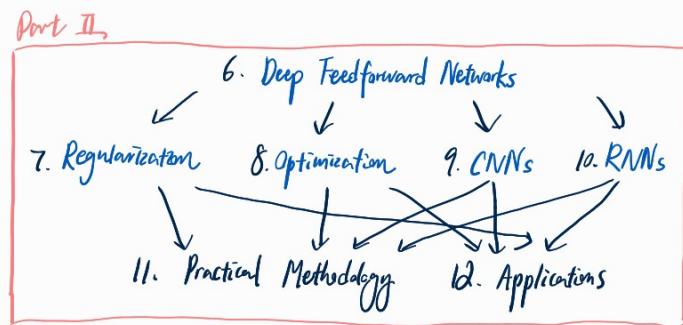


Part II: established DL algorithm



## Chapter 6 Deep Feedforward Networks

Deep Feedforward Network

Feedforward Neural Network

Multilayer Perceptrons (MLP)

Feed forward  
Network

Information flow through

- ① the function evaluated from  $\vec{x}$
- ② the intermediate computations used to define  $f$
- ③ finally to the output  $\vec{y}$

No feedback connections from output into itself  
otherwise Recurrent Neural Network,  
composing many functions with a DAG

Depth of the model  $\longrightarrow$  Number of layers (length of the chain of functions)

Width of the model  $\longrightarrow$  Dimensionality of the hidden layers (typically vector-valued)

$\hookrightarrow$  Each layer: many units (vector-to-scalar function) working in parallel

Motivation:

Linear models =  
e.g. linear & logistic regression

- ✓ fit efficiently & reliably  
in closed form or with convex optimization
- ✗ model capacity limited to linear functions  
apply linear model on transformed input  $\phi(\vec{x})$  where  $\phi$  nonlinear

Choosing  $\phi$ :

① Generic  $\phi$  =

- e.g. infinite-dimensional  $\phi$  like RBF kernel
- ✓ high-dim  $\phi \rightarrow$  enough capacity to fit  
✗ generalization to test set is poor
- ✗ very generic  $\phi \rightarrow$  usually based only on the principle of local smoothness  
✗ not encoding enough prior

② Manually engineer  $\phi$  =  
✗ huge human efforts with specialized practitioners  
✗ little transfer between domains

③ Deep Learning = learn the  $\phi$  in the Kernel trick in Linear Models

$$\vec{y} = f(\vec{x}; \theta; \vec{w}) = \phi(\vec{x}; \theta)^T \vec{w}$$

use to learn  $\phi$   
map from  $\phi(\vec{x})$  to the output  
from a broad class of functions

the only approach to give up the convexity of the training problem  
but benefits outweigh the harms

- ✓ can be highly generic by using a very broad family  $f(\vec{x}; \theta)$
  - ✓ human practitioners can encode their knowledge to help generalization by designing families  $f(\vec{x}; \theta)$  that they expect to perform well
- \* only need to find the function family,  
not the precise right function

## 6.1 Example: Learning XOR

Target function:  $y = f^*(\vec{x})$

Model function:  $y = f(\vec{x}; \theta)$

Learning Algorithm: adopt the parameters  $\theta$  to make  $f$  as similar as  $f^*$

① Loss function  $\xrightarrow{\text{choose}}$  MSE loss  $X = \{(0,0), (0,1), (1,0), (1,1)\}$

$$J(\theta) = \frac{1}{4} \sum_{\vec{x} \in X} (f^*(\vec{x}) - f(\vec{x}; \theta))^2$$

② Form of the model  $f(\vec{x}; \theta) \xrightarrow{\text{choose}}$  Linear model,  $\vec{\theta} = \{\vec{w}, b\}$

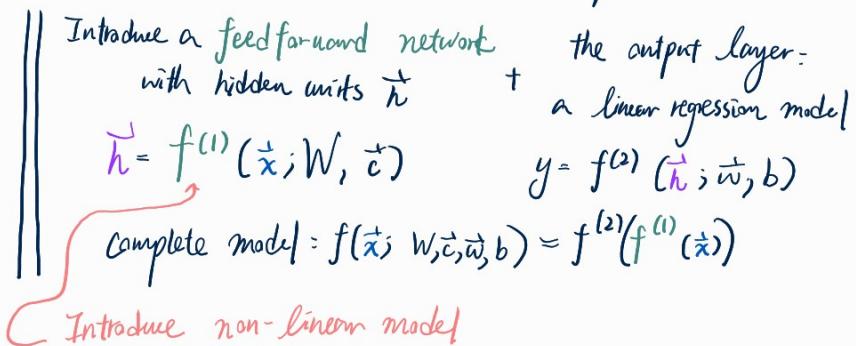
$$f(\vec{x}; \vec{w}; b) = \vec{x}^T \vec{w} + b$$

③ Minimize  $J(\theta) \rightarrow$  can be done in closed form w.r.t.  $\vec{w}$  and  $b$

$$\vec{w} = \vec{0}, b = \frac{1}{2}$$

$\rightarrow$  simply outputs 0.5 everywhere  $\times$

To solve: use a model that learns a different feature space  
in which a linear model is able to represent the solution



Most neural network: an affine transformation (controlled by learned parameter)

scalar to scalar

a fixed (non-linear) activation function

$$\vec{h} = g(W^T \vec{x} + \vec{c})$$

typically an element-wise activation function

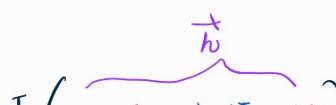
$$\vec{h}_i = g(\vec{x}^T W_{:,i} + c_i)$$

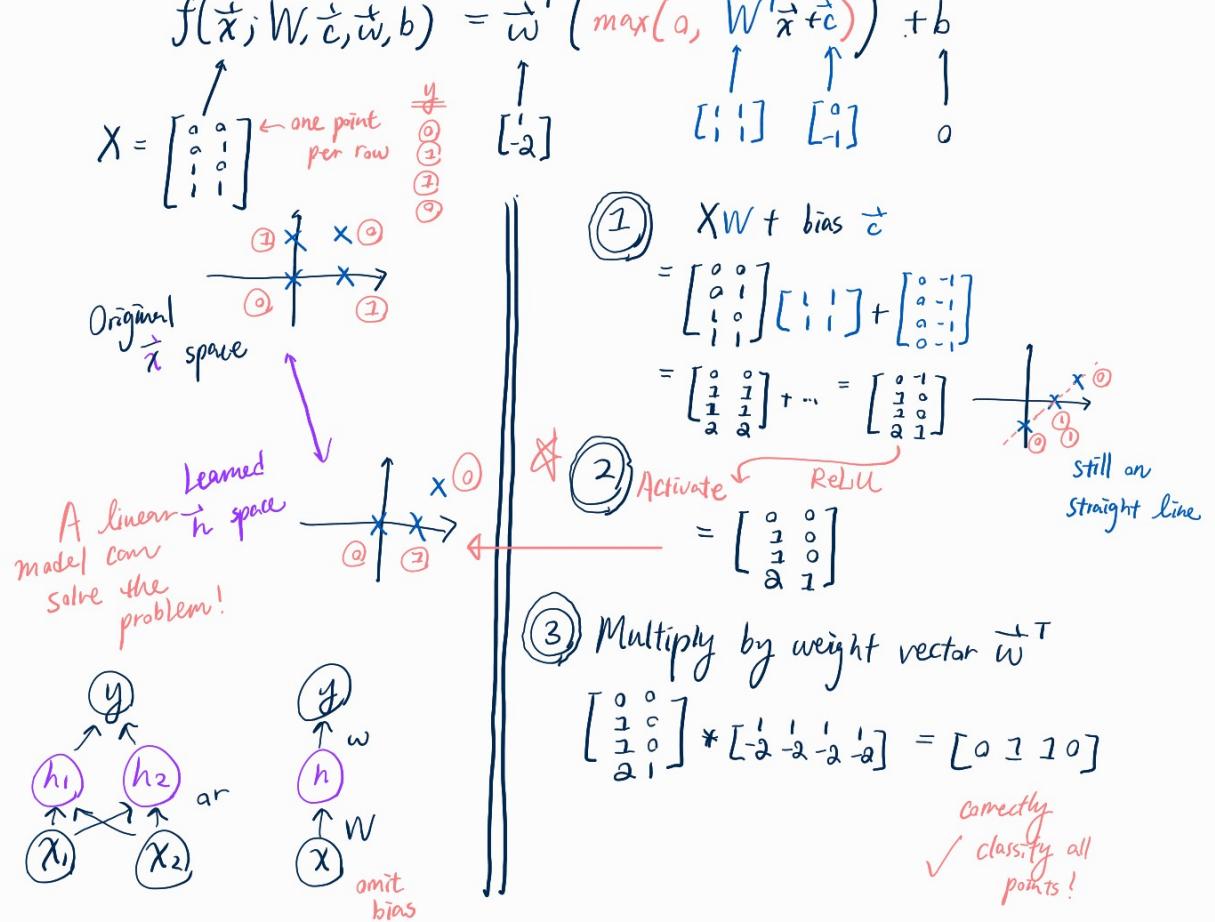
typical Rectified Linear Unit

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



Complete Network:





## 6.2 Gradient-Based Learning

Neural Networks vs Linear model:

Non-linearity of a neural network

causes most interesting loss functions to become nonconvex.

→ Usually aim to drive the cost function to a very low value, converges starting from any initial parameters rather than convex optimization algorithms with global convergence guarantees or linear equation solvers (e.g. logistic regression or SVMs)

→ Stochastic gradient descent on

nonconvex loss functions → no such convergence guarantees

→ sensitive to initial parameters

! Important: Feedforward Network → init weights to small random values

bias to 0 / small positive numbers

### 6.2.1 Cost Functions

Most cases: parametric model defines a distribution

$p(\vec{y} | \vec{x}; \theta)$  and use the principle of maximum likelihood

→ Cross Entropy between the training data

and the model's predictions as the cost function

Simpler approach =

rather than predicting a complete probability distribution over  $\vec{y}$ , merely predict some statistics of  $\vec{y}$  conditioned on  $\vec{x}$

(specialized loss function to train a predictor of these estimates)

Total cost function = primary cost function + regularization term

### 6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Cost function  $\rightarrow$  negative likelihood / cross entropy

$$J(\theta) = - \mathbb{E}_{\vec{x}, \vec{y} \sim p_{\text{data}}} \log p_{\text{model}}(\vec{y} | \vec{x})$$

true  
far any model  
(not only linear model)  
when predict Gaussian  
mean of Gaussian  
depends on the model parameters and can depend on the model parameters and can discard some terms that do not depend only on variance and do not need to parametrize

e.g. when  $p_{\text{model}}(\vec{y} | \vec{x}) = \mathcal{N}(\vec{y}; f(\vec{x}; \theta), I)$ ,  
then recover  $J(\theta) = \text{Mean Squared Error cost} + \text{const}$

i.e. specifying a model  $p(\vec{y} | \vec{x})$  = determines the cost function

✓ Power of deriving cost function from maximum likelihood

\* Unusual property of Cross-Entropy:

No minimum value  $\leftarrow$  Regularization techniques modify the learning problem s.t. the model cannot reap unlimited reward

### 6.2.1.2 Learning Conditional Statistics

Instead of learning a full probability distribution  $p(\vec{y} | \vec{x}; \theta)$ , want to just learn one conditional statistic of  $\vec{y}$  given  $\vec{x}$

e.g. prefer  $f(\vec{y}; \theta)$  to predict the mean of  $\vec{y}$

View the cost function as a function rather than just a function  
 $\Rightarrow$  learning to choose a function, instead of choosing a set of parameters  
 $\Rightarrow$  e.g. can design the cost function to have its minimum occur at some specific desirable function, e.g. a function that maps  $\vec{x}$  to  $\mathbb{E}(\vec{y} | \vec{x})$

Requires a mathematical tool: Calculus of Variations

Gives two results

$$\textcircled{1} \quad f^* = \underset{f}{\operatorname{argmin}} \mathbb{E}_{\vec{x}, \vec{y} \sim p_{\text{data}}} \|\vec{y} - f(\vec{x})\|^2 = \mathbb{E}_{\vec{y} \sim p_{\text{data}}(\vec{y} | \vec{x})} [\vec{y}]$$

If we train on infinitely many samples from the true data generating distribution  
minimizing the MSE cost function  $\rightarrow$  gives a function that predicts the mean of  $\vec{y}$  for each value of  $\vec{x}$

$$\textcircled{2} \quad f^* = \underset{f}{\operatorname{argmin}} \mathbb{E}_{\vec{x}, \vec{y} \sim p_{\text{data}}} \|\vec{y} - f(\vec{x})\|_1 = \text{gives a function that predicts the median of } \vec{y} \text{ for each value of } \vec{x}$$

Mean Absolute Error cost function

X Both poor result when using gradient-based optimization,

(Output units that saturates produce very small gradient when combining these cost functions)

⇒ Cross-entropy is more popular even when not necessary to estimate  $p(\vec{y}|\vec{x})$

## 6.2.2 Output Units

Choice of cost function → coupled with choice of output unit

the form of the cross-entropy function determines how to represent output

### 6.2.2.1 Linear Units for Gaussian Output Distributions

Linear Units: given features  $\vec{h}$ ,

a layer of linear output units produces  $\vec{y} = W^T \vec{h} + b$

Often used =

produce the mean of a conditional Gaussian distribution

Maximum likelihood → minimizing the MSE

✓ linear units does not saturate → good for gradient-based optimization

### 6.2.2.2 Sigmoid Unit for Bernoulli Output Distributions

$$\text{Sigmoid Output Unit: } \vec{y} = \delta(\vec{w}^T \vec{h} + b) \quad \begin{matrix} \uparrow & \underbrace{\quad\quad\quad}_{\text{activation}} \\ \text{Sigmoid} & \delta(\vec{x}) = \frac{\exp(\vec{x})}{1 + \exp(\vec{x})} \end{matrix}$$

Often used =

the maximum likelihood approach to define a Bernoulli distribution over  $y$  conditioned  $x$  on binary classification (predicting the value of a binary variable)

*the log in the cost function*  
*undoes the exp of the sigmoid*  $\rightarrow J(\vec{o}) = -\log P(\vec{y}|\vec{x})$  *Softplus*  
 $= \vec{y} \ln(1 - \vec{y}) + (1 - \vec{y}) \ln \vec{y}$  where  $z$  is the logit

⇒ Saturates only when  $(1-\vec{y})z$  is very negative

i. Sigmoid  $\delta$  exclusively used in Maximum Likelihood with a log Loss function, otherwise other losses easily saturates when  $\delta$  saturates

i.e.  $y=1$  and  $z$  very positive  
 $y=0$  and  $z$  very negative

In the limit of extremely correct  $z$ , the softplus does not shrink the gradient  
✓ Gradient-based learning can act quickly

### 6.2.2.3 Softmax Units for Multinomial Output Distributions

$$\text{Softmax Function: } \text{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \leftarrow \vec{z} = W^T \vec{h} + b, z_i = \log \tilde{P}(y=i|\vec{x}) \quad \text{unnormalized}$$

A "softened" version of argmax → argmax gives one-hot vector  
softmax is continuous and differentiable

Often used =

Represent a probability distribution over an discrete variables with  $n$  possible values

More rare = used within model itself to choose between one of  $n$  diff. options - for some internal values

(Similarly) Works well when training the softmax to output  $y$  using maximum log-likelihood

The log loss "Undoes"  
softmax

$$\log \text{softmax}(\vec{z})_i = z_i - \log \sum_j \exp(z_j)$$

the exp in sum =  $\sum_i \exp(z_i)$ 

$\underbrace{\exp(z_i)}_{\text{Input } z_i}$   
 always have an direct contribution  
 to the cost function  
 $\rightarrow$  Want 1 satrate

learning can proceed  
 even if the 2nd term  
 became very small

$\Rightarrow$  Saturates when the difference between input values became extreme  
 e.g. Numerically stable variant "Winner-take-all"

$$\text{Softmax}(\vec{z}) = \text{Softmax}(\vec{z} - \max_i z_i)$$

## 6.2.2.4 Other Output Types

Generally:

If define a conditional distribution  $p(\vec{y}|\vec{x}; \theta)$ ,the principle of maximum likelihood suggests using  $-\log p(\vec{y}|\vec{x}; \theta)$  as cost function

Neural Network = a function  $f(\vec{x}; \vec{\theta})$  not direct predictions of  $\vec{y}$   
 whose outputs are  $\vec{w}$ , the parameters of a distribution over  $\vec{y}$  and  
 loss function is  $-\log p(\vec{y}/\vec{w}(\vec{x}))$

Examples:

Learning the variance of a conditional Gaussian for  $\vec{y}$ Multimodal regression  $\rightarrow$  Gaussian mixtures in Neural Network

effective in  
 generative models of  
 speech / movement of physical objects

 $\rightarrow$  Mixture density networks

## 6.3 Hidden Units

- Requires Trial-and-Error
- Can safely disregard the non-differentiability of the hidden unit activation functions during implementation, return one-side limit // numerical error so  $\epsilon \neq 0$

### 6.3.1 Rectified Linear Units and Their Generalizations

$$g(z) = \max\{0, z\}$$

✓ similar to linear units  $\rightarrow$  easy to optimize

✓ derivative remain large when it is active

✓ second order derivative is 0  $\rightarrow$  no second-order effects

**tips** = initialize  $\vec{b}$  (in  $\vec{n} = g(W^T \vec{x} + \vec{b})$ ) to small positive values e.g. 0.1  
 s.t. the ReLU unit is active initially

**X** Cannot learn via gradient-based methods on examples for which their activation is 0

generalizations of ReLU guarantees non-zeros

① when  $z_i < 0$  =

$$\vec{n} = g(\vec{z}, \vec{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

Absolute value rectification

$$\alpha_i = -1 \Rightarrow g(z) = |z|$$

eg. object recognition  
 look for features that are invariant

Leaky ReLU  $\alpha \approx$  very small e.g. 0.01 under a polarity reversal of the input illumination

Parametric ReLU or PReLU  $\alpha_z$  a learnable parameter

## ② Maxout units Generalize further

$g(\vec{z})_i = \max_{j \in G^{(i)}} z_j \Rightarrow$  divides  $\vec{z}$  into  $k$  groups of values, each unit outputs the max element of one of these groups  
learn piece-wise linear functions ("the activation function itself")

- ✓ with large enough  $k$ , can approximate any convex function with arbitrary fidelity
- parametrized by  $k$  weight vectors → require more regularization
- ✓ help fight Catastrophic forgetting (neural network forget what they learned) since it has some redundancy when each unit is driven by multiple filters

## 6.3.2 Logistic Sigmoid and Hyperbolic Tangent

Commonly used before ReLU is introduced

Logistic Sigmoid activation  $g(z) = \delta(z)$  closely related

Hyperbolic Tangent activation  $g(z) = \tanh(z)$  }  $\tanh(z) = 2\delta(2z) - 1$

- discouraged to be used as activation in feedforward network because of easily saturated (only use as output units with appropriate cost func.)
- If must use sigmoidal activation function, tanh performs better because  $\tanh(0) = 0$  (resembles identity function near 0)

## 6.3.3 Other Hidden Units

Many unpublished hidden units perform comparably to the traditional ones  
↳ uninterested since so common

Some special highlights:

- no activation e.g. some layers are purely linear reduce parameters
- Softmax units

→ Radial basis function (RBF)

$$h_i = \exp(-\vec{z}_i^T \|W_{i,:} - \vec{x}\|^2)$$

active when  $\vec{x} \rightarrow W_{i,:}$   
saturates to 0 for most  $\vec{x}$ , difficult to optimize

→ Softplus

$$g(a) = \gamma(a) = \log(1 + \exp(a))$$

smooth version of Rectifier  
differentiable everywhere

empirically may not work better than ReLU

→ Hard tanh

$$g(a) = \max(-1, \min(1, a))$$

unlike the rectifier,  
it is bounded

## 6.4 Architecture Design

1st layer =

$$\vec{h}^{(1)} = g^{(1)}(W^{(1)\top} \vec{x} + b^{(1)})$$

and layer:

$$\vec{h}^{(2)} = g^{(2)}(W^{(2)\top} \vec{h}^{(1)} + b^{(2)})$$

and so on

Main considerations:  
depth of the network  
width of each layer

## 6.4.1 Universal Approximation Properties and Depth

### The Universal Approximation Theorem

- A feedforward network with
  - ↳ a linear output layer
  - ↳ at least one hidden unit layer with any "squashing" activation function
- can approximate any Borel measurable function from one-dimensional space to another with any desired nonzero amount of error,
- provided that the network is given enough hidden units
- ✓ guaranteed that a large MLP can represent any function
- ✗ not guaranteed that the training algorithm can learn that function  
 => No-free-lunch Theorem = no universally superior ML algorithm
- ? How large the network needed to be?
  - Single-layer is sufficient
  - but the layer is large = worst case exponential # of hidden units (one unit to one input configuration)
  - Using deeper model
    - ✓ reduce # units
    - ✓ reduce generalization error

## 6.4.2 Other Architectural Considerations

CNN, RNN, skip connections, dropout

## 6.5 Back-Propagation and Other Differentiation Algorithms

Forward propagation = when the input  $\vec{x}$  provides the initial info that then propagates up to the hidden units of each layer and finally produces  $\vec{y}$

during training, continue until produces  $J(\theta)$  the scalar loss

Back-propagation / Backprop = info from the cost then flow backward to compute the gradient

### 6.5.1 Computational Graphs

### 6.5.2 Chain Rule of Calculus

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \quad \frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$n \times m$  Jacobian matrix

$$\nabla_{\vec{x}} \vec{z} = \left( \frac{\partial \vec{z}}{\partial \vec{x}} \right)^T \nabla_{\vec{y}} \vec{z} \quad \nabla_{\vec{x}} \vec{z} = \sum_j (\nabla_{\vec{x}} Y_j) \frac{\partial z}{\partial Y_j}$$

$\vec{x} \in \mathbb{R}^m \quad \vec{y} \in \mathbb{R}^n$

Tensor Notation

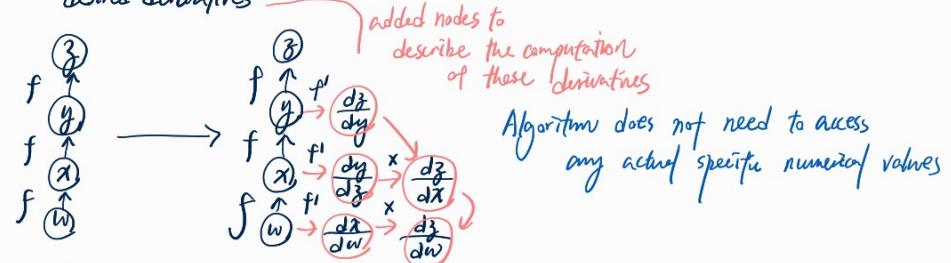
6.5.3 Recursively Applying the Chain Rule to obtain Backprop

6.5.4 Back-Propagation Computation in Fully Connected MLP

6.5.5 Symbol-to-Symbol Derivatives

Symbol-to-Number differentiation: Torch, Caffe

Symbol-to-Symbol differentiation: Theano, Tensorflow  
provides a symbolic description of the desired derivatives



6.5.6 General Back-Propagation

6.5.7 Example: Back-Propagation for MLP Training

6.5.8 Complications

- operations that return more than one tensor
- memory consumption (e.g. summation of many tensors; maintain only a single buffer)
- handle various data types (e.g. 16-bit, 32-bit floats, integers and add to this buffer until completed)
- undefined gradients

6.5.9 Differentiation outside the Deep Learning Community

The field of Automatic Differentiation = e.g. Reverse mode accumulation  
determine the order of evaluation (backprop is only one approach)  
to get lowest computational cost (NP-complete)

6.5.10 Higher-Order Derivatives

Support higher-order derivatives = Theano and Tensorflow

Instead of explicitly computing the Hessian (easily infeasible to even represent, too many parameters)

Krylov Methods

Iterative techniques for e.g. approx. inverting a matrix  
finding approx. to eigenvectors/eigenvalues  
using only matrix-vector products

## 6.6 Historical Notes

Chain Rules: 17-th century

Gradient Descent: 19-th century

Perception = 1940s (inability to learn XoR)

Efficient Chain Rule based on DP = 1960s and 1970s

Neural Network Research = early 1990s

Modern Deep Learning Renaissance = starts in 2006

From 1980s = some backprop and gradient descent in use

Improvement from 1986 to 2015 =

① Larger dataset

② Bigger Neural network due to more powerful computers & better software infrastructure

Replacement of MSE with cross-family of loss functions

Replacement of Sigmoid hidden units with piecewise linear units e.g. ReLU  
since around 2009

Since 2012 =

gradient-based learning in feedforward networks

→ viewed as a powerful technique

## Chapter 7 Regularization for Deep Learning

Regularization: strategies to reduce test error,  
possibly at the expense of increased training error

Strategies =

adding restrictions on the parameter values

adding extra terms in the objective function (also a soft constraint on the parameter value)

designed to encode specific prior knowledge

designed to express a general preference for a simpler model class

to promote generalization

Sometimes necessary to make an undetermined problem determined

Ensembled methods = combine multiple hypotheses to explain the training data

Deep learning:

based on regularizing estimators =

trading increased bias for reduced variance

high: underfitting

change in resampling

high: overfitting

Three Situations of the model family being trained =

① Excluded the true data-generating process

- underfitting, high bias
- ② matched the true data-generating process
  - ③ included the generating process  
but also many others
- Regularization:  
 more from  
 ③ to ②
- variance dominates the  
 estimation error

Practically,

almost always situation ②: true data-generating process is too complex  
aim to find the best fitting model (min. generalization error)  
that is (large and) regularized appropriately

## 7.1 Parameter Norm Penalties

Limit the capacity of models (NN, linear/logistic regression)  
by adding a parameter norm penalty

$$\tilde{J}(\theta; X, \vec{y}) = J(\theta; X, \vec{y}) + \alpha \Delta_1(\theta)$$

where  $\alpha \in [0, \infty]$  a hyperparameter

Typically,  $\Delta_1$  only penalizes the weights of the affine transformation at each layer  
leaving the bias unregularized (can introduce underfitting)

### 7.1.1 $L^2$ Parameter Regularization

Commonly known as Weight Decay

$$\Delta_2(\theta) = \frac{1}{2} \|\vec{w}\|_2^2$$

Also Ridge Regression or Tikhonov regularization

Total Objective function:

$$\tilde{J}(\vec{w}; X, \vec{y}) = \frac{\alpha}{2} \vec{w}^T \vec{w} + J(\vec{w}; X, \vec{y})$$

Parameter Gradient:

$$\nabla_{\vec{w}} \tilde{J}(\vec{w}; X, \vec{y}) = \alpha \vec{w} + \nabla_{\vec{w}} J(\vec{w}; X, \vec{y})$$

Single Gradient Step:

$$\vec{w} \leftarrow (1 - \varepsilon \alpha) \vec{w} - \varepsilon \nabla_{\vec{w}} J(\vec{w}; X, \vec{y})$$

Quadratic Approximation of the Objective 100% exact linear regression model with MSE

$$\hat{J}(\theta) = J(\vec{w}^*) + \frac{1}{2} (\vec{w} - \vec{w}^*)^T H (\vec{w} - \vec{w}^*) \quad H: \text{Hessian of } J \text{ wrt. } \vec{w}$$

$\hat{J}$  is minimum when  $\nabla_{\vec{w}} \hat{J}(\vec{w}) = H(\vec{w} - \vec{w}^*) = 0$

Adding the weight decay gradient:

$$\alpha \vec{w} + H(\vec{w} - \vec{w}^*) = 0 \quad \vec{w}: \text{denote the minimum}$$

$$\vec{w} = (H + \alpha I)^{-1} H \vec{w}^* \quad H \text{ real and symmetric}$$

$$= Q(L_1 + \alpha I)^{-1} L_1 Q^T \vec{w}^* \quad \text{decompose into } H = Q D Q^T$$

### Effect of weight decay:

rescale  $\vec{w}^*$  along the axis defined by the eigenvectors of  $H$  by  $\frac{\lambda_i}{\lambda_i + \alpha}$

When  $\lambda_i \gg \alpha$ , regularization effect is small

$\lambda_i \ll \alpha$ , shrink to nearly 0 magnitude

### Example in Linear Regression:

Total Objective function =

$$(X\vec{w} - \vec{y})^T (X\vec{w} - \vec{y}) + \frac{1}{2}\alpha \vec{w}^T \vec{w}$$

Solution form =

$$\vec{w} = (\underbrace{X^T X}_{\text{diagonal entries: covariance matrix}} + \alpha I)^{-1} X^T \vec{y}$$

Effect =

the learning algo "perceives" the input  $X$  as having higher variance,  
 so shrink the weights on features whose variance with the output target  
 is low compared to the added variance

### 1.1.2 L1 Regularization

$$\mathcal{L}_1(\theta) = \|\vec{w}\|_1 = \sum_i |w_i|$$

### Example in Linear Regression

Total Objective function :

$$\tilde{J}(\vec{w}; X, \vec{y}) = \alpha \|\vec{w}\|_1 + J(\vec{w}; X, \vec{y})$$

Subgradient =

$$\nabla_{\vec{w}} \tilde{J}(\vec{w}; X, \vec{y}) = \alpha \cdot \text{sign}(\vec{w}) + \nabla_{\vec{w}} J(X, \vec{y}; \vec{w})$$

Effect =

Regularization contribution to the gradient is a constant  
 instead of scaling linearly with each  $w_i$

### Quadratic Approximation of the Objective:

$$\text{Gradient } \nabla_{\vec{w}} \hat{J}(\vec{w}) = H(\vec{w} - \vec{w}^*)$$

Further simplifying assumption:

Hessian is diagonal;  $H = \text{diag}([H_{1,1}, \dots, H_{n,n}])$  where  $H_{i,i} > 0$   
 holds if data has been preprocessed to remove correlation between the  
 input features, e.g. by PCA

$$\hat{J}(\vec{w}; X, \vec{y}) = J(\vec{w}^*; X, \vec{y}) + \sum_i \left[ \frac{1}{2} H_{ii} (\vec{w}_i - \vec{w}_i^*)^2 + \alpha |\vec{w}_i| \right]$$

Analytical Solution :

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}$$

①  $w_i^* \leq \frac{\alpha}{H_{i,i}}$  → pushes  $w_i$  to zero

②  $w_i^* > \frac{\alpha}{H_{i,i}}$  → regularization only shifts  $w_i$  in that direction by  $\frac{\alpha}{H_{i,i}}$

Similarly for  $w_i^* < 0$  → either zero or less negative by  $\frac{\alpha}{H_{i,i}}$

Effect =

Solution that is more sparse

Compared to L2 regularization

$$\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^* \rightarrow \tilde{w}_i \text{ is non-zero when } w_i \text{ is non-zero}$$

Common: Feature Selection Mechanism

LASSO (Least Absolute Shrinkage and Selection Operator)

Integrates an  $L^1$  penalty

with a linear model and a least-square cost function.

suggesting features to be safely discarded

$L^2$  Regularization =

= MAP Bayesian inference with a Gaussian prior on the weights

$L^1$  Regularization:

= MAP Bayesian inference's log-prior term when the prior is an Isotropic Laplace distribution over  $\vec{w} \in \mathbb{R}^n$

$$\log p(\vec{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) \propto \alpha \|\vec{w}\|_1$$

## 7.2 Norm Penalties as Constrained Optimization

Constraint  $\|\vec{w}\|_2 \leq k$ :

construct a generalized Lagrange function:

$$L(\theta, \alpha; X, \vec{y}) = J(\theta; X, \vec{y}) + \alpha (\|\vec{w}\|_2 - k)$$

solution:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \max_{\alpha \geq 0} L(\theta, \alpha)$$

Effect of the constraint:

fix  $\alpha^*$ , view the problem as a function of  $\theta$ :

$$\theta^* = \underset{\theta}{\operatorname{argmin}} L(\theta, \alpha^*)$$

$$= \underset{\theta}{\operatorname{argmin}} J(\theta; X, \vec{y}) + \alpha^* \|\vec{w}\|_2$$

exactly the same as the regularized training of minimizing  $\hat{J}$   
 can view Parameter Norm Penalty → constraints on the weights

$\|\vec{w}\|_2$  is  $L^2$  norm → weights constrained to lie in an  $L^2$  ball

$\|\theta\| \leq k \rightarrow$  weights constrained to lie in an  $L_1$  ball  
larger  $\alpha \rightarrow$  smaller constraint region  
but do not directly know the  $k$

## Explicit Constraints & Reprojection but not enforcing constraints with penalties -

- ✓ can modify Stochastic Gradient Descent
  - to project  $\theta$  back to the nearest point  $\|\theta\| < k$  when we know the approx.  $k$
- ✓ penalties can cause non-convex optimization procedures
  - to get stuck in local minima corresponding to small  $\theta$
  - so explicit constraints by reprojection work better
- ✓ impose stability on the optimization procedure
  - e.g. reprojection prevents numerical overflow
  - for some positive feedback loops that keeps increasing weights when the learning rate is high
  - e.g. constraint the norm of each column of the matrix (instead of the  $\ell_1$  norm)

## 7.3 Regularization and Under-Constrained Problems

Many linear models and PCA =

depends on  $(X^T X)^{-1}$  but not possible if  $X^T X$  singular has no variance in some direction  
Regularized matrix  $X^T X + \alpha I$  is guaranteed to be invertible

e.g. data-generating distribution  
or when no variance is observed e.g.  $m > n$

↳ then have closed-form solutions

Guarantees the convergence of iterative methods applied to underdetermined problems

e.g. stops gradient descent when the weights are too high to cause an overflow

Apply to Basic Linear Algebra

The Moore-Penrose pseudoinverse stabilizes underdetermined linear equations with regularization

$$X^+ = \lim_{\alpha \rightarrow 0} (X^T X + \alpha I)^{-1} X^T$$

→ perform linear regression with weight decay

## 7.4 Dataset Augmentation

Create fake data and add to the Training dataset

e.g. classification

effective: object recognition  
speech recognition

Noise Injection in the input to a neural network

improve robustness of the neural network  
can apply to hidden units  
dropout = constructing new inputs by  
multiplying by noise

Important to take the effect

of data augmentation when comparing ML algorithms

e.g. transforming the input

✓ Gaussian Noise = still part of the ML algo

✗ Specific to domain = should be considered as separate  
e.g. cropping image preprocessing steps

## 7.5 Noise Robustness

For same models, addition of noise with infinitesimal variance  
at the input of the model

= imposing a penalty on the norm of the weights

Adding noise to the weights

primarily used in RNN

a Stochastic implementation of Bayesian inference over the weights

Consider the model weights to be uncertain,  
representable via a probability distribution that reflects  
this uncertainty

Encouraging stability of the function

Least-squares cost function

$$J = \mathbb{E}_{p(x,y)} [(\hat{y}(\vec{x}) - y)^2]$$

Random perturbation added to the network weights

$$\epsilon_w \sim \mathcal{N}(\epsilon; \vec{\theta}, \mathbf{I})$$

Objective function

$$\tilde{J}_w = \mathbb{E}_{p(\vec{x}, y, \epsilon_w)} [(\hat{y}_{\epsilon_w}(\vec{x}) - y)^2]$$

$$= \mathbb{E}_{p(\vec{x}, y, \epsilon_w)} [\hat{y}_{\epsilon_w}^2(\vec{x}) - 2y\hat{y}_{\epsilon_w}(\vec{x}) + y^2]$$

For small  $\epsilon$ , equivalent to

having an additional regularization term =

$$\gamma \mathbb{E}_{p(\vec{x}, y)} [\|\nabla_{\epsilon_w} \hat{y}(\vec{x})\|^2]$$

Effect:

goes to the region of parameter space where  
small perturbations of the weights have a relatively  
small influence on the outputs

### 7.5.1 Injecting Noise at the Output Targets

Most datasets have some mistakes in the  $y$  labels  
 harmful to maximize  $\log p(y|\vec{x})$  when  $y$  is a mistake  
 ↓  
 Model the noise on the labels  
 assume the label  $y$  is correct with probability  $1-\epsilon$   
 ✓ easy to incorporate into the cost function analytically  
 rather than explicitly drawing noise samples

### Label Smoothing

Replacing the hard 0 and 1 classification targets  
 with targets of  $\frac{\epsilon}{k-1}$  and  $1-\epsilon$  respectively  
 (softmax with  $k$  outputs)

- Can use standard Cross-entropy loss
- ✓ prevent the pursuit of hard probabilities without discouraging correct classification

## 7.6 Semi-Supervised Learning

Both unlabeled examples from  $P(\vec{x})$  and labeled examples from  $P(\vec{x}, \vec{y})$  are used to estimate  $P(\vec{y}|\vec{x})$  or predict  $\vec{y}$  from  $\vec{x}$

Deep Learning =

refers to learning a representation  $\vec{h} = f(\vec{x})$

Graph = learn a representation s.t. examples from the same class then a linear classifier have similar representations in the new space might achieve better generalization

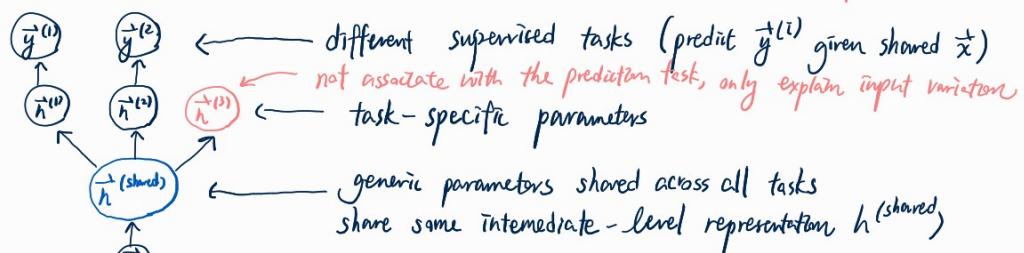
E.g. PCA as a preprocessing step before applying a classifier on the projected data

Instead of having separated unsupervised and supervised components in the model, one can construct a model in which

expresses a Generative model of  $P(\vec{x})$  or  $P(\vec{x}, \vec{y})$  → Unsupervised / Generative criterion  $-\log P(\vec{x})$  or  $-\log P(\vec{x}, \vec{y})$   
 a prior share parameters with a Discriminative model of  $P(\vec{y}|\vec{x})$  → Supervised criterion  $-\log P(\vec{y}|\vec{x})$

## 7.7 Multitask Learning

Improve generalization by pooling the examples arising out of several tasks (can be seen as soft constraints imposed on the parameters)



Underlying prior belief: among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks

## 7.8 Early Stopping

Everytime the error on the validation set improves,  
we store a copy of the model parameters.

Terminate the algorithm when no parameters have improved over  
the best recorded validation error for some pre-specified  
number of iterations  $\rightarrow$  Early Stopping

- ✓ very efficient hyperparameter selection algorithm  
parameter: # of training steps  
controlling the effective capacity of the model

✗ cost: run validation set evaluation periodically during training  
ideally done in parallel in a separate machine / CPU // GPU

✗ cost to obtain a copy of the hyperparameters  
(generally negligible)

✓ unobtrusive form of regularization (no change in objective function, etc.)

✗ validation set won't be used in training

$\rightarrow$  can use it during second-pass training

① best parameters in 1st pass with all data  
② continue training from the best parameters in 1st pass  
with all data

Why:

early stopping restricts the optimization procedure  
to a relatively small volume of parameter space in the neighborhood  
of the initial parameter value  $\theta_0$

## 7.9 Parameter Tying and Parameter Sharing

Another way to express our prior knowledge:

there are some dependences between the model parameters

e.g. the parameters should be close to each other

$\rightarrow$  add parameter norm penalty  $\Omega_2(w^{(A)}, w^{(B)}) = \|w^{(A)} - w^{(B)}\|_2^2$

$\rightarrow$  force sets of parameters to be equal i.e. parameter sharing

✓ only a subset of parameters needs to be stored in memory

### 7.9.1 Convolutional Neural Networks

CNN on computer vision  $\rightarrow$  most popular and extensive use of parameter sharing

e.g. images  $\rightarrow$  invariant to translation

CNN  $\rightarrow$  share parameters across multiple image locations

i.e. can find a cat with the same cat detector

whether the cat appear at column 2 or 21 in the image

## 7.10 Sparse Representations

Weight decay: place a penalty on the model parameters

Another strategy: place a penalty on the activations of the units in a neural network  
encouraging the activations to be sparse

Representational sparsity

e.g. sparse  $\vec{v}$  (many elements of the representation are zero)

Representational regularization

some sorts of mechanisms in parameters regularization

e.g. Norm regularizers

L<sub>1</sub>-norm penalty regularization of representations

$$\tilde{J}(\theta; X, \vec{y}) = J(\theta; X, \vec{y}) + \alpha \|\vec{h}\|_1$$

L<sub>1</sub> penalty on representation  $\rightarrow$  representation sparsity

$$\|\vec{h}\|_1 = \sum_i |h_i|$$

Orthogonal Matching Pursuit (OMP-k)

encodes an input  $\vec{x}$  with representation  $\vec{h}$  that

$$\underset{\substack{\vec{h} \\ \|\vec{h}\|_0 \leq k}}{\operatorname{argmin}} \|\vec{x} - W\vec{h}\|_2^2$$

# non-zero entries of  $\vec{h}$  can be solved efficiently  
when  $W$  is orthogonal

e.g. OMP-1 can be very effective feature extractor for deep architectures

## 7.11 Bagging and Other Ensemble Methods

Bagging (bootstrap aggregating) reduces generalization error

by combining several models =

train several models separately, vote on the output for the test examples  
by construct different datasets with sampling (allow repetition/duplicates)  
some model, training algo, and objective functions

An example of Model Averaging strategy, known as Ensembled method

Usually better:

consider  $k$  regression models, each make error  $\varepsilon_i$  on each example, with errors drawn from a zero-mean multivariate normal distribution

$$\text{with variances } \mathbb{E}[\varepsilon_i^2] = \nu,$$

$$\text{covariances } \mathbb{E}[\varepsilon_i \varepsilon_j] = C$$

then the error made by the average prediction of all ensemble models is  $\frac{1}{k} \sum_i \varepsilon_i$ , and the expected squared error is:

$$\begin{aligned} \mathbb{E}\left[\left(\frac{1}{k} \sum_i \varepsilon_i\right)^2\right] &= \frac{1}{k^2} \mathbb{E}\left[\sum_i \left(\varepsilon_i^2 + \sum_{j \neq i} \varepsilon_i \varepsilon_j\right)\right] \\ &= \frac{1}{k} \nu + \frac{k-1}{k} C \end{aligned}$$

But if errors are perfectly uncorrelated, if errors are perfectly correlated,

$C=0 \Rightarrow$  expected squared error is only  $\frac{1}{k} \nu \Rightarrow$  inversely proportional to the ensemble size

\* On average, the ensemble perform at least as good as any of its members, and if the members make independent errors, the ensemble will perform significantly better

→ Benchmark comparisons are usually with a single model

!!! Not all ensembled methods are fair regularization

BreastTune constructs an ensemble with higher model capacity

## 7.12 Dropout

Dropout: making bagging practical of very large neural networks.

trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network  
e.g. remove a unit by multiplying its output value by zero

To train with dropout

- ① use a minibatch-based learning algo that makes small steps  
e.g. SGD
- ② each time when loading a mini-batch,  
randomly sample a different binary mask  
to apply to all the input and hidden units in the network
- ③ The probability of sampling a mask value of 1 is a hyperparameter  
typically, input unit  $\rightarrow 0.8$   
hidden unit  $\rightarrow 0.5$
- ④ Run forward propagation, back-propagation, and learning updates

Dropout compared to Bagging

Bagging :  
<sup>(a)</sup> models are all independent, <sup>(b)</sup> each model converges to its respective training set

Dropout =  
<sup>(a)</sup> models share parameters,

<sup>(b)</sup> each model inherit a different subsets of parameters from the parent network

<sup>(b)</sup>  
a tiny fraction of the possible subnetworks are each trained for a single step  
Beyond these, dropout follows bagging  
e.g. each subnetwork encounters a subset of training set

Weight Scaling Inference Rule

Approximate ensemble by evaluating  $p(y|\vec{x})$  in one model:  
the model with all units, but with the weights going out of unit  $i$   
multiplied by the probability of including unit  $i$

No theoretical argument yet, but empirically works well

This weight scaling rule is exact for:

- many models without nonlinear hidden units
- regression networks with conditionally normal output

✓ Dropout can be more effective than other

Standard computationally inexpensive regularizers

e.g. weight decay, filter norm constraints, sparse activity regularizations

✓ computationally cheap for cost per step (only defining binary mask)

✓ works well with nearly all models that uses a distributed representation  
and can be trained with SGD

✗ Cost of using dropout in a complete system can be significant

dropout for regularization  $\Rightarrow$  reduces the effective capacity of a model

to offset  $\rightarrow$  must increase the size of the model  
e.g. much larger model, more iterations of the training algo

X less effective when extremely few labeled training examples

! Regularization effect of the bagged ensemble is only achieved when the stochastically sampled ensemble members are trained to perform well independently of each other

**Dropout key insight:** training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing.

**Ensembled models that share hidden units:** each hidden unit must perform well regardless of which other hidden units in the model (biology: evolutionary pressure)

**Multiplicative Noise:** avoid pathological solution to the noise robustness problem  
(e.g. make  $h_i$  very large so noise  $\epsilon_i$  insignificant  $\leftarrow$  won't work)  
for additive noise

## 7.13 Adversarial Training

Adversarial example =

examples that are intentionally constructed by an optimization to search for an input  $x'$  near  $x$  s.t. the model output is very different at  $x'$

Adversarial training =

train on adversarially perturbed examples from the training set to reduce error rate on the original i.i.d. test set

Primary causes of Adversarial examples:

Excessive linearity e.g. NN are built with primary linear building blocks

Adversarial training Some built overall functions are highly linear

discourages by encouraging the network to be  $\rightarrow$  easy to optimize, but the value of a linear function can change very rapidly

locally constant in the neighborhood of the training data  $\rightarrow$  e.g. if each input changed by  $\epsilon$ , then a linear function with weights  $\vec{w}$  can change by as much as  $\epsilon \|\vec{w}\|_1$   $\rightarrow$  large when  $\vec{w}$  high-dimensional

local constancy prior

Purely linear models e.g. logistic regression cannot resist adversarial examples

Neural Network can represent functions from nearly linear to nearly local constant

$\rightarrow$  can capture linear trend in the training data + learn to resist local perturbation

## Virtual Adversarial Examples

adversarial examples generated using not the true label but a label provided by a trained model

\* train the classifier to assign the same label to  $x$  and  $x'$  where originally it outputs different labels  $\hat{y}$  and  $y'$

- ↳ encourages the classifier to learn a function that is robust to small changes along the manifold where the unlabeled data lie
- Assumption: different classes usually lie on disconnected manifolds
- ✓ a means of accomplishing semi-supervised learning

## 7.14 Tangent Distance, Tangent Prop and Manifold Tangent Classifier

Overcame the curse of dimensionality:

manifold hypothesis  $\rightarrow$  assume data lies near a low-dim manifold

### Tangent Distance algorithm

a nonparametric nearest neighbour algo in which the generic Euclidean distance is not used,

but a metric that is derived from knowledge of the manifold near which probability concentrates

$\rightarrow$  cheap alternative that makes sense locally:

approximate manifold  $M_i$  by its tangent plane at  $x_i$ ,

then measure the distance between the two tangents / between a tangent plane and a point

achieved by solving a low-dim linear system

(in the dimension of the manifolds)

requires specifying the tangent vectors.

### Tangent Prop algorithm

for known manifold tangent vectors  $\vec{v}^{(i)}$  at  $\vec{x}$ , (derived or prior)

require local invariance by requiring  $\nabla_{\vec{x}} f(\vec{x})$  to be orthogonal to  $\vec{v}^{(i)}$  at  $\vec{x}$  adding a regularization penalty

$$\mathcal{L}_t(f) = \sum_i ((\nabla_{\vec{x}} f(\vec{x}))^T \vec{v}^{(i)})^2$$

Expect the classifier to change rapidly as it moves the direction normal to the manifold, but not change very much when tangent

### Tangent Prop vs Data Augmentation

Similar: user encodes the prior knowledge by specifying a set of transformations that should not alter the output of the network

difference: Tangent Prop

✓ analytically regularizes, intellectually elegant

✗ only regularizes the model to resist infinitesimal perturbations; explicit data augmentation confers resistance to larger perturbations

✗ infinitesimal approach poses difficulties for models based on rectified linear units

### Double BackProp vs Adversarial Training

Double Backprop: regularizes the Jacobian to be small

Adversarial Training: the "non-infinitesimal" version of double backprop

Both require the model to be invariant to All directions (instead of specific) of changes  
in the input as long as the change is small

## Manifold tangent classifier

- ① use an autoencoder to learn the manifold structure  
by unsupervised learning
- ② use these tangents to regularize a neural net classifier  
as in tangent prop.