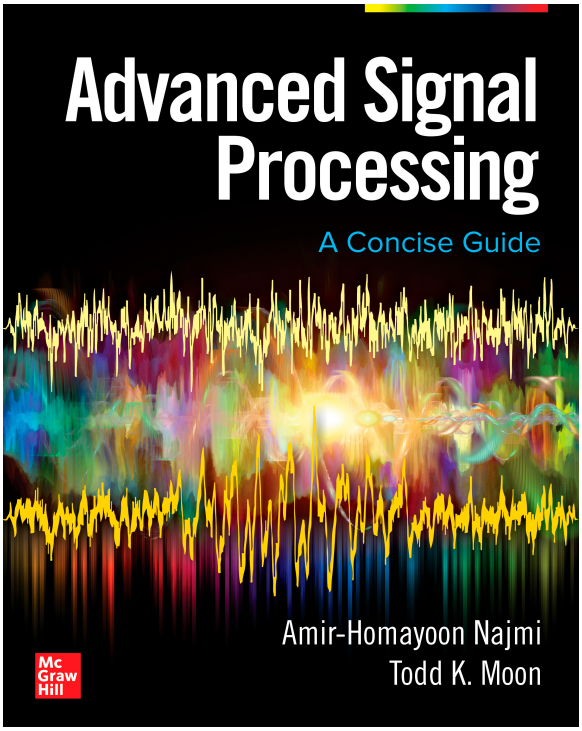
Modern Neural Networks

Amir-Homayoon Najmi, Patrick R. Emmanuel, and Todd K. Moon

1. ABSTRACT

Deep neural networks have been tremendously successful in many areas from speech and image recognition to genomics. This article explores and provides insight into modern neural network concepts and applications; it is based on a chapter in the textbook Advanced Signal Processing: A Concise Guide *published by McGraw Hill Professional in August 2020 (https://www.mhprofessional.com/najmi).*



2. Introduction

Neural networks have become a major field of research in machine learning, with applications to complex problems such as optimization, pattern recognition, and system identification. The most important among many reasons to use the human nervous system (a network of nearly 90 billion interconnected neurons) as a model is the fact that the human brain is able to successfully deal with complex problems such as face recognition. Conversely, better neural network models and a fundamental understanding of their inner workings will hopefully enable us to understand the human brain better; if our models learn to recognize, generalize, and discriminate complex patterns, perhaps they will reveal how the brain uses the same identified mechanisms in its processes. The potential to apply useful ideas from a model of the human nervous system to difficult problems is perhaps the most immediate reason neural networks have gained such prominence in modern computer applications.

The most fundamental structure of a neural network is based on a single neuron, illustrated in figure 1, consisting of a cell body called the soma, several extensions of the soma called dendrites, and the axon, which is a single nerve fiber connecting the soma to thousands of other neurons. The axon and dendrites can be thought of as insulated conductors with different impedances that transmit electrical signals to the neuron. The connection between neurons can occur on the soma or on junctions on the dendrites known as synapses that regulate signals between neurons. The totality of the neurons with their dendrites and axon connections and synapses form a neural network.

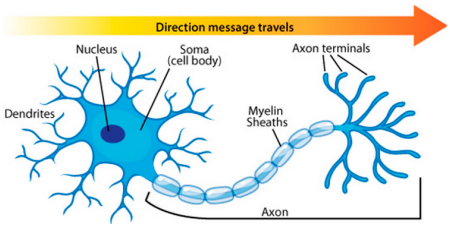


Fig. 1. Anatomy of a neuron (by permission from ASU's Ask A Biologist https://askabiologist.asu.edu).

Success of models of the human neural network based on the perceptron with activation function σ, as described in the next section 3, is closely connected with theorem 1 which is known as the universal approximation theorem of neural networks [1], and is illustrated in figure 2 for the one dimensional space $\mathbb{R}$.

Theorem 1. *If $f(\boldsymbol{x})$ is a continuous function defined on a compact subset of $\mathbb{R}^D$, then for any $\epsilon > 0$ there exists a positive integer N, real numbers $b_n \in \mathbb{R}$, real vectors $\boldsymbol{w_n} \in \mathbb{R}^D$, and real numbers $\alpha_n \in \mathbb{R}$, for which*

$$|f(\boldsymbol{x}) - h(\boldsymbol{x})| < \epsilon, \quad h(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \sigma\left(\boldsymbol{w}_n^T \boldsymbol{x} + b_n\right),$$

where the function σ satisfies the following conditions:

- *σ is sigmoidal, i.e., $\sigma(v) \to 1$ as $v \to \infty$, and $\sigma(v) \to 0$ as $v \to -\infty$.*
- *σ is discriminatory, i.e., it cannot map the linear variety $\boldsymbol{w}_n^T \boldsymbol{x} + b_n$ to a set of measure 0.*

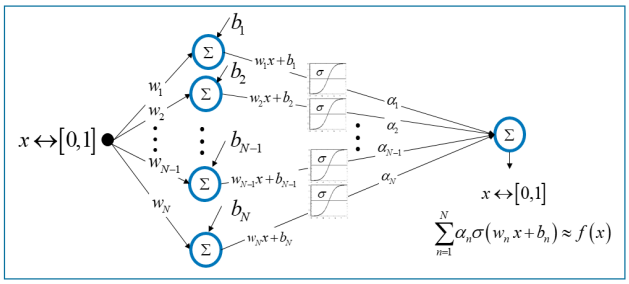


Fig. 2. The universal approximation theorem 1: $x \in [0,1] \subset \mathbb{R}$.

The second condition in theorem 1 on the function σ ensures that the inverse function $\sigma^{-1}(.)$ is well-defined and continuous on $[0,1]$. Both conditions on the function σ are satisfied by the sigmoid function

$$\sigma(v) = 1/(1+e^{-v}).$$

Using the sigmoid function it is not difficult to visualize the result of the theorem for an arbitrary continuous function $f(x)$ of a single variable $x \in [a,b] \subset \mathbb{R}$. We note that $\sigma(wx+b)$ is equal to $1/2$ at $x = -b/w$ at which point its derivative is $w/4$. Changing the ratio $-b/w$ simply shifts the sigmoid function by that amount while changing w changes the gradient at that point. Thus, instead of the pair of parameters (w,b) it is more convenient to work with parameters (w,s) where $s \equiv -b/w$, and so we consider the function $\sigma(w(x-s))$. Figure 3 shows the function $\sigma(w(x-0.499)) - \sigma(w(x-0.501))$ for three different values of w with x ranging in $[0.48, 0.52]$. Clearly, a pair of sigmoids are required to produce (any) one of the functions depicted in figure 3 that when appropriately shifted and scaled can produce an approximation to any given continuous function defined on a compact subset of $\mathbb{R}$ to any degree of accuracy. The approximation improves as the

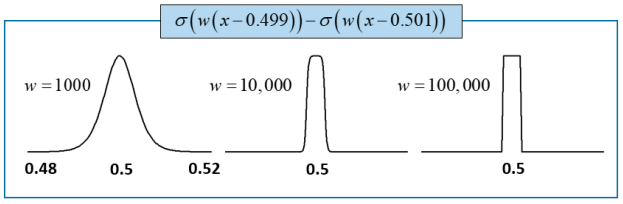


Fig. 3. The difference between two sigmoid functions $\sigma(w(x-s+0.001)) - \sigma(w(x-s-0.001))$ for $s = 0.5$, $w = 1000, 10,000, 100,000$, for $x \in [0.48, 0.52]$.

number of shifts s_n are increased, thus increasing the number of nearly rectangular bumps centered on s_n and covering the interval $[0,1]$.

The application of theorem 1 to a neural network with a single hidden layer requires the insertion of a final sigmoid function to obtain the network output. We will introduce the neuron model in section 3 and neural networks in section 4.

Although it appears that a sigmoidal function is necessary for the approximation theorem to hold, current practice has replaced the sigmoid [2] with other activation functions such as the ReLU (see section 3 and figure 6). The practical challenge is how to learn the parameters of a neural network that can produce arbitrarily close approximations to any continuous function.

3. Perceptron and the Neuron Model

The simplest model of a neuron is the classical perceptron [3] as shown in figure 4. Signals on dendrites are modeled by a real vector $\boldsymbol{x} \equiv [x_1, \ldots, x_N]^T$, whose N elements are linearly combined using N elements of a real weight vector $\boldsymbol{w} \equiv [w_1, \ldots, w_N]^T$, producing a unity output when that linear combination is greater than or equal to an internal threshold value $-b$. This threshold computing unit is known as a perceptron; it divides the N dimensional space $\mathbb{R}^N$ into two regions separated by a decision boundary given by the hyperplane $\boldsymbol{w}^T\boldsymbol{x} + b = 0$.

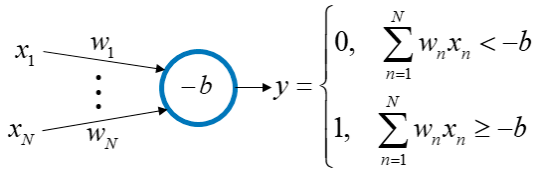


Fig. 4. A threshold computing unit model of a neuron.

By defining an additional input equal to b we can set the internal threshold to 0; the input b is equivalent to an input of 1 multiplied by a weight b; thus, the output of the classical perceptron is $y = b + \boldsymbol{w}^T\boldsymbol{x}$. In the modern development of neural networks, the output of a classical perceptron is passed through a possibly nonlinear, or piecewise linear, activation function σ; the resulting threshold computing unit is known as an artificial neuron (often simply referred to as a neuron), and is illustrated in figure 5.

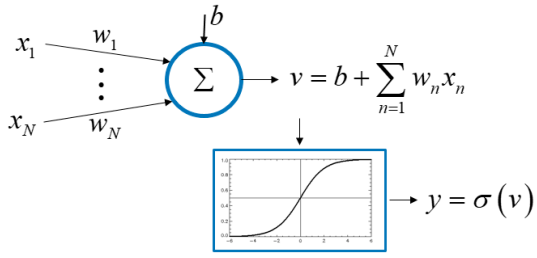


Fig. 5. An artificial neuron.

Some common activation functions [4] illustrated in figure 6, are:

- The sigmoid (or logistic) function $\sigma(v) = 1/(1 + e^{-v})$ (this is used in figure 5).
- The hyperbolic tangent $\sigma(v) = (e^v - e^{-v})/(e^v + e^{-v})$.
- The Rectified linear unit (ReLU) $\sigma(v) = \text{Max}(0, v)$.
- The softplus function, a smooth approximation to ReLU, $\sigma(v) = \ln(1+e^v)$. Note that the derivative of the softplus is the sigmoid function.
- Leaky Rectified linear unit (LReLU) is much the same as ReLU except that it allows for a small gradient for negative input, $\sigma(v) = v$ for $v \geq 0$, and αv for $v < 0$.
- Exponential linear unit (ELU), $\sigma(v) = v$ for $v \geq 0$, and $\alpha(e^v - 1)$ for $v < 0$.

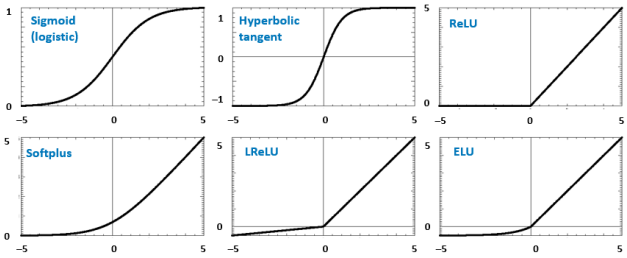


Fig. 6. Some activation functions.

A single neuron can be used as a linear classifier. For instance, using the sigmoid function, the output of a single neuron can be interpreted as the probability of class membership in a binary classification problem. Suppose that the outcome of a Bernoulli trial is a discrete random variable y taking the values 1 or 0 with probabilities p and $1-p$, where p is a function of the input variable x. Then, in N independent trials the conditional likelihood function is

$$\prod_{n=1}^{N} \Pr\left(y = y_n \,|x = x_n\right) = \prod_{n=1}^{N} p(x_n;\theta)^{y_n} \left(1 - p\left(x_n;\theta\right)\right)^{1-y_n},$$

for some parameter θ. The logistic regression model is given by

$$\ln \frac{p}{1-p} = b + \boldsymbol{w}^T\boldsymbol{x},$$

whose solution

$$p = \frac{1}{1 + e^{-b-\boldsymbol{w}^T\boldsymbol{x}}}$$

is the nonlinear output of the sigmoid function for a neuron (this is also the Boltzmann distribution for a two-state system whose states differ in energy by $b + \boldsymbol{w}^T\boldsymbol{x}$). The misclassification error is minimized when we actually predict $y = 1$ for $p \geq 0.5$, and $y = 0$ for $p < 0.5$; i.e., $y = 1$ when $b + \boldsymbol{w}^T\boldsymbol{x} \geq 0$ and $y = 0$ otherwise. The decision boundary is the solution to $b + \boldsymbol{w}^T\boldsymbol{x} = 0$ and so logistic regression is a linear classifier. Note that the distance of any point $\boldsymbol{x}$ to the boundary is given by $\left|b + \boldsymbol{w}^T\boldsymbol{x}\right| / \sqrt{b^2 + \boldsymbol{w}^T\boldsymbol{w}}$ which also defines the class probability. Substituting the logistic model into the likelihood function we can proceed to determine the optimal parameters b and $\boldsymbol{w}$ by maximizing the likelihood function using numerical techniques.

Classification problems with nonlinear decision boundaries, however, require a network of neurons. For instance, figure 7 shows two patterns: pattern (a) can be separated by a straight line, but pattern (b) cannot.

