widely applied deep learning model applied to sequential data in practical applications.

LSTM and gated RNNs use self-loops to create paths where the gradient can flow for long durations. But one does not want the duration to be fixed.

Gating controls the weight on a self-loop by another hidden unit, so that the time scale can change dynamically. This makes the weight context dependent.

LSTMs are the most popular of all gated RNNs. LSTM recurrent networks have "cells" whose self-loop is comprised of nodes which are updated in a context (current and recent input) dependent way, enabling the network to control how long it remembers input history. LSTMs actually learn four separate neural network gates (or channels) which are responsible for different aspects of the learning process:

- the input (g)
- the input gate (i),
- the forget gate (f),
- the output gate (o).

These are commonly abbreviated as "IFOG".

Each of the input gate, forget gate, and output gate takes in the current input ($\boldsymbol{x}_t$) as well as the output of the previous step ($\boldsymbol{h}_{t-1}$). These vectors are concatenated and then fed into each of the networks. Each network contains its own set of weights and intercepts which must be learned separately. The "IFO" of IFOG uses sigmoid activation functions while the input "G" of IFOG uses a tanh activation function.

The input gate is used to control the how much influence the current input ($\boldsymbol{x}_t, \boldsymbol{h}_{t-1}$) has on the output and the value of the state unit s_t, while the forget channel controls how much information in s_{t-1} one ought to forget given the current input. Finally, the output channel controls how much information one would like this step of the network to produce.

$$z_t = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} \longrightarrow \begin{array}{c} i_t = \sigma\big(W_i z_t + b_i\big) \\ f_t = \sigma\big(W_f z_t + b_f\big) \\ o_t = \sigma\big(W_o z_t + b_o\big) \\ g_t = tanh\big(W_g z_t + b_g\big) \end{array} \longrightarrow \begin{array}{c} s_t = f_t \odot s_{t-1} + i_t \odot g_t \\ h_t = tanh(s_t) \odot o_t \end{array}$$

$$...h_{t-1} \rightarrow \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} \rightarrow \begin{array}{c} LSTM \\ s_{t-1} \uparrow\downarrow s_t \end{array} \rightarrow h_t \rightarrow \begin{bmatrix} \boldsymbol{x}_{t+1} \\ \boldsymbol{h}_t \end{bmatrix} \rightarrow \begin{array}{c} LSTM \\ s_t \uparrow\downarrow s_{t+1} \end{array} \rightarrow \boldsymbol{h}_{t+1} \rightarrow$$

