CS282 Lecture 5: Survey of Architectures and Problems

Lecturer: Anant Sahai, Scribes: Gabrielle Hoyer, Kevin Tsai

September 2022

1. Last Lecture Recap

In Lecture 4, we talk about using regularization to prevent our dependence on direction in the SVD space (direction with small singular values) that we don't trust. Various methods of regularization are mentioned and discussed, in particular we analyze:

1.1 Explicitly Adding Terms to the Cost Function

This corresponds to transforming the optimization problem to:

$$\min_{\boldsymbol{\theta}} \quad (l_{\text{train}}(\boldsymbol{\theta}) + R(\boldsymbol{\theta}))$$

where $R(\boldsymbol{\theta})$ is a regularization term, e.g., for l2 regularization this could be:

$$R(\boldsymbol{\theta}) = \lambda ||\boldsymbol{\theta}||_2^2$$

Under l2 regularization and least squares loss function, the transformed problem is the classical ridge regression with solution $\boldsymbol{\theta}^*$ where X is the data matrix and $\mathbf{y}$ is the target vector.

$$\boldsymbol{\theta}^* = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

1.2. Weight Decay (Explicit in the Algorithm)

With ridge regression, the penalty term $\lambda ||\boldsymbol{\theta}||_2^2$ inside the cost function has a gradient that is proportional to the weight ($\boldsymbol{\theta}$) itself. When performing gradient descent, this amounts to scaling down the norm of the weight (provided that the learning rate is sufficiently small), i.e., $\boldsymbol{\theta}_{t+1} = (1 - 2\lambda\zeta)\boldsymbol{\theta}_t +$ *usual terms for gradient descent*, where ζ is the learning rate (this is like pulling the weights down). This form of regularization is listed separately from the first one even though it might appear to be equivalent; this is because for applications other than ridge regression (with gradient descent), sometimes a general optimizer (or algorithm) might not work well with the l2-penalized cost function, in which case, we can still perform the weight decay regardless.

1.3. Data Augmentation

It is also possible to mimic regularization by inserting artificial observations into the data matrix, i.e.

$$\begin{bmatrix} X \\ \sqrt{\lambda} I_d \end{bmatrix} \boldsymbol{\theta} \approx \begin{bmatrix} \mathbf{y} \\ \mathbf{0}_d \end{bmatrix}$$

This is important for neural networks, e.g., adding flipped/rotated images into the training data for a convolutional neural network.

1.4. Feature Augmentation

Instead of extending the data matrix row-wise, we can also extend it column-wise, e.g., by appending artificial features to each existing observation.

$$\begin{bmatrix} X & \sqrt{\lambda} I_d \end{bmatrix} \begin{bmatrix} \boldsymbol{\theta} \\ \boldsymbol{\theta}' \end{bmatrix} \approx \begin{bmatrix} \mathbf{y} \end{bmatrix}$$

This is also important for neural networks; we will see in section 2 that we have just as many weights (parameters) as we have features in a neural net, and these many features can have the desired regularization effect.

1.5. Implicit Regularization during Optimization

Optimization algorithms such as gradient descent implicitly implement regularization, even when the objective function does not include the regularization penalty. Features (in the SVD space) are more favored if they correspond with large singular values (i.e., in each iteration, they take bigger steps), whereas those with small singular values move hardly at all. For those "good" features with large singular values, we hope that they are likely to correspond to true patterns. On the contrary, those with small singular values, we hope they are spurious signals.

In addition to the regularization effect introduced by how the optimization algorithm responds to different singular values, early stopping is another implicit mechanism we have for most iterative algorithms; this effectively prevents the weights from becoming arbitrarily large.

1.6. Bias-variance Trade-off

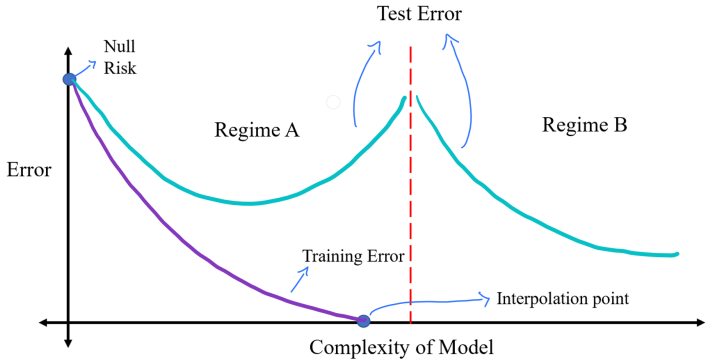

Figure 1: Training Error and Test Error vs Model Complexity

A common plot in traditional machine learning is the evolution of training error and test error as the complexity of a model increases. The complexity of a model can be the number of parameters estimated, the degrees of polynomials used, or the number of hidden layers/units in a neural network, etc. Figure 1 shows a simple demonstration of such a plot: When the model has zero complexity (e.g., an intercept model) the training error is equal to the test error and their specific values are called the null risk (note that they need not be identical). As the model becomes more complicated, the training error usually decreases faster than the test error, and at some point, the test error ceases to decrease and begins to increase (what people usually call overfitting) whereas the training error approaches zero

eventually (which could potentially be concerning if there is non-reducible error in the data generating process and a zero error implies the model is fitting the noise). The reason why the test error begins to increase can be attributed to the increase in estimation error with the increased complexity and the decrease in approximation error (from the more complex model) can not compensate for that.

