COMS 4776 Lecture 9: Generalization

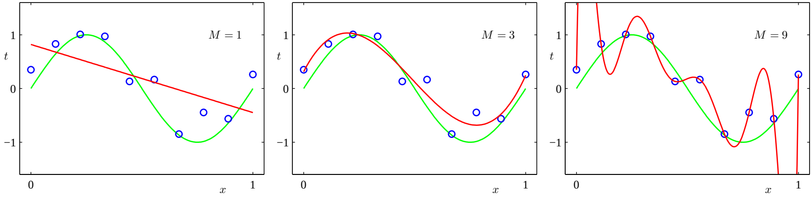
Richard Zemel

Overview

- We've focused so far on how to *optimize* neural nets — how to get them to make good predictions on the training set.
- How do we make sure they generalize to data they haven't seen before?
- Even though the topic is well studied, it's still poorly understood.

Generalization

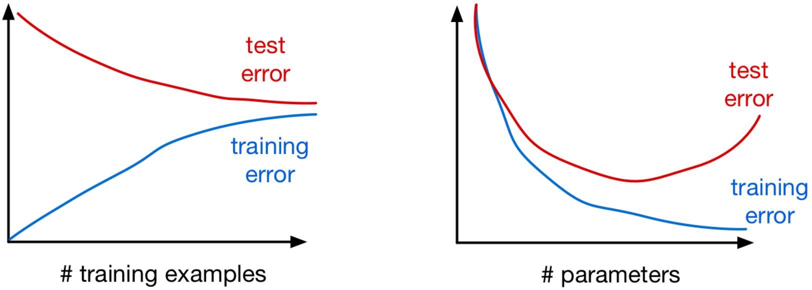
Recall: overfitting and underfitting



We'd like to minimize the generalization error, i.e. error on novel examples.

Generalization

- Training and test error as a function of # training examples and # parameters:



Our Bag of Tricks

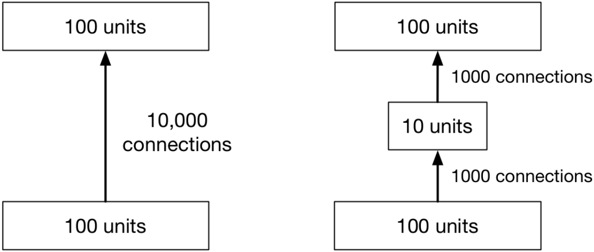
- How can we train a model that's complex enough to model the structure in the data, but prevent it from overfitting? I.e., how to achieve low bias and low variance?
- Our bag of tricks
  - data augmentation
  - reduce the number of paramters
  - weight decay
  - early stopping
  - ensembles (combine predictions of different models)
  - stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks.

Data Augmentation

- The best way to improve generalization is to collect more data!
- Suppose we already have all the data we're willing to collect. We can augment the training data by transforming the examples. This is called data augmentation.
- Examples (for visual recognition)
  - translation
  - horizontal or vertical flip
  - rotation
  - smooth warping
  - noise (e.g. flip random pixels)
- Only warp the training, not the test, examples.
- The choice of transformations depends on the task. (E.g. horizontal flip for object recognition, but not handwritten digit recognition.)

Reducing the Number of Parameters

- Can reduce the number of layers or the number of paramters per layer.
- Adding a linear bottleneck layer is another way to reduce the number of parameters:



- The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
- Remember how linear layers don't make a network more expressive? They might still improve generalization.

Weight Decay

- We've already seen that we can regularize a network by penalizing large weight values, thereby encouraging the weights to be small in magnitude.

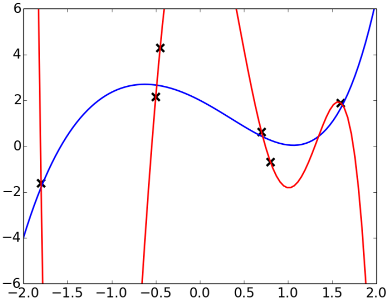
$$\mathcal{J}_{\mathrm{reg}} = \mathcal{J} + \lambda \mathcal{R} = \mathcal{J} + \frac{\lambda}{2} \sum_j w_j^2$$

- We saw that the gradient descent update can be interpreted as weight decay:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\
&= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}
\end{aligned}$$

Weight Decay

Why we want weights to be small:



$$y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2$$

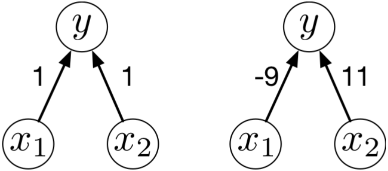
$$y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12$$

The red polynomial overfits. Notice it has really large coefficients.

Weight Decay

Why we want weights to be small:

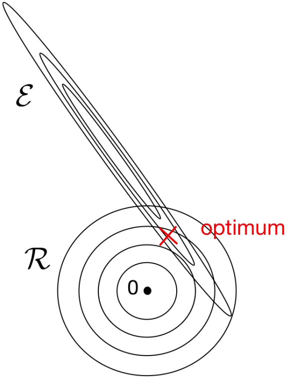
- Suppose inputs x_1 and x_2 are nearly identical. The following two networks make nearly the same predictions:



- But the second network might make weird predictions if the test distribution is slightly different (e.g. x_1 and x_2 match less closely).

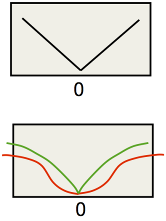
Weight Decay

- The geometric picture:

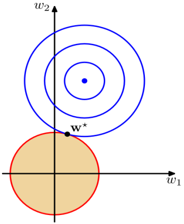


Weight Decay

- There are other kinds of regularizers which encourage weights to be small, e.g. sum of the absolute values.
- These alternative penalties are commonly used in other areas of machine learning, but less commonly for neural nets.
- Regularizers differ by how strongly they prioritize making weights exactly zero, vs. not being very large.

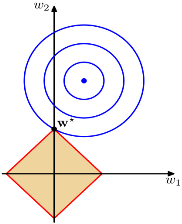


— Hinton, Coursera lectures



L2 regularization

$$\mathcal{R} = \sum_i w_i^2$$



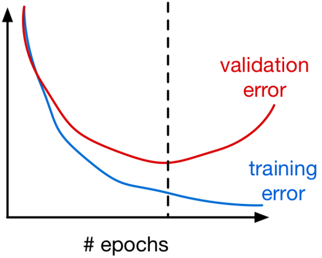
L1 regularization

$$\mathcal{R} = \sum_i |w_i|$$

— Bishop, *Pattern Recognition and Machine Learning*

Early Stopping

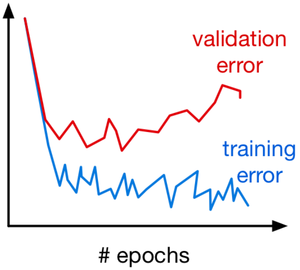
- We don't always want to find a global (or even local) optimum of our cost function. It may be advantageous to stop training early.



- Early stopping: monitor performance on a validation set, stop training when the validtion error starts going up.

Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.



- Determining when the validation error has actually leveled off can be tricky.

Early Stopping

- Why does early stopping work?
  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.
  - If you are using sigmoidal units, and the weights start out small, then the inputs to the activation functions take only a small range of values.
    - Therefore, the network starts out approximately linear, and gradually becomes more nonlinear (and hence more powerful).

Ensembles

- If a loss function is convex (with respect to the predictions), you have a bunch of predictions, and you don't know which one is best, you are always better off averaging them.

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1$$

- This is true no matter where they came from (trained neural net, random guessing, etc.). Note that only the loss function needs to be convex, not the optimization problem.
- Examples: squared error, cross-entropy, hinge loss
- If you have multiple candidate models and don't know which one is the best, maybe you should just average their predictions on the test data. The set of models is called an ensemble.
- Averaging often helps even when the loss is nonconvex (e.g. 0–1 loss).

Ensembles

- Some examples of ensembles:
  - Train networks starting from different random initializations. But this might not give enough diversity to be useful.
  - Train networks on differnet subsets of the training data. This is called bagging.
  - Train networks with different architectures or hyperparameters, or even use other algorithms which aren't neural nets.
- Ensembles can improve generalization quite a bit, and the winning systems for most machine learning benchmarks are ensembles.
- But they are expensive, and the predictions can be hard to interpret.

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units (i.e. sets their activations to zero).

$$h_j = \begin{cases} \phi(z_j) & \text{with probability } 1 - \rho \\ 0 & \text{with probability } \rho, \end{cases}$$

  where ρ is a hyperparameter.
- Equivalently,

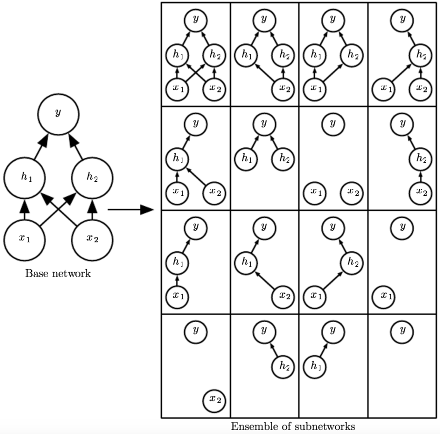
$$h_j = m_j \cdot \phi(z_j),$$

  where m_j is a Bernoulli random variable, independent for each hidden unit.
- Backprop rule:

$$\overline{z_j} = \overline{h_j} \cdot m_j \cdot \phi'(z_j)$$

Stochastic Regularization

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

Dropout

Dropout at test time:

- Most principled thing to do: run the network lots of times independently with different dropout masks, and average the predictions.
    - Individual predictions are stochastic and may have high variance, but the averaging fixes this.
- In practice: don't do dropout at test time, but multiply the weights by $1 - \rho$
    - Since the weights are on $1 - \rho$ fraction of the time, this matches their expectation.

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time: $\tilde{y}^{(i)} = 2 \sum_j m_j^{(i)} w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$. $\mathbb{E}_m[\tilde{y}^{(i)}] = y^{(i)}$.

$$\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N} \sum_{i=1}^{N} \mathbb{E}_m[(\tilde{y}^{(i)} - t^{(i)})^2]$$

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time: $\tilde{y}^{(i)} = 2\sum_j m_j^{(i)} w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$. $\mathbb{E}_m[\tilde{y}^{(i)}] = y^{(i)}$.

$$\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N} \mathbb{E}_m[(\tilde{y}^{(i)} - t^{(i)})^2]$$

The bias-variance decomposition of the squared error gives:

$$\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\mathrm{Var}_m[\tilde{y}^{(i)}]$$

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time: $\tilde{y}^{(i)} = 2\sum_j m_j^{(i)} w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$. $\mathbb{E}_m[\tilde{y}^{(i)}] = y^{(i)}$.

$$\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N} \mathbb{E}_m[(\tilde{y}^{(i)} - t^{(i)})^2]$$

The bias-variance decomposition of the squared error gives:

$$\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\mathrm{Var}_m[\tilde{y}^{(i)}]$$

Assume weights, inputs and masks are independent and $\mathbb{E}[x] = 0$.

$$\begin{aligned}\mathbb{E}_m[\mathcal{J}] &= \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\sum_j \mathrm{Var}_m[2m_j^{(i)} x_j^{(i)} w_j] \\ &= \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2}\sum_j \mathrm{Var}[x_j] w_j^2\end{aligned}$$

Stochastic Regularization

- Dropout can help performance quite a bit, even if you're already using weight decay.
- Lots of other stochastic regularizers have been proposed:
  - Batch normalization (mentioned last week for its optimization benefits) also introduces stochasticity, thereby acting as a regularizer.
  - The stochasticity in SGD updates has been observed to act as a regularizer, helping generalization.
    - Increasing the mini-batch size may improve training error at the expense of test error!

Our Bag of Tricks

- Techniques we just covered:
  - data augmentation
  - reduce the number of paramters
  - weight decay
  - early stopping
  - ensembles (combine predictions of different models)
  - stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks.