Regularization

Reading

- Deep Learning: chapter 7
  - https://www.deeplearningbook.org/contents/regularization.html

- Regularization overview, with a deep learning bias

Regularization

- Occam's Razor
  - Try to use simplest model family possible

- Neural nets can easily overfit any dataset we have come up with
  - Regularization adds constraints to keep models well-behaved
- A bit of a funky concept
  - We want to minimize the loss, but we also want to minimize it the right way!
  - Comes to indicate that our losses could be improved

Parameter Norm Penalties

- One of the most popular regularizations
- Suppose original loss is $J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$
- Come up with an extra term $\Omega(\boldsymbol{\theta})$ that penalizes the parameters
- Final loss becomes

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

  – where α is a (small) hyper-parameter
- Often reduces variance at the expense of some bias

L_2 Penalty

- Most standard penalty

$$\Omega(\boldsymbol{\theta}) = \boldsymbol{w}^T\boldsymbol{w}$$

  - Recall that $\boldsymbol{\theta} = [\boldsymbol{w}, \boldsymbol{b}]$
- Usually applied only to weights, not to biases
  - Regularizing biases leads to underfitting without major variance benefits
- Also known as weight decay
  - Recall that weights are updated as follows

$$\boldsymbol{w}' = \boldsymbol{w} - \epsilon\nabla\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$$

  - With L_2 penalty, the update is

$$\begin{aligned}\boldsymbol{w}' &= \boldsymbol{w} - \epsilon\left(2\alpha\boldsymbol{w} + \ \nabla J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})\right)\\ &= (1 - 2\alpha\epsilon)\boldsymbol{w} - \epsilon\nabla J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})\end{aligned}$$

L_2 Penalty, cont'd

- If true loss is quadratic, L_2 penalty penalizes learning in directions where the loss isn't affected
  - Prevents learning spurious functionality due to overparameterization (proof in book)
- Same idea in general – keep weights small unless necessary
  - Simplifies models and improves robustness
- In linear regression, makes fitting more robust to variance

$$-2\boldsymbol{X}\boldsymbol{y} + 2\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{w} + 2\alpha\boldsymbol{w} = 0$$

- Then $\boldsymbol{w}^* = \left(\boldsymbol{X}\boldsymbol{X}^T + \alpha\boldsymbol{I}\right)^{-1}\boldsymbol{X}\boldsymbol{y}$
- I have also used L_2 penalty in my research

L_1 Penalty

- A slightly less standard penalty

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

- Note that the derivative of $\Omega(\boldsymbol{\theta}) = sign(\boldsymbol{w})$
- So the weight update is now

$$\boldsymbol{w}' = \boldsymbol{w} - \epsilon\left(\alpha sign(\boldsymbol{w}) + \nabla J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})\right)$$

  - i.e., a constant factor along the direction of the 1-norm
  - Might lead to sparser weight matrices (more 0s)
- Hard to derive nice mathematical formulae
- Overall regularization effect is similar to L_2

Norm Penalties as Constrained Optimization

- Instead of penalizing bigger weights, we can impose an explicit constraint

$$\boldsymbol{\theta}^* = argmin_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$$
$$subject\ to \quad \Omega(\boldsymbol{\theta}) < k$$

- Explicit constraints may stabilize the learning process in certain cases (since the loss is simplified)
  - However, it may hurt in others
  - Constrained non-convex optimization is a hard problem
- Can also reformulate the problem as

$$\boldsymbol{\theta}^* = argmin_{\boldsymbol{\theta}} \max_{\alpha} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k)$$

  - Same idea, optimization algorithm slightly different
  - Dual formulation of the constrained problem above

Data Augmentation

- Using fake data for training is not always a good idea!
  - But sometimes OK…
- In image classification, we can usually generate “new” data from a given dataset
  - Rotate, translate, add white noise to images
  - Useful because it discourages learning spurious relationships (similar to regularization)
  - You can overdo it, though. Thoughts?

Weight Noise Robustness

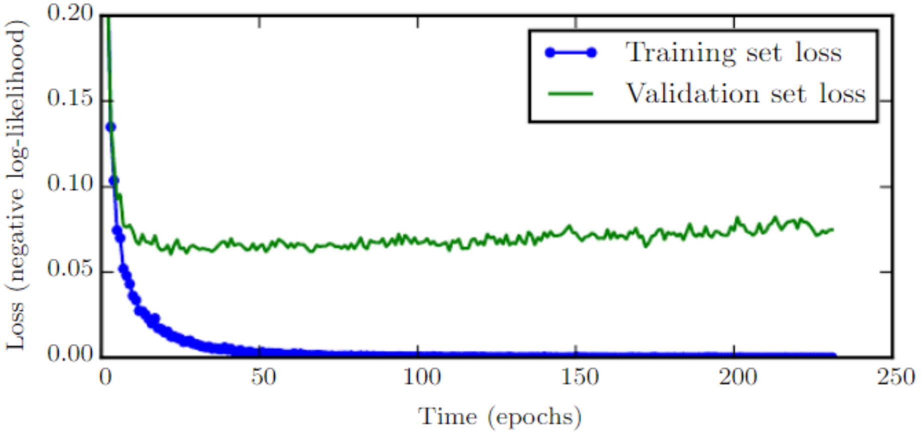
- Can also apply noise to the weights
  - E.g., in Bayesian neural networks every weight is drawn from a Gaussian with learned parameters
  - Pushes weights to region where model is less sensitive to perturbations
- Can also do it at the output layer
  - “Soft” labels aka label smoothing
    - E.g., one-hot labels are not (0,1) but maybe (0.1, 0.9)
  - Discourages the NN from learning very big weights in trying to approximate the hard 0/1 outputs

Semi-supervised learning

- A lot of deep learning has to do with learning representations of the training data that are separated in some embedding space
- What if we learn the embedding separately from the classifier?
  - i.e., learn a generative model of the data first
  - An active research area, improves robustness a great deal
- Many types of generative models, such as generative adversarial networks (GANs), variational autoencoders (VAEs) and others

Early Stopping

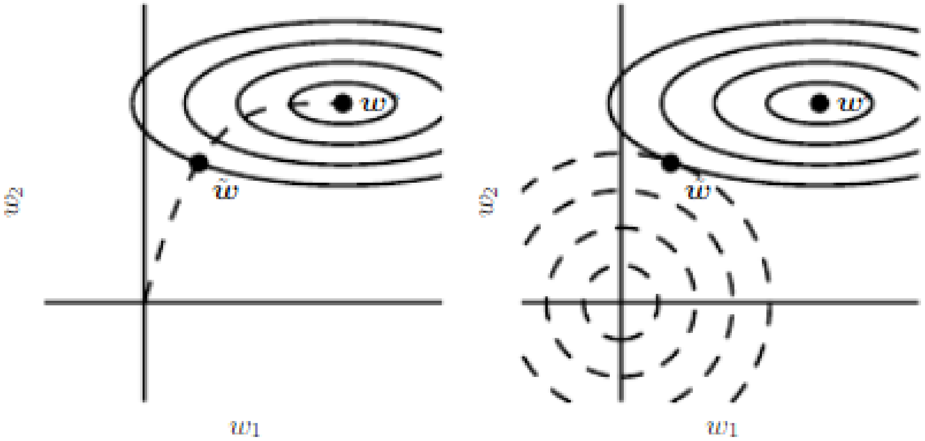
- Often, training loss keeps decreasing while validation loss starts increasing
  - A sign of overfitting



- Can stop training as soon as this happens
  - (or save trained weights frequently and go back to that checkpoint)
  - Technique called **early stopping**

Early Stopping, cont'd

- Simple and effective
- Essentially another hyper-parameter
- Periodically storing weights is not a major overhead
- However, if early stopping is necessary, then you're violating Occam's Razor
  - If your model overfits drastically, you should consider using a simpler model
- Related to L_2 penalty



Ensemble Methods

- A very old and effective idea in ML
- Train multiple models on the same task and average their outputs
- Very effective if done well (i.e., models make independent errors)
- Suppose you have k models, each makes (regression) error ϵ_i

$$\epsilon_i = f_i(\boldsymbol{X}) - Y$$

  – To simplify math, assume $\mathbb{E}[\epsilon_i] = 0$

- The ensemble's output is then

$$f(\boldsymbol{X}) = \frac{1}{k}\sum_i f_i(\boldsymbol{X})$$

Ensemble Methods, cont'd

- Suppose you have k models, each makes error ϵ_i (where ϵ_i is a zero-mean random variable)

$$\epsilon_i = f_i(\boldsymbol{X}) - Y$$

  - To simplify math, assume $\mathbb{E}[\epsilon_i] = 0$

- The ensemble's output is then

$$f(\boldsymbol{X}) = \frac{1}{k}\sum_i f_i(\boldsymbol{X})$$

- What is the ensemble's error?

$$\begin{aligned}\epsilon &= f(\boldsymbol{X}) - Y \\ &= \frac{1}{k}\sum_i (f_i(\boldsymbol{X}) - Y) \\ &= \frac{1}{k}\sum_i \epsilon_i\end{aligned}$$

Ensemble Methods, cont'd

- What is the ensemble's expected error?
  - We know $\mathbb{E}[\epsilon] = 0$, since each $\mathbb{E}[\epsilon_i] = 0$
- But the variance can be drastically reduced
- Suppose they have same variance $\mathbb{E}\left[\epsilon_i^2\right] = v$, and covariances are $\mathbb{E}\left[\epsilon_i \epsilon_j\right] = c$

Ensemble Methods, cont'd

- Suppose $\mathbb{E}\left[\epsilon_i^2\right] = v$ and $\mathbb{E}\left[\epsilon_i \epsilon_j\right] = c$
- Expected squared error (i.e., variance) is

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

$$= \mathbb{E}\left[\frac{1}{k^2}(\epsilon_1 + \cdots + \epsilon_k)(\epsilon_1 + \cdots + \epsilon_k)\right]$$

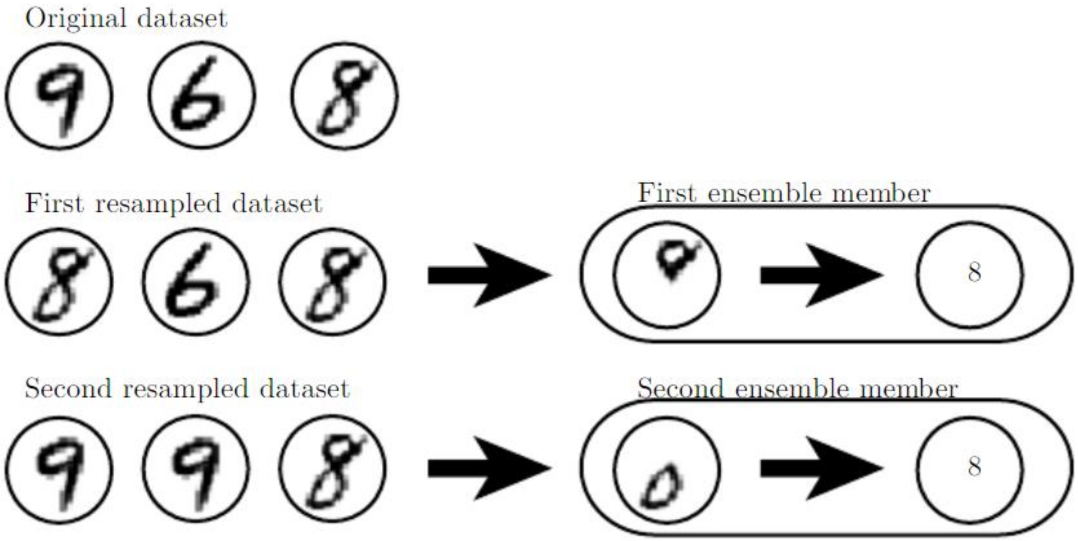
$$= \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{i \neq j} \epsilon_i \epsilon_j\right)\right]$$

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

- If perfect correlation, $v = c$, average doesn't help
- If no correlation, $c = 0$, squared error inversely proportional to number of models

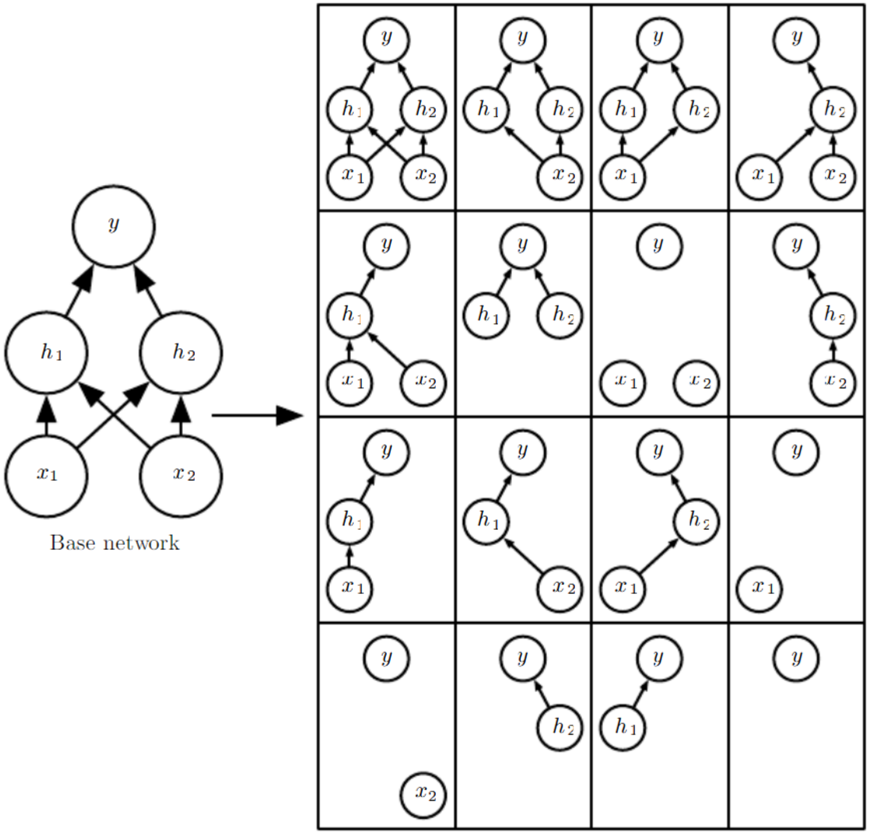
Ensemble Methods, cont'd

- Suppose you resample original dataset and train a different model each time
  - Models learn different important features
  - More robust overall since spurious features averaged out
  - This is the idea of boosting



Dropout

- Generated a lot of attention a few years ago
- A computationally cheap way to approximate ensemble of methods
- The “ensemble” is the set of all subnetworks of a given NN
  - To eliminate a neuron, just multiply its output by 0
- Slightly different from classic ensembles since data is the same

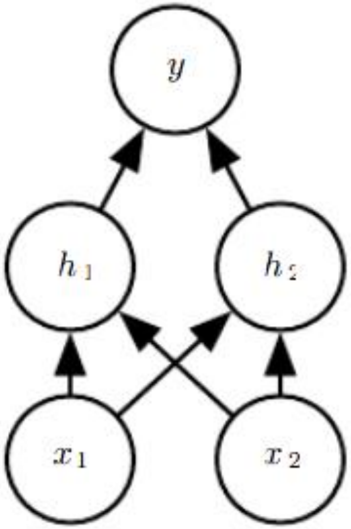


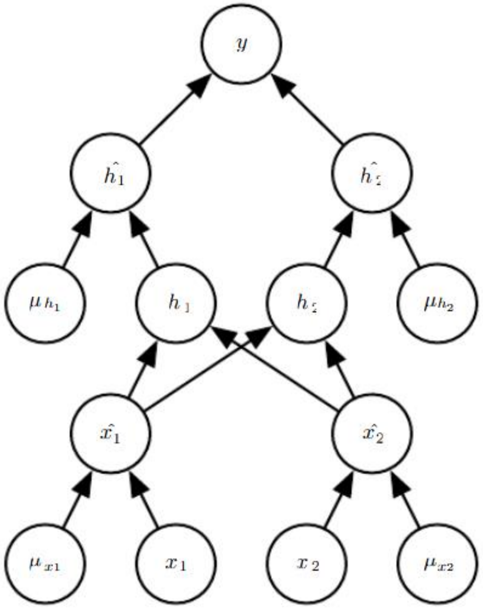
Base network

Ensemble of subnetworks

Dropout, cont'd

- During training, select a random bit mask μ for each iteration of gradient descent
  - Enumerating all subnetworks is intractable
  - E.g., keep input neurons with a probability of 0.8 and hidden neurons with probability of 0.5
  - Once you have determined μ, train as before using backprop





Dropout Prediction

- Suppose we have trained our model with dropout
- How do we predict the label given a new example?
- Ideally, we enumerate all subgraphs and compute the mean of all subnetwork outputs
  - What's the challenge with this?
    - Exponentially many subnetworks
- One option is to sample a number of masks and average over those (a reasonably good estimate of the true average)
  - Not deterministic
- Better idea: output expected value of each neuron (how?)
  - multiply all weights by the keep probability, p
  - binary variable with parameter p

Dropout Prediction

- One idea that works very well in practice is to multiply all weights by the keep probability, p
  - Most common choice
- Another idea is to sample masks and compute geometric mean

$$\mathbb{P}_{ensemble}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} \mathbb{P}_{dropout}(y \mid \boldsymbol{x}, \boldsymbol{\mu})}$$

  - Normalize over classes (doesn't sum up to 1 otherwise)
  - For some architectures, this is the same as multiplying the weights by p
    - E.g., networks with one layer and a softmax output
    - See proof in book

Beware of overfitting!

- Neural networks can perfectly overfit any existing dataset
  - Even if you randomly shuffle the labels
- In some sense, not clear why NNs perform as well as they do
- Always a good idea to use as small a model as possible
  - If your training accuracy is significantly higher than test accuracy, then likely you need to regularize or reduce your model
- Deep learning is a powerful tool but it requires a strong understanding of statistics

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. "Understanding deep learning (still) requires rethinking generalization." *Communications of the ACM* 64, no. 3 (2021): 107-115.

Summary

- Many, many ways to regularize
- Usually trial and error is the best approach
- If you set up everything well (right model, right features, etc.), you may not even need much regularization
- An L_2 regularization will typically get you a long way