DATA 586: Advanced Machine Learning

2023W2

Shan Du

When to Use Deep Learning

- We revisit our *Hitters* dataset where the goal is to predict the *Salary* of a baseball player in 1987 using his performance statistics from 1986.
- After removing players with missing responses, we are left with 263 players and 19 variables. We randomly split the data into training set of 176 players (two thirds), and a test set of 87 players (one third).

When to Use Deep Learning

- We used three methods for fitting a regression model to these data:
  - A linear model was used to fit the training data, and make predictions on the test data. The model has 20 parameters.
  - The same linear model was fit with lasso regularization. The tuning parameter was selected by 10-fold cross-validation on the training data. It selected a model with 12 variables having nonzero coefficients.
  - A neural network with one hidden layer consisting of 64 ReLU units was fit to the data. This model has 1,345 parameters.

When to Use Deep Learning

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1345	257.4	0.54

TABLE 10.2. *Prediction results on the* `Hitters` *test data for linear models fit by ordinary least squares and lasso, compared to a neural network fit by stochastic gradient descent with dropout regularization.*

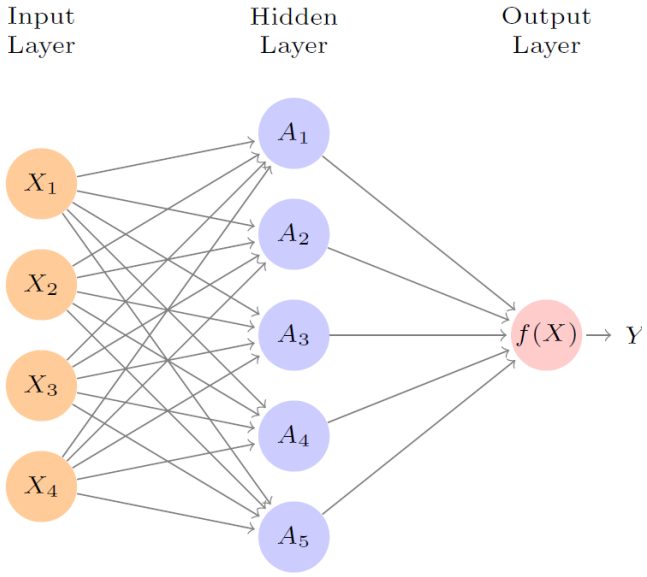
- Linear models are much easier to present and understand than the neural network.
- *Occam's razor* principle: when faced with several methods that give roughly equivalent performance, pick the simplest.

When to Use Deep Learning

- Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.

Fitting a Neural Network

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$



- All the parameters $\beta = (\beta_0, \beta_1, \ldots, \beta_K)$ and $w_k = (w_{k0}, w_{k1}, \ldots, w_{kp}), k = 1, \ldots, K$ need to be estimated from data.

Fitting a Neural Network

- Given observations $(x_i, y_i), i = 1, \ldots, n$, we could fit the model by solving a nonlinear least squares problem

