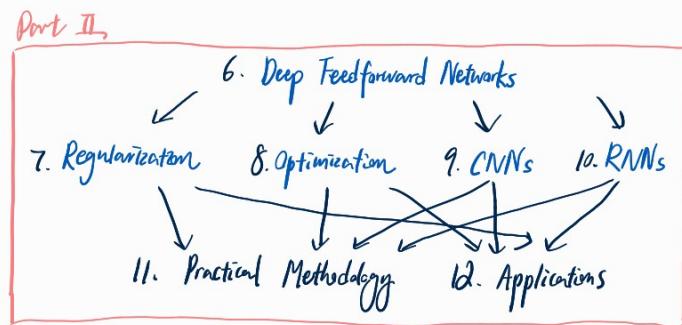


Part II: established DL algorithm



## 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  $X = \{(\vec{x}_0), (\vec{x}_1), (\vec{x}_2), (\vec{x}_3)\}$

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

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

Introduce a feedforward network with hidden units  $\vec{h}$  + the output layer: a linear regression model

$$\vec{h} = f^{(1)}(\vec{x}; W, \vec{c}) \quad y = f^{(2)}(\vec{h}; \vec{w}, b)$$

complete model:  $f(\vec{x}; W, \vec{c}, \vec{w}, b) = f^{(2)}(f^{(1)}(\vec{x}))$

Introduce non-linear model

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

scalar to scalar  $\downarrow$  a fixed (non-linear) activation function

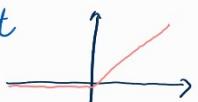
$$\vec{h} = g(W^T \vec{x} + \vec{c})$$

typically an element-wise activation function

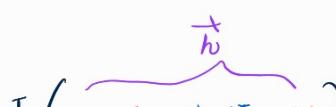
$$\vec{h}_i = g(\vec{x}^T W_{:,i} + c_i)$$

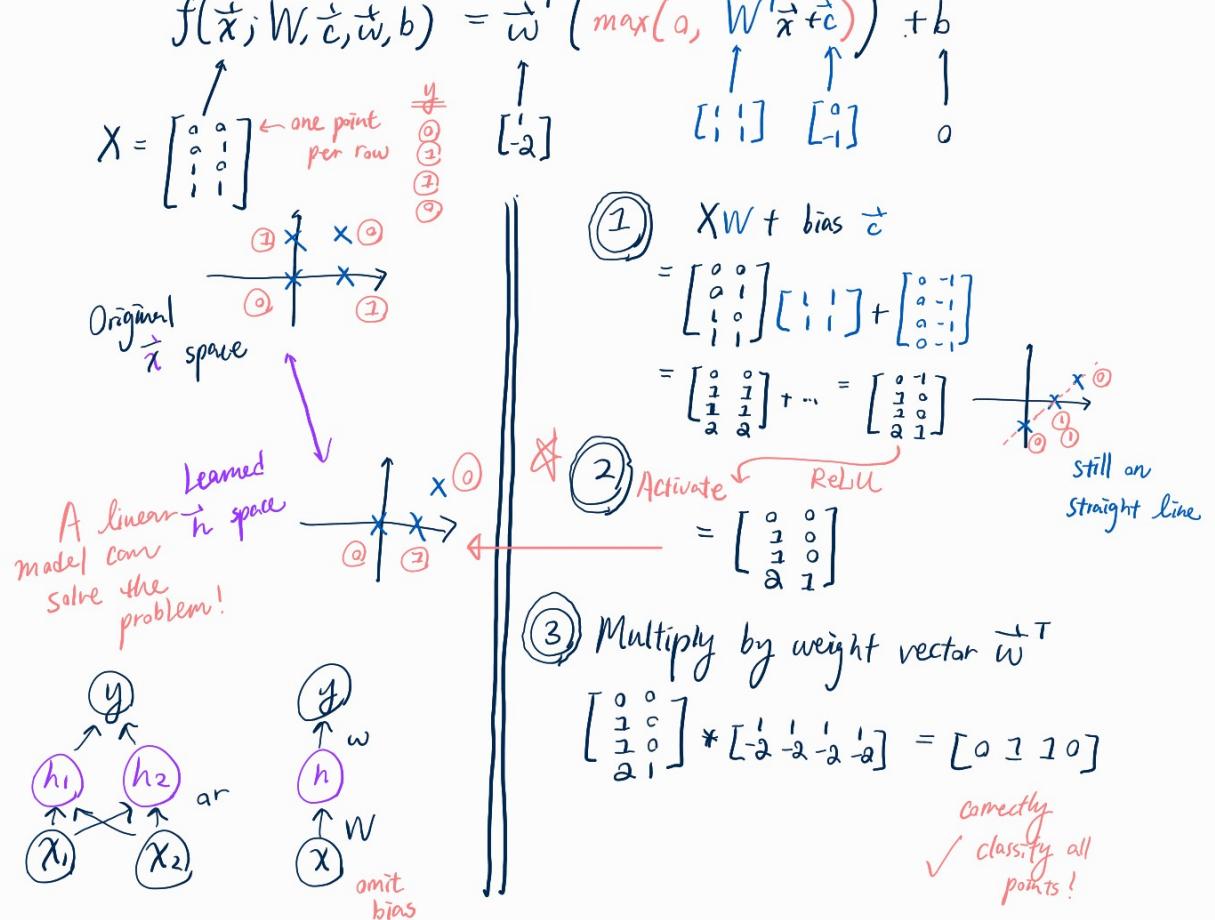
typical Rectified Linear Unit

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



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

biases 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

typically can discard some terms that do not depend on the model parameters and can discard

e.g. when  $p_{\text{model}}(\vec{y} | \vec{x}) = \mathcal{N}(\vec{y}; f(\vec{x}; \theta), I)$ , depends only on variance  
then recover  $J(\theta) = \text{Mean Squared Error cost} + \text{const}$  ↴ and do not need to parametrize

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 ← 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 functions  
 ⇒ learning to choose a function, instead of choosing a set of parameters  
 ⇒ e.g. can design the cost function to have its minimum occur at some specific desirable function, e.g. a function that map  $\vec{x}$  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 → 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  $\vec{y} = \vec{W}^T \vec{h} + b$

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*  $J(\vec{o}) = -\log P(\vec{y}|\vec{x})$  *Softplus*  $\vec{y}(\vec{x}) = \log(1 + \exp(\vec{x}))$

Otherwise saturation of sigmoid  $\Rightarrow$  good pages  $= \vec{y}(1 - 2y)\vec{z}$  where  $\vec{z}$  is the logit

⇒ Saturates only when  $(1 - 2y)\vec{z}$  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  $\vec{z}$  very positive  
 $y=0$  and  $\vec{z}$  very negative

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

### 6.2.2.3 Softmax Units for Multinomial Output Distributions

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

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...  $\rightarrow$   $\sum_i \exp(z_i)$   
 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  $w$ , the parameters of a distribution over  $\vec{y}$  and  
 loss function is  $-\log p(\vec{y}/w(\vec{x}))$

Examples:

Learning the variance of a conditional Gaussian for  $\vec{y}$

Multimodal regression  $\rightarrow$  Gaussian mixtures in Neural Network

effective in generating 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

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

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

Tensor Notation

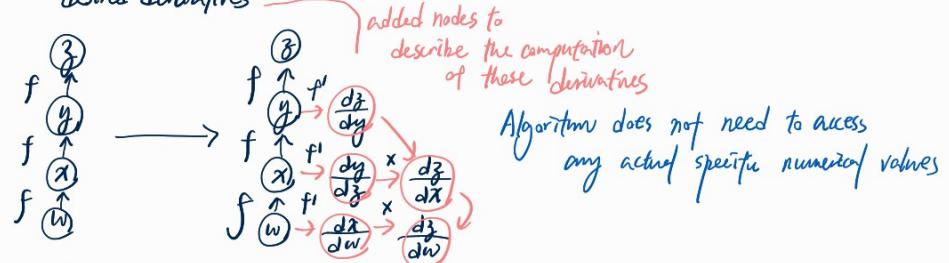
6.5.3 Recursively Applying the Chain Rule to obtain Backprop

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:

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

$$\vec{w} = (H + \alpha I)^{-1} H \vec{w}^*$$

$$= Q(L_1 + \alpha I)^{-1} L_1 Q^T \vec{w}^* \quad H \text{ real and symmetric}$$

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 :

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

Subgradient =

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

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 correspond 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  
eg 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 =

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

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

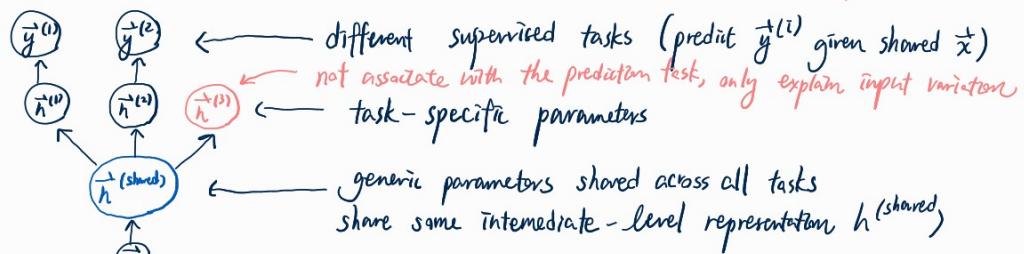
E.g. PCA as a preprocessing step before applying a classifier on the projected data

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 in 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

→ explicitly introduce a 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.

# Chapter 8 Optimization for Training Deep Models

Focus on one particular case of optimization:

finding the parameters  $\theta$  of a neural network

to significantly reduce a cost function  $J(\theta)$ ,

which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms

## 8.1 How Learning differs from Pure Optimization

Machine Learning: reduce the cost function  $J(\theta)$  in the hope of improving the performance measure ← indirect

Pure Optimization = minimizing  $J(\theta)$  is the goal itself

Typical cost function:

$$J^*(\theta) = \underset{\substack{(\vec{x}, y) \text{ w/p data} \\ \text{data-generating distribution } p_{\text{data}}}}{\mathbb{E}} [L(f(\vec{x}; \theta), y)]$$

per-example loss function

### 8.1.1 Empirical Risk Minimization

$J^*(\theta)$  over the true underlying data distribution  $p_{\text{data}}$  → Risk

Risk optimization = an optimization solvable by optimization algo if we knew  $p_{\text{data}}$

when we don't know but only have training samples  
→ a machine learning problem

Empirical Risk = replacing the true distribution  $p(\vec{x}, y)$  with the empirical distribution  $\hat{p}(\vec{x}, y)$

$$\underset{\vec{x}, y \sim \hat{p}_{\text{data}}(\vec{x}, y)}{\mathbb{E}} [L(f(\vec{x}; \theta), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\vec{x}^{(i)}; \theta), y^{(i)})$$

## → Empirical Risk Minimization

- X prone to overfitting (remember the training set)
- X many cases are infeasible e.g. 0-1 loss, no useful derivatives

Must use a slightly different approach with a more different quantity

### 8.1.2 Surrogate Loss Functions and Early Stopping

Surrogate Loss Function = act as a proxy but has advantages

- ✓ even when the expected 0-1 loss is 0, it can improve the robustness of the classifier by further pulling the classes apart etc.
  - e.g. negative log-likelihood of the correct class
  - or surrogate for the 0-1 loss

Usually halts when reaching the early-stopping criterion → that is based on the true underlying loss function,  
even if gradient is still large  
→ unlike pure optimization that only considered to be converged when gradient is small e.g. 0-1 loss on validation set

### 8.1.3 Batch and Minibatch Algorithms

Maximum likelihood estimation problems when viewed in log space.

$$\theta_{\text{ML}} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^m \log p_{\text{model}}(\vec{x}^{(i)}, y^{(i)}, \theta)$$

Maximizing this sum is equivalent to maximizing the expectations over the empirical distribution defined by the training set

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

Most of the properties of  $J$  are also expectations over the training set

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\vec{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{model}}(\vec{x}, y; \theta)$$

- X Computing the expectation exactly is very expensive  
→ instead randomly sampling a small number of examples

#### Batch (or Deterministic) Gradient methods

process all the training examples simultaneously in a large batch

(but "batch" refer to the minibatch by minibatch stochastic gradient descent)

#### Stochastic (or Online) methods

a single example at a time (online → drawn from continuous streams)

#### Minibatch (Stochastic) methods (now commonly call stochastic methods)

use more than one but fewer than all training examples

## Size of Minibatch factors

- larger batch size only gives less than linear returns in estimating the gradient accurately
  - // Standard Error of the mean from  $n$  samples:  $\frac{\sigma}{\sqrt{n}}$
- small batch size offers a regularization effect
  - might need smaller learning rate and thus longer training time
- methods that compute updates only based on gradient  $\vec{g}$ 
  - relatively robust, can handle batch size e.g. 100
- second-order methods which use the Hessian matrix  $H$  to compute update  $H^{-1}\vec{g}$ 
  - requires larger batch size e.g. 10,000
  - because small changes in estimation of  $\vec{g}$  affect  $H^{-1}\vec{g}$  even if  $H$  is perfect
- multicore architectures are usually underutilized by extremely small batches → no reduction in time processing a batch size below some abs. minimum
- GPUs = common to use power of 2 batch sizes for better runtime
- memory scales with the batch size when parallel processing
  - the limiting factor of most hardware setup
- \* Crucial to select minibatches randomly
- \* compute unbiased estimate of the true generalization error as long as no examples are repeated

## §.2 Challenges in Neural Network Optimization

Traditionally, ML avoided the difficulty of general optimization by designing the objective function and constraints s.t. the problem is convex.

Training neural networks → must confront the general nonconvex case

### §.2.1 Ill-Conditioning

Second-order Taylor series expansion of the cost function predicts a gradient descent step of  $-\epsilon \vec{g}$  will add to the cost

$$\frac{1}{2} \epsilon^2 \vec{g}^T H \vec{g} - \epsilon \vec{g}^T \vec{g}$$

squared gradient norm

if this exceeds  $\vec{g}^T \vec{g}$  → ill-condition becomes a problem

Normally, we want learning becomes very slow because even if  $\vec{g}^T \vec{g}$  is strong, the gradient norm to decrease the learning rate must shrink to compensate for in the optimization; → even stronger curvature. the gradient norm might still increase

Test Rule: monitor gradient norm

## 8.2.6 Local Minima

With non-convex functions in neural nets,  
possible to have many local minima due to model identifiability problem.  
e.g. weight space symmetry by rearranging the hidden units  
however usually not a problem since these local minima are equivalent to the cost function

Stuck in local minima (rather than finding global minima)

is only a problem when they have high cost

Test Rule = monitor gradient norm

if it does not shrink, then it's not a local minima/critical point problem

## 8.2.3 Plateaus, Saddle Points and other Flat Regions

In high-dimensional nonconvex problems,

local minima (maxima) are rare compared to saddle points

Empirically, gradient descent can escape saddle points well

→ designed to move downhill and not explicitly designed to seek a critical point

X Newton's method: designed to seek a critical point,  
easily stuck in a saddle point

explains why second-order methods have not succeeded in  
replacing gradient descent for (high-dim) neural network training  
→ requires modification

## 8.2.4 Cliffs and Exploding Gradient

Cliffs: extremely steep regions, caused by multiplication of several weights  
common in RNN / long temporal sequences (one factor for each time step)

Gradient Clipping: when the traditional GD algo proposes making a very large step,  
the gradient clipping heuristics (that jumps too far)  
intervene to reduce the step size

## 8.2.5 Long-Term Dependencies

Example: a computational graph contains a path that consists of multiplying by a matrix  $W$ . After  $t$  steps  $\rightarrow W^t$

Suppose  $W$  has an eigen decomposition  $W = V \text{diag}(\lambda) V^{-1}$

$$W^t = (V \text{diag}(\lambda) V^{-1})^t = V \text{diag}(\lambda)^t V^{-1}$$

any eigenvalues  $\lambda_i \neq 1$  will either explode or vanish

⇒ The vanishing and exploding gradient problem

RNN uses the same  $W$  at each timestep → more common

## 8.2.6 Inexact Gradient

Imperfections in the gradient estimate

## 8.2.7 Poor Correspondence between Local and Global Structure

Much of the runtime of training can be due to the length of trajectory  
needed to arrive at the solution:

Gradient Descent / all learning algorithms  
→ based on making small local moves  
→ previous sections: focus on correcting the directions of these local moves  
that can be difficult to compute  
→ high computational cost

Existing research direction → aimed at finding good initial points  
for problems that have difficult global structure

### 8.2.8 Theoretical Limits of Optimization

Develop more realistic bounds on the performance of optimization algorithms  
→ important goal for machine learning research

## 8.3 Basic Algorithms

### 8.3.1 Stochastic Gradient Descent

Possible to obtain an unbiased estimate of the gradient  
by taking the average gradient on a minibatch of  $m$  examples  
drawn i.i.d from the data-generating distribution.

