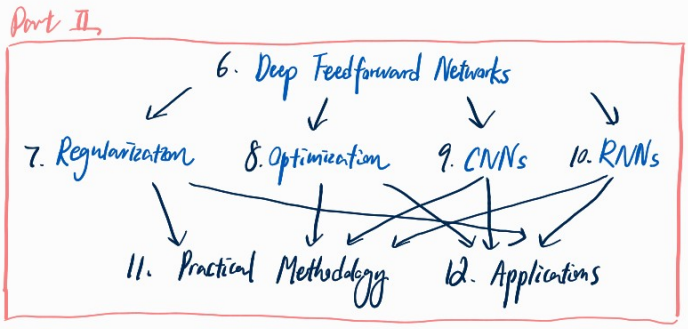
Author:
kiking

Part II: established DL algorithm



Chapter 6 Deep Feedforward Networks

Deep Feedforward Network
<>
Feedforward Neural Network
<>
Multilayer Perceptrons (MLP)

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

Feedforward → No feedback connections from output into itself
otherwise Recurrent Neural Network

Network → composing many functions with a DAG

Depth of the model → Number of layers (length of the chain of functions)
Width of the model → Dimensionality of the hidden layers (typically vector-valued)
↳ 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 $\phi(x)$ where ϕ nonlinear

Choosing ϕ:

① Generic ϕ: e.g. infinite-dimensional ϕ like RBF kernel
✓ high-dim ϕ → enough capacity to fit
✗ generalization to test set is poor
✗ Very generic ϕ → usually based only on the principle of local smoothness
✗ not encoding enough prior

② Manually engineer ϕ: ✗ huge human efforts with specialized practitioners
✗ little transfer between domains

③ Deep Learning: learn the ϕ in the Kernel trick in Linear Models

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

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

Author:
kiking

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 $\phi(\vec{x};\theta)$

→ ✓ human practioners can encode their knowledge to help generalization
by designing families $\phi(\vec{x};\theta)$ 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 θ to make f as similar as f^*

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

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

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

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

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

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

→ simply outputs 0.5 everywhere ✗

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) + a fixed (non-linear) activation function

scalar to scalar

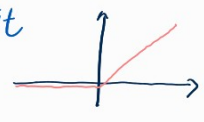
$$\vec{h} = g(W^T\vec{x} + \vec{c})$$

typically an element-wise activation function

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

typical Rectified Linear Unit

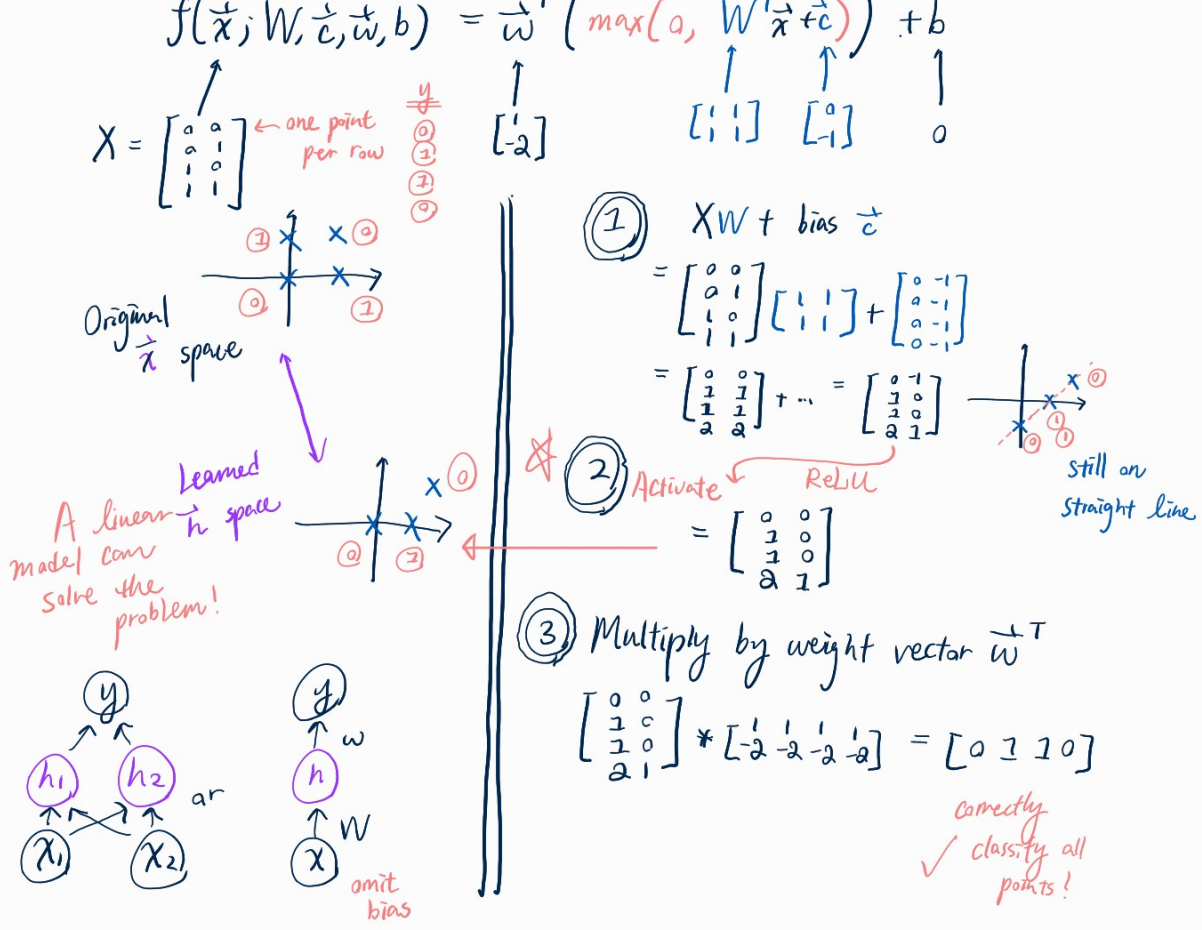
$g(z) = \max(0,z)$



Complete Network:

$\vec{h}$

Author:
kiking



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, rather than convex optimization algorithms with global convergence guarantees (converges starting from any initial parameters) (e.g. logistic regression or SVMs) or linear equation solvers

→ Stochastic gradient descent on nonconvex loss functions → no such convergence guarantees → sensitive to initial parameters

! Important: Feed-forward Network → init weights to small random values, bias to 0 / small positive numbers

6.2.1 Cost Functions

Most cases: parametric model defines a distribution $p(\vec{y} \mid \vec{x}; \theta)$ 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

Author:
kiking

(+ weight decay approach)

6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Cost function → negative likelihood / cross entropy

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

true for any model (not only linear model)

when predict mean of Gaussian (typically can discard some terms that do not depend on the model parameters and can discard)

e.g. when $p_{model}(\vec{y}|\vec{x}) = \mathcal{N}(\vec{y}; f(\vec{x};\theta), I)$,

then recover $J(\theta)$ = Mean Squared Error cost + const

depends only on variance 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 $\log p(\vec{y}|\vec{x})$ 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. predictor $f(\vec{y};\theta)$ to predict the mean of $\vec{y}$

View the cost function as a *functional* rather than just a function

⇒ learning to choose a function, instead of choosing a set of parameters

⇒ e.g. can design the cost functional 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 result

① $f^* = \arg\min_f \mathbb{E}_{\vec{x},\vec{y}\sim p_{data}} \|\vec{y} - f(\vec{x})\|^2 = \mathbb{E}_{\vec{y}\sim p_{data}(\vec{y}|\vec{x})}[\vec{y}]$

