Parameter Norm Penalties

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
6. Parameter tying and parameter sharing
7. Sparse representations
8. Bagging and other ensemble methods
9. Dropout
10. Adversarial training
11. Tangent methods

Topics in Parameter Norm Penalties

1. Overview (limiting model capacity)
2. L^2 parameter regularization
3. L^1 regularization

Limiting Model Capacity

- Regularization has been used for decades prior to advent of deep learning
- Linear- and logistic-regression allow simple, straightforward and effective regularization strategies
  - Adding a parameter norm penalty $\Omega(\theta)$ to the objective function J :

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

    - where $\alpha\varepsilon[0,\theta)$ is a hyperparameter that weight the relative contribution of the norm penalty term Ω
      - Setting α to 0 results in no regularization. Larger values correspond to more regularization

Norm Penalty

- When our training algorithm minimizes the regularized objective function

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

  - it will decrease both the original objective J on the training data and some measure of the size of the parameters θ
- Different choices of the parameter norm Ω can result in different solutions preferred
  - We discuss effects of various norms

No penalty for biases

- Norm penalty Ω penalizes only weights at each layer and leaves biases unregularized
  - Biases require less data to fit than weights
  - Each weight specifies how variables interact
    - Fitting weights requires observing both variables in a variety of conditions
- Each bias controls only a single variable
  - We do not induce too much variance by leaving biases unregularized
- $\boldsymbol{w}$ indicates all weights affected by norm penalty
- $\boldsymbol{\theta}$ denotes both $\boldsymbol{w}$ and biases

Different or Same αs for layers?

- Sometimes it is desirable to use a separate penalty with a different α for each layer

$$\boxed{J^{!}(\theta;X,y) = J(\theta;X,y) + \alpha\Omega(\theta)}$$

- Invariance under linear transformation $\mathbf{T}$ is one case
  - i.e., we want neural net to perform the same when the inputs are transformed

- But this creates too many hyperparameters
  - Search space reduced by using same hyperparameters

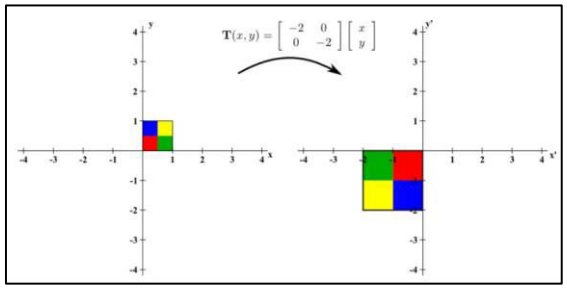
Linear Transformation T

- Consider a simple linear transformation of the input

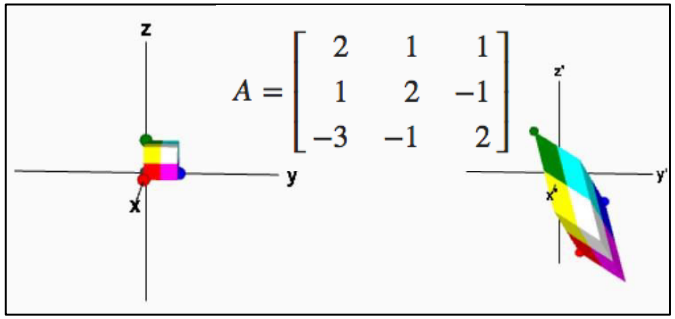
Two-variables
x and y

$$\mathbf{T}(x, y) = (ax + by, cx + dy) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{T}(\mathbf{x}) = A\mathbf{x}$$



Three variables *x,y* and *z*



Weight decay and invariance

- Suppose we train two 2-layer networks
  - *First network*: trained using original data: $\boldsymbol{x}=\{x_i\}$, $\boldsymbol{y}=\{y_k\}$
  - *Second network*: input and/or target variables are transformed by one of the linear transformations

$$x_i \rightarrow x'_i = ax_i + b \qquad y_k \rightarrow y'_k = cy_k + d$$

- Consistency requires that we should obtain equivalent networks that differ only by linear transformation of the weights

For first layer:

$$w_{ji} \rightarrow \frac{1}{a} w_{ji}$$

and/or

And/or or second layer:

$$w_{kj} \rightarrow c w_{kj}$$

Simple weight decay fails invariance

- Simple weight decay $\tilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{\alpha}{2}\boldsymbol{w}^{\mathsf{T}}\boldsymbol{w}$
- Treats all weights and biases on equal footing
  - While resulting w_{ji} and w_{kj} should be treated differently
  - Consequently networks will have different weights and violate invariance
- We therefore look for a regularizer invariant under the linear transformations
  - Such a regularizer is $\frac{\alpha_1}{2}\sum_{w\in W_1} w^2 + \frac{\alpha_2}{2}\sum_{w\in W_2} w^2$
    - where W_1 are weights of first layer and
    - W_2 are the set of weights in the second layer
      - This regularizer remains unchanged under the weight transformations provided the parameters are rescaled using

$$\lambda_1 \rightarrow a^{1/2}\lambda_1 \quad \text{and} \quad \lambda_2 \rightarrow c^{-1/2}\lambda_2$$

Weight decay used in practice

- Because it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use same weight decay at all layers to reduce search space

L^2 parameter Regularization

- Simplest and most common kind
- Called *Weight decay*
- Drives weights closer to the origin
  - by adding a regularization term to the objectve function

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

- In other communities also known as *ridge regression* or *Tikhonov regularization*

Gradient of Regularized Objective

- Objective function (with no bias parameter)

$$J'(w;X,y) = \frac{\alpha}{2} w^T w + J(w;X,y)$$

- Corresponding parameter gradient

$$\nabla_w J'(w;X,y) = \alpha w + \nabla_w J(w;X,y)$$

- To perform single gradient step, perform update:

$$w \leftarrow w - \varepsilon \left(\alpha w + \nabla_w J\left(w; X, y\right)\right)$$

- Written another way, the update is

$$w \leftarrow (1 - \varepsilon\alpha) w - \varepsilon \nabla_w J\left(w; X, y\right)$$

  - We have modified learning rule to shrink $\boldsymbol{w}$ by constant factor $1\text{-}\varepsilon\alpha$ at each step

To study effect on entire training

- Make quadratic approximation to the objective function in the neighborhood of minimal unregularized cost $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$
- The approximation is given by

  $J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T H(\boldsymbol{w} - \boldsymbol{w}^*)$
- Where H is the Hessian matrix of J wrt $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$

Effect of L^2 regularization on optimal w

Objective function: $J'(w;X,y) = \frac{\alpha}{2} w^T w + J(w;X,y)$

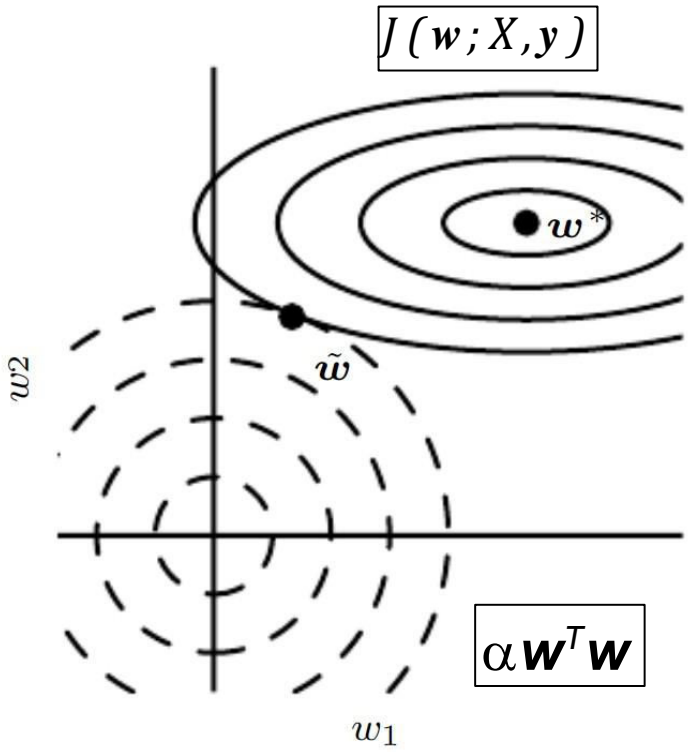
Solid ellipses:

contours of equal value of unregularized objective $J(w;X,y)$

Dotted circles:

contours of equal value of L^2 regularizer $\alpha \boldsymbol{w}^T \boldsymbol{w}$

At point $\tilde{\boldsymbol{w}}$ competing objectives reach equilibrium



Along w_1, eigen value of Hessian of J is small. J does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because J does not have a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to 0.

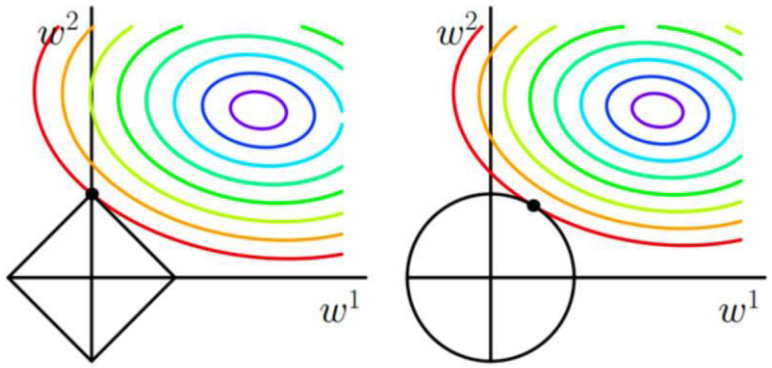
Along w_2, J is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little

L^1 Regularization

- L^2 weight decay is common weight decay
- Other ways to penalize model parameter size
- L^1 regularization is defined as

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

  – which sums the absolute values of parameters



(a) ℓ_1-ball meets quadratic function. ℓ_1-ball has corners. It's very likely that the meet-point is at one of the corners.

(b) ℓ_2-ball meets quadratic function. ℓ_2-ball has no corner. It is very unlikely that the meet-point is on any of axes."

Image Source: https://zhuanlan.zhihu.com/p/28023308

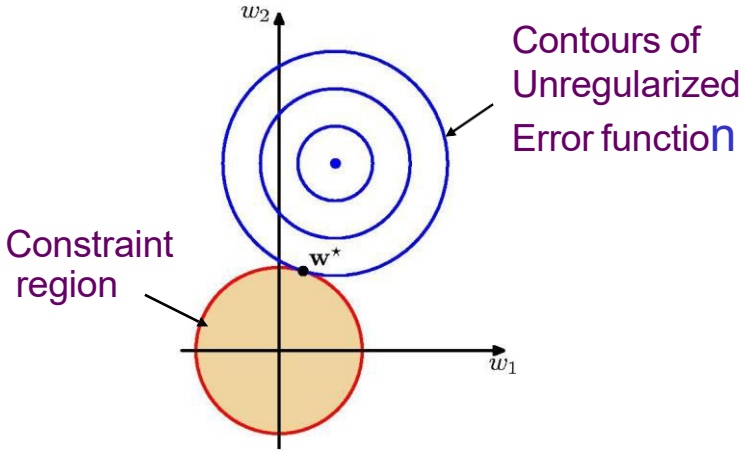
Sparsity and Feature Selection

- The sparsity property induced by L^1 regularization has been used extensively as a feature selection mechanism
  - Feature selection simplifies an ML problem by choosing subset of available features
- LASSO (Least Absolute Shrinkage and Selection Operator) integrates an L^1 penalty with a linear model and least squares cost function
  - The L^1 penalty causes a subset of the weights to become zero, suggesting that those features can be discarded

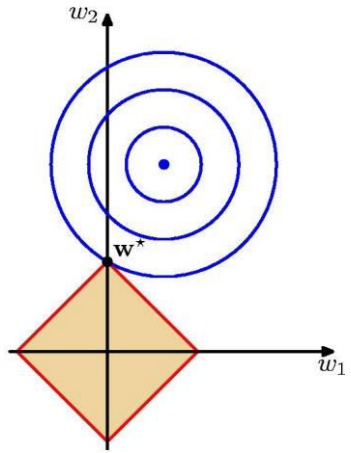
Sparsity with Lasso constraint

- With $q=1$ and λ is sufficiently large, some of the coefficients w_j are driven to zero
- Leads to a sparse model
  - where corresponding basis functions play no role
- Origin of sparsity is illustrated here:

Quadratic solution where w_1^* and w_0^* are nonzero



Minimization with Lasso Regularizer
A sparse solution with $w_1^*=0$



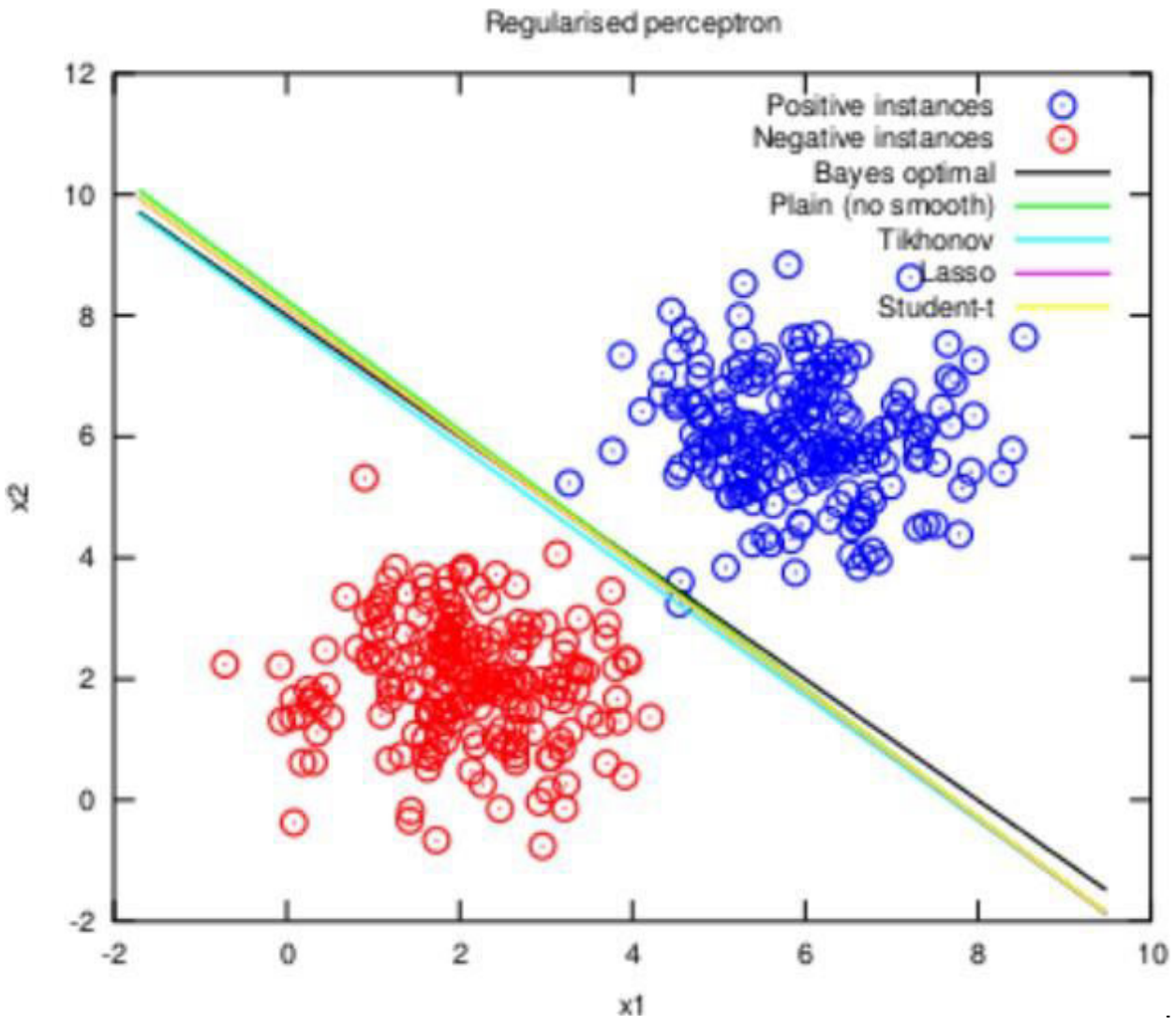
Regularization with linear models

Norm Regularization:

$$Tikhonov: \quad \lambda \sum_{\forall i} w_i^2$$

$$Lasso: \quad \lambda \sum_{\forall i} |w_i|$$

$$Student-t: \quad \lambda \sum_{\forall i} \log(1+w_i^2)$$



Norm Penalties as Constrained Optimization

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
6. Parameter tying and parameter sharing
7. Sparse representations
8. Bagging and other ensemble methods
9. Dropout
10. Adversarial training
11. Tangent methods

Topics in Norm Penalty Optimization

1. Lagrangian formulation
2. KKT multiplier
3. Equivalence to norm penalty
4. Explicit constraints and Reprojection

Constrained Optimization

- Consider the cost function regularized by a norm penalty $\tilde{J}(\theta;X,y) = J(\theta;X,y) + \alpha\Omega(\theta)$
- Recall we can minimize a function subject to constraints by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties
  - Each penalty is a product between a coefficient called a Karush-Kuhn-Tucker (KKT) multiplier and a function representing whether constraint is satisfied

Lagrange Formulation

- If we wanted to constrain $\Omega(\theta)$ to be less than some constant k, we could construct a generalized Lagrange function

$$L(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha\left(\Omega(\theta) - k\right)$$

- The solution to the constrained problem is given by

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

- Solving this problem requires modifying both θ and α
  - Many different procedures are possible

Insight into effect of constraint

- We can fix α^* and view the problem as just a function of θ:

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

- This is exactly the same as the regularized training problem of minimizing $\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$
- We can thus think of the parameter norm penalty as imposing a constraint on the weights

How α influences weights

- If Ω is L^2 norm
  - weights are then constrained to lie in an L^2 ball
- If Ω is the L^1 norm
  - Weights are constrained to lie in a region of limited L^1 norm
- Usually we do not know size of constraint region that we impose by using weight decay with coefficient α^* because the value of α^* does not directly tell us the value of k
  - Larger α will result in smaller constraint region
  - Smaller α will result in larger constraint region

Reprojection

- Sometimes we may wish to use explicit constraints rather than penalties
  - We can modify SGD to take a step downhill on $J(\theta)$ and then project θ back to the nearest point that satisfies $\Omega(\theta) < k$
  - This useful when we have an idea of what values of k is appropriate and we do not want to spend time searching for the value of α that corresponds to this k
- Rationale for explicit constraints/Reprojection

1. Dead weights
2. Stability

1. Eliminating dead weights

- A reason to use explicit constraints and reprojection rather than enforcing constraints with penalties:
  - Penalties can cause nonconvex optimization procedures to get stuck in local minima corresponding to small θ
    - This manifests as training with *dead units*
  - Explicit constraints implemented by reprojection can work much better because they do not encourage weights to approach the origin

2. Stability of Optimization

- Explicit constraints with reprojection can be useful because thse impose some stability on the optimization procedure
- When using high learning rates, it is possible to encounter a positive feedback learning loop in which large weights induce large gradients, which then induce a large update of the weights
  - Can lead to numerical overflow
- Explicit constraints with reprojection prevent this feedback loop from continuing to increase magnitudes of weights without bound

Regularization and Underconstrained Problems

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
6. Parameter tying and parameter sharing
7. Sparse representations
8. Bagging and other ensemble methods
9. Dropout
10. Adversarial training
11. Tangent methods

Topics in Underconstrained Problems

1. Need for proper definition of ML problems
    1. Underconstrained closed-form solution
    2. Underconstrained iterative solution
2. Regularization for Logistic Regression with linearly separable classes
3. Moore Penrose inverse

Under constrained closed-form linear regression

- Regularization is often necessary for a machine learning problem to be properly defined
- Many linear models including PCA and linear regression depend on inverting the $\Phi^T\Phi$ matrix

$$y(\mathbf{x},\mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$$

$$\mathbf{w}_{ML} = \Phi^{+}\mathbf{t}$$

$$\Phi^{+} = (\Phi^T\Phi)^{-1}\Phi^T$$

$$\Phi = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \dots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & & & \\ & & & \\ \phi_0(\mathbf{x}_N) & & & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}$$

Samples X Dimensions

- This is not possible when $\Phi^T\Phi$ is singular
  - This happen with fewer samples than dimensions

Underconstrained logistic regression

- Logistic regression with linearly separable classes
  - Task is under-determined

$$\boxed{\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} - \eta \nabla E_n} \qquad \boxed{\nabla E_n = (y_n - t_n)\phi_n} \qquad \text{where } y_n = \sigma(\boldsymbol{w}^{\mathrm{T}}\phi_n)$$

- If $\boldsymbol{w}$ separates perfectly, so will $2\boldsymbol{w}$ and with higher likelihood
  - SGD will continually increase $\boldsymbol{w}$ and will never halt
    - Solution left to the overflow handler

Solution for Undercontrained Iterative

- Logistic regression with linearly separable classes is under-determined

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} - \eta \nabla E_n \qquad \nabla E_n = (y_n - t_n)\phi_n$$

where $y_n = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}_n)$

- Solution: Most forms of regularization guarantee convergence of iterative methods
  - e.g., weight decay

$$\nabla E_n = -\sum_{n=1}^{N} \{ t_n - \mathrm{w}^{\tau}\boldsymbol{\varphi}(\mathrm{x}_n) \} \boldsymbol{\varphi}(\mathrm{x}_n)^{\tau}$$

without regularization

$$\nabla E_n = \left[-\sum_{n=1}^{N} \{ t_n - \mathrm{w}^{\tau}\boldsymbol{\varphi}(\mathrm{x}_n) \} \boldsymbol{\varphi}(\mathrm{x}_n)^{\tau} \right] + \lambda \mathrm{w}$$

with regularization

  - Stops when slope of likelihood equals weight decay coefficient

Regularization in linear algebra problems

- Several basic linear algebra problems use regularization
- We can solve underdetermined linear equations using Moore Penrose inverse

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

- Recall closed-form solution to linear regression with design matrix ϕ

$$\Phi^{+}=(\Phi^{T}\Phi)^{-1}\Phi^{T}$$

Data Set Augmentation

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
6. Parameter tying and parameter sharing
7. Sparse representations
8. Bagging and other ensemble methods
9. Dropout
10. Adversarial training
11. Tangent methods

Topics in Data Augmentation

1. More data is better
2. Augmentation for classification
3. Caution in data augmentation
4. Injecting noise
5. Benchmarking using augmentation
6. Ex: Heart disease diagnosis using Deep Learning

More data is better

- Best way to make a ML model to generalize better is to train it on more data
- In practice amount of data is limited
- Get around the problem by creating synthesized data
- For some ML tasks it is straightforward to synthesize data

Augmentation for classification

- Data augmentation is easiest for classification
  - Classifier takes high-dimensional input $\boldsymbol{x}$ and summarizes it with a single category identity y
  - Main task of classifier is to be invariant to a wide variety of transformations
- Generate new samples $(\boldsymbol{x}, y)$ just by transforming inputs
- Approach not easily generalized to other problems
  - For density estimation problem
    - it is not possible generate new data without solving density estimation

Effective for Object Recognition

- Data set augmentation very effective for the classification problem of object recognition
- Images are high-dimensional and include a variety of variations, may easily simulated
- Translating the images a few pixels can greatly improve performance
  - Even when designed to be invariant using convolution and pooling
- Rotating and scaling are also effective

Main data augmentation methods

Original	Flip	Rotation	Random crop
			
• Image without any modification	• Flipped with respect to an axis for which the meaning of the image is preserved	• Rotation with a slight angle • Simulates incorrect horizon calibration	• Random focus on one part of the image • Several random crops can be done in a row

Color shift	Noise addition	Information loss	Contrast change
			
• Nuances of RGB is slightly changed • Captures noise that can occur with light exposure	• Addition of noise • More tolerance to quality variation of inputs	• Parts of image ignored • Mimics potential loss of parts of image	• Luminosity changes • Controls difference in exposition due to time of day

Remark: data is usually augmented on the fly during training.

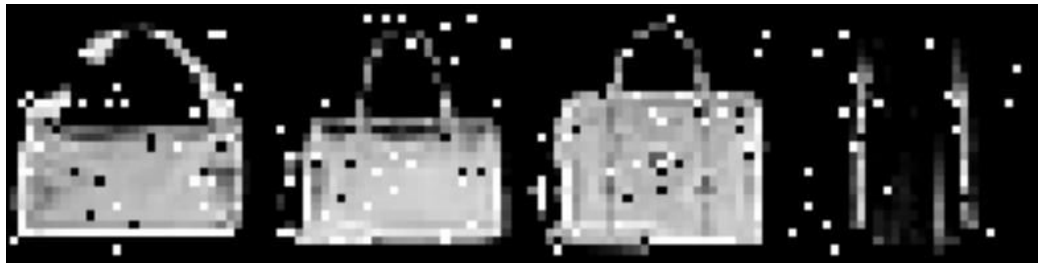
Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

Fashion MNIST Noisy Images

Gaussian Noise

Noise on the objects only and not in the background.