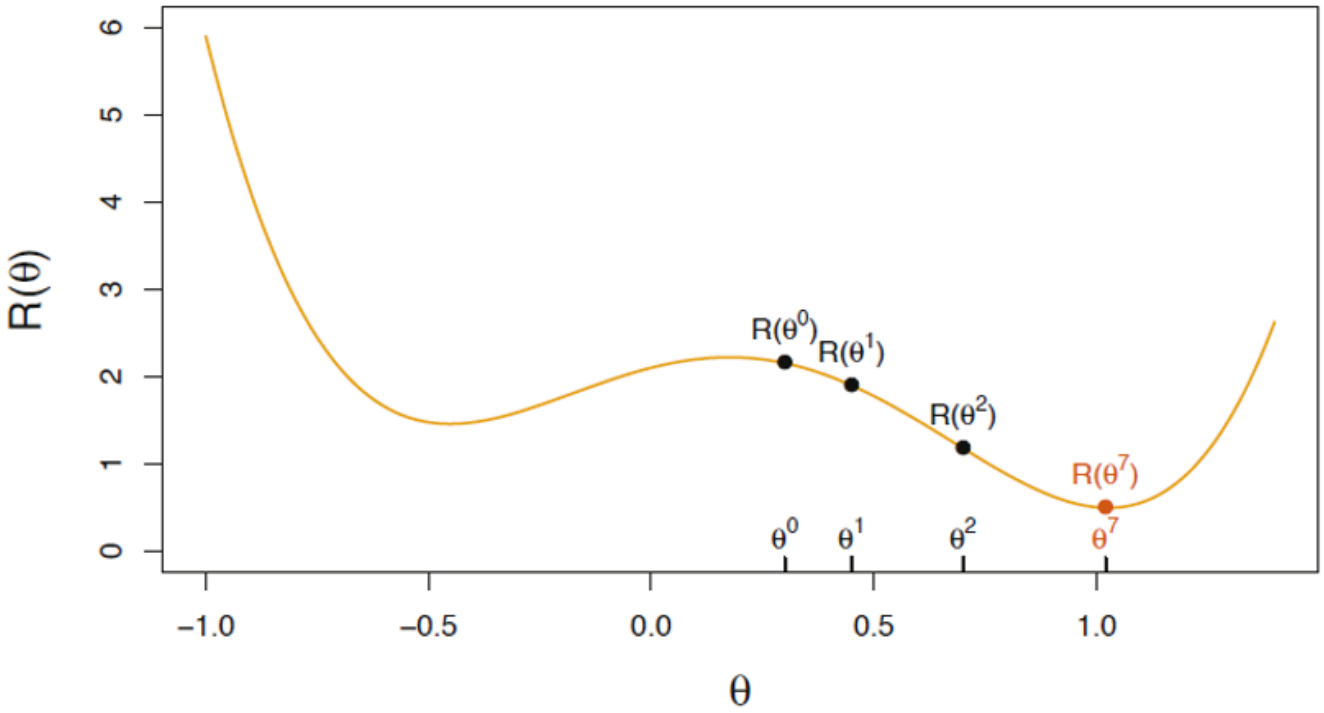
$$\underset{\{w_k\}_1^K, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij})$$

Fitting a Neural Network

- The problem is nonconvex in the parameters, and hence there are multiple solutions.



Fitting a Neural Network

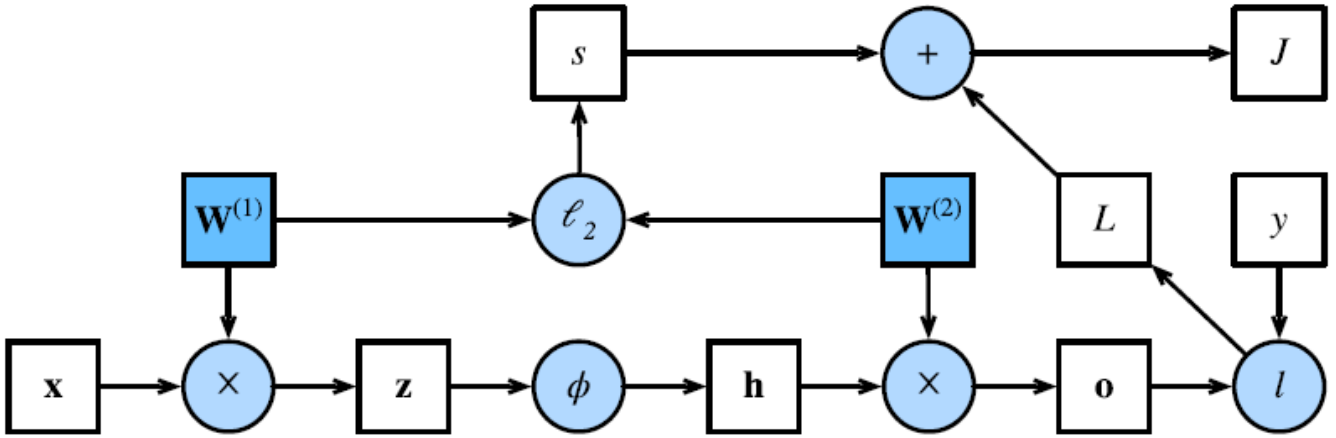
- There are two solutions: one is a *local minimum* and the other is a *global minimum*.
- Two general strategies are employed when fitting neural networks:
  - Slow Learning: the model is fit in a somewhat slow iterative fashion, using *gradient descent*. The fitting process is then stopped when overfitting is detected.
  - Regularization: penalties are imposed on the parameters, usually lasso or ridge.

Forward Propagation

- Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Computational Graph

- Plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation.
- Squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows are primarily rightward and upward.



Computational graph of forward propagation.

Gradient Descent

- With gradient descent, when we cannot solve the optimization analytically, we can still often train models effectively in practice.
- Gradient descent, the key technique for optimizing nearly any deep learning model, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function.

Gradient Descent

- Suppose we represent all the parameters in one long vector θ. Then we can rewrite the objective as

$$R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$$

where we make explicit the dependence of f on the parameters.

Gradient Descent

- The idea of gradient descent is very simple.
  1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
  2. Iterate until the objective fails to decrease:

     (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e., such that $\mathrm{R}(\theta^{t+1}) < \mathrm{R}(\theta^t)$.

     (b) Set $t \leftarrow t + 1$.

Gradient Descent

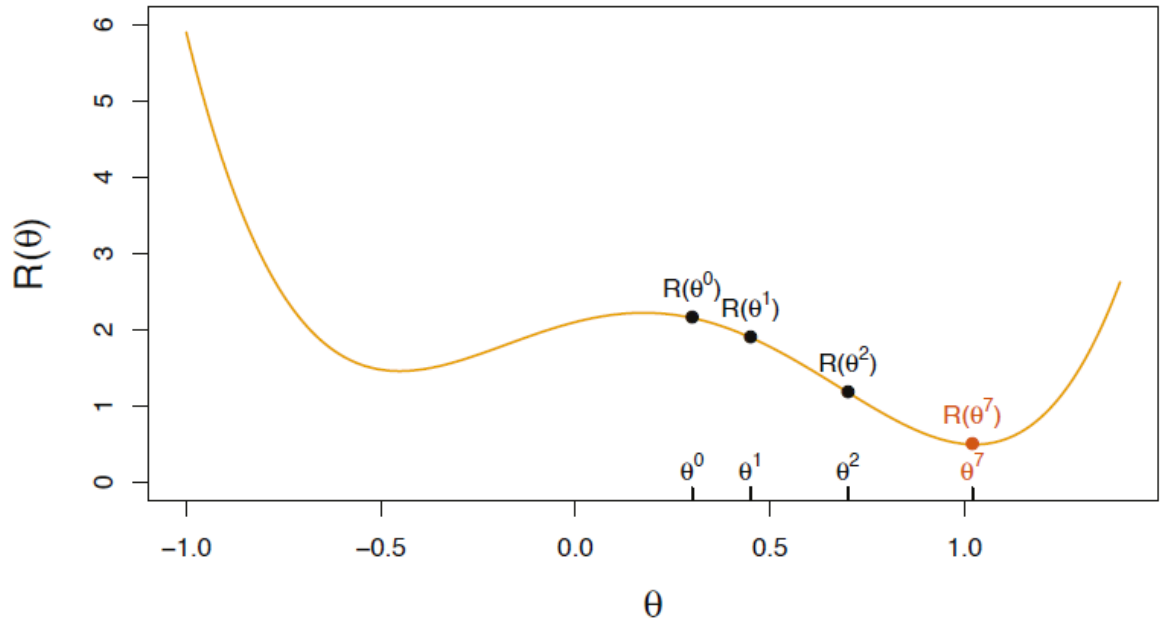


FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.*

Gradient Descent

- One can visualize standing in a mountainous terrain, and the goal is to get to the bottom through a series of steps.
- As long as each step goes downhill, we must eventually get to the bottom.
- In this case we were lucky, because with our starting guess θ^0 we end up at the global minimum. In general, we can hope to end up at a (good) local minimum.

Backpropagation

- How do we find the directions to move θ so as to decrease the objective $R(\theta)$?
- The *gradient* of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point:

$$\nabla R(\theta^m) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta^m}.$$

Backpropagation

- The subscript $\theta = \theta^m$ means that after computing the vector of derivatives, we evaluate it at the current guess, θ^m.
- This gives the direction in θ-space in which $R(\theta)$ increases most rapidly. The idea of gradient descent is to move θ a little in the opposite direction (since we wish to go downhill):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m).$$

Backpropagation

- For a small enough value of the *learning rate* ρ, this step will decrease the objective $R(\theta)$; i.e., $R(\theta^{m+1}) \leq R(\theta^m)$. If the gradient vector is zero, then we may have arrived at a minimum of the objective.
- Since $R(\theta) = \sum_{i=1}^{n} R_i(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$ is a sum, its gradient is also a sum over the n observations, so we will just examine one of these terms,

Backpropagation

$$R_i(\theta) = \frac{1}{2}(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}))^2$$

- To simplify the expressions to follow, we write $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$. We first take the derivative with respect to β_k:

$$\frac{\partial R_i(\theta)}{\partial \beta_k} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} = -(y_i - f_\theta(x_i)) \cdot g(z_{ik})$$

Backpropagation

- And now we take the derivative with respect to

$$\frac{\partial R_i(\theta)}{\partial w_{kj}} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}}$$
$$= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}.$$

- Notice that both these expressions contain the residual $y_i - f_\theta(x_i)$. we see that a fraction of that residual gets attributed to each of the hidden units according to the value of $g(z_{ik})$ we see a similar attribution to input j via hidden unit k. So the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule — a process known as *backpropagation* in the neural network literature.

Backpropagation

- How to use Gradient Decent to minimize error:
  - https://www.python-unleashed.com/post/derivation-of-the-binary-cross-entropy-loss-gradient
  - https://www.youtube.com/watch?v=z_xiwjEdAC4

Minibatch Stochastic Gradient Descent

- The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset.
- In practice, this can be extremely slow: we must pass over the entire dataset before making a single update, even if the update steps might be very powerful.
- Even worse, if there is a lot of redundancy in the training data, the benefit of a full update is even lower.

Minibatch Stochastic Gradient Descent

- The other extreme is to consider only a single example at a time and to take update steps based on one observation at a time.
- The resulting algorithm, stochastic gradient descent (SGD) can be an effective strategy, even for large datasets.
- Unfortunately, SGD has drawbacks, both computational and statistical.

Minibatch Stochastic Gradient Descent

- One problem arises from the fact that processors are a lot faster multiplying and adding numbers than they are at moving data from main memory to processor cache. It is up to an order of magnitude more efficient to perform a matrix vector multiplication than a corresponding number of vector-vector operations.
- This means that it can take a lot longer to process one sample at a time compared to a full batch.
- A second problem is that some of the layers, such as batch normalization, only work well when we have access to more than one observation at a time.

Minibatch Stochastic Gradient Descent

- The solution to both problems is to pick an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a *minibatch* of observations.
- The specific choice of the size of the said minibatch depends on many factors, such as the amount of memory, the number of accelerators, the choice of layers, and the total dataset size. Despite all of that, a number between 32 and 256, preferably a multiple of a large power of 2, is a good start. This leads us to *minibatch stochastic gradient descent* which is the state of the art for learning deep neural networks.

Minibatch Stochastic Gradient Descent

- In its most basic form, in each iteration t, we first randomly sample a minibatch B_t consisting of a fixed number $|B|$ of training examples.
- We then compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters.
- Finally, we multiply the gradient by a predetermined small positive value η, called the *learning rate*, and subtract the resulting term from the current parameter values.

Minibatch Stochastic Gradient Descent

- We can express the update as follows:

$$(w, b) \leftarrow (w, b) - \frac{\eta}{|B|} \sum_{i \in B_t} \partial_{(w,b)} l^{(i)}(w, b)$$

- In summary, minibatch SGD proceeds as follows: (i) initialize the values of the model parameters, typically at random; (ii) iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient. For quadratic losses and affine transformations, this has a closed-form expansion:

Minibatch Stochastic Gradient Descent

$$w \leftarrow w - \frac{\eta}{|B|} \sum_{i \in B_t} \partial_w l^{(i)}(w, b)$$

$$= w - \frac{\eta}{|B|} \sum_{i \in B_t} \mathrm{x}^{(i)} (w^T \mathrm{x}^{(i)} + \mathrm{b} - y^{(i)})$$

$$b \leftarrow b - \frac{\eta}{|B|} \sum_{i \in B_t} \partial_b l^{(i)}(w, b)$$

$$= b - \frac{\eta}{|B|} \sum_{i \in B_t} (w^T \mathrm{x}^{(i)} + \mathrm{b} - y^{(i)})$$

Minibatch Stochastic Gradient Descent

- Since we pick a minibatch B we need to normalize by its size $|B|$. Frequently minibatch size and learning rate are user-defined. Such tunable parameters that are not updated in the training loop are called *hyperparameters*.
- They can be tuned automatically by a number of techniques, such as Bayesian optimization.
- In the end, the quality of the solution is typically assessed on a separate validation dataset (or validation set).

Minibatch Stochastic Gradient Descent

- Linear regression happens to be a learning problem with a global minimum (with some conditions). However, the loss surfaces for deep networks contain many saddle points and minima.
- Fortunately, we typically don't care about finding an exact set of parameters but merely any set of parameters that leads to accurate predictions (and thus low loss).
- In practice, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training sets*. The more formidable task is to find parameters that lead to accurate predictions on previously unseen data, a challenge called *generalization*.

Predictions

- Given the trained model, we can now make predictions for a new example.
- Deep learning practitioners have taken to calling the prediction phase *inference* but this is a bit of a misnomer - *inference* refers broadly to any conclusion reached on the basis of evidence, including both the values of the parameters and the likely label for an unseen instance. If anything, in the statistics literature *inference* more often denotes parameter inference and this overloading of terminology creates unnecessary confusion when deep learning practitioners talk to statisticians.

Vectorization for Speed

- When training our models, we typically want to process whole minibatches of examples simultaneously.
- Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

Model Complexity

- In classical theory, when we have simple models and abundant data, the training and generalization errors tend to be close. However, when we work with more complex models and/or fewer examples, we cannot conclude based on fitting the training data alone that our model has discovered any generalizable pattern.

Model Complexity

- When a model is capable of fitting arbitrary labels, low training error does not necessarily imply low generalization error. *However, it does not necessarily imply high generalization error either*!
- All we can say confidently is that low training error alone is not enough to certify low generalization error.
- Deep neural networks turn out to be just such models: while they generalize well in practice, they are too powerful to allow us to conclude much on the basis of training error alone. In these cases we must rely more heavily on our holdout data to certify generalization after the fact. Error on the holdout data, i.e., validation set, is called the *validation error*.

Data Size

- Another big consideration to bear in mind is dataset size.
- Fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting.
- As we increase the amount of training data, the generalization error typically decreases. Moreover, in general, more data never hurts. For a fixed task and data distribution, model complexity should not increase more rapidly than the amount of data.

Data Size

- Given more data, we might attempt to fit a more complex model. Absent sufficient data, simpler models may be more difficult to beat.
- For many tasks, deep learning only outperforms linear models when many thousands of training examples are available.
- In part, the current success of deep learning owes considerably to the abundance of massive datasets arising from Internet companies, cheap storage, connected devices, and the broad digitization of the economy.

Model Selection

- Typically, we select our final model, only after evaluating multiple models that differ in various ways (different architectures, training objectives, selected features, data preprocessing, learning rates, etc.). Choosing among many models is aptly called *model selection*.

Model Selection

- We should never rely on the test data for model selection. And yet we cannot rely solely on the training data for model selection either because we cannot estimate the generalization error on the very data that we use to train the model.
- The common practice to address the problem of *training on the test set* is to split our data three ways, incorporating a *validation set* in addition to the training and test datasets.

Cross-Validation

- When training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set.
- One popular solution to this problem is to employ *K-fold cross-validation*. Here, the original training data is split into K non-overlapping subsets. Then model training and validation are executed K times, each time training on $K-1$ subsets and validating on a different subset (the one not used for training in that round).
- Finally, the training and validation errors are estimated by averaging over the results from the K experiments.