It is possible, however, that the test error in the overfit regime can again start to decrease when the model becomes even more complicated. This is apparently what practitioners of deep neural networks observe and most deep neural networks that work reasonably well are thought to be in this regime (Regime B in Figure 1). One would be concerned if a traditional machine learning model ends up with zero training error (whatever it means in the context) as it indicates overfitting, but when it comes to deep neural networks, people would actually only start to consider the model if it can attain zero training error because they believe a well-designed deep neural net should be operating in Regime B. The 4th (feature augmentation) and 5th (implicit regularization) regularization methods mentioned above may be at work to contribute to the reduction in test error in this regime (a potential decrease in approximation error is also a possible cause of reduction). It is likely that these two regularization methods help control the estimation error when deep neural networks perform well.

Although we know the 4th and 5th regularization methods are important for neural networks, for practical reasons, it is often the case that practitioners would try all 5 methods of regularization mentioned above. One of the reasons why someone might want to do that is because these methods do not work independently from each other; e.g., if we want to take advantage of implicit regularization and we want the feature direction with large singular values to match what we believe to be true, we probably need other regularization techniques to encourage the behavior.

Last, despite the fact that our discussion centers around training a neural network, the behavior in Regime B (small test error with a really complex model) can also be observed in other traditional machine learning models, e.g., tree-based models and kernel methods. One of the first realizations of such behavior is said to have been discovered with boosted trees.

2. Features

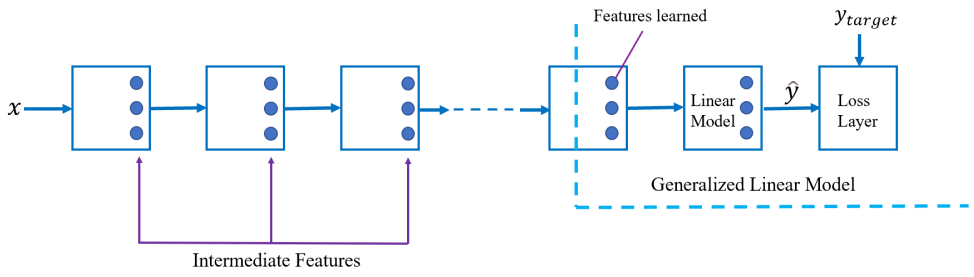

Figure 2: Generalized Linear Model with intermediate features

2.1 Neural Net Represented as a Generalized Linear Model

When thinking about deep neural networks, it is useful to consider the optimization algorithm from a local linear perspective. The below equation states an interesting relationship between $\hat{y}$, our model, x, the training data, and θ, the learned weights. Specifically, the equation describes that $\hat{y}(x)$ for given current parameters relative to the parameters to change, is equal to $\hat{y}$ for x relative to current parameters and outcomes, plus the partial derivative of $\hat{y}$ relative to all parameters, evaluated at θ_0, times the desired

hypothetical parameter change.

$$\hat{y}(x) = \hat{y}(x, \theta_0) + \left.\frac{d\hat{y}}{d\theta}\right|_{\theta_0} \cdot \Delta\vec{\theta}$$

From this perspective, one can see that despite the many possible layers within a neural net, the algorithm is working on a generalized linear model (though differently centered) with a feature corresponding to every parameter. The $\Delta\vec{\theta}$ vector is the size of the number of parameters that can be learned; therefore, $\frac{d\hat{y}}{d\theta}$ is this same size. In this way, the generalized linear model can be described by the following equation in which our model, $\vec{\theta}$ is acting on some featurization vector, $\vec{\phi}(x)$.

$$y = \vec{\theta}^T \vec{\phi}(x)$$

Specifically, this construes the lifting of x to a set of features, which are functions of x, to return scalars. This is turn is multiplied by the weights, and the output is this linear combination. While we do not fully understand the nuances of deep neural networks, we understand linear models very well, thus the simplification of our optimization problem by our algorithm in a local perspective is quite useful.

When thinking about our deep neural network, it is important to understand that the Gradient Descent direction is not determined from the features of the penultimate layer,"features learned" in Figure 2, but rather the the derivative feature matrix $\frac{d\hat{y}}{d\theta}$ described in our equation. Indeed, it is this feature matrix that helps determine the singular value, big or tiny σ_i, in Gradient Descent, thereby determining our next trusted direction to move.

2.2 Discussion Example

Below is an example from our latest discussion. Figure 3 displays a neural network with affine layers, ReLU non-linearity, and a fully-connected layer. This neural network has a single hidden layer, a scalar input and scalar output. The neural network has a width of k. The parameters of this network can be viewed in two different ways:

1) $3k + 1$ total parameters, stemming from kW_1, kb_1, kW_2, kb_2

2) $k + 1$ parameters from the generalized linear model view.

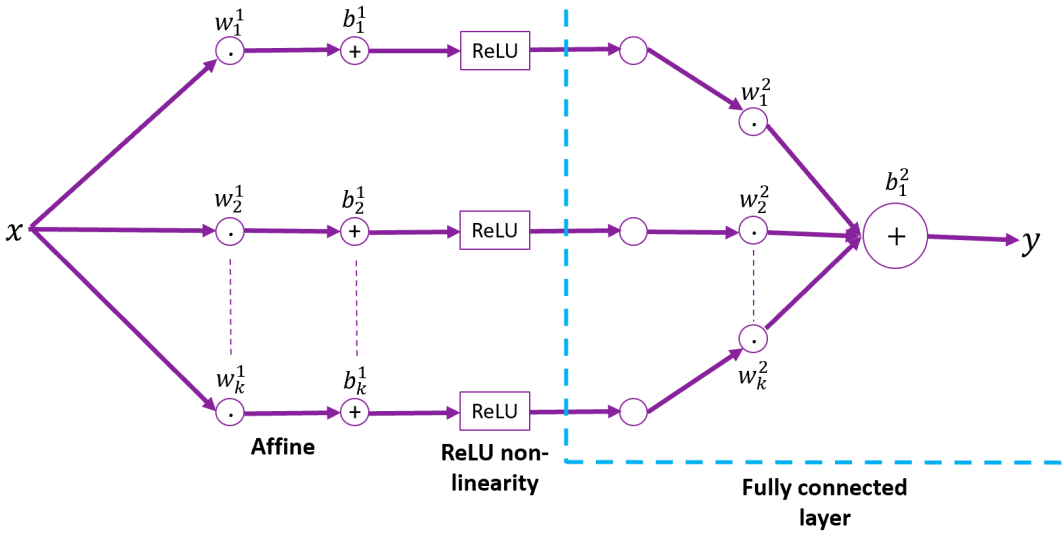

Figure 3: ReLU Neural Network

Consider the following equations. The partial derivative of our network output y in respect to our weights, W, and biases, b, elucidate the dependence of earlier layer features on the features of the later layers. This is interesting, but also introduces the necessity of decoupling these features, a topic of normalization which will be discussed at a later time.

$$\frac{dy}{db_j^2}(x) = 1$$

$$\frac{dy}{dW_j^2}(x) = max(0, W_j^1 x + b_j^1)$$

$$\frac{dy}{db_j^1}(x) = \begin{cases} 0 & ReLU - off \\ W_j^2 & ReLU - on \end{cases}$$

$$\frac{dy}{dW_j^1}(x) = \begin{cases} 0 & ReLU - off \\ W_j^2 x & ReLU - on \end{cases}$$

The above example corresponds to a neural network with a single output. In the case of a network with multiple outputs, there will be features which correspond to each output. Furthermore, in the case of the general linear model each output of the penultimate layer is assigned its own weight. However, in the case of a deep neural network such as for multi-classification, there will be features corresponding to each output with weight sharing.

In the case of a very large linear model with many features, each feature performs a small part of the work of fitting the residual; the amount each corresponding weight moves is quite small. Interestingly, there is a hypothesis in the field to consider which states that for a very large model, a sufficient distance from initialization is never reached to change the features (or derivatives of the features) in a notable way, nor is it truly necessary. This is a perspective that deep neural nets do indeed behave quite like a general linear model.

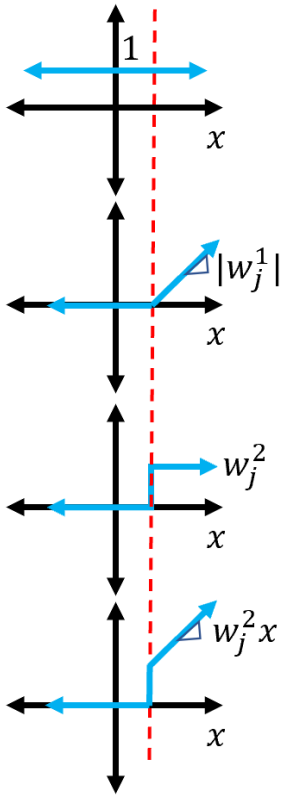

Figure 4: Gradient plots, First: $\frac{dy}{db_j^2}$ Second: $\frac{dy}{dW_j^2}$ Third: $\frac{dy}{db_j^1}$ Fourth: $\frac{dy}{dW_j^1}$

3. Survey

There are a variety of Neural Network architectures and specific applications in which they can be used. Technical application domains and the approaches in which you engage with them will be further explored in detail at a future time. Our multi-dimensional architecture-application grid can be seen below:

Neural Net Architectures (families)					
Area of Use	Multilayer Perceptron	Convolutional NN	Recurrent NN	Graph NN	Transformers
Vision Natural Lang. Processing Time Series Recommendation Engines Scientific Control					