Advanced Computational Econometrics: Machine Learning

Chapter 7: Neural Networks and Deep Learning

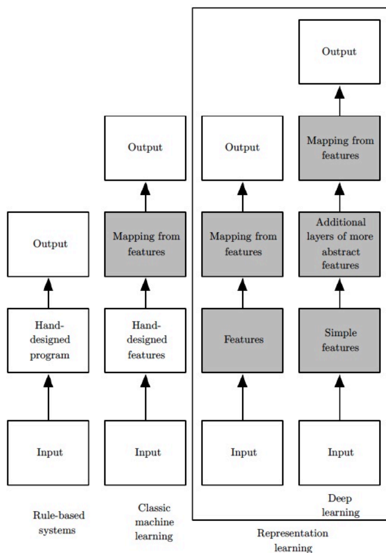
Thierry Denœux

Spring 2022

Neural networks

- A class of learning methods that was developed in AI with inspiration from neuroscience.
- The central idea is to learn simultaneously
  - New predictors (activation of "hidden neurons") and
  - A linear regressor or classifier in the predictor space.
- The result is a powerful learning method, with widespread applications in many fields.
- In recent years, there has been a surge of interest in deep networks/learning, with applications to computer vision and natural language processing.
- There exist many neural network models. In this course we describe the most widely used multilayer feedforward neural networks:
  - Multilayer perceptrons
  - Deep convolutional networks

Neural networks within AI

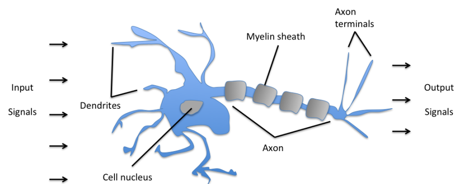


Historical perspective

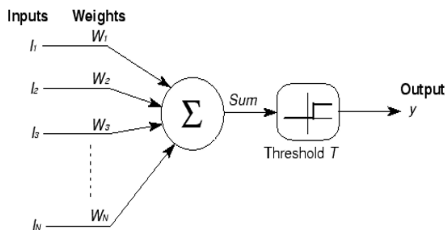
- Three main phases:
  1. Perceptron (1955-1965)
  2. Multi-layer neural networks (1985-1995)
  3. Deep networks (2010-)
- The history can be summarized by a list of 4 influential papers.

McCulloch-Pitts model

- W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115-133, 1943.
- Main idea: biological neurons modeled as simple logic gates with binary outputs.



Schematic of a biological neuron.



McCulloch-Pitts neuron as a logic gate

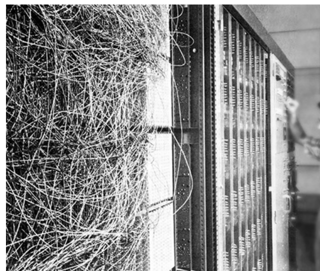
- Assume
  - The inputs I_j take values in $\{0, 1\}$ for $j = 1, \ldots, N$ (1 codes for "TRUE" and 0 for "FALSE")
  - The weights W_j are all equal to 1.
- If the threshold is $T = N$, the output is

$$y = \begin{cases} 1 & \text{if } \sum_{j=1}^{N} I_j - N \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

  It is equal to 1 iff $I_j = 1$ for all j: the neuron computes a logical AND.
- If $T = 1$, $y = 1$ iff $I_j = 1$ for some j: the neuron computes a logical OR.

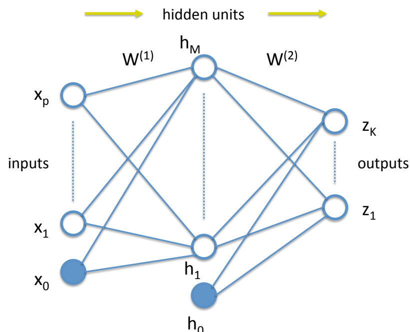
Perceptron

- F. Rosenblatt. *The perceptron, a perceiving and recognizing automaton (Project PARA)*. Cornell Aeronautical Laboratory, 1957.
- Main idea: an algorithm to learn the weights of a McCulloch-Pitts neuron. Implementation in a machine call the "Mark 1 perceptron".
- The algorithm worked only for two linearly separable classes and it was very slow.



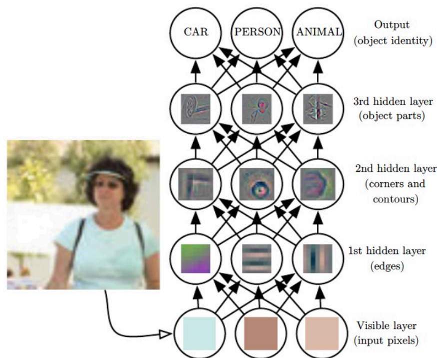
Multilayer neural networks

- D. E. Rumelhart, G. E. Hinton and R. J. Williams (1986). Learning representations by back-propagating errors. *Nature*, 323 (6088):533–536.
- Main ideas: train neural networks with (one or two) hidden layers using an efficient algorithm for computing the gradient of the error (back-propagation algorithm).



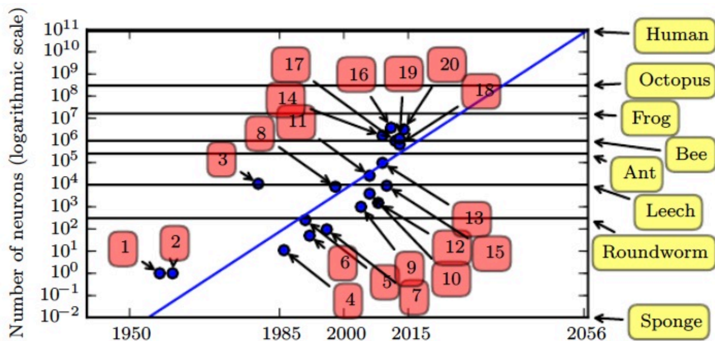
Deep networks

- Y. LeCun, Y. Bengio and G. Hinton (2015). Deep learning. *Nature*, 521:436–444.
- Main idea: train neural networks with many hidden layers that encode more and more abstract features.



Increase of neural network size

Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years.



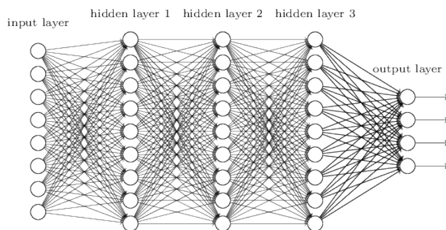
1: Perceptron; 4: Early back-propagation network; 8: LeNet-5 (LeCun et al., 1998b); 20: GoogLeNet

Overview

1. Multilayer feedforward neural networks
   - Hidden units
   - Output units
   - Architecture
2. Learning
3. Complexity control

Definition

- A multilayer feedforward neural network (multilayer perceptron, MLP) is composed of computational units (neurons) arranged in layers: one input layer, one or several hidden layers and one output layer
- Neurons in each layer (expect the input one) are connected to all neurons in the previous layers through weighted connections.
- The information flows from the input layer to the output layer.



Overview

1. Multilayer feedforward neural networks
   - Hidden units
   - Output units
   - Architecture
2. Learning
3. Complexity control

Equation of hidden units

- Each hidden neuron m computes a net input

$$z_m = \sum_{j=1}^{p} w_{mj} x_j + w_{m0} = w_m^T x + w_{m0}$$

  - w_{mj} is the connection weight between input unit j and hidden unit m
  - w_m is the vector of weights of unit m
  - w_{m0} is a bias term (may be seen as the weight of a connection from an input unit with constant input 1).
- The output of unit m is

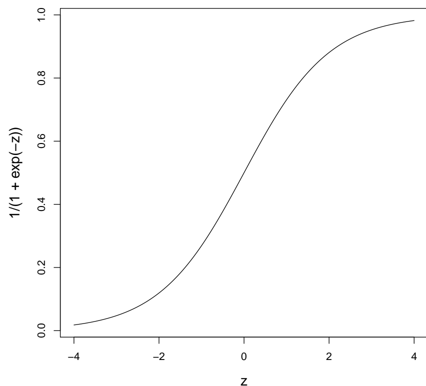
$$h_m = g(z_m),$$

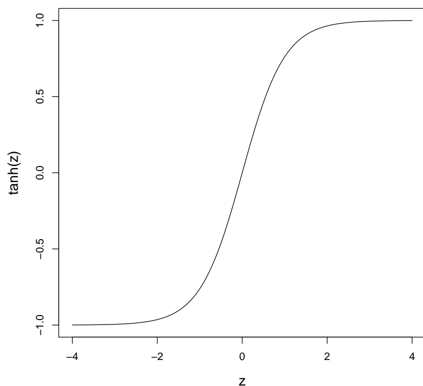
where g is a nonlinear activation function.

Sigmoid activation functions

- The first generation of multi-layer networks used the logistic activation function
$$g(z) = \Lambda(z) = \frac{1}{1+e^{-z}} \in [0,1]$$
taking values in $[0,1]$, or the hyperbolic tangent activation function
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\Lambda(2z) - 1 \in [-1,1]$$
- These activation functions are said to be "sigmoid" (S-shaped).
- Sigmoid units saturate across most of their domain, and are only strongly sensitive to their input when z is near 0. This can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged.

Sigmoid activation functions





Rectified linear units

- Rectified linear units (ReLU) use the activation function

$$g(z) = \max(0, z).$$

- Rectified linear units are easy to optimize because they are similar to linear units: the only difference is that a rectified linear unit outputs zero across half its domain.
- This makes the derivatives through a rectified linear unit remain large whenever the unit is active.

Rectified linear unit activation functions

ReLU

pmax(0, z)

z

Radial basis function (RBF) units

- In a radial basis function (RBF) unit, the output is computed as a function of the distance (typically, Euclidean) between x and the unit's weight vector w_m:

$$h_m = g(-\gamma_m \|x - w_m\|),$$

  where $\gamma_m > 0$ is a scaling parameter.

- Usually, g is decreasing, $g(0) = 1$ and $\lim_{d \to \infty} g(d) = 0$, such as

$$g(d) = \exp(-d^2)$$

- This kind of unit becomes more active as x becomes closer to weight vector w_m, which can be seen as a template or prototype.
- Because it saturates to 0 for most x, it can be difficult to optimize.

Overview

1. Multilayer feedforward neural networks
   - Hidden units
   - Output units
   - Architecture
2. Learning
3. Complexity control

Output units for regression

- A neural network can be used for regression or classification.
- For regression, there is only one linear output unit ($K = 1$). However, we can easily generalize the model to $K > 1$ outputs.
- The k-th output is computed as

$$z_k = \sum_{m=1}^{M} w_{km} h_m + w_{k0} = w_k^T h + w_{k0}$$

$$\widehat{y}_k = z_k$$

- The output units are similar to hidden units, except that their activation function is linear.

Output units for binary classification

- For binary classification, we usually have one output unit with a logistic activation function:

$$\widehat{y} = \Lambda\left(\sum_{m=1}^{M} w_m h_m + w_0\right) = \Lambda(w^T h + w_0) \in [0, 1]$$

  This output can be made to approximate the conditional probability $P(x) = \mathbb{P}(Y = 1|x)$.

- When there is no hidden layer, the input-output equation is

$$\widehat{y} = \Lambda(w^T x + w_0)$$

  It is exactly the binomial logistic regression model.

Output units for c-class classification

- For c-class classification, there are $K = c$ output units with the kth unit modeling the probability of class k. We use the softmax function

$$\widehat{y}_k = g_k(z) = \frac{\exp(z_k)}{\sum_{\ell=1}^{K} \exp(z_\ell)}$$

with $z_k = w_k^T h + w_{k0}$ and $z = (z_1, \ldots, z_K)$.

- This is exactly the transformation used in the multinomial logistic regression model; it produces positive probability estimates that sum to one.

Overview

1. Multilayer feedforward neural networks
   - Hidden units
   - Output units
   - Architecture
2. Learning
3. Complexity control

Architecture design

- A key design consideration for neural networks is determining the architecture, i.e., the overall structure of the network: how many units it should have and how these units should be connected to each other.
- In multilayer feedforward networks, groups of units (layers) are arranged in a chain structure, with each layer being a function of the layer that precedes it. The vector of outputs from the 1st layer is

$$\underbrace{h^{(1)}}_{M_1\times 1} = g^{(1)}(\underbrace{W^{(1)}}_{M_1\times p}\underbrace{x}_{p\times 1} + \underbrace{w_0^{(1)}}_{M_1\times 1}),$$

the second-layer output vector is

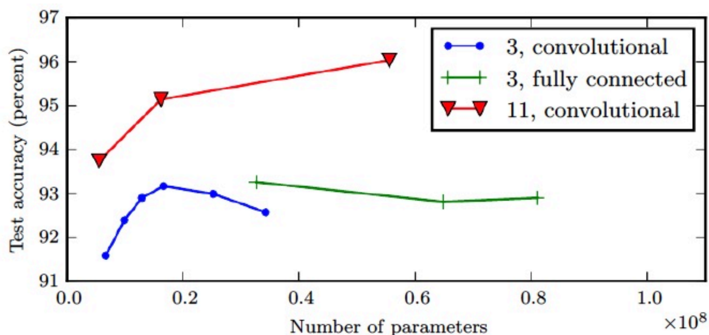
$$\underbrace{h^{(2)}}_{M_2\times 1} = g^{(2)}(\underbrace{W^{(2)}}_{M_2\times M_1}\underbrace{h^{(1)}}_{M_1\times 1} + \underbrace{w_0^{(2)}}_{M_2\times 1}),$$

and so on. Here, $W^{(l)}$ is the matrix of weights for connections into hidden layer l.

How many layers?

- It can be shown that neural networks with only one hidden layer of nonlinear units are universal approximators: they can approximate any sufficiently smooth (e.g., continuous) function to any desired accuracy.
- However, the hidden layer may be infeasibly large and may fail to learn and generalize correctly.
- In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
- Empirical results show that deeper models tend to perform better, not merely because the model is larger.

Example



This experiment from Goodfellow et al. (2014) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million.

Overview

1. Multilayer feedforward neural networks

2. Learning
   - Loss functions
   - Back-propagation
   - Optimization algorithms
   - Weight initialization
   - NNs with R

3. Complexity control

Overview

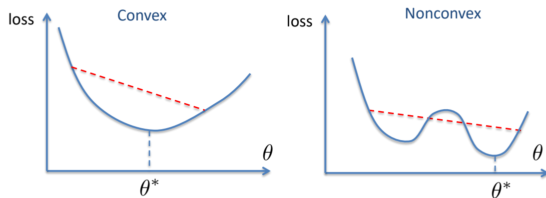
1 Multilayer feedforward neural networks

2 Learning
- Loss functions
- Back-propagation
- Optimization algorithms
- Weight initialization
- NNs with R

3 Complexity control

Gradient-based learning

- Designing and training a neural network is not much different from training any other machine learning model by minimizing a loss (cost, error) function.
- The largest difference between the models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
- This means that neural networks are usually trained using iterative, gradient-based optimization algorithms that merely drive the cost function to a local minimum.



Loss function

- We first need to define a loss function $\mathcal{L}(\widehat{y}, y)$.
- Given a learning set $\{(x_i, y_i)\}_{i=1}^n$, we then minimize the average loss

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x_i; \theta), y_i),$$

where
  - θ denote the vector of all connection weights (the learnable parameters) and
  - $f(x; \theta)$ the vector of outputs for input vector x.
- This is an estimate of the expected loss

$$\mathbb{E}_{X,Y} \mathcal{L}(f(X; \theta), Y)$$

Loss function for regression

- For regression, we often use the sum-of-squares loss function:

$$\mathcal{L}(f(x;\theta), y) = \|y - f(x;\theta)\|^2 = \sum_{k=1}^{K} (y_k - f_k(x;\theta))^2$$

- Minimizing $J(\theta)$ is equivalent to maximizing the conditional likelihood, assuming a Gaussian error model

$$Y = f(x;\theta) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I}_K)$$

(see next slide).

Sum-of-squares loss and Gaussian error assumption

- Assuming $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I}_K)$, the conditional likelihood is

$$L(\theta) = \prod_{i=1}^{n} p(y_i \mid x_i) \propto \prod_{i=1}^{n} \exp\left(-\frac{1}{2\sigma^2}\|y_i - f(x_i;\theta)\|^2\right)$$

$$\propto \exp\left(-\frac{1}{2\sigma^2} \underbrace{\sum_{i=1}^{n} \|y_i - f(x_i;\theta)\|^2}_{\mathcal{L}(f(x_i;\theta),y_i)}\right)$$

- Maximizing the conditional likelihood is equivalent to minimizing the average sum-of-squares loss.

Loss function for classification

- For classification we use cross-entropy (deviance) loss function:

$$\mathcal{L}(f(x;\theta), y) = -\sum_{k=1}^{c} y_k \log f_k(x;\theta),$$

where $y_k = I(y = k)$. The corresponding classifier is

$$C(x) = \arg\max_k f_k(x;\theta).$$

- If $f_k(x;\theta)$ is a model of $P_k(x) = \mathbb{P}(Y = k \mid X = x)$, then $J(\theta)$ equals minus the log-likelihood $\ell(\theta)$ (see next slide).

Cross-entropy and conditional likelihood

- The conditional likelihood is

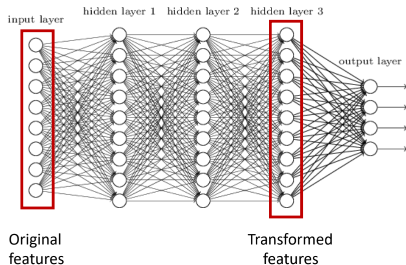
$$L(\theta) = \prod_{i=1}^{n} \mathbb{P}(Y_i = y_i \mid X_i = x_i)$$
$$= \prod_{i=1}^{n} \prod_{k=1}^{c} f_k(x_i; \theta)^{y_{ik}}$$

- The conditional log-likelihood is

$$\ell(\theta) = \sum_{i=1}^{n} \underbrace{\sum_{k=1}^{c} y_{ik} f_k(x_i; \theta)}_{-\mathcal{L}(f(x_i;\theta), y_i)}$$

Relation with logistic regression

- With the logistic ($c = 2$) or softmax ($c > 2$) activation function and the cross-entropy error function, the neural network model is exactly a logistic regression model in which the hidden unit outputs are the predictors.
- All the parameters are estimated by maximum likelihood.
- Training a neural network is equivalent to learning a transformation from the input space to a feature space, and learning a logistic regression classifier in the feature space at the same time.



Overview

1 Multilayer feedforward neural networks

2 Learning
- Loss functions
- Back-propagation
- Optimization algorithms
- Weight initialization
- NNs with R

3 Complexity control

Principle

- The vector θ^* minimizing $J(\theta)$ does not have a closed-form expression: we need to use an iterative optimization algorithm.
- Most optimization algorithms require to compute the gradient of $J(\theta)$ at each step.
- The gradient of $J(\theta)$ can be easily derived using the chain rule for differentiation.
- The corresponding algorithm is called back-propagation (BP).
- For ease of exposition, we present the BP algorithm in the case of two hidden layers. Generalization to any number of hidden layers is straightforward.

Propagation equations and loss function

- Propagation equations (see next slide):

$$z_m = \sum_j w_{mj} x_j + w_{m0}, \quad h_m = g(z_m), \quad m = 1, \ldots, M$$

$$z_q = \sum_m w_{qm} h_m + w_{q0}, \quad h_q = g(z_q), \quad q = 1, \ldots, Q$$

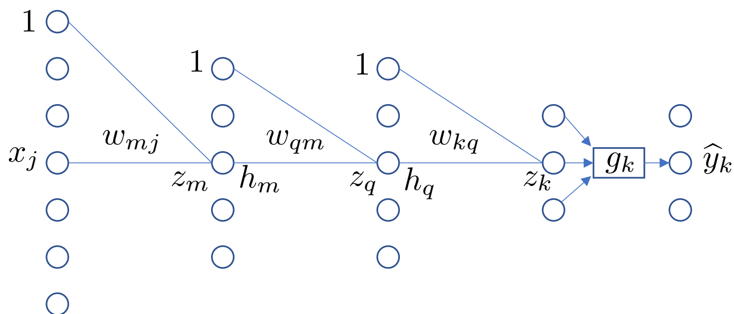
$$z_k = \sum_q w_{kq} h_q + w_{k0}, \quad k = 1, \ldots, K$$

$$\widehat{y}_k = g_k(z_1, \ldots, z_K) = f_k(x, \theta), \quad k = 1, \ldots, K$$

- Loss function:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x_i; \theta), y_i).$$

Notations



Derivatives w.r.t to the w_{kq}

- We compute the gradient of $\mathcal{L}(f(x;\theta),y)$.
- The derivatives w.r.t to the weights w_{kq} can be computed as

$$\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial w_{kq}} = \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_k}}_{\delta_k} \underbrace{\frac{\partial z_k}{\partial w_{kq}}}_{h_q}$$

with

$$\delta_k = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial \widehat{y}_{k'}} \frac{\partial \widehat{y}_{k'}}{\partial z_k}$$

- With the sum-of-squares criterion and linear output units, we have $\mathcal{L}(f(x;\theta),y) = \sum_k (\widehat{y}_k - y_k)^2$ and $\widehat{y}_k = z_k$, so

$$\delta_k = 2(\widehat{y}_k - y_k)$$

Derivatives w.r.t to the w_{qm}

The derivatives w.r.t to the weights w_{qm} can be computed as

$$\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial w_{qm}} = \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_q}}_{\delta_q} \underbrace{\frac{\partial z_q}{\partial w_{qm}}}_{h_m}$$

with

$$\delta_q = \sum_k \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_k}}_{\delta_k} \underbrace{\frac{\partial z_k}{\partial h_q}}_{w_{kq}} \underbrace{\frac{\partial h_q}{\partial z_q}}_{g'(z_q)}$$

$$\delta_q = g'(z_q) \sum_k \delta_k w_{kq}$$

Derivatives w.r.t to the w_{mj}

The derivatives w.r.t to the weights w_{mj} can be obtained as

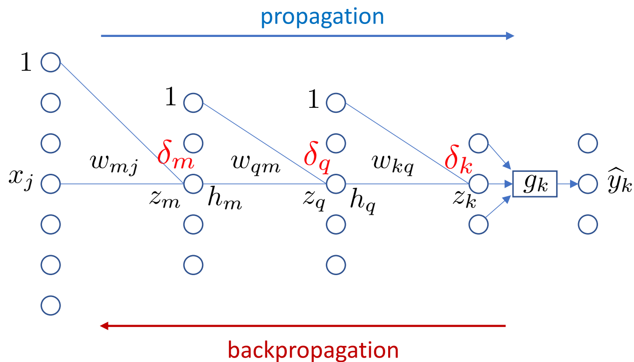
$$\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial w_{mj}} = \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_m}}_{\delta_m} \underbrace{\frac{\partial z_m}{\partial w_{mj}}}_{x_j}$$

with

$$\delta_m = \sum_q \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_q}}_{\delta_q} \underbrace{\frac{\partial z_q}{\partial h_m}}_{w_{qm}} \underbrace{\frac{\partial h_m}{\partial z_m}}_{g'(z_m)}$$

$$= g'(z_m) \sum_q \delta_q w_{qm}$$

Back-propagation algorithm



Advantage of back-propagation

- Each gradient evaluation requires $O(N)$ operations, where N is the number of weights in the network. Consequently, the algorithm can be applied to large networks.
- The calculations in back-propagation are local: each hidden unit passes and receives information only to and from units that share a connection. Hence, backpropagation can be implemented efficiently on a parallel architecture computer.

Overview

1. Multilayer feedforward neural networks

2. Learning
   - Loss functions
   - Back-propagation
   - Optimization algorithms
   - Weight initialization
   - NNs with R

3. Complexity control

Batch learning with gradient descent

- The simplest approach to using gradient information is gradient descent: we update the weights by making a small step in the direction of the negative gradient, so that

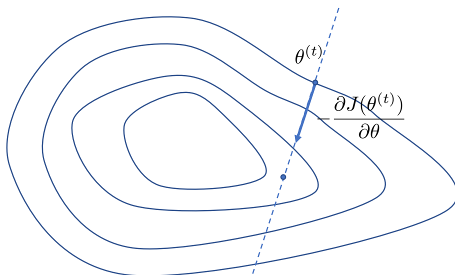
$$\begin{aligned}\theta^{(t+1)} &= \theta^{(t)} - \eta_t \frac{\partial J(\theta^{(t)})}{\partial \theta} \\ &= \theta^{(t)} - \eta_t \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \mathcal{L}(f(x_i, \theta^{(t)}), y_i)}{\partial \theta}\end{aligned}$$

  Coefficient η_t is called the learning rate.

- The error function is the average loss over the training set, and so each step requires that the entire training set be processed in order to evaluate the gradient. This is called batch learning.

Batch gradient descent with line search

The learning rate η_t for batch learning was originally taken to be a constant (learning is then very slow); it can also be optimized by a line search, which minimizes the error function at each update.



Quasi-Newton methods

- Faster learning can be achieved using more powerful optimization algorithms.
- The Newton-Raphson method cannot be used, because the second derivative matrix of J (the Hessian) is very large.
- Quasi-Newton methods are based on approximations of the Hessian. For instance, a diagonal approximation can be computed in $O(N)$ time. Other methods like the BFGS algorithm update the Hessian estimate by analyzing successive gradient vectors.

Stochastic gradient descent

- Batch learning is not feasible with very large learning sets. Learning can then be carried out online – processing each observation one at a time, updating the weights after each learning example, and cycling through the learning set many times.
- In this case, the update equation become

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\partial \mathcal{L}(f(x_t, \theta^{(t)}), y_t)}{\partial \theta}$$

  where (x_t, y_t) is the learning example presented at iteration t.
- Here, $\frac{\partial \mathcal{L}(f(x_t,\theta^{(t)}),y_t)}{\partial \theta}$ can be seen as an estimate of $\frac{\partial \mathbb{E}\mathcal{L}(f(X,\theta^{(t)}),Y)}{\partial \theta}$.
- Online training (also called stochastic gradient descent – SGD) allows the network to handle very large training sets, and also to update the weights as new observations come in.

Minibatch

- $\frac{\partial \mathcal{L}(f(x_t,\theta^{(t)}),y_t)}{\partial \theta}$ is a high-variance estimate of $\frac{\partial \mathbb{E}\mathcal{L}(f(X,\theta^{(t)}),Y)}{\partial \theta}$.
- In practice, we often average the gradient over a randomly selected subset of $\nu \ll n$ learning examples $\{(x_{i_1}, y_{i_1}), \ldots, (x_{i_n}, y_{i_\nu})\}$ called a minibatch.
- The update equation is then

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{1}{\nu} \sum_{j=1}^{\nu} \frac{\partial \mathcal{L}(f(x_{i_j}, \theta^{(t)}), y_{i_j})}{\partial \theta}$$

- A minibatch is randomly selected before each weight update.

SGD algorithm with a fixed learning rate

Require: Learning rate η
Require: Initial parameter θ
while stopping criterion not met **do**
Sample a minibatch $\{(x_{i_1}, y_{i_1}), \ldots, (x_{i_n}, y_{i_\nu})\}$ of ν examples from the training set
Compute gradient estimate $\widehat{\boldsymbol{g}} = \frac{1}{n} \sum_{j=1}^{\nu} \frac{\partial \mathcal{L}(f(x_{i_j}, \theta), y_{i_j})}{\partial \theta}$
Update the learning rate η
Apply update: $\theta \leftarrow \theta - \eta \widehat{\boldsymbol{g}}$
end while

Learning rate

- It is necessary to gradually decrease the learning rate over time. The decrease should be neither too fast, nor too slow to guarantee convergence.
- In practice, it is common to decay the learning rate linearly until some iteration τ:

$$\eta_t = \begin{cases} \left(1 - \frac{t}{\tau}\right)\eta_0 + \frac{t}{\tau}\eta_\tau & \text{if } t < \tau \\ \eta_\tau & \text{if } t \geq \tau \end{cases}$$

- Parameter τ may be set to the number of iterations required to make a few hundred passes through the training set.
- While SGD remains a very popular optimization strategy, learning with it can sometimes be slow, and several accelerated learning algorithms have been proposed.

The RMSprop algorithm

- The RMSprop algorithm uses a separate learning rate for each parameter and automatically adapts these learning rates throughout the course of learning.
- Idea: the parameters with smaller partial derivative of the loss should have a correspondingly larger learning rate.
- RMSprop adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of an exponentially decaying average of historical squared values of the gradient.
- RMSProp is currently one of the most widely used methods for training deep neural networks.

RMSprop algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter θ
Require: Small constant δ, usually 10^{-6} used to stabilize division by small numbers
Initialize accumulation variables $\mathbf{r} = 0$
while stopping criterion not met **do**
Sample a minibatch $\{(x_{i_1}, y_{i_1}), \ldots, (x_{i_n}, y_{i_\nu})\}$ of ν examples from the training set
Compute gradient estimate $\widehat{\mathbf{g}} = \frac{1}{n} \sum_{j=1}^{\nu} \frac{\partial \mathcal{L}(f(x_{i_j}, \theta^{(t)}), y_{i_j})}{\partial \theta}$
Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho) \widehat{\mathbf{g}} \odot \widehat{\mathbf{g}}$ (multiplication applied element-wise)
Compute parameter update: $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \widehat{\mathbf{g}}$ (division and square root applied element-wise)
Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

Overview

1. Multilayer feedforward neural networks
2. Learning
   - Loss functions
   - Back-propagation
   - Optimization algorithms
   - Weight initialization
   - NNs with R
3. Complexity control

Importance of starting values

- Most learning algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, as well as the speed of convergence.
- Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly.

Some heuristics

- Some heuristics are available for choosing the initial scale of the weights.
- One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from

$$\mathcal{U}\left(-\frac{1}{\sqrt{m}}, +\frac{1}{\sqrt{m}}\right).$$

- Other authors suggest using the normalized initialization:

$$w_{ij} \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, +\sqrt{\frac{6}{m+n}}\right).$$

- It is best to standardize all inputs to have mean zero and standard deviation one, or to belong to $[-1, 1]$.

Overview

1 Multilayer feedforward neural networks

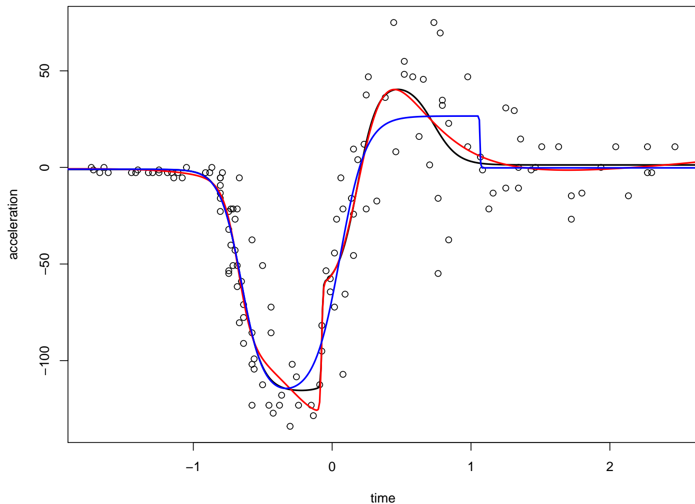
2 Learning
- Loss functions
- Back-propagation
- Optimization algorithms
- Weight initialization
- NNs with R

3 Complexity control

Shallow NN training using the nnet package

```
library('MASS')
mcycle.data<-data.frame(mcycle,x=scale(mcycle$times))
test.data<-data.frame(x=seq(-2,3,0.01))

library('nnet')

nn1<- nnet(accel ~ x, data=mcycle.data, size=5, linout = TRUE)
pred1<- predict(nn1,newdata=test.data)

nn2<- nnet(accel ~ x, data=mcycle.data, size=5, linout = TRUE)
pred2<- predict(nn2,newdata=test.data)

nn3<- nnet(accel ~ x, data=mcycle.data, size=5, linout = TRUE)
pred3<- predict(nn3,newdata=test.data)
```

Results



Deep NN training using the keras package

```
library(keras)

model <- keras_model_sequential()

model %>% layer_dense(units = 30, activation = 'relu', input_shape = 1) %>%
layer_dense(units = 20, activation = 'relu') %>%
layer_dense(units = 5, activation = 'relu') %>%
layer_dense(units = 1, activation = 'linear')


model %>% compile(loss = 'mean_squared_error', optimizer = optimizer_rmsprop())

history <- model %>% fit(mcycle.data$x, mcycle.data$accel,
           epochs = 2000, batch_size = 30)

x=seq(-2,3,0.01)
pred <- predict(model, x)
```

Training error

```
plot(history$metrics$loss,type="l",lwd=3,,xlab="epochs",ylab="loss")
```

loss
3000
2500
2000
1500
1000
500
0 500 1000 1500 2000
epochs

Result

acceleration
50
0
−50
−100
−1
0
1
2
time

Overview

1. Multilayer feedforward neural networks
2. Learning
3. Complexity control
   - Exploring different architectures
   - Regularization
   - Early stopping
   - Dropout
   - Weight sharing

Necessity of complexity control

- Because of the universal approximation property of neural networks, the training error can, in principle, be made arbitrarily small by increasing the number of hidden units.
- However, a large neural network will be prone to overfitting and will typically have bad generalization performance.
- We need to control the complexity of the model. Many approaches have been proposed. We will review some of these approaches:
  1. Exploring different architectures
  2. Early stopping
  3. Regularization
  4. Dropout
  5. Weight sharing, as implemented in convolutional networks

Overview

1. Multilayer feedforward neural networks

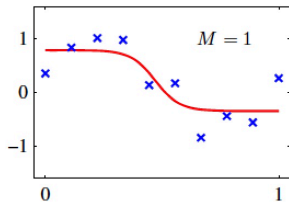
2. Learning

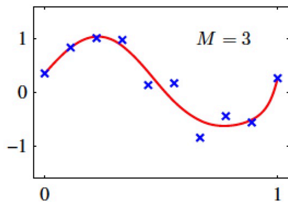
3. Complexity control
   - Exploring different architectures
   - Regularization
   - Early stopping
   - Dropout
   - Weight sharing

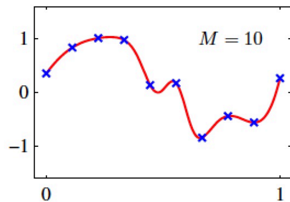
Optimizing the number of hidden units

- The most basic approach is to explore different architectures.
- If we limit ourselves to shallow networks with one hidden layer, a simple way to define a family of nested models is to consider networks with different values M of hidden units.
- For each network, the generalization error is estimated using a validation set or using cross-validation, and the best value of M is selected.
- However, we have seen that better results may be often be obtained with deeper architectures. Considering architectures with different numbers of hidden layer considerably enlarges the search space.

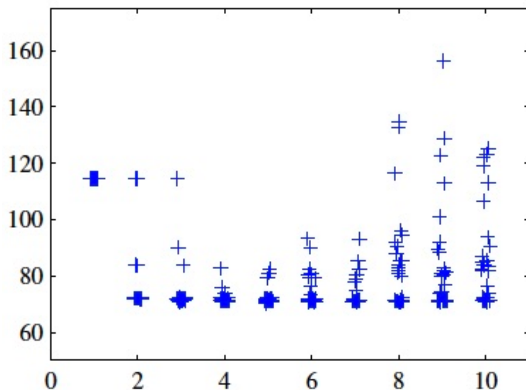
Example







Sum-of-squares test error as a function of M



Plot of the sum-of-squares test error for a polynomial data set versus the number of hidden units in the network, with 30 random starts for each network size, showing the effect of local minima.

Overview

1. Multilayer feedforward neural networks
2. Learning
3. Complexity control
   - Exploring different architectures
   - Regularization
   - Early stopping
   - Dropout
   - Weight sharing

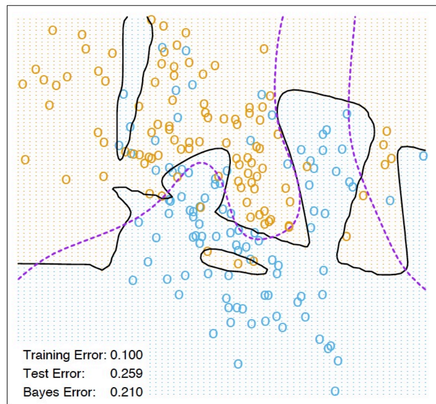
Weight decay

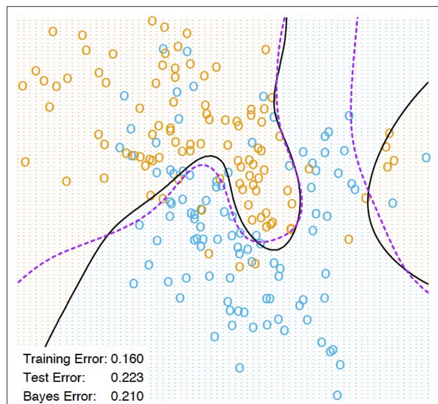
- An alternative approach to control the complexity of a neural network is to choose a relatively large value for M and to add a norm penalty term (regularizer) to the error function.
- The simplest regularizer is the quadratic (L_2), giving a regularized error

$$\widetilde{J}(\theta) = J(\theta) + \lambda \theta^T \theta$$

- This regularizer is also known as weight decay. It is similar to ridge regression. The regularization coefficient λ is usually determined by cross-validation.
- This regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector θ.
- Other regularizer choices, such as L_1 (Lasso) correspond to a different prior (Laplace).

Example 1: classification





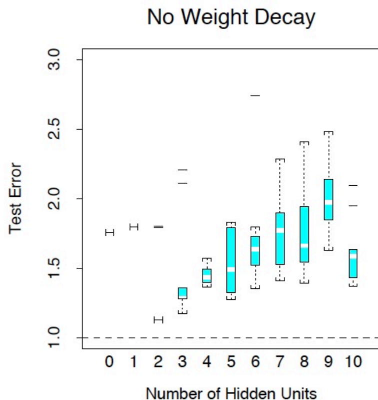
Example 2: regression

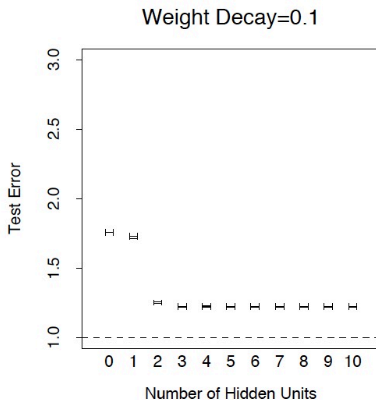
- Model $Y = f(X) + \varepsilon$ with

$$f(X) = \Lambda(a_1^T X) + \Lambda(a_2^T X),$$

  $X = (X_1, X_2)$, $a_1 = (3, 3)$, $a_2 = (3, -3)$, $\mathrm{Var}(f(X))/\mathrm{Var}(\varepsilon) = 4$.
- Training set of size 100, test set of size 10,000.
- Neural networks with weight decay and various numbers of hidden units.
- 10 random starting weights for each configutation.

Results without and with weight decay





Influence of the weight decay hyper-parameter

Sum of Sigmoids, 10 Hidden Unit Model

Test Error

1.0 1.2 1.4 1.6 1.8 2.0 2.2

0.00 0.02 0.04 0.06 0.08 0.10 0.12 0.14

Weight Decay Parameter

Generalization

- More generally, we can penalize each layer of weights with a different coefficient.
- For instance, in the case of one hidden layer, we have

$$\lambda_1 \sum_{m=1}^{M} \sum_{j=1}^{p} w_{mj}^2 + \lambda_2 \sum_{k=1}^{K} \sum_{m=1}^{M} w_{km}^2$$

- This corresponds to the Gaussian prior

$$p(\mathbf{w} \mid \lambda_1, \lambda_2) \propto \exp\left(-\lambda_1 \sum_{m=1}^{M} \sum_{j=1}^{p} w_{mj}^2 - \lambda_2 \sum_{k=1}^{K} \sum_{m=1}^{M} w_{km}^2\right)$$

- This approach has some theoretical advantages (it makes the regularizer equivariant under linear transformations of the inputs and outputs), but we now have two (or more) hyperparameters to fix.

Shallow NN training with weight decay using nnet

```
library('MASS')
mcycle.data<-data.frame(mcycle,x=scale(mcycle$times))
test.data<-data.frame(x=seq(-2,3,0.01))

library('nnet')

nn1<- nnet(accel ~ x, data=mcycle.data, size=2, linout = TRUE, decay=0)
pred1<- predict(nn1,newdata=test.data)

nn2<- nnet(accel ~ x, data=mcycle.data, size=10, linout = TRUE, decay=0)
pred2<- predict(nn2,newdata=test.data)

nn3<- nnet(accel ~ x, data=mcycle.data, size=10, linout = TRUE, decay=1)
pred3<- predict(nn3,newdata=test.data)
```

Results

M=2
M=10, $\lambda = 0$
M=10, $\lambda = 1$

accel

x

50 0 −50 −100

−1 0 1 2

Selection of λ by 10-fold cross-validation

CV

700
650
600

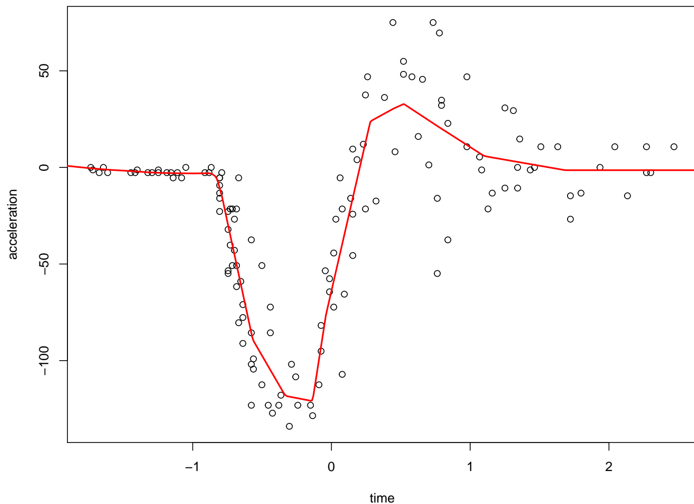
0 2 4 6 8 10

lambda

Deep NN training with weight decay using keras

```
library('keras')

model <- keras_model_sequential()
model %>%
layer_dense(units = 50, activation = 'relu', input_shape = 1,
kernel_regularizer = regularizer_l2(l=0.1)) %>%
layer_dense(units = 30, activation = 'relu',
kernel_regularizer = regularizer_l2(l=0.1)) %>%
layer_dense(units = 20, activation = 'relu',
kernel_regularizer = regularizer_l2(l=0.1)) %>%
layer_dense(units = 1, activation = 'linear')

model %>% compile(loss = 'mean_squared_error',optimizer = optimizer_rmsprop())
history <- model %>% fit(mcycle.data$x, mcycle.data$accel, epochs = 2000,
           batch_size = 30)
pred <- predict(model, x)
```

Results

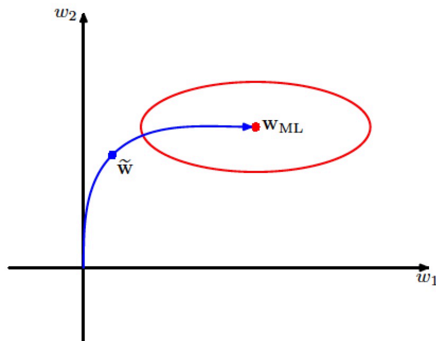


Overview

1. Multilayer feedforward neural networks
2. Learning
3. Complexity control
   - Exploring different architectures
   - Regularization
   - Early stopping
   - Dropout
   - Weight sharing

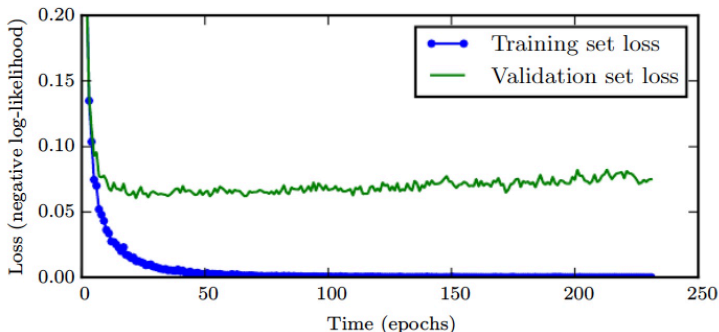
Early stopping

- An alternative to regularization as a way of controlling the effective complexity of a network is early stopping: we train the model only for a while, and stop well before we approach a local minimum.



- Since the weights start at a highly regularized (linear) solution, this has the effect of shrinking the final model toward a linear model.
- A validation dataset is needed for determining when to stop, since we expect the validation error to start increasing.

Example



Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or epochs). The training loss decreases consistently over time, but the validation loss eventually begins to increase again, forming an asymmetric U-shaped curve.

Overview

1. Multilayer feedforward neural networks
2. Learning
3. Complexity control
   - Exploring different architectures
   - Regularization
   - Early stopping
   - Dropout
   - Weight sharing

Dropout as a substitute of bagging

- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks.
- Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Principles of the dropout method

- To train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as SGD.
- Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each unit is sampled independently from all of the others.
- The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5.
- We then run forward propagation, back-propagation, and the learning update as usual.

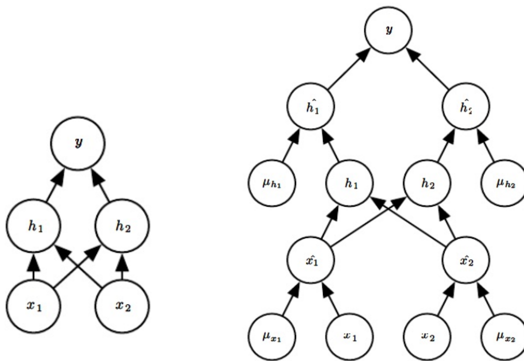
Example

Base network

Ensemble of subnetworks

Formal analysis

- More formally, suppose that a binary mask vector $\boldsymbol{\mu}$ specifies which units to include, and $J(\theta, \boldsymbol{\mu})$ defines the average loss of the model defined by parameters θ and mask $\boldsymbol{\mu}$.



Formal analysis (continued)

- At each iteration the average update is

$$\mathbb{E}_{\boldsymbol{\mu}}\left[\frac{\partial J(\theta,\boldsymbol{\mu})}{\partial\theta}\right]=\frac{\partial\mathbb{E}_{\boldsymbol{\mu}}J(\theta,\boldsymbol{\mu})}{\partial\theta}$$

- Then dropout training consists in minimizing $\mathbb{E}_{\boldsymbol{\mu}}J(\theta,\boldsymbol{\mu})$.
- The expectation contains exponentially many terms but we can obtain an unbiased estimate of its gradient by sampling values of $\boldsymbol{\mu}$.

Difference with bagging

Dropout training is not quite the same as bagging:

Bagging	Dropout
Models are all independent	Models share parameters, each model inheriting a different subset of parameters from the parent neural network
Each model is trained to convergence on its respective training set	Only a tiny fraction of the possible sub-networks are each trained for a single step; the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters

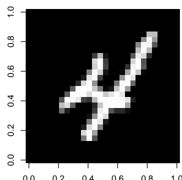
Beyond these differences, dropout follows the bagging algorithm. For example, the training set for each sub-network (a minibatch) is a subset of the original training set sampled with replacement.

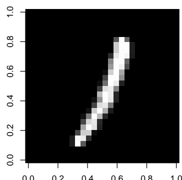
Prediction

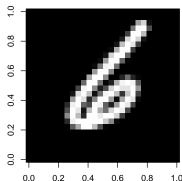
- To make a prediction, a bagged ensemble must accumulate votes from all of its members. In the case of dropout, this is impractical because there are exponentially many models.
- A good heuristic is to evaluate the output of one model: the model with all units, but with the weights going out of each unit i multiplied by the probability of including unit i. In this way, we capture the right expected value of the output from each unit. This approach is called the weight scaling inference rule.
- Because we usually use an inclusion probability of $1/2$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training.
- There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

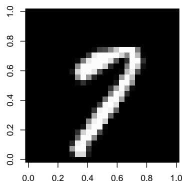
Example: MNIST dataset

Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.









Input/output formatting

```
library('keras')

mnist <- dataset_mnist()
X_train <- mnist$train$x
Y_train <- mnist$train$y
X_test <- mnist$test$x
Y_test <- mnist$test$y

# reshape
x_train <- array_reshape(X_train, c(nrow(X_train), 784))
x_test <- array_reshape(X_test, c(nrow(X_test), 784))

# rescale
x_train <- x_train / 255
x_test <- x_test / 255

y_train <- to_categorical(Y_train, 10)
y_test <- to_categorical(Y_test, 10)
```

Model definition and learning

```
model <- keras_model_sequential()

model %>%
layer_dense(units = 256, activation = 'relu', input_shape = 784) %>%
layer_dropout(rate = 0.4)%>%
layer_dense(units = 128, activation = 'relu') %>%
layer_dropout(rate = 0.3) %>%
layer_dense(units = 10, activation = 'softmax')

model %>% compile(
loss = 'categorical_crossentropy',
optimizer = optimizer_rmsprop(),
metrics = c('accuracy'))

history <- model %>% fit( x_train, y_train,
epochs = 50, batch_size = 128, validation_split = 0.2)
```

Learning curves

Results

```
model %>% evaluate(x_test, y_test)

$loss
0.1159833

$acc
0.982
```

Overview

1 Multilayer feedforward neural networks

2 Learning

3 Complexity control
- Exploring different architectures
- Regularization
- Early stopping
- Dropout
- Weight sharing

Parameter sharing

- Another way to reduce the number of degrees of freedom in a model is to force sets of parameters to be equal. This method of regularization is often referred to as parameter/weight sharing.
- A significant advantage of parameter sharing over regularization via a norm penalty is that only a subset of the parameters (the unique set) need to be stored in memory. In large networks, this can lead to significant reduction in the memory requirement of the model.
- This idea is implemented in convolutional neural networks (CNNs).