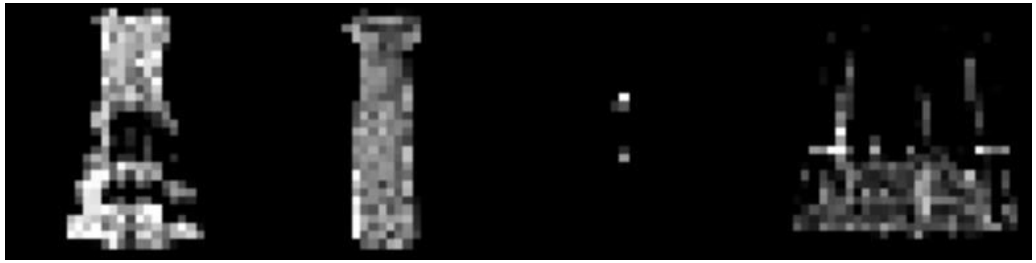
Salt and Pepper Noise



Mixture of black and white noise on objects as well as background.

https://debuggercafe.com/adding-noise-to-image-data-for-deep-learning-data-augmentation/?fbclid=IwAR2KWCN5HExb8EmpW5Z6vsM_7y6n8le6-rcHvEwia6pze3DLM9hZEKU1arc

Speckle Noise



Caution in Data Augmentation

- Not apply transformation that would change the class
- OCR example: ‘b’ vs ‘d’ and ‘6’ vs ‘9’
  - Horizontal flips and 180 degree rotations are not appropriate ways
- Some transformations are not easy to perform
  - Out of plane rotation cannot be implemented as a simple geometric operation on pixels

Injecting noise

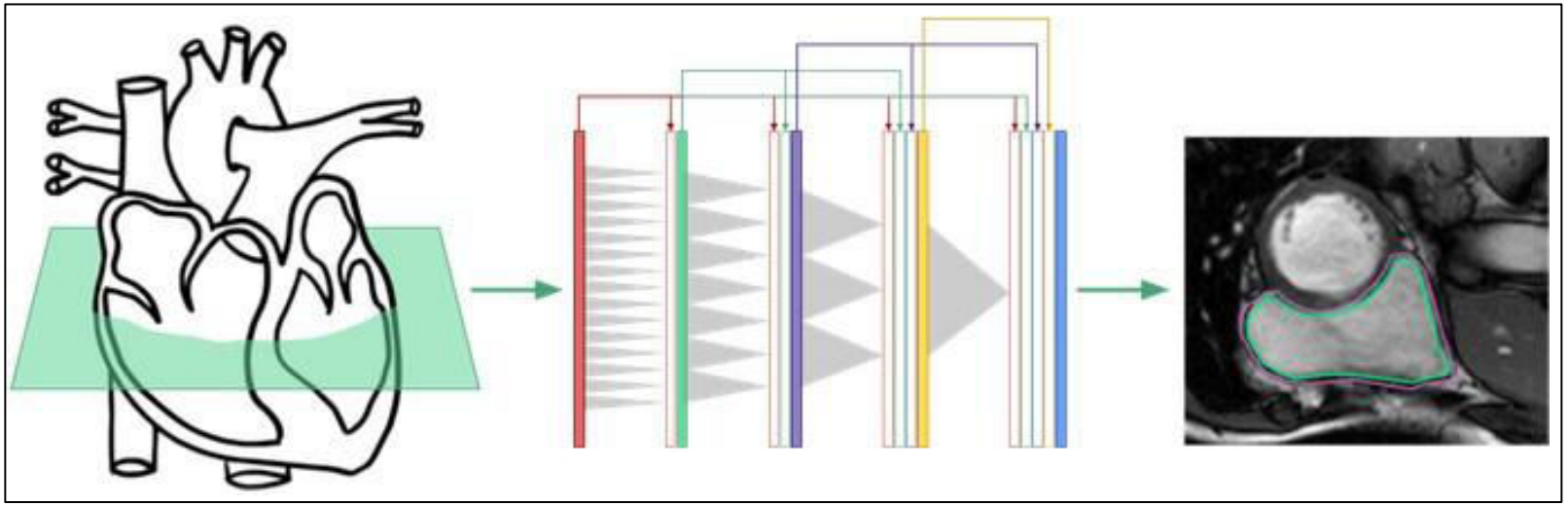
- Injecting noise into the input of a neural network can be seen as data augmentation
- Neural networks are not robust to noise
- To improve robustness, train them with random noise applied to their inputs
  - Part of some unsupervised learning, such as denoising autoencoder
- Noise can also be applied to hidden units
- Dropout, a powerful regularization strategy, can be viewed as constructing new inputs by multiplying by noise

Benchmarking using augmentation

- Hand-designed data set augmentation can dramatically improve performance
- When comparing ML algorithms A and B, same data set augmentation should be used for both
  - If A performs poorly with no dataset augmentation and B performs well with synthetic transformations of the input, reason may be the data set rather than algorithm
- Adding Gaussian noise is considered part of ML while cropping input images is not

Ex: Image segmentation for heart disease

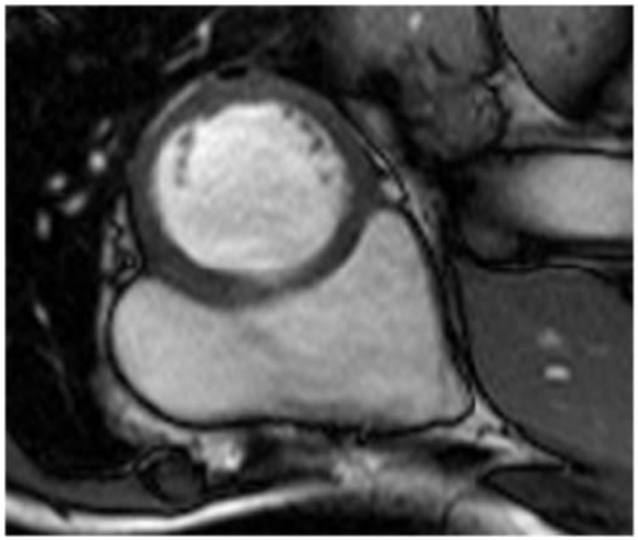
- To determine *ejection fraction*: which measures of how well a heart is functioning
  - After relaxing to its *diastole* so as to fully fill with blood, what percentage is pumped out upon contracting to its *systole?*
    - This metric relies on segmenting right ventricles (RVs) in cardiac magnetic resonance images (MRIs)

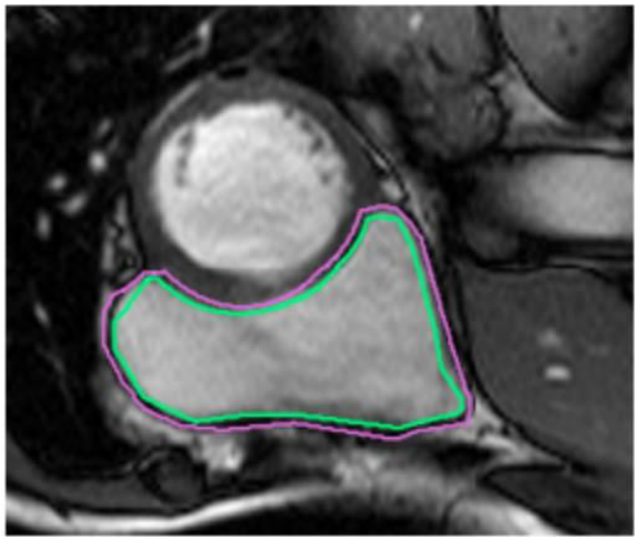


https://blog.insightdatascience.com/heart-disease-diagnosis-with-deep-learning-c2d92c27e730

Problem Description

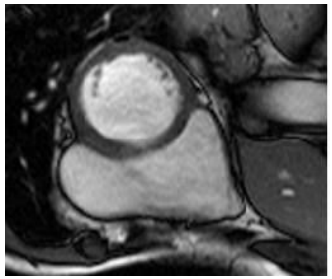
- Develop system to segment RV in cardiac MRI
  - Currently handled by classical image processing
- RV has irregularly shaped thin walls: inner and outer walls (endocardium and epicardium)
  - Manually drawn contours shown:

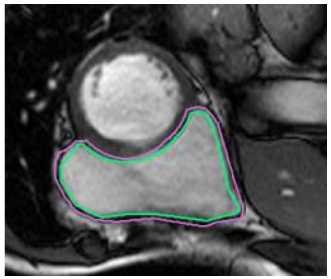


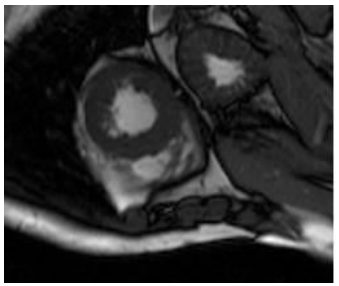


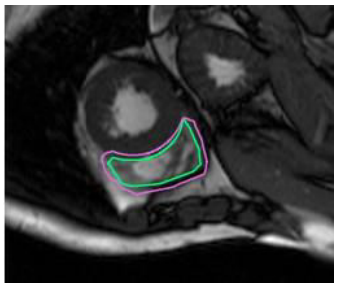
RV segmentation is difficult

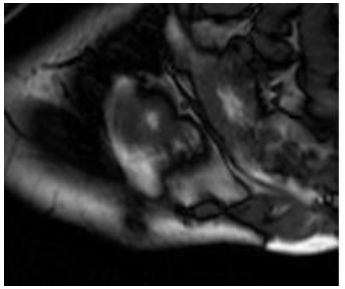
- Left ventricle segmentation is easier
  - LV is a thick-walled circle
    - Kaggle 2016 competition
- Right ventricle segmentation is harder
  - Complex crescent shape
  - Easy and hard cases

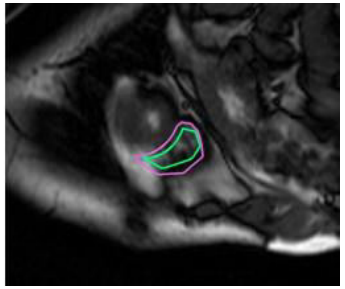










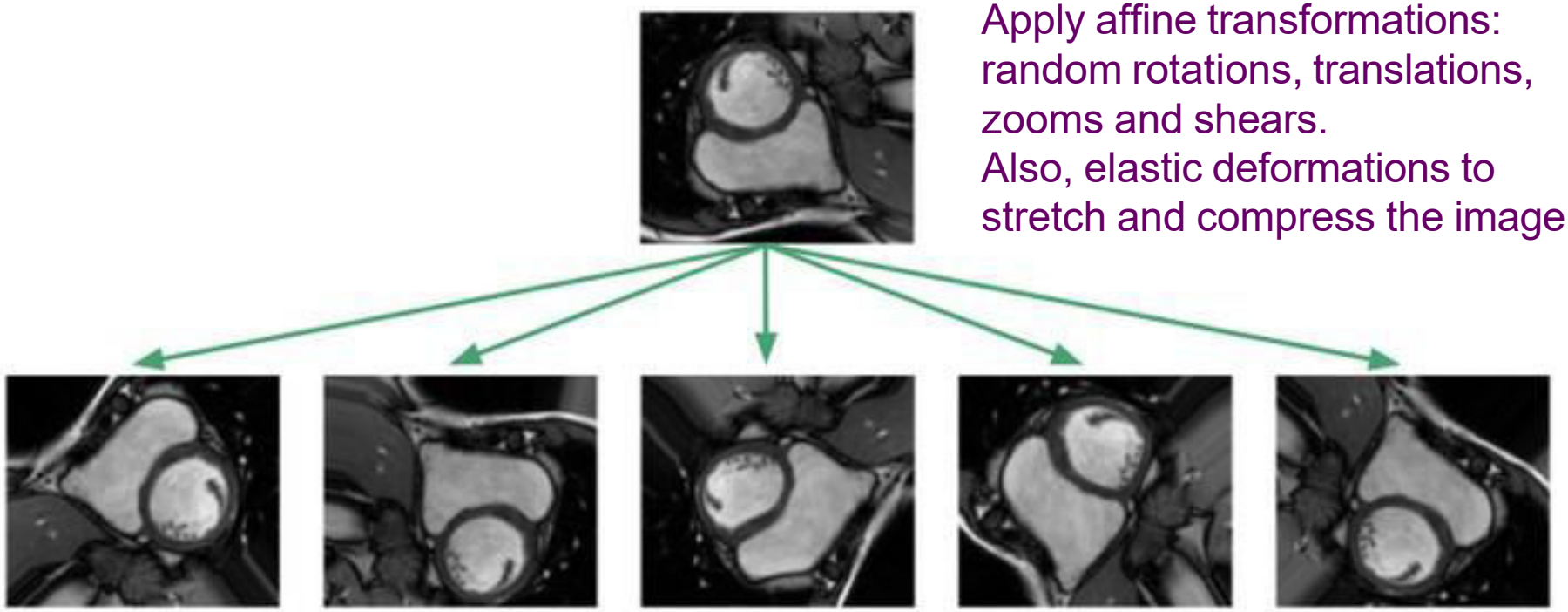


Task: determine whether each pixel is part of RV or not

Need for Data augmentation

- Dataset: 243 physician-segmented images of 16 patients.
  - 3697 additional unlabeled images, useful for unsupervised or semi-supervised techniques
    - Generalization to unseen images would be hopeless!
  - Typical situation in medical settings where labeled data is expensive.