The procedure commonly used to train many neural networks is based on gradient descent [5] and known as backpropagation [6]. The gradient descent algorithm is an iterative method to reach a local minimum of an error surface that is a positive function of a vector $\boldsymbol{w} \in \mathbb{R}^N$ (we take the vector to

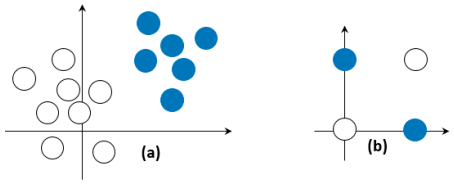


Fig. 7. Pattern (a) shows linear separation; no straight line can separate the pattern in (b).

be real), namely, $\mathcal{E}(\boldsymbol{w})$ (if the surface is convex then gradient descent finds the global minimum). We wish to choose the increment $\Delta\boldsymbol{w}$ so that we descend in the direction of maximum change of the error surface. Proceeding with a Taylor series approximation

$$\mathcal{E}(\boldsymbol{w}+\Delta\boldsymbol{w}) \approx \mathcal{E}(\boldsymbol{w}) + \Delta\boldsymbol{w}^T\,\nabla_{\boldsymbol{w}}\mathcal{E},$$

we choose the increment

$$\Delta\boldsymbol{w} = -\mu\nabla_{\boldsymbol{w}}\mathcal{E},$$

where $\mu > 0$ is the learning rate, leading to a decrease in the error, i.e.,

$$\mathcal{E}(\boldsymbol{w}+\Delta\boldsymbol{w}) \approx \mathcal{E}(\boldsymbol{w}) - \mu\left|\nabla_{\boldsymbol{w}}\mathcal{E}\right|^2 < \mathcal{E}(\boldsymbol{w}).$$

There is no guarantee, of course, that the global minimum can be achieved for a non-convex error surface. Nevertheless, gradient descent, or modifications to it, are used in training most artificial neural networks that use supervised learning. We illustrate this algorithm in the training of a single neuron to approximate a specific value c for an input data vector $\boldsymbol{x}$. In practice, we would have a collection of input vectors $\boldsymbol{x}^{[k]}$ forming our training data with a weight vector $\boldsymbol{w}$, but for now we consider only a single instance and drop the superscript k. Thus, our aim is to start with a random set of elements for the weight $\boldsymbol{w}$ and the bias $\boldsymbol{b}$ vectors (see section 10 for more details), and to determine the optimal values that produce a final error less than some prescribed $\epsilon > 0$ using iterative techniques.

Let us define $w_0 \equiv b$, $x_0 \equiv 1$, a new input vector $\boldsymbol{x} = [x_0, \ldots, x_N]^T$ and a new weight vector $\boldsymbol{w} = [w_0, \ldots, w_N]^T$, each with $N+1$ elements, and a loss function (an error surface)

$$\mathcal{E}(\boldsymbol{w}) \equiv (y-c)^2 = (\sigma(\boldsymbol{w}^T\boldsymbol{x}) - c)^2.$$

The iterative algorithm should stop at the final weight vector $\boldsymbol{w}_f$ when $\mathcal{E}(\boldsymbol{w}_f) \leq \epsilon$. The gradient descent algorithm is defined by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \mu\frac{\partial\mathcal{E}(\boldsymbol{w})}{\partial\boldsymbol{w}}.$$

The derivative of the loss function $\mathcal{E}(\boldsymbol{w})$ with respect to the weight vector is

$$\frac{\partial\mathcal{E}(\boldsymbol{w})}{\partial\boldsymbol{w}} = 2\big(y(\boldsymbol{w}) - c\big)\sigma'\boldsymbol{x}, \quad \sigma' \equiv \frac{d\sigma(v)}{dv}, \quad v = \boldsymbol{w}^T\boldsymbol{x},$$

where $y(\boldsymbol{w})$ indicates the dependence of the neuron output on the weight vector. This gives the weight vector update

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - 2\mu\Big(y(\boldsymbol{w}) - c\Big)\sigma'\boldsymbol{x}.$$

When training data consists of K vectors $\boldsymbol{x}_k$, $k = 1, \ldots, K$, the average squared error is minimized

$$\mathcal{E} = \frac{1}{K}\sum_{k=1}^{K}\mathcal{E}_k, \quad \mathcal{E}_k \equiv (y_k - c)^2,$$

and the weight vector update is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - 2\mu\frac{1}{K}\sum_{k=1}^{K}\Big(y_k(\boldsymbol{w}) - c\Big)\sigma'\boldsymbol{x}_k.$$

In section 4 we will study artificial neural networks, i.e., connected networks of individual neurons. The backpropagation of gradient function follows a similar form when the gradient is calculated at an output layer, but will be different if the gradient is calculated at a hidden (not the output) layer. Different forms of loss functions for neural networks will be discussed in section 6.

4. Fully Connected Feed Forward Neural Networks

A Feed forward artificial neural network can be constructed by attaching layers of neurons whose outputs become inputs to the neurons in the next layer [7]. A feed forward network is fully connected when every neuron in one layer is connected to every neuron in the next layer. Figure 8 shows one such network whose input layer has N elements (data samples x_1 through x_N), three hidden layers with 3, 2, and 3 neurons, respectively, and an output layer with two neurons. The entire network is denoted by the equation $\boldsymbol{y} = h(\boldsymbol{x})$ where the input $\boldsymbol{x}$ is an $N \times 1$ vector and the output $\boldsymbol{y}$ is a 2×1 vector. The upper part of figure 8 shows the network connections with inputs and outputs; a square box represents a neuron as shown in the lower part of figure 8. Layer $l \geq 1$ has N_l neurons, each of which has N_{l-1} inputs and a single output. Each connection has a weight $w_{ij}^{(l)}$ that multiplies the output of neuron $i \geq 1$ in the previous layer $l-1$ to neuron $j \geq 1$ in layer l; the latter neuron has a bias $b_j^{(l)}$ which is equivalently defined by $w_{0j}^{(l)}$. In this example layer $l = 0$ is the input layer, $l = 4$ is the output layer, and $l = 1, 2, 3$ are the three hidden layers. Inputs to layer 1 are the N data samples of $\boldsymbol{x}$. The output of neuron j of the a hidden layer $l = 1, 2, 3$ is denoted by $y_j^{(l)}$ while the final output of neuron j in the output layer 4 is denoted by y_j, all of which are denoted by the vector $\boldsymbol{y}$ with length M which denotes the number of final outputs. Thus, for a network with $L+1$ layers consisting of one input layer ($l = 0$), $L-1$ hidden layers ($l = 1, \ldots, L-1$), and one output layer ($l = L$), the output of neuron j in layer $l \geq 1$ is

$$y_j^{(l)} = \sigma\Big(w_{0j}^{(l)} + \sum_{k=1}^{N_l} w_{kj}^{(l)} y_k^{(l-1)}\Big), \quad 1 \leq l \leq L, \; 1 \leq j \leq N_l,$$

where $y_j^0 \equiv x_j$ is the data at the input layer $l = 0$, and $y_j^L \equiv y_j$ is the output of the output layer $l = L$.

The depiction of theorem 1, for a continuous function defined on the closed interval $[0, 1]$, in figure 2 can be readily generalized to apply to a neural network with a single hidden layer by simply including an activation function before the final output. If the output is to constitute an approximation to

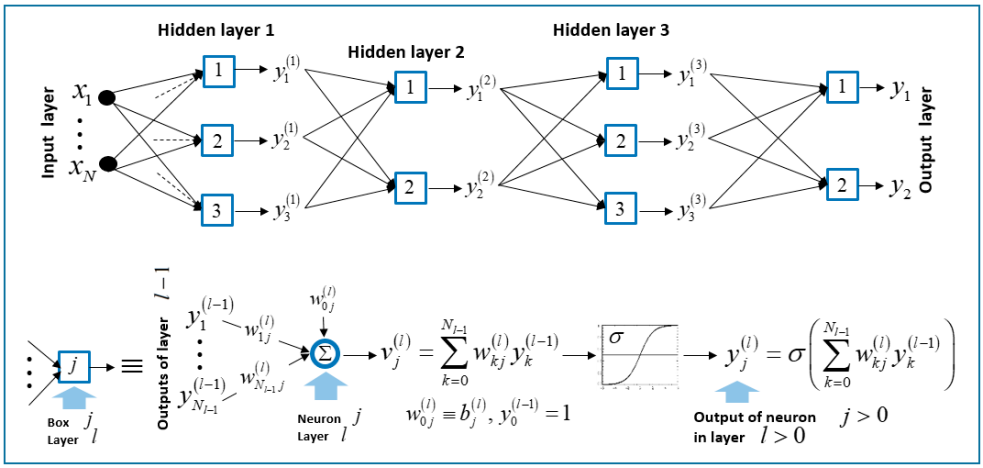


Fig. 8. An artificial neural network $\boldsymbol{y} = h(\boldsymbol{x})$ whose input layer ($l = 0$) has N elements, three hidden layer ($l = 1, 2, 3$) with $3, 2, 3$ neurons, respectively, and an output layer ($l = 4$) with 2 neurons. Each square box in the network represents a neuron shown on the bottom.

a function $f(x)$, then the input to this final activation function must be of the form $\sigma^{-1}[f(x)]$. The general requirements on the activation in theorem 1 ensure that the inverse function exists and is continuous in the same interval in which $f(x)$ is continuous and so the inverse function can be uniformly approximated by a linear combination of function $\sigma(w_n x + b_n)$. Thus, theorem 1 can be restated as follows.

Theorem 2. *Given a continuous function $f(\boldsymbol{x})$ defined on a compact subset of $\mathbb{R}^D$, and $\epsilon > 0$, a neural network with a single hidden layer and N neurons, and a sigmoidal activation function exists whose output $h(\boldsymbol{x})$ uniformly approximates $f(\boldsymbol{x})$ with $|f(\boldsymbol{x}) - h(\boldsymbol{x})| < \epsilon$.*

5. The Backpropagation Algorithm

Training of most feed forward neural networks uses the steepest descent method in a backpropagation algorithm [6]. We have already seen how to compute the required derivatives with respect to the weight functions at an output layer. The backpropagation algorithm allows us to compute the derivatives with respect to the weight functions at all hidden layers, once we have the derivatives at an output layer. To derive the hidden layer derivatives we refer to figure 9 that shows the connection of an output of neuron k in layer $l-2$ to neuron i in layer $l-1$ whose output goes through neuron j in layer l to finally arrive at $y_j^{(l)}$.
Using figure 9 we have

$$y_j^{(l)} = \sigma(v_j^{(l)}), \;\; v_j^{(l)} = \sum_{i=0}^{N_l} w_{ij}^{(l)} y_i^{(l-1)}, \;\; w_{0j}^{(l)} \equiv b_j^{(l)}, \; y_0^{(l-1)} = 1.$$

Let us assume there are M final outputs with squared errors $\mathcal{E}_m = (y_m - c_m)^2$, $1 \le m \le M$ whose sum divided by M is the average output error. Differentiating $\mathcal{E}_m$ with respect to the weights at layer $l-1$ gives

$$\frac{\partial \mathcal{E}_m}{\partial w_{ki}^{(l-1)}} = \frac{\partial \mathcal{E}_m}{\partial v_i^{(l-1)}} \frac{\partial v_i^{(l-1)}}{\partial w_{ki}^{(l-1)}} = \frac{\partial \mathcal{E}_m}{\partial v_i^{(l-1)}} y_k^{(l-2)}.$$

The last derivative on the right hand side is

$$\frac{\partial \mathcal{E}_m}{\partial v_i^{(l-1)}} = \sum_{j=1}^{N_l} \frac{\partial \mathcal{E}_m}{\partial v_j^{(l)}} \frac{\partial v_j^{(l)}}{\partial v_i^{(l-1)}} = \sum_{j=1}^{N_l} \frac{\partial \mathcal{E}_m}{\partial v_j^{(l)}} \; w_{ij}^{(l)} \sigma'\left(v_i^{(l-1)}\right),$$

which is the backpropagation formula to compute all gradients down to the first hidden layer, starting with the output layer L, and the derivative at layer L, namely,

$$\frac{\partial \mathcal{E}_m}{\partial v_j^{(L)}} = 2(y_m - c_m)\sigma'(v_m^{(L)}).$$

The gradient update at layer 1 is

$$\Delta w_{ij}^{(1)} = -\mu \, \frac{\partial \mathcal{E}_m}{\partial w_{ij}^{(1)}} = -\mu \; x_i \sum_{k=1}^{N_1} \frac{\partial \mathcal{E}_m}{\partial v_j^{(2)}} \; w_{jk}^{(2)} \sigma'\left(v_j^{(1)}\right).$$

We will study examples of neural network training and loss functions in section 6. For now we should emphasize the importance of the nonlinear activation in the design of an artificial neural network. In general, an activation such as the sigmoid function that has saturation levels in both directions leads to the *vanishing gradient* problem which can permanently deactivate many of the neurons in the network, thus decreasing the capacity of the network. In addition, the output of the sigmoid is not centered at zero and this can cause jumps in gradient updates; if the incoming data is all positive then during backpropagation gradients become all positive or all negative; for this reason, the hyperbolic tangent is preferred over the sigmoid, but it too suffers from the vanishing gradient problem [8].

The ReLU and some of its variants (Noisy ReLU, Leaky ReLU, and Exponential Linear Units) alleviate the vanishing gradient problem and as a result have, in the last few years, completely replaced the older sigmoid and hyperbolic tangent activations. The ReLU itself suffers from the *dead neuron* problem when for high learning rates some neurons never activate for the entire training data. This problem has been solved by lowering the learning rate or using other variants

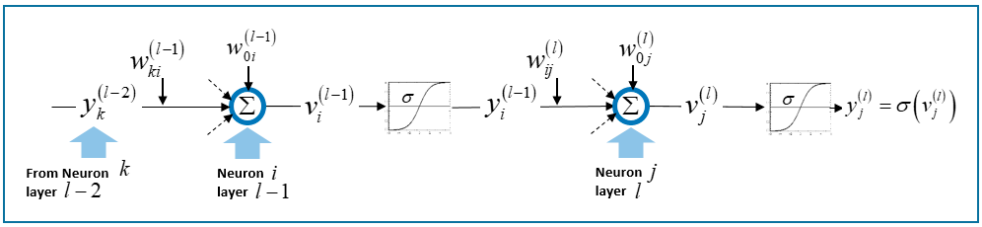


Fig. 9. A portion of a feed forward neural network: $y_k^{(l-2)}$ is the output of neuron k in layer $l-2$, $y_i^{(l-1)}$ is the output of neuron i in layer $l-1$, and $y_j^{(l)}$ is the output of neuron j in layer l.

of the ReLU such as the Leaky ReLU when the parameter α is chosen to be a small number such as 0.01. In practice, it is best to start with ReLU with lower learning rates while monitoring the fraction of dead neurons.

Although activation functions in the hidden layers are often chosen to be the same (e.g., ReLU), the output layer activation is selected depending on the loss function $\mathcal{E}$ the neural network is designed to minimize (see section 6 for a discussion of loss functions), and the function $h(\boldsymbol{x})$ that the network is designed to compute. For instance, in a regression problem if the output values are in the range $[-A, +A]$, $A > 0$, then we could use the **tanh** nonlinearity, while if the output values are non-negative then the ReLU activation is appropriate. However, in regression problems a nonlinearity is often not used in the output layer; in other words, activation is the identity operation.

When the network is used as a binary classifier the sigmoid (logistic) function is the most commonly used activation at the output layer, in which case the output is the conditional probability (conditioned on the input to the network) of belonging to the positive class (typically denoted by 1 in a binary vector $[0, 1]$). When the network is used to classify among $N_c > 2$ classes of data, then the **softmax** activation function is used to minimize the cross-entropy loss function (see section 6)

$$\textbf{softmax}(\boldsymbol{v}) = [e^{v_1}, \ldots, e^{v_{N_c}}]^T / S, \quad S \equiv \sum_{n=1}^{N_c} e^{v_n},$$

where $\boldsymbol{v}$ is the output before the non-linearity as defined in figure 9. To avoid numerical instability the **softmax** nonlinearity is usually calculated by multiplying the numerator and the denominator by $e^{-v_{\max}}$. When $N_c = 2$, $\sigma(v_1) + \sigma(v_2) = 1$ and this reduces to the sigmoid (logistic) nonlinearity.

