Discussion of Optimization Techniques

Vahid Tarokh
ECE 685D, Fall 2025

Introduction

- **We will quickly review optimization algorithms and SGD.**
- **Discuss more why they are preferred more than batch methods**
- **Discuss mathematical analysis for (and applications to) SGD**
- **Discuss extensions/variants (RMSprop, ADAgrad, ADAM)**

- **Important Note: Source of some of my slides (with great appreciation and acknowledgements)**
  - **Professor David Carlson Slides**
  - **Professor Ruslan Salakhutdinov's slides (available online).**

Optimization Goal

- Have some model or network parameterized by $\boldsymbol{w}$
- Goal: given data, find the best $\boldsymbol{w}$
- What is the best $\boldsymbol{w}$?
  - Gives the best prediction (or other metric) on the *true* task
  - *True* task refers to future, unseen data (i.e. real-world performance)
  - Quick reminder: often estimate performance with a test set
- Many examples shown so far:
  - Image recognition
  - Object detection
  - Text classification
  - Etc.

Optimization Goal

- In optimization, this amounts to solving for $\boldsymbol{w}^*$

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \mathbb{E}_{p_{true}(\mathbf{x},y)}[\ell(h_{\boldsymbol{w}}(\mathbf{x}), y)]$$

- $\ell(\cdot,\cdot)$ it the "loss," can be defined in many ways. Some examples:
  - squared loss: $\ell(\hat{y}, y) = ||\hat{y} - y||_2^2$
  - (binary) cross-entropy loss: $\ell(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$
  - Negative Log-Likelihood: $\ell(h_{\boldsymbol{w}}(\mathbf{x}), y) = -\log p_{\boldsymbol{w}}(\boldsymbol{x}, y)$
- $h_{\boldsymbol{w}}(\mathbf{x})$ defines a transformation from the data to the output space – only function that changes when the parameters do (e.g. in logistic regression $h_{\boldsymbol{w}}(\mathbf{x}) = \sigma(\boldsymbol{w}^T\mathbf{x})$)
- $p_{true}(\mathbf{x}, y)$ is the probability distribution over data (unknown!)

Empirical Risk Minimization

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \mathbb{E}_{p_{true}(\mathbf{x},y)}[\ell(h_{\boldsymbol{w}}(\mathbf{x}), y)]$$

- Issue: The probably distribution $p_{true}(\mathbf{x}, y)$ is unknown!
- But we have N data examples $\mathcal{D} = \{\mathbf{x}_n, y_n\} \sim p_{true}(\mathbf{x}, y)$
- Will approximate the above with finite data examples (i.e. "Empirical Risk Minimization" (ERM))

$$\min_{\boldsymbol{w}} \frac{1}{N} \sum_{n=1}^{N} \ell(h_{\boldsymbol{w}}(\mathbf{x}_n), y_n)$$

  - Why does this matter?

Why does using finite data matter?

- Using empirical samples is biased, leads to overfitting

Definition (Overfitting)

Overfitting occurs when the learned parameters $\boldsymbol{w}$ capture random error or noise. This implies the parameters $\boldsymbol{w}$ are "learning" the noise rather than properties of the data. Mathematically:

$$\mathbb{E}_{p(\mathbf{x},y)}[\ell(h_{\boldsymbol{w}}(\mathbf{x}), y)] > \frac{1}{N} \sum_{n=1}^{N} \ell(h_{\boldsymbol{w}}(\mathbf{x}_n), y_n)$$

or,

"generalization error" > "training error"

Consequences for Iterative Optimizers

- No benefit to exact optimization, only need "**moderate**" accuracy as fast as possible
- *modus operandi* in big data: use stochastic iterative methods
  - Less information per iteration, but many many more iterations in the same amount of time

- We assume $\boldsymbol{w} \in \mathbb{R}^D$, but can easily consider constrained set (most important thing is dimensionality D)
- The above optimization goal is often rewritten for simplicity:

$$\min_{\boldsymbol{w}} \frac{1}{N} \sum_{n=1}^{N} f_n(\boldsymbol{w}), \quad f_n(\boldsymbol{w}) = \ell(h_{\boldsymbol{w}}(\mathbf{x}_n), y_n)$$

Binary Logistic Regression

- A canonical model for classification is logistic regression:

$$P(y = 1|\mathbf{x}) = \sigma(\boldsymbol{w}^T\mathbf{x}), \quad \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Why is this a good example for optimization?
  - This gives our logistic loss function
  - Can simply derive constants used in convergence analysis (allowing theorems to have *precise* values)

Introduction and Intuition of Stochastic Gradients

(Stochastic) Gradient Descent:

Gradient Descent (GD)

With step size α_k, use updates:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \left[\frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\boldsymbol{w}_k) \right]$$

Stochastic Gradient Descent (SGD)

With step size α_k and random index i_k, use updates:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \left[\nabla f_{i_k}(\boldsymbol{w}_k) \right]$$

Note that $\mathbb{E}_{p(i_k)} \left[\nabla f_{i_k}(\boldsymbol{w}_k) \right] = \left[\frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\boldsymbol{w}_k) \right]$ (unbiased). (Underlying i.i.d assumption in our dataset, $\mathcal{D}$)

How expensive are these iterative algorithms?

- Gradient descent takes:
  - $\mathcal{O}(N)$ to estimate gradients
  - $\mathcal{O}(1)$ to update the parameters
- Stochastic gradient descent takes:
  - $\mathcal{O}(1)$ to estimate gradients
  - $\mathcal{O}(1)$ to update the parameters
- What about Newton's method (classical optimization approach)?
  - $\mathcal{O}(N)$ to estimate gradients and $\mathcal{O}(ND^2)$ to estimate Hessian
  - $\mathcal{O}(D^3)$ to update the parameters
- Consider implication for GoogLeNet image classifier on the ImageNet dataset, with $N = 10^6$ and $D \simeq 5 \times 10^6$

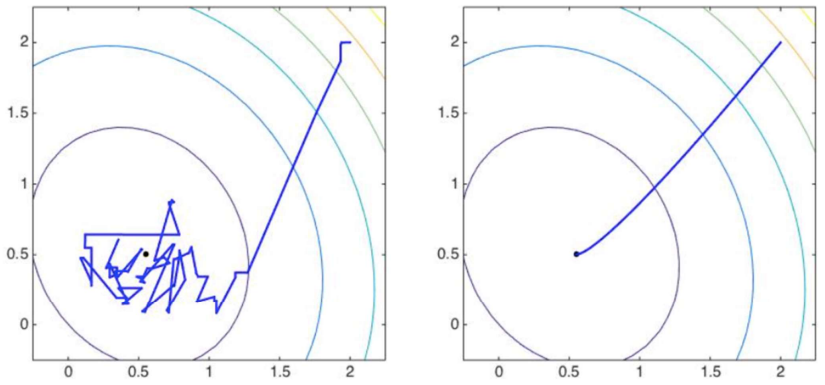


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Batch gradient descent.

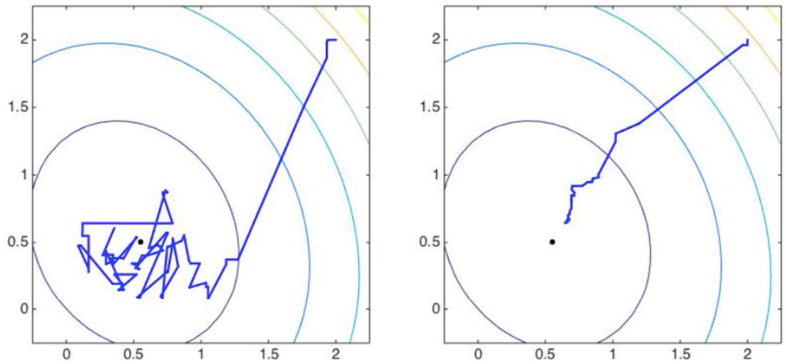


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Stochastic gradient descent with diminishing step size.

When will Stochastic Gradient Descent work?

Necessary Definitions for Convergence Analysis

Definition (Lipschitz-continuous functions)

Given an objective function $F : \mathbb{R}^D \to \mathbb{R}$, the function is Lipschitz continuous if

$$|F(y) - F(x)| \leq L_0 ||\boldsymbol{y} - \boldsymbol{x}||_2$$

Necessary Definitions for Convergence Analysis

Definition (Lipschitz-continuous objective gradients)

Given an objective function $F : \mathbb{R}^D \to \mathbb{R}$ is continuously differentiable, the gradient of F is Lipschitz continuous with Lipschitz constant $L > 0$ if

$$||\nabla F(\boldsymbol{y}) - \nabla F(\boldsymbol{x})||_2 \leq L||\boldsymbol{y} - \boldsymbol{x}||_2.$$

For continuous second gradients, note $L \geq \max_{\boldsymbol{x}} ||\nabla^2 F(\boldsymbol{x})||_{S_\infty}$

- The matrix $|| \cdot ||_{S_\infty}$ norm is defined as the largest singular value of the matrix (also known as the spectral norm, the Schatten-∞ norm, and the matrix 2-norm):

$$||\mathbf{A}||_{S_\infty} = \max_{||\boldsymbol{x}||_2 \leq 1} ||\mathbf{A}\boldsymbol{x}||_2$$

Lipschitz Gradient for Logistic Regression

- Remember the cost function in logistic regression:

$$\min_{\boldsymbol{w}} F(\boldsymbol{w}) = \min_{\boldsymbol{w}} \frac{1}{N} \sum_{n=1}^{N} -y_n \log(\sigma(\boldsymbol{w}^T \mathbf{x})) - (1 - y_n) \log(1 - \sigma(\boldsymbol{w}^T \mathbf{x}))$$

- Hessian is given by

$$\nabla^2 F(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} \left(\sigma(\boldsymbol{w}^T \mathbf{x}_n)(1 - \sigma(\boldsymbol{w}^T \mathbf{x}_n)) \right) \mathbf{x}_n \mathbf{x}_n^T$$

Lipschitz Gradient for Logistic Regression

- First term is bound by:

$$0 \leq \left(\sigma(\boldsymbol{w}^T \mathbf{x})(1 - \sigma(\boldsymbol{w}^T \mathbf{x})) \right) \leq \frac{1}{4}$$

- Linear algebra allows us to state:

$$\mathbf{0} \preceq \nabla^2 F(\boldsymbol{w}) \preceq \frac{1}{4N} \sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^T$$

($\mathbf{A} \preceq \mathbf{B}$ means that $\mathbf{B} - \mathbf{A}$ is a positive semidefinite matrix, e.g. $\boldsymbol{x}^T \mathbf{A} \boldsymbol{x} \leq \boldsymbol{x}^T \mathbf{B} \boldsymbol{x}$ for any $\boldsymbol{x}$ if matrices are symmetric)

- A simple bound is $L \boxed{\geq} \frac{1}{4} \mathbb{E}_{\mathcal{D}}[||\mathbf{x}||_2^2]$

Convexity

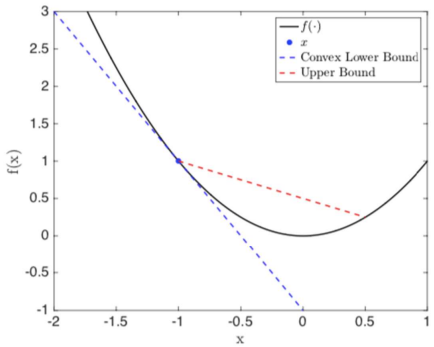
- A convex function has two definitions:

$\forall \mathbf{x},\ \mathbf{y} \in \mathcal{X}, t \in [0, 1]$

$f(t\mathbf{y} + (1 - t)\mathbf{x}) \leq tf(\mathbf{y}) + (1 - t)f(\mathbf{x})$

$f(\mathbf{y}) \geq f(\mathbf{x}) + (\nabla f(\mathbf{x}))^T (\mathbf{y} - \mathbf{x})$

- First definition is more general, gradient doesn't always exist



Strong Convexity

Definition (Strong Convexity:)

An objective function $F : \mathbb{R}^d \to \mathbb{R}$ is strongly convex in that there is a constant $c > 0$ such that

$$F(\boldsymbol{y}) \geq F(\boldsymbol{x}) + \nabla F(\boldsymbol{x})^T (\boldsymbol{y} - \boldsymbol{x}) + \frac{1}{2} c ||\boldsymbol{y} - \boldsymbol{x}||_2^2$$

Then F will have a unique minimizer $\boldsymbol{x}_*$ with function value F_*. If only $c = 0$ holds, then the function is convex.

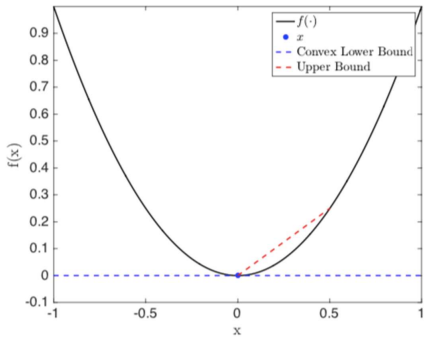
- This definition covers a significant subset of machine learning problems (e.g. penalized logistic regression, etc.)
- Does *not* cover models such as LDA, PFA, RBM, SBN, MLP, CNN, RNN, etc.
  - Gives clear theoretical results, and is very useful for intuitions in these problems

Implications

- Convex function implies a local minima is a global minima
- The gradient at the optima (if gradient exists)

$$\nabla f(\mathbf{x}) = \mathbf{0}$$

- Most state-of-the-art machine learning models are not convex
- However, very useful for intuition – nonconvex functions are often convex around a local minima



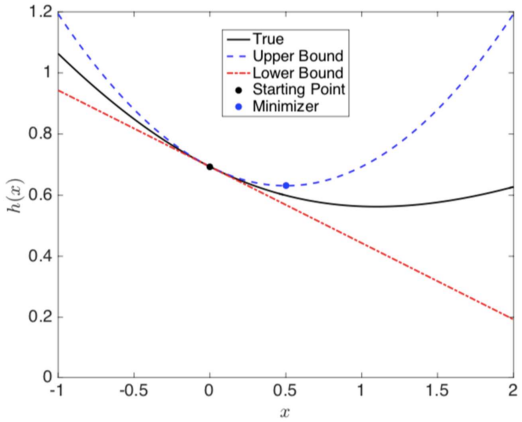


Figure: Examples of the upper bound on logistic loss. The Lipschitz gradient provides a conservative upper bound on the function value.

Consequences for Gradient Methods

Lemma (Decreasing sequence for gradient descent)

Consider a function F with Lipschitz gradient with constant L. For the deterministic (i.e. batch) gradient descent method with iterates

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha \nabla F(\boldsymbol{w}_k),$$

then the Lipschitz gradient upper bound with this sequence yields

$$\begin{aligned} F(\boldsymbol{w}_{k+1}) &\leq F(\boldsymbol{w}_k) - [\nabla F(\boldsymbol{x})]^T(\alpha \nabla F(\boldsymbol{w}_k)) + \frac{1}{2}L||\alpha \nabla F(\boldsymbol{w}_k)||_2^2, \\ &\leq F(\boldsymbol{w}_k) - \alpha||\nabla F(\boldsymbol{w}_k)||_2^2 + \frac{\alpha^2 L}{2}||\nabla F(\boldsymbol{w}_k)||_2^2, \\ &\leq F(\boldsymbol{w}_k) + (\frac{L\alpha^2}{2} - \alpha)||\nabla F(\boldsymbol{w}_k)||_2^2. \end{aligned}$$

"Optimal" Step Size

Lemma (Optimal step size for gradient descent)

Consider the upper bound for gradient descent

$$F(\boldsymbol{w}_{k+1}) \leq F(\boldsymbol{w}_k) + (\frac{L\alpha^2}{2} - \alpha)||\nabla F(\boldsymbol{w}_k)||_2^2.$$

The RHS (optimal guaranteed improvement) is minimized at $\alpha = \frac{1}{L}$,

$$F(\boldsymbol{w}_{k+1}) \leq F(\boldsymbol{w}_k) - \frac{1}{2L}||\nabla F(\boldsymbol{w}_k)||_2^2.$$

What are the implications of the upper bound?

- If the gradient is nonzero, the next iteration will have a lower function value (sequence is non-increasing)
- If the function has a minimum F^*, will converge to a fixed point (i.e. $\nabla F(\boldsymbol{w}) = \mathbf{0}$)

Implications

Algorithm (Minibatch Gradient Estimator)

Inputs: Data $\{\boldsymbol{x}_n, y_n\}_{n=1,\ldots,N}$, *minibatch size B, parameters* $\boldsymbol{w}_k$
Sample B minibatch indicies $\{i_1, \ldots, i_B\}$
Return gradient estimate: $\tilde{\mathbf{g}}_k \leftarrow \frac{1}{B}\sum_{m=1}^{B} \nabla f_{i_m}(\boldsymbol{w}_k)$

Lemma (Sequence for stochastic gradient)

Consider a function F with Lipschitz gradient with constant L. For the minibatch gradient descent method with iterates

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \tilde{g}_k,$$

then the sequence satisfies in expectation

$$\mathbb{E}[F(\boldsymbol{w}_{k+1})] - F(\boldsymbol{w}_k) \leq -\alpha_k [\nabla F(\boldsymbol{w}_k)]^T \mathbb{E}[\tilde{g}_k] + \frac{\alpha_k^2 L}{2} \mathbb{E}[||\tilde{g}_k||_2^2]$$

Consequences for an unbiased estimator

- First, we consider an unbiased gradient estimator (i.e $\mathbb{E}[\tilde{g}_k] = g_k$)
- Assume the variance is bounded (i.e. $\text{var}(||\tilde{g}_k||_2) \leq M$)
- Then the previous lemma reveals that

$$\mathbb{E}[F(\boldsymbol{w}_{k+1})] - F(\boldsymbol{w}_k) \leq \left(\frac{\alpha_k^2 L}{2} - \alpha_k\right) ||g_k||_2^2 + \frac{\alpha_k^2 LM}{2}$$

- *Only* difference to deterministic case is the term $\frac{\alpha_k^2 LM}{2}$
- Question: When is a gradient step expected to improve our cost function?

Consequences for an unbiased estimator

- Consider a step size $\alpha_k = \frac{1}{L}$

$$\mathbb{E}[F(\boldsymbol{w}_{k+1})] - F(\boldsymbol{w}_k) \leq \frac{1}{2L}(M - ||g_k||_2^2)$$

- Whether the function is improved depends on the quantity $M - ||g_k||_2^2$
  - Often, M is fairly constant, but $||g_k||_2^2 \to 0$ at the optimum
- Smaller step sizes allow for further optimization
  - Smaller step sizes take (much) longer if not necessary!

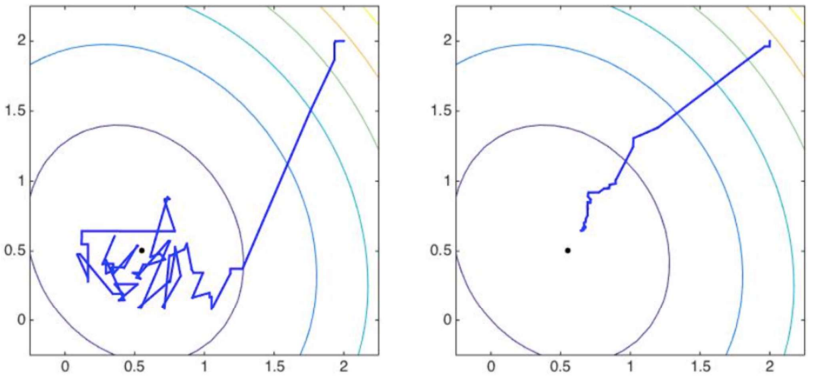


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Stochastic gradient descent with diminishing step size.

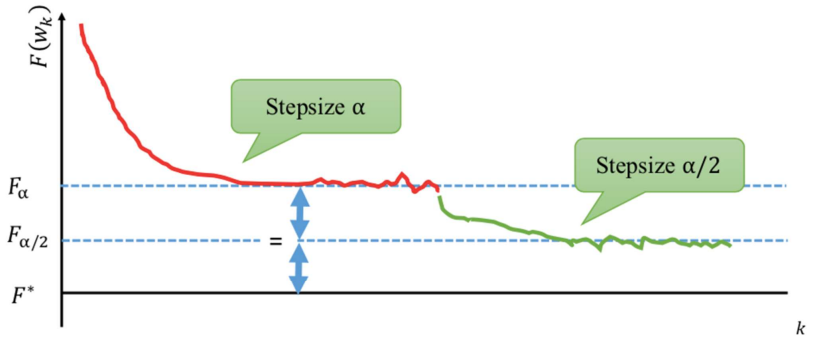


Figure: Reducing the step size allows the objective to reach closer to the optimal value.

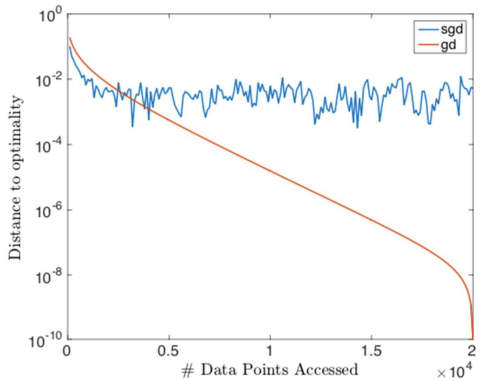


Figure: In the beginning, SGD has an exponential decay, but on small datasets GD will catch up and pass SGD.

Summary/Recap

- Rigorous analysis depends on constants that can be derived for many standard models
  - Hard to derive for deep neural networks, but intuition is important for understanding how these methods work
- The variance in the gradient estimator limits how well the function can be optimized
- Reducing the step size *or* increasing the minibatch size allows the optimization algorithm to reach closer to the optimum

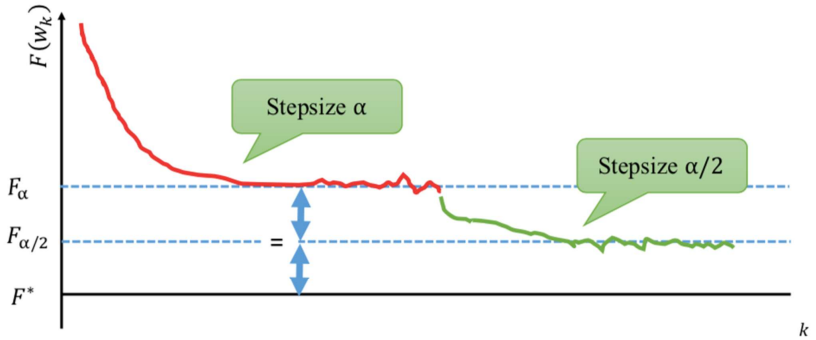


Figure: Reducing the step size allows the objective to reach closer to the optimal value.

SGD with diminishing step sizes

Algorithm (Minibatch Gradient Descent)

Inputs: $\beta, \gamma, \kappa, \boldsymbol{w}_0$
Initialize: $k \leftarrow 0$
for k=0,... ***do***
  Estimate gradient: $\tilde{\mathbf{g}}_k$
  Calculate step size: $\alpha_k = \beta(\gamma + k)^{-\kappa}$
  Update parameters: $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \tilde{\mathbf{g}}_k$
end for

Theorem (Convergence of minibatch gradient descent)

For $\kappa = 1$, $\beta \geq \frac{1}{c}$ *and* $\gamma \geq \beta L$, *the expected optimality gap satisfies*

$$\mathbb{E}[F(\boldsymbol{w}_k) - F_*] \leq \frac{\nu}{\gamma + k}, \quad \nu = \max\left\{(\gamma+1)(F(\boldsymbol{w}_1) - F_*), \frac{\beta^2 LM}{2(\beta c - 1)}\right\}$$

- F is assumed to be a strongly convex function with constant c and Lipsitz gradient with constant L

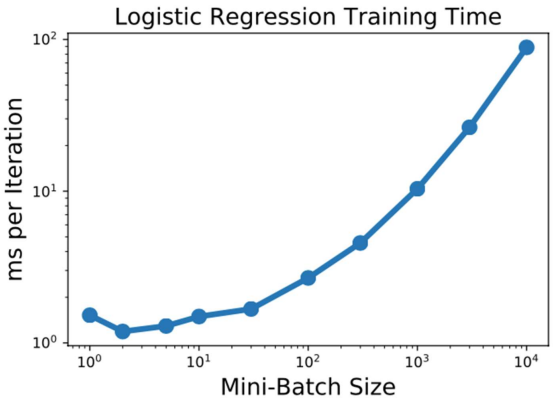
Thoughts and conclusions up to this point in time

- Practically, SGD can dominate batch methods to reasonable accuracy
- The limitations of the stochastic gradient method are:
  - Constants (and hence step sizes) are usually unknown
  - Variance in the gradient estimation limits convergence (much of the recent literature and our forthcoming discussion will discuss ways of minimizing variance)
  - Does not utilize the curvature in space, can be very poorly conditioned.
- So far, we assumed the gradient estimates were unbiased
  - Often gradients depend on a sequential procedure (RBMs, PFA, variational methods, recurrent neural nets) – gradients are not necessarily unbiased

Improving Accuracy

- One of the fundamental quantities that control the convergence of stochastic gradient is the variance M
- Complex approach: develop algorithmic methods to reduce variance (included in slides, will not get to today)
- Simple approach: increase minibatch size. Typically $M \propto 1/(\text{minibatch size})$
- Problem: Time to estimate the gradient is $\propto$ (minibatch size)
- No theoretical benefit to using minibatch over a single example – in fact, theoretical convergence rate is typically worse.

Empirical Minibatch Timing



Nestrov Acceleration Methods

Nesterov's Accelerated Gradient Descent:

- First define the following sequences:

$$\lambda_0 = 0\,, \qquad \lambda_k = \frac{1 + \sqrt{1 + 4\lambda_{k-1}{}^2}}{2}, \qquad \gamma_k = \frac{1 - \lambda_k}{\lambda_{k+1}}$$

- For $k = 1\ldots$ do (for some initial point $w_1 = t_1$):

$$t_{k+1} = w_k - \frac{1}{\beta}\nabla F(w_k),$$
$$w_{k+1} = (1 - \gamma_k)t_{k+1} + \gamma_k t_k,$$

- The update involves computing of gradient in time steps k and $k + 1$ (using momentum)

Nesterov's Accelerated Gradient Descent:

- Nesterov methods invented originally by Yurii Nesterov is used for acceleration of the first order methods.
- Remember that function F with **only Lipchitz gradient** assumption has sublinear rate $\mathcal{O}(\frac{1}{k})$ convergence.
- Nesterov method establishes a better rate like $\mathcal{O}(\frac{1}{k^2})$.
- In addition to the gradient information from ***the previous iteration***, gradients from other iteration(s) are contributed with appropriate weights to the calculation of the current estimate of the parameter.

Theorem (Nesterov 1983). ***Let F be a convex and β-smooth function, then the Nesterov's Accelerated Gradient Descent satisfies for all $k > 1$:***

$$F(w_k) - F(w_*) \leq \frac{2\beta||w_1 - w_*||_2^2}{k^2}$$

Improving Stochastic Methods with Momentum (Acceleration methods)

Including Momentum in SGD

Algorithm (Minibatch Gradient Descent with Momentum)

Inputs: α, β
Initialize: $k \leftarrow 0, \mathbf{m}_k$
for *k=0,...* ***do***
  Estimate gradient: $\tilde{\mathbf{g}}_k$
  Update with momentum: $\mathbf{m}_{k+1} \leftarrow \beta \mathbf{m}_k + \tilde{\mathbf{g}}_k$
  Update parameters: $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha \mathbf{m}_{k+1}$
end for

- Classically, momentum "dampens" the highly oscillatory terms
- Analysis reduces dependency of $\frac{L}{c}$ (a worst-case condition number of the Hessian) to $\sqrt{\frac{L}{c}}$ for a strongly convex model.

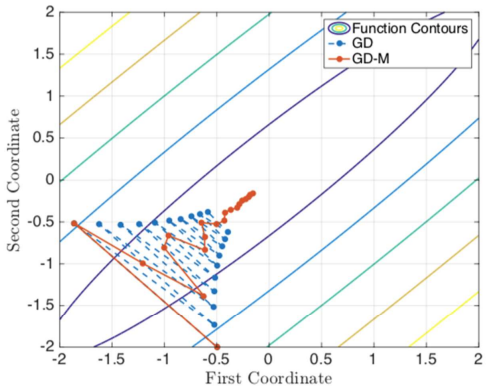


Figure: Effect of using momentum for a skewed 2D Gaussian. Gradient descent bounces back and forth, but using momentum averages out the first dimension and finds the correct path.

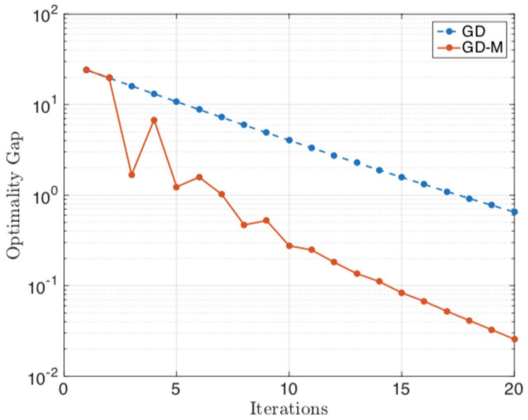


Figure: Effect of using momentum for a skewed 2D Gaussian in the previous figure. Using momentum greatly improves the convergence speed.

Two Viewpoints on Momentum

- Define momentum updates as exponential smoothing

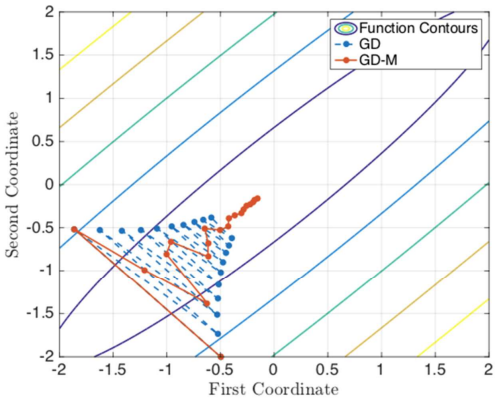
$$\tilde{\mathbf{g}} = (1-\beta)\sum_{i=0}^{k}\beta^i \tilde{\mathbf{g}}_{k-i}.$$

- Note: $\sum_{i=0}^{\infty}\beta^i = (1-\beta)^{-1}$
- Then SGD with momentum is implemented with the update

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \tilde{\mathbf{g}}.$$

- This can be viewed as:
  - A filter that dampens out transient patterns
  - A bias-variance tradeoff (decrease variance for an small bias) on noisy gradient estimates

Dampening of Transient Patterns



Momentum as De-noising Gradient Estimates

- Examining the term

$$\tilde{\mathbf{g}} = (1-\beta)\sum_{i=0}^{}\beta^i \tilde{\mathbf{g}}_{k-i}.$$

- If $\mathbb{E}[\tilde{\mathbf{g}}_k] = \mathbf{g}_k$ and $\text{var}(||\mathbf{g}_k||_2) \leq M$, then as $k \to \infty$

$$\text{var}(||\tilde{\mathbf{g}}_k||_2) = (1-\beta)^2 \frac{1}{1-\beta^2} M = \frac{1-\beta}{1+\beta} M.$$

- Decreases variance by a multiplicative factor of $\frac{1-\beta}{1+\beta}$
- Downside: introduces bias (i.e. $\mathbb{E}[\tilde{\mathbf{g}}_k] \neq \mathbf{g}_k$)

Higher Order Methods

Higher Order Methods

- SGD only uses gradient (first-order) information
- Sometimes the *curvature* is drastically different depending on the direction
  - Recall the momentum example!
- Can change the direction based on curvature
- step size tuning can pose problems, can use curvature to estimate an appropriate step size

Second Order Methods

Definition (Second order approximation)

Consider a second order expansion:

$$\begin{aligned} q_k(\boldsymbol{w}) =& F(\boldsymbol{w}_k) + \nabla F(\boldsymbol{w}_k)^T(\boldsymbol{w} - \boldsymbol{w}_k) \\ &+ \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_k)^T \hat{B}^{-1}(\boldsymbol{w} - \boldsymbol{w}_k). \end{aligned}$$

Move $\boldsymbol{w}$ in the direction that minimizes q_k:

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \hat{B} \nabla F(\boldsymbol{w}_k).$$

Note: $\alpha_k = 1$ will minimize $q_k(\boldsymbol{w})$.

Newton's Methods

Algorithm (Newton's method)

Let $\hat{B}^{-2} = \nabla^2 F(\boldsymbol{w}_k)$ (i.e. the Hessian). Then

$$\begin{aligned} \boldsymbol{w}_{k+1} &= \arg\min_{\boldsymbol{w}} q_k(\boldsymbol{w}) \\ &= \boldsymbol{w}_k - [\nabla^2 F(\boldsymbol{w}_k)]^{-1} \nabla F(\boldsymbol{w}_k) \end{aligned}$$

If certain conditions are satisfied, this approach converges with quadratic convergence $\mathcal{O}(\rho^{k^2})$.

- Convergence is great, but many issues:
  - Gradient typically costs $\mathcal{O}(ND)$
  - Forming the Hessian typically costs $\mathcal{O}(ND^2)$
  - Applying the Hessian to the gradient typically costs $\mathcal{O}(D^3)$
  - Often assumptions are not satisfied, may diverge!
- Can often get a "good enough" solution from SGD by the time Newton runs a single iteration for big data and big models

Practical Approaches

- A large neural network may have $>>$ 1 million parameters
  - Neither feasible to calculate Hessian, nor apply it
- Still want to be able to use curvature information
- Common approach is to use a diagonal approximation

$$\begin{aligned} q_k(\boldsymbol{w}) =& F(\boldsymbol{w}_k) + \nabla F(\boldsymbol{w}_k)^T(\boldsymbol{w} - \boldsymbol{w}_k) \\ &+ \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_k)^T \text{diag}(\mathbf{b})^{-1}(\boldsymbol{w} - \boldsymbol{w}_k) \end{aligned}$$

with the minimizer with $\alpha_k = 1$ as

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \mathbf{b} \odot \nabla F(\boldsymbol{w}_k)$$

- Several approaches to developing $\mathbf{b}$

Consequences of diagonal approximations

- The follow algorithms *do not* improve the convergence rate
- They can improve optimizations constants (provably on regret)
  - The constants may be dramatically improved
- Biggest reason to use the following methods is their *robustness* to settings
  - SGD methods are very sensitive to the step size sequences
  - Many of the so-called "adaptive metric" methods can use the same settings across many models and datasets

Adagrad Algorithm

Algorithm (ADAgrad)

Inputs: $\epsilon, \gamma, \boldsymbol{w}_0$
Initialize: $k \leftarrow 0$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for *k=0,...* ***do***
  Estimate gradient: $\tilde{\mathbf{g}}_k$
  Update sum-of-squares: $\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{v}_k + \tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k$
  Calculate element-wise step sizes: $\alpha_k = \gamma \mathbf{1} \oslash \left(\epsilon + \sqrt{\boldsymbol{v}_{k+1}}\right)$
  Update parameters: $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \odot \tilde{\mathbf{g}}_k$
end for

- Key idea: sparsely occurring features may be *very* informative – only decrease step size when information relating to that parameter is seen
- *Provably* improves regret bounds compared to online (stochastic) gradient descent

RMSprop Algorithm

Algorithm (RMSprop)

Inputs: $\epsilon, \gamma, \beta, \boldsymbol{w}_0, \alpha_k$
Initialize: $k \leftarrow 0, \boldsymbol{v}_0 \leftarrow \mathbf{0}$
for *k=0,...* ***do***
  Estimate gradient: $\tilde{\mathbf{g}}_k$
  Update sum-of-squares: $\boldsymbol{v}_{k+1} \leftarrow (1-\beta)\boldsymbol{v}_k + \beta(\tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k)$
  Calculate element-wise preconditioner: $\mathbf{b}_k = \gamma \mathbf{1} \oslash \left(\epsilon + \sqrt{\boldsymbol{v}_{k+1}}\right)$
  Update parameters: $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \mathbf{b}_k \odot \tilde{\mathbf{g}}_k$
end for

- Will converge if $\alpha \to 0$ with at rate $t^{\frac{1}{2}}$
- Typically a constant step size is used $\alpha_k = \bar{\alpha} \simeq 10^{-3}$
- In practice, has been largely replaced the ADAM optimizer

- Comments on Adagrad

- ADAgrad is a great algorithm when features are sparse
- Step sizes can diminish much too quickly (no forgetting–once an element-wise step size goes small, it stays small)
- Intuitive to add a "forgetting" term, especially in deep learning

- Comments on RMSprop

- Ad-hoc: originally no convergence guarantee
- No inclusion of momentum
- Adaptive Moments (ADAM) addressed these issues
- RMSprop had significant practical successes, but is less used now

ADAM Algorithm

Algorithm (ADAM)

Inputs: $\alpha_{1,\ldots,T}, \beta_1, \beta_2, \boldsymbol{w}_0$

Initialize: $k \leftarrow 0, \mathbf{m}_0^{(1)} \leftarrow \mathbf{0}, \mathbf{m}_0^{(2)} \leftarrow \mathbf{0}$

for *k=0,...* ***do***

Estimate gradient: $\tilde{\mathbf{g}}_k$

Update first moment: $\mathbf{m}_{k+1}^{(1)} \leftarrow \beta_1 \mathbf{m}_k^{(1)} + (1-\beta_1)\tilde{\mathbf{g}}_k$

"Debias" first moment $\tilde{\mathbf{m}}_{k+1}^{(1)} \leftarrow \mathbf{m}_{k+1}^{(1)} (1-\beta_1^{(k+1)})^{-1}$

Update second moment: $\mathbf{m}_{k+1}^{(2)} \leftarrow \beta_2 \mathbf{m}_k^{(2)} + (1-\beta_2)\tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k$

"Debias" second moment $\tilde{\mathbf{m}}_{k+1}^{(2)} \leftarrow \mathbf{m}_{k+1}^{(2)} (1-\beta_2^{(k+1)})^{-1}$

Update parameters: $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_t(\tilde{\mathbf{m}}_{k+1}^{(1)}) \oslash \left(\epsilon + \sqrt{\tilde{\mathbf{m}}_{k+1}^{(2)}}\right)$

end for

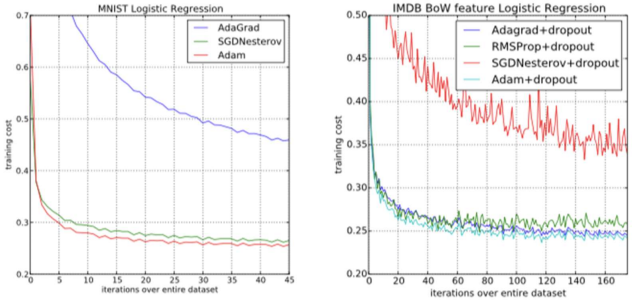


Figure: Comparison of several learning rules on large-scale logistic regression.

Analysis of ADAM Algorithm

- RMSprop inspired ADAM – if $\beta_1 = 0$ then they are very similar algorithms
- Provable regret bound under a decreasing step size
- Can typically use "standard" parameters for many problems, e.g.

$$\alpha_k = \bar{\alpha} = 10^{-3},\ \beta_1 = .9,\ \beta_2 = .999$$

- Reminder: β_1 going higher reduces variance of gradients, but can add bias

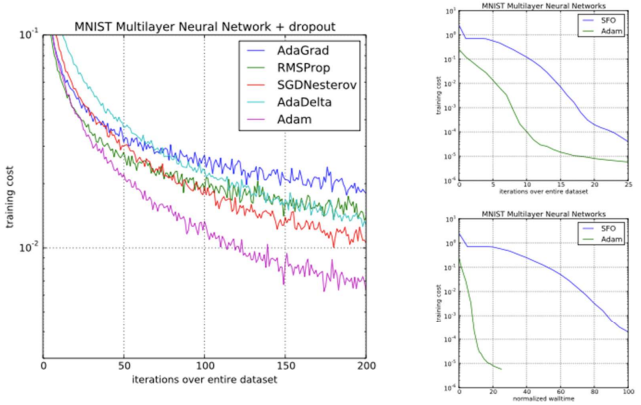


Figure: Comparison of several learning rules on deep neural networks.

Conclusions

- Stochastic gradient descent will converge to the optimum if given enough time
  - Will converge to a fixed point in nonconvex problems as well
- Adding in adaptive higher order information can reduce parameter tuning and account for curvature information
- Stochastic gradient methods are effective at quickly obtaining a reasonable solution
- Machines are learned with stochastic gradient methods
- Many theoretical issues but many practical successes
- Still a very active area of research!
- Many tuning parameters within these algorithms
  - Hopefully their meaning and how to set them is clearer now

Convolutional Neural Networks and Applications to Object Classification

Vahid Tarokh

ECE685D, Fall 2025

Introduction

•We next focus on convolutional neural networks. These have been extremely successful in image classification algorithms.

•Important Note: Source of some of my slides (with great appreciation and acknowledgements):

- **Dive into Deep Learning**
- **Professor David Carlson's Slides**
- **Professor Hugo Larochelle's slides**
- **Professor Ruslan Salakhutdinov's slides (available online)**

•Some tutorial slides were borrowed from Rob Fergus
https://sites.google.com/site/deeplearningsummerschool2016/speakers

•Marc'Aurelio Ranzato's CVPR 2014 tutorial on Convolutional Nets
https://sites.google.com/site/lsvrtutorialcvpr14/home/deeplearning

- **Much of the material in this lecture was borrowed from class on NN:**
  https://sites.google.com/site/deeplearningsummerschool2016

Computer Vision

• Design algorithms that can process visual data to accomplish a given task:

- For example, object recognition: Given an input image, identify which object it contains

Image Classification

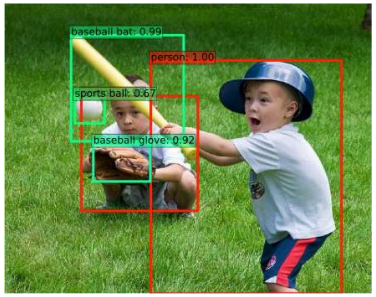
Question: “What is this an image of?”

Answer: “95% probability this is a ballplayer”

Object Detection

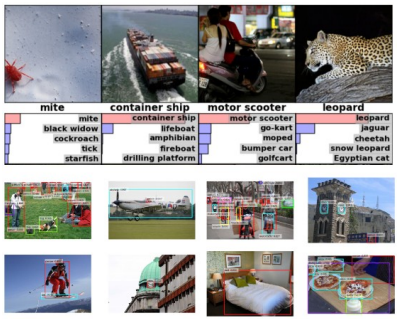
Question: “What are all the objects in this object and where are they?”

Answer:



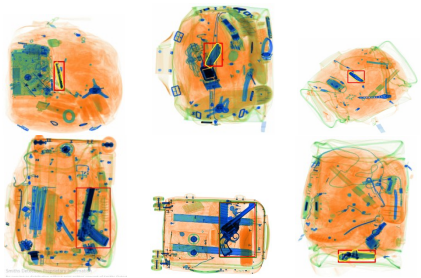
Classification versus detection

- Very related, but different problems.
- Many aspects will be shared between the two problems
- In detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image.
- There could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.



Application: Self-Driving Cars

Application: TSA



Useful Properties

We have some important properties that are useful

- Translation Invariance
- Scale Invariance
- Rotation Invariance

Some are built into the algorithm, and some come from the structure of the dataset

Convolutional Neural Networks

• Our goal is to design neural networks that are specifically adapted for such problems

- Must deal with very high-dimensional inputs: 150 x 150 pixels = 22500 inputs, or 3 x 22500 if RGB pixels
- Can exploit the 2D topology of pixels (or 3D for video data)
- Can build on invariance to certain variations: translation, illumination, etc.

• Convolutional networks leverage these ideas

- Local connectivity
- Parameter sharing
- Convolution
- Pooling / subsampling hidden units

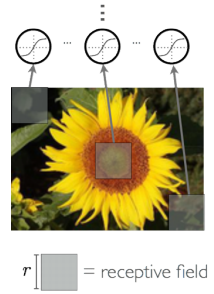
Local Connectivity

• Use local connectivity of hidden units

- Each hidden unit is connected only to a sub-region (patch) of the input image
- It is connected to all channels: 1 if grayscale, 3 (R, G, B) if color image

• Why local connectivity?

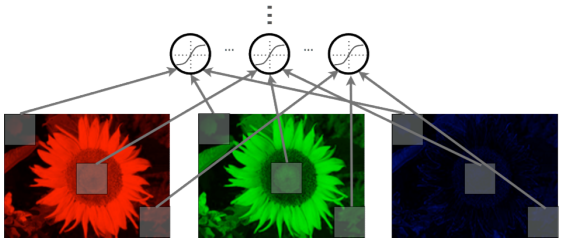
- Fully connected layer has a lot of parameters to fit, requires a lot of data
- Spatial correlation is local



Local Connectivity

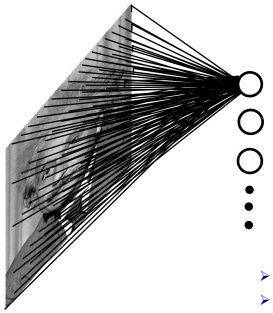
• Units are connected to all channels:

- 1 channel if grayscale image,
- 3 channels (R, G, B) if color image



Local Connectivity

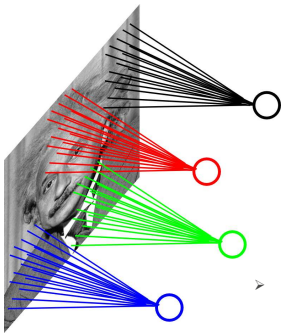
• Example: 200x200 image, 40K hidden units, ~2B parameters!



- Spatial correlation is local
- Too many parameters, will require a lot of training data!

Local Connectivity

• Example: 200x200 image, 40K hidden units, filter size 10x10, 4M parameters!



- This parameterization is good when input image is registered

Convolutional Neural Networks

• Our goal is to design neural networks that are specifically adapted for such problems

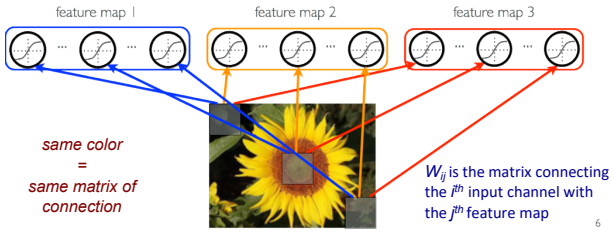
- Must deal with very high-dimensional inputs: 150 x 150 pixels = 22500 inputs, or 3 x 22500 if RGB pixels
- Can exploit the 2D topology of pixels (or 3D for video data)
- Can build in invariance to certain variations: translation, illumination, etc.

