CSE 475: Statistical Methods in AI **Monsoon 2018**

Lec 17: Multi Layer Perceptrons (MLP)

Lecturer: C. V. Jawahar *Date: Sep. 3, 2018*

17.57 Single Layer Perceptrons

We know about the linear classifiers that classify based on the simple rule:

$$Sign(\mathbf{w}^T\mathbf{x}).$$

Such a classifier classifies a sample into positive class if $\mathbf{w}^T\mathbf{x}$ is ≥ 0 and negative class if $\mathbf{w}^T\mathbf{x}$ negative. We also know the perception algorithm that learns the $\mathbf{w}$ for a linear separable problem.

We can look at this as a single layer neural network with inputs as $x^1, x^2, \ldots, x^d$ and the weights (on the edges) $w_1, w_2, \ldots w_d$ gets multiplied and added.

There are two computations that are happening with in the neuron. (1) Multiply and add (i.e., compute $\mathbf{w}^T\mathbf{x}$. (2) pass through the nonlinearity $\phi()$, also known as the activation. In this case the nonlinearity $\phi()$ is

$$\phi(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Single Layer Perceptron or simple perceptron has only one layer and the above mentioned activation function. This can classify samples into two class with a linear decision boundary.

Activation function: This nonlinearity $\phi()$ is often called activation function. A disadvantage of the above activation function is that it is not differentiable. A logistic/sigmoid function is often used as the nonlinearity

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

As shown in the figure 17.3, this function ranges between 0 and 1. And also

Figure 17.3: Sigmoid or Logistic Function $\phi(x) = \frac{1}{1+e^{-x}}$

17.58 MLP

We can connect many single layer neural networks to form a multi layer neural network. Eventually this models the transformation of the input $\mathbf{x}$ to $\mathbf{y}$. From this point of view, MLPs can be used for either regression or classification.

In the case of classification, it is the practice to represent the class-ID as a one hot vector. i.e., if there are C classes, there will be C neurons in the output. The the desired class is p, then we represent the output as a C dimensional vector with zero every where except at pth location.

17.59 Notations

The layer to layer transition in a typical MLP can be understood as a matrix multiplication followed by the activation computation.

17.60 Design Choices

17.60.1 Why do we need a nonlinearity?

In typical MLPs, it is assumed that all neutrons in a layer gets connected to all the neurons in the next layer. i.e., the layer is fully connected. Say $\mathbf{x_i}$ is the representation (output of the neurons at the ith layer and $\mathbf{x_{i+1}}$ is the representation at the $i+1$ th layer, then we can compute

$$\mathbf{x_{i+1}} = \phi(\mathbf{W}\mathbf{x}_i)$$

If there was no nonlinearity, then multiple layer perceptron could have reduced to a single layer perceptron.

Example: If $\mathbf{x_2} = \mathbf{W}'\mathbf{x}_1$ and $\mathbf{x_3} = W''\mathbf{x}_2$, then we can in fact write $\mathbf{x_3} = \mathbf{W}'''\mathbf{x_1}$. For some $\mathbf{W}''' = \mathbf{W}' \cdot \mathbf{W}''$

17.60.2 Architecture

A typical MLP what is given to us is the input, output pairs $(\mathbf{x_i}, \mathbf{y_i})$ $i = 1, \ldots N$. Typically $\mathbf{x}$ and $\mathbf{y}$ are vectors. The number of neurons in the first/input layer is the dimensionality of $\mathbf{x}$. Number of neurons in the output layer is the dimensionality of $\mathbf{y}$. We have two things in our control (i) Number of hidden layers (ii) number of neurons in each hidden layer. Typically, we go for 2 or 3 hidden layers. With number of layers increasing, the network becomes deep and the learning problem becomes difficult due to issues like vanishing gradient.

The number of neurons in each layer is typically larger than the number of neurons that are required at the input or output layers. Note that typical neural networks are over parameterized. i.e., there are more neurons and weights than what is required for solving the problem. (Indeed it may be possible to prune and get a smaller network once the network is trained.)

17.60.3 Loss Function

- **MSE**
- **Cross Entropy**
- **Regularized Losses**

17.60.4 Activation Functions

Here are some examples of popular activation functions:

1. **Sigmoid**
2. **tanh**
3. **ReLu**
4. **Leakly Relu**

17.61 Expressive Power of MLP

17.61.1 AND and OR

Consider the problem of implementing "AND" and "OR" with single layer perceptrons.

x^1	x^2	AND	OR	EXOR
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	1	1	0

Let us implement these logics with a single layer neural network with an activation $\phi(x) = 1$ iff $x \geq 0$. With the introduction of bias, what we want to lean is $w_0, w_1 and w_2$.

It can be seen that the following weights satisfy:

- **AND**: $w_0 = w_1 = w_2 =$
- **OR**: $w_0 = w_1 = w_2 =$

Q: Are they the only possible weights that satisfy?

Q: Can we get a valid solution with no bias?

Q: Can you find weights corresponding to NAND and NOR?

Q: Here, we used $0, 1$ logic. Assume we use $-1, 1$ representation, can you redesign the networks?

If we plot this data, we can see that these are linearly separable.

17.61.2 EXOR

However, that is not true for EXOR. It can be easily seen that no solution of the form $w_2x^2 + w_1x^1 + w_0$ is going to work for EXOR.

17.61.3 Representational Power

17.62 Learning

The key question is on "How do we train the neural network." This is possibly more important than how do we decide the number of neurons in each layer or number of layers for a beginner.

Even for an experiences, the learning process is not that simple. Experience in carefully doing an experiment is required to get the best.

The popular algorithm for training neural networks is called "Error Backpropagation Algorithm" or popularly known as "backpropagation algorithm (BP)".

CSE 475: Statistical Methods in AI | **Monsoon 2018**

Lec 18: Back Propagation

Lecturer: C. V. Jawahar | *Date: Oct 11, 2018*

18.63 High Level Picture

Let us consider an MLP as a sequence/chain of computational blocks. (see figure 18.4). In practice, these blocks have a matrix multiplication and an activation function of the form $\mathbf{x}_{n+1} = \phi(\mathbf{W_n}\mathbf{x}_n)$. Here, $\mathbf{x}_n$ corresponds to the number of neurons in the n th layer. Note that the number of neurons in each layer may be different. This implies that $\mathbf{W}_n$ matrix need not be square. Needless to say, they could be different for each layer.

There is a loss layer at the end of the chain. This module computes the discrepancy of the last output ($\mathbf{x}_{p+1}$) with the expected value ($\mathbf{y}$) and compute a scalar loss measure.

The objective of learning is to find the parameters (i.e., $\mathbf{W}$ matrices) that minimize the loss.

Assuming that there are p layers, we have matrices $\mathbf{W}_1, \dots \mathbf{W}_p$ that parameterize the neural network. In otherwise, we have that many parameters to learn.

Our gradient descent learning rule will allow us to learn in the form of

$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^k - \eta \frac{\partial L}{\partial \mathbf{W}^k} \tag{18.38}$$

As in the previous gradient descent schemes, we can start with a random (or preferably a smart) initialization of the weight matrices in the zero iteration (initialization) and update it with every iteration k, until some convergence criteria is met.

However, the problem is not simple. The loss depends only on the $\mathbf{x}_{p+1}$ and the true prediction $\mathbf{y}$. Then how can the partial derivatives in equation 18.38 be nonzero?. On a closer look, we realize that $\mathbf{x}_{p+1}$ depends on the previous weight matrix $\mathbf{W}_p$, and also $\mathbf{x}_p$.

18.64 Derivatives

Computation of the partial derivatives is not that complex, if we use our familiar chain rule. We make an assumption at this stage:

1. **criteria -A** For each block, we know how to compute the partial derivative of the output with respect to that of input.
$$i.e., \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n}$$
2. **criteria -B** For each block we know how to compute the partial derivative of the output with respect to that of the learnable parameters (say $\mathbf{W}$).
$$i.e., \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{W_n}}$$
Some of the blocks may also have learnable parameters θ other than weights. In such case, we also should know the
$$i.e., \frac{\partial \mathbf{x}_{n+1}}{\partial \theta}$$

We can compute the partial derivative of the loss with respect to any of the learnable parameters using the chain rule. This allows us to learn the weights using the gradient update rule in equation 18.38.

18.64.1 Loss Layer

Consider the final loss layer which computes loss from $\mathbf{x}_{p+1}$ and $\mathbf{y}$ for each sample. An example of the loss is

$$\mathcal{L} = \sum_{i=1}^{N} ||\mathbf{x}_{p+1} - \mathbf{y}||^2 = \sum_{i=1}^{N} [\mathbf{x}_{p+1} - \mathbf{y}]^T [\mathbf{x}_{p+1} - \mathbf{y}] \tag{18.39}$$

In this case $\frac{\partial \mathcal{L}}{\mathbf{x_p}}$ is easily computable. Note that $\mathbf{y}$ is constant/fixed. It is part of the data or gound truth.

Q: Do compute $\frac{\partial L}{\mathbf{x_p}}$ for three popular loss functions. (or loss functions that you think are meaningful.)

Figure 18.4: MLP as a sequence of computational blocks. Input $\mathbf{x}$ is same as $\mathbf{x_1}$. Output $\mathbf{x}_{p+1}$ is compared to the true output in the loss layer.

18.64.2 Typical Fully Connected Layer

A typical fully connected later (i.e., all neurons in layer k is connected to all the neurons in layer $k+1$) can be represented as

$$\mathbf{x}_{n+1} = \phi(\mathbf{W_n}\mathbf{x}_n)$$

To make the equations simpler, let us define a temporary variable $\mathbf{t}$, and rewrite the above as two steps.

$$\mathbf{t} = \mathbf{W_n}\mathbf{x}_n \tag{18.40}$$

$$\mathbf{x}_{n+1} = \phi(\mathbf{t}) \tag{18.41}$$

We need to compute $\frac{\partial \mathbf{x}_{n+1}}{\mathbf{W_n}}$ and $\frac{\partial \mathbf{x}_{n+1}}{\mathbf{x_n}}$.
These two are nothing but:

$$\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{W_n}} = \phi'(\mathbf{t}) \cdot \frac{\partial \mathbf{t}}{\partial \mathbf{W_n}} \tag{18.42}$$

$$\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x_n}} = \phi'(\mathbf{t}) \cdot \frac{\partial \mathbf{t}}{\partial \mathbf{x_n}} \tag{18.43}$$

All the partial derivates on the right side are straightforward to compute.

$\phi'(\mathbf{t})$ is a vector of $\phi'(t_i)$. i.e., element-wise evaluation of the derivative.

Note: (i) Look some where else for how the matrix vector derivates are computed. Tom Minka's notes shared in the past is worth. (ii) See discussions somewhere else for How to compute $\phi'()$ for some of the popular activation functions.

18.65 Forward and Backward Passes

We are given $(\mathbf{x_i}, \mathbf{y_i})$ $i = 1, \ldots, N$. Our objective is to learn the weight matrices $\mathbf{w}_i$.

18.65.1 Forward Pass

For each sample in the training data we can give $\mathbf{x}_i$ as input and go through it through a series of matrix multiplications and activations. Finally the network predicts $\mathbf{x}_{p+1}$. We compute the loss per sample using equation 18.39 or similar other loss equations. Finally the total loss $\mathcal{L}$ is computed as the sum of loss over all the samples.

Forward pass is straightforward. It involves many matrix multiplications. This leaves scope for parallelization and running on dedicated hardware at high speed.

18.65.2 Backward Pass

Example 1 Let us consider the situation, we want to update the weight matrix of the last block as

$$\mathbf{w}_p^{k+1} \leftarrow \mathbf{W}_p^k - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$$

How do we compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$?

Chain rule helps us to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p}$ as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_p} \cdot \frac{\partial \mathbf{x}_p}{\partial \mathbf{W}_p}$$

The first term is available (see the text next to equation 18.39) and the next term is available with the definition of the block (see criteria B).

Example 2 Now let us try updating the weights in the last but one block.

$$\mathbf{w}_{p-1}^{k+1} \leftarrow \mathbf{W}_{p-1}^k - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{p-1}}$$

Similar to the previous case, we can compute

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{p-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_p} \cdot \frac{\partial \mathbf{x}_p}{\partial \mathbf{x}_{p-1}} \cdot \frac{\partial \mathbf{x}_{p-1}}{\partial \mathbf{W}_{p-1}}$$

We already know the availability of the first and last term (from the previous example of updating $\mathbf{W}_p$). We also know that the middle term is available from our criteria-A.

Example 3 Now we can compute

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_p} \cdot \frac{\partial \mathbf{x}_p}{\partial \mathbf{x}_{p-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \cdot \frac{\partial \mathbf{x}_2}{\partial \mathbf{W}_1}$$

The point to note in the backward computation is that the partial derivatives required for the computation is available already, if we have updated the weights backwards.

18.66 Backpropagation

The error backpropagation or backpropagation can be summarized as:

1. Initialize the network with random weights.
2. For all the samples, compute the output of the neural network $\mathbf{x}_{p+1}$
3. Compute the loss for the full batch of N samples as
$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_{p+1}^{i}, \mathbf{y}_i)$$
4. Adjust all the parameters (such as weight matrices) as
$$\theta^{k+1} \leftarrow \theta^{k} - \Delta\theta$$
or
$$\theta^{k+1} \leftarrow \theta^{k} - \eta\frac{\partial L}{\partial \theta}$$
5. Repeat steps 2-4 until convergence.

18.66.1 Refinements over backpropagation algorithm

Over years, backpropagation algorithm has been refined significantly with many minor but critical innovations. What all can change in the simple version we saw early?

1. **step 1:** Initialization can be smarter.
2. **step 2:** Computing loss over a full batch and updating it once is not the best.
3. **step 3:** Loss function can be different. There re many other loss functions available beyond MSE.
4. **step 4:** Update rule can be different. What we saw here is too simple.

18.67 Closer Look at the Derivatives (*)

Now that we had seen the larger picture of backpropagation and the chain rule for computing the derivatives, let us have a closer look at the equations 18.42 and 18.43.

$$t_n = W_n\mathbf{x}_n \quad ; \quad \mathbf{x}_n = \phi(\mathbf{t}_n)$$

$$\mathbf{x}_{n+1} : q \times 1 \quad ; \quad \mathbf{x}_n : p \times 1 \quad ; \quad W_n : q \times p \quad ; \quad \mathbf{t}_n : q \times 1$$

$$\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{x}_n} = \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{t}_n}\frac{\partial \mathbf{t}_n}{\partial \mathbf{x}_n} \quad (18.44)$$
$$\frac{\partial \mathbf{x}_{n+1}}{\partial W_n} = \frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{t}_n}\frac{\partial \mathbf{t}_n}{\partial W_n} \quad (18.45)$$

It should be noted that in equation 18.45, even though $\mathbf{t}_n$ is of dimension $q \times 1$ and W_n is dimension $q \times p$, the derivative $\frac{\partial \mathbf{t}_n}{\partial W_n}$ has a maximum of $q \times p$ non-zero values. This is because, the i^{th} element of $\mathbf{t}_n$ depends only on the i^{th} row of W_n. As such, for the purpose of equation 18.45, we assume that the derivative $\frac{\partial \mathbf{t}_n}{\partial W_n}$ is represented by a $q \times p$ matrix with the i^{th} row containing the derivative of the i^{th} element of $\mathbf{t}_n$ with respect to the i^{th} row of W_n. Also to be noted here is that the derivative $\frac{\partial \mathbf{x}_{n+1}}{\partial \mathbf{t}_n}$ would be a $q \times q$ dimensional diagonal matrix, because the i^{th} element of $\mathbf{x}_{n+1}$ depends only on the i^{th} element of $\mathbf{t}_n$.

CSE 475: Statistical Methods in AI **Monsoon 2018**

Lec 19: More on Backpropagation

Lecturer: C. V. Jawahar *Date: Oct. 22, 2018*

We had seen the back-propagation algorithm as one that iteratively minimizing the loss/error over the samples. We discussed the algorithm as two steps:

1. **Forward Pass:** Do a forward pass for all the samples and compute the loss over the training set.
2. **Backward Pass:** Do refine all the learnable parameters (weights) iteratively as:

$$\theta^{k+1} \leftarrow \theta^k - \eta \frac{\partial J}{\partial \theta}$$

To make the discussions and notations simple, we assume that all the learnable parameters are part of θ, and the loss/objective is $J(\theta)$. We use this notation throughout.

19.68 Stochasticity

In practice we do not compute the loss over all the samples and then update the parameters in one go. We do this over a randomly selected subset of the samples. This leads to stochastic mini batch backpropagation algorithm.

Note that the batch mode of the BP computes loss over all the samples. This is actually an approximation of the "true" gradient which we are not able to compute, since we do not know the analytic function form. If the true gradient can be approximated with sum of gradients over a number of samples, thus approximation can be computed from a subset of the samples also. However, computing the gradient from a smaller set may have larger error than that computed from all the samples. However, this is much more efficient. Therefore, we can have BP implemented as batch, single sample and mini-batch. Minibatch is preferred.

The convergence and properties of stochastic gradient descent methods have been analyzed in both convex minimization and stochastic approximation. In many related areas, it has been shown that the stochastic gradient descent will converge to the batch (not stochastic) estimates in many practical situations.

19.68.1 Epochs and Iterations

In neural network literature, it is common to use the word epoch. In the neural network terminology, one epoch consists of one forward pass and one backward pass of all the training examples. However, as we discussed above, batch size (i.e., the number of training examples in one forward/backward pass) could be much smaller than the entire data. Though it is possible to randomly create the batch size every time, it becomes computationally efficient to create random batches once (in the beginning of the training) and use the same batches throughout. When the batch size is large, the memory requirement of the training could increase.

19.69 Sub-gradients

Another issue is the sub-gradient. many functions that we use in the modern deep neural networks are not truely differentiable. Eg. ReLU. How do we handle these?

Eg1: ReLu A popular activation function is ReLU

$$\phi(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Eg2: Hinge Loss Hinge loss has been a key component in the success of SVMs.

$$max(0, 1 - t \cdot y)$$

Though in these two cases, the function is not continuous, we can compute the derivatives at each point, except the point of discontinuity. This is done with the help of "subgradients". In mathematics, the subgradient, generalizes the derivative to convex functions which are not necessarily differentiable.

19.70 Refinements

19.70.1 Initialization

. Initialization is very important for any iterative solution to the non-convex optimization. It is always feared that performance of the neural networks could be highly dependent on this lucky initialization. (Is it really true? Did you find so?)

Q: If we initialize all the weights as zero. What would happen? If we initialize all the weights as non-zero, but a constant value, what could happen?

It is observed that the random initializations are ideal for the neural networks. In the earlier days, it was common to try multiple initializations and pick the best. Even if we randomly initialize the weights, the output of a neuron depends on the inputs. The variance of the output directly depends on the inputs. Glorot and Bengio (2010) proposed to randomly initialize the weights with a variance that depends on the number of incoming (fan-in) and outgoing (fan-out) connections. This method (popularly known as Xavier's initilaization (AISTAT 2010) works well in practice.

19.70.2 Better Update Rules

We know the update rule as:

$$\theta^{k+1} \leftarrow \theta^k - \Delta\theta$$

Gradient Update In the simple gradient descent (and backpropagation) we saw, the change in weight $\Delta\theta$ is proportional to the derivative of the loss/objective.

$$\Delta\theta = \eta \frac{\partial L}{\partial \theta}$$

Momentum Momentum based optimization provided better convergence properties, and results. The idea is simple. If we conistently change the weights/parameters in certain direction, we can make larger steps instead of small steps.

$$\Delta\theta = \eta \frac{\partial L}{\partial \theta} + \gamma \Delta\theta^{t-1}$$

If we change the directions frequently, then we make small step. Connecting to the analogies from physics, we make big steps, if the surface is smooth. If surface is rough, we make small steps. Typically the momentum is set to 0.9. This classical momentum term could carry the ball beyond the minimum point. Ideally, we would like "the ball" to slow down when the ball reaches the minimum point and the slope starts increasing. This is achieved by the Nesterov momentum.

Another aspect is that we have only one parameter for the learning rate. Can we have different parameters/scales for each dimension? Adaptive gradients and its variations are aimed at this.

Other Recent Advances(*) A number of refinements have surfaced in the recent years:

- Nistrong Moment [Nesterov 1983]
- AdaGrad [Duchi 2011]
- AdaDelta [Zeiler 2012]
- RMSProp [Tieleman and Hinton, 2012]

Though some of these refinements were recent, they have all reached the end users through the popular libraries for neural networks.

The ADAptive Moment Estimation (ADAM) [Kingma and Ba, ICLR 2014] is a popular refinement of the update rule similar to the momentum techniuqes. The main difference between Adam and its two predecessors (RMSprop and AdaDelta) is that the updates are estimated by using both the first moment and the second moment of the gradient. A running average of gradients (mean) is maintained along with a running average of the squared gradients.

19.70.3 Termination Criteria

Termination is often when there is no significant change in the loss/objective. However, a larger number of iterations can lead to overfitting. Often an "early stop" is performed by looking at how the loss/objective change over the validation data set. When the loss starts increasing on the validation data, iterations are stopped.

19.71 Loss Functions

We know loss function as a measure of discrepancy between the ground truth (true value) and the predicted output. The mean square error

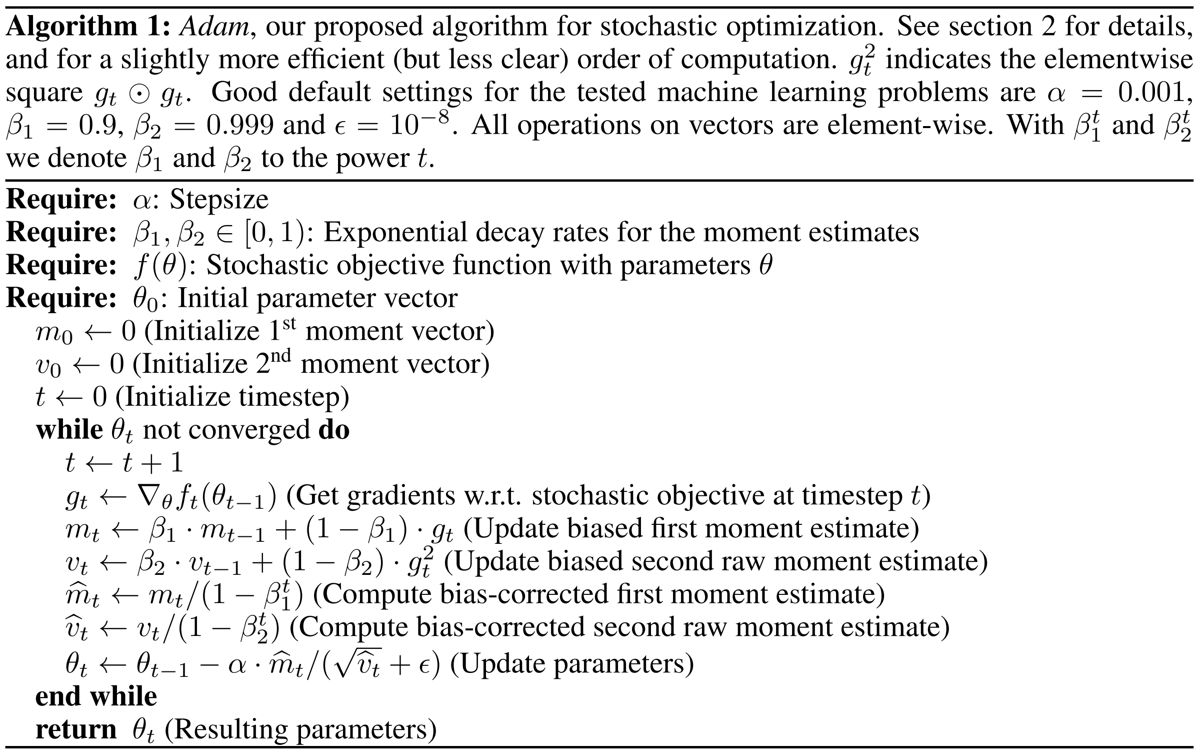
$$\sum_{i=1}^{N} (t_i - o_i)^2$$

is a useful measure in this regard. This is popular for many regression tasks, where t_i and o_i are reals. If there are more than one neurons in the output layer (say M), then one can compute it by summing up over all the M neurons

$$\sum_{i=1}^{N} \sum_{j=1}^{M} (t_i^j - o_i^j)^2$$

This is not the apt measure for classification related tasks. When the output has M neuons and output of these neurons corresponds to the probabilities to these classes, then it is typical to use softmax in the output layer.

$$o^k = \frac{e^{o^k}}{\sum_{j=1}^{M} e^{o^j}}$$



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 19.5: Adam: Algorithm. taken from Kingma and Ba, ICLR 2014

. The softmax output o^k could be considered as the probability to be in class k. p^k.

The popular loss in this case is cross entropy loss

$$= \sum_{j=1}^{M} y^j \log p^j$$

for a M class classification problem. To appreciate this loss better, let us look at how can we compute the distance/dissimilarity between two probabilistic distributions. The classical measure for this is called Bhattacharya distance (or some people also call it as KL-divergemce)

$$D_{KL}(p(x)||q(x)) = \sum_x p(x) \ln \frac{p(x)}{q(x)}$$

A symmetric version $(D_{KL}(p(x)||q(x)) + D_{KL}(q(x)||p(x)))$ is also preferred in many cases. See Wikipedia on KL divergnce to appreciate the relationship between cross entropy and KL divergence.

19.71.1 Regularization

It is common to regularize the neural networks in different ways to prevent overfitting. One classical method is to look for simple/small neural network that can solve the problem of interest. The simplicity of such a neural network is measured with the number of non-zero weights (L0 norm of the weights) or sum of absolute value of the weights (L1 norm of the weights). Both L0 and L1 norms are hard to work with in most cases. In practice L2 norm is a good choice and the new loss then becomes old loss plus the L2 norm of the weights in the neural networks.

$$J' = J + ||\theta||_2^2$$

Please note that the new term is an addition. (remember $u+v$) and the new update rule will be almost the same as the old update rule plus an additional term corresponding to the L2 norm of the weights.

One can attempt to minimize $L0$ norm by pruning (removing or explicitly making it as zero) some of the tiny weights. Usually removal of some of the tiny weights need not change the network performance (output values) significantly. One of two iterations of the backpropagation algorithm can recover any loss in performance due to this. However, having zero elements in the weight matrices will not save memory or computation if your inherent data structure is matrix/tensor. One needs to use appropriate sparse matrix computing techniques in this case.

19.72 Second Order Methods

The popular backpropagation algorithm that we studied till now uses only the first order derivatives. It takes linear steps along the negative gradient. Assume we have knowledge about the curvature/shape of the curve, then we could obtain better estimate of the solution in every step. Second order method uses Hessian (matrix of second order derivatives) in every step. (Recollect our discussions related to second order gradient descent methods elsewhere.)

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 J}{\theta_1^2} & \frac{\partial^2 J}{\theta_1 \theta_2} & \cdots & \frac{\partial^2 J}{\theta_1 \theta_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 J}{\theta_N \theta_1} & \frac{\partial^2 J}{\theta_N \theta_2} & \cdots & \frac{\partial^2 J}{\theta_N^2} \end{bmatrix}$$

Remembering the truncated Taylor series expansion,

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla J(\theta_0) + (\theta - \theta_0)^T H(\theta - \theta_0)$$

We could take the derivative and equate to zero:

$$\nabla_\theta J(\theta) = \nabla_\theta J(\theta_0) + H(\theta - \theta_0) = \mathbf{0}$$

or

$$\theta = \theta_0 - \mathbf{H}^{-1} \nabla_\theta J(\theta_0)$$

If the function was quadratic, this step directly takes us to the minimum. (Q: why?). If not, this takes us to the minima of the quadratic approximation.

In general, we need to now compute g, compute H and then update as $\theta^{k+1} \leftarrow \theta^k - \mathbf{H}^{-1}\mathbf{g}$ Where $\mathbf{g} = \nabla_\theta J(\theta)$

The above mentioned second order method (alias Newtons method) requires, at every iteration, calculating and then inverting the Hessian. If the network has N parameters, then inverting the Hessian is $O(N^3)$. This renders Newtons method impractical for the modern deep neural networks.

When the Hessian has negative eigenvalues, then steps along the corresponding eigenvectors are gradient ascent steps. To counteract this, it is possible to regularize the Hessian, so that the updates become:

$$\theta \leftarrow \theta^k - (\mathbf{H} + \alpha I)^{-1} \nabla_\theta J(\theta_0)$$

As α becomes larger, this turns into first order gradient descent with learning rate $\frac{1}{\alpha}$.

19.73 Discussions (*)

19.73.1 Vanishing and Exploding Gradients

We had seen the derivation of the gradients using chain rule. If the gradients are small, then the product can "vanish" very fast. Similarly if the gradients are larger, then the products can explode very fast.

When cost function has "cliffs", where small changes in θ, drastically change the cost function. (This usually happens if parameters are repeatedly multiplied together, as in recurrent neural networks.) Similar to exploding gradients, repeated multiplication of a matrices/vectors can cause vanishing gradients.

These problems of vanishing and exploding problems have bothered for long time in numerically training deep neural networks.

19.73.2 More on Second Order Methods

To get around the problem of having to compute and invert the Hessian, quasi-Newton methods are often used. Amongst the most well-known is the BFGS (Broyden Fletcher Goldfarb Shanno) update.Quasi-Newton methods usually require a full batch (or very large minibatches) since errors in estimating the inverse Hessian can result in poor steps.

CG methods are also beyond the scope of this class, but we bring it up here in case helpful to look into further. Again, ECE 236C is recommended if youd like to learn more about these techniques.

- CG methods find search directions that are conjugate with respect to the Hessian, i.e., that $g_k^T H g_{k1} = 0$.
- It turns out that these derivatives can be calculated iteratively through a recurrence relation.
- Implementations of Hessian-free CG methods have been demonstrated to converge well (e.g., Martens et al., ICML 2011).

19.73.3 Further Reading

This area has many advanced reading material. Interested students may read the original papers, tutorial notes online. They are beyond the scope of this course.