6. Loss Functions in Neural Network Training

Let $\boldsymbol{y} = h(\boldsymbol{x}; \boldsymbol{w})$ denote the network output for input $\boldsymbol{x}$ and weight vector $\boldsymbol{w}$, and consider training the network using training data $\boldsymbol{x}_k$ to reach a desired output data $\boldsymbol{d}_k$, $1 \le k \le K$. A regression problem loss function is

$$\mathcal{E} = \frac{1}{K} \sum_{k=1}^{K} \left\| h(\boldsymbol{x}_k; \boldsymbol{w}) - \boldsymbol{d}_k \right\|^2.$$

Note that this loss function requires the last activation function to be the identity, as illustrated in figure 10 describing a network with L hidden layers whose first $L-1$ layers with activation σ, and the last layer L with identity activation. The input to the network is an $N \times 1$ vector $\boldsymbol{x}$ with elements $x_1, \ldots, x_N$ and the output $\boldsymbol{y} = \boldsymbol{v}^{(L)}$ has N elements.

Now consider a two-class classifier neural network with a sigmoid activation at the output layer, and training data $\boldsymbol{x}_k$ belonging to classes 0 and 1. In this case the network output determines class, e.g., we assign $\boldsymbol{x}_k$ to class 1 when $y_k \ge 0.5$ and $\boldsymbol{x}_k$ to class 0 when $y_k < 0.5$ (in practice the threshold is chosen according to some optimality criterion). Then the appropriate cost function is the binary cross-entropy function defined by

$$\mathcal{E} = -\frac{1}{K} \sum_{k=1}^{K} \Big(d_k \ln(y_k) + (1 - d_k) \ln(1 - y_k) \Big),$$

where $d_k = 0$ or 1.

When the network is used to classify data into $N_c > 2$ classes using the **softmax** activation at the output layer, the associated loss function is the cross-entropy defined by

$$\mathcal{E} = -\frac{1}{K} \sum_{k=1}^{K} \sum_{n=1}^{N_c} d_{kn} \ln y_{kn}, \quad d_{kn} = 0 \text{ or } 1.$$

For instance, consider a 3-class problem with classes A, B, and C, and a desired vector of the form [A, B, C, C, B, A]. In order to train the network we encode this vector as a *one-hot encoded matrix* whose elements are the desired values d_{kn},

$$\begin{aligned} A \to [1,0,0] = d_{1n}, \quad B \to [0,1,0] = d_{2n}, \\ C \to [0,0,1] = d_{3n}, \\ d_{5n} = d_{2n}, \quad d_{4n} = d_{3n}, \quad d_{6n} = d_{1n}. \end{aligned}$$

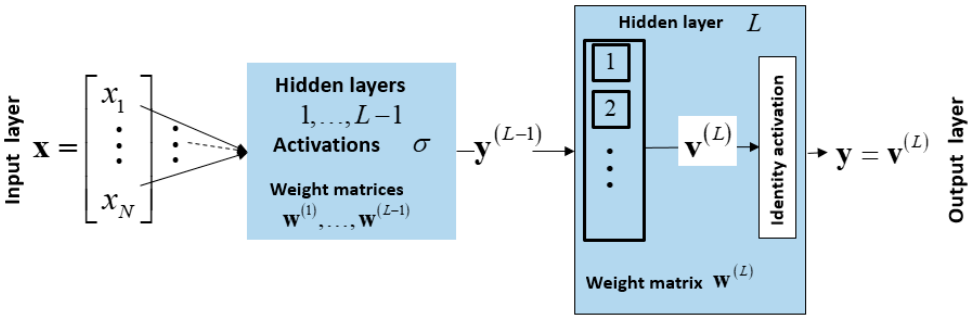


Fig. 10. A fully connected regression neural network with L hidden layers: the first $L-1$ layers have activation σ and weight matrices $\boldsymbol{w}^{(1)},\ldots,\boldsymbol{w}^{(L-1)}$, and the last hidden layer has weight matrix $\boldsymbol{w}^{(L)}$ with identity activation so that $\boldsymbol{y} = \boldsymbol{v}^{(L)}$.

7. Gradient Descent Variants

The gradient descent algorithm

$$\boldsymbol{g}^{(n)} \equiv \nabla_{\boldsymbol{w}^{(n)}} \mathcal{E}\left(\boldsymbol{w}^{(n)}\right), \quad \boldsymbol{w}^{(n+1)} \leftarrow \boldsymbol{w}^{(n)} - \mu \boldsymbol{g}^{(n)},$$

is often slow to converge. An approach to accelerate learning is momentum optimization [5] that introduces a momentum vector $\boldsymbol{m}$ to store and use previous step's gradient direction. Classical momentum algorithms accumulate a decaying sum of the previous gradients into a momentum vector $\boldsymbol{m}$ and use this in the update instead of the gradient,

$$\boldsymbol{m}^{(n+1)} \leftarrow \beta \boldsymbol{m}^{(n)} + \boldsymbol{g}^{(n)}, \quad \boldsymbol{w}^{(n+1)} \leftarrow \boldsymbol{w}^{(n)} - \mu \boldsymbol{m}^{(n+1)}.$$

Learning is accelerated in any direction along which the gradient is relatively stable across training steps, but learning is slowed in any direction along which the gradient is oscillatory; β is a friction coefficient to ensure that the momentum gradually decreases to zero.

If the momentum vector is pointing in the right direction then it may be more accurate to evaluate the gradient a little further along than the current position. The Nesterov accelerated gradient (NAG) method is equivalent to improving the momentum vector and achieves a much better bound than standard gradient descent by evaluating the gradient at the updated value of momentum $\boldsymbol{m}^{(n+1)}$. In the momentum update $\boldsymbol{m}^{(n)}$ does not depend on the gradient $\boldsymbol{g}^{(n)}$; Nesterov's algorithm introduces a dependence in the form

$$\boldsymbol{g}^{(n)} \equiv \nabla_{\boldsymbol{w}^{(n)}} \mathcal{E}\left(\boldsymbol{w}^{(n)} - \beta\mu \boldsymbol{m}^{(n)}\right)$$

when using the update equation.

To address the problem of learning in a "long narrow valley" one might try to cut across the slope heading towards the global minimum, gaining progress on the variable that needs the most change (the long valley), even though it is not approaching the global minimum in the steepest direction. AdaGrad (adaptive subgradient descent) scales down the gradient in the steepest direction by its norm (so instead of going straight downhill it traverses less steep directions); it uses different learning rates on different parameters by adaptively adjusting the rates according to the "steepness" in each component [5]

$$\boldsymbol{g}^{(n)} \equiv \nabla_{\boldsymbol{w}^{(n)}} \mathcal{E}\left(\boldsymbol{w}^{(n)}\right), \quad \nu^{(n+1)} \leftarrow \beta\nu^{(n)} + \left|\nu^{(n)}\right|^2,$$
$$\boldsymbol{w}^{(n+1)} \leftarrow \boldsymbol{w}^{(n)} - \frac{\mu \boldsymbol{g}^{(n)}}{\sqrt{\nu^{(n+1)} + \varepsilon}}.$$

This algorithm accelerates learning along directions that have changed slightly but suffers from the exploding norm problem that halts learning altogether. A simple cure is to use an exponentially weighted adaptive norm calculation for ν, known as RMSProp,

$$\nu^{(n+1)} \leftarrow \alpha\nu^{(n)} + (1-\alpha)\left|\nu^{(n)}\right|^2, \quad 0 \ll \alpha < 1.$$

A combination of RMSProp and the classical momentum method is Adam (adaptive momentum estimation) [5],

$$\boldsymbol{g}^{(n)} \equiv \nabla_{\boldsymbol{w}^{(n)}} \mathcal{E}\left(\boldsymbol{w}^{(n)}\right), \quad \boldsymbol{m}^{(n+1)} \leftarrow \beta \boldsymbol{m}^{(n)} + (1-\beta)\boldsymbol{g}^{(n)},$$
$$\boldsymbol{m}^{(n+1)} \leftarrow \frac{\boldsymbol{m}^{(n+1)}}{(1-\beta)},$$
$$\nu^{(n+1)} \leftarrow \alpha\nu^{(n)} + (1-\alpha)\left|\nu^{(n)}\right|^2, \quad \nu^{(n+1)} \leftarrow \frac{\nu^{(n+1)}}{(1-\alpha)},$$
$$\boldsymbol{w}^{(n+1)} \leftarrow \boldsymbol{w}^{(n)} - \mu \boldsymbol{m}^{(n)} / \sqrt{\nu^{(n+1)} + \varepsilon}.$$

Exponentially weighted momentum adaptive methods often use a time-dependent parameter, e.g., $\beta^{(n)} = 0.99\left(1 - 0.5 \times 0.96^{n/250}\right)$. Figure 11 shows a comparison between gradient descent and AdaGrad/Adam-type algorithms.

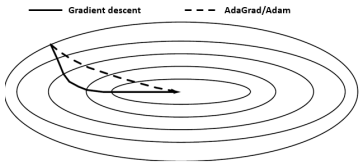


Fig. 11. Gradient descent compared with AdaGrad/Adam.

8. Single-Hidden-Layer and Multiple-Hidden-Layer Neural Networks

As theorem 1 suggests, arbitrarily complicated functions can be approximated by a neural network with a single

hidden layer. But deep networks with more than one hidden layer can have much higher efficiency with far fewer (possibly exponentially fewer) neurons per layer. Deep networks have been successfully used in difficult problems such as speech recognition and image classification. Development of a neural network often starts with one or two layers, and then increasing the number of hidden layers until overfitting is observed or suspected.

The number of input neurons is determined by the input data. For instance, to classify images of hand-written characters, the input images are small, say $30 \times 30 = 900$ pixels, and if no data reduction prior to the neural network is performed then 900 neurons are needed in the first layer. If the characters are classified into $26 + 26$ letters, 10 numeric digits, and a dozen punctuation marks, then 74 neurons are needed in the final output layer.

To illustrate the effect of more hidden layers in reducing the number of neurons with no loss in performance we examine the spiral data consisting of two paired sets $(\boldsymbol{x}_1, \boldsymbol{y}_1)$ and $(\boldsymbol{x}_2, \boldsymbol{y}_2)$, which we construct by choosing $\boldsymbol{z} = 13.6\sqrt{\boldsymbol{u_0}}$ and

$$\boldsymbol{x}_1 = -\boldsymbol{z}\cos(\boldsymbol{z}) + \sigma\boldsymbol{u}_1, \quad \boldsymbol{y}_1 = +\boldsymbol{z}\sin(\boldsymbol{z}) + \sigma\boldsymbol{u}_2,$$
$$\boldsymbol{x}_2 = +\boldsymbol{z}\cos(\boldsymbol{z}) + \sigma\boldsymbol{u}_3, \quad \boldsymbol{y}_2 = -\boldsymbol{z}\sin(\boldsymbol{z}) + \sigma\boldsymbol{u}_4,$$

where $\boldsymbol{u}_k$, $0 \le k \le 4$, are random vectors whose elements are uniformly distributed in $[0, 1]$, and σ denotes the strength of the noise (0.5 in our example); element by element multiplication and association is implied in the equations.

Figure 12 shows the classification boundaries for three neural networks consisting of a single hidden layer with 10 (a), 1000 (b), and 5000 (c) neurons, respectively. Clearly, increasing the number of neurons improves the network performance as expected by the universal approximation theorem 1.

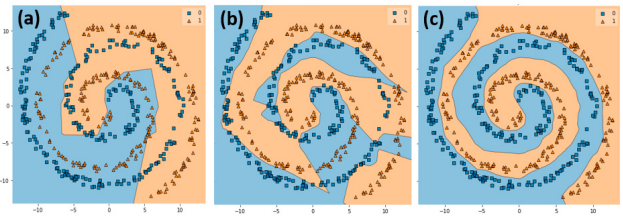


Fig. 12. Classification boundaries for three neural networks with a single hidden layer and 10 (a), 1000 (b), and 5000 (c) neurons, respectively.

Figure 13 shows the performance with more hidden layers but with a significantly reduced number of neurons. Image (a) shows the classification boundaries using 2 hidden layers with 10 neurons each, while image (b) shows the boundaries for 3 hidden layers with 10 neurons each; both images show similar performance to the single hidden layer network with 5000 neurons, but a reduction in the total number of neurons by a factor of approximately 150. We note that to illustrate the power of deep representations compared with shallow ones, we limited the number of epochs (see section 9) to 100; increasing the number of epochs does allow for better classifications using single-layer networks (even with fewer neurons) at the expense of significantly more training time. All models were optimized using Adam (see section 7).

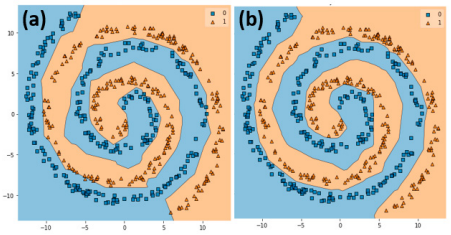


Fig. 13. Classification boundaries for two neural networks with 2 (a) and 3 (b) hidden layers, each with 10 neurons.

9. Mini-Batch Training and Normalization

When performing gradient descent adaptation the average of the loss over the entire training set can be used to compute gradients for backpropagation; this is known as batch gradient descent and can be extremely time consuming and memory inefficient for large data sets. Batch training offers a more stable estimate of the error gradient which, although useful in some problems, can actually reduce performance by converging to less optimal network weights. The extreme alternative to this is to use a random ordering of the training samples, update the weights based on the loss calculated for the first training data, and use the weights as initial weights for training using the second training data and so on, to finish through the entire training set of size N_T. The process can be repeated starting with the weights from the first round, and a new random ordering of the training data. Each round is known as an epoch, by the end of which the network has seen all the training data. After completing N_E epochs, i.e., when the network has been exposed to the training data N_E times, we would have a total of $N_T \times N_E$ weight updates. This method is known as stochastic gradient descent or SGD; it can be useful in on-line learning when training can occur as new samples arrive.