• Convolutional networks leverage these ideas

- Local connectivity
- Parameter sharing
- Convolution
- Pooling / subsampling hidden units

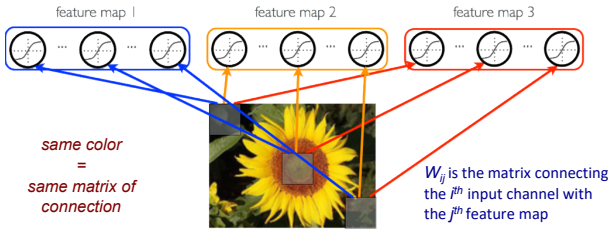
Parameter Sharing

- Share matrix of parameters across some units
  - Units that are organized into the 'feature map" share parameters
  - Hidden units within a feature map cover different positions in the image



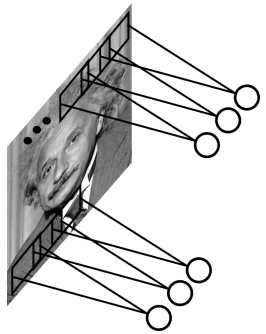
Parameter Sharing

- Why parameter sharing?
  - Reduces even more the number of parameters
  - Will extract the same features at every position (features are "equi-variant")



Parameter Sharing

- Share matrix of parameters across certain units



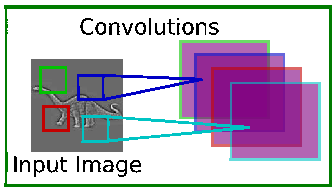
  - Convolutions with certain kernels

Computer Vision

- Our goal is to design neural networks that are specifically adapted for such problems
  - Must deal with very high-dimensional inputs: 150 x 150 pixels = 22500 inputs, or 3 x 22500 if RGB pixels
  - Can exploit the 2D topology of pixels (or 3D for video data)
  - Can build in invariance to certain variations: translation, illumination, etc.
- Convolutional networks leverage these ideas
  - Local connectivity
  - Parameter sharing
  - Convolution
  - Pooling / sub-sampling hidden units

Parameter Sharing

- Each feature map forms a 2D grid of features
  - can be computed with a discrete convolution ($*$) of a kernel matrix k_{ij} which is the hidden weights matrix W_{ij} with its rows and columns flipped



Jarret et al. 2009

$$y_j = g_j \tanh(\sum_i k_{ij} * x_i)$$

- x_i is the i^{th} channel of input
- k_{ij} is the convolution kernel
- g_j is a learned scaling factor
- y_j is the hidden layer

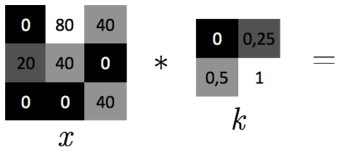
can add bias

Discrete Convolution

- Discrete convolution between **one** kernel (filter) and **one** channel image (matrix; 2-d tensor)
- This is not quite convolution; instead, it is correlation

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

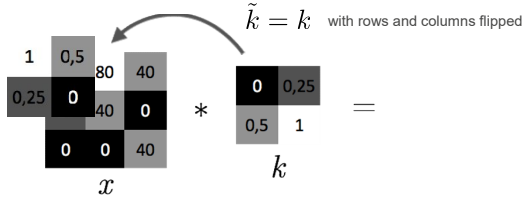
- Example:



Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

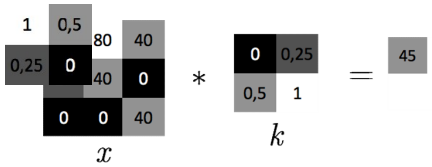
- Example:



Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

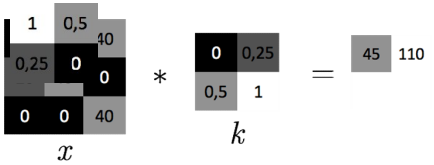
- Example: 1 x 0 + 0.5 x 80 + 0.25 x 20 + 0 x 40 = 45



Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

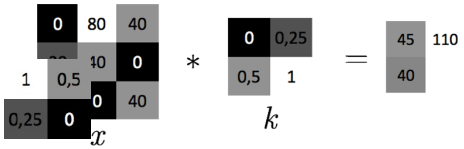
• Example: 1 x 80 + 0.5 x 40 + 0.25 x 40 + 0 x 0 = 110



Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

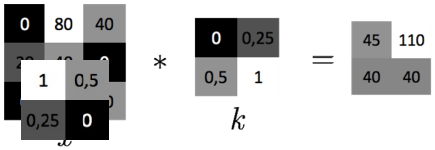
• Example: 1 x 20 + 0.5 x 40 + 0.25 x 0 + 0 x 0 = 40



Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} . k_{p,q}$$

• Example: 1 x 40 + 0.5 x 0 + 0.25 x 0 + 0 x 40 = 40



Discrete Convolution

• Pre-activations from channel x_i into feature map y_j can be computed by:

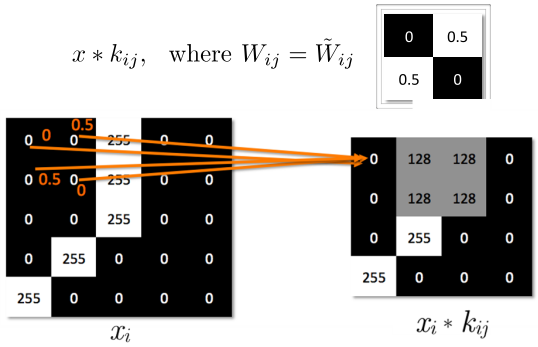
- getting the convolution kernel where $k_{ij} = W_{ij}$ from the connection matrix W_{ij}
- applying the correlation $x_i * k_{ij}$
- **We abuse the terminology and notation and refer and use the same notation for convolution and correlation when there is no ambiguity.**
  - This is equivalent to computing the discrete correlation of x_i with W_{ij}
  - Discrete convolution in general form (for f^{th} output filter (kernel), for c^{th} input channel)
  - k is a 4-d Tensor which is convolved to the 3-d Tensor input x

$$(x * k)_{fij} = \sum_{c} \sum_{p,q} x_{c,i+p,j+q} . k_{c,p,q,f}$$

Example

• Illustration:

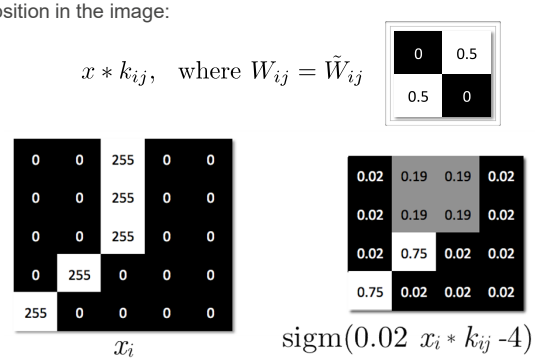
$x * k_{ij}, \quad \text{where } W_{ij} = \tilde{W}_{ij}$



Example

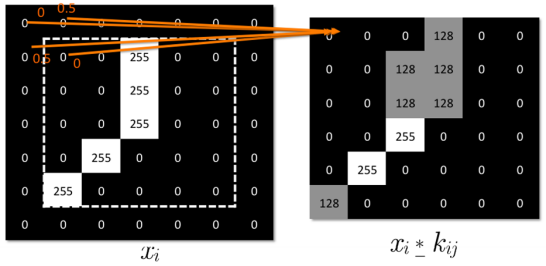
• With a non-linearity, we get a detector of a feature at any position in the image:

$x * k_{ij}, \quad \text{where } W_{ij} = \tilde{W}_{ij}$



Example

• Can use "zero padding" to allow going over the borders (*)



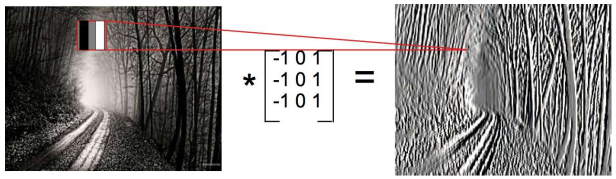
CNN Convolution Parameters

CNN — Parameters

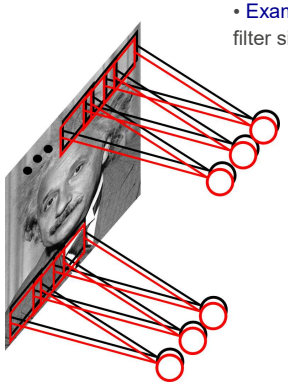
- **Filters**: Represents the amount of filters in a CL.
- **Kernel Size**: Defines the dimensions of the filters.
- **Stride**: Sets the size of the filter shift step.
- **Padding**: defines whether or not there is entry zeroing, influencing the output dimensions:

CNN Convolution Parameters

Example



Multiple Feature Maps



• Example: 200x200 image, 100 filters, filter size 10x10, 10K parameters

Convolutional Neural Networks

• Our goal is to design neural networks that are specifically adapted for such problems

- Must deal with very high-dimensional inputs: 150 x 150 pixels = 22500 inputs, or 3 x 22500 if RGB pixels
- Can exploit the 2D topology of pixels (or 3D for video data)
- Can build in invariance to certain variations: translation, illumination, etc.

• Convolutional networks leverage these ideas

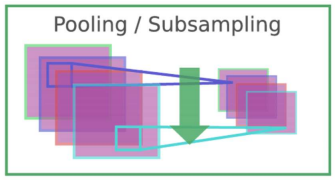
- Local connectivity
- Parameter sharing
- Convolution
- Pooling / subsampling hidden units

Pooling

• Pool hidden units in same neighborhood

- pooling is performed in non-overlapping neighborhoods (subsampling)

$$y_{ijk} = \max_{p,q} x_{i,j+p,k+q}$$



Jarret et al. 2009

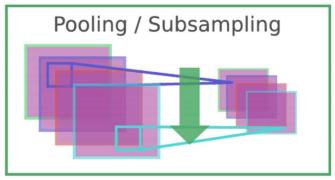
- x_i is the i^{th} channel of input
- $x_{i,j,k}$ is value of the i^{th} feature map at position j,k
- p is vertical index in local neighborhood
- q is horizontal index in local neighborhood
- y_{ijk} is pooled / subsampled layer

Pooling

• Pool hidden units in same neighborhood

- an alternative to "max" pooling is "average" pooling

$$y_{ijk} = \frac{1}{m^2} \sum_{p,q} x_{i,j+p,k+q}$$

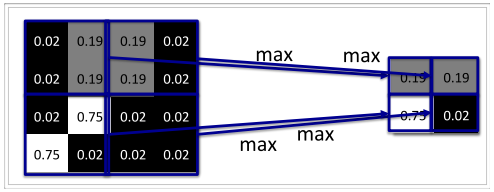


Jarret et al. 2009

- x_i is the i^{th} channel of input
- $x_{i,j,k}$ is value of the i^{th} feature map at position j,k
- p is vertical index in local neighborhood
- q is horizontal index in local neighborhood
- y_{ijk} is pooled / subsampled layer
- m is the neighborhood height/width

Example: Pooling

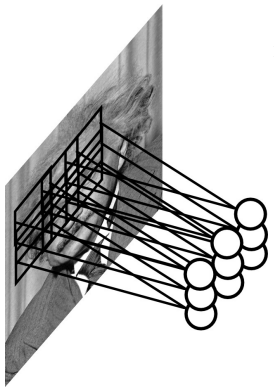
• Illustration of pooling/subsampling operation



• Why pooling?

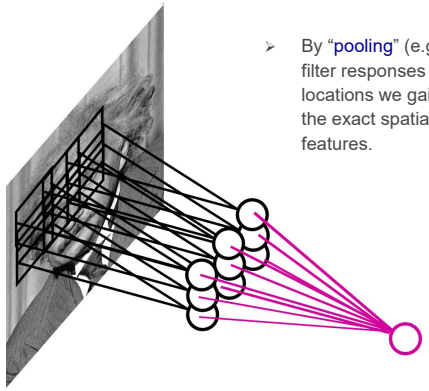
- Introduces invariance to local translations
- Reduces the number of hidden units in hidden layer

Example: Pooling



- can we make the detection robust to the exact location of the eye?

Example: Pooling

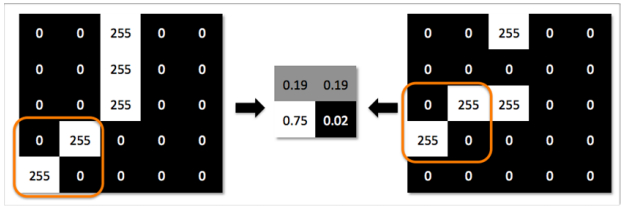


- By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Translation Invariance

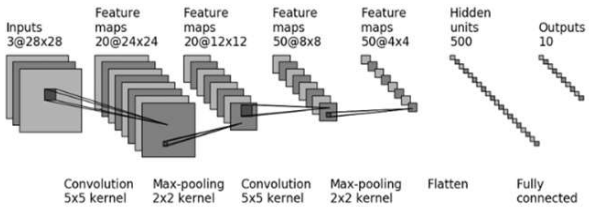
• Illustration of local translation invariance

- both images result in the same feature map after pooling



Convolutional Network

• Convolutional neural network alternates between the convolutional and pooling layers



Structure of LeNet-5.

From Yann LeCun's slides

Convolutional Network

• For classification: Output layer is a regular, fully connected layer with softmax non-linearity

- Output provides an estimate of the conditional probability of each class

• The network is trained by stochastic gradient descent

- Backpropagation is used similarly as in a fully connected network
- We have seen how to pass gradients through element-wise activation function
- We also need to pass gradients through the convolution operation and the pooling operation

Gradient of Pooling Layer

• Let l be the loss function

- For max pooling operation $y_{ijk} = \max_{p,q} x_{i,j+p,k+q}$, the gradient for x_{ijk} is

$$\nabla_{x_{ijk}} l = 0, \text{ except for } \nabla_{x_{i,j+p',k+q'}} l = \nabla_{y_{ijk}} l$$

- where p', q' = argmax $x_{i,j+p,k+q}$
- In other words, only the "winning" units in layer x get the gradient from the pooled layer
- For the average operation $y_{ijk} = \frac{1}{m^2} \sum_{p,q} x_{i,j+p,k+q}$, the gradient for x_{ijk} is

$$\nabla_x l = \frac{1}{m^2} \text{upsample}(\nabla_y l)$$

where you should calculate upsample(.) as an excerise.

Gradient of Convolutional Layer

The goal is to compute the gradient of the loss function w.r.t. to the weights of the filters in layer h^u and input X^{u-1} given the gradient in the layer h^u.

Remember $h^u = g(w^u * X^{u-1})$, where g is some nonlinear operator (nonlinear activation, pooling,..)

Assume you have computed the gradient of loss function, l, up to the **current hidden convolutional layer h^u**, i.e., $\partial_{h^u_{ijk}} \triangleq \nabla^u_{h_{ijk}} l = \frac{\partial l}{\partial h^u_{ijk}}$.

Here h stands for "hidden" layer and has been introduced for the ease of notation. Also, index i denotes the channel number of the current hidden layer h_u. That is, $i = 1,2, \dots, C_u$ and $u = 1,2, \dots L$, where C_u denotes the number of channels in layer u and L is the total number of layers.

For simplicity drop the channel index and the layer number. So, $\partial_{h_{ij}} \triangleq \frac{\partial l}{\partial h_{ij}}$.

All the computations should be done for all the channels and for all the convolutional layers.

Similarly define the gradient w.r.t. to the filter coefficients as $\partial_{w_{ij}} \triangleq \frac{\partial l}{\partial w_{ij}}$

Gradient of Convolutional Layer -- Continue

Now, we establish the gradient operation visually [1].

Assume in forward pass, we have convolved a 3x3 input with a kernel 2X2 which outputs a 2X2 matri

We calculate this for $g(z) = z$. Please extend to the general case as an exercise

With the notation from the previous slide, we can write:

$$\begin{aligned}
\partial_{w_{11}} &= X_{11}\partial_{h_{11}} + X_{12}\partial_{h_{12}} + X_{21}\partial_{h_{21}} + X_{22}\partial_{h_{22}} \\
\partial_{w_{12}} &= X_{12}\partial_{h_{11}} + X_{13}\partial_{h_{12}} + X_{22}\partial_{h_{21}} + X_{23}\partial_{h_{22}} \\
\partial_{w_{21}} &= X_{21}\partial_{h_{11}} + X_{22}\partial_{h_{12}} + X_{31}\partial_{h_{21}} + X_{32}\partial_{h_{22}} \\
\partial_{w_{22}} &= X_{22}\partial_{h_{11}} + X_{23}\partial_{h_{12}} + X_{32}\partial_{h_{21}} + X_{33}\partial_{h_{22}}
\end{aligned}$$

Gradient of Convolutional Layer -- Continue

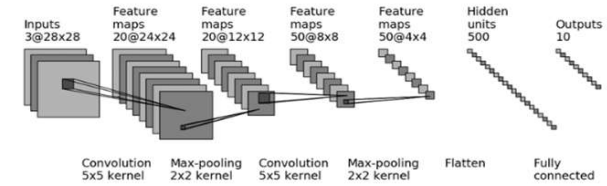
- This is our old friend, discrete convolution operator:

$$\partial_W = X * \partial_h$$

- Where $\partial_{w_{ij}} = \sum_{p,q} X_{i+p,j+q} \partial_{h_{ij}}$
- Similarly, we can compute the gradient of the loss w.r.t. X (input layer , or X^{u-1}) since we need these gradients in order to propagate the gradient to the layers towards input of the CNN.

Convolutional Network

• Convolutional neural network alternates between the convolutional and pooling layers



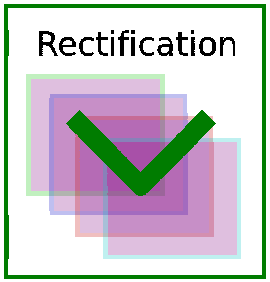
Structure of LeNet-5.

• Need to introduce other operations that can improve object recognition.

Rectification

• Rectification layer: $y_{ijk} = |x_{ijk}|$

- introduces invariance to the sign of the unit in the previous layer
- for instance, loss of information of whether an edge is black-to-white or white-to-black



Local Contrast Normalization

• Perform local contrast normalization

Local average

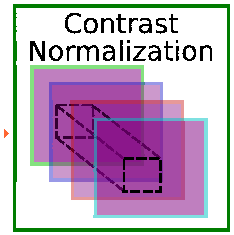
$$v_{ijk} = x_{ijk} - \sum_{ipq} \alpha_{pq} x_{i,j+p,k+q}$$

$$y_{ijk} = v_{ijk} / \max(c, \sigma_{jk})$$

Local stdev

$$\sigma_{jk} = (\sum_{ipq} \alpha_{pq} v^2_{i,j+p,k+q})^{1/2}, \quad \sum_{pq} \alpha_{pq} = 1$$

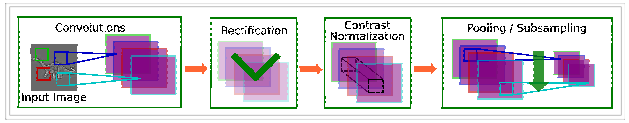
where c is a small constant to prevent division by 0 and $\alpha_{pq} \geq 0$.



