Insurance analytics

Neural networks

Katrien Antonio
LRisk - KU Leuven and ASE - University of Amsterdam

May 14, 2019

Acknowledgement

- Some of the figures in this presentation are taken from
  - Michael A. Nielsen (2015). Neural networks and deep learning. Determination Press.
  - all the beautiful work by prof. Taylor Arnold

Today's mission

- Today's mission:
  - de-mystify neural networks
  - sketch different types of neural networks and their applications
  - a discussion of specific considerations to keep in mind when using these predictive modeling techniques with frequency/severity data.

A famous challenge

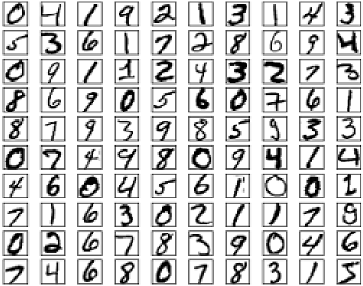
Recognizing handwritten digits

- Consider this

- Humans have a primary visual cortex (V_1) with 140M neurons, with 10s billions connections.
- Next to V_1, also V_2, V_3, V_4 and V_5, doing complex image processing.
- A supercomputer in our head!

A famous challenge

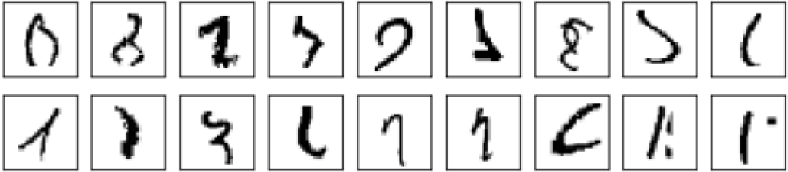
Recognizing handwritten digits



The goal: infer rules for recognizing handwritten digits.

A famous challenge

Recognizing handwritten digits



The winner: well-designed neural nets classify 9 979 out of 10 000 images correctly (in 2013).

Types of neural networks

What's in a name?

- Different types of neural networks and their applications:
  - ANN: Artificial Neural Network

    for regression and classification problems, with vectors as input data
  - CNN: Convolutional Neural Network

    for image processing, image/face/... recognition, with images as input data
  - RNN: Recurrent Neural Network

    for sequential data such as text or time series.

Types of neural networks
A bit of history

[HERE COMES HISTORY]

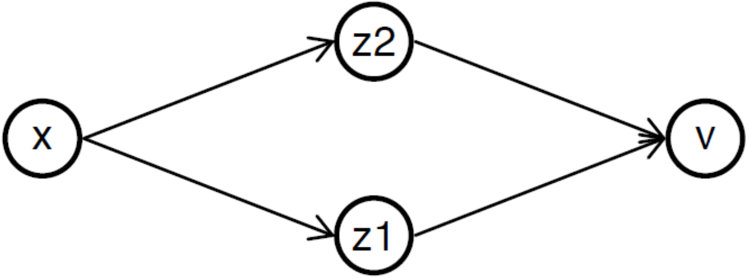
Artificial neural networks

A simple neural network

- De-mystify artificial neural networks (ANNs):
  - a collection of inter-woven linear models
  - extending linear approaches to detect non-linear interactions in high-dimensional data.

Artificial neural networks

A simple neural network



The goal: predict a scalar response y from scalar input x.

Artificial neural networks

A simple neural network

- Some terminology:
  - x is the input layer
  - v is output layer
  - middle layers are hidden layers
  - four neurons: x, z_1, z_2 and v.

Artificial neural networks

A simple neural network

- First, apply two independent linear models:

$$\begin{aligned} z_1 &= b_1 + x \cdot w_1 \\ z_2 &= b_2 + x \cdot w_2, \end{aligned}$$

  using four parameters: two intercepts and two slopes.

- Next, construct another linear model with the z_j as inputs:

$$\hat{y} := v \; = \; b_3 + z_1 \cdot u_1 + z_2 \cdot u_2.$$

Artificial neural networks

A simple neural network

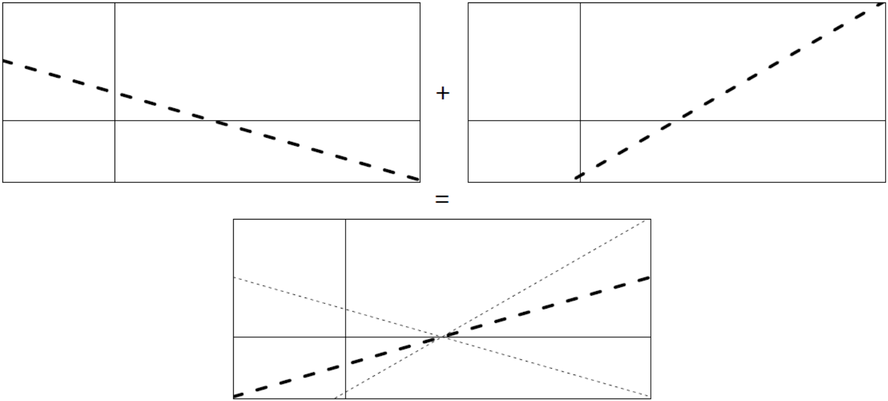
- Putting it all together:

$$
\begin{aligned}
v &= b_3 + z_1 \cdot u_1 + z_2 \cdot u_2 \\
&= b_3 + (b_1 + x \cdot w_1) \cdot u_1 + (b_2 + x \cdot w_2) \cdot u_2 \\
&= (b_3 + u_1 \cdot b_1 + u_2 \cdot b_2) + (w_1 \cdot u_1 + w_2 \cdot u_2) \cdot x \\
&= (\text{intercept}) + (\text{slope}) \cdot x.
\end{aligned}
$$

- Model is over-parametrized, with infinitely many ways to describe same model.

Artificial neural networks

A simple neural network



Artificial neural networks
Activation function

- Capture non-linear relationships between x and y:

$$v = b_3 + \sigma(z_1) \cdot u_1 + \sigma(z_2) \cdot u_2,$$

where $\sigma(.)$ is an activation function, a mapping from $\mathbb{R}$ to $\mathbb{R}$.

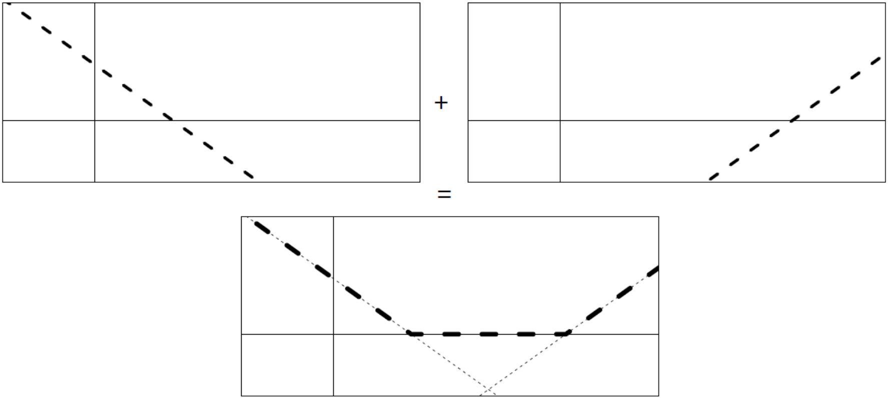
- For example, the rectified linear unit (ReLU) activation function:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

Adding an activation function greatly increases the set of possible relations between x and v!

Artificial neural networks

Activation function



More activation functions: sigmoid, hyperbolic tan, leaky rectified linear unit, maxout.

Artificial neural networks
Artificial vs biological

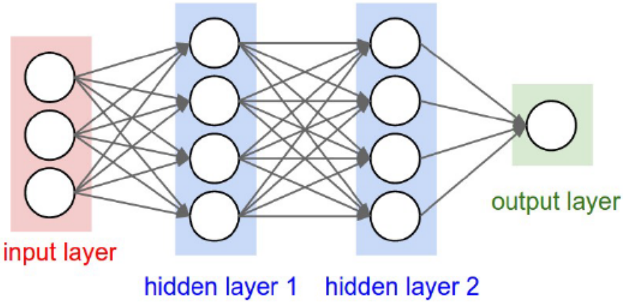
[HERE THE ANALOGY WITH BIO STUFF]

Artificial neural networks
The architecture

- Artificial Neural networks (NNs):
  - a collection of neurons
  - organized into an ordered set of layers
  - directed connections pass signals between neurons in adjacent layers
  - to train: update parameters describing the connections by minimizing loss function over training data
  - to predict: pass $\boldsymbol{x}_i$ to first layer, output of final layer is $\hat{y}_i$.
- The network is *dense* if XXX.

Artificial neural networks

The architecture



A basic neural network. Source : http://blog.christianperone.com

This is a feedforward neural network - no loops!

Artificial neural networks

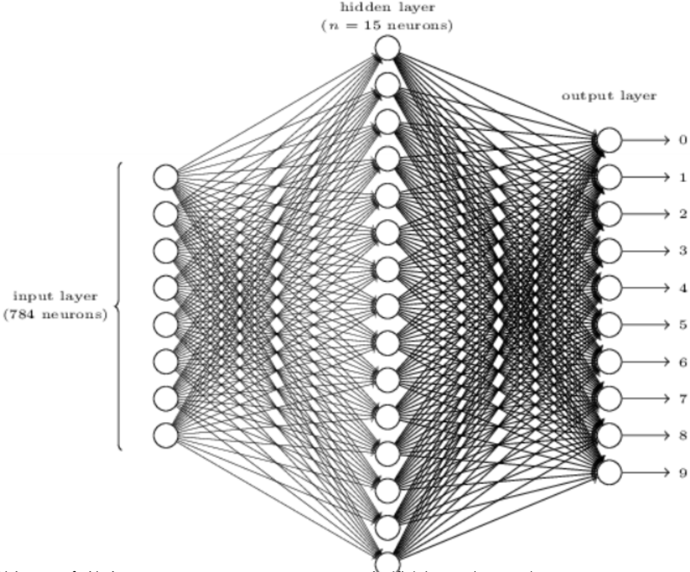
Back to the famous challenge

- The goal: try to recognize individual hand-written digits
  - take e.g. 28 by 28 greyscale image
  - $784 = 28 \times 28$ input neurons, with intensities between 0 (white) and 1 (black)
  - output layer has 10 layers

    neuron with highest activation value fires.

Artificial neural networks

Back to the famous challenge



Stochastic gradient descent

Basic idea - GD

- We want to find

$$\min_{w} f(w),$$

then gradient descent updates as follows (cfr. lecture on *boosting methods*)

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w f(w_{\text{old}}),$$

with learning rate η. Move in the direction the function locally decreases the fastest!

- A good choice for NNs because faster second-order methods involve the Hessian and this is infeasible when having tons of parameters.

Stochastic gradient descent

From GD to SGD

- Let $\mathcal{L}(w; y_i, x_i)$ be the loss function and w the parameters. For example,

$$\begin{aligned} \mathcal{L}(w; y_i, x_i) &= \frac{1}{2n} \sum_i \left(\hat{y}_i(w) - y_i\right)^2 \\ &= \frac{1}{n} \sum_i \mathcal{L}_i(w; y_i, x_i). \end{aligned}$$

- With the gradient descent update rule

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w \mathcal{L}(w),$$

where $\nabla_w \mathcal{L}(w) = \frac{1}{n} \sum_i \nabla_w \mathcal{L}_i$.

Stochastic gradient descent

From GD to SGD

- Run over each of the training observations, and get update $w^{(n)}$

$$
\begin{array}{rcl}
\left(w^{(0)} - (\eta/n) \cdot \nabla_{w^{(0)}} \mathcal{L}_1\right) & \rightarrow & w^{(1)} \\
\left(w^{(1)} - (\eta/n) \cdot \nabla_{w^{(0)}} \mathcal{L}_2\right) & \rightarrow & w^{(2)} \\
 & \vdots & \\
\left(w^{(n-1)} - (\eta/n) \cdot \nabla_{w^{(0)}} \mathcal{L}_n\right) & \rightarrow & w^{(n)},
\end{array}
$$

with the gradients calculated separately for each training input.

Stochastic gradient descent

From GD to SGD

- With a final tweak

$$
\begin{aligned}
\left(w^{(0)} - (\eta/n) \cdot \nabla_{w^{(0)}} \mathcal{L}_1\right) \quad &\rightarrow \quad w^{(1)} \\
\left(w^{(1)} - (\eta/n) \cdot \nabla_{w^{(1)}} \mathcal{L}_2\right) \quad &\rightarrow \quad w^{(2)} \\
&\vdots \\
\left(w^{(n-1)} - (\eta/n) \cdot \nabla_{w^{(n-1)}} \mathcal{L}_n\right) \quad &\rightarrow \quad w^{(n)}.
\end{aligned}
$$

- One pass through entire dataset is an epoch.

Stochastic gradient descent

Mini-batches

- Stochastic gradient descent picks randomly m training inputs $x_1, \ldots, x_m$, a mini-batch:

$$\frac{\sum_{j=1}^m \nabla \mathcal{L}_{x_j}}{m} \approx \frac{\sum_x \nabla \mathcal{L}_x}{n} = \nabla C.$$

- Thus,

$$\nabla \mathcal{L} \approx \frac{1}{m} \sum_{j=1}^{m} \nabla \mathcal{L}_{x_j}.$$

With update rule

$$w_{\text{new}} = w_{\text{old}} - \frac{\eta}{m} \sum_j \nabla_w \mathcal{L}(w),$$

where j runs over all training samples in the current mini-batch.

Stochastic gradient descent

- Partition input randomly into disjoint groups $M_1,\ M_2, \ldots, M_{n/m}$.
- Updates:

$$
\begin{aligned}
w_{k+1} &= w_k - \frac{\eta}{m} \sum_{i \in M_1} \nabla \mathcal{L}_i \\
w_{k+2} &= w_{k+1} - \frac{\eta}{m} \sum_{i \in M_2} \nabla \mathcal{L}_i \\
&\vdots \\
w_{k+n/m+1} &= w_{k+n/m} - \frac{\eta}{m} \sum_{i \in M_{n/m}} \nabla \mathcal{L}_i .
\end{aligned}
$$

Backward propagation of errors

- Compute the gradient of the loss function wrt all trainable parameters:
  - tons of parameters
  - need for efficient algorithm to calculate gradient
  - generic algorithm usable for arbitrary number of layers and neurons in each layer.
- The strategy (Rumelhart et al., 1986, Nature)

  backwards propagation of errors or backpropagation

  uses chain rule for derivatives.

Backward propagation of errors

- Using the NN language:
  - intercept called *the bias*
  - slopes called *weights*
  - L layers in total, with input layer denoted as layer 0
  - use a (from *activation*) to denote the output of a given layer.
- Let's start with a single layer network (called an artificial neuron).

Backward propagation of errors

Artificial neuron

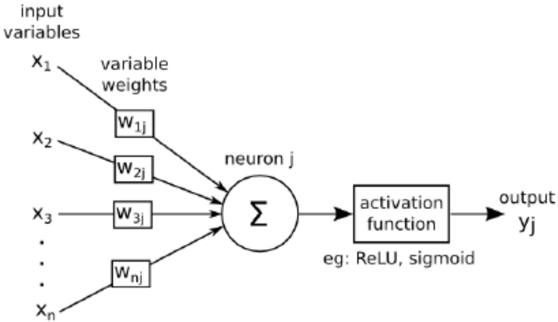
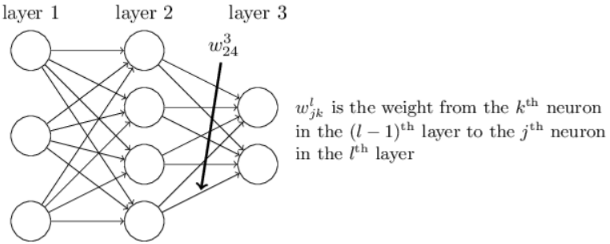


Figure 1: source: andrewjames turner.co.uk

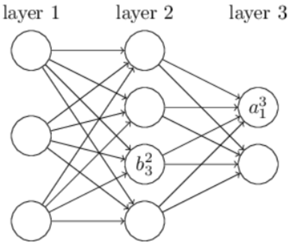
Backward propagation of errors

- Use w^l_{jk} to denote the weight for the connection:
  - from neuron k in layer $(l-1)$
  - to neuron j in layer l.



Backward propagation of errors

- Use
  - b^l_j for the bias of neuron j in layer l
  - a^l_j for the activation of neuron j in layer l.



Backward propagation of errors

- d-dimensional inputs

$$a^0 = x,$$

- weight $w^1_{j,k}$ on kth neuron in layer 0 within jth neuron in layer 1

$$w^1_j = (w^1_{j,1}, \ldots, w^1_{j,d})$$

- with bias term b^1_j

$$\begin{aligned} z^1_j &= w^1_{j,1} \cdot x_1 + \ldots + w^1_{j,d} \cdot x_d + b^1_j \\ &= < w^1_j, x > + b^1_j, \end{aligned}$$

- apply activation function

$$a^1_j = \sigma(z^1_j),$$

the output of neuron j in layer 1.

Backward propagation of errors

- With L layers in total: (in vectorized form)

$$\begin{aligned} z^l &= w^l \circ a^{l-1} + b^l \text{ (the weighted input)} \\ a^l &= \sigma(z^l), \end{aligned}$$

  for l from 1 up to (and including) L.

- Finally,

$$a^L = \hat{y}.$$

Backward propagation of errors

- With a loss function f, e.g. squared error loss

$$f(y, a^L) \;=\; \frac{1}{2n}(y - a^L)^2.$$

- Equations defining backpropagation:

starting point - gradient terms of b_j^l

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_j^l} &= \frac{\partial \mathcal{L}}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} \\ &= \frac{\partial \mathcal{L}}{\partial z_j^l} \cdot 1 = \frac{\partial \mathcal{L}}{\partial z_j^l}.\end{aligned}$$

Thus, problem centers around derivatives wrt z^l!

Backward propagation of errors

- With a loss function f, e.g. squared error loss

$$\mathcal{L}(y, a^L) \;=\; (y - a^L)^2.$$

- Equations defining backpropagation:

starting point - gradient terms of weights w^l_{jk}

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w^l_{jk}} &= \frac{\partial \mathcal{L}}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial w^l_{jk}} \\ &= \frac{\partial \mathcal{L}}{\partial z^l_j} \cdot a^{l-1}_k. \end{aligned}$$

Thus, problem centers around derivatives wrt z^l!

Backward propagation of errors

- The derivatives in layer L are straightforward to calculate

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z_j^L} &= \sum_k \frac{\partial \mathcal{L}}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L} \\
&= \frac{\partial \mathcal{L}}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \\
&= \frac{\partial \mathcal{L}}{\partial a_k^L} \cdot \sigma^{'}(z_j^L).
\end{aligned}
$$

An equation for the error in the output layer!

Backward propagation of errors

- The derivatives wrt z^l can be written as a function of derivatives wrt z^{l+1}.

- Using the chain rule

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z_j^l} &= \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \cdot w_{kj}^{l+1} \sigma'(z_j^l),
\end{aligned}
$$

where we use

$$
z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}.
$$

Backward propagation of errors

- The backpropagation alogrithm:

  1. Input x, set a^1 for the input layer.
  2. Feedforward: for each $l = 2, 3, \ldots, L$ compute
  $$z^l \quad = \quad w^l \cdots a^{l-1} + b^l \text{ and } a^l = \sigma(z^l).$$
  3. Output error: compute $\frac{\partial \mathcal{L}}{\partial z_j^L} = \frac{\partial \mathcal{L}}{\partial a_k^L} \cdot \sigma^{'}(z_j^L)$.
  4. Backpropagate the error: for each $l = L-1, \ldots, 2$ compute $\frac{\partial \mathcal{L}}{\partial z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \cdot w_{kj}^{l+1} \sigma^{'}(z_j^l)$.
  5. Output: the gradient of the loss function $\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \cdot a_k^{l-1}$ and $\frac{\partial \mathcal{L}}{\partial b_j^l} = \frac{\partial \mathcal{L}}{\partial z_j^l}$.

Improving SGD and regularization

- Several improvements covered in the literature
  - to reduce overfitting
  - to reduce getting stuck in local saddle points
  - to converge in a smaller number of epochs.

Improving SGD and regularization

- Update the weight initialization $\rightsquigarrow$ might drastically improve performance of a model with many layers.
- Early stopping to prevent overfitting:
  - calculate validation after each epoch
  - stop when this no longer improves
  - NNs are motivated by an optimization problem, but do not attempt to solve the optimization task.

Improving SGD and regularization

- Prevent overfitting via regularization (cfr. lecture on Lasso and friends)
  - add (e.g.) ℓ_2-norm

$$f_\lambda(w, b) \quad = \quad f(w, b) + \frac{\lambda}{2} \cdot \|w\|_2^2,$$

    with weights w and bias terms b, where only weights are penalized

  - with gradient

$$\nabla_w f_\lambda \quad = \quad \nabla_w f(w, b) + \lambda \cdot w$$

    such that

$$\begin{aligned} w_{\text{new}} \quad &\leftarrow \quad w_{\text{old}} - \eta \cdot \nabla_w f_\lambda(w_{\text{old}}) \\ &\leftarrow \quad w_{\text{old}} - \eta \cdot [\nabla_w f(w_{\text{old}}) + \lambda \cdot w] \\ &\leftarrow \quad [1 - \eta \cdot \lambda] \cdot w_{\text{old}} - \eta \cdot f(w_{\text{old}}). \end{aligned}$$

Improving SGD and regularization

- Dropout:
  - set activations to zero
  - randomly, with fixed probability p
  - both in forward propagation as well as backpropagation
  - only in training, all nodes turned on during prediction.
- Adjust SDG itself, or use different optimization strategy.

Improving SGD and regularization
Dropout

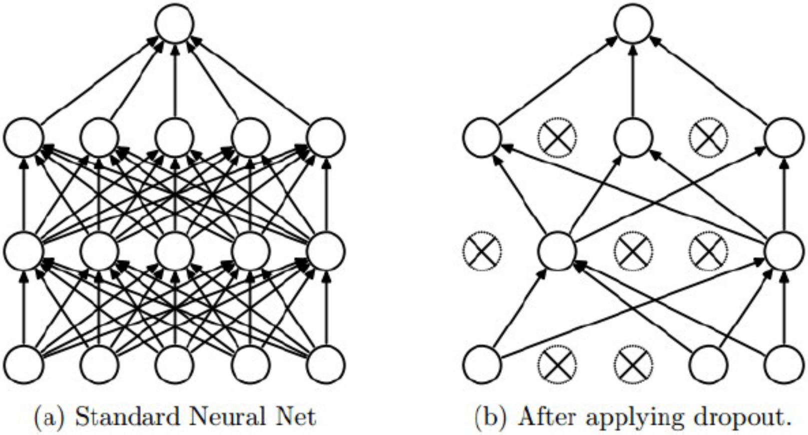


Figure 5: Dropout - source: http://blog.christianperone.com/

Where to finetune your ANN?

Classification with Artificial Neural Networks

- Use a multi–valued output layer for classification tasks.
- One-hot encoding applied to output vector y

$$\begin{pmatrix} 2 \\ 4 \\ \vdots \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

- Use softmax function as activation in final layer

$$\begin{aligned} a_j^L &= \text{softmax}(z_j^L) \\ &= \frac{e^{z_j}}{\sum_k e^{z_k}}. \end{aligned}$$

Classification with neural networks

- Instead of using squared error loss, use categorical cross-entropy

$$f(a^L, y) \;\;=\;\; -\sum_i y_i \cdot \log(a_k^L),$$

  a the loss function.

- Work out the derivatives to set-up the backpropagation with this loss function.

Convolutional neural networks
The motivation

- Prediction tasks with images as inputs are a popular application of NNs.
- With a limited number of input pixels, put a weight on each pixel and use ANN.
- With large images use convolutional layers.

Convolutional neural networks

A convolution?

- The discrete convolution between f and g is

$$(f * g)(x) \;=\; \sum_{t} f(t) \cdot g(x+t).$$

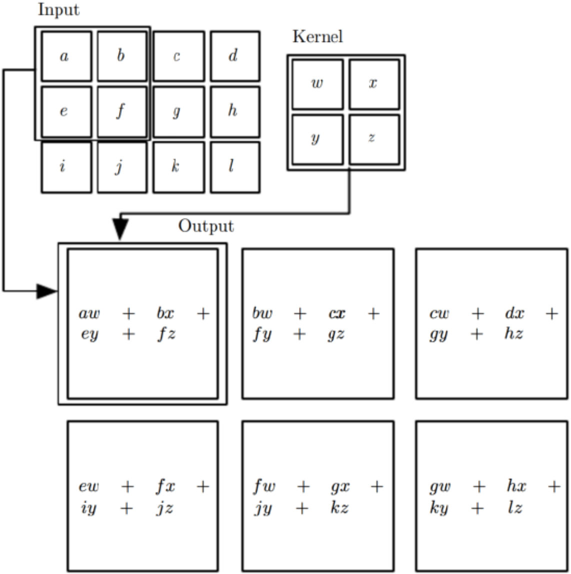
- With 2-dimensional signals (e.g. images), consider 2D-convolutions

$$(K * I)(i,j) \;=\; \sum_{m,n} K(m,n) \cdot I(i+m, j+n),$$

with K a convolutional kernel applied to a $2D$ signal (or image) I.

Convolutional neural networks

A convolution pictured



Convolutional neural networks

- With large images use convolutional layers:
  - apply small set of weights to subsections of the image
  - same weights across the image
  - use a kernel matrix, e.g. for a black and white image

$$K = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

  Take pixel value and subtract from this the pixel value to its immediate lower right.

Convolutional neural networks

[HERE SOME PICTURES]

Convolutional neural networks

- In a CNN:
  - apply several such kernels in a layer
  - with weights learned during training of the algorithm
  - different convolutions pick up different features (e.g. edges, texture, basic object types).
- With input image in color: kernel matrix K is 3-dimensional array with weights applied to each color channel.

Convolutional neural networks

Mathematical notation

- Input data using two indices to represent the two spatial dimensions

$$a^0_{i,j} \;\; = \;\; x_{i,j}.$$

- With kernels of size k_1-by-k_2 (e.g. 3-by-3 in image processing), denote the output of the first hidden layer

$$z^1_{i,j,k} \;\; = \;\; \sum_{m=0}^{k_1} \sum_{n=0}^{k_2} w^1_{m,n} \cdot a^0_{i,j} + b^1_k.$$

- Re-parametrize z^1 as

$$z^1_q \;\; = \;\; z^1_{i,j,k},\; q = (i-1) \cdot W \cdot K + (j-1) \cdot K + j,$$

with K the total number of kernels and W the width of the input image. [SOME MORE STUFF HERE?]

Recurrent neural networks

- To infer sequential data such as text or time series.
- A hidden layer at time t depends on
  - the entry at time t, x_t
  - but also on the same hidden layer at time $t-1$
  - or on the output at time $t-1$.

Recurrent neural networks
Unrolled

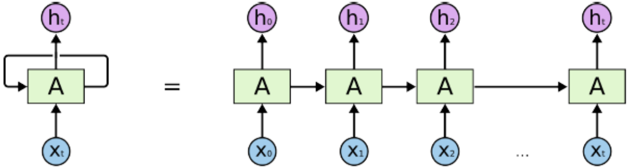


Figure 19: Unrolled representation of a RNN. Source : Understanding LSTM Networks by Christopher Olah - http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Yes, we CANN1

[SOME STUFF HERE]

Implementation

- The R package `tensorflow`
  - gives access to TensorFlow library in XXX
  - allows to work efficiently with multidimensional arrays
  - allows a generic form of backpropagation.
- The R package `keras`
  - gives access to Keras
  - comes on top of TensorFlow
  - building NNs out of layer objects
  - ANNs, CNNs, RNNs.