Deep Learning Study Lecture Note

Lecture 2 : MultiLayer Perceptron

Kim Tae Young

January 3, 2020

2.1 Introduction

In this lecture, I will cover some loss function to use, activation functions, definition of MLP, and batch normalization, dropout.

2.2 Activation Function

Recall the Universal Approximation Theorem. You can see activation function f there. These are some conditions for function f required to approximate other functions.

Condition of activation function	Approximable functions
Continuous, Sigmoidal	$C(I_n)$
Monotonic, Sigmoidal	$C(I_n)$
Sigmoidal	$C(I_n)$
$f \in L^1(\mathbb{R})$, $\int f(t)dt \neq 0$	$L^1(I_n)$
Continuous, Sigmoidal	$L^2(I_n)$
Indicator of Rectangle	$L^1(I_n)$
$f \in L^1(\mathbb{R})$, $\int f(t)dt \neq 0$	$L^1(\mathbb{R}^n)$

Figure 2.1: Condition for Activation function

Definition 2.2.1. *A function f is sigmoidal if*

$$f(t) \to \begin{cases} 1 & as\ t \to \infty \\ 0 & as\ t \to -\infty \end{cases}$$

So it is reasonable to use continuous, sigmoidal function for our activation function. These two functions are mostly used.

$$tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$$

$$sigmoid(t) = \frac{1}{1 + e^{-t}}$$

However, the main problem of these two activation functions is that in training, gradient vanishing occurs. If we find maximum of derivative, we get $tanh'(0) = 1, sigmoid'(0) = \frac{1}{4}$. Meaning that every time we apply activation function, gradient decreases. So instead, ReLU(Rectified Linear Unit) function is well used for activation function.

$$ReLU(t) = \begin{cases} t & t \geq 0 \\ 0 & t < 0 \end{cases}$$

It seems that ReLU should not work, since it is almost everywhere linear, and it is not sigmoidal. However in practice, it works well.

There are also other activation functions like LeaklyReLU, ReLU6, SELU, etc. If you are intersted, please see Other Activation Functions.

2.3 Definition of MLP

Multi Layer Perceptron is simple neural network that uses fully connected (or linear) layer. Here, every edge multiplies with its weight, then each node sums all

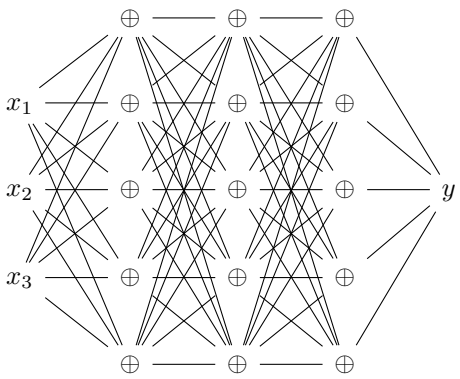

Figure 2.2: Multi Layer Perceptron

of its inputs and bias, then apply activation function to it. The result becomes output of each node. The layer that gets input is called input layer, where last layer is called output layer and the others are called hidden layer.

The hyperparameters for MLP are number of hidden layers, dimension of hidden layer, and activation function. We call N-layer neural network, when the neural network has N layers excepting the input layer. So the neural network over there is 4-layer neural networks. Also, we have hidden dimension 5. Usually, we use same dimension for every hidden layer.

The number of parameter usually becomes measure for the complexity of neural network, where high complexity means that neural network can approximate more complex functions. However, higher complexity also means that overfitting can occur. So adjusting number of layer and dimension of hidden layer is one key of regulating overfitting.

You can calculate number of parameter by following equation.

$$\#\{parameters\} = (d_{in}+1) \times d_{hid} + (d_{hid}+1) \times d_{hid} \times (n-1) + (d_{hid}+1) \times d_{out}$$

where n is number of layer, d_{in}, d_{hid}, d_{out} are dimensions of input, hidden, output layer, respectively. Here, $+1$ is added for bias.

If you remember Universal Approximation Theorem, it seems one single layer is enough. So why we need, or use multi layer perceptron? The reason is explained in [1]

First, we assume using ReLU activation function, where it is viewed as dividing current space by hyperplane. The divided spaces are called linear region, where definition for linear regions is following.

Definition 2.3.1. *The linear region of a piecewise linear function $F : \mathbb{R}^{n_0} \to \mathbb{R}^m$ is a maximal connected subset of the input space $\mathbb{R}^{n_0}$ on which F is linear.*

First, let's analyze single layer neural network. Since ReLU has irregular (Here meaning non linearity) behavior at zero, the function $x \mapsto ReLU(W_i x + b_i)$ has irregular behavior at hyperplane $H_i = \{x \in R^{n_0} : W_i x + b_i = 0\}$. Then the linear regions that is splitted by these hyperplane is analyzed by hyperplane arrangement. Since arrangement of n_1 hyperplane in $\mathbb{R}^{n_0}$ has at most $\sum_{j=0}^{n_0} \binom{n_1}{j}$, single layer neural netowrk with n_0 inputs and n_1 hidden unit has maximum $\sum_{j=0}^{n_0} \binom{n_1}{j}$ linear regions.

Now we will focus on deep neural network. First, we will define identification of input neighborhoods.

Definition 2.3.2. *A map F identifies two neighborhoods S, T of its input domains if $F(S) = F(T)$.*

Now we will consider function $F_l : \mathbb{R}^{n_0} \to \mathbb{R}^{n_l}$, where $F_l = h_l \circ \ldots \circ h_1$ for $h_i = g_i \circ f_i$. Then we will denote image of F_l as $S_l \subseteq \mathbb{R}^{n_l}$. And given subset $R \subseteq S_l$, we will let P_R^l as the set of subsets $\bar{R}_1, \ldots, \bar{R}_k \subseteq S_{l-1}$ that are mapped by h_l onto R, that is, $h_l(\bar{R}_1) = \ldots = h_l(\bar{R}_k) = R$.

Then the number of separate input space neighborhoods that are mapped to a common neighborhood $R \subseteq S_l \subseteq \mathbb{R}^{n_l}$ is given recursively as

$$\mathcal{N}_R^l = \sum_{R' \in P_R^l} \mathcal{N}_{R'}^{l-1} \qquad \mathcal{N}_R^0 = 1$$

Then we have following result,

Lemma 2.3.3. *The maximal number of linear regions of the functions computed by an L-layer neural network is at least $\mathcal{N} = \sum_{R \in P^L} \mathcal{N}_R^{L-1}$ where P^L is a set of neighboorhoods in distinct linear regions of the function computed by the last hidden layer.*

Also, by constructing deep neural network well, we can show following theorem.

Theorem 2.3.4. *The maximal number of linear regions of the functions computed by a neural network with n_0 input units and L hidden layers, with $n_i \geq n_0$ rectifiers at the i-th layer, is lower bounded by*

$$\left(\prod_{i=1}^{L-1} \lfloor \frac{n_i}{n_0} \rfloor^{n_0}\right) \sum_{j=0}^{n_0} \binom{n_L}{j}$$

Now remember the narratives. In neuroscientific narrative, MLP is simulation of neuron. The input connection becoms axon from a neuron, and weight is synapse, then in cell body we sum all input then apply activation, which works as threshold for excitatory and inhabitory. Then finally output works as outgoing axon. Note that this is not exact simulation of neuron.

In probablistic narrative, we view distribution as linear sum of latent variables, where latent variables are L-1-th layer. Then again vieweing distribution of latent variables as linear sum of L-2-th layer, and recursively following.

Finally in representation narrative, each layer is viewed as 'folding' the space, when using ReLU as activation function.

2.4 Loss function

Loss function is what determining optimization direction. Here, I will explain various loss functions that are implemented in torch.nn. If you'd like to see more, see loss functions

Using loss functions depend on problem, whether we are doing (un-)supervised learning, and whether we are embedding to discrete space or continuous space.

2.4.1 Regression

The main loss we use in regression loss is MSE loss and L1 loss.

Definition 2.4.1. *L1 loss is criterion that measures mean absolute error.*

$$l(x, y) = \frac{1}{N} \sum_{i=1}^{N} |x_i - y_i|$$

Definition 2.4.2. *MSE loss is criterion that measures mean squared error.*

$$l(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x_i - y_i)^2$$

There are also loss functions like smoothL1, which performs robust loss by giving lower gradient to outlier.

2.4.2 Classification

There are several different problems in classification, here I will show classical classification and multi class classification.

NLL loss and Cross Entropy loss are mainly used loss for classification problem. However, Cross Entropy loss is more widely used.

Definition 2.4.3. *NLL(Negative Log Likelihood) loss is useful to train a classification problem with C classes. The input is expected to contain log probabilities of each class, and the target is given as class index in* $[0, C-1]$*, where C is number of classes. Here,* w_i *is weight for each class i, when given data is not balanced. (For example, if we have 100 data but 80 of them are class A and the rest are class B,* $w_1 = 0.8$*,* $w_2 = 0.2$*.)*

$$l(x, y) = \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} - w_{y_n} x_{n,y_n}$$

Definition 2.4.4. *The cross entropy loss is also used in classification problem with C classes. Cross entropy loss is combination of log softmax function and NLL loss, where log softmax is applied to obtain log probability in neural network.*

So cross entropy loss expects raw scores for each class, and target is also class index.

$$l(x, y) = -\sum_{n=1}^{N} \log \frac{\exp(x_{n,y_n})}{\sum_j \exp(x_{n,j})}$$

One benefit of using Cross Entropy Loss is that torch optimizes this loss, since when we manually implements cross entropy loss, we computes log value, resulting floating point underflow.

Definition 2.4.5. *Multi Margin Loss is well used in multi class classification, meaning that each data may be element of several classes. This usually happens in object detection, since there might be several objects of different class in image.*

$$l(x, y) = \frac{\sum_i \max(0, margin - x_{n,y_n} + x_{n,i})^p}{C}$$

The default value for margin and p are both 1.

2.4.3 Metric Learning

Metric Learning is major problem for semisupervised learning, where we'd like to map items in same class on nearby space, and we wants items in different class has distance larget than some fixed value, Δ.

Definition 2.4.6. *Hidge Embedding Loss receives input tensor x and labels tensor y. This is used on measuring whether two inputs are similar or dissimilar, by using losses in regression for x.*

$$l(x, y) = \frac{1}{N} \sum_{n=1}^{N} \begin{cases} x_n & \text{if } y_n = 1 \\ \max\{0, \Delta - x_n\} & \text{if } y_n = -1 \end{cases}$$

Usually, we use as following.

$$l(x_1, x_2, y) = l(||x_1 - x_2||, y)$$
$$l(x_1, x_2, y) = l(||x_1 - x_2||_2, y)$$

Here label 1 means that x_1, x_2 are in positive pair, meaning they are in same class. Conversely, label -1 means that x_1, x_2 are in negative pair, they are different class.

2.4.4 Regularization Loss

Now I'd like to show regularization loss, which is used to regularize overfitting. Since higher value of weight and bias means the neural network is more complex, so we'd like to reduce each value of weight and bias.

Definition 2.4.7. *L1 loss (Lasso Regularization) is loss that applied to every tensor in neural network.*

$$l(NN) = \lambda \sum_{\theta \in NN} |\theta|$$

Definition 2.4.8. *L2 loss (Ridge Regularization) is loss that applied to every tensor in neural network.*

$$l(NN) = \lambda \sum_{\theta \in NN} \theta^2$$

2.5 Dropout

Dropout is well used regularization method in MLP, and even CNN, RNN.

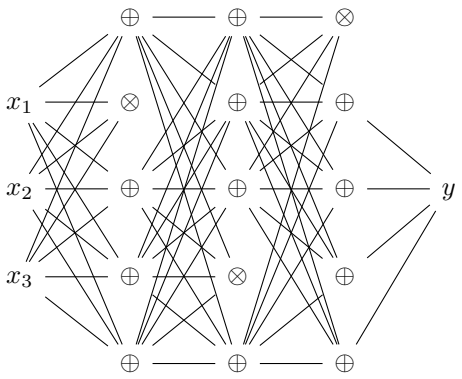

Figure 2.3: Dropout network with p=0.2

As you can see in figure, node $\otimes$ are dropped out. The node's are dropped out for given dropout rate.

Before dropout, regularizations are well used for resolving overfitting, however it didn't work well. The main reason of this is co-adaption.

Since every weights are learned together, there exists some of the connections that have more predictive capability than the others. And in repeated training, these strong connections are learned more, where weak ones are ignored, or less learned. Therefore only small amount of note is trained. Since L1, L2 regularization regularize based on the predictive capability of the connections, it couldn't be prevented by these. As a result, larger neural network size had no benefit.

There dropout emerged. Here, I will show that dropout do regularization, by showing that expectation of gradient value is same as original neural network with regularization. I'll assume linear activation $f(x) = x$ for ease. Let our loss for original neural network and dropout neural network as

$$\mathcal{L}_N = \frac{1}{2}(t - \sum_{i=1}^{n} w_i' I_i)^2$$

$$\mathcal{L}_D = \frac{1}{2}(t - \sum_{i=1}^{n} \delta_i w_i I_i)^2$$

Here, δ_i denotes bernoulli random variable, meaning $\delta_i \sim Bernoulli(p)$.

Then we can compute gradient for dropout neural network as

$$\frac{\partial \mathcal{L}_D}{\partial w_i} = -t\delta_i I_i + w_i \delta_i^2 I_i^2 + \sum_{j=1, j\neq i}^{n} w_j \delta_i \delta_j I_i I_j$$

With $w_i' = p_i w_i$ we have gradient of original neural network

$$\frac{\partial \mathcal{L}_N}{\partial w_i} = -tp_i I_i + w_i p_i^2 I_i^2 + \sum_{j=1, j\neq i}^{n} w_j p_i p_j I_i I_j$$

Finally we have

$$E[\frac{\partial \mathcal{L}_D}{\partial w_i}] = -tp_i I_i w_i p_i^2 I_i^2 + w_i Var(\delta_i) I_i^2 + \sum_{j=1, j\neq i}^{n} w_j p_i p_j I_i I_j = \frac{\partial \mathcal{L}_N}{\partial w_i} + w_i p_i (1-p_i) I_i^2$$

Then you can see final term is gradient of L2 regularization term, having regularization effect.

So as we can see here, Dropout has regularization effect, also regularizing co-adaption. Instead, we usually have trade-off that adding dropout increases learning time.

2.6 Batch Normalization

Finally we will see batch normalization. Batch normalization is as the name suggests, normalizes inputs in each batch. It is computed as following.

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \times \gamma + \beta$$

So for every layer, we apply batch normalization, where γ, β are 1-D tensor with same dimension to input, and it is also learned while training. I will not compute gradients for these two values.

The main effect of batch normalization is first, faster training speed. Also, batch normalization prevents gradient explosion or gradient vanishing, which makes some parameter to not learnable. And finally, applying batch normalization makes hyper parameter tuning more robust, meaning we don't need to tune a lot.

Then what is reason of these effects? In the first paper that batch normalization was introduced, internal covaraite shift was suspected as the reason. Meaning, ICS disturbs training, and batch normalization reduces ICS. One recent research first showed that ICS does not disturb training, and also batch normalization even does not decrease ICS. This was showed by two experiment, first adding noise to batchnorm which increases ICS, however it showed better result then vanilar neural network. Second the paper defined some notion related to ICS, and computed it to show that applying batchnorm didn't decrease ICS.

Definition 2.6.1. *Let $\mathcal{L}$ be the loss, $W_1^{(t)}, \ldots, W_k^{(t)}$ be the parameters, and $(x^{(t)}, y^{(t)})$ be the batch of input label pairs used to train the network at time t. The internal covariate shift (ICS) of activation i at time t is difference $||G_{t,i} - G'_{t,i}||_2$, where*

$$G_{t,i} = \nabla_{W_i^{(t)}} \mathcal{L}(W_1^{(t)}, \ldots, W_k^{(t)}; (x^{(t)}, y^{(t)}))$$
$$G_{t,i} = \nabla_{W_i^{(t)}} \mathcal{L}(W_1^{(t+1)}, \ldots, W_{i-1}^{(t+1)}, W_i^{(t)}, W_{i+1}^{(t)}, \ldots, W_k^{(t)}; (x^{(t)}, y^{(t)}))$$

Instead, the same paper argues that batch norm's result is due to smoothing effect of it. We can see neural network training problem as optimization problem of minimizing loss over space of weights and bias. Then adding batch normalization makes this loss function more smooth and lipscitz. This result is written as following.

Theorem 2.6.2. *For a BatchNorm network with loss $\hat{\mathcal{L}}$ and an identical non-BN nework with identical loss $\mathcal{L}$,*

$$||\nabla_{y_j} \hat{\mathcal{L}}||^2 \leq \frac{\gamma^2}{\sigma_j^2}(||\nabla_{y_j} \mathcal{L}||^2 - \frac{1}{m}\langle 1, \nabla_{y_j} \mathcal{L}\rangle - \frac{1}{m}\langle \nabla_{y_j} \mathcal{L}, \hat{y}_j\rangle^2)$$

So now our question is, why lipscitzness of optimization space affects training? The main reason is predictivity of gradient descent. As a derivative denotes, what we expect at each step of gradient descent is decrease of loss, of similar amount. And higher lipscitzness of optimization space yields higher predictivity.

2.7 Weight Initialization

Since gradient descent algorithm is not optimization algorithm that converges to global minimum, the initialization of parameters are crucial. Then first, you may think initializing every weight as 0, which results absolutely no learning at all. And similarly, if we initialized every weight lower, then gradient will be too small so that learning speed is slow. In contrary, if initilized weight are too large, the value whould be too large to useful. So the problem is what value for standard deviation is correct.

From here, fan_in denotes input dimension of that weight matrix, and fan_out denotes output dimension of that weight matrix. Also these initializations only talks about weight matrices. For bias, we usually take 0 as default value.

Definition 2.7.1. *Xavier Init (also known as Glorot Init) is well used initialization in neural network with non rectified activation.*

Xavier Uniform Init initializes weights from uniform distribution $U(-b, b)$ where $a = gain \times \sqrt{\frac{6}{fan\_in + fan\_out}}$. Xavier Normal Init initializes weights from normal distribution $\mathcal{N}(0, \sigma^2)$ where $\sigma = gain \times \sqrt{\frac{2}{fan\_in + fan\_out}}$.

Sigmoid	1
Tanh	$\frac{3}{5}$
ReLU	$\sqrt{2}$
LeaklyReLU	$\sqrt{\frac{2}{1+a^2}}$

Figure 2.4: Gain value table

Here, gain is recommended value for each activation function, defined as following. These values are chosen by experiments.

Definition 2.7.2. *He Init (also known as Kaiming Init) is well used initialization in neural network with rectifed activation.*

He Uniform Init initializes weights from uniform distribution $U(-b, b)$ *where* $b = \sqrt{\frac{6}{(1+a^2) \times fan\_in}}$. *He Normal Init initializes weights from normal distribution* $\mathcal{N}(0, \sigma^2)$ *where* $\sigma = \sqrt{\frac{2}{(1+a^2) \times fan\_in}}$. *Here, a is negative slope of Leakly ReLU. If you use ReLU, it is 0 by default.*

Here, the main idea is to preserves variance when applying each layers. First we assume single linear layer, and our inputs to have zero mean.

$$y = w_1 x_1 + \ldots w_n x_n + b$$

Then

$$var(y) = var(w_1 x_1 + \ldots + w_n x_n + b)$$

Given following equation

$$var(w_i x_i) = E(x_i)^2 var(w_i) + E(w_i)^2 var(x_i) + var(w_i) var(x_i)$$

since we have $E(x_i) = E(w_i) = 0$, $var(w_i x_i) = var(w_i) var(x_i)$. Also, assuming each weight and inputs comes from independent distribution, we have

$$var(y) = var(w_1) var(x_1) + \ldots + var(w_n) var(x_n)$$

and they are identically distributed, $var(y) = N var(w_i) var(x_i)$. Since we wants our output has same distribution, letting $var(y) = var(x_i)$ gives $var(w_i) = \frac{1}{N}$, having result of $\sigma = \sqrt{\frac{1}{fan\_in}}$. Then why we also add fan_out in xavier init? This is to also preserve back propagation signal.

Bibliography

[1] Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, Yoshua Bengio *On the number of linear regions of deep neural neworks.*
`https://arxiv.org/abs/1402.1869`

[2] Chitta Ranjan. *Understanding Dropout with the Simplified Math behind it*
https://towardsdatascience.com/simplified-math-behind-dropout-in-deep-learning-6d50f3f47275

[3] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry. *How Does Batch Normalization Help Optimization?*
`https://arxiv.org/abs/1805.11604`

[4] Prateek Joshi. *Understanding Xavier Initialization In Deep Neural Networks*
https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/