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 $\vec{x}$

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

Mean Absolute Error cost function

✗ Both poor result when using gradient-based optimization

Author:
kiking

(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

how to represent output

the form of the cross-entropy function ← determines

6.2.2.1 Linear Units for Gaussian Output Distributions

Linear Units: given features $\vec{h}$,
a layer of linear output units produces $\vec{y} = W^T\vec{h} + b$

Often used:

produce the mean of a conditional Gaussian distribution

Maximum likelihood → minimizing the MSE

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

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

Sigmoid Output Unit: $\vec{y} = \sigma(\vec{w}^T\vec{h} + b)$

activation ← σ; linear layer output $\vec{z}$

Sigmoid $\sigma(\vec{x}) = \frac{\exp(\vec{x})}{1+\exp(\vec{x})}$

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)

$J(\vec{\theta}) = -\log P(\vec{y}|\vec{x})$

the log in the cost function undoes the exp of the sigmoid

$= \zeta((1-2y)z)$ where z is the logit

Softplus $\zeta(\vec{x}) = \log(1+\exp(\vec{x}))$

Otherwise saturation of sigmoid ✗ → good gradient

⇒ Saturates only when $(1-2y)z$ is very negative

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

In the limit of extremely correct z, the softplus does not shrink the gradient

✓ Gradient-based learning can act quickly

i: Sigmoid σ exclusively used in Maximum Likihood with a log Loss function, otherwise other losses easily saturates when σ saturates

6.2.2.3 Softmax Units for Multinoulli Output Distributions

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

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

Often used:

Represent a probability distribution over a 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

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

Author:
kiking

the exp in Softmax

input z_i always have an direct contribution to the cost function → won't saturate

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

⇒ Saturates when the difference between input values became extreme

"Winner-take-all"

e.g. Numerically stable variant

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

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 ω, the parameters of a distribution over $\vec{y}$ and
loss function is $-\log p(\vec{y}|\omega(\vec{x}))$

Examples:

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

Multimodal regression → Gaussian mixtures in Neural Network
→ Mixture density networks
effective in generative models of speech / movement of physical objects

6.3 Hidden Units

- Requires Trial-and-Error
- Can safely disregard the non-differentiability of the hidden unit activation functions

ReLU

during implementation, return one-side limit // numerical error so $\varepsilon \neq 0$

6.3.1 Rectified Linear Units and Their Generalizations

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

✓ similar to linear units → easy to optimize
✓ derivative remain large when it is active
✓ second order derivative is 0 → no second-order effects

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

✗ cannot learn via gradient-based methods on examples for which their activation is 0

generalizations of ReLU guarantees non-zero

① when $z_i < 0$ =

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

Absolute value rectification

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

e.g. object recognition
look for features that are invariant

Author:
kiking

Leaky ReLU α_i very small e.g. 0.01 — under a polarity reversal of the input illumination

Parametric ReLU or PReLU α_i a learnable parameter

(2) Maxout units Generalize further

$g(\vec{z})_i = \max_{j \in G^{(i)}} z_j$ ⟹ divides $\vec{z}$ into k groups of values, each unit output 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 learn) 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 $g(z) = \sigma(z)$

Hyperbolic Tangent activation $g(z) = \tanh(z)$

} closely related $\tanh(z) = 2\sigma(2z) - 1$

↳ 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)

$h_i = \exp\left(-\frac{1}{\sigma_i^2}\|W_{:,i} - \vec{x}\|^2\right)$ — active when $\vec{x} \to W_{:,i}$, saturates to 0 for most $\vec{x}$, difficult to optimize

→ Softplus

$g(a) = \zeta(a) = \log(1 + \exp(a))$ — smooth version of Rectifier, differentiable everywhere, empirically may not working better than ReLU

→ Hard tanh

$g(a) = \max(-1, \min(1, a))$ — unlike the rectifier, it is bounded

6.4 Architecture Design

1st layer:

