University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 11: October 4

Lecturer: Alexander Schwing Scribe: Zhenfeng Chen

1 Notation

Notation	Meaning
C	Weight decay (regularization constant)
$\mathbf{w}, \mathbf{W}$	Weight
$\hat{y}$	Ground truth labels
$L(y^{(i)}, \hat{y})$	Loss
$F(\mathbf{w}, x, y)$	General function

Table 1: Notation Definition

2 Recap

The recap section covers the content in **Lecture 10 Deep Net (layers)**.

2.1 General Framework

In the previous lecture, we introduce a framework that can be used to describe objective function in a general way:

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

The limitation of the framework is that $\mathbf{w}^T \psi(x, y)$ is the linear combination of the parameters $\mathbf{w}$ and feature space $\psi(x, y)$. The model can only be learned linearly. To fix this issue, we should include non-linearity into the framework. Use kernels is one way to solve it. Another way is that we can replace $\mathbf{w}^T \psi(x, y)$ with a general function $F(\mathbf{w}, x, y) \in \mathbb{R}$. After replacing $\mathbf{w}^T \psi(x, y)$ by $F(\mathbf{w}, x, y)$, we have

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

As mentioned in the previous lecture, logistic regression can be written in a form of

$$\min_{\mathbf{w}} \frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in \mathcal{D}} \left(\log(1 + \exp(-y^{(i)} \mathbf{w}^T \phi(x^{(i)}))) \right) \tag{3}$$

Similarly, SVM for binary classification can be written as

$$\min_{\mathbf{w}} \frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in \mathcal{D}} \left(\max\{0, 1 - y^{(i)} \mathbf{w}^T \phi(x^{(i)})\} \right) \tag{4}$$

For deep net, $F(\mathbf{w}, x, y) \in \mathbb{R}$ is formed by a group of differentiable composite functions:

$$F(\mathbf{w}, x, y) = f_1(\mathbf{w}_1, y, f_2(\mathbf{w}_2, f_3(\mathbf{w}_3(...f_n(\mathbf{w}_n, x)...))) \in \mathbb{R} \tag{5}$$

To perform inference using the model, we do:

$$y^* = \operatorname{argmax} F(\mathbf{w}, x, y) \tag{6}$$

To solve the problem, we need to plug in the ground truth $\hat{y}$ to find the maximal. It is also known as exhaustive search over all the classes. Gradient descent cannot be applied to this problem, because we are searching over discrete space instead of continuous space.

2.2 Layer

To build a deep net, there are some options for individual functions that can be used in each layer.

2.2.1 Fully Connected Layer

A fully connected layer can be written as:

$$y = \mathbf{W}x + \mathbf{b}. \tag{7}$$

Weight $\mathbf{W}$ and bias $\mathbf{b}$ are trainable parameters. The limitation of using fully connected layer is that we may have too many parameters to store. For example, for a layer with 2,048 nodes connected to another layer with 2,048 nodes, we will have $2,048 \times 2,048 = 4,194,304$ parameters to store in the model. For an image with 256×256 pixels, the size of parameters becomes huge ($2^3 2$). We should come up with a way to reduce the size of parameters. Sharing weights is one of the ideas.

2.2.2 Convolutions

One solution to use fewer parameters in the model is that, we can make the parameters be able to share. For example, in convolutional networks, a kernel/filter can be applied to features at different locations. Therefore, convolution layer can be used to build a deeper network with few parameters. With the same image of size 256×256, applying 5×5 convolution operation will only require 25 trainable parameters.

2.2.3 ReLU

ReLU(Rectified linear units) is usually used as activation function that can be written as:

$$f(x) = max(0, x). \tag{8}$$

A modified version of ReLU function called **Leaky ReLU** can be written as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \tag{9}$$

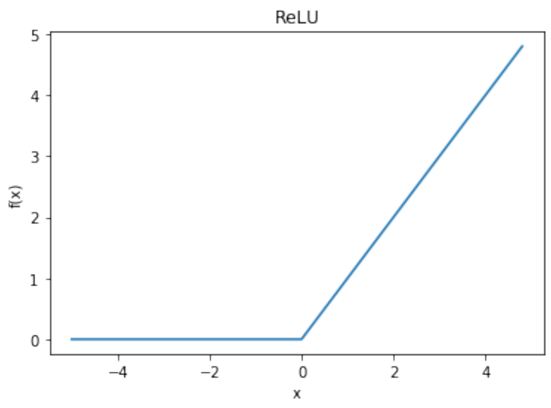

Figure 1: ReLU

Leaky ReLU will provide a small gradient if the unit is not active. It helps solve the vanishing

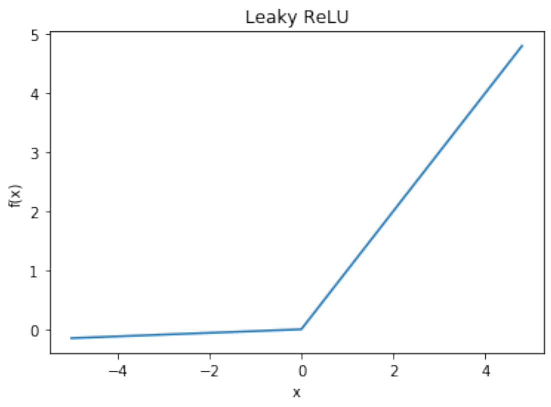

Figure 2: Leaky ReLU

gradient problem.

2.2.4 Pooling

Pooling is an efficient way to reduce the number of parameters in the model. It usually comes after the convolutional layer. Pooling can down-sample the features and help generalization of the model. The reason why we can do pooling is that, we care about the existence of the object instead of the location of the object in an image.

Two types of pooling layers were introduced in the class.

- Max Pooling

  Apply a max pooling filter with a stride, as shown in fig 3.

- Average Pooling

  Instead of output the max value within a kernel, average pooling will output the average value over all the values in the kernel.

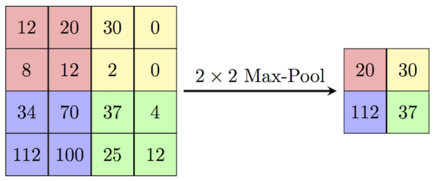

Figure 3: Maxpooling (source: wiki)

In practice, there is no a rule for choosing max pooling or average pooling in the deep net. Personally, I preferred to use max pooling because it helps to capture the extreme features, e.g. corners and edges.

2.2.5 Softmax

Softmax function is defined as

$$p(\hat{y}|x) = \frac{\exp F(\mathbf{w}, x, \hat{y})}{\sum_{\tilde{y}} \exp F(\mathbf{w}, x, \tilde{y})}. \tag{10}$$

Using softmax has several advantages. It avoids 'negative probability' by using exponential.

For example, given

$$F(\mathbf{w}, x, 1) = 11, F(\mathbf{w}, x, 2) = 7, F(\mathbf{w}, x, 3) = 3, F(\mathbf{w}, x, 4) = -2,$$

$$P(y = 4|x, \mathbf{w}) = \frac{\exp F(\mathbf{w}, x, 4)}{\exp\left(F(\mathbf{w}, x, 1) + F(\mathbf{w}, x, 2) + F(\mathbf{w}, x, 3) + F(\mathbf{w}, x, 4)\right)} > 0$$

The derivative of softmax function $p_j = \frac{\exp x_j}{\sum_k \exp x_k}$ is

$$\frac{\partial p_j}{\partial x_j} = p_i \left(\mathbb{1}(i = j) - p_j\right). \tag{11}$$

Relate to the cross entropy, we can calculate the derivative of cross entropy with softmax.

$$\begin{aligned}\frac{\partial L}{\partial x_i} &= -\sum_k y_k \frac{\partial \log(p_k)}{\partial x_i} \\ &= p_i - y_i\end{aligned}$$

A detailed proof can be found at https://deepnotes.io/softmax-crossentropy.

2.2.6 Dropout

Dropout layer is to randomly inactive some nodes in each iteration, see figure 4.

2.3 Loss Functions

Loss function is a measure of performance of model in the prediction.

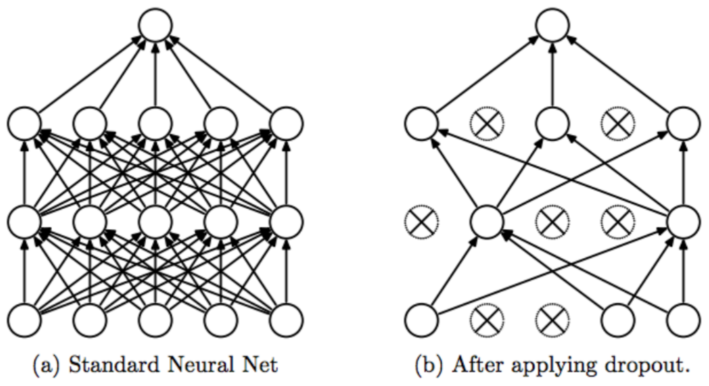

Figure 4: Dropout (source: [1])

- CrossEntropyLoss

$$
\begin{aligned}
loss(x, class) &= -\log \frac{\exp(x[class])}{\sum_j \exp(x[j])} \\
&= -x[class] + \log \left(\sum_j \exp(x[j]) \right)
\end{aligned}
$$

- NLLLoss

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

  To avoid overflow issue, the NLL loss is intended to be used in a combination with log softmax for its numerical robustness because of the 'log-sum-exp trick'. Log softmax can be written as

$$
\begin{aligned}
f_i(x) &= \log \frac{\exp x_i}{\sum_j \exp x_j} \\
&= \log \exp x_i - \log \sum_j \exp x_j \\
&= x_i - c - \log \sum_j \exp(x_j - c)
\end{aligned}
$$

  When $x_i = \max_j(x_j)$ and $c = x_i$, $f_i(x) = 0 - 0 = 0$. Otherwise, $f_i(x) < 0$.

- MSELoss

$$loss(x, y) = \frac{1}{n} \sum_i \mid x_i - y_i \mid^2$$

- BCELoss

$$loss(x, y) = -\frac{1}{n} \sum_i \left(y_i \log(x_i) + (1 - y_i) \log(1 - x_i)\right)$$

  $x_i \in (0, 1)$ because of $\log(x_i)$ and $\log(1 - x_i)$. Therefore x_i should be probability.

- BCEWithLogitsLoss

$$loss(x, y) = -\frac{1}{n} \sum_i (y_i \log(\sigma(x_i)) + (1 - y_i) \log(1 - \sigma(x_i)))$$

  where the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$.

- L1Loss
- KLDivLoss

The usage of loss functions in detail can be found in Pytorch documentation.

2.4 Example: build a deep net

A detailed documentation for implement a deep net can be found here.
To build the deep net in the following example, we will use

- **nn.Conv2d**: a 2D convolution
- **nn.Linear**: dense layer
- **F.max_pool2d/nn.MaxPool2d**: a 2D max pooling
- **F.relu/nn.ReLU**: ReLU function
- **view**: flatten the feature matrix

Besides the layer/method we mentioned before, dropout layer and batch normalization are usually added in the deep net for regularization.

2.4.1 Source code

```
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
```

2.4.2 Compute input size

By default, in nn.Conv2d(), stride = 1, and padding = 0. In max_pool2d, stride = kernel_size, and padding = 0. When there is no dilation, the output size of convolutional layer can be computed as:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding[0] - kernel\_size[0]}{stride[0]} + 1 \right\rfloor \tag{12}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times padding[1] - kernel\_size[1]}{stride[1]} + 1 \right\rfloor \tag{13}$$

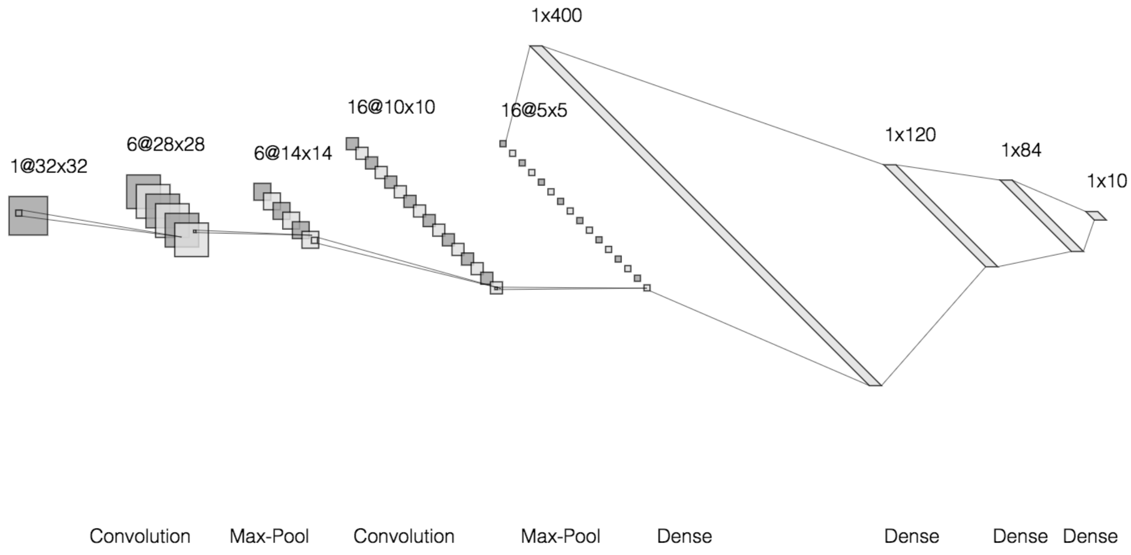

Figure 5: Deep Net

Similarly, the output size of max pooling layer without dilation can be computed as:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding[0] - kernel\_size[0]}{stride[0]} + 1 \right\rfloor \tag{14}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times padding[1] - kernel\_size[1]}{stride[1]} + 1 \right\rfloor \tag{15}$$

We notice that it is the same as what we did in compute output size for convolutional layers.

3 Backpropagation

The objective function for deep net can be written as

$$\min_{\mathbf{w}} \frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in \mathcal{D}} \left(\ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \right). \tag{16}$$

which can be referred to as maximizing the cross entropy.

$$\max_{\mathbf{w}} -\frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in \mathcal{D}} \sum_{\hat{y}} \delta(\hat{y} = y^{(i)}) \ln p(\hat{y}|x^{(i)}) \tag{17}$$

To optimize this, we use stochastic gradient descent with momentum. The gradient of the objective function is

$$C\mathbf{w} + \sum_{i \in \mathcal{D}} \sum_{\hat{y}} \left(p(\hat{y}|x^{(i)}) - \delta(\hat{y} = y^{(i)}) \right) \frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}}. \tag{18}$$

$p(\hat{y}|x)$ can be computed using soft-max

$$p(\hat{y}|x) = \frac{\exp F(\mathbf{w}, x, \hat{y})}{\sum_y \exp F(\mathbf{w}, x, y)}$$

while $\frac{\partial F(\mathbf{w}, x^{(i)}, \hat{y})}{\partial \mathbf{w}}$ can be computed via backpropagation.
Computing gradient of deep nets is expensive. We would like to use backpropagation to avoid repeated computation. To apply backpropagation, we will use chain rule, which is defined as:

$$\frac{\partial f(g(w))}{\partial w} = \frac{\partial f}{\partial g} \frac{\partial g(w)}{\partial w} \tag{19}$$

or

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \tag{20}$$

We will use the example in the lecture to show how backpropagation works. In the lecture, we have

$$F(\mathbf{w}, x, y) = f_1(\mathbf{w}_1, y, f_2(\mathbf{w}_2, f_3(\mathbf{w}_3, x))).$$

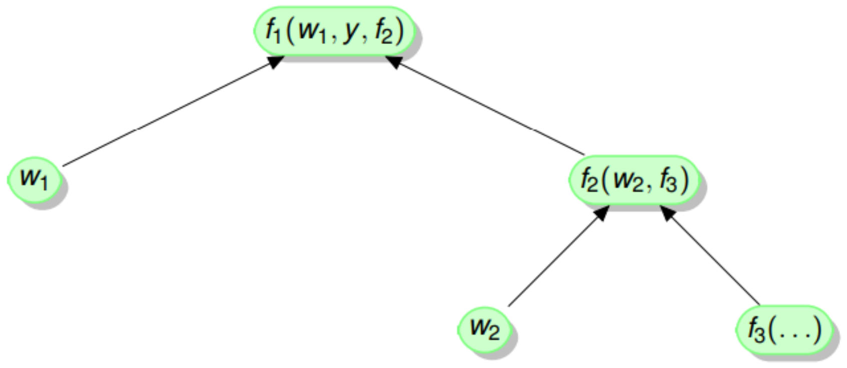

Figure 6: Function $F(\mathbf{w}, x, y)$

To compute the gradient of parameters, we have

$$\frac{\partial F(\mathbf{w}, x, y)}{\partial w_3} = \frac{\partial f_1}{\partial x_1} \frac{\partial x_1}{\partial x_2} \frac{\partial x_2}{\partial \mathbf{w}_3}$$

$$\frac{\partial F(\mathbf{w}, x, y)}{\partial w_2} = \frac{\partial f_1}{\partial x_1} \frac{\partial x_1}{\partial \mathbf{w}_2}$$

where

$$x_2 = f_3(\mathbf{w}_3, x)$$

and

$$x_1 = f_2(\mathbf{w}_2, x_2).$$

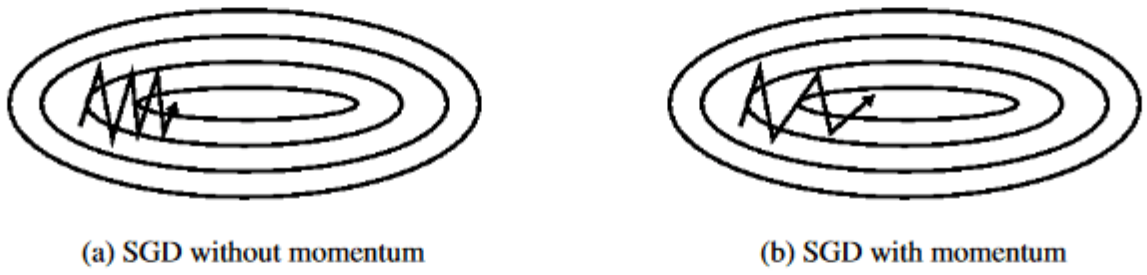

Figure 7: Momentum

We noticed that $\frac{\partial f_1}{\partial x_1}$ was used twice in the computation. To avoid repeated computation in the training/learning, we should store the intermediate results in the forward pass. Some functions, usually referring to X + activation (ReLU, sigmoid, tanh) can be combined together when we store the intermediate result. Because in the activation function, we don't have parameters to learn. Therefore we don't do backpropagation over the activation functions. When we use the model for inference, the intermediate results can be thrown away.

One thing we should notice is that the loss function for neural net is non-convex, i.e. global optimum is not guaranteed.

4 Questions

- What is momentum?

  In the slide, it is mentioned that we can use stochastic gradient descent with momentum to optimize our loss function. Given two images in figure 7, we notice that momentum makes SGD converge faster towards the local optimum. In class, we use an example to demonstrate momentum. Imagine we put a ball on the hill and let it roll down, the ball will move faster and faster along one direction, which is the gradient point.

- Can deep net achieve 100% accuracy?

  It depends on the task. Ideally, given infinite time and dataset, deep nets can match any functions. However, in most of the real world application, we have limited computing resources and data. What we can do is to improve the accuracy and model generalization as much as possible. It is also known as a trade-off between bias and variance.

References

[1] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014.