Advanced Computational Econometrics: Machine Learning

Chapter 10: Neural Networks and Deep Learning

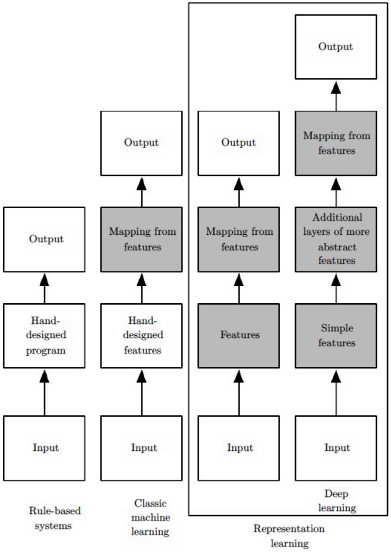
Thierry Denœux

July-August 2019

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 chapter we describe the most widely used multilayer feedforward neural networks:
  - Multilayer perceptrons
  - Deep convolutional networks

Neural networks within AI



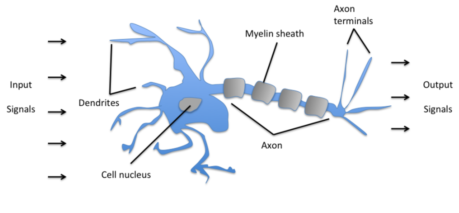
Historical perspective

Three phases:

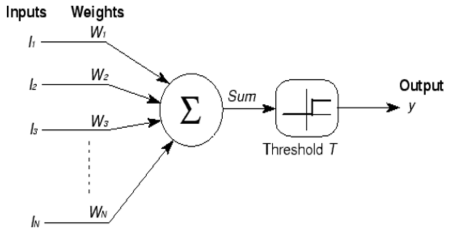
1. Perceptron (1955-1965)
2. Multi-layer neural networks (1985-1995)
3. Deep networks (2010-)

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.

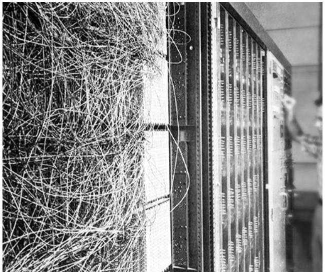


Schematic of a biological neuron.



Perceptron

- F. Rosenblatt. *The perceptron, a perceiving and recognizing automaton (Project PARA)*. Cornell Aeronautical Laboratory, 1957.
- Main idea: an algorithm to learn weights so as to solve binary classification tasks. (Mark 1 perceptron: custom-built hardware designed for image recognition with an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.)



Perceptron algorithm

Error function

- The perceptron learning algorithm tries to find a separating hyperplane with equation $x^T\beta + \beta_0 = 0$ by minimizing the distance of misclassified points to the decision boundary
- A response $y_i \in \{-1, 1\}$ is misclassified if $y_i(x_i^T\beta + \beta_0) < 0$. The goal is to minimize
$$D(\beta, \beta_0) = -\sum_{i \in \mathcal{M}} y_i(x_i^T\beta + \beta_0),$$
where $\mathcal{M}$ indexes the set of misclassified points
- The quantity is nonnegative and proportional to the distance of the misclassified points to the decision boundary.

Perceptron algorithm

Learning rule

- The gradient (assuming $\mathcal{M}$ is fixed) is given by

$$\frac{\partial D(\beta, \beta_0)}{\partial \beta} = -\sum_{i \in \mathcal{M}} y_i x_i, \quad \frac{\partial D(\beta, \beta_0)}{\partial \beta_0} = -\sum_{i \in \mathcal{M}} y_i$$

- The algorithm in fact uses stochastic gradient descent: rather than computing the sum of the gradient contributions of each observation followed by a step in the negative gradient direction, a step is taken after each observation is visited.
- The misclassified observations are visited in some sequence and the parameters β are updated via

$$\begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} \leftarrow \begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} + \eta \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix},$$

where the learning rate η can be taken to be 1 without loss in generality.

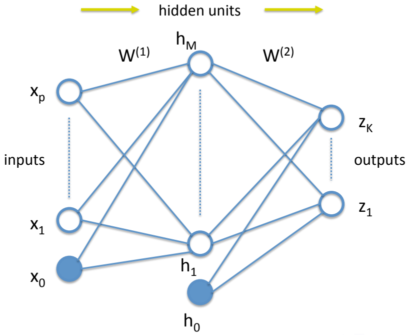
Perceptron algorithm

Properties

- If the classes are linearly separable, it can be shown that the algorithm converges to a separating hyperplane in a finite number of steps
- However, there are a number of problems with this algorithm:
    - The "finite" number of steps can be very large. The smaller the gap, the longer the time to find it.
    - When the data are not separable, the algorithm will not converge, and cycles develop. The cycles can be long and therefore hard to detect.
    - The perceptron algorithm does not generalize readily to $K > 2$ classes.

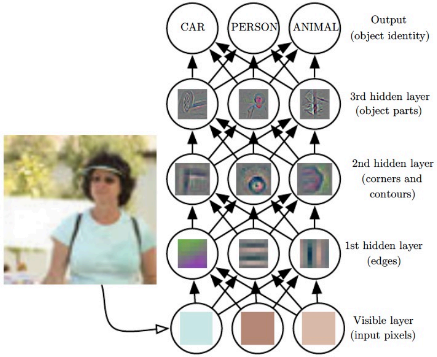
Multilayer neural networks

- 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).



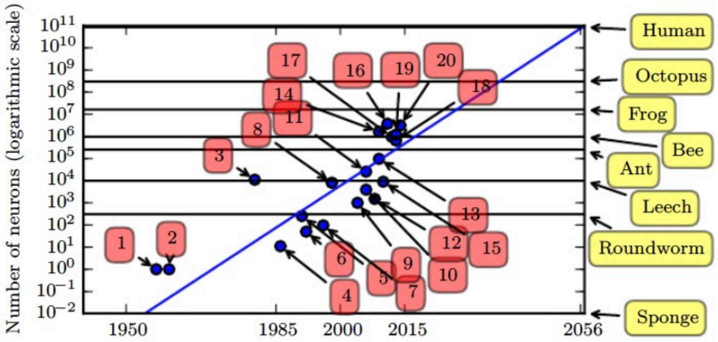
Deep networks

- 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.


CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

Increase of neural network size

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

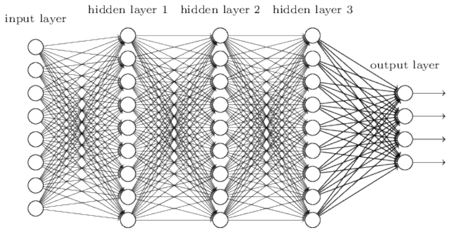


1: Perceptron; 4: Early back-propagation network; 8: LeNet-5 (LeCun et al., 1998b); 20: GoogLeNet

Overview

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

Equation of hidden units

- Each hidden neuron m computes a weighted sum of its inputs

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

where w_{mj} is the connection weight between input unit j and hidden unit m, w_m is the vector of weights of unit m, and w_{m0} is a bias term (which 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 logistic sigmoid activation functions

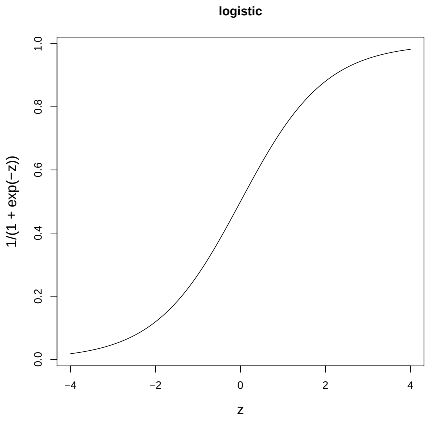
$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \in [0,1]$$

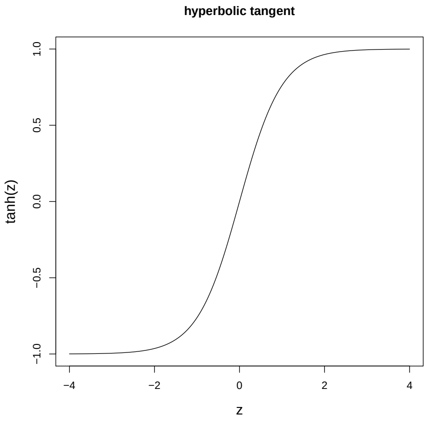
taking values in $[0,1]$, or hyperbolic tangent activation functions

$$g(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} = 2\sigma(2z)-1 \in [-1,1]$$

- Sigmoidal units saturate across most of their domain, and are only strongly sensitive to their input when z is near 0. The widespread saturation of sigmoidal units 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

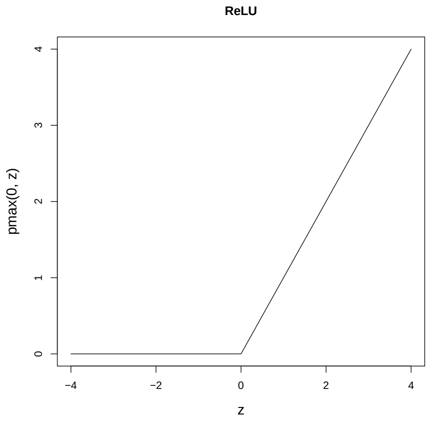
$$g(z) = \max(0, z).$$

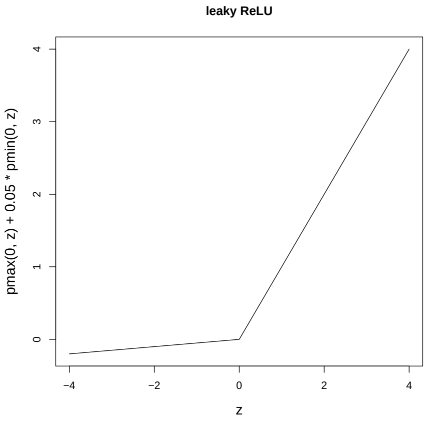
- Rectified linear units are easy to optimize because they are similar to linear units: the only difference is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active.
- Variation: leaky ReLU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

$$g(z) = \max(0, z) + \alpha \min(0, z),$$

where α takes a small value such as 0.01.

Rectified linear unit activation functions





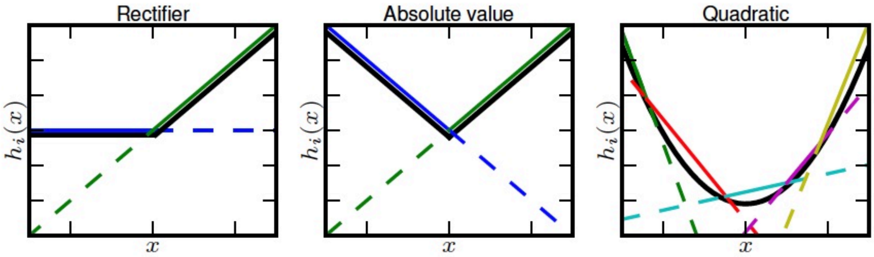
Maxout units

- Maxout units generalize rectified linear units.
- Assume that the previous layer has M linear units with outputs $\boldsymbol{z} = (z_1, \ldots, z_M)$
- Instead of applying an element-wise function $g(z)$, maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of one of these groups

$$\begin{aligned} g(\boldsymbol{z})_i &= \max_{j \in G(i)} z_j \\ &= \max_{j \in G(i)} w_j^T x + w_{j0} \end{aligned}$$

  where $G(i)$ is the set of indices into the inputs for group i.
- Maxout units can approximate arbitrary convex functions by piecewise linear functions.
- They can thus be seen as learning the activation function itself rather than just the relationship between units.

Convex function approximation by maxout units



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, $g(0) = 1$ and $\lim_{d \to \infty} g(d) = 0$, such as
$$g(d) = \exp(-d^2)$$
- This kind of unit becomes more active as x approaches a template or prototype w_m. Because it saturates to 0 for most x, it can be difficult to optimize.

Overview

Output units for regression

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

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

- 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 sigmoid activation function:

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

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

- This is exactly the transformation used in the binary logistic regression model: binary logistic regression correspond to a neural network without hidden units.

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 $z_k = w_{k0} + w_k^T h$ and $z = (z_1, \ldots, z_K)$.
- This is exactly the transformation used in the multi-class logistic regression model; it produces positive probability estimates that sum to one.

Overview

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.
- Most neural networks architectures arrange groups of units (layers) in a chain structure, with each layer being a function of the layer that preceded it. The vector of outputs from the 1st layer is

$$h^{(1)} = g^{(1)}\left(W^{(1)}x + w_0^{(1)}\right),$$

the second-layer output vector is

$$h^{(2)} = g^{(2)}\left(W^{(2)}h^{(1)} + w_0^{(2)}\right),$$

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

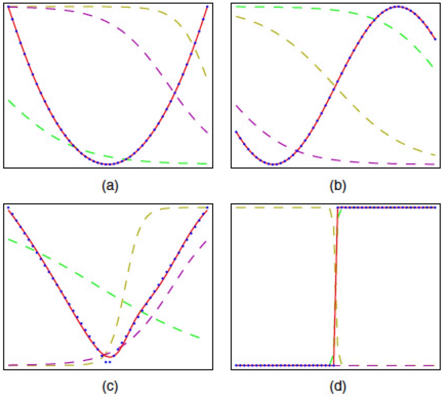
Universal approximation properties

Proposition (Universal approximation theorem)

A feedforward network with a linear output layer and one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any continuous function on a closed and bounded input subset of $\mathbb{R}^p$ with any desired non-zero amount of error, provided that the network is given enough hidden units.

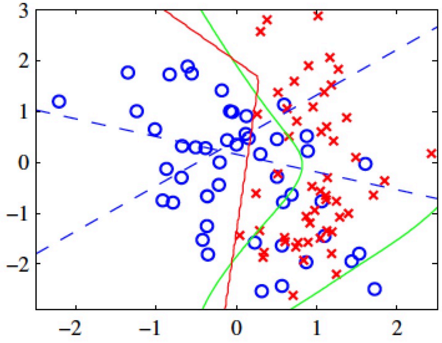
- A neural network may also approximate any function mapping from any finite dimensional discrete space to another.
- Universal approximation theorems have also been proved for a wider class of activation functions, including the now commonly used rectified linear unit.

Example: function approximation



Four functions approximated from $n = 50$ points using a neural network with $M = 3$ sigmoidal hidden units and $K = 1$ linear output unit.

Example: classification

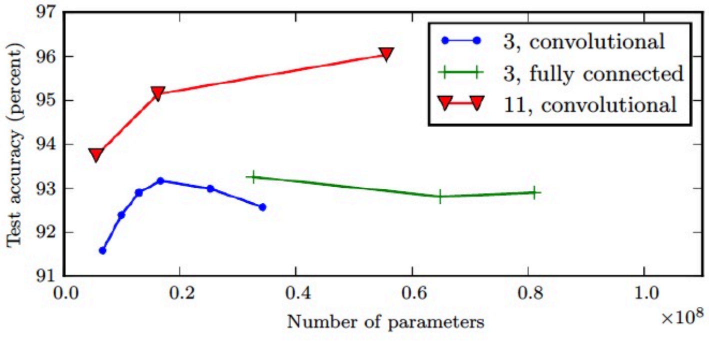


Two-class classification using a neural network with two inputs, $M = 2$ hidden units and a single output having a logistic sigmoid activation function. The dashed blue lines show the $h = 0.5$ contours for each of the hidden units, and the red line shows the $\widehat{y} = 0.5$ decision surface for the network. The green line denotes the Bayes decision boundary.

Practical utility of deep networks

- A feedforward network with a single layer is sufficient to represent any function, but the 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

Overview

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 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:

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

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

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

Loss function for classification

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

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

  where $y_k = 1$ if the learning example belongs to class k and $y_k = 0$ otherwise. The corresponding classifier is $C(x) = \arg\max_k f_k(x;\theta)$.
- If $f_k(x;\theta)$ is a model of $\mathbb{P}(Y = k \mid X = x)$, then $J(\theta)$ equals minus the log-likelihood $\ell(\theta)$.
- 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 the hidden units, and all the parameters are estimated by maximum likelihood.

Overview

Principle

- The generic approach to minimizing $J(\theta)$ is by gradient descent.
- The gradient of the error $J(\theta)$ 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 one hidden layer. Generalization to multiple hidden layers is straightforward.

Propagation equations and loss function

- Propagation equations:

$$\begin{aligned}
z_m &= \sum_{j=1}^{p} w_{mj} x_j + w_{m0}, \quad m = 1, \ldots, M \\
h_m &= g(z_m), \quad m = 1, \ldots, M \\
z_k &= \sum_{m=1}^{M} w_{km} h_m + w_{k0}, \quad k = 1, \ldots, K \\
\widehat{y}_k &= g_k(z_1, \ldots, z_K), \quad k = 1, \ldots, K \\
&= f_k(x, \theta)
\end{aligned}$$

- Loss function:

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

Derivatives w.r.t to the hidden-to-output weights

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

$$\frac{\partial \mathcal{L}(f(x;\theta), y)}{\partial w_{km}} = \frac{\partial \mathcal{L}(f(x;\theta), y)}{\partial z_k} \frac{\partial z_k}{\partial w_{km}} = \delta_k h_m$$

with

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

- With the sum-of-squares criterion and linear output units ($\widehat{y}_k = z_k$), we simply have

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

Derivatives w.r.t to the input-to-hidden weights

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

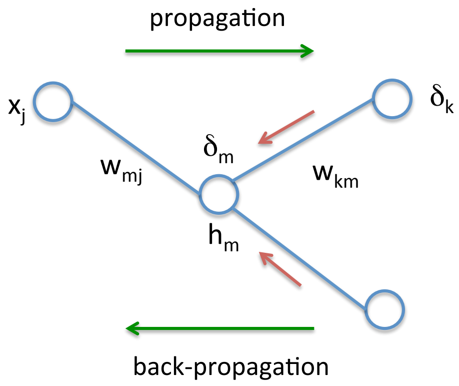
$$\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial w_{mj}} = \frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_m}\frac{\partial z_m}{\partial w_{mj}} = \delta_m x_j$$

with

$$\delta_m = \frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_m} = \sum_{k=1}^{K} \underbrace{\frac{\partial \mathcal{L}(f(x;\theta),y)}{\partial z_k}}_{\delta_k} \underbrace{\frac{\partial z_k}{\partial h_m}}_{w_{km}} \underbrace{\frac{\partial h_m}{\partial z_m}}_{g'(z_m)}$$

$$= g'(z_m)\sum_{k=1}^{K}\delta_k w_{km}$$

Back-propagation algorithm



1. Apply an input vector x_i to the network and forward propagate through the network to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units.
3. Backpropagate the δ_k's to obtain δ_m for each hidden unit in the network.
4. Evaluate the required derivatives.

Advantage of back-propagation

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

Overview

Batch learning with gradient descent

- The simplest approach to using gradient information is to choose the weight update to comprise 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 defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate the gradient. This is called called batch learning.

Learning rate tuning and optimization algorithms

- The learning rate η_t for batch learning was originally taken to be a constant; it can also be optimized by a line search that minimizes the error function at each update.
- 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) can be very large.
- Quasi-Newton methods are based on approximations of the Hessian. For instance, a diagonal approximation can be computed in $O(N_W)$ 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 gradient after each training case, and cycling through the training cases 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 training example presented at iteration t.
- 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

- 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

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

Learning rate

- In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration t as η_t.
- A sufficient condition to guarantee convergence of SGD is that

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

- In practice, it is common to decay the learning rate linearly until iteration τ:

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

  with $\gamma = t/\tau$.
- Usually τ may be set to the number of iterations required to make a few hundred passes through the training set.

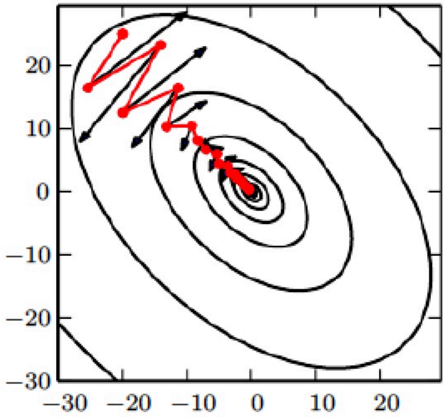
Momentum

- While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow.
- The method of momentum is designed to accelerate learning, especially in regions of high curvature (where the direction of the gradient changes a lot between two weight updates).
- The method of momentum accumulates an exponentially decaying moving average of past gradients and continues to move in their direction:

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

$$\theta \leftarrow \theta + \Delta\theta$$

Momentum



The red path is the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point.

Setting α

- In the method of momentum, the size of the step depends on how large and how aligned a sequence of gradients are.
- If the gradient $\boldsymbol{g}$ is constant, the norm of the weight update follows an arithmetico-geometric sequence

$$\|\Delta\theta_t\| = \alpha\|\Delta\theta_{t-1}\| + \eta\|\boldsymbol{g}\|$$

  Its limit is

$$\lim_{t\to\infty} \|\Delta\theta_t\| = \frac{\eta\|\boldsymbol{g}\|}{1-\alpha}$$

- It is thus helpful to think of the momentum hyperparameter in terms of $1/(1-\alpha)$. For example, $\alpha = .9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.
- Common values of α used in practice include .5, .9, and .99.

Overview

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, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
- The initial parameters need to “break symmetry” between different units: If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.
- 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:

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

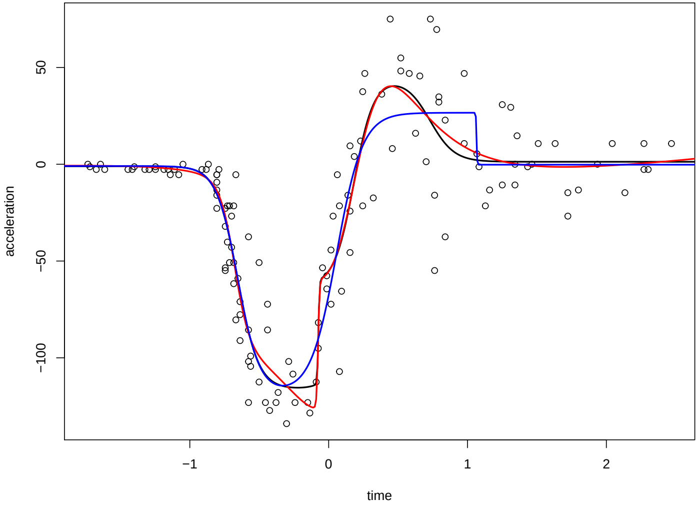
- It is best to standardize all inputs to have mean zero and standard deviation one, or to belong to $[0, 1]$.

Overview

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,)
pred2<- predict(nn2,newdata=test.data)
```

Results

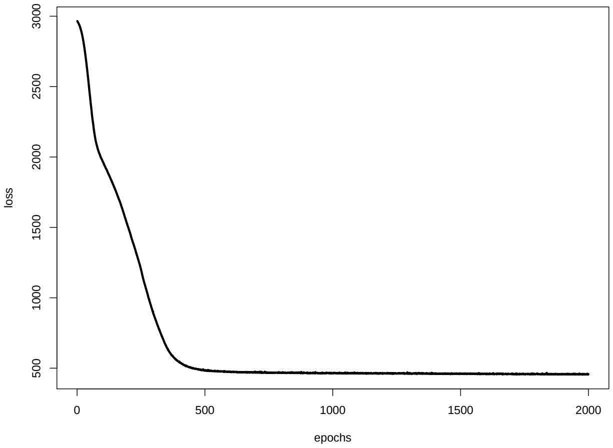


Deep NN training using the keras package

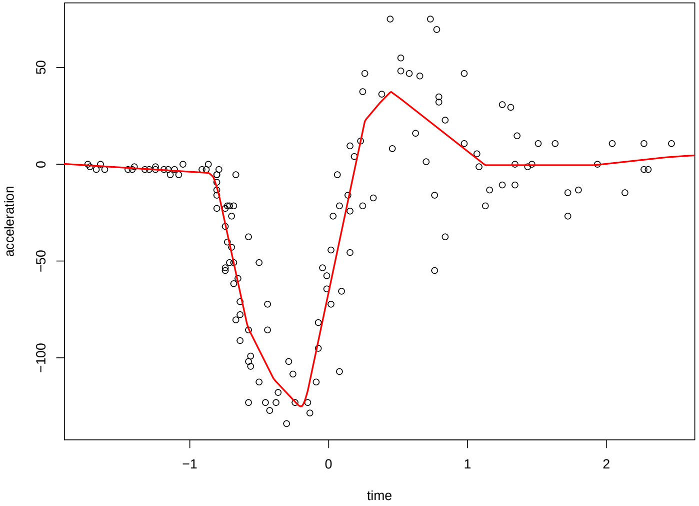
```
library(keras)

model <- keras_model_sequential()

model %>% layer_dense(units = 30, activation = 'relu', input_shape = 1) %>%
layer_dense(units = 20, activation = 'relu',name="cache1") %>%
layer_dense(units = 5, activation = 'relu',name="cache2") %>%
layer_dense(units = 1, activation = 'linear',name="sortie")


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")
```



Result



Overview

Necessity of complexity control

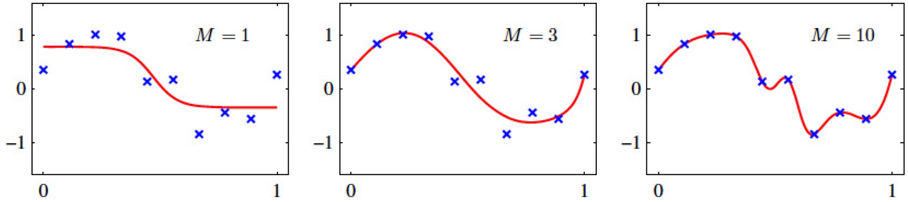
- Because of the universal approximation property of neural network, the training error can, in principle, be made arbitrarily small by increasing the number of hidden units.
- However, a large neural network will be prone to overfitting and will typically have bad generalization performance.
- We need 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

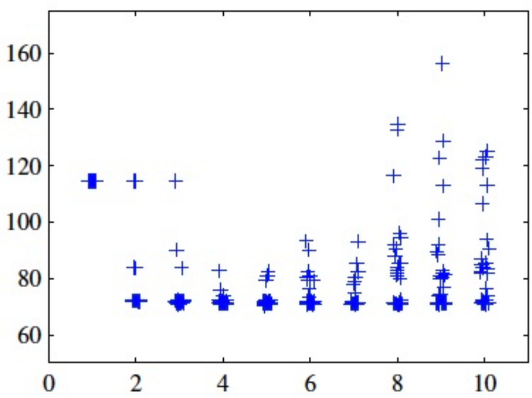
Optimizing the number of hidden units

- 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 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-set 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

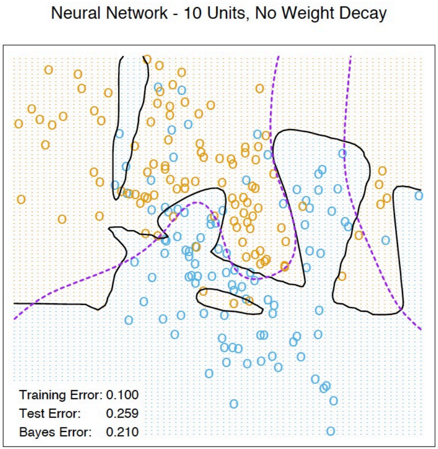
Weight decay

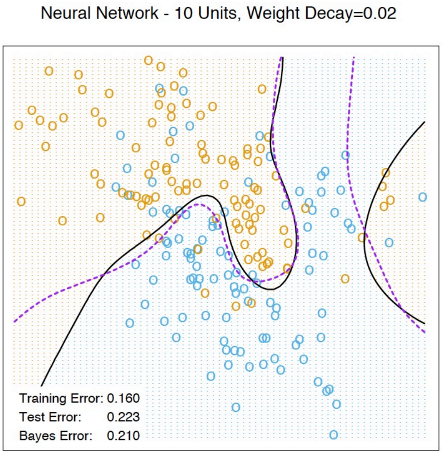
- An alternative approach to control the complexity of a neural network is to choose a relatively large value for M and to add a norm penalty term (regularizer) to the error function.
- The simplest regularizer is the quadratic (L_2), giving a regularized error

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

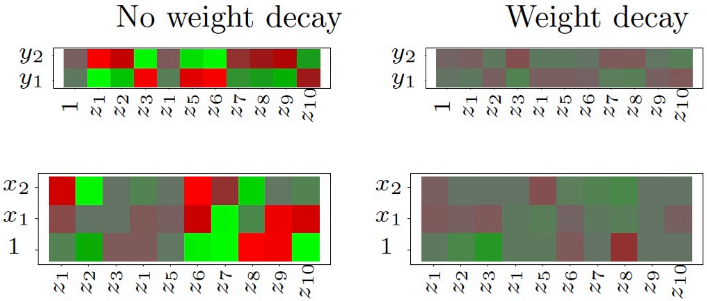
- This regularizer is also as weight decay. It is similar to ridge regression. The regularization coefficient λ is usually determined by cross-validation.
- This regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector θ.
- Other regularizer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

Example 1: classification





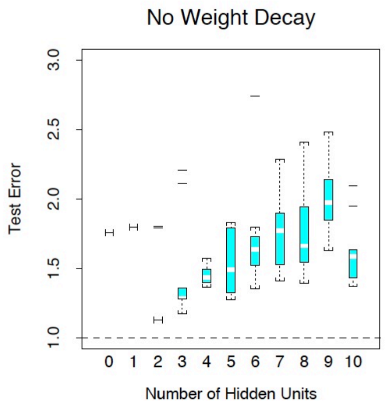
Heat maps of estimated weights

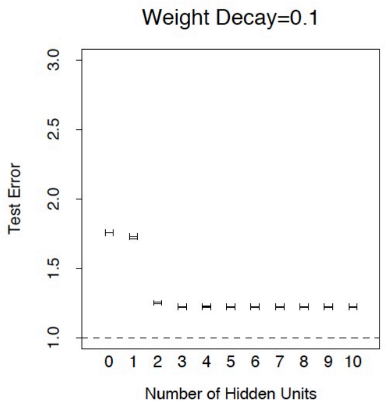


Example 2: regression

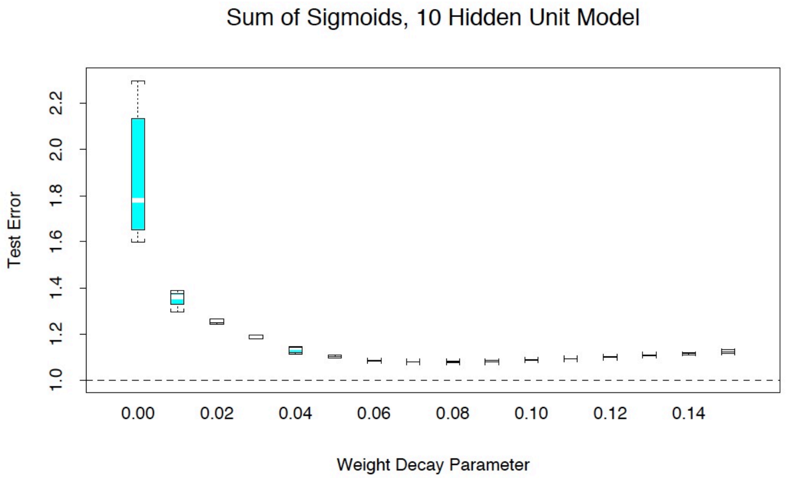
- Model $Y = f(X) + \varepsilon$ with
$$f(X) = \sigma(a_1^T X) + \sigma(a_2^T X),$$
$X = (X_1, X_2)$, $a_1 = (3, 3)$, $a_2 = (3, -3)$, $\text{Var}(f(X))/\text{Var}(\varepsilon) = 4$.
- Training sample of size 100, a test sample of size 10,000.
- Neural networks with weight decay and various numbers of hidden units.
- Average test error for each of 10 random starting weights.

Results without and with weight decay





Influence of the weight decay hyper-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

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

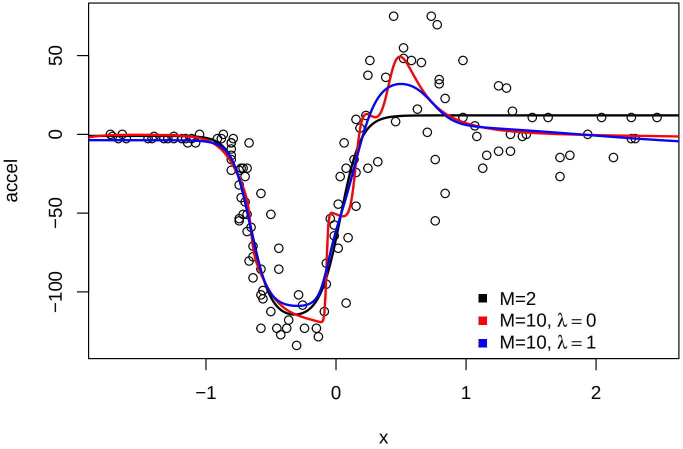
- This corresponds to the Gaussian prior

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

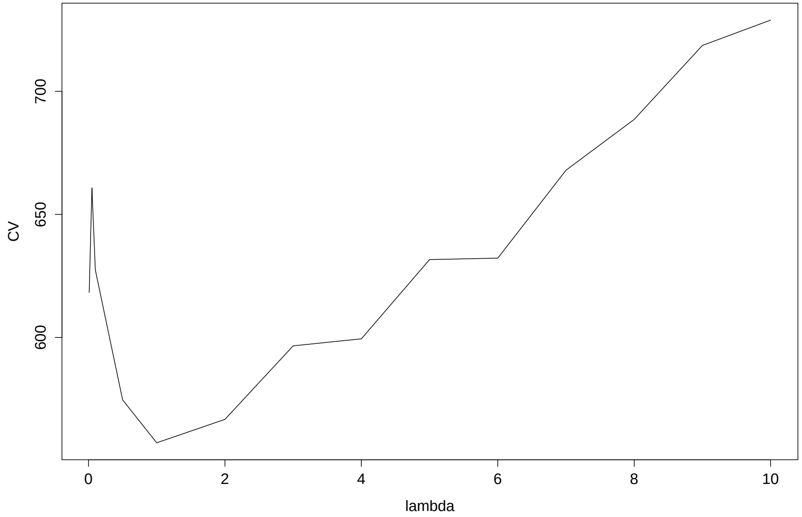
Shallow NN training with weight decay using nnet

```
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



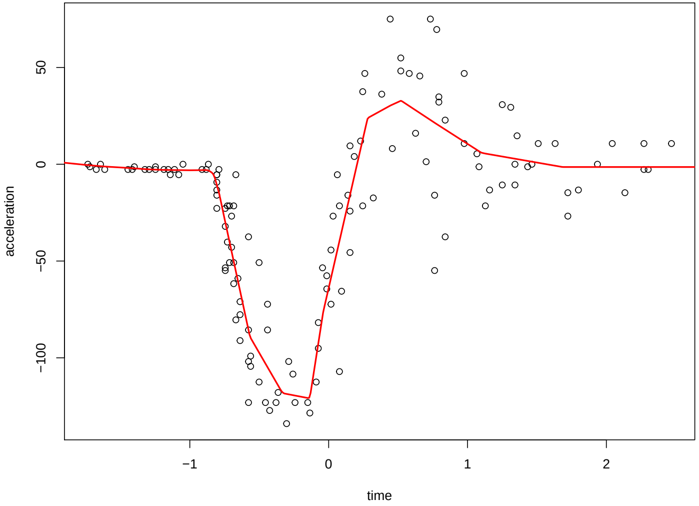
Selection of λ by 10-fold cross-validation



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',name="cache1",
kernel_regularizer = regularizer_l2(l=0.1)) %>%
layer_dense(units = 20, activation = 'relu',name="cache2",
kernel_regularizer = regularizer_l2(l=0.1)) %>%
layer_dense(units = 1, activation = 'linear',name="sortie")

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)
```

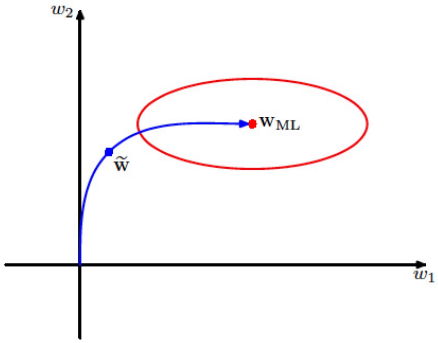
Results



Overview

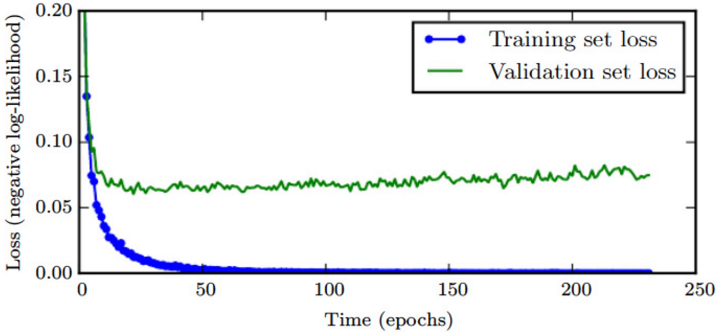
Early stopping

- 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 the global 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 useful 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). Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

Overview

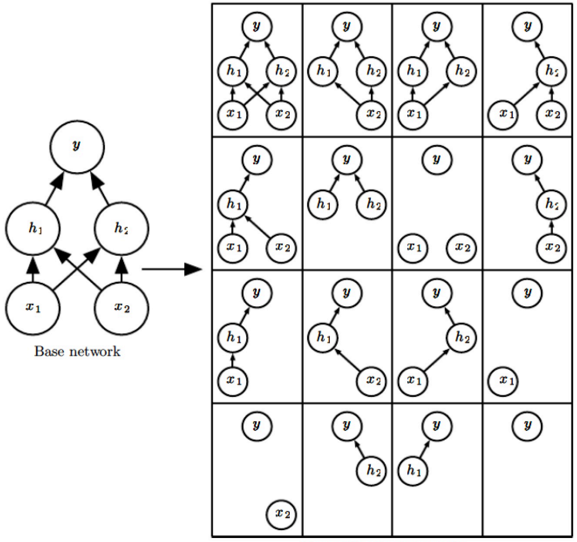
Dropout as a substitute of bagging

- 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

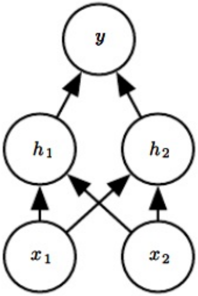


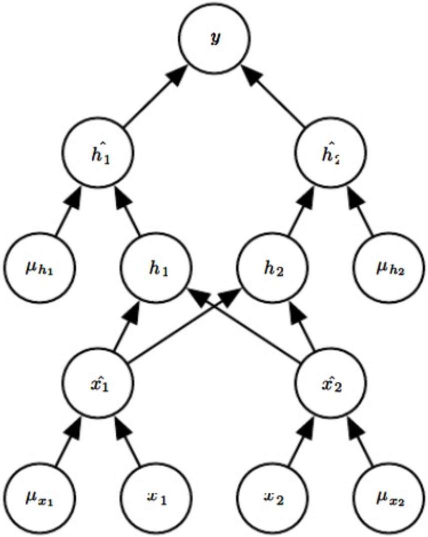
Ensemble of subnetworks

Formal analysis

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

Illustration of dropout





Difference with bagging

Dropout training is not quite the same as bagging:

Bagging	Dropout
Models are all independent	Models share parameters, with 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, and the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters

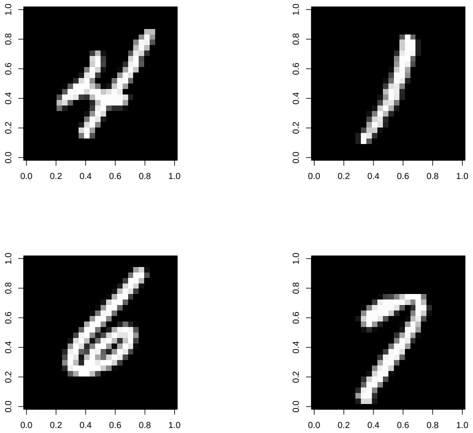
Beyond these differences, dropout follows the bagging algorithm. For example, the training set encountered by each sub-network is indeed a subset of the original training set sampled with replacement.

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 unit i multiplied by the probability of including unit i. The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach 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.



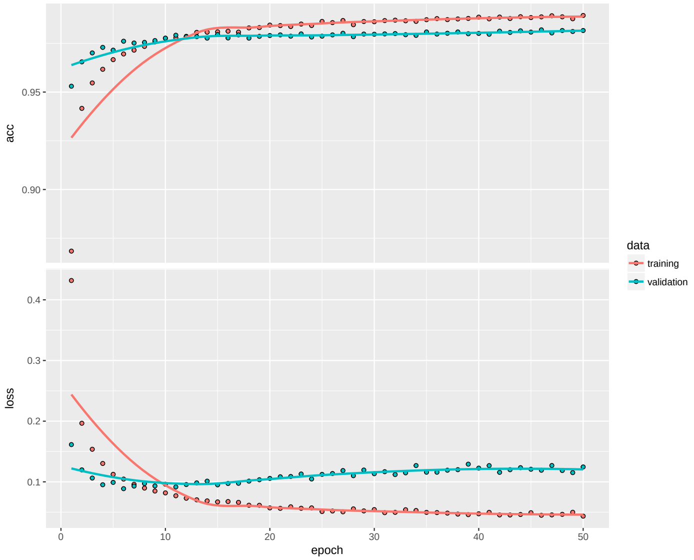
Input/output formatting

```
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

Parameter sharing

- While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: to force sets of parameters to be equal.
- This method of regularization is often referred to as parameter/weight sharing, because we interpret the various models or model components as sharing a unique set of parameters.
- A significant advantage of parameter sharing over regularizing the parameters via a norm penalty is that only a subset of the parameters (the unique set) need to be stored in memory.
- In certain models – such as convolutional neural networks – this can lead to significant reduction in the memory requirement of the model.

Overview

Convolutional neural networks

- Convolutional networks, also known as convolutional neural networks or CNNs, are designed to process data that have a known, grid-like topology.
- Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, image data, which can be thought of as a 2D grid of pixels, and videos, which are 3D data (2D grid of pixels + time).
- Convolutional networks have been tremendously successful in practical applications.
- Convolutional networks are simply neural networks that use convolution (a specialized kind of linear operation) in place of general matrix multiplication in at least one of their layers.

Overview

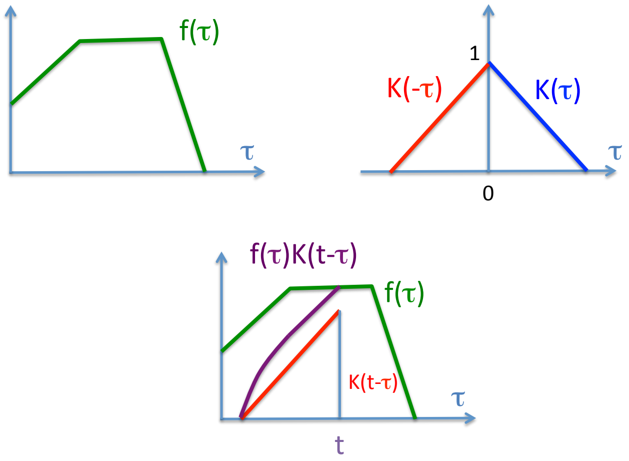
Convolution (1D)

- Let $f(t)$ and $K(t)$ be two functions from $\mathbb{R}$ to $\mathbb{R}$.
- Their convolution is the function $s(t) = (f * K)(t)$ defined as

$$s(t) = (f * K)(t) = \int_{-\infty}^{+\infty} f(\tau)K(t-\tau)d\tau.$$

- The first argument (in this example, the function f) to the convolution is often referred to as the input and the second argument (in this example, the function K) as the kernel.
- The convolution of f with kernel K can be described as a weighted average of the function $f(\tau)$ where the weighting is given by $K(-\tau)$ simply shifted by amount t.

Illustration of 1D convolution



1D convolution (continued)

- Commutativity: $f * K = K * f$. Proof: let $u = t - \tau$. We have

$$\begin{aligned}(f * K)(t) &= \int_{-\infty}^{+\infty} f(\tau)K(t-\tau)d\tau \\ &= \int_{+\infty}^{-\infty} f(t-u)K(u)(-du) \\ &= \int_{-\infty}^{+\infty} K(u)f(t-u)du = (K * f)(t)\end{aligned}$$

- Discrete version: for functions f and K from $\mathbb{Z}$ to $\mathbb{R}$,

$$(f * K)(i) = \sum_{n=-\infty}^{+\infty} f(n)K(i-n)$$

Convolution (2D)

- In machine learning applications, the input is usually a multidimensional array (tensor) of data and the kernel is usually a tensor of parameters that are adapted by the learning algorithm. These functions are zero everywhere but the finite set of points for which we store the values.
- If we use a two-dimensional image I as our input, we use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_{m,n} I(m,n)K(i-m,j-n),$$

- Thanks to commutativity, we can write equivalently

$$S(i,j) = (K * I)(i,j) = \sum_{m,n} I(i-m,j-n)K(m,n),$$

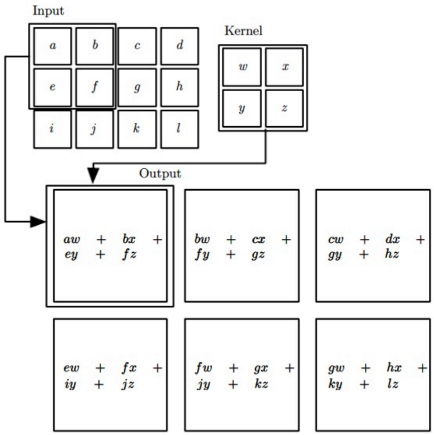
Cross-correlation

- The commutative property of convolution arises because we have flipped the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property.
- Many neural network libraries actually implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

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

- In practice, whether we flip the kernel or not is immaterial.

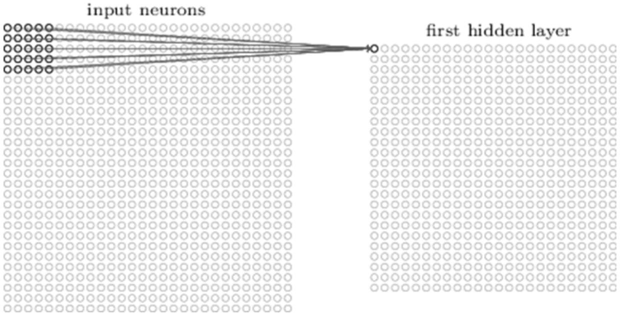
Computation of 2-D convolution



An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts.

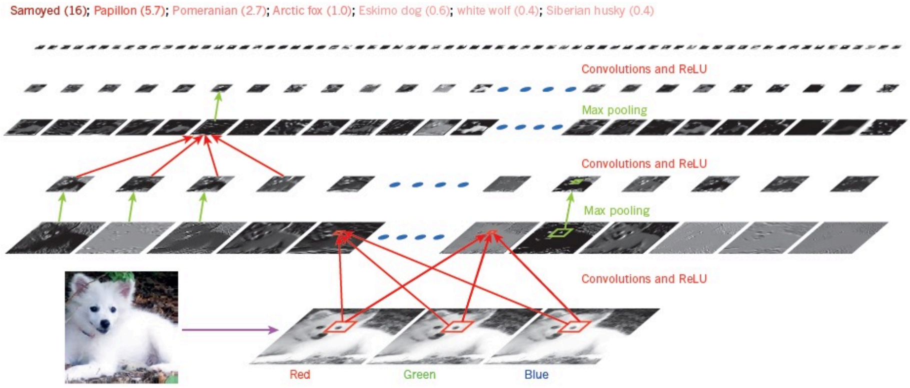
Implementation

- In the convolutional layer the units are organized into planes, each of which is called a feature map.
- Units in a feature map each take inputs only from a small subregion of the image called a receptive field, and all of the units in a feature map are constrained to share the same weight values.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

A more realistic convolutional network


Samoyed (16); Papillon (5.7); Pomeranian (2.7); Arctic fox (1.0); Eskimo dog (0.6); white wolf (0.4); Siberian husky (0.4)
Convolutions and ReLU
Max pooling
Convolutions and ReLU
Max pooling
Convolutions and ReLU
Red
Green
Blue

Each rectangular image is a feature map corresponding to the output for one of the learned features, detected at each of the image positions. Information flows bottom up, with lower-level features acting as oriented edge detectors, and a score is computed for each image class in output.

Motivation

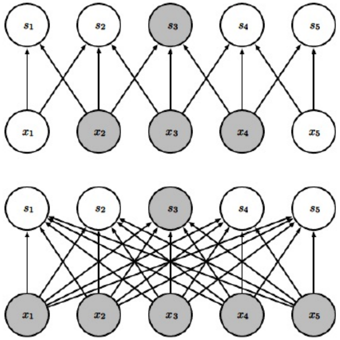
Convolution leverages three important ideas that can help improve a machine learning system:

1. Sparse connectivity
2. Parameter sharing and
3. Equivariance to translation

Sparse connectivity

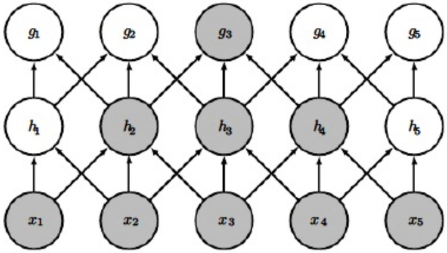
- When processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels.
- As a consequence, computing the output requires fewer operations. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime.
- For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m.

Sparse connectivity (continued)



We highlight one output unit, s_3, and the input units in x that affect this unit. These units are known as the receptive field of s_3. (Top) When s is formed by convolution with a kernel of width 3, only three inputs affect s_3. (Bottom) When s is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3.

Sparse connectivity (continued)

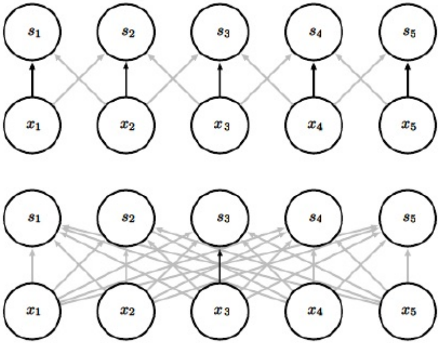


In a deep convolutional network, units in the deeper layers may indirectly interact with a larger portion of the input. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

Parameter sharing

- Parameter sharing refers to using the same parameter for more than one function in a model.
- In a feature map, all units share the same weight vector (corresponding to the kernel of the convolution). Instead of learning a separate set of parameters for every unit in the map, we learn only one set.
- This does not affect the runtime of forward propagation – it is still $O(k \times n)$ – but it does further reduce the storage requirements of the model to k parameters.
- Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

Parameter sharing



Equivariance to translation

- To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$.
- In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g.
- For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I', then applied the transformation g to the output.

Equivariance to translation (continued)

- When applied to images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output.
- This is useful when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations.
- For example, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

Edge detection example

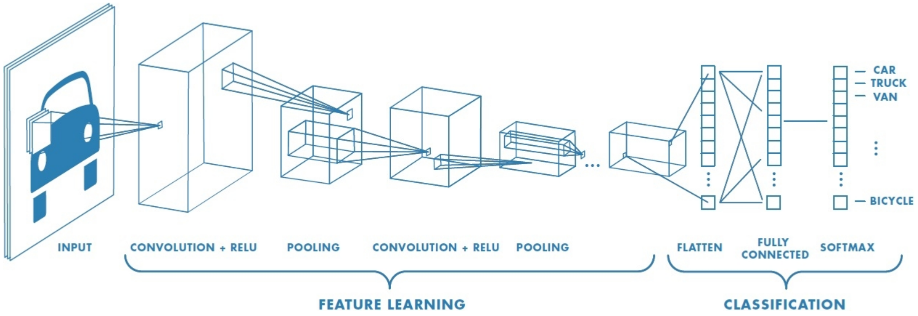
$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 10 & 10 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -30 & 0 & 30 & 30 \\ -30 & 0 & 30 & 30 \\ -30 & 0 & 30 & 30 \\ -30 & 0 & 30 & 30 \end{bmatrix}$$

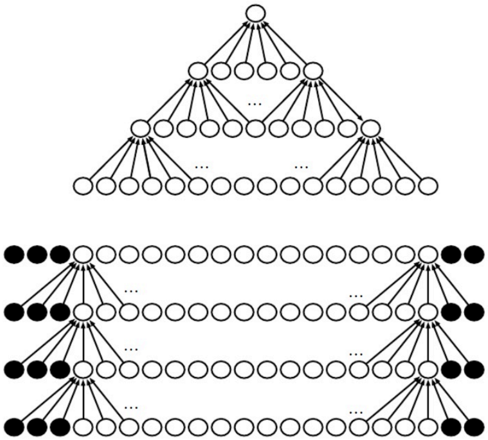
3D convolution

- In many applications, the input is not just a grid of real values. Rather, it is a grid of vector-valued observations.
- For example, a color image has a red, green and blue intensity at each pixel.
- In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position.
- When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel.

Example


INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
CAR
TRUCK
VAN
BICYCLE
FEATURE LEARNING
CLASSIFICATION

Zero-padding



Without zero-padding, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently.

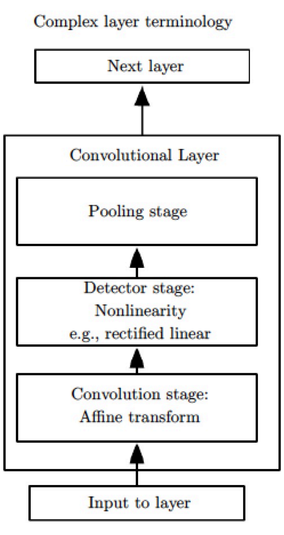
Overview

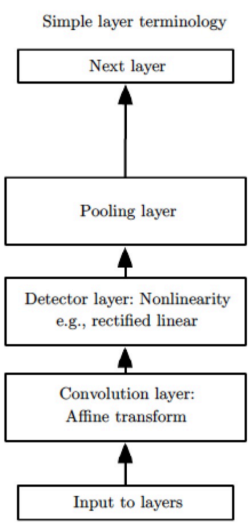
Components of a CNN layer

A typical layer of a convolutional network consists of three stages:

1. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations.
2. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage.
3. In the third stage, we use a pooling function to modify the output of the layer further.

Components of a CNN layer

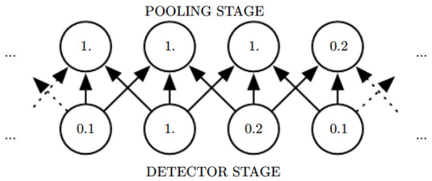

Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer

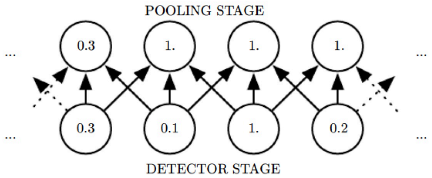

Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity e.g., rectified linear
Convolution layer: Affine transform
Input to layers

Pooling

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
- For example, the max pooling operation reports the maximum output within a rectangular neighborhood.
- Other popular pooling functions include
  - The average of a rectangular neighborhood
  - The L_2 norm of a rectangular neighborhood, or
  - Weighted average based on the distance from the central pixel.

Max pooling introduces invariance to translation



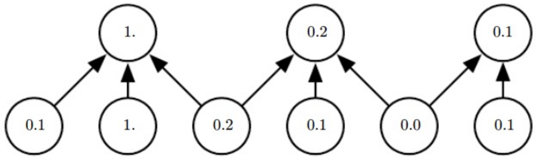


In the bottom figure, the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

Pooling with down-sampling

- Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart.
- This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process.

Pooling with down-sampling (continued)

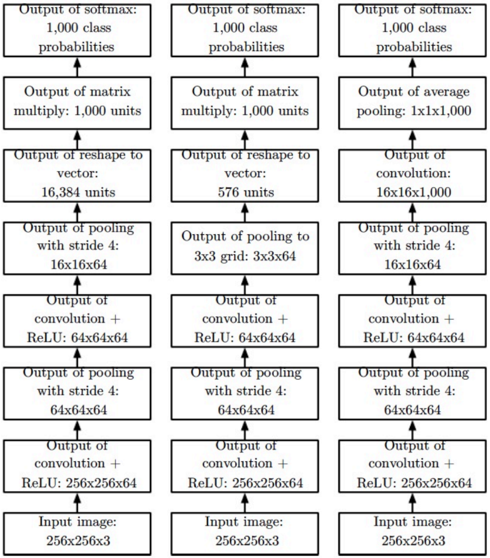


Here we use max-pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.

Handling inputs of varying sizes

- For many tasks, pooling is essential for handling inputs of varying size.
- For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size.
- For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

Examples of architectures

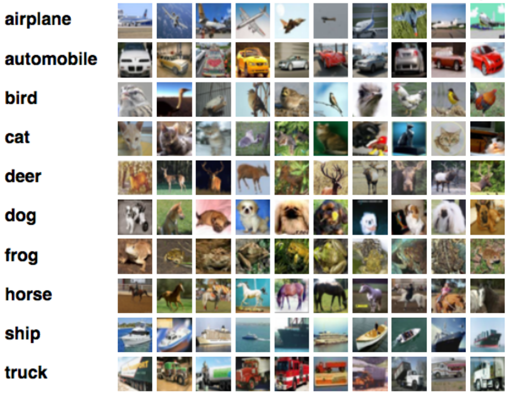

Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 16,384 units
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 576 units
Output of pooling to 3x3 grid: 3x3x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of average pooling: 1x1x1,000
Output of convolution: 16x16x1,000
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3

Explanation of the previous figure

1. (Left) A CNN that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier.
2. (Center) A CNN that processes a variable-sized image, but still maintains a fully connected section. This network uses a pooling operation with variably-sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network.
3. (Right) A CNN that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

Example: CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.



Data preparation

```
library('keras')

cifar10 <- dataset_cifar10()

# Image display
library('imager')
i<-sample(50000,1)
plot(permute_axes(as.cimg(cifar10$train$x[i,,,]),"yxzc"),axes=FALSE)

# Feature scale RGB values in test and train inputs
x_train <- cifar10$train$x/255
x_test <- cifar10$test$x/255
y_train <- to_categorical(cifar10$train$y, num_classes = 10)
y_test <- to_categorical(cifar10$test$y, num_classes = 10)
```

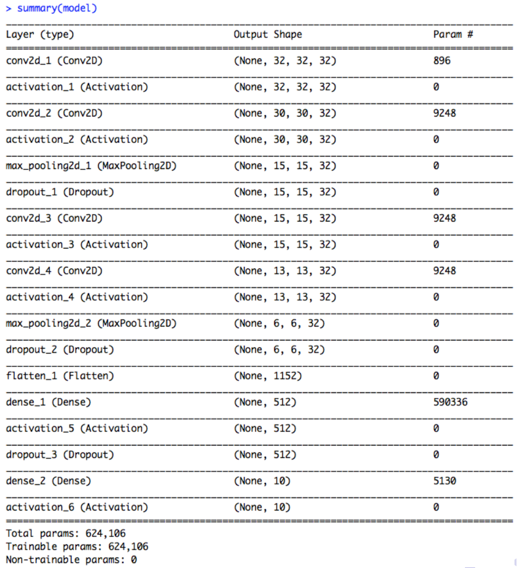
Model definition

```
model <- keras_model_sequential()
model %>%

# Start with hidden 2D convolutional layer being fed 32x32 pixel images
layer_conv_2d(
filter = 32, kernel_size = c(3,3), padding = "same",
input_shape = c(32, 32, 3)
) %>%
layer_activation("relu") %>%

# Second hidden layer
layer_conv_2d(filter = 32, kernel_size = c(3,3)) %>%
layer_activation("relu") %>%

# Use max pooling
layer_max_pooling_2d(pool_size = c(2,2)) %>%
layer_dropout(0.25) %>%
```

Model definition (continued)

```
# 2 additional hidden 2D convolutional layers
layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same") %>%
layer_activation("relu") %>%
layer_conv_2d(filter = 32, kernel_size = c(3,3)) %>%
layer_activation("relu") %>%

# Use max pooling once more
layer_max_pooling_2d(pool_size = c(2,2)) %>%
layer_dropout(0.25) %>%

# Flatten max filtered output into feature vector and feed into dense layer
layer_flatten() %>%
layer_dense(512) %>%
layer_activation("relu") %>%
layer_dropout(0.5) %>%

# Outputs from dense layer are projected onto 10 unit output layer
layer_dense(10) %>%
layer_activation("softmax")
```

Model summary



```
> summary(model)
________________________________________________________________________________
Layer (type)                        Output Shape                     Param #
================================================================================
conv2d_1 (Conv2D)                   (None, 32, 32, 32)               896
________________________________________________________________________________
activation_1 (Activation)           (None, 32, 32, 32)               0
________________________________________________________________________________
conv2d_2 (Conv2D)                   (None, 30, 30, 32)               9248
________________________________________________________________________________
activation_2 (Activation)           (None, 30, 30, 32)               0
________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)      (None, 15, 15, 32)               0
________________________________________________________________________________
dropout_1 (Dropout)                 (None, 15, 15, 32)               0
________________________________________________________________________________
conv2d_3 (Conv2D)                   (None, 15, 15, 32)               9248
________________________________________________________________________________
activation_3 (Activation)           (None, 15, 15, 32)               0
________________________________________________________________________________
conv2d_4 (Conv2D)                   (None, 13, 13, 32)               9248
________________________________________________________________________________
activation_4 (Activation)           (None, 13, 13, 32)               0
________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)      (None, 6, 6, 32)                 0
________________________________________________________________________________
dropout_2 (Dropout)                 (None, 6, 6, 32)                 0
________________________________________________________________________________
flatten_1 (Flatten)                 (None, 1152)                     0
________________________________________________________________________________
dense_1 (Dense)                     (None, 512)                      590336
________________________________________________________________________________
activation_5 (Activation)           (None, 512)                      0
________________________________________________________________________________
dropout_3 (Dropout)                 (None, 512)                      0
________________________________________________________________________________
dense_2 (Dense)                     (None, 10)                       5130
________________________________________________________________________________
activation_6 (Activation)           (None, 10)                       0
================================================================================
Total params: 624,106
Trainable params: 624,106
Non-trainable params: 0
```

Compilation and training

```
opt <- optimizer_rmsprop(lr = 0.0001, decay = 1e-6)
model %>% compile(
loss = "categorical_crossentropy",
optimizer = opt,
metrics = "accuracy"
)

# Training --------------------------------

model %>% fit(
x_train, y_train,
batch_size = 32,
epochs = 200,
validation_data = list(x_test, y_test),
shuffle = TRUE
)
```

Final remarks

- Since the early 2000's, CNNs have been applied with great success to the detection, segmentation and recognition of objects and regions in images. These were all tasks in which labelled data was relatively abundant, such as traffic sign recognition, the detection of faces, text, pedestrians and human bodies in natural images. A major recent practical success of CNNS is face recognition.
- Importantly, images can be labelled at the pixel level, which has applications in technology, including autonomous mobile robots and self-driving cars. Other applications gaining importance involve natural language understanding and speech recognition.
- Recent CNN architectures have 10 to 20 layers of ReLUs, hundreds of millions of weights, and billions of connections between units. Whereas training such large networks could have taken weeks only two years ago, progress in hardware, software and algorithm parallelization have reduced training times to a few hours.

Final remarks (continued)

- The performance of CNN-based vision systems has caused most major technology companies, including Google, Facebook, Microsoft, IBM, Yahoo!, Twitter and Adobe, as well as a quickly growing number of start-ups to initiate research and development projects and to deploy CNN-based image understanding products and services.
- CNNs are easily amenable to efficient hardware implementations in chips. A number of companies such as NVIDIA, Mobileye, Intel, Qualcomm and Samsung are developing CNN chips to enable real-time vision applications in smartphones, cameras, robots and self-driving cars.