Transformed data



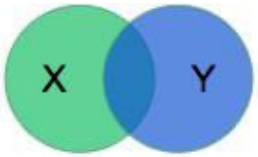
Apply affine transformations:
random rotations, translations,
zooms and shears.
Also, elastic deformations to
stretch and compress the image

Goal: prevent network from memorizing just the training examples, and force it to learn that the RV is a solid, crescent-shaped object in a variety of orientations.

Apply transformations on the fly so the network sees new random transformations during each epoch.

Peformance Evaluation

- Training: 20% of images as validation set
  - RV challenge: separate test set of another 514 MRI images derived from a separate set of 32 patients
- Performance metric
  - The model will output a mask *X* delineating what it thinks is the RV, and the dice coefficient compares it to the mask *Y* produced by a physician via:

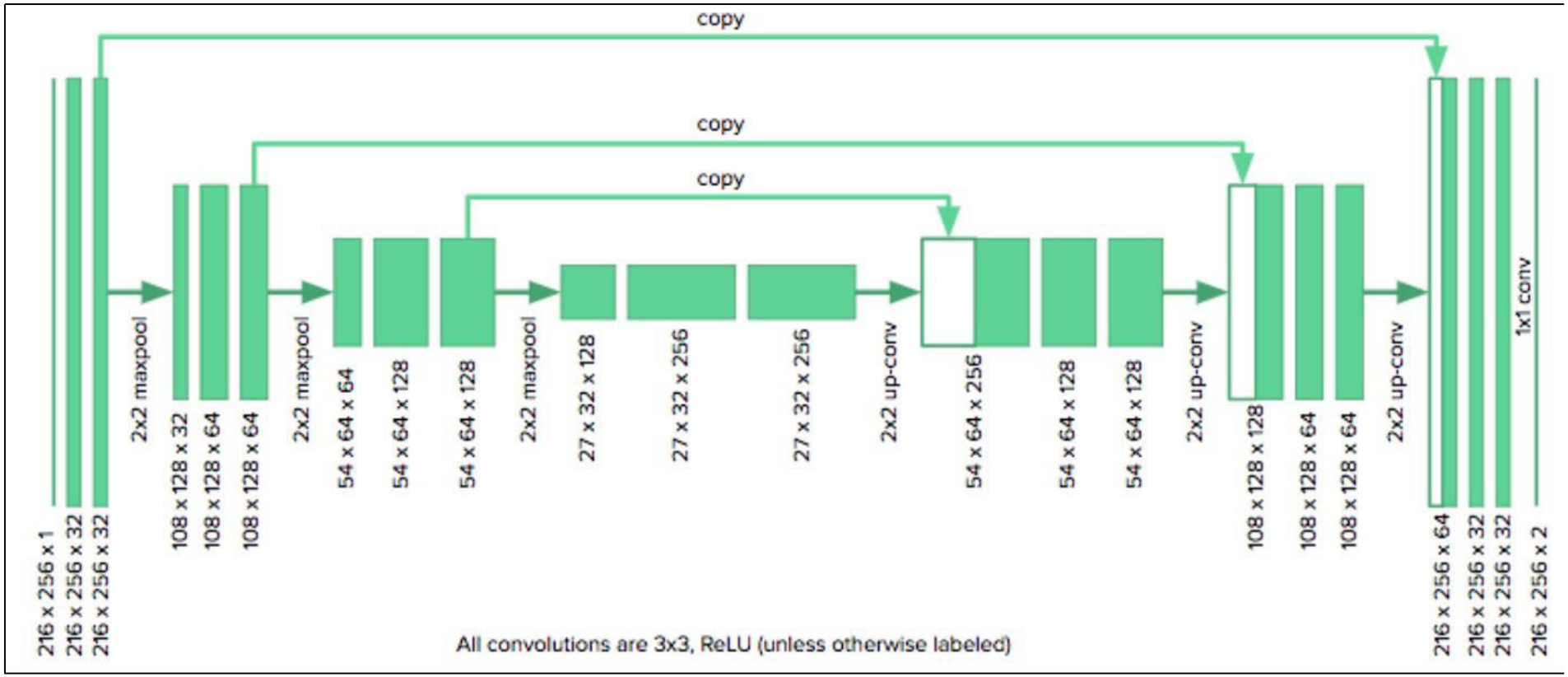


$$\text{dice}(X, Y) = \frac{2X \cap Y}{X + Y}$$

Metric is (twice) the ratio of the intersection over the sum of areas.
It is 0 for disjoint areas, and 1 for perfect agreement.

E.g., model performance is written as 0.82 (0.23), where the parentheses contain the standard deviation.

Deep Learning Architecture



U-net architecture

- Train network with only 30 images using augmentation and pixel-wise reweighting

- It consists of a contracting path, which collapse image into high level features,
- Uses the feature information to construct a pixel-wise segmentation mask.
- Copy and concatenate connections pass information from early feature maps to later portions of the network tasked with constructing the segmentation mask.

Implementation

- Implemented in Keras
  - Code available in Github
    - https://github.com/chuckyee/cardiac-segmentation
- Baseline is fully convolutional network (FCN)
- Endocardium and epicardium performance

Method	Train	Val	Test	Params
Human	–	–	0.90 (0.10)	–
FCN (Tran 2017)	–	–	0.86 (0.20)	~11M
U-net	0.93 (0.07)	0.86 (0.17)	0.77 (0.30)	1.9M
Dilated u-net	0.94 (0.05)	0.90 (0.14)	**0.88 (0.18)**	3.7M
Dilated densenet	0.94 (0.04)	0.89 (0.15)	0.85 (0.20)	**0.19M**

Method	Train	Val	Test	Params
Human	–	–	0.90 (0.10)	–
FCN (Tran 2017)	–	–	**0.84 (0.21)**	~11M
U-net	0.91 (0.06)	0.82 (0.23)	0.79 (0.28)	1.9M
Dilated u-net	0.92 (0.08)	0.85 (0.19)	**0.84 (0.21)**	3.7M
Dilated densenet	0.91 (0.10)	0.87 (0.15)	0.83 (0.22)	**0.19M**

Acknowledgemnts

1. Goodfellow, I., Bengio, Y., and Courville, A. Deep Learning, MIT Press 2016
2. Yee, C-H., "Heart Disease Diagnosis with Deep Learning: State-of-the-art results with 60x fewer parameters" https://blog.insightdatascience.com/heart-disease-diagnosis-with-deep-learning-c2d92c27e730

Noise Robustness

Sargur N. Srihari
srihari@buffalo.edu

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
9. Parameter tying and parameter sharing
10. Sparse representations
11. Bagging and other ensemble methods
12. Dropout
13. Adversarial training
14. Tangent methods

Topics in Noise Robustness

- Noise applied at inputs
- Noise applied to weights
- Injecting noise at the output targets

Noise injection is powerful

- Noise applied to inputs is a data augmentation
  - For some models addition of noise with infinitesimal variance at the input is equivalent to imposing a penalty on the norm of the weights, e.g., $\lambda \boldsymbol{w}^{\mathrm{T}} \boldsymbol{w}$
- Noise applied to hidden units
  - Noise injection can be much more powerful than simply shrinking the parameters
  - Noise applied to hidden units is so important that it merits its own separate discussion
    - Dropout is the main development of this approach

Adding Noise to Weights

- This technique primarily used with RNNs
- This can be interpreted as a stochastic implementation of Bayesian inference over the weights
  - Bayesian learning considers model weights to be uncertain and representable via a probability distribution $p(\boldsymbol{w})$ that reflects that uncertainty,
  - Adding noise to weights is a practical, stochastic way to reflect this uncertainty

Noise at weights encourages stability

- Noise applied to weights is equivalent to traditional regularization, encouraging stability
- This can be seen in a regression setting
  - Train $\hat{y}(\boldsymbol{x})$ to map $\boldsymbol{x}$ to a scalar using least squares between model prediction $\hat{y}(\boldsymbol{x})$ and true values y :

$$J = E_{p(\boldsymbol{x},y)}\left[\left(\hat{y}(\boldsymbol{x}) - y\right)^2\right]$$

    - Training set: m labeled samples $\{(\boldsymbol{x}^{(1)},y^{(1)}),\ldots,(\boldsymbol{x}^{(m)},y^{(m)})\}$
  - We perturb each input with $\varepsilon_W \sim N(\varepsilon;0,\eta I)$
  - For small η, this is equivalent to a regularization term $\eta E_{p(\boldsymbol{x},y)}\left[\left\|\nabla_W \hat{y}(\boldsymbol{x})\right\|^2\right]$
    - It encourages parameters to regions where small perturbations of weights have small influence on output

Injecting Noise at Output Targets

- Most datasets have some mistakes in y labels
  - Harmful to maximize $\log p(y|\boldsymbol{x})$ when y is a mistake
- To prevent it we explicitly model noise on labels
  - Ex: we assume training set label y is correct with probability $1\text{-}\varepsilon$, and otherwise any of the other labels may be correct
  - This can be incorporated into the cost function
    - Ex: *Local Smoothing* regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\varepsilon/(k\text{-}1)$ and $1\text{-}\varepsilon$ respectively

Semi-Supervised Learning

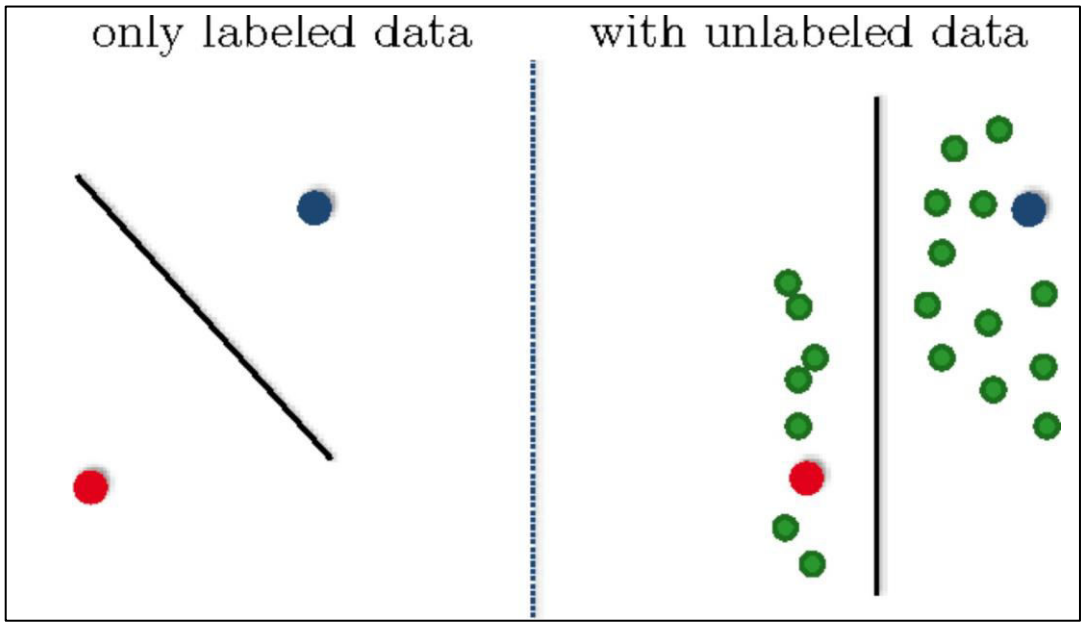
Sargur N. Srihari
srihari@buffalo.edu

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
9. Parameter tying and parameter sharing
10. Sparse representations
11. Bagging and other ensemble methods
12. Dropout
13. Adversarial training
14. Tangent methods

Task of Semi-supervised Learning

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

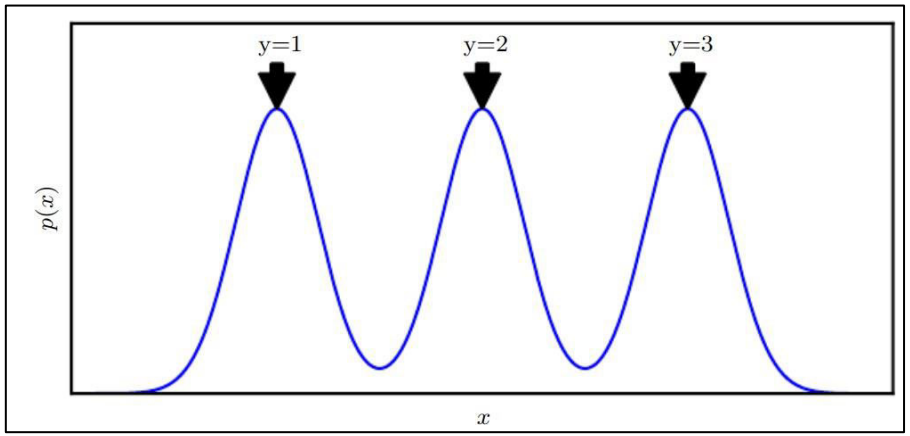


How semi-supervised succeeds

- $p(x)$: a mixture over three components, $y \in \{1,2,3\}$
  - If components well-separated:
    - modeling $p(x)$ reveals where each component is
      - A single labeled example per class enough to learn $p(x|y)$
      - Which we can use to predict $p(y|x)$

capital letters, small letters, digits

x = no. of black pixels



$p(x)$ has three modes

$p(x|y)$ is a univariate Gaussian for y=1,2,3

Task of Semi-supervised Learning

- Both unlabeled examples from $P(\boldsymbol{x})$ and labeled examples from $P(\boldsymbol{x},y)$ are used to estimate $P(y/\boldsymbol{x})$ or predict y from $\boldsymbol{x}$
- In the context of deep learning it refers to learning a representation $\boldsymbol{h}=f(\boldsymbol{x})$
- The goal is to learn a representation so that examples from the same class have similar representations

How unsupervised learning helps

- Unsupervised learning can provide useful clues for how to group examples in representational space
- Examples that cluster tightly in the input space should be mapped to similar representations
- A linear classifier in the new space may achieve better generalization
- A variant is the application of PCA as a preprocessing step before applying a classifier to the projected data

Sharing Parameters

- Instead of separate unsupervised and supervised components in the model, construct models in which generative models of either $P(\boldsymbol{x})$ or $P(\boldsymbol{x},y)$ shares parameters with a discriminative model of $P(y|\boldsymbol{x})$
- One can then trade-off the supervised criterion $-\log P(y|\boldsymbol{x})$ with the unsupervised or generative one (such as $-\log P(\boldsymbol{x})$ or $-\log P(\boldsymbol{x},y)$)
  - The generative criterion then expresses a prior belief about the solution to the supervised problem
    - viz., structure of $P(\boldsymbol{x})$ is connected to structure of $P(y|\boldsymbol{x})$ in a way that is captured by shared parameterization

Multi-task Learning

Sargur N. Srihari
srihari@buffalo.edu

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
6. Parameter tying and parameter sharing
7. Sparse representations
8. Bagging and other ensemble methods
9. Dropout
10. Adversarial training
11. Tangent methods

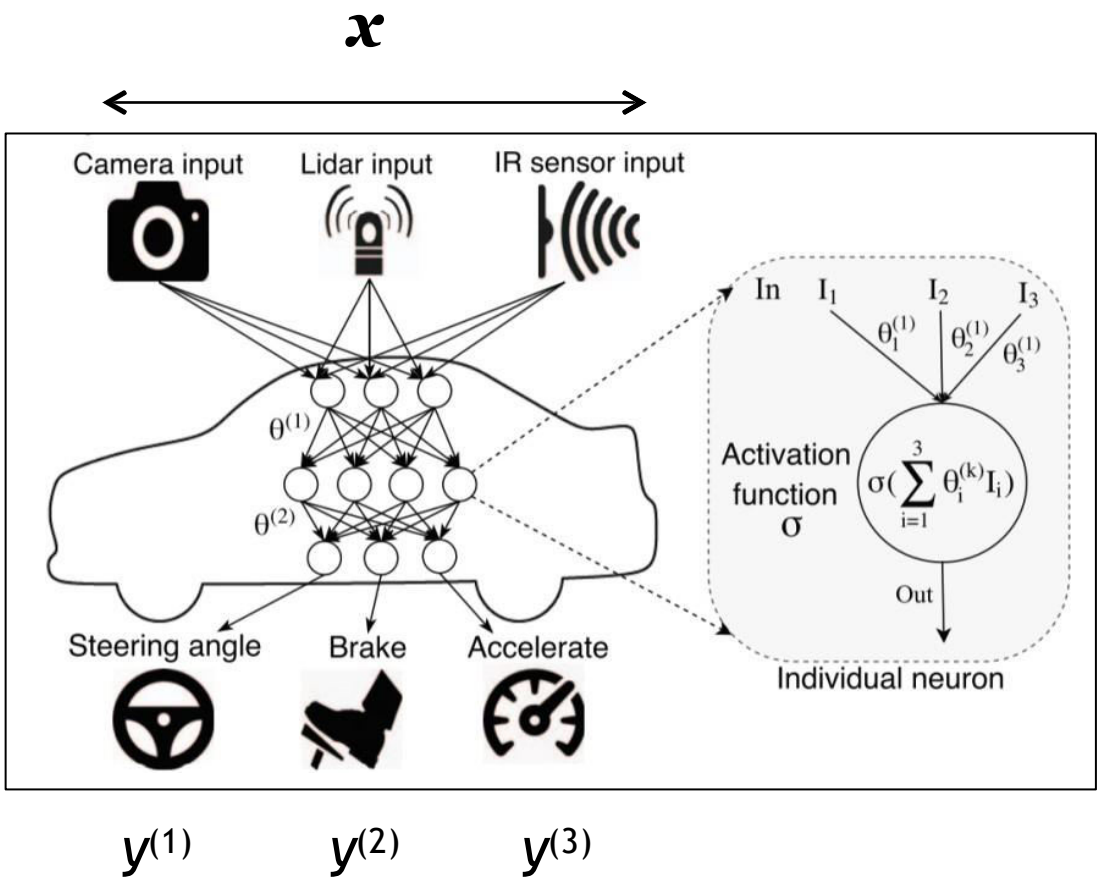
Sharing parameters over tasks

- Multi-task learning is a way to improve generalization by pooling the examples out of several tasks
  - Examples can be seen as providing soft constraints on the parameters
- In the same way that additional training examples put more pressure on the parameters of the model towards values that generalize well

Common form of multitask learning

- Different supervised tasks, predicting $\boldsymbol{y}^{(i)}$ given $\boldsymbol{x}$
- Share the same input $\boldsymbol{x}$, as well as some intermediate representation $\boldsymbol{h}^{(\text{shared})}$ capturing a common pool of factors

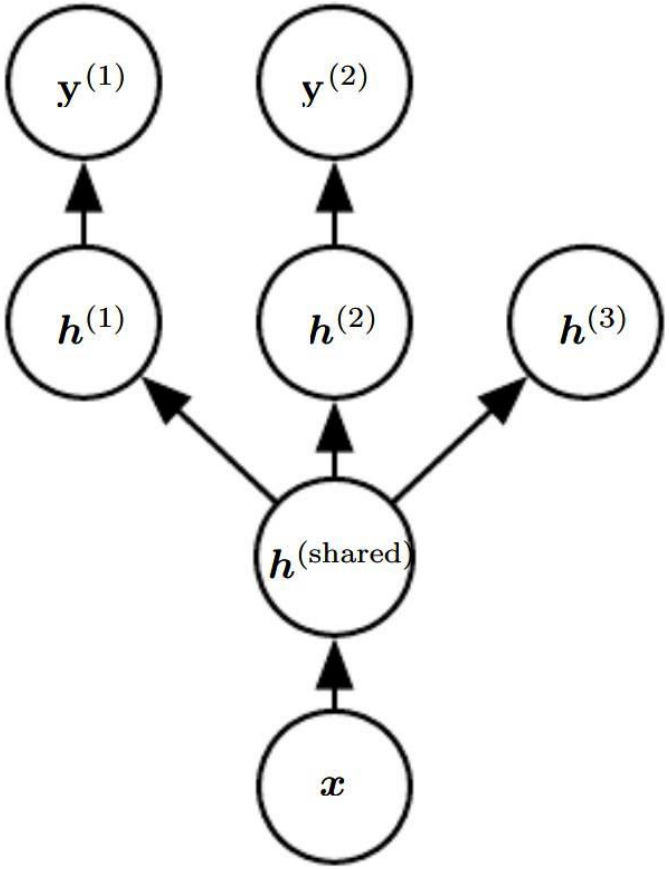
Ex: Autonomous Navigation



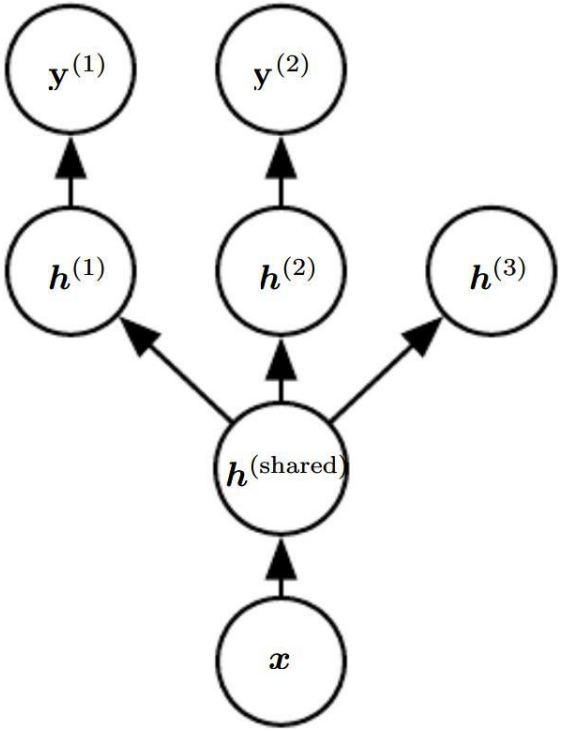
Common multi-task situation

- Common input but different target random variables
  - Lower layers (whether feedforward or includes a generative component with downward arrows) can be shared across such tasks.
    - Task-specific parameters $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$ can be learned on top of those yielding a shared representation $\boldsymbol{h}^{(\text{shared})}$

Common pool of factors explain variations of Input $\boldsymbol{x}$ while each task is associated with a Subset of these factors



Multi-task in Unsupervised Learning



- In the unsupervised learning context
- some of the top level factors are associated with none of the output tasks $\boldsymbol{h}^{(3)}$

  These are factors that explain some of the input variations but not relevant for predicting $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$

Model can be divided into two parts

1. Task specific parameters
   - Which only benefit from the examples of their task to achieve good generalization
     - These are the upper layers of the neural network
2. Generic parameters
   - Shared across all tasks
     - Which benefit from the pooled data of all tasks
     - These are the lower levels of the neural network

Benefits of multi-tasking

- Improved generalization and generalization error bounds
  - achieved due to shared parameters
    - For which statistical strength can be greatly improved
      - In proportion to the increased no. of examples for the shared parameters compared to the scenario of single-task models
- From the point of view of Deep Learning, the underlying prior belief is the following:
  - Among the factors that explain the variations observed in the data associated with different tasks, some are shared across two or more tasks

Early Stopping

Sargur N. Srihari
srihari@cedar.buffalo.edu

This is part of lecture slides on :
http://www.cedar.buffalo.edu/~srihari/CSE676

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
9. Parameter tying and parameter sharing
10. Sparse representations
11. Bagging and other ensemble methods
12. Dropout
13. Adversarial training
14. Tangent methods