The concept of stochastic mini-batch training fits somewhere between SGD and batch training. To illustrate, consider the set of indices $[1, \ldots, 20]$ that refer to 20 training samples. Choosing a mini-batch size of 4 we construct 5 mini-batches containing random indices (typically without replacement) between 1 and 20, and begin training with initial random weights $\{\boldsymbol{w}\}_0$ on mini-batch 1 resulting in updated weights $\{\boldsymbol{w}\}_1$. The weights $\{\boldsymbol{w}\}_1$ become the initial weights for training on mini-batch 2, and so on until we have the weights $\{\boldsymbol{w}\}_5$ at the end of epoch 1, which consists of 5 mini-batches. The weights $\{\boldsymbol{w}\}_5$ now become the initial weights for training through 5 mini-batches of epoch 2 which consists of another random set of training indices. In this example training ends after the set of 5 updates through epoch 3, with $\{\boldsymbol{w}\}_{15}$ as the final weights for the network, as illustrated in figure 14.

Given a mini-batch size of M_{mB}, there are N_T/M_{mB} mini-batches in each epoch, and the total number of weight updates after training through N_E epochs is $N_E \times N_T/M_{mB}$. Mini-batch size is usually much smaller than the number of training samples and should divide it; smaller mini-batch sizes increase noise in the gradients which, ironically, may be useful as an implicit regularizer [9], while larger sizes have less noise in

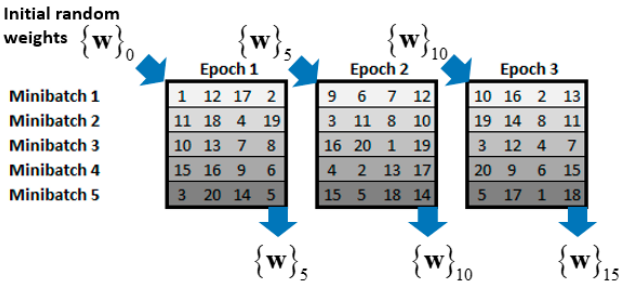


Fig. 14. Example of mini-batch training: training indices $[1, \ldots, 20]$, mini-batch size of 4, three epochs, resulting in 15 weight updates.

the gradients but they can train faster. Mini-batch size is often chosen to be a power of 2, e.g., $4, 8, 16, 32, \ldots$.

An important issue in the training phase of deep neural networks is the fact that the distribution of each layer's activations change as the distribution of inputs to that layer change through change in parameters of the previous layers; this change of distribution of network activations during training is known as *internal covariate shift* [10] and often slows convergence. Convergence can be accelerated by whitening the input sequences to all layers; whitening includes decorrelation as well as normalization. Currently, normalization alone (excluding decorrelation) via mini-batch statistics is the preferred method and is applied to each activation separately, i.e., each element of each activation vector is normalized by the mean and standard deviation of the set of its values within a mini-batch [10]. For instance, let $\boldsymbol{x} = [x_1, \ldots, x_d]$ denote an input vector to a fully connected network, and let $\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$, ..., denote the training data. Let $\boldsymbol{x}^{(k_1)}, \ldots, \boldsymbol{x}^{(k_m)}$ denote the members of a mini-batch of size m, and let $\boldsymbol{y}^{(k_1)}, \ldots, \boldsymbol{y}^{(k_m)}$ denote the outputs of a layer associated with this mini-batch. Batch normalization introduces two new $d \times 1$ vector parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, for each mini-batch, that are learnt along with the rest of the network parameters, and are defined by the Batch Normalizing Transform $\mathbf{BNT}_{\gamma,\beta} : \boldsymbol{x} \rightarrow \boldsymbol{y}$

$$\boldsymbol{\mu} \equiv \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}^{(k_i)}, \quad \boldsymbol{\sigma}^2 \equiv \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}^{(k_i)} - \boldsymbol{\mu})^2,$$
$$\hat{\boldsymbol{x}}^{(k_i)} = \frac{\boldsymbol{x}^{(k_i)} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}}, \quad 0 < \varepsilon \ll 1,$$
$$\boldsymbol{y}^{(k_i)} = \boldsymbol{\gamma} \otimes \hat{\boldsymbol{x}}^{(k_i)} + \boldsymbol{\beta}, \quad i = 1, \ldots, m,$$

where $\otimes$ indicates element by element vector multiplication. Batch normalized networks are trained using batch gradient descent or stochastic mini-batch descent with $m > 1$, and backpropagation of the usual derivatives of the loss function with respect to the weights in addition to the derivatives with respect to the BNT variables [10]. While the normalized activations $\hat{\boldsymbol{x}}^{(k_i)}$ remain internal to the network, reducing covariate shift and accelerating network training, the BNT transformed values $\boldsymbol{y}^{(k_i)}$ are passed to other network layers. The learnt variables $\boldsymbol{\gamma}, \boldsymbol{\beta}$ applied to the normalized activations allow the BNT to represent the identity transformation and preserve network capacity.

To achieve its promise, batch normalization must be accompanied by changes in several other training parameters . Some of these changes include increasing the learning rate parameter, removing dropout (see section 10), reducing L_2 weight regularization, and shuffling training samples more thoroughly so the same samples do not always appear in a mini-batch together [10].

10. Network Initialization

An important issue in network training is the initialization of the weights [11]. Zero initial values lead to equal loss derivatives with respect to all the weights, which will lead to the same value for all the weights in every iteration during training. The neural network then essentially becomes a linear model. Hence random initial values should be the starting values. There are two possible issues with random initial values: they could lead to vanishing gradients (the vanishing gradient problem), or divergent gradients (the exploding gradient problem). One way to avert the vanishing gradient problem is to use ReLU activations.

Methods to prevent the exploding gradient problem include gradient clipping (setting a threshold value for the magnitude of the gradients that if exceeded will set them to the threshold), or modifying the parameters of random number generators (uniform or Gaussian) that are used to produce the initial values. Let us define $[-r, +r]$, $r > 0$, to be the interval in which random numbers with uniform density are generated. Gaussian random numbers are assumed to be zero-mean and are defined by their variance σ. Table 1 shows random number settings that are functions of the number of inputs to a layer N_{in}, and the number of outputs of that layer N_{out}. These initializations are known as Xavier initialization, or (for the ReLU activation) He initialization [12].

Activation	**Uniform**: $[-r, r]$	**Normal**: σ
Logistic	$r = \sqrt{6}/r_0$	$\sigma = \sqrt{2}/\sigma_0$
tanh	$r = 4\sqrt{6}/r_0$	$\sigma = 4\sqrt{2}/\sigma_0$
ReLU	$r = \sqrt{22}/r_0$	$\sigma = 2/\sigma_0$
	$r_0^2 \equiv N_{\text{in}} + N_{\text{out}}$	$\sigma_0^2 \equiv N_{\text{in}} + N_{\text{out}}$

TABLE 1
Random Number Generator Settings for Network Initialization

11. Regularization

Overfitting is a major issue in neural networks when the neural network is trained to achieve excellent performance (such as low classification error) on the training data, but in doing so it becomes so specialized that it is unable to do well on any other data set [13]. An overfit neural network loses the ability to generalize, i.e., it is unable to apply what it has learned from the training data to other data it has never been exposed to.

As a practical matter, in any regression or classification problem, it is important to partition the data into two portions: a training portion and a test and validation portion. Typically, about 80–90% of the available data is used for training, with the remaining 10–20% for testing and validation. The important criterion is not so much how well the neural network performs on the training data, but how it does on the test data. The real validation in neural network training is then performed on the test data, not on the training data. In order to reliably measure the network's ability to generalize, we should never mix training and test data. Methods to reduce overfitting are known as regularization.

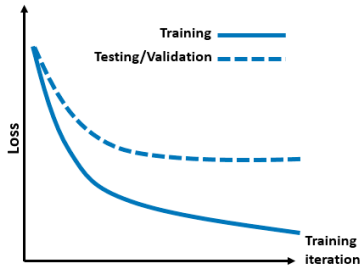


Fig. 15. An overfit neural network.

When training a neural network we often see a continual decrease in training loss accompanied by a drop in test data loss for some time before the testing loss begins to plateau or increase, as illustrated in figure 15. The gap between training and validation performance is sometimes referred to as the *generalization gap* [14]; this is a clear evidence of overfitting when the neural network has learned to represent the training data so well that it has lost the ability to generalize to holdout data.

In general controlling a neural network's ability to fit training data reduces to managing its representational capacity. Recall from theorem 1 that with enough neurons (and appropriate activation functions), standard multi-layer perceptrons are universal function approximators. Thus, sufficiently reducing the number of neurons in a neural network will reduce its representation capacity, inhibiting the model's ability to (over)fit training data. We can also control the capacity of our models by limiting the depth of the network, bounding the norm of the weight matrices with L_1 or L_2 regularization, injecting noise into the input/hidden layers of the network, and performing more general input data manipulations called *data augmentation* [15] where we perturb/transform the input data in such a way as to prevent the neural network from placing emphasis on spurious features of the data; e.g., for a cat vs dog image classifier we might perform color transformations to overcome biases in the color of the animals in the training set. These and other methods explicitly control what neural network models can learn from the data during parameter optimization (SGD and its variants) and are thus effective regularizers for practical applications.

Early stopping is another effective method to avoid overfitting [16]: when a new best loss value on the validation set is found, the current best neural network weights are saved, and training stops if the validation loss has not improved within a specified number of training iterations. The number of training iterations (or epochs) before halting is termed known as a "stopping criterion" and prevents model weights from continuing to improve on the training set while making little progress on the validation/test set (see figure 15).

Another method is the surprising idea of dropout [17], [18], when at every training step, every neuron, including input layer neurons but excluding output layer neurons, will be dropped with a probability p. A neuron that is dropped is simply ignored during this training step. The probability p is the dropout rate, and is typically set to 0.25. The idea is that the neurons that are not dropped at a given iteration must adapt to the data, becoming as useful as possible on their own and not co-adapting to their neighboring neurons. We may think of neural networks with different dropout configurations as different neural networks. If N neurons may be dropped then there are 2^N different neural networks (the set of all subsets of the given N neuron). The training with dropout gives, roughly, the average result of a large number of neural networks. This is related to the idea of boosting, in which the results of many weak classifiers are combined to produce a strong classifier result.

An improved implementation of the original concept of dropout is *inverted dropout*. For a dropout rate of p, when testing the network (without dropout), a neuron is connected to, on average, $1/(1-p)$ as many inputs as it was during training, i.e., each neuron has a total input signal that is, on average, $1/(1-p)$ as large as what the network was trained on. To compensate for this, we multiply each neuron's connection weight by $1-p$ after training.

Another way of regularizing a neural network is to tie parameters together; for instance, we may require that some weights should be equal. This provides more training data per weight. A common way of tying parameters is to use convolutional neural networks (see section 12).

12. Convolutional Neural Networks (CNN)

In the fully connected network described in section 4 every input to layer l is connected to every output of the previous layer $l-1$ using the weight functions. A fully connected network with many hidden layers and many neurons in each layer can contain tens of thousands of weights which, in cases of small training data sets, can lead to overfitting issues. A far more important problem with fully connected networks in pattern recognition problems is the absence of shift (translation) invariance, or insensitivity to local distortions. For instance, spoken words can have different speed, pitch, and

intonation, which can cause substantial variations in time location of important features in the input data. Although, a fully-connected network can in principle, learn to produce shift invariant outputs, the required number of training sets and the network size may be prohibitively large. Furthermore, fully connected networks do not take advantage of the high correlations between nearby data points.

The usual definition of a convolution between a sequence $[y_1, \ldots, y_N]$ and a set of weights, or kernel, $[w_1, \ldots, w_M]$, is the same as a correlation between $[y_1, \ldots, y_N]$ and $[w_M, \ldots, w_1]$; in machine learning parlance correlations between two sequences is referred to as "convolution"; a justification for this might be that weight vectors are found through training and so any specific initial indexing is arbitrary—if we index the weights $N, \ldots, 1$ then a convolution is the same as correlation with indices $1, \ldots, N$.

An explicitly shift invariant architecture that learns local features first, and then recognizes spatial or temporal patterns by combining those local features, is the convolutional neural network (CNN) [19], [20] which consists of a number of layers, a number of convolutional neurons in each layer (the same as the number of outputs of that layer), a number of weight filters, or kernels, (total number of filters in each neuron equals the number of neurons in the previous convolutional layer), a bias vector for each convolutional neuron, and a *flattening layer* to produce scalar outputs in the output layer. Many convolutional networks designed to work on images include a *pooling layer* to perform down sampling after a chosen convolutional layer; the down sampling is sometimes achieved through convolution with strides greater than 1, when a large input into one layer produces a smaller input to the next layer. It is, however, more common to use a down sampling method known as *max-pooling*.

The convolutional network architecture is believed to approximate the human visual cortex which extracts local features, e.g., horizontal edges, vertical edges, and local patterns [20]. The hierarchical viewpoint suggests the importance of multi-layer networks for pattern recognition. A CNN commonly has inputs with a depth dimension; for instance, imagery in RGB channels provide input with depth 3, or IR time series measurements in 2 frequency bands provide input with depth 2. In section 13 we will discuss a time series problem to classify six human activities based on nine measurements that provide time series input with depth 9. For time series inputs with a depth dimension, outputs of each hidden layer (before activation) are called *feature vectors*; when inputs are two dimensional images with depth, then outputs of each hidden layer (before activation) are called *feature maps*. As explained below, outputs of a convolutional hidden layer include layer biases as additive quantities.

Figure 16 illustrates the concept of a CNN for time series classification with an input depth dimension of 5, i.e., the input layer consists of five (equal length) time series $\boldsymbol{x}_k$, $1 \le k \le 5$, representing 5 different measurements for classification. There are three hidden layers with 3, 2, and 4 convolutional units (neurons), respectively, a flattening layer with input vectors $\boldsymbol{y}_k$, $1 \le k \le 4$, with a single $Q \times 1$ vector output, that leads to a single scalar y in the output layer. Each of the hidden layers have weight filters, or kernels, denoted by $\boldsymbol{w}_{ij}^{(l)}$, where $1 \le l \le 3$ is the layer number, j is the convolutional unit (neuron) in that layer whose input is denoted by index i. The output of convolutional neuron j in hidden layer $1 \le l \le 3$ before (nonlinear) activation is the feature vector for neuron j and layer l, and is denoted by the vector $\boldsymbol{v}_j^{(l)}$. The bias for a single convolutional neuron is a single scalar value which facilitates the neuron's ability to learn patterns in a predictable and consistent manner when presented with similar input data. Therefore, the bias vector $\boldsymbol{b}_j^{(l)}$ is always taken to be $\boldsymbol{b}_j^{(l)} = b_j^{(l)} \times [1, \ldots, 1]$.

The output of the (nonlinear) activation for neuron j and layer l is denoted by $\boldsymbol{y}_j^{(l)}$. The number of convolutional neurons in each layer determines the depth dimension of the layer's feature vectors and the layer's output vectors. For instance, $\boldsymbol{v}_1^{(1)}$, $\boldsymbol{v}_2^{(1)}$, and $\boldsymbol{v}_3^{(1)}$ are the feature vectors at hidden layer 1 while $\boldsymbol{y}_1^{(1)}$, $\boldsymbol{y}_2^{(1)}$, and $\boldsymbol{y}_3^{(1)}$ are the final outputs of layer 1 (after the nonlinearity).

