coursera

Stanford University

Andrew Ng

Summary

Neural Networks and Deep Learning

---

by Amber Brands

July 26, 2018

[1] https://www.coursera.org/specializations/deep-learning

Contents

1 Neural Networks and Deep Learning

1.1 Introduction to Deep Learning

1.1.1 What is a Neural Network?

Neural networks are computing systems inspired by biological neural networks. Such systems learn (progressively improve performance) to do tasks considering examples generally without task-specific programming. For example, a neural network could be used for predicting housing prices y based on several input features x (e.g. size, number of bedrooms, zip code, wealth etc.) (see Figure 1).

1.1.2 Supervised Learning with Neural Networks

In supervised learning you train an algorithm given labelled training data. Examples of supervised learning are predicting housing prices (real estate), the possibility of user's clicking on add (online advertising), object recognition (photo tagging), text transcript (speech recognition), Chinese translation (machine translation) and position of other cars (autonomous driving). For different types of application we use different types of neural networks (NN), such as standard (real estate, online advertising), convolutional (photo tagging), recurrent (speech recognition and machine translation) and custom/hybrid/more custom (autonomous driving) NN. Moreover, within supervised learning you can distinguish structured data and unstructured data, where structured data means that each of the features has a well-defined meaning (user age, ID, size, etc). and unstructured data are well less defined (raw audio, images and text). Thanks to artificial intelligence and deep learning, computers are recently able to recognize unstructured data much better.

1.1.3 Why is Deep Learning taking off?

When performances of traditional learning algorithms such as SVM, and logistic regression, against amount of data, you'll see that performance plateaus at some point. However, when you use more complicated algorithms (such as small, medium and large NN), performance will take on a linear line. This can also be formulates as scale drives deep learning progress where performance depends both on the amount of data as well as the size of the network that is used. In the regime of small training sets, it is not clear which type of network yields the highest performance as data size increases. Three major points which influence performance are data, computation and algorithms. For instance, since the last couple of years, the sigmoid function in machine learning has been replaced by a ReLU function (Rectifier Linear Unit), resulting in a higher gradient over all values and therefore a gradient descent that works much faster. The process of training a neural network is highly iterative including an idea, code and experiment.

1.2 About this Course

In this specialization we will focus on the following specializations:

- Neural Networks and Deep Learning
- Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
- Structuring Machine Learning Projects
- Convolutional Neural Networks
- Sequence Models

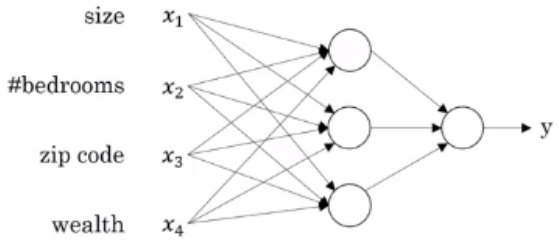


Figure 1: Neural Network for housing prices

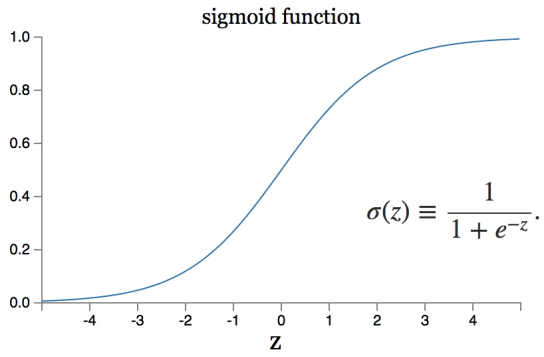


Figure 2: Sigmoid function used for Logistic Regression

1.3 Logistic Regression as a Neural Network

1.3.1 Binary Classification

Logistic regression is an algorithm for binary classification. An example of binary classification is object recognition. Let's create a pixel vector X including containing pixel values for red, green and blue colors. For instance, when we have a 64×64 image, we end up with $64 \times 64 \times 3 = 12288$ input values. For the following courses we will use the following denotations:

- (x, y) where $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
- m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$.
- $X = \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(m)} \\ | & | & & | \end{pmatrix}$
- $Y_{shape} = (1, m)$

1.3.2 Logistic Regression

Given an input feature vector X, we want to predict the output value $\hat{y} = P(y = 1|x)$ with parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, our output will be $\hat{y} = \sigma(w^T x + b)$ (for the function of σ see Figure 2, where $\sigma(z) = \frac{1}{1+e^{-z}}$). Now, if z is very large positive number, $\sigma(z) \approx 1$, while if z is a very large negative number $\sigma(z) \approx 0$. Now when you implement this algorithm, you have to find the correct parameters w and b, where $\hat{y}$ correctly predicts the outcome y. Another common way to notate the parameters is $\hat{y} = \sigma(\Theta^T x)$, where Θ_0 equals b. However, this notation will not be used in this course.

1.3.3 Logistic Regression Cost Function

To recap from the previous section, $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$. Now given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$, we want $\hat{y}^{(i)} \approx y^{(i)}$. The loss (error) function $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ is not a option here because this function is non-convex (i.e. gradient descent will get stuck in local optima). A similar, but convex loss (error) function is follows:

$$L(\hat{y}, y) = -(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})) \tag{1}$$

With this logistic regression loss function, you'll get the following intuition (for one training example):

- If $y = 1$, then $L(\hat{y}, y) = -log(\hat{y})$, therefore we want $log(\hat{y})$ to be large, and consequently $\hat{y}$ to be large as well.
- If $y = 0$, then $L(\hat{y}, y) = -log(1 - \hat{y})$, therefore we want $1 - \hat{y}$ to be large, and consequently $\hat{y}$ to be small.

For an entire training set, the loss function, also known as the cost function is as follows:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \cdot log(\hat{y}^{(i)}) + (1 - y^{(i)}) \cdot log(1 - \hat{y}^{(i)})] \tag{2}$$

So in summary, the loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

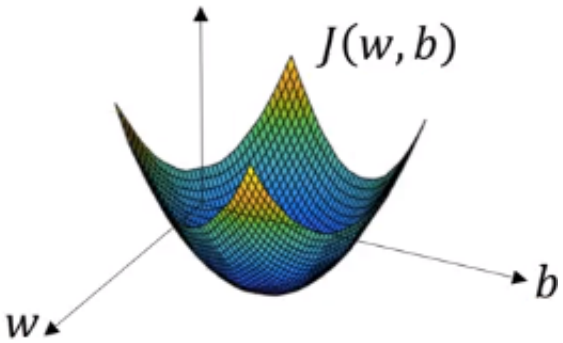


Figure 3: Cost function J for the parameters w and b. This is a convex function, because it has one global optimum (and no local optima)

1.3.4 Gradient Descent

To recap from earlier sections: $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$, and:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \cdot log(\hat{y}^{(i)}) + (1-y^{(i)}) \cdot log(1-\hat{y}^{(i)})] \tag{2}$$

Now, we want to find w, b that minimize $J(w,b)$ (see Figure 3). This function is also known as a convex function, because it has one global optimum (and no local optima). We do this by initializing random values for w and b. Subsequently, gradient descent takes a step in the deepest downhill direction (where J takes on a smaller value). Gradient descent is written down formally:

$$w := w - \alpha \frac{\partial J(w)}{\partial w} \tag{3}$$

$$b := b - \alpha \frac{\partial J(b)}{\partial b} \tag{4}$$

where α is the learning rate, followed by the derivative terms indicating the slope of the function. The slope of the function is the height divided by the width. When the derivative is positive, you end up subtracting a value of w therefore lowering the value of the parameter. In contrast, when the derivative is negative, you eventually will add a value to w increasing the value of the parameter.

1.3.5 Derivatives

As was mentioned earlier, the derivative is the slope which is computed as the height divided by the width ($d = \frac{height}{width}$). This is written formally as $\frac{df(a)}{a} = \frac{d}{da}f(a)$. For a straight line, this means that we you move a variable to the right with an infinitely small amount, the output of the function will shift with a constant value. Next we will look at non-linear examples where the derivative can have different values.

1.3.6 More Derivative Examples

When you have non-linear functions (e.g. $f(a) = a^2$), the derivative will have a different value depending on which value you take as input variable. In fact, the formula that determines the change in the derivative when the input variable is changes is in fact $2a$. More formally:

- when $f(a) = a^2$, then $\frac{d}{da}f(a) = 2a$.
- when $f(a) = a^3$, then $\frac{d}{da}f(a) = 3a^2$.
- when $f(a) = log_e(a)/ln(a)$, then $\frac{d}{da}f(a) = \frac{1}{a}$.

1.3.7 Computation Graph

When compute $J(a,b,c) = 3(a+bc)$, we first need to compute $u = bc$, then $v = a + u$, and finally $J = 3v$. These subsequent steps can be visualized in a computation graph (see Figure 4). Now one step of backward propagation on a computation graph yields the derivative of the final output variable. For more explanation, see the next section.

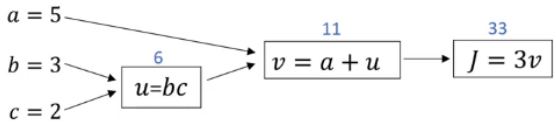


Figure 4: Computation graph for $J(a, b, c) = 3(a + bc)$

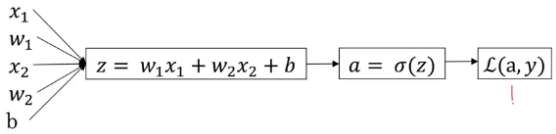


Figure 5: Computation graph for Logistic Regression

1.3.8 Derivatives with a Computation Graph

Now say we want to compute $\frac{dJ}{dv}$, where $J = 3v$ (and $v = 11$). Meaning that the increase in v is three-times the increase in J ($\frac{dJ}{dv} = 3$). Now let's look at another example, namely $\frac{dJ}{da} = 3$, because when you change a, you change v and therefore J, also written as $\frac{dJ}{da} = \frac{dJ}{dv} = \frac{dv}{da}$. In calculus, this is also called the chain rule. Because you calculate backwards (from right to left), this is part of backpropagation. This can be applied to learning algorithm where $\frac{dFinalOutputVar}{dvar}$. However, since this is a very long variable name, you could also call this $dJdvar$. However, since you are always taking the variables with respect to dJ, we'll introduce a new notation. In the code we write we'll use $dvar$ which represents that quantity ($\frac{dJ}{dv} = dv$). In other words, in this class the coding devention $dvar$ represents the derivative of a final output variable with respect to various intermediate quantities. The derivatives for all the variables can be computed as follows:

- $\frac{dJ}{dv} = 3$:
  $v = 11 \to v = 11,001$
  $J = 33 \to v = 33,003$
- $\frac{dJ}{da} = 3$: when
  $a = 5 \to a = 5,001$
  $v = 11 \to v = 11,001$
  $J = 33 \to v = 33,003$
  $(\frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da} = 3 \times 1)$.
- $\frac{dJ}{du} = 3$:
  $u = 6 \to u = 6,001$
  $v = 11 \to v = 11,001$
  $J = 33 \to v = 33,003$
  $(\frac{dJ}{du} = \frac{dJ}{dv}\frac{dv}{du} = 3 \times 1)$.
- $\frac{dJ}{db} = 6$:
  $b = 3 \to b = 3,001$
  $u = 6 \to u = 6,002$
  $(\frac{dJ}{db} = \frac{dJ}{du}\frac{du}{db} = 3 \times 2)$.
- $\frac{dJ}{dc} = 9$:
  $c = 2 \to b = 2,001$
  $u = 6 \to u = 6,003$
  $(\frac{dJ}{dc} = \frac{dJ}{du}\frac{du}{dc} = 3 \times 3)$.

1.3.9 Logistic Regression in Gradient Descent

To recap logistic regression from earlier sections, $z = w^T x + b$, where $\hat{y} = a = \sigma(z)$ and $L(a, y) = -(y \cdot log(a) + (1 - y) \cdot log(1 - a))$. When these elements are put in an computation graph (see Figure 5), for backpropagation the following derivatives should be computed (for one single example):

- $da = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$
- $dz = \frac{dL}{dz} = \frac{dL(a,y)}{dz} = a - y$
- $dw_1 = \frac{dL}{dw_1} = x_1 \cdot dz$
- $dw_2 = \frac{dL}{dw_2} = x_2 \cdot dz$
- $db = dz$

with gradient descent for one example (for the entire training set we use the partial derivative ∂):

$$w_1 := w_1 - \alpha \cdot dw_1 \tag{3}$$

$$w_2 := w_2 - \alpha \cdot dw_2 \tag{3}$$

$$b := b - \alpha \cdot db \tag{4}$$

1.3.10 Gradient Descent on m Examples

Now we want to perform gradient descent not for one example, but for our entire training set. The cost function is as follows:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \cdot log(a^{(i)}) + (1-y^{(i)}) \cdot log(1-a^{(i)})] \tag{2}$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$ and you have the parameters $dw_1^{(i)}, dw_2^{(i)}$ and $db^{(i)}$. Now you get the following cost function:

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)}) \tag{5}$$

$$\frac{\partial}{\partial w_2} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_2} L(a^{(i)}, y^{(i)}) \tag{5}$$

In this code we use w_1 and w_2 as accumulators averaged over our entire training set. Using explicit for-loops result in a less efficiently and computationally expensive code. A solution for this is vectorization.

1.4 Python and Vectorization

1.4.1 Vectorization

See file Jupyter "$Machine Learning_N N.ipynb$" (directory "C:\Users\Amber\OneDrive\AI\PythonJupyter"). Many deep learning algorithms are run on GPU, while we run our code and demo's on CDU. GPU and CPU stand for "Graphics Processing Unit" and , "Central Processing Unit", respectively. It turns out that both GPU and CDU have parallel instructions called SIMD which stand for "Single Intuition Multiple Data". What this means is that functions which don't need explicit for-loops are run very efficiently.

1.4.2 More Vectorization Examples

Now when you look at the non-vectorized code of our logistic regression:

$J = 0$, $dw1 = 0$, $dw2 = 0$, $db = 0$
for i = 1 to m:
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J+ = -[y^{(i)} \cdot log(a^{(i)}) + (1-y^{(i)}) \cdot log(1-a^{(i)})]$
    $dz^{(i)} = a^{(i)}(1-a^{(i)})$
    $dw_1+ = x_1^{(i)} dz^{(i)}$
    $dw_2+ = x_2^{(i)} dz^{(i)}$
    $db+ = dz^{(i)}$
$J = J/m$, $dw_1 = dw_1/m$, $dw_2 = dw_2$, $db = db/m$

We not want to eliminate the second for-loop that computes dw_1 and dw_2. This can be done as follows:

$dw = np.zeros((n-x,1))$ (instead of $dw1 = 0$, $dw2 = 0$)
$dw+ = x^{(i)} dz^{(i)}$ (instead of $dw_1+ = x_1^{(i)} dz^{(i)}$ and $dw_2+ = x_2^{(i)} dz^{(i)}$)
dw/m (instead of $dw_1 = dw_1/m$, $dw_2 = dw_2$)

1.4.3 Vectorizing Logistic Regression

Now we will vectorize are code for forward propagation even further eventually yielding a code that uses no for-loop (even no for-loop to iterative over the training examples m). Earlier we defined the matrix $X = \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{pmatrix}$ with dimensions (n_x, m) (number of input features $\times$ number of training examples, $\mathbb{R}^{n_x \times m}$). When you assume $z^{(i)} = w^T x^{(i)} + b$, you want compute a vector $[z^1, z^2, \dots, z^m] = w^T X + [b, b, \dots, b]$, where $b \in \mathbb{R}^{1\times m}$. In other words $w^T \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{pmatrix} +$

$[b, b, \ldots, b] = [w^T x^{(1)} + b, w^T x^{(2)} + b, \ldots, w^T x^{(m)} + b]$. The Python command is as follows: $Z = np.dot(w.T, X) + b$, where Python automatically expands b. Next, we want to find a way to compute $[a^1, a^2, \ldots, a^m]$ at the same time. $A = [a^1, a^2, \ldots, a^m] = \sigma(Z)$. Now we don't have to loop over the training examples, instead you can compute A for all training examples at the same time.

1.4.4 Vectorizing Logistic Regression's Gradient Output

Now we discuss how to perform vectorizing logistic regression gradient descent.

- You want to compute $dZ = [dz^{(1)}, dz^{(2)}, \ldots, dz^{(m)}]$ where $dz^{(i)} = a^{(i)} - y^{(i)}$. Moreover, $A = [a^1, a^2, \ldots, a^m]$ and $Y = [y^1, y^2, \ldots, y^m]$. Consequently, $dZ = A - Y$. Next, we want to compute db and dw.
- You want to compute $db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$, which in Python can be done as follows: $db = \frac{1}{m} np.sum(dZ)$.
- You want to compute $dw = \frac{1}{m} X dZ^T = \frac{1}{m} \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \ldots & x^{(m)} \\ | & | & & | \end{pmatrix} \begin{pmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{pmatrix} = \frac{1}{m}[x^{(1)} dz^{(1)}, x^{(2)} dz^{(2)}, \ldots, x^{(m)} dz^{(m)}] \in \mathbb{R}^{n \times 1}$.

Not let's put all the code together and implement logistic regression as a vectorized line of code. Our non-vectorized code looks as follows:

$J = 0$, $dw1 = 0$, $dw2 = 0$, $db = 0$
for i = 1 to m:
  $z^{(i)} = w^T x^{(i)} + b$
  $a^{(i)} = \sigma(z^{(i)})$
  $J += -[y^{(i)} \cdot log(a^{(i)}) + (1 - y^{(i)}) \cdot log(1 - a^{(i)})]$
  $dz^{(i)} = a^{(i)}(1 - a^{(i)})$
  $dw_1 += x_1^{(i)} dz^{(i)}$
  $dw_2 += x_2^{(i)} dz^{(i)}$
  $db += dz^{(i)}$
$J = J/m$, $dw_1 = dw_1/m$, $dw_2 = dw_2$, $db = db/m$

In the last sections we made the following alterations:

1. Instead of looping over $dw1$ and $dw2$ we replaced this with a vector value dw where $dw += x^{(i)} dz^{(i)}$.
2. We also vectorized our code in a way that the outer loop that iterates over the training examples is replaced as well where $Z = w^T X + b = np.dot(w.T, X) + b$ where $A = \sigma(Z)$. Next, we compute $dZ = A - Y$. Finally, $dw = \frac{1}{m} X dZ^T$ and $db = \frac{1}{m} np.sum(dZ)$. Subsequently, w and b are updated by $w = w - \alpha dw$ and $b = b - \alpha db$, respectively.

This code iterates one times over your training set. To iterate multiple number of times a for-loop is inevitable.

1.4.5 Broadcasting in Python

Broadcasting is a technique in Python that makes your code run more efficiently. An example is calculating % of calories from carb, protein and fat in 100g of different foods (e.g. apples, beef, eggs and potatoes). Now the question is, whether this can be done without using an explicit for-loop. See file Jupyter "$Machine Learning_N N.ipynb$" (directory "C:\Users\Amber\OneDrive\AI\PythonJupyter"). In Pyton, the following two lines will compute the amount of calories (in percentage):

```
cal = A.sum(axis = 0)
percentage = 100* A/(cal.reshape(1,4))
```

"axis = 0" sums horizontally and "axis = 1" sums vertically. "reshape" is here redundant since the matrix has already the right size. In fact, you divided a 3×4 matrix by a 1×4 matrix. The general principle of broadcasting in Python is as follows: $(m \times n)$ $+/-/\times$ $(1 \times n) = (m \times n)$.

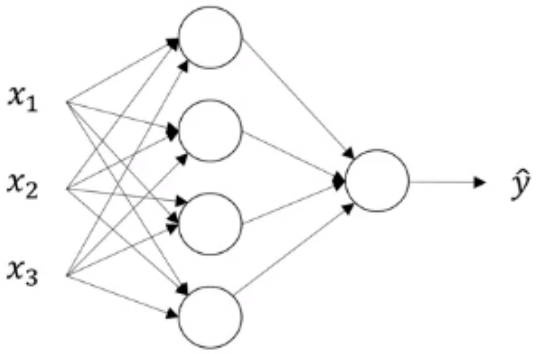


Figure 6: Neural network with one hidden layer

1.4.6 A note on Python/Numpy Vectors

Some comments concerning Python with regard to vectors and matrices:
a = np.random.randn(5) is called a "rank 1 array" (do not use when working with matrices and vectors!). Instead:
a = np.random.rand(5,1) → a.shape = (5,1) which is called a "column vector"
a = np.random.rand(1,5) → a.shape = (1,5) which is called a "row vector"
assert(a.shape == (5,1)) exertions statements to make sure the vector has the dimensions you desire.

1.4.7 Explanation of Logistic Regression Cost Function

If $y = 1$: $p(y|x) = \hat{y}$
If $y = 0$: $p(y|x) = 1 - \hat{y}$

$p(y|x) = \hat{y}^y(1-\hat{y})^{(1-y)}$, because:

If $y = 1$: $p(y|x) = \hat{y}(1-\hat{y})^0$
If $y = 0$: $p(y|x) = \hat{y}^0(1-\hat{y})^1 = 1 \times (1-\hat{y}) = 1 - \hat{y}$

$log(p(y|x)) = log(\hat{y}^y(1-\hat{y})^{(1-y)}) = y \cdot log(\hat{y}) + (1-y) \cdot log(1-\hat{y}) = -L(\hat{y}, y)$

This is how the cost function works on one example. Moreover, on an entire training set, the cost function works as follows:

$p(\text{labels in training set}) = \Pi_{i=1}^{m} p(y^{(i)}|x^{(i)})$

$log(p(\ldots)) = \Sigma_{i=1}^{m} log(p(y^{(i)}|x^{(i)}))$, where $-L(\hat{y}, y) = log(p(y^{(i)}|x^{(i)})) = -\Sigma_{i=1}^{m} L(\hat{y}, y)$

Eventually, you want to minimize the cost: $J(w, b) = \frac{1}{m}\Sigma_{i=1}^{m} L(\hat{y}, y)$

1.5 Shallow Neural Network

1.5.1 Neural Networks Overview

In this section you will learn how to implement a neural network. Last sections we discussed logistic regression (see Figure 5). A neural network looks a bit differently (see Figure 1), containing different layers. The notation that will be used from now on is $z^{[1]} = W^{[1]}x + b^{[1]}$, where the number between the bracket superscript refers to a specific layer ($z^{[i]}$, refers to the ith layer in the neural network).

1.5.2 Neural Network Representation

In Figure 6 is a neural network shown with one hidden layer. x_i is called the input layer, the middle layer is called the hidden layer and the last layer is called the output layer which predicts the value of y (notated as $\hat{y}$). The hidden layer is called hidden because the values are not visible in the training set. The vector notating the input features will be defined as $a^{[0]} = X$ ("activations of the input layer"). The activations of the hidden layer are denoted as $a^{[1]}$, and the output layer

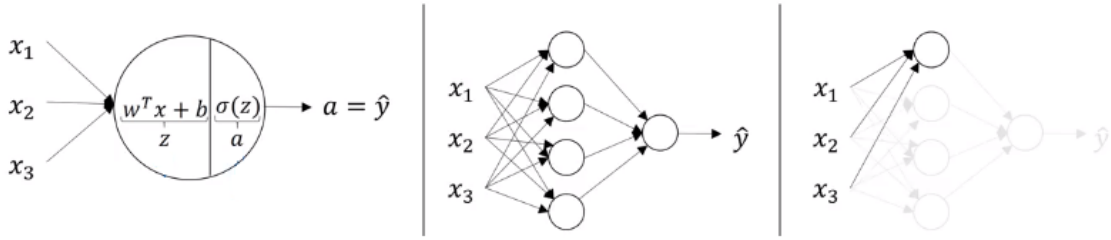


Figure 7: Logistic regression (left) and neural network (middle) representation

as $a^{[2]}$. In this case $a^{[1]}$ has a 4×1 dimensions because there are four nodes present in this hidden layer. This type of network is called a "two neural network" (the input layer is considered and counted as an official layer).

1.5.3 Computing a Neural Network's Output

In Figure 7, a representation is shown of logistic regression (left) and a neural network (middle). Now let's focus on one node (right) which receives multiple inputs.

- The selected node in the first hidden layer computes two values, namely $z^{[1]} = w^{[1]T}x + b^{[1]}$ and $a_1^{[1]} = \sigma(z_1^{[1]})$.
  - $a_i^{[l]}$ refers to the $i$the node in the lth layer

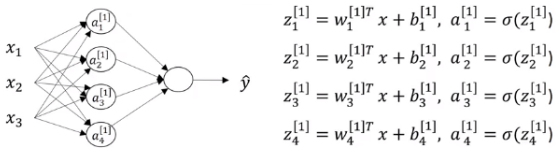


Figure 8: Neural network computation hidden layer

Next, we vectorize these four equations:

$$z^{[1]} = \begin{pmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{pmatrix} = \begin{pmatrix} w_1^{[1]T} X + b_1^{[1]} \\ w_2^{[1]T} X + b_2^{[1]} \\ w_3^{[1]T} X + b_3^{[1]} \\ w_4^{[1]T} X + b_4^{[1]} \end{pmatrix} = \begin{pmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{pmatrix} \text{ and } a^{[1]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{pmatrix} = \sigma(z^{[1]})$$

with vector $W^{[1]}$ and $b^{[1]}$ referring to the entire vector for parameter w and b in layer 1, respectively. So given input x:

- $z^{[1]} = W^{[1]}x + b^{[1]}$ *with* $(4 \times 1) = (4 \times 3)(3 \times 1) + (4 \times 1)$
- $a^{[1]} = \sigma(z^{[1]})$ *with* $(4 \times 1) = (4 \times 1)$
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ *with* $(1 \times 1) = (1 \times 4)(4 \times 1) + (1 \times 1)$
- $a^{[2]} = \sigma(z^{[2]})$ *with* $(1 \times 1) = (1 \times 1)$

1.5.4 Vectorizing across multiple examples

When you have m training examples, you need the computation discussed in the previous section m times eventually yielding $a^{[l](m)}$ in the lth layer for the mth training example and $[\hat{y}^{(1)}, \ldots, \hat{y}^{(m)}]$:

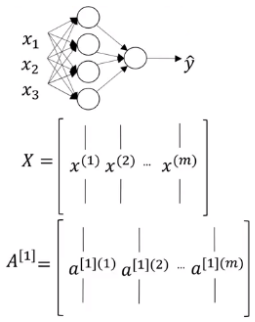
for i = 1 to m:
$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$

Recall that we define $X = \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{pmatrix}$. The vectorized implementation across multiple examples is as follows:

$Z^{[1]} = W^{[1]}X + b^{[1]}$
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

Eventually you'll end up with matrix A, where horizontally are the values for the different training examples m and vertically the different hidden layers l. This also holds true for the matrix Z and X (with matrix X the vertical values represent the different features).

1.5.5 Explanation for Vectorized Implementation



for i = 1 to m
$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$

$Z^{[1]} = W^{[1]}X + b^{[1]}$
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

where $X = A^{[0]}$.

1.5.6 Activation Functions

So far we used the Sigmoid function as an activation function (see Figure 2).

$$a = sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

However, there are alternative functions that could be used. An alternative function is the "tanh" function, which is a shifted function of the sigmoid function with a range between -1 and 1. This almost always works better than the sigmoid function because the mean is closer to 0, which is good when you want to center your data. An exception is the output layer, because the value of $\hat{y}$ is either 0 or 1 (which is also the range of the sigmoid function). Indexing activation function is done by subscript brackets. The tanh activation function is as follows:

$$a = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{7}$$

A downside of both the sigmoid as the tanh function is that when z takes either very small or very large values, the function will plateau and the slope will be close to 0, which can slow down gradient descent. To solve this, we can use another function called the Rectifier Linear Unit function (ReLU), which has the following formula:

$$a = ReLU(z) = max(0, z) \tag{8}$$

One major benefit is the reduced likelihood of the gradient to vanish. This arises when $a > 0$. In this regime the gradient descent has a constant value. In contrast, the gradient of the sigmoid activation function becomes increasingly small as the absolute value of x increases. The constant gradient descent of ReLUs results in faster learning. The other benefit of ReLUs is sparsity. Sparsity arises when $a \leq 0$. The more such units that exist in a layer the more sparse the resulting representation. The sigmoid activation function on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations. Now here are some rules of thumb when deciding which activation function to use:

- When you have a 0 to 1 range, a sigmoid function is the best option (in practice, this only applies to the output units).

- For all other ranges, the ReLU is the most optimal default activation function to use. One disadvantage is that the derivative is 0 when the value of z is negative. In practice this is fine, however, an alternative is the leaky ReLU, which has a very small slope when z is negative.

$$a = LeakyReLU(z) = max(0.01z, z) \tag{9}$$

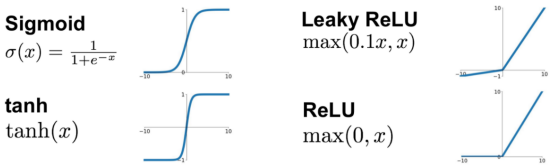


Figure 9: Non-linear activation functions

1.5.7 Why do you need non-linear activation functions?

An example of a linear activation function is $g(z) = z$. This results in $a^{[i]} = z^{[i]}$. If you were to use linear functions, the output will be a linear function as well. This makes the use of hidden layers not useful any more, because the composition of two linear functions is a linear function. A case in which you do use a linear activation function is the output layer when y takes on continuous values so that $\hat{y}$ must takes on a real value as well.

1.5.8 Derivatives of activation functions

When you implement backpropagation in your neural network you need to compute the slope of the activation functions. As was mentioned earlier, the slop is the heigh divided by the width. Let's take the sigmoid function ($g(z) = \frac{1}{1+e^{-z}}$)), where $\frac{d}{dz}g(z) =$ slope of $g(x)$ at z which is equal to $g(z)(1 - g(z))$.

- When z is very large, $g(z)$ will be close to 1: $\frac{d}{dz}g(z) = 1(1-1) \approx 0$
- When z is very small, $g(z)$ will be close to 0: $\frac{d}{dz}g(z) = 0(1-0) \approx 0$
- When z is equal to 0, $g(z)$ will be close to 0.5: $\frac{d}{dz}g(z) = \frac{1}{2}(1-\frac{1}{2}) \approx \frac{1}{4}$

The derivative of the sigmoid function can also be notated as $g'(z) = a(1-a)$. Now let's look at the *tanh* activation function, where $\frac{d}{dz}g(z) = 1 - (tanh(z))^2$.

- When z is very large, $tanh(z)$ will be close to 1: $\frac{d}{dz}g(z) \approx 0$
- When z is very small, $tanh(z)$ will be close to -1: $\frac{d}{dz}g(z) \approx 0$
- When z is equal to 0, $tanh(z)$ will be close to 0: $\frac{d}{dz}g(z) \approx 1$

The derivative of the *tanh* activation function can also be notated as $g'(z) = 1 - a^2$. Finally let's look at the ReLU and Leaky ReLU activation functions.

- ReLU: $g'(z) = 0$ when $z < 0$
  $g'(z) = 1$ when $z > 0$
- Leaky ReLU: $g'(z) = 0.01$ when $z < 0$
  $g'(z) = 1$ when $z > 0$

For both ReLU and Leaky ReLU $z = 0$ is not defined.

1.5.9 Gradient descent for Neural Networks

Let's recap the neural network with on hidden layer. In this network the parameters we have are $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ with $n_x = n^{[0]}, n^{[1]}, n^{[2]}$ and where $n^{[2]} = 1$ (output layer). The dimensions of our parameters are as follows:

- $w^{[1]} \rightarrow (n^{[1]}, n^{[0]})$
- $b^{[1]} \rightarrow (n^{[1]}, 1)$

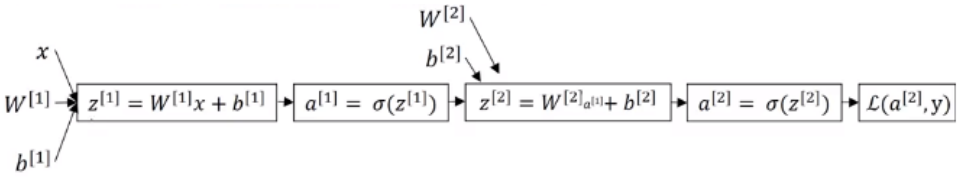


Figure 10: Computation graph neural network with one hidden layer

- $w^{[2]} \rightarrow (n^{[2]}, n^{[1]})$
- $b^{[2]} \rightarrow (n^{[2]}, 1)$

The cost function for binary classification is $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\Sigma_{i=1}^{m} L(\hat{y}, y)$ where $\hat{y} = a^{[2]}$. Next you need to compute the gradient descent:

Repeat {
Compute prediction $(\hat{y}^{(i)}, \ldots, \hat{y}^{(i)})$
$dW^{[1]} = \frac{dJ}{dW^{[1]}}, db^{[1]} = \frac{dJ}{db^{[1]}}, dW^{[2]} = \frac{dJ}{dW^{[2]}}, db^{[2]} = \frac{dJ}{db^{[2]}}$
$W^{[1]} = W^{[1]} - \alpha dW^{[1]}$
$b^{[1]} = b^{[1]} - \alpha db^{[1]}$
$W^{[2]} = W^{[2]} - \alpha dW^{[2]}$
$b^{[2]} = b^{[2]} - \alpha db^{[2]}$
}

This is one iteration of gradient descent, for convergence you nee to repeat this multiple times. Now how to compute $dW^{[1]}, db^{[1]}, dW^{[2]}, db^{[2]}$? The formulas for computing the derivatives are as follows. First we compute forward propagation for all our training examples:

$Z^{[1]} = W^{[1]}X + b^{[1]}$
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

Then we compute the backward propagation over all training examples by using the derivatives:

$Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$
$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$

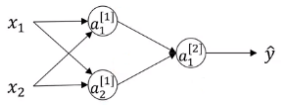
$dZ^{[1]} = W^{[2]T} dZ^{[2]} \times g'^{[1]}(Z^{[1]})$ (element-wise product)
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$
$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

1.5.10 Backpropagation intuition

Our computation graph for a neural network with one hidden layer looks as follows (see Figure 10). With backpropagation the parameters are computed in the following order: $da^{[2]} \rightarrow dz^{[2]} \rightarrow dW^{[2]} \rightarrow db^{[2]} \rightarrow da^{[1]} \rightarrow dz^{[1]} \rightarrow dW^{[1]} \rightarrow db^{[1]}$. In supervised learning we don't take the derivative of x, because these values are fixed.

1.5.11 Random Initialization

When you train a neural network it is important to initialize the parameters randomly. When all weights take on value 0, then the nodes in the hidden unit will be identical/symmetric. Therefore, after every single iteration, both units in the hidden layer will exert the same activation function. You can initialize your weight for the example network below by the following line of code:



$$W^{[1]} = np.random.randn((2,2)) \times 0.01$$
$$b^{[1]} = np.zero((2,1))$$
$$W^{[2]} = np.random.randn((1,2)) \times 0.01$$
$$b^{[1]} = 0$$

The reason for multiplying with a small number is when you take a large value, your activation function will be saturated (slope of 0), which slows down the rate of learning.

1.6 Deep Neural Network

1.6.1 Deep L-layer neural network

In Figure 11 different types of neural networks are shown. The logistic regression model is a more "shallow" model compared to a 5-hidden layer network which is considered a "deep" model. For the following sections L is defined as the number of layers, $n^{[l]}$ the number of nodes in layer l and $a^{[l]}$ the activations in layer l. For further notations see section 6.

1.6.2 Forward Propagation in a Deep Network

First we'll discuss how forward propagation works on a single example

- Layer 1: $z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$ with $a^{[1]} = g^{[1]}(z^{[1]})$
- Layer 2: $z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$ with $a^{[2]} = g^{[2]}(z^{[2]})$
- Layer 4: $z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$ with $a^{[4]} = g^{[4]}(z^{[4]})$
- *General rule*: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$ with $a^{[l]} = g^{[l]}(z^{[l]})$

The following computes it for the entire training set in a vectorized manner:

- Layer 1: $Z^{[1]} = W^{[1]}Z^{[0]} + b^{[1]}$ with $A^{[1]} = g^{[1]}(Z^{[1]})$
- Layer 2: $Z^{[2]} = W^{[2]}Z^{[1]} + b^{[2]}$ with $A^{[2]} = g^{[2]}(Z^{[2]})$
- Layer 4: $\hat{Y} = g(Z^{[4]}) = A^{[4]}$
- *General rule*: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ with $A^{[l]} = g^{[l]}(Z^{[l]})$

This is one place where you have to use an explicit for-loop that computes the activations for the different layers.

1.6.3 Getting your matrix dimensions right

When you implement a deep learning network, it is important to realise the dimensions of the different layers. The dimensions of parameters in the different layers when implementing forward propagation for one example are as follows:

- $z^{[l]}, a^{[l]} = (n^{[l]}, 1)$ (equal to dimensions of $dz^{[l]}$ and $da^{[l]}$).
- $x^{[l]} : (n^{[0]}, 1)$

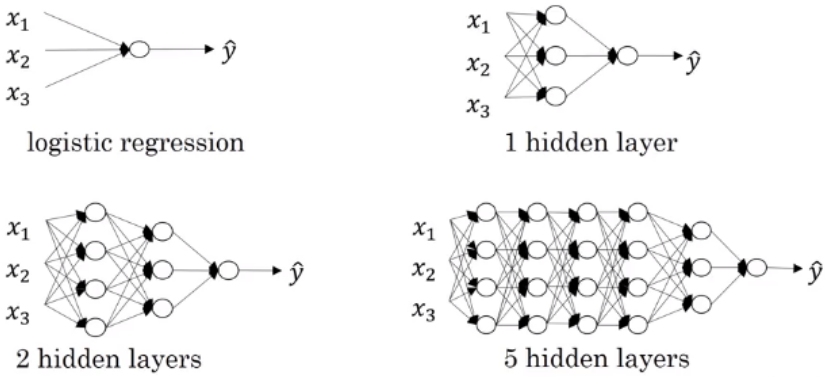


Figure 11: Different types of neural networks

- $w^{[l]} : (n^{[l]}, n^{[l-1]})$ (equal to dimensions of $dw^{[l]}$)
- $b^{[l]} : (n^{[l]}, 1)$ (equal to dimensions of $db^{[l]}$)

When applying a vectorized implementation over the entire training set, the dimensions will alter slightly:

- $Z^{[l]}, A^{[l]} = (n^{[l]}, m)$ (when $l = 0$, then $A^{[0]} = X = (n^{[0]}, m)$(equal to dimensions of $dZ^{[l]}$ and $dA^{[l]}$).
- $X^{[l]} : (n^{[0]}, m)$
- $W^{[l]} : (n^{[l]}, n^{[l-1]})$ (equal to dimensions of $dw^{[l]}$)
- $b^{[l]} : (n^{[l]}, m)$ (equal to dimensions of $db^{[l]}$)

1.6.4 Why deep representations?

Now why are multiple layers needed in deep neural networks? The different layers can learn different features. The structured way of implementing a network enables different layers that process different levels of complexity. The early layers learn "simple" features whereas the later layers learn more "complex" features. It is hypothesized that the human brain processes information in a similar way. Informally, based on the circuit theory, there are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

1.6.5 Building blocks of deep neural networks

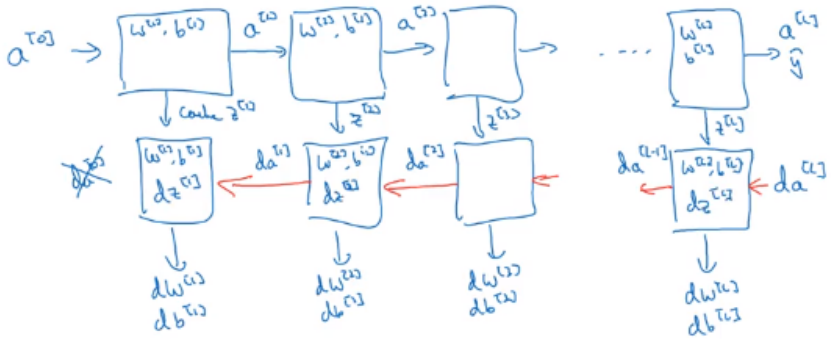


Figure 12: Forward and backward implementation functions

1.6.6 Forward and Backward Propagation

Let's see how to implement the forward and backward propagation function discussed in the previous section.

- For forward propagation for layer l:
  - Input $a^{[l-1]}$
  - Output $a^{[1]}$, cache $z^{[1]}$ (and $w^{[1]}, b^{[1]}$)
  - Vectorized: $Z^{[1]} = W^{[1]}A^{[1-1]} + b^{[1]}$ with $A^{[1]} = g^{[1]}(Z^{[1]})$
- For backward propagation for layer l:
  - Input $da^{[l]}$
  - Output $da^{[1-1]}$, $dW^{[1]}, db^{[1]}$
  - Vectorized:
    $dZ^{[1]} = dA^{[1]}g^{\cdot[1]}(Z^{[1]})$
    $dW^{[1]} = \frac{1}{m}dZ^{[1]}A^{[1-1]T}$
    $db^{[1]} = \frac{1}{m}np.sum(dZ^{[1]}, axis = 1, keepdims = \text{“}True\text{“})$
    $dA^{[1-1]} = W^{[1]T}dZ^{[1]}$

1.6.7 Parameters vs Hyperparameters

The parameters you model are $W^{[1:L]}$ and $b^{[1:L]}$, hyperparameters include the learning rate (α), the number of iterations, the number of hidden layers (L), the number of hidden units ($n^{[l]}$) and the choice of activation functions. These hyperparameters determine and control the final values of $W^{[1:L]}$ and $b^{[1:L]}$. Subsequently, applied deep learning is a very empirical process where you have an idea, you code it and experiment to determine whether the implemented network works well.

2 Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

2.1 Setting up your Machine Learning Application

2.1.1 Train/Dev/Test sets

Applied ML is a highly iterative process, where multiple aspects determine how well your code works (e.g. number of layers, number of hidden units, learning rates, activation functions etc.). Machine learning applications include NLP (natural language processing), speech, structural data (e.g. advertisement, search engines, security, logistics). One way to improve your algorithm is to divide your data in the following three sets:

- Training set
- Cross-validation/Development set (abbreviated with "dev")
- Test set

In the previous era, the proportion of each set was about 60/20/20%. Currently however, big data sets are divided a bit differently. How this division exactly looks like depends on the algorithm you want to implement. Moreover, a increasingly applied strategy is "mismatched train/test distribution", where the training and test set come from different distributions. Finally, remember that it is okay if you don't have a test set (only development set).

2.1.2 Bias/Variance

We'll now discuss the terms bias and variance, where bias refers to underfitting data and variance refers to overfitting the data. The two key numbers to observe are the train set error and the dev set error:

- An algorithm has a bias when both the train set error and and the dev set error are *high* ($\approx 15\%$).
- An algorithm has a variance when the train set error is *low* ($\approx 1\%$) and the dev set error is *high* ($\approx 15\%$).

In this case it is assumed that human error would be approximately 0%. When human error would be higher ($\approx 15\%$), then the train and dev set error will be higher as well.

2.1.3 Basis Recipe for Machine Learning

After you have initially implemented a model, first determine whether the algorithm has a high bias. When this is the case, try a bigger network, longer training and/or optimization algorithms (will be discussed later) so that the training set will fit well. Consequently, determine whether the algorithm has a high variance. When this is the case, solve this by feeding more data to the network, apply regularization and/or CNN architecture. The dev set can be used to determine when the implementation problem is due to high bias or variance. A broadly discussed topic is "bias/variance tradeoff", which is the problem of simultaneously minimizing the bias and variance. Nowadays, however, there are some techniques to minimize one of the two separately.

2.2 Regularizing your neural network

2.2.1 Regularization

We'll discuss regularization in terms of logistic regression. The cost function for logistic regression was as follows:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{2}$$

The regularized form is as follows:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_2^2 \tag{10}$$

where you apply L_2 regularization. L_2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

$$L_2 \textit{ regularization: } ||w||_2^2 = \Sigma_{j=1}^{n_x} w_j^2 = w^T w \textit{ (squared euclidian norm)} \tag{11}$$

Why do you regularize only the parameter w and not b as well? This is because w is usually a high dimensional factor, whereas b is a single number. Consequently, theoretically, you could add the regularization term for b as well, but in practice this does not make a large difference. Moreover, you could also add L_1 regularization:

$$L_1 \textit{ regularization: } = \frac{\lambda}{m}\Sigma_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}||w||_1 \tag{12}$$

In this case, w will be sparse, meaning that the w vector will have a lot of zeros. In these equations λ is the regularization parameter (denote as "lambd", because "lambda" is a fixed value in Python). Now, in a neural network the cost function looks as follows:

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{13}$$

With regularization, the function looks as follows:

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\Sigma_{l=1}^{L}||w^{[l]}||^2 \tag{14}$$

where

$$||w^{[l]}||^2 = \Sigma_{i=1}^{n^{[l-1]}}\Sigma_{j=1}^{n^{[1]}}(w_{ij}^{[l]})_F^2 \tag{15}$$

This is also called the "Frobenius norm". Now how do you apply gradient descent with regularization?

$$dw^{[l]} = \textit{ (from backprop) } + \frac{\lambda}{m}w^{[l]} \tag{16}$$

where $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$. This is also called "weight decay", because you also multiply dw by $1 - \alpha\frac{\lambda}{m}$, which always has a value smaller than 1.

2.2.2 Why regularization reduces overfitting?

Now how does regularization prevent overfitting/high variance? When you add the regularization term (and set λ to a high value), you penalize w resulting in values of w that are close to 0. Eventually, you end up with a smaller/less complex neural network that is less prone to overfitting. Moreover, when you take the $tanh(z)$, you end up with a linearised function when you have values of z (and consequently, values of w) that are relatively small. A linearised function can only fit simple algorithms (as opposed to non-linear functions).

2.2.3 Dropout Regularization

Another powerful technique to minimize overfitting is dropout regularization. With dropout you randomly shut down some neurons in each iteration. In other words, you go through the layers and eliminate nodes on e.g a 50/50% percent change. Subsequently, you run the algorithm with the diminished network and compute the cost. The idea behind dropout is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time. Now how do you implement dropout? The most common technique is called the "inverted dropout" (example layer 3):

- $d3 = np.rnadom.rand(a3.shape[0], a3.shape[1]) < keep.prob$ where $keep.prob$ is the chance that nodes will be eliminated (e.g. $keep.prob = 0.8$.
- $a3 = np.multiply(a3, d3)$ (element-wise multiplication).
- $a3/ = keep.prob$ (for scaling):
  - Let's say you have 50 units in the third layer and $keep.prob = 0.8$ you have on average 10 units shut off. This has to be divided by the $keep.prob$ to not reduce the expected value $z^{[3]}$.

With every iteration you implement the hidden units again (repeat the above lines of code). Important note: at test time you don't apply dropout because you don't want your output to be random.

2.2.4 Understanding Dropout

Why does this technique work as well as regularization? One reason is that you can't rely on any one feature, so spreading weights is a good approach (e.g. shrinking the weights). Moreover, it is possible to alter $keep.prob$ per layer when you're worried that the different layers are differently thrown for overfitting. One big downside of regularization dropout is that the cost function isn't well defined and harder to calculate. Therefore, when computing a graph for J against the number of iterations it is less informative.

2.2.5 Other regularization methods

When you have overfitting data, you can apply other regularization techniques that increase the data on your dataset and can therefore improve your algorithm. A few techniques are flipping images horizontally, taking different crops of the image, apply random distortions, rotations etc. Moreover, you can also apply "early stopping", where you stop iterating through the algorithm when the dev set error is at the lowest point. By stopping halfway will yield a mid-size $||W||_F^2$ but will prevent your algorithm from overfitting too much (i.e. low training set error and high dev set error). A principal by which implementing a deep learning network works is called "orthogonalization", where you focus on one task at the time:

1. Optimize cost function $\rightarrow$ gradient descent,...
2. Not overfit $\rightarrow$ regularization,...

The main downside you can not work on minimizing the cost function and prevent overfitting separately. Rather than using early stopping is "L_2-regularization" where you can iterate through the algorithm as many times as desired. A downside for this is that the value of λ is not known, which demands running the algorithm multiple times to determine the desired value. This we'll be discussed later in greater detail.

2.3 Setting up your optimization problem

2.3.1 Normalizing inputs

Scaling your dataset includes subtract the mean. You end up with a dataset with a mean around approximately 0.

$$\begin{aligned} \mu &= \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} \\ x &:= x - \mu \end{aligned} \tag{17}$$

And next apply normalized variance to scale your different features. When you don't scale your features, the values may lay in different ranges which ultimately will slow down the learning algorithm.

$$\begin{aligned} \sigma^2 &= \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} .* 2 \\ x/ &= \sigma^2 \end{aligned} \tag{18}$$

2.3.2 Vanishing/Exploding gradients

Vanishing/Exploding gradients refers to the problem that when training a deep learning algorithm the derivatives can take on very large or very small values. When you have a deep learning networks with many hidden units you either get exploding variations ($w^{[l]} > 1$) or vanishing ($w^{[l]} < 1$) derivatives/gradients. This makes it difficult for gradient descent to converge. It turns out that their is a partial solution where you carefully decide the initialization of the weights.

2.3.3 Weight Initialization for Deep Networks

To prevent exploding/vanishing gradients, you implement the weights with the following lines of code:

- When you have multiple input features where $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n$ (where $b = 0$).
  - when n is large $\rightarrow$ smaller w_i, and $Var(w:) = \frac{1}{n}$ (when using a ReLU function $Var(w:) = \frac{2}{n}$)
- $w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$.
  - when using a ReLU function $w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$.
  - when using a tanh function $w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$.

This implementation makes sure that the weights are set around the value 1, so vanishing/exploding gradients will occur less quickly.

2.3.4 Numerical approximation of gradients

Gradient checking will assure that the backpropagation works as intended. We can approximate the derivative of our cost with computing the derivative at a certain point in the range of $f(\Theta - \epsilon) : f(\Theta + \epsilon)$, where ϵ is a small value (e.g. $\epsilon = 0.01$)(two-sided difference).

$$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta + \epsilon)}{2\epsilon} \tag{19}$$

You can also implement one-sided difference ($\frac{J(\Theta+\epsilon)-J(\Theta)}{\epsilon}$). However, this technique has a much lower accuracy.

2.3.5 Gradient checking

Gradient checking is a technique that helps you discover bugs in your algorithm.

1. Take $W^{[1]}, b^{[L]}, \ldots, W^{[1]}, b^{[L]}$ and concatenate into a big vector Θ.
2. Take $dW^{[1]}, db^{[L]}, \ldots, dW^{[1]}, db^{[L]}$ and reshape into a big vector $d\Theta$ (same dimension as Θ).
3. Now, is $d\Theta$ the gradient descent of $J(\Theta)$? This is how you implement gradient checking:
   - For each i:
     $d\Theta^{[i]}_{approx} = \frac{J(\Theta_1,\Theta_2,\ldots,\Theta_i+\epsilon)-J(\Theta_1,\Theta_2,\ldots,\Theta_i-\epsilon)}{2\epsilon} \approx d\Theta^{[i]} = \frac{\partial J}{\partial \Theta_{[i]}}$
   - Next you want to know if $d\Theta_{approx} \approx d\Theta$ (vectors have same dimensions).
   - Compute/Check Euclidean distance and normalize: $\frac{||d\Theta_{approx}-d\Theta||_2}{||d\Theta_{approx}||_2+||d\Theta||_2}$.
   - $\epsilon = 10^{-7}$, when the gradient check is smaller than ϵ it is fine, when the value is higher there is a chance you have a bug in your algorithm.

2.3.6 Gradient Checking Implementation Notes

Here are some implementation notes for gradient checking:

- Don't use in training - only to debug.
- If algorithm fails gradient check, loot at components to try to identify bug.
- Remember/Include regularization.
- Doesn't work with dropout (first compute gradient check without dropout ($keep.prob = 1$) and next apply dropout).
- Run at random initialization; perhaps again after some training.

2.4 Optimization algorithms

2.4.1 Mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples. However, when you have a large dataset it is computationally very expensive to iterate over all you're training examples before applying gradient descent. As a solution you can use mini-batch gradient descent, where you divide your training set in mini-batch (e.g. 1000 training examples per mini-batch), where $x^{\{i\}}$ and $y^{\{i\}}$ are denoted as the i^{th} mini-batch. Next is implementing a vectorized mini-batch gradient descent $(X^{\{i\}}, y^{\{i\}})$:

for $t = 1, \ldots, 5000$ (size of the mini-batch)
  Forward propagation on $X^{\{t\}}$:
  $Z^{[1]} = W^{[l]}X^{\{t\}} + b^{[1]}$
  $A^{[1]} = g^{[1]}(Z^{[1]})$
  $\vdots$
  $Z^{[L]} = W^{[L]}X^{\{t\}} + b^{[L]}$
  $A^{[L]} = g^{[L]}(Z^{[L]})$ (vectorized implementation)
  Compute cost $J^{\{t\}} = \frac{1}{1000}\Sigma^{l}_{i=1}L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000}\Sigma_l||w^{[l]}||^2_F$ (only using $\{X^{\{i\}}, y^{\{i\}}\}$)
}

This is "1 epoch" where you pass through the training set one time. You can add another for loop to iterate over the training set multiple times. Building mini-batches includes two steps:

1. Shuffle: Create a shuffled version of the training set (X, Y) as shown below. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the i^{th} column of X is the example corresponding to the i^{th} label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.
2. Partition: Partition the shuffled (X, Y) into mini-batches of size $mini\_batch\_size$ (for example 64). Note that the number of training examples is not always divisible by $mini\_batch\_size$. The last mini batch might be smaller, but you don't need to worry about this.

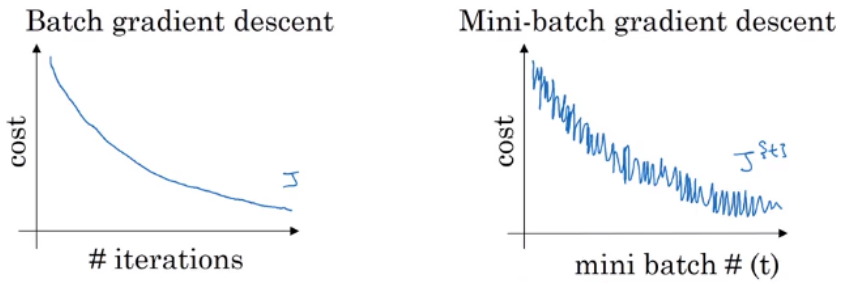


Figure 13: Cost J against number of iterations for batch and mini-batch gradient descent

2.4.2 Understanding mini-batch gradient descent

With batch gradient descent you expect that the cost J will decrease with the number of iterations. On the other hand, with mini-batch descent, the cost J may not decrease with every iteration (see Figure 13). Moreover, you also need to choose the size of your mini-batch:

- When the size of the mini-batch is equal to m, you're implementing *batch gradient descent*.
  - Batch gradient descent we'll be able to take relatively large gradient steps. A downside is that gradient descent will take a long time per iteration (large memory required). In other words, convergence to a global minimum will take a long time.
- When the size of the mini-batch is equal to 1 you're implementing *stochastic gradient descent*.
  - Stochastic gradient descent we'll be able to only take relatively small gradient steps. This gradient descent will never converge but we'll be in close proximity of the global optimum.
- In practice, you take a size between 1 and m which will results in fast learning. With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

Generally, if the dataset is small ($m \approx 2000$) then batch gradient is the best option. When deciding to use mini-batch gradient descent, standard sizes include 64, 128, 256, 512, 1024 (n^i, fits CPU/GPU memory best).

2.4.3 Exponentially weighed averages

Exponentially weighted (moving) average (EWMA) is a calculation to analyse data points by creating series of averages of different subsets of the full data set. The general formula for calculating moving averages is: $v_t = \beta v_{t-1} + (1-\beta)\Theta_t$. Subsequently, the amount of data you are averaging over can be computed with $\frac{1}{1-\beta}$, where a large value for β corresponds to a large number of data (which results in a smoother and slower adapting curve when plotted in a figure). A weighted moving average allows us to smooth out trends over time so it's easier to get a read on how the trend is progressing while giving us the ability to add a weighting factor so that some of the data will be treated as "more important" than other data

2.4.4 Understanding exponentially weighted averages

To recap, $v_t = \beta v_{t-1} + (1-\beta)\Theta_t$. Now discuss the effect of the value of β.

$$
\begin{aligned}
v_{100} &= 0.9v_{99} + 0.1\Theta_{100} \\
v_{99} &= 0.9v_{98} + 0.1\Theta_{99} \\
v_{98} &= 0.9v_{97} + 0.1\Theta_{98} \\
&\vdots
\end{aligned}
$$

Then the following applies:

$$
\begin{aligned}
v_{100} &= 0.1\Theta_{100} + 0.9v_{99} \\
v_{100} &= 0.1\Theta_{100} + 0.9(0.1\Theta_{99} + 0.9v_{98}) \\
v_{100} &= 0.1\Theta_{100} + 0.9(0.1\Theta_{99} + 0.9(0.1\Theta_{98} + 0.9v_{97})) \\
v_{100} &= 0.1\Theta_{100} + 0.1 \times 0.9 \times \Theta_{99} + 0.1 \times (0.9)^2\Theta_{98} + 0.1 \times (0.9)^2\Theta_{97} + 0.1 \times (0.9)^4\Theta_{96} + \ldots
\end{aligned}
$$

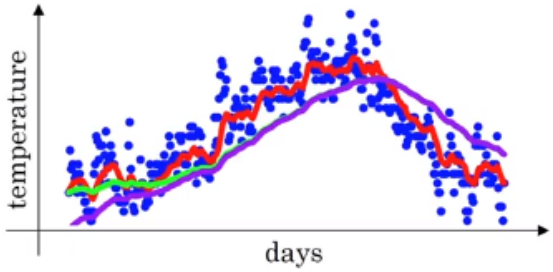


Figure 14: Exponentially weighted average without (purple) and with (green) bias correction with $v_t = \beta v_{t-1} + (1-\beta)\Theta_t$

One way to compute this in a graph, is an exponentially decaying function, referring to an object that decreases at a rate proportional to its current value. Rather than decreasing linearly, the weight for an EWMA decreases exponentially for each time period further in the past. Additionally, the result of an EWMA is cumulative because it contains the previously calculated EWMA in its calculation of the current EWMA. Because of this, all the data values have some contribution in the result, though that contribution diminishes as each next period is calculated. When implementing exponentially weighted averages:

$V_\Theta := 0$
Repeat {
$V_\Theta = \beta V_\Theta + (1-\beta)\Theta_t$
}

2.4.5 Bias correction in exponentially weighted averages

When the value for β is relatively high (e.g. $\beta = 0.98$) then the curve of v_t will be smooth. When using the formula $v_t = \beta v_{t-1} + (1-\beta)\Theta_t$ you'll notice that the curve will start at a value close to zero (see Figure 14, purple line), because you initially set V_0 to 0, the first averaged value will be relatively low. To solve this you can perform bias correction, which takes $V_t = \frac{V_t}{1-\beta_t}$ (instead of $V_t = V_t$). Consequently, as t becomes large, β^t will approach 0 and bias correction will not make much of a difference.

