Fundamentals of Media Processing

Lecturer:
池畑　諭（Prof. IKEHATA　Satoshi）
児玉　和也（Prof. KODAMA Kazuya）

Support:
佐藤　真一（Prof. SATO　Shinichi）
孟　洋（Prof. MO　Hiroshi）

Course Overview (15 classes in total)

1-10 Machine Learning by Prof. Satoshi Ikehata

11-15 Signal Processing by Prof. Kazuya Kodama

Grading will be based on the final report.

10/16 (Today) Introduction Chap. 1

Basic of Machine Learning (Maybe for beginners)

10/23 Basic mathematics (1) (Linear algebra, probability, numerical computation) Chap. 2,3,4

10/30 Basic mathematics (2) (Linear algebra, probability, numerical computation) Chap. 2,3,4

11/6 Machine Learning Basics (1) Chap. 5

11/13 Machine Learning Basics (2) Chap. 5

Basic of Deep Learning

11/20 Deep Feedforward Networks Chap. 6

11/27 Regularization and Deep Learning Chap. 7

12/4 Optimization for Training Deep Models Chap. 8

CNN and its Application

12/11 Convolutional Neural Networks and Its Application (1) Chap. 9 and more

12/18 Convolutional Neural Networks and Its Application (2) Chap. 9 and more

Regularization for Deep Learning

Parameter Norm Penalties

- Regularization is important for the neural networks to work for the generalization (by avoiding over-under-fitting)
- Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\theta)$ to the objective function J as: $\tilde{J}(\boldsymbol{\theta}; X, \boldsymbol{y}) = J(\boldsymbol{\theta}; X, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$
- Different choices for the parameter norm can result in different solutions being preferred. It is often happing that we use different α for different layers
- In neural networks, we typically chose to parameter norm penalty that penalizes only the weights of the affine transformation at each layer and leave the bias unregularized since the bias does not require much data to fit

L2 Parameter Regularization (1)

- L2 parameter norm is called as weight decay, ridge regression or Tikhonov regularization

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

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

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \epsilon\left(\alpha\boldsymbol{\omega} + \nabla_{\boldsymbol{\omega}}J(\boldsymbol{\omega}; X, \boldsymbol{y})\right) = (1 - \epsilon\alpha)\boldsymbol{\omega} - \epsilon\nabla_{\boldsymbol{\omega}}J(\boldsymbol{\omega}; X, \boldsymbol{y})$$

- We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step

L2 Parameter Regularization (2)

- Assuming that $J(\boldsymbol{\omega}; X, \boldsymbol{y}) = J(\boldsymbol{\omega})$, then the second order Tayler expansion around a critical point $\boldsymbol{\omega}^*$ is

$$J(\boldsymbol{\omega}) = J(\boldsymbol{\omega}^*) + \frac{1}{2}(\boldsymbol{\omega} - \boldsymbol{\omega}^*)^T H(\boldsymbol{\omega} - \boldsymbol{\omega}^*), \quad \nabla_{\boldsymbol{\omega}} J(\omega^*) = 0$$

- $\hat{J}(\boldsymbol{\omega}) = J(\boldsymbol{\omega}) + \frac{\alpha}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}$, the minimum of $\hat{J}(\boldsymbol{\omega})$ occurs where its gradient:

$$\alpha\boldsymbol{\omega} + H(\boldsymbol{\omega} - \boldsymbol{\omega}^*) = 0$$

$$\boldsymbol{\omega} = (H + \alpha I)^{-1} H \boldsymbol{\omega}^*$$

$H = Q\Lambda Q^T$
(eigen decomposition)

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

- As α increases, weight decay rescales $\boldsymbol{\omega}^*$ along the axis defined by the eigenvectors of H.

L2 Parameter Regularization (3)

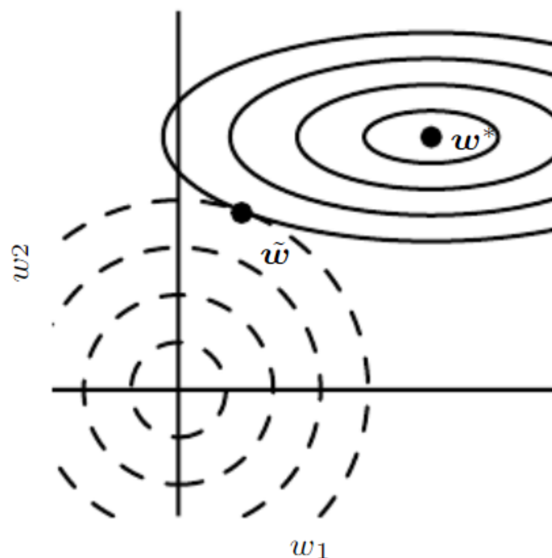


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function 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.

L2 Parameter Regularization (4)

- Assuming that $J(\boldsymbol{\omega}; X, \boldsymbol{y}) = (X\boldsymbol{\omega} - \boldsymbol{y})^{\boldsymbol{T}}(X\boldsymbol{\omega} - \boldsymbol{y}) + \frac{\alpha}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}$, then:

$$\boldsymbol{\omega}^* = (X^T X)^{-1} X^T \boldsymbol{y}$$

$$\boldsymbol{\omega} = (X^T X + \alpha I)^{-1} X^T \boldsymbol{y}$$

- L2 regularization causes the learning algorithm to "perceive" the input X as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance

L1 Parameter Regularization (1)

- L1 regularization results in the solution that is more sparse

- L1 regularization on the model parameter $\boldsymbol{\omega}$ is defined as

$$\Omega(\boldsymbol{\omega}) = \|\boldsymbol{\omega}\|_1$$

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

$$\nabla_{\boldsymbol{\omega}}\tilde{J}(\boldsymbol{\omega}; X, \boldsymbol{y}) = \alpha\text{sign}(\boldsymbol{\omega}) + \nabla_{\boldsymbol{\omega}}J(\boldsymbol{\omega}; X, \boldsymbol{y})$$

- Assuming that the Hessian matrix H is diagonal (i.e., if the data for the linear regression problem has been preprocessed to remove all correlation between the input features by e.g., PCA),

$$\hat{J}(\boldsymbol{\omega}; X, \boldsymbol{y}) = J(\boldsymbol{\omega}^*; X, \boldsymbol{y}) + \sum_i \left(\frac{1}{2}H_{i,i}(\omega_i - \omega_i^*)^2 + \alpha|\omega_i|\right)$$

L1 Parameter Regularization (2)

$$\hat{J}(\boldsymbol{\omega}; X, \boldsymbol{y}) = J(\boldsymbol{\omega}^*; X, \boldsymbol{y}) + \sum_i \left(\frac{1}{2} H_{i,i} (\omega_i - \omega_i^*)^2 + \alpha |\omega_i| \right)$$

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

1. In the case where $\omega_i^* \leq \alpha / H_{i,i}$, the optimal value of ω_i under the regularized objective is simply ω_i=0. This occurs because the contribution of J to the regularized objective function $\tilde{J}$ is overwhelmed, in direction *i*, by L1 regularization (***sparsity***)
2. In the case where $\omega_i^* > \alpha / H_{i,i}$, the regularization does not move the optimal value of ω_i to zero but just instead shifts it in that direction by a distance equal to $\alpha / H_{i,i}$

Norm Penalties as Constrained Optimization

- Assume we want to minimize a function subject to constrains, we can construct a generalized Lagrange function composed of KKT multiplier and a function representing whether the constraint is satisfied. If we constraint Ω to be less than *k*:

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

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

$$\boldsymbol{\theta}^* = \arg\min_{\theta} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha; X, \boldsymbol{y})$$

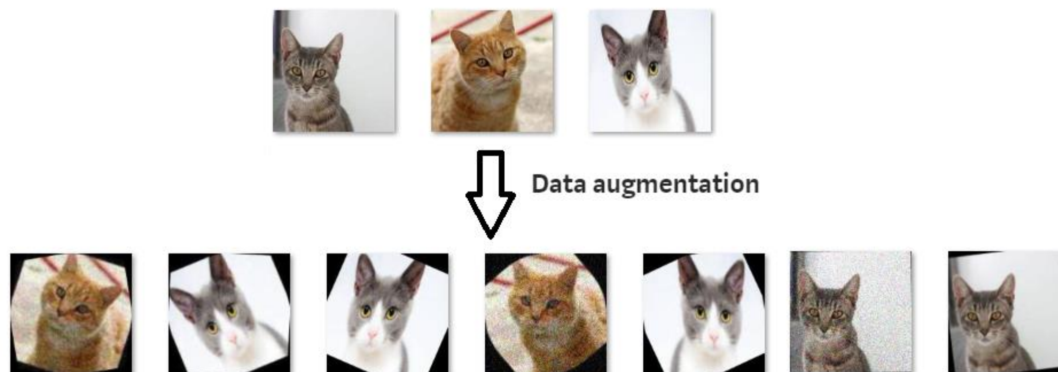
- The reason to use explicit constraints rather than penalties ($\text{i.e.,}\ \min \alpha\boldsymbol{\omega}^T\boldsymbol{\omega} \ \rightarrow\ \text{s.t.}, \alpha\boldsymbol{\omega}^T\boldsymbol{\omega} \leq k$) is;
  a. When we need proper *k*, and do not waste time to find α
  b. Penalties try to $\boldsymbol{\omega}$ to be zero, which may make the gradient extreamely small (***dead unit***), instead, the explicit constraint imposes some stability on the optimization procedure

Dataset Augmentation

- The best way to make a machine learning model generalize better is to train it on more data. However if it is difficult, we can create fake data and add it to the training set. This is called ***data augmentation***

- In the image classification task, it is common to flip and rotate images

- Since the neural network is not robust to noises, it is also common to inject noises to input. Noises are often added to hidden units to improve the robustness (Poole2014)



https://medium.com/@thimblot/data-augmentation-boost-your-image-dataset-with-few-lines-of-python-155c2dc1baec

Noise Robustness

- For some models, the addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop1995). In the general case, noise injection can be much more powerful than simply shrinking the parameters, especially when the noises are added to hidden units (e.g., dropout)

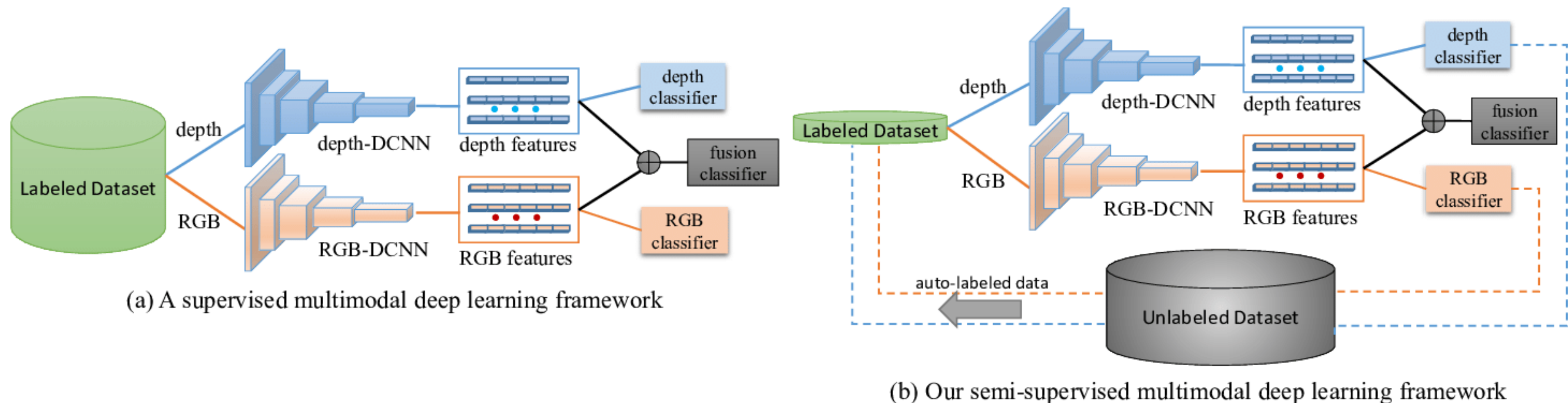
- Another way that noise has been used is by adding it to the weights. Noise applied to the weights can be interpreted as equivalent to a more traditional form of regularization, encouraging stability of the function to be learned

- The training label is often incorrect. To improve the robustness of the model to the errors in the data, we may inject noise at the output target in the form of ***label smoothing*** (Salimans2016)**:** by replacing the classification output (e.g., 1 to 0.7~1.2, 0 to 0 to 0.3)

Semisupervised Learning

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

- In the context of deep learning, semisupervised learning usually refers to learning a representation $h = f(x)$. The goal is to learn a representation so that examples from the same class have similar representations



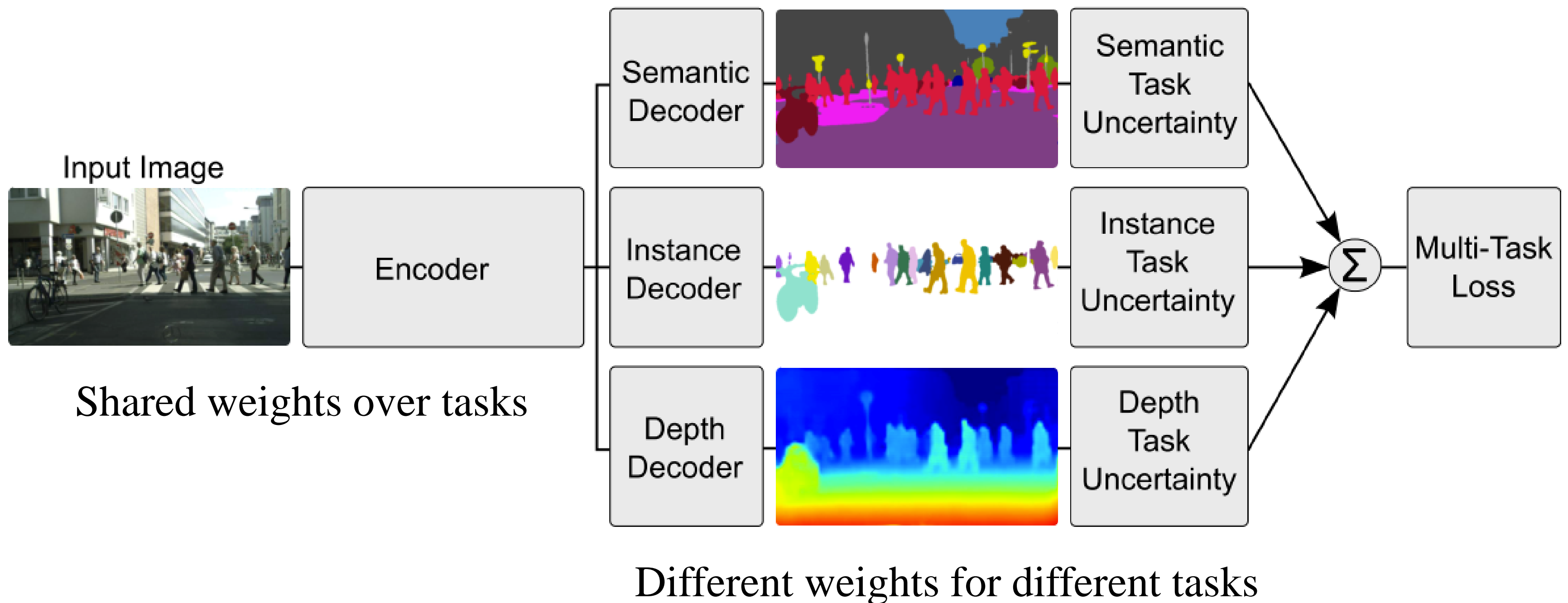
(a) A supervised multimodal deep learning framework

(b) Our semi-supervised multimodal deep learning framework

https://www.researchgate.net/figure/The-structures-of-a-supervised-and-b-semisupervised-multimodal-deep-learning-for_fig1_301227188

Multitask Learning

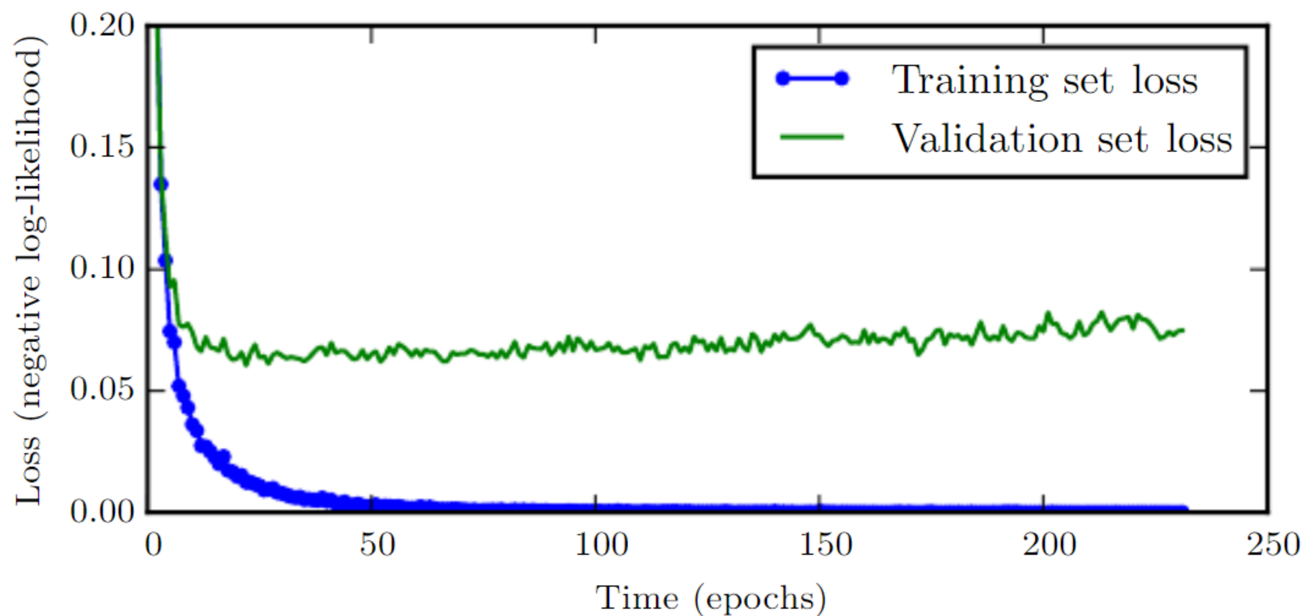
- Multitask Learning (Caruana1993) is a way to improve generalization by pooling the examples arising out of several tasks



http://ruder.io/multi-task/

Early Stopping (1)

- Validation error generally begins rise after some iterations due to overfitting
- To avoid this, we can use ***early stopping***, which can be seen as a very efficient hyperparameter selection algorithm (the number of training steps are also considered as hyperparameter)
- Early stopping is considered as a L2 regularizer when then number of steps is small (i.e., α decreases as steps increases, see details in the book)



Early Stopping (2)

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^*.

Early Stopping (3)

- First training with validation set to decide the number of steps. Then use all data to estimate the final parameters

Algorithm 7.2 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.

Early Stopping (4)

- Another strategy is to keep the parameters obtained from the first round of training and then continue training, but now using all the data until it falls below the value of the training set objective at which the early stopping procedure halted

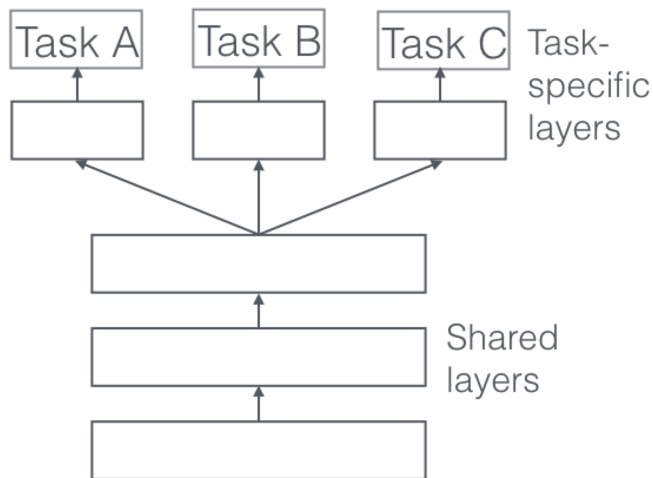
---

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

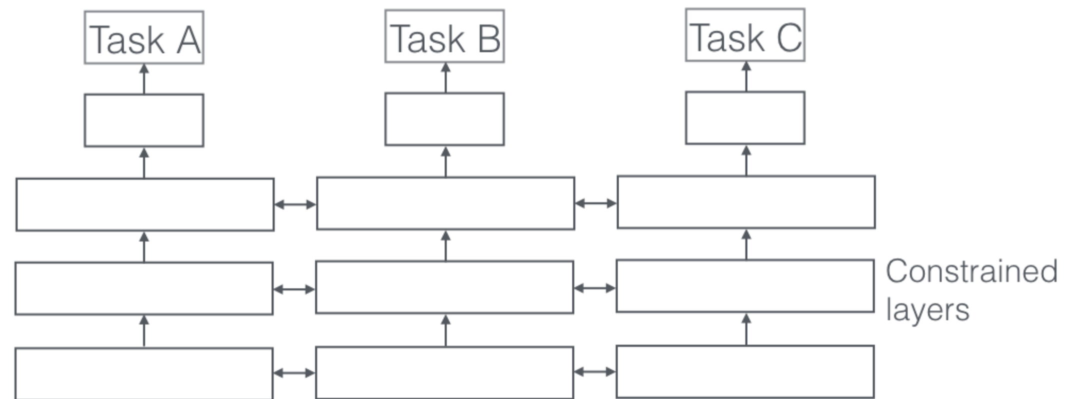
Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ 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

Parameter Tying and Parameter Sharing

- Assume we have model A with parameter ω_A and model B with parameter ω_B. If we know the tasks are similar enough, we can leverage this information to constrain the model



Hard parameter sharing
$(\boldsymbol{\omega}_A = \boldsymbol{\omega}_B)$



Soft parameter sharing
$(\|\boldsymbol{\omega}_A - \boldsymbol{\omega}_B\| \leq \epsilon)$

http://ruder.io/multi-task/

Representational Sparsity

- We can place the penalty on the activations of the units in a neural network, encouraging their activations to be sparse

$$\tilde{J}(\boldsymbol{\theta}; X, \boldsymbol{y}) = J(\boldsymbol{\theta}; X, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$ Penalty on the parameter

$$\tilde{J}(\boldsymbol{\theta}; X, \boldsymbol{y}) = J(\boldsymbol{\theta}; X, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h})$$ Penalty on the activation

$$\Omega(\boldsymbol{h}) = \|\boldsymbol{h}\|_1$$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad \boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\quad \boldsymbol{h} \in \mathbb{R}^n$

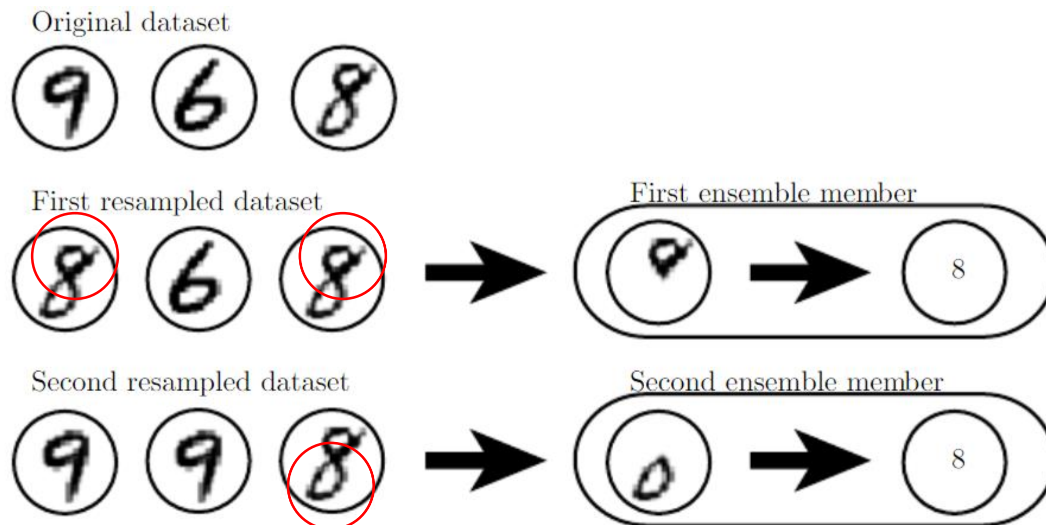
- Other representational sparsity penalties include one derived from a Student t prior on the representation (Olshusen1996), KL divergence penalty (Larochelle2008), regularizing the average activation across several examples (Goodfellow2009) and orthogonal pursuit (Pati1993)

Bagging and Other Ensemble Methods

- ***Bagging*** (short for **b**ootstrap **agg**regat**ing**), model averaging or ensemble method is a technique for reducing generalization error by training several uncorrelated models separately, then have all the models vote on the output for test examples (Breiman1994).

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] = \frac{1}{k}\mathbb{E}(\epsilon_i^2) \because \mathbb{E}(\epsilon_i\epsilon_j) = 0$$

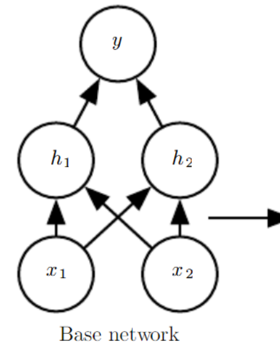
Replacement from the original

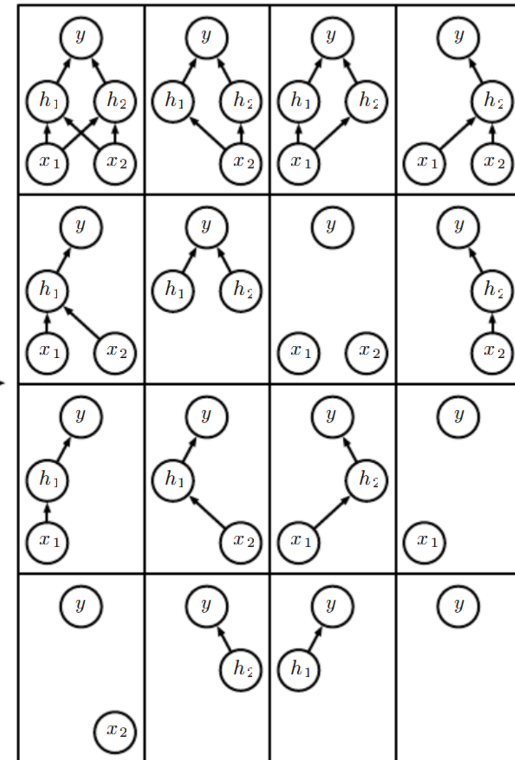


Dropout (1)

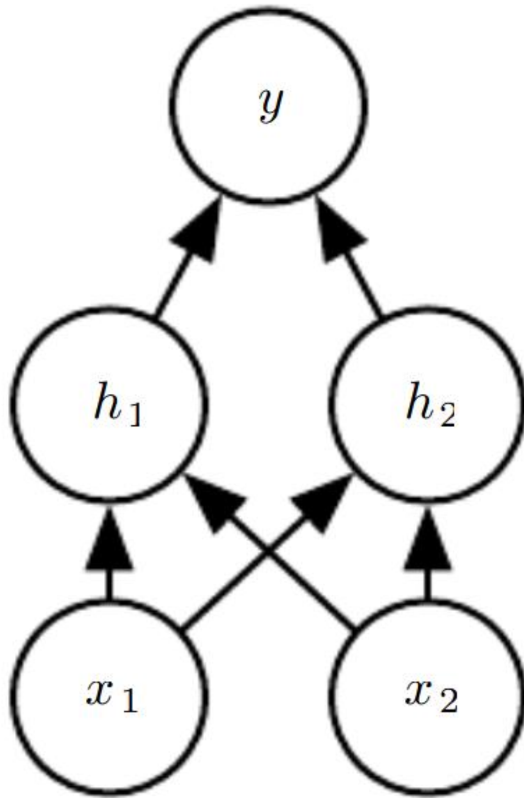
- ***Dropout*** (Srivastava2014) trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network. We can do it by *simply multiplying output values of each unit by zero*

- To train with dropout, we use a minibatch-based learning algorithm such as stochastic descent. Each time we load an example into a minibatch, we randomly sample (p_{dropout}; e.g., 0.8 for input, 0.5 for hidden units) a different binary mask to apply to all the input and hidden units in the network



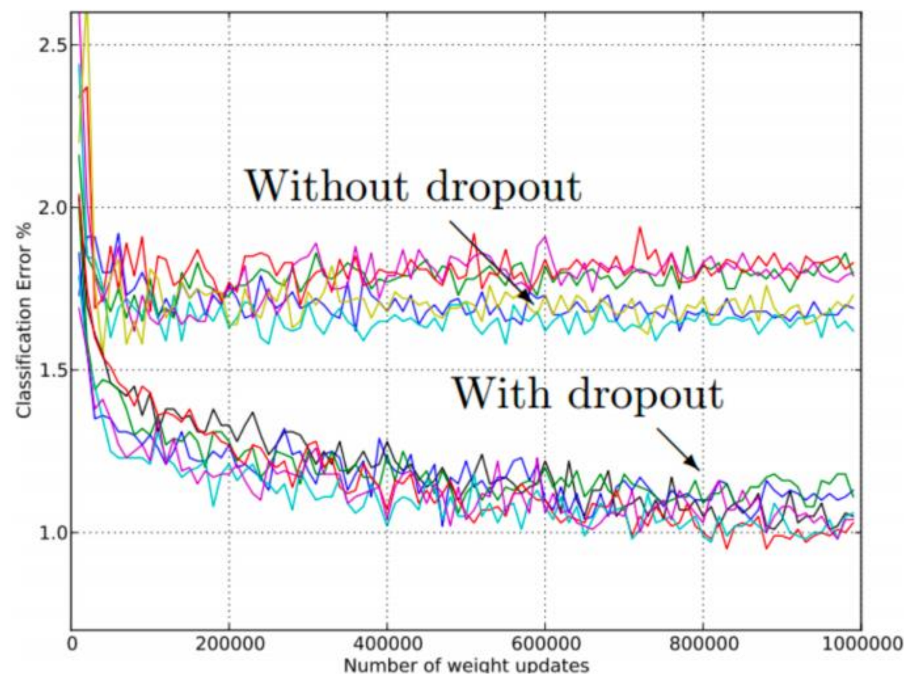


Dropout (2)



Dropout (3)

- Different from bagging, dropout shares the parameters among models, and each model differs slightly for each step. Beyond these dropout follows the bagging algorithm (e.g., the training set encountered by each subnetwork is indeed a subset of the original training set sampled with replacement)



https://cedar.buffalo.edu/~srihari/CSE676/7.12%20Dropout.pdf

Dropout (4)

- Prediction of the network with dropout
  - Arithmetic mean: the arithmetic mean of the probability distribution for each model with different mask ($\boldsymbol{\mu}$; 10-20 masks practically) is given by:

$$p_{ensemble}(y|\boldsymbol{x}) = \sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu})p(y|\boldsymbol{x}, \boldsymbol{\mu})$$

  - Geometric mean: the unnormalized probability distribution defined directly by the geometric mean is given by

d is the number of units dropped

$$\tilde{p}_{ensemble}(y|\boldsymbol{x}) = \sqrt[2d]{\prod_{\boldsymbol{\mu}} p(y|x, \boldsymbol{\mu})} \qquad p_{ensemble}(y|\boldsymbol{x}) = \frac{\tilde{p}_{ensemble}(y|\boldsymbol{x})}{\sum \tilde{p}_{ensemble}(y|\boldsymbol{x})}$$

- Key insight is that with geometric mean, we can approximate $p_{ensemble}$ by evaluating $p(y|x, \boldsymbol{\mu})$ in one model: the model with all units, but with the weights going out of unit *i* multiplied by the probability of including unit *i* (***weight scaling inference rule***)

Dropout (5)

- Advantages of dropout is:
  - Computationally cheap. O(n) computation per example per update to generate n random binary numbers and multiply them by the state. Depending on the implementation, but it generally requires O(n) memory to store those binary numbers
  - It does not significantly limit the model or training procedure

- While the computationally efficient, we may need larger size of network to make the dropout work effectively. In addition, we may need sufficiently amount of training data for making dropout effective

- Another deep learning algorithm, ***batch normalization***, reparametrizes the model in a way that introduces both additive and multiplicative Nosie (dropout is only multiplicative) on the hidden units at training time, often makes dropout unnecessary

Adversarial Training

- Even neural networks have a nearly 100 percent error rate on examples x' that are intentionally constructed so that they are close to x but the model output is quite different. Those examples are called as ***adversarial examples***
- Goodfellow analyzed this was caused by the highly linearity of the model (even with the nonlinear activation). To avoid this, the network should be trained with adversarial examples to assign the same label to x and x'

$+ .007 \times$

$=$

$\boldsymbol{x}$

$y =$"panda"
w/ 57.7%
confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
w/ 8.2%
confidence

$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
w/ 99.3%
confidence

Tangent Propagation

- ***Tangent propagation*** trains network so that the network outputs similar labels along the manifold that was manually specified
- Working in similar manner with data augmentation or adversarial training (try to learn the function that does not change the output)
- For eliminating the need to know the tangent vectors a priori, ***manifold tangent classifier*** (Rifai2011) had been proposed taking advantages of autoencoder to estimate the manifold tangent vectors

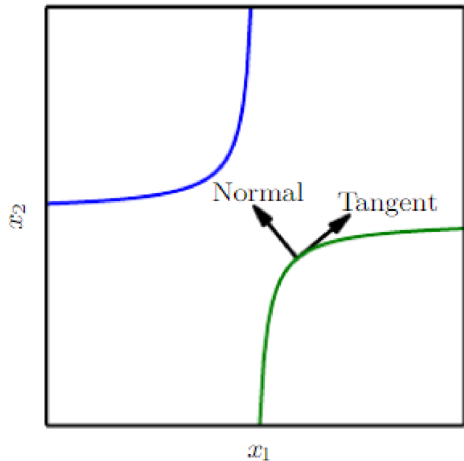


Figure 7.9: Illustration of the main idea of the tangent prop algorithm (Simard *et al.*, 1992) and manifold tangent classifier (Rifai *et al.*, 2011c), which both regularize the classifier output function $f(\boldsymbol{x})$. Each curve represents the manifold for a different class, illustrated here as a one-dimensional manifold embedded in a two-dimensional space. On one curve, we have chosen a single point and drawn a vector that is tangent to the class manifold (parallel to and touching the manifold) and a vector that is normal to the class manifold (orthogonal to the manifold). In multiple dimensions there may be many tangent directions and many normal directions. We expect the classification function to change rapidly as it moves in the direction normal to the manifold, and not to change as it moves along the class manifold. Both tangent propagation and the manifold tangent classifier regularize $f(\boldsymbol{x})$ to not change very much as $\boldsymbol{x}$ moves along the manifold. Tangent propagation requires the user to manually specify functions that compute the tangent directions (such as specifying that small translations of images remain in the same class manifold), while the manifold tangent classifier estimates the manifold tangent directions by training an autoencoder to fit the training data. The use of autoencoders to estimate manifolds is described in chapter 14.