ECBM E4040
Neural Networks and Deep Learning

Regularization

Zoran Kostić
Columbia University
Electrical Engineering Department
& Data Sciences Institute

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

References and Acknowledgments

- Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville, http://www.deeplearningbook.org/ , chapter 7.
- Lecture material by bionet group / Prof. Aurel Lazar (http://www.bionet.ee.columbia.edu/)
- Trevor Hastie, Robert Tibshirani, Jerome Friedman, “The Elements of Statistical Learning” https://web.stanford.edu/~hastie/ElemStatLearn//printings/ESLII_print12.pdf

Regularization

In mathematics and statistics, particularly in the fields of machine learning and inverse problems, regularization is a process of introducing additional information

- **in order to solve an ill-posed problem, or**
- **to prevent overfitting the training data.**

With the hope that this improves generalization, i.e., the ability to correctly handle data that the network has not trained on.

We will study regularization.

This guy will help!

Regularization from a Bayesian Perspective

The Role of a Prior

Regularization From a Bayesian Perspective

There is a deep connection between the Bayesian perspective on estimation and the process of regularization. Many forms of regularization can be given a Bayesian interpretation.

Given a dataset $(\mathbf{x}^1, \ldots, \mathbf{x}^m)$, the posterior $p(\boldsymbol{\theta}|\mathbf{x}^1, \ldots, \mathbf{x}^m)$ can be obtained by combining the data likelihood $p(\mathbf{x}^1, \ldots, \mathbf{x}^m|\boldsymbol{\theta})$ with the prior belief on the parameter encapsulated by $p(\boldsymbol{\theta})$:

$$\log p(\boldsymbol{\theta}|\mathbf{x}^1, \ldots, \mathbf{x}^m) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\mathbf{x}^i|\boldsymbol{\theta}) + \text{constant}$$

where the constant is $-\log Z$, with the normalization constant Z that depends only on the data.

When maximizing over $\boldsymbol{\theta}$, this constant does not matter.

Regularization From a Bayesian Perspective

In the context of maximum likelihood learning, the introduction of the prior distribution plays the same role as a regularizer in that it can be seen as a term (the first one below)

$$\log p(\boldsymbol{\theta}|\mathbf{x}^1, \ldots, \mathbf{x}^m) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\mathbf{x}^i|\boldsymbol{\theta}) + \text{constant}$$

added to the objective function that is added (to the second term, the log-likelihood) in hopes of achieving better generalization, despite of its detrimental effect on the likelihood of the training data.

Classical Regularization

Parameter Norm Penalty

L^2 Parameter Regularization

L^1 Parameter Regularization

Classical Regularization

We denote the regularized objective function by $\tilde{J}$:

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

where α is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function $J(\mathbf{x}; \boldsymbol{\theta})$.

The hyperparameter α is a non-negative real number. Setting $\alpha = 0$ results in no regularization. Larger values of α correspond to more regularization.

Note that for neural networks, we typically use a parameter norm penalty Ω that only penalizes the interaction weights. The offsets are left unregularized. The offsets typically require less data to fit accurately than the weights.

L^2 Parameter Regularization

The L^2 parameter norm penalty commonly known as weight decay is given by $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\mathbf{w}||_2^2$.

As we will see, the L^2 regularization strategy drives the parameters closer to the origin. To simplify the presentation, we assume that $\boldsymbol{\theta} = \mathbf{w}$ (no offset term). The gradient of the total objective function amounts to

$$\nabla_{\mathrm{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathrm{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}).$$

A single gradient step for updating the weights is

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon(\alpha \mathbf{w} + \nabla_{\mathrm{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})),$$

or

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon \nabla_{\mathrm{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})).$$

L^2 Parameter Regularization

To simplify the analysis, we consider a quadratic approximation to the objective function in the neighborhood of the empirically optimal value of the weights $\mathbf{w}^*$.

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

where $\mathbf{H}$ of J with respect to $\mathbf{w}$ evaluated at $\mathbf{w}^*$.

There is no first order term in this quadratic approximation, because $\mathbf{w}^*$ is defined to be a minimum, where the gradient vanishes.

Likewise, because $\mathbf{w}^*$ is a minimum, we can conclude that $\mathbf{H}$ is positive semi-definite and

$$\nabla_w \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*).$$

L^2 Parameter Regularization

The location of the minimum of the regularized objective function is given by

$$\alpha \mathbf{w} + \mathbf{H}(\mathbf{w} - \mathbf{w}^*) = 0,$$

or

$$(\mathbf{H} + \alpha \mathbf{I})\mathbf{w} = \mathbf{H}\mathbf{w}^*,$$

or finally,

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

The regularization term moves the optimum from $\mathbf{w}^*$ to $\tilde{\mathbf{w}}$. As α approaches 0, $\tilde{\mathbf{w}}$ approaches $\mathbf{w}^*$.

L^2 Parameter Regularization

Here we investigate the case when α grows.

Because $\mathbf{H}$ is real and symmetric, we can decompose it into a diagonal matrix

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$$

with $\mathbf{\Lambda}$ having real eigenvalues, $\mathbf{Q}$ orthonormal eigenvectors and $\mathbf{Q}^{-1} = \mathbf{Q}^T$.
We have

$$\begin{aligned}\tilde{\mathbf{w}} &= (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T + \alpha\mathbf{I})^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{w}^* \\ &= [\mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})\mathbf{Q}^T]^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{w}^* \\ &= \mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})^{-1}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{w}^*,\end{aligned}$$

and finally

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

L^2 Parameter Regularization

$\mathbf{Q}^T\tilde{\mathbf{w}}$ is rotating our solution parameters $\tilde{\mathbf{w}}$ into the basis defined by the eigenvectors of $\mathbf{Q}$ of $\mathbf{H}$.

Consequently, the effect of weight decay is to rescale the coefficients of the eigenvectors. The i'th component is rescaled by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$.