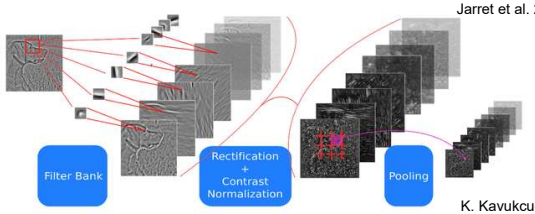
- reduces unit's activation if neighbors are also active
- creates competition between feature maps
- scales activations at each layer better for learning

Convolutional Network

• These operations are inserted after the convolutions and before the pooling



Jarret et al. 2009



K. Kavukcuoglu

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

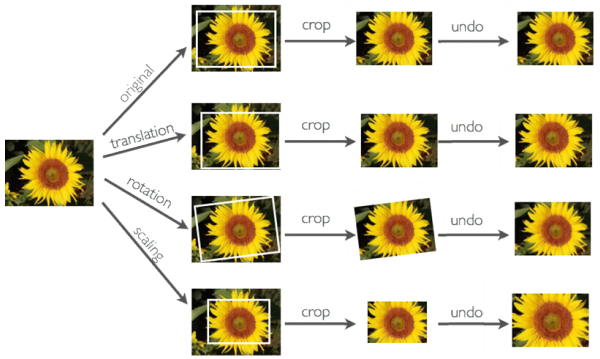
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learned linear transformation to adapt to non-linear activation function (**γ and β are trained**)

Invariance by Dataset Expansion (Augmentation)

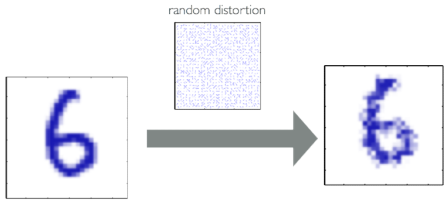
- Invariances built-in in convolutional network:
  - small translations: due to convolution and max pooling
  - small illumination changes: due to local contrast normalization
- It is not invariant to other important variations such as rotations and scale changes
- However, it's easy to artificially generate data with such transformations
  - could use such data as additional training data
  - neural network can potentially learn to be invariant to such transformations

Generating Additional Examples



Elastic Distortions

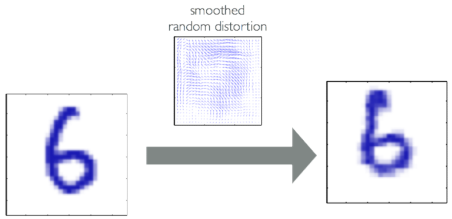
- Can add "elastic" deformations (useful in character recognition)
- We can do this by applying a "distortion field" to the image
  - a distortion field specifies where to displace each pixel value



Bishop's book

Elastic Distortions

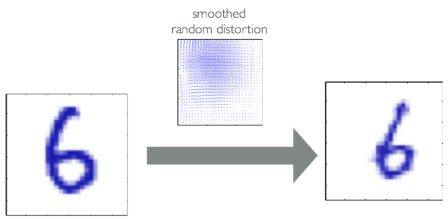
- Can add ‘‘elastic’’ deformations (useful in character recognition)
- We can do this by applying a ‘‘distortion field’’ to the image
  - a distortion field specifies where to displace each pixel value



Bishop’s book

Elastic Distortions

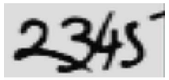
- Can add ‘‘elastic’’ deformations (useful in character recognition)
- We can do this by applying a ‘‘distortion field’’ to the image
  - a distortion field specifies where to displace each pixel value



Bishop’s book

Conv Nets: Examples

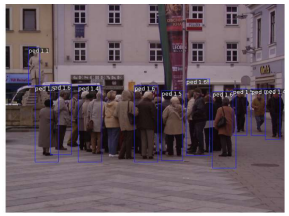
- Optical Character Recognition, House Number and Traffic Sign classification



Ciresan et al. “MCDNN for image classification” CVPR 2012
Wan et al. “Regularization of neural networks using dropconnect” ICML 2013
Goodfellow et al. “Multi-digit nuber recognition from StreetView...” ICLR 2014
Jaderberg et al. “Synthetic data and ANN for natural scene text recognition” arXiv 2014

Conv Nets: Examples

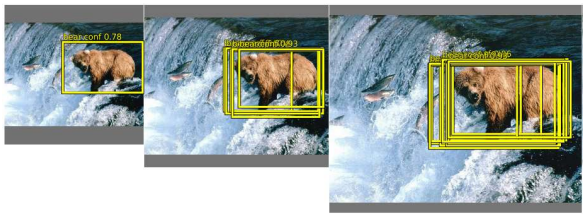
- Pedestrian detection



Sermanet et al. “Pedestrian detection with unsupervised multi-stage..” CVPR 2013

Conv Nets: Examples

- Object Detection



Sermanet et al. “OverFeat: Integrated recognition, localization” arxiv 2013
Girshick et al. “Rich feature hierarchies for accurate object detection” arxiv 2013
Szegedy et al. “DNN for object detection” NIPS 2013

ImageNet Dataset

- 1.2 million images, 1000 classes

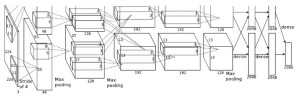
Examples of Hammer

Deng et al. “Imagenet: a large scale hierarchical image database” CVPR 2009

Important Breakthroughs

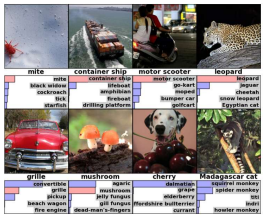
- Deep Convolutional Nets for Vision (Supervised)

  Krizhevsky, A., Sutskever, I. and Hinton, G. E., ImageNet Classification with Deep Convolutional Neural Networks, NeurIPS, 2012.



IMAGENET

1.2 million training images
1000 classes

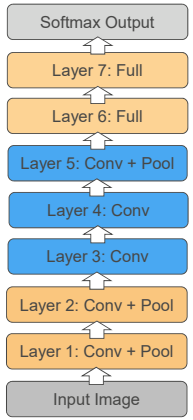


Architecture

- How can we select the right architecture:
  - Manual tuning of features is now replaced with the manual tuning of architectures
- Depth
- Width
- Parameter count

AlexNet

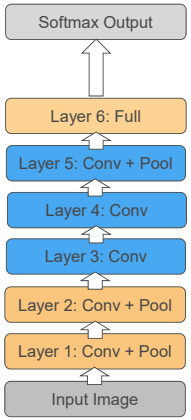
- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error



[From Rob Fergus' CIFAR 2016 tutorial]

AlexNet

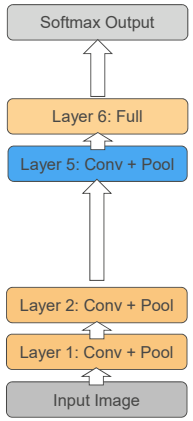
- Remove top fully connected layer 7
- Drop ~16 million parameters
- Only 1.1% drop in error!



[From Rob Fergus' CIFAR 2016 tutorial]

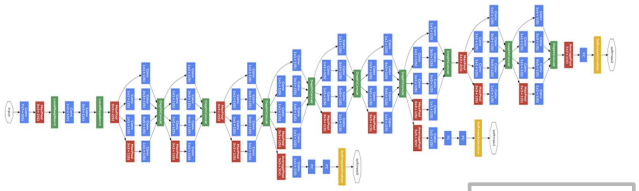
AlexNet

- Let us remove upper feature extractor layers and fully connected:
  - Layers 3,4, 6 and 7
- Drop ~50 million parameters
- **33.5 drop in error!**
- Depth of the network is the key.



[From Rob Fergus' CIFAR 2016 tutorial]

GoogLeNet



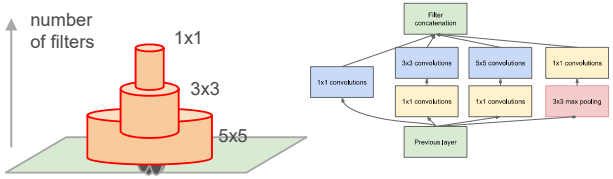
- 24 layer model that uses so-called inception module.

Convolution
Pooling
Softmax
Other

[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

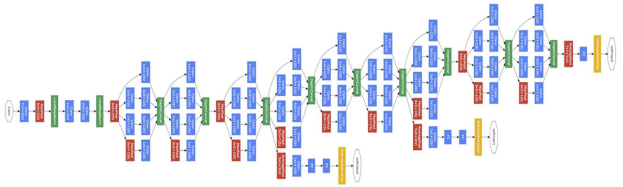
GoogLeNet

- GoogLeNet inception module:
  - Multiple filter scales at each layer
  - Dimensionality reduction to keep computational requirements down



[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

GoogLeNet

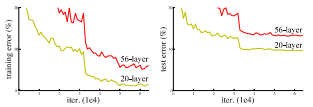


- Width of inception modules ranges from 256 filters (in early modules) to 1024 in top inception modules.
- Can remove fully connected layers on top completely
- Number of parameters is reduced to 5 million
- 6.7% top-5 validation error on Imagnet

[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

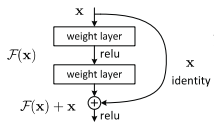
Residual Networks

Really, really deep convolutional nets do not train well,
E.g. CIFAR10:



Key idea: introduce "pass through" into each layer

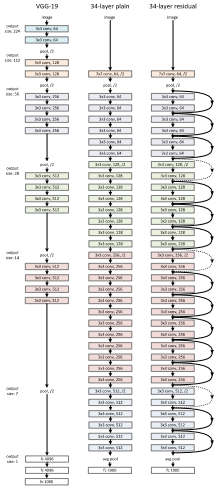
Thus only residual now needs to be learned



method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43[†]
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except [†] reported on the test set).

With ensembling, 3.57% top-5 test error on ImageNet



[He, Zhang, Ren, Sun, CVPR 2016]

Choosing the Architecture

- Task dependent
- Cross-validation
- [Convolution → pooling]* + fully connected layer
- The more data: the more layers and the more kernels
  - Look at the number of parameters at each layer
  - Look at the number of flops at each layer
- Computational resources

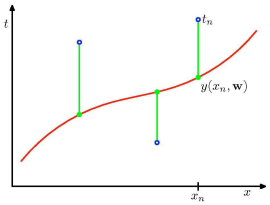
[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Underfitting, Overfitting and Training Tricks

Vahid Tarokh
ECE 685D, Fall 2025

Example: Polynomial Curve Fitting

• As for the least squares example: we can minimize the sum of the squares of the errors between the predictions $y(x_n, \mathbf{w})$ for each data point x_n and the corresponding target values t_n.



Loss function: sum-of-squared error function:

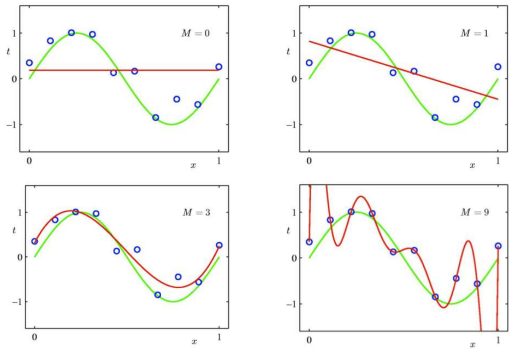
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2.$$

• Similar to the linear least squares: Minimizing sum-of-squared error function has a unique solution $\mathbf{w}^*$.

• The model is characterized by M+1 parameters $\mathbf{w}^*$.

• How do we choose M? ! **Model Selection**.

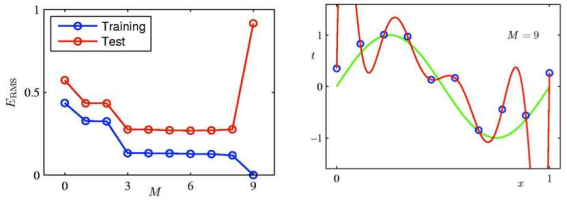
Some Fits to the Data



For M=9, we have fitted the training data perfectly.

Overfitting

• Consider a separate **test set** containing 100 new data points generated using the same procedure that was used to generate the training data.

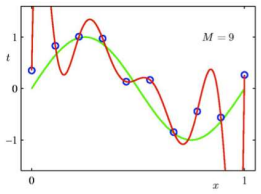


• For M=9, the training error is zero ! The polynomial contains 10 degrees of freedom corresponding to 10 parameters $\mathbf{w}$, and so can be fitted exactly to the 10 data points.

• However, the test error has become very large. Why?

Overfitting

	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

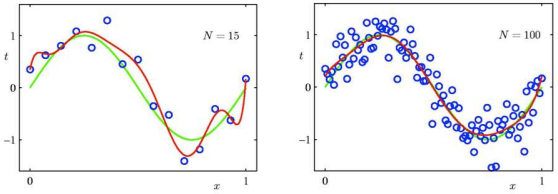


- As M increases, the magnitude of coefficients gets larger.
- For M=9, the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

  More flexible polynomials with larger M tune to the random noise on the target values.

Varying the Size of the Data

9th order polynomial

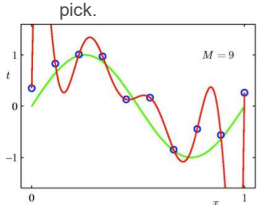


- For a given model complexity, the overfitting problem becomes less severe as the size of the dataset increases.

- However, the number of parameters is not necessarily the most appropriate measure of the model complexity.

Generalization

- The goal is achieve good **generalization** by making accurate predictions for new test data that is not known during learning.
- Choosing the values of parameters that minimize the loss function on the training data may not be the best option.
- We would like to model the true regularities in the data and ignore the noise in the data:
  - It is hard to know which regularities are real and which are accidental due to the particular training examples we happen to pick.



- Intuition: We expect the model to generalize if it explains the data well given the complexity of the model.
- If the model has as many degrees of freedom as the data, it can fit the data perfectly. But this is not very informative.
- Some theory on how to control model complexity to optimize generalization.

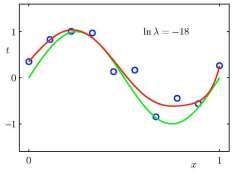
A Simple Way to Penalize Complexity

One technique for controlling over-fitting phenomenon is **regularization**, which amounts to adding a penalty term to the error function.

penalized error function ↘ target value ↘ regularization parameter

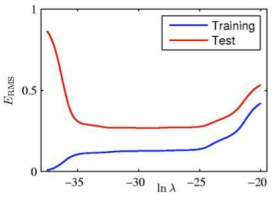
$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

where $\|\mathbf{w}\| = \mathbf{w}^T\mathbf{w} = w_1^2 + w_2^2 + ... + w_M^2$ and λ is called the regularization term. Note that we do not penalize the bias term w_0.



- The idea is to "shrink" estimated parameters towards zero (or towards the mean of some other weights).
- Shrinking to zero: penalize coefficients based on their size.
- For a penalty function which is the sum of the squares of the parameters, this is known as "**weight decay**", or "**ridge regression**".

Regularization



	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

Graph of the root-mean-squared training and test errors vs. $\ln\lambda$ for the M=9 polynomial.

How to choose λ?

Cross Validation

If the data is plentiful, we can divide the dataset into three subsets:

- Training Data: used to fitting/learning the parameters of the model.
- Validation Data: not used for learning but for selecting the model, or choosing the amount of regularization that works best.
- Test Data: used to get performance of the final model.

For many applications, the supply of data for training and testing is limited.
To build good models, we may want to use as much training data as possible.
If the validation set is small, we get noisy estimate of the predictive performance.

S fold cross-validation

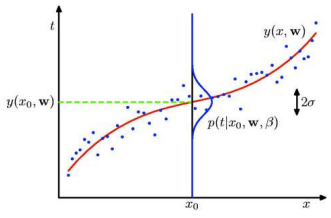
• The data is partitioned into S groups.
• Then S-1 of the groups are used for training the model, which is evaluated on the remaining group.
• Repeat procedure for all S possible choices of the held-out group.
• Performance from the S runs are averaged.

Probabilistic Perspective Of Polynomial Regression

• So far we saw that polynomial curve fitting can be expressed in terms of error minimization. We now view it from probabilistic perspective.
• Suppose that our model arose from a statistical model:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon,$$

where ϵ is a random error having Gaussian distribution with zero mean, and is independent of $\mathbf{x}$.