2.4.6 Gradient descent with momentum

When you're gradient descent takes on an zigzag-like path towards the global minimum (see Figure 15), you eventually want slower learning on the y-axis and faster learning on the x-axis. This could be achieved by using momentum. Momentum remembers to update the parameters at each iteration, and determines the next update as a linear combination of the gradient and the previous update. In other words, momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable v . Formally, this will be the exponentially weighted average of the gradient on previous steps. You can also think of v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill. The implementation is as follows:

On iteration t:
Compute dW and db on current mini-batch.
$v_{dW} = \beta v_{dW} + (1-\beta)dW$
$v_{db} = \beta v_{db} + (1-\beta)db$
$W := W - \alpha v_{dW}$, $b := b - \alpha v_{db}$

This algorithm smooths out the steps of gradient descent by taking the averages over the different batches. Consequently, the average of the y-axis will be close to zero therefore, gradient descent will be slower, whereas for the axis, gradient descent will be faster. The hyperparameters include α, β where a frequently used value for β is 0.9 (initial values for v_{dw} and v_{db} are set to 0). Moreover, sometimes you see that the term $(1-\beta)$ is omitted. In practice this does not make much of a difference. Important notes:

- The velocity v is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps
- If $\beta = 0$, then this just becomes standard gradient descent without momentum.

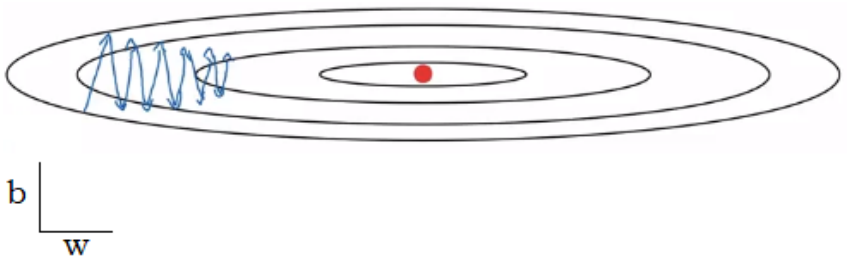


Figure 15: An example of gradient descent where the path to convergence has a zigzag-like movement in the vertical direction resulting in slower learning. To speed up gradient descent algorithm that can be implemented are *momentum* and *RSMprop*.

Now how to choose β?

- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.
- Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

So to summarize, momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent. You have to tune a momentum hyperparameter β and a learning rate α.

2.4.7 RMSprop

Another algorithm used to speed up gradient descent is RMSprop (Root Mean Square Propagation). Recall our previous examples where you get a zigzag-like path towards a global minimum in the vertical direction (see Figure 15). As mentioned previous, the goal is to speed up convergence in the horizontal direction (denoted as b) and slow it down in the vertical direction (denoted as w). The RMSprop algorithm is implemented as follows:

On iteration t:
Compute dW and db on current mini-batch.
$s_{dW} = \beta s_{dW} + (1-\beta)dW^2$ (element-wise)
$s_{db} = \beta s_{db} + (1-\beta)db^2$ (element-wise)
$W := W - \alpha \frac{dW}{\sqrt{s_{dW}}}$, $b := b - \alpha \frac{db}{\sqrt{s_{db}}}$

What this algorithm does is keeping an exponentially weighted average of the squares of the derivatives. In this example, s_{dW} is quite small therefore gradient descent for dW will be small and therefore slow in the horizontal direction. In contrast, the value for s_{dW} will be quite large resulting in a large value for db, and consequently a faster gradient descent in the vertical direction. This enables you to choose a higher learning rate α without the risk of divergence. In practice, W and b are multidimensional elements, therefore this example is extremely simplified. Notation note, to avoid confusion, the β for RMSprop is from here on out denoted as β_2. To add numerical stability (i.e. to avoid dividing by 0), a hyperparameter is added to the gradient descent: $\frac{dW}{\sqrt{s_{dW}}+\epsilon}$ and $\frac{db}{\sqrt{s_{db}}+\epsilon}$, where ϵ takes on a small value (e.g. $\epsilon = 10^{-8}$).

2.4.8 Adam optimization algorithm

Frequently, optimization algorithms work on specific problems and can not be extrapolated very well. However, some optimization algorithms work on a wide range of deep learning architectures, for instance the adam optimization algorithm. This algorithm is implemented as follows:

$v_{dW} = 0$, $s_{dW} = 0$, $v_{db} = 0$, $s_{db} = 0$

On iteration t:
Compute dW and db on current mini-batch.
$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$, $v_{db} = \beta_1 v_{db} + (1-\beta_1)db$

$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2$, $s_{dw} = \beta_2 s_{db} + (1-\beta_2)db^2$
$V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t}$, $V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$
$S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}$, $S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$
$W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}}+\epsilon}$, $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{s_{db}^{corrected}}+\epsilon}$

This algorithm combines the effect of gradient descent with momentum together with gradient descent with RMSprop. This is a commonly used learning algorithm that is proven to be very effective for many different neural networks of a very wide variety of architectures. Now, this algorithm has the following hyperparameters:

- α
- $\beta_1 = 0.9$ (momentum $\rightarrow dW$)
- $\beta_2 = 0.9999$ (RMSprop $\rightarrow dW^2$)
- $\epsilon = 10^{-8}$ does not needs to be tuned.

Most practitioners set β_1, β_2 and ϵ to fixed values and vary the learning rate α. Now where does the term "adam" stands for? Adam stand for "Adaptive moment estimation", where the first moment computes the exponentially weighted mean of the derivatives (momentum) and second moment computes the exponentially weighted average of the squares (RSMprop). So to summarize, adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSprop (described in lecture) and Momentum. How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

Some advantages of adam include relatively low memory requirements (though higher than gradient descent and gradient descent with momentum). Moreover, adam usually works well even with little tuning of hyperparameters (except α).

2.4.9 Learning rate decay

During the initial steps of learning you can have a large learning rate α, whereas when you get closer to the optimum, you need your learning rate α to be smaller. This can be solved with learning rate decay. Learning rate decay can be implement as follows:

α_0, $decay.rate = 1$

1 epoch = 1 pass through the entire data set
$\alpha = \frac{1}{1+\ decay\ rate\ *\ epoch\text{-}num} * \alpha_0$

Now the learning rate gradually decrease with every epoch. You can try a variety of values for you hyperparameters (α_0 and $decar.rate$). Other learning rate decay methods are:

- Using formula's
  - $\alpha = 0.95^{epoch.num} * \alpha_0$ (exponentially decay)
  - $\alpha = \frac{k}{\sqrt{epoch.num}} * \alpha_0$ or $\alpha = \frac{k}{\sqrt{t}} * \alpha_0$
  - Learning rate which decreases in discrete steps
- Manually decay where you tune α by hand. This only works when you're using small models.

2.4.10 The problem of local optima

Global and local optima include values for the parameters where gradient descent is equal to 0. The early view considered local optima as values for the parameters that are suboptimal, where a model has multiple local optima and one global optimum (see Figure 16a). However, nowadays it is believed that most points of zero gradients are saddle points, which is a point where gradient descent is equal to 0, while it is clearly not a local minimum as the cost function J has a smaller function value (see Figure 16b). One problem that can slow down the algorithm is a plateau, which is a region where the derivative is close to 0 for a long time (see Figure 16c). While it is unlikely to get stuck in bad local optima, plateaus can make your learning very slow. However, optimization algorithms such as adam can solve this.

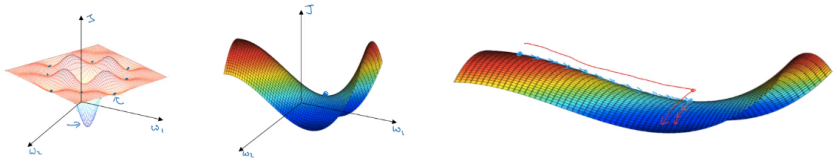


Figure 16: Cost J plotted against the values of the different parameters. (A) Classical view where an algorithm has multiple local minima and one global minimum. (B) Current view where the cost function has saddle point. (C) Plateau during which the cost J decreases very slowly with every iteration.

2.5 Hyperparameter tuning

2.5.1 Tuning process

We'll now discuss some guidelines for how to systematically organize your hyperparameter tuning process, which hopefully will make it more efficient to converge on a good setting of the hyperparameters. Here is a list with hyperparameters ordered by importance of tuning:

1. The value of the learning rate α.
2. $\beta = 0.9$, the number of hidden units and the mini-batch size.
3. The number of layers and the learning rate decay.
4. $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Now how do you select the range of values to explore? In the early days it was common practice to sample the value in a grid of hyperparameter 1 and hyperparameter 2. This practice works best when the number of hyperparameter is relatively small. In deep learning however, it is recommended to randomly choose the values, so you explore a greater set of values for the different parameters (see Figure 17). Moreover, once you've set a range of values that work well, you can tune in to explore further in the selected range.

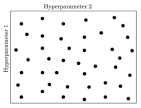


Figure 17: Grid with randomly chosen values for hyperparameter 1 and hyperparameter 2

2.5.2 Using an appropriate scale to pick hyperparameters

When picking hyperparameters randomly, you still need to decide the range in which you choose the values. Often a linear scale is not the appropriate scale for your hyperparameters. Alternatively, a log scale often gives much better values. Implementing a range of values scaled by a log can be done as follows. You first assign $r = -4 * np.random.rand()$ which computes a value between -4 and 0. Subsequently, you assign $\alpha = 10^r$ which computes a range of values between 10^{-4} and 10^0. Finally, one other tricky case is choosing hyperparameters for exponentially weighted averages. Typically, for β you want to have a range of values between 0.9 and 0.999, and consequently the range of $(1 - \beta)$ must lie between 0.1 (10^{-1}) and 0.001 (10^{-3}). What you do is $r \in [-3, -1]$, where $1 - \beta = 10^r$, and $\beta = 1 - 10^r$. Now why is it so bad to sample on a linear scale? When β is close to 1, the sensitivity of the results changes, even with small changes to β.

2.5.3 Hyperparameters tuning in practice: Pandas vs. Caviar

It is important to re-test your hyperparameters occasionally. One technique is babysitting one model where you compute the cost J over different days ("Panda strategy"). Another approach is training many models in parallel, where in the end, you choose the one that works most efficient ("Caviar strategy").

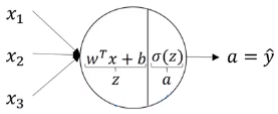


Figure 18: Neural network representation

2.6 Batch Normalization

2.6.1 Normalizing activations in a network

When training a model such as logistic regression, normalizing can speed up your algorithm:

$$\mu = \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} \qquad x := x - \mu \tag{17}$$

$$\sigma^2 = \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} .* 2 \qquad x/ = \sigma^2 \tag{18}$$

When you have a multi-layered network you want to normalize the activation $a^{[i]}$ values of every hidden layer as to train the weights $W^{[i]}$ and biases $b^{[i]}$ faster. In practice normalizing $z^{[i]}$ is applied more frequently. Implementing batch norm goes as follows:

Given some intermediate values $z^{(i)}$ for a specific layer in a NN

$\mu = \frac{1}{m}\Sigma_i z^{(i)}$
$\sigma^2 = \frac{1}{m}\Sigma_i (z^{(i)} - \mu)^{(2)}$
$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$

Here, γ and β are parameters which values you can change. This allows you to have different hidden layer values (to distinguish between different layers in the network) instead of altering only the input layer.

2.6.2 Fitting Bach Norm into a neural network

As mentioned earlier, you can distinguish two different elements in the nodes in the different layers in a neural network, namely $z^{[i]}$ and $a^{[i]}$ (see Figure 18). If you were to apply batch norm, you apply batch norm to $z^{[1]}$ giving you a new normalized value which you feed into the activation function $a^{[1]}$:

$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{(1)} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{(1)}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{(2)} \rightarrow a^{[2]} = g^{[2]}(\tilde{z}^{(2)}) \dots$$

So the parameters of your network will be $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[l]}, b^{[l]}$ with additional parameters $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[l]}, \gamma^{[l]}$. For these parameters you can use gradient descent to update the values with every iteration. In practice (deep learning frameworks), you don't need to implement all these values manually. In practice, batch norm is applied using mini-batches

$$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{(1)} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{(1)}) \dots$$

Where you apply gradient descent followed by the subsequent mini-batch:

$$X^{\{2\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{(1)} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{(1)}) \dots$$

And so on...

$$X^{\{3\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{(1)} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{(1)}) \dots$$

One detail is concerns setting the parameters$(w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]})$. As was discussed earlier, $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$. However,

what mini-batch gradient descent does is cancelling out the constants ($b^{[l]}$) by the mean subtraction step. So using batch norm you eliminate the constant giving $z^{[l]} = W^{[l]}a^{[l-1]}$, where a constant β is added in a later stage. So let's put this altogether and describe how you implement gradient descent using batch norm (assuming you use mini-batch):

for $t = 1, \ldots, numMiniBatches$
  Compute forward propagation on $X^{\{t\}}$
    In each hidden layer, use batch norm to replace $z^{[l]}$ with $\tilde{z}^{(l)}$.
  Use backpropagation to compute $dW^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$
  Update parameters:
    $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$
    $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$
    $\gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$

This also works with momentum, RMSprop and Adam.

2.6.3 Why does Batch Norm work?

"Covariate shift" refers to the change in the distribution of network activations due to the change in network parameters during training. As you know, in neural networks, the output of the first layer feeds into the second layer, the output of the second layer feeds into the third, and so on. When the parameters of a layer change, so does the distribution of inputs to subsequent layers. These shifts in input distributions can be problematic for neural networks, especially deep neural networks that could have a large number of layers. Batch normalization is a method intended to mitigate internal covariate shift for neural networks. Batch normalization limits the amount to which updating the parameters in the earlier layers can affect the distribution of values that a later layer receives and therefore has to learn on. And so, batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on. Finally, here are some notes on batch normalization as regularization:

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that mini-batch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect. By adding noise the reliance of specific units is reduced.

2.6.4 Bach Norm at test time

Batch norm processes your data one mini batch at a time, but the test time you might need to process the examples one at a time. Let's see how you can adapt your network to do that. To recap:

$\mu = \frac{1}{m}\Sigma_i z^{(i)}$
$\sigma^2 = \frac{1}{m}\Sigma_i (z^{(i)} - \mu)^{(2)}$
$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$

Notice that μ^2 and σ^2 which you need for this scaling calculation are computed on the entire mini-batch. But the test time you might not have a mini-batch of a large set of examples to process at the same time. So, you need some different way of coming up with μ^2 and σ^2. What is done in order to apply a neural network at test time is coming up with some separate estimate of μ^2 and σ^2 based on your training set. During a typical implementation of batch norm, what you do is estimate this using a exponentially weighted average where the average is across the mini-batches.

2.7 Mutli-class classification

2.7.1 Softmax Regression

Till this far, we discussed only binary classification problems. We'll now discuss multi-class classification, where C denotes the number of classes ($n^{[L]} = C$) and the dimension of $\hat{y}$ is $(C, 1)$. The standard model for getting your network to do this uses what's called a Softmax layer, and the output layer in order to generate these multi-class outputs. Say you have $z^{[L]} = W^{[L]}a^{[L-1]}$. Subsequently, you compute the activation function as follows:

1. You create a temporal variable $t = e^{(z^{[L]})}$ with dimensions $(C, 1)$.

2. $a_i^{[L]} = \frac{t_i}{\Sigma_{j=1}^{C} t_j}$ with dimensions $(C, 1)$

For clarification, let's look at an example. Say you have $Z^{[L]} = \begin{pmatrix} 5 \\ 2 \\ -1 \\ 3 \end{pmatrix}$, where $t = \begin{pmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{pmatrix} = \begin{pmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{pmatrix} \cdot \Sigma_{j=1}^{4} t_j = 176.3$ and

$a^{[L]} = \frac{t}{176.3} = \frac{e^{-5}}{176.3} = 0.842$ (highest probability)
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-2}}{176.3} = 0.042$
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-1}}{176.3} = 0.002$
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-3}}{176.3} = 0.114$

So this algorithm takes the vector $z^{[L]}$ and compute the probabilities of the different classes that sum to 1.

2.7.2 Training a softmax classifier

Let's take a look again at the example mentioned earlier. $Z^{[L]} = \begin{pmatrix} 5 \\ 2 \\ -1 \\ 3 \end{pmatrix}$, where $t = \begin{pmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{pmatrix} = \begin{pmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{pmatrix} \cdot \Sigma_{j=1}^{4} t_j = 176.3$

and

$a^{[L]} = \frac{t}{176.3} = \frac{e^{-5}}{176.3} = 0.842$ (highest probability)
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-2}}{176.3} = 0.042$
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-1}}{176.3} = 0.002$
$a^{[L]} = \frac{t}{176.3} = \frac{e^{-3}}{176.3} = 0.114$

A "hard max" layer will look at the elements and assign a 1 to the node with the highest probability and a 0 to the rest of the nodes. So softmax and hard max will give the following output, respectively: $a^{[L]} = \begin{pmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.

Softmax regression generalizes logistic regression to C classes. If $C = 2$, softmax reduces logistic regression. Now how do you train a neural network using a softmax layer? Say you have $y = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ and $\hat{y} = \begin{pmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{pmatrix}$. What the loss function does is look at whatever is the ground true class in your training set, and it tries to make the corresponding probability of that class as high as possible. If you're familiar with maximum likelihood estimation statistics, this turns out to be a form of maximum likelihood estimation.