$...$

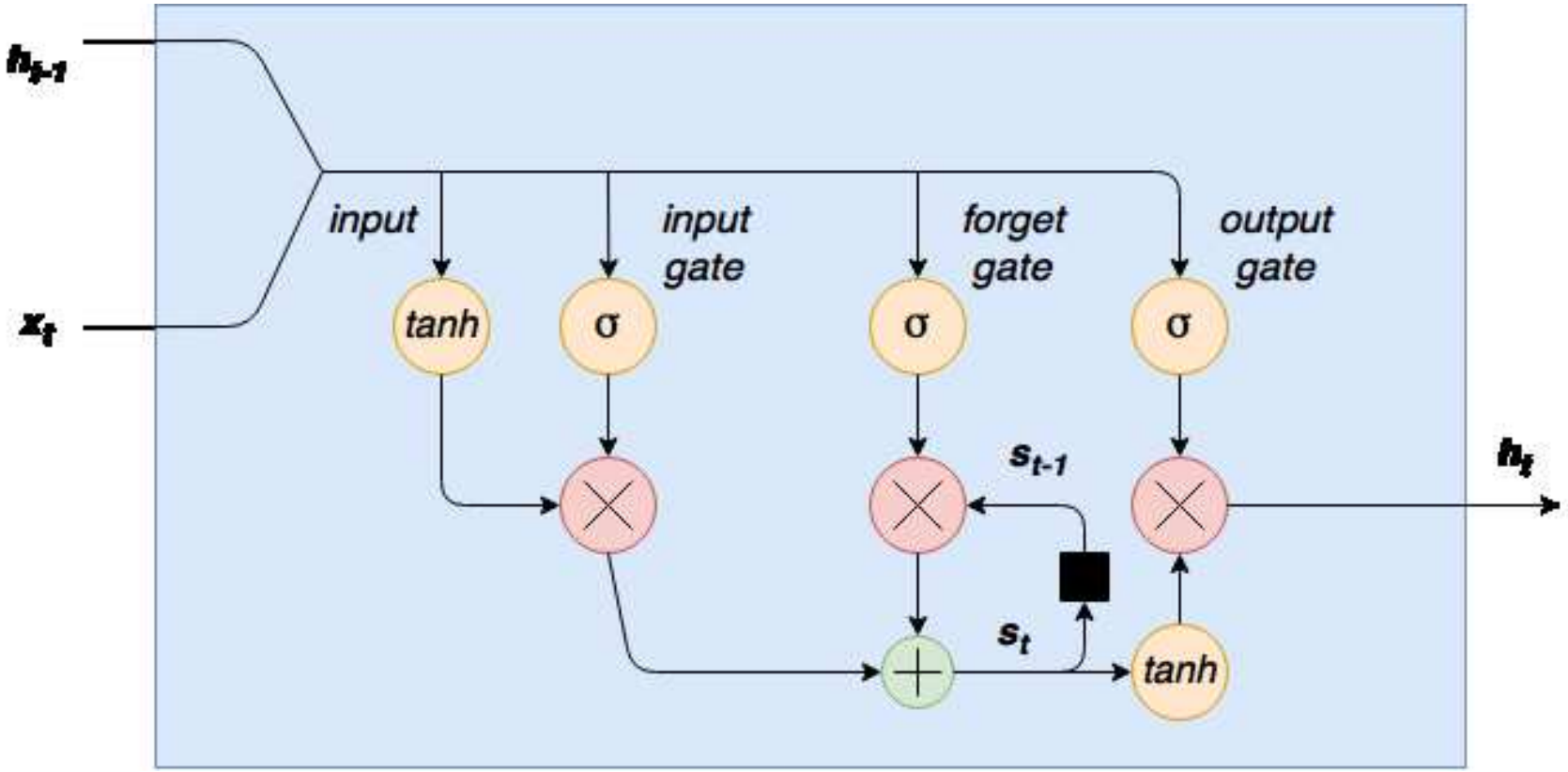


Figure 5: LSTM Diagram

4.5 Variants of LSTM

The **peephole** LSTM unit is an LSTM architecture that allows the gates to access a **Constant Error Carousel** (CEC). The CEC puts allows the IFOG cell to include an input that is unchanged.

This means that the history is carried forward without compression (i.e., no weights, no non-linearities) and the only scaling is from the outputs of the IFOG units. This is a good way of letting the network decide whether it has seen a specific patttern before.

output
output gate
CEC
forget gate
1.0
input gate
input

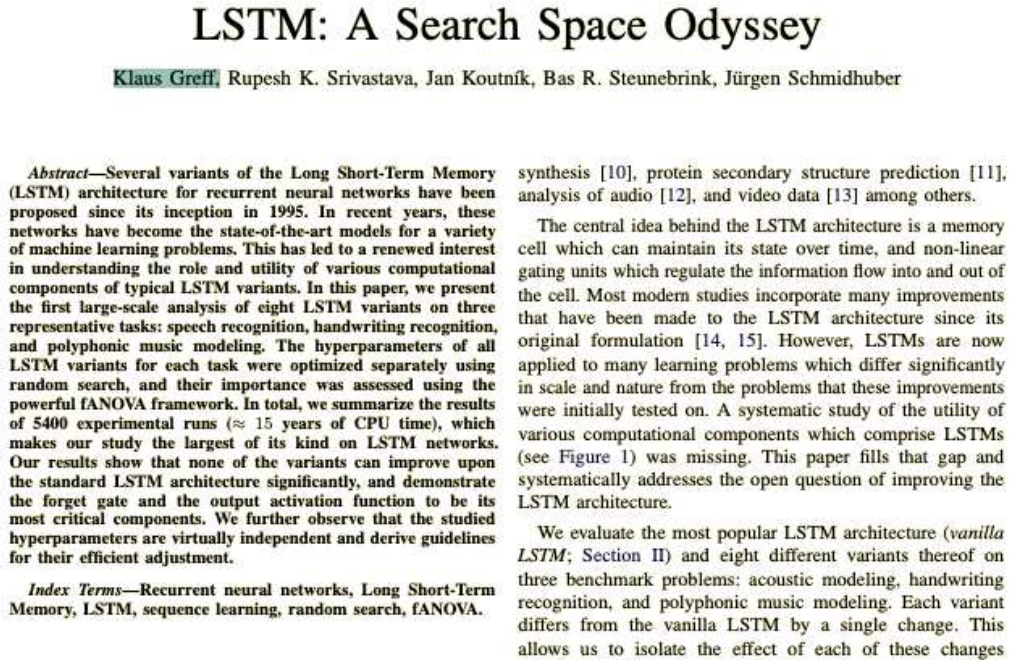
LSTMs have been criticized for being too redundant. Some prefer to use a single gate to control the amount of "forgetting" and "updating" of the state unit. One common RNN which does this is the **gated recurrent units** (GRUs), which use a "reset" gate r_t as an update gate u_t instead of IFOG.

$$h_t = u_{t-1} + (1 - u_{t-1})\sigma\left(W_h \begin{bmatrix} x_t \\ r_{t-1} \odot h_{t-1} \end{bmatrix} + b_h\right)$$

$$u_t = \sigma\left(W_u \begin{bmatrix} x_t \\ h_t \end{bmatrix} + b_u\right)$$

$$r_t = \sigma\left(W_r \begin{bmatrix} x_t \\ h_t \end{bmatrix} + b_r\right)$$

Experiments have shown that although several variations of the LSTM model and GRU model exist, no other architectures have been found which significantly improve the LSTM's performance.



LSTM: A Search Space Odyssey

Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, Jürgen Schmidhuber

***Abstract*—Several variants of the Long Short-Term Memory (LSTM) architecture for recurrent neural networks have been proposed since its inception in 1995. In recent years, these networks have become the state-of-the-art models for a variety of machine learning problems. This has led to a renewed interest in understanding the role and utility of various computational components of typical LSTM variants. In this paper, we present the first large-scale analysis of eight LSTM variants on three representative tasks: speech recognition, handwriting recognition, and polyphonic music modeling. The hyperparameters of all LSTM variants for each task were optimized separately using random search, and their importance was assessed using the powerful fANOVA framework. In total, we summarize the results of 5400 experimental runs (≈ 15 years of CPU time), which makes our study the largest of its kind on LSTM networks. Our results show that none of the variants can improve upon the standard LSTM architecture significantly, and demonstrate the forget gate and the output activation function to be its most critical components. We further observe that the studied hyperparameters are virtually independent and derive guidelines for their efficient adjustment.**

***Index Terms*—Recurrent neural networks, Long Short-Term Memory, LSTM, sequence learning, random search, fANOVA.**

synthesis [10], protein secondary structure prediction [11], analysis of audio [12], and video data [13] among others.

The central idea behind the LSTM architecture is a memory cell which can maintain its state over time, and non-linear gating units which regulate the information flow into and out of the cell. Most modern studies incorporate many improvements that have been made to the LSTM architecture since its original formulation [14, 15]. However, LSTMs are now applied to many learning problems which differ significantly in scale and nature from the problems that these improvements were initially tested on. A systematic study of the utility of various computational components which comprise LSTMs (see Figure 1) was missing. This paper fills that gap and systematically addresses the open question of improving the LSTM architecture.

We evaluate the most popular LSTM architecture (*vanilla LSTM*; Section II) and eight different variants thereof on three benchmark problems: acoustic modeling, handwriting recognition, and polyphonic music modeling. Each variant differs from the vanilla LSTM by a single change. This allows us to isolate the effect of each of these changes

Figure 6: Study by Google investigating 5400 LSTM variants

4.6 Vanishing and Exploding Gradients

RNNs are prone to vanishing gradients, so Martens and Sutskever (2011) propose multiplying the gradient by the inverse Hessian under the assumption that the Hessian will shrink as rapidly as the gradient. However, this results in a need for a larger batch size and a much larger computational burden.

An alternative method (Pascanuet al., 2013) is to add the regularization term

$$\Omega = \sum_t \left(\frac{|| \bigtriangledown_{h_t} L \frac{\partial h_t}{\partial h_{t-1}} ||}{|| \bigtriangledown_{h_t} L ||} - 1 \right)^2$$

which will encourage the gradient to maintain it's magnitude as it flows backward in time.

RNNs are also apt to have exploding gradients. This can cause the the next step in the update to overshoot, undoing the advantage of previous work. One wants learning rates that decay slowly enough that consecutive steps have approximately the same learning rate.

The most common method for handling exploding gradients is to use **gradient clipping** which sets a maximum threshold (either elementwise or on the norm) for the magnitude of the gradient. If the gradient exceeds this threshold, the gradient will be "clipped" down to the threshold. For example, if g is the gradient and we use a threshold of $\lambda > 0$, then:

$$\Big(||g|| > \lambda\Big) \Longrightarrow \Big(g \leftarrow \frac{g}{||g||}\lambda\Big)$$

$$\Big(||g|| \leq \lambda\Big) \Longrightarrow \Big(g \leftarrow g\Big)$$

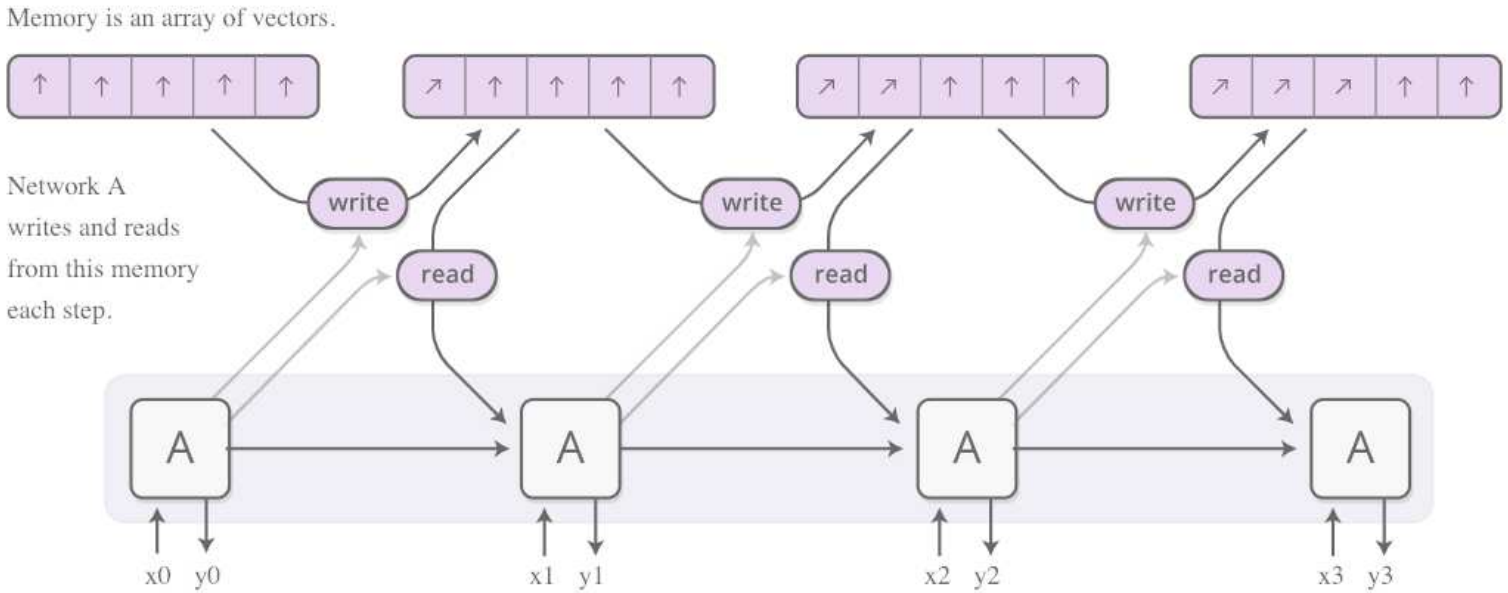
If one still has problems with an exploding or vanishing gradient, take a random step.

RNNs with LSTMs are good at storing the "general theme" of a sequence of input data when performing tasks such as sentiment analysis, but there is no part of their architecture that explicitly stores "facts" of data.

In order to incorporate that capability, some neural networks inspired by RNNs (sometimes called "Augmented RNNs") have been developed which possess the ability to store certain facts. One common augmented RNN is the **Neural Turing Machine** (NTM), which reads from and writes to explicit memory cells at each time step.

Neural Turing Machines

Neural Turing Machines [2] combine a RNN with an external memory bank. Since vectors are the natural language of neural networks, the memory is an array of vectors:



NTMs and other augmented RNNs access memory cells in an odd way: they read from and write to all memory cells at every time step, but in different amounts. This is accomplished by using an **attention mechanism** which determines how much weight to read from or write to each of the memory cells given the current attention and input. Such memory cells can be modified to include not only scalar values, but also entire vectors, allowing **content-based addressing** where groups of memories (such as phrases of related words) are read together and written to together.

Sparse attention mechanisms are an active area of DNN research and seem to have important potential (Bricken, 2021).

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

- from the input to the hidden state,
- from the previous hidden state to the next hidden state,
- from the hidden state to the output.

In an RNN with basic architecture, each of these three blocks is related to a single weight matrix and corresponds to a shallow transformation. A "shallow transformation" means that the transformation would be represented by a single layer within a multilayer perceptron (vanilla feedforward NN). Typically this shallow transformation is a learned affine transformation followed by a fixed nonlinearity.

Many different researchers—Graves et al. (2013), Pascanu et al. (2014), Bengio (1996), and Jaeger (2007)—have suggested introducing depth into each of these operations. One needs enough depth to perform the required mappings; examples are shown below:

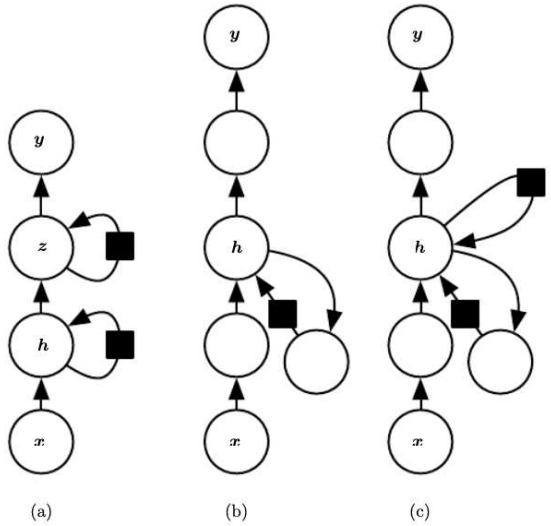


Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu *et al.*, 2014a). *(a)* The hidden recurrent state can be broken down into groups organized hierarchically. *(b)* Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts. This may lengthen the shortest path linking different time steps. *(c)* The path-lengthening effect can be mitigated by introducing skip connections.

5. Closing Comments

We have covered feedforward DNNs, and CNNs and RNNs. The material is pretty complex and often ad hoc. Theory is slowly being developed, but the applications are charging forward.

We have not covered **Variational Autoencoders** or **Generative Adversarial Networks**, both of which have important applications. Instead we have focused on CNNs, useful for astrostatistics image data, and RNNs, useful for time-varying astrostatistical signal.