Thus we have:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}),$$

where β is a precision parameter, corresponding to the inverse variance.

We will use probability distribution and probability density interchangeably. It should be obvious from the context.

Maximum Likelihood

If the data are assumed to be independently and identically distributed (*i.i.d assumption*), the likelihood function takes form:

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = \prod_{i=1}^{N} \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}).$$

It is often convenient to maximize the log of the likelihood function:

$$\ln p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = \underbrace{-\frac{\beta}{2}\sum_{n=1}^{N}(y(\mathbf{x}_n, \mathbf{w}) - t_n)^2}_{\beta E(\mathbf{w})} + \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi).$$

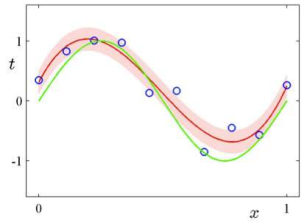
• Maximizing log-likelihood with respect to $\mathbf{w}$ (under the assumption of a Gaussian noise) is equivalent to minimizing the *sum-of-squared error* function.
• Determine $\mathbf{w}_{ML}$ by maximizing log-likelihood. Then maximizing w.r.t. β :

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n}(y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n)^2.$$

Predictive Distribution

Once we determined the parameters **w** and β, we can make prediction for new values of **x**:

$$p(t|\mathbf{x}, \mathbf{w}_{ML}, \beta_{ML}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}_{ML}), \beta_{ML}^{-1}).$$



Maximum Likelihood

• As before, assume observations arise from a deterministic function with an additive Gaussian noise:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon,$$

which we can write as:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}).$$

• Given observed inputs $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, and corresponding target values $\mathbf{t} = [t_1, t_2, ..., t_N]^T$, , under i.i.d assumption, we can write down the likelihood function:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{i=1}^{N} \mathcal{N}(t_n|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n), \beta),$$

where $\boldsymbol{\phi}(\mathbf{x}) = (\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), ..., \phi_{M-1}(\mathbf{x}))^T$.

Maximum Likelihood

Taking the logarithm, we obtain:

$$\begin{aligned}\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) &= \sum_{i=1}^{N} \ln \mathcal{N}(t_n|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n), \beta) \\ &= \underbrace{-\frac{\beta}{2}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)\right)^2}_{\text{sum-of-squares error function}} + \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi).\end{aligned}$$

Differentiating and setting to zero yields:

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^{N} \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\} \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} = \mathbf{0}.$$

Maximum Likelihood

Differentiating and setting to zero yields:

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^{N} \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\} \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} = \mathbf{0}.$$

Solving for **w**, we get:

$$\mathbf{w}_{\mathrm{ML}} = \underbrace{\left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}}_{\text{Depends on Data}}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}$$

The Moore-Penrose pseudo-inverse of $\boldsymbol{\Phi}^{\dagger}$

where $\boldsymbol{\Phi}$ is known as the design matrix:

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}.$$

Sequential Learning

• The training data examples are presented one at a time, and the model parameters are updated after each such presentation (online learning):

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \bigtriangledown E_n$$

weights after seeing training case t+1

learning rate

vector of derivatives of the squared error w.r.t. the weights on the training case presented at time t.

• For the case of sum-of-squares error function, we obtain:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \left(t_n - \mathbf{w}^{(t)^T} \boldsymbol{\phi}(\mathbf{x}_n)\right) \boldsymbol{\phi}(\mathbf{x}_n).$$

• Stochastic gradient descent: The training examples are picked at random (dominant technique when learning with very large datasets).

• Care must be taken when choosing learning rate to ensure convergence.

Regularized Least Squares

• Let us consider the following error function:

$$E_D(\mathbf{w}) + \lambda E_W(\mathbf{w})$$

Data term + Regularization term

λ is called the regularization coefficient.

• Using sum-of-squares error function with a quadratic penalization term, we obtain:

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$$

which is minimized by setting:

Depends on Data

$$\mathbf{w} = \left(\lambda\mathbf{I} + \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}.$$

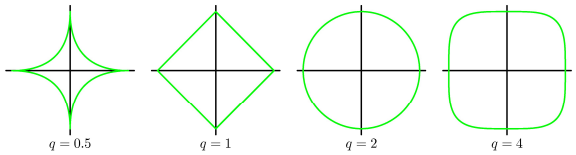
Ridge regression

The solution adds a positive constant to the diagonal of $\boldsymbol{\Phi}^{\mathbf{T}}\boldsymbol{\Phi}$. This makes the problem nonsingular, even if $\boldsymbol{\Phi}^{\mathbf{T}}\boldsymbol{\Phi}$ is not of full rank (e.g. when the number of training examples is less than the number of basis functions).

Other Regularizers

Using a more general regularizer, we get:

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q$$



Lasso Quadratic

The LASSO

• Penalize the absolute value of the weights:

$$\mathbf{w}^{lasso} = \underset{\mathbf{w}}{\operatorname{argmin}} \left[\frac{1}{2}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{M-1}|w_j|\right].$$

• For sufficiently large λ, some of the coefficients will be driven to exactly zero, leading to a sparse model.

• The above formulation is equivalent to:

$$\mathbf{w}^{lasso} = \underset{\mathbf{w}}{\operatorname{argmin}} \underbrace{\frac{1}{2}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)\right)^2}_{\text{unregularized sum-of-squares error}}, \text{ subject to } \sum_{j=1}^{M-1}|w_j| \leq \tau.$$

• The two approaches are related using Lagrange multiplies.

• The LASSO solution is a quadratic programming problem: can be solved efficiently.

Review of Inference

Assume that the training examples are drawn independently from the set of all possible examples, or from the same underlying distribution $p(\mathbf{x}, t)$.

We also assume that the training examples are identically distributed (i.i.d assumption).

Assume that the test samples are drawn in exactly the same way -- i.i.d from the same distribution as the training data.

These assumptions make it unlikely that some strong regularity in the training data will be absent in the test data.

Statistical Decision Theory

- We now develop a small amount of theory that provides a framework for developing many of the models we consider.
- Suppose we have a real-valued input vector **x** and a corresponding target (output) value t with joint probability $p(\mathbf{x}, t)$. distribution:
- Our goal is predict target t given a new value for **x**:
  - for regression: t is a real-valued continuous target.
  - for classification: t is a categorical variable representing class labels.

The joint probability distribution $p(\mathbf{x}, t)$ provides a complete summary of uncertainties associated with these random variables.

Determining $p(\mathbf{x}, t)$ from training data is known as the inference problem.

Example: Classification

Medical diagnosis: Based on the X-ray image, we would like determine whether the patient has cancer or not.

The input vector **x** is the set of pixel intensities, and the output variable t will represent the presence of cancer, class C_1, or absence of cancer, class C_2.

C_1: Cancer present

C_2: Cancer absent

x -- set of pixel intensities

Choose t to be binary: t=0 correspond to class C_1, and t=1 corresponds to C_2.

Inference Problem: Determine the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$, or equivalently $p(\mathbf{x}, t)$. However, at the end, we must make a decision of whether to give treatment to the patient or not.

Example: Classification

Informally: Given a new X-ray image, our goal is to decide which of the two classes that image should be assigned to.

- We could compute conditional probabilities of the two classes, given the input image:

posterior probability of C_k given observed data.

probability of observed data given C_k

prior probability for class C_k

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}, \mathcal{C}_k)}{\sum_{k=1}^{K} p(\mathbf{x}, \mathcal{C}_k)} = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}$$

Bayes' Rule

- If our goal to minimize the probability of assigning **x** to the wrong class, then we should choose the class having the highest posterior probability.

Expected Loss

• Loss Function: overall measure of loss incurred by taking any of the available decisions.

Suppose that for **x**, the true class is C_k, but we assign **x** to class j ! incur loss of L_{kj} (k,j element of a loss matrix).

Consider medical diagnosis example: example of a loss matrix:

Truth \ Decision	cancer	normal
cancer	0	1000
normal	1	0

Expected Loss: $\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, C_k)\, \mathrm{d}\mathbf{x}$

Goal is to choose decision regions $\mathcal{R}_j$ as to minimize expected loss.

Regression

Let **x** ∈ R^d denote a real-valued input vector, and t ∈ R denote a real-valued random target (output) variable with joint the $p(\mathbf{x}, t)$. distribution

• The decision step consists of finding an estimate y(**x**) of t for each input **x**.

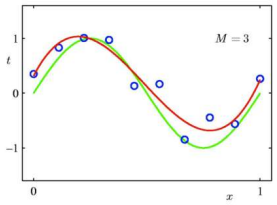
• To quantify what it means to do well or poorly on a task, we need to define a loss (error) function: $L(t, y(\mathbf{x}))$.

• The average, or expected, loss is given by:

$$\mathbb{E}[L] = \int\int L(t, y(\mathbf{x})) p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$

• If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int\int \big(t - y(\mathbf{x})\big)^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$



Squared Loss Function

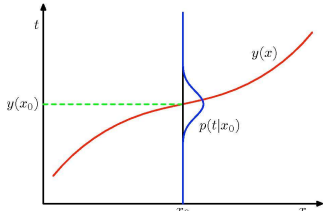
• If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int\int \big(t - y(\mathbf{x})\big)^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$

• Our goal is to choose y(x) so as to minimize the expected squared loss.

• The optimal solution (if we assume a completely flexible function) is the conditional average:

$$y(\mathbf{x}) = \int t p(t|\mathbf{x})\mathrm{d}t = \mathbb{E}[t|\mathbf{x}].$$



The regression function y(**x**) that minimizes the expected squared loss is given by the mean of the conditional distribution $p(t|\mathbf{x})$.

Squared Loss Function

• If we use squared loss, we obtain:

$$\begin{aligned}(y(\mathbf{x}) - t)^2 &= (y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t|\mathbf{x}] - t)^2 \\ &= \big(y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\big)^2 + 2\big(y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\big)\big(\mathbb{E}[t|\mathbf{x}] - t\big) + \big(\mathbb{E}[t|\mathbf{x}] - t\big)^2.\end{aligned}$$

• Plugging into expected loss:

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \int \mathrm{var}\,[t|\mathbf{x}]\, p(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

expected loss is minimized when $y(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}]$.

intrinsic variability of the target values.

Because it is independent noise, it represents an irreducible minimum value of expected loss.

Other Loss Function

• Simple generalization of the squared loss, called the *Minkowski* loss:

$$\mathbb{E}[L] = \int\int \left(t - y(\mathbf{x})\right)^q p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$

• The minimum of $\mathbb{E}[L]$ is given by:

- the conditional mean for q=2,
- the conditional median when q=1

Bias-Variance Decomposition

• Introducing a regularization term can help us control overfitting. But how can we determine a suitable value of the regularization coefficient?

• Let us examine the expected squared loss function. Remember:

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \underbrace{\iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t}$$

intrinsic variability of the target values: The minimum achievable value of expected loss

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int t p(t|\mathbf{x})\,\mathrm{d}t.$$

• If we model $h(\mathbf{x})$ using a parametric function $y(\mathbf{x}, \mathbf{w})$, then from a Bayesian perspective, the uncertainly in our model is expressed through the posterior distribution over parameters **w**.

• We first look at the frequentist perspective.

Bias-Variance Decomposition

• From a frequentist perspective: we make a point estimate of $\mathbf{w}^*$ based on the dataset D.

• We next interpret the uncertainly of this estimate through the following thought experiment:

- Suppose we had a large number of datasets, each of size N, where each dataset is drawn independently from $p(\mathbf{x}, t)$.
- For each dataset D, we can obtain a prediction function $y(\mathbf{x}; \mathcal{D})$.
- Different datasets will give different prediction functions.
- The performance of a particular learning algorithm is then assessed by taking the average over the ensemble of these datasets.

• Let us consider the expression:

$$\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2.$$

• Note that this quantity depends on a particular dataset D.

Bias-Variance Decomposition

• Consider:

$$\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2.$$

• Adding and subtracting the term $\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]$, we obtain

$$\begin{aligned}
&\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 \\
&\quad +2\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}.
\end{aligned}$$

• Taking the expectation over $\mathcal{D}$, the last term vanishes, so we get:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2\right] \\
&= \underbrace{\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\right]}_{\text{variance}}.
\end{aligned}$$

Bias-Variance Trade-off

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

Average predictions over all datasets differ from the optimal regression function.

Solutions for individual datasets vary around their averages -- how sensitive is the function to the particular choice of the dataset.

Intrinsic variability of the target values.

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right] p(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

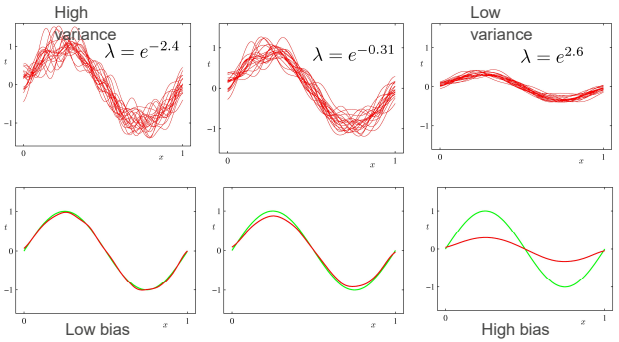
$$\text{noise} = \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t$$

• Trade-off between bias and variance: With very flexible models (high complexity) we have low bias and high variance; With relatively rigid models (low complexity) we have high bias and low variance.

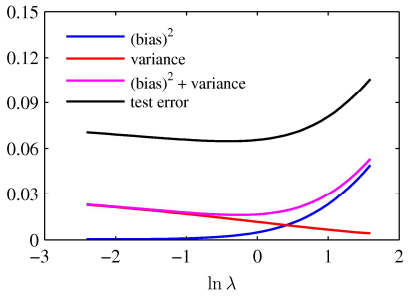
• The model with the optimal predictive capabilities has to balance between bias and variance.

Bias-Variance Trade-off

• Consider the sinusoidal dataset. We generate 100 datasets, each containing N=25 points, drawn independently from $h(x) = \sin 2\pi x$.



Bias-Variance Trade-off



From these plots note that over-regularized model (large λ) has high bias, and under-regularized model (low λ) has high variance.

Beating the Bias-Variance Trade-off

• We can reduce the variance by averaging over many models trained on different datasets:

- In practice, we only have a single observed dataset. If we had many independent training sets, we would be better off combining them into one large training dataset.

• Given a standard training set D of size N, we could generate new training sets, of size N, by sampling examples from D uniformly and with replacement.

- This is called bagging and it works quite well in practice (ad hoc).

• Given enough computation, we could also resort to the Bayesian framework:

- Combine the predictions of many models using the posterior probability of each parameter vector as the combination weight.

Applications to Deep Networks

Overfitting Issues in Deep Networks

Deep nets may have many hidden layers

With limited training data, over-fitting may happen

Methods to reduce overfitting (Regularization)

- Cross-validation set (discussed before)
- Weight regularization, e.g., L_1 and L_2 regularization (discussed before)
- Dropout

In order to understand dropout, we need to first understand bagging.

Bagging (bootstrap aggregating) is a method of averaging over several models to improve generalization. The idea is to train several different models separately, then have all the models vote on the output for test examples.

This is an example of a general strategy in machine learning called model averaging.

- Bagging: average the predictions of all possible settings of the parameters

Bagging

Consider for example a set of k regression models. Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $E(\epsilon_i^2) = v$ and co-variances $E(\epsilon_i \epsilon_j) = c$.

Then the error made by the average prediction of all the ensemble models has variance $\frac{v}{k} + c\frac{k-1}{k}$

If $c = 0$, the bagging reduces square error by a factor of k. If $c = v$, then no gains is achieved.

Typically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset.

Model i is then trained on dataset i. The differences between which examples are included in each dataset result in differences between the trained models.

Bagging

In bagging:

- The classification probability of the ensemble of neural networks is given by the arithmetic mean of all the corresponding distributions
  - Model i produces the prediction probability as $p^{(i)}(y|\boldsymbol{x})$
  - Prediction of ensemble of k models is the arithmetic mean $\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y|\boldsymbol{x})$
  - This gives equal weight to ensemble elements predictions
  - It is possible to give unequal weights to ensemble elements giving $\sum_{i=1}^{k} w_i\, p^{(i)}(y|\boldsymbol{x})$ with $w_i \geq 0$ and $\sum_i^k w_i = 1$.
  - Use geometric mean rather than arithmetic mean of the ensemble member's predicted distributions is *intuitively equivalent* to ensemble log-likelihood optimization.

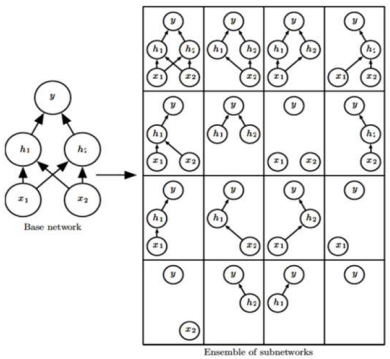
Bagging

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all the models are trained on the same dataset.

Differences in random initialization, in random selection of mini-batches, or in outcomes of nondeterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

This means that bagging can be very useful in neural networks.

Subnetworks Example



Dropout

Specifically, dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network.

Parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

Dropout aims to approximate the above, but with an exponentially large number of neural networks.

Note that in most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.

Dropout

In bagging, each model is trained to converge on its respective training set.

In dropout, typically most models are not explicitly trained at all.

- A small fraction of the possible subnetworks trained for a single step.

Model parameters for subnetworks are shared

The parameter sharing causes the remaining subnetworks to perform well too.

Dropout

Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent.

Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network.

- The mask for each unit is sampled independently from all the others.
- The probability of sampling a mask value of one (causing a unit to b included) is a hyper-parameter fixed before training begins.
- It is not a function of the current value of the model parameters or the input example.
- Typically, an input unit is included with probability 0.8, and a hidden unit is included with probability 0.5.

We then run forward propagation, back-propagation, and the learning update as usual.

Mathematical Model of Dropout

Denote μ as a mask vector

- which units to include and which ones to be removed

Denote $J(\theta, \mu)$ as the cost of the model prediction

Training with dropout consists of minimizing $\mathbb{E}_{\mu}(J(\theta, \mu))$

- Expected value contains exponential # of terms

One can get an unbiased estimate of its gradient by sampling values of μ

Mask for dropout training

Denote μ as a mask vector

- each subnetwork is this defined by mask vector μ
  - μ determines which units to include and to remove

Hence, each subnetwork outputs a probability distribution $p(y \mid x, \mu)$.

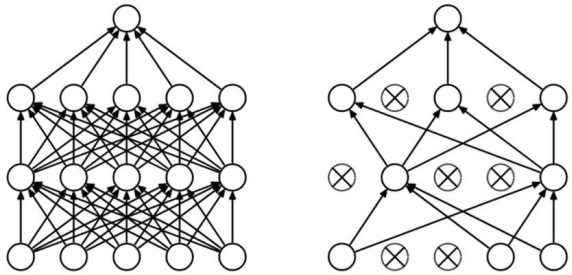
If subnetworks are chosen according to probability/weight $p(\mu)$, then the arithmetic mean **over all masks** is given $\sum_{i=1}^{k} p^{(i)}(y|\boldsymbol{x})\, p(\mu)$

$p(\mu)$ is the distribution used to sample μ at training time

Geometric mean is preferred (average $log\,[\,p^{(i)}(y|\boldsymbol{x})]$).

This sum may include an exponential number of terms.

Visualization of Drop-out



(a) Standard Neural Net (b) After applying dropout.

Dropout

Approximate the sum using sampling

1 - By averaging together the output from many masks

- 10-20 masks are sufficient for good performance

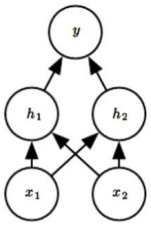
2 - Use geometric mean rather than arithmetic mean of the ensemble member's predicted distributions

$$p_{ensemble}(y|x) = \sqrt[2^d]{\prod_{\mu} p(y|x,\mu)}$$

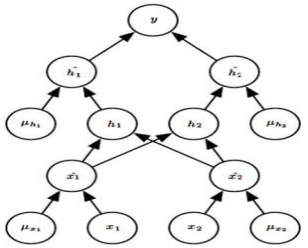
3- We have to make sure that none of the models assign probability zero to any event

Forward Propagation with Dropout

Base network



Forward propagation with dropout



μ is a random binary vector (mask) with one entry for each input

The probability of each entry being 1 is a hyper-parameter, usually 0.5 for the hidden layer, and 0.8 for the input.

Each unit is multiplied by the corresponding mask μ

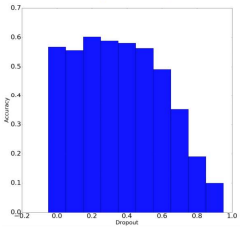
Equivalent to randomly selecting one of the subnetworks of previous slide

What is the effect of probability of drop

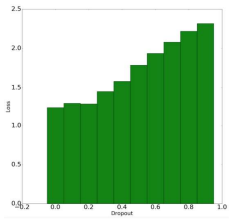
CIFAR-10 test dataset

- Three convolution layers of size 64, 128 and 256
- Followed by two densely connected layers of size 512
- output layer dense layer of size 10

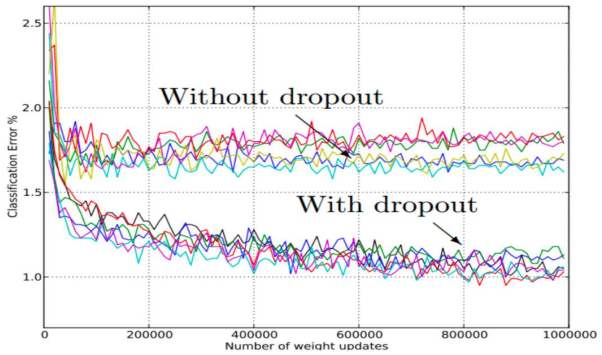
Accuracy vs dropout



Loss vs dropout



Classification accuracy with or without Dropout

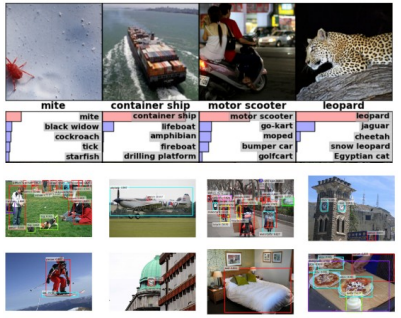


Convolutional Neural Networks Applications to Object Detection

Vahid Tarokh
ECE685D, Fall 2025

Classification versus detection

- Very related, but different problems.
- Many aspects will be shared between the two problems
- In detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image.
- There could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.

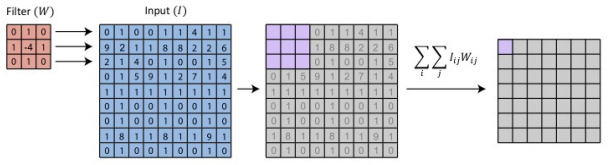


Pieces of a Deep Algorithm

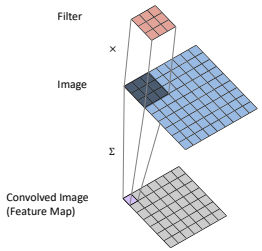
Object detection is very related to image classification. We use similar types of algorithms and related data types.

	Image Classification	Object Detection
Model	CNN Feature Extractor + Softmax Classifier	SSD, Faster R-CNN
Data	Imagenet, CIFAR-10, e.g.	COCO, PASCAL VOC
Scalar loss function (Objective)	CE loss + Regularization	SSD Loss, Faster R-CNN Loss, (both very complicated)
Optimization Algorithm and Hyperparameters	Stochastic Gradient Descent (SGD), Adam, Momentum, etc.	

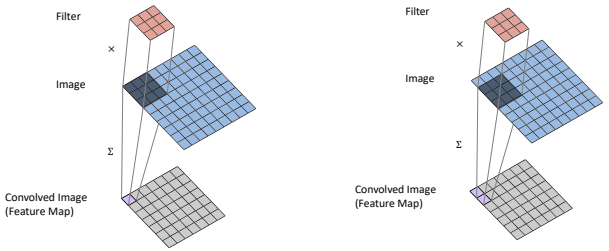
Review: 2D Convolution



Review: 2D Convolution



Review: 2D Convolution



CNN Convolution Parameters

CNN — Parameters

- **Filters**: Represents the amount of filters in a CL.
- **Kernel Size**: Defines the dimensions of the filters.
- **Stride**: Sets the size of the filter shift step.
- **Padding**: defines whether or not there is entry zeroing, influencing the output dimensions:

CNN Convolution Parameters

Review of Useful Properties

Recall that we have some important properties that are useful in object detection

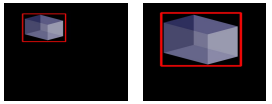
- **Translation Invariance**

This is addressed with convolutional features.

- **Rotation Invariance**

We handle this by data augmentation and other techniques in the dataset.

- **Scale Invariance**



We handle by data diversity and rescaling feature maps.

Some are built into the algorithm, and some come from the structure of the dataset

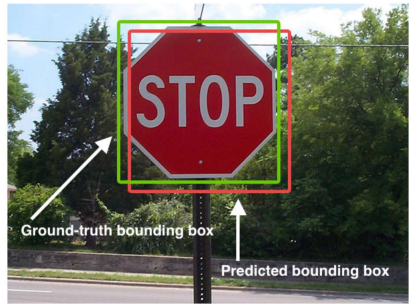
We need a metric to evaluate our model

Classification – did we choose the correct class?

Object detection – is our object correct?

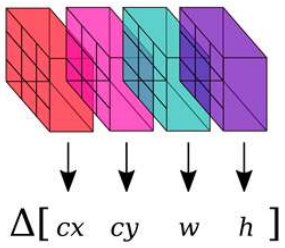
How do we determine if we got the correct location?

We want the bounding box to be good enough.



Detection and Regression

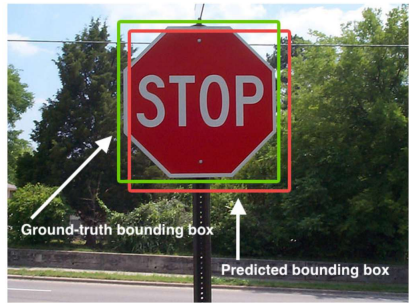
- Finding the bounding box can be thought of as a "**regression problem**".
- Given the image, we must find the center coordinates, width, and height of the box.
- This means we need to output a 4-dimensional vector corresponding to the box for each object of interest.
- We need to understand what is in the box too.
- Need a measure of how good the box is.



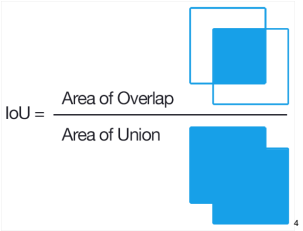
Intersection-over-Union

- The **intersection-over-union** metric is one way of defining this.
- Compute the **Intersection-the** common area covered by the ground-truth bounding box and the predicted bounding box.
- Compute the **Union-the** total area covered by either the ground-truth bounding box and predicted bounding box.

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}}$$



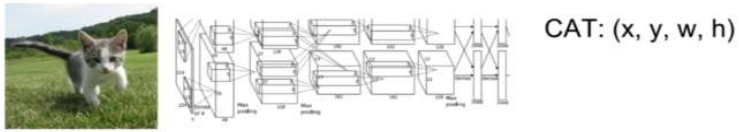
Intersection-over-Union



Intersection-over-Union

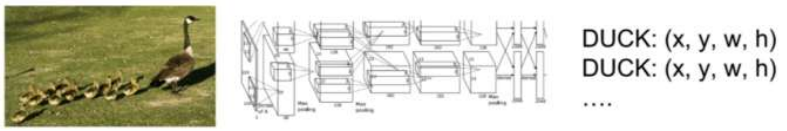
From Wikipedia

The idea



- Images are fed through a convolutional feature extractor.
- Bounding box regression and classification occur in one stage relative to a set of anchor boxes.
- This is a new step that we must figure out.

The idea



We might not necessarily draw just one bounding box in an object detection case, there could be many bounding boxes representing different objects of interest within the image and **we would not know how many beforehand**.

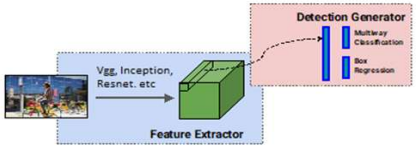
The Major Barrier to Overcome

- The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that
- The length of the output layer is variable — not constant, this is because the number of occurrences of the objects of interest is not fixed.
- A naive approach to solve this problem would be to take different regions of interest from an image and use a CNN to classify the presence of the object within that region.
- The problem with this approach is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, we would have to select a huge number of regions and this could computationally blow up.
- Therefore, algorithms like R-CNN, YOLO etc., have been developed to find these occurrences and find them fast.

FROM CONVOLUTIONAL FEATURES TO OBJECT DETECTION

Object Detection Algorithm

- Images are fed through a convolutional feature extractor.
- Bounding box regression and classification occur in one stage relative to a set of anchor boxes.
- This is a new step that we have to figure out.



R-CNN

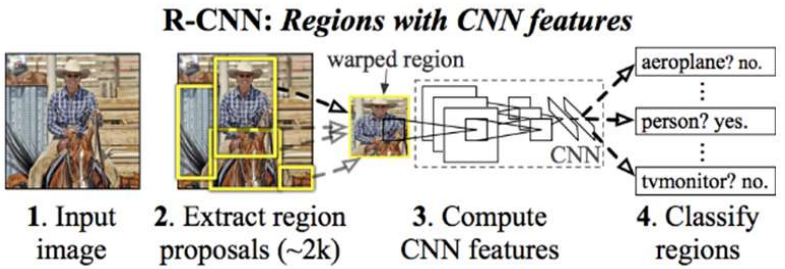
- To bypass the problem of selecting a huge number of regions, Girshick proposed a method where a selective search is used to extract just 2000 regions from the image (These regions are called ***region proposals***).
- Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions.
- These 2000 region proposals are generated using the **selective search algorithm**:
- Selective Search (High Level Description):
  1. Generate initial sub-segmentation, we generate many candidate regions
  2. Use greedy algorithm to recursively combine similar regions into larger ones
  3. Use the generated regions to produce the final candidate region proposals

Details are given in
https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013/UijlingsIJCV2013.pdf

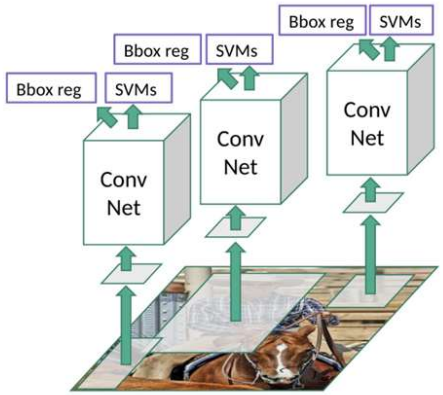
R-CNN

- These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output.
- The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into a support vector machine (SVM) to classify the presence of the object within that candidate region proposal.
- In addition to predicting the presence of an object within the region proposals, the algorithm also predicts **four** values which are offset values to increase the precision of the bounding box.
- For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half.
- Therefore, the offset values help in adjusting the bounding box of the region proposal.

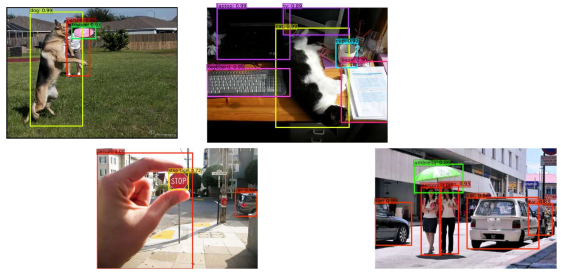
R-CNN



R-CNN



Sample Detections



R-CNN

- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

FAST R-CNN

- The approach is similar to the R-CNN algorithm.
- Instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map.
- From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a Region of Interest (RoI) pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer.

