Introduction to Data Science: Neural Networks and Deep Learning

Héctor Corrada Bravo

University of Maryland, College Park, USA

CMSC 320: 2020-05-10

Historical Overview

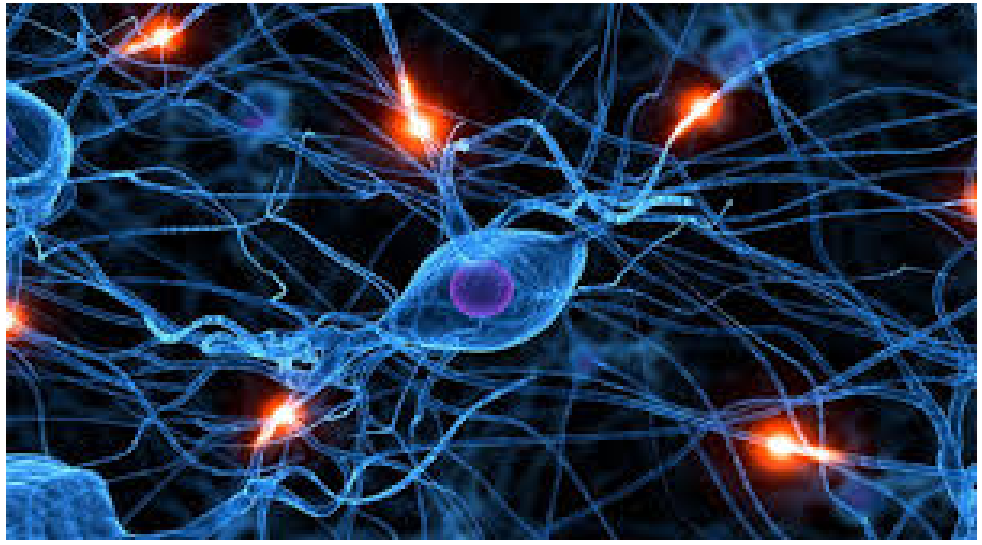
Neural networks are a decades old area of study.

Initially, these computational models were created with the goal of mimicking the processing of neuronal networks.

Historical Overview

Inspiration: model neuron as processing unit.

Some of the mathematical functions historically used in neural network models arise from biologically plausible activation functions.



Historical Overview

Somewhat limited success in modeling neuronal processing

Neural network models gained traction as general Machine Learning models.

Historical Overview

Strong results about the ability of these models to approximate arbitrary functions

Became the subject of intense study in ML.

In practice, effective training of these models was both technically and computationally difficult.

Historical Overview

Starting from 2005, technical advances have led to a resurgence of interest in neural networks, specifically in *Deep Neural Networks*.

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Advances in neural network architecture design and network optimization

Deep Learning

Advances in computational processing:

- powerful parallel processing given by Graphical Processing Units

Advances in neural network architecture design and network optimization

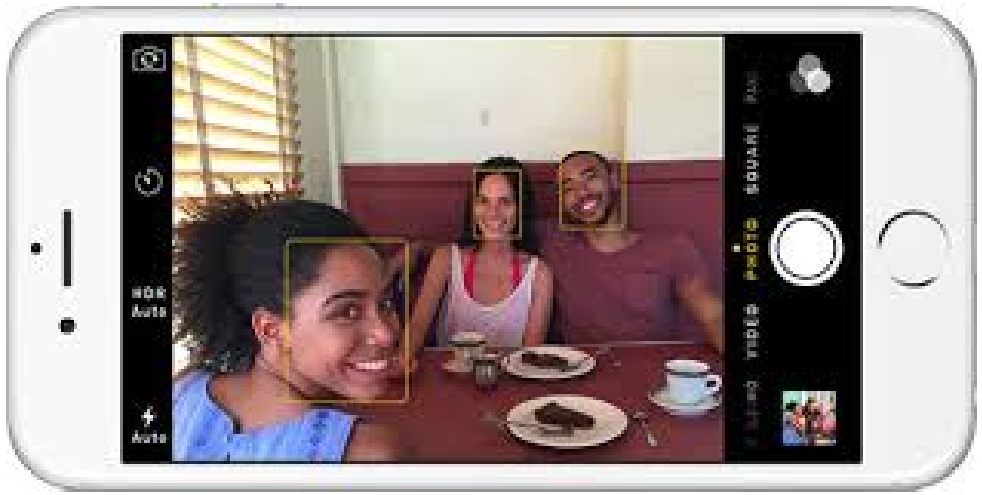
Researchers apply Deep Neural Networks successfully in a number of applications.

Deep Learning

Self driving cars make use of Deep Learning models for sensor processing.

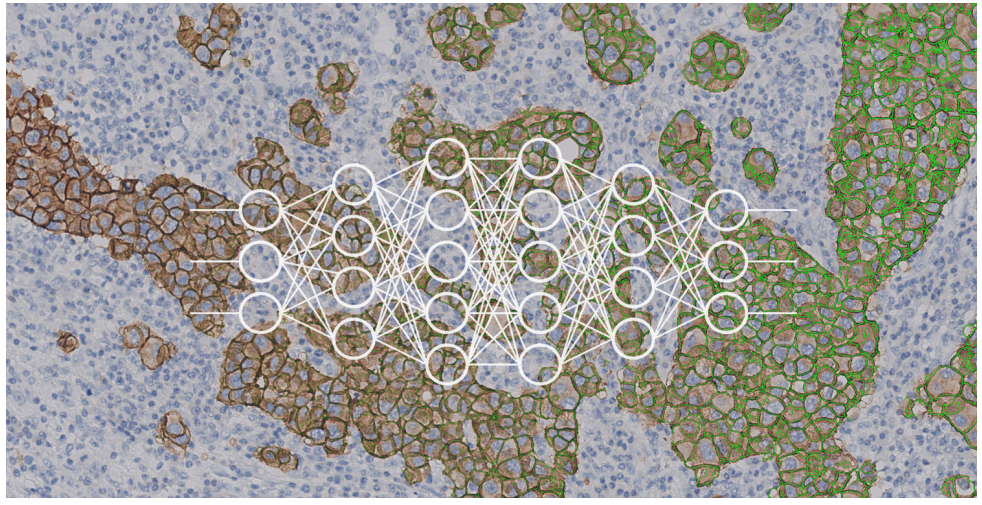
Deep Learning

Image recognition software uses Deep Learning to identify individuals within photos.



Deep Learning

Deep Learning models have been applied to medical imaging to yield expert-level prognosis.



Deep Learning

An automated Go player, making heavy use of Deep Learning, is capable of beating the best human Go players in the world.

Neural Networks and Deep Learning

In this unit we study neural networks and recent advances in Deep Learning.

Projection-Pursuit Regression

To motivate our discussion of Deep Neural Networks, let's turn to simple but very powerful class of models.

As per the usual regression setting, suppose

- given predictors (attributes) $\{X_1, \ldots, X_p\}$ for an observation
- we want to predict a continuous outcome Y.

Projection-Pursuit Regression

The Projection-Pursuit Regression (PPR) model predicts outcome Y using function $f(X)$ as

$$f(X) = \sum_{i=1}^{M} g_m(\mathbf{w}'_m X)$$

where:

- $\mathbf{w}_m$ is a p-dimensional *weight vector*
- so, $\mathbf{w}'X = \sum_{j=1}^{p} w_{mj} x_j$ is a linear combination of predictors x_j
- and g_m, $m = 1, \ldots, M$ are univariate non-linear functions (a smoothing spline for example)

Projection-Pursuit Regression

Our prediction function is a linear function (with M terms).

Each term $g_m(\mathbf{w}_m' X)$ is the result of applying a non-linear function to, what we can think of as, a *derived feature* (or derived predictor) $V_m = \mathbf{w}_m' X$.

Projection-Pursuit Regression

Here's another intuition. Recall the Principal Component Analysis problem we saw in the previous unit.

Given:

- Data set $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, where $\mathbf{x}_i$ is the vector of p variable values for the i-th observation.

Return:

- Matrix $[\phi_1, \phi_2, \ldots, \phi_p]$ of *linear transformations* that retain *maximal variance*.

Projection-Pursuit Regression

Matrix $[\phi_1, \phi_2, \ldots, \phi_p]$ of *linear transformations*

You can think of the first vector ϕ_1 as a linear transformation that embeds observations into 1 dimension:

$$Z_1 = \phi_{11} X_1 + \phi_{21} X_2 + \cdots + \phi_{p1} X_p$$

where ϕ_1 is selected so that the resulting dataset $\{z_1, \ldots, z_n\}$ has *maximum variance*.

Projection-Pursuit Regression

$$f(X) = \sum_{i=1}^{M} g_m(\mathbf{w}'_m X)$$

In PPR we are reducing the dimensionality of X from p to M using linear projections,

And building a regression function over the representation with reduced dimension.

Projection-Pursuit Regression

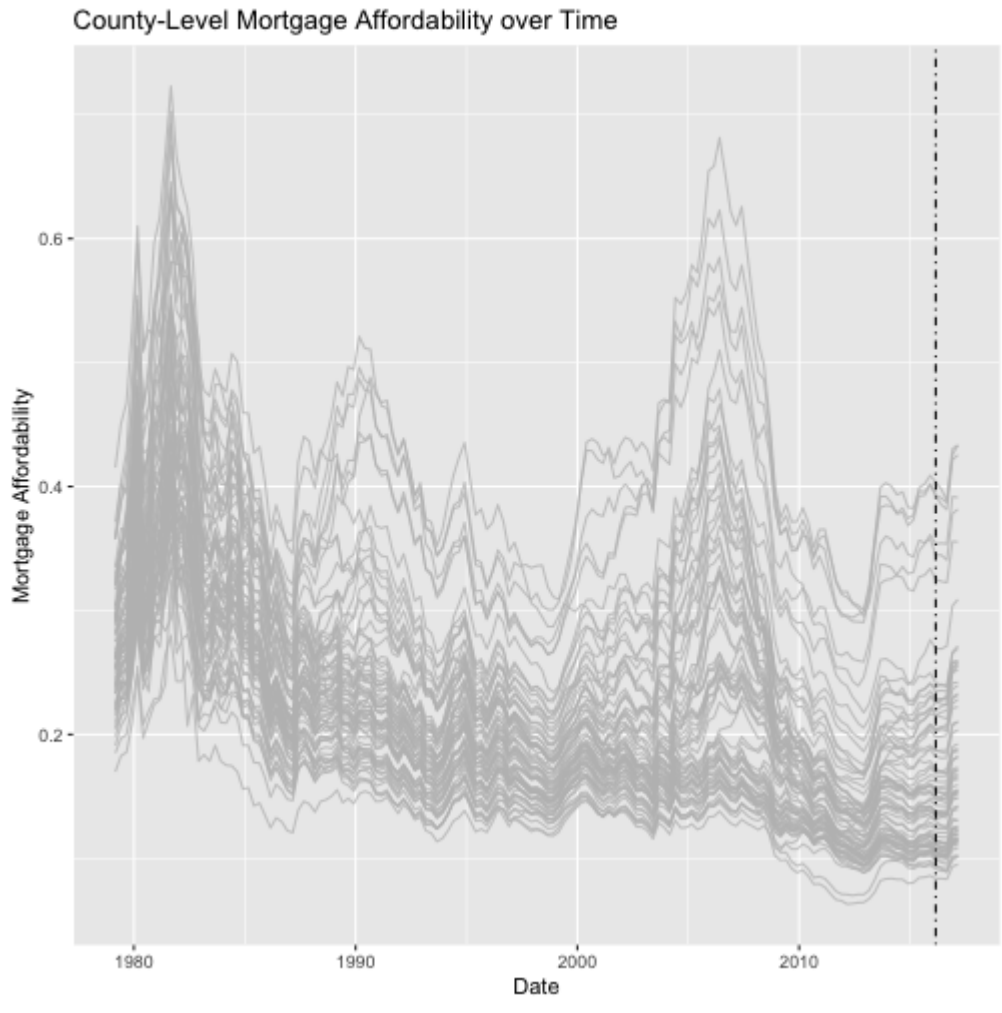
Let's revist the data from our previous unit and see how the PPR model performs.

