USC Viterbi
School of Engineering
*Daniel J. Epstein Department of Industrial and Systems Engineering*

Predictive Analytics (ISE529)

# Deep Learning (II)

Dr. Tao Ma
ma.tao@usc.edu

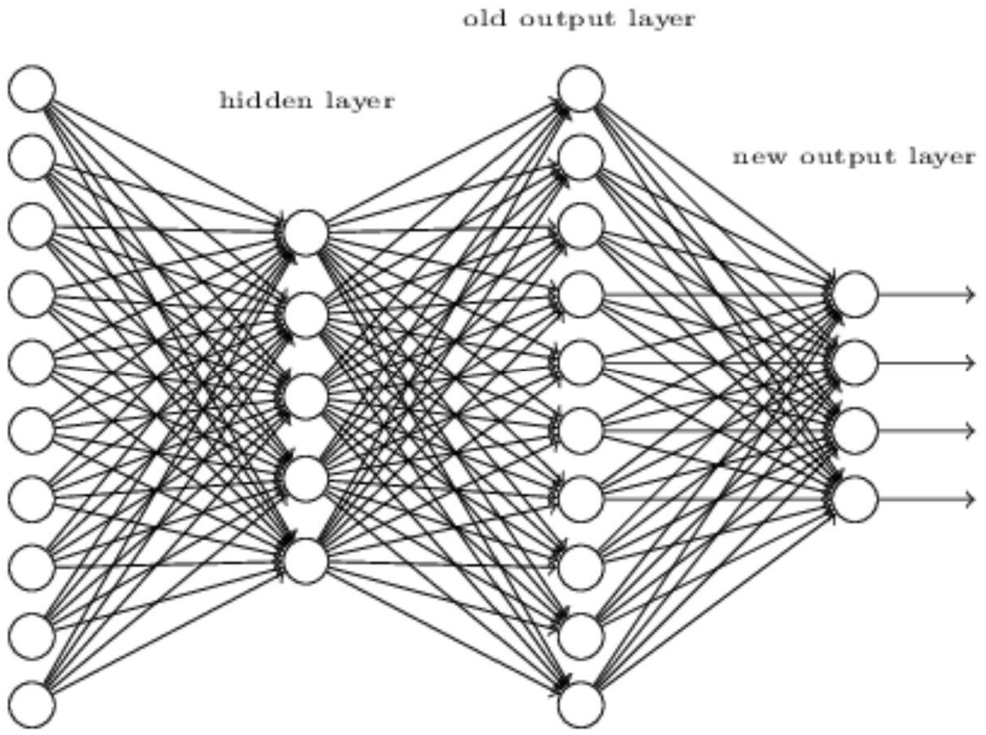
*Tue/Thu, May 22 - July 1, 2025, Summer*

USC University of Southern California

# Deep Neural Network

One input layer + many hidden layers + output layer

- Approximate complex nonlinear relationship, feature transformation
- A hidden layer is viewed as a linear model, a stack of linear models



For example: CNN, RNN, Elman, Jordan, Boltzmann, etc.

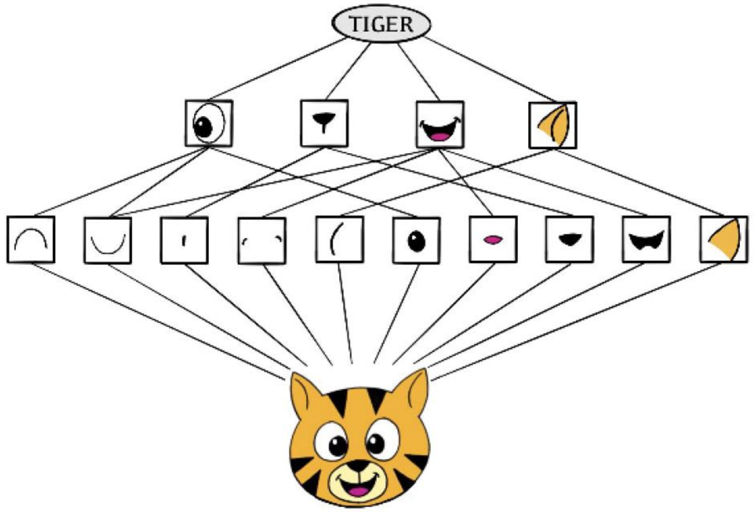
## CONVOLUTIONAL NEURAL NETWORKS

## Convolutional Neural Network

Convolutional neural networks (CNNs) are being widely used in computer vision, digit recognition, image classificationand natural language processing. The major differences between conventional neuron networks and CNNs are the size and structure. The size of a CNN can be much larger, consisting of tens of layers, each layer has tens or thousands of neurons. The layers of CNNs are not necessarily fully connected.

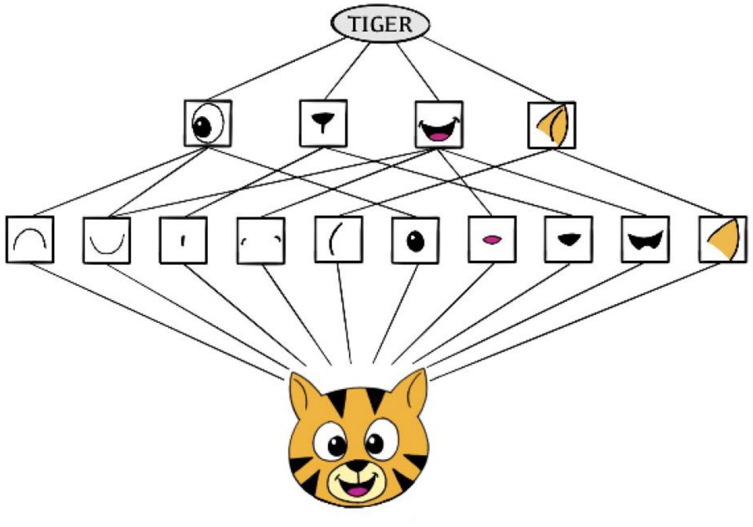
# Convolutional Neural Networks

Neural networks around 2010 with big successes in image classification. A special family of neural networks, ***Convolutional Neural Networks,*** (CNNs) has evolved for classifying images and has shown spectacular success on a wide range of problems.. CNNs mimic to some degree how humans classify images, by recognizing specific features or patterns anywhere in the image that distinguish each object class.



The network takes in the image and identifies local features. It then combines the local features (low-level features, such as small edges, patches of color) in order to create compound features (higher-level features, such as parts of ears, eyes, so on). These compound features contribute to the probability of any given output class label, such as “tiger”.

# Convolutional Neural Networks



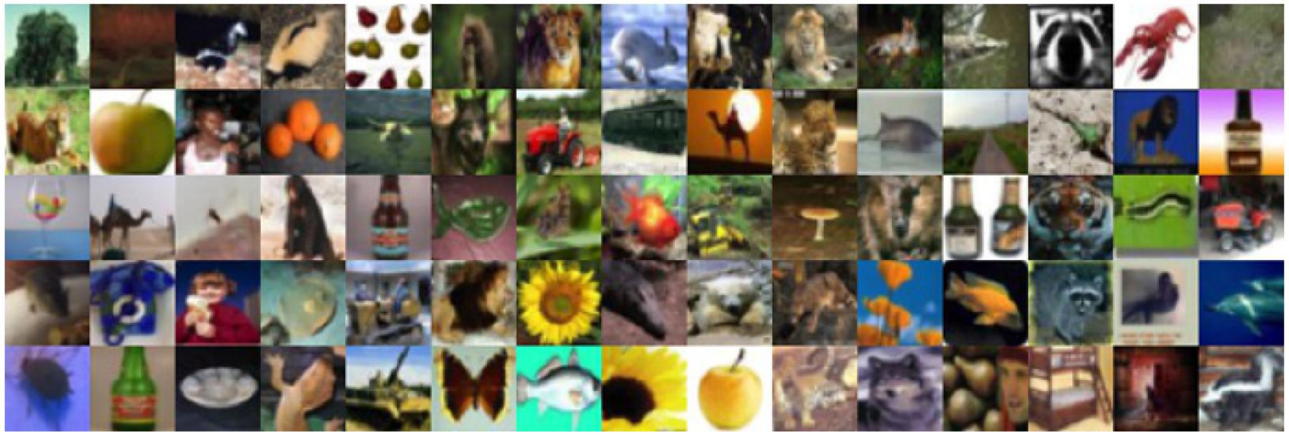
How does a convolutional neural network build up this hierarchy?

It achieves this with two specialized types of hidden layers, called **convolution** layers and **pooling** layers.

Convolution layers search for small patterns in the image, whereas pooling layers reduce dimension to select a prominent subset. Morden CNN architectures make use of many convolution and pooling layers.

# Preprocess Images to Feature Map

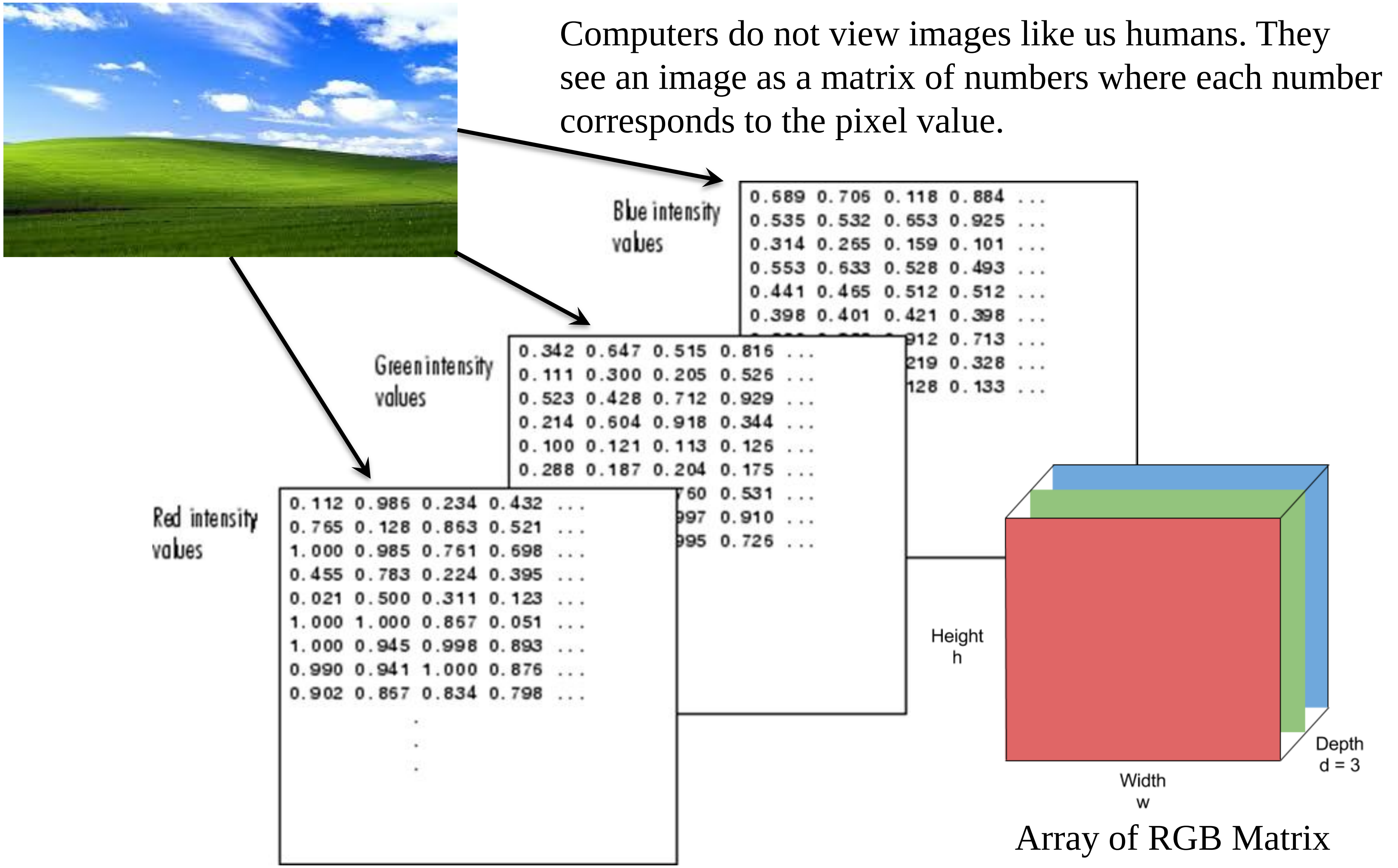
Around 2010, massive databases of labeled images were being accumulated, with ever-increasing numbers of classes, such as a well-known image database the *CIFAR100* database, which consists of 60,000 images labeled with 100 different classes.