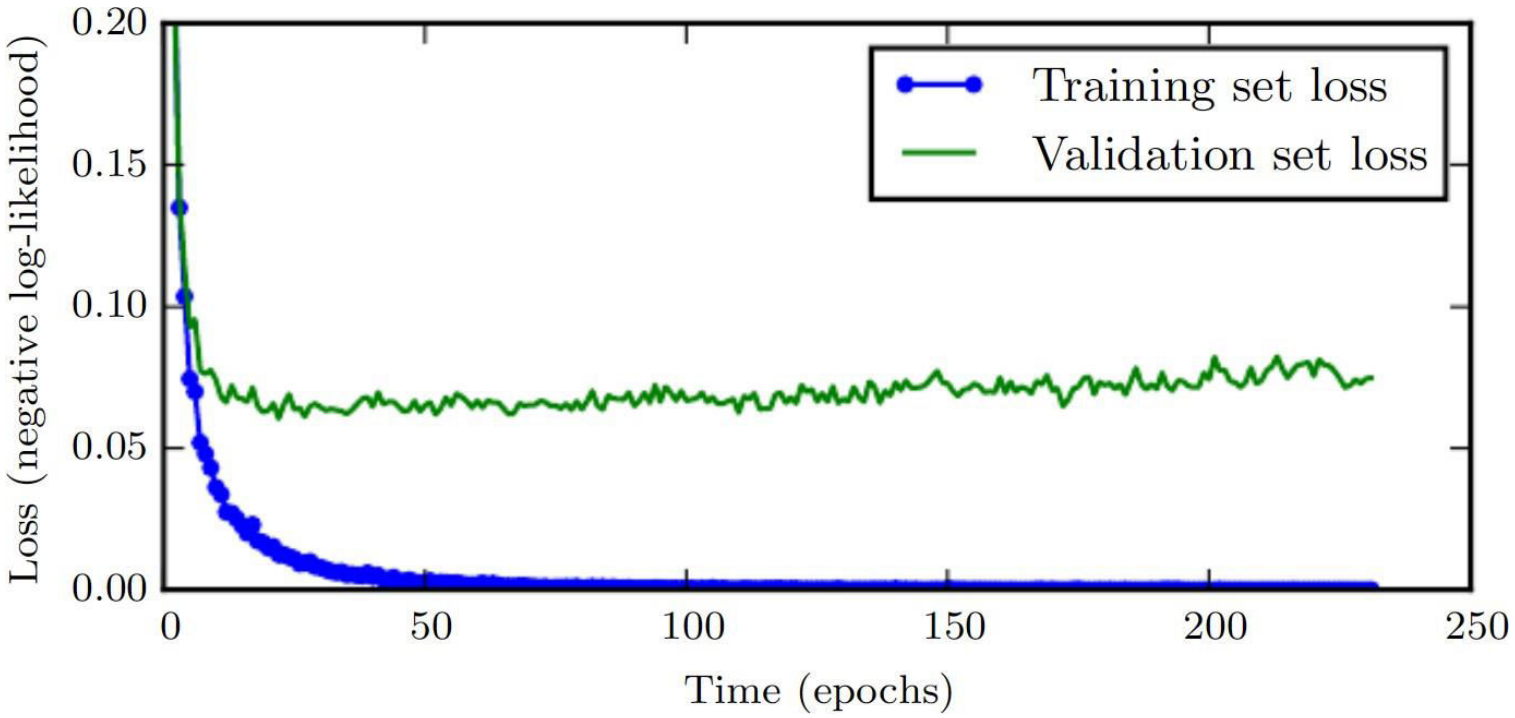
Topics in Early Stopping

1. Learning Curves
2. Early Stopping Meta-Algorithm
3. Early Stopping vs L^2 Regularization

Increase in validation set error

- When training large models with sufficient representational capacity to overfit the task, training error decreases steadily over time, but validation set error begins to rise again
- An example of this behavior is shown next

Learning Curves



Shows how negative log-likelihood loss changes over time (indicated as no. of Training iterations over the data set, or epochs).
In this example, we train a maxout network on MNIST (maxout generalizes RELU further)
Training objective decreases consistently over time, but validation set average Loss eventually begins to increase again forming an asymmetric U-shaped curve

Saving parameters

- We can thus obtain a model with better validation set error (and thus better test error) by returning to the parameter setting at the point of time with the lowest validation set error
- Every time the error on the validation set improves, we store a copy of the model parameters.
- When the training algorithm terminates, we return these parameters, rather than the latest set

Early stopping meta algorithm

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*

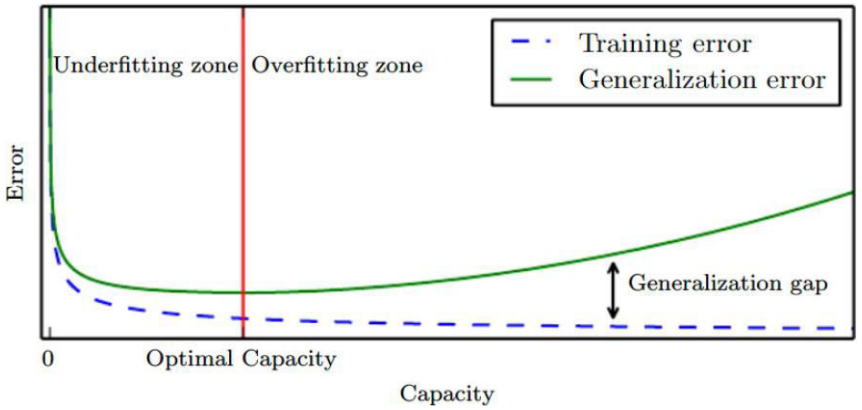
Algorithm determines the best amount of time to train. The meta algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Strategy of Early Stopping

- The above strategy is known as Early Stopping
- It is the most common form of regularization in deep learning
- Its popularity is due to its effectiveness and its simplicity

Early Stopping as Hyperparameter Selection

- We can think of early stopping as a very efficient hyperparameter selection algorithm
  - In this view no. of training steps is just a hyperparameter
  - This hyperparameter has a U-shaped validation set performance curve
  - Most hyperparameters have such a U-shaped validation set performance curve, as seen below



In the case of early stopping we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set

Costs of Early Stopping

- Cost of this hyperparameter is running validation evaluation periodically during training
  - Ideally done in parallel to training process on a separate machine
    - Separate CPU or GPU from main training process, Or
    - Using small validation set or validating set less frequently
- Need to maintain a copy of the best parameters
  - This cost is negligible because they can be stored on a slower, larger memory
    - E.g., training in GPU, but storing the optimal parameters in host memory or on a disk drive

Early Stopping as Regularization

- Early stopping is an unobtrusive form of regularization
- It requires almost no change to the underlying training procedure, the objective function, or the set of allowable parameter values
- So it is easy to use early stopping without damaging the learning dynamics
  - In contrast to weight decay, where we must be careful not to use too much weight decay
    - Otherwise we trap the network in a bad local minimum corresponding to pathologically small weights

Use of a second training step

- Early stopping requires a validation set
  - Thus some training data is not fed to the model
- To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed
  - In the second extra training step, all the training data is included
- There are two basic strategies for the second training procedure

First Strategy for Retraining

- One strategy is to initialize the model again and retrain on all the data
- In the second training pass, we train for the same no. of steps as the early stopping procedure determined was optimal in first pass
- Whether to retrain for the same no. of parameter updates or the same no of passes through the data set?
  - On the second round, each pass through dataset will require more parameter updates because dataset is bigger

First meta algorithm for retraining

A meta algorithm for using early stopping to determine how long to train, then retraining on all the data

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

Second strategy for retraining

- Keep all the parameters obtained from the first round of training and then *continue* training but now require using all the data
- We no longer have a guide for when to stop in terms of the no of steps
- Instead we monitor the average loss function on the validation set and continue training until it falls below the value of the training set objective of when early stopping halted

Second meta algorithm for retraining

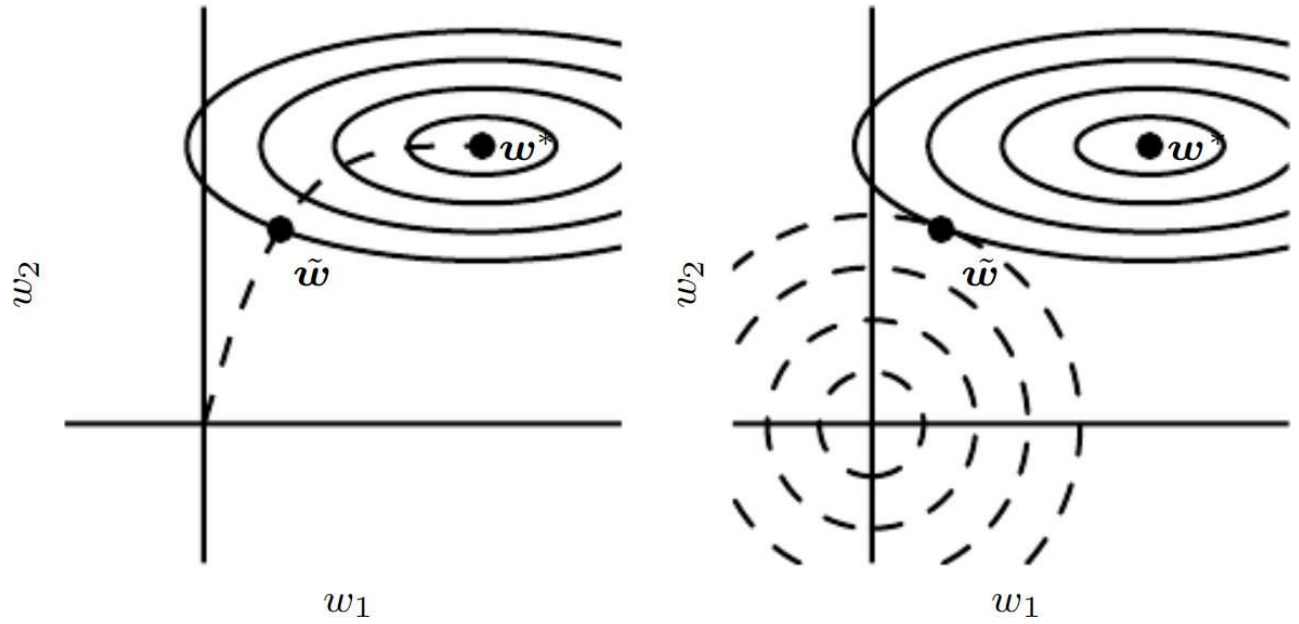
Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
end while

Early stopping as a regularizer

- So far we have stated that early stopping is a regularization strategy
  - But supported the claim only by showing learning curves where the validation set error has a U-shaped curve
- What is the actual mechanism by which early stopping regularizes the model?
  - Early stopping has the effect of restricting the optimizing procedure to a relatively small volume of parameter space in the neighborhood of the initial parameters $\boldsymbol{\theta}_0$

Early Stopping vs L^2 regularization



- Two weights, Solid contour lines: contours of negative log-likelihood
- Left: dashed lines indicates trajectory of SGD. Rather than stopping at point $\boldsymbol{w}^*$ that minimizes cost, early stopping results in an earlier point in trajectory
- Right: dashed circles indicate contours of L^2 penalty which causes the minimum of the total cost to lie nearer the origin than the minimum of the the unregularized cost

Bagging and Other Ensemble Methods

Sargur N. Srihari
srihari@buffalo.edu

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
9. Parameter tying and parameter sharing
10. Sparse representations
11. Bagging and other ensemble methods
12. Dropout
13. Adversarial training
14. Tangent methods

What is bagging?

- It is short for *Bootstrap Aggregating*
- It is a technique for reducing generalization error by combining several models
  - Idea is to train several models separately, then have all the models vote on the output for test examples
- This strategy is called *model averaging*
- Techniques employing this strategy are known as *ensemble methods*
- Model averaging works because different models will not make the same mistake

Ex: Ensemble error rate

- Consider set of k regression models
  - Each model makes error ε_i on each example, $i=1,..N$
  - Errors drawn from a zero-mean multivariate normal with variance $E[\varepsilon_i^2]=v$ and covariance $E[\varepsilon_i\varepsilon_j]=c$
    - Error of average prediction of all ensemble models: $\frac{1}{k}\sum_i \varepsilon_i$
  - Expected squared error of ensemble prediction is

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

    - If errors are perfectly correlated, $c=v$, and mean squared error reduces to v, so model averaging does not help
    - If errors are perfectly uncorrelated and $c=0$, expected squared error of ensemble is only v/k
      - Ensemble error decreases linearly with ensemble size

Ensemble vs Bagging

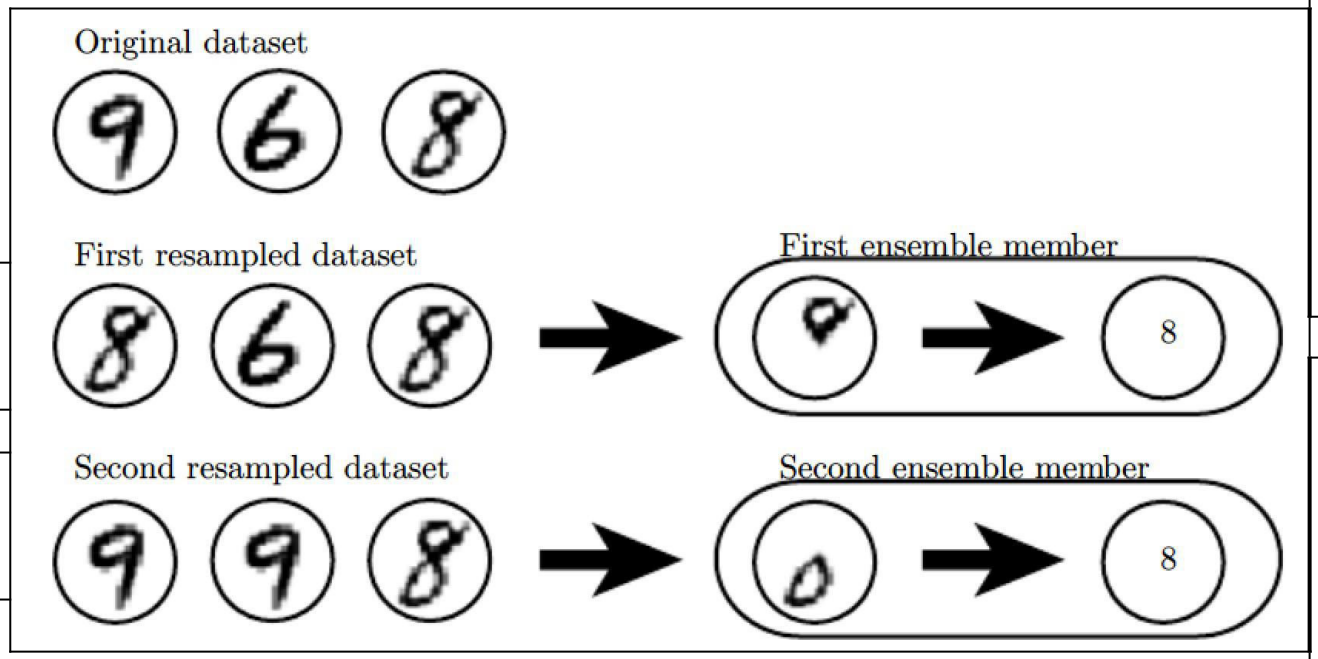
- Different ensemble methods construct the ensemble of models in different ways
  - Ex: each member of ensemble could be formed by training a completely different kind of model using a different algorithm or objective function
- Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times

The Bagging Technique

- Given training set D of size N, generate k data sets of same no of examples as original by sampling with replacement
  - Some observations may be repeated in each D_i the rest being duplicates. This is known as a bootstrap sample
    - The differences in examples will result in differences between trained models
  - The k models are combined by averaging the output (for regression) or voting (for classification)
- An example is given next

Example of Bagging Principle

- Task of training an 8 detector
- Bagging training procedure
  - make different data sets by resampling the given data set



Omit 9
Repeat 8

Omit 6
Repeat 9

Detector learns that loop on top Is an 8

Detector learns that loop at bottom is an 8

- Each detector is brittle
- Their average is robust achieving maximum confidence when both loops are present

Neural nets and bagging

- Neural nets reach a wide variety of solution points
  - Thus they benefit from model averaging when trained on the same dataset
  - Differences in:
    - random initializations
    - random selection of minibatches, in hyperparameters,
  - cause different members of the ensemble to make partially independent errors

Model averaging is powerful

- Model averaging is a reliable method for reducing generalization error
  - Machine learning contests are usually won by model averaging over dozens of models
    - Ex: Netflix grand prize
- Since model averaging performance comes at the expense of increased computation and memory, benchmark comparisons are made using a single model

Boosting

- Incrementally adding models to the ensemble
  - After a weak learner is added, the data are reweighted:
    - examples that are misclassified gain weight and examples that are classified correctly lose weight
- Has been applied to ensembles of neural networks, by incrementally adding neural networks to the ensemble
- Also interpreting a neural network as an ensemble, incrementally adding hidden units to the network

Adaboost (informal)

- Consider 2-class problem
  - Data set $\mathrm{X}=\{\mathbf{x}_1,..\mathbf{x}_N\}$
  - Binary target variables $t_1,..t_N$, t_n c $\{-1,1\}$
  - Each data point has weight w_n, initially set to $1/N$
  - We have a procedure available for training a base classifier using weighted data to give $y(\mathbf{x})$ c$\{-1,1\}$
- At each stage, train a new classifier
  - using a data set in which the weights are adjusted giving higher weight to those misclassified
- When desired no of base classifiers are trained they are combined to form a committee
  - With different weights to base classifiers

Adaboost Algorithm

1. Initialize $w^{(1)}_n = 1/N$ for $n=1,..N$
2. For $m=1,..M$
   a) Fit a classifier $y_m(\mathbf{x})$ by minimizing weighted error

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)$$

where $I(y_m(x_n) \neq t_n)$ is the indicator function and equals 1 when $I(y_m(x_n) \neq t_n)$ and 0 otherwise

   b) Evaluate the quantities $\epsilon_m = \dfrac{J_m}{\sum_{n=1}^{N} w_n^{(m)}}$

and then use these to evaluate $\alpha_m = \ln\left\{\frac{1-\epsilon_m}{\epsilon_m}\right\}$

   c) Update the data weighting coefficients

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m I(y_m(x_n) \neq t_n)\}$$

3. Make predictions using final model

$$Y_M(x) = sign\left(\sum_{m=1}^{M} \alpha_m y_m(x)\right)$$

ϵ_m: weighted error rate of classifier

α_m: weighting coeffts give greater weight to more accurate classifiers

$w_n^{(m+1)}$: increase weight of misclassified data with exponential error

Dropout

Sargur N. Srihari
srihari@buffalo.edu

This is part of lecture slides on :
http://www.cedar.buffalo.edu/~srihari/CSE676

Regularization Strategies

1. Parameter Norm Penalties
2. Norm Penalties as Constrained Optimization
3. Regularization and Under-constrained Problems
4. Data Set Augmentation
5. Noise Robustness
6. Semi-supervised learning
7. Multi-task learning
8. Early Stopping
9. Parameter tying and parameter sharing
10. Sparse representations
11. Bagging and other ensemble methods
12. Dropout
13. Adversarial training
14. Tangent methods

Topics in Dropout

- What is dropout?
- Dropout as an ensemble method
- Mask for dropout training
- Bagging vs Dropout
- Prediction intractability

Overfitting in Deep Neural Nets

- Deep nets have many non-linear hidden layers
  - Making them very expressive to learn complicated relationships between inputs and outputs
  - But with limited training data, many complicated relationships will be the result of training noise
    - So they will exist in the training set and not in test set even if drawn from same distribution
- Many methods developed to reduce overfitting
  - Early stopping with a validation set
  - Weight penalties (L^1 and L^2 regularization)
  - Soft weight sharing

Regularization with unlimited computation

- Best way to regularize a fixed size model is:
  - Average the predictions of all possible settings of the parameters
  - Weighting each setting with the posterior probability given the training data
    - This would be the Bayesian approach
- Dropout does this using considerably less computation
  - By approximating an equally weighted geometric mean of the predictions of an exponential number of learned models that share parameters

Dropout is a bagging method

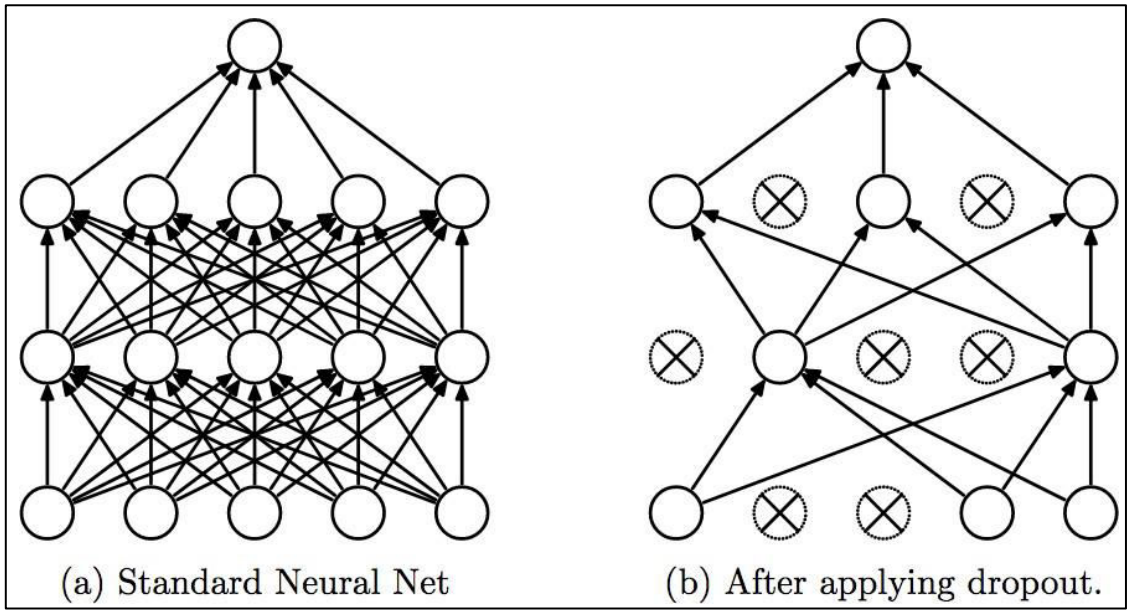
- Bagging is a method of averaging over several models to improve generalization
- Impractical to train many neural networks since it is expensive in time and memory
  - Dropout makes it practical to apply bagging to very many large neural networks
    - It is a method of bagging applied to neural networks
- Dropout is an inexpensive but powerful method of regularizing a broad family of models

Removing units creates networks

- Dropout trains an ensemble of all subnetworks
  - Subnetworks formed by removing non-output units from an underlying base network
- We can effectively remove units by multiplying its output value by zero
  - For networks based on performing a series of affine transformations or on-linearities
  - Needs some modification for radial basis functions based on difference between unit state and a reference value

Dropout Neural Net

- A simple way to prevent neural net overfitting



Drop hidden and visible units from net, i.e., temporarily remove it from the network with all input/output connections. Choice of units to drop is random, determined by a probability p, chosen by a validation set, or equal to 0.5

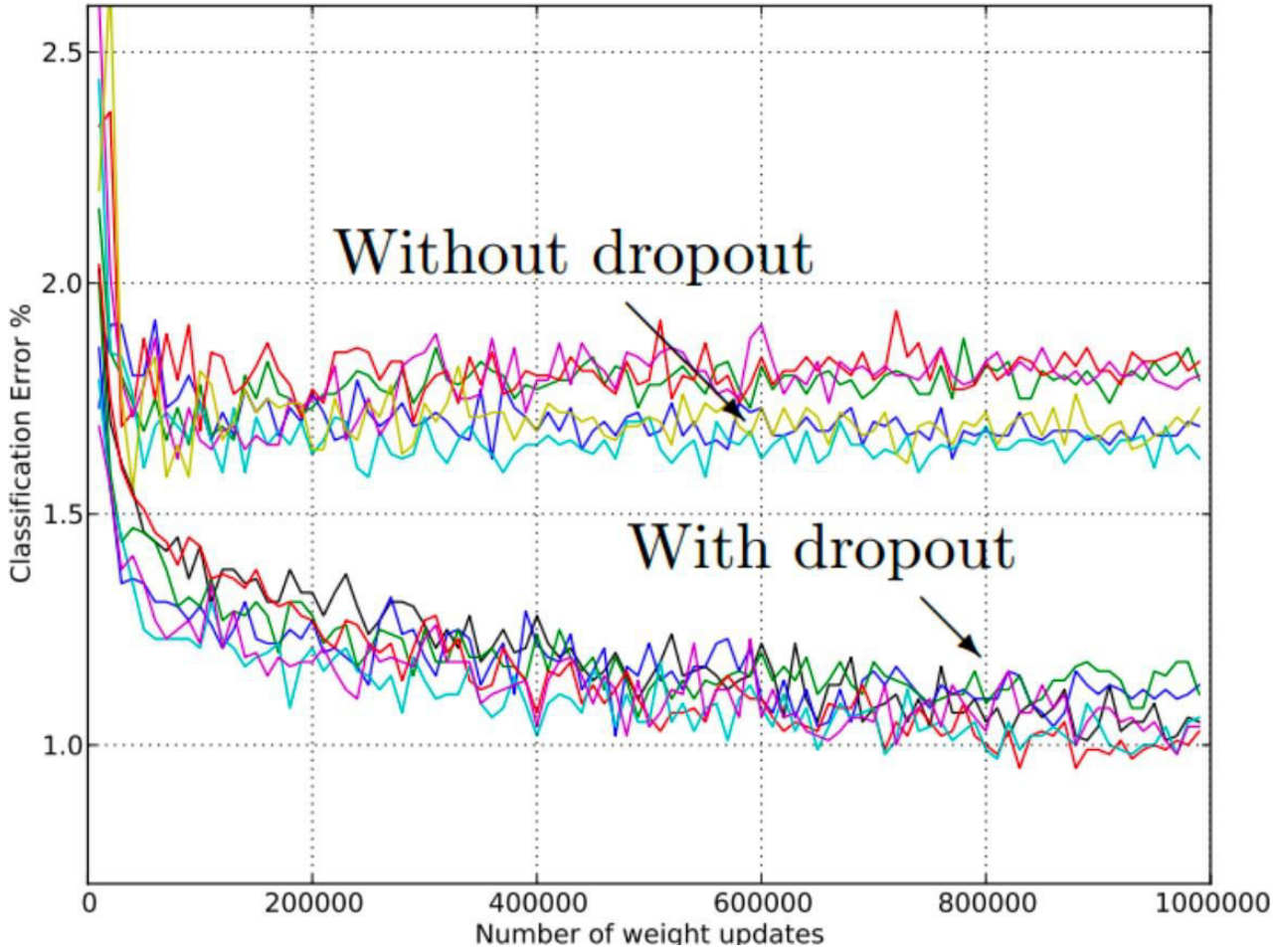
(a) A standard neural net with two hidden layers

(b) A thinned net produced by applying dropout, crossed units have been dropped

Bernoulli Distribution

$$f(x|p) = \begin{cases} p & x = 1 \\ 1-p & x = 0 \end{cases}$$

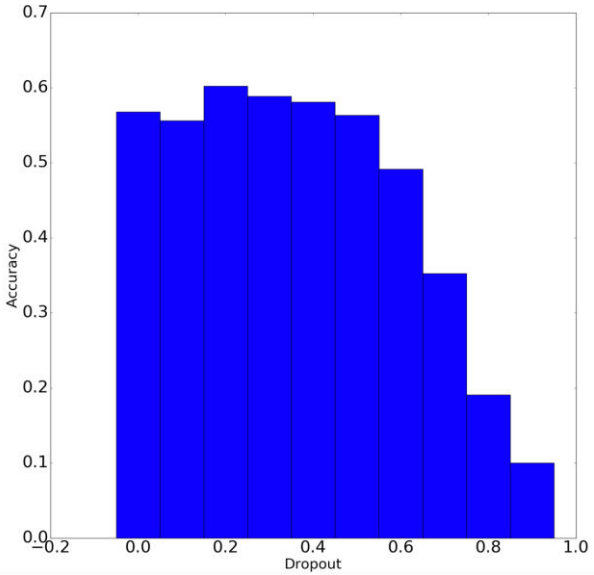
Performance with/without Dropout



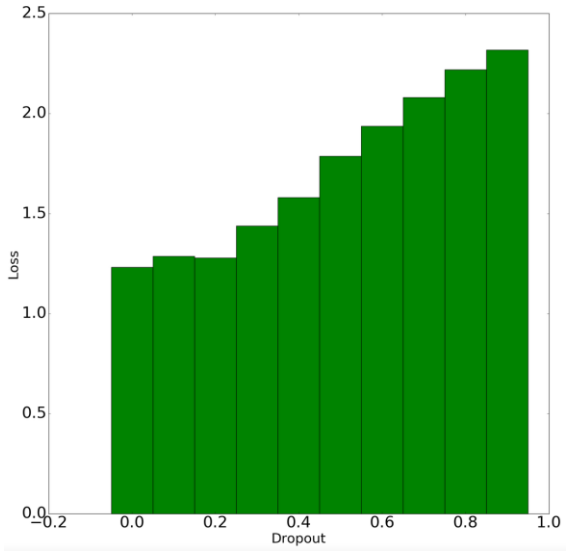
Dependence on p

Deep net in Keras
Validate on CIFAR-10 dataset
Network built had three convolution layers of size 64, 128 and 256 followed by two densely connected layers of size 512 and an output layer dense layer of size 10

Accuracy vs dropout



Loss vs dropout

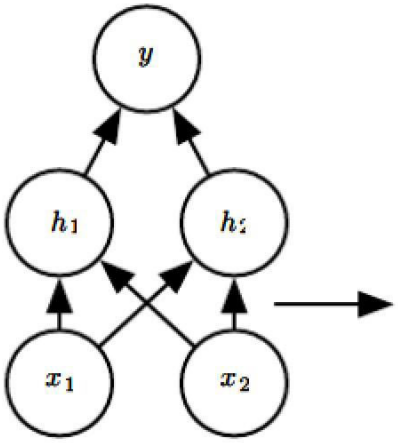


Dropout as bagging

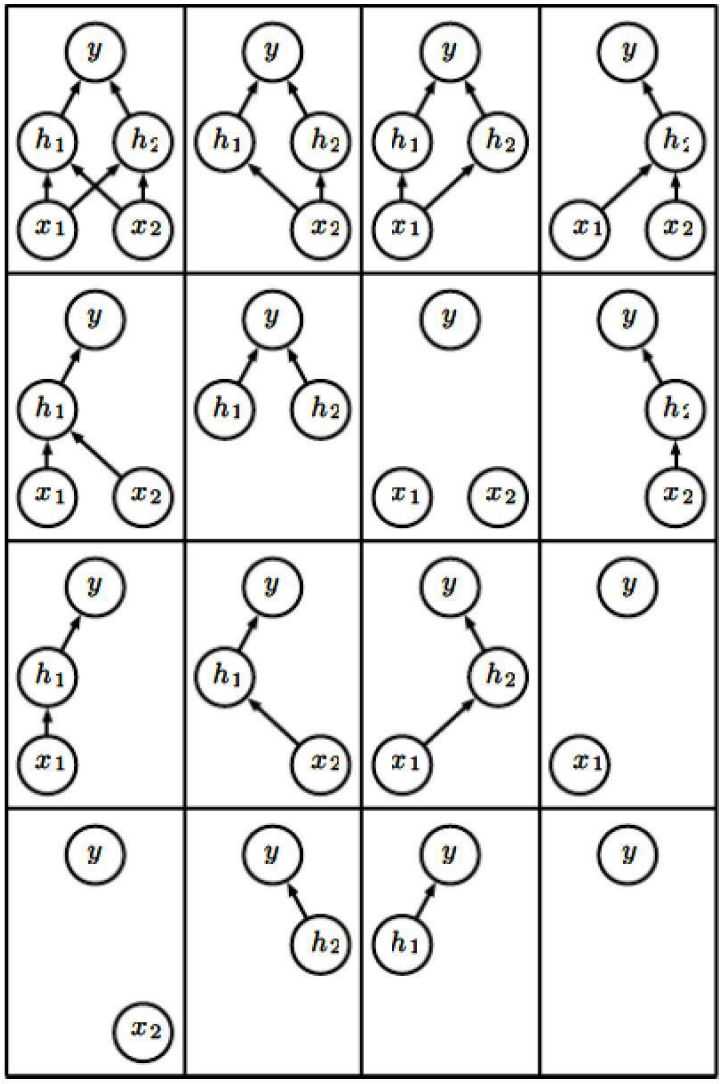
- In bagging we define *k* different models, construct *k* different data sets by sampling from the dataset with replacement, and train model *i* on dataset *i*
- Dropout aims to approximate this process, but with an exponentially large no. of neural networks

Dropout as an ensemble method

- Remove non-output units from base network.
- Remaining 4 units yield 16 networks



Base network


Ensemble of subnetworks

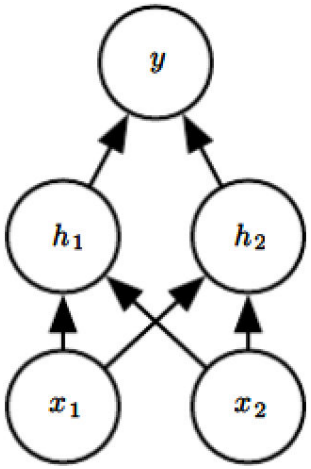
- Here many networks have no path from input to output
- Problem insignificant with large networks

Mask for dropout training

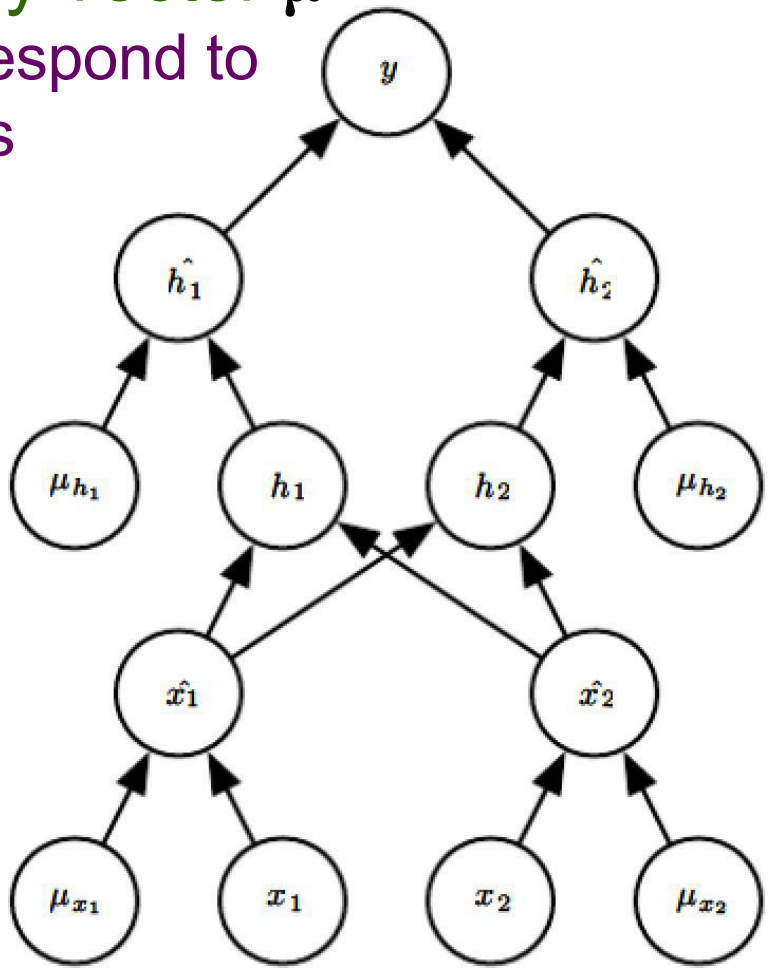
- To train with dropout we use minibatch based learning algorithm that takes small steps such as SGD
- At each step randomly sample a binary mask
  - Probability of including a unit is a hyperparameter
    - 0.5 for hidden units and 0.8 for input units
- We run forward & backward propagation as usual

Forward Propagation with dropout

Feed-forward network



- Network with binary vector $\boldsymbol{\mu}$ whose elements correspond to input and hidden units
- Elements of $\boldsymbol{\mu}$
- With probability of 1 being a hyperparameter
  - 0.5 for hidden
  - 0.8 for input
- Each unit is
  - Multiplied by corresponding mask



- Forward prop as usual
- Equivalent to randomly selecting one of the subnetworks of previous slide

Formal description of dropout

- Suppose that mask vector $\boldsymbol{\mu}$ specifies which units to include
- Cost of the model is specified by $J(\boldsymbol{\theta},\boldsymbol{\mu})$
- Drop training consists of minimizing $E_{\boldsymbol{\mu}}(J(\boldsymbol{\theta},\boldsymbol{\mu}))$
- Expected value contains exponential no. of terms
- We can get an unbiased estimate of its gradient by sampling values of $\boldsymbol{\mu}$

Bagging training vs Dropout training

- Dropout training not same as bagging training
  - In bagging, the models are all independent
  - In dropout, models share parameters
    - Models inherit subsets of parameters from parent network
    - Parameter sharing allows an exponential no. of models with a tractable amount of memory
- In bagging each model is trained to convergence on its respective training set
  - In dropout, most models are not explicitly trained
    - Fraction of sub-networks are trained for a single step
    - Parameter sharing allows good parameter settings

Prediction: Bagging vs. Dropout

- Bagging:
  - Ensemble accumulates votes of members
  - Process is referred to as inference
    - Assume model needs to output a probability distribution
    - In bagging, model i produces $p^{(i)}(y|\boldsymbol{x})$
    - Prediction of ensemble is the mean $\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y|\boldsymbol{x})$
- Dropout:
  - Submodel defined by mask vector $\boldsymbol{\mu}$ defines a probability distribution $p(y|\boldsymbol{x},\boldsymbol{\mu})$
  - Arithmetic mean over all masks is $\sum_{\mu} p(y|\boldsymbol{x},\mu)$
    - Where $p(\boldsymbol{\mu})$ is the distribution used to sample $\boldsymbol{\mu}$ at training time

Intractability of prediction

- Dropout prediction is $\sum_{\mu} p(y \mid \boldsymbol{x}, \mu)$
- It is intractable to evaluate due to an exponential no. of terms
- We can approximate inference using sampling
  - By averaging together the output from many masks
    - 10-20 masks are sufficient for good performance
- Even better approach, at the cost of a single forward propagation:
  - use geometric mean rather than arithmetic mean of the ensemble member's predicted distributions