This is a time series dataset of mortgage affordability as calculated and distributed by Zillow: https://www.zillow.com/research/data/.

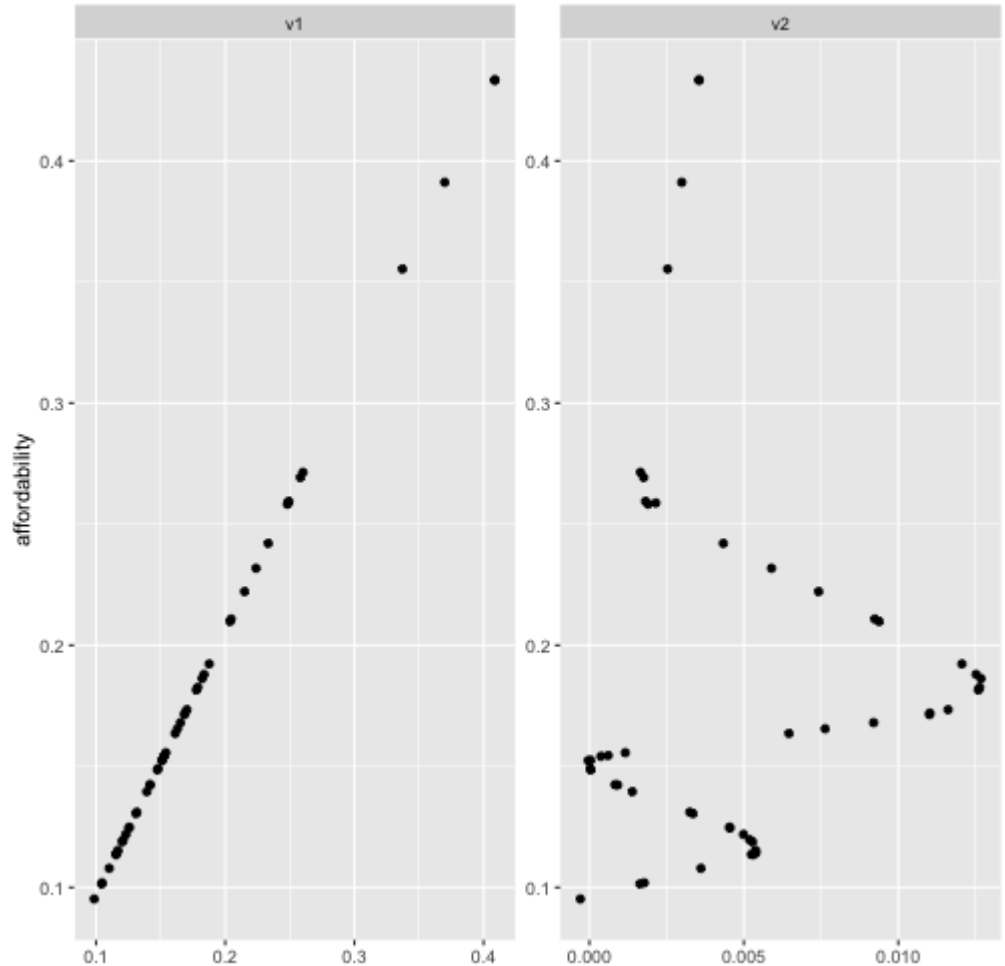
The dataset contains affordability measurements for 76 counties with data from 1979 to 2017. Here we plot the time series of affordability for all counties.

Projection-Pursuit Regression

We will try to predict affordability at the last time-point given in the dataset based on the time series up to one year previous to the last time point.



Projection-Pursuit Regression



Projection-Pursuit Regression

So, how can we fit the PPR model?

As we have done previously in other regression settings, we start with a loss function to minimize

$$L(g, W) = \sum_{i=1}^{N} \left[y_i - \sum_{m=1}^{M} g_m(\mathbf{w}'_m x_i) \right]^2$$

Use an optimization method to minimize the error of the model.

For simplicity let's consider a model with $M = 1$ and drop the subscript m.

Projection-Pursuit Regression

Consider the following procedure

- Initialize weight vector $\mathbf{w}$ to some value $\mathbf{w}_{\text{old}}$
- Construct derived variable $v = \mathbf{w}_{\text{old}}$
- Use a non-linear regression method to fit function g based on model $E[Y|V] = g(v)$. You can use additive splines or loess

Projection-Pursuit Regression

- Given function g now update weight vector $\mathbf{w}_{\text{old}}$ using a gradient descent method

$$\begin{aligned} \mathbf{w} = & \quad \mathbf{w}_{old} + 2\alpha \sum_{i=1}^{N} (y_i - g(v_i)) g'(v_i) x_i \\ = & \quad \mathbf{w}_{old} + 2\alpha \sum_{i=1}^{N} r_i x_i \end{aligned}$$

where α is a learning rate.

Projection-Pursuit Regression

$$\begin{aligned} \mathbf{w} = & \quad \mathbf{w}_{old} + 2\alpha \sum_{i=1}^{N} (y_i - g(v_i)) g'(v_i) x_i \\ = & \quad \mathbf{w}_{old} + 2\alpha \sum_{i=1}^{N} \tilde{r}_i x_i \end{aligned}$$

In the second line we rewrite the gradient in terms of the residual r_i of the current model $g(v_i)$ (using the derived feature v) weighted by, what we could think of, as the *sensitivity* of the model to changes in derived feature v_i.

Projection-Pursuit Regression

Given an updated weight vector $\mathbf{w}$ we can then fit g again and continue iterating until a stop condition is reached.

Projection-Pursuit Regression

Let's consider the PPR and this fitting technique a bit more in detail with a few observations

We can think of the PPR model as composing three functions:

- the linear projection $\mathbf{w}'x$,
- the result of non-linear function g and, in the case when $M > 1$,
- the linear combination of the g_m functions.

Projection-Pursuit Regression

To tie this to the formulation usually described in the neural network literature we make one slight change to our understanding of *derived feature*.

Consider the case $M > 1$, the final predictor is a linear combination $\sum_{i=1}^{M} g_m(v_m)$.

We could also think of each term $g_m(v_m)$ as providing a *non-linear* dimensionality reduction to a single *derived feature*.

Projection-Pursuit Regression

This interpretation is closer to that used in the neural network literature, at each stage of the composition we apply a non-linear transform to the data of the type $g(\mathbf{w}'x)$.

Projection-Pursuit Regression

The fitting procedure propagates errors (residuals) down this function composition in a stage-wise manner.

Feed-forward Neural Networks

We can now write the general formulation for a feed-forward neural network.

We will present the formulation for a general case where we are modeling K outcomes $Y_1, \ldots, Y_k$ as $f_1(X), \ldots, f_K(X)$.

Feed-forward Neural Networks

In multi-class classification, categorical outcome may take multiple values

We consider Y_k as a discriminant function for class k,

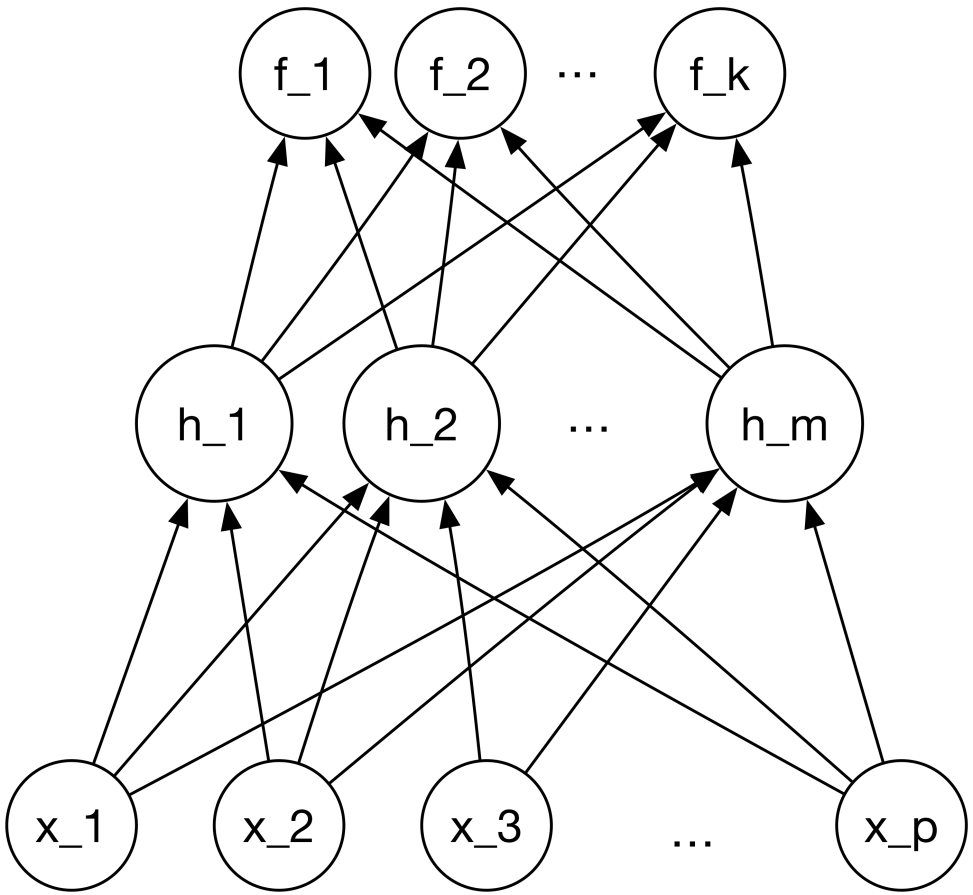
Final classification is made using $\arg\max_k Y_k$. For regression, we can take $K = 1$.

Feed-forward Neural Networks

A single layer feed-forward neural network is defined as

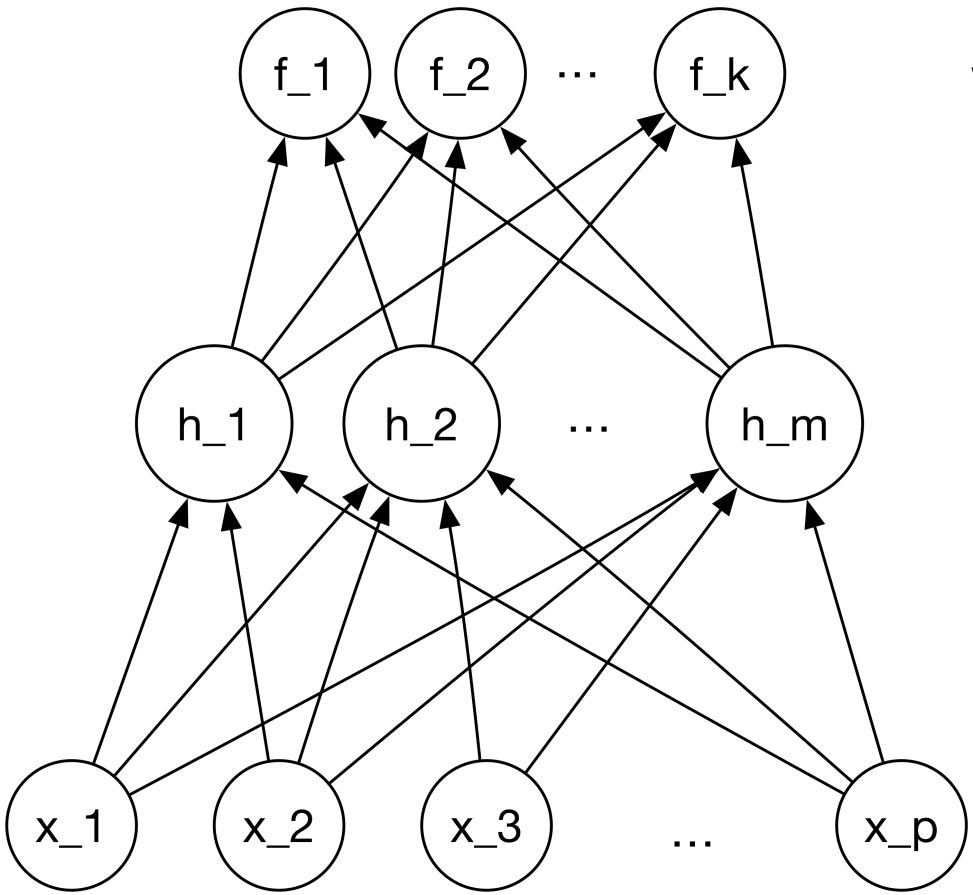
$$h_m = g_h(\mathbf{w}'_{1m} X),\ m = 1, \ldots, M$$
$$f_k = g_{fk}(\mathbf{w}'_{2k} \mathbf{h}),\ k = 1, \ldots, K$$

Feed-forward Neural Networks



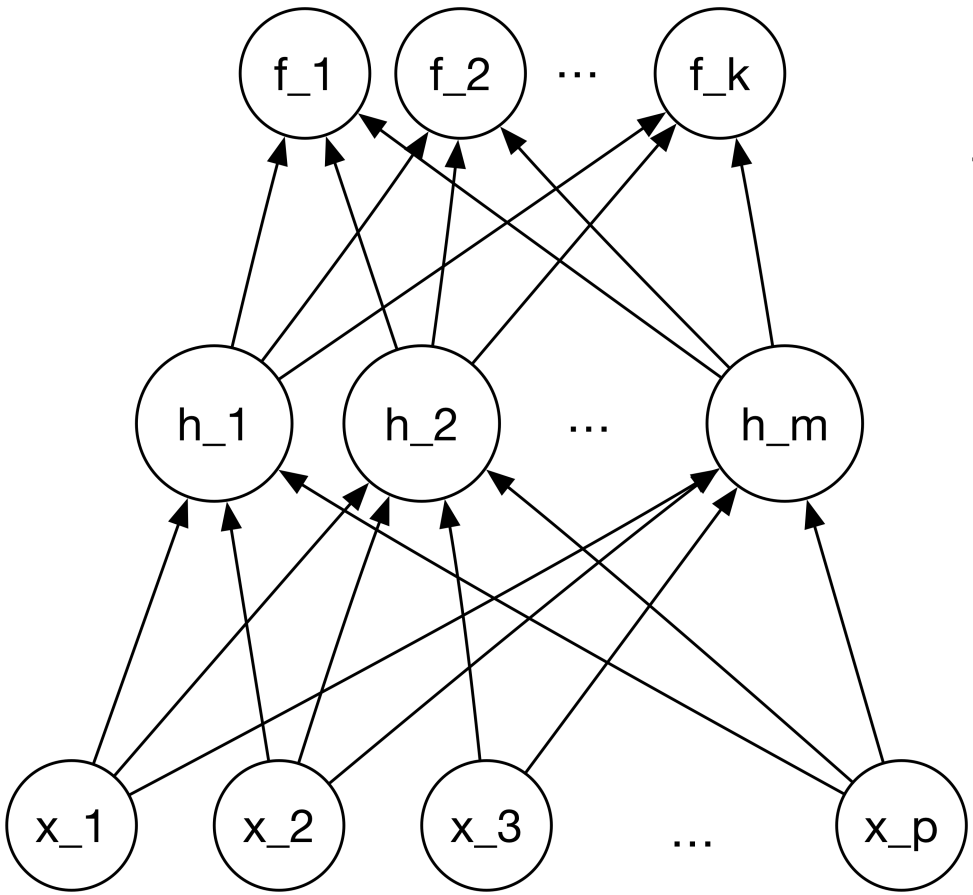
The network is organized into *input*, *hidden* and *output* layers.

Feed-forward Neural Networks



Units h_m represent a *hidden layer*, which we can interpret as a *derived* non-linear representation of the input data as we saw before.

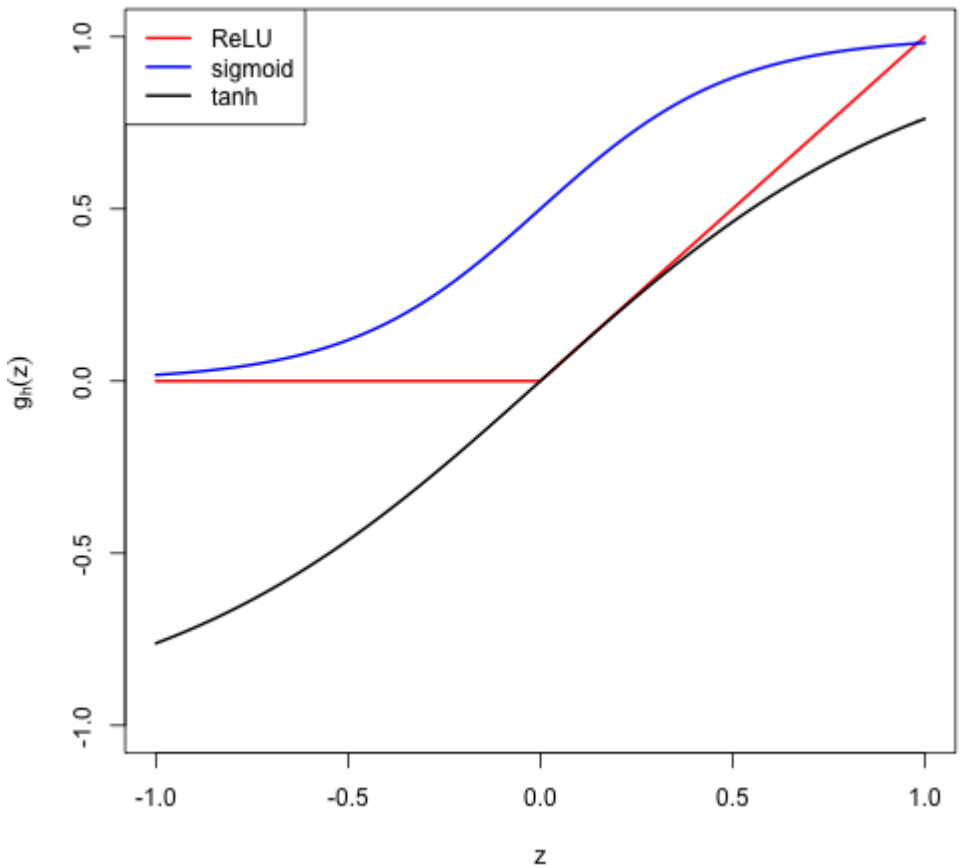
Feed-forward Neural Networks



Function g_h is an *activation* function used to introduce non-linearity to the representation.