The output vectors $\boldsymbol{v}$ of each hidden layer, namely, the layer's feature vectors, are computed by correlating the input vectors with their corresponding weight vectors, summing the resulting vectors and adding the bias term. The final outputs $\boldsymbol{y}$ of the layer are found by passing the result through the point-by-point nonlinear activation function σ, as illustrated on the bottom portion of figure 16. For instance, the output time series of the convolutional unit 2 in hidden layer 2, namely $\boldsymbol{y}_2^{(2)}$, is found from

$$\boldsymbol{y}_2^{(2)} = \sigma\left(\boldsymbol{v}_2^{(2)}\right), \;\; \boldsymbol{v}_2^{(2)} = \boldsymbol{w}_{12}^{(2)} * \boldsymbol{y}_1^{(1)} + \boldsymbol{w}_{22}^{(2)} * \boldsymbol{y}_2^{(1)} + \boldsymbol{w}_{32}^{(2)} * \boldsymbol{y}_3^{(1)} + \boldsymbol{b}_2^{(2)},$$

where "$*$" denotes correlation between the associated time series and $\boldsymbol{b}_2^{(2)} = b_2^{(2)} \times [1, \ldots, 1]$. In the rest of this section we will omit the bias term for notational convenience.

If we denote the filter (kernel) dimension at hidden layer l by M_l (which we assume to be odd), $l = 1, 2, 3$, and denote by N the dimension of each of the 5 input layer vectors, then the correlation equation is performed according to a "centered output prescription". Thus, we pad the time series $\boldsymbol{y}$ on either side by $(M_l - 1)/2$ zeros and denote the resulting $(N + M_l - 1) \times 1$ vector by $\boldsymbol{y}_0$. Next we align the filter weights $\boldsymbol{w}$ with $\boldsymbol{y}_0$ so that their first elements match and we multiply the elements 1 through M_l and sum them to obtain the first element of the correlation vector. Then we slide the filter along by one element and repeat the process to obtain a total of N correlation values; the last correlation value is found when the last element of the filter is aligned with the last element of $\boldsymbol{y}_0$. This procedure is illustrated in figure 17 for an input sequence with 4 elements and a filter with 3 elements; the input sequence is padded with one zero on either side and 4 correlation sequence values are calculated.

The correlations shown in figure 16 are, in practice, carried out as matrix multiplications with zero-padded vectors. Figure 18 shows the procedure to calculate the sum $\boldsymbol{w}_1 * \boldsymbol{x}_1 + \boldsymbol{w}_2 * \boldsymbol{x}_2 + \boldsymbol{w}_3 * \boldsymbol{x}_3$ for a set of three 4×1 input vectors, and three filters (kernels) with 3 elements each, with the final result having 4 elements. The 3×3 section of the input data that is being multiplied by the 3 elements of the weight vector is known as a receptive field. The weight matrix, at each correlation

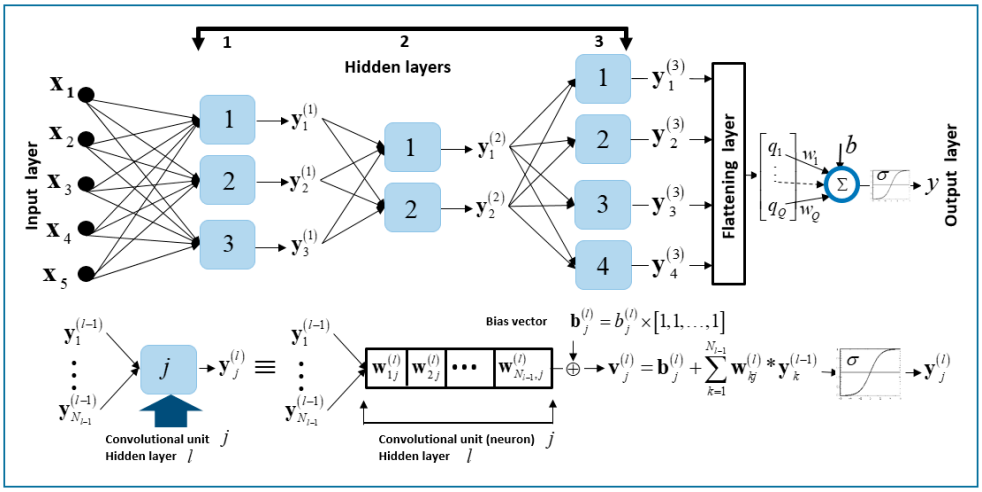


Fig. 16. A CNN with 5 time series of equal length in the input layer, 3 hidden layers, and an output layer with 1 scalar output. The bottom portion describes each convolutional unit (neuron), where $\boldsymbol{w} * \boldsymbol{y}$ indicates correlation between the two sequences $\boldsymbol{w}$ and $\boldsymbol{y}$. The flattening layer in this example produces 1 scalar output for a 2-class classification network.

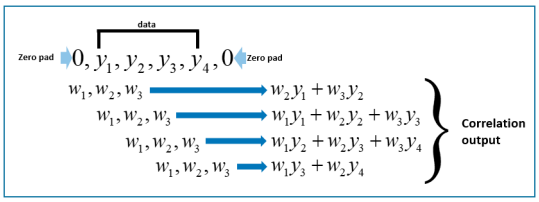


Fig. 17. Correlation procedure for an input sequence with 4 elements and a filter with 3 elements, resulting in an output with 4 elements.

lag, produces a single output, and weights in different layers provide local receptive field connections.

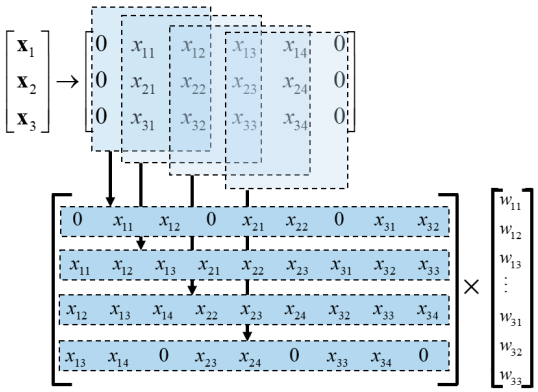


Fig. 18. Zero padding and matrix multiplication procedure to calculate the sum $\boldsymbol{w}_1 * \boldsymbol{x}_1 + \boldsymbol{w}_2 * \boldsymbol{x}_2 + \boldsymbol{w}_3 * \boldsymbol{x}_3$.

The flattening layer, in general, produces a single vector output of some length (a hyper-parameter of the model), which can then be fed into a fully connected layer with **softmax** activation for classification, or if the length of the output vector is the same as the number of classes then it can be fed directly into a **softmax** activation. One way to flatten is to simply stack all the vector inputs into a long vector; for instance, the quantities q_k, $1 \le k \le Q$ in figure 16 could be equivalent to the vector $[\boldsymbol{y}_1^{(3)}, \ldots, \boldsymbol{y}_4^{(3)}]$. Another method, known as global average pooling (GAP) [21] is to average each of the input vectors, and then producing a single vector of all the averages; for example, in figure 16, q_k, $1 \le k \le 4$, could be averages of each of the 4 feature vectors $\boldsymbol{y}_1^{(3)}, \ldots, \boldsymbol{y}_4^{(3)}$.

When two-dimensional arrays (images) are used as inputs to a CNN, then for each convolutional neuron weight matrix (filter or kernel) the corresponding images are zero-padded around their boundaries, and correlation results for all images are summed and added to a bias array (image of the same dimensions) to produce an output image (also known as feature maps—see later in this section). Figure 19 shows two 5×5 images being correlated with two distinct 3×3 weight matrices (filters or kernels) and the correlation results are added to produce the 5×5 output image (we are neglecting the additional 5×5 bias image). The two images in figure 19 might represent input with a depth of 2 at the input layer of a neural network while the weight matrices might be those associated with one of the neurons in the first hidden layer. Alternatively, the two images might be two feature maps (outputs of a hidden layer) going into a neuron in the next hidden layer.

Figure 20 shows an input of depth 3 RGB images into a CNN. The first hidden layer has 5 convolutional neurons, each of which has 3 (the same number of input image depth) weight matrices (kernels) of size 3×3. There are, therefore, 5 feature maps, i.e., 5 image outputs from the hidden layer, whose size will be the same as the input image size for unit stride, or smaller if strides greater than 1 are used. Let us denote a feature map by $F_{ij}^{(d)}$, $d = 1, \ldots, 5$. Given the input image matrices R_{ij}, G_{ij}, B_{ij}, and weight matrices (kernels) by $w_{ij}^{(d,n)}$, $n = 1, 2, 3$, then we have

$$F_{km}^{(d)} = \sum_{i,j=0}^{2} \left(w_{ij}^{(d,1)} R_{i+k,j+m} + w_{ij}^{(d,2)} G_{i+k,j+m} + w_{ij}^{(d,3)} B_{i+k,j+m} \right),$$

where $k, m = 0, \ldots$ refer to the elements of each feature map.

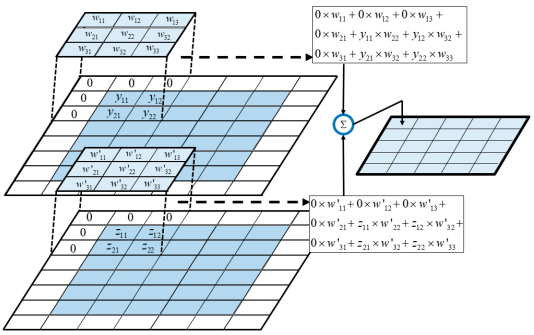


Fig. 19. Zero padding 5×5 images and correlating with two 3×3 filter matrices.

The outputs of a hidden layer are, of course, the feature maps plus layer biases that will form the input to a nonlinearity.

If the input images are square with dimension $N \times N$, weight matrices have odd dimension $M \times M$, and correlations are performed with stride $S = 1$ and padding with $(M-1)/2$ zeros, then $k, m = 0, \ldots, N-1$ and feature maps have dimension $N \times N$; if correlation stride $S > 1$ then feature maps have dimension $((N-1)/S+1) \times ((N-1)/S+1)$ - figure 20 is drawn to indicate $S > 1$ (feature map sizes are smaller than input images).

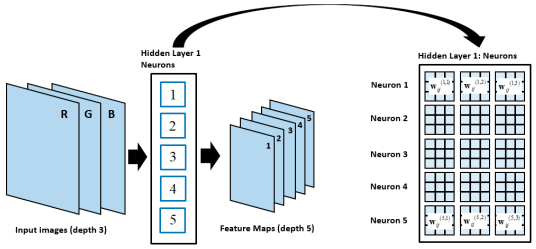


Fig. 20. Input layer images with depth 3, first hidden layer with 5 convolutional neurons, and 5 feature maps - box on the right shows a detailed picture of all the weight matrices for all 5 neurons.

Figure 21 describes the sum of correlations of the three weight matrices of the first neuron with the three input images to produce feature map 1: if the 3×3 squares below the three weight matrices $\boldsymbol{w}^{(1,1)}$, $\boldsymbol{w}^{(1,2)}$, $\boldsymbol{w}^{(1,3)}$, are denoted by $\tilde{\boldsymbol{R}}$, $\tilde{\boldsymbol{G}}$, and $\tilde{\boldsymbol{B}}$ (portions of the appropriately zero-padded matrices), respectively, then the sum of simple dot products, namely, $\boldsymbol{w}^{(1,1)} \cdot \tilde{\boldsymbol{R}} + \boldsymbol{w}^{(1,2)} \cdot \tilde{\boldsymbol{G}} + \boldsymbol{w}^{(1,3)} \cdot \tilde{\boldsymbol{B}}$ produces the indicated element of the feature map 1 matrix.

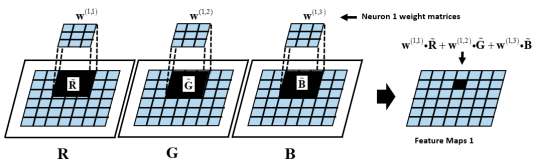


Fig. 21. Feature map 1 generated from stride 1 sliding correlation values obtained by sum of dot products of neuron 1 weight matrices with correspondingly aligned 3×3 image sections (white sections around RGB images indicate zero padding).

In image classification applications a weight matrix (kernel) may be thought of as being matched to some particular feature of the input image. For example, two matrices below show sensitivity to horizontal structures and to vertical structures, respectively; i.e., using the first matrix in an input to one layer will emphasize horizontal features of that image in the input to the next layer.

$$\text{Horizontal structures} \leftrightarrow \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\text{Vertical structures} \leftrightarrow \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

13. Time Series Classification With A Convolutional Neural Network

Let us consider the multi-class classification problem of Human Activity Recognition (HAR) data set from the University of California at Irvine (UCI) machine learning repository [22]. Data consists of $10,299$ instances of triaxial accelerometer and gyroscope time series measurements representing six activities, namely, walking, walking upstairs, walking downstairs, sitting, standing, and laying. Each activity has 9 associated 128-point time series: body acceleration, body gyro, and total acceleration, for all three axes. Figure 22 shows data for one instance of three of the most dissimilar of the 6 activities, namely, walking, sitting and standing; thus, the inputs to a classification CNN are 9 time series (3 sets of triaxial measurements).

An example CNN to classify the six activities of the HAR data is depicted in figure 23. It consists of an input layer of nine 128×1 vectors representing the accelerometer and gyro measurements, four hidden layers, a flattening layer with 6 outputs that go through a **softmax** nonlinearity to produce the final 6 class probabilities for classifying the six activities. Hidden layer 1 has 32 convolutional neurons that each have 9 filters with 3 elements each. Hidden layers $2, 3$ have 32 convolutional neurons that each have 32 filters with 3 elements each (recall that the number of filters in each convolutional neuron is the same as the number of input vectors into the layer). Hidden layer 4 has 6 convolutional neurons (to match the number of classes) that each have 32 filters with 3 elements each.

The GAP flattening layer has 6 scalar outputs that are passed through a **softmax** nonlinearity to produce the final output layer of 6 scalar values representing the class probabilities. Training was performed on $7,352$ instances and validation was done on the remaining $2,949$ instances.

Figure 24 shows the confusion matrix for the classification of the six activities, together with a table summarizing the results. *precision* is the probability of correct classification; it is calculated by dividing each diagonal number by the sum of the elements of the row passing through that diagonal element. *recall* is the probability of correct prediction and is found by dividing the diagonal element by the sum of the elements of the column through that element. *F1 score* is the harmonic mean of precision and recall, and *support* is the number

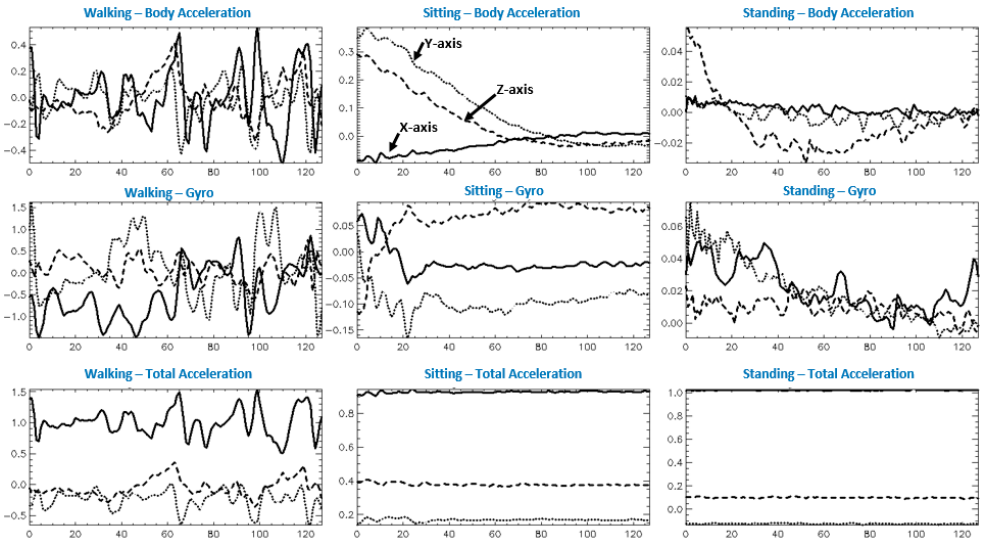


Fig. 22. Vector time series (three spatial axes) for three of six activities of the HAR data from UC Irvine: body acceleration, gyro, and total acceleration for Walking, Sitting, and Standing.

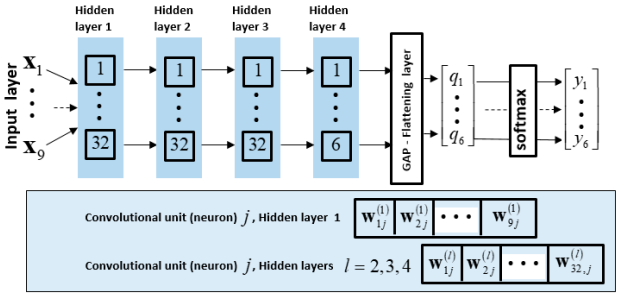
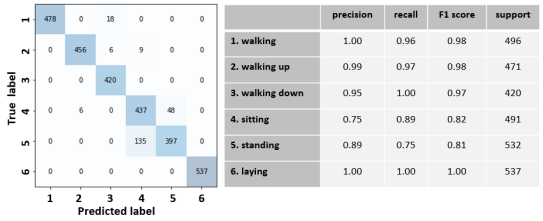


Fig. 23. CNN to classify six activities of the HAR data set.



	precision	recall	F1 score	support
1. walking	1.00	0.96	0.98	496
2. walking up	0.99	0.97	0.98	471
3. walking down	0.95	1.00	0.97	420
4. sitting	0.75	0.89	0.82	491
5. standing	0.89	0.75	0.81	532
6. laying	1.00	1.00	1.00	537

Fig. 24. Summary of results for CNN classification of HAR data.

of instances of each activity used for validation (with each instance associated with nine 128×1 vectors). The model has $7,506$ parameters with an overall accuracy of 95.24%, which compares well with the best reported result of 96.7% achieved using deep recurrent neural networks (with an unknown number of parameters). A set of 561 hand-engineered features derived from the time series (a reduction of more than 50% from the 9×128 time series values) have been classified using a logistic regression network with an overall classification accuracy of 96.2%, while a fully connected network with 3-hidden layers and $\approx 350,000$ parameters achieved 96% overall accuracy using the same 561 features. It is possible to improve the CNN accuracy by using more complicated architectures known as *ResNets*, i.e., fully convolutional networks with residual connections, to achieve 96.9% overall accuracy, but at the cost of increasing the network parameters to $\approx 1,000,000$.

14. Image Classification With A Convolutional Network

The MNIST (Modified National Institute of Standards and Technology) data [23] consists of hand-written gray-scale 28×28-pixel images of digits from 0 to 9, ten samples of which are shown in figure 25.

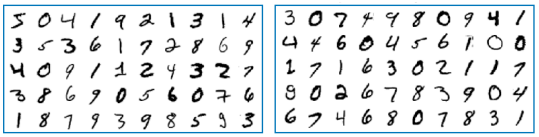


Fig. 25. Samples of hand-written images from MNIST data.

We used a CNN to classify hand-written digits consisting of 4 hidden layers, a flattening layer with 10 outputs that go through a **softmax** nonlinearity to produce the final 10 class probabilities. Hidden layer 1 has 32 convolutional neurons that each have a single filter, while layers $2, 3$ have 32 neurons each of which has 32 filters. The last hidden Layer 4 has 10 neurons with 32 filter each. Hidden layer 4 has 10 convolutional neurons that each have 32 filters; all filters have size 3×3. Figure 26 shows the CNN that we used. We chose a mini-batch size of 2^5 which corresponds to $1,875$ weight updates

for each epoch, and we trained the network for 25 epochs. This network had a total of 19,358 trainable parameters and achieved 97.8% accuracy on the test set. More sophisticated network architectures with considerably more parameters have achieved accuracies in excess of 99%.

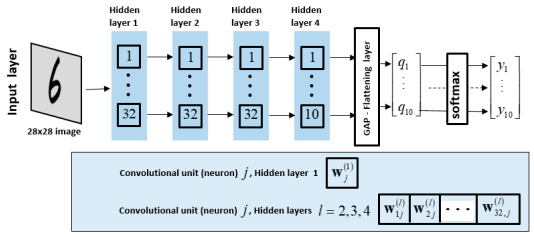


Fig. 26. CNN to classify the MNIST hand-written images of digits $0-9$.

Figures 27, 28, 29 and 30 show the activation outputs for convolutional layers 1 through 4.

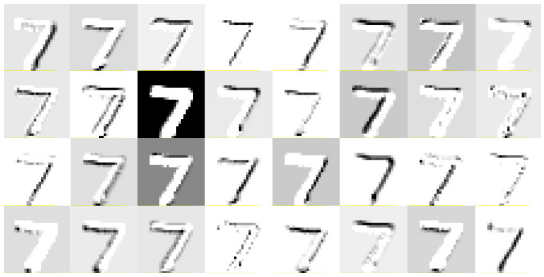


Fig. 27. Activation outputs for class label 7 for convolutional layer 1.

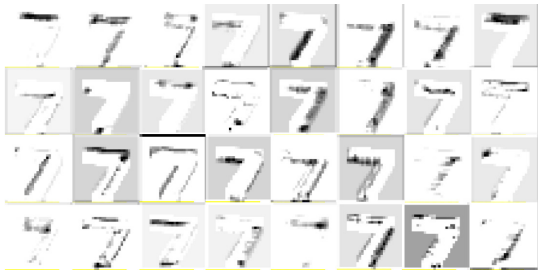


Fig. 28. Activation outputs for class label 7 for convolutional layer 2.

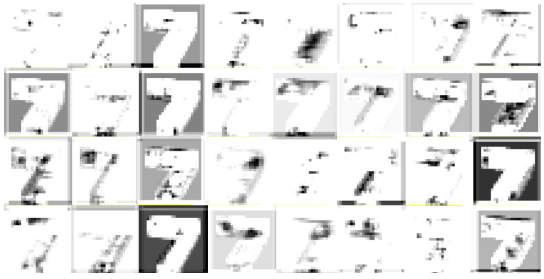


Fig. 29. Activation outputs for class label 7 for convolutional layer 3.

Fig. 30. Activation outputs for class label 7 for convolutional layer 4.

Earlier layers tend to detect more primitive aspects of the image and subsequent layers tend to learn more complex features. The final convolutional layer learns a representation that maximizes the probability of class membership. For instance, in figure 30, the image that looks more like the correct class (in this case, the number 7) is, in fact, produced by neuron 8. Note that the neuron number corresponds to the actual digit plus one, since classes are numbered 0 to 9.

15. Recurrent Neural Networks (RNN)

The neural networks considered so far have been static, i.e., for a given input vector, they produce a single output, or a vector; they resemble combinatorial digital logic composed of gates, and are incapable of keeping track of passage of time. Recurrent neural networks (RNN) [24], like digital circuits with memory, have an internal state which holds memory of previous values; these together with the current input value are used to make a decision. They are particularly useful for tasks such as speech recognition or connected handwriting recognition, when modeling data sequences that depend on previous values, and they can be combined with convolutional layers to extend the effective neighborhood of pixels.

Consider a recurrent neuron (RN) as depicted on the left hand side of figure 31: at time step t it receives an input $\boldsymbol{x}_t$ (that we assume to be a vector with length d), in addition to the scalar valued hidden state output of the previous step h_{t-1}. The final output of the RN y_t at time step t is found from its hidden state h_t through a simple feed forward neuron with a scalar weight w_{hy} (the subscript indicates "hidden to y") and scalar bias b_y,

$$y_t = w_{hy} h_t + b_y .$$

The behavior of an RN, including only the input and the hidden state, can be understood by unrolling it through time as shown on the right hand side of figure 31, which also introduces the notation for a summation unit followed by an activation function. If we introduce a "x to hidden" weight vector $\boldsymbol{w}_{xh}$ of dimension d and a scalar "hidden to hidden" weight w_{hh} then the output of the recurrent neuron at time step t is

$$h_t = \sigma\left(\boldsymbol{w}_{xh}^T \boldsymbol{x}_t + w_{hh} h_{t-1} + b\right),$$

where b is the recurrent neuron bias and σ denotes the activation function. Note that the weights and bias are shared at all time steps, i.e., they do not change as a function of time step t. The hidden state h_t at one time step t is often the only quantity of interest, and so figure 31 shows a scalar valued hidden state on the left hand side. Clearly, the process of

calculating h_t yields all previous hidden states $\ldots, h_{t-2}, h_{t-1}$ too. Putting all these values into an $N \times 1$ vector (the number of time steps) produces an $N \times 1$ vector $\boldsymbol{h}^{[t]}$; the superscript is to prevent confusion with the notation $\boldsymbol{x}_t$ which is a $d \times 1$ vector denoting the data vector at time step t, and whose elements are not necessarily a time series. For instance, $\boldsymbol{x}_t$ could denote a collection of economic indices at time step t that are used to predict the Dow Jones industrial average y_t from the hidden state h_t. Thus, in our discussion of a single RN we consider the scalar hidden value h_t and not the vector consisting of hidden states at t and all previous time steps.

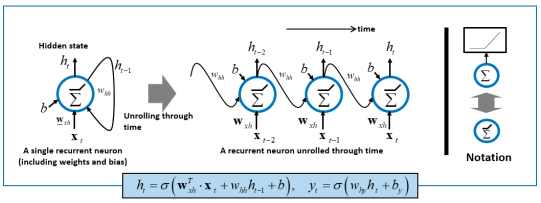


Fig. 31. An RN neuron and its unrolling through time.

The basic recurrent neuron concept can be generalized to a layer of J recurrent neurons as illustrated in figure 32. The outputs of individual recurrent neurons in each layer form the vector output of the layer, e.g., if $h_{t-1}^{(j)}$ is the scalar output of recurrent neuron j, then

$$\boldsymbol{h}_{t-1} = [h_{t-1}^{(1)}, \ldots, h_{t-1}^{(J)}]^T,$$

is the vector output of the layer at time step $t-1$, which is fed into the layer at time step t. Thus, at time step t the hidden state is

$$\boldsymbol{h}_t = \sigma\left(\boldsymbol{w}_{xh}\boldsymbol{x}_t + \boldsymbol{w}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}\right),$$

where $\boldsymbol{x}_t$ is a $d \times 1$ vector, $\boldsymbol{h}_t$ and $\boldsymbol{b}$ are $J \times 1$ vectors, $\boldsymbol{w}_{xh}$ is a $J \times d$ matrix, and $\boldsymbol{w}_{hh}$ is a $J \times J$ matrix. Each neuron in the layer has its own "x to hidden" weight vector and all these weight vectors form the rows of the full weight matrix $\boldsymbol{w}_{xh}$. Similarly, the "hidden to hidden" weight matrix is constructed from those associated with each neuron while $\boldsymbol{b}$ is the vector of individual biases for each RN in the layer. The final output of this layer is given by a feed forward neuron with the hidden state as input,

$$\boldsymbol{y}_t = \boldsymbol{w}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y,$$

where $\boldsymbol{w}_{hy}$ has dimension $J \times J$ and $\boldsymbol{y}_t$ and $\boldsymbol{b}_y$ have dimension $J \times 1$.

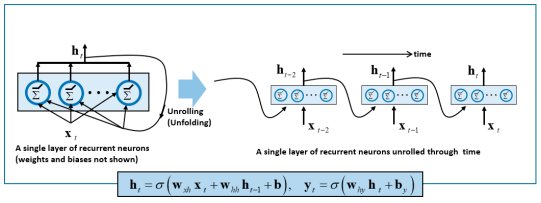


Fig. 32. A layer of recurrent neurons and its unrolling through time.

Recurrent neural networks are used in some distinct architectures as illustrated in figure 33; here we show the final output of each RN without the intermediate hidden state, i.e., each box in the figure indicates an RN layer together with a final feed forward NN to produce the output $\boldsymbol{y}_t$:

- (a) shows the basic configuration in which a sequence of input vectors, in this case a sequence of 4 elements, produces a sequence of output vectors. This architecture can be used, for example, to reproduce or predict a time series.
- (b) shows a configuration with a sequence of input vectors and only a single output. This might be used, for example, in a time series prediction application, where the input is, say, a sequence of market related information and the output is the 1-day prediction of a particular stock value.
- (c) shows a sequence with a single input and a sequence of outputs. An application might use an image as input image, to produce a sequence of words composing a caption for the image.
- (d) has a sequence of inputs and a sequence of outputs, but there is a delay between input and output. This configuration is referred to as a decoder. A suggested application is language translation, in which the input and output are sequences of words. This architecture delays the producing of outputs, i.e., the translation, until a few words have been processed.

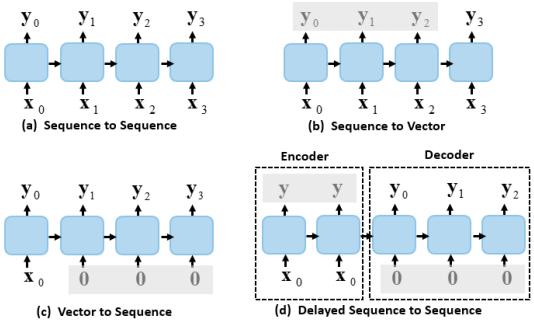


Fig. 33. Different recurrent network architectures.

The key to training a RNN is to unroll through time and then apply backpropagation in the usual way [25]. For instance, consider the delayed sequence to sequence structure shown in figure 34 (once again, we show the final output and not the hidden states). At time step t training data $\boldsymbol{x}_{t-4}$, $\boldsymbol{x}_{t-3}$ (or a mini-batch at that time) are presented, resulting in outputs $\boldsymbol{y}_{t-2}, \boldsymbol{y}_{t-1}, \boldsymbol{y}_t$, based on the current weights and biases (dotted lines). Once the cost function for the current data is computed, its gradient with respect to the weights and biases is backpropagated (solid lines) through all neurons that influence the outputs (in the present example, the neuron with input $\boldsymbol{x}_{t-4}$).

Recurrent neural networks are particularly susceptible to the vanishing gradient problem: states that are too far from the current state contribute nothing to the learning, yet the network must learn long-term dependencies in the data. A solution to this problem is based on the concept of a Long Short-Term Memory (LSTM) recurrent neuron [26] with a $d \times 1$ input $\boldsymbol{x}_t$ and hidden state vector $\boldsymbol{h}_t$.

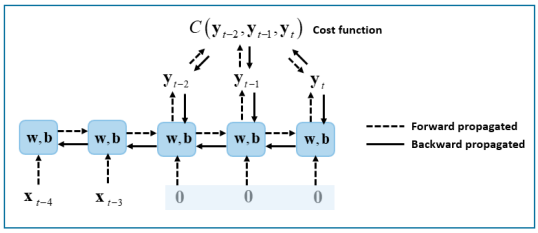


Fig. 34. Data flow for backpropagation training of an RNN.

The central idea to remembering inputs over a long time is that of a gated cell state (or gated memory unit) $\boldsymbol{c}_t$ which contains all the information up to time step t; it is gated so as to give the cell the ability to store information (opening the gate) or deleting it (closing the gate). An LSTM RN has additional gates that control the flow of data to update the cell state, i.e., how old memory and new memory are to be combined. The gates are: input, input modulation, forget, and output. The full operation of an LSTM neuron, illustrated in figure 35, is described by:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{w}_{xi}\boldsymbol{x}_t + \boldsymbol{w}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i),\\
\boldsymbol{g}_t &= \tanh(\boldsymbol{w}_{xg}\boldsymbol{x}_t + \boldsymbol{w}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g),\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{w}_{xf}\boldsymbol{x}_t + \boldsymbol{w}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f),\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{w}_{xo}\boldsymbol{x}_t + \boldsymbol{w}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o),
\end{aligned}$$

each defined with their own weight matrices and bias vectors. The quantities $\boldsymbol{i}_t$, $\boldsymbol{g}_t$, and $\boldsymbol{f}_t$ are computed first, and together with the previous cell state $\boldsymbol{c}_{t-1}$ are used to obtain the present cell state which together with $\boldsymbol{o}_t$ is used to compute the LSTM hidden state output, as illustrated in figure 35,

$$\boldsymbol{c}_t = \boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t, \quad \boldsymbol{h}_t = \boldsymbol{o}_t \otimes \tanh(\boldsymbol{c}_t),$$

where $\otimes$ indicates element-by-element multiplication. The final **softmax** output is

$$y_t = \textbf{softmax}(\boldsymbol{w}_h^T \boldsymbol{h}_t + b_h).$$

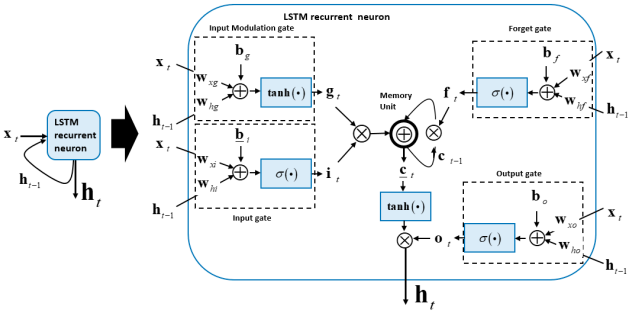


Fig. 35. A single LSTM recurrent neuron: $\boldsymbol{i}_t$ and $\boldsymbol{g}_t$ and $\boldsymbol{f}_t$, together with the previous cell state $\boldsymbol{c}_{t-1}$, are used to calculate the present cell state $\boldsymbol{c}_t$ which together with $\boldsymbol{o}_t$ produce the LSTM output $\boldsymbol{y}_t$; element by element multiplication and addition are denoted by $\otimes$ and $\oplus$, respectively. The previous output $\boldsymbol{y}_{t-1}$ and present data $\boldsymbol{x}_t$ are input to all four gates.

We now show a LSTM regression analysis using the airline passenger data [27] which is an example of a non-stationary seasonal time series. Our network consists of a single LSTM layer with $J = 300$ memory units, and feature dimension $d = 16$, i.e., we use the previous 16 values $x_{t-16}, \ldots, x_{t-1}$ to predict the present value x_t. The output of the LSTM layer is fed into a single output neuron (with hyperbolic tangent activation) and is optimized with the minimum squared error cost function. Figure 36 shows the actual passenger data, the portion used for LSTM training and the training predictions, and the validation portion of the data together with the LSTM predictions. Although the network was never exposed to the validation data (to the right of the dotted line) during training, it clearly learnt the seasonal and non-stationary characteristics of the data.

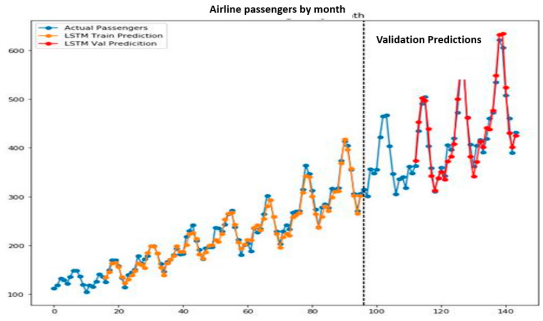


Fig. 36. LSTM prediction of airline passenger data.

More recently, gated recurrent units (GRU) have been found to have similar performance to LSTM RNNs but with fewer parameters. GRUs have two gates

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{w}_{xi}\boldsymbol{x}_t + \boldsymbol{w}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i),\\
\boldsymbol{u}_t &= \sigma(\boldsymbol{w}_{xu}\boldsymbol{x}_t + \boldsymbol{w}_{hu}\boldsymbol{h}_{t-1} + \boldsymbol{b}_u),
\end{aligned}$$

and

$$\begin{aligned}
\hat{\boldsymbol{h}}_t &= \tanh(\boldsymbol{w}_{xh}\boldsymbol{x}_t + \boldsymbol{w}_{hh}\ \boldsymbol{h}_{t-1} \otimes \boldsymbol{i}_t + \boldsymbol{b}_h),\\
\boldsymbol{h}_t &= (1 - \boldsymbol{u}_t) \otimes \boldsymbol{h}_{t-1} + \boldsymbol{u}_t \otimes \hat{\boldsymbol{h}}_t.
\end{aligned}$$

16. Unsupervised Learning

All neural networks described so far fit the *Supervised Learning* category, i.e., networks are trained to learn a function that maps input data to associated output labels with the goal of generalizing to new samples. The output labels act as a guide for training, i.e., we minimize the difference between actual output labels and estimated output labels. Thus, backpropagation learning is used to train neural networks with desired output labels corresponding to every training input.

Interestingly, there are neural network applications that do not need output labels corresponding to input data. Such applications of neural networks fall under the category of *Unsupervised Learning* [28]. While less common than supervised networks, unsupervised networks are used in a variety of applications. In this section we use unsupervised neural networks for data compression, reconstruction, and generative modeling. We demonstrate these capabilities with a class of neural network models known as *autoencoders* [29].

The basic configuration of an autoencoder is shown in figure 37. The training data is presented as both input and output data, i.e., the input data act as "targets" for the network, and the network is trained to reproduce the input data as accurately as possible. This raises the question of the usefulness of such a network if the output is essentially the same as the input. The key to the utility of the autoencoder is in the hidden layers. Typically, as suggested by figure 37, the hidden layers become successively smaller as the inner-most hidden layer, often called the *bottleneck layer*, is approached; the bottleneck layer corresponds to the layer with the lowest dimension in the network. Having fewer neurons to work with in each successive layer, the neural network learns to find an efficient representation of the data, i.e., it learns to perform data compression. The successive decrease in dimensionality of the hidden layers forces the network to send maximally useful information through the bottleneck layer since the network is tasked with reconstructing the input data as accurately as possible. Thus, the bottleneck layer represents the input data in an optimally compressed form and in much smaller dimension than the original input dimension.

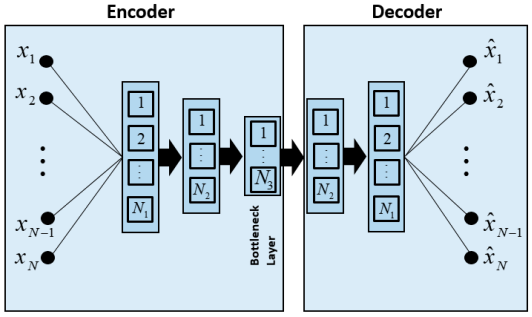


Fig. 37. Basic autoencoder configuration with input $N \times 1$ data vector. The encoder portion has three hidden layers with $N_1 > N_2 > N_3$ neurons, with the last hidden layer known as the bottleneck layer. The decoder portion has two hidden layers with N_2 and N_1 neurons, respectively. The output vector has the same dimension N as the input vector.

For instance, consider the MNIST data set of 28×28 images of digits $0-9$, each of which is represented as a vector $\boldsymbol{x}_k$ with 784 elements. Since the images represent 10 digits, it seems unlikely that there are 784 independent dimensions in the data set. Although there are a number of swoops, loops, and strokes in all the digits, it seems reasonable to expect to represent the data with a smaller number of dimensions than 784. For example, we could train an autoencoder whose bottleneck layer has 100 neurons. If it turns out that this autoencoder is able to reproduce the input data accurately, then we say that the intrinsic data dimension is closer to 100 rather than the ambient dimension of 784. Exploring with different number of neurons in the bottleneck layer, a sense of the intrinsic dimension of the original data can be obtained.

We may view an autoencoder as a model with two distinct pieces as illustrated in figure 37: the layers from the input to the bottleneck layer form the *encoder* while the layers after the bottleneck layer, all the way to reconstructed output layer, form the *decoder*. The encoder encodes the input data by embedding it into a new *latent space* whose dimension is close to the intrinsic dimension of the data. The encoder output, often referred to as *latent code*, is then passed into the decoder for reconstruction. The process is exactly the same as data compression and reconstruction but now we view the latent code $\boldsymbol{z}$ from a different perspective that will show its utility as more than a compressed representation of the original input $\boldsymbol{x}$: if we knew the distribution function $f(\boldsymbol{z})$ of the latent code $\boldsymbol{z}$, then we could generate data with the same (unknown) distribution of the input by simply feeding into the decoder realizations from the distribution $f(\boldsymbol{z})$. We shall see how this can be accomplished using a form of unsupervised learning known as a generative model. Whereas in discriminative models (supervised learning) networks learn the conditional probabilities $P(\boldsymbol{y}|\boldsymbol{x})$, where $\boldsymbol{y}$ is a set of classification labels, generative network models learn the joint probability density function $f(\boldsymbol{x})$ of the input vector $\boldsymbol{x}$. Thus, generative models allow us to create data realizations $\boldsymbol{x}$ once the joint density $f(\boldsymbol{x})$ has been learnt. The ability to create unlimited data realizations from a comparatively small sample is of enormous significance in data analysis when actual data being modeled is difficult to obtain because of processing or acquisition constraints.

17. Generative Adversarial Networks

A flexible class of generative models is known as Generative Adversarial Networks or GAN(s) [30] which typically consist of two competing neural networks, the generator **G** and the discriminator **D**, in a zero-sum game (a game in which the algebraic sum of all participants' gains and losses equals zero) [31] where each neural network has objectives counter to the other.

The generator network **G** has the goal of generating "fake" samples $\boldsymbol{x}_f$ that accurately mimic realizations from the distribution function $f(\boldsymbol{x})$, while the goal of the discriminator network **D** is to differentiate between "genuine" data realizations $\boldsymbol{x}$ and generated ones $\boldsymbol{x}_f$, and ultimately reject the fake realizations. Thus, **G** tries to generate fake samples from the true distribution $f(\boldsymbol{x})$ that look sufficiently genuine that **D** fails to reject them. The discriminator **D**, on the other hand, has the goal of accurately differentiating between generated samples $\boldsymbol{x}_f$ and genuine ones $\boldsymbol{x}$. The two models "learn" by minimizing their respective losses: the generator **G** minimizes its loss when it can successfully "fool" the discriminator **D** into classifying one of its generated samples as a genuine sample, while the discriminator **D** minimizes its loss when it can correctly reject the generated samples.

Figure 38 illustrates the GAN architecture in more detail; the generator and discriminator networks can be fully connected or convolutional. GANs have two distinct modes of training to facilitate the zero-sum game between **G** and **D**: the Generator and the Discriminator train in sequence and each network attempts to minimize its own loss. Note that the loss is always calculated at the output of **D** but the calculation could use different functions depending on the network that is being trained. Typical losses used to optimize GANs are Wasserstein and least-squares loss functions [32], [33].

In the first training phase, **D** attempts to minimize its error (e.g., cross entropy, minimum squared error (MSE), etc.) by

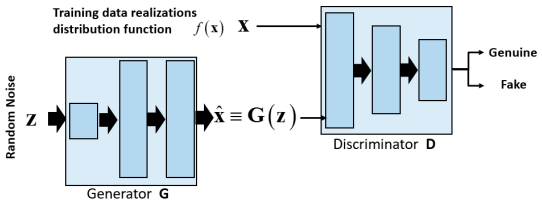


Fig. 38. A typical Generative Adversarial Network (GAN). Each rectangle represents a hidden layer of arbitrary size except for: the output layer of **G** which must match the dimension of $\boldsymbol{x}$, and the output of **D** which must allow for binary classification (i.e., single sigmoid node or two **softmax** nodes).

discriminating between the output of **G** and the real samples $\boldsymbol{x}$. Initially **D** easily rejects the output of **G** since **G** has not updated its weights; it produces samples that look like random noise. In the next training phase we optimize **G** so that it can generate samples that **D** will fail to classify as fake samples. During this phase the samples generated by **G** are sent to **D** for classification and the resulting cost is used to update the weights of **G** via the backpropagation algorithm we have described in section 5. This gives **G** a chance to improve itself in order to fool **D** more effectively, but now **D** needs to improve itself in order to deal with the better samples now being generated by **G**. The training phases alternate allowing **G** and **D** to compete until their respective losses stabilize and **G** produces samples that are sufficiently good that **D** fails to reject them. At this point **G** can generate samples that mimic the training data and can be used purely as a generative model.

It is important to note that when optimizing **D** the label on the output of **G** is "fake" and the label on true samples is "genuine", but when optimizing **G** the label on the output of **G** is "genuine" and so is the label on $\boldsymbol{x}$ — this is necessary in order to gauge how *genuine* the fake samples look to **D**, as shown in figure 39.

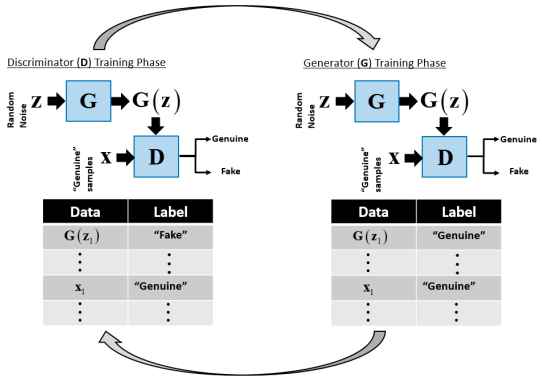


Fig. 39. The two phases of the GAN training process.

GANs are difficult to train; they are typically very sensitive to hyper-parameter tuning [34]. While progress has been made in this regard often GANs converge to solutions that are not very good, or can diverge altogether. Sometimes when GANs do converge, they do so in ways that do not accurately model the underlying distribution of the data. One of the most common issues encountered in GAN training is *mode collapse* [34] when the generator learns to only generate a single (or near single) mode in an underlying multi-modal distribution. Essentially, GANs can arrive at solutions that maximally fool the discriminator (i.e. images look like sufficiently realistic samples from $\boldsymbol{x}$ when evaluated by **D**) but lack the actual variation present in the underlying data. If we consider the MNIST digits, this would be the equivalent of a generator **G** learning to only generate realistic 1's and 7's leaving all the other digits unrepresented, regardless of the input $\boldsymbol{z}$ into **G**.

Given the above issues with GAN training we wish to learn a generative model that avoids them. Autoencoders seem to be a good class of models since they explicitly model the entire data and converge using modern training techniques. Although "vanilla" autoencoders are not generative models, they can be sufficiently enhanced with ideas from the GAN paradigm to turn them into generative models. This class of unsupervised models is called adversarial autoencoders or AAE. AAEs provide the generative capabilities of GANs with the training stability and convergence properties of autoencoders, i.e., they explicitly control for mode collapse [35].

Figure 37 shows how an autoencoder can be viewed as the combination of two distinct parts: an encoder **enc** and a decoder **dec**. The encoder takes some input vector $\boldsymbol{x}$ and outputs a latent code $\mathbf{enc}(\boldsymbol{x})$ that is fed into the decoder producing the reconstructed output $\hat{\boldsymbol{x}} \equiv \mathbf{dec}(\mathbf{enc}(\boldsymbol{x}))$. As stated above, if $\mathbf{enc}(\boldsymbol{x})$ followed some probability distribution $f(\boldsymbol{x})$, then we could generate samples from this distribution and feed them to the decoder for data generation. This is precisely what adversarial autoencoders learn to do; they learn to optimally compress and reconstruct the input while simultaneously forcing $\mathbf{enc}(\boldsymbol{x})$ to follow an arbitrary prior distribution $f(\boldsymbol{x})$ (typically a joint Gaussian distribution). Once the network converges we can sample $f(\boldsymbol{x})$ and feed that data to the decoder and generate data that mimic the variation in the training set without worrying about mode collapse.

To train an AAE the latent code representation must be regularized in such a way as to *force* it to follow a prior probability distribution. We do this by incorporating a discriminator $\mathrm{D}_{\mathrm{AAE}}$ into the autoencoder learning process. Much like regular GANs the discriminator in this setup takes two inputs. One input comes from a multi-dimensional probability distribution (which has the same dimension as the autoencoder latent code $enc(x)$) called the *real samples*, and the other input to the $\mathrm{D}_{\mathrm{AAE}}$ comes from the bottleneck layer output code $\mathbf{enc}(\boldsymbol{x})$ called the *fake samples*. To train the AAE we alternate between two training phases much like with GANs: the first phase consists of training $\mathbf{D}_{\mathrm{AAE}}$ to discriminate between samples from a real probability distribution and the latent code of our autoencoder $\mathbf{enc}(\boldsymbol{x})$. The next phase consists of training the autoencoder to perform its regular reconstruction task while simultaneously minimizing its loss with respect to how believable the latent codes it produces are to $\mathbf{D}_{\mathrm{AAE}}$. In other words, it learns to do reconstruction while ensuring that the latent code produced by the encoder follows the specified prior distribution. Figure 40 shows an example of a vanilla AAE as a traditional autoencoder coupled with a discriminator that takes $f(\boldsymbol{x})$ and $\mathbf{enc}(\boldsymbol{x})$ as inputs. Forcing the AAE to learn to reconstruct the data while producing $\mathbf{enc}(\boldsymbol{x})$ samples

that fool $\mathbf{D}_{\text{AAE}}$ is the reason models in this class are called *adversarial*; $\mathbf{D}_{\text{AAE}}$ does not want to be fooled by the output of the encoder, while the encoder tries to fool the discriminator into classifying its output $\mathbf{enc}(\boldsymbol{x})$ as coming from the real prior distribution $f(\boldsymbol{x})$.

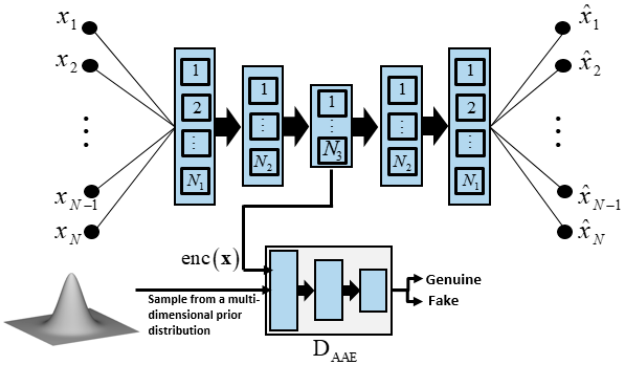


Fig. 40. Adversarial Autoencoder (AAE): an autoencoder coupled with a discriminator, trained using the same alternating phases as GANs.

We demonstrate an AAE application using the MNIST data set. Rather than using the vanilla AAE we use a supervised version that learns to disentangle "style" (variation) from "content" (class label) in the training images. In this setup we can hold a class label fixed and observe the variation present in class by sampling from our prior distribution and passing that as input to our decoder for image generation. We can also hold variation constant and observe how the factors of variation we have isolated affect the different classes in our data. To train this supervised variant of an AAE the only modification we make is to add a one-hot label to the input of our decoder; otherwise the training process is the same as with the standard AAE formulation we described above. The supervised AAE allows the generator to associate the data it reconstructs with class labels. Once the model converges we can fix the one-hot label (i.e. content input) and sample our prior distribution (i.e. style input), or vice versa, and generate new data samples. Figure 41 illustrates a supervised AAE architecture.

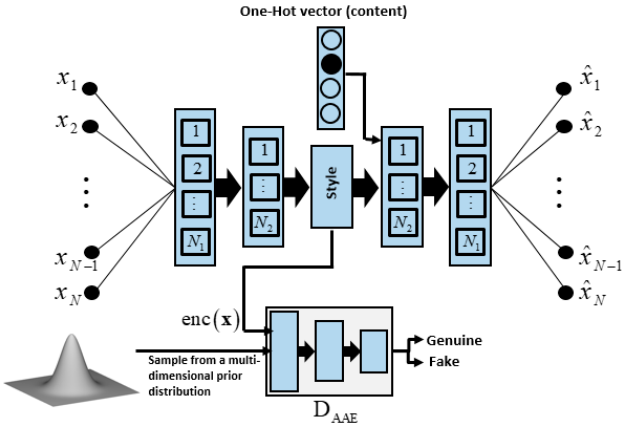


Fig. 41. Supervised Adversarial Autoencoder (AAE).

Figure 42 (a replication of the results in the original AAE paper [35]) shows the output of our trained supervised AAE that clearly demonstrates the ability of the model to separate content and style in the images.

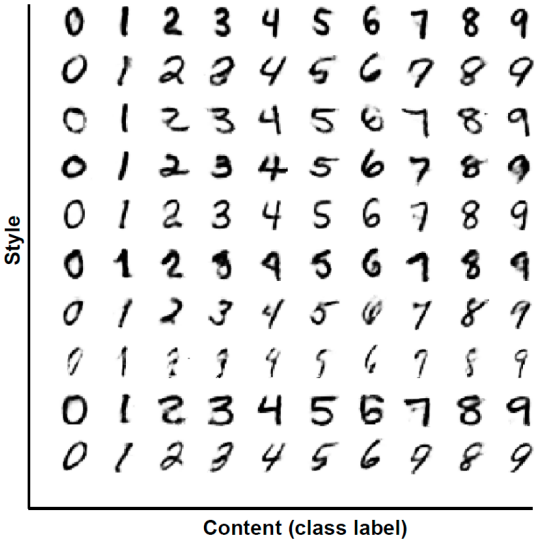


Fig. 42. Trained supervised AAE separating content and style.

References

[1] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control Signals Systems*, vol. 2, pp. 303–314, 1989.

[2] K. Hornik, M. Stichcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[3] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[4] D. Pedamonti, "Comparison of non-linear activation functions for deep neural networks on MNIST classification task," https://arxiv.org/pdf/1804.02763.pdf, arXiv:1804.02763v1 - 8 Apr 2018.

[5] S. Ruder, "An overview of gradient descent optimization algorithms," https://arxiv.org/pdf/1609.04747.pdf, arXiv:1609.04747v2 - 15 Jun 2017.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[7] P. D. Wasserman and T. Schwartz, "Neural Networks, Part 2: What they are and why everybody is so interested in them now," https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=2091, pp. 10–15, IEEE Expert, Spring 1988.

[8] M. Nielsen, "Why are deep neural networks hard to train?" http://neuralnetworksanddeeplearning.com/chap5.html.

[9] P. Chaudhari and S. Soatto, "Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks," https://arxiv.org/pdf/1710.11029.pdf, arXiv:1710.11029v2 - 16 Jun 2018.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," https://arxiv.org/pdf/1502.03167.pdf, arXiv:1502.03167v3 - 2 Mar 2015.

[11] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[12] S. K. Kumar, "On weight initialization in deep neural networks," https://arxiv.org/pdf/1704.08863.pdf, arXiv:1704.08863v2 - 2 May 2017.

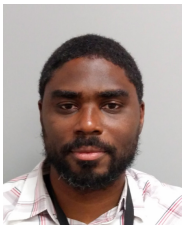
[13] P. Domingos, "A few useful things to know about machine learning," http://www.astro.caltech.edu/~george/ay122/cacm12.pdf.

[14] Y. Jiang, D. Krishnan, H. Mobahi, and S. Bengio, "Predicting the generalization gap in deep networks with margin distributions," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HJlQfnCqKX

[15] C. Shorten and T. Khoshgoftaar, "A survey on image data augmentation for deep learning," https://doi.org/10.1186/s40537-019-0197-0, 2019.

[16] R. Caruana, S. Lawrence, and L. Giles, "Over-fitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*. MIT Press, 2000, pp. 381–387.
[17] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," https://arxiv.org/pdf/1207.0580.pdf, arXiv:1207.0580v1 - 3 Jul 2012.
[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.
[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, pp. 2278–2324, 1998.
[20] K. Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biol. Cybernetics*, vol. 36, pp. 193–202, 1980.
[21] M. Lin, Q. Chen, and S. Yan, "Network in network," https://arxiv.org/pdf/1312.4400.pdf, arXiv:1312.4400v3 - 4 Mar 2014.
[22] D. Dua and C. Graff, "UCI machine learning repository," http://archive.ics.uci.edu/ml, 2017.
[23] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/.
[24] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," https://arxiv.org/pdf/1506.00019.pdf, arXiv:1506.00019v4 - 17 Oct 2015.
[25] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, pp. 1550–1560, 1990.
[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.
[27] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis: Forecasting and Control*. Hoboken, N.J.: Wiley, 2008.
[28] Z. Ghahramani, "Unsupervised learning," *ML Summer Schools — 2003 Advanced Lectures on Machine Learning*, vol. LNAI 3176, 2004.
[29] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," *JMLR: Workshop and Conference Proceedings*, vol. 27, pp. 37–50, 2012.
[30] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," https://arxiv.org/pdf/1406.2661.pdf, arXiv:1406.2661v1 - 10 Jun 2014.
[31] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton, N.J.: Princeton University Press, 1944.
[32] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," https://arxiv.org/pdf/1701.07875.pdf, arXiv:1701.07875v3 - 6 Dec 2017.
[33] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley, "Least squares generative adversarial networks," https://arxiv.org/pdf/1611.04076.pdf, arXiv:1611.04076v3 - 5 Apr 2017.
[34] N. Kodali, J. Abernethy, J. Hays, and Z. Kira, "On convergence and stability of GANs," https://arxiv.org/pdf/1705.07215.pdf, arXiv:1705.07215v5 - 5 Apr 2017.
[35] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey, "Adversarial Autoencoders," https://arxiv.org/pdf/1511.05644.pdf, arXiv:1511.05644v2 - 25 May 2016.

A.-H. Najmi completed the Mathematical Tripos at Cambridge University and received the D.Phil. degree in theoretical physics from St. Catherine's College, Oxford University. He taught at the University of Utah and conducted research in seismic imaging at Shell Oil Bellaire Research center in Houston before joining the Johns Hopkins University Applied Physics Laboratory, where he also holds a joint faculty appointment at the Johns Hopkins School of Medicine. In addition to publications in diverse areas of theoretical physics, geophysics, and biophysics, he has published a graduate textbook on wavelets (John Hopkins University Press, 2012) and a graduate textbook on advanced signal processing (McGraw Hill, 2020).



Patrick Emmanuel received his Bachelor's degree in mathematics and computer science from Palm Beach Atlantic University. He is currently pursuing a master's degree in applied mathematics from the Johns Hopkins University. He has been at the Applied Physics Laboratory since 2017 where he conducts applied deep learning research and systems development.

T. Moon received his bachelors degree summa cum laude in electrical engineering and mathematics from Brigham Young University. He completed his Ph.D. in Electrical Engineering at the University of Utah. He has been at Utah State University since 1991, where he is now professor and head of the Electrical and Computer Engineering Department. His publications in signal processing and digital communications include three textbooks in signal processing and error correction coding.