Lecture 13

Neural Networks

ESL 11.3 – 11.7

Jilles Vreeken
Aleksandar Bojchevski

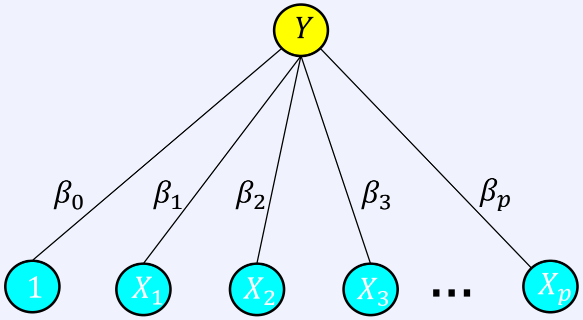
UNIVERSITÄT DES SAARLANDES

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Another look at (Logistic) Regression

The model is $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$

- equivalently $Y = \beta_0 + \beta^T X$ where β and X are now vectors
- equivalently $Y = \beta^T X$ for an X where we added a constant 1 feature



For binary classification we apply the sigmoid function σ

- $Y = \sigma(\beta_0 + \beta^T X)$
- σ squishes the output between 0 and 1
- Y can be interpreted as the probability of class 1

We can only model linear functions!

Introducing Non-linearity

General recipe: transform X into some other space Z and fit a linear model on Z

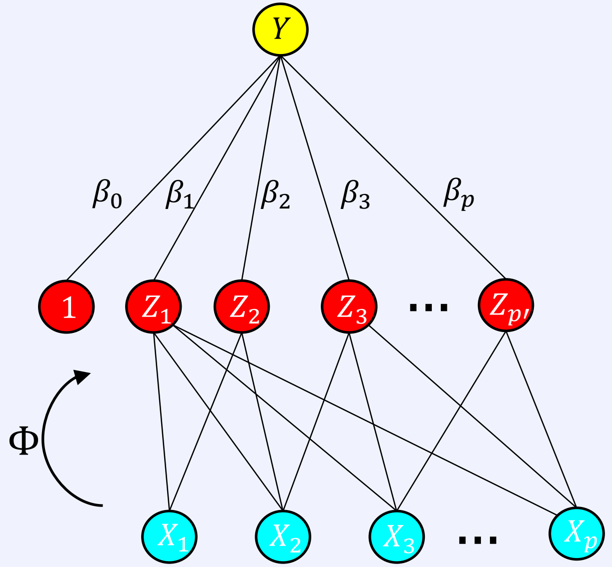
Polynomial regression: add higher-order terms

- $X_1, X_1^2, X_2, X_2^2, \ldots, X_p, X_p^2$
- rename $Z_1 = X_1,\ Z_2 = X_1^2,\ Z_3 = X_2, \ldots, Z_{p'} = X_p^2$
- we can have interaction terms, e.g. $Z_i = X_2 \cdot X_5$

SVM: apply a kernel $K(\cdot,\cdot)$ equivalent to a transformation Φ

- $Z = \Phi(X)$ is now the RKHS (called Ψ before)

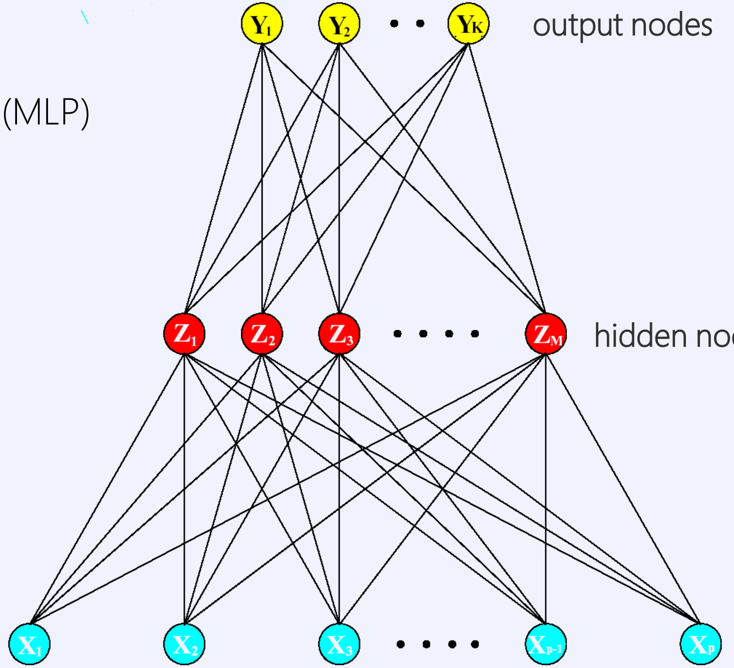
These transformations are fixed. Why don't we learn them?



Neural Networks

Single hidden layer neural network

- aka feed-forward NN, aka multilayer perceptron (MLP)
- represented by a network diagram



Works for regression and classification

- regression: typically $K = 1$ but we can handle multiple quantitative responses
- K-class classification: K output units, where Y_k models the probability of class k

Neural Networks

The derived (hidden) features are defined as

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \dots, M$$

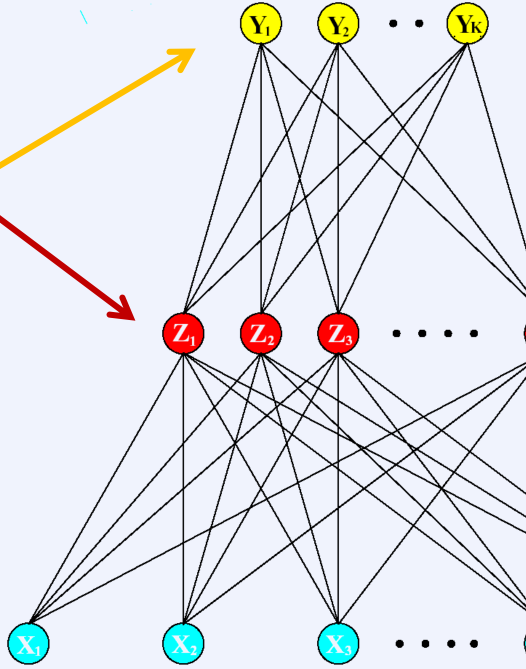
The output features are defined as

$$T_k = \beta_{0k} + \beta_k^T Z, k = 1, \dots, K$$

$$Y_k = f_k(X) = g_k(T), \; k = 1, \dots, K$$

The output transformation $g_k(T)$

- regression $g_k(T) = T_k$, no transformation
- classification $g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$, the softmax ensures Y_k are positive and sum to 1



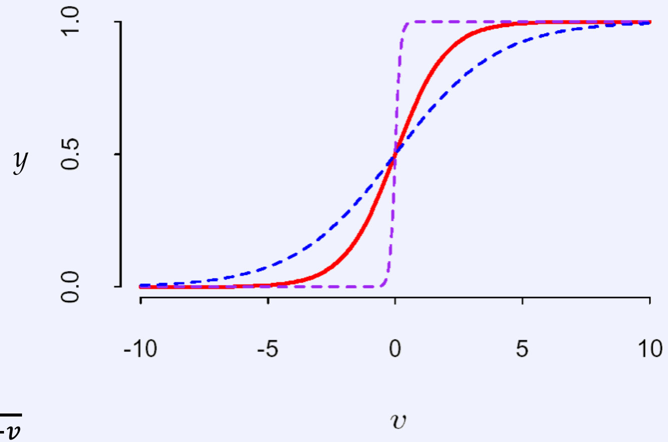
The activation function σ

Why do we need an activation function?

- ignoring the bias we have $Z_m = \sigma(\alpha_m^T X)$ and $T_k = \beta_k^T Z$

If σ is the identity function then

- $T_k = \beta_k^T Z = \beta_k^T(\alpha_m^T X)$
- equivalent to $T_k = \tilde{\beta} Z$ for some $\tilde{\beta}$
- the final model is still linear



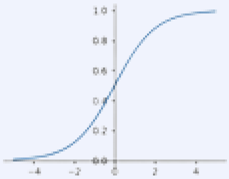
A useful activation function is the sigmoid $\sigma(v) = \frac{1}{1+e^{-v}}$

- each hidden node projects the data along a specific direction α_m and applies a sigmoid along this direction
- $\sigma(s(v - v_0))$ shifts the inflection point from $\mathbf{0}$ to v_0 and scales the function by a factor s

Activation functions

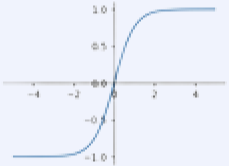
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



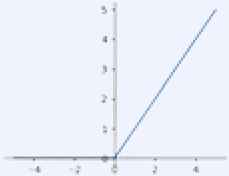
tanh

$$\tanh(x)$$



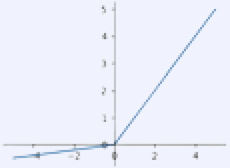
ReLU

$$\max(0, x)$$



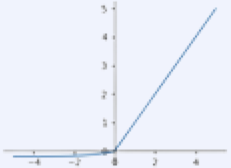
Leaky ReLU

$$\max(0.1x, x)$$



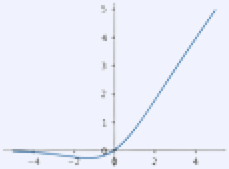
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Swish

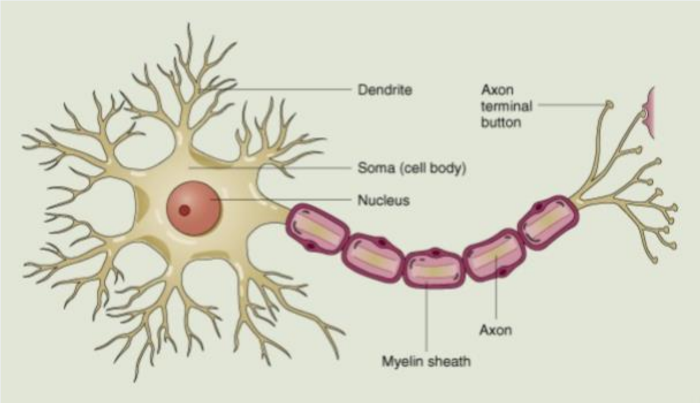
$$x \cdot \sigma(x)$$



Naming of Neural Networks

Naming from (loose) analogy to neurons in the brain

- **dendrites** receive analog input (post-synaptic potentials)
- the **soma** integrates these potentials
- if result exceeds a threshold it fires a sequence of spikes (action potentials) down the **axon**
- firing frequency rises with the total potential
- arrival of a spike at the **axon terminal** causes **neurotransmitter** to be released into the **synaptic cleft**.
- higher frequency of spikes = more transmitter released



Originally the activation function σ was a step function realizing a firing threshold

- was not appropriate for optimization

Fitting a Neural Network

The unknown parameters of a neural net (NN) are the sets of weights θ

- $\{\alpha_{0m}, \alpha_m; m = 1,2, \dots, M\}$ $\quad M(p+1)$
- $\{\beta_{0k}, \beta_k; k = 1,2, \dots, K\}$ $\quad K(M+1)$

Regression: typically use sum of squares as error measure $R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} \left(y_{ik} - f_k(x_i)\right)^2$

Classification: typically use cross entropy $R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$

- with the final classifier as $\arg\max_k f_k(x)$

With softmax and cross entropy a NN equals a linear logistic regression model in its hidden units, but the overall model is non-linear since Z is a non-linear transformations of X

Fitting a Neural Network

All parameters are estimated by maximum likelihood

The global minimizer is likely to overfit the data, thus we need regularization

- through a penalty term or
- by early stopping

Training by gradient descent is now called back propagation

- detailed for square-error loss in the next slide

Recall the chain rule: for $f(x) = d(c(b(a(x))))$ we have:

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

Fitting a Neural Network

The model definition:

- $Z_m = \sigma(\alpha_{0m} + \alpha_m^T X),\ m = 1, .., M$
- $T_k = \beta_{0k} + \beta_k^T Z,$ and $Y_k = f_k(X) = g_k(T), k = 1, \dots, K$

So we have $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i),\ z_i = (z_{1i}, z_{2i}, \dots, z_{Mi}),\ i = 1, \dots, N$

The error is $R(\theta) = \sum_{i=1}^{N} R_i = \sum_{i=1}^{N} \sum_{k=1}^{K} \big(y_i - f_k(x_i)\big)^2$ with the derivative as

$$\frac{\partial R_i}{\partial \beta_{km}} = -2\big(y_{ik} - f_k(x_i)\big) g_k'\big(\beta_k^T z_i\big) z_{mi}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2\big(y_{ik} - f_k(x_i)\big) g_k'\big(\beta_k^T z_i\big) \beta_{km} \sigma'(\alpha_m^T x_i) x_{il}$$

(Rumelhart et al, 1986)

Fitting a Neural Network

- $\frac{\partial R_i}{\partial \beta_{km}} = \underbrace{-2\big(y_{ik} - f_k(x_i)\big)g_k'\big(\beta_k^T z_i\big)}_{\delta_{ki}} z_{mi}$ $\quad (*)$
- $\frac{\partial R_i}{\partial \alpha_{ml}} = \underbrace{-\sum_{k=1}^{K} 2\big(y_{ik} - f_k(x_i)\big)g_k'\big(\beta_k^T z_i\big)\beta_{km}\sigma'(\alpha_m^T x_i)}_{s_{mi}} x_{il}$

We use the gradients in the following gradient descent formula where γ_r is the learning rate

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} + \gamma_r \sum_{i=1}^{N} \partial R_i / \partial \beta_{km}^{(r)}$$

$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} + \gamma_r \sum_{i=1}^{N} \partial R_i / \partial \alpha_{ml}^{(r)}$$

back propagation equations

By $(*)$ the derivatives have the simplified form $\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$ and $\frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}$

where δ_{ki} and s_{mi} are the current error terms and satisfy $s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}$

(Rumelhart et al, 1986)

Fitting a Neural Network

The gradient descent updates are

- $\beta_{km}^{(r+1)} = \beta_{km}^{(r)} + \gamma_r \sum_{i=1}^{N} \partial R_i / \partial \beta_{km}^{(r)}$ and $\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} + \gamma_r \sum_{i=1}^{N} \partial R_i / \partial \alpha_{ml}^{(r)}$ $(**)$
- with derivatives $\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$ and $\frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}$

The updates in $(**)$ can be done by a two-pass algorithm

1. Forward pass
   - fix current weights and compute the predicted output values Y_k
2. Backward pass
   - compute the errors δ_{ki}
   - "back propagate" the errors to obtain the values s_{mi}
   - use these values to compute the gradients and do the update $(**)$

Training with the cross entropy error is analogous

(Rumelhart et al, 1986)

Remarks on Backpropagation

Backpropagation is local in nature

- every weight depends only on the weights of neurons connected to the respective neuron
- the algorithm hence allows for trivial parallelization

We discussed (full) batch learning, where all training data is processed simultaneously

- there is also an online version where training data is fed continually in a repeating cycle
- great for very large training sets
- stochastic gradient descent on mini-batches of data

- larger batch ⇒ less noise in the gradient estimate
- but some noise can be good, e.g. act as a regularizer and help us escape bad local minima

(Robbins & Munro, 1951) (ESL 11.4)

Remarks on Backpropagation

Learning rate γ_r usually a constant

- can be optimized by a line search along the direction of the gradient
- should decrease to 0 in online setting
- variant of stochastic approximation, ensures convergence if $\gamma_r \to 0,\ \sum_r \gamma_r = \infty,\ \sum_r \gamma_r^2 < \infty$
- first-order methods can be very slow, sadly, Newton's method is not appropriate since the second derivative of R can be very large

Backpropagation is not just the chain rule, with automatic differentiation (Pytorch, Tensorflow)

- it is a particularly efficient strategy for computing the chain rule
- evaluating $\nabla f(x)$ provably as fast as evaluating $f(x)$
- code for $\nabla f(x)$ can be automatically derived even if we have control flow structures like loops
- it operates on a more general family of functions: programs which have intermediate variables

(Robbins & Munro, 1951) (ESL 11.4)

Issues with Training

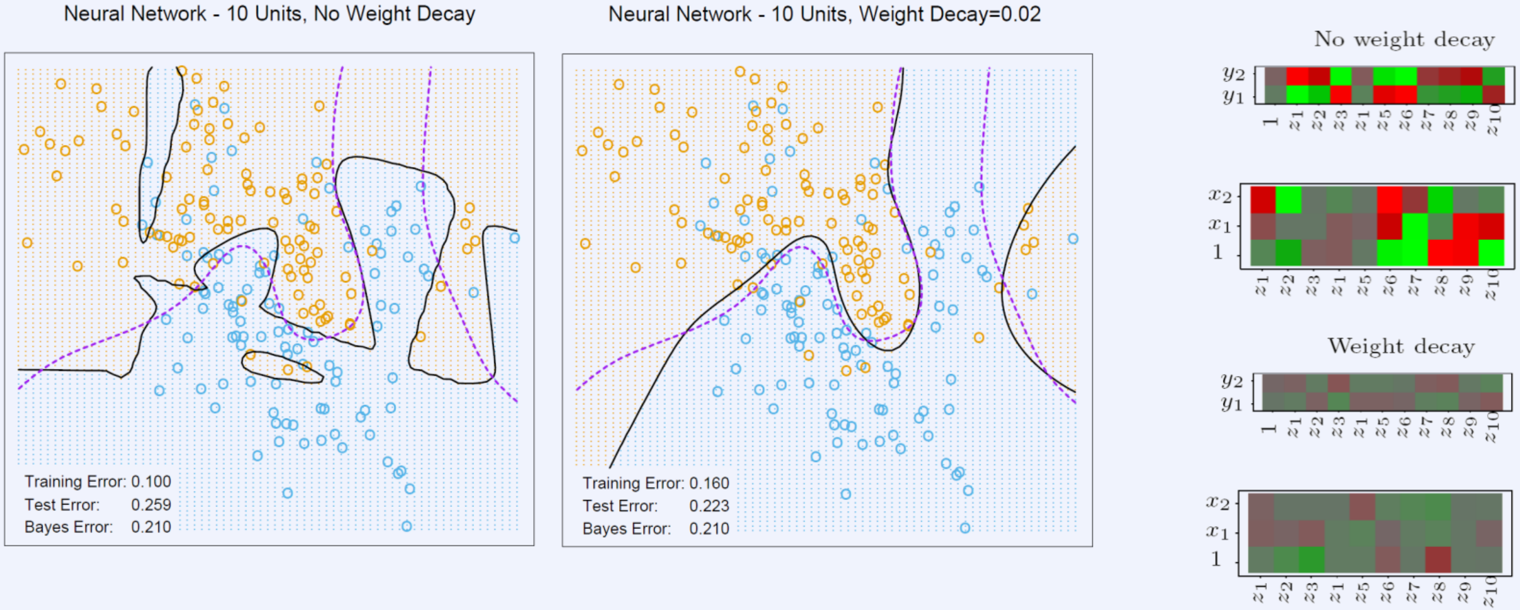
Starting values – initialization of the parameters

- large initial weights usually lead to poor solutions
- starting with 0 for all weights never changes anything
- if weights are near-zero, the operative part of the sigmoid is near-linear and such is the model
- usually chosen randomly near-zero (e.g. Xavier Glorot)
- model hence starts out as linear, and chooses direction where non-linearity is needed

Overfitting

- early stopping amounts to shrinking the model towards a more linear solution
- weight decay is analogue to ridge regression, adds a penalty to the error
  $R(\theta) + \lambda J(\theta)$ *with* $J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2$
- cross validation used to choose λ respectively adds $2\beta_{km}$ and $2\alpha_{ml}$ to the gradient

The Effect of Weight Decay



- alternative **weight elimination** penalty $J(\theta) = \sum_{km} \frac{\beta_{km}^2}{1+\beta_{km}^2} + \sum_{ml} \frac{\alpha_{ml}^2}{1+\alpha_{ml}^2}$ shrinks smaller weights more

Issues with Training

Scaling the inputs

- determines the scaling of the weights in the bottom layer(s)
- can have a large effect on the outcome

Best to normalize the inputs to mean 0 and standard deviation 1

- treats all inputs equally in regularization
- allows meaningful ranges for initial weights, e.g. uniform $[-0.7, +0.7]$

Issues with Training

Number of hidden units

- better too many than too few
- with too few, not enough flexibility for capturing the non-linear effect
- with too many, the superfluous ones can be shrunk during regularization

Typically 50 to 100 hidden units

- increasing with number of inputs and training instances
- need not use CV to find the optimal number if you use CV to tune λ
- choice guided partly by background knowledge

Modern NNs are heavily overparametrized

Issues with Training

Number of hidden layers

- allows hierarchical extraction of features at different levels of resolution
- choice guided partly by background knowledge

$R(\theta)$ potentially possesses very many minima

- thus need to try several starting configurations
- good idea to combine models by averaging their output
- bagging is another possibility (changes training data instead of starting values)

Example Sigmoids and Radials

Generate data from two additive error models

- sum of sigmoids: $Y = \sigma(a_1^T X) + \sigma(a_2^T X) + \epsilon_1$ with $a_1 = (3,3), a_2 = (3,-3)$
- radial: $Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2$ with $\phi(t) = \frac{1}{\sqrt{2\pi}} e^{-t^2/2}$
- $X = (X_1, X_2, \dots, X_p)$ with each X_i being a standard Gaussian variable
- ϵ_i are Gaussian errors, variance chosen such that S/N ratio = 4

Signal to Noise (S/N) ratio $\frac{Var(E[Y|X])}{Var(Y - E[Y|X])} = \frac{Var(f(x))}{Var(\epsilon)}$

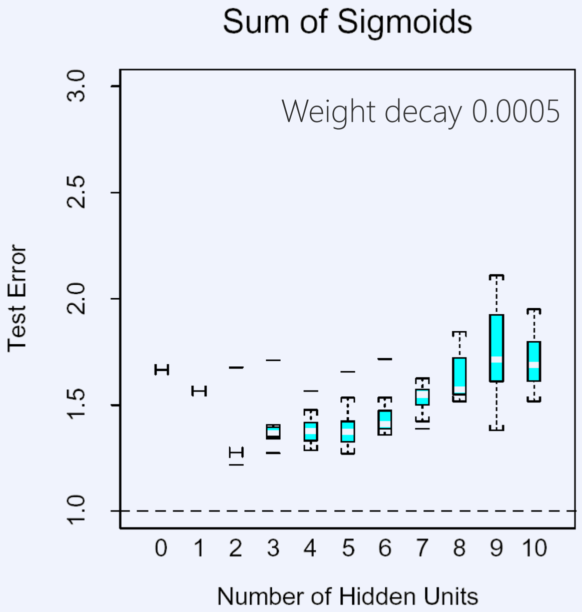
- 100 training samples and 10,000 test samples

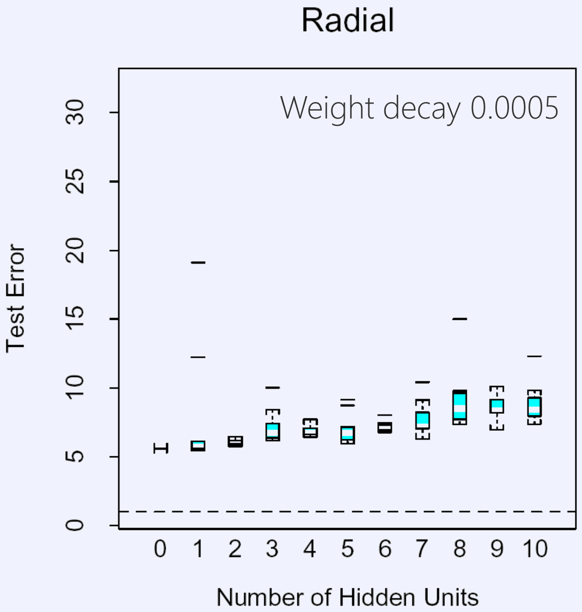
Fit neural network with weight decay $(\lambda = 0.0005)$ and various numbers of hidden units

- record average test error $E_{\text{test}}[Y - \hat{f}(X)]^2$ over 10 random initializations

Example Sigmoids and Radials

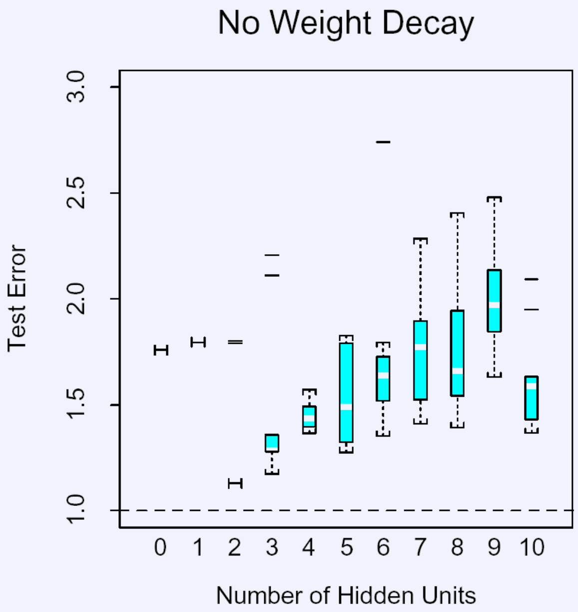
- radial is the “worst” case since no preferred directions (each hidden neuron represents a direction), and worse than the best constant model (which has relative error 5 for a S/N ratio of 4)

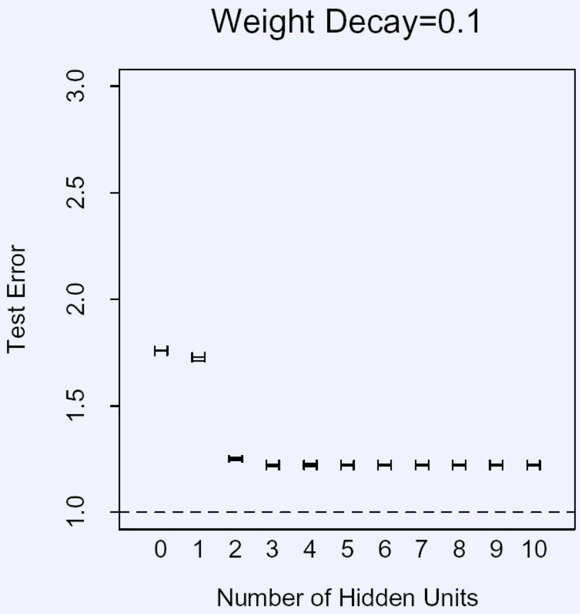




Example Sigmoids – Effect of Weight Decay

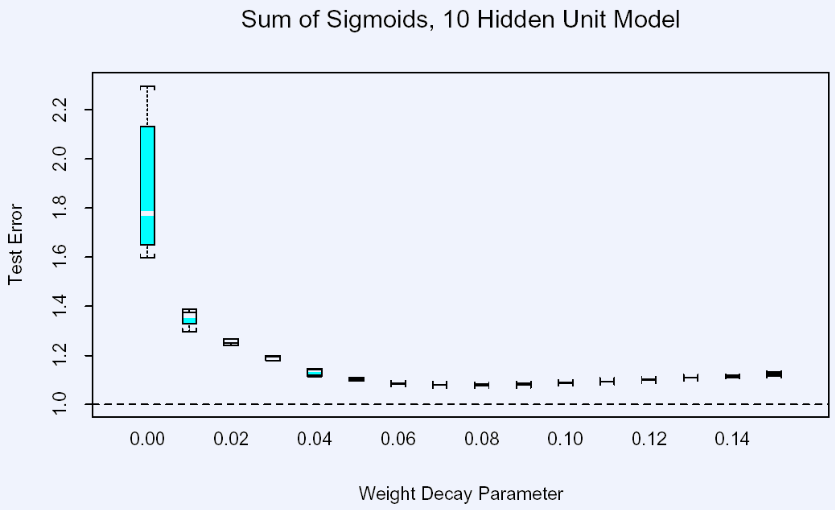
Weight decay helps to reduce the test error





Example Sigmoids – Deterimining λ

$\lambda = 0.1$ is about optimal



Example Sigmoids and Radials

We need to select

- the number of hidden units M
- the weight decay parameter λ

One possibility

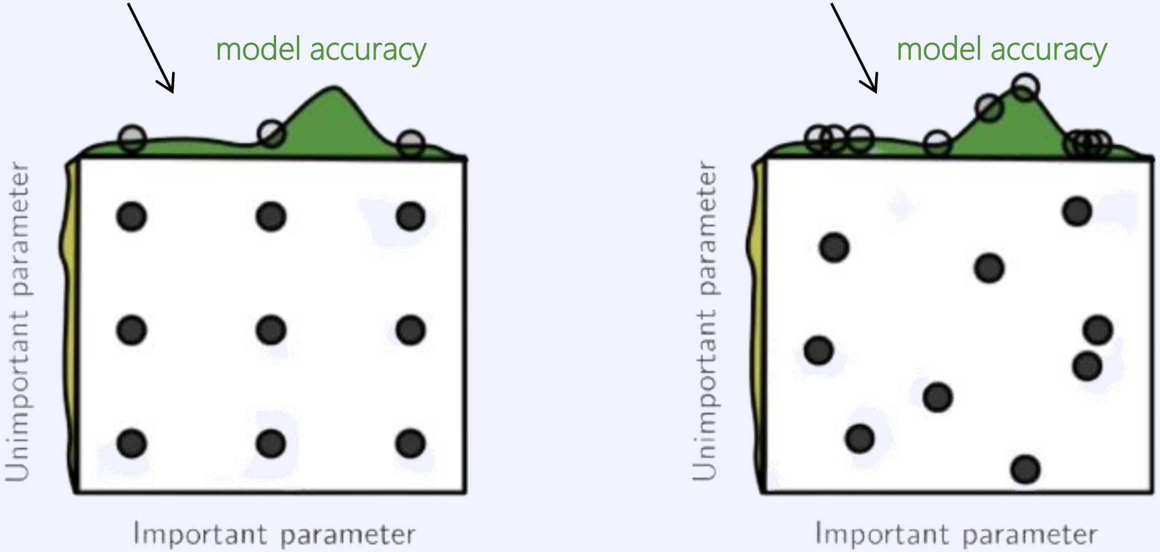
- fix either parameter at the point of the least constrained model (to allow flexibility)
- choose the other parameter with CV
- here least constrained is $\lambda = 0$, $M = 10$
- for our example optimizing λ was more effective than optimizing M

Another possibility: try different configurations of (λ, M)

Note on Hyperparameter Tuning

Another possibility: try different combinations of (λ, M)

Careful: **grid** search is usually much worse than **random** search



(Bergstra and Bengio, "Random Search for Hyper-Parameter Optimization")

Example Zip Code Data

Data is 16x16 8-bit grayscale images

- 256 inputs to the neural net, one per pixel
- 320 digits in training set, 160 digits in test set

A black-box neural net is not appropriate

- pixel representation lacks invariances, e.g. to small rotations
- early-day neural nets hence yielded low accuracy (4.5%)
- here we report on breakthrough effort to overcome this

Sigmoidal output units

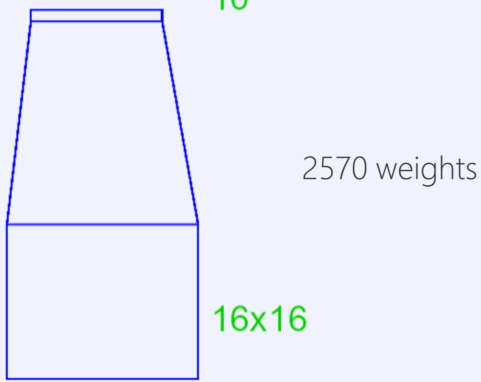
- fit with square error function
- online training
- training error 0% (more parameters than observations)

(ESL 11.7, Le Cun 1989)

Example Zip Code Data

Five classification networks

- Net-1: no hidden layers, equivalent to logistic regression

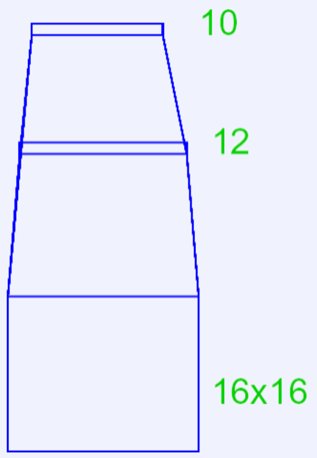


(ESL 11.7, Le Cun 1989)

Example Zip Code Data

Five classification networks

- Net-1: no hidden layers, equivalent to logistic regression
- Net-2: 1 layer, 12 units, fully connected



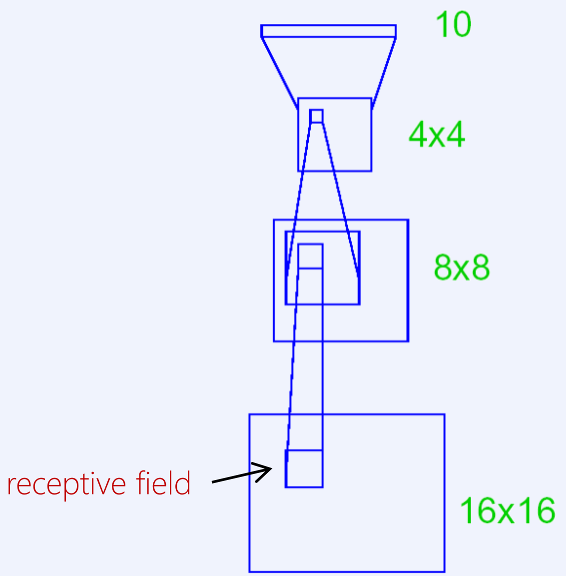
(ESL 11.7, Le Cun 1989)

Example Zip Code Data

Five classification networks

- Net-1: no hidden layers, equivalent to logistic regression
- Net-2: 1 layer, 12 units, fully connected
- Net-3: 2 layers locally connected

- **local connectivity**: receptive field of adjacent units in the first (second) hidden layer are two (one) units apart

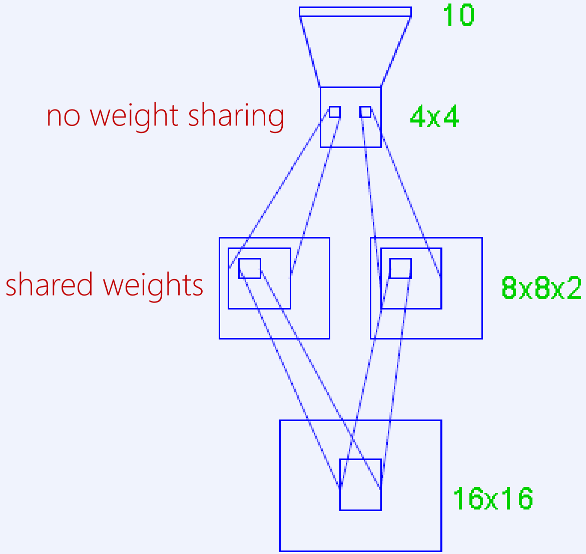


(ESL 11.7, Le Cun 1989)

Example Zip Code Data

Five classification networks

- Net-1: no hidden layers, equivalent to logistic regression
- Net-2: 1 layer, 12 units, fully connected
- Net-3: 2 layers locally connected
- Net-4: like Net-3 with weight sharing



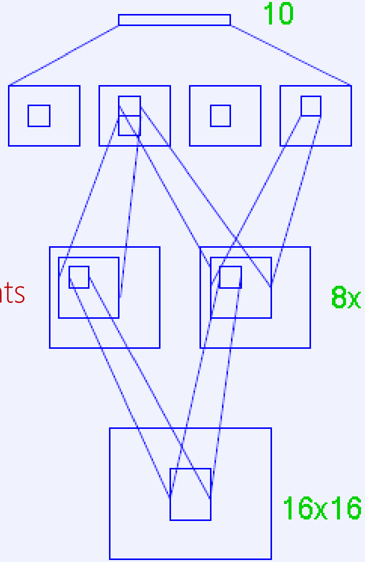
- **local connectivity**: receptive field of adjacent units in the first (second) hidden layer are two (one) units apart
- **shared weights**: same weights among all receptive fields in a feature map, but individual bias

(ESL 11.7, Le Cun 1989)

Example Zip Code Data

Five classification networks

- Net-1: no hidden layers, equivalent to logistic regression
- Net-2: 1 layer, 12 units, fully connected
- Net-3: 2 layers locally connected
- Net-4: like Net-3 with weight sharing
- Net-5: like Net-4 with 2 levels of weight sharing

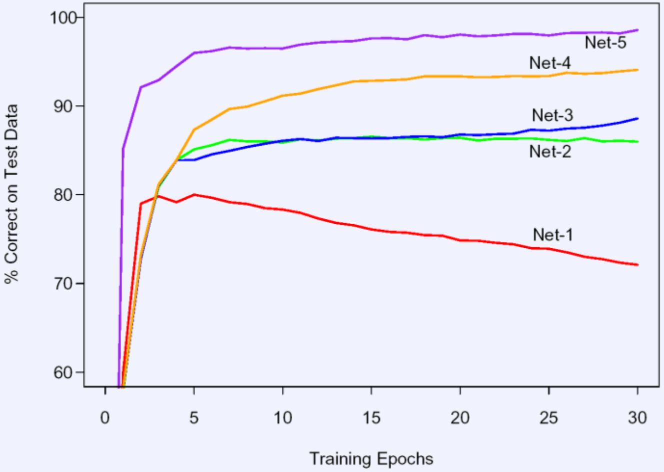


- **local connectivity**: receptive field of adjacent units in the first (second) hidden layer are two (one) units apart

- **shared weights**: same weights among all receptive fields in a feature map, but individual bias

Example Zip Code Data

NNs are especially effective for problems with high signal-to-noise ratio & spatial redundancy
However they lack the ability to interpret the prediction

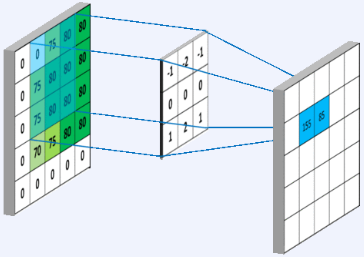


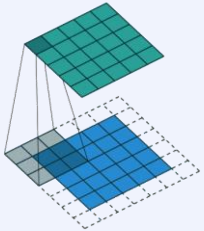
Network	Links	Weights	% Correct
Net-1	2570	2570	80.0
Net-2	3214	3214	87.0
Net-3	1226	1226	88.5
Net-4	2266	1132	94.0
Net-5	5194	1060	98.4

(ESL 11.7, Le Cun 1989)

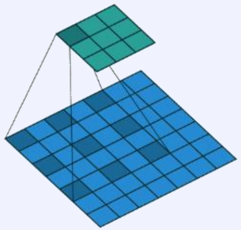
Convolutional Neural Networks

We slide a kernel/filter over the image

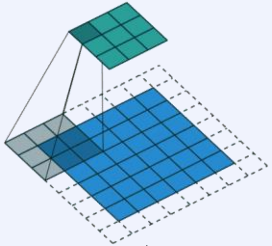




padding

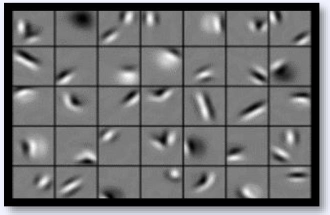


dilation

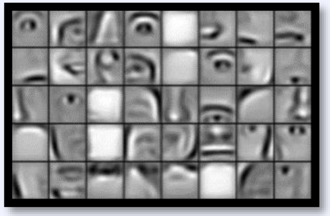


stride

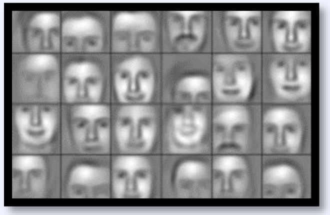
We can visualize the learned filters in each layer



layer 1 filter



layer 2 filter



layer 3 filter

Deep Learning

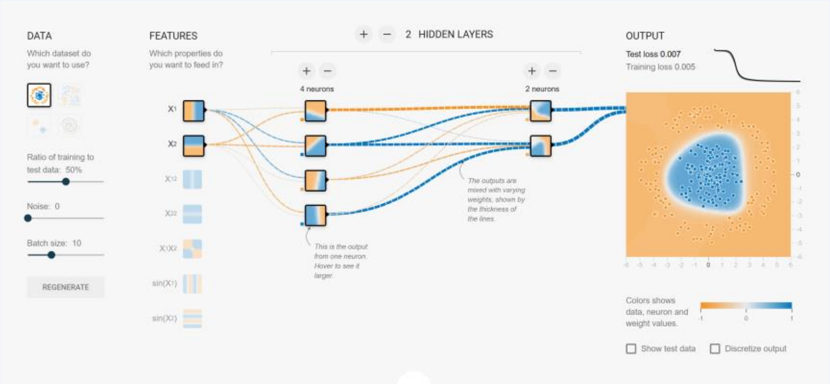
Current hype in neural networks

- build deep (multilayer) and wide networks
- layers learn a hierarchy of problem features, as exemplified by the digit example above
- adjust training schedule (some layers learn faster than others)
- results in high accuracy models
- especially suitable for image analysis and natural language problems

Very well written online textbook: Neural Networks and Deep Learning by Michael Nielsen

A Neural Network Playground

[Tinker with a NN directly in your browser](#) [Tinker with a CNN in your browser](#)



Summary

Neural networks automatically learn the transformation from the data

Feed-forward NN: several layers of linear transformations followed by a nonlinearity

We train NNs with gradient descent (backpropagation via automatic differentiation)

There are many hyperparameters to tune: num. neurons, num. hidden layers, weight decay

Suitable for problems with high SNR and (spatial) redundancy: images, text, audio, graphs, …