Critical Parameter: Learning Rate

necessary to decrease the learning rate over time

because SGD gradient estimator introduces a source of noise (random sampling)  
that does not vanish even when arriving at a minimum

(in contrast, batch gradient descent gets gradient = 0 at minimum  
→ can use a fixed learning rate)

Sufficient condition to SGD convergence

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

$\varepsilon_k$ : learning rate  
at iteration  $k$

Common: decay linearly until iteration  $T$

$$\varepsilon_k = (1-\alpha) \varepsilon_0 + \alpha \cdot \varepsilon_T \quad \text{with } \alpha = \frac{k}{T}$$

$T$ : # required to make a few hundred passes through the training set

$\varepsilon_T$ : 1% of  $\varepsilon_0$   
 $\varepsilon_0$ : pick one that is higher than the best-performing lr after the first 100 iterations or so

Best to choose the learning rate by  
plotting the objective function over time

(more art  
than science)

Study the convergence rate of the optimization algorithm

$$\text{Excess Error } J(\theta) - \min_{\theta} J(\theta)$$

the amount of which the current cost function exceeds the min. possible cost

SGD on convex problem:  $O(\frac{1}{\sqrt{k}})$  after  $k$  iterations

Strongly convex problem:  $O(\frac{1}{k})$  Cannot improve these bounds unless extra conditions are assumed

Batch GD converges better in theory,  
but Cramér-Rao bound states that generalization error  
cannot decrease faster than  $O(\frac{1}{K})$   
therefore might not worth to pursue

Also SGD makes rapid initial progress while evaluating for very few examples  
 $\rightarrow$  outweigh its slow asymptotic convergence

Trade-off: gradually increase the minibatch size during learning.

### 8.3.2 Momentum

SGD is popular but slow.

Momentum is designed to accelerate learning esp. of  
 accumulates an exponentially decaying  
 moving average of past gradients and continues to move in their direction

Update Rule:

$$\vec{v} \leftarrow \alpha \vec{v} - \epsilon \nabla_{\theta} \left( \frac{1}{m} \sum_{i=1}^m L(f(\vec{x}^{(i)}; \theta), \vec{y}^{(i)}) \right)$$

$$\theta \leftarrow \theta + \vec{v}$$

$\alpha \in [0, 1]$  hyperparameter  
 how quickly the past contributions

size of each step: exponentially decay

depends on how large and how aligned a sequence of gradients are

common  $\alpha = 0.5, 0.9, 0.99$  (can be adaptive,  
 less important than shrinking start with small then raised)  
 the learning rate  $\epsilon$

$\vec{v}$ : "velocity". With "unit mass",  
 regard as the momentum of the particle  
 where the negative gradient is a force  
 moving a particle through the param space

### 8.3.3 Nesterov Momentum

A variant of momentum inspired by Nesterov's accelerated gradient method

Update Rule:

$$\begin{aligned} \vec{v} &\leftarrow \alpha \vec{v} - \epsilon \nabla_{\theta} \left[ \frac{1}{m} \sum_{i=1}^m L(f(\vec{x}^{(i)}; \theta + \alpha \vec{v}), \vec{y}^{(i)}) \right] \\ \theta &\leftarrow \theta + \vec{v} \end{aligned}$$

gradient is evaluated

For Batch GD,

after the current velocity is applied

Convergence of the excess error =  $O(\frac{1}{K^2})$  improved from  $O(\frac{1}{K})$   
 but does not improve for SGD

## 8.4 Parameter Initialization Strategies

Difficult to design parameter initialization strategy;

the only certainty is the initial parameters needed to "break symmetry" between diff. units  
 If two hidden units with the same activation function are connected to the same inputs,  
 then these units must have different init parameters.  
 so that each unit compute a different function

Common: initialize the weight to values

drawn from a Gaussian or Uniform distribution

The scale of the init distribution  $\rightarrow$  large effect on the outcome + generalization ability $\rightarrow$  larger init weight =

✓ stronger symmetry-breaking effect

✗ might result in exploding values

✗ extreme values causing activation functions to saturate

 $\rightarrow$  Optimization perspective suggests larger weight,

Regularization perspective encourages making smaller

e.g. early stopping  $\rightarrow$  expresses a prior that the final parameters  
should be close to the initial parameters $\rightarrow$  Heuristics = a fully connected layer with  $m$  inputs and  $n$  outputsSample each weight from  $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$   $U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}})$  Normalized InitializationGood idea to treat the initial scale of the weights for each layer  
as a hyperparameter; dense/spARSE initialization also a hyperparameter

Good Rule of Thumb:

look at the range/standard deviation of activations/gradient  
on a single minibatch of dataif unacceptably small activations  $\rightarrow$  increase the weight s.t. can propagate forward through the network

✓ less computationally costly to optimize

Setting bias = 0 is compatible with most weight initialization schemes

Setting variance/precision = usually can set to 1 safely

Initialize via ML  $\rightarrow$  might yield faster convergence, better generalization because  
they encode info about the distribution in the init parameters of the model

## 8.5 Algorithms with Adaptive Learning Rates

If we believe that the directions of sensitivity one axis aligned,

it can make sense to use a separate learning rate for each parameter  
and automatically adapt these learning ratesDelta-bar-delta: early heuristic approach  $\rightarrow$  if the partial derivative of the loss  
wrt the model parameter  
can only be applied on full batch optimization  
 $\rightarrow$  if the partial derivative of the loss  
wrt the model parameter  
remains the same sign  $\rightarrow$  increase the learning rate  
change the sign  $\rightarrow$  decrease the learning rate

Recently, more mini-batch-based methods =

### 8.5.1 AdaGrad

Accumulate squared gradient =

$$\vec{r} \leftarrow \vec{r} + \vec{g} \odot \vec{g}$$

Compute update =

$$\Delta \theta \leftarrow -\frac{\epsilon}{\vec{g} + \vec{r}} \odot \vec{g}$$

for numerical stability,  
small  $\sim 10^{-7}$

Apply update =

### 8.5.2 RMSProp

Accumulate squared gradient =

$$\vec{r} \leftarrow p \vec{r} + (1-p) \vec{g} \odot \vec{g}$$

Compute update =

$$\Delta \theta \leftarrow -\frac{\epsilon}{\vec{g} + \vec{r}} \odot \vec{g}$$

decay rate  $p$

Apply update =

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

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

Adapt the learning rates of all model parameters  $\theta$  by scaling them inversely proportional to  $\sqrt{\sum_i \text{historical squared values of the gradient}}$

Parameters with a larger partial derivative of the loss  $\rightarrow$  a corresponding rapid decrease in their learning rate

✓ convex optimization  $\rightarrow$  desirable theoretical properties

✗ training deep neural network  $\rightarrow$

accumulation of squared gradient from the beginning can result in a premature/excessive decrease in the effective learning rate

$\rightarrow$  only perform well for some but not all deep learning problems

Modifies AdaGrad to perform better in nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average.

AdaGrad:

✗ shrink the learning rate w.r.t. the entire history, the learning rate could be too small when arriving the local convex bowl

RMSProp:

✓ discard the history from the extreme past by an exponentially decaying average s.t. it can converge rapidly after finding a local convex bowl

one of the go-to for DL

### 8.5.3 Adam

Update biased first/second moment estimate, Correct bias in first/second moment

$$\begin{aligned} \hat{s} &\leftarrow p_1 \hat{s} + (1-p_1) \hat{g} & \hat{r} &\leftarrow p_2 \hat{r} + (1-p_2) \hat{g} \odot \hat{g} \\ \hat{s} &\leftarrow \frac{\hat{s}}{1-p_1 t} & \hat{r} &\leftarrow \frac{\hat{r}}{1-p_2 t} \end{aligned}$$

Compute update:

$$\Delta\theta = -\varepsilon \frac{\hat{s}}{\hat{g} + \sqrt{\hat{r}}}$$

Apply update:

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

Adam  $\rightarrow$  "Adaptive Moments"

a variant of RMSProp and momentum with:

① momentum = as an estimate of the **first-order moment** (with exponential weighting) of the gradient

$\leftrightarrow$  if apply momentum to the rescaled gradient, no clear theoretical motivation

② includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin

$\leftrightarrow$  RMSProp also has the second-order moment, but lacks the correction factor

✓ fairly robust to the choice of hyperparameters

### RMSProp with Nesterov Momentum

Compute gradient:

$$\hat{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\vec{x}^{(i)}; \theta + \alpha \vec{v}), \vec{y}^{(i)})$$

Accumulate squared gradient:

$$\hat{r} \leftarrow p \hat{r} + (1-p) \hat{g} \odot \hat{g}$$

Compute update:

$$\vec{v} \leftarrow \alpha \vec{v} - \frac{\varepsilon}{\sqrt{\hat{r}}} \odot \hat{g}$$

Apply update:

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

### 8.5.4 Choosing the Right Optimization Algorithm

Most popular: SGD, SGD with momentum,

RMSProp, RMSProp with momentum,  
AdaDelta, Adam

# Chapter 9 Convolutional Network

Convolutional Neural Networks: process data that has a grid-like topology

e.g. time-series data  $\rightarrow$  1-D grid taking samples at regular time intervals

image data  $\rightarrow$  2-D grid of pixels

## 9.1 The Convolution Operation

Example:

①  $x(t)$ : location of a spaceship at time  $t$

suppose this is noisy, want a averaged measurement  
and also a weighted average that gives more weight to the recent measurements

②  $w(a)$ : weighting function,  $a$  is the age of measurement  $\leftarrow$  must be a valid

Convolution = probability density function

$$s(t) = \int x(a) w(t-a) da$$

$a=0 = t \leftarrow \text{location 0} * \text{weight at age } t$  (oldest)

$a=1 = t-1$

$a=a = t-2$

$\vdots$

$a=t = 0 \leftarrow \text{location } t * \text{weight at age } 0$  (latest)

$$= (x * w)(t)$$

$$\text{approx. by } \sum_{a=-\infty}^{\infty} x(a) w(t-a)$$

$$\text{approx. by } \sum_{a=K_1}^{K_2} x(a) w(t-a)$$

$x(t) =$

Input

$w(a) =$

kernel

output =  
feature map

only a finite set of points  
that matters

2-D image & kernel:

$$\begin{aligned} S(i, j) &= (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i-m, j-n) \\ &= (K * I)(i, j) = \sum_m \sum_n I(i-m, j-n) K(m, n) \text{ "flipping"} \\ &= (K * I)(i, j) = \sum_m \sum_n I(2im, j+in) K(m, n) \text{ Cross-correlation} \end{aligned}$$

## 9.2 Motivation

Sparse interactions, Parameter sharing, Equivariant representations

traditional NN =  
every input unit connects  
with every output unit

traditional NN:  
each element of the weight layer  
is used exactly once when computing  
the output of a layer

Equivariance to translation  
e.g.  $f(x)$  is equivalent to  $g$   
if  $f(g(x)) = g(f(x))$

CNN =  
kernel smaller than the input  
 $\rightarrow$  sparse connectivity

CNN:  
each member of the kernel is  
used at every position of the input  
 $\rightarrow$  reduce parameters,  
dramatically more efficient

e.g. images  
useful to share "edges"  
parameters

## 9.3 Pooling

CNN

Input to

Convolutional stage

$\rightarrow$  Detector stage

$\rightarrow$  Pooling stage

Next



## Pooling

Replaces the output of a net at a certain location with a summary statistics of the nearby outputs

Max pooling = reports max. output within a rectangular neighborhood

✓ make the representation approx. invariant to small translations of the input

✓ useful when we care existence of the feature than its location

⇒ An infinitely strong prior that the function the layer learns must be invariant to small translations

✓ Fewer pooling units than detector units

⇒ computational efficiency

## 9.4 Convolution and Pooling as an Infinitely Strong Prior

Imagine a CNN as being similar to a fully connected net, but with an infinitely strong prior over its weights:

→ weights for one hidden unit must be identical to the weights of its neighbor but shifted in space

→ weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit

Key insight =

Convolution and pooling → can cause underfitting

## 9.5 Variants of the Basic Convolution Function

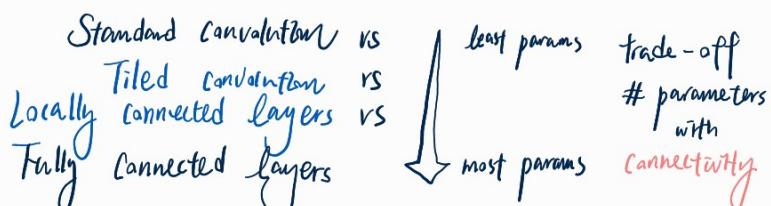
Stride: sample only every s pixels in each direction in the output  
(downsampling the output of the full convolution)

Zero-padding: control the kernel width and the output size separately

padding: {  
optimal padding: ① Valid Convolution: no zero padding (easily shrink to 1x1)  
less between ② Same Convolution: keep input = output size

but: pixels at border influence less than pixels at center → border pixels underrepresented

③ Full Convolution: enough zeros are added for every pixel  
to be visited k times in each direction difficult to learn a kernel  
border pixels are functions of fewer pixels → that performs well at all positions



Crucial operations -	Convolution
to train feedforward	Backprop from output to weights
CNN	Backprop from output to inputs

## 9.6 Structured Outputs

CNN → can be used to output a high-dim structured object, rather than just predicting a class label for a classification task or a real value for a regression task

## 9.7 Data Types

- CNN = ✓ reduce computational cost  
✓ reduce overfitting  
✓ process inputs with varying spatial extents  
(cannot be represented by traditional / matrix multiplication-based NN)  
via diff. # of times applying kernel / pooling layer

	Single Channel	Multichannel
1D	audio waveform	skeleton animation
2D	Fourier transform with audio	Color image
3D	CT scan	Color Video

## 9.8 Efficient Convolution Algorithms

Convolution = convert input + kernel to the frequency domain via Fourier Transform,  
perform point-wise multiplication of the two signals,  
convert back to the time-domain via Fourier Transform

(can be faster than the naive implementation of discrete convolution)

Separable kernel: can be expressed as outer product of (one vector per dimension)  
= several one-dim convolutions

(composed approach sig. faster than performing a d-dim convolution with their products)

## 9.9 Random or Unsupervised Features

One way to reduce the cost of CNN training:

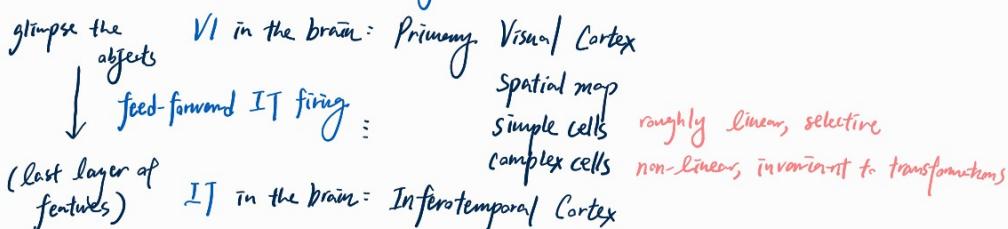
use features that are not trained in a supervised fashion  
(avoid a complete run of forward + backward propagation through the entire network)

Convolutional kernels without supervised training

- initialize randomly
- design by hands
- learn with unsupervised criterion
- greedy layer-wise pretraining

## 9.10 The Neuroscientific Basis for Convolutional Networks

CNN vs Mammalian Vision System



Most V1 cells have weights that are described by Gabor functions

$$\text{Response of a simple cell: } s(I) = \sum_{x \in X} \sum_{y \in Y} w(x,y) I(x,y)$$

Applying weight  $w(x,y)$  to the location  $(x,y)$

2-D Image

$w(x,y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \underbrace{\alpha \exp(-\beta_x x'^2 - \beta_y y'^2)}_{\substack{\text{Gating term: responds only to} \\ \text{centered at } (x_0, y_0), \\ \text{respond to changes in brightness} \\ \text{as more along or line rotated by } \tau \text{ radians} \\ \text{from the horizontal}}} \cos(f x' + \phi)$

define coordinate systems: where  $x' = (x-x_0) \cos(\tau) + (y-y_0) \sin(\tau)$   
 $y' = -(x-x_0) \sin(\tau) + (y-y_0) \cos(\tau)$

responds to changing brightness along  $x'$   
 a function of  $(x', y')$

## 9.11 Convolutional Networks and the History of Deep Learning

CNN: some of the first working deep networks

trained with back-propagation

carried the torch for the rest of deep learning

## Chapter 10: Sequential Modeling: Recurrent and Recursive Nets

Recurrent neural networks (RNNs): processing sequential data

a sequence of values  $\vec{x}^{(1)}, \dots, \vec{x}^{(T)}$

Parameter sharing: shares the same weight across several time steps

each member of the output is produced using the same update rule

applied to the previous outputs  $\rightarrow$  parameter sharing through a very deep computational graph

### 10.1 Unfolding Computation Graphs

$$\vec{h}^{(t)} = f(\vec{h}^{(t-1)}, \vec{x}^{(t)}; \theta) \quad \vec{h} \text{ represents the state}$$

*≈ past sequence of inputs up to t*

*unfolding*  $= g^{(t)}(\vec{x}^{(t)}, \vec{x}^{(t-1)}, \dots, \vec{x}^{(2)}, \vec{x}^{(1)})$

✓ before unfolding: learn a single  $f$  instead of a separate  $g(t)$  for all possible  $t$

$\rightarrow$  allows generalization to sequence lengths that did not appear in the training set

### 10.2 Recurrent Neural Networks

Any function computable by a Turing machine  
 can be computed by such a Recurrent network of a finite size

For each time step  $t=1$  to  $t=T$ :

$$\vec{a}^{(t)} = \vec{b} + W \vec{h}^{(t-1)} + U \vec{x}^{(t)}$$

where

$$\vec{h}^{(t)} = \tanh(\vec{a}^{(t)})$$

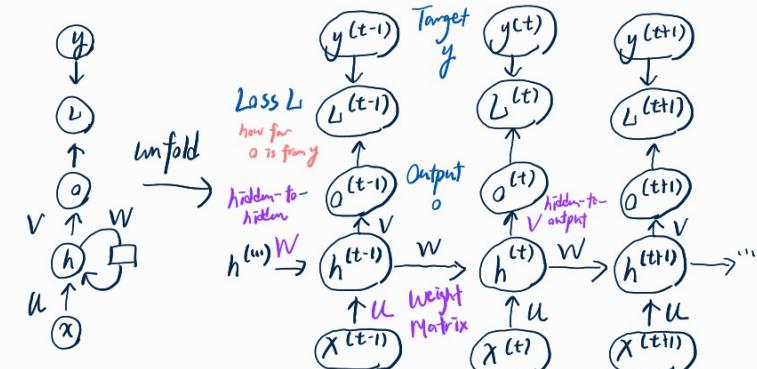
$$\text{Output } \vec{o}^{(t)} = \vec{c} + V \vec{h}^{(t)}$$

$$\text{Predictor } \hat{y}^{(t)} = \text{softmax}(\vec{o}^{(t)})$$

Total Loss

$$L = \left( \{ \vec{x}^{(1)}, \dots, \vec{x}^{(T)} \}, \{ \hat{y}^{(1)}, \dots, \hat{y}^{(T)} \} \right)$$

$$= \sum_{t=1}^T L_t$$



requires Back-propagation through time (BPTT)

$$= - \sum_t \log p_{\text{model}}(\vec{y}^{(t)} | \{\vec{x}^{(1)}, \dots, \vec{x}^{(t)}\})$$

✓ very powerful  
✗ expensive to train, cannot be parallelized

### 10.2.1 Teacher Forcing and Networks with Output Recurrence

Recurrent connections

only from output to hidden units

- ✗ less powerful due to lack of hidden-to-hidden recurrence;  
require outputs to capture all past information  
but outputs are trained to match  $y$

- ✓ training can be parallelized  
no need to compute the previous output  
because the training set produced the ideal value of the output

→ suitable for Teacher Forcing

during training, the model receives the ground truth output  $\vec{y}^{(t)}$  as input at  $t+1$ .

Maximum Likelihood criterion

$$\begin{aligned} & \log p(\vec{y}^{(1)}, \vec{y}^{(2)} | \vec{x}^{(1)}, \vec{x}^{(2)}) \\ &= \log p(\vec{y}^{(2)} | \vec{y}^{(1)}, \vec{x}^{(1)}, \vec{x}^{(2)}) + \log p(\vec{y}^{(1)} | \vec{x}^{(1)}, \vec{x}^{(2)}) \end{aligned}$$

- ✗ if the network is used in an closed-loop mode  
fed-back inputs during training could be too different than that at test time

### 10.2.2 Computing the Gradient in a Recurrent Neural Network

Straightforward = generalized back-propagation to the unrolled computational graph

### 10.2.3 Recurrent Networks as Directed Graphical Model

Above example: losses  $L^{(t)}$  are cross-entropies  
between training targets  $\vec{y}^{(t)}$  and outputs  $\vec{o}^{(t)}$

In principle, possible to use any loss

usually wish to interpret the RNN output as a probability distribution,

then use the cross-entropy with that probability distribution to define the loss

e.g. MSE: cross-entropy loss associated with an output distribution that is a unit Gaussian

When using a predictive log-likelihood training objective,

train RNN to estimate the conditional distribution of the next sequence element  $\vec{y}^{(t)}$  given the past inputs, e.g.

maximize  $\log p(\vec{y}^{(t)} | \vec{x}^{(1)}, \dots, \vec{x}^{(t)})$  or

$\log p(\vec{y}^{(t)} | \vec{x}^{(1)}, \dots, \vec{x}^{(t)}, \vec{y}^{(1)}, \dots, \vec{y}^{(t-1)})$

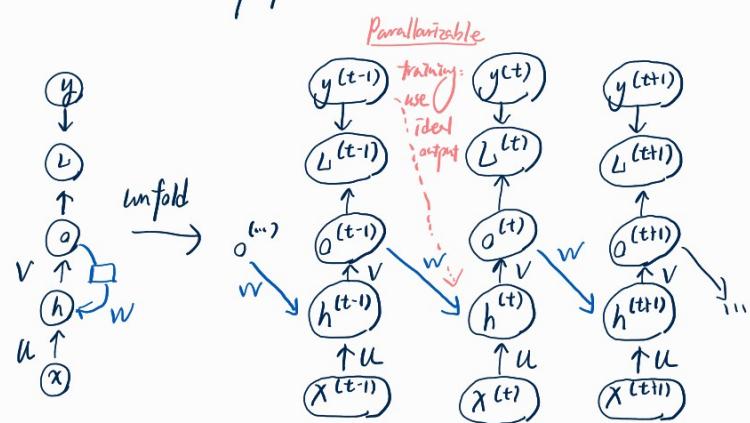
If the model includes connections  
from the output at one timestep to the next timestep

if we feed the actual  $\vec{y}$  values back into the network,

the directed graphical model contains edges from all past  $\vec{y}^{(i)}$  to the current  $\vec{y}^{(t)}$  value

Interpret RNN as a graphical model

one way is to view it as a complete graph that can represent direct dependencies between any pair of  $y$  values

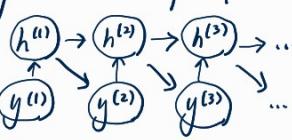


another way: include the hidden units  $\vec{h}$

✓ efficient parametrization

✗ more difficult parameter optimization

RNN assumption: the conditional probability distribution over the variables at  $t+1$  given those at  $t$  is stationary



Determine the length of the sequence

→ add a special symbol corresponding to the end of a sequence (when output is taken from vocabulary)

→ introduce an extra Bernoulli output to either continue or halt at each time step  
*more general!*

→ adds an extra output to predict the sequence length integer  $T$

#### 10.2.4 Modelling Sequences Conditioned on Context with RNNs

RNNs allow the extension of the graphical model view to represent not only a joint distribution over the  $y$  variables, but also a conditional distribution over  $y$  given  $\vec{x}$

$$P(\vec{y}; \theta) \rightarrow P(\vec{y} | w) \text{ when } w = \vec{x} \rightarrow P(\vec{y} | w(\vec{x})) \text{ make } w \text{ a function of } \vec{x}$$

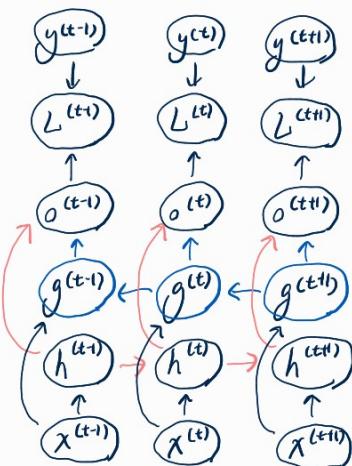
Can provide  $\vec{x}$  as an extra input to RNN when generating the  $\vec{y}$  sequence

- e.g. ① extra input at each time step
- ② as the initial state  $\vec{h}^{(0)}$
- ③ both

"context": the input to RNN

#### 10.3 Bidirectional RNNs

Motivation: output prediction of  $\vec{y}^{(t)}$  that depends on the whole input sentence



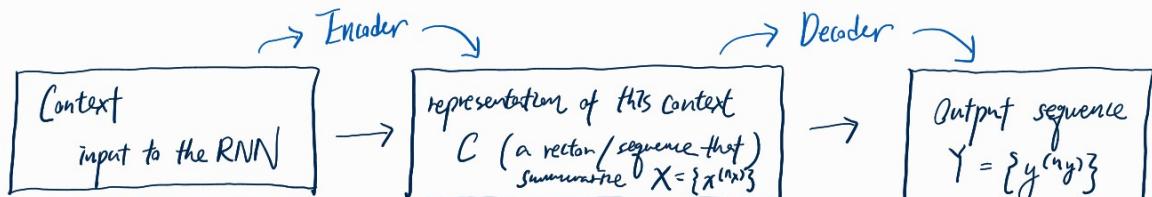
$\vec{h}^{(t)}$ : sub-RNN that moves forward through time

$\vec{g}^{(t)}$ : sub-RNN that moves backward through time

$\vec{o}^{(t)}$ : depends on both the past and the future but is most sensitive to the input around  $t$

#### 10.4 Encoder - Decoder Sequence-to-Sequence Architectures

Motivation: train an RNN to map an input sequence to an output sequence which is not necessarily of the same length



Sequence-to-sequence architecture

the two RNNs are trained jointly to maximize the

average of  $\log P(\vec{y}^{(1)} | \vec{x}^{(1)}) P(\vec{y}^{(2)} | \vec{x}^{(2)}) \dots P(\vec{y}^{(n)} | \vec{x}^{(n)})$  over all pairs of  $\vec{x}$  and  $\vec{y}$  sequences

## 10.5 Deep Recurrent Networks

RNN can be decomposed into

- ① from input to hidden state
- ② from previous hidden state to the next hidden state
- ③ from the hidden state to the output

Shallow transformation = a layer in MLP (learned affine transformation)

Deep = separate MLP (deep) for  
each of the three blocks

followed by a fixed non-linearity

## 10.6 Recursive Neural Networks

Recursive nets vs Recurrent nets

→ tree-like                      → chain-like

✓ depth  $O(\log \tau)$  vs depth  $O(\tau)$

(deal with long-term dependencies)

e.g. NLP

structure of the parse tree of the sentence

## 10.7 The Challenge of Long-Term Dependencies

$$\vec{h}^{(t)} = (W^t)^T \vec{h}^{(0)}$$

$$= Q^T \Lambda^T Q \vec{h}^{(0)} \quad \text{if } W = Q \Lambda Q^T$$

and  $Q$  orthogonal

The eigenvalues are raised to the power of  $t$ ,  
causing eigenvalues with magnitude  $< 1$  decay to zero  
 $> 1$  to explode

## 10.8 Echo State Networks

Echo State Networks (ESNs)

termed Reservoir Computing = hidden units form a reservoir of temporal features  
that may capture different aspects of the history of inputs

via: set the input and recurrent weights s.t. the dynamical system is near the edge of stability

Strategy: fix the Jacobian weights to have some fixed spectral radius  $\max_{\text{its eigenvalues}}$  of the absolute values of  
where information is carried forward through time but does not explode  
because of the stabilizing effect of saturating nonlinearities like tanh

## 10.9 Leaky Units and Other Strategies for Multiple Time Scales

Deal with long-term dependencies:

model that operates at multiple time scales  $\rightarrow$  fine-grained time scales (handle small details)  
 $\rightarrow$  coarse time scales (transfer info from the distant past to the present more efficiently)

### 10.9.1 Adding Skip Connections through Time

Skip connections e.g. recurrent connections with a time delay of  $d$

$\rightarrow$  gradients diminish exponentially as a function of  $\frac{d}{T}$  rather than  $T$

### 10.9.2 Leaky Units and a Spectrum $\rightarrow$ Different Time Scales

Obtain paths on which the product of derivatives is close to 1  
 → have units with linear self-connections and a weight near 1 on these connections

Leaky units : units with linear self-connections

similar to accumulating a running average  $\mu^{(t)} \leftarrow \alpha \mu^{(t-1)} + (1-\alpha) v^{(t)}$

when  $\alpha \approx 1 \rightarrow$  remember info about the past for a long time

✓ Adjusting real-valued  $\alpha \rightarrow$  more smooth and flexible

than the integer-valued skip length in skip-connections

### 10.9.3 Removing Connections

Actively remove length-one connections,  
 replacing them with longer connections

skip-connections:  
 only add connections  
 not remove

## 10.10 The Long, Short-Term Memory and Other Gated RNNs

Gated RNNs → long short-term memory  
 → networks based on the gated recurrent unit

Leaky Units : accumulate information over a long duration

Gated RNNs : learn to decide when to forget the old state

### 10.10.1 LSTM

#### Long Short-term Memory (LSTM)

Internal recurrence: Self-loops to produce paths where the gradient can flow for long durations  
 on top of outer recurrence of RNN: Weight on this self-loop is gated (controlled by another hidden unit)  
 and conditioned on context (rather than fixed)

✓ time scale of integration can be changed dynamically  
 based on the input sequences

\* LSTM cell internal state

$$S_i^{(t)} = f_i^{(t)} S_i^{(t-1)} + g_i^{(t)} \delta(b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} h_j^{(t-1)})$$

forget gate      external input gate      bias      input weights      recurrent weights

\* Output

$$h_i^{(t)} = \tanh(S_i^{(t)}) g_i^{(t)} \text{ Output gate}$$

\* Forget gate \* Crucial

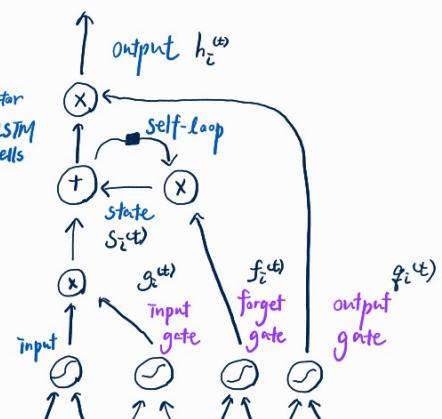
$$f_i^{(t)} = \delta(b_i^f + \sum_j U_{ifj}^f x_j^{(t)} + \sum_j W_{ifj}^f h_j^{(t-1)})$$

\* External Input gate

$$g_i^{(t)} = \delta(b_i^g + \sum_j U_{igj}^g x_j^{(t)} + \sum_j W_{igj}^g h_j^{(t-1)})$$

\* Output gate

$$g_i^{(t)} = \delta(b_i^o + \sum_j U_{ijo}^o x_j^{(t)} + \sum_j W_{ijo}^o h_j^{(t-1)})$$



### 10.10.2 Other Gated RNNs

#### Gated Recurrent Units (GRUs)

A single gating unit simultaneously controls the forgetting vector  
 and the decision to update the state unit

$$h_i^{(t)} = \underbrace{u_i^{(t-1)} h_i^{(t-1)}}_{\text{update gate}} + (1 - \underbrace{u_i^{(t-1)}}_{\text{reset gate}}) \delta \left( b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} r_j^{(t-1)} h_j^{(t-1)} \right)$$

Act like **conditional leaky integrators** → **linear gate**

Control which parts of the state get used to compute the next target state  
 → additional non-linear effect between past & future state

**\* Update gate**  
 $u_i^{(t)} = \delta \left( b_i + \sum_j U_{ij}^{(t)} x_j^{(t)} + \sum_j W_{ij}^{(t)} h_j^{(t-1)} \right)$

**\* Reset gate**  
 $r_i^{(t)} = \delta \left( b_i^{(t)} + \sum_j U_{ij}^{(t)} x_j^{(t)} + \sum_j W_{ij}^{(t)} h_j^{(t-1)} \right)$

## 10.11 Optimization for Long-Term Dependencies

Idea: Second-order optimization algo  $\approx \frac{\text{first derivative}}{\text{second derivative}}$   
 If keep second-order derivative  
 shrinks at a similar rate to first-order derivative  
 → then the ratio remains constant (tackle vanishing/exploding gradient)

✗ second-order methods:

- high computational cost
- need large minibatch
- tendency to be attracted to saddle points

→ largely replaced by simply using SGD applied to LSTMs

### 10.11.1 Clipping Gradients

Avoid a large parameter gradient throw the parameters very far (exploding gradient)

Clipping the gradient just before the parameter update:

- ① Clip the parameter gradient from a minibatch element-wise
- ② Clip the norm  $\|\vec{g}\|$  of the gradient  $\vec{g}$  ✓ gradient of all parameters  
 If  $\|\vec{g}\| > r$ ,  $\vec{g} \leftarrow \frac{\vec{g}}{\|\vec{g}\|} \cdot r$  is renormalized jointly with a single scaling factor,
- ③ Simply take a random step when the gradient magnitude is above a threshold guarantee each step still in the gradient direction

### 10.11.2 Regularizing to Encourage Information Flow

Idea: make the gradient vector  $\nabla_{h^{(t)}} L$  being back-propagated to maintain its magnitude

i.e.  $(\nabla_{h^{(t)}} L) \frac{\partial h^{(t)}}{\partial h^{(t-1)}}$  as large as  $\nabla_{h^{(t)}} L$

## 10.12 Explicit Memory

NN excels at storing implicit knowledge ("how a cat looks different from a dog") but struggles to memorize facts ("a cat is a kind of animal")

Memory Networks requires a supervision signal on how to use memory cells

Neural Turing Machine: content-based soft attention mechanism

(NTM) network outputs an internal state that chooses which cells to I/O

Attention Mechanism  $\approx$  Read: weighted average of cells  
 Write: modify multiple cells by different amount

# Chapter 11 Practical Methodology

## 11.1 Performance Metrics

Precision, Recall, PR curve, F-score, Coverage

## 11.2 Default Baseline Models

Problem complexity

solve by a few linear weights  $\rightarrow$  simple statistical model e.g. Logistic Regression  
AI - complete category  $\rightarrow$  Deep Learning models

Structure of data

- \* fixed-size vectors as input  $\rightarrow$  feedforward network with fully connected layers
- \* unknown topological structure input (e.g. images)  $\rightarrow$  convolutional network
  - piecewise linear units (ReLUs, Leaky ReLUs, PReLU, Maxout)
- \* Sequences for input/output  $\rightarrow$  gated recurrent nets (LSTM or GRU)

Optimization Algorithm

SGD with momentum with a decaying learning rate

Adam

Batch normalization drastic effect esp. for CNN  
networks with sigmoidal nonlinearities

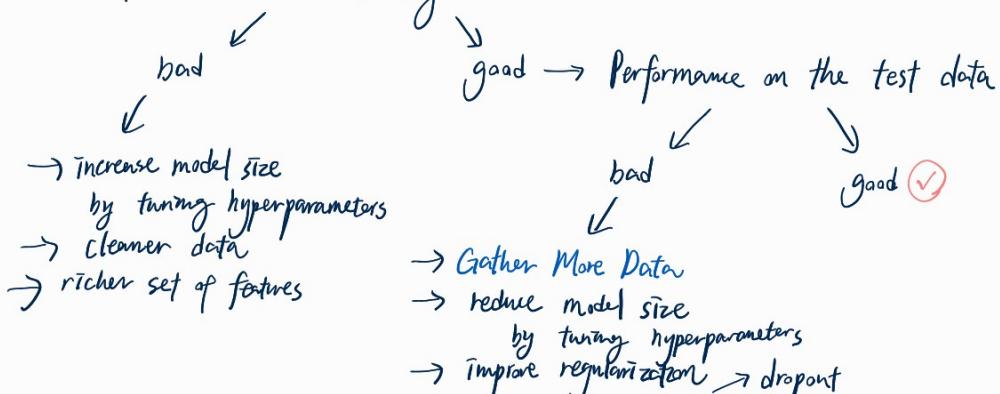
Regularization

Early stopping

Batch normalization

## 11.3 Determine whether to Gather More Data

Performance on the training data:



## 11.4 Selecting Hyperparameters

### 11.4.1 Manual Hyperparameter Tuning

Primary Goal: adjust the effective capacity of the model  
to match the complexity of the task

Effective Capacity constrained by

- ① representational capacity of the model

- ② ability of the learning algo to minimize the cost function  
③ the degree to which the cost function & training procedure regularize the model

### \* Most important = Learning Rate

Others =

# hidden units  
width of convolutional kernel  
implicit zero padding  
weight decay coefficient  
dropout rate

↑↑  
↑↑  
↑↑  
↓↓  
↓↓

High Training Error → increase model capacity  
High Test Error → reduce model capacity

help increase  
model capacity

## 11.4.2 Automatic Hyperparameters Optimization Algorithms

### Hyperparameter Optimization Algorithms

wrap a learning algorithm and chooses its parameters

### 11.4.3 Grid Search

Select a small finite set of values to explore for each hyperparameter  
(usually of log scale)

### 11.4.4 Random Search

define a marginal distribution per hyperparameter

e.g. Bernoulli or multinoulli for binary/discrete hyperparameters

Uniform distribution on a log-scale for positive real-valued hyperparameters

e.g.  $\log\text{-learning-rate} \sim u(-1, -5)$

$\text{learning-rate} = 10^{\log\text{-learning-rate}}$

$\log\text{-num-of-hidden-units} \sim u(\log(50), \log(5000))$

! should not discretize / bin the values of the hyperparameters

✓ simple to program

✓ more convenient to use

✓ converges much faster to good values due to no wasted runs compared with grid search

### 11.4.5 Model-Based Hyperparameter Optimization

Build a model of the validation set error

then propose new hyperparameter guesses by performing optimization within the model

e.g. Bayesian regression model to estimate the expected validation error +

Trade-off between

the uncertainty around this expectation

Exploration vs exploitation

(high uncertainty  
of large improvement)

(confident to perform  
well but similar performance)

X require the training experiment to run to a completion  $\rightarrow$  slower than manual

## 11.5 Debugging Strategies

- [ design a simple case that the correct behavior can be predicted
- [ design a test that exercises one part of the NN implementation in isolation
- Debugging tests

- \* Visualize the model in action
- \* Visualize the worst mistakes
- \* Reason about software using training and test errors
- \* Fit a tiny dataset
- \* Compare back-propagated derivatives to numerical derivatives (eg finite differences)
- \* Monitor histograms of activations and gradient
  - eg magnitude of parameter gradients to the magnitude of the parameters  
 $\checkmark 1\%$ , but not  $50\%$  or  $0.001\%$
- \* Test against guarantees of deep learning algorithms

## 11.6 Example = Multi-Digit Number Recognition

Street-view transcription system

- ① Data Collection
- ② Performance metrics + desired values
  - 98% accuracy  $\leftarrow$  sacrifices coverage
  - Goal: maximize coverage while keeping 98% accuracy.  
(eventually = 95% coverage)
- ③ Establish baseline system
  - CNN with rectified linear units
  - output layer:  $n$  different softmax units to predict  $n$  characters  
(as if classification)

- ④ Iteratively refine the baseline
  - Network refuse to predict when  $p(y|x) < t$  for some threshold  $t$
  - $\rightarrow$  initial  $p(y|x)$ : ad-hoc multiplying all softmax units
  - Improvement: compute a principled log-likelihood

- ⑤ Decide Underfitting or Overfitting
  - train set error  $\approx$  test set error  $\rightarrow$  underfitting
- ⑥ Visualize the model's worst errors
  - $\rightarrow$  cropping issues: modify cropping region size brings 10% coverage
- ⑦ Adjusting hyperparameters
  - $\rightarrow$  a few more % to the coverage

# Chapter 12 Applications

## 12.1 Large-Scale Deep Learning

Size of neural networks is critical

### 12.1.1 Fast CPU Implementations

Careful implementation of numerical computation routines can yield a large payoff.  
(e.g. fixed-point arithmetic vs floating-point arithmetic)

### 12.1.2 GPU Implementations

#### Graphics Processing Units (GPUs)

- many simple computations that do not involve much branching (if-branch)
- independent computations (may be parallelized)
- require massive buffers of memory
- ✓ high degree of parallelism
- ✓ high memory bandwidth
- ✗ low clock speed
- ✗ less branching capability

#### general purpose GPUs (GP-GPUs)

not only for gaming rendering

Software libraries: Theano, TensorFlow, Torch

### 12.1.3 Large-Scale Distributed Implementations

#### Data parallelism / Model parallelism

↓

#### Asynchronous Stochastic Gradient Descent

shares the memory representing the parameters

### 12.1.4 Model Compression

Model Compression = reduce the cost of inference end user is more resource-constrained than the developer

e.g. when the model size is only necessary because of limited training samples:  
after fitting, use the model to generate new samples,  
then train a smaller model to match on the new samples

### 12.1.5 Dynamic Structure

Conditional computation e.g. Cascade of a sequence of classifiers

Dynamic Structure examples

Decision trees

Mixture of Experts

Attention mechanism

✗ reduced degree of parallelism

### 12.1.6 Specialized Hardware Implementations of Deep Networks

ASICs (application-specific integrated circuits)

FPGAs

## 12.2 Computer Vision

### 12.2.1 Preprocessing

#### 12.2.1.1 Contrast Normalization

Contrast: magnitude of the difference between the bright and the dark pixels in an image

→ In DL, standard deviation of the pixels in an image or region of an image

##### Global Contrast Normalization (GCN)

subtract the mean from each image,

then rescale it s.t. the standard deviation across its pixels is equal to same constant  $s$

$$X'_{ijk} = s \cdot (X_{ijk} - \bar{X}) \cdot \left[ \max \left\{ \epsilon, \sqrt{\lambda + \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 (X_{ijk} - \bar{X})^2} \right\} \right]^{-1}$$

Standard deviation  
 denominator      constraint the bias  
 (regularization)      Contrast of an image

$$\text{where } \bar{X} = \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 X_{ijk} \text{ mean intensity of the entire image}$$

$X_{ijk}$  = red intensity at row  $i$ , column  $j$   
 2 green  
 3 blue

##### Spherical / Whitening

rescaling the principal components to have equal variance,  
 s.t. the multivariate normal distribution used by PCA has spherical contours

##### Local contrast normalization

GCN does not ensure that edges within the dark region stand out

Local contrast normalization → contrast is normalized across each small window rather than over the image as a whole

#### 12.2.2 Dataset Augmentation

Classifiers can benefit from random translations, rotations, flips of the input, random perturbation of the image colors, non-linear geometric distortions of the input

## 12.3 Speech Recognition

Map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker

##### Automatic Speech Recognition (ASR)

traditionally

hidden Markov models (HMMs)	sequence of phonemes
Gaussian mixture models (GMMs)	association between acoustic features and phonemes
GMM-HMM systems	

Benchmark: TIMIT corpus (39 phonemes to discriminate)

then move to deep-learning

(long short-term memory (LSTM))

## 12.4 Natural Language Processing

### 12.4.1 n-grams

Language model = defines a probability distribution over sequence of tokens in a natural language

Models based on n-grams = define the conditional probability of the n-th token given the preceding n-1 tokens

$$P(x_1, \dots, x_t) = \underbrace{P(x_1, \dots, x_{n-1})}_{\text{preceding } n-1 \text{ tokens}} \prod_{t=1}^n P(x_t | x_{t-n+1}, \dots, x_{t-1})$$

unigram, bigram, trigram  
 $n=1$        $n=2$        $n=3$

✓ maximum likelihood estimate can be computed

Simply by counting how many times each possible n-gram occurs in the training set

✓ core building blocks for many decades

✗  $P_n$  estimated from training set counts can be zero

even though  $(x_{t-n+1}, \dots, x_t)$  may appear in the test set

Fundamental  
Limitation

→ Smoothing (assign non-zero values to all possible next words)

Back-off methods (mixture model with high-order & low-order n-gram models, look-up the lower-order n-grams if the frequency is too low)

✗ Curse of dimensionality:  $|V|^n$  possible n-grams and  $|V|$  is large

→ Class-based language models (clustering to form word categories,

✗ much info can be lost use word-classes instead of individual words to represent the context on the right side of the conditioning bar)

### 12.4.2 Neural Language Models

#### Neural Language Models (NLM)

use distributed representation of words to overcome the curse of dimensionality

Word embeddings: word representations

### 12.4.3 High-Dimensional Outputs

Naive approach for distribution representation:

apply an affine transformation from a hidden representation to the output space, then apply the softmax function

$$a_i = b_i + \sum_j w_j h_j \quad \forall i = \{1, \dots, |V|\}$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_k e^{a_k}} \quad \text{top hidden layer}$$

$h_h$ : thousands  
 $|V|$ : hundred of thousands  $\Rightarrow$  dominates the computation of most NLMs

## 12.9.21 Use of a Smart List

## Splitting the vocabulary ✓

into a *Shortlist*  $L_s$  of most frequent words  $\Rightarrow$  neural net  
 and a tail  $T = V \setminus L_s$  of more rare words  $\Rightarrow$   $n$ -gram model  
 Combine the two predictions

where neural nets also predict the probability that a word appearing after context C belongs to the tail list

$$P(y=i|c) = \sum_{i \in L} P(y=i|c, i \in L) (1 - P(i \in T|c)) + \sum_{i \in L} P(y=i|c, i \in T) P(i \in T|c)$$

provided by NLM  
 provided by n-grams

- X NLM only limit to most frequent words  
→ least useful

## 12.4.3.8 Hierarchical Softmax

decompose probabilities hierarchically = from  $|V|$  to  $\log |V|$   
by forming hierarchy of word categories

probability of a word = summation over all the paths that lead to the word

- ✓ Computation benefits in both train & test time
  - ✗ worst result than sampling-based methods

## 12.4.3.3 Importance Sampling

Avoid explicitly computing the contribution of the gradient from all the words that do not appear in the next position

→ Sample only a subset of the words

## Gradient

$$\begin{aligned}
 \frac{\partial \log P(y|c)}{\partial \alpha} &= \frac{\partial \log \text{softmax}_y(\frac{1}{\alpha})}{\partial \alpha} \\
 &= \frac{\partial}{\partial \alpha} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \\
 &= \frac{\partial}{\partial \alpha} (a_y - \log \sum_i e^{a_i}) \\
 &= \underbrace{\frac{\partial a_y}{\partial \alpha}}_{\substack{\text{positive} \\ \text{phase}}} - \underbrace{\sum_i P(y=i|c) \frac{\partial a_i}{\partial \alpha}}_{\substack{\text{negative} \\ \text{phase}}} \text{ pushing } a_i \text{ down via}
 \end{aligned}$$

## Biased Importance Sampling

Approximate the negative phase term

$$\sum_{i=1}^{|V|} P(z_i | c) \frac{\partial a_z}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^m w_i \frac{\partial a_{z_i}}{\partial \theta} \quad \text{where} \quad w_i = \frac{p_{n_i}}{\frac{q_{n_i}}{f_{n_i}}}$$

far a negative word  
n<sub>i</sub> when sampled

## 18.4.3.4 Noise-Contrastive Estimation and Ranking Loss

Early example: view the NLM output of each word as a score  
tries to make the score of the correct word  $a_y$   
to be ranked higher than the other scores  $a_i$

Ranking loss:  $L = \sum_i \max(0, 1 - a_y + a_i)$  gradient is 0 when  
 $a_y > a_i$  by a margin of 1  
X not providing estimated conditional probabilities

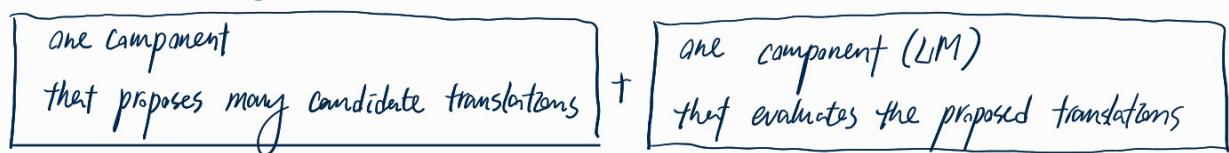
Recent example: Noise-contrastive estimation

#### 12.4.4 Combining Neural Language Models with n-grams

Add model capacity by ensembling NLM and n-gram language model

#### 12.4.5 Neural Machine Translation

Translation System:



Conditional n-gram models

MLP estimates  $P(t_1, t_2, \dots, t_k | s_1, s_2, \dots, s_n)$   
phrase  $t$  in target language   phrase  $s$  in source language  
X require fixed length of the preprocessed sentences

Encoder-Decoder RNNs

✓ variable input/output length

#### 12.4.5.1 Using an Attention Mechanism and Aligning Pieces of Data

Attention mechanism

read the whole sentence/paragraph,  
produce each word one at a time, each time focusing on a different part of the input  
to gather the semantic details required to produce the next output word.

Attention-based systems

- ① A process that reads raw data and converts them into distributed representations
- ② A list of vector vectors as "memory" to store the output of the reader
- ③ A process that exploits the content of the memory,  
being able to put attention on the content of one/a few memory element

✓ based on weighted average  $\rightarrow$  smooth, differentiable approximation essentially a weighted average  
that can be trained with existing optimization algorithms

## 12.4.6 Historical Perspective

1986: distributed representation

1990: an embedding for a word

1991~1996 = input as a sequence of characters

2001: NLM

2011: character-based models

2015: individual bytes of Unicode characters  
NLP applications

## 12.5 Other Applications

### 12.5.1 Recommender Systems

Major applications

① online advertising > rely on predicting the association

② item recommendations > between a user and an item

probabilistic classification → probability of some actions (buy the product)

regression + expected gain (depend on product value)

#### Collaborative Filtering

if user 1 & 2 like items A, B and C and user 1 also likes item D

⇒ user 2 also likely likes item D

non-parametric approaches: nearest neighbor

parametric approaches:

often rely on learning a distributed representation for each user and item

bilinear prediction

dot product between user embedding and the item embedding

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{uj} B_{j,i} \rightarrow \text{minimize MSE}(\hat{R}_{u,i}, R_{u,i})$$

Matrix of predictions    bias for user    bias for item    user embeddings    item embeddings

obtain user, item embeddings:

factorize  $R = \begin{matrix} U \\ A \end{matrix} \begin{matrix} D \\ V \end{matrix}'$  by SVD

B

✓ perform very well in Netflix prizes (2007)

a component of the ensemble models by most of the competitors

| Based on restricted Boltzmann machines (RBMs)

| with the use of neural networks

X cold-start recommendation problem (new users/items)

#### Content-based Recommender Systems

introduce extra info about the users / items

### 12.5.1.1 Exploration versus Exploitation

Ordinary supervised learning → Reinforcement Learning  
↓ issue

Contextual bandits

an action is taken in the context of some input variables  
that can inform the decision

when use recommendation system to  
collect the data

→ biased & incomplete view  
of the user preferences

→ no info about the outcome  
if showing other items

→ only the reward for the  
selected actions is observed

→ special case:

learner only takes a  
single action and receives a  
single reward

(ordinary supervised learning : ←  
no trade-off,  
the supervised signal always specify)

Exploration vs Exploitation

↓

taking actions  
specifically for more  
training data

↓

taking actions  
that come from the  
currently best learned policy (with high reward)

prominent factor: time-scale

long time to  
acquire reward

vs short time to  
acquire reward

## 12.5.2 Knowledge Representation, Reasoning and Question Answering

### 12.5.2.1 Knowledge, Relations and Question Answering

Determine how distributed representations can be trained  
to capture relations between entities

e.g. sentence (subject, verb, object) (subject, adjective)

↓

(entity<sub>i</sub>, relation<sub>j</sub>, entity<sub>k</sub>) (entity<sub>i</sub>, attribute<sub>j</sub>)

Knowledge Base (a form of relations database)

e.g. WordNet, Wikibase, Freebase, OpenCyc

Research: Link Prediction

Word-sense disambiguation

Question-Answering system