EECS 182/282A Deep Neural Networks Lecture 4 - 09/06/2022

Lecture 4: Basic Principles Part II

Instructor: Anant Sahai *Scribe: Austin Zane*

1 Regularization

1.1 Explicit Regularization

This is a recap of the previous lecture. For the ordinary least squares problem $\boldsymbol{X}\boldsymbol{w} \approx \boldsymbol{y}$, the solution is given by

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y}.$$

However, this can give us very large and not very useful values for the parameter vector. In such cases, it is often beneficial to "regularize" the parameters when training so that they stay reasonable. A common choice is ridge regularization, which uses a modified version of the least-squares loss function:

$$\arg\min_{\boldsymbol{w}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2.$$

A bit of vector calculus shows that the new solution for $\hat{\boldsymbol{w}}$ is given by

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y} = \boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y},$$

where the first formula is the classic ridge form and the second is kernel ridge form. The equivalence can be seen as follows,

$$\begin{aligned}
\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y} &= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{X}^\mathsf{T}\right)\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda\boldsymbol{I}\right)\boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y} \\
&= \boldsymbol{X}^\mathsf{T}\left(\boldsymbol{X}\boldsymbol{X}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y}.
\end{aligned}$$

There is a way of thinking of the first formulation as solving the primal and the second as solving the dual.

1.2 Data and Feature Augmentation

Instead of explicitly changing the loss function, we can add d "fake" data points to our data matrix to achieve the same regularizing effect (as proven in previous lecture):

$$\begin{bmatrix}\boldsymbol{X} \\ \sqrt{\lambda}\boldsymbol{I}_d\end{bmatrix}\boldsymbol{w} \approx \begin{bmatrix}\boldsymbol{y} \\ \boldsymbol{0}_d\end{bmatrix},$$

where the new data matrix is in $\mathbb{R}^{(n+d)\times d}$ and the new response is a vector in $\mathbb{R}^{n+d}$. This is an important concept and can be thought of as improving the conditioning number of the data matrix. Plugging the new data matrix and response vector into the classic OLS solution immediately gives us the solution for the ridge regularized problem.

Another equivalent option is adding n fake features:

$$\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{w}\\ \boldsymbol{f}\end{bmatrix} = \boldsymbol{y}.$$

We use the Moore-Penrose pseudoinverse to solve for the new weight vector,

$$\begin{bmatrix}\boldsymbol{w}\\ \boldsymbol{f}\end{bmatrix} = \begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}^{\dagger}\boldsymbol{y} = \begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\left(\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\right)^{-1}\boldsymbol{y}.$$

We are only interested in solving for $\boldsymbol{w}$, so we disregard the n rows corresponding to the false parameter $\boldsymbol{f}$. In the end, we are left with the minimum norm solution to the under-determined system specified above,

$$\boldsymbol{w} = \boldsymbol{X}^{\mathsf{T}}\left(\begin{bmatrix}\boldsymbol{X} & \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\begin{bmatrix}\boldsymbol{X}^{\mathsf{T}}\\ \sqrt{\lambda}\boldsymbol{I}_n\end{bmatrix}\right)^{-1}\boldsymbol{y} = \boldsymbol{X}^{\mathsf{T}}\left(\boldsymbol{X}\boldsymbol{X}^{\mathsf{T}} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{y}.$$

1.3 Using Singular Value Decomposition to Simplify Regularization

Using the singular value decomposition (SVD) in place of $\boldsymbol{X}$ in the above equations allows us to simplify things and make the algorithms far easier to handle. Namely, we are able to update each weight individually instead of solving systems of equations. This can be seen by taking the SVD of $\boldsymbol{X}$ and solving the unregularized problem

$$\begin{aligned}\boldsymbol{X}\boldsymbol{w} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathsf{T}}\boldsymbol{w} &\approx \boldsymbol{y}\\ \Rightarrow\ \boldsymbol{\Sigma}\widetilde{\boldsymbol{w}} &\approx \widetilde{\boldsymbol{y}},\end{aligned}$$

where $\widetilde{\boldsymbol{w}} := \boldsymbol{V}^{\mathsf{T}}\boldsymbol{w}$, $\widetilde{\boldsymbol{y}} := \boldsymbol{U}^{\mathsf{T}}\boldsymbol{y}$, and $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times d}$. Note that $\boldsymbol{U}$, $\boldsymbol{V}$ are orthogonal matrices so they merely rotate our vectors while preserving their norms. Let σ_i denote the i^{th} singular value. Because $\boldsymbol{\Sigma}$ is diagonal and we are assuming that $n > d$, only the first d equations will be meaningful here:

$$\begin{aligned}\text{for } d \text{ equations: } \sigma_i\widetilde{\boldsymbol{w}}[i] \approx \widetilde{\boldsymbol{y}}[i] \ \Rightarrow\ \widetilde{\boldsymbol{w}}[i] &\approx \frac{1}{\sigma_i}\widetilde{\boldsymbol{y}}[i],\\ \text{for } n-d \text{ equations: } 0 &\approx \widetilde{\boldsymbol{y}}[i].\end{aligned}$$

As previously stated, using these coordinates removes the system of equations and allows us to simply solve for the individual weight components. Observe that we are dividing by the singular values so problems may arise if they become too small (i.e. if the matrix is ill-conditioned). This is the underlying cause of OLS sometimes giving us wild values.

Next, we consider using ridge regression. In this setting, the solution is obtained by plugging the SVD of $\boldsymbol{X}$

into the previously mentioned classic solution for ridge regression.

$$\begin{aligned}\hat{\boldsymbol{w}} &= \left(\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\boldsymbol{V}^\mathsf{T} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{U}^\mathsf{T}\boldsymbol{y} \\ &= \boldsymbol{V}\left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{V}^\mathsf{T}\boldsymbol{V}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}} \\ &= \boldsymbol{V}\left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}} \\ \Rightarrow \widehat{\widetilde{\boldsymbol{w}}} &= \left(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Sigma}^\mathsf{T}\widetilde{\boldsymbol{y}}.\end{aligned}$$

In the end, we are left with solutions of the form

$$\widetilde{\boldsymbol{w}}[i] = \frac{\sigma_i}{\sigma_i^2 + \lambda}\widetilde{\boldsymbol{y}}[i].$$

If $\lambda << \sigma_i^2$, then we are in the same situation as the unregularized case. If $\lambda >> \sigma_i^2$, then the weights are forced to stay small instead of behaving wildly.

1.4 Implicit Regularization

Implicit regularization is the regularization that occurs when we aren't consciously doing any regularization. When performing explicit regularization as above, we must specify a specific regularization hyperparameter and modify the training data, model architecture, or loss function. In contrast, implicit regularization is an unexpected benefit stemming from our choice of optimizer. We will see that choosing gradient descent as our optimization algorithm, combined with the large size of DNNs, provides enough regularization for DNNs to generalize well without us intentionally restricting the parameters.

To gain intuition, let's look at gradient descent (GD) updates for OLS in SVD coordinates:

$$\widetilde{\boldsymbol{w}}_{t+1} = \widetilde{\boldsymbol{w}}_t + 2\eta\boldsymbol{\Sigma}^\mathsf{T}\left(\widetilde{\boldsymbol{y}} - \boldsymbol{\Sigma}\widetilde{\boldsymbol{w}}_t\right),$$

where $\widetilde{\boldsymbol{w}}_t$ represents the parameter vector at the current step, $\widetilde{\boldsymbol{w}}_{t+1}$ represents the updated parameter vector, $\boldsymbol{\Sigma}$ represents the diagonal matrix of singular values from the above SVD, $\widetilde{\boldsymbol{y}} := \boldsymbol{U}^\mathsf{T}\boldsymbol{y}$ as defined above, and η is our learning rate hyperparameter.

Note that because we are dealing with a diagonal matrix, this works out to updating each component of the weight vector individually. They don't interact with each other at all during GD, so each is being modified as follows:

$$\widetilde{\boldsymbol{w}}_{t+1}[i] = \widetilde{\boldsymbol{w}}_t[i] + 2\eta\sigma_i\left(\widetilde{\boldsymbol{y}}[i] - \sigma_i\widetilde{\boldsymbol{w}}_t[i]\right).$$

This is potentially unstable because we are not reducing $\widetilde{\boldsymbol{w}}_t[i]$ at each step as we did in the last lecture. This means that, subject to a bounded input, we might get an unbounded output if we allow the algorithm to run forever. Observe that the stationary point is the solution we discussed earlier, $\widetilde{\boldsymbol{w}}[i] = \frac{1}{\sigma_i}\widetilde{\boldsymbol{y}}[i]$. If σ_i is tiny, then we find ourselves in a bad situation.

Let's carefully calculate the first few steps of GD to see what's going on:

$$\begin{aligned}&\widetilde{\boldsymbol{w}}_0[i] = 0 \\ &\widetilde{\boldsymbol{w}}_1[i] = 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i] \\ &\widetilde{\boldsymbol{w}}_2[i] = 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i] + 2\eta\sigma_i\left(\widetilde{\boldsymbol{y}}[i] - \sigma_i 2\eta\sigma_i\widetilde{\boldsymbol{y}}[i]\right) \approx 4\eta\sigma_i\widetilde{\boldsymbol{y}}[i].\end{aligned}$$

Observe that this is roughly a linear function with an extremely small slope if σ_i is tiny. In this situation, GD barely moves in the early stages even though it will eventually converge to a very large value, as previously discussed. Together with early stopping, this means that GD is trying to do something like ridge regularization for us because it will resist enlarging the directions corresponding to small singular values. Early stopping is when we stop the training process because validation performance has gotten worse or has not improved for a long time. It is important to note that GD, when initialized at zero, will converge to the minimum-norm solution. This is a good exercise for the reader to verify.

To summarize, there are three kinds of regularization: explicit regularization, data augmentation (adding fake observations or features), and implicit regularization (optimizer has implicit regularizing effect). Regarding DNNs, the combination of the min-norm seeking behavior of gradient descent and the feature augmentation that is implicit when using large networks gives a lot of regularization even if we weren't thinking about it.

2 Trade-offs Between Qualitatively Different Sources of Error

Suppose we have learned a model $\hat{\theta} \to f_{\hat{\theta}}$. At "test time", we look at the error, $\left(Y_{observed} - f_{\hat{\theta}}(x)\right)$. There are three main sources of error:

1. *Irreducible error*: This is due to noise or randomness from $Y|X$ itself. In short, there is some level of noise that is impossible for our model to account for. One possible situation is that the underlying response is a deterministic function of X (i.e. not random), but there is randomness in the measurement. It is possible that our model perfectly predicts the underlying signal, but the model will still disagree with $Y_{observed}$ and contribute to the error. In the classic setup of $y = f_{true}(x) + \boldsymbol{\varepsilon}_{noise}$, the $\boldsymbol{\varepsilon}_{noise}$ term contributes to irreducible error.
2. *Approximation error*: This error comes from limited expressive power of f_θ as a finitely-parameterized model. In other words, our model isn't "flexible" enough to capture the true signal. For example, trying to fit the function $y = \cos(x)$ using a single 6th degree polynomial model, $f_\theta(x) = \sum_{i=0}^{6} a_i x^i$.
3. *Estimation error*: There are two components to this type of error, both of which are well-covered in prerequisite machine learning courses:
   (a) *Bias*: Bias captures the systematic error of our learning algorithm and training data in terms of making predictions. We write this mathematically by $\mathbb{E}_{\boldsymbol{\varepsilon},\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(X) - Y \ |X]$. Note that $\boldsymbol{\varepsilon}$ respresents the randomness in $Y|X$ and $\mathcal{D}$ represents the randomness in the training process used to get $\hat{\theta}$. For example, this could include the training data set we used or the splits made in random forest trees. Traditionally, X is separate from $\mathcal{D}$ and is not random. It is common to look at the squared bias to prevent positive and negative biases for different observations from canceling.
   (b) *Variance*: This is the variable part of error. We write it as $\mathbb{E}_{\mathcal{D}}[(f_{\hat{\theta}(\mathcal{D})}(X) - \mathbb{E}_{\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(X)])^2|X]$, where $\mathcal{D}$ again represents the randomness of the training process. It describes how much our prediction "shakes" as a function of the randomness in the training.

This perspective can be useful and many papers utilize the bias-variance decomposition in their derivations. However, it doesn't always match what our intuition might be, especially in deep learning settings. The following figure is intended to help us understand this point.

This drawing is for high-level intuition, so we musn't allow ourselves to become confused over the exact dimensions and projections. We first turn our attention to the figure on the left. The line passing through

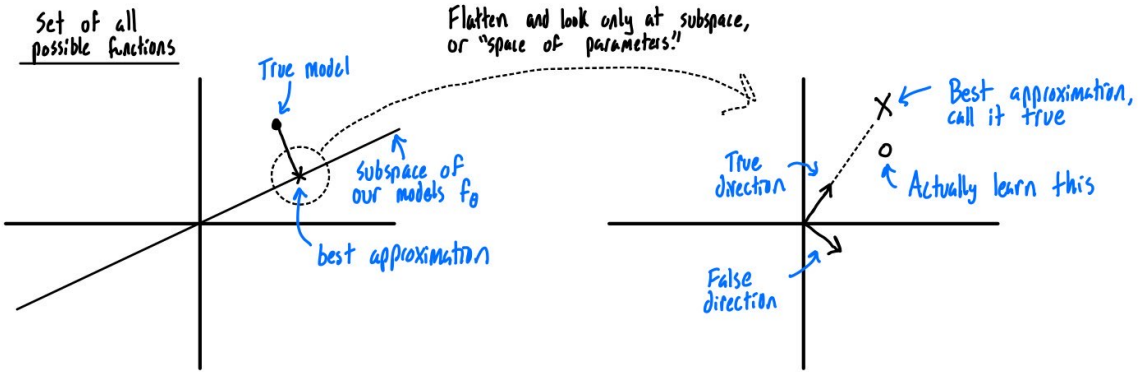

Figure 1: Approximation and Estimation Error

the origin represents the subspace models that we are considering. Note that we are only capable of providing estimates that fall along this line. As we can see, the true model is not in this subspace so we will suffer a certain amount of approximation error.

Turning our attention to the figure on the right, we project the true model onto the subspace of models that we are considering. The result is the best possible approximation and is the goal of our learning algorithm. We see in the figure that our predicted model is not quite equal to the best approximation. In this perspective, instead of thinking of things in terms of bias and variance, we consider how much of the "true" and "false" directions we are incorporating into our prediction.

We make this a bit more formal by discussing *survival* and *contamination*. Survival reflects, in expectation, how much of the true pattern survives the estimation/learning process. Contamination is how much useless information get "learned", e.g. spurious features that our model is capable of picking up but don't help with prediction. Survival and contamination can be thought of as the intuition behind bias and variance, respectively.

3 What are "Features"?

The following is a simplified sketch of a neural network with ℓ hidden layers.

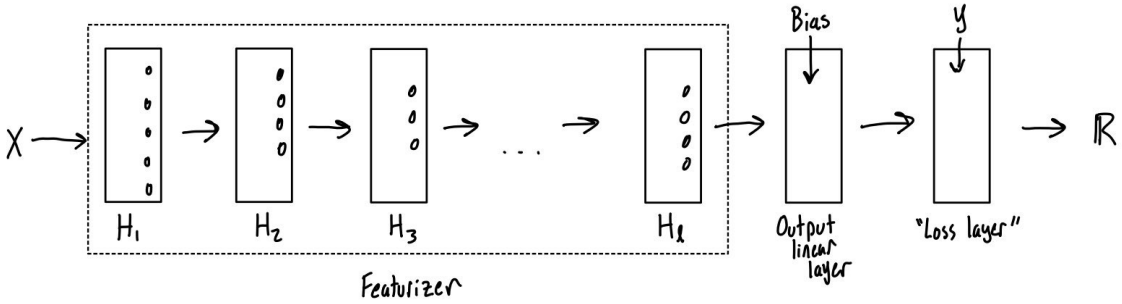

Figure 2: Simplified DNN

If we treat everything before the output of H_ℓ as a black box, we can think of things from the perspective of a generalized linear model. We have a featurizer, some linear function of those new features, and a loss that we are optimizing. The featurizer lifts or distills the input X into a nicer feature space. In this perspective, the "learned" features are the outputs of the penultimate layer in the featurizer, H_ℓ. We want the featurization to be data-driven instead of hand-picked, so the layers essentially find a representation of the data that allows the generalized linear model (GLM) to work well.

However, there is another important point of view. Suppose a generalized linear model is given by $\hat{y} = \sum_{i=1}^{\lambda} w_i \phi_i(\boldsymbol{x})$, where $\boldsymbol{\phi(x)}$ is the output of the "featurizer" in the figure above. When we are actually using the model on new data, these are simply the features. However, from the training perspective, they also determine the gradient. The derivative of the linear model with respect to the i^{th} parameter is $\frac{\partial \hat{y}}{\partial w_i} = \phi_i(\boldsymbol{x})$.

In short, we don't understand nonlinear systems well, so our standard approach to understanding a nonlinear system is local linearization. We zoom in until things are roughly linear around a certain point. In terms of DNNs, we say that the deep network is Taylor expanded around the features in such a way that it is some constant term plus a local GLM in which small increments of the features change the predictions in a small way. This will be covered in greater detail during the next lecture.