CS7140: Advanced Machine Learning **Spring 2020**

Lecture 8: Learning as Inference

Lecturer: Rose Yu *Scribes: Yushu Wu, Zixuan Liang*

8.1 Learning as Inference

8.1.1 Motivation Example: Logistic Regression

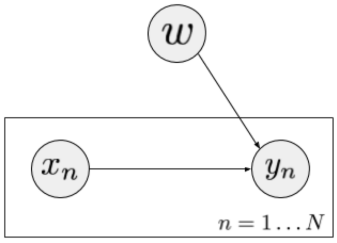

Figure 8.1: Logistic Regression Model

Figure 8.1 presents the *logistic regression* in graphic model language. In the *logistic regression* model, with two variables x, y, has parameter w that governed how y is related to x. In other way, the probability is shown as below.

$$P(y = 1|x, w) = \frac{1}{1 + \exp(-w^\top x)} \tag{8.1}$$

The equation (8.1) is basically the *sigmoid* function applied to linear combination of features. In this simple one-neural example, we have inputs x, a neural network implemented by the weight w and an activate function which is the *sigmoid* function over $w^T x$.
In general, any non-linear function can be chosen as an activate function. A neural network implements a non-linear function $f(x; w)$ with inputs x, parameterized by weights w.

In *logistic regression*, cross entropy (equation (8.2)) is implemented as loss function to estimated the parameters in graphical model. The reason to choose cross entropy is that the distribution assumption for y is Bernoulli distribution. According to Bernoulli distribution, the joint likelihood of the binary classification model is $f(x_n; w)^{y_n}(1-f(x_n; w))^{1-y_n}$. Implementing the log likelihood yields the *cross entropy* loss function.

$$L(w) = -\sum_n \left[y_n \log f(x_n; w) + (1 - y_n) \log (1 - f(x_n; w))\right] \tag{8.2}$$

Figure 8.2 is the neural network implementation of logistic regression. As shown in the figure, there are inputs $x_1 \dots x_N$ and bias which is top node 1. Then there is the parameters weights w. The sum operation with non-linear activation function gives us the output feature. In general, there can be multiple sums corresponding to one layer and multiple layers which makes it a feedforward deep neural network.

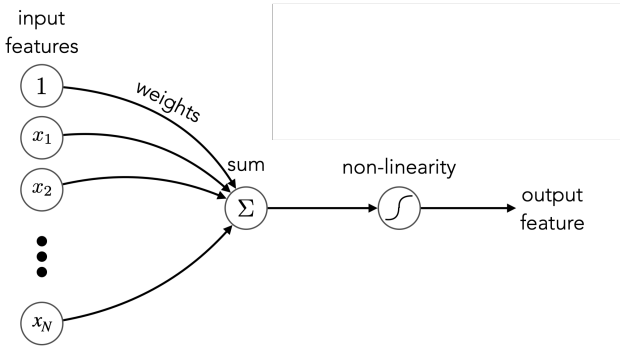

Figure 8.2: Logistic Regression Neural Network

Given the loss function, which in this case is cross entropy, and variables that we want to optimize, the simplest algorithm is *back propagation*(gradient descent).

$$\frac{\partial L}{\partial w_j} = \sum_n -\left(y_n - f\left(x_n; w\right)\right) x_{n,j} \tag{8.3}$$

8.1.2 Probabilistic Interpretation

The neural networks are very flexible and they can over-fit to training data easily, which means the training loss would be very low and the testing loss probably explode.
To avoid this situation, regularization is deployed to the neural network. In this case, two terms regularization is implemented.

$$J(w) = L(w) + \alpha R(w) \tag{8.4}$$

where in Equation (8.4), the $L(w)$ is the original loss function, which is cross entropy, and $R(w)$ is the regularization over the parameter w, α balancing the prior believe versus the observation of data.

$$L(w) = -\sum_n \left[y_n \log f\left(x_n; w\right) + \left(1 - y_n\right) \log\left(1 - f\left(x_n; w\right)\right)\right] \tag{8.5}$$

$$R(w) = \frac{1}{2} \sum_i w_i^2 \tag{8.6}$$

The activation of a single neuron $f(x; w) \equiv P(y = 1|x, w)$ defines the probability input x belongs to positive class.
Regularizer can also be interpreted similarly, as a log prior probability over the parameters $P(w|\alpha) = \frac{1}{Z} \exp\left(-\alpha R(w)\right)$. This regularizer form is a familiar form of probability that mentioned in *Markov Random Field*. *Objective function* $J(w)$ has a log probability of a posterior distribution. In fact, the parameters themselves are random variables.

8.1.3 Simple Example: a neuron with two weights

In the case of a neuron with just two inputs and no bias,

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2)}} \tag{8.7}$$

we can plot the posterior probability of $\mathbf{w}$, $P(\mathbf{w}|D, \alpha) \propto \exp(-M(\mathbf{w}))$. In Figure 8.3, the left column shows the data we received. The second column is the value of $L(w)$, the likelihood of the data. The third column is the posterior, which is the probability over the parameters w. Different rows are the different number of data observed.

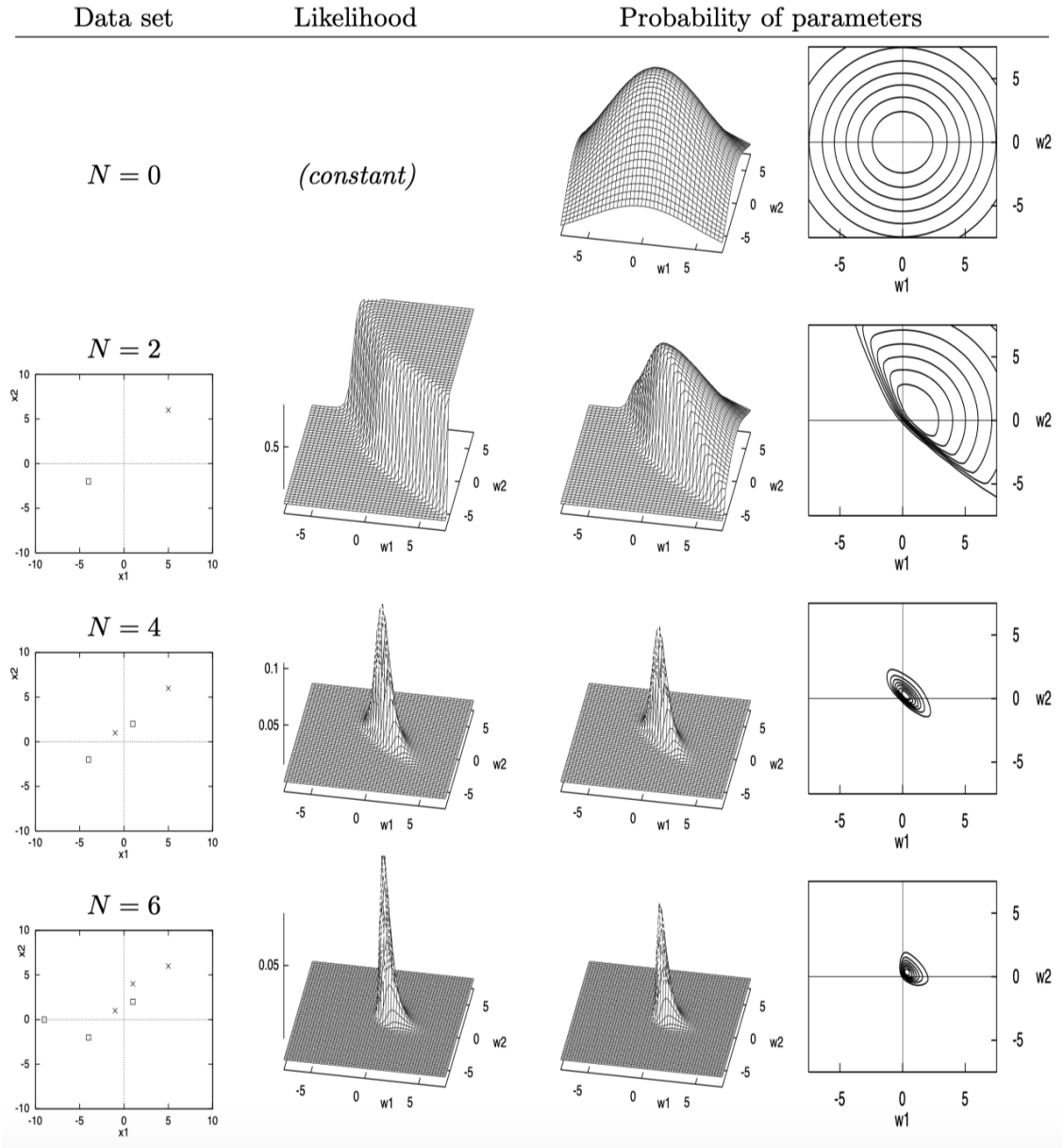

Figure 8.3: The Bayesian interpretation and generalization of traditional neural network learning. Evolution of the probability distribution over parameters as data arrive.

8.1.4 Making predictions

Having the posterior probability of $w, P(w|D,\alpha)$

$$P(w|D,\alpha) = \frac{1}{Z}\exp\left(-J(w)\right) \tag{8.8}$$

where the $Z = \int d^K w \exp\left(-J(w)\right)$. The *Bayesian* prediction of a new datum y_{N+1} involves *marginalizing* over the parameters. Then the predictive probability of a new data y_{N+1} at location x_{N+1} is:

$$P\left(y_{N+1} = 1|x_{N+1}, D, \alpha\right) = \int d^K w f(x;w)\frac{1}{Z}\exp(-J(w)) \tag{8.9}$$

Bayesian inference for general data modelling problems may be implemented by **exact methods** and *approximate inference*. For neural networks *Monte Carlo methods* and *Gaussian approximation methods* are two common approaches.

8.1.4.1 Langevin Monte Carlo

In general *Monte Carlo*, its task of evaluating the integral of a distribution $f(w^{(w)})$ is solved by treating $y(\mathbf{x^{(N+1)}};\mathbf{w})$ as a function f of $\mathbf{w}$ whose mean is

$$\mathbb{E}[f(w)] \approx \frac{1}{R}\sum_r f\left(w^{(r)}\right) \tag{8.10}$$

where $\mathbf{w}^{(r)}$ are samples from the posterior distribution Equation (8.7).

The *Langevin method* may be summarized as *gradient descent with added noise*. A noise vector $\mathbf{P}$ is generated from a Gaussian with unit variance, $p \sim \mathcal{N}(0,1)$. The gradient $\mathbf{g}$ is computed and a step in $\mathbf{w}$ is made, given by

$$\Delta\mathbf{w} = -\frac{1}{2}\epsilon^2\mathbf{g} + \epsilon\mathbf{p} \tag{8.11}$$

Notice that if the $\epsilon\mathbf{p}$ term were omitted this would simply be gradient descent with learning rate $\eta = \frac{1}{2}\epsilon^2$. This step $\mathbf{w}$ is accepted or rejected depending on the change in the value of the objective function $J(w)$. If ϵ is set to too large a value, moves may be rejected. If it is set to a very small value, progress around the state space will be slow.

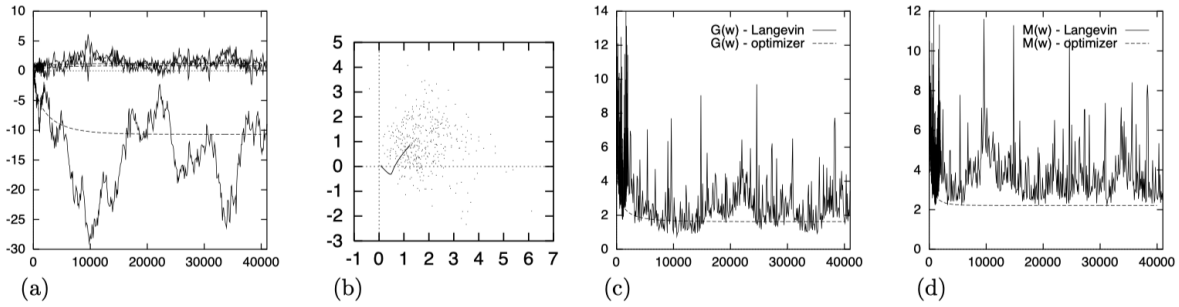

Figure 8.4: A single neuron learning under the Langevin Monte Carlo method. (a)Evolution of weights w_0, w_1 and w_2 as a function of number of iterations. (b) Evolution of weights w_1 and w_2 in weight space.(c) The error function $G(\mathbf{w})$ as a function of number of iterations. (d) The objective function as function $M(\mathbf{x})$ of number of iterations.

8.1.4.2 Gaussian Approximation

Revisiting the Equation (8.9), it is preferable to approximate probability distributions by *Gaussian*. Making a *Gaussian approximation* to the posterior probability, the Taylor expansion is

$$J(w) \approx J(w_0) + \frac{1}{2}(w - w_0)^T \mathbf{A}(w - w_0) + \dots \tag{8.12}$$

where the $A_{ij} \equiv \frac{\partial^2}{\partial w_i w_j} J(w)$ is the *Hessian matrix*.

Here we define our *Gaussian approximation*:

$$Q\left(w; w_0, \mathbf{A}\right) = \left[\det(\mathbf{A}/2\pi)\right]^{1/2} \exp\left[-\frac{1}{2}\left(w - w_0\right)^\top \mathbf{A}\left(w - w_0\right)\right] \tag{8.13}$$

We can think of the matrix $\mathbf{A}$ as defining **error** bars on w. To be precise, Q is a normal distribution whose covariance matrix is $\mathbf{A^{-1}}$. Then the integral can be computed by using Gaussian posterior:

$$P(w|D, \alpha) \approx \left(\frac{1}{Z_Q}\right) \exp\left(-\frac{1}{2}\Delta w^\top A \Delta w\right) \tag{8.14}$$

where $\Delta w = w - w_0$

8.2 Optimization

8.2.1 Stochastic Gradient Descent

This is for large scale dataset. We choose a subset of the dataset which we call "mini-batch", then we compute the gradient of this subset. The update rule is as follow:

$$w = w - \alpha \tilde{\nabla}_w L \tag{8.15}$$

Where w is the weights, α is learning rate, $\tilde{\nabla}_w L$ is the approximated gradient of loss funcion L obtained with the mini-batch data. It is only an approximation because we only use a subset of the data to compute the gradient.

8.2.2 Optimizing Deep Neural Nets

Deep neural nets are non-convex functions. For convex function there is only one local optimum which is also the global optimum. However, for non-convex functions with many local optima, when optimizing the function it would be very likely to be stuck on one local optimum and fail to find the global optimum. SGD is not enough for non-convex functions, such as Rosenbrock function. Applying SGD to this kind of function, it is likely that the optimization progress would be very slow.

8.2.3 Accelerating Stochastic Gradient Descent

8.2.3.1 Momentum

Introducing momentum into SGD. It is a moving average of historical gradients. In other words, it keep track of previous gradients and apply a decay factor to it. In this way the update rule is:

$$v_t = \alpha v_{t-1} - \epsilon \tilde{\nabla}_w L(w_t) \tag{8.16}$$

$$w_{t+1} = w_t + v_t \tag{8.17}$$

8.2.3.2 Nesterov Momentum

It has stronger theoretical guarantee of convergence. The update rule is:

$$v_t = \alpha v_{t-1} - \epsilon \tilde{\nabla}_w L(w_t + \alpha v_{t-1}) \tag{8.18}$$

$$w_{t+1} = w_t + v_t \tag{8.19}$$

Compared to normal momentum method, the only difference is the gradient is not computed at the current position, instead it "foresees" the next position with a "pre-update", and use the pre-udpated position to compute the gradient. In this way, if the alogirithm "sees" that the next gradient would be larger than the current gradient, the update would be larger and vice versa. Therefore it can make the algorithm converge more quickly and smoothly.

8.2.4 Adaptive Learning Rate

Choosing suitable size of learning rate is also important. In some directions the curve is smooth, so we want to have large learning rate. In other directions the curve is steep, so we want to have small learning rate. All adaptive learning rate algorithms approximate the curvage of the function with Hessian and adapt the learning rate accordingly.

8.2.4.1 AdaGrad

The gradient is computed as:

$$g = \tilde{\nabla}_w L(w_t) \tag{8.20}$$

The approximated Hessian is computed as:

$$r_t = r_{t-1} + g \odot g \tag{8.21}$$

Then the update rule is:

$$w_{t+1} = w_t - \frac{\epsilon}{\delta + \sqrt{r}} \odot g \tag{8.22}$$

8.2.4.2 RMSProp

The gradient is computed as:

$$g = \tilde{\nabla}_w L(w_t) \tag{8.23}$$

The approximated Hessian is computed as:

$$r_t = \rho r_{t-1} + (1-\rho) g \odot g \tag{8.24}$$

Then the update rule is:

$$w_{t+1} = w_t - \frac{\epsilon}{\delta + \sqrt{r}} \odot g \tag{8.25}$$

8.2.5 Comparison of Optimization Algorithms

Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast, while Momentum and Nesterov Momentum are led off-track, evoking the image of a ball rolling down the hill. Nesterov Momentum, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

For saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD. Notice here that SGD, Momentum, and Nesterov Momentum find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.

8.2.6 Regularization

Neural nets are so flexible that they can even overfit random noise. Regularization is to solve the problem of overfitting. It includes methods introducing uncertainty such as SGD, dropout and batch normalization. It also includes methods introducing constraints to the neural nets, such as early stopping and weight penalties.

8.2.6.1 Dropout

Dropout is a method used by all deep neural networks with fully connected layers currently. It constraints the complexity of the neural nets, and it is also an effective way of ensemble learning. How it works is in training phase, for every neuron the weight is set to be 0 with probability p, in testing phase all neurons are active but the weights multiply $1 - p$ to match the random dropout in training phase.

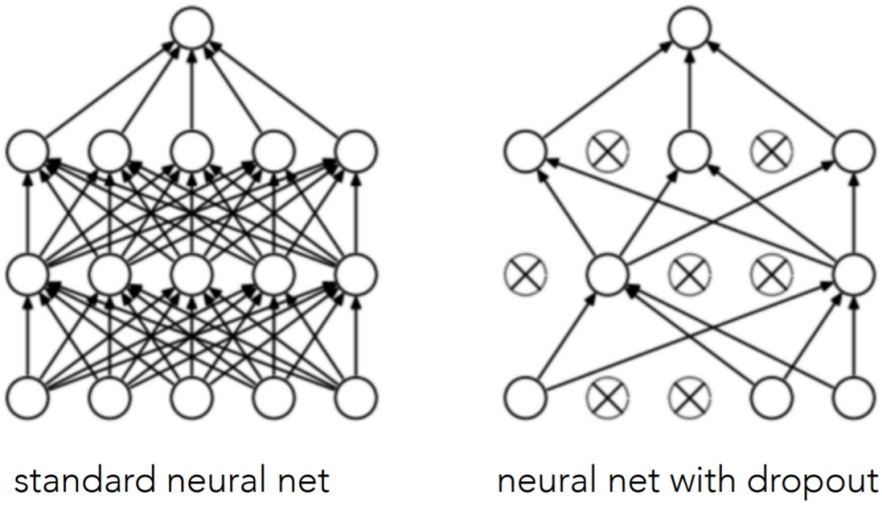

Figure 8.5: Droppping Out Units in Neural Network

8.2.6.2 Batch Normalization

It normalizes each layer's each layer's activations according to the statistics of the mini-batch as follow:

$$s^{(l)} \leftarrow \gamma \frac{s^l - \mu_B}{\sigma_B} + \beta \tag{8.26}$$

Where $s^{(l)}$ is the activation of layer l, μ_B, σ_B are the batch mean and standard deviation. γ and β are additional parameters.

The reason of why batch normalization works is related to a concept called "internal covariate shift". In machine learning a classic assumption is the data distributions between source domain and target domain are the identical. If not there would be new machine learning problems such as transfer learning and domain adaptations, etc. As for covariate shift, it is under the assumption that the data distributions between source domain and target domain are different. It means the conditional probability of source domain and target domain are the same, but the marginal probability are different. Now think about a neural network, the distribution of output of each layer is obviously different from the input, but the label which they correspond to remain the same. Therefore neural network is a covariate shift problem. Researchers found out that using batching normalization to fix each layer's mean and standard deviation, in this way even if the activation or the gradient is very small, it can be scaled up so BN could solve the common "gradient vanishing" problem of deep neural network.

References

[M03] D. MacKay, "Information Theroy, Inference, and Learning Algorithms," *Cambridge University Press 2003*, 2003, pp. 492–503.