Each image has a resolution of 32 × 32 pixels, with three eight-bit numbers per pixel representing *red*, *green* and *blue*. The numbers for each image are organized in a three-dimensional array called a **feature map**. The first two axes are spatial (both are 32-dimensional), and the third is the ***channel*** axis, representing the three colors.

# Preprocess Images to Feature Map

Computers do not view images like us humans. They see an image as a matrix of numbers where each number corresponds to the pixel value.



```
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
0.398 0.401 0.421 0.398 ...
                  912 0.713 ...
                  219 0.328 ...
                  128 0.133 ...
```

```
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
0.288 0.187 0.204 0.175 ...
             760 0.531 ...
             997 0.910 ...
             995 0.726 ...
```

```
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
            .
            .
            .
```

Array of RGB Matrix

## CONVOLUTION OPERATION

# Convolution Operation

- A convolution layer is made up via a large number of convolution **filters**. Each filter is a template that determines whether a **local feature** is present in an image.
- A convolution filter relies on a very simple operation, called a ***convolution***, which is basically equivalent to repeatedly multiplying matrix elements and then adding the results.
- Example: consider a very simple example of a 4 × 3 image:

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

- Now consider a 2 × 2 filter of the form $\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$
- Convolve the image with the filter, we get the result

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

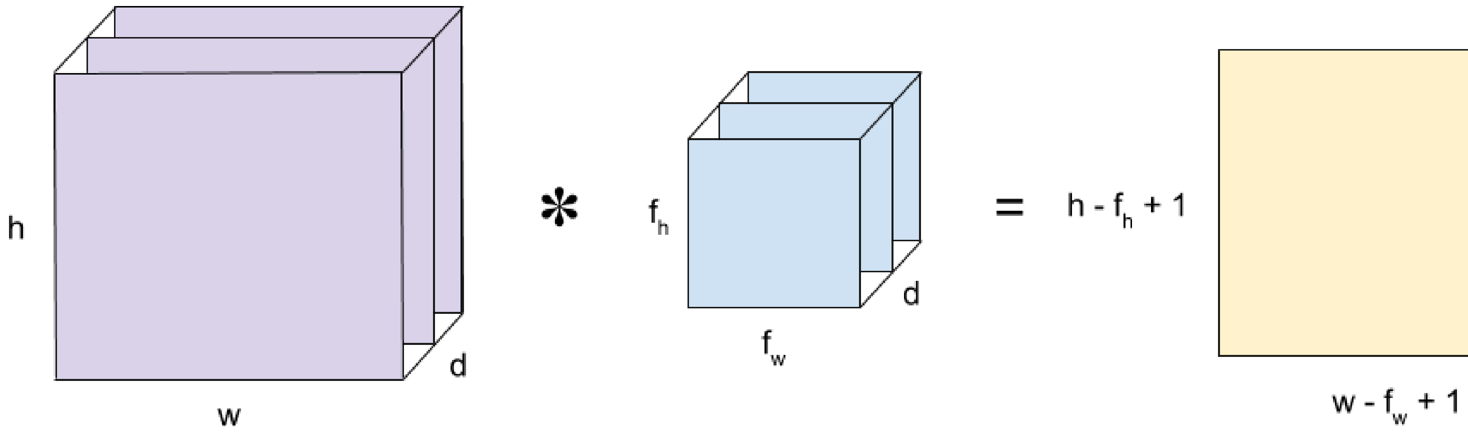
# Convolution Operation

Convolution of image is to extract features from an input image.

Convolution is a mathematical operation that takes two inputs:

- An image matrix (volume) of dimension (h x w x d)
- A filter ($f_h$ x $f_w$ x d)

and produce a "Feature Map" matrix as output (h - $f_h$ + 1) x (w – $f_w$ + 1) x 1

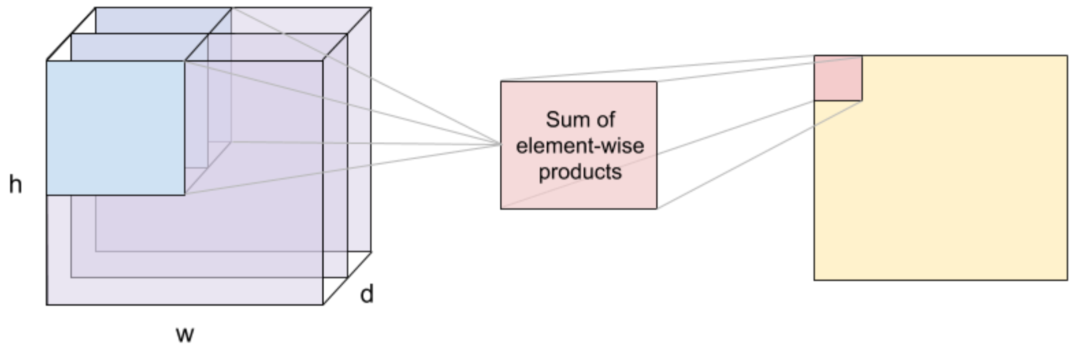


## Convolution Operation

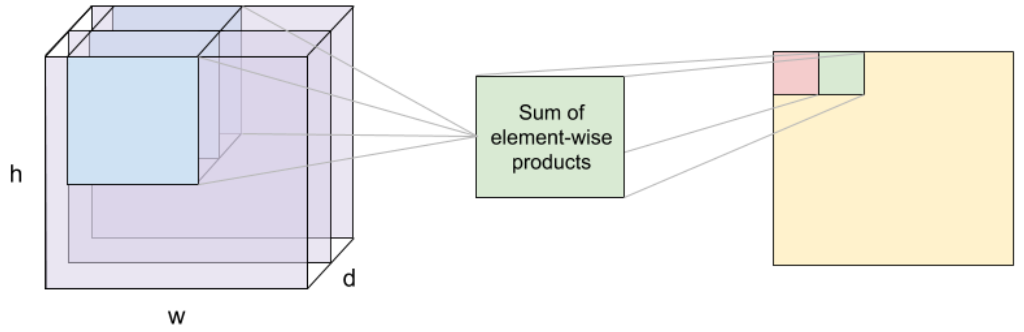
We compute the convolution as follows:

Start with the top-left pixel of the output:

1. Place the filter on the top-left corner of the input image
2. Calculate the element-wise product of each corresponding pixel value
3. Sum the products to obtain the top-left pixel value of the output



Next, shift the filter to the right by **one position** and repeat steps 2. and 3. to obtain the next pixel value of the output



# Convolution Operation

An example of the convolution of 6 x 6 image matrix is shown as below:

$$
\begin{bmatrix} 1 & 3 & 5 & 2 & 6 & 1 \\ 7 & 0 & 2 & 4 & 1 & 3 \\ 5 & 1 & 3 & 6 & 2 & 7 \\ 2 & 4 & 3 & 0 & 0 & 1 \\ 1 & 7 & 3 & 9 & 2 & 1 \\ 3 & 6 & 2 & 4 & 1 & 0 \end{bmatrix}
*
\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}
=
\begin{bmatrix} 3 & -8 & 1 & 1 \\ 6 & -5 & 5 & -1 \\ -1 & -3 & 5 & 6 \\ -2 & 4 & 5 & 11 \end{bmatrix}
$$

6 x 6 image matrix

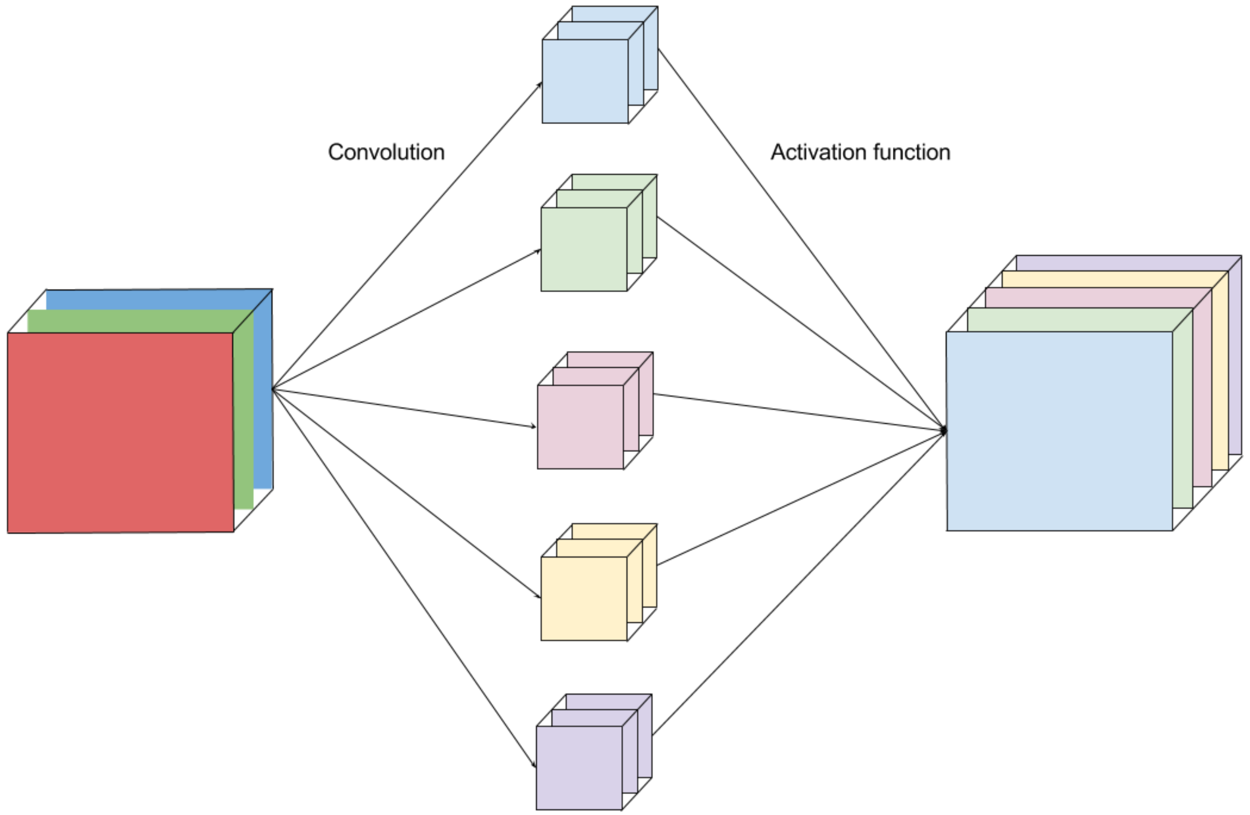
3 x 3 filter matrix

Convolved feature matrix

- We can think of the filter weights as the parameters going from an input layer to a hidden layer, with one hidden neuron for each pixel in the convolved image.
- We can think of the original image as the input layer in a convolutional neural network, and the convolved images as the neurons in the first hidden layer.

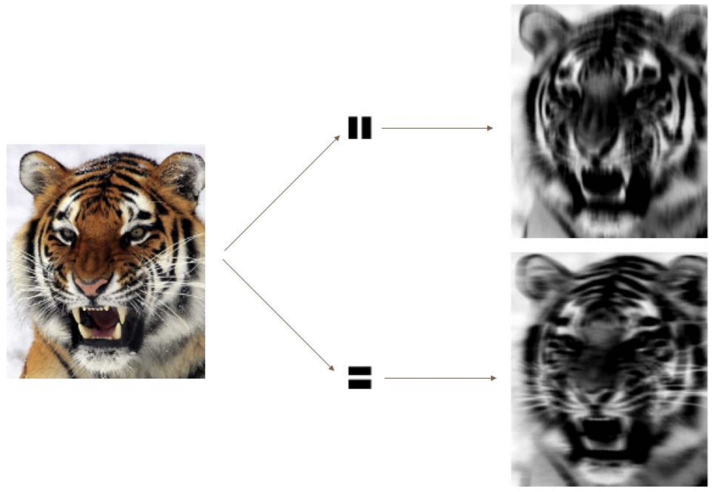
# Convolution Operation

A convolution layer is composed of *N* filters of the same size and depth. We convolve each filter with the input volume and get *N* outputs. The outputs are passed to some activation function, such as *ReLU*, for example. Finally, those *N* outputs is stacked together into a $(h - f_h + 1) \times (w - f_w + 1) \times N$ feature maps.



# Example

- If a $2 \times 2$ submatrix of the original image resembles the convolution filter, then it will have a large value in the convolved image; otherwise, it will have a small value. Thus, the convolved image highlights regions of the original image that resemble the convolution filter.
- In general convolution filters are small $\ell_1 \times \ell_2$ matrices, with positive integers that are not necessarily equal.



Convolution filters find **local features** in an image of tiger, such as edges and small shapes.

We apply the two small convolution filters in the middle. The convolved images highlight areas in the original image where details similar to the filters are found.

Specifically, the top convolved image highlights the tiger's **vertical** stripes, whereas the bottom convolved image highlights the tiger's **horizontal** stripes.

## Example

- In a convolution layer, we use a whole bank of **predefined filters** to pick out a variety of differently-oriented edges and shapes in the image.
- Some additional details:
  - Since the input image is in color, it has three **channel**s represented by a three-dimensional feature map (matrices). Each channel is a two-dimensional ($32 \times 32$) feature map — one for red, one for green, and one for blue. A single convolution filter will also have three channels, one per color, each of dimension $3\times3$, with potentially different filter weights. The results of the three convolutions **are summed to** form a two-dimensional output feature map.
  - If we use $K$ different convolution filters at this first hidden layer, we get $K$ two-dimensional output feature maps, which together are treated as a single three-dimensional feature map. Each of the $K$ output feature maps is seen as a separate channel of information, so now we have $K$ channels in contrast to the three-color channels of the original input feature map.

# Example

- Some additional details (cont'd):
    - The three-dimensional feature map is just like the *activations* in a hidden layer of a simple neural network, but organized and produced in a spatially structured way.
    - We typically apply the ReLU activation function to the convolved image. This step is sometimes viewed as a separate layer in the convolutional neural network, referred to as a *detector layer*.

# Convolution Operation

**Convolution** is formally called **cross-correlation** in the mathematics literature.

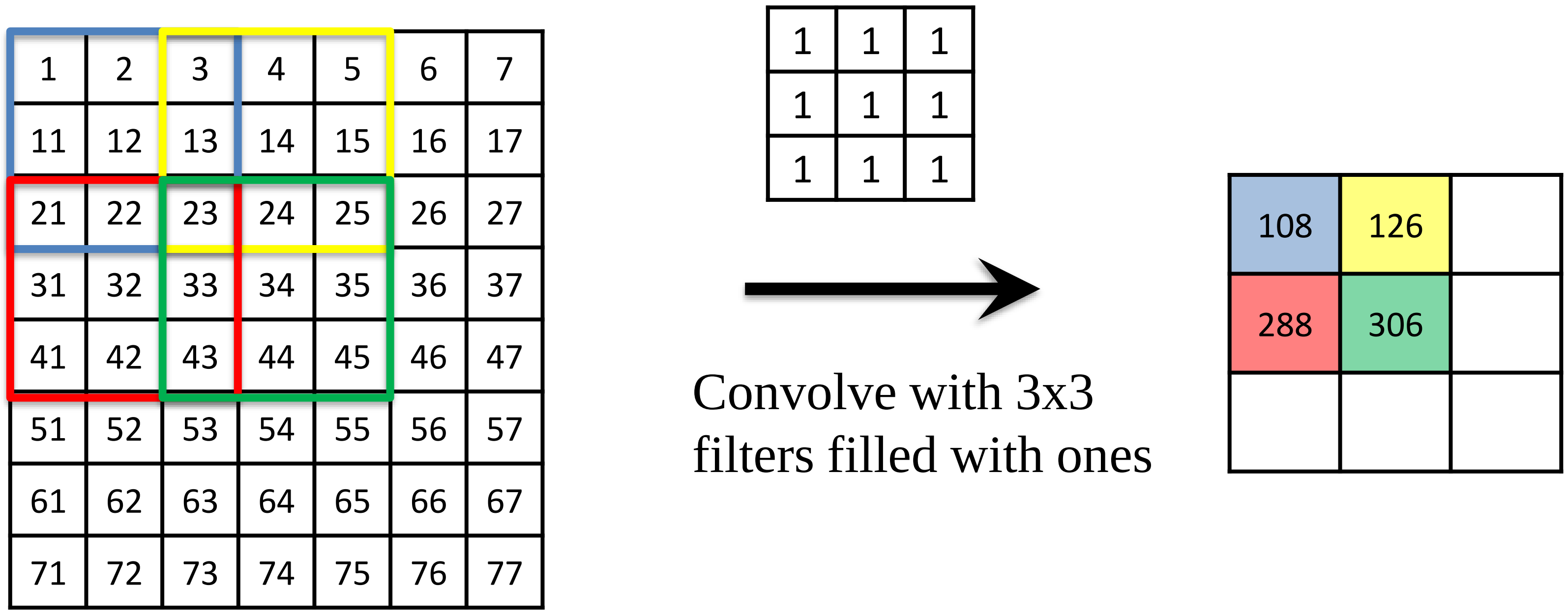
Performing convolutions on images instead of directly connecting each pixel to the neural network units has two main advantages:

1. Reduces the number of parameters we need to learn. Instead of learning the weights connecting each input pixel, we only need to learn the weights of the filter in a lower dimensional space.

2. Preserves locality. We don’t have to flatten the image matrix into a vector; thus the relative positions of the image pixels are preserved.

# Strides

*Stride* is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.



| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 |

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

Convolve with 3x3 filters filled with ones

| | | |
|---|---|---|
| 108 | 126 | |
| 288 | 306 | |
| | | |

**Stride of 2 pixels**

## Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

- Zero-padding, pad the picture with zeros so that it fits
- Valid padding, drop the part of the image where the filter did not fit and keeps only valid part of the image.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 3 | 5 | 2 | 6 | 1 | 0 |
| 0 | 7 | 0 | 2 | 4 | 1 | 3 | 0 |
| 0 | 5 | 1 | 3 | 6 | 2 | 7 | 0 |
| 0 | 2 | 4 | 3 | 0 | 0 | 1 | 0 |
| 0 | 1 | 7 | 3 | 9 | 2 | 1 | 0 |
| 0 | 3 | 6 | 2 | 4 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

padding applied to an image matrix with one pixel.

Notice the zeros in the outer rows and columns

## Activation Function

### Non-Linearity (ReLU)

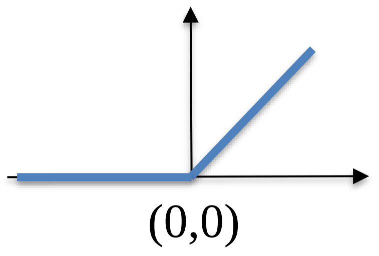
ReLU stands for Rectified Linear Unit for a non-linear operation. The output is

$$f(x) = \max(0, x)$$

ReLU's purpose is to introduce non-linearity in CNN. Since the real-world data would want CNN to learn would be non-negative linear values. Other nonlinear functions such as **tanh** or **sigmoid** can also be used. However, ReLU is better than the other two.

|  |  |  |  |
|---|---|---|---|
| 15 | 20 | -10 | 35 |
| 18 | -110 | 25 | 100 |
| 20 | -15 | 25 | -10 |
| 101 | 75 | 18 | 23 |

ReLU transfer function



|  |  |  |  |
|---|---|---|---|
| 15 | 20 | 0 | 35 |
| 18 | 0 | 25 | 100 |
| 20 | 0 | 25 | 0 |
| 101 | 75 | 18 | 23 |

## POOLING OPERATION

# Pooling Operation

- After the convolutional layer, it is a common practice to pass these values into the next layer which is known as the ***pooling*** layer. Spatial pooling also called *subsampling* or *downsize sampling* which **reduces the dimensionality** of each feature matrix but retains important information.

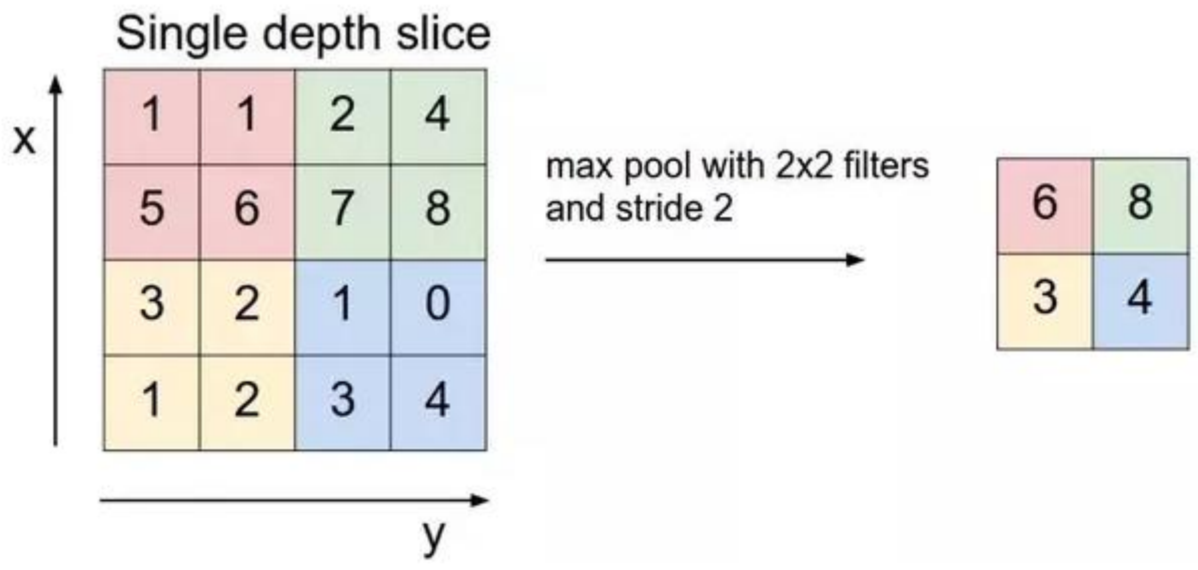
- While there are a number of possible ways to perform pooling (e.g., Max Pooling, Average Pooling, Sum Pooling), the max pooling operation summarizes each non-overlapping (e.g., 2x2) block of pixels in an image using the maximum value in the (2x2) block.

- This reduces the size of the image by a factor of two in each direction, and it also provides some **location invariance**: i.e., as long as there is a large value in one of the four pixels in the block, the whole block registers as a large value in the reduced image.

## Pooling Operation

Here is a simple example of max pooling:

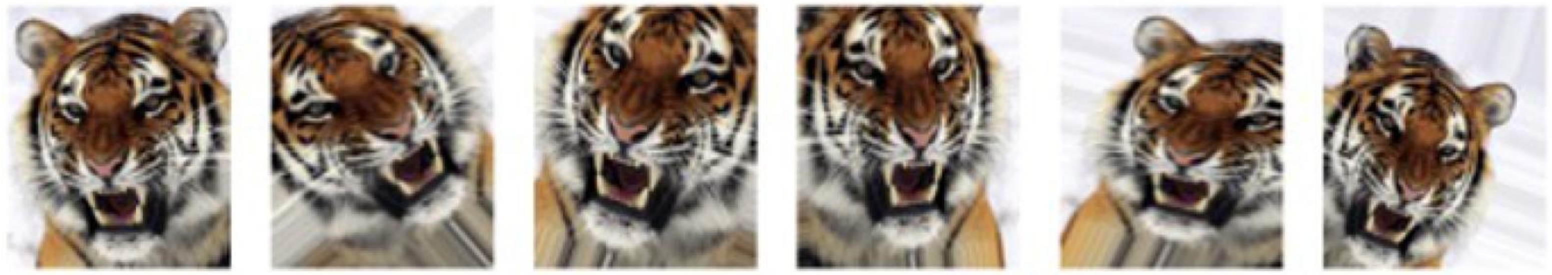
$$\text{Max pool}\begin{bmatrix}1 & 2 & 5 & 3\\ 3 & 0 & 1 & 2\\ 2 & 1 & 3 & 4\\ 1 & 1 & 2 & 0\end{bmatrix}\rightarrow\begin{bmatrix}3 & 5\\ 2 & 4\end{bmatrix}$$



## DATA AUGMENTATION

# Data Augmentation

An additional important trick used with image modeling is data augmentation. Essentially, each training image is replicated many times, with each replicate randomly distorted in a natural way such that human recognition is unaffected.



The original image (leftmost) is distorted in natural ways to produce different images with the same class label. These distortions do not fool humans, and act as a form of regularization when fitting the CNN.

Typical distortions are zoom, horizontal and vertical shift, shear, small rotations, and in this case horizontal flips. At face value this is a way of increasing the training set considerably with somewhat different examples, and thus protects against overfitting.

# Data Augmentation

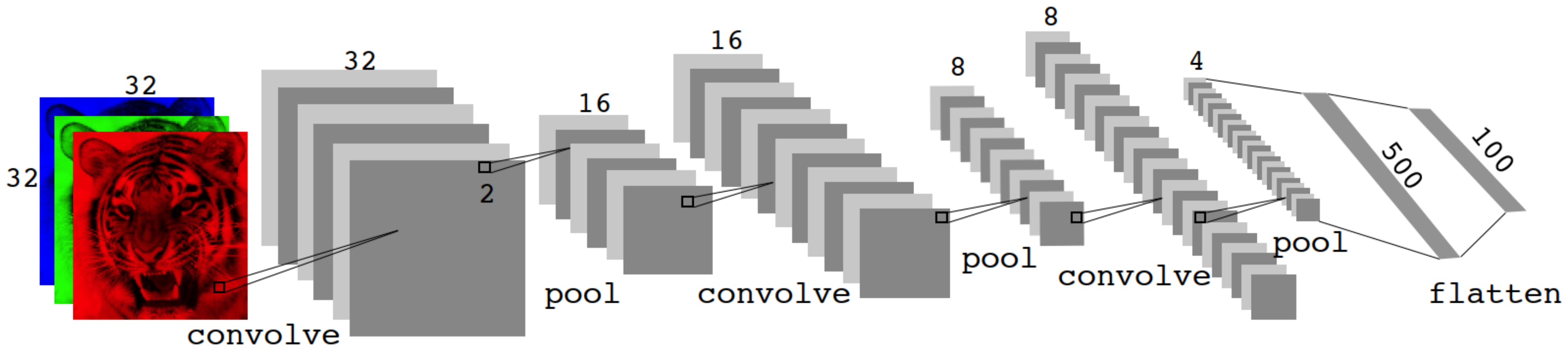
- In fact, we can see this as a form of **regularization**: we build a cloud of images around each original image, all with *the same label*. This kind of fattening of the data is similar in spirit to ridge regularization.

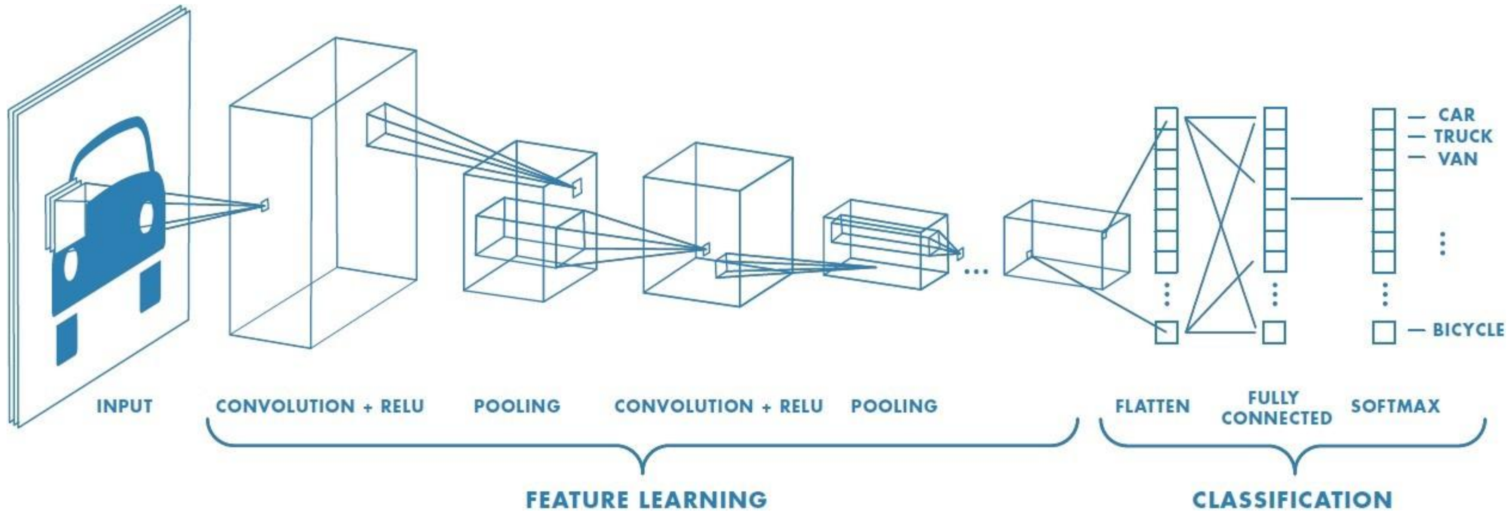
- The stochastic gradient descent algorithms for fitting deep learning models repeatedly process randomly-selected batches of, say, 128 training images at a time.

- This works hand-in-glove with augmentation, because we can distort each image in the batch on the fly, and hence do not have to store all the new images.

## ARCHITECTURE OF A CONVOLUTIONAL NEURAL NETWORK

# Architecture of CNN

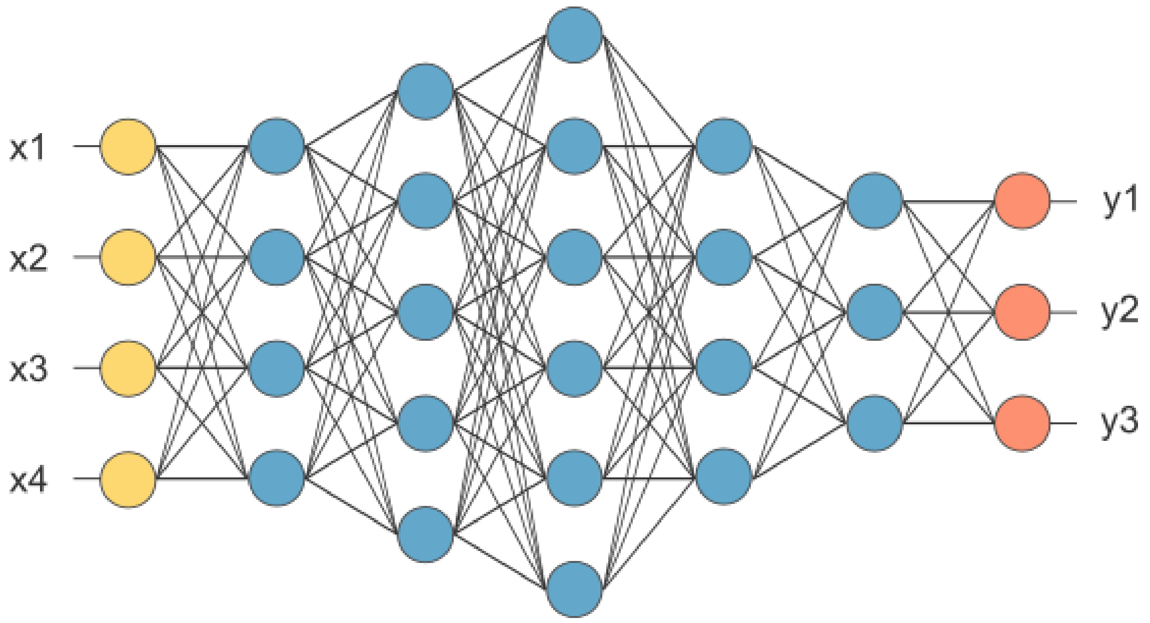




Examples of Complete CNN Architecture

# Fully Connected (FC) Layer

Flattened as FC layer **after pooling layer**



The feature map matrices will be converted as vector ($x_1$, $x_2$, $x_3$, …). With the fully connected layers, we combined these features together to create a model. Finally, we have an activation function such as ***softmax*** to classify the outputs, such as tiger, cat, dog, car, or truck etc.

# Convolutional Neural Networks

- Each subsequent convolve layer is similar to the first. It takes as input the three-dimensional feature map from the previous layer and treats it like a single multi-channel image. Each convolution filter learned has as many channels as this feature map.
- Since the channel feature maps are reduced in size after each pool layer, we usually increase the number of filters in the next convolve layer to compensate.
- Sometimes we repeat several convolve layers before a pool layer. This effectively increases the dimension of the filter.
- These operations are repeated until the pooling has reduced each channel feature map down to just a few pixels in each dimension. At this point the three-dimensional feature maps are **flattened** — the pixels are treated as separate units — and fed into one or more **fully-connected** layers before reaching the output layer, which is a ***softmax*** activation.

# Convolutional Neural Networks

## Summary

- Preprocess images: convert each image into a 3-channel matrices
- Feed our input image into the convolutional layer.
- **Convolutional layers**: perform convolution on the image. Choose parameters for convolution including stride, padding, and filter size. Add as many convolutional layers as possible until satisfied. Apply *ReLU* activation to the matrix.
- **Pooling Layers**: perform pooling on the output to reduce the dimension.
- **FC Layers**: flatten the output of the last pooling layer and feed into a fully connected layer; output the class using an activation function such as *softmax* function.

## Parallel Computation

Since the CNN models are typically very large, model **parallelization** is often necessary. In some cases, even a single layer of neurons cannot fit in a single machine's memory. The first way of parallelizing the model is to store each layer on a different machine.

In a forward pass, each machine receive data from the input machine, compute the output and pass it to the machine holding the next layer.

In a backward pass, each machine receives gradients from the next machine, compute the gradients of its parameters and feed them back to the machine holding the previous layer.

For a fully connected layer, there can be so many parameters that even splitting them onto several machines is necessary. In this case, communications happen if there is a forward or backward pass across the split boundaries.

## Convolutional Neural Networks

To fit large CNNs, it is common to use a **parameter server** to hold all model parameters. Several machines will then work as a bank, each bank has a replica of model from the parameter server, compute gradient and send the gradients back to the **model server**. Each bank of machines performs computation on a **stale model**. Each bank has an iteration number. When it tries to update the model, the update can be rejected if the stale model is too old.

Since there are too many parameters in a CNN, it is prone to overfitting. There are several heuristics of preventing this from happening. For example, we can use **regularization** when training the network. Another approach is **dropout**, which deactivates some neurons with some probability when training.

References
R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, 1995.

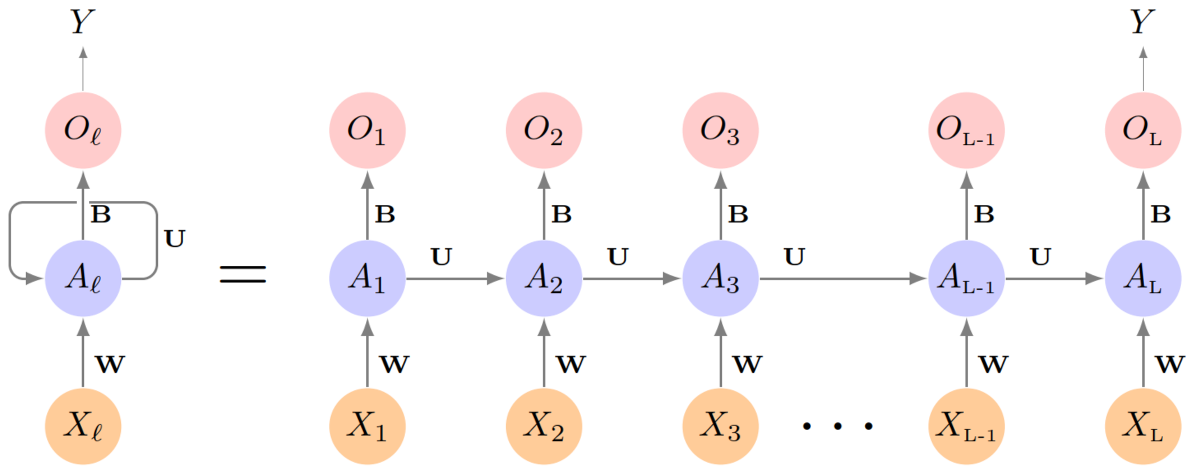
## RECURRENT NEURAL NETWORKS

# Recurrent Neural Networks

Many data sources are sequential in nature and call for special treatment when building predictive models. Examples include:

- Documents such as book and movie reviews, newspaper articles, and tweets. The sequence and relative positions of words in a document capture the narrative, theme and tone, and can be exploited in tasks such as *topic classification, sentiment analysis, and language translation*.
- Time series of temperature, rainfall, wind speed, air quality, and so on. We may want to *forecast the weather several days ahead*, or climate several decades ahead.
- Financial time series, where we track market indices, trading volumes, stock and bond **prices**, and exchange rates. Here prediction is often difficult, but as we will see, certain *indices can be predicted* with reasonable accuracy.
- Recorded speech, musical recordings, and other sound recordings. We may want to give a *text transcription of a speech*, or perhaps a *language translation*. We may want to assess the quality of a piece of music or assign certain attributes.

# Recurrent Neural Networks

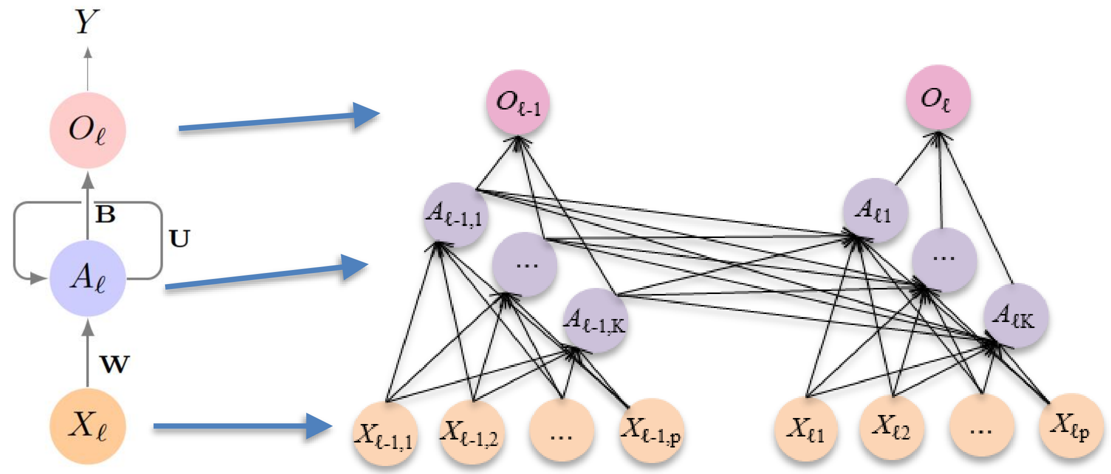


Schematic of a simple recurrent neural network.

The figure illustrates the structure of a very basic RNN with a sequence $X = \{X_1, X_2, . . . , X_L\}$ as input, a simple scalar $Y$ as output, and a hidden-layer sequence $\{A_\ell\}_1^L = \{A_1, A_2, . . . , A_L\}$. Each $X_\ell$ is a vector;

- The sequence is processed one vector $X_\ell$ at a time.
- The activations $A_\ell$ in the hidden layer is calculated by taking as input the vector $X_\ell$ and the activation vector $A_{\ell-1}$ from the previous step in the sequence.
- Each Aℓ feeds into the output layer and produces a prediction $O_\ell$ for $Y$. $O_L$, the last of these, is the most relevant.

# Recurrent Neural Networks

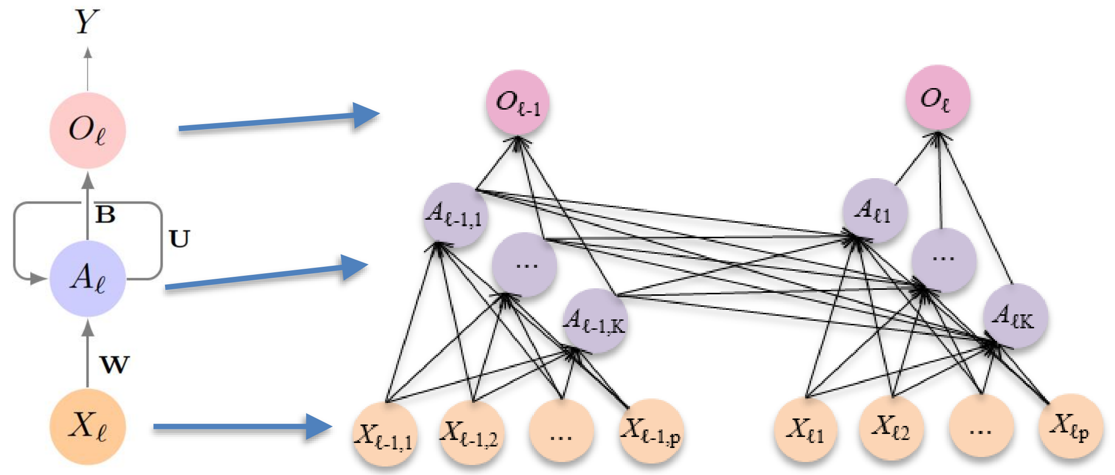


Schematic of a detailed recurrent neural network

In detail, suppose each vector $X_\ell$ of the input sequence has ***p*** components $X_\ell^T = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p})$, and the hidden layer consists of ***K*** units $A_\ell^T = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K})$.

- The matrix ***W*** contains $K \times (p+1)$ shared weights $w_{kj}$ for the input-to-hidden layers.
- The matrix ***U*** contains $K \times K$ shared weights $u_{ks}$ for the hidden-to-hidden layers.
- The vector ***B*** contains $K + 1$ shared weights $\beta_k$ for the output layer.

# Recurrent Neural Networks



Schematic of a detailed recurrent neural network

The activation $A_{\ell K}$ is computed as

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right)$$

and the output $O_\ell$ is computed as

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}$$

for a quantitative response, or with an additional sigmoid activation function for a binary response.

# Recurrent Neural Networks

- Notice that the same weights ***W***, ***U*** and ***B*** are used as we process **each element** in the sequence, i.e., they are not functions of $\boldsymbol{\ell}$. This is a form of weight sharing used by RNNs, and similar to the use of filters weight in convolutional neural networks.
- As we proceed from beginning to end, the activations $A_\ell$ accumulate a history of what has been seen before, so that the learned context can be used for prediction.
- For regression problems the loss function for an observation ($X$, $Y$) is $(Y - O_L)^2$ which only references the final output $O_L = \beta_0 + \sum_{k=1}^{K} \beta_k A_{Lk}$. Thus $O_1$, $O_2$, . . . , $O_{L-1}$ are not used.
- When we fit the model, each element $X_\ell$ of the input sequence $X$ contributes to $O_L$ via the chain and hence contributes **indirectly** to learning the shared parameters ***W***, ***U*** and ***B*** via the loss function.
- With ***n*** input sequence/response pairs (xi, yi), the parameters are found by minimizing the sum of squares

$$\sum_{i=1}^{n}(y_i - o_{iL})^2 = \sum_{i=1}^{n}\Big(y_i - \big(\beta_0 + \sum_{k=1}^{K}\beta_k g\big(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s}\big)\big)\Big)^2.$$

