University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 11 Backpropagation: October 2

Lecturer: Alexander Schwing Scribe: Yifeng Fan

The goal of this lecture:

- Understanding forward and backward pass.
- Learning about backpropagation.
- Learning about Deep Neural Network.

1 Recap

In this section let us retrospect what we have learned before. Please jump to section 2 directly if you are familiar with previous lectures.

1.1 General model

All the topics of this course are related to machine learning. As shown in wikipedia: "Machine learning is a field of artificial intelligence that uses statistical techniques to give computer systems the ability to "learn" (e.g., progressively improve performance on a specific task) from data". In this way, suppose we have n data samples, where the i^{th} data has value $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ (you can think about $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ are two vectors), we would like to let our computer learns, given an arbitrary input $\boldsymbol{x}$, what is the corresponding prediction of $\boldsymbol{y}$. The basic algorithm is to build a prediction model (or a function) f with parameters $\boldsymbol{w}$, which outputs $f(\boldsymbol{w}, \boldsymbol{x})$ for the input $\boldsymbol{x}$. In order to find the proper choice of $\boldsymbol{w}$, we define a loss function $L = L(\boldsymbol{w})$, where the optimal $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w})$. The data set that used to find $\boldsymbol{w}^*$ is called training data.

1.2 Linear regression and Binary classification (Lecture 2&3)

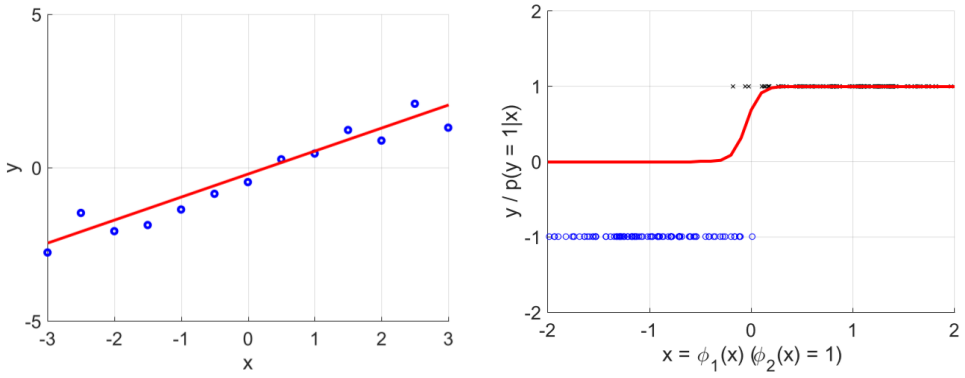
We firstly focus on two simple problems: linear regression and binary classification:

- **Linear regression:** For linear regression, we assume that our data should be well linearly modeled as $y = \boldsymbol{w}^\top \boldsymbol{x} + b$, then our goal is to find a line (actually a plane in the high dimensional data) that cross our data samples with minimal error. Usually the model and the loss function are given as:

$$f(\boldsymbol{w}, \boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b, \quad L(\boldsymbol{w}) = \frac{1}{2} \sum_{i=1}^{n} (y^{(i)} - f(\boldsymbol{w}, \boldsymbol{x}^{(i)}))^2. \tag{1}$$

  Here we use the least square error as the loss function. You can see the 'linear' comes from the linear property of our prediction $f(\boldsymbol{w}, \boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$, later on you will see usually this linear assumption is not feasible for most of problems.

- **Binary classification:** For binary classification, we would like to divide our data into two categories, here we assume for input data $\boldsymbol{x}$, its output class $\hat{y} \in \{-1, 1\}$. Then the probability



(a) Linear regression (b) Binary classification

Figure 1: An example of *left:* linear regression, *right:* binary classification. The blue and black scatter plots are the data samples, and the red line is the output of our method. Please note the difference, that the prediction is a value on a straight line for linear regression, or a class label as blue or black for binary classification.

that $\hat{y} = 1$ is given as:

$$p(\hat{y} = 1|\boldsymbol{x}) = \frac{1}{1 + \exp\left(-\boldsymbol{w}^\top \boldsymbol{x}\right)}. \tag{2}$$

and for $\hat{y} = -1$ we have:

$$p(\hat{y} = -1|\boldsymbol{x}) = 1 - p(\hat{y} = 1|\boldsymbol{x}) = \frac{1}{1 + \exp\left(\boldsymbol{w}^\top \boldsymbol{x}\right)}. \tag{3}$$

Note that these two cases can be combined as:

$$p(\hat{y}|\boldsymbol{x}) = \frac{1}{1 + \exp\left(-\hat{y}\boldsymbol{w}^\top \boldsymbol{x}\right)}. \tag{4}$$

Here we will not illustrate too much on why we use such a form of probability (please refer to Lecture 3). Based on the probability definition, our model is to choose the $\hat{y}$ that maximize the probability as:

$$f(\boldsymbol{w}, \boldsymbol{x}) = \arg\max_{\hat{y} \in \{-1,1\}} p(\hat{y}|\boldsymbol{x}) = \arg\max_{\hat{y} \in \{-1,1\}} \frac{1}{1 + \exp\left(-\hat{y}\boldsymbol{w}^\top \boldsymbol{x}\right)} \tag{5}$$

And the corresponding loss function is to combine the probability for our whole training data set:

$$L(\boldsymbol{w}) = \ln \prod_{i=1}^{n} p(y^{(i)}|\boldsymbol{x}^{(i)}) \tag{6}$$

$$= -\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right). \tag{7}$$

Notice that here we assume our data set is independent identical distributed (i.i.d), which is an usual case in real problems.

Based on the definition of loss function, we could find the optimal $\boldsymbol{w}^*$ either by analytic solution or iterative methods. Usually analytic solutions do not exist and we use iterative methods instead. Please refer to Lecture 4&7 for detailed iterative methods. In Figure. 1 we show an example of linear regression and binary classification.

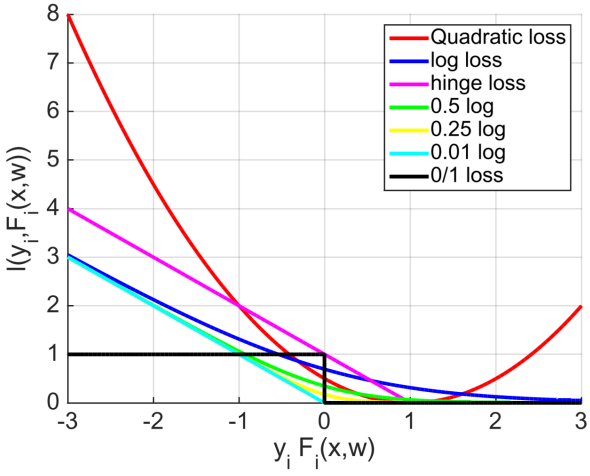


Figure 2: Comparison of different loss functions. Logistic regression uses 'log loss' and SVM uses 'hinge loss'. It is clear that both loss functions are quite similar. Due the space limits we will not illustrate other loss functions.

1.3 Multi-class classification (Lecture 8&9)

So far we have talked about binary classification, but most classification problems involve more than two categories. For example, we hope to classify a bunch of images with more than two classes, which is very common. To address this, we could follow the similar way as in binary classification: suppose there are K categories and for the k^{th} category we define the parameter vector $\boldsymbol{w}_k$ and we concatenate all $\boldsymbol{w}$ as a long vector $\boldsymbol{w} = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k\}$. For input data $\boldsymbol{x}$ and output class $\hat{y}$, the probability $p(\hat{y} = k|\boldsymbol{x})$ is given as:

$$p(\hat{y} = k|\boldsymbol{x}) = \frac{\exp\left(\boldsymbol{w}_k^\top \boldsymbol{x}\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{w}_j^\top \boldsymbol{x}\right)}, \tag{8}$$

and we predict the class of $\boldsymbol{x}$ by maximizing such probability as:

$$f(\boldsymbol{w}, \boldsymbol{x}) = \underset{k\in\{1,\ldots,K\}}{\arg\max}\ p(\hat{y} = k|x) = \underset{k\in\{1,\ldots,K\}}{\arg\max} \frac{\exp\left(\boldsymbol{w}_k^\top \boldsymbol{x}\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{w}_j^\top \boldsymbol{x}\right)}. \tag{9}$$

Moreover, the loss function is defined as:

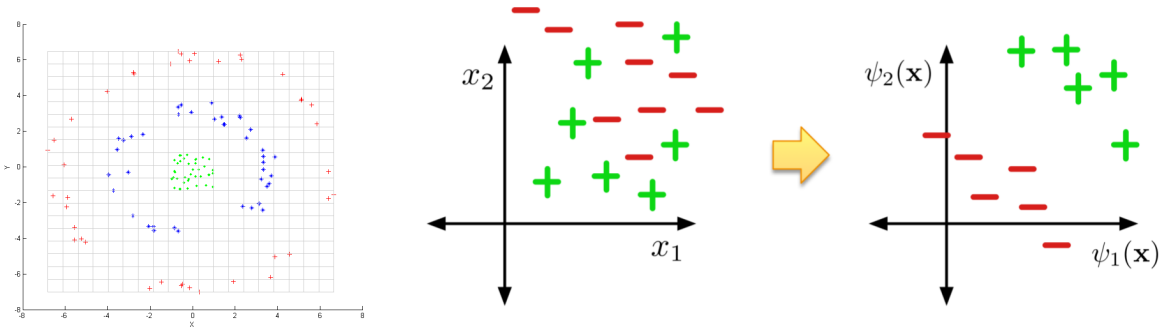
$$L(\boldsymbol{w}) = \ln \prod_{i=1}^{n} p(y^{(i)}|\boldsymbol{x}^{(i)}) \tag{10}$$

$$= \sum_{i=1}^{n} \ln \left[\sum_{\hat{y}=1}^{K} \exp\left(-\hat{y}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right] - \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right). \tag{11}$$

This is the loss function of multi-class logistic regression, and there are other types of loss functions, such as Support Vector Machine (SVM) that uses hinge loss (please refer to Lecture 8). Actually all of these loss functions have similar classification performance, since their asymptotic behavior are the same, as shown in Figure. 2.

1.4 How to handle non-linear function? (Lecture 9&10)

So far, all the methods we have touched are based on linear model, which means all the methods can only applies to the data that can be approximated or classified by linear function. However what if we want to classify non-linear models such as quadratic function or functions with higher orders? The left plot in Figure. 3 displays a non linear classification task.



(a) (b)

Figure 3: *Left:* An example of non-linear classification task, data are divided into 3 classes with different colors (red, blue and green). Figure copied from Wikipedia. *Right:* An illustration of non-linear mapping. Figure copied from CSC321 in University of Toronto.

To handle this problem, we present two possible solutions:

(1)Kernel trick(Lecture 9):

Kernel trick starts from the observation that, for two data points with feature maps $x^{(i)}$ and $x^{(j)}$, all of the linear methods above calculate their affinity by the inner product $\left\langle x^{(i)}, x^{(j)} \right\rangle$. Then kernel trick is to replace such inner product by a function $\kappa(x^{(i)}, x^{(j)})$ as so called kernel. Specifically, replacing inner product with kernel indicates that we map our features to a higher dimensional space V as $v(\boldsymbol{x}^{(i)})$ (usually unknown, but we do not care), which could sometimes solve such non-linear problems. In Lecture 9 we have shown different common kernels.

kernel helps, but it still has two major problems:

- The choice of kernels need to be specified in advance, and this can require a lot of engineering work. In other words, there is still no general instructions for what kind of kernels we should use.
- Although kernel could represent data sample by some high dimensional features, what we do as following is still running the linear models on such high dimensional space. This sometimes fails to solve the problem, since the model is still linear to the parameter $\boldsymbol{w}$.

Therefore a more practical approach is needed to solve this problem.

(2)Neural Network (Lecture 10):

Let us recall the loss function for the multi-class logistic regression in Eq. (11) as:

$$L(\boldsymbol{w}) = \sum_{i=1}^{n} \ln \left[\sum_{\hat{y}=1}^{K} \exp\left(-\hat{y}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right) \right] - \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right). \tag{12}$$

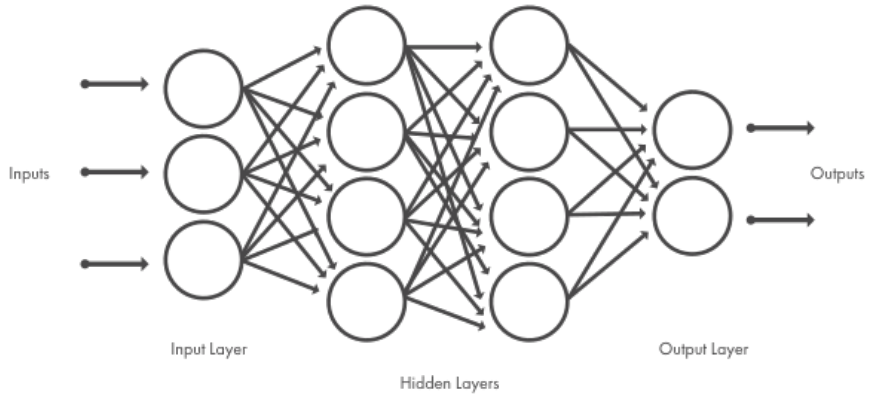


Figure 4: An example of artificial Neural Network with 3 hidden 4-dimensional layers, 3-dimensional input and 2-dimensional output.

If we use kernel trick, we actually replace $\boldsymbol{x}^{(i)}$ by $v(\boldsymbol{x}^{(i)})$ as:

$$L(\boldsymbol{w}) = \sum_{i=1}^{n} \ln \left[\sum_{\hat{y}=1}^{K} \exp\left(-\hat{y}\boldsymbol{w}^\top v(\boldsymbol{x}^{(i)})\right) \right] - \exp\left(-y^{(i)}\boldsymbol{w}^\top v(\boldsymbol{x}^{(i)})\right). \tag{13}$$

In order to avoid the drawbacks of kernel trick, the idea of neural network is to replace the whole part $-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}$ within the exponential function by an arbitrary function $F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, y^{(i)})$ as:

$$L(\boldsymbol{w}) = \sum_{i=1}^{n} \ln \left[\sum_{\hat{y}=1}^{K} \exp\left(F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, \hat{y})\right) \right] - \exp\left(F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, y^{(i)})\right). \tag{14}$$

As a result, neural network can be thought as a more general way of learning non-linear feature mappings, where we input our data $x^{(i)}$ and get the output $F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, y^{(i)})$ for the category $\hat{y}$. Comparing with kernel trick, neural network does not require the specific definition of kernel, also it could be a non-linear function of parameters $\boldsymbol{w}$.

1.5 How to construct a neural network?

Since neural network is inspired by the biological neural networks that constitute animal brains. The artificial neural network has the similar structure. In Figure. 4 we show a simple 2-hidden-layer neural network, where the input and outputs nodes represent input data $\boldsymbol{x}$ and output $F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, y^{(i)})$ correspondingly. They are connected by hidden layers with arbitrary number of nodes.

For each layer, suppose the input is $\boldsymbol{x}$ and output $\boldsymbol{a}$, it can be written as:

$$\boldsymbol{a} = \sigma\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right) \tag{15}$$

where $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{b}$ is the bias. σ is usually a non-linear activation function, actually we have seen it in linear regression as $\sigma(z) = z$, and in Logistic regression as $\sigma(z) = 1/(1 - \exp(-z))$. **A neural network is just a combination of lots of these units**. Each one performs a very simple and stereotyped function, but in aggregate they can do some very useful computations. For a given structure, you can think about neural network as an expression of a range of non-linear functions. What we need is to find the optimal non-linear function that fits our problem, i.e., finding the optimal parameter $\boldsymbol{w}$ based on the training data and loss function.

2 Backpropagation

In section 1 we have shown that neural network could be a powerful tool for representing non-linear functions. Then given the loss function $\mathcal{L}(\boldsymbol{w})$, we need to find the optimal $\boldsymbol{w}^*$ that minimize $\mathcal{L}(\boldsymbol{w})$ by iterative method, which involves computing the derivatives of $\mathcal{L}(\boldsymbol{w})$. This seems easy when L is simple, such as linear regression, but can be horrible for the realistic neural network case, since $\mathcal{L}(\boldsymbol{w})$ could be extremely complicated due to the large number of nodes, weights and depth in the network. In this section we introduce an efficient approach to compute the derivatives: backpropagation.

2.1 A toy example of derivative

To emphasize the difficulty of computing derivatives, we could have a try on computing the derivatives manually. Considering a binary classification problem as following: For simplicity, let us assume we have univariate inputs and a single training example (x, t). The predictions are a linear function followed by a sigmoidal nonlinearity. Finally, we use the squared error loss function. The model and loss function are as follows:

$$z = wx + b \tag{16}$$

$$y = \sigma(z) = \frac{1}{1 + \exp(-z)} \tag{17}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2 + \lambda\frac{w^2}{2} \tag{18}$$

Please note all the variables are simplified to 1-dimension instead of vectors. To solve this problem, we compute the derivatives $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$ respectively, which can be expressed as:

$$\mathcal{L} = \frac{1}{2}(\sigma(wx + b) - t)^2 + \lambda\frac{w^2}{2}. \tag{19}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx + b) - t)^2 + \lambda\frac{w^2}{2}\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx + b) - t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx + b) - t)\frac{\partial}{\partial w}(\sigma(wx + b) - t) + \lambda w \\
&= (\sigma(wx + b) - t)\sigma'(wx + b)x + \lambda w.
\end{aligned} \tag{20}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx + b) - t)^2 + \lambda\frac{w^2}{2}\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx + b) - t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx + b) - t)\frac{\partial}{\partial b}(\sigma(wx + b) - t) \\
&= (\sigma(wx + b) - t)\sigma'(wx + b).
\end{aligned} \tag{21}$$

In this case maybe you would feel the computation for derivatives is quite easy (several steps above seem to be trivial), but please remember this is the simplest case with only 1 layer, 1-dimension input and output. For the realistic neural network with tens of layers and high dimensional vectors, computing the derivatives for different parameters one by one could be very time consuming.

From the derivative example above we observe two drawbacks for the computation:

- **Cumbersome:** The calculations are cumbersome. As in the example, for each step of derivative we need to check each component if it is related to the variables w or b, while this is doable for this simple example, it could be a disaster for the realistic neural network with complicated expression.
- **Redundant:** There are two many redundant work during the computation. For instance the first two steps for $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$ are identical in the example above. Moreover, the final expressions for $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$ contains repeated terms, such as $(\sigma(wx+b)-t)\sigma'(wx+b)$, these term has been computed twice during our calculation. Therefore we hope to find a better way to reduce this redundant work.

The idea behind backpropagation is to share the repeated computations wherever possible. We will see the backpropagation calculations, if done properly, are very clean and modular.

2.2 Preliminary: Multivariable chain rule

Backpropagation is based on chain rule. For the univariate case, the chain rule is given as:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(g(t)) = f'(g(t))g'(t). \tag{22}$$

Roughly speaking, increasing t by some infinitesimal quantity $\mathrm{d}t$ "causes" g to change by the infinitesimal $g(t)\mathrm{d}t$. This in turn causes f to change by $f(g(t))g(t)\mathrm{d}t$.

The multivariable Chain Rule is a generalization of the univariate one. Suppose we have a function f with two variables $x(t)$ and $y(t)$, both variables are controlled by t. If we want to compute $\frac{\mathrm{d}f}{\mathrm{d}t}$. Changing t slightly has two effects: it changes x slightly, and it changes y slightly. Each of these effects causes a slight change to f. For infinitesimal changes, these effects combine additively. The Chain Rule, therefore, is given by:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{23}$$

Please note that we compute the partial derivative $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at first, which indicates in this step, we assume x and y are independent (although they are not). In the following section we will introduce how chain rule helps.

2.3 An example of chain rule: how backpropogation works?

Consider the following function f:

$$F(\boldsymbol{W}, x, y) = f_1(w_1, y, f_2(w_2, f_3(w_3, f_4(w_4, f_5(...))))) \tag{24}$$

You could consider $F(\boldsymbol{W}, x, y)$ is the function of a N-layer network. In order to train this network we compute the derivative with respect to the corresponding parameters $w_1, w_2, \ldots, w_N$. These are

given as:

$$\frac{\partial}{\partial w_1} F(\boldsymbol{W}, x, y) = \frac{\partial f_1}{\partial w_1}, \tag{25}$$

$$\frac{\partial}{\partial w_2} F(\boldsymbol{W}, x, y) = \frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial w_2}, \tag{26}$$

$$\frac{\partial}{\partial w_3} F(\boldsymbol{W}, x, y) = \underbrace{\frac{\partial f_1}{\partial f_2}}\frac{\partial f_2}{\partial f_3}\frac{\partial f_3}{\partial w_3}, \tag{27}$$

$$\frac{\partial}{\partial w_4} F(\boldsymbol{W}, x, y) = \underbrace{\frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}}\frac{\partial f_3}{\partial f_4}\frac{\partial f_4}{\partial w_4}, \tag{28}$$

$$\vdots \quad \vdots \quad \vdots \tag{29}$$

$$\frac{\partial}{\partial w_N} F(\boldsymbol{W}, x, y) = \underbrace{\frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}\frac{\partial f_3}{\partial f_4}\frac{\partial f_4}{\partial f_5}\cdots}\frac{\partial f_N}{\partial w_N}. \tag{30}$$

Probably you could notice that, a huge portion of these derivatives has been computed recursively (See the underbrace part above), since we are using chain rule repeatedly. Therefore, in order to avoid repeat computation, a possible way is to compute such derivatives from w_1 to w_N, and save the components for further usage.

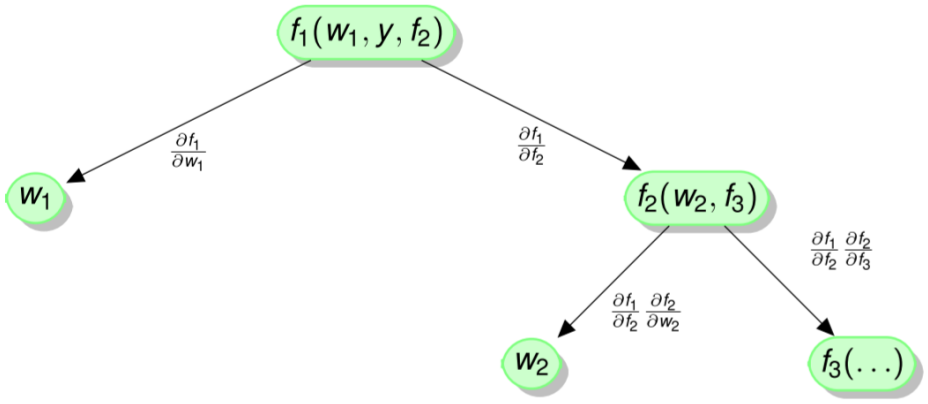
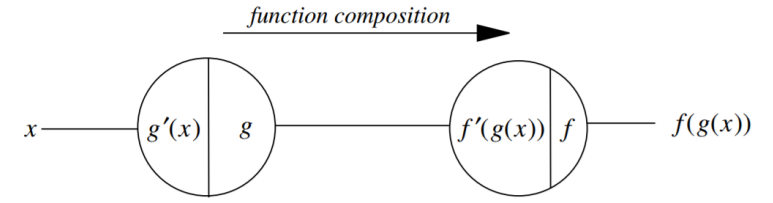


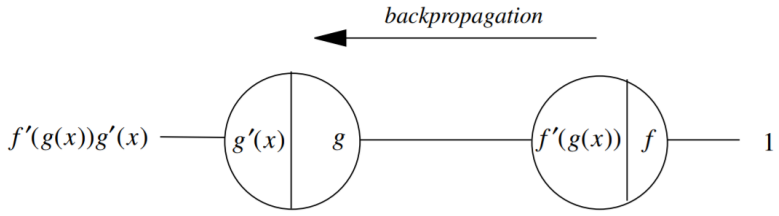
Figure 5: An example of backpropogation, we start from compute

2.4 Computation graph

Computation graph is a graph structure that the nodes correspond to all the values that are used to compute, such as $w_1, \ldots, w_n$ and $f_1, \ldots, f_N$, and the edges indicate which values are computed from which other values, for example f_1 should be computed by f_2 and w_1, so it is connected with nodes f_2 and w_1. In Figure. 5 we show the computation graph for our example above. You will find that all the functions (neural networks) can be illustrated clearly by a computation graph. Moreover, computation graph is a tree structure, where the children of each node are the values that contain this node, for example $f_1(w_1, y, f_2)$ should be the children of the node w_1, this definition could help us understand backpropogation further.



(a) Forward pass



(b) Backward pass

Figure 6: An example of *Top:* forward pass, *Bottom:* backward pass. The network function is given as $f(g(x))$. In the forward pass we compute from the input x to the output $f(g(x))$. Then in the backward pass we compute from the output to the input.

2.5 Complete backpropogation algorithm

Let us start with the formal definition of the backpropogation algorithm. Let $v_1, v_2, \ldots, v_N$ denote all of the nodes in the computation graph, in a topological ordering (A topological ordering is any ordering where parents come before children). We wish to compute all of the derivatives v_i, although we may only be interested in a subset of these values. The backpropagation algorithm contains two steps:

- **Forward pass:** In the forward pass, we compute the values on all of the nodes in our computation graph, since in backward pass we need to compute the values of derivatives, which need the value of each node. We start from the parents nodes (i.e., from $v_1, v_2, \ldots$ to v_N) to its children nodes, since children nodes are defined by its parents node. You can imagine this is a process from the input side of our neural network to the output. In the top plot of Figure.6 we show a simple example of forward pass.
- **Backward pass:** In the backward pass, we compute the derivatives of each node from an opposite direction as in the forward pass (i.e., from $v_N, v_{N-1}, \ldots$ to v_1). For each node, we compute the derivative based on the derivatives of its children nodes, then save it for computing the derivatives of its parents. For example in the bottom plot of Figure.6, we compute $f'(g(x))$ and save it, then we compute the derivative of its parent $g(x)$ as $f'(g(x))g'(x)$. This can be thought as a result of chain rule.

In Algorithm. 1 we present the pseudo codes of backpropagation algorithm, which has two merits: No redundant computation since it takes the advantage of chain rule. Also the procedure of backpropagation is modular, as we divide the computation of derivatives into small patches, once we would like to modify some parts of our network, what we need is to only change the expression inside these parts, and keep other expression fixed. Otherwise we need to re-compute the whole things for the naive method.

Algorithm 1: Backpropagation

Input: Initial computation graph G, with nodes $\{v_1, v_2, \ldots, v_N\}$.

Output: Derivatives of all nodes, denote as $\{\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_N\}$

Forward pass: **for** $i = 1, \ldots, N$ **do**

  Compute v_i as a function of its parents nodes, denote as $\mathbf{Pa}(v_i)$

end

$\bar{v}_N = 1$

Backward pass: **for** $i = N-1, \ldots, 1$ **do**

  Compute $\bar{v}_i$ as a function of its children nodes, denote as $\mathbf{Ch}(v_i)$

$$\bar{v}_i = \sum_{j \in \mathbf{Ch}(v_i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

  and we need to save this $\bar{v}_i$ for further usage.

end

2.6 Examples of Backpropogation

In this section we take several examples of backpropagation as an exercise.

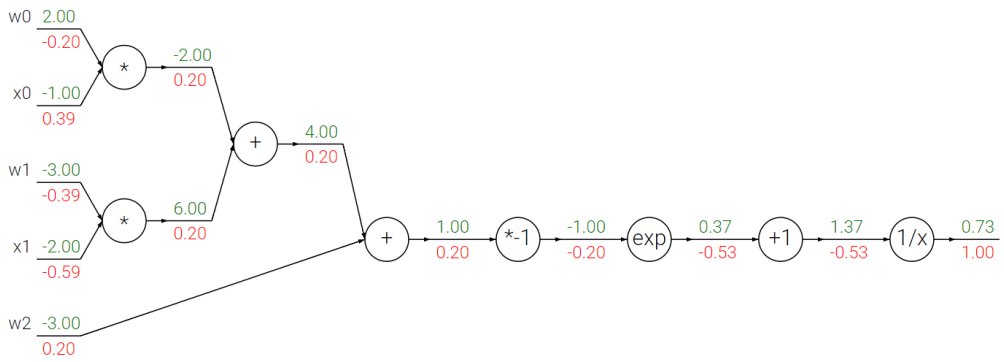


Figure 7: Example 1: The computation graph. The forward pass result is shown in green color and the backward pass result is shown in red color. Please note that the backward computation for the sigmoid function is unnecessary (see Remark 1). Figure copied from CS231n website of Stanford University.

- **Example 1:** Let us consider the following function:

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \tag{31}$$

  Actually this is a network layer with 2-dimension input and it uses sigmoid activation function. Then we hope to compute the derivatives of the parameters w and x (x could be the output of previous networks, then we also need to compute its derivatives). We show the computation graph in Figure. 7 and apply the backpropagation algorithm based on it. You could easily verify the result if interest.

Remark 1: It is important to notice that sometimes we do not need to compute the derivatives of all the nodes in computation graph. In the example above, let us denote the input of sigmoid activation function σ as:

$$z = w_0x_0 + w_1x_1 + w_2, \quad \sigma(z) = \frac{1}{1+e^{-z}}. \tag{32}$$

It is interesting to see the derivatives of z can be expressed as:

$$\frac{\partial f}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = (1-\sigma(z))\sigma(z). \tag{33}$$

Therefore, by using this property we could save a great number of time, since we compute the derivative of z directly without additional calculation for every node that uses sigmoid activation function. Similar results can be found for other functions, such as Relu ($\max(0, z)$) or tanh activation function. Please have a try if interest.

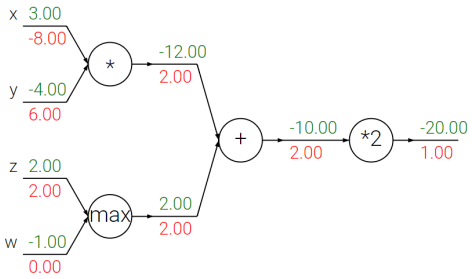


Figure 8: Example 2: The computation graph. The forward pass result is shown in green color and the backward pass result is shown in red color. Figure copied from CS231n website of Stanford University.

- Example 2: Let us consider another example:

$$f(x, y, z, w) = 2(xy + \max(z, w)). \tag{34}$$

The computation graph is shown in Figure. 8, please check the result if interest.

Remark 2: The point we want to emphasize is the three most commonly used gates in neural networks (add,mul,max), they all have very simple interpretations in terms of how they act during backpropagation, and we could utilize these properties while programming our neural network.

Add gate: The **add gate** always takes the gradient on its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass.

Max gate: The **max gate** routes the gradient. Unlike the add gate which distributed the gradient unchanged to all its inputs, the max gate distributes the gradient (unchanged) to exactly one of its inputs (the input that had the highest value during the forward pass).

Multiply gate: The **multiply gate** is a little less easy to interpret. Its local gradients are the input values (except switched), and this is multiplied by the gradient on its output during the chain rule.

2.7 Non-convexity of neural network: How to initialize?

In the Lecture 4 and 7 we have shown that if a function that we want to optimize is convex, then the local optimal is also the global optimal. This property let us compute the linear regression, binary classification and other linear models efficiently. However, due to the complex structure of neural network, most of them are non-convex, or sometimes hard to prove their convexity. Therefore we have the following observation:

- Due to the non-convexity, we are no longer guarantee to find the optimal minimum. All the results we get are local optimal, while it is hard to verify if it is global optimal or not.
- Different Initialization values matters. In other words, different initialization strategies could lead to different result.
- A famous initialization strategy is given by Glorot and Bengio in 2010 [2]. In their model, suppose a given layer of neural network has input vector $\boldsymbol{x} \in \mathbb{R}^m$, and output $\boldsymbol{y} \in \mathbb{R}^n$, then we have:
$$y = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{35}$$
Therefore $\boldsymbol{W}$ should be a m by n matrix. The initialization strategy is to set $\boldsymbol{W}$ as:
$$w_{ij} \sim \mathbf{U}\left[-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right]. \tag{36}$$
Later on they argue another strategy as:
$$w_{ij} \sim \mathbf{U}\left[-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right]. \tag{37}$$
The experimental result shows their advantage on the convergence speed. Please refer to the main paper if interest.

3 Deep neural network

A deep neural network (DNN for brevity) is an artificial neural network (ANN) with multiple layers between the input and output layers. It has been shown to have great performance on many artificial intelligence tasks. In this section we will briefly go through the development of DNN in recent years.

3.1 Advantage and disadvantage of DNN

Advantage: The most important advantage of deep neural network, is the ability of automatically learning feature space transformations (hierarchical abstractions of data) such that data is easily separable at the output. For example in the image classification task, if we construct a deep convolutional neural network and take a great number of images as the training data, then it could automatically learns the hierarchical features of different objects. In Figure. 9 we show some features that it learned. For the faces catogory, the features in 2^{nd} layer are different parts of face, and the features in the 3^{rd} layer are different whole faces. Also we can observe similar scenes on other objects categories. This is why DNN has a great performance on image classification.

Disadvantage: Due to the complex structure of DNN, the training process usually require a great number of computational resource (GPUs) to make the training procedure efficient, also it demands significant amount of training data to guarantee that it learns the right features. This can be thought as the biggest disadvantage of DNN.

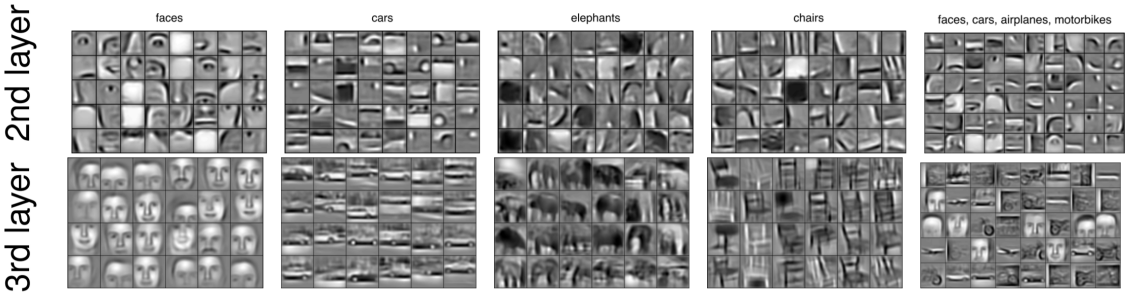


Figure 9: The features that learned for different objects. It is clear that deep neural network can learn the features of different objects very well. Moreover, the features of deeper layer are more hierarchical.

3.2 Why DNN becomes popular?

Generally there are four reasons that make DNN be popular:

- Sufficient computational resources: we are able to training a DNN within a short time period (usually several days).
- Sufficient data: due to the improvement of Internet and hardware storage, we are able to collect a huge amount of training data, for example images or audios.
- Sufficient algorithm advance: in recent years we have witnessed tons of algorithms about how to designing a better DNN.
- Sufficient evidence that it works: In a lot of tasks, the experimental results by DNN have shown a huge amount of improvement than the previous methods. Such as image classification.

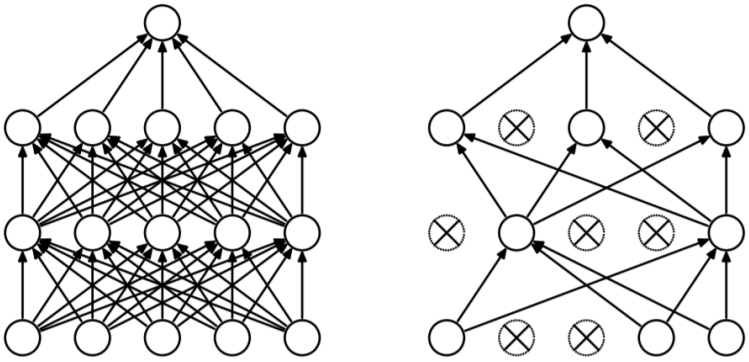
In summary, this combination lead to significant performance improvements on many datasets

3.3 Some algorithmic advances:

Here we briefly introduce several algorithmic advances about training DNN, please refer to the main papers if interest.

- **Rectified linear unit** ($\max\{0, x\}$) **[7]:** Rectified linear unit (Relu) activation function[] is used to solve the **vanishing gradient problem**. The problem is, in some cases, the gradient of our neural network will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. For example if we use the famous sigmoid function as the activation function:
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{38}$$
when z is quite large or small the gradient could be close to zero. While if we use the Relu $\max\{0, x\}$ instead, the gradient could always be a constant if it is activated.
- **Dropout [9]: Overfitting** is another spiny problem in training neural network, it means during the training process, our neural network corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations

reliably. To address this problem we could use a simple strategy called dropout. What dropout does is during each iteration, we randomly set the weight of each neuron to be zero, as a consequence it could prevent some weights to be dominant in the network. In Figure. 10 we show a simple diagram about dropout.



(a) Standard Neural Net (b) After applying dropout.

Figure 10: Dropout Neural Net Model. *Left*: A standard neural net with 2 hidden layers. *Right*: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. Figure copied from [9].

- **Good initialization heuristics:** As we show in the section 2.7, although finding the optima for Neural network is tricky, we still have a great number of initialization strategies that speed up the convergence rate.
- **Batch-Normalization during training [4]:** Batch normalization is another simple strategy that could increase the stability and convergence speed of DNN. What it does is, for each layer between the activation function, we normalize the input data of each node to be zero mean and vairance one. This process make the input of each node has the nearly identical normal distribution. We present the algorithm in Figure. 11. For the details please refer to the main paper.

3.4 Popular architectures and ImageNet Challenge

In this section we introduce several popular DNN architectures, from LeNet in 1998 to ResNet in 2015. Although all of them are only designed for image classification task, a lot of architectures that using the similar structure has been applied to other fields.

- **LeNet[6]:** LeNet, with 7-layers, is the first succesful Convolutional Neural Network (CNN for brevity) that invented by Yann Lecun in 1998, which is used to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel images. In Figure. 12 we show its structure. LeNet can be thought as a pioneer of CNN, where lots of important concepts of CNN comes from it. However at that time due to the lack of computation resource, people did not pay too much attention on it since its performance is not surprising, as what we have seen in the deep CNN.

- **AlexNet [5]:** Inspired by LeNet, AlexNet was developed in 2012 by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. It has the similar structure as LeNet but was deeper, bigger,

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 11: Batch Normalizing Transform, applied to activation x over a mini-batch. Figure copied from [4].

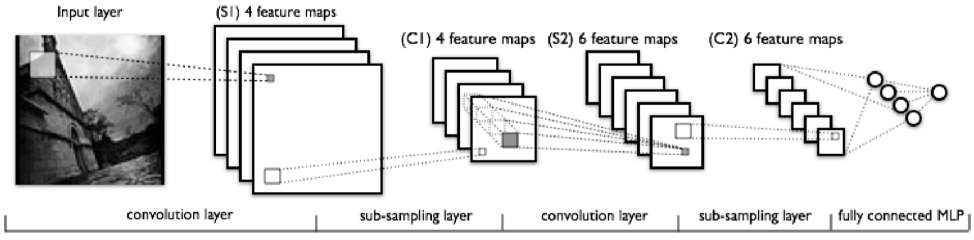


Figure 12: The structure of LeNet, some basic ideas of CNN such as sub-sampling and local filtering, were derived from it.

and featured Convolutional Layers stacked on top of each other. The structure was shown in Figure. 13. AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The main contribution of AlexNet is that, it is the first time to prove that DNN can be used on realistic Computer Vision tasks, also it could be thought as the first DNN that popularized the Deep Learning field in Computer Vision. Later on we will introduce the ImageNet ILSVRC challenge.

- **GoogleNet [10]:** Since the emergence of AlexNet, more DNNs that used to do image classification came up. The ImageNet ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google (so we call as GoogleNet). Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. In Figure. 14 we show the Inception Module that used in Googlenet, you can see the interesting thing is that the size of filters are considerablely smalled then AlexNet.

- **VGGNet [8]:** The runner-up in ImageNet ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution

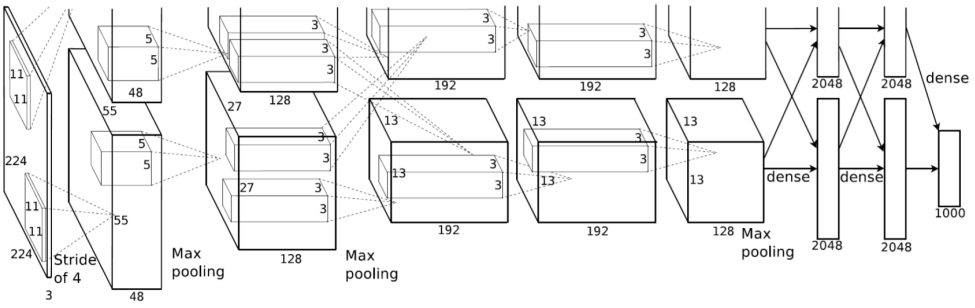


Figure 13: The structure of AlexNet, it inherits the basic idea of LeNet, but dramatically increase the number of filters.

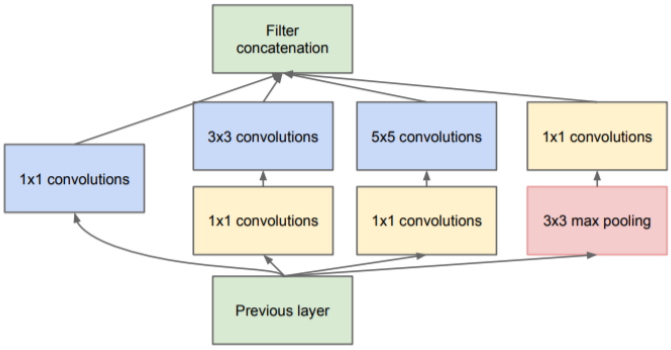


Figure 14: The structure of inception module with dimensionality reduction in GoogleNet.

was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Their pretrained model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

- **ResNet [3]:** Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. The reader is also referred to Kaiming's presentation (video, slides), and some recent experiments that reproduce these networks in Torch. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 10, 2016).

ImageNet [1]: The ImageNet project is a large visual database designed for use in visual object recognition software research. Over 14 million URLs of images have been hand-annotated by ImageNet to indicate what objects are pictured; in at least one million of the images, bounding boxes are also provided. ImageNet contains over 20 thousand categories; a typical category, such as "balloon" or "strawberry", contains several hundred images.

ImageNet Challenge: Since 2010, the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is a competition where research teams evaluate their algorithms on the given data set,

and compete to achieve higher accuracy on several visual recognition tasks. All the architectures that we present above, excepet LeNet, were submitted to the ILSVRC and got top scores. Benefits from DNN architecture, the error rate of the best team on each year has been dramaticly decreased from 20% to nearly 5%, from 2011 to 2016. As we can see in Figure. 15.

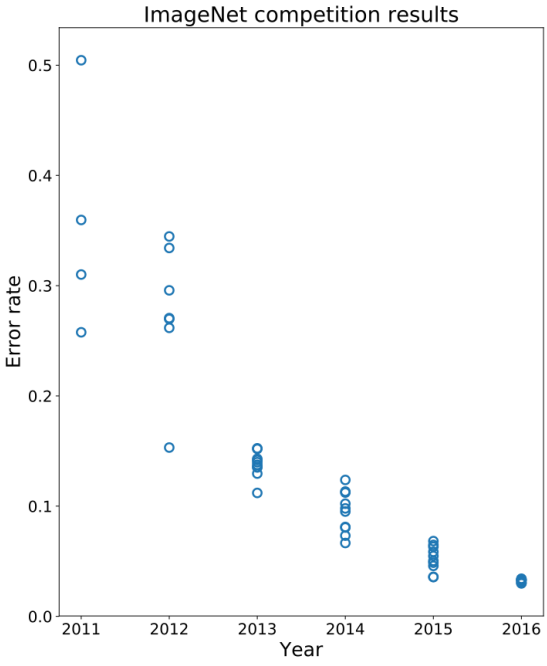


Figure 15: Error rate history on ImageNet (showing best result per team and up to 10 entries per year). Figure copied from Wikipedia.

4 Quiz:

- **What are deep nets?** A: A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers.
- **How do deep nets relate do SVMs and logistic regression?** A: As a generalization of the function $F(\boldsymbol{w}, \boldsymbol{x}^{(i)}, y^{(i)})$.
- **What is back-propagation in deep nets?** A: A framework that compute the gradient efficiently.
- **What components of deep nets do you know?** A: Please refer to the section 3.
- **What algorithms are used to train deep nets?** A: Please refer to the section 3.

Acknowledge: I would like to thanks to the course website of CS231n in Stanford University, and CSC321 in University of Toronto for providing the figures and some illustrations. Also thanks to Alex and TAs who provide the figures in previous lecture notes.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[2] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Y. LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20, 2015.

[7] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[8] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.