2.8 Introduction to programming frameworks

2.8.1 Deep learning frameworks

Deep learning frameworks include a wide range of deep learning algorithms with which you can work with and where you don't need to build a deep learning network from scratch. Examples of deep learning frameworks are:

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Important criteria for using specific deep learning frameworks are the ease of programming (development and deployment), running speed and truly open (open source with good governance).

2.8.2 TensorFlow

We'll now discuss the deep learning framework TensorFlow. As an example we have the following cost function $J(w) = w^2 - 10w + 25$. Now we'll use TensorFlow in order to automatically find appropriate values for W and b. The following line of code can be applied:

$import\ numpy\ as\ np$
$import\ tensorflow\ as\ tf$

$coefficients = np.arry([1], [-10], [25])$

$w = tf.Varaible([0], dtype = tf.float32)$
$x = tf.placeholder(tf.float32, [3, 1])$
$cost = x[0][0] * w * *2 + x[1][0] * w + x[2][0]\#(w - 5)^2$
$train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)$
$init = tf.global_variables_initializer()$
$session = tf.Session()$
$sessionrun(init)$
$print(session.run(w))$

$for\ i\ in\ range(1000):$
$\quad session.run(train, feed\_dict = \{x : coefficients\})$
$print(session.run(w))$

An advantage is that TensorFlow has already built in the backpropagation function. Consequently, you don't need to compute this yourself. To use TensorFlow you need to perform the following actions: (1) initialize your variables, (2) create a session and (3) run the operations inside the session. Next, you'll also have to know about placeholders. A placeholder is an object whose value you can specify only later. When you first define a variable you do not have to specify a value for it. A placeholder is simply a variable that you will assign data to only later, when running the session. We say that you feed data to these placeholders when running the session. To summarize:

- TensorFlow is a programming framework used in deep learning
- The two main object classes in TensorFlow are Tensors and Operators.
- When you code in TensorFlow you have to take the following steps:
  - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
  - Create a session
  - Initialize the session
  - Run the session to execute the graph
- You can execute the graph multiple times as you've seen in model()
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.

3 Structuring Machine Learning Projects

3.1 Introduction to ML Strategy

3.1.1 Why ML strategy

Now how to structure your machine learning (ML) project? Let's say, you have build a classifier, but you want to increase the accuracy. Some ideas of how to increase the performance are:

- Collect more data
- Collect more diverse training set
- Train algorithm longer with gradient descent
- Try Adam instead of gradient descent
- Try bigger network
- Try smaller network
- Try dropout
- Add L_2 regularization
- Network architecture
  - Activation functions
  - # hidden units
  - ...

The problem is that you have many options and trying randomly can be time consuming. The following sections explain how you can quickly and effectively pursue ideas to increase the performance of your algorithm.

3.1.2 Orthogonalization

One of the challenged of building ML systems is that there are many variables you can vary. One of the things noticeable about the most effective machine learning people is they are very clear-eyed about what to tune in order to try to achieve one effect. This is a process we call orthogonalization. Orthogonalization refers to distinctive mechanisms controlling separate entities as opposed to one combined mechanisms controlling multiple features. Now how does this translate to machine learning? First, you need to fit the training set well on the cost function. Subsequently, you hope it works well on the dev set and finally on the test set as well ultimately performing well in the real world, where you have distinctive mechanisms to work on the different data sets (e.g. training, dev, test, real world) also known as orthogonalization. So to summarize, for an algorithm to work well you want to use orthogonalization, where you set up distinctive goals ("Chain of assumptions in ML"):

- Fit training set well on cost function (by bigger network, Adam optimization, etc.)
- Fit dev set well on cost function (by regularization, bigger training set, etc.)
- Fit test set well on cost function (by bigger dev set, etc.)
- Performs well in real world (by changing dev set or cost function, etc.)

3.2 Setting up your goal

3.2.1 Single number evaluation metric

Setting up a single number evaluation metric for your problem quickly shows you if the new thing you just tried works better or worse than your last idea. Let's look at en example where you have build two subsequent classifiers A and B that recognizes images of cats, where each classifier has a certain percentage of recall and precision. Here, precision refers to the fraction of the predicted cat images that were recognized correctly, whereas recall refers to the fraction of actual cat images that were recognized correctly. The problem with using precision and recall as your evaluation metric is that if classifier A does better on recall and the classifier B does better on precision, then you're not sure which classifier is better. As a solution, you can use a new metric that combines precision and recall, called the F_1 score which takes the average of precision (P) and recall (R) ($F_1 = \frac{2}{\frac{1}{P}+\frac{1}{R}}$) also called the "Harmonic mean". You measure precision and recall with your dev set. So having a dev set together with a single real number evaluation metric tends to speed up iterating or the iterative process.

3.2.2 Satisficing and Optimizing metric

It's not always easy to combine all the things you care about into a single row number evaluation metric. In those cases it is sometimes useful to set up an optimizing matrix. In such a matrix you have an "optimizing" and a "satisficing" metric. The optimizing metric is the outcome that is most relevant and which you want to increase as much as possible (e.g. accuracy). The satisficing metric is an outcome that needs to be under a specific value (e.g. running time) but has less relevance. More generally, when you have N metrics, you have 1 optimizing metric and $N-1$ satisficing metrics.

3.2.3 Train/dev/test distributions

The way you set up your train/dev/test set determines greatly the progression speed. One important note is to extract your dev and test set from the same distributions by randomly shuffling the data and dividing it in a dev and test set. One guideline is to choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.

3.2.4 Size of the dev and test sets

The old way of splitting data is a train and test set of 70% and 30%, respectively. With in additional dev set the distribution for the train, dev and test set looks more like 60%, 20% and 20%, respectively. With smaller amount of data these distributions are pretty reasonable (up to 10.000 training examples). Nowadays however, people are used to work with much larger data sets (e.g. 1.000.000 examples). With these amounts of data the train, dev and test set take on distributions around 98%, 1% and 1%, respectively. Now let's discuss the size of the test set. The guideline for the test set is to be big enough to give high confidence in the overall performance of your system. Moreover, sometimes it is okay not to have a test set (i.e. only a train and dev set).

3.2.5 When to change dev/test sets and metrics

Let's say you want to build a cat classifier, where as a metric you choose the classification error. You have build to algorithms A and B with an error of 3% and 5%, respectively. Now, you notice that algorithm A is letting through a lot of pornographic images. Form your company points of view, algorithm B is a better algorithm since it doesn't show many pornographic images. Now, your metric and dev set prefer algorithm A, while you and your users prefer algorithm B. This is a sing that you should change your evaluation metric or even your dev and test set. The error here is defined as follows:

$$\frac{1}{m_{dev}} \Sigma_{i=1}^{m_{dev}} L\{y_{pred}^{(i)} \neq y^{(i)}\} \tag{20}$$

This formula basically counts up the number of misclassified examples. The problem with this evaluation metric is that they treat pornographic and non-pornographic images equally but you really want your classifier not mislabel pornographic images. One way to change this evaluation metric is adding a weight term.

$$\frac{1}{m_{dev}} \Sigma_{i=1}^{m_{dev}} w^{(i)} L\{y_{pred}^{(i)} \neq y^{(i)}\} \tag{21}$$

where

$$w^{(i)} = \begin{cases} 1 \text{ if } x^{(i)} \text{ is non-porn} \\ 10 \text{ if } x^{(i)} \text{ is porn} \end{cases} \tag{22}$$

So, if you find that your evaluation metric is not giving the correct rank order preference for what is actually a better algorithm that there's a time to think about defining a new evaluation metric. So far, we've only discussed how to define a metric to evaluate classifiers which is an example of orthogonalization. To formulate this more step-wise:

1. So far we've only discussed how to define a metric to evaluate classifiers (place target)
2. Worry separately about how to do well on this metric (shoot/aim at target).

Moreover, an additional guideline is if doing well on your metric + dev/test set does not correspond to doing well on your application, then change your metric and/or dev/test set.

3.3 Comparing to human-level performance

3.3.1 Why human-level performance?

In the last few years, a lot more machine learning teams have been talking about comparing the machine learning systems to human-level performance. Why is this? First, due to the advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance. Second, it turns out that the workflow of designing and building a machine learning system, is much more efficient when you're trying to do something that humans can also do. So in those settings, it becomes natural to talk about comparing, or trying to mimic human-level performance. On a lot of machine learning tasks progress tends to be relatively rapid as you approach human level performance. However, after a while, the algorithm surpasses human-level performance after which the progress in accuracy slows down. However, the hope is that it achieves some theoretical optimum level of performance. Subsequently, over time, as you keep training the algorithm, the performance approaches but never surpasses some theoretical limit also called the "Bayes optimal error". In other words, humans are quit good at a lot of talks. So long as ML is worse than humans you can:

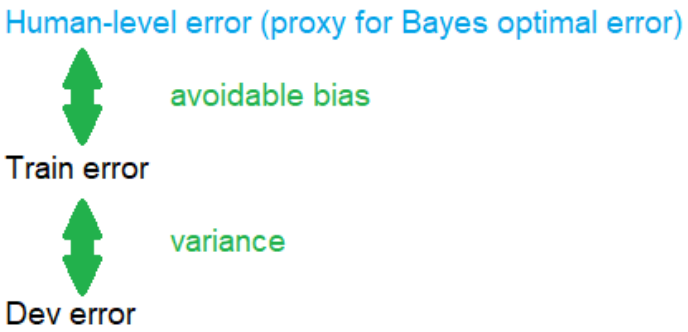


Figure 19: In terms of terminology, we are calling the approximation of Bayes error and the training error the avoidable bias. The difference between the train and dev error is called the variance. So the goal is to perform the training performance in order to reduce the avoidable bias as much as possible.

- Get labelled data from humans
- Gain insight from manual error analysis: Why did a person get this right?
- Better analysis of bias/variance

3.3.2 Avoidable bias

When the train and dev error is much higher than the human error, it is a good strategy to focus on decreasing the bias. In contract, when the train error is similar to the human error, you may want to decrease the difference between the train and dev error by decreasing the variance. So depending on what human-level error is which we assume is close to the Bayes optimal error (human-level error $\approx$ Bayes optimal error), you can choose to focus on the bias or variance reduction tactics. In terms of terminology, we are calling the approximation of Bayes error and the training error the avoidable bias (see Figure 19). Moreover, the difference between the train and dev error is called the variance. So the goal is to perform the training performance in order to reduce the avoidable bias as much as possible. "Avoidable" refers to a certain amount of error you cannot get below. Depending on the avoidable bias, you focus on either bias or variance reduction tactics.

3.3.3 Understanding human-level performance

As was mentioned earlier, in ML human-level error is used as a proxy for the Bayes optimal error. Now let's say you have a medical image classification problem where humans level performance is as follows:

- Typical human: 3% error
- Typical doctor: 1% error
- Experienced doctor: 0.7% error
- Team of experienced doctors: 0.5% error

Now how do you define "human-level" error? To answer this question, bare in mind that you want the human-level error to be a proxy for the Bayes optimal error. Therefore, in this typical example you can conclude that the Bayes optimal error $\leq 0.5\%$. In this setting, one tactic is to set human-level performance to 0.5%. Other techniques is to choose an error of a typical doctor. Which value you choose depends on the goal of your classifier. So instead of comparing the training error to 0%, you set the goal to human-level error which is a proxy for Bayes error. This works well until you surpass human-level performance.

3.3.4 Surpassing human-level performance

Now let's discuss what your strategy should be when you surpass human-level performance. Let's look at two examples. The first example has the following errors:

- Team of humans: 0.5% error
- One human: 1% error
- Training error: 0.6% error

- Dev error: 0.8% error

In this case, your avoidable bias is the difference between the training set and the team of humans. Now look at another example:

- Team of humans: 0.5% error
- One human: 1% error
- Training error: 0.3% error
- Dev error: 0.4% error

Determining the avoidable bias becomes a bit more complicated. Now, have you actually overfit the data by making the train error smaller than the error of the team of humans or is the Bayes optimal error much lower than 0.5%? You actually don't have enough information to know what the valid explanation is. Your options on making progress are less clear now. Problems where ML significantly surpasses human-level performance include online advertising, product recommendations, logistics (predicting transit time), loan approvals, etc. These problems include statistical data, not natural perception. Moreover, these problems include teams that have access to large amounts of data. Others include speech recognition, some image recognition and medical applications (ECG, cancer detection etc.).

3.3.5 Improving your model performance

The two fundamental assumptions of supervised learning are:

1. You can fit the training set pretty well ($\approx$ avoidable bias)
2. The training set performance generalizes pretty well to the dev/test set ($\approx$ variance).

In order to reduce (avoidable) bias and variance it is recommended to look at several elements. First is the difference between your train error and your proxy for Bayes error giving you a sense of the avoidable bias. Additionally, looking at the difference between your train and dev error giving you a sense of the variance you have. In other words, how much harder you should be working to make your performance generalize from the training set to the dev set that wasn't trained on explicitly. To reduce avoidable bias you can apply tactics such as training a bigger model, train longer/better optimization algorithms (momentum, RMSprop, Adam) or NN architecture/hyperparameters search (RNN/CNN). In order to reduce the variance you can add more data, regularization (L_2, dropout, data augmentation) or try NN architecture/hyperparameter search (RNN/CNN).

3.4 Error analysis

3.4.1 Carrying out error analysis

One way of error analysis is looking manually at mislabelled dev set examples and count up how many are mislabelled with the same labels (e.g. how many examples are mislabelled for dogs). Let's say your building a cat classifier and you want to reduce the error by (1) fix pictures of dogs being recognized as cats, (2) fix great cats (lion, panthers, etc.) being misrecognized and (3) improve performance on blurry images. What you can do is compute a table with the mislabelled examples in the rows and the three strategies in the columns. By looking which fraction of the mislabelled examples are due to one of the category errors, it can gives you a better idea which error category to pursue. So to summarize, to carry out error analysis, you should find a set of mislabelled examples, either in your dev set or test set and look at the mislabelled examples for false positives and false negatives. Subsequently, you can just count up the number of errors that fall into various different categories. During this process, you might be inspired to generate new categories By counting up the fraction of examples that are mislabelled in different ways, will often help you to prioritize. Or give you inspiration for new directions to go in. Now as you're doing error analysis, sometimes you notice that some of your examples in your dev sets are mislabelled. So what do you do about that? Let's discuss that in the next section.

3.4.2 Cleaning up incorrectly labelled data

Deep learning algorithms are quite robust to random errors in the training set. They are however less robust to systematic errors. So for instance, when you classifier labels white dogs standard as cats, this could hurt your algorithm. Now how about incorrectly labelled examples in your dev or test set? It is recommended to determine the overall dev set error and distinguish which fraction is due to incorrect labels and due to other causes. When a large fraction of your dev set error is due to incorrect labels, it may be wise to fix up the incorrect labels in your dev set. The actual goal of a dev set is to help you select between multiple classifiers. A few other guidelines are as follows:

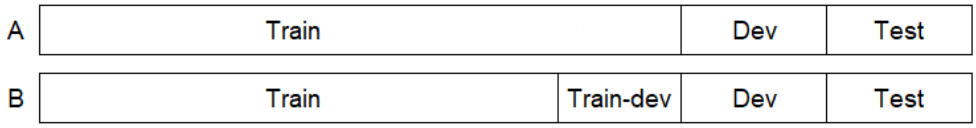


Figure 20: Distribution of the data when the train and dev/test set come from (A) the same or (B) a different distribution. In the case of having a train and dev/test set from a different distribution, you implement and additional dataset called the dev-set, ending up with four different datasets from two different distributions.

- Apply same process to your dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples your algorithm got right as well as ones it got wrong.
- Train and dev/test data may now come from slightly different distributions.

3.4.3 Build your first system quickly, then iterate

A strategy for building an algorithm is act quickly then iterate. In other words, set up a dev/test set and metric, build initial system quickly and afterwards use bias/variance analysis and error analysis to prioritize the next steps. This strategy is a good one when you want to build something from the ground without a lot of background knowledge. When there is already a lot known about the project (e.g. academic literature), which is the case for face recognition, you can start with an algorithm that is more complex.

3.5 Mismatched training and dev/test set

3.5.1 Training and testing on different distributions

Deep learning algorithms have a huge hunger for training data. They just often work best when you can find enough label training data to put into the training set. This has resulted in many teams sometimes taking whatever data they can find and just feeding it to the training set. Even if some of this data, or even maybe a lot of this data, doesn't come from the same distribution as your dev and test data. So in a deep learning era, more and more teams are now training on data that comes from a different distribution than your dev and test sets. Let's take as an example the cat classifier. Say you have a dataset ($\approx$ 210.000) with a large fraction of data collected from webpages ($\approx$ 200.000) and a smaller fraction of data collected from mobile app ($\approx$ 10.000), now what is a good strategy for dividing your data used in the train/dev/test sets?

- One tactic is to shuffle the data and randomly divide in a train ($\approx$ 205.000), dev ($\approx$ 2.500) and test ($\approx$ 2.500) set. One advantage of this tactic is that the train, dev and test set come from the same distributions. One major disadvantage however, is if you look at your dev set, of these 2.500 examples, a lot of it will come from the web page distribution of images, rather than what you actually care about, which is the mobile app distribution of images.
- Another way of distributing your data is by training your data on the web-page images and use the mobile app images for the dev and test set. In other words, training and testing on different distributions. The advantage of splitting up your data this way is that you aim the target where you want it to be, namely on the data from the mobile app. The disadvantage is that your training distribution is different from your dev and test set. however, this will give you better performance over the long term. How this works exactly will be discussed later.