VGG16: Very Deep Convolutional Networks for Large-Scale Image Classification"

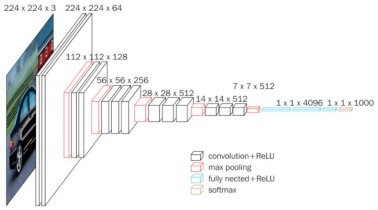
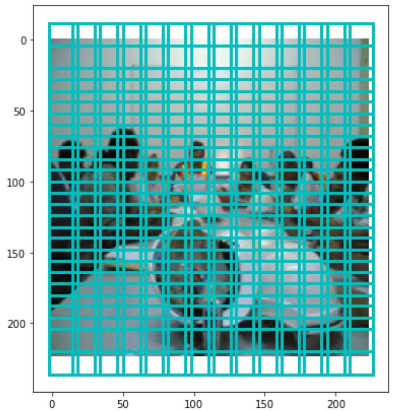


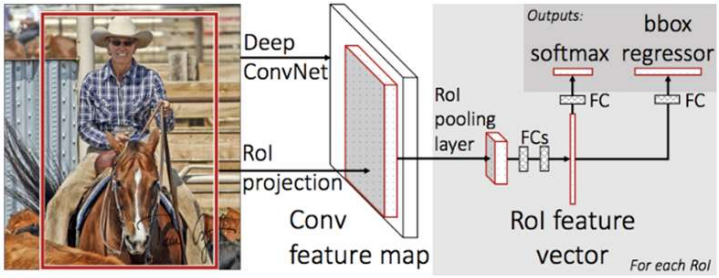
Image copyright Simonyan & Zisserman, 2015

- Input normalized to 224×224 pixels, 3 color channels.
- Last convolutional layer is 14×14 pixels, 512 channels. Call this $\vec{f}[m, n]$, where $\vec{f} \in \Re^{512}$, $0 \leq (m, n) \leq 13$.
- Output FCN trained for object recognition: 1000 different object types.

Last convolutional layer contains 196 features



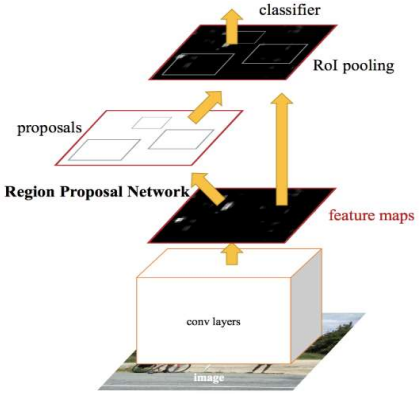
Fast R-CNN



FASTER R-CNN

- Both R-CNN & Fast R-CNN use selective search to find out the region proposals.
- Selective search is a slow and time-consuming process affecting the performance of the network.
- Ren et al, came up with an object detection algorithm that eliminates the selective search algorithm and lets the network learn the region proposals.

Faster R-CNN



FASTER R-CNN

- Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map.
- Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals.
- The predicted region proposals are then reshaped using a RoI pooling layer.
- This is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

VGG16: Very Deep Convolutional Networks for Large-Scale Image Classification"

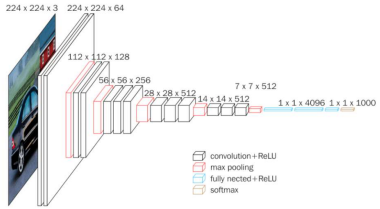


Image copyright Simonyan & Zisserman, 2015

- Input normalized to 224×224 pixels, 3 color channels.
- Last convolutional layer is 14×14 pixels, 512 channels. Call this $\vec{f}[m,n]$, where $\vec{f} \in \Re^{512}$, $0 \leq (m,n) \leq 13$.
- Output FCN trained for object recognition: 1000 different object types.

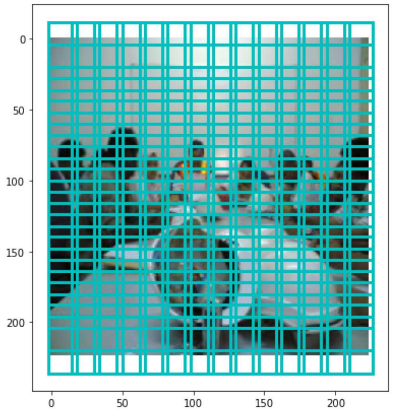
- Faster RCNN assumes that the original image is 1064×1064 pixels, which is then downsampled to the 224×224-pixel size required as input to VGG16.
- There are 4 layers of max pooling before the last conv layer, so each feature vector in the last conv layer represents

$$\left(2^4\left(\frac{1064}{224}\right)\right) \times \left(2^4\left(\frac{1064}{224}\right)\right) = 76 \times 76 \ \frac{\text{input pixels}}{\text{feature vector}}.$$

- The last conv layer contains

$$\left(\frac{224}{2^4}\right) \times \left(\frac{224}{2^4}\right) = 14 \times 14 = 196 \text{ feature vectors.}$$

Last convolutional layer contains 196 features



ROI = 3×3 grid of VGG16 feature vectors

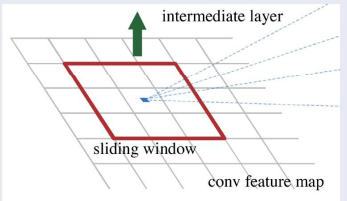


Image copyright Ren, He, Girshick & Sun, 2016

The region proposal network takes, as input, the concatenation of nine neighboring feature vectors from the VGG16 layer:

$$\vec{x}_{m,n} = \begin{bmatrix} \vec{f}[m-1,n-1] \\ \vec{f}[m-1,n] \\ \vdots \\ \vec{f}[m+1,n+1] \end{bmatrix}$$

Notice, we could think of this as another convolutional layer, but Ren et al. treat it as $14 \times 14 = 196$ different FCNs.

Features and Original Image

The $(m,n)^{\text{th}}$ feature vector, $\vec{f}_{m,n}$, covers a particular block of pixels in the input image:

$$(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI}) = (76n, 76m, 228, 228)$$

- Each $\vec{x}[m,n]$ covers 76×76 input pixels.
- Each $\vec{f}_{m,n}$ is $(3 \cdot 76) \times (3 \cdot 76) = 228 \times 228$.
- $m \rightarrow y$ is the vertical axis, $n \rightarrow x$ horizontal.

Suppose the nearest true object is in rectangle $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$. We want to somehow encode the difference between where we are now $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$ and where we want to be $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$. Fast RCNN does this using the following target vector, $\vec{y}_r$, for the neural network:

$$\vec{y}_r = \begin{bmatrix} \frac{x_{REF} - x_{ROI}}{w_{ROI}} \\ \frac{y_{REF} - y_{ROI}}{h_{ROI}} \\ \ln\left(\frac{w_{REF}}{w_{ROI}}\right) \\ \ln\left(\frac{h_{REF}}{h_{ROI}}\right) \end{bmatrix}$$

The neural net is trained to find a $\hat{y}_r$ that is as close as possible to $\vec{y}_r$ (minimum MSE).

Training a bounding box regression network

The network is now trained with two different outputs, $\hat{y}_c$ and $\hat{y}_r$. The total loss is

$$\mathcal{L} = \mathcal{L}_c + \mathcal{L}_r$$

where $\mathcal{L}_c$ is BCE for the classifier output:

$$\mathcal{L}_c = -\frac{1}{n}\sum_{i=1}^{n}(y_{c,i}\ln\hat{y}_{c,i} + (1-y_{c,i})\ln(1-\hat{y}_{c,i}))$$

and $\mathcal{L}_r$ is zero if $y_c = 0$ (no object present), and MSE if $y_c = 1$:

$$\mathcal{L}_r = \frac{1}{2n}\sum_{i=1}^{n} y_{c,i}\|\vec{y}_{r,i} - \hat{y}_{r,i}\|^2$$

Summary of Approach

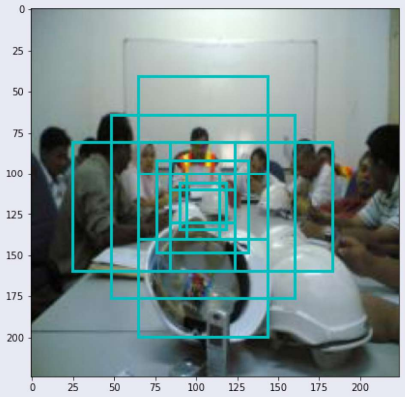
- An ROI network has a 4608d input, corresponding to a 3×3 grid of 512d feature vectors from the last conv layer of a VGG16 object recognizer.
- Faster-RCNN defines 9 different anchors centered on each ROI.
- W.r.t. each anchor, we define the classification target $y_c = 1$ if $IOU > 0.7$, otherwise $y_c = 0$.
- If $y_c = 1$, then we define a regression target $\vec{y}_r$, specifying how much the REF bbox differs from the anchor.

3 sizes, 3 aspect ratios

The Faster RCNN paper described 9 anchors per ROI:

- 3 different anchor sizes: 128×128, 256×256, and 512×512.
- 3 different aspect ratios: $1:2$, $1:1$, and $2:1$

9 anchors per ROI



"Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," Ren, He, Girshick & Sun, 2016

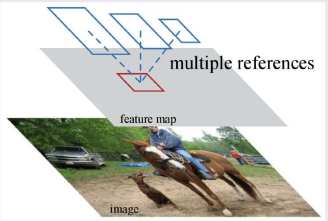


Image copyright Ren, He, Girchick & Sun, 2016

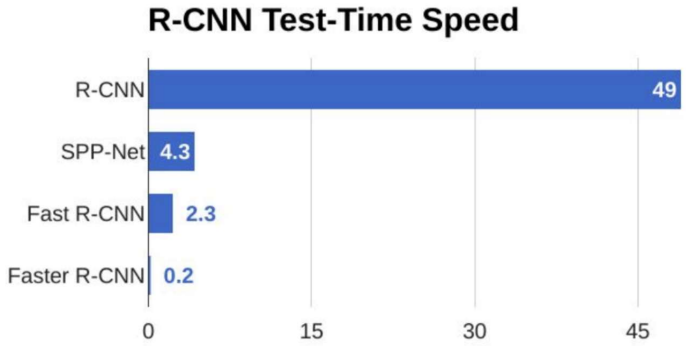
- Each candidate bounding box computes 9 different regression outputs, each of which is a 4-vector (x,y,w,h)
- The 9 different regression outputs from each bbox are w.r.t. 9 different "anchor" rectangles, each offset from the input ROI. Thus:

$$\text{anchor} = \text{ROI} + \text{known shift}$$
$$\text{object} = \text{anchor} + \text{regression}$$

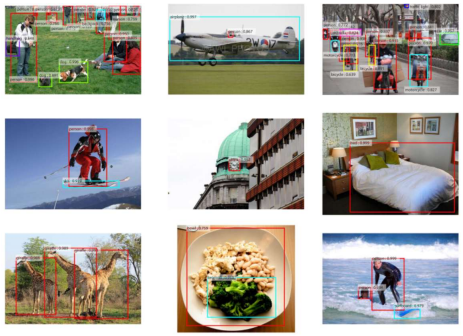
- The ROI is $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$.
- The anchor is (x_a, y_a, w_a, h_a).
- The true object is located at $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$.
- The regression target is:

$$\vec{y}_r = \begin{bmatrix} \frac{x_{REF} - x_a}{w_a} \\ \frac{y_{REF} - y_a}{h_a} \\ \ln\left(\frac{w_{REF}}{w_a}\right) \\ \ln\left(\frac{h_{REF}}{h_a}\right) \end{bmatrix}$$

Speeds



Sample Detections



How can this be learned?

- Object detection adds *a lot* of complexity compared to a CNN
- Many practical details become important for effective training
- Please read the papers if you go to implement this yourself

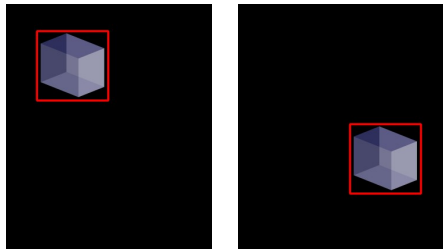
Let's go back to our goals:

We have some important properties that are useful in object detection

- Translation Invariance
- Scale Invariance
- Rotation Invariance

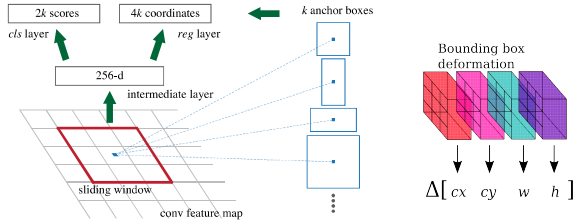
Some are built into the algorithm, and some come from the structure of the dataset

Translation Invariance

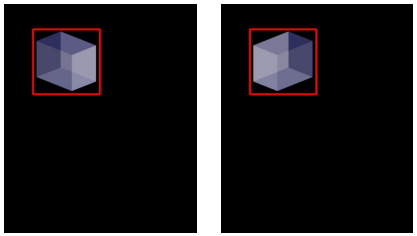


This is addressed with convolutional features…

Translation invariance comes from the sliding windows (same as conv.)

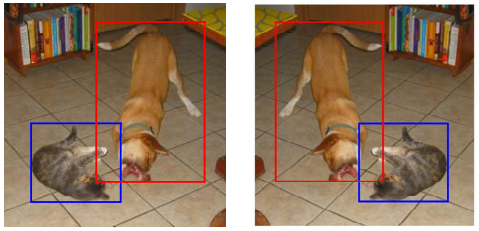


Rotation Invariance



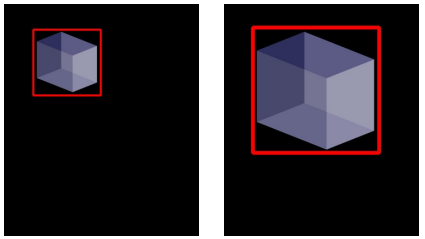
handled by data augmentation and other techniques in the dataset.

Data Augmentation



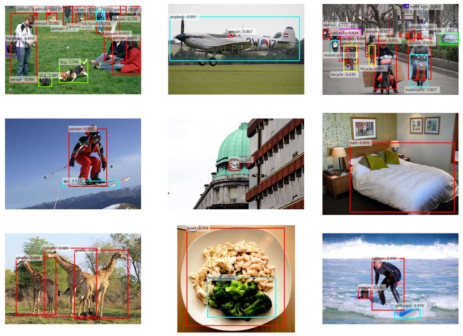
Feed in mirros, rotations, shifts, etc.

Scale Invariance



handled by data diversity and rescaling feature maps.

Sample Detections



How do we measure performance

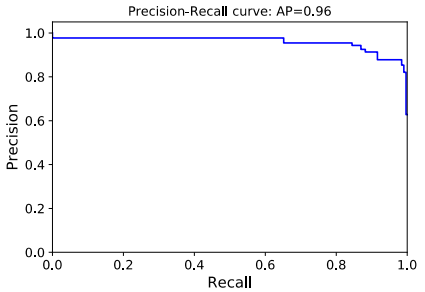
- We use mean average precision (mAP) over all our classes
- Must use a precision-recall metric *because there are so many ways to get things wrong*

Types of Predictions

	Predict Negative	Predict Positive
True Label is Positive	False Negatives (FN)	True Positives (TP)
True Label is Negative	True Negatives (TN)	False Positives (FP)

Precision is given by (TP)/(TP+FP).
Recall is given by (TP)/(TP+FN).

Precision-Recall



Average Precision is the integral of this curve.

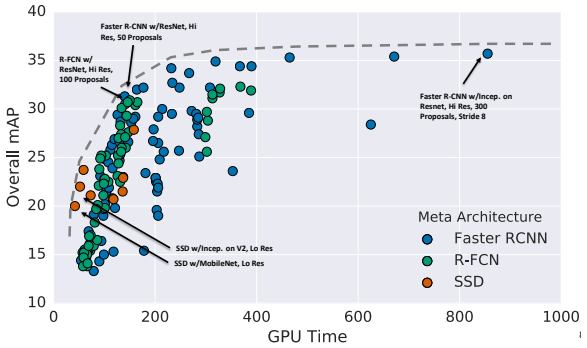
Mean Average Precision

- Average Precision (AP) isn't perfect. Nevertheless, it works well for a single class.
- When we have multiple classes, we can take the "Mean Average Precision," which is the average precision over multiple classes:
- $$mAP\ (or\ MAP) = \ mean\ (AP\ for\ each\ class).$$
- Other common metrics include:

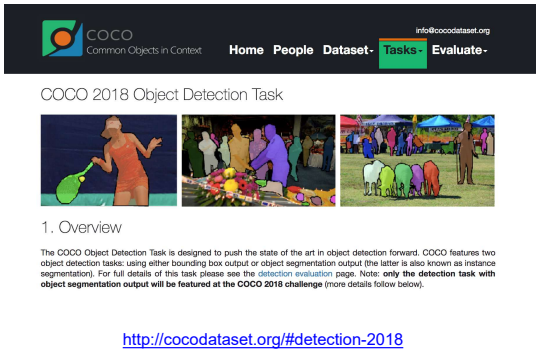
$$F_1 = 2\ (Precision * Recall)/(Precision + Recall)$$

- MRR = mean reciprocal rank (useful when only one answer is appropriate out of many)

Performance Tradeoffs



Where is object detection going?



http://cocodataset.org/#detection-2018

Resources for training your own object detector

Object detection in PyTorch
https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection