University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 10: October 2 and October 4

Lecturer: Alexander Schwing Scribe: Chen Chen

1 Introduction

In this lecture, we started to learn Deep Neural Networks. The things we discussed include the motivation behind using deep net, the general framework and the different functions or layers that can be used, some examples of network architecture. And finally we touched a little on how to train the network.

2 Motivation and Framework of Deep Net

In previous lectures, we have generated a framework that can generalize to SVM and logistic regression:

$$\min_{\mathbf{w}} \frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}} \epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)},\hat{y}) + \mathbf{w}^\intercal \psi(x^{(i)},\hat{y})}{\epsilon} - \mathbf{w}^\intercal \psi(x^{(i)},y^{(i)}) \tag{1}$$

As we discussed in the previous lecture, this framework is limited because it has linearity in the feature space $\psi(x,y)$. We used kernels to make transformation in the feature space to fix this problem last time, but it is still a linear model in the parameters $\mathbf{w}$. Therefore, to overcome this limitation, we proposed a new solution. Instead of using $\mathbf{w}^{\mathbf{T}}\psi(x,y)$ in the above equation, we now use a general function $F(\mathbf{w},x,y)$. The framework is now changed to:

$$\min_{\mathbf{w}} \frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}} \epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)},\hat{y}) + F(\mathbf{w},x^{(i)},\hat{y})}{\epsilon} - F(\mathbf{w},x^{(i)},y^{(i)}) \tag{2}$$

Note that $F(\mathbf{w},x,y)$ is a general function that takes in weight $\mathbf{w}$, data x and labels y as inputs and generate a real number as output. Now, from this new framework, we can get to logistic regression, binary SVM, multiclass regression, multiclass SVM and deep neural networks by choosing appropriate ϵ and $F(\mathbf{w},x,y)$. For this lecture, we will be focusing on deep neural networks since the other ones have already been covered in the previous lectures.

3 Function choices for Deep Net

3.1 General idea of deep net functions

Now that we got the framework for deep net, we need to think about what function $F(\mathbf{w},x,y) \in \mathbb{R}$ to choose. Usually, for deep neural networks, we can choose any differentiable composite function as long as we can perform gradient descent later on for training. But there is one important factor that should be taken into account when choosing the function. That is we would want inference of a data point to be quick. This means given weight $\mathbf{w}$ and a data point x, we can get the classification label y easily. The general method to compute the classification label y would be computing $\arg\max_{\hat{y}} F(\mathbf{w}, x, \hat{y})$ with $\{\hat{y} \mid \hat{y} \in \{1, ..., k\}\}$ using exhaustive search. So, to speed up this process, it is often common to have y close to the final output in the function as shown in equation 3 (there are recent attempts of moving y somewhere inside, but it will be out of scope of this class). One possible function choice is:

$$F(\mathbf{w}, x, y) = f_1(\mathbf{w_1}, y, f_2(\mathbf{w_2}, f_3(...f_n(\mathbf{w_n}, x)...))) \tag{3}$$

In this function, each individual function f_i takes in weight $\mathbf{w_i}$ and either data x or the function output from f_{i+1} until the outermost layer f_1 where y is taken as one of the inputs to get the final output. Inference with a data point x would be fast because for the same x and different $\hat{y}$, we only have to compute the inside layers output once and get the final output with different $\hat{y}$ quickly by computing $f_1(\mathbf{w_1}, \hat{y}, f_2)$. Moreover, note that on the right-hand side of this function, $\mathbf{w_i}$ does not have to be same dimension. $\mathbf{w}$ in $F(\mathbf{w}, x, y)$ is just one long vector that represents all the $\mathbf{w_i}$.

In a more general form, we would say that the functions can be represented by an acyclic graph (computation graph). This means there are certain orders to compute each part of the graph to get the final result. This is what calculators do to perform operations between scalars. It is actually the same mechanism in deep learning frameworks such as PyTorch and Tensorflow. The only differences are that they can compute on any type of data (tensors in this case) and they can compute gradients. To give a concrete example, for the following function:

$$F(\mathbf{w}, x, y) = f_1(\mathbf{w_1}, f_2(\mathbf{w_2}, f_3(...))) \tag{4}$$

Take weights, data and functions as the nodes of the graph, we have a computation graph that can represent this whole function as shown in Figure 1. Note that computation graph is directed. The direction of the edges indicate the order of the computations.

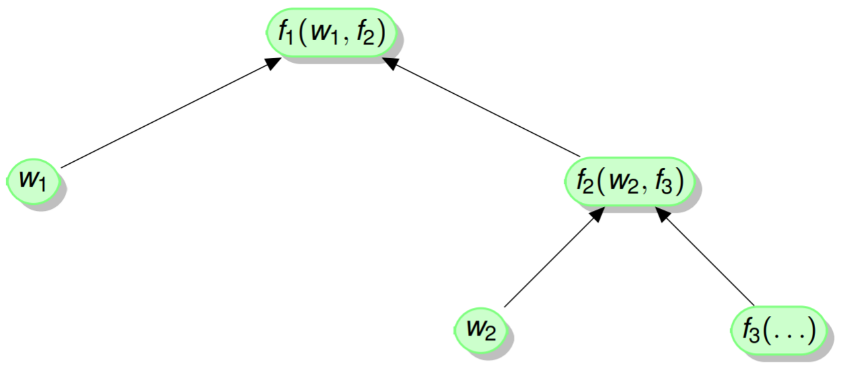

Figure 1: Computation graph representation of $F(\mathbf{w}, x, y) = f_1(\mathbf{w_1}, f_2(\mathbf{w_2}, f_3(...)))$

3.2 Individual functions (layers) of deep net

After we had a general idea of what the $F(\mathbf{w}, x, y)$ should be like, we can now explore what could the individual functions (layers) f_i be. There are several common ones that we discussed in this lecture:

- Fully connected layers
- Convolutions

- Rectified linear units (ReLU)
- Maximum/Average pooling
- Softmax layer
- Dropout

3.2.1 Fully connected layers

A fully connected layer means every node in one layer connects to every node in the next layer. Figure 2 gives a brief idea of how a two-layer fully connected network would look like. Think of each node as data point value and each edge as a weight value, we get value of each node in the second layer by adding up all the weighted values from the first layer, and each node in the third layer by adding up all the weighted values from the second layer. Is it also common to have a bias value for each node starting from the second layer. Therefore, a simple way to represent one fully connected layer would be $\mathbf{Wx} + \mathbf{b}$. The trainable parameters $\mathbf{w}$ would be weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ in this case.

One issue with fully connected layers is that the size of the parameters would grow super fast when the size of the input data grows. For instance, in Figure 2, if we only look at the first and second layer, both the input ($\mathbf{x}$) and output data ($\mathbf{Wx} + \mathbf{b}$) have size $5 * 1$. To get full connections, we will need $5 * 5 = 25$ edges as weights, which means the weight matrix $\mathbf{W}$ has size $5 * 5$ and the bias vector $\mathbf{b}$ has size $5 * 1$. Now imaging we are dealing with an image of size $256 * 256$ as input and we want to have an output of identical size. To get full connections, we will now have a weight matrix $\mathbf{W}$ of size $(256 * 256) * (256 * 256) = 2^{32} = 4294967296$. It would soon be impossible to compute if we add more layers.

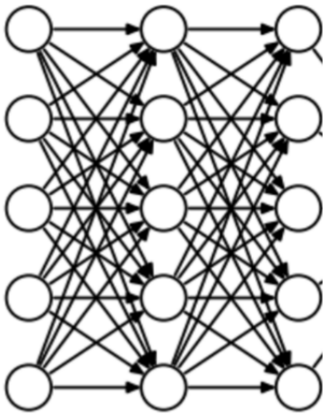

Figure 2: Two-layer fully connected network

3.2.2 Convolutions

To reduce the size of parameters, we can use the idea of sharing weights. Convolution is one of the ways to achieve this by moving a filter (kernel) of smaller size around an input data of larger size and get an output by performing some computation between the filter and the input data the filter covers when it moves around. For example, in Figure 3, a filter of size $3 * 3$ is applied on an input data of size $5 * 5$. The computation performed is a element-wise multiplication between the filer and the covered section of input followed by a summation. Therefore, the output would be 93 on the right-hand size. This convolution is actually called a box filter that gets the average pixel value of every $3 * 3$ block of the input data. If the input data is an image, it will have the effect of blurring.

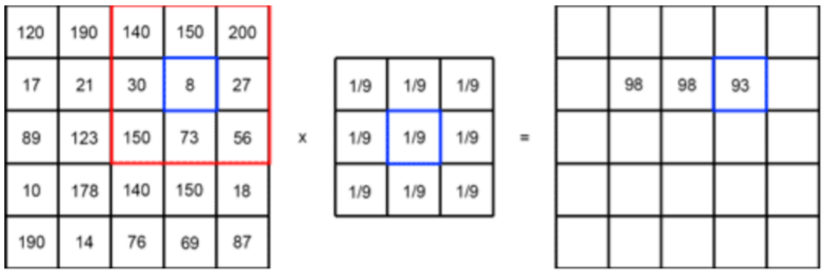

Figure 3: Convolution with box filter

One thing to note is that on the edges, the filter could not fully align with the input data. So it would be impossible to perform the same computation as before. There are multiply ways to deal with this issue. One way is to pad the input data so that the output could have the same shape as the input. Another way is to ignore values on the edges and the output would have a smaller size. Moreover, when the filter is moving around the input data, it can take different size of stride. This is the number of steps the filter take for each move and taking a stride larger than 1 would also result in reducing the output size. In Figure 4, the filter has size $3 * 3$, the input has size $7 * 7$ and the stride is 2 and there is no padding. So, the output would be $3 * 3$ and can be computed by $(input\_width + 2 * padding - filter\_width)/stride + 1 = (7 + 0 - 3)/2 + 1 = 3$.

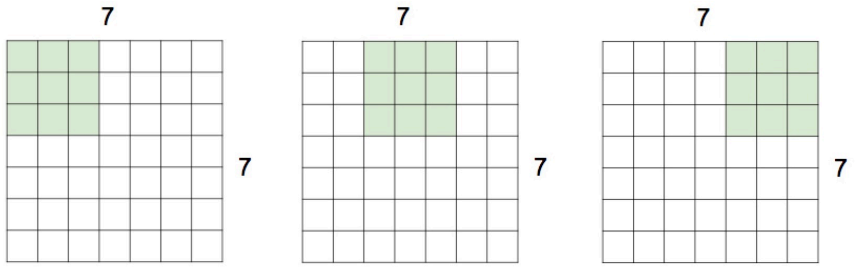

Figure 4: Convolution with stride 2 [2]

The trainable parameter **w** in a convolution layer would be the filters and the bias. each filter has a size of $width * height * depth$, and usually, multiple filters will be used on each layer of data so the size of the filters should be multiplied by the filter count as well. The size of the bias would be the same size of each output layer.

3.2.3 ReLu

ReLu is a function that assign 0 to any value in the input that is less than 0. So it won't affect the size of the input and it won't have any trainable parameters. The function representation for ReLu is $max\{0, x\}$.

3.2.4 Maximum/Average pooling

Maximum/Average pooling is also a convolution computation. But this time, the function of the filter is just taking the maximum or the average of the input area it covers. Figure 5 shows a more intuitive example of max pooling. The filter is of size $2 * 2$ and the stride is size 2.

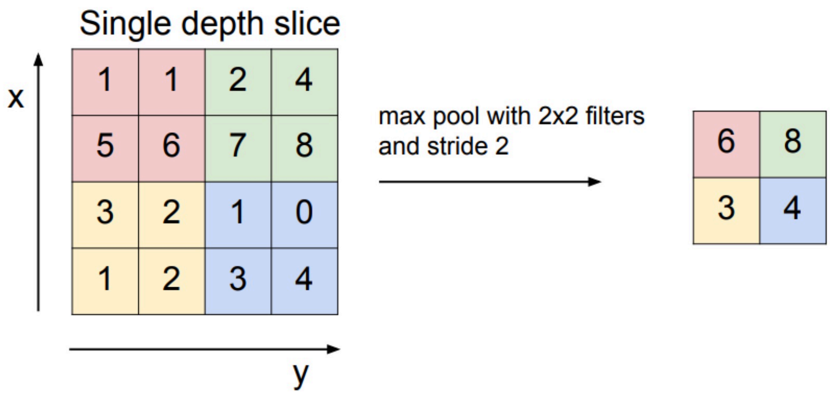

Figure 5: Max pooling layer

The idea of average pooling is exactly the same as max pooling. The only difference is that instead of taking the maximum, we will compute the average and use that as output. So, in Figure 5, the blue area on the bottom right would have value $(1 + 0 + 3 + 4)/4 = 2$ instead of 4.

Note that in a pooling layer, the stride is usually more than 1. This is because we want to substantially reduce the size of the input by applying pooling. In the above example, we reduced the size by a factor of 2 in each dimension. Polling layers do not have trainable parameters since itself does not provide any new information to the network.

3.2.5 Softmax layer

The softmax layer translate logits (scores) to probabilities. So it can take in a vector of logits and output a vector of probabilities. The function that can be used to describe softmax operation is:

$$softmax_i(x) = \frac{\exp x_i}{\sum_j \exp x_j} \tag{5}$$

The softmax function has a nice property that the output vector will sum up to 1, and each output would be in the range of [0,1]. So, it is reasonable to view them as probabilities. Softmax layer also has no trainable parameter.

3.2.6 Dropout layer

The idea of dropout layer is that in each forward pass, we randomly set some activations to zero. As shown in Figure 6, The nodes with x is where dropout happens. Dropout can help prevent co-adaptation of features and reduce overfitting. Another interpretation is that dropout is training a large ensemble of models that share parameters since it randomly silence some parameters every time. [1] The probability of dropping is a hyperparameter, and is commonly set to 0.5. Since this value is usually picked rather than trained, we could think of dropout layer as having no trainable parameters.

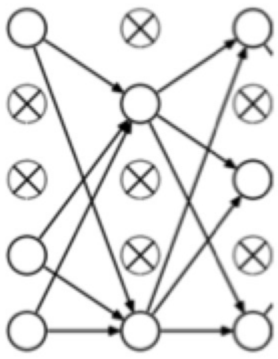

Figure 6: Dropout layer

4 Example Function Architectures

Now that we have looked at many individual functions (layers), we can look at some examples that put them together to build networks.

4.1 LeNet

The first example is an early deep net created by Yann LeCun in 1998 named LeNet. Figure 7 shows the basic architecture of LeNet. From the image input, there is a convolution layer, a pooling layer, another convolution layer, another pooling layer, and finally a fully connected layer. If we look at the dimensions, we can see that for the first convolution layer, 4 filter are used to get 4 feature maps. Then pooling is used to down-sample the feature maps to get 4 smaller feature maps. From here, we now use a 3-dimensional filter with depth 4 to perform convolution. This time, 6 3-dimensional filters are used and we get 6 feature maps. The size of the 6 feature maps is reduced by applying another pooling on them, and finally, the 6 feature maps go through a fully connected layer with 4 outputs. Each output corresponds to a different class.

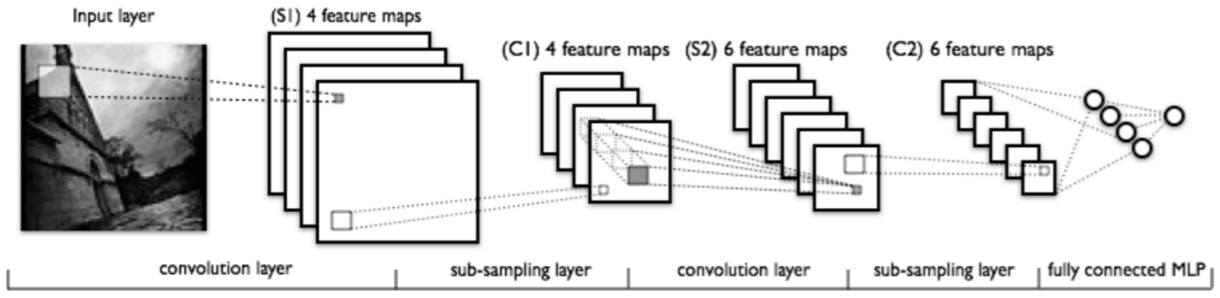

Figure 7: Architecture of LeNet

A more common version of LeNet (LeNet-5) is shown in Figure 8. The basic constructor is exactly the same as before, with increased number of feature maps in the middle. There are also more than one fully connected layers in the end, and the output has 10 different classes because this network is used to recognize handwritten digits. More details can be found in Yann LeCun's paper. [4]

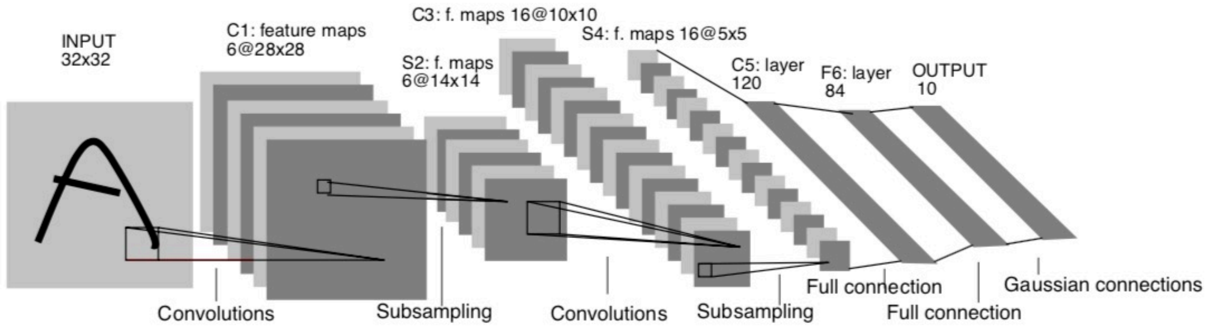

Figure 8: Architecture of LeNet-5

4.2 AlexNet

The next example is AlexNet. This is the first deep neural network that achieved a revolutionary high accuracy in the ImageNet challenge, a large scale visual recognition challenge that evaluates algorithms for object detection and image claasification at large scale. The Figure 9 shows a general structure of AlexNet, which consists of convolution layer, max pooling, another convolution layer, another max pooling, three convolution layers, another max pooling, and finally, three fully connected layers. The actual network is more complicated and uses other layers such ReLu, dropout layer, normalization layer and so on. More details can be found in this paper [3]. The reason that it has a size of 1000 as output is that the ImageNet dataset has 1000 classes. AlexNet utilized vectorized computation and used GPUs to train the network. The reason that it has two parts as shown in Figure 9 is that the GPUs they used did not have enough memory to hold all the data and parameters so they have to spread the network across two GPUs.

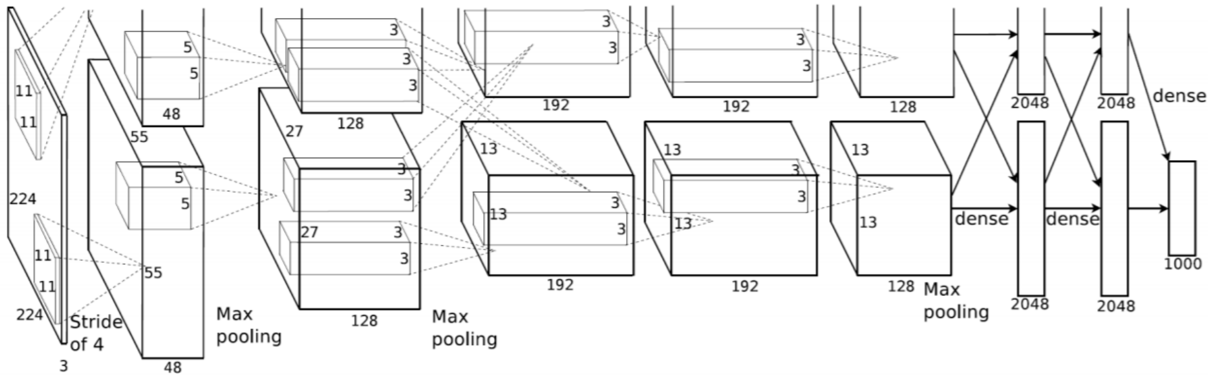

Figure 9: Architecture of AlexNet

4.3 Another deep net

Finally, we have another deep net example. This example is very similar to LeNet. As shown in Figure 10, it consists of a convolution layer, a pooling layer, another convolution layer, another pooling layer, and finally two fully connected layers. It has an output of size 4, which corresponds to four different types of objects. As shown in this example, the input image is some boats and the outputs indicate that the probability of this image being a boat is 0.94.

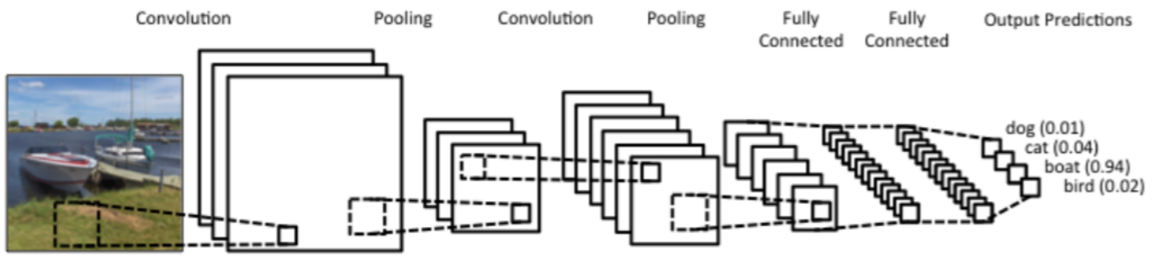

Figure 10: Architecture of another deep net example

All the above examples are similar in the sense that the spatial resolution of the inputs are decreased gradually by pooling and the number of channels are increased by having more filters.

These nets are structurally simple in that a layer's output is used as input for the next layer. But this is not required in building deep networks.

5 Deep Net Training

To perform training on deep net, what we want to do is to minimize the cost function, which we can get from our generalized framework by taking $\epsilon = 1$ and $L(y^{(i)}, \hat{y}) = 0$. So the function we are minimizing is:

$$\min_{\mathbf{w}} \frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}} \ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \tag{6}$$

This is often referred to as maximizing the regularized cross entropy:

$$\max_{\mathbf{w}} -\frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}} \sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \tag{7}$$

So the question is how do we get from the equation 6 to equation 7. First of all, we should know what is cross entropy. It is a way to transform score values to a probability value between [0, 1] so that we can compare the probability of each label $\hat{y}$ being the classification label for a certain data. The formula for cross entropy is:

$$p(\hat{y}|x) = \frac{\exp F(\mathbf{w}, x, \hat{y})}{\sum_{\tilde{y}} \exp F(\mathbf{w}, x, \tilde{y})} \tag{8}$$

Then, take a look at $p_{GT}^{(i)}(\hat{y})$. Since it is ground truth, this value is 1 when $\hat{y} == y^{(i)}$ and 0 when $\hat{y} \neq y^{(i)}$. Putting the above information together, we now get:

$$\max_{\mathbf{w}} -\frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \quad \text{with} \begin{cases} p_{GT}^{(i)}(\hat{y}) = \delta(\hat{y} = y(i)) \\ p(\hat{y}|x) \propto \exp F(\mathbf{w}, x, \hat{y}) \end{cases} \tag{9}$$

In equation 9, $\delta(condition)$ is just a function that outputs 1 when the condition inside is true and outputs 0 when it is false. With this information, we can start to prove that equation 6 and equation 7 are equivalent.

Proof: First, we can deduct this:

$$\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x) = \ln p(y^{(i)}|x) \tag{10}$$

Look back to equation 6, if we take out part of this equation and rewrite it we can get:

$$\begin{aligned} &\ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \\ =&\ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - \ln \exp F(\mathbf{w}, x^{(i)}, y^{(i)}) \\ =&\ln \frac{\sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y})}{\exp F(\mathbf{w}, x^{(i)}, y^{(i)})} = -\ln \frac{\exp F(\mathbf{w}, x^{(i)}, y^{(i)})}{\sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y})} \\ =&-\ln p(y^{(i)}|x^{(i)}) = -\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \end{aligned} \tag{11}$$

Now, from equation 11 to equation 6 and equation 7, it is just summing up all data points $x^{(i)}$ and adding the regularization $\max_{\mathbf{w}} -\frac{C}{2}$, and we can get:

$$\begin{aligned} &\frac{C}{2}\|\mathbf{w}\|_2^2 + \sum_{i\in\mathcal{D}} \ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \\ =&\frac{C}{2}\|\mathbf{w}\|_2^2 - \sum_{i\in\mathcal{D}}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \end{aligned} \tag{12}$$

Finding the minimum of left-hand side of equation 12 is just the same as finding the maximum of the negative of the right-hand side, which proves that equation 6 and equation 7 are indeed equivalent.

Note that C in this equation is the regularization constant (weight decay) that helps to reduce overfitting.

6 Deep Learning with Frameworks

Cross entropy loss can be easily computed in deep learning frameworks such as PyTorch:

$$\min_{\mathbf{w}} \underbrace{\frac{C}{2}\|\mathbf{w}\|_2^2}_{\text{weight decay}} \underbrace{-\sum_{i\in\mathcal{D}}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)})}_{\text{torch.nn.CrossEntropyLoss}(gt, F)} \tag{13}$$

Although frameworks are really helpful when we are constructing machine learning models, we need to make sure that we know what we are doing. When using frameworks, the two main choices we make are the composite function $F(\mathbf{w}, x, y)$ and the appropriate loss function. we should know the dimensions of the inputs, weights and outputs. If we are using some build-in function interfaces from a framework, we need to refer to the documentations and get the information of the function parameters.

In the lecture, we took a look at these loss functions in PyTorch:

- CrossEntropyLoss

  ```
  loss(x, class) = -log(exp(x[class]) / (\sum_j exp(x[j])))
                 = -x[class] + log(\sum_j exp(x[j]))
  ```

  Note that in this function, the x is the F we had earlier, which is the logits (scores) rather than the probability. If we provide the probability as input, we will get double exponentiation.

- NLLLoss (Negative Log Likelihood Loss)

  ```
  loss(x, class) = -x[class]
  ```

  For Negative Log Likelihood Loss, x is not F any more. It is now the log of softmax. The reason for the function to require the log softmax instead of softmax is that having log could achieve numerical robustness by using the 'log-sum-exp trick'.
  If we take a look at Log Softmax:

  $$f_i(x) = \log \frac{\exp x_i}{\sum_j \exp x_j} \tag{14}$$

  We can see that if x_j is a very large number, $\exp x_j$ could easily overflow. By adding a log, we now have:

  $$\log \sum_j \exp x_j = c + \log \sum_j \exp(x_j - c) \quad \text{with} \quad c \in \mathbb{R} \tag{15}$$

  If we choose c to be $\max_j x_j$, $(x_j - c)$ will always be a negative value, $\exp(x_j - c)$ will always be smaller than 1, log of a value smaller than 1 would not cause problem, and finally we just add $\max_j x_j$ to get the final value.

- MSELoss

  ```
  loss(x, y) = 1/n \sum_i |x_i - y_i|^2
  ```

- BCELoss (Binary Cross Entropy Loss)

  ```
  loss(o, t) = -1/n \sum_i(t[i]*log(o[i])+(1-t[i])*log(1-o[i]))
  ```

  This loss function is used for binary classification. Here, o should be a probability since it can only take value in $[0, 1]$ to make log legitimate and t can only be value 1 or 0. In fact, the o here is simply the sigmoid of data points.

- BCEWithLogitsLoss (Binary Cross Entropy With Logits Loss)

  ```
  loss(o, t) = -1/n \sum_i(t[i]*log(sigmoid(o[i]))
                          +(1-t[i])*log(1-sigmoid(o[i])))
  ```

  Now, o is no longer a probability. It is now the logits and will be applied a sigmoid by the function itself. This method is numerically more stable than the BCELoss.

- L1Loss

- KLDivLoss

In the end, we looked at an example network written in PyTorch.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

In this example, we can get some information by looking at the code. We can tell taht the network consists of two convolution layers and three fully connected layers. The filter size is $5 * 5$ for both convolution layers. The first layer has 1 input channel and 6 output channels, and the second layer has 6 input channels and 16 output channels. Then we have the three fully connected layers with input and output dimension showed as the parameters for each layer.

The forward function is our function F as described above. It takes in the input x and applied convolution layer1, relu, pooling, convolution layer2, relu, another pooling, fully connected layer1, relu, fully connect layer2, relu and finally fully connected layer3 to get the output.

From here, we can also deduct the input size of x by computing backwards. What we know is that before x was fed to the first fully connected layer, it had 16 channels with a size of $5 * 5$. The pooling would reduce the size by a factor of 2 on each dimension. So, before the second pooling, the input size was $10 * 10$. Relu doesn't change size, so we can just ignore it. Since the filter size was $5 * 5$, the stride size was 1, and we don't have any padding, we can get the input size before the second convolution layer using $(input\_width + 2 * padding - filter\_width)/stride + 1 = 10$, and get $input\_width = (10 - 1) * 1 + 5 = 14$. Then, we can just do the same operations and get the original input width by $original\_input\_width = (14 * 2 - 1) * 1 + 5 = 32$. Now we know the original input was size $32 * 32$.

References

[1] Cnn architectures. http://cs231n.stanford.edu/slides/2018/cs231n$_2018_lecture09.pdf$.

[2] Convolutional neutral network. http://cs231n.stanford.edu/slides/2018/cs231n$_2018_lecture05.pdf$.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, November 1998.