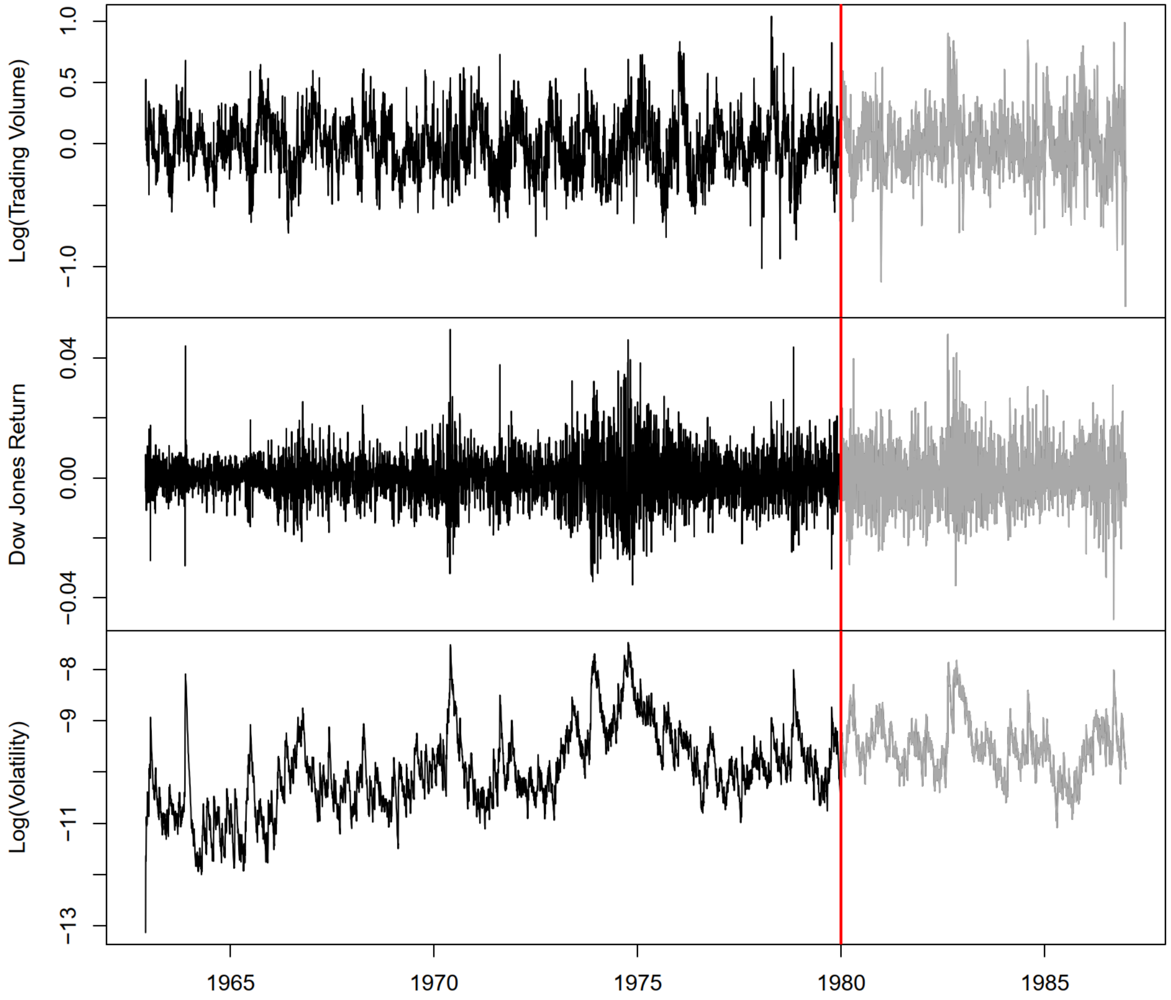
# Recurrent Neural Networks

- Since the intermediate outputs $O_\ell$ are not used, one may well ask *why they are there at all?*

- First of all, they come for free and provide an evolving prediction for the output. Furthermore, for some learning tasks the response is also a sequence, and so the output sequence $\{O_1, O_2, \ldots, O_L\}$ is explicitly needed.

# Example: Time Series Forecasting

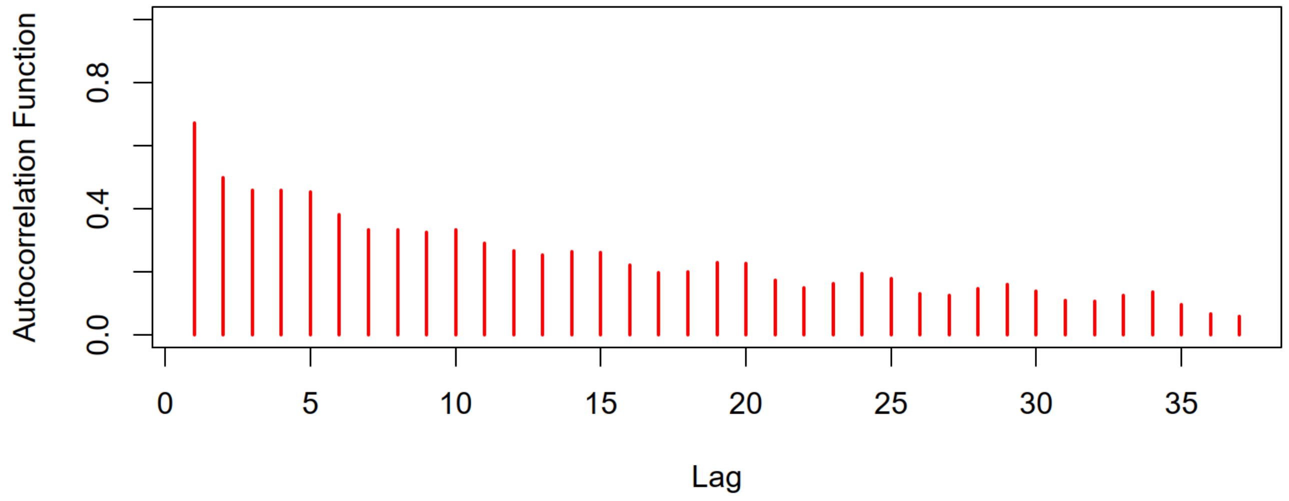
We illustrate RNN use in a financial time series forecasting problem. The figure shows historical trading statistics from the New York Stock Exchange (1962-1986).



# Example: Time Series Forecasting

Predicting stock prices is useful for planning trading strategies. The measurements on day *t* are denoted by $v_t$, $r_t$, $z_t$ for *log_volume*, *DJ_return* and *log_volatility*. There are a total of T = 6,051 such triples.

The day-to-day observations are not independent of each other. The series exhibit ***autocorrelation*** — in this case values nearby in time tend to be similar to each other.



The autocorrelation function for *log_volume*. We see that nearby values are strongly correlated, with correlations above 0.2 as far as 20 days apart.

# Example: RNN forecaster

- We wish to predict a value $v_t$ from past values $v_{t-1}$, $v_{t-2}$, . . ., and also to make use of past values of the other series $r_{t-1}$, $r_{t-2}$, . . . and $z_{t-1}$, $z_{t-2}$, . . .

- How do we represent this problem in terms of the structure of RNN?

- The idea is to extract many short mini-series of input sequences $X = \{X_1, X_2, \ldots, X_L\}$ with a predefined length $L$ (called the lag), and a corresponding target $Y$ . They have the form

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \; X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \cdots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \; \text{and } Y = v_t.$$

- Each $X_\ell$ consists of the three measurements, a 3-vector.

- Each value of $t$ makes a separate $(X, Y)$ pair, for $t$ running from L + 1 to $T$ .

# Example: Comparison with AR model

- For the NYSE data we use the past five trading days to predict the next day's trading volume. Hence, $L = 5$. Since $T = 6{,}051$, we can create 6,046 such $(X, Y)$ pairs.
- We fit this model with $K = 12$ hidden units using the 4,281 training sequences, and then used it to forecast the 1,765 values of log_volume in the test set. We achieve an $R^2 = 0.42$ on the test data.
- A traditional autoregression (AR) linear model, construct a response vector *y* and a matrix *M* of predictors for least squares regression as follows:

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}$$

## Example: Comparison with AR model

- Fitting a regression of *y* on *M*, we get a traditional autoregression (AR) linear model:

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}$$

- and is called an order-*L* autoregressive model, or simply AR(*L*).
- For the NYSE data, an AR model with $L = 5$ achieves a test $R^2$ of 0.41, slightly inferior to the 0.42 achieved by the RNN.

## Long Short-Term Memory (LSTM) Models

The Problem with Sequence Models

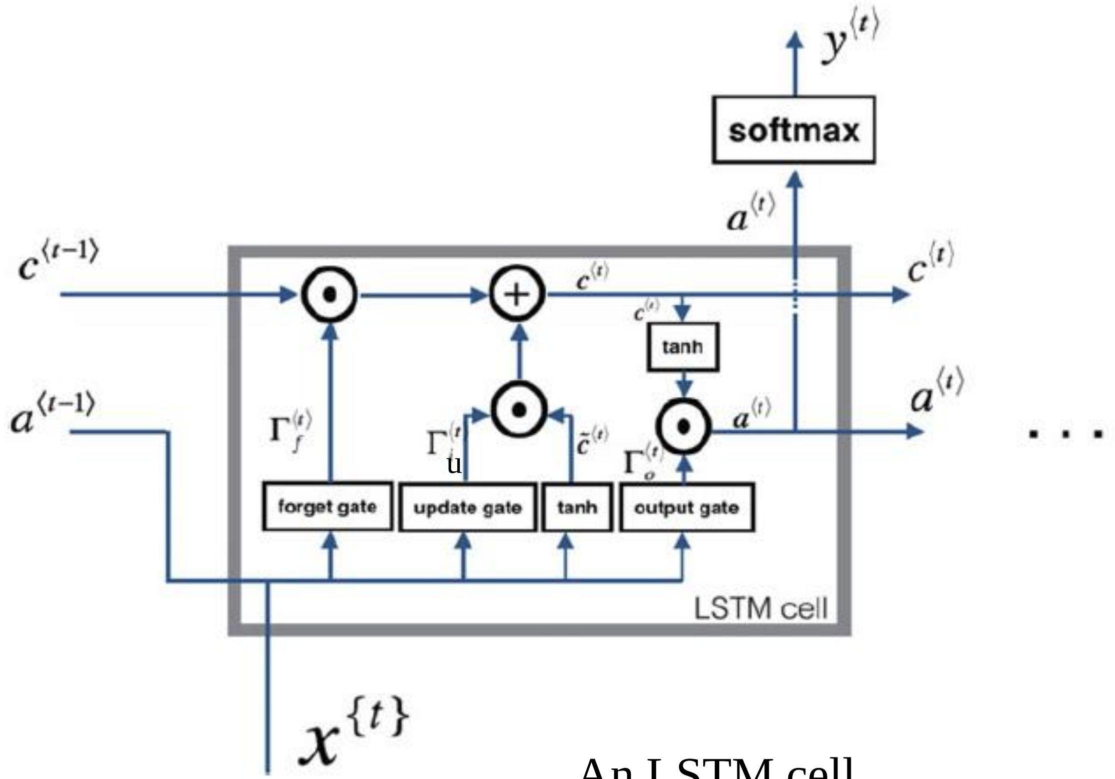
1. Vanishing/Exploding Gradients
In plain sequence models, the more time-steps we have, the more chances we have that backpropagation gradients are either exploding or vanishing.

2. The Problem of Long-Term Dependencies
When the gap between the relevant information and the time-step grows, RNNs become unable to connect the information.

Long Short-Term Memory networks (LSTMs) can overcome the above two problems. LSTMs are a special type of recurrent neural networks which are capable of learning long-term dependencies.

# Long Short-Term Memory (LSTM) Models

**Two tracks** of hidden-layer activations are maintained, so that when the activation $A_\ell$ is computed, it gets input from hidden units both **further back in time**, and **closer in time** — a so-called *LSTM-RNN*. With long sequences, this *LSTM-RNN* overcomes the problem of early signals being washed out by the time they get propagated through the chain to the final activation vector $A_L$.



$y^{\langle t \rangle}$
softmax
$a^{\langle t \rangle}$
$c^{\langle t-1 \rangle}$
$c^{\langle t \rangle}$
tanh
$a^{\langle t-1 \rangle}$
$\Gamma_f^{\langle t \rangle}$
$\Gamma_u^{\langle t \rangle}$
$\tilde{c}^{\langle t \rangle}$
$\Gamma_o^{\langle t \rangle}$
forget gate
update gate
tanh
output gate
LSTM cell
. . .
$x^{\{t\}}$

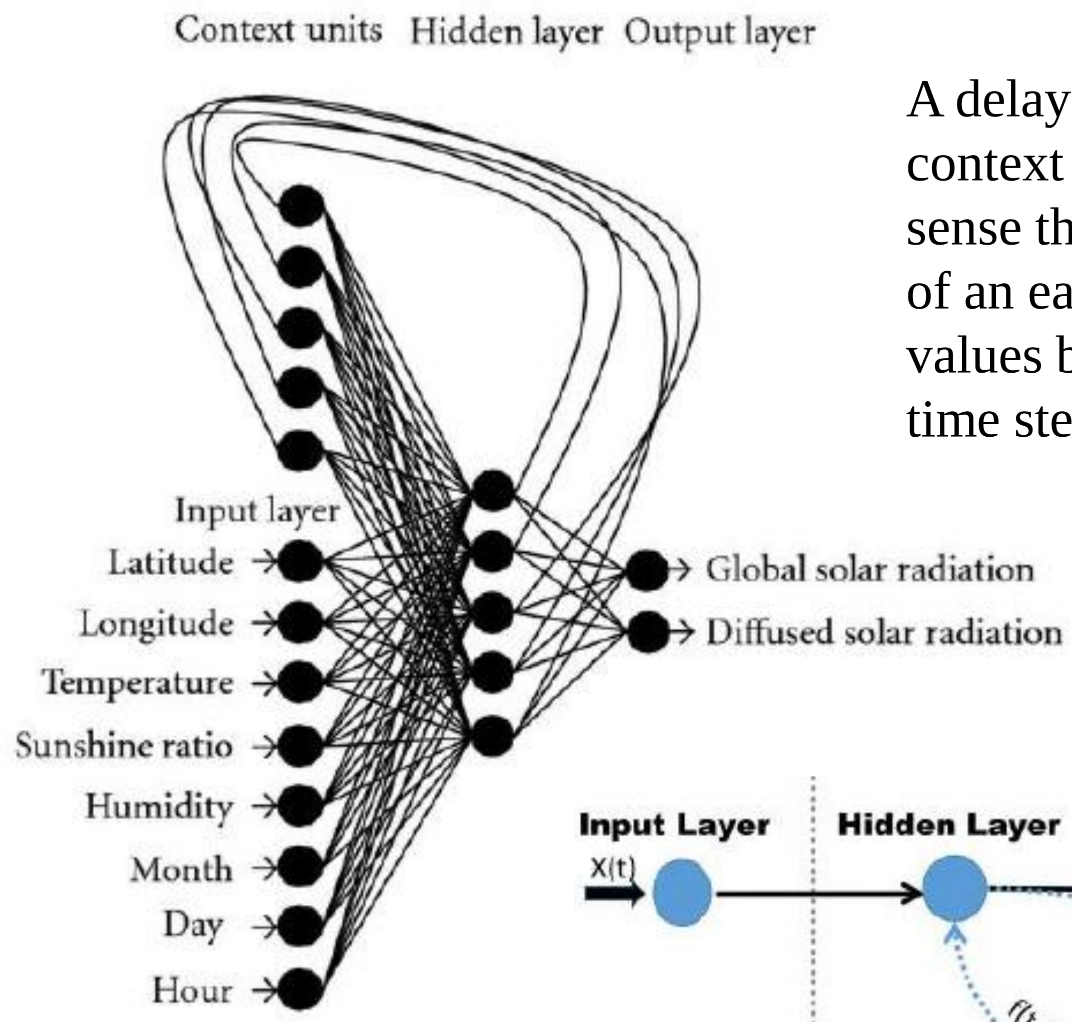
An LSTM cell

## ELMAN NEURAL NETWORKS

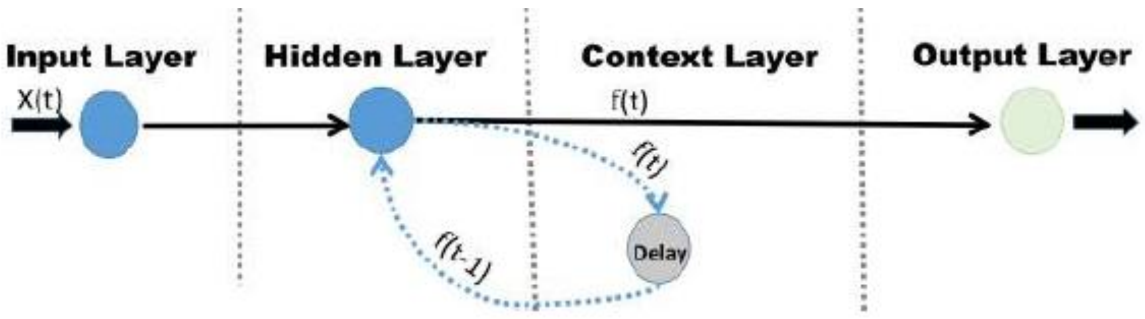
# Elman Neural Networks

Elman neural network is a recurrent neural network. In a recurrent neural network, neurons connect back to other neurons, information flow is multi-directional, so the activation of neurons can flow around in a loop. This type of neural network has a sense of time and memory of earlier networks states which enables it to learn sequences which vary over time, perform classification tasks and develop predictions of future states. As a result, recurrent neural networks are used for classification, stochastic sequence modeling and associative memory tasks.

# Elman Neural Networks



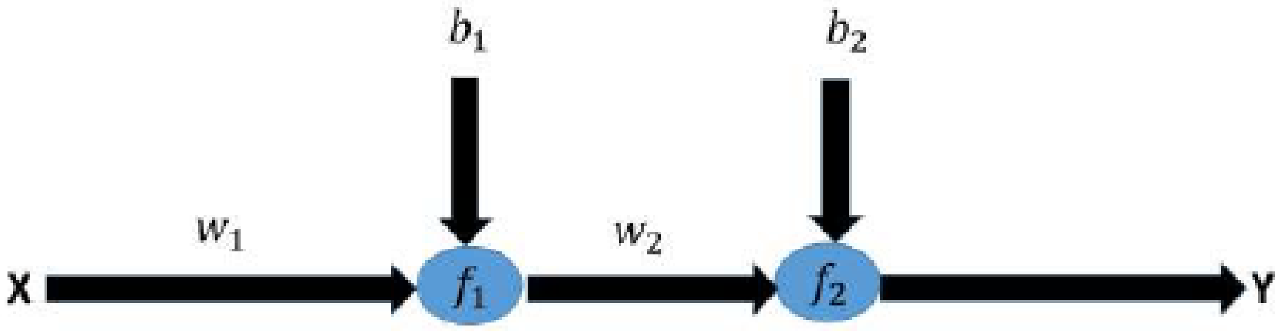
A delay neuron is introduced in the context layer. It has a memory in the sense that it stores the activation values of an earlier time step. It releases these values back into the network at the next time step.



# Elman Neural Networks

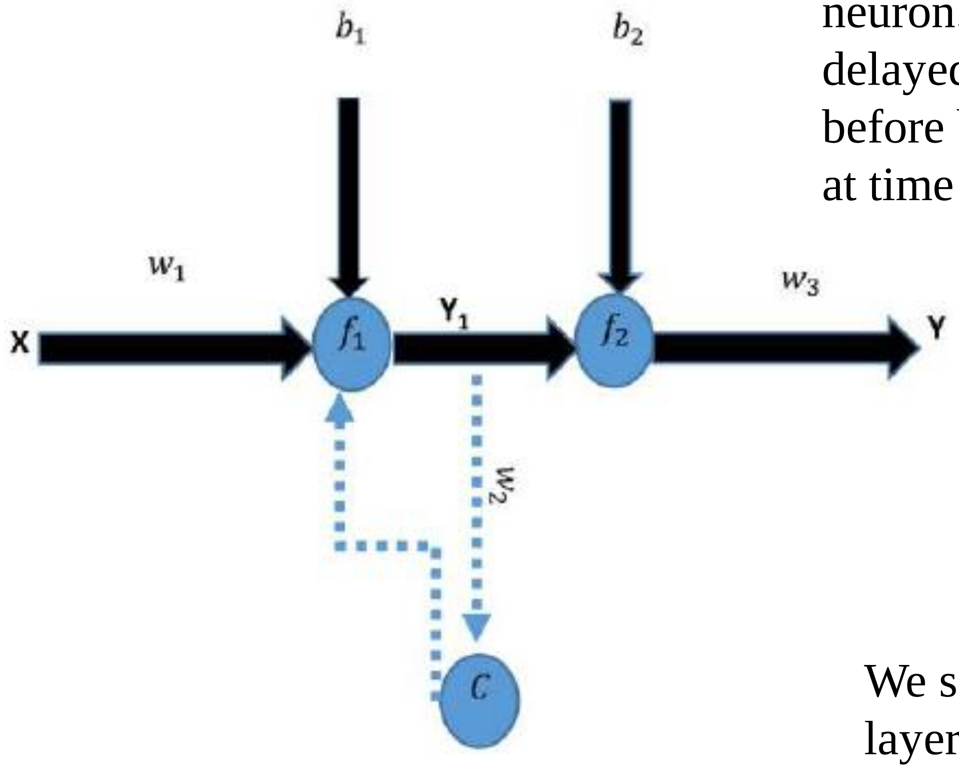
Suppose we have a neural network consisting of only two neurons. The network has one neuron in each layer. Each neuron has a bias denoted by $b_1$ for the first neuron, and $b_2$ for the second neuron. The associated weights are $w_1$ and $w_2$ with activation function $f_1$ and $f_2$. The output *Y* as a function of the input attribute *X* is given by:

$$Y = f_2\left(w_2 f_1\left(w_1 X + b_1\right) + b_2\right)$$



Two neuron network

# Elman Neural Networks



Two node Elman network

There is a context neuron which feeds the activation from the hidden layer neuron back to that same neuron. The activation from the context neuron is delayed by one time step and multiplied by $w_2$ before being fed back into the network. The output at time *t* is given by:

$$Y[t] = f_2\left(w_3 f_1\left(w_1 X[t] + w_2 C + b_1\right) + b_2\right)$$

where,

$$C = Y_1[t-1]$$

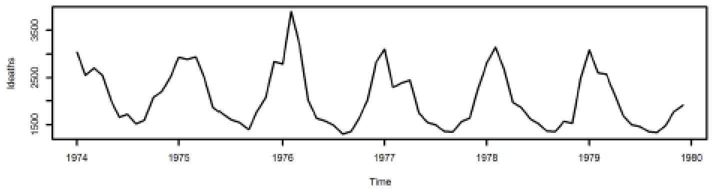
We see, in this simple example, that the hidden layer is fully connected to inputs and the network exhibits **recurrent** connections; and the use of **delayed** memory creates a more **dynamic** neural network system.

# Elman Neural Networks

Elman neural networks are akin to the multi-layer perceptron augmented with one or more context layers. The number of neurons in the context layer is equal to the number of neurons in the hidden layer. In addition, the context layer neurons are fully connected to all the neurons in the hidden layer.

The neurons in the context layer are included because they remember the previous internal state of the network by storing hidden layer neuron values. The stored values are delayed by one time step; and are used during the next time step as additional inputs to the network.

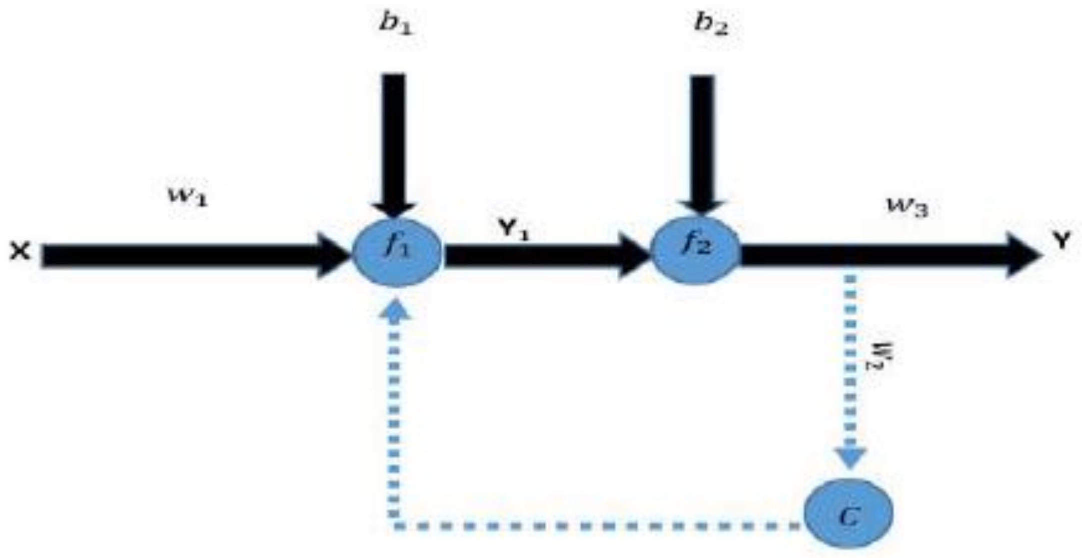
Elman Neural Networks are useful in applications where we are interested in **predicting** the next output in given sequence. Their dynamic nature can capture time-dependent patterns. Elman networks are particularly useful for modeling **time series data**.



## JORDAN NEURAL NETWORKS

# Jordan Neural Networks

Jordan neural networks are similar to the Elman neural network. The only difference is that the context neurons are fed from the output layer instead of the hidden layer. The activation of the output nodes is recurrently copied back into the context nodes. This provides the network with a memory of its previous state.



A simple Jordan neural network

# Jordan Neural Networks

Jordan Neural Networks appear capably of modeling time series data for prediction, and are useful for classification problems.

Standardize the attribute data prior to using a neural network model. Whilst there are no fixed rules about how to normalize inputs, here are four popular choices for an attribute xi:

$$z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

$$z_i = \frac{x_i - \overline{x}}{\sigma_x}$$

$$z_i = \frac{x_i}{\sqrt{SS_i}}$$

$$z_i = \frac{x_i}{x_{max} + 1}$$

where

$SS_i$ is the sum of squares of $x_i$, and $\overline{x}$ and $\sigma^2$ are the mean and standard deviation of $x_i$.

## RESTRICTED BOLTZMANN MACHINES

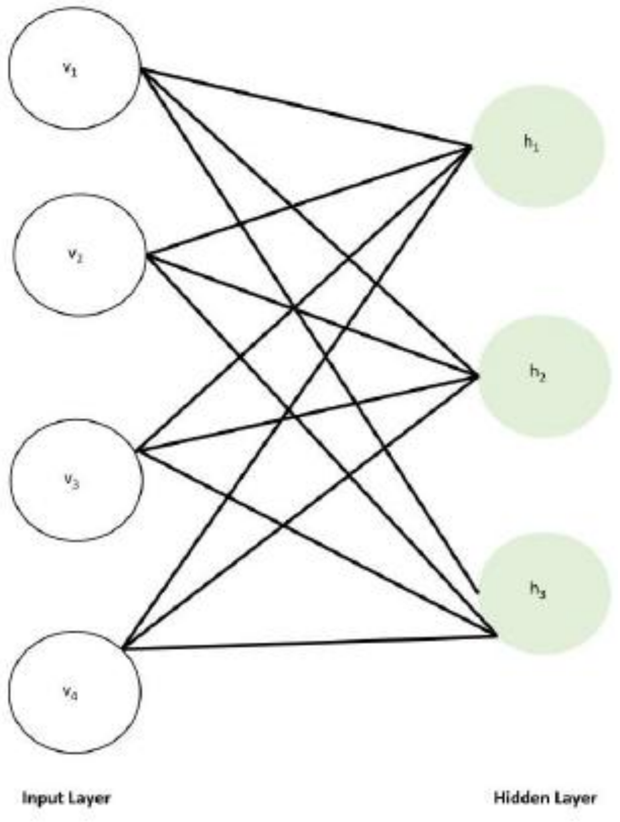
# Restricted Boltzmann Machines

Restricted Boltzmann Machine (RBM) is an unsupervised learning model that approximates the *probability density function* of sample data. The “restricted” part of the name points to the fact that there are no connections between units in the same layer. The weights of the model are learned by **maximizing the likelihood function of the samples**.

Since it is used to approximate a *probability density function*, it is often referred to in the literature as a **generative model**. Generative learning involves making guesses about the probability distribution of the original input in order to reconstruct it.

A **deep belief network** (DBN) is a probabilistic generative multilayer neural network composed of several **stacked Restricted Boltzmann Machines**.

# Restricted Boltzmann Machines



Graphical representation of an RBM model

It is composed of two layers in which there are a number of units with inner layer connections. Connections between layers are symmetric and bidirectional, allowing information transfer in **both directions**.

It has one visible layer containing four nodes and one hidden layer containing three nodes. The visible nodes are related to the input attributes so that for each unit in the visible layer, the corresponding attribute value is observable.

Visible layer node has binary state $v_i = 0$ or $v_i = 1$ that make stochastic decisions about whether to transmit that input or not.; hidden layer node contains binary state $h_j = 0$ or $h_j = 1$. For each unit or node in the hidden layer, the corresponding value is unobservable and it needs to be inferred

# Restricted Boltzmann Machines

The RBM is a probabilistic energy-based model, meaning the probability of a specific configuration of the visible and hidden units is proportional to the negative exponentiation of an energy function, *E* (*v*, *h*). For each pair of a visible vector and a hidden vector the probability of the pair (*v*, *h*) is defined as follows:

$$P(v,h) = \frac{1}{Z} \exp -\tilde{E}(v,h)$$

where the denominator *Z* is a normalizing constant known as the partition function. The partition function sums over all possible pairs of visible and hidden variables is computed as:

$$Z = \sum_{v} \sum_{h} \exp -\tilde{E}(v,h)$$

It is therefore a normalizing constant such that *P*(*v*, *h*) defines a probability distribution over all possible pairs of *v* and *h*.

# Restricted Boltzmann Machines

As there are no intra-layer connections within an RBM the energy of a joint configuration (*v*, *h*), which defines a bipartite structure, is given by:

$$\tilde{E}(v,h) = -\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij} v_i h_j - \sum_{i=1}^{m} v_i b_i - \sum_{j=1}^{n} h_j d_j$$

where $w_{ij}$ is the weight associated with the link between $v_i$ and $h_j$ . Intuitively, weights identify the correlation of the nodes. Larger weights imply a larger possibility that connected nodes **concur**. The parameters $b_i$ and $d_j$ are the biases of the jth hidden and ith visible nodes respectively; and *m* and *n* are the number of nodes in the visible and hidden layer.

Another way to think about a RBM is as a parametric model of the joint probability distribution of visible and hidden variables.

# Restricted Boltzmann Machines

Since the hidden units of the RBM only connect to units **outside** of their specific layer, they are **mutually independent given the visible units**. The conditional independence of the units in the same layer is a nice property because it allows us to factorize the **conditional distributions of the hidden units** given the visible units as:

$$P(h|v) = \prod_j P(h_j|v) \quad \text{with} \quad P(h_j = 1|v) = \prod_j sigmoid\left(\sum_i w_{ij}v_i + d_j\right)$$

The conditional distribution of the visible units can be factorized in a similar way:

$$P(v|h) = \prod_i P(v_i|h) \quad \text{with} \quad P(v_i = 1|h) = \prod_i sigmoid\left(\sum_j w_{ij}h_j + b_i\right)$$

These conditional probabilities are important for the iterative updates between hidden and visible layers when training an RBM model. In practice, *Z* is difficult to calculate so computation of the joint distribution *P*(*v*, *h*) is typically intractable.

# Restricted Boltzmann Machines

The goal in training an RBM is to adjust the parameters of the model, denoted by Θ, so that the log-likelihood function using the training data is maximized. The gradient of **the log-likelihood** is:

$$\frac{\partial}{\partial \Theta} L(\Theta) = -\left\langle \frac{\partial \tilde{E}(v;\Theta)}{\partial \Theta} \right\rangle_{data} + \left\langle \frac{\partial \tilde{E}(v;\Theta)}{\partial \Theta} \right\rangle_{model}$$

where $<\bullet>_{data}$ represents the expectation of all visible nodes *v* in regard to the data distribution and $<\bullet>_{model}$ denotes the model joint distribution defined by $P(v, h)$.

# When to Use Deep Learning

*Should we discard all our older tools, and use deep learning on every problem with data?*

- Let's revisit our *Hitters* dataset where the goal is to predict the Salary of a baseball player with 263 players observations and 19 variables.
- Split the data into a training set of 176 players (two thirds), and a test set of 87 players (one third).
- Three methods were used to fit a regression model to these data.
    - A linear model that has 20 parameters.
    - The same linear model was fit with lasso regularization by 10-fold cross-validation. It selected a model with 12 variables having nonzero coefficients.
    - A neural network with one hidden layer consisting of 64 ReLU units was fit to the data by stochastic gradient descent with a batch size of 32 for 1,000 epochs, and 10% dropout regularization. This model has 1,345 parameters.

# When to Use Deep Learning

| Model | # Parameters | Mean Abs. Error | Test Set $R^2$ |
|---:|---:|---:|---:|
| Linear Regression | 20 | 254.7 | 0.56 |
| Lasso | 12 | 252.3 | 0.51 |
| Neural Network | 1345 | 257.4 | 0.54 |

- Prediction results on the Hitters test data with three methods.
- Similar performance for all three models, which are all respectable.
- With great ease we obtained linear models that work well. However, we spent a fair bit of time fiddling with the configuration parameters of the neural network to achieve these results.
- Hence, in cases like this we are much better off following **the parsimony principle**: when faced with several methods that give roughly equivalent performance, **pick the simplest**.

# When to Use Deep Learning

- We have a number of very powerful tools at our disposal, including neural networks, random forests and boosting, support vector machines, general linear models, to name a few, and variants of these.
- If we can produce models with the simpler tools that perform at the same level as complex tools, Wherever possible, it makes sense to try the simpler models first, because they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches.
- Typically, we expect **deep learning** to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.