3.5.2 Bias and Variance with mismatched data distributions

Estimating the bias and variance of your learning algorithm really helps you prioritize what to work on next. But the way you analyse bias and variance changes when your training set comes from a different distribution than your dev and test sets. Let's look again at the cat classifier example. Assume a humans gets $\approx 0\%$ error (close to Bayes optimal error), during error analysis you look both at the train and dev set error. When the train and dev set come from the same distribution and the errors are 1% and 10%, respectively, you can assume the variance is relatively high. However, when the train and dev set come from a different distribution, the difference between the train and dev set error could also be due to the fact that the examples in the dev set are harder to classify. So the problem with this is two things changed when running the algorithm, namely (1) the algorithm saw data in the training set but not in the dev set and (2) the distribution of data in the dev set is different from the data in the train set. In order to tease out these two effects it would be useful to define a new piece of data, namely the training-dev set which has the same distribution as the training set, but is not used for training. So earlier you had three different data set, whereas now you have four (see Figure 20). So just and the dev and test set have the same distribution, the train and train-dev set also have the same distribution. The

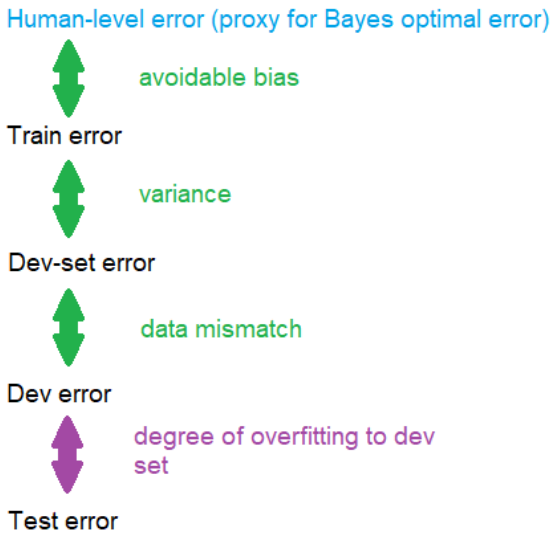


Figure 21: Error analysis based on differences in the train, train-dev, dev and test error leading to avoidable bias, variance, data mismatch or degree of overfitting to the dev set.

difference however is that now you train your neural network only on the train set while you won't run it on the train-dev portion of the data. In order to carry out error analysis, you should look at the error of your classifier on the train set, on the train-dev set as well as on the dev set. Now when there is a big difference between the train and train-dev error and a small difference between the train-dev and dev error, you know you have a variance problem. On the other hand, when the difference between the train and train-dev/dev error is large, you know the variance is quit low and rather you have a data mismatch problem. So, regarding error analysis, depending on the difference between the human-level, train, train-dev and dev error, you either have a problem of avoidable bias, variance or data mismatch (see Figure 21). Technically you could also add one more thing, which is the test set performance. However, you shouldn't be doing development on your test set because you don't want to overfit your test set. But if you also look at this, then this gap in error between the dev and test set tells you the degree of overfitting to the dev set. So if there's a large gap between your dev set performance and your test set performance, it means you maybe overtuned to the dev set. As a solution you could find a bigger dev set. Moreover, remember that your dev set and your test set come from the same distribution. So the only way for there to be a huge gap, for it to do much better on the dev set than the test set, is if you somehow managed to overfit the dev set. And if that's the case, what you might consider doing is going back and just getting more dev set data. So in the case of the cat classifier, where you have a large dataset of web images and a small set of mobile app images, where the ultimate goal is to train the classifier in a way that is is able to recognize the blurry, low quality images of the users (mobile app), the error analysis looks as follows:

Table 1: My caption

Error/Dataset	Web	Mobile app
Human-level	"Human-level"	
Trained data	"Train"	
Untrained data	"Train-dev"	"Dev/Test"

3.5.3 Addressing data mismatch

Addressing data mismatch can be done with several strategies. One tactic is to carry out manual error analysis to try understand difference between training and dev/test sets. Another strategy is to make training data more similar or collect more data similar to dev/test sets (e.g. similar noise in train data, or creating artificial data synthesis). These tactics however, are not systematic strategies. Artificial data synthesis can work very well. But, if you're using artificial data synthesis, just be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

3.6 Learning from multiple tracks

3.6.1 Transfer learning

One of the most powerful ideas in deep learning is that sometimes you can take knowledge the neural network has learned from one task and apply that knowledge to a separate related task, also known as transfer learning. Let's say you have trained a neural network on image recognition. If you want to take this neural network and adapt, or transfer, what is learned to a different task, such as radiology diagnosis, what you can do is take the last output layer of the neural network and just delete that and delete also the weights feeding into that last output layer and create a new set of randomly initialized weights just for the last layer and have that now output radiology diagnosis. So to be concrete, during the first phase of training when you're training on an image recognition task, you train all of the usual parameters for the neural network (i.e. all the weights, all the layers). Now you have something that learns to make image recognition predictions. Having trained that neural network, what you now do is implement transfer learning and swap in a new data set (X,Y), where X are the radiology images and Y the diagnosis you want to predict. Consequently, you initialize the last layers' weights randomly ($w^{[L]}, b^{[L]}$) and train the neural network on this new data set. You have a couple options on how to retrain the neural network with the new radiology data. The rule of thumb is when you have a small data set you can retrain the weights of the last layer and keep the rest of the parameters fixed. When you have a larger dataset you can also retrain all the layers of the neural network. The training on the first dataset is also called the "pre-training" and the training on the second dataset is also called "fine-tuning". Transfer learning is useful when you have a lot of data for the problem you're transferring from and usually relatively less data for the problem you're transferring to. More formally, if you're trying to learn from some Task A and transfer some of the knowledge to some Task B, then transfer learning makes sense when...

- ... task A and B have the same input X.
- ... you have a lot more data for task A than task B.
- ... low level features from A could be helpful for learning B.

3.6.2 Multi-task learning

Whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B. In multi-task learning, you start off simultaneously, trying to have one neural network to do several things at the same time. In multi-task learning, $y^{(i)}$ is a vector predicting the presence of several objects (e.g. traffic image with a stop sign, pedestrian, cars and traffic light). In the case of the traffic image $y^{(i)}$ and $\hat{y}^{(i)} \in \mathbb{R}^4$. Subsequently, $\frac{1}{m}\Sigma_{i=1}^{m}\Sigma_{j=1}^{4}L(\hat{y}_j^{(i)}, y_j^{(i)})$. Now the main difference between this and softmax regression is that unlike softmax regression, which assigned a single label to a single example, this one image can have multiple labels. When you train a neural network to reduce the loss, you are actually performing multi-task learning. Another thing you do is just train four separate neural networks, instead of train one network to do four things. But if some of the earlier features in neural network can be shared between these different types of objects, then you find that training one neural network to do four things results in better performance than training four completely separate neural networks to do the four tasks separately. Multi-task learning makes sense when...

- ... training on a set of tasks that could benefit from having shared lower-level features.
- ... usually: Amount of data you have for each task is quit similar.
- ... can train a big enough neural network to do well on all the tasks.

In practice, transfer learning is applied much more often than multi-task learning.

3.7 End-to-end deep learning

3.7.1 What is end-to-end deep learning?

One of the most exciting recent developments in deep learning, has been the rise of end-to-end deep learning. So what is end-to-end learning? Briefly, there have been some data processing systems, or learning systems that require multiple stages of processing. What end-to-end deep learning does, is it can take all those multiple stages, and replace it usually with just a single neural network. An example of end-to-end learning is with speech recognition. Traditionally, speech recognition demanded several stages (audio $\rightarrow$ features $\rightarrow$ phonemes $\rightarrow$ words $\rightarrow$ transcript). So, in contrast to this pipeline with a lot of stages, what end-to-end deep learning does, is you can train a huge neural network to just input the audio clip, and have it directly output the transcript. One of the challenges of end-to-end deep learning is that you need a large amount of data for it to work well. Another example where end-to-end learning is applied is with face recognition

(e.g. image courtesy of Baidu), where employees are granted access to a building through a camera which takes an image of their face. In this case, face recognition can be implemented in one of two ways. One way is end-to-end deep learning where the image is fed to a neural network which processes and identifies the person in the photo. The second approach differs from end-to-end learning in that is divides the process in two stages. The first stage is that the software only processes the face (cuts out the rest of the image). The second step includes running the image through a data base of all employees. This is different from earlier computed end-to-end deep learning neural networks which process the entire image. There are two reasons why this two step approach works better. One of the reasons is that each of the two problems you are solving is much simpler. Second, you have a lot of data for each of the two sub-tasks. Another applications is machine translation which usually includes many different stages. In this case, end-to-end deep learning works quit well, because today it is possible to obtain a large dataset. One last example is estimation the age of a child based on hand X-ray images. A none end-to-end, multi-step approach is taking an image and segment all the different bones, estimating the child's age (Image $\rightarrow$ bones $\rightarrow$ age). This approach works pretty well. In contrast, when you go from the image to the age directly, you would need a lot of data.

3.7.2 Whether to use end-to-end deep learning

Let's take a look at the pros and cons of end-to-end learning. Pros are:

- Let the data speak. So if you have enough data, if you train a big enough neural network, hopefully the neural network will figure it out. By having a pure machine learning approach, your neural network learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.
- Less hand-designing of components needed.

Cons are:

- May need large amount of data.
- Excludes potentially useful hand-designed components.

The key question as to whether use end-to-end deep learning is: Do you have sufficient data to learn a function of the complexity needed to map x to y?

4 Convolutional Neural Networks

4.1 Convolutional Neural Networks

4.1.1 Computer Vision

Computer vision is an interdisciplinary field that deals with how computers can be made for gaining high-level understanding from digital images or videos. From the perspective of engineering, it seeks to automate tasks that the human visual system can do. Computer vision can be really exciting for two reasons. First, rapid advances in computer vision are enabling brand new applications to view, though they were just impossible a few years ago. Second, if you don't end up building computer vision systems exclusively, computer vision research community has been so creative and so inventive in coming up with new neural network architectures and algorithms, is actually inspire that creates a lot cross-fertilization into other areas as well. Problems of computer vision include image classification, object detection and Neural Style Transfer (new type of art work). One of the challenges of computer vision is that the input can be really vague. When you have a high resolution image, the neural network you have to build in order for the algorithm to work will be really large. A solution for this is applying a convolution operation, which is one of the major building blocks of convolutional neural networks.

4.1.2 Edge Detection Example

The convolution operation is one of the fundamental building blocks of a convolutional neural network. Let's look at an example, namely edge detection, and more specific, vertical edge detection. Let's say you have a 6×6 gray scale image. In order to detect edges you construct a 3×3 matrix which is called a filter (sometimes called a kernel). Next, you are going to convolve the 6×6 matrix with the 3×3 filter (denoted with an asterisk $*$) ending up with a 4×4 image:

3	**0**	**1**	2	2	7	4
1	**5**	**8**	9	9	3	1
2	**7**	**2**	5	5	1	3
0	1	3	1	1	7	8
4	2	1	6	6	2	8
2	4	5	2	2	3	9

$*$

1	0	-1
1	0	-1
1	0	-1

$=$

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

This turns out to be a vertical detection, where the first and third matrix are interpreted as an image, and the second as a filter. Now why is this computing vertical edge detection? The 3×3 filter can be visualized as having lighter pixels on the left (1) and darker pixels on the right (-1) and intermediate pixels in the middle (0) (see Figure 22).

4.1.3 More Edge Detection

The 3×3 filter mentioned earlier allows you to detect vertical edges, while the following filter is able to detect horizontal edges: So different filters allow you to find vertical and horizontal edges. It turns out that the three by three vertical edge

1	1	1
0	0	0
-1	-1	-1

detection filter we've used is just one possible choice. And historically, in the computer vision literature, there was a fair amount of debate about what is the best set of numbers to use. For instance, you can also use a filter where in the center the numbers are higher than in the borders. This is also called a Sobel filter. Another filter is a Scharr filter, which has different kind of properties.

4.1.4 Padding

The rule of thumb is when you have an $n \times n$ image with a convolution filter of $f \times f$, the result is a $n - f + 1 \times n - f + 1$ image. The two downsides to this is every time you apply a convolutional filter, the image shrinks. As a result, this filtering processing can be done a limited amount of times. The second downside is that the corner pixels are used only one time as output, while the other pixels are filtered multiple times. This way, it is more likely that you throw away useful information near the edge of the image. To solve these two problems you can apply padding. Padding includes adding and additional border to the image, where you mange to preserve the size of the image and the output becomes $n + 2p - f + 1 \times n + 2p - f + 1$. If you want you can also pad the borders with more than 1 pixel. So to summarize, padding is useful for two reasons:

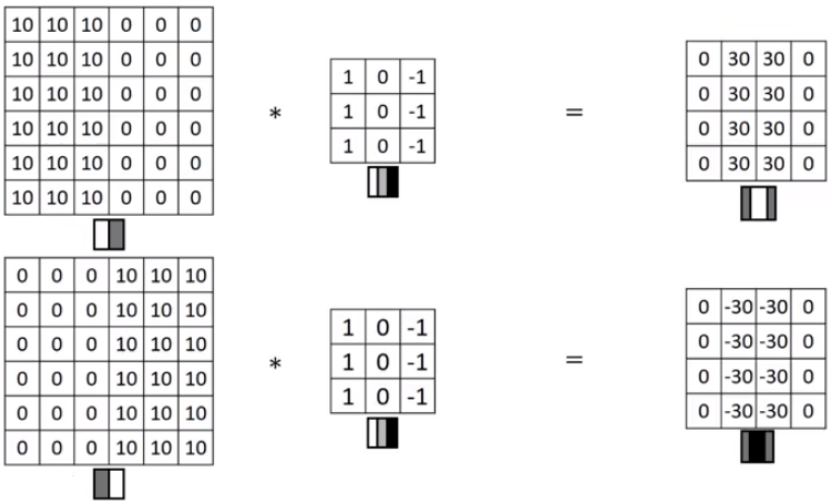


Figure 22: Edge detection.

- It allows you to use a convolutional layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.

In terms of how much to pad, there are two options:

- "Valid" (no padding): $n \times n * f \times f \rightarrow n - f + 1 \times n - f + 1$
- "Same:": Pad so that output size is the same as the input size $n + 2p - f + 1 \times n + 2p - f + 1$ where $p = \frac{f-1}{2}$
  - Usually, f takes on an odd number. Reasons for this is because when f is even, than you need some asymmetric padding. So only if f is odd, this type of same convolution gives a natural padding region. Second, when you have an odd dimension filter, it has a central position and sometimes in computer vision its nice to have a distinguisher. In other words, it's nice to have a pixel you can call the central pixel so you can talk about the position of the filter.

4.1.5 Strided Convolutions

Strided convolutions is another basic building block of convolutions as used in Convolution Neural Networks. Stride controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. When the stride is 2 (or rarely 3 or more) then the filters jump 2 pixels at a time as they slide around. The receptive fields overlap less and the resulting output volume has smaller spatial dimensions. The size of the output image will take the following rule of thumb: $\frac{n+2p-f}{s} + 1 \times \frac{n+2p-f}{s} + 1$. It can happen that the output size is not an integer. In such a case you take the floor integer (round down) as the output size. So to summarize, when you have a $n \times n$ image and a $f \times f$ filter with a padding p and stride s, the size of your output image is as follows:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

One technical note, if you reading a math textbook the way the convolution is defined before doing the element-wise product and summing, there's actually one other step that you'll take first. This includes taking the $f \times f$ filter and slip it on the horizontal as well as the vertical axis. The way we define the convolution operation in these sections is that we've skipped this narrowing operation. Technically, what we're actually doing, the operation we've been using for the last few videos is sometimes called cross-correlation instead of convolution. But in the deep learning literature by convention, we just call this a convolutional operation. Just to summarize, by convention in machine learning, we usually do not bother with this skipping operation and technically, this operation is maybe better called cross-correlation but most of the deep learning literature just calls it the convolution operator.

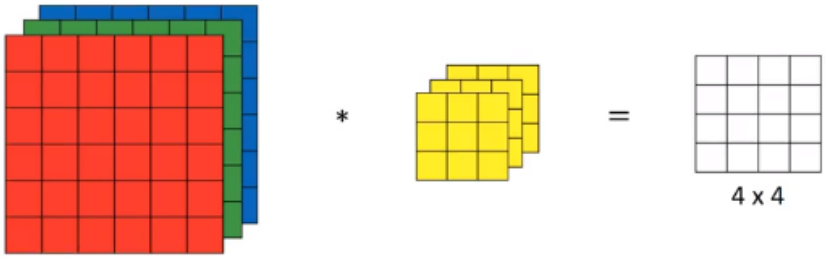


Figure 23: Convolution with RGB images. As input a $6 \times 6 \times 3$ image and a $3 \times 3 \times 3$ filter resulting in an output of a 4×4 image.

4.1.6 Convolutions Over Volume

When dealing with RGB images, the input you feed to your convolutional neural network includes both a 3-dimensional image as well as 3-dimensional filter (height $\times$ width $\times$ # channel). The number of channels (or depth) in your image must match the number of channels in your filter. Let's look at en example (see Figure 23). When computing the first pixel of the output image, you have $3 \times 3 \times 3$ parameters with should be multiplied with the corresponding numbers from the red, green and blue channels of the image. Depending on how you set the parameters in the three channels, enables you to develop different feature detectors. Now that you know how to convolve on volumes, there is one last idea that will be crucial for building convolutional networks, which is detecting multiple features (e.g. vertical and horizontal edges). This can be done by applying different filters to your image, yielding multiple output images. By stacking this output images you can combine different feature detections. To summarize, if you have a $n \times n \times n$ image, with a $f \times f \times f$ filter (number of channels must be equal for the image and filter), you get a $n - f + 1 \times n - f + 1 \times n'_c$ output where n'_c denotes the number of filters (assuming you don't apply padding and striding).

4.1.7 One Layer of a Convolutional Network

Before yielding an output image you have to apply a bias and non-linearity (e.g. ReLU) to the output image yielded by multiplying the image with the filer. After this, you can stack the output images. This sequence of actions includes one layer of a convolutional neural network. Extrapolating this to non-convolutional networks will give the following similarities in computation. As you remember, one layer of a non-convolutional network has the following structure (with the corresponding convolution neural network components:

- $z^{[l]} = W^{[l]} \times a^{[l-1]} + b^{[l]} \rightarrow$ output image = filter $\times$ input image + bias (where $b \in \mathbb{R}$)
- $a^{[l]} = g(z^{[l]}) \rightarrow$ output image = ReLU(output image)

Now let's discuss the notation. If layer l is a convolutional layer:

- $f^{[l]}$ = filter size
- $p^{[l]}$ = padding
- $s^{[l]}$ = stride
- $n_c^{[l]}$ = number of filters
- Each filter is $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
- Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ where $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
- Bias: $n_c^{[l]}$, for practical reason dimensions are $(1, 1, 1, n_c^{[l]})$
- Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
- Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- $n_{H/W}^{[l]} = \left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$

4.1.8 Simple Convolutional Network Example

Let's take a look at a a deep learning convolutional network (ConvNet), more specifically a cat classifier (image size is illustrated in the order $n_H \times n_W \times n_c$):

$$39 \times 39 \times 3 \xrightarrow[\text{10 filters}]{f^{[1]}=3 \quad s^{[1]}=1 \quad p^{[1]}=0} 37 \times 37 \times 10 \xrightarrow[\text{20 filters}]{f^{[2]}=5 \quad s^{[2]}=2 \quad p^{[2]}=0} 17 \times 17 \times 20 \xrightarrow[\text{40 filters}]{f^{[3]}=5 \quad s^{[3]}=2 \quad p^{[3]}=0} 7 \times 7 \times 40$$

The output image will be flattened out, yielding a vector with dimensions $(1960, 1)$ $(7 \times 7 \times 40 = 1960)$. In the case of object recognition, the vector will be fed into softmax in order to make predictions for the final output. There are different types of layer in a convolutional network:

- Convolution layer (CONV)
- Pooling layer (POOL)
- Fully connected layer (FC)

4.1.9 Pooling layers

We'll now discuss the pooling layer, which are used to reduce the size of the representation in order to speed up the computation as well as make some of the features that are detected slightly more robust. One example is max pooling, which is a sample-based discretization process. The objective is to down-sample an input representation, reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned. This is done to in part to help overfitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation. Max pooling is done by applying a max filter non-overlapping subregions of the initial representation. Let's say we have a 4 matrix representing our initial input. Let's say, as well, that we have a 2×2 filter that we'll run over our input. We'll have a stride of 2 for stepping over our input and therefore it won't overlap regions. For each of the regions represented by the filter, we will take the max of that region and create a new, output matrix where each element is the max of a region in the original input. One interesting property of max pooling is that it has a set of hyperparameters but it has no parameters to learn. Once you fix the filter size and striding, it's just a fixed computation and gradient descent doesn't change anything. Another type of pooling is called average pooling, where you take for each filter the average value instead of the maximum value. So to summarize...

- Hyperparameters are f (filter size) and s (stride) ($f = 2$, $s = 2$ are commonly used values)
- Subsequently, you can perform max or average pooling (usually, this type of pooling does not apply padding).
- Pooling applies to all of your channels, so the number of channels will be the same for your input and output image (n_c)
- No parameters to learn (no use to apply backpropagation).

4.1.10 CNN Example

Now let's look at an example of a convolutional neural network, where you have an input RGB image and try do perform handwritten digit recognition (LeNet-5)

$$32\times32\times3 \xrightarrow[\text{6 filters}]{f=5 \quad s=1} 28\times28\times6 \text{ (CONV1)} \xrightarrow[\text{6 filters}]{f=2 \quad s=2} 14\times14\times6 \text{ (POOL1)} \xrightarrow[\text{16 filters}]{f=5 \quad s=1} 10\times10\times16 \text{ (CONV2)} \xrightarrow[\text{16 filters}]{f=2 \quad s=2}$$
$$5\times5\times16 \text{ (POOL2)} \rightarrow 400 \xrightarrow{W^{[3]}(120,400) \quad b^{[3]}(120)} FC3(120) \rightarrow FC4(84) \rightarrow \text{softmax} \quad \hat{y}(10)$$

Written more formally in terms of activation shape, activation size and the number of parameters in the network, see the table bellow. When you look at the table, you may notice a few things. First, the POOL layers don't have any parameters. Second, the convolutional layers tend to have fewer parameters, while the number of parameters in the fully connected layers are relatively high. And finally, as you ascend deeper into the neural network, the activation size tends to decrease.

	Activation shape	Activation size	# parameters
Input:	(32,32,3)	3072	0
CONV1 (f = 5, s = 1)	(28,28,6)	6272	208
POOL1	(14,14,6)	1568	0
CONV2(f = 5, s = 1)	(10,10,16)	1600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48 001
FC4	(84,1)	84	10 081
Softmax	(10,1)	10	841

4.1.11 Why Convolutions?

Now why using convolutional layers additionally to fully connected layers? The advantages are (1) parameter sharing and (2) sparsity of connections. Parameter sharing includes a feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image. Sparsity of connections means that in each layer, each output value depends only on a small number of inputs.

4.2 Case studies

4.2.1 Why look at case studies?

As an outline what we will do in the next sections is discuss several classic networks (LeNet-5, AlexNet, VGG). Moreover, we'll also discuss ResNet and Inception. Even if you don't end up building computer vision applications yourself, I think you'll find some of these ideas very interesting and helpful for your work.

4.2.2 Classic Networks

Let's first discuss a few classical networks, namely LeNet-5, AlexNet and VGG. As was mentioned earlier, LeNet-5 is classical network designed for handwritten digit recognition (see Figure 24A). This network has an input of gray-scale image (i.e. 1 channel). Several patterns can be observed in this neural network. First, the number of parameters is relatively small ($60K$ parameters). Second, the numbers of channels tend to increase as you go deeper into the network, while the size of the activation tends to decrease. Third, the type of arrangement is quite common ($CONV \rightarrow POOL \rightarrow CONV \rightarrow POOL \rightarrow FC \rightarrow FC \rightarrow OUTPUT$). For more advanced comments, see the literature[2]. Earlier, researchers applied sigmoid/tanh non-linearity as opposed to ReLU for computational reasons. Nowadays, due to the increased processing speed of computers, the appliance of non-linearity over input images has changed. Another classical network is the AlexNet, named after Alex Krizhevsky who was the first author of the paper describing this neural network[3] (see Figure 24B). This neural network has a lot of similarities to LeNet-5, only much larger in size ($\approx 60M$ parameters). Another aspect of this architecture making it far more useful than LeNet is the ReLU function. Moreover, this network was able to train on multiple GPUs making it more efficient to run. This network also made use of a different type of layer, namely a Local Response Normalization. The last network we'll discuss is the VVG-16 neural network, which made use of a much simpler network with fewer parameters, with the same hyperparameters for the different layers ($CONV = 3 \times 3$ filter , $s = 1$, *same*; $MAX - POOL = 2 \times 2$, $s = 2$)[4] (see Figure 24C). This network has a large number of parameters ($\approx 130M$ parameters). The simplicity and uniformity of the architecture makes it an appealing neural network to use.

4.2.3 ResNets

Very deep neural networks are difficult to train due to vanishing and exploding gradient types of problems. A possible approach is to skip connections which allows you to take the activation from one layer and feed it to another layer deeper in the neural network. These networks are called ResNet which enables you to train very deep networks. ResNets are built out of something called residual blocks, in which a short cut/skip connection is created after a linear part but before a ReLU part. When training a plain network, you'll see that adding layers will result in a lower training error. However, when adding more layers, the training error will go up again. In theory, a deeper network should result in a lower training error, in reality, it rises again. When one uses ResNet, the performance of training error will decrease, even when the network contains many layers. This helps with the vanishing and exploding gradient types problems.

4.2.4 Why ResNets Work

Let's look at one example that shows the efficiency of ResNets. Residual networks use identity functions to transfer results from previous layers unaltered. When adding layers to a large neural network, it is quite easy for the network to learn the identity matrix despite the addition of layers. That is why adding a residual block (with for instance two layers) in the middle or at the end doesn't hurt the performance. However, our goal is not to not just not hurt performance, it is to help performance. So you can imagine that if all of these heading units actually learned something useful than maybe the network can do even better than learning the identity function.

4.2.5 Networks in Networks and 1×1 Convolutions

In terms of designing content architectures, one of the ideas that really helps is using a one by one convolution/network in network (i.e. $1 \times$ filter). When having only one channel, it doesn't really make a difference. However, when you have

[2] LeCun *et al.*, 1998. Gradient-based learning applied to document recognition

[3] Krizhevsky *et al.*, 2012. ImageNet classification with deep convolutional networks.

[4] Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition.

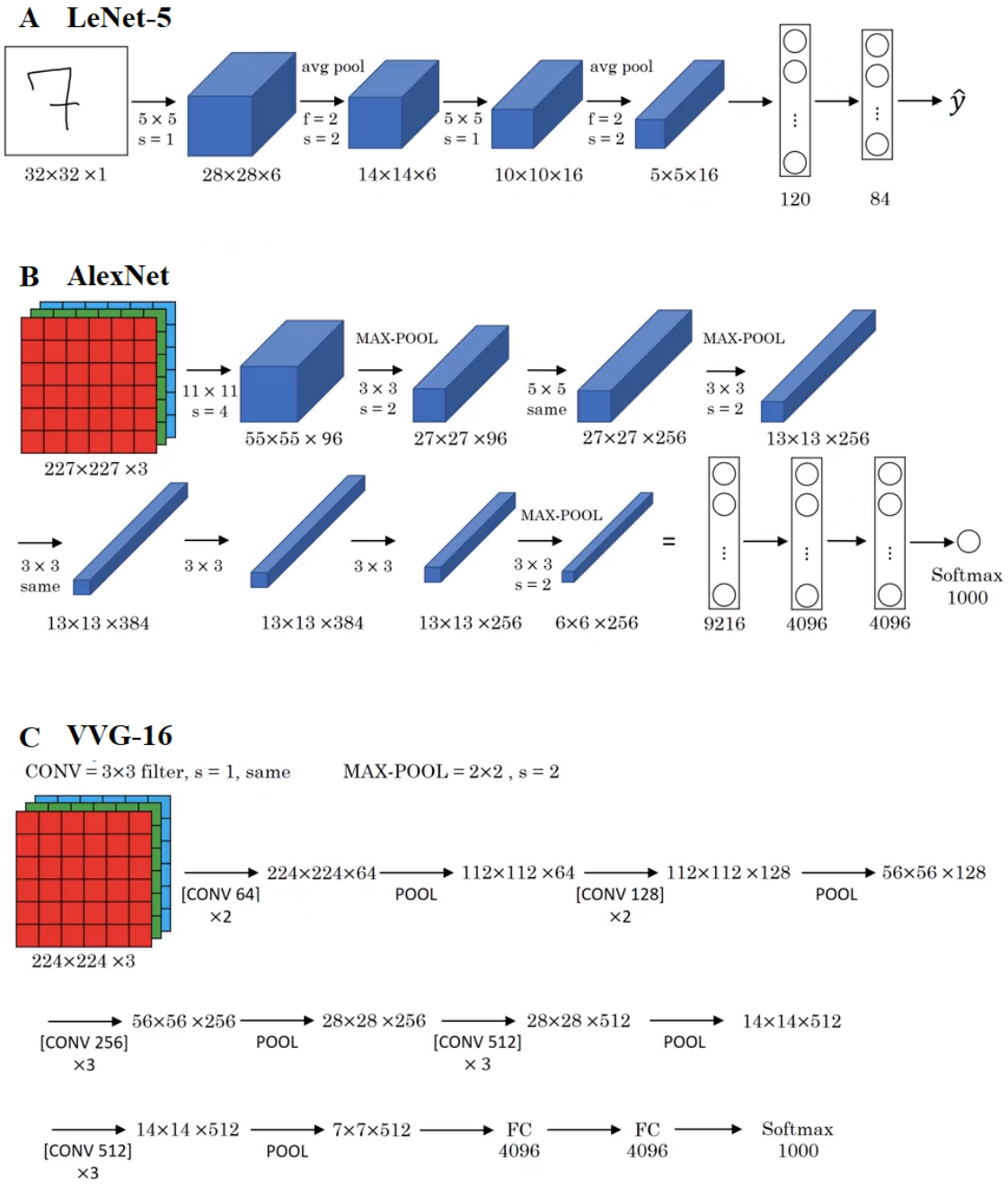


Figure 24: Architectures of different types of neural networks. (A) LeNet-5, (B) AlexNet and (C) VVG-16 (16-layered neural network).

a input of multiple channel, a 1×1 convolution can be helpful. A $1 \times$ one filter will use a element-wise multiplication across the different channels followed by a ReLU function. You can think of a 1×1 convolution as a fully connected network[5]. 1×1 convolution can be helpful when you intend to shrink the number of channels (n_c). Pooling also has the purpose to shrink a network, however, this technique can only alter the dimensions of n_h and n_w. Network in network can be helpful to build an inception network. Finally, ResNet blocks with the short cut also makes it very easy for one of the blocks to learn an identity function. This means that you can stack on additional ResNet blocks with little risk of harming training set performance. (There is also some evidence that the ease of learning an identity function–even more than skip connections helping with vanishing gradients–accounts for ResNets' remarkable performance). Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different.

- The identity block: The identity block is the standard block used in ResNets, and corresponds to the case where the input activation (say $a^{[l]}$) has the same dimension as the output activation (say $a^{[l+2]}$).
- The convolutional bock: You can use this type of block when the input and output dimensions don't match up. The difference with the identity block is that there is a convolutional layer in the short cut path. The convolutional layer in the short cut path is used to resize the input x to a different dimension, so that the dimensions match up in the final addition needed to add the short cut value back to the main path. Moreover, the convolutional layer on the short cut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.

4.2.6 Inception Network Motivation

An inception network approximates a sparse CNN with a normal dense construction, where the width and number of the convolutional filters is kept small. Moreover, it uses convolutions of different sizes to capture details at varied scales (1×1, 3×3, 5×5). The basic idea is that instead of you needing to pick one of these filter sizes or pooling, you can do them all and just concatenate all the outputs. There is one problem with the inception layer as we've described it here, namely the problem of the computational cost. Say you have an input:

$$28 \times 28 \times 192 \xrightarrow{CONV\ 5\times5\times32} 28 \times 28 \times 32 \approx 120M.$$

This can become computationally very expensive operations. An alternative architecture is as using a 1×1 convolution also called a "bottleneck layer", where you shrink the representations before increasing the size again.

$$28 \times 28 \times 192 \xrightarrow[one filter:1\times1\times192]{CONV\ 1\times1\times16} 28 \times 28 \times 16 \xrightarrow[one filter:5\times5\times16]{CONV\ 5\times5\times32} 28 \times 28 \times 32 \approx 12.4M$$

4.2.7 Inception Network

The inception model takes as activation a previous activation (see Figure 25). In this figure is one example inception module (bottom figure). An inception network takes all these different modules/blocks together (top figure). One detail (not shown in the figures) are different additional side branches that can be added to the network. These side branches take some hidden layers and use this layer to make a prediction (using a softmax function). What these side branches do is to ensure that the features computed, even in the hidden units, are useful/valid for predicting the output image. This appears to have a regularizing effect on the inception network, preventing it from overfitting. The network shown is also called GoogLeNet.

4.3 Practical advices for using ConvNets

4.3.1 Using Open-Source implementation

Online open-source documentation (e.g. GitHub) can help you building and training deep networks.

4.3.2 Transfer Learning

If you are building a computer vision application rather than training the ways from scratch, from random initialization, you often make much faster progress (i.e. downloading weights from the internet). The computer vision research community has been pretty good at posting lots of data sets on the internet (e.g. Net, MS COCO, Pascal types). By means of these datasets you can transfer hyperparameters and weights to train your network much faster. This is also called transfer learning. When you have a network, you can freeze a portion of the hidden units and train the others. In all the different applications of deep learning, computer vision is one where transfer learning is something that you should always do unless you have an exceptionally large dataset to train everything else from scratch yourself.

[5] Lin *et al.*, 2013. Network in network.

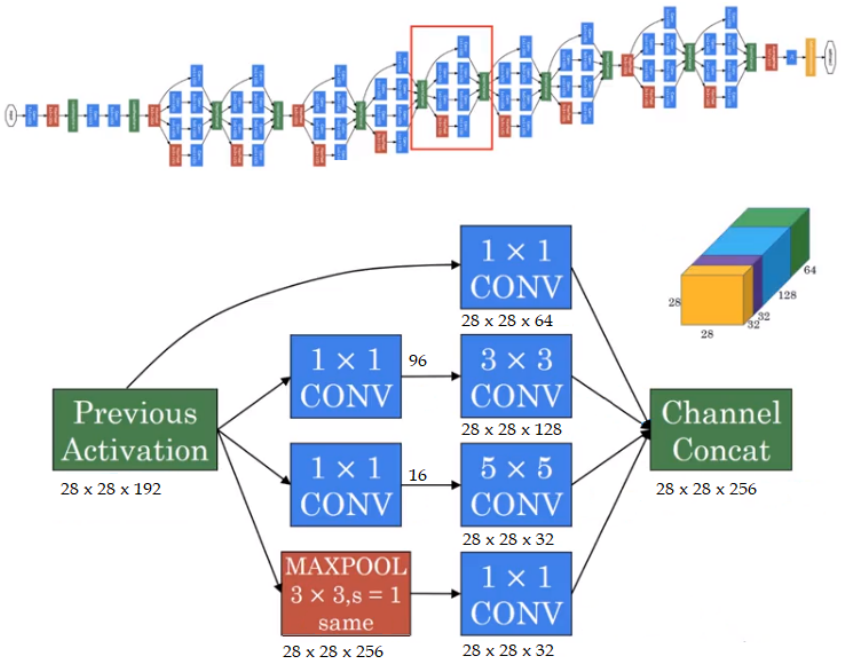


Figure 25: Module of inception network.

4.3.3 Data Augmentation

Data augmentation is one of the techniques often used to improve performance. A common augmentation method is mirroring, where pictures doubled by being vertically flipped. Another method is random cropping, although this isn't a perfect method, in practice is can be very helpful. Other techniques include rotations, shearing and local warping, however, being used a lot less. Another technique is colour shifting, where you apply different distortions to the RGB channels. By taking different values for RGB, images can be distorted. RGB values are computed using PCA colour augmentation, however, this will not be discussed any further.

4.3.4 State of Computer Vision

You can think of computer vision problems on a spectrum where on the one hand you have little data and on the other hand a lot of data (from little to a lot: object detection, image recognition, speech recognition). If you look across a broad spectrum of machine learning problems, you can get away with simpler algorithms and less hand-engineering when you have a large dataset. In contrast, when you don't have that much data, people engage more often in more hand-engineering ("hacks"). Usually there are two sources of knowledge, namely (1) labelled data and (2) hand engineered features/network architectures/other components. Due to lack of sufficient data, computer vision relies more on hand engineering nowadays. When we talk about object detection, you see that the algorithms become even more complex with more specialized components. When having little data transfer learning can be very useful. Tips for doing well on benchmarks/winning competitions:

- Ensembling
  - Train several networks independently and average their outputs.
- Multi-crop at test time
  - Run classifier on multiple versions of test images and average results

Moreover, use open source code:

- Use architectures of networks published in the literature.
- Use open source implementations if possible.
- Use pre-trained models and fine-tune on your dataset.

4.3.5 Keras

Keras was developed to enable deep learning engineers to build and experiment with different models very quickly. Just as TensorFlow is a higher-level framework than Python, Keras is an even higher-level framework and provides additional abstractions. Being able to go from idea to result with the least possible delay is key to finding good models. However, Keras is more restrictive than the lower-level frameworks, so there are some very complex models that you can implement in TensorFlow but not (without more difficulty) in Keras. That being said, Keras will work fine for many common models. Keras is very good for rapid prototyping. In just a short time you will be able to build a model that achieves outstanding results. Note that Keras uses a different convention with variable names than we've previously used with numpy and TensorFlow. In particular, rather than creating and assigning a new variable on each step of forward propagation such as X, Z1, A1, Z2, A2, etc. for the computations for the different layers, in Keras code each line above just reassigns X to a new value using X = ... In other words, during each step of forward propagation, we are just writing the latest value in the computation into the same variable X. The only exception was X_input, which we kept separate and did not overwrite, since we needed it at the end to create the Keras model instance. To train and test a model, four steps have to be taken in Keras:

1. Create the model by calling the model function.
2. Compile the model by calling: model.compile(optimizer = "...", loss = "...", metrics = ["accuracy"])
3. Train the model on train data by calling model.fit(x = ..., y = ...)
4. Test the model on test data by calling model.evaluate(x = ..., y = ...)

4.4 Detection algorithms

4.4.1 Object Localization

Image classification includes labelling objects in images (e.g. "car"). When speaking of localization you the algorithm is also responsible for putting a bounding box around the object, this is called classification with localization. Moreover, a detection problem includes a problem where you have multiple objects in an image which you want to both classify and localize. One speaking of a classification problem, you can train a convolutional network with a softmax function at the end of the pipeline for different objects (e.g. pedestrian, car, motorcycle and background). When dealing with a classification with localization, there needs to be an additional output unit representing the bounding box (b_x, b_y, b_h, b_w). So to summarize, the network contains four different classes and outputs b_x, b_y, b_h, b_w as well as the class label probabilities (1-4).

The target label y will be defined as follows: $y = \begin{pmatrix} p_c \\ b_x \\ b_y \\ b_h \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$, where the first component p_c indicates whether there is an object? C_1, C_2 and C_3 points out the object class (i.e. pedestrian, car or motorcycle). Now let's discuss the loss function $L(\hat{y}, y)$. The loss function is defined as follows: $L(\hat{y}, y) = (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \ldots + (\hat{y}_8 - y_8)^2$ if $y_1 = 1$ (an object is in the image). The other case, when $y_1 = 0$, the loss can be defined as $L(\hat{y}, y) = (\hat{y}_1 - y_1)^2$.

4.4.2 Landmark Detection

Landmark detection includes a network which outputs x and y coordinates of important points in an image. This can be used for face recognition, emotion recognition (for instance snapchat filters). Another application where landmark detection is used is to detect body poses.