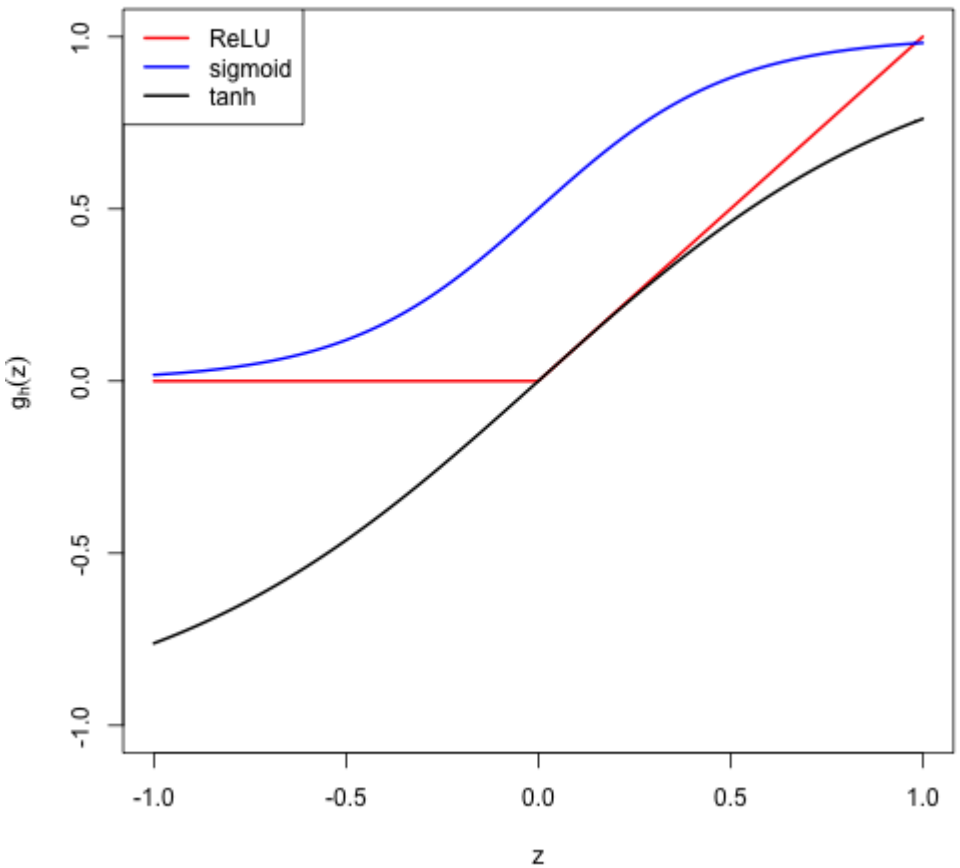
Feed-forward Neural Networks

Historically, the sigmoid activation function was commonly used $g_h(v) = \frac{1}{1+e^{-v}}$ or the hyperbolic tangent.

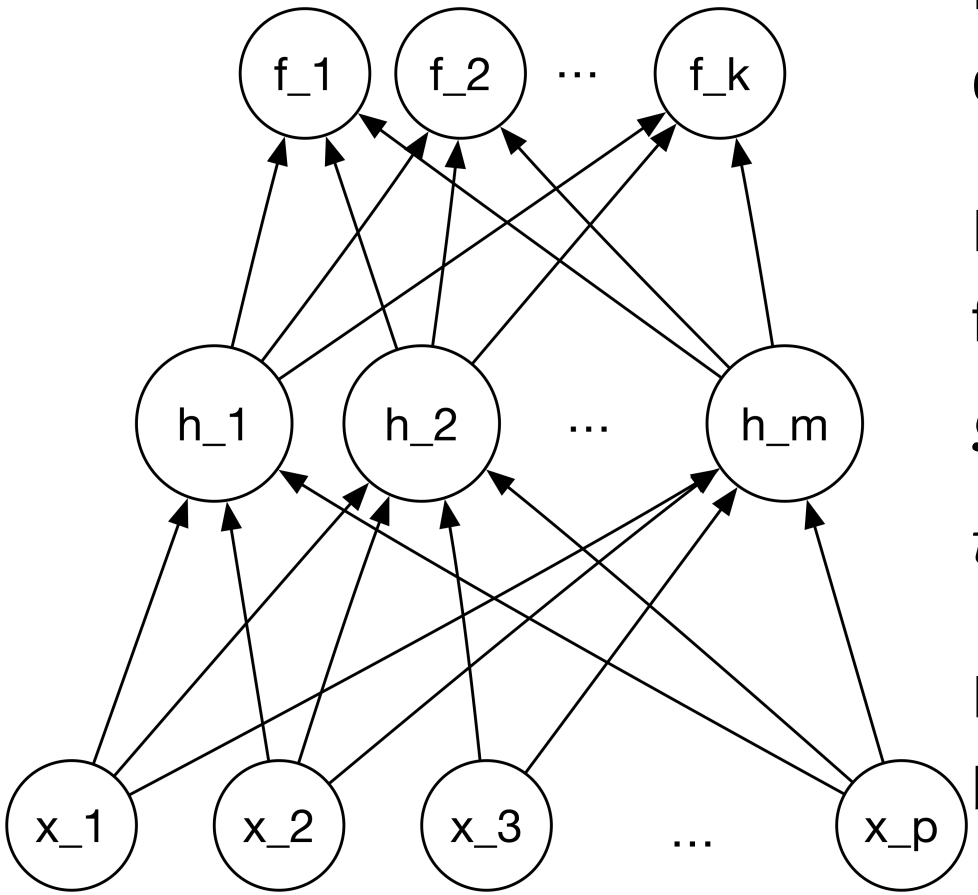


Feed-forward Neural Networks

Nowadays, a rectified linear unit (ReLU) $g_h(v) = \max\{0, v\}$ is used more frequently in practice. (there are many extensions)



Feed-forward Neural Networks



Function g_f used in the output layer depends on the outcome modeled.

For classification a *soft-max* function can be used $g_{fk}(t_k) = \frac{e^{t_k}}{\sum_{l=1}^{K} e^{t_k}}$ where $t_k = \mathbf{w}'_{2k}\mathbf{h}$.

For regression, we may take g_{fk} to be the identify function.

Feed-forward Neural Networks

The single-layer feed-forward neural network has the same parameterization as the PPR model,

Activation functions g_h are much simpler, as opposed to, e.g., smoothing splines as used in PPR.

Feed-forward Neural Networks

A classic result of the Neural Network literature is the universal function representational ability of the single-layer feed-forward neural network with ReLU activation functions (Leshno et al. 1993).

Feed-forward Neural Networks

A classic result of the Neural Network literature is the universal function representational ability of the single-layer feed-forward neural network with ReLU activation functions (Leshno et al. 1993).

However, the number of units in the hidden layer may be exponentially large to approximate arbitrary functions.

Feed-forward Neural Networks

Empirically, a single-layer feed-forward neural network has similar performance to kernel-based methods like SVMs.

This is not usually the case once more than a single-layer is used in a neural network.

Fitting with back propagation

In modern neural network literature, the graphical representation of neural nets we saw above has been extended to *computational graphs*.

Fitting with back propagation

In modern neural network literature, the graphical representation of neural nets we saw above has been extended to *computational graphs*.

Especially useful to guide the design of general-use programming libraries for the specification of neural nets.

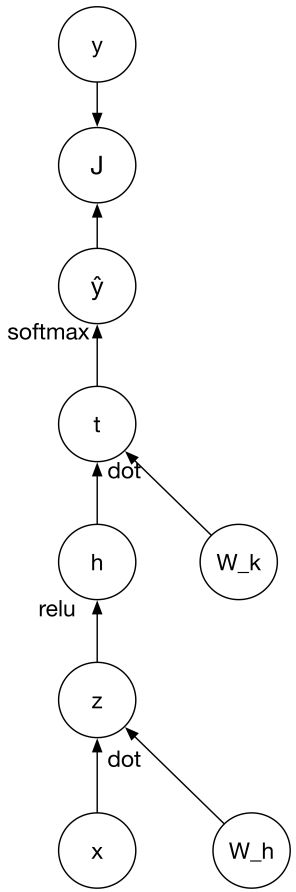
Fitting with back propagation

In modern neural network literature, the graphical representation of neural nets we saw above has been extended to *computational graphs*.

Especially useful to guide the design of general-use programming libraries for the specification of neural nets.

They have the advantage of explicitly representing all operations used in a neural network which then permits easier specification of gradient-based algorithms.

Fitting with back propagation



Fitting with back propagation

Gradient-based methods based on stochastic gradient descent are most frequently used to fit the parameters of neural networks.

Fitting with back propagation

Gradient-based methods based on stochastic gradient descent are most frequently used to fit the parameters of neural networks.

These methods require that gradients are computed based on model error.

Fitting with back propagation

Gradient-based methods based on stochastic gradient descent are most frequently used to fit the parameters of neural networks.

These methods require that gradients are computed based on model error.

The layer-wise propagation of error is at the core of these gradient computations.

Fitting with back propagation

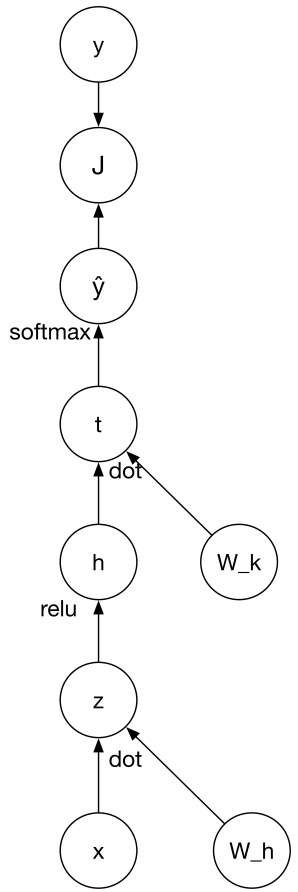
Gradient-based methods based on stochastic gradient descent are most frequently used to fit the parameters of neural networks.

These methods require that gradients are computed based on model error.

The layer-wise propagation of error is at the core of these gradient computations.

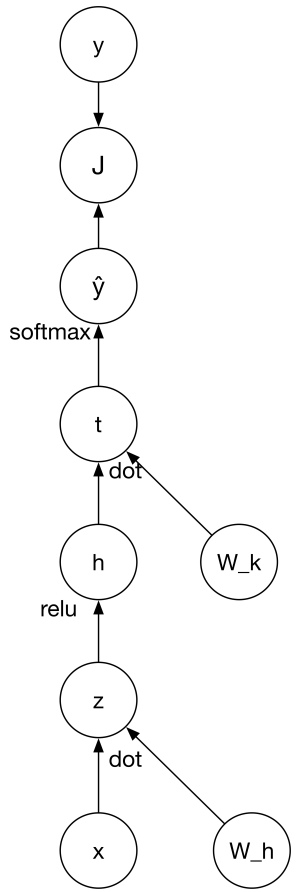
This is called back-propagation.

Fitting with back propagation



Assume we have a current estimate of model parameters, and we are processing one observation x (in practice a small batch of observations is used).

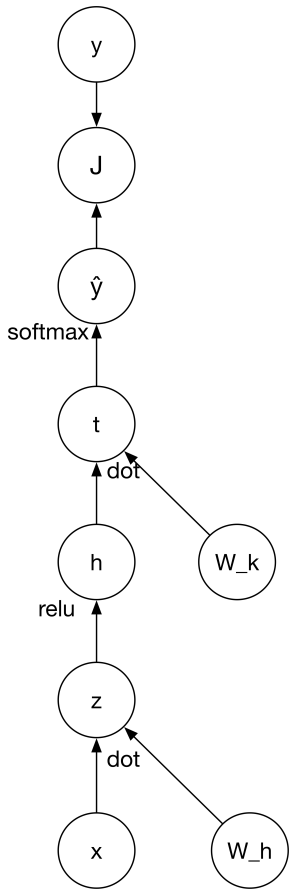
Fitting with back propagation



First, to perform back propation we must compute the error of the model on observation x given the current set of parameters.

To do this we compute all activation functions along the computation graph from the bottom up.

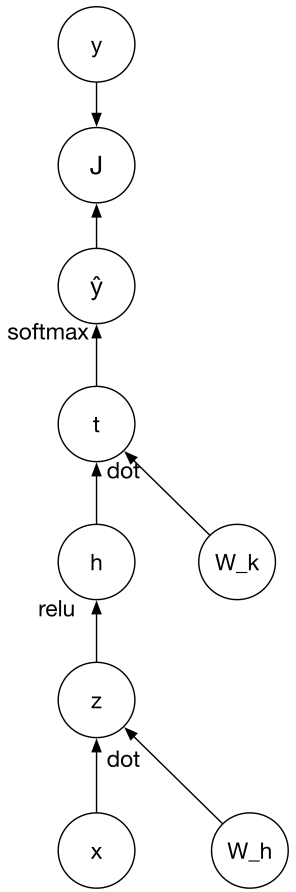
Fitting with back propagation



Once we have computed output $\hat{y}$, we can compute error (or, generally, cost) $J(y, \hat{y})$.

Once we do this we can walk back through the computation graph to obtain gradients of cost J with respect to any of the model parameters applying the chain rule.

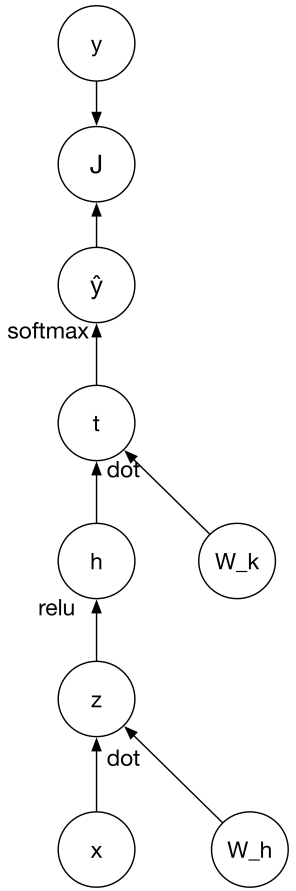
Fitting with back propagation



We will continously update a gradient vector ∇.

First, we set $\nabla \leftarrow \nabla_{\hat{y}} J$

Fitting with back propagation



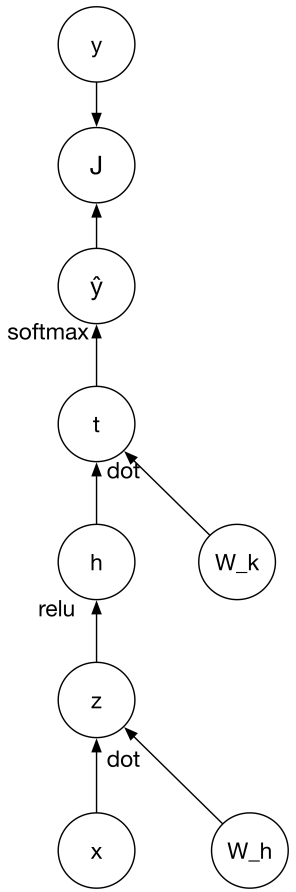
Next, we need the gradient $\nabla_t J$

We apply the chain rule to obtain $\nabla_t J = \nabla \odot f'(t)$

- f' is the derivative of the softmax function
- $\odot$ is element-wise multiplication.

Set $\nabla \leftarrow \nabla_t J$.

Fitting with back propagation

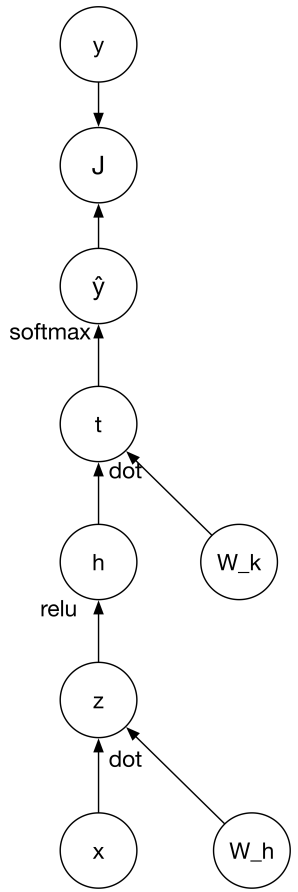


Next, we want to compute $\nabla_{W_k} J$.

We can do so using the gradient we just computed ∇ since $\nabla_{W_k} J = \nabla_t J \nabla_{W_k} t$.

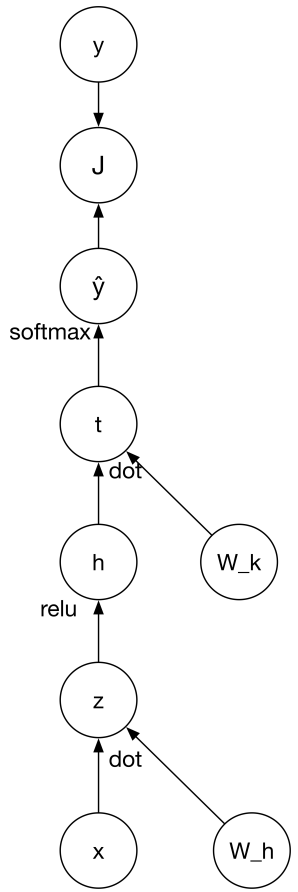
In this case, we get $\nabla_{W_k} J = \nabla \mathbf{h}'$.

Fitting with back propagation



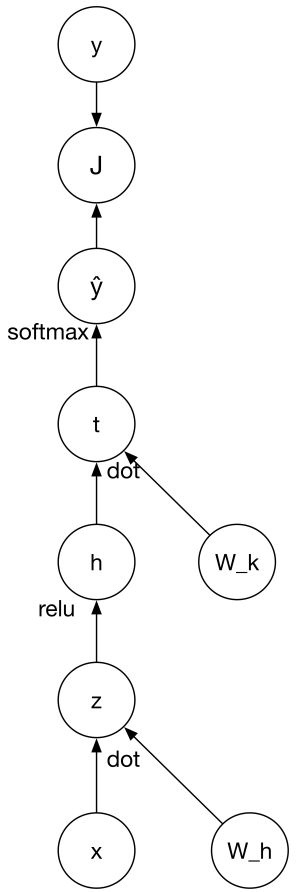
At this point we have computed gradients for the weight matrix W_k from the hidden layer to the output layer, which we can use to update those parameters as part of stochastic gradient descent.

Fitting with back propagation



Once we have computed gradients for weights connecting the hidden and output layers, we can compute gradients for weights connecting the input and hidden layers.

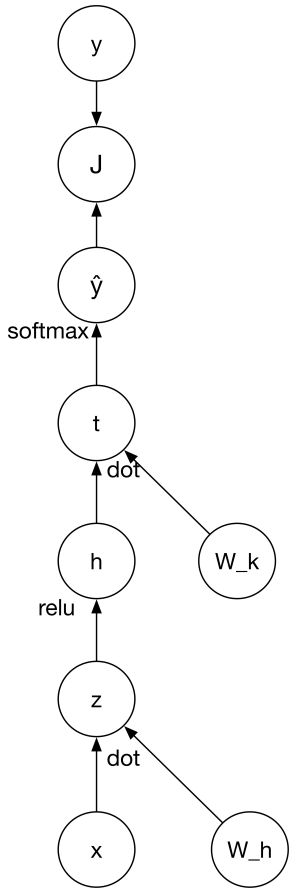
Fitting with back propagation



We require $\nabla_{\mathbf{h}} J$, we we can compute as $W_k' \nabla$ since ∇ currently has value $\nabla_t J$.

At this point we can set $\nabla \leftarrow \nabla_{\mathbf{h}} J$.

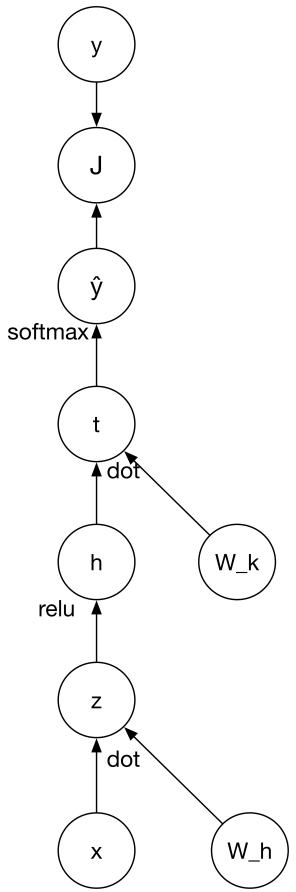
Fitting with back propagation



Finally, we set $\nabla \leftarrow \nabla_{\mathbf{z}} J = \nabla \cdot g'(z)$ where g' is the derivative of the ReLU activation function.

This gives us $\nabla_{W_h} J = \nabla \mathbf{x}'$.

Fitting with back propagation



At this point we have propagated the gradient of cost function J to all parameters of the model

We can thus update the model for the next step of stochastic gradient descent.

Practical Issues

Stochastic gradient descent (SGD) based on back-propagation algorithm as shown above introduces some complications.

Scaling

The scale of inputs x effectively determines the scale of weight matrices W

Scale can have a large effect on how well SGD behaves.

In practice, all inputs are usually standardized to have zero mean and unit variance before application of SGD.

Initialization

With properly scaled inputs, initialization of weights can be done in a somewhat reasonable manner

Randomly choose initial weights in $[-.7, .7]$.

Overfitting

As with other highly-flexible models we have seen previously, feed-forward neural nets are prone to overfit data.

Overfitting

As with other highly-flexible models we have seen previously, feed-forward neural nets are prone to overfit data.

We can incorporate penalty terms to control model complexity to some degree.

Architecture Design

A significant issue in the application of feed-forward neural networks is that we need to choose the number of units in the hidden layer.

Architecture Design

A significant issue in the application of feed-forward neural networks is that we need to choose the number of units in the hidden layer.

We saw above that a wide enough hidden layer is capable of perfectly fitting data.

Architecture Design

A significant issue in the application of feed-forward neural networks is that we need to choose the number of units in the hidden layer.

We saw above that a wide enough hidden layer is capable of perfectly fitting data.

We will also see later that in many cases making the neural network deeper instead of wider performs better.

Architecture Design

A significant issue in the application of feed-forward neural networks is that we need to choose the number of units in the hidden layer.

We saw above that a wide enough hidden layer is capable of perfectly fitting data.

We will also see later that in many cases making the neural network deeper instead of wider performs better.

In this case, models may have significantly fewer parameters, but tend to be much harder to fit.

Architecture Design

Ideal network architectures are task dependent

Require much experimentation

Judicious use of cross-validation methods to measure expected prediction error to guide architecture choice.

Multiple Minima

As opposed to other learning methods we have seen so far, the feed-forward neural network yields a non-convex optimization problem.

Multiple Minima

As opposed to other learning methods we have seen so far, the feed-forward neural network yields a non-convex optimization problem.

This will lead to the problem of multiple local minima in which methods like SGD can suffer.

Multiple Minima

As opposed to other learning methods we have seen so far, the feed-forward neural network yields a non-convex optimization problem.

This will lead to the problem of multiple local minima in which methods like SGD can suffer.

We will see later in detail a variety of approaches used to address this problem.

Multiple Minima

As opposed to other learning methods we have seen so far, the feed-forward neural network yields a non-convex optimization problem.

This will lead to the problem of multiple local minima in which methods like SGD can suffer.

We will see later in detail a variety of approaches used to address this problem.

Here, we present a few rule of thumbs to follow.

Multiple Minima

The local minima a method like SGD may yield depend on the initial parameter values chosen.

Multiple Minima

The local minima a method like SGD may yield depend on the initial parameter values chosen.

One idea is to train multiple models using different initial values and make predictions using the model that gives best expected prediction error.

Multiple Minima

The local minima a method like SGD may yield depend on the initial parameter values chosen.

One idea is to train multiple models using different initial values and make predictions using the model that gives best expected prediction error.

A related idea is to average the predictions of this multiple models.

Multiple Minima

The local minima a method like SGD may yield depend on the initial parameter values chosen.

One idea is to train multiple models using different initial values and make predictions using the model that gives best expected prediction error.

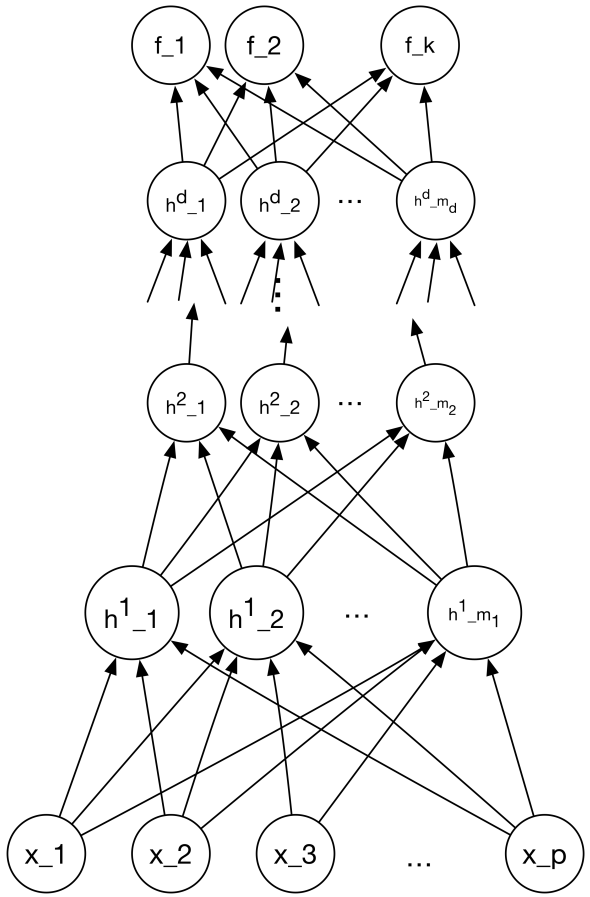
A related idea is to average the predictions of this multiple models.

Finally, we can use *bagging* as described in a previous session to create an ensemble of neural networks to circumvent the local minima problem.

Summary

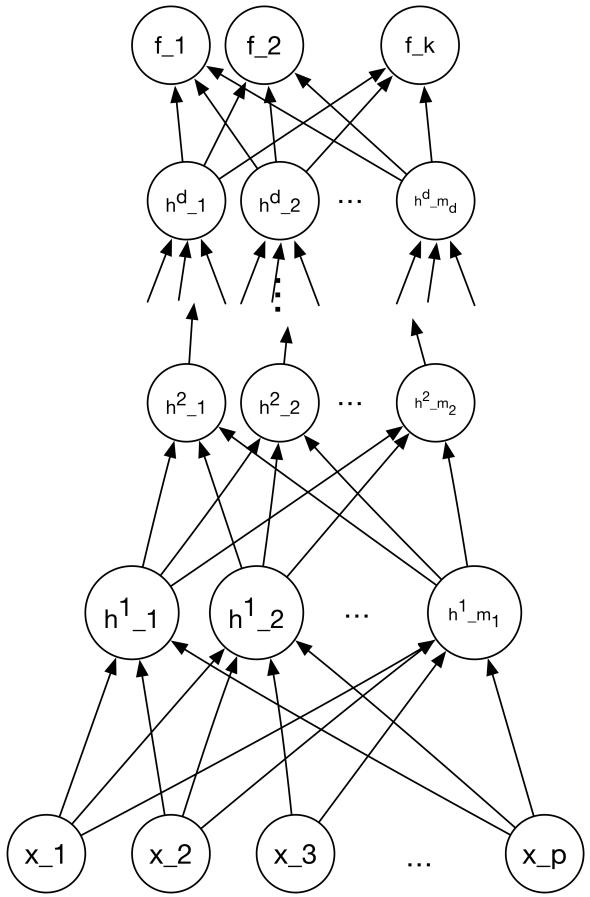
- Neural networks are representationally powerful prediction models.
- They can be difficult to optimize properly due to the non-convexity of the resulting optimization problem.
- Deciding on network architecture is a significant challenge. We'll see later that recent proposals use deep, but thinner networks effectively. Even in this case, choice of model depth is difficult.
- There is tremendous excitment over recent excellent performance of deep neural networks in many applications.

Deep Feed-Forward Neural Networks



The general form of feed-forward network can be extended by adding additional *hidden layers*.

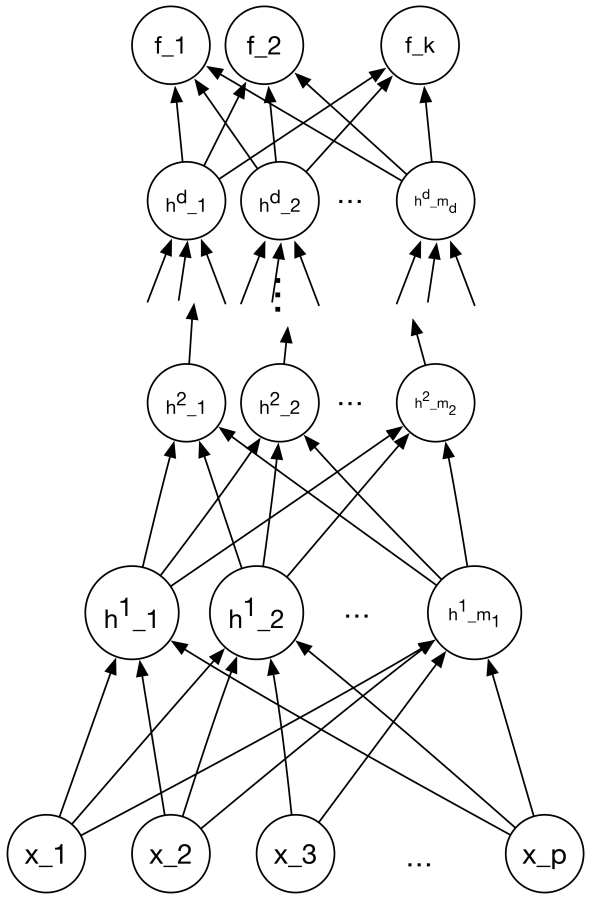
Deep Feed-Forward Neural Networks



The same principles we saw before:

- We arrange computation using a computing graph
- Use Stochastic Gradient Descent
- Use Backpropagation for gradient calculation along the computation graph.

Deep Feed-Forward Neural Networks

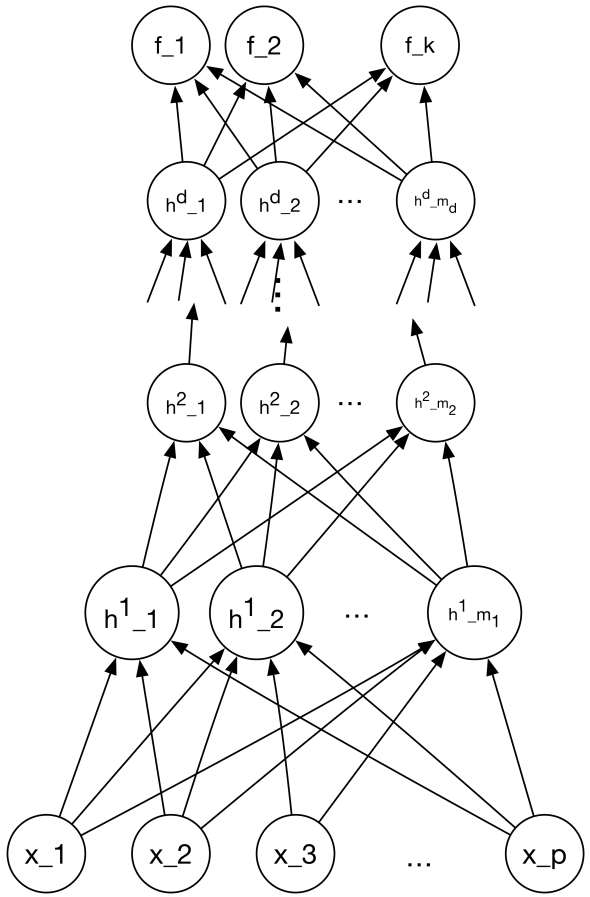


Empirically, it is found that by using more, thinner, layers, better expected prediction error is obtained.

However, each layer introduces more linearity into the network.

Making optimization markedly more difficult.

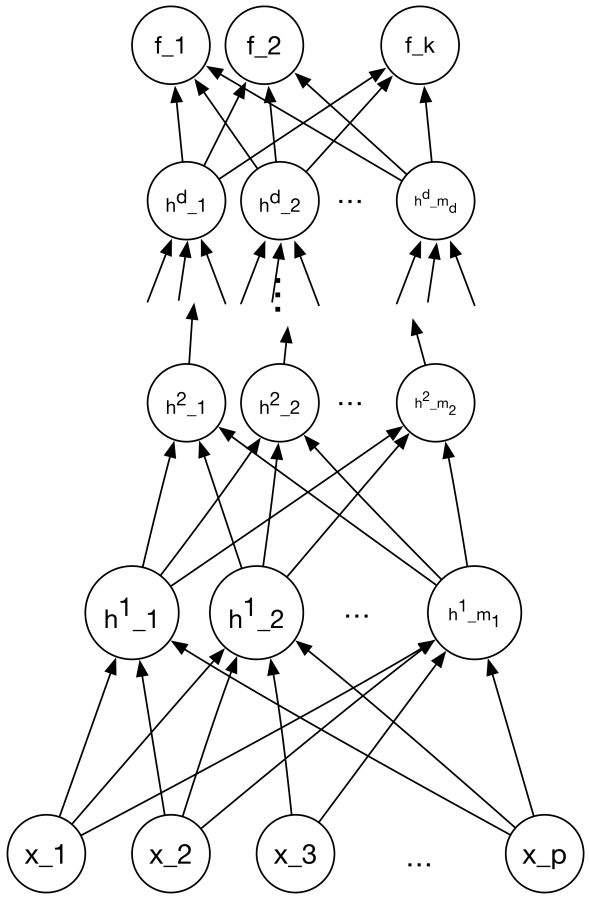
Deep Feed-Forward Neural Networks



We may interpret hidden layers as progressive derived representations of the input data.

Since we train based on a loss-function, these derived representations should make modeling the outcome of interest progressively easier.

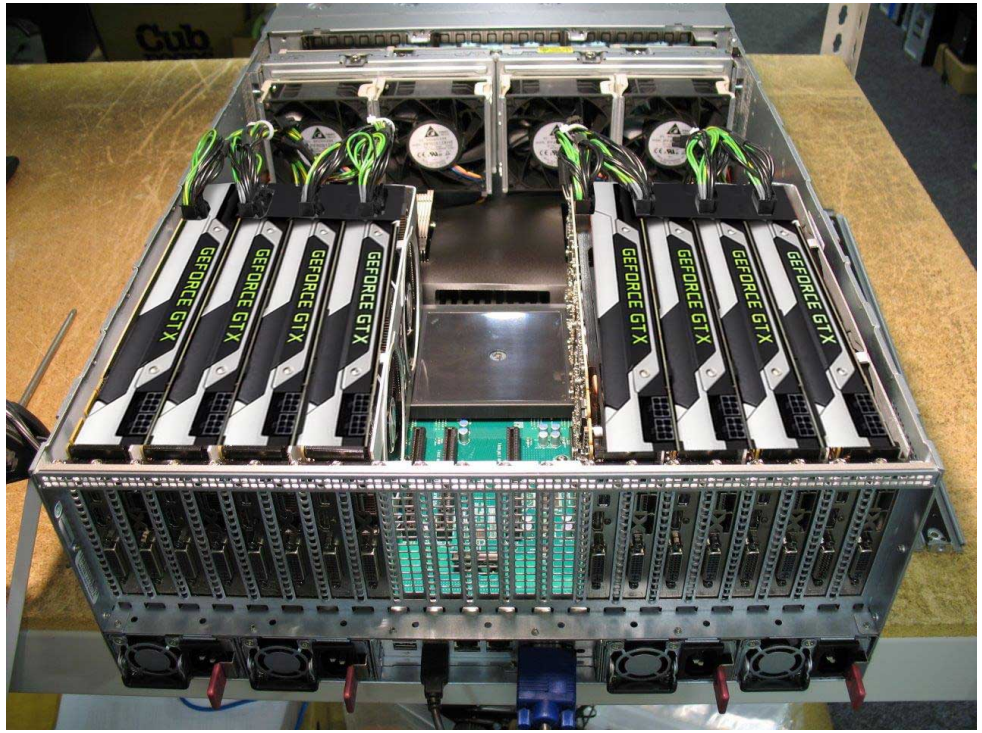
Deep Feed-Forward Neural Networks



In many applications, these derived representations are used for model interpretation.

Deep Feed-Forward Neural Networks

Advanced parallel computation systems and methods are used in order to train these deep networks, with billions of connections.



Deep Feed-Forward Neural Networks

Advanced parallel computation systems and methods are used in order to train these deep networks, with billions of connections.

The applications we discussed previously build this type of massive deep network.

Deep Feed-Forward Neural Networks

Advanced parallel computation systems and methods are used in order to train these deep networks, with billions of connections.

The applications we discussed previously build this type of massive deep network.

They also require massive amounts of data to train.

Deep Feed-Forward Neural Networks

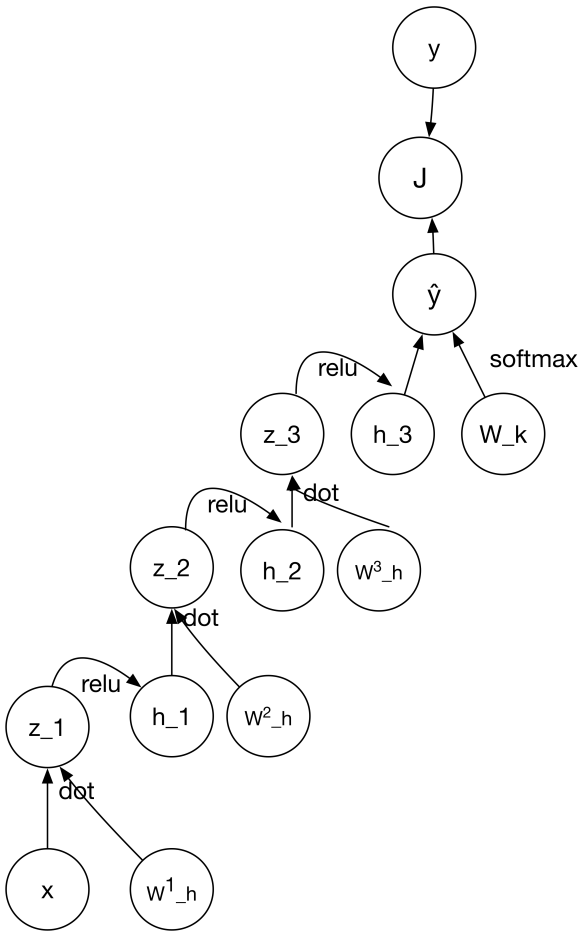
Advanced parallel computation systems and methods are used in order to train these deep networks, with billions of connections.

The applications we discussed previously build this type of massive deep network.

They also require massive amounts of data to train.

However, this approach can still be applicable to moderate datasizes with careful network design, regularization and training.

Supervised Pre-training

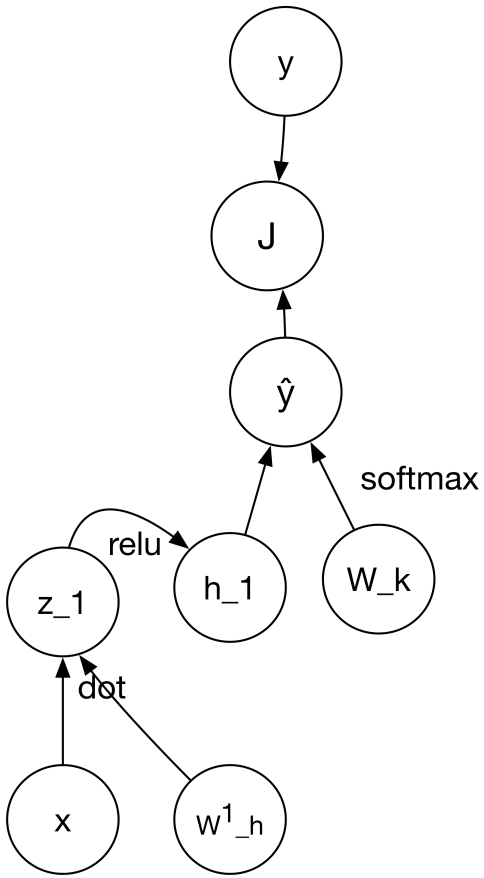


A clever idea for training deep networks.

Train each layer successively on the outcome of interest.

Use the resulting weights as initial weights for network with one more additional layer.

Supervised Pre-training

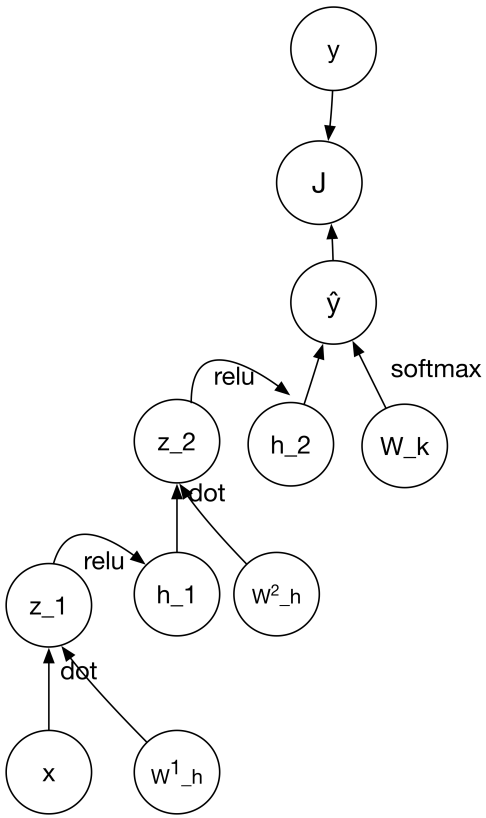


Train the first layer as a single layer feed forward network.

Weights initialized as standard practice.

This fits W_h^1.

Supervised Pre-training



Now train two layer network.

Weights W_h^1 are initialized to result of previous fit.

Supervised Pre-training

This procedure continues until all layers are trained.

Hypothesis is that training each layer on the outcome of interest moves the weights to parts of parameter space that lead to good performance.

Minimizing updates can ameliorate dependency problem.

Supervised Pre-training

This is one strategy others are popular and effective

- Train each layer as a single layer network using the hidden layer of the previous layer as inputs to the model.
- In this case, no long term dependencies occur at all.
- Performance may suffer.

Supervised Pre-training

This is one strategy others are popular and effective

- Train each layer as a single layer on the hidden layer of the previous layer, but also add the original input data as input to every layer of the network.
- No long-term dependency
- Performance improves
- Number of parameters increases.

Parameter Sharing

Another method for reducing the number of parameters in a deep learning model.

When predictors X exhibit some internal structure, parts of the model can then share parameters.

Parameter Sharing

Two important applications use this idea:

- Image processing: local structure of nearby pixels
- Sequence modeling: structure given by sequence

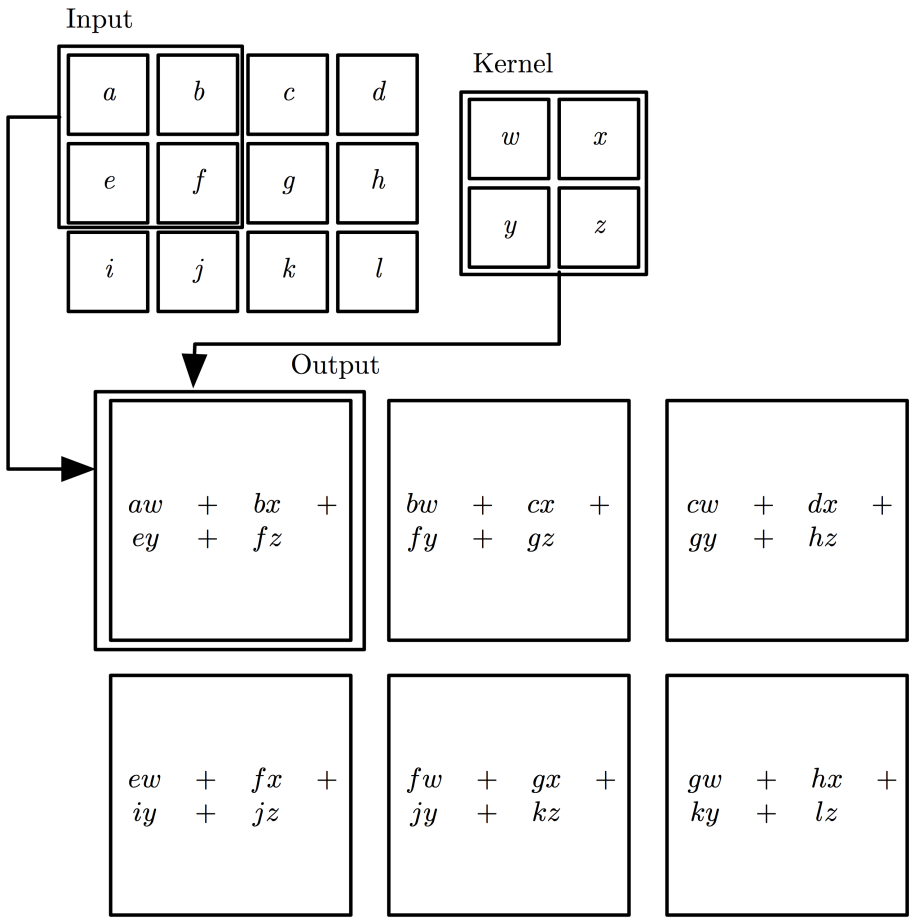
The latter includes modeling of time series data.

Parameter Sharing

Convolutional Networks are used in imaging applications.

Input is pixel data.

Parameters are shared across nearby parts of the image.

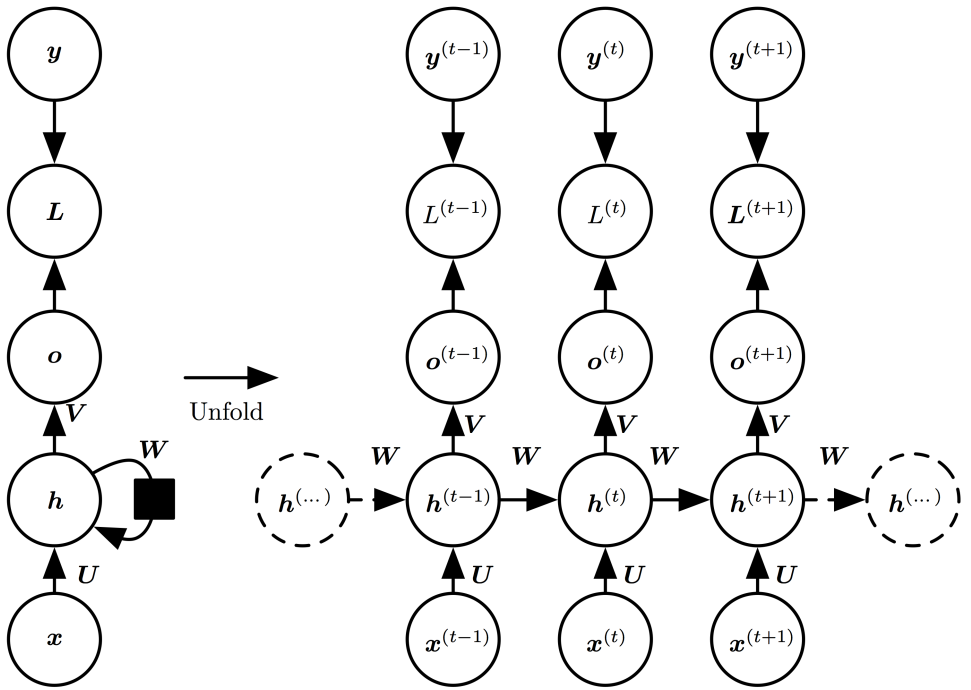


Recurrent Networks

Recurrent Networks are used in sequence modeling applications.

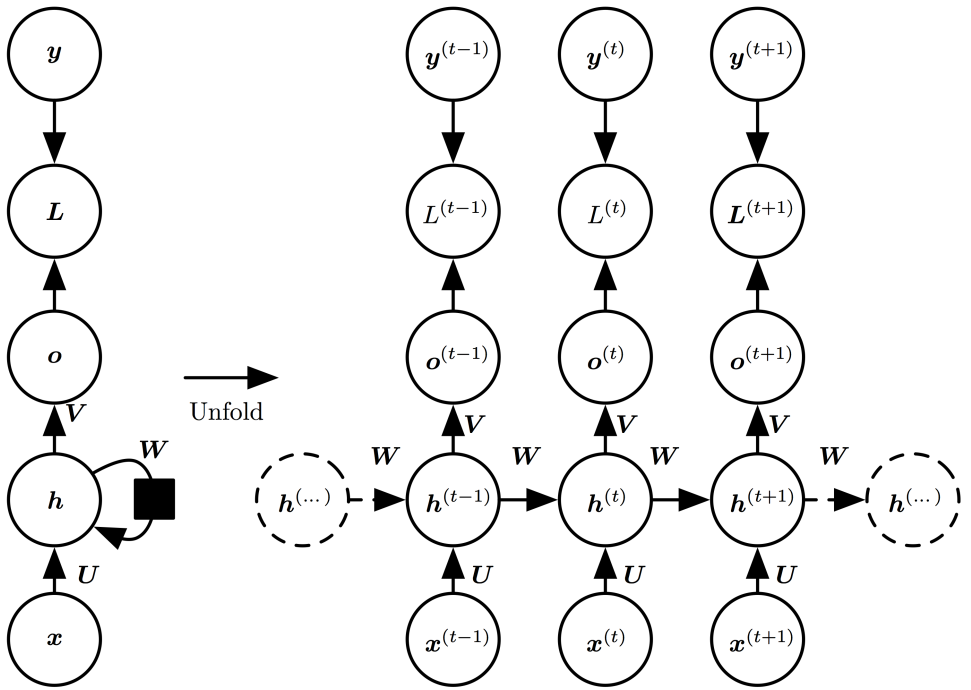
For instance, time series and forecasting.

Parameters are shared across a time lag.



Recurrent Networks

The *long short-term memory* (LSTM) model is very popular in time series analysis



Example

Learn to add: "55+22=77"

https://keras.rstudio.com/articles/examples/addition_rnn.html

Example

Learn to add: "55+22=77"

https://keras.rstudio.com/articles/examples/addition_rnn.html

Addition encoded as sequence of one-hot vectors:

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 5    0    0    0    0    0    0    0    1    0     0
## 5    0    0    0    0    0    0    0    1    0     0
## +    0    1    0    0    0    0    0    0    0     0
## 2    0    0    0    0    1    0    0    0    0     0
## 2    0    0    0    0    1    0    0    0    0     0
##   [,11] [,12]
```

Example

Learn to add: "55+22=77"

https://keras.rstudio.com/articles/examples/addition_rnn.html

Result encoded as sequence of one-hot vectors

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 7    0    0    0    0    0    0    0    0    0     1
## 7    0    0    0    0    0    0    0    0    0     1
##   [,11] [,12]
## 7     0     0
## 7     0     0
```

Example

Learn to add: "55+22=77"

https://keras.rstudio.com/articles/examples/addition_rnn.html

Result encoded as sequence of one-hot vectors

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 7    0    0    0    0    0    0    0    0    0     1
## 7    0    0    0    0    0    0    0    0    0     1
##   [,11] [,12]
## 7     0     0
## 7     0     0
```

This is a **sequence-to-sequence** model. Perfect application for

Example

```
## _________________________________________________________
## Layer (type)             Output Shape          Param #
## =========================================================
## lstm_10 (LSTM)           (None, 128)           72192
## _________________________________________________________
## repeat_vector_5 (Repeat (None, 3, 128)        0
## _________________________________________________________
## lstm_11 (LSTM)           (None, 3, 128)        131584
## _________________________________________________________
## time_distributed_5 (Tim (None, 3, 12)         1548
## _________________________________________________________
## activation_5 (Activatio (None, 3, 12)         0
```

Summary

Deep Learning is riding a big wave of popularity.

State-of-the-art results in many applications.

Best results in applications with massive amounts of data.

However, newer methods allow use in other situations.

Summary

Many of recent advances stem from computational and technical approaches to modeling.

Keeping track of these advances is hard, and many of them are ad-hoc.

Not straightforward to determine a-priori how these technical advances may help in a specific application.

Require significant amount of experimentation.

Summary

The interpretation of hidden units as *representations* can lead to insight.

There is current research on interpreting these to support some notion of statistical inference.

Excellent textbook: http://deeplearningbook.org