University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 11: October 4

Lecturer: Alexander Schwing Scribe: Junfeng Guan

Overview

In lecture 10 we introduced Deep Nets and Deep Learning. We looked into functions/layers, as well as architectures of function through examples such as LeNet and AlexNet. Following on that, this lecture we will focus on deep net training, including backpropagation, initialization, and choosing loss functions and composite functions. We will also discuss features of deep nets, such as its pros and cons, and the reason for its rapid popularization over the last decades. After this lecture, student should try to implement their first deep nets.

Recap

0.1 General framework

Our earlier framework is:

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

It has limitations because of the linearity in its feature space $\psi(x, y)$. To fix this issue, we introduce kernels, replacing $\mathbf{w}^T \psi(x, y)$ with a general function $F(\mathbf{w}, x, y) \in \mathbb{R}$. As a result, we got the general framework:

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

Notice that, even with the general framework, we are still learning a model linear in the parameters $\mathbf{w}$.

Based on the selection of function $F(\mathbf{w}, x, y) \in \mathbb{R}$, constants C and ϵ, as well as the task loss L, the general framework can represents:

- **Logistic regression**
- **Binary SVM**
- **Multinomial logistic regression**
- **Multiclass SVM**
- **Deep Learning**

0.2 Deep Learning

The above generalized framework also adapts to solve deep learning problems. When we choose $F(\mathbf{w}, x, y)$ to be differentiable and representable by an acyclic graph as in Figure 1.

$$F(\mathbf{w}, x, y) = f_1(\mathrm{w}_1, y, f_2(\mathrm{w}_2, f_3(...f)n(\mathrm{w}_n, x)...))) \in \mathbb{R}, (y \in \{1...K\})$$

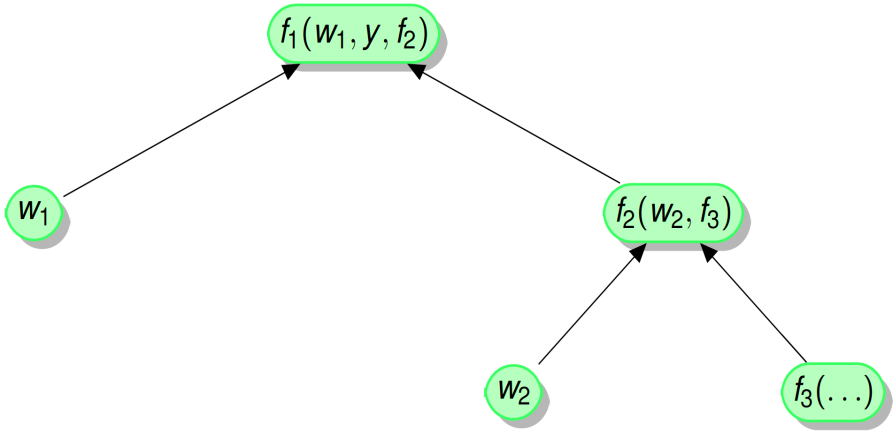

Figure 1: Acyclic graph of F

Here functions/layers f_1, f_2 can be:

- **Fully connected layers**
- **Convolutions**
- **Rectified linear units (ReLU): max{0,x}**
- **Maximum-/Average pooling**
- **Soft-max layer**
- **Dropout**

0.3 Deep net training

Deep net training is to minimize the multiclass logistic loss,

$$\min \mathbf{w} \frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in D} \left(ln \sum_{\hat{y}} expF(\mathbf{w}, x^{(i)}, \hat{(}y)) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \right)$$

which is equivalent to maximizing the regularized cross entropy which we will examine in detail in this lecture.

1 Program

Recall the general framework set up for Multiclass Logistic Loss in recap section above. The loss function used for Deep Learning algorithms is similar to that described there. In particular the loss function is given by the equation below,

The general frame work is often written as maximizing the regularized cross entropy:

$$\min_{\mathbf{w}} -\frac{C}{2}\left\|\mathbf{w}\right\|_2^2 + \sum_{i\in D}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) ln p(\hat{y}|x^{(i)})$$

Since it does not have a closed form solution, we need to optimizes with respect to $\mathbf{w}$ using Stochastic gradient descent (SGD).

1.1 Gradient

The gradient is

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

To compute it numerically we need to compute:

- $p(\hat{y}|x) = \frac{expF(\mathbf{w},x,\hat{y})}{\sum_{\tilde{y}} expF(\mathbf{w},x,\tilde{y})}$ via soft-max which takes logits F as input
- $\frac{\partial F(\mathbf{w},x^{(i)},\hat{y})}{\partial \mathbf{w}}$ via backpropagation

1.2 Backpropagation

Suppose we have a differentiable composite function $F(\mathbf{w}, x, y)$ as:

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

with activations:

$$\alpha(x) = \begin{cases} x_2 = f_3(\mathrm{w}_3, x) \\ x_1 = f_2(\mathrm{w}_2, x_2) \end{cases}$$

Then we can calculate $\frac{\partial F(\mathbf{w},x,y)}{\partial \mathrm{w}_3}$ and $\frac{\partial F(\mathbf{w},x,y)}{\partial \mathrm{w}_2}$ using the chain rule:

$$\frac{\partial F(\mathbf{w}, x, y)}{\partial \mathrm{w}_3} = \frac{\partial f_1}{\partial x_1}\cdot\frac{\partial x_1}{\partial x_2}\cdot\frac{\partial x_2}{\partial \mathrm{w}_3} = \underbrace{\frac{\partial f_1}{\partial f_2}}\cdot\frac{\partial f_2}{\partial f_3}\cdot\frac{\partial f_3}{\partial \mathrm{w}_3}$$

$$\frac{\partial F(\mathbf{w}, x, y)}{\partial \mathrm{w}_2} = \frac{\partial f_1}{\partial x_1}\cdot\frac{\partial x_1}{\partial \mathrm{w}_2} = \underbrace{\frac{\partial f_1}{\partial f_2}}\cdot\frac{\partial f_2}{\partial \mathrm{w}_2}$$

Notice that in the above example, the computation of $\frac{\partial f_1}{\partial f_2}$ is part of both derivatives so in the traditional forward pass manner it will be repeated twice.

Therefore to avoid such redundant computations, we do backpropagation on an acyclic graph that computes chain rule derivatives in a top-down mannerx. Nodes in this graph are weights, data, and functions. Figure 2 demonstrates the backward pass of the computation. $\frac{\partial f_1}{\partial f_2}$ is computed first and saved. Then we can repeatedly use chain rule to calculate $\frac{\partial F(\mathbf{w},x,y)}{\partial \mathrm{w}_2}$ and $\frac{\partial F(\mathbf{w},x,y)}{\partial \mathrm{w}_3}$ with the saved upper layer gradients (e.g. $\frac{\partial f_1}{\partial f_2}$) to efficiently computes the overall gradient.

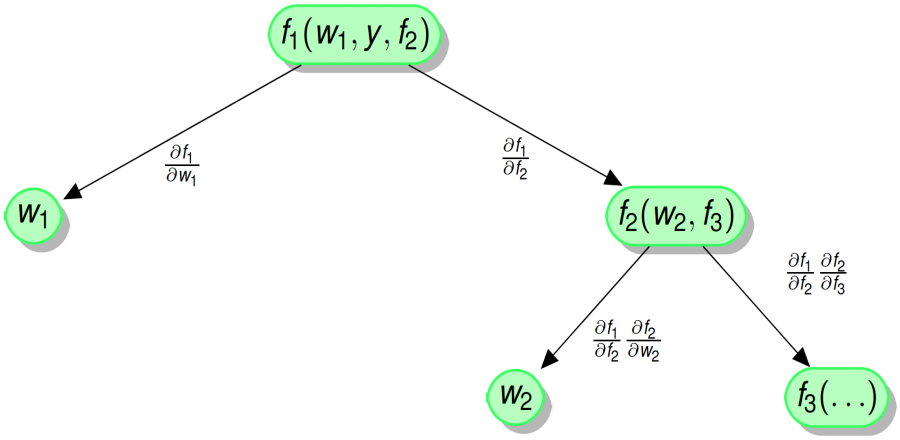

Figure 2: Backpropagation on an acyclic graph

forward pass softmax probabilities compute backward pass on computational graph

1.3 Storing intermediate results

Forward pass only needs the derivative calculated but nothing in the computation, so it is memory-wise efficient. On the contrary, backpropagation needs to keep intermediate results that will be reused. Therefore, training is more computationally and memory-wise demanding. Sometimes, if limited on memory, we should also do recomputation backpropagation.

For Learning and Inference, we should store intermediate result differently to be efficient in memory usage as well. For Inference, we can discard any intermediate result and only pass a single value. On the other hand, for learning, we should store intermediate results for fully connected layer and convolution. Some functions can be combined to reduce intermediate storage.

Difference between activation functions and layers: Layers are operations that store intermediate precisely, while activation functions are in place operations, which do not store intermediate results.

Student should try to implement a simple deep net framework. Starts from scalers instead of tensors or matrices. They can refer to the PyTorchECE544.py example from Lecture 10.

2 Initialization

2.1 Non-convexity

Since $F(\mathbf{w}, x, y)$ is no longer constrained in any form, the loss function is generally no longer convex. In other words, it is not guaranteed to reach the global optimum.

2.2 Initialization

The non-convexity of neural networks also means that initialization of the weights matters, because bad selection of initialization of weights may lead to stuck at suboptimal local minimums. Unfortunately, initialization is not yet a well understood problem in general, but there are a few heuristics commonly used. They should break symmetry and be random. Two of those heuristics are:

Random uniform

$$Uniform\Big(-\frac{1}{\sqrt{fanin}}, \frac{1}{\sqrt{fanin}}\Big)$$

Glorot and Bengio (2010)

$$Uniform\Big(-\sqrt{\frac{6}{fanin + fanout}}, \sqrt{\frac{6}{fanin + fanout}}\Big)$$

3 Features of Deep Nets

3.1 Feature transformation example

From the example of feature transformation example shown in Figure 3, we can tell the difference between layers. Lower layers learn mapping of edges, higher layer (e.g. 2nd layer in the example) learn mappings of parts of the structures (eyes, wheels), while the even higher layer (e.g. 3rd layer in the example) learn mapping of canonical images (faces and objects).

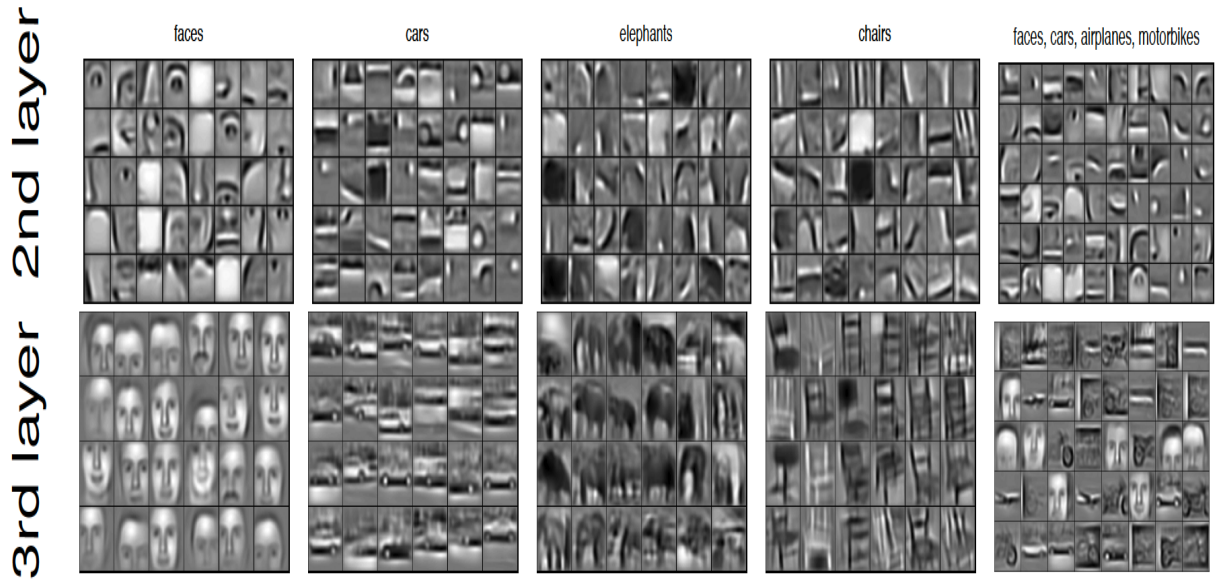

Figure 3: Feature transformations learned by deep networks

Notice that a deep net with a single fully connected layer is equivalent to logistic regression.

3.2 Advantages

Deep nets automatically learn feature space transformations (hierarchical abstractions of data) such that data is easily separable at the output.

3.3 Disadvantages

Compared to the usage of features, deep nets requires both huge computational power (GPUs) and significant amounts of training data. Without either one of these prerequisites, output might suffer.

3.4 Recent popularity

In the past decade we have seen a significant raise in popularity, thanks to the:

- **Improvement in computational resources**
- **Increasing availability of data**
- **Algorithmic advances**
- **Convincing proof of working**

3.5 Algorithmic advances

- **Rectified linear unit (ReLU)** $(max\{0, x\})$ **activation function instead of sigmoid**
  Fixed the vanishing gradient problem such that gradient gets very close to zero for lower layers close to the input
- **Dropout**
  Decorrelates different units so that different features can be learned, and models become more robust
- **Good initialization heuristics**
  Avoids getting stuck at suboptimal local optima due to non-convexity
- **Batch-Normalization during training**
  Normalizes data when training really deep nets by subtracting mean and dividing by standard deviation, which makes networks more robust to bad initialization.

4 Choose Deep Learning packages

When implementing deep net framework, there are two essential choices to make for deep learning packages. They are:

- **Choosing an appropriate loss function**
- **Designing a composite function** $F(\mathbf{w}, x, y)$

Also it is important to know what you are doing, for instance, knowing all the dimensions.

4.1 Loss functions

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

- **MSELoss**
  loss(x, y) = 1/n \sum_i |x_i - y_i|^2
- **BCELoss**
  loss(x, y) = -1/n \sum_i i(t[i]*log(o[i])+(1-t[i])*log(1-o[i]))
- **BCEWithLogitsLoss**
  loss(x, y) = -1/n \sum_i (t[i]*log(sigmoid(o[i]))
  +(1-t[i])*log(1-sigmoid(o[i])))
- **L1Loss**
- **KLDivLoss**

This form of NLLLoss: loss(x, class) = -x[class], is intended to be used together with 'LogSoftmax':

$$f_i(x) = log\frac{expx_i}{\sum_j expx_j}$$

Because it is more robust numerically due to the 'log-sum-exp trick'

$$log\sum_j expx_j = c + log\sum_j exp(x_j - c)$$

4.2 Popular architectures

- **LeNet**
- **AlexNet**
- **VGG (16/19 layers, mostly 3×3 convolutions)**
- **GoogLeNet (inception module) Figure 4**
- **ResNet (residual connections) Figure 5**

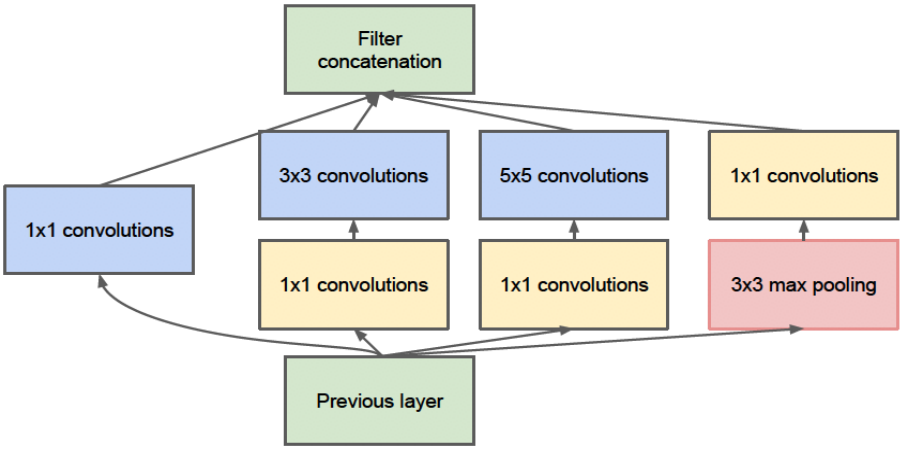

Figure 4: GoogLeNet

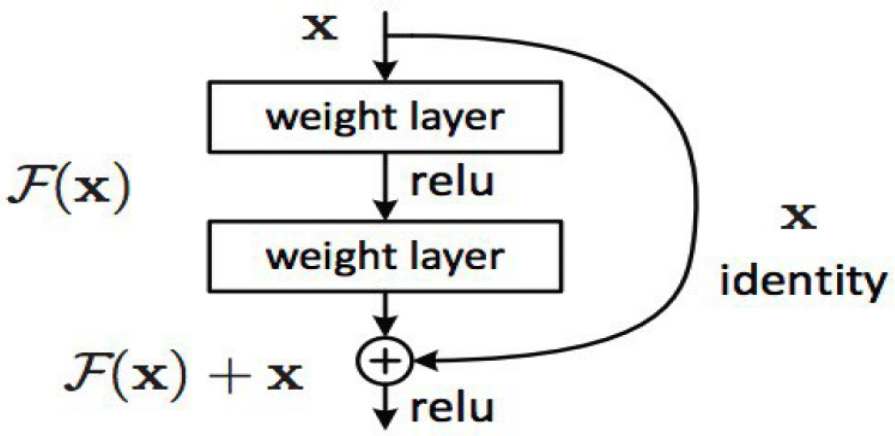

Figure 5: ResNet

5 Imagnet Challenge

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. It has a large dataset of 1.2 million images within 1000 categories. From the top-5 winning algorithms over years shown in Figure 6, we can tell that the sufficient computational resources provided by GPUs enables a significant improvement of detection error rate started from AlexNet. Besides, the dramatic increase of depth of deep net in ResNet from 22 layers to 152 layers powered up another breakthrough. Over years, rectifier has becomes the most popular activation function, since rectified linear unit (ReLU) overcomes sigmoid units in simplifying the optimization.

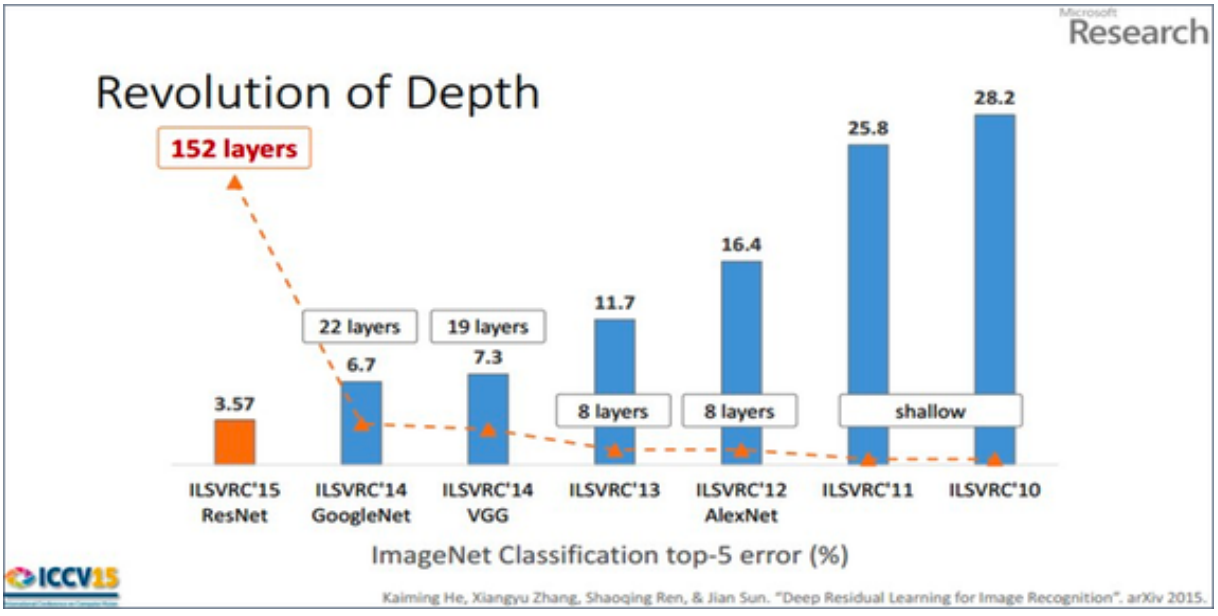

Figure 6: Imagnet Challenge top-5 error rate