4.4.3 Object Detection

You have now learned about object localization and landmark detection. Now let's build up to an object detection algorithm. You will learn how to use a convolutional neural network to perform object detection using something called sliding windows. Let's say you want to build a car detection algorithm and first train a convolutional network with a labelled training set, where the dataset includes cropped images. Once you have trained this convolutional network you can then use it in a sliding window detection algorithm. When you have a test image, you start by picking a certain window size which you feed into the convolutional network which makes a prediction (is it a car or not?). You repeat this until you have slid the window across every position in the image. Depending on which stride you choose, you pass

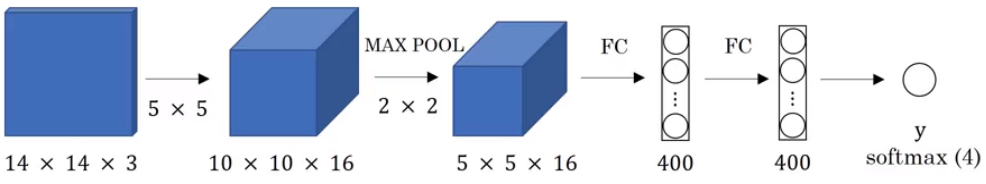
Figure 26: Object detection using sliding windows.

overlapping images into the network. The next step is to repeat is for a larger window size (and so on). The hope is that as long as there is a car somewhere in the image, there will be at some point a window size which detects this (see Figure 26). A large disadvantage of sliding windows detection is the computational cost. Due to the fact that you crop out so many different square regions in the image and running these cropped images independently through a convolutional network it is computationally a very expensive operations. The cost diminished by taking a larger stride reducing the number of windows you need to pass through, however, this may hurt performance.

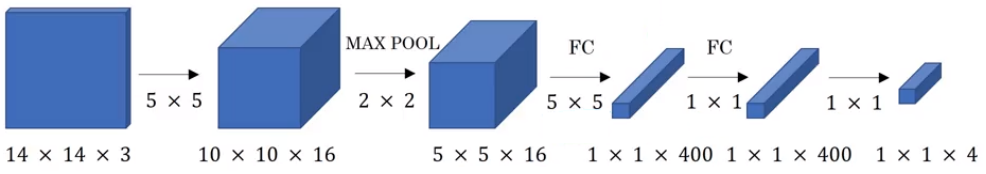
4.4.4 Convolutional Implementation of of Sliding Windows

In the last section you learned about the sliding detection algorithm which turned out to be very slow. In this section you will learn how to implement that algorithm in a convolutional manner. First, you will be explained how to turn fully connected layers in your network to convolutional layers. Say you have the following convolutional network (see Figure 27a). First, we propose an alteration to the output unit which consists of four numbers corresponding to the cause probabilities of the four classes the softmax unit is classified amongst. Now how to change the fully connected layers to convolutional layers? This is done by applying a 5×5 followed by a 1×1 convolution (see Figure 27b). Now we will discuss how you can have a convolutional implementation of sliding windows object detection[6]. Let's say, that during training your convolutional network inputs $14 \times 14 \times 3$ (see Figure 28a) images and you're testing images are of size $16 \times 16 \times 3$. In the original window sliding algorithms you might want to input the blue region into a convolutional network with stride 2. You end up yielding four different images that need to be run by the network. As you can imagine, this process is highly duplicative. Now what the convolutional implementation of sliding windows does is it allows the four runs share a gross part of the computation. So instead of forcing you to run four propagation on four subsets of the input image independently, it combines all four into one form of computation where computation power is shared by regions of the image that have overlap. So in the output layer, the upper left corner corresponds to the red cropped image, the right left corner corresponds to the blue blue cropped image, the lower left corner corresponds to the orange image and the lower right corner corresponds to the green image (see Figure 28b). So instead of doing it in sequence you can implement the entire image, making al the predictions at the same time by one forward pass through the convolutional network and hopefully have it recognized the object. One weakness of this approach is that the position of the bounding box might not be accurate. In the next section, we will discuss how this problem is fixed.

[6]Sermanet *et al.*, 2014. OverFeat: Integrated recognition, localization and detection using convolutional networks

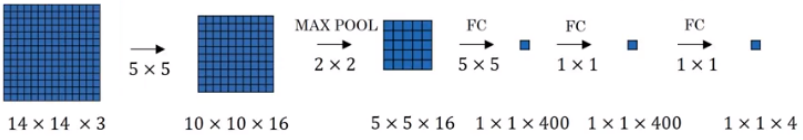


(a)

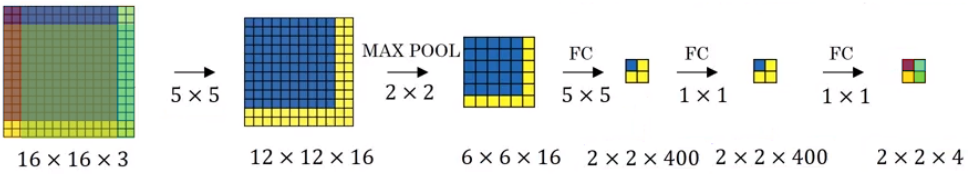


(b)

Figure 27: Object detection using sliding windows. (a) Convolutional neural network with 2 fully connected (FC) layers. (b) Convolution network where the FC layers are turned into convolutional layers.



(a) Training images

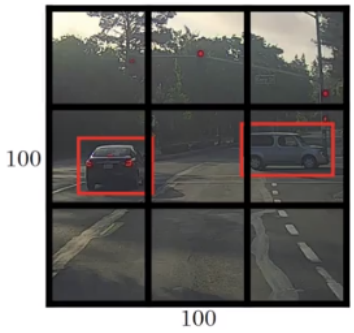


(b) Testing images

Figure 28: Object detection using sliding windows. (a) Convolutional neural network with training images of size $14 \times 14 \times 3$ as input. (b) Convolution network with testing images of size $16 \times 16 \times 3$ as input. Order of colour from left to right, top to bottom: red, blue, orange, green.

4.4.5 Bounding Box Predictions

In the last section you have learned to use a convolutional implementation of sliding windows, which it computationally more efficient. However, the problem remains of that this approach does not output the most accurate bounding boxes. A way to fix this is the YOLO algorithm which is an abbreviation of "You only look once"[7]. Let's say you have a 100×100 image where you use a 3×3 grid. The basic idea is that you are going to take the image classification and localization algorithm discussed in previous sections and apply it to the nine grids. First, as you remember, the defining of the labels was as follows. For each of the nine grid cells you specify a label y where $y = \begin{pmatrix} p_c & b_x & b_y & b_h & C_1 & C_2 & C_3 \end{pmatrix}$, where p_c indicates whether an object is present (value of 0 or 1), $b_x - b_h$ specifying the bounding box is there is an image and $C_1 - C_3$ indicating the class (e.g. pedestrian, car or motorcycle). Now what the YOLO algorithm does, it takes the midpoint of each of the two objects and assigns the object to the grid cell containing the midpoint. So for each of the nine grid cells you end up with a 8-dimensional vector, resulting in a target output volume of $3 \times 3 \times 8$. In order to train your convolutional neural network, you have an input image of $100 \times 100 \times 3$. The advantage of this algorithm is that the neural network outputs precise bounding boxes. This is done as follows. What do you is you feed the network an input image and consequently perform forward propagation and then for each of the nine target outputs see whether there is an image present in that grid cell, what the values are for the bounding box and to which class the object belongs. As long as you have only one object in a grid cell this algorithm should run okay. Detecting multiple object in one grid cell is something that will be discusses later. In practice, larger grids are used, making the respective grid much finer reducing the chance that there are multiple objects in one grid cell. Moreover, the way you assign a object to a grid cell is by looking at the midpoint. So, even if the object extends to multiple grid cells, that object is assigned to only one of them. So, notice two things:



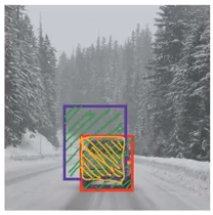
- This is similar to the image classification and localization discusses earlier, where it outputs the bounding box coordinates explicitly. This allows your network to output bounding boxes of any aspect ratio. as well as output more precise coordinates that aren't just dictated by the strip size of you sliding window classifier.
- This is a convolutional implementation meaning that you don't implement this algorithm nine times. Instead, you use a single convolutional implementation using one constant with a lot of shared computations between the different grid cells. This makes this algorithm really fast.

4.4.6 Intersection Over Union

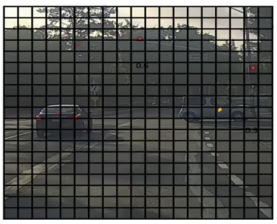
So how do you tell whether your object algorithm is working well? You can determine this using the Intersection over Union function (IoU) . In the object detection task you are expected to localize the object as well. Now what the IoU does is computing the intersection over union of two bounding boxes (e.g. right figure, purple and red bounding boxes).

[7] Redmon *et al.*, 2015. You Only Look Once: Unified real-time object detection

The union of these two bounding boxes is the total area within the bounding boxes (green and yellow) and the intersection is the area where they overlap (yellow). What IoU does it computes the size of the intersection divided by the size of the union (i.e. $\frac{size\ of\ intersection}{size\ of\ union}$). By convention, the answer is "correct" if $IoU \geq 0.5$. Ideally, the IoU is 1, however, in practice this doesn't happen very often. What motivated the definition if IoU as a way of evaluating whether or not an object localization algorithm is accurate or not. More generally, IoU is a measure of the overlap between two bounding boxes (degree of similarity).



4.4.7 Non-max Suppression



One of the problems of object detection is that your algorithm may find multiple detections of the same objects. Non-max suppression is a way for you to make sure that your algorithm detects each object only once. The figure on the right has a 19×19 grid and while technically this car has only one midpoint, it is quit possible that that other cells might also indicate that the midpoint is in their grid eventually ending up with multiple detections of each object. What non-max suppression does it cleaning these incorrect detections. Concretely, it first looks at the probabilities associated with these detections, namely c_p, taking the largest probability (i.e. most confident detection) and highlight it. Having done that, the non-max suppression looks at the remaining rectangles and all the ones with a high IoU (i.e. high overlap) will get suppressed. In the end every rectangle is either highlighted or suppressed. Now let's right down the non-suppression algorithm min pseudocode and simplify y in the car detection example, where $y = \begin{pmatrix} p_c & b_x & b_y & b_h \end{pmatrix}$ (i.e. only one class). Say that we will discard all the boxes with $p_c \leq 0.6$. Next, while there are any remaining boxes:

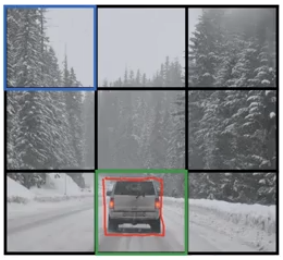
- Pick the box with the largest p_c. Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step.

Now, when you have multiple output classes it is wise to run non-max suppression algorithm separately for every class.

4.4.8 Anchor Boxes

One of the problem with object detection is that each of the grid cells can only detect one object. A way to make it possible for a grid cell to detect multiple object/overlapping objects is using Anchor Boxes. The idea of anchor boxes is that you pre-define two different shapes called anchor boxes or anchor boxes shapes. Subsequently, you are able to associate two predictions with the two anchor boxes. In general, you might use more than two anchor boxes. What you will do is define y for both anchor boxes, where $y = \begin{pmatrix} p_c & b_x & b_y & b_h & C_1 & C_2 & C_3 & p_c & b_x & b_y & b_h & C_1 & C_2 & C_3 \end{pmatrix}$. In this example anchor box 1 represent the first 8 components and anchor box 2 the other 8 components. Now what you do is compare the detect object's rectangle with the anchor boxes. So, previously each object in training image is assigned to grid cell that contains that object's midpoint, where the target output was $3 \times 3 \times 8$. With two anchor boxes, each object in training is assigned to grid cells that contains object's midpoint and anchor box for the grid with the highest (i.e. grid cell and anchor box). Consequently the target output will be $3 \times 3 \times 2 * 8$ (i.e. $3 \times 3 \times 16$). Two cases that this algorithm doesn't handle very well is when you have two anchor boxes but three objects in the same grid cell or when two objects in the same grid are both most associated with one anchor boxes. However, this should not happen very often (especially when you have a large number of grids). Finally, how do you choose the anchor boxes? Most people just choose by hand five or ten anchor box shapes that spans a variety of shapes that seems to cover the types of objects you seem to detect. A much more advantaged version however is to use a K-means algorithm to group together two types of object shapes you tend to get. This is however, a more advanced way to automatically choose the anchor box shapes.

4.4.9 YOLO Algorithm



So far, you have seen most of the components of object detection. Now, let's put all these components together to form the YOLO object detection algorithm. First, let's so how you construct your training set. Say you want to classify three types of objects (e.g. pedestrian, car and motorcycle) with two anchor boxes where $y = \begin{pmatrix} p_c & b_x & b_y & b_h & C_1 & C_2 & C_3 & p_c & b_x & b_y & b_h & C_1 & C_2 & C_3 \end{pmatrix}$, y will be $\times\ 3\ \times\ 2(\#\ \text{anchors})\ \times\ 8\ (5\ +\ \#\ \text{classes})$ (i.e. $3 \times 3 \times 16$). For the first grid cell (blue) with no object y will be as follows: $y = \begin{pmatrix} 0 & ? & ? & ? & ? & ? & ? & ? & 0 & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$. Now say you pass a grid cell that does contain an object (green), y might look like this: $y = \begin{pmatrix} 0 & ? & ? & ? & ? & ? & ? & ? & 1 & b_x & b_y & b_h & b_w & 0 & 1 & 0 \end{pmatrix}$. Eventually you end up with nine 16-dimensional vectors. Keep in mind that in practice, the number of grid cells will be much higher. So far, this include training with a convolutional network where you input a $100 \times 100 \times 3$ image and yield a output target of size $3 \times 3 \times 16$. Next, you algorithm is ready to make predictions. Finally you run this through non-max suppression:

- When using two anchor boxes, for each gird cell, you get two predicting bounding boxes.
- Get rid of low probability predictions.
- If you have three classes, for each class (pedestrian, car and motorcycle) use non-max suppression to generate final predictions.

4.4.10 Region Proposals

If you look at the object detection literature, there is a set of ideas called region proposals. If you recall the sliding windows idea you would take a trained classifier and run it across all of these different windows and run the detector to see whether there is an object present. You could this using a convolutional network, however a downside is that the algorithm is crossing a lot of regions with no objects. What is proposed to use a R-CNN, which stand for Regions with Convolutional Networks[8]. What it does, it picks a few regions that make sense to run your classifier on. In order to figure out what could be potential objects you must run a segmentation algorithm that results in the output to the right. Subsequently, you could focus on the blobs since these might be objects. It turns out that the R-CNN algorithm is still quit slow. That's why faster algorithms are being developed:

- R-CNN: Propose regions. Classify proposed regions one a time. Output label + bounding box.
- Fast R-CNN[9]: Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions. One of the problems of this algorithm is that the clustering step is still quit slow.
- Faster R-CNN[10]: Use convolutional network to propose regions.

4.5 Face Recognition

4.5.1 What is face recognition?

First, let's discuss some denotation regarding face verification vs. face recognition

Verification

- Input image, name/ID
- Output whether the input image is that of the claimed person

Recognition

- Has a database of K persons
- Get an input image
- Output ID if the image is any of the K persons (or "not recognized")

4.5.2 One Shot Learning

One-shot learning in face recognition is means that for face recognition applications you need to be able to recognize a person given just one single image or just one example of a person's face. Historically, deep learning algorithms don't work well if you have only one training example. Let's say you have a database of four employees and someone shows up at the office and want to let through at the turn style. In this case you need to learn from one example to recognize the person again. What you can do is feed an input image to a CNN and have it output a label using softmax. This doesn't work well on a small training set to make a robust network. To carry out face recognition, one-shot learning is instead learn a similarity function which outputs $d(img1, imag2) =$ *degree of difference between images*, where you have a threshold τ. Consequently, *if* $d(img1, img2) \leq \tau$ then the person in the both images is the same, whereas *if* $d(img1, img2) > \tau$ the person in both images is a different person.

[8] Girshik *et al.*, 2013: Rich feature hierarchies for accurate object detection and semantic segmentation
[9] Girshik, 2015: Fast R-CNN
[10] Ren *et al.*, 2016: Faster R-CNN: Towards real-time object detection with region proposal networks.

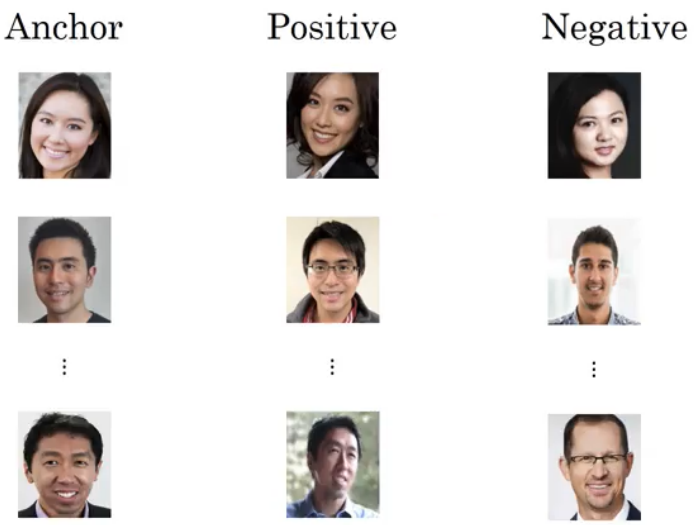


Figure 29: Triplet loss function compares pairs of images. In the terminology of the triplet loss, you are always going to look at one anchor A image and a positive image P (i.e. same person) as well as an anchor image and a negative image N (i.e. a different person)

4.5.3 Siamese Network

A good way to apply the functionality of the function d is by using a Siamese network[11]. You are used to feeding a input image (say $x^{(1)}$) into a convolutional network and through a sequence of pooling and fully connected layers you end up with a feature vector sometimes this is fed to a softmax function in order to make a classification. In this section we will not focus on a softmax function but on a fully connected layer that lies deeper in the network (say of 128 numbers). We will denote this 128-dimensional list with $f(x^{(1)})$ and you should think of the element as an encoding of the input image $x^{(1)}$. The way you can build a face recognition system is, i.e. you want to compare $x^{(1)}$ with another picture $x^{(2)}$, what you can do is feed the second picture to the same neural network with the same parameters and get a different 128-dimensional vector encoding the second picture $f(x^{(2)})$. Now you can think of d as the following:

$$d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(2)})||_2^2 \tag{23}$$

This is also called a Siamese network architecture. So to put it more formally, you have parameters of a network which define an encoding $f(x^{(i)})$. Subsequently you want to learn parameters so that:

- if $x^{(1)}, x^{(2)}$ are the same person, $||f(x^{(1)}) - f(x^{(2)})||^2$ is small.
- if $x^{(1)}, x^{(2)}$ are different persons, $||f(x^{(1)}) - f(x^{(2)})||^2$ is large.

4.5.4 Triplet Loss

One way to learn the parameters of the neural network so that it gives you a good encoding for you pictures of faces is to define an applied gradient descent on the triplet loss function[12]. To apply the triplet loss, you need to compare pairs of images (see Figure 29). In the terminology of the triplet loss, you are always going to look at one anchor A image and a positive image P (i.e. same person) as well as an anchