ELEMENTS OF DATA SCIENCE AND STATISTICAL LEARNING

SPRING 2020

Week 13

OUTLINE

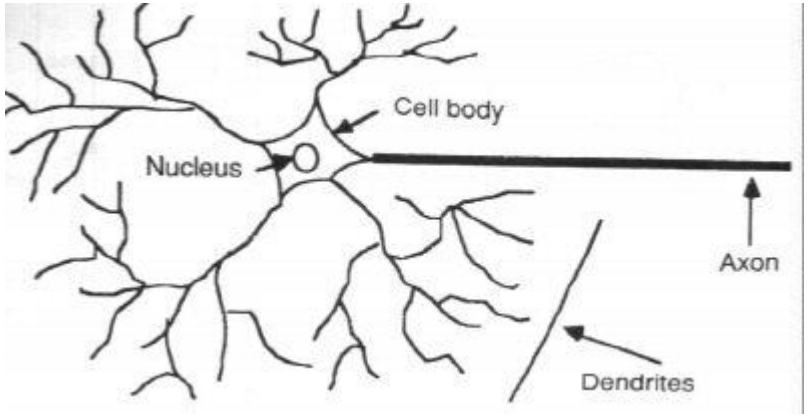
- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

NEURAL NETWORKS: THE BEGINNING

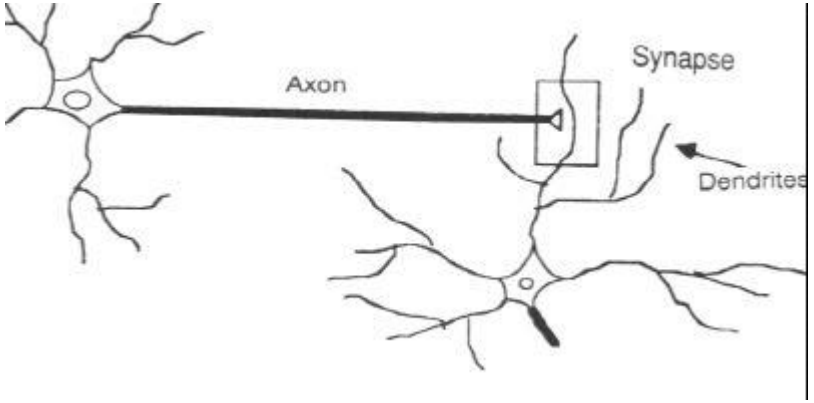
- Artificial Neural Network (ANN) is one of the earliest approaches in machine learning/artificial intelligence
  - Inspired by early advances in neuroscience during the first half of the 20th century,
  - By the advent of modern computers in the 1940-50ies…
  - And probably (to some extent) by the early hopes to teach those computers how to think and reason
- The early idea was to build a mathematical/computational model of a brain, or rather of interacting neurons – hence the name
  - One may argue how accurate this model might be, but this is not very relevant for us
- ANN is a very rich and flexible model with uneven history
  - Fell out of grace and was all but dead more than once
  - Always came back stronger, modern day "Deep Learning" techniques are refinement of ANN
  - A very good and accessible overview of neural networks and (mostly) their philosophy and history is given in the first chapter of the "Deep Learning" book (free online at http://www.deeplearningbook.org/). If you really want to know more about deep learning, just keep reading past chapter 1…

INSPIRATION: A NEURON

- A neuron grows out multiple processes: one of them is the *axon* and the rest are *dendrites*. Information is encoded by spikes of electrical activity.
  - Dendrites are "inputs": they *sense* the spikes;
  - Axon is the output: depending on the combination and strength of the signals on the dendrites, neuron may start "firing" the spikes through its axon

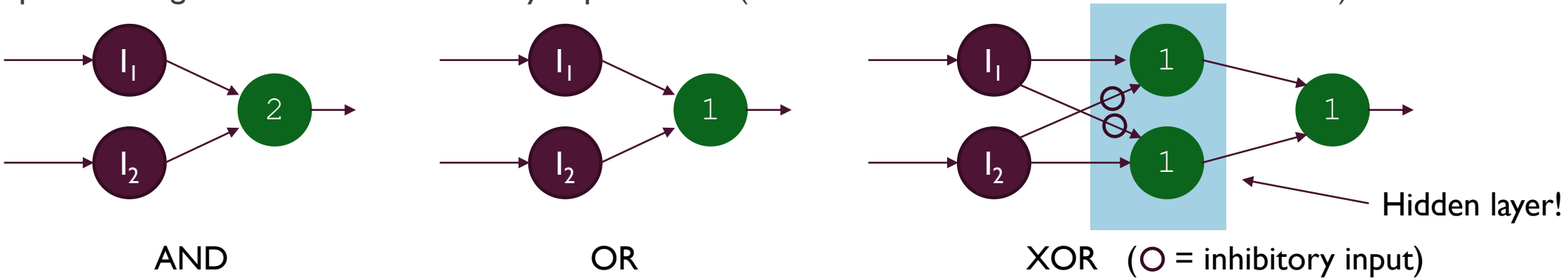


- Neurons are organized into complex networks: dendrites (inputs) of one neuron form synapses ("contacts") with axons (outputs) of other neuron(s)
  - Both excitatory and inhibitory synapses exist
  - Information is transferred through and transformed/computed upon by the network of neurons
  - Learning, forming memories etc = forming/strengthening synapses



ARTIFICIAL NEURON

- The simplest model was introduced in 1943 by McCulloch and Pitts
  - In the initial model only binary inputs and outputs were considered, and some restrictions on weights were in place; the original model also defined an inhibitory input with absolute priority (i.e. with inhibitory input on, the output is always 0)
  - Inputs $I_1,\ldots,I_p$ are summed up, $S = \sum_{k=1}^{p} w_k I_k$ (here we use w=1)
  - A threshold T is applied: output $O = f(S) = \begin{cases} 1 \text{ if } S \geq T \text{ and no inhibition} \\ 0 \text{ otherwise} \end{cases}$ [*f(x)* is a *transfer function* – a simple step function in this case]
- Simple rules/logical functions can be easily implemented (number in the unit denotes the threshold T):



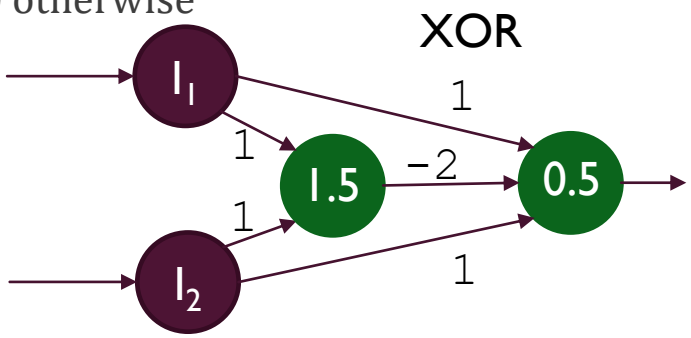
AND OR XOR (O = inhibitory input)

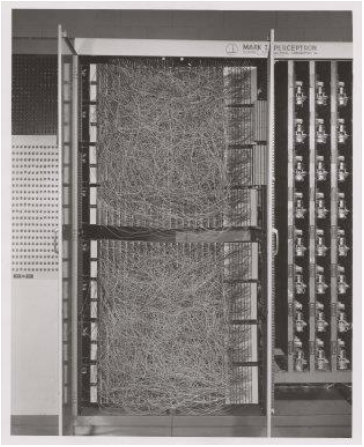
OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

THE PERCEPTRON

- A modification to the original model (by Frank Rosenblatt at Cornell Aeronautical Lab, 1957)
  - No dedicated inhibitory input
  - Arbitrary weights w_k (both positive and negative weights are allowed); $O = f(S) = \begin{cases} 1 \text{ if } S \geq T \\ 0 \text{ otherwise} \end{cases}$
  - And most importantly: learning algorithm!
- Famous!
  - NYT: "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

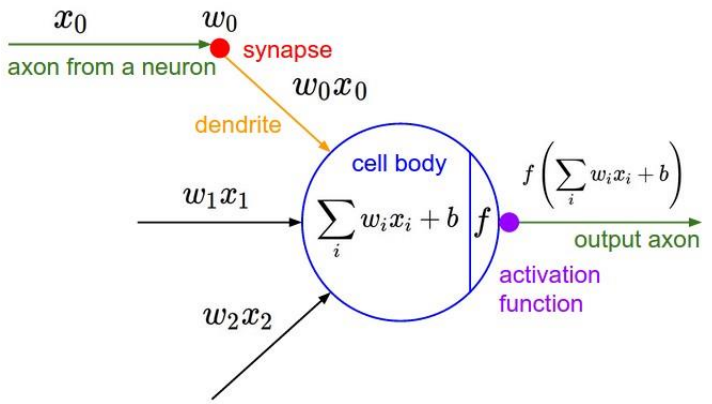




Mark I Perceptron
Could recognize simple images!

PERCEPTRON LEARNING ALGORITHM

- Given the training data $D=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\ldots,(\boldsymbol{x}_n,y_n)\}$, where $\boldsymbol{x}_j$ is a p-dimensional vector of predictors:
  - Initialize weights w_i to 0 or some random numbers
  - For each example j=1…n:
    - Calculate output $\hat{y}_j = f(\boldsymbol{x}_j) = f(\boldsymbol{w} \cdot \boldsymbol{x}_j)$
    - Update the weights $w_i \leftarrow w_i + (y_j - \hat{y}_j)x_{ji}$
  - Note that:
    - We are still considering binary variables here
    - We do not update weights for predictors that are 0 (when $x_{ji}=0$, it does not contribute to the output $f(x_j)$!)
    - If true outcome is 1 and we predicted 0, we increase the weights (so we will eventually cross the threshold from below!)
    - If true outcome is 0 and we predicted 1, we decrease the weights (so we will eventually cross the threshold from above)
  - *Online learning* is possible (learn as examples come one by one)!

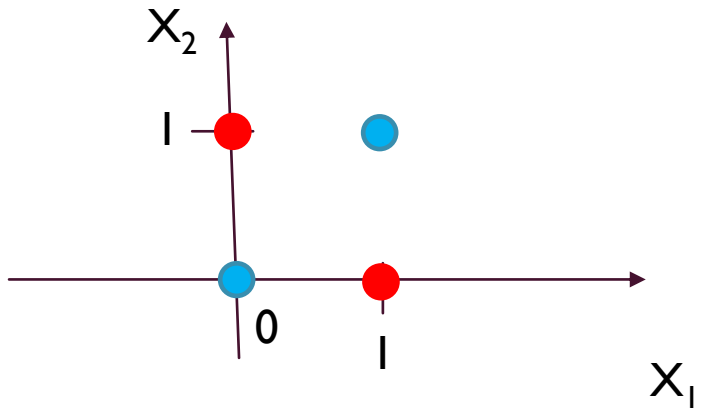


- The algorithm *converges*, but not always…
  - What condition do you think is required for convergence?

DECISION BOUNDARY OF A SINGLE LAYER PERCEPTRON

- The outcome 1 is computed as $\boldsymbol{w} \cdot \boldsymbol{x} > T$, or equivalently we can introduce the *bias* coefficient $w_0 = -T$ at each unit and write $w_0 + \boldsymbol{w} \cdot \boldsymbol{x} > 0$
- This is a *linear boundary*.
  - Data must be linearly separable (or nearly separable)
  - If data are indeed separable, many equivalent boundaries may exist. Support vector classifier is a regularization of the same problem: finds among those boundaries the one with the greatest margin!

This is why we need multiple layers in order to implement XOR: no linear boundary can separate output 0 (blue) from from 1 (red)

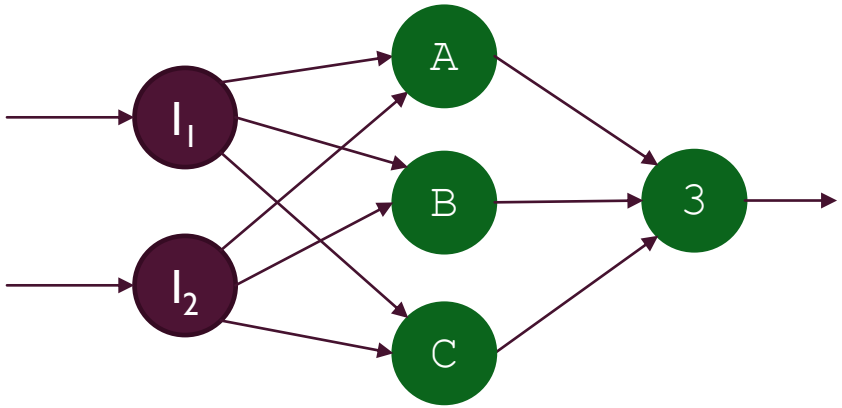


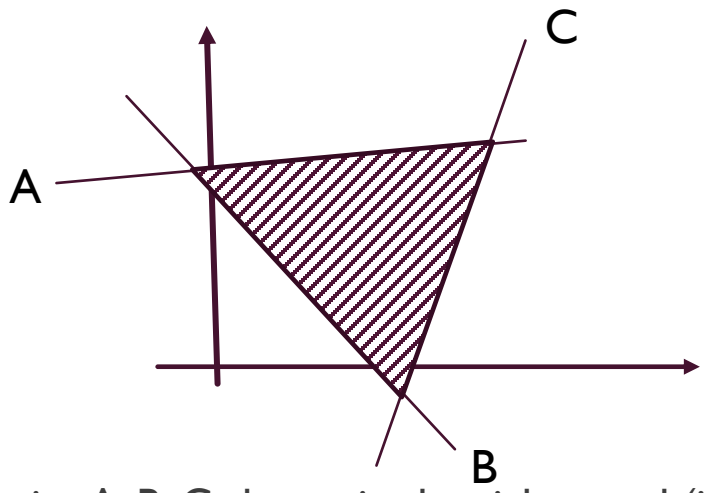
OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

DECISION BOUNDARY OF A MULTI-LAYER PERCEPTRON

- Consider the following multi-layer network:

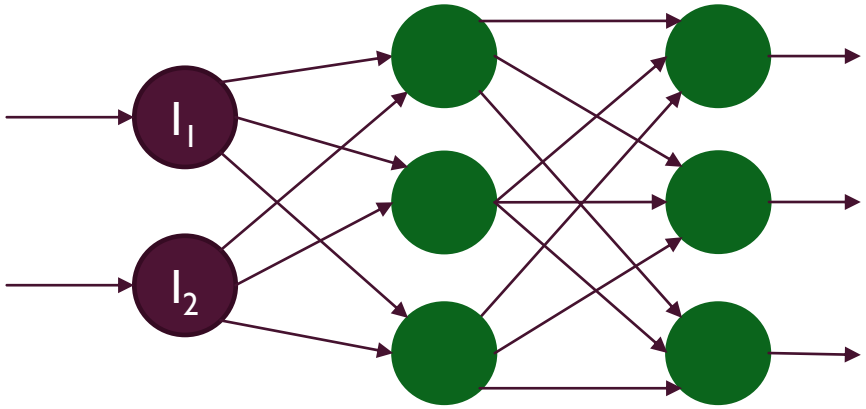




- Let us assume that the units A, B, C define, respectively, the decision boundaries A, B, C shown in the right panel (it is always possible, just choose the appropriate vector of weights ***w*** orthogonal to a desired line)
- Let us further assume that the unit A outputs 1 when x is below the line A and units B, C output 1 when x is above their respective decision boundaries (either side can be chosen to output 1 by changing signs of the coefficients)
- The output node is a trivial summation unit: takes all inputs with equal weights w=1 and has the threshold 3 (equivalently, w_0=-3). In other words, it outputs 1 only when all its 3 inputs are equal to 1. This corresponds to the shaded area in the plot.
- Multilayer perceptron can implement very complex decision boundaries and disjoint regions (try coming up with an example for the latter case)!

LEARNING MULTIPLE CLASSES

- Multi-level classification naturally fits into the ANN framework (less so with step function transfer=binary outcomes 0 or 1, but we will fix that soon enough)
  - Potentially, we might be able to design the right network topology and to train it in such a way that different possible sequences of binary outputs represent different classes
  - Or if we can make the output a continuous function (soon!), we can e.g. predict the outcome that corresponds to the output layer neuron with the largest value



Up to 8 possible outcome classes (000, 001, …)
Or 3 classes if we use which.max(outcome)

AI WINTER

- In 1969, Minsky and Papert publish a famous book, *Perceptron*, in which they present detailed analysis of perceptrons and point out some serious deficiencies.
  - The book is often credited with causing much disenfranchisement with ANN in the research community (but this could have at least as much to do with the typical hype-disillusionment cycle). But one way or another, the funding dried up and research in ANN nearly stopped soon after. Minsky himself coined the "AI winter" term some 15 years later in anticipation of yet another period of disappointment in AI.

  - The book is also often misquoted – Minsky and Papert did not "show" that "XOR is impossible" (because it is possible, of course, with a multilayer network). Instead, they have shown that some functions can not be computed with ANN unless at least one unit in the intermediary layer connects to all inputs. Another problem was that the learning algorithm would not work for multilayer networks.

OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

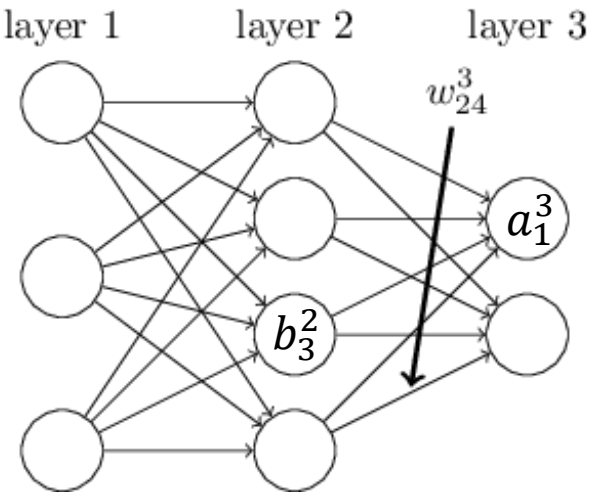
MODERN NEURAL NETS AND BACKPROPAGATION

- While other ANN variants and backpropagation in some form were apparently known to some researchers, the lack of interest in ANN delayed the broad introduction of backpropagation until the 1980ies
- It *is* possible to propagate the misclassification error back and update *all* weights (including those in the hidden layer(s)); differentiable transfer function helps too.
  - Utility of backpropagation was demonstrated quite dramatically in 1989 by Yann LeCun by building an ANN classifier for handwritten digits (the very dataset we were playing with last couple of weeks).
- Transfer function:
  - Logistic: output $O = \frac{\exp(\boldsymbol{w}\cdot\boldsymbol{x}+w_0)}{1+\exp(\boldsymbol{w}\cdot\boldsymbol{x}+w_0)}$: what do you think a single neuron unit becomes then?
  - Hinge
  - Tanh
- The discussion of backpropagation thereafter follows this online book: http://neuralnetworksanddeeplearning.com/chap2.html

BACKPROPAGATION ALGORITHM

- Consider (1) a matrix w_{jk}^l of the weights between k-th unit in layer $l-1$ and j-th unit in layer l, (2) a vector b_j^l of biases (i.e. w_0 values) in j-th unit in layer l, and (3) the vector a_j^l of "activations" (or outputs) of j-th unit in layer l:
- With these notations the activation of neuron j in layer l is:

$$a_j^l = f\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$



- Or, in matrix notation:

$$\boldsymbol{a}^l = f\left(w^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l\right)$$

where $\boldsymbol{a}^l$ and $\boldsymbol{b}^l$ are the column vector of all outputs and biases in layer l, and w^l is the matrix of weights: the j-th *row* of w^l contains the weights used for summing up inputs (the outputs from the previous layer!) into the j-th neuron in layer l. The transfer function f is now understood as acting on each element of its column vector argument.

BACKPROPAGATION, CONTINUED

- Let us further define:
  - The vector z^l of weighted inputs into units in level l, or for the j-th element (j-th unit):

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l = (w^l \cdot \boldsymbol{a}^{l-1} + \boldsymbol{b}^l)_j$$

  - The cost function *C* we want to minimize ("fitting" a model always means improving some score, right?). For simplicity let us consider quadratic cost for a single observation (**x**,y) (regression problem!):

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2 ,$$

where L is the number of layers (i.e. we take the outputs from the last layer as predicted values – remember that we can have multiple output neurons j=1…m !)

  - The "error" of neuron j in layer l:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

(strictly speaking it's not the "error", but the *effect* of the neuron on the output: by how much a small change in that particular neuron's weighted input is going to affect the cost function). Note that if we know δ_j^l, then we also know how we should wiggle the neuron's input z_j^l in order to decrease the cost

ERROR BACKPROPAGATION, OUTPUT LAYER

- As the name of the method amply suggests, we start from the last layer. Remember that the output of that layer is the predicted outcome $a^L = f(z^L)$, and the cost *C* is in fact expressed in terms of a^L. Thus, using the chain rule, the errors in the output layer can be written as:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} f'(z_j^L)$$

Note that this is easily computable: we know the transfer function *f*, thus we also know what its derivative *f'* is, so we only need to substitute z_j^L (which we are going to compute anyway in the process of calculating the predictions of the current version of the network). Cost function is also known, and so is its derivative. In our example of the quadratic cost function, $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$.

- We can rewrite all the equations for j=1…m in a vector form if we introduce the vector $\nabla_a C$ (j-th component of this vector is simply a derivative of C wrt the output a_j^L), called a *gradient*:

$$\delta^L = \nabla_a C \odot f'(z^L)$$

where we use the "Hadamard product" – it's simply an elementwise product of two vectors (this is exactly how R would multiply c(1,2,3)*c(4,5,6) !

ERROR BACKPROPAGATION, INTERMEDIATE LAYERS

- We can continue applying the chain rule and obtain the expressions for the errors of neurons in intermediate layers. Indeed, for a neuron j in layer l we have (note that $a_j^l = f(z_j^l)$, the very same way we just had it for the output layer of course):

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} f'(z_j^l)$$

In addition, let us express the derivative wrt a_j^l via the derivative(s) wrt z_j^{l+1}, ie. weighted inputs in the *next* layer. Since *each* of those weighted inputs includes the output from our neuron j in layer l , we now have to write:

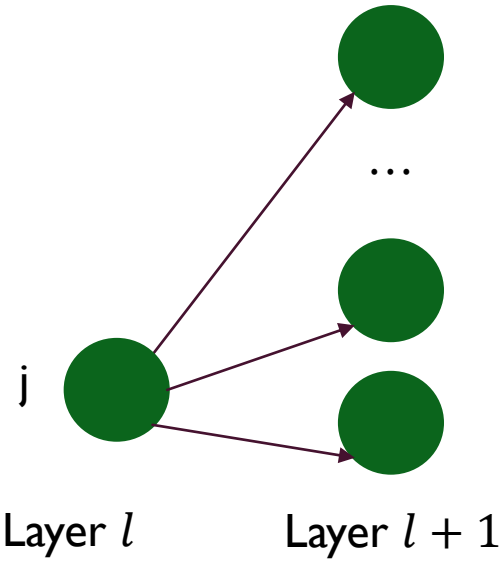
$$\frac{\partial C}{\partial a_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1}$$

(remember the definitions of the error and of the weighted input z!)

Combining the results and writing them in vector form we obtain:

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1} \right) \odot f'(z^l)$$

This is again easily computable. Note how the error propagates *backwards* from $l+1$ to l !!



BACKPROPAGATION, THE RESULT

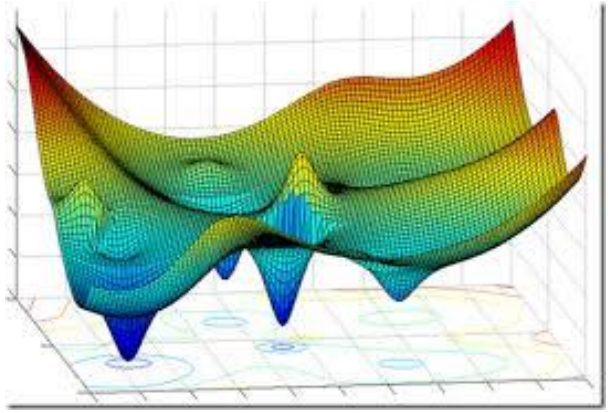
- But why did we have to compute the errors δ? Don't we want to update parameters w, b of our model?
- It turns out, the derivatives wrt the parameters can be easily expressed in terms of δ. The full backpropagation algorithm boils down to just 4 equations (first two are the error backpropagation equations, just reproduced here for completeness):

$$\delta^L = \nabla_a C \odot f'(z^L)$$

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1} \right) \odot f'(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}\,\delta_j^l$$



Now, for each training example (x,y), we can compute the prediction of our ANN on x, calculate the cost function, backpropagate the error (first two equations) and compute the derivatives of the cost with respect to each model parameter (last two equations). With all the derivatives in hand, we can update all the parameters in the direction, in which C decreases (gradient descent)!

UNIVERSALITY THEOREM

- Neural networks are extremely powerful (in theory): the universality theorem states that for *any* function f(x) (where x can be a vector variable, and where the function f can have vector values itself), no matter how complex and "wiggly", a neural network can be built that would provide arbitrarily good approximation to f(x)
  - Of course, the better approximation we want, the more complex network we might need, so this statement is not necessarily practical, NNs are computationally heavy!
  - You can read more on universality here: http://neuralnetworksanddeeplearning.com/chap4.html
- In practice, relatively (!) simple NNs perform surprisingly well. It is not completely clear why we can build modern powerful deep learning networks that demonstrate tremendous performance on very difficult tasks. Yes, they are complex… Bur not *that* complex.
  - Some interesting arguments can be found here (warning: not a tutorial, a full-fledged research paper is discussed and cited in this editorial, and you can try reading the original paper too): https://www.technologyreview.com/s/602344/the-extraordinary-link-between-deep-neural-networks-and-the-nature-of-the-universe/
  - Aside from philosophy, which is more questionable of course, the paper contains nice demonstration of ANN that computes a product (and by extension any Taylor expansion!)

OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

EXAMPLE: CARDIAC ARRHYTHMIA DATABASE

- Another dataset from UCI: multilevel classification
  - Large number of parameters (279) including patient's data (height, weight, age *etc*) and a number of ECG parameters; outcome: one of 16 diagnostic classes (1=normal, 2-16 are different classes of arrhythmia)

```
ad=read.table("http://archive.ics.uci.edu/ml/machine-learning-
databases/arrhythmia/arrhythmia.data",header=F,sep=",",as.is=T,na="?")
ad=subset(ad,V280!=7 & V280!=8 & V280!=9 & V280 != 14 & V280 != 15) # too few observations in these classes
table(ad$V280)
  1   2   3   4   5   6  10  16
245  44  15  15  13  25  50  22
t.ad=table(ad$V280)
for( cl in 1:length(t.ad) ) {
   ad[[paste("class",names(t.ad)[cl],sep=".")]] = ifelse(ad$V280==as.numeric(names(t.ad)[cl]),1,0)
}
apply(ad,2,FUN=function(x) sum(is.na(x))) # will show that V14 is mostly NA (> 300 cases)
ad=ad[,-14]  # remove useless column V14
fm=as.formula(paste("class.1+class.2+class.3+class.4+class.5+class.6+class.10+class.16~",
    paste("V",c(1:13,15:279),sep="",collapse="+"),sep="") ) # create formula; note multi-level outcome!
ad=ad[!apply(ad,1,FUN=function(x) any(is.na(x))),] # remove remaining observations with NAs
dim(ad)
[1] 403 287
```

TRAINING NEURAL NETWORK

- We will use library 'neuralnet' (there are many libraries out there these days!)
  - Network topology: specify the number of units in each hidden layer using parameter 'hidden'. Here we use one hidden layer of 20 nodes. For two hidden layers with 15 and 10 nodes we would ask for `hidden=c(15,10)`

```
library(neuralnet)
Mn=neuralnet(fm,data=ad,hidden=20,linear=FALSE,err.fct="ce")
vars=paste("V",c(1:13,15:279),sep="")
result=apply(compute(Mn,ad[,vars])$net.result,1,which.max) # or, equivalently:
result=apply(predict(Mn,ad[,vars]),1,which.max)
# we chose to use only 8 diagnosis classes out of the whole table; our 7th and 8th class
# variables are diagnosis classes 10 and 16 (see the formula used):
sum(diag(table(ad$V280,ifelse(result==7,10,ifelse(result==8,16,result)))))
[1] 281  # only 70% correct predictions on the training set!

# Compare to:
library(e1071)
fm1=as.formula(paste("as.factor(V280)~",paste("V",c(1:13,15:279),sep="",collapse="+"),sep=""))
Ms=svm(fm1,data=ad,kernel="poly",degree=3,coef0=1,C=0.1,scale=T)
table(ad$V280,predict(Ms))
      1   2   3   4   5   6  10  16
  1 237   0   0   0   0   0   0   0
  2   0  36   0   0   0   0   0   0
  3   0   0  13   0   0   0   0   0
  4   0   0   0  14   0   0   0   0
  5   0   0   0   0  13   0   0   0
  6   0   0   0   0   0  24   0   0
  10  0   0   0   0   0   0  48   0
  16  0   0   0   0   0   0   0  18
```

WHY WAS FIT SO BAD WITH NEURAL NET?

- Consider the number of parameters that need to be optimized:
  - ~280 inputs feeding into each of 20 hidden nodes ~ 5600 parameters
  - 20 hidden units feeding into 8 output units = another 168 (if we also count bias terms as we should)
  - Total of ~ 5800 parameters, with only 400 examples
  - Why were we able to get any fit at all if the problem is so under-defined?
  - Online learning! We can update coefficients on every single example!
- Additional considerations
  - Most frequent (normal) class represents well over 50% of observations
  - Some attributes are constant or nearly constant
  - Probably not all attributes are equally useful
  - Neural net starts with random weights
    - Results and rate of convergence may vary from trial to trial

ARRHYTHMIA DATASET: SUBSET OF IMPORTANT VARIABLES

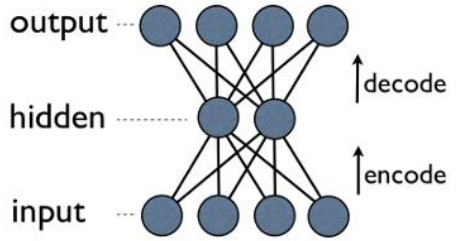
- Suppose we know that some variables are more "important" than others - how can we figure out this, btw?

```
> impVars=c("V15","V224","V93","V91","V228","V267","V277","V257","V197","V177","V113",
+ "V243","V117","V114","V242","V211","V171","V76","V28","V181","V7","V163","V4","V192")
> fmImp=as.formula(paste0(paste0("class.",c(1:6,10,16),collapse="+"),"~",paste0(impVars,collapse="+")))
> set.seed(123);invisible(rnorm(10000))  # fix for reproducibility and warm-up; always check how it varies!
> system.time(MnImp<-neuralnet(fmImp,data=ad,hidden=c(20,10,5),linear=FALSE,err.fct="sse",threshold = 1))
   user  system elapsed
   1.39    0.00    1.44
> resultImp=apply(compute(MnImp,ad[,impVars])$net.result,1,which.max) # or, equivalently:
> resultImp=apply(predict(MnImp,ad[,impVars]),1,which.max)
> table(ad$V280,ifelse(resultImp==7,10,ifelse(resultImp==8,16,resultImp)))
      1   2   3   4   5   6  10  16
  1 234   1   0   1   0   0   1   0
  2  23  10   0   2   0   1   0   0
  3   1   0  12   0   0   0   0   0
  4   1   1   0  12   0   0   0   0
  5   4   0   0   0   9   0   0   0
  6  11   1   0   0   0  11   1   0
  10  7   1   0   0   0   0  39   1
  16  8   0   0   0   0   0   2   8
> sum(diag(table(ad$V280,ifelse(resultImp==7,10,ifelse(resultImp==8,16,resultImp)))))
[1] 335    # still not quite the result we got with SVM but better than the first attempt
```

- Some classes are easier to predict than others

VARIATIONS

- What we considered so far (and what is mostly used) is a simple *feed-forward* network
  - It is possible to have a more complex networks with connections skipping layers and with *feedback loops* (very difficult to reason about such networks, although optimization methods do exist). Feedback provides memory.
- Feature selection.
  - Consider the weights w_{jk}: imagine our familiar digit recognition problem. Each image = vector of 784 pixels (inputs), multiplied by 784 corresponding weights to get the weighted input into j-th hidden unit. Large w_{jk} means the pixel matters; small weight means the pixel is irrelevant as the input to this particular unit. We can *plot* these 784 weights the same way we plot a vector of 784 pixels. The result shows a "mask" that selects features learned by the particular neuron (you can see some interesting patterns!)
- Extending the latter idea: *autoencoders* : unsupervised(!) method
  - Take inputs x, and consider x also as the training output
  - Can we "predict" x from x via the (smaller) hidden layer?
  - Hidden layer learns compressed (and potentially important and meaningful) representation of the data!



- ….And many more (clustering, deep learning, …)

SOME R TRICKERY: EXPANDING FORMULAE

- Neuralnet() has a shortcoming: it does not like the ~ . formula. This is very inconvenient if we have many predictors

```
expand.formula = function(f,data=NULL)  {
    # convert formula into a plain string, f.str becomes something like
    # "Y~X1+X2" or "Y~.":
    f.str = deparse(f)
    # check if we are dealing with a shortcut formula Y~. (we simply
    # check if the last character in the formula string is ".”). If we are,
    # we also verify that the dataframe to take variables from is actually
    # provided; if the formula is not a shortcut, then this function has nothing
    # left to do and returns the original formula immediately:
    if ( grepl("\\.$",f.str)[1] ) {
     if ( is.null(data) ) { stop("Shortcut formula ~. requires a dataframe") }
    } else {
       return(f)
    }
    # if we got to this point (i.e. we did not execute the return statement
    # above), we are dealing with the shortcut formula ~. and need to expand:
    # replace everything starting with any spaces followed by '~' through
    # the end with an empty string, this leaves us with the dependent
    # variable name; e.g. if the formula string was "Y~X",
    # dependent.name will be now set to "Y" (you can further modify this code to
    # deal with multi-class formulas such as "Y1+Y2+Y3~."):
```

EXPANDING FORMULAE: CONTINUED

```
    dependent.name = sub("\\s*~.*","",f.str) # substitute using regular expression
    n = colnames(data)   # get the names of all variables (columns) in the 'data'
    # remove the dependent variable name from the list of variables provided
    # with the data object (if it is not there, the following code has no effect):
    n = n[ n != dependent.name ]
    # we want to use ALL the remaining variables in the formula,
    # this is what Y~. means! Make a string of "name1 + name2 + …" :
    rhs = paste(n,collapse=" + ")
    # now substitute "." In "Y~." with the rhs name1+name2+…
    f.str = sub("\\.$",rhs,f.str)
    # convert modified string back to formula object, make sure we bind
    # the new formula to the right environment:
    f = as.formula(f.str,env=environment(f))
    return(f)
}
```

NEW FUNCTION INTERFACE

- Now we could do something like neuralnet(expand.formula("Y~.",data=d), data=d, …) – still a little cumbersome
- Let's also fix the neuralnet itself (write a wrapper that would take care of the formula):
  - Note that we are presenting simplified code here that can deal only with a binary class output Y~., you can parse LHS (split on '+') if you want to generalize this code to deal with multi-class formulas such as Y1+Y2+Y3 ~ .

```
neuralnet.fx = function(f,data,...)  {
    # in the function definition line we ask for the two first arguments
    # to be assigned to the formal variables 'f' and 'data', while '...'
    # means "any number of additional arguments is allowed"; when our
    # function is actually called with extra arguments,
    # R will store them for us so that we still can access and parse them
    # if we need to, but they will not be automatically assigned to
    # any named formal variables in the function

    # if we called our wrapper with ~. in the
    # formula, at this point the dot shortcut will be expanded into the
    # names of the variables provided in 'data'; if we did not use the dot,
    # the formula will stay unchanged, that's why we needed expand.formula()
    # to be "smart" about it:
    f = expand.formula(f,data)
```

NEW FUNCTION INTERFACE

```
# if observations have NA in any measurement (variable), we will need
# to drop that measurement as a whole - meaning from the dataframe 'data' *AND* from
# the dependent variable in order to keep all the data in sync.
# Let us see where the outcome data are hiding (it might be a separate external
# variable rather then a column in the provided data frame!); in order to do that, we
# first need to extract the name of outcome variable from the formula;
# using the same approach as the one we used in expand.formula() earlier, we get:
f.str = deparse(f)
# if the formula is too long (which it might become after we explicitly
# inserted the names of all the columns), deparse() may introduce
# "line breaks", i.e. instead of a single string it will return a
# vector of strings; if this happened, here we collapse them back:
f.str = paste(f.str,collapse="")
dependent.name = sub("\\s*~.*","",f.str)
# if the dependent variable is found among the columns of 'data', we will
# process it later. However, if the dependent variable is not in 'data'
# but is supposed to be found in the environment, we got some magic to do:
if ( ! dependent.name %in% names(data) ) {
```

NEW FUNCTION INTERFACE

```
# Imagine that the dependent.name variable
# holds the string value "outcome" (i.e. the formula was "outcome ~ ..."). 
# Thus when this formula is used for fitting, the variable with the name
# 'outcome' will be looked up in the environment the formula points to.
# The get() command takes the name of the variable
# to extract the value for, and the environment to extract it from, so we copy into
# the local variable dependent.data the value of the variable 'outcome'!
# The beauty of it is that the formula can contain any name of the
# dependent variable of course; we extract that *name* at run-time,
# and pull the value of the variable with *that* specific name from the environment!
# Note that this won't work if the lhs of the formula contains some expression
# rather than just the variable name, e.g. sqrt(Y) or as.factor(Y);
# this can be fixed, but we don't need it right now.
# Following line of code extracts the value of the variable with the *name* stored
# in another variable (called dependent.name), from a specified environment:
dependent.data = get(dependent.name,envir=environment(f))
```

NEW FUNCTION INTERFACE

```
    # we now have a copy of dependent variable data vector in our hands,
    # so we can filter and un-factor it as we please; but the formula still
    # refers to the original variable ("outcome" in our example),
    # how can we make the formula refer to *our* local data instead?
    # Overwriting the value of the original outcome variable is possible,
    # but that means corrupting data in the global environment - a very bad
    # idea. Instead, we will bind the outcome data vector to our
    # local copy of the 'data' (remember, here we are dealing with the case
    # when the outcome variable is NOT in the data frame, so it's ok).
   # The model-fitting function is going to look in the dataframe first, before looking
    # in the enclosing environment, so it will work just like we want it:
    data=cbind(dependent.data,data)
    # set the column name to the name of outcome used in the formula:
    names(data)[1] = dependent.name # now the model will find dependent data in the dataframe!
}
# now the dataframe 'data' definitely keeps the outcome variable too -
# either because it did so from the start, or because we found those
# outcome data and manually added them to the dataframe. All that is left
# is to remove rows with NA from 'data' and un-factor the outcome if needed (neuralnet does
# not like factors either):
```

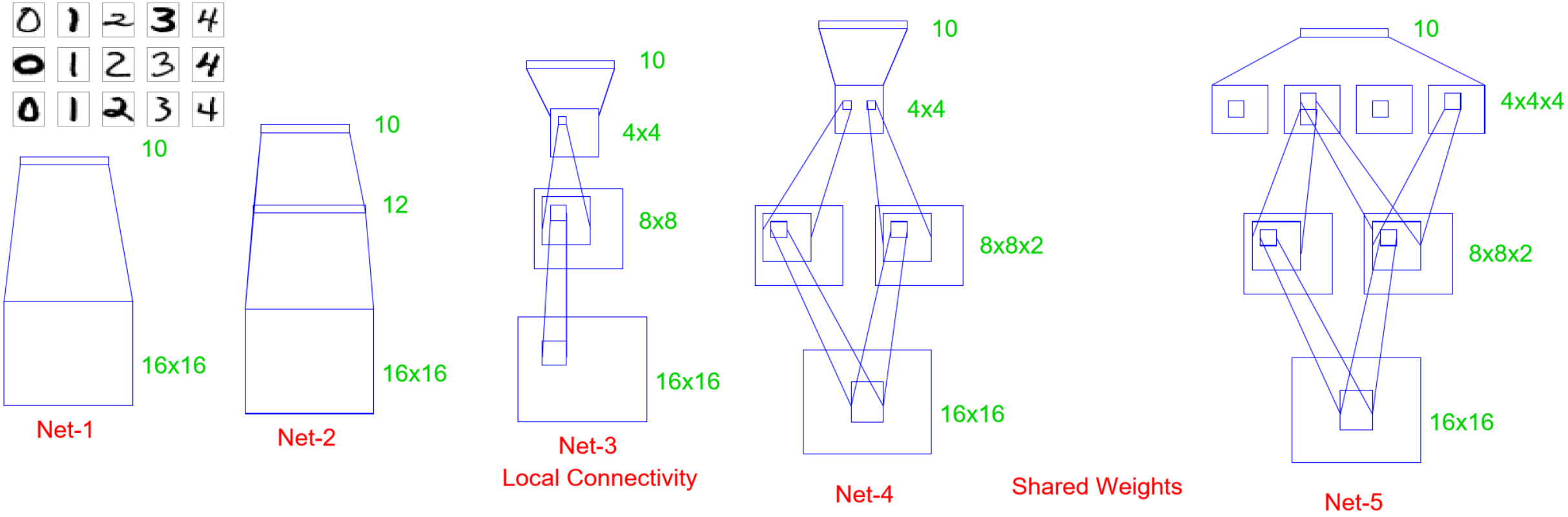
NEW FUNCTION INTERFACE

```
    has.na = apply(data,1,function(x) { any(is.na(x)) } )
    # use drop=F to ensure that even if the data are actually a matrix and the result of the
    # following operation is just one row, that row is still kept as (one-row) matrix and
    # not auto-converted to a vector (that's what R would normally do by default):
    data = data[! has.na,,drop=F]
    if ( is.factor( data[,dependent.name] ) ) {
         data[,dependent.name] = as.numeric(as.vector(data[,dependent.name]))
    }
    # now call the original neuralnet function with the fixed formula and data.
    # The "..." in the function call means: take all
    # the extra parameters we grabbed with '...' in the function definition above,
    # and stick them all here, i.e. we are just passing them all through :
    neuralnet(f,data,...)
}
```

OUTLINE

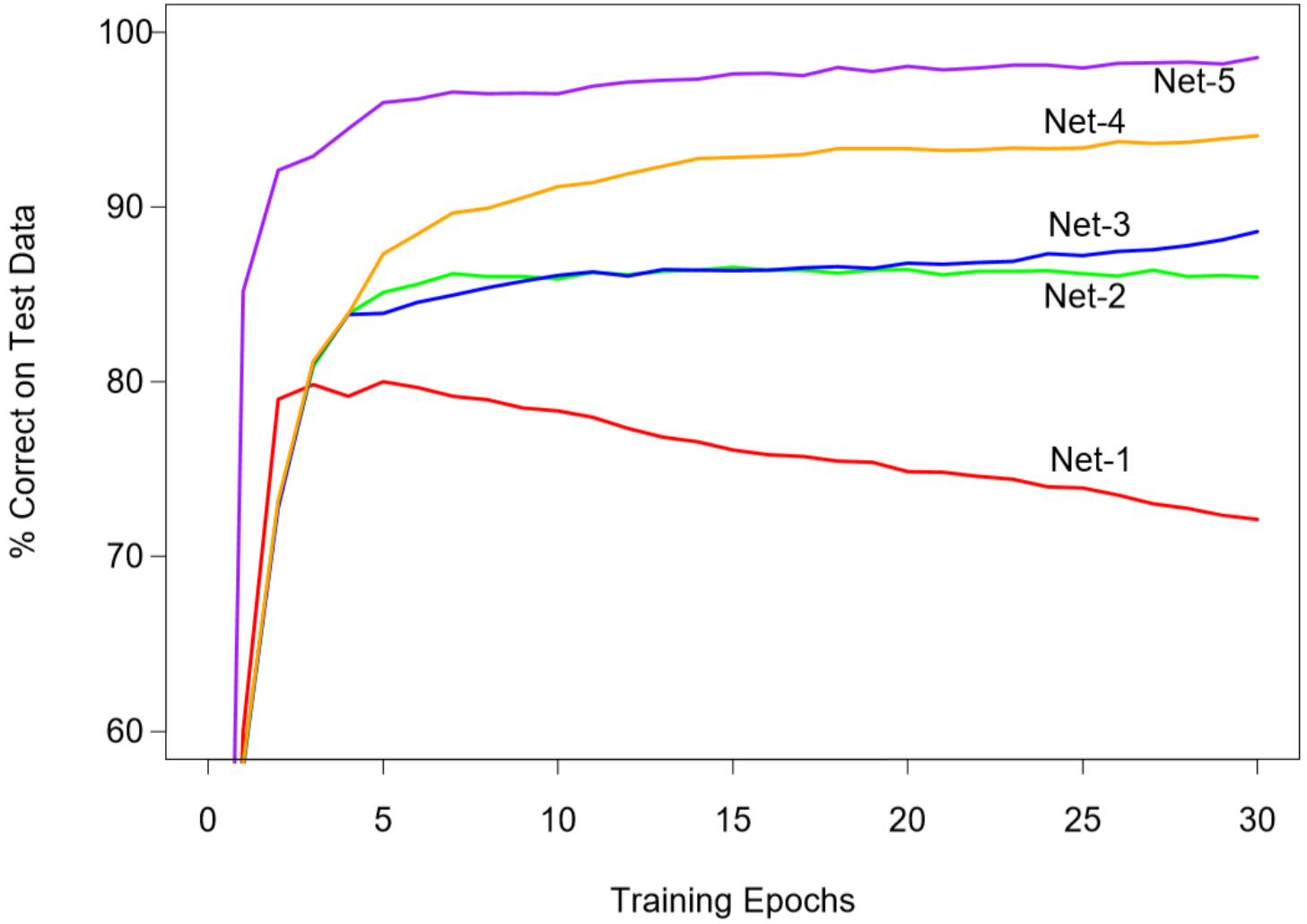
- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

MORE EXAMPLES: ZIP CODE (FROM ESL)



- Examples of handwritten digits represented as 16×16 8-bit grayscale images that have to be classified to classes "0" through "9". Five different network architectures were used to classify these observations (N(train) = 320, N(test) = 160)

MORE EXAMPLES: ZIP CODE (FROM ESL)



- Networks 3, 4 and 5 introduce constraints specific to this problem.
- Restrictions on connections and weights are designed to increase importance of local features that is expected to make fit more robust to rotations etc.
- ANN performance can be greatly improved by developing deep insights about the configuration of the underlying problem
- Training error = 0% - N(train)=320
- "The clever design of network Net-5 … was the result of many person years of experimentation" (ESL Ch.11.7)

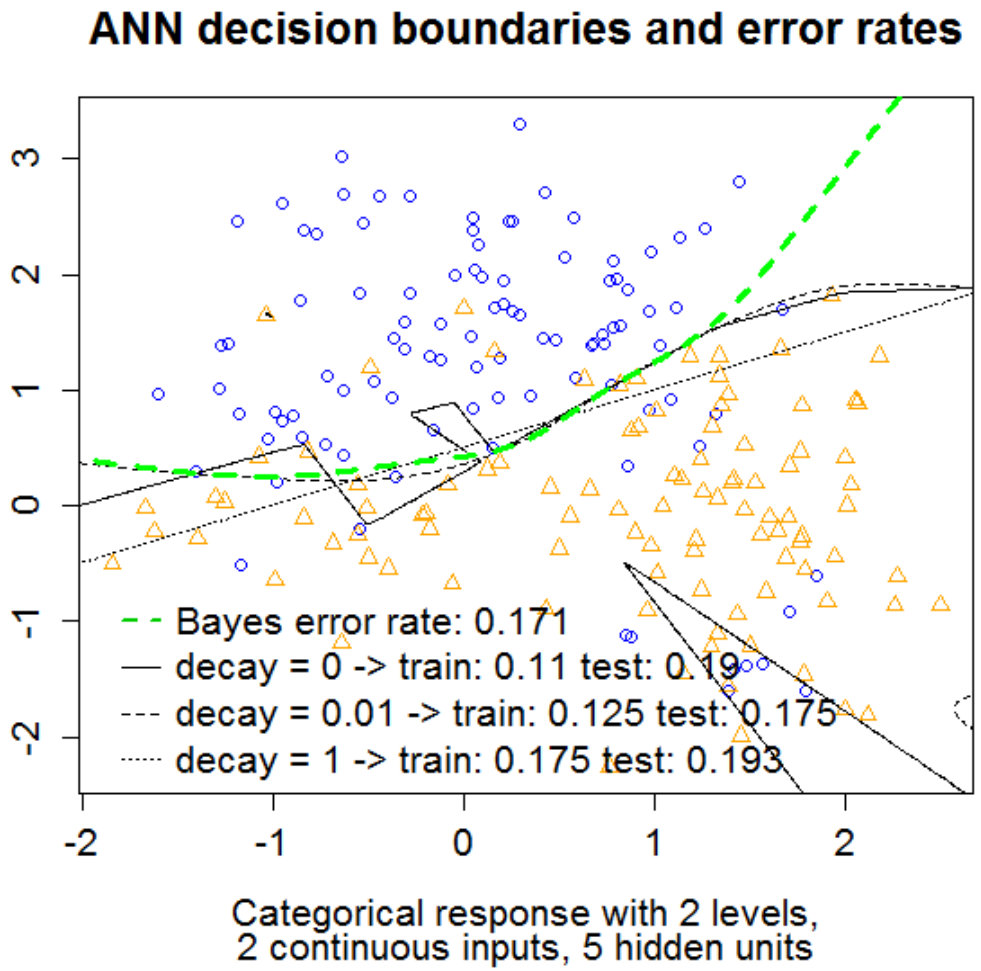
OVERFITTING AND REGULARIZATION

- Depending on number of observations, attributes, hidden layers and hidden nodes number of parameters (weights) in ANN can be large enough to overfit the model
- Similarly to other regularized (e.g. ridge/lasso) approaches, a penalty on weights can be added to the target function for minimization
  - looks like "neuralnet" does not support regularization, while "nnet" does ("decay" parameter controls magnitude of penalty)
- Other forms of regularization include weight elimination and early stopping
- Increase in sample size can attenuate the effect of overfitting and thus the impact of regularization / weight decay
- Let's look at an example of the weight decay impact on "nnet" results

OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

EXAMPLE: WEIGHT DECAY (IN NNET)



- Data for each class generated by sampling from 10 (ten) bivariate normal distributions with their centers sampled in their turn from bivariate normal with 25x higher variance and centered at (0,1) and (1,0) for each class respectively (see code at the next slide)
- Green dashes indicate Bayes classifier boundary
- Black lines/dashes/dots indicate ANN classification boundaries with three levels of weight decay ("nnet")
- Training (n=200) error is the lowest when there is no decay (an overfit model with higher test error)
- Test (n=20K) is the lowest (of the three ANNs) for the model with modest decay – numerically close to that of Bayes classifier
- As in lasso/ridge optimal regularization depends on sample size and signal magnitude as well

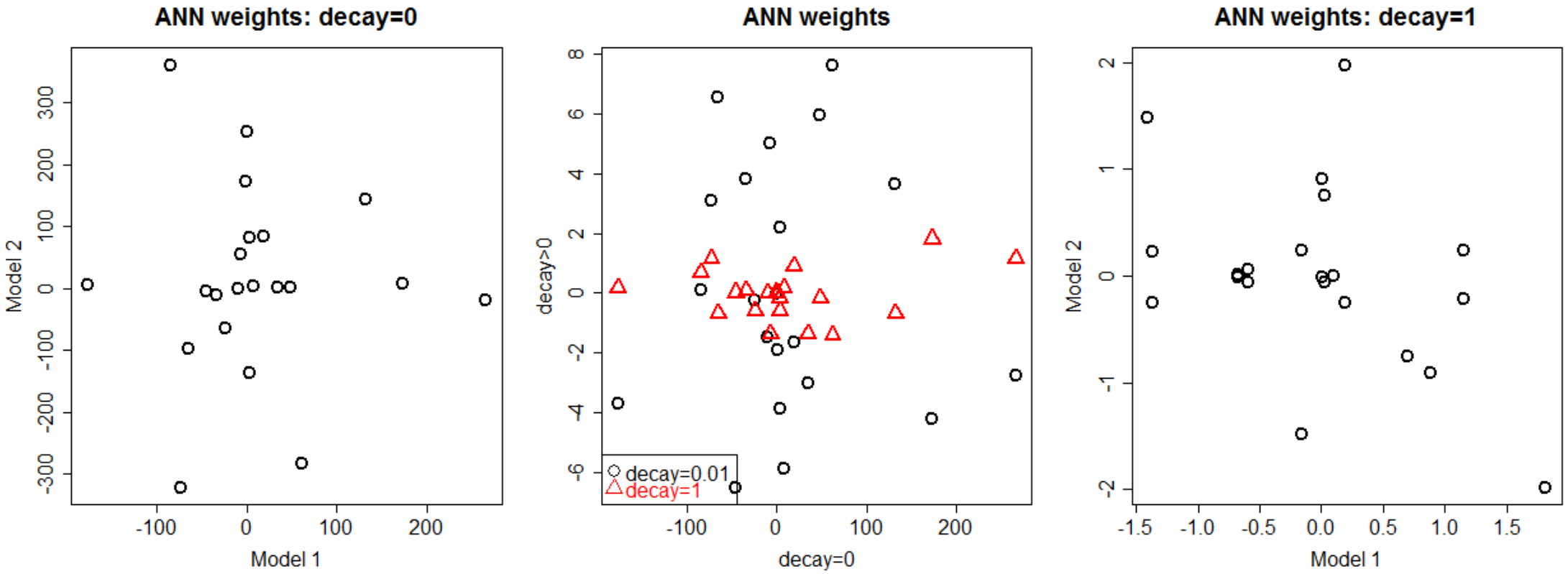
SIMULATION SETUP

```
library(nnet); set.seed(1); n.mu <- 10; sd.sim <- sqrt(1/5); x.grid <- (-400:400)/100
mu.c1.x1 <- rnorm(n.mu); mu.c1.x2 <- rnorm(n.mu, 1); mu.c2.x1 <- rnorm(n.mu, 1); mu.c2.x2 <- rnorm(n.mu)
set2size <- list(train = 100, test = 10000); set2data <- list()
for (ds in names(set2size)) {
  c1.ids <- sample(n.mu, set2size[[ds]], replace = T)
  c2.ids <- sample(n.mu, set2size[[ds]], replace = T)
  c1.train.x1 <- rnorm(set2size[[ds]], mu.c1.x1[c1.ids], sd.sim)
  c1.train.x2 <- rnorm(set2size[[ds]], mu.c1.x2[c1.ids], sd.sim)
  c2.train.x1 <- rnorm(set2size[[ds]], mu.c2.x1[c2.ids], sd.sim)
  c2.train.x2 <- rnorm(set2size[[ds]], mu.c2.x2[c2.ids], sd.sim)
  set2data[[ds]] <- data.frame(x1 = c(c1.train.x1, c2.train.x1),x2 = c(c1.train.x2, c2.train.x2),y = c(rep("c1",
set2size[[ds]]), rep("c2", set2size[[ds]])))
}
class.dens <- function(x1, x2, mu.1, mu.2, sd.1, sd.2 = sd.1) {
  ret.dens <- 0
  for (i in 1:length(mu.1)) {
    ret.dens <- ret.dens + dnorm(x1, mu.1[i], sd.1) * dnorm(x2, mu.2[i], sd.2)
  }
  ret.dens/length(mu.1)
}
d.test <- set2data[["test"]]
c2.test.dens <- class.dens(d.test$x1, d.test$x2, mu.c2.x1, mu.c2.x2, sd.sim)
c1.test.dens <- class.dens(d.test$x1, d.test$x2, mu.c1.x1, mu.c1.x2, sd.sim)
bayes.rate <- sum(as.numeric(c2.test.dens > c1.test.dens) + 1 != as.numeric(d.test$y))/dim(d.test)[1]
c1.grid.dens <- outer(x.grid, x.grid, class.dens, mu.c1.x1, mu.c1.x2, sd.sim)
c2.grid.dens <- outer(x.grid, x.grid, class.dens, mu.c2.x1, mu.c2.x2, sd.sim)
```

CODE FOR PREVIOUS FIGURE

```
cl.clr <- c("blue", "orange")[as.numeric(set2data[["train"]]$y)]
plot(set2data[["train"]]$x1, set2data[["train"]]$x2, col = cl.clr, pch = as.numeric(factor(cl.clr)), xlab =
"Categorical response with 2 levels,", main = "ANN decision boundaries and error rates", sub = "2 continuous
inputs, 5 hidden units", ylab = "")
decays <- c(0, 0.01, 1)
d.train <- set2data[["train"]]
d.test <- set2data[["test"]]
for (i.nnet in 1:length(decays)) {
  x.nnet <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = decays[i.nnet])
contour(x.grid, x.grid, outer(x.grid, x.grid, function(x, y) predict(x.nnet, data.frame(x1 = x, x2 = y),
type = "raw")), levels = 0.5, drawlabels = F, add = T, lty = i.nnet)
  train.err <- sum(predict(x.nnet, type = "class") != d.train$y)/dim(d.train)[1]
  test.err <- sum(predict(x.nnet, d.test, type = "class") != d.test$y)/dim(d.test)[1]
  points(c(-1.8, -1.6), rep(-1.4 - 0.4 * (i.nnet - 1), 2), type = "l", lty = i.nnet)
  text(-1.6, -1.4 - 0.4 * (i.nnet - 1), paste("decay =", decays[i.nnet], "-> train:", signif(train.err, 3),
"test:", signif(test.err, 3)), pos = 4)
}
points(c(-1.8, -1.6),c(-1,-1),type="l",lwd=2,col=3,lty=2)
text(-1.6, -1, paste("Bayes error rate:", signif(bayes.rate, 3)), pos = 4)
contour(x.grid, x.grid, c1.grid.dens - c2.grid.dens, levels = 0, add = T, drawlabels = F, lty = 2, col =
"green", lwd = 3)
```

REGULARIZATION AND ANN WEIGHTS



- Increased decay decreases magnitude of the weights, but still does not ensure their repeatability between two ANNs optimized initiated with different starting values

CODE FOR THE PREVIOUS PLOTS

```
nnetRes0 <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = 0)
nnetRes0a <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = 0)
nnetRes0.01 <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = 0.01)
nnetRes1 <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = 1)
nnetRes1a <- nnet(y ~ x1 + x2, d.train, size = 5, maxit = 10000, decay = 1)

old.par <- par(mfrow=c(1,3),ps=18)
plot(nnetRes0$wts,nnetRes0a$wts,lwd=2,cex=2,main="ANN weights: decay=0",xlab="Model 1",ylab="Model 2")
plot(nnetRes0$wts,nnetRes0.01$wts,lwd=2,cex=2,main="ANN weights",xlab="decay=0",ylab="decay>0")
points(nnetRes0$wts,nnetRes1$wts,col=2,pch=2,lwd=2,cex=2)
legend("bottomleft",c("decay=0.01","decay=1"),col=1:2,pch=1:2,text.col=1:2,pt.cex=2)
plot(nnetRes1$wts,nnetRes1a$wts,lwd=2,cex=2,main="ANN weights: decay=1",xlab="Model 1",ylab="Model 2")
par(old.par)
```

OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

ANOTHER EXAMPLE: DETECTION OF RETINOPATHY

JAMA | Original Investigation | INNOVATIONS IN HEALTH CARE DELIVERY

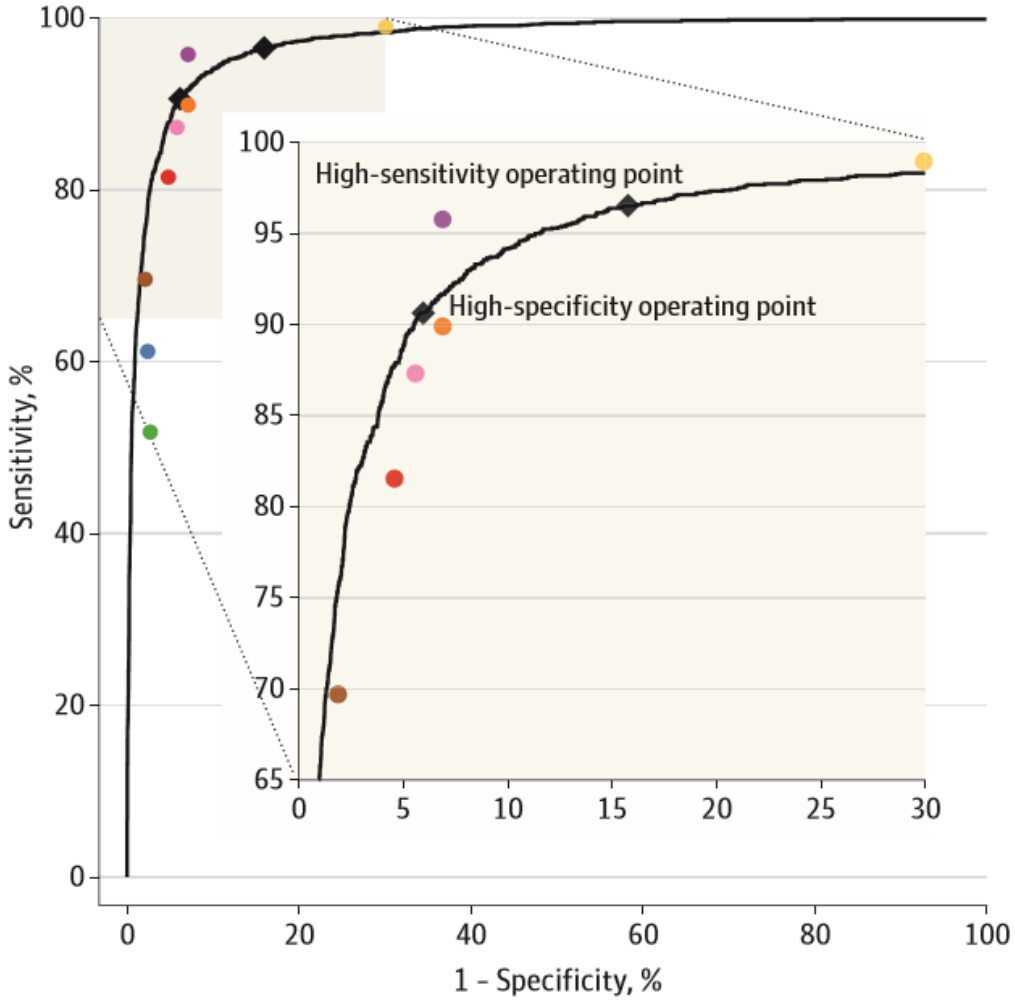
Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs

Varun Gulshan, PhD; Lily Peng, MD, PhD; Marc Coram, PhD; Martin C. Stumpe, PhD; Derek Wu, BS; Arunachalam Narayanaswamy, PhD; Subhashini Venugopalan, MS; Kasumi Widner, MS; Tom Madams, MEng; Jorge Cuadros, OD, PhD; Ramasamy Kim, OD, DNB; Rajiv Raman, MS, DNB; Philip C. Nelson, BS; Jessica L. Mega, MD, MPH; Dale R. Webster, PhD

JAMA. 2016 Dec 13;316(22):2402-2410. doi: 10.1001/jama.2016.17216

DESIGN AND SETTING A specific type of neural network optimized for image classification called a deep convolutional neural network was trained using a retrospective development data set of 128 175 retinal images, which were graded 3 to 7 times for diabetic retinopathy, diabetic macular edema, and image gradability by a panel of 54 US licensed ophthalmologists and ophthalmology senior residents between May and December 2015. The resultant algorithm was validated in January and February 2016 using 2 separate data sets, both graded by at least 7 US board-certified ophthalmologists with high intragrader consistency.

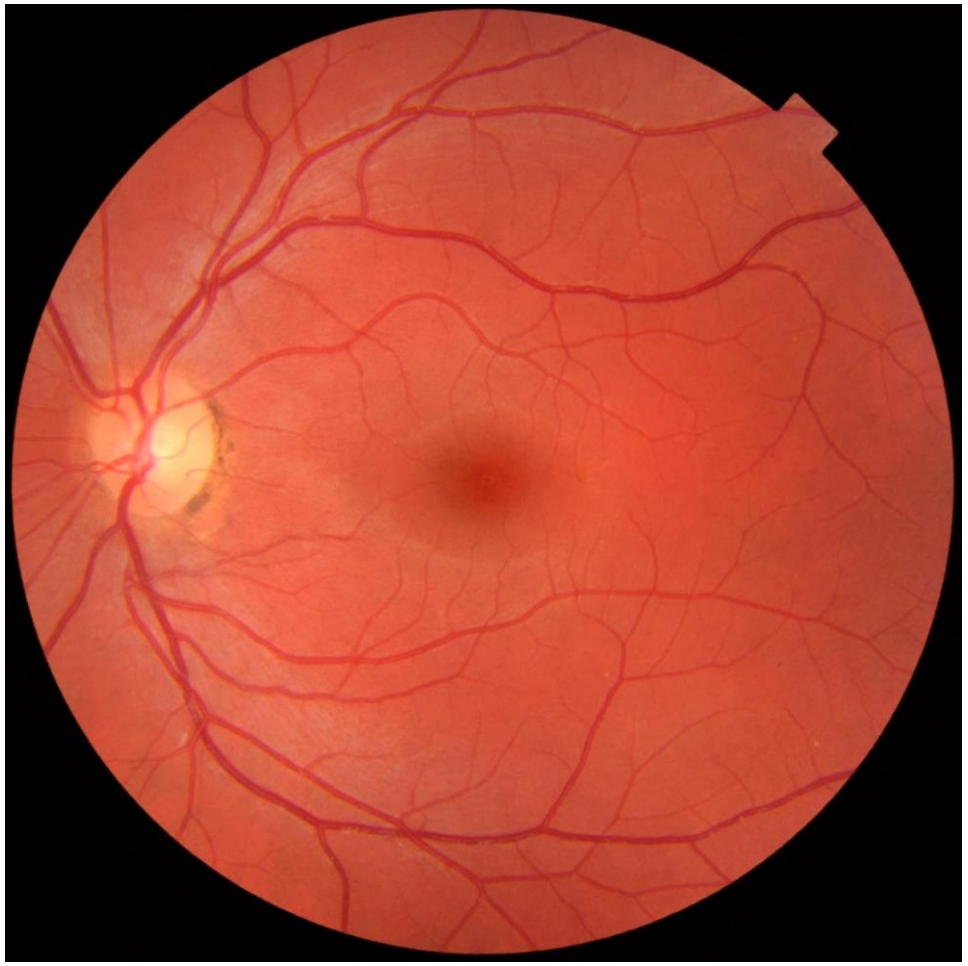
Finding In 2 validation sets of 9963 images and 1748 images, at the operating point selected for high specificity, the algorithm had 90.3% and 87.0% sensitivity and 98.1% and 98.5% specificity



- N(train)=128175, N(test)={9963,1748}, ANN optimized for image classification

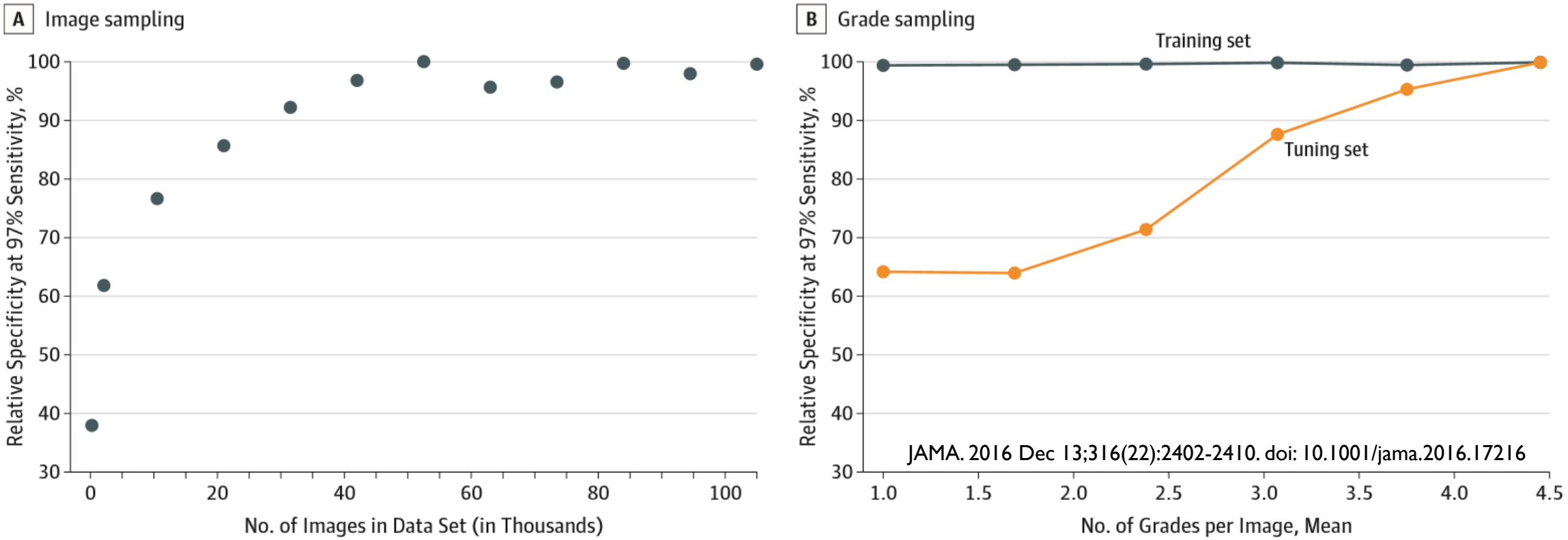
DETECTION OF RETINOPATHY (CNTD.)

- ANN "computes diabetic retinopathy severity from the intensities of the pixels in a fundus image"
- "The network used in this study is a convolutional neural network … that first combines nearby pixels into local features, then aggregates those into global features"
- "The specific neural network used in this work is the Inception-v3 architecture proposed by Szegedy et al."
- "To speed up the training, batch normalization as well as preinitialization using weights from the same network trained to classify objects in the ImageNet dataset were used"
- "Because … [of] a large number of parameters (22 million), an early stopping criteria (that stops training when peak AUC is reached on a separate tuning set) was used to terminate training before convergence." (80%/20% traning/tuning)
- "An ensemble of 10 networks trained on the same data was used"



JAMA. 2016 Dec 13;316(22):2402-2410. doi: 10.1001/jama.2016.17216

DETECTION OF RETINOPATHY (CNTD.)



- A) Increase in the size of training sample increases specificity on tuning data; B) consensus is more predictable (average of multiple grades by several ophthalmologists is better predicted than that by one or two)

OUTLINE

- Neural Networks:
  - History
  - Perceptron
  - Simple networks
  - Logistic transfer function and backpropagation
- Examples:
  - Arrhythmia
  - Zip codes
  - Weights and regularization
  - Detection of retinopathy
- Summary

"THERE IS QUITE AN ART IN TRAINING NEURAL NETWORKS" (ESL CH.11.5)

- Choice of the starting values for the model: for gradient descent to proceed, it has to start somewhere — this starting position can affect the quality of the resulting model
- Overfitting prevention: typical ANN model has fairly large number of parameters increasing possibility of overfit — this can be avoided by constraining minimization procedure
- Predictor values may have to be scaled so that the starting values can be on the same scale and their contributions to regularization are comparable
- Network "architecture": how many hidden units to use? Is more than one hidden layer needed? Local connectivity? Shared weights?
- Because of many local minima the result of minimization is sensitive to the starting point in the space of parameters and this multiplicity of final solutions has to be accounted for

SUMMARY OF ANN PROPERTIES

- A systematic approach to model development using nonlinear transformation of linear combinations of inputs
- Greater model flexibility than that of linear methods, extensively studied and applied to numerous practical problems
- Can be optimized to achieve superior performance on par with the best methods in the field – particularly efficiently when signal is strong as compared to noise and relationship between inputs and outputs is well understood
- Notoriously difficult to interpret due to multiple nonlinear contributions from each input and because weights on similarly performing models can be vastly different from one training run to another
- Most suitable for well-understood problems with strong signal when accurate prediction is a goal in itself (i.e. model interpretation is less important)