Along the directions where the eigenvalues of $\mathbf{H}$ are relatively large, for example, where $\lambda_i >> \alpha$, the effect of regularization is relatively small. However, components with $\lambda_i << \alpha$ will be shrunk to have nearly zero magnitude.

L^1 Parameter Regularization

L^1 regularization on the model parameter $\mathbf{w}$ is defined by

$$\Omega(\boldsymbol{\theta}) = ||\mathbf{w}||_1 = \sum_i |\mathbf{w}_i|,$$

that is, as the sum of absolute values of the individual parameters.

The gradient of the regularized objective function is

$$\nabla_{\mathrm{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \beta \mathrm{sign}(\mathbf{w}) + \nabla_{\mathrm{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

where $\mathrm{sign}(\mathbf{w})$ is the sign of $\mathbf{w}$ applied element-wise.

Not that the regularization contribution to the gradient no longer scales linearly with $\mathbf{w}$, but instead it is a constant factor with a sign equal to $\mathrm{sign}(\mathbf{w})$.

L^1 Parameter Regularization

The gradient of the approximation is given by

$$\nabla_w \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*).$$

where $\mathbf{H}$ is the Hessian matrix of J with respect to $\mathbf{w}$ evaluated at $\mathbf{w}^*$.

To gain insight, we assume that the Hessian is diagonal, that is, $\mathbf{H} = \text{diag}([\gamma_1, ..., \gamma_N])$, where each $\gamma_i > 0$.

With this assumption, the solution of the minimum of the L^1 regularized objective function decomposes into a system of equations of the form:

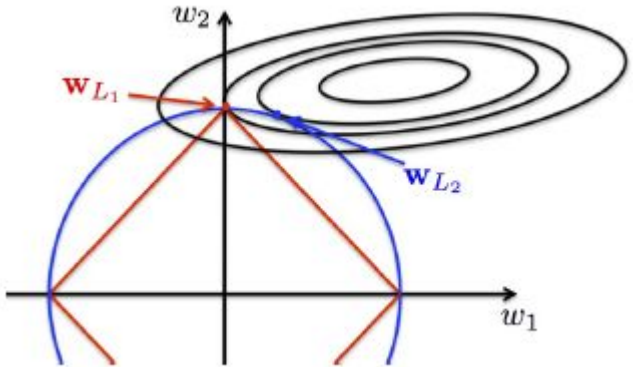
$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{1}{2}\gamma_i(\mathbf{w}_i - \mathbf{w}_i^*)^2 + \beta|\mathbf{w}_i^*|.$$

L^1 Parameter Regularization

For each dimension i, the optimal solution is of the form

$$\mathbf{w}_i = \text{sign}(\mathbf{w}_i^*) \max(|\mathbf{w}_i^*| - \frac{\beta}{\gamma_i}, 0).$$

L^1 regularization prefers sparser solutions i.e it prefers some parameters to have optimal value of zero.



L^1 Parameter Regularization

Most weights are driven to zero.

Why do that?

- **The gradient of this regularizer does not flatten out as it approaches zero and pushes unneeded weights down.**
- **This can be used as a method of feature selection.**

L^1 and L^2 Parameter Regularization, in Summary

- Regularization is any component of the model, training process or prediction procedure which is included to account for limitations of the training data, including its finiteness.
- Most classical regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\mathbf{\Omega}(\theta)$ to the objective function J. We denote the regularized objective function by $\tilde{J}$:

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

- For L^2 regression, $\mathbf{\Omega}(\theta) = \frac{1}{2}|\mathbf{w}||_2^2$
- For L^1 regression, $\mathbf{\Omega}(\theta) = |\mathbf{w}||_1$

Regularization: Why Make Coefficients Small

The magnitude of coefficients increases exponentially with an increase in model complexity.

What does a large coefficient imply?

- **a lot of emphasis on a particular feature, i.e. the particular feature is a good predictor for the outcome. When it becomes too large, the algorithm starts modelling intricate relations to estimate the output and ends up overfitting to the particular training data.**

The higher the model complexity, the more likely is the possibility of overfitting (remember 9th degree polynomial fit to a quadratic function).

Intuition says: impose a constraint on the magnitude of coefficients to reduce model complexity.

analyticsvidhya.com, L1 and L2

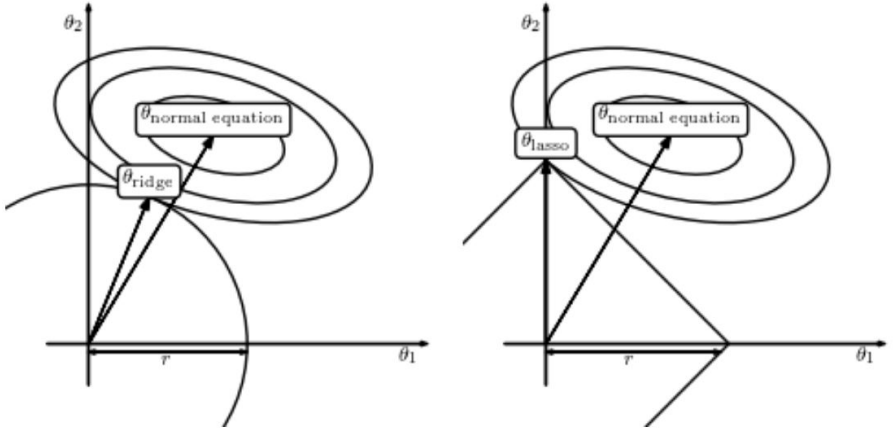
L^1 vs.L^2 Parameter Regularization

- **A geometric interpretation of regularization**
  - **The right figure shows L1 regularization (LASSO regression) and the left figure L2 regularization (ridge regularization). The ellipses indicate the posterior distribution for no prior or regularization. The solid lines show the constraints due to regularization (limiting Θ^2 for ridge regression and abs(Θ) for LASSO regression). The corners of the L1 regularization create more opportunities for the solution to have zeros for some of the weights.**

3.4 Shrinkage Methods in "The Elements of Statistical Learning"

https://web.stanford.edu/~hastie/ElemStatLearn//printings/ESLII_print12

from: http://www.astroml.org/book_figures/chapter8/fig_lasso_ridge.htm



Regularization by Constrained Optimization

We can also constrain the norm to be smaller than some value, rather than imposing a penalty on it. This is a case of constrained optimization.

Such constrained optimization can be solved by projecting the weight vector on the constraint space after every update. This can allow us to have a higher learning rate and train networks faster.

In the case of least squares linear regression, if the system is underconstrained, the matrix X^TX will not be invertible. However, the matrix $X^TX + I$ will always be invertible.

Regularizations that can be Approximated by Penalty Regularizations

Regularization via Injecting Noise

Early Stopping

Regularization via Injecting Noise

Neural networks can be sensitive to noisy inputs.

It is possible to use noise as part of a regularization strategy.

Under certain assumptions on the noise model and certain approximations, we can show such regularization strategies as implementing a penalty based regularization.

We can either add noise to the inputs during training or inject noise into the weights during training.

Regularization via Injecting Noise at the Inputs

- With each input presentation to the model, we also include a random perturbation, $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \nu \mathbf{I})$
- We will analyze injecting noise at the inputs in the context of regression, where we are interested in learning a model $\hat{y}(\mathbf{x})$
- The cost function then becomes

$$\begin{aligned}\tilde{J}_{\mathrm{x}} &= \mathbb{E}_{p(\mathrm{x},y,\boldsymbol{\epsilon})}[(\hat{y}(\mathbf{x}+\boldsymbol{\epsilon}) - y)^2] \\ &= \mathbb{E}_{p(\mathrm{x},\boldsymbol{\epsilon})}[\hat{y}^2(\mathbf{x}+\boldsymbol{\epsilon})] - 2\mathbb{E}_{p(\mathrm{x},\boldsymbol{\epsilon})}[y\hat{y}(\mathbf{x}+\boldsymbol{\epsilon})] + \mathbb{E}_{p(y)}[y^2]\end{aligned}$$

- Assuming that the noise is small, we can model its effect using the Taylor series expansion

$$\hat{y}(\mathbf{x}+\boldsymbol{\epsilon}) = \hat{y}(\mathbf{x}) + \boldsymbol{\epsilon}^T \nabla_{\mathrm{x}} \hat{y}(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}^T \nabla_{\mathrm{x}}^2 \hat{y}(\mathbf{x})\boldsymbol{\epsilon} + \mathcal{O}(\boldsymbol{\epsilon}^3)$$

Regularization via Injecting Noise at the Inputs

- Plugging the taylor series we get

$$\tilde{J}_x \approx \mathbb{E}_{p(x,\epsilon)}\left[\left(\hat{y}(\mathbf{x}) + \epsilon^T \nabla_x \hat{y}(\mathbf{x}) + \frac{1}{2}\epsilon^T \nabla_x^2 \hat{y}(\mathbf{x})\epsilon\right)^2\right]$$

$$- 2\mathbb{E}_{p(x,\epsilon)}\left[y\left(\hat{y}(\mathbf{x}+\epsilon) = \hat{y}(\mathbf{x}) + \epsilon^T \nabla_x \hat{y}(\mathbf{x}) + \frac{1}{2}\epsilon^T \nabla_x^2 \hat{y}(\mathbf{x})\epsilon\right)\right] + \mathbb{E}_{p(y)}[y^2]$$

$$= \mathbb{E}_{p(x,y)}[(\hat{y}(\mathbf{x}) - y)^2] + \mathbb{E}_{p(x,\epsilon)}\left[\hat{y}(\mathbf{x})\epsilon^T \nabla_x^2 \hat{y}(\mathbf{x})\epsilon + \left(\epsilon^T \nabla_x \hat{y}(\mathbf{x})\right)^2 + \mathcal{O}(\epsilon^3)\right]$$

$$- \mathbb{E}_{p(x,y,\epsilon)}\left[y\epsilon^T \nabla_x^2 \hat{y}(\mathbf{x})\epsilon\right]$$

$$= J + \nu\mathbb{E}_{p(x,y)}\left[(\hat{y}(\mathbf{x}) - y)\nabla_x^2 \hat{y}(\mathbf{x})\right] + \nu\mathbb{E}_{p(x,y)}\left[||\nabla_x \hat{y}(\mathbf{x})||^2\right]$$

Regularization via Injecting Noise at the Inputs

- If we minimize this objective function, by taking the functional gradient of $\hat{y}(x)$ and setting it to zero, we get

$$\hat{y}(\mathbf{x}) = \mathbb{E}_{p(y|\mathrm{x})}[y] + \mathcal{O}(\nu)$$

- This implies that $\nu\mathbb{E}_{p(\mathrm{x},y)}\left[(\hat{y}(\mathbf{x}) - y)\nabla_{\mathrm{x}}^2\hat{y}(x)\right]$ reduces to $\mathcal{O}(\nu^2)$
- Therefore,

$$\tilde{J}_{\mathrm{x}} \approx J + \nu\mathbb{E}_{p(\mathrm{x},y)}\left[\|\nabla_{\mathrm{x}}\hat{y}(\mathbf{x})\|^2\right] + \mathcal{O}(\nu^2)$$

- This regularization term has the effect of penalizing large gradients of the function $\hat{y}(\mathbf{x})$. That is, it has the effect of reducing the sensitivity of the output of the network with respect to small variations in its input $\mathbf{x}$.
- We can interpret this as attempting to build in some local robustness into the model and thereby promote generalization. We note also that for linear networks, this regularization term reduces to simple weight decay

Regularization via Injecting Noise at the Weights

- Through similar analysis, we can approximate the cost function in the case where we add noise $\epsilon_w \sim \mathcal{N}(\mathbf{0}, \eta \mathbf{I})$ at the weights by the following

$$\tilde{J}_w \approx J + \eta \mathbb{E}_{p(x,y)} \left[\|\nabla_w \hat{y}(x)\|^2 \right] + \mathcal{O}(\eta^2)$$

- This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output.

- In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights.

Regularization by Early Stopping

Early stopping can be viewed as regularization in time.

Intuitively, a training procedure like gradient descent will tend to learn more and more complex functions as the number of iterations increases.

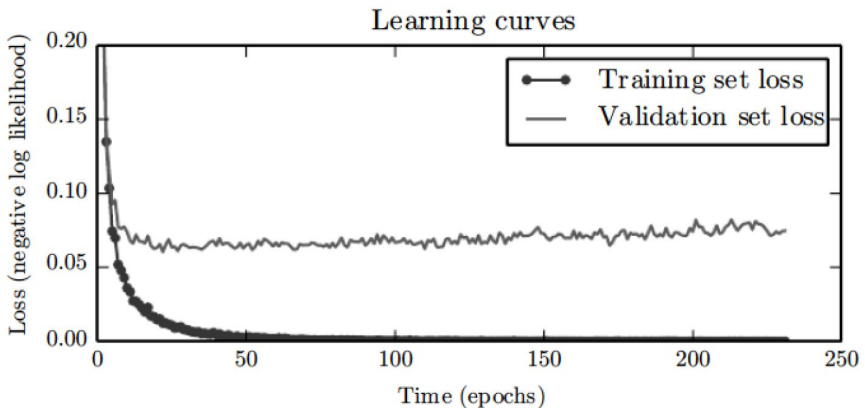
By regularizing on time, the complexity of the model can be controlled, improving generalization.

Regularization by Early Stopping

When training large models with large capacity, we often observe that training error decreases steadily over time, but validation set error begins to rise again.

Instead of running our optimization algorithm until we reach a (local) minimum of validation error, we run it until the error on the validation set has not improved for some amount of time.

Every time the error on the validation set improves, we store a copy of the model parameters.



Early Stopping 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^*

Early Stopping Analysis

- Consider the quadratic approximation of the cost function in the neighbourhood of the empirically optimal value of the weights $\mathbf{w}^*$

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

- Thus,

$$\nabla_w \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*)$$

- For simplicity, we assume $\mathbf{w}^{(0)} = \mathbf{0}$
- Then, the gradient descent updates can be expressed as

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \mathbf{H}(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*)$$

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \eta \mathbf{H})(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*)$$

Early Stopping Analysis (compare to L^2 regularization)

- With $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$, we rewrite the above as

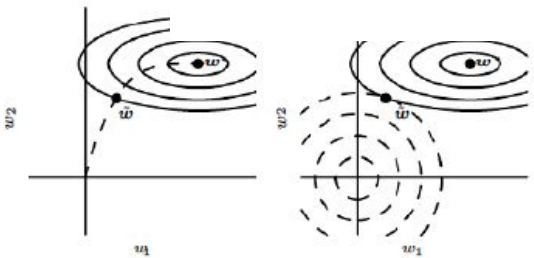
$$\mathbf{Q}^\top(\mathbf{w}^{(\tau)} - \mathbf{w}^*) = (\mathbf{I} - \eta\mathbf{\Lambda})\mathbf{Q}^\top(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*)$$

- Assuming $\mathbf{w}^{(0)} = \mathbf{0}$

$$\mathbf{Q}^\top\mathbf{w}^{(\tau)} = \left(\mathbf{I} - (\mathbf{I} - \eta\mathbf{\Lambda})^\tau\right)\mathbf{Q}^\top\mathbf{w}^*$$

- The closed form solution of L^2 regularization can be written as

$$\mathbf{Q}^\top\tilde{w} = \left(\mathbf{I} - (\mathbf{I} + \alpha\mathbf{\Lambda})^{-1}\alpha\right)\mathbf{Q}^\top\mathbf{w}^*$$



Regularization for Deep Learning

Dataset Augmentation

Adversarial Training

Parameter Sharing

Multi-Task Learning

Bootstrap Aggregating (Bagging)

Dropout

Dataset Augmentation (More Data)

More complex models require more training data and training data is always finite.

- **It is not always possible to have large datasets at the outset:**
  - **For certain tasks, especially image classification, it may be possible to generate "fake" data.**
  - **For example, operations like translating the training images a few pixels in each direction can often greatly improve generalization**
  - **crop, rotate, translate, synthetically generate using computer graphics**
- **One must be careful not to apply transformations that would ~~change the correct class~~.**
  - **For example, optical character recognition tasks require recognizing the difference between 'b' and 'd' and the difference between '6' and '9', so horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.**

Dataset Augmentation

Color changes

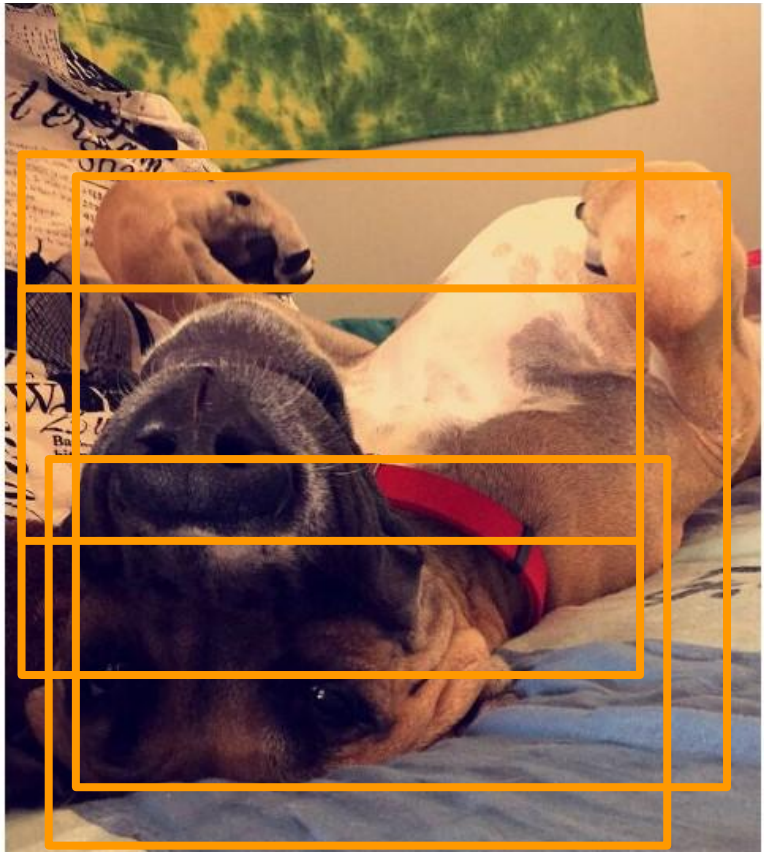
Horizontal Flips

Dataset Augmentation

Resize image

Dataset Augmentation

Training: sample random crops



Dataset Augmentation

Computer Animation

?

Dataset Augmentation

Melange of :

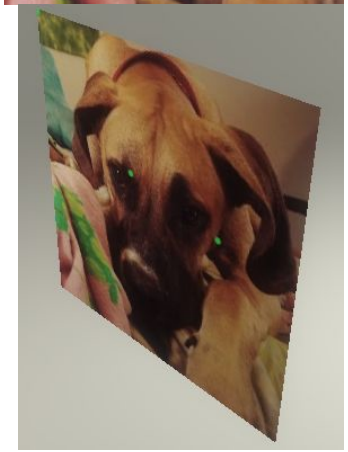
- translation

- rotation

- stretching

- shearing

- lens distortions



Adversarial Training

Intentionally constructed, hard to "see", purposely wrong examples.

Even neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intentionally constructed

- **by using an optimization procedure to search for an input x' near a data point x such that the model output is very different at x'.**
- **In many cases, x' can be so similar to x that a human observer cannot tell the difference between the original example and the adversarial example.**

Adversarial Training

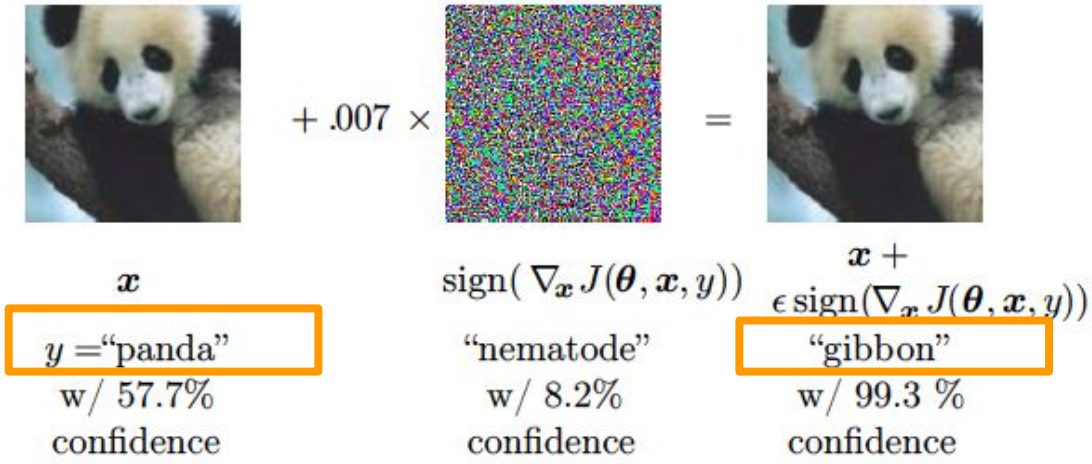
- Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result.
- To understand the implications of it, let's consider a linear mapping,

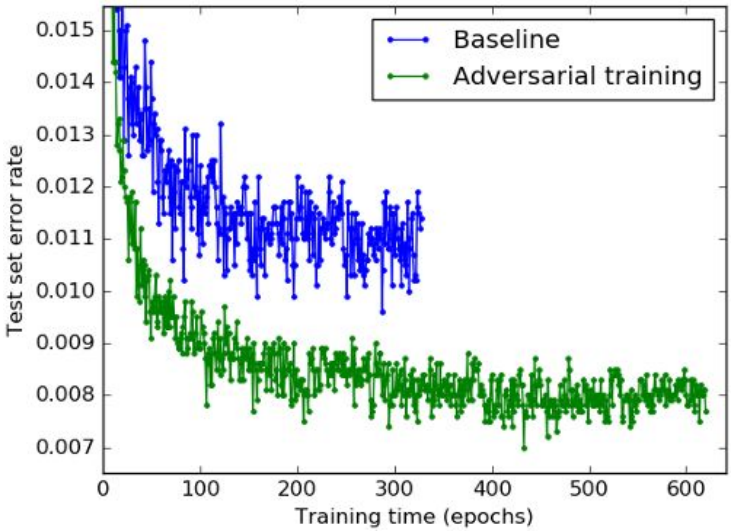
$$\hat{y} = \mathbf{w}^\top \mathbf{x}$$

- If we add a small $\boldsymbol{\epsilon}$ to the input in the linear case we get $\hat{y} = \mathbf{w}^\top \mathbf{x} + \mathbf{w}^\top \boldsymbol{\epsilon}$. We can maximize this increase by assigning $\boldsymbol{\epsilon} = \epsilon \ \text{sign}(\mathbf{w})$. When $\mathbf{w}$ is high dimensional, this can result in a very large change in the output.

Adversarial Training

One can reduce the error rate on the original i.i.d. test set via adversarial training: training on adversarially perturbed examples from the training set.





Parameter Sharing

Another way to regularize or reduce the complexity of a deep model is through parameter sharing.

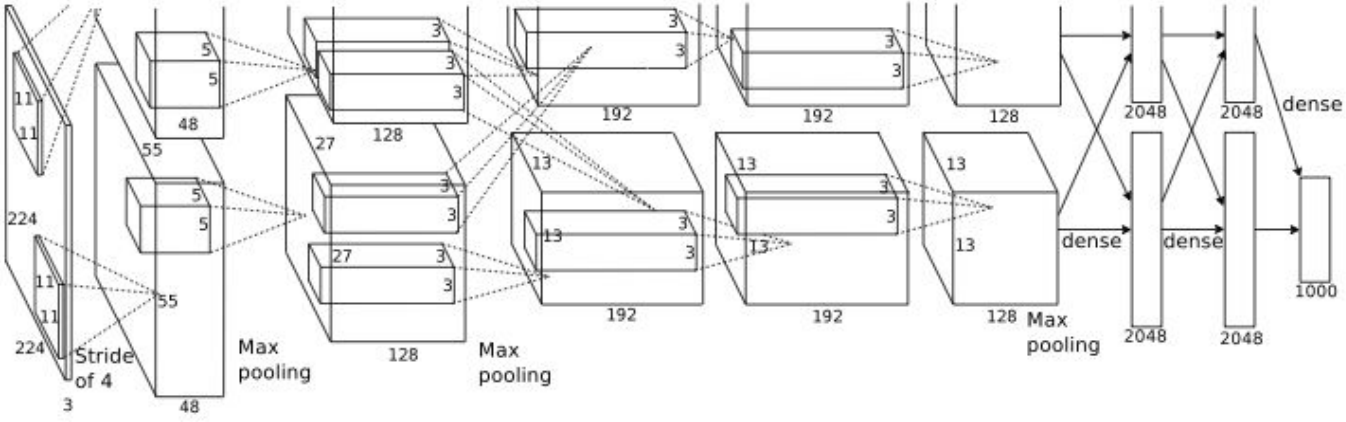
Various components of the model are made to share a unique set of parameters

The most popular and extensive use of parameter sharing occurs in convolutional neural networks (CNNs) applied to computer vision.

Example

ConvNet

http://libccv.org/doc/doc-convnet/



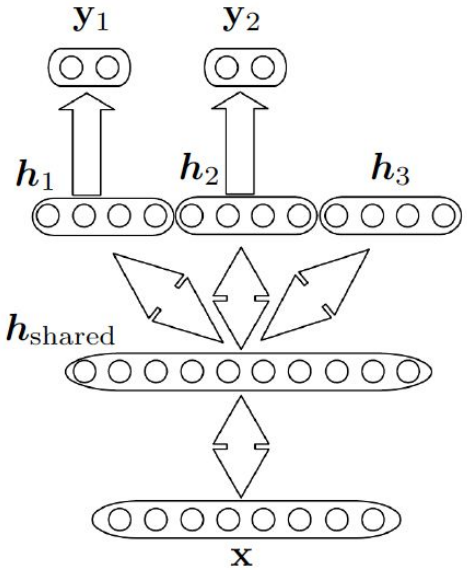
Multi-Task Learning

Parameter sharing restricts different components of the model to share parameters for a particular task.

In multi-task learning, on the other hand, a part of the model is shared across various tasks.

Biological vision systems share initial processing across all tasks as well.

Assuming that sharing is justified, when a part of a model is shared across tasks, that part of the model is more constrained towards "good" values (in the context of generalization).

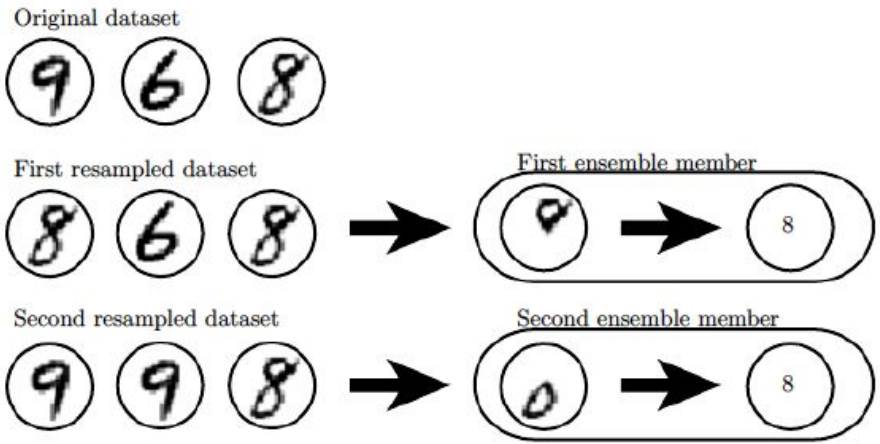


Bootstrap Aggregating (Bagging)

Train a model by random sampling with replacement (re-use training samples).

- **Repeat this problem k times to produce k separate models**
- **Have all models vote on output for test samples**

Observe how first model is good for "detecting" the top circle...



Bootstrap Aggregating (Bagging)

- Bagging (short for bootstrap aggregating) is a technique for reducing generalization error by combining several models
- Consider for example a set of k regression models, eaxh of which makes an error, ϵ_i on each example. Let $\mathbb{E}(\epsilon_i) = 0, \mathbb{E}(\epsilon_i^2) = \nu, \mathbb{E}(\epsilon_i \epsilon_j) = c$. Then

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k}\nu + \frac{k-1}{k}c$$

- Bagging involves constructing the k models by training them on k different datasets obtained by the process of bootstrapping. Bootstrapping involves uniformly sampling from the training dataset with replacement.

Bagging -> Ensemble Modelling

Model averaging is one of the ensemble methods:

- **Ensemble models are being used by winning models in many competitions**
- **Ensemble methods are memory hogs, and since they require running several models to train and evaluate -> they are slow**
- **Ensemble models are good at improving accuracy (lower error rates), but they are not friendly for scaling**

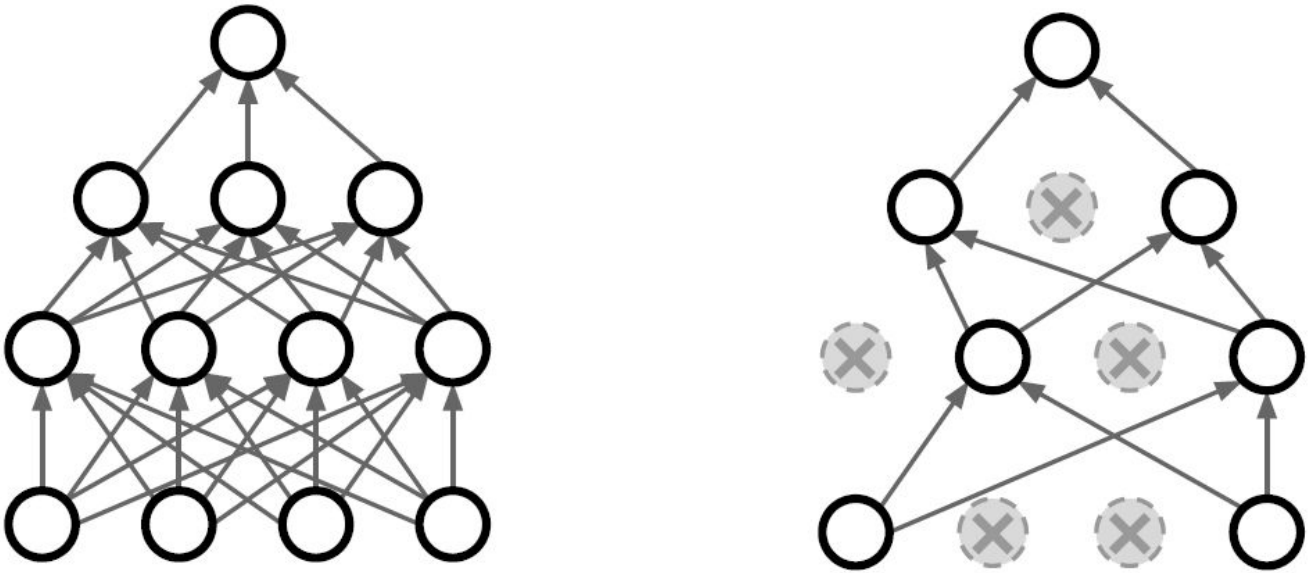
Dropout

Densely connected layers occupy most of the parameters, so the ANN is prone to overfitting.

One method to reduce overfitting is dropout:

- **At each training stage, individual nodes are either "dropped out" of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left**
- **Incoming and outgoing edges to a dropped-out node are also removed.**
- **Only the reduced network is trained on the data in that stage.**
- **The removed nodes are then reinserted into the network with their original weights.**
- **https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf**

Dropout



Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Dropout

Dropout can be thought of as a method of making bagging practical for large neural networks.

Dropout trains the ensemble consisting of all sub-networks that can be formed by removing units from an underlying base network.

A fully connected layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible active/inactive combinations; compared to $\sim 10^{82}$ atoms in the universe.

Dropout can be more effective than other standard computationally inexpensive regularizers, such as weight decay.

A practical look: https://visualstudiomagazine.com/articles/2014/05/01/neural-network-dropout-training.aspx

Dropout

In the training stages, the probability that a hidden node will be dropped is usually 0.5.

For input nodes, the probability of dropping should be lower (~0.2), because information is explicitly lost when some input nodes are ignored.

At testing time after training has finished, we would ideally like to find a sample average of all possible 2^n dropped-out networks

- This is unfeasible for large values of n.

Dropout

One can find an approximation of the sample average of all possible networks,

by using the full network with each node's output weighted by a factor of p,

so the expected value of the output of any node is the same as in the training stages.

- This is an important contribution of the dropout method: it effectively uses 2^n neural nets constructed from the mother-net, making it feasible to have effective model combination
- At test time only a single network needs to be tested.

Dropout

By avoiding training all nodes on all training data, dropout decreases overfitting.

The method also significantly improves the speed of training. This makes model combination practical, even for deep neural nets.

The technique seems to reduce node interactions, leading them to learn more robust features that better generalize to new data.

Dropout - An Implementation Sketch

Dropout Naive

Apply dropout during training

Scale the activations

```
1  ##############################################################################
2  # Dropout
3  # **** Dropout Naive (Not Recommended) *****
4
5  p = 0.5 # Probability of keeping a neuron active; higher values = less dropout
6
7  def train_step(X):      # X is the matrix containing data
8      # Example: 3-layer neural network------------------------------
9      # forward pass ------------------------------------------------
10     H1 = np.maximum (0, np.dot(W1, X) + b1)
11     U1 = np.random.rand(*H1.shape) < p   # dropout: mask for layer 1    <---
12     H1 *= U1 # dropout application: layer 1                            <---
13     H2 = np.maximum (0, np.dot(W2, H1) + b2)
14     U2 = np.random.rand(*H2.shape) < p   # dropout: mask for layer 2    <---
15     H2 *= U2 # dropout application: layer 2                            <---
16     out = np.dot(W3, H2) + b3
17     # backward pass -----------------------------------------------
18     # Compute_gradients on(through) non-dropped nodes, not shown
19     #  ...
20     # Execute parameter update, not shown
21     #  ...
22
23 def predict(X)
24     # forward pass using ensemble results -------------------------
25     H1 = np.maximum(0, np.dot(W1, X ) + b1) * p # p: scales the activations <---
26     H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # p: scales the activations <---
27     out = np.dot(W3, H2) + b3
28
```

Dropout - An Implementation Sketch
Dropout Inverted

Apply dropout during training

No change! (No scaling)

```
31  ##################################################################################
32  # Dropout
33  #  **** Dropout Inverted (Recommended) *****
34
35  p = 0.5 # Probability of keeping a neuron active; higher values = less dropout
36
37  def train_step(X):      # X is the matrix containing data
38      # Example: 3-layer neural network------------------------------
39      # forward pass ------------------------------------------------
40      H1 = np.maximum (0, np.dot(W1, X) + b1)
41      U1 = (np.random.rand(*H1.shape) < p) / p # dropout: mask for layer 1    <---
42      H1 *= U1 # dropout application: layer 1                                 <---
43      H2 = np.maximum (0, np.dot(W2, H1) + b2)
44      U2 = (np.random.rand(*H2.shape) < p) / p   # dropout: mask for layer 2  <---
45      H2 *= U2 # dropout application: layer 2                                 <---
46      out = np.dot(W3, H2) + b3
47      # backward pass -----------------------------------------------
48      # Compute_gradients on(through) non-dropped nodes, not shown
49      #  ...
50      # Execute parameter update, not shown
51      #  ...
52
53  def predict(X)
54      # forward pass using ensemble results -------------------------
55      H1 = np.maximum(0, np.dot(W1, X ) + b1) # Observe: scaling with p not necessary <-
56      H2 = np.maximum(0, np.dot(W2, H1) + b2) # Observe: scaling with p not necessary <-
57      out = np.dot(W3, H2) + b3
58
```

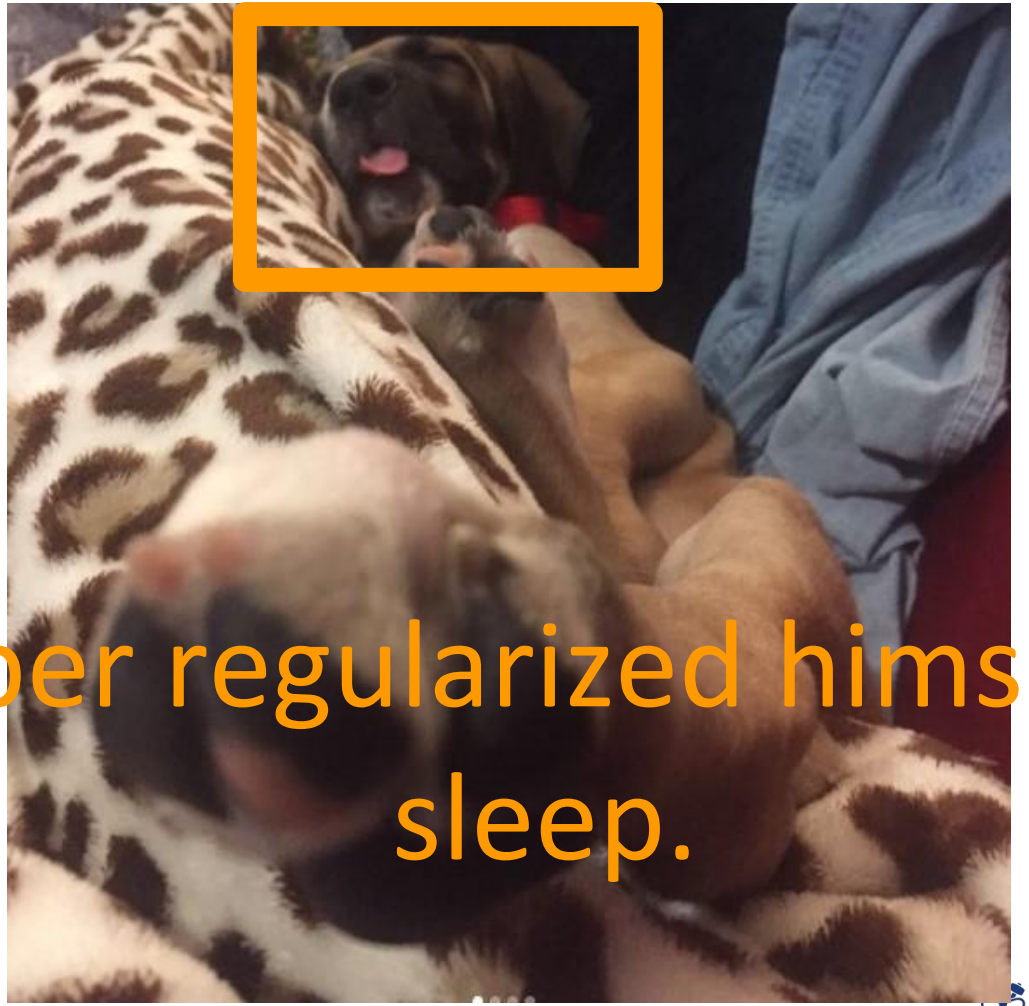
DropConnect

- DropConnect is the generalization of dropout in which each connection, rather than each output unit, can be dropped with probability 1-p.
  - Each unit thus receives input from a random subset of units in the previous layer.
- DropConnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output vectors of a layer. In other words, the fully connected layer with DropConnect becomes a sparsely connected layer in which the connections are chosen at random during the training stage.
- DropConnect becomes Dropout if all connections into the neuron (unit) are dropped.

Regularization (recap)

Regularization is a process of introducing additional information,

with the hope that this improves generalization, i.e., the ability to correctly handle data that the network has not trained on.



Timber regularized himself to sleep.