Author:
kiking

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

2nd layer: $\vec{h}^{(2)} = g^{(2)}(W^{(2)T}\vec{h}^{(1)} + b^{(2)})$

and so on

Main consideration:
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 at 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

Author:
kiking

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

$n \times m$ Jacobian matrix

$$\nabla_{\vec{x}} z = \left(\frac{\partial \vec{y}}{\partial \vec{x}}\right)^T \nabla_{\vec{y}} z \qquad \in \mathbb{R}^m \quad \in \mathbb{R}^n$$

Tensor Notation

$$\nabla_X z = \sum_j (\nabla_X Y_j)\frac{\partial z}{\partial Y_j}$$

6.5.3 Recursively Applying the Chain Rule to obtain Backprop

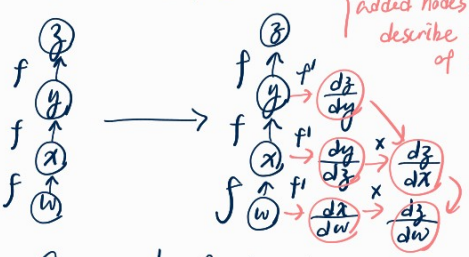
6.5.4 Back-Propagation Computation in Fully Connected MLP

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



Algorithm does not need to access any actual specific numerical values

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 and add to this buffer until completed)

→ handle various data types
e.g. 16-bit, 32-bit floats, integers

→ undefined gradients

6.5.9 Differentiation outside the Deep Learning Community

The field of Automatic Differentiation: e.g. Reverse mode accumulation (backprop is only one approach)

determine the order of evaluation 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 too many to even represent, parameters)

Krylov Methods

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

Author:
kiking

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: same 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 (high = underfitting) for reduced variance (change in resampling; high = overfitting)

Three Situations of the model family being trained:

① excluded the true data-generating process

Author:
kiking

underfitting, high bias

② matched the true data-generating process

③ included the generating process but also many others — variance dominates the estimation error

Regularization: more from ③ to ②

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

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

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

Typically Ω 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

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

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)

$$\hat{J}(\theta) = J(\vec{w}^*) + \frac{1}{2}(\vec{w} - \vec{w}^*)^T H (\vec{w} - \vec{w}^*)$$

H: Hessian of J w.r.t. $\vec{w}$ evaluated at $\vec{w}^*$

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

Adding the weight decay gradient:

$$\alpha\tilde{w} + H(\tilde{w} - \vec{w}^*) = 0$$

$\tilde{w}$: denote the minimum

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

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

H real and symmetric, decompose into $H = Q\Lambda Q^T$

Author:
kiking

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:

$$\vec{w} = (X^TX + \alpha I)^{-1}X^T\vec{y}$$

X^TX: diagonal entries = covariance matrix

Effect:

the learning algo "perceive" 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

7.1.2 L^1 Regularization

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

Example in Linear Regression

Total Objective function:

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

Subgradient:

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

Effect:

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

Quadratic Approximation of the Objective:

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

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

Analytical Solution:

Author:
kiking

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

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

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

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

Effect:

Solution that is more sparse

Compared to L2 regularization

$\tilde{w}_i = \frac{H_{i,i}}{H_{i,i}+\alpha} w_i^* \rightarrow \tilde{w}_i$ is non-zero when w_i 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 isotopic 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 $\Omega(\theta)$ to be less than k:

construct a generalized Lagrange function:

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

solution:

$$\theta^* = \arg\min_\theta \max_{\alpha, \alpha \ge 0} L(\theta, \alpha)$$

Effect of the constraint:

fix α^*, view the problem as a function of θ:

$$\theta^* = \arg\min_\theta L(\theta, \alpha^*)$$

$$= \arg\min_\theta J(\theta; X, \vec{y}) + \alpha^* \Omega(\theta)$$

exactly the same as the regularized training of minimizing $\tilde{J}$

Can view Parameter Norm Penalty → constraints on the weights

Ω is L^2 norm → weights constrained to lie in an L^2 ball

Author:
kiking

L^2 is L norm → weights constrained to lie in an L^2 ball

larger α → 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 θ back to the nearest point $\Omega(\theta) < k$
when we know the approx. k

✓ penalties can cause non-convex optimization procedures
to get stuck in local minima corresponding to small θ

so explicit constraints by reprojection work better

✓ impose stability on the optimization procedure
e.g. reprojection prevents numerical overflow
for some positive feedback loop that keeps increasing weights
when the learning rate is high
e.g. constraint the norm of each column of the matrix (instead of the Frobenius norm)

7.3 Regularization and Under-Constrained Problems

Many linear models and PCA:
depends on $(X^TX)^{-1}$ but not possible if X^TX singular
e.g. data-generating distribution has no variance in some direction or when no variance is observed e.g. $m<n$

Regularized matrix $X^TX + \alpha I$
is guaranteed to be invertible

↳ then have closed-form solutions

Guarantee 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 stabilizing underdetermined linear equations with regularization

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

→ 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

Author:
kiking

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 preprocessing steps
eg cropping image

7.5 Noise Robustness

For some 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-square cost function

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

Random perturbation added to the network weights

$$\varepsilon_w \sim \mathcal{N}(\varepsilon; \vec{0}, \eta I)$$

Objective function (perturbed model)

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

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

For small η, equivalent to having an additional regularization term :

$$\eta\, \mathbb{E}_{p(\vec{x},y)}\left[\|\nabla_w \hat{y}(\vec{x})\|^2\right]$$

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

7.5.1 Injecting Noise at the Output Targets

Author:
kiking

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 $1-\varepsilon$
✓ 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 $\frac{\varepsilon}{k-1}$ and $1-\varepsilon$ 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 $\vec{h} = f(\vec{x})$
Goal = learn a representation s.t. examples from the same class
have similar representations
then a linear classifier
in the new space might achieve better generalization

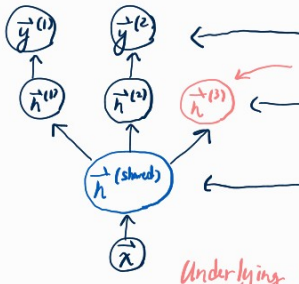
E.g. PCA as a preprocessing step before applying a classifier on the projected data

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

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

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)



different supervised tasks (predict $\vec{y}^{(i)}$ given shared $\vec{x}$)
not associate with the prediction task, only explain input variation
task-specific parameters
generic parameters shared across all tasks
share some intermediate-level representation $h^{(shared)}$

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

Author:
kiking

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 → 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
→ 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 θ_0

7.9 Parameter Tying and Parameter Sharing

Another way to express our prior knowledge:
there are some dependencies between the model parameters
e.g. the parameters should be close to each other
→ add parameter norm penalty $\Omega(w^{(A)}, w^{(B)}) = \|w^{(A)} - w^{(B)}\|_2^2$
→ 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: most popular and extensive use of parameter sharing
e.g. images → invariant to translation
CNN → share parameters across multiple image locations
i.e. can find a cat with the same cat detector
whether the cat appear at column i or $i+1$ 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 $\vec{h}$ (many elements of the representation are zero)
Representational regularization
same sorts of mechanisms in parameters regularization
e.g. Norm

Author:
kiking

L^1 -norm penalty regularization of representations

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

L^1 penalty on representation → representational sparsity

$$\Omega(\vec{h}) = \|\vec{h}\|_1 = \sum_i |h_i|$$

Orthogonal Matching Pursuit (OMP-k)

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

$$\underset{\vec{h}}{\operatorname{argmin}} \|\vec{x} - W\vec{h}\|_2^2 \quad \text{subject to } \|\vec{h}\|_0 \le k$$

$\|\vec{h}\|_0$: # nonzero entries of h; $W\vec{h}$: 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)

same 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 ε_i on each example, with errors drawn from a zero-mean multivariate normal distribution

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

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:

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \varepsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\varepsilon_i^2 + \sum_{j\neq i}\varepsilon_i\varepsilon_j\right)\right]$$

$$= \frac{1}{k}v + \frac{k-1}{k}c$$

But if errors are perfectly uncorrelated, $c = 0 \Rightarrow$ expected squared error is only $\frac{1}{k}v \Rightarrow$ inversely proportional to the ensembled size

if errors are perfectly correlated, $c = v \Rightarrow$ expected squared error is v, not helping

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 for regularization

Boosting constructs an ensemble with higher model capacity

Author:
kiking

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 → 0.8
hidden unit → 0.5
④ Run forward propagation, back-propagation, and learning updates

Dropout compared to Bagging
Bagging: (a) models are all independent, (b) each model converges to its respective training set
Dropout: (a) models share parameters,
each model inherit a different subsets of parameters from the parent network
(b) 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 → reduces the effective capacity of a model

Author:
kiking

to offset → 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 ensembled 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 solutions to the noise robustness problem
(eg. make h_i very large so noise ε_i insignificant ← 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
Some built overall functions are highly linear
→ easy to optimize, but the value of a linear function can change very rapidly
e.g. if each input changed by ε, then a linear function with weights $\vec{w}$ can change by as much as $\varepsilon\|\vec{w}\|_1$ → large when $\vec{w}$ high dimensional

Adversarial training discourages by encouraging the network to be locally constant in the neighborhood of the training data
→ explicitly introduce a local constancy prior

Purely linear models eg. logistic regression cannot resist adversarial examples
Neural Network can represent functions from nearly linear to nearly local constant
⇒ 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

Author:
kiking

↳ 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

Overcome the curse of dimensionality:

manifold hypothesis → 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

→ 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 a priori)

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

$$\Omega(f) = \sum_i \left((\nabla_{\vec{x}} f(\vec{x}))^T \vec{v}^{(i)} \right)^2$$

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

Tangent Prob 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

Author:
kiking

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 regularizes 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 θ 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:

$$J^*(\theta) = \mathbb{E}_{(\vec{x},y)\sim p_{data}} L(f(\vec{x};\theta), y)$$

($f(\vec{x};\theta)$: predicted output; L: per-example loss function; p_{data}: data-generating distribution)

8.1.1 Empirical Risk Minimization

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

Risk optimization: an optimization solvable by optimization algo
- if we knew p_{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 $\hat{p}(\vec{x}, y)$

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

Author:
kiking

⇒ Empirical Risk Minimization

✗ prone to overfitting (remember the training set)

✗ 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 as a surrogate for the 0-1 loss

Usually halts when reaching the early-stopping criterion → that is based on the true underlying loss function, e.g. 0-1 loss on validation set

even if gradient is still large

→ unlike pure optimization that only considered to be converged when gradient is small

8.1.3 Batch and Minibatch Algorithms

Maximum likelihood estimation problems when viewed in log space:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(\vec{x}^{(i)}, y^{(i)}, \theta)$$

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

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

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

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

✗ 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

Author:
kiking

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

8.2 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

8.2.1 Ill-Conditioning

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

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

$\frac{1}{2}\varepsilon^2\vec{g}^T H\vec{g}$: if this exceeds $\varepsilon\vec{g}^T\vec{g}$ → ill-conditioning becomes a problem
$\varepsilon\vec{g}^T\vec{g}$: squared gradient norm

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

Test Rule: monitor gradient norm

Author:
kiking

8.2.2 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 its 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, (that jumps too far)
the gradient clipping heuristics 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 → W^t
Suppose W has an eigendecomposition $W = V \operatorname{diag}(\lambda) V^{-1}$

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

any eigenvalues $\lambda_i \neq 1$ will either explode or vanish
⇒ The vanishing and exploding gradient problem
RNN reuses 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:

Author:
kiking

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

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

ε_k: learning rate at iteration k

Common: decay linearly until iteration τ

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

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

ε_τ: 1% of ε_0

ε_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

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

Author:
kiking

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
→ 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 high curvature, small but consistant curvatures or noisy gradient
accumulates an exponentially decaying
moving average of past gradients and continues to move in their direction

Update Rule:

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

$\vec{v}$: "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

$\alpha \in [0,1)$ hyperparameter
how quickly the past contributions exponentially decay

Size of each step:
depends on how large and how aligned a sequence of gradients are

Common α: 0.5, 0.9, 0.99 (can be adaptive, start with small then raised)
less important than shrinking the learning rate ε

8.3.3 Nesterov Momentum

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

Update Rule:

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

gradient is evaluated after the current velocity is applied

For Batch GD,
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

Author:
kiking

drawn from a Gaussian or Uniform distribution

The scale of the init distribution → large effect on the outcome + generalization ability

→ larger init weight:
- ✓ stronger symmetry-breaking effect
- X might result in exploding values
- X extreme values causing activation functions to saturate

→ Optimization perspective suggests larger weight,
Regularization perspective encourages making smaller
e.g. early stopping → expresses a prior that the final parameters should be close to the initial parameters

→ Heuristics: a fully connected layer with m inputs and n outputs

Sample each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$ $U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}})$ Normalized Initialization

Good 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 mini-batch of data

if unacceptably small activations → 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 → 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 are axis aligned, it can make sense to use a separate learning rate for each parameter and automatically adapt these learning rates

Delta-bar-delta: early heuristic approach → if the partial derivative of the loss w.r.t the model parameter
- remains the same sign → increase the learning rate
- change the sign → decrease the learning rate

can only be applied on full batch optimization

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:

$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \vec{g}$ δ for numerical stability, small $\approx 10^{-7}$

Apply update:

8.5.2 RMSProp

Accumulate squared gradient:

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

Compute update:

$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \vec{g}$ decay rate ρ

Apply update:

Author:
kiking

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

Adapt the learning rates of all model parameters θ by scaling them inversely prop. to $\sqrt{\Sigma_i}$ historical squared values of the gradient

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

✓ convex optimization → desirable theoretical properties

✗ training deep neural network →
accumulation of squared gradient from the beginning can result in a premature/excessive decrease in the effective learning rate
→ only perform well for some but not all deep learning problems

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

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

$\vec{s} \leftarrow \rho_1 \vec{s} + (1-\rho_1)\vec{g}$ $\quad \vec{r} \leftarrow \rho_2 \vec{r} + (1-\rho_2)\vec{g} \odot \vec{g}$

$\hat{s} \leftarrow \frac{\vec{s}}{1-\rho_1^t}$ $\quad \hat{r} \leftarrow \frac{\vec{r}}{1-\rho_2^t}$

Compute update: $\rho_1, \rho_2 \in [0,1)$ suggests 0.9, 0.99

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

Apply update:
$\theta \leftarrow \theta + \Delta\theta$

Adam → "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 ⟷ 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 ⟷ 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:
$\vec{g} \leftarrow \frac{1}{m} \nabla_\theta \Sigma_i L(f(\vec{x}^{(i)}; \theta + \alpha\vec{v}), \vec{y}^{(i)})$

Accumulate squared gradient:
$\vec{r} \leftarrow \rho\vec{r} + (1-\rho)\vec{g} \odot \vec{g}$

Compute update:
$\vec{v} \leftarrow \alpha\vec{v} - \frac{\varepsilon}{\sqrt{r}} \odot \vec{g}$

Apply update:
$\theta \leftarrow \theta + \Delta\theta$

8.5.4 Choosing the Right Optimization Algorithm

Most popular: SGD, SGD with momentum,

Author:
kiking

RMSProp, RMSProp with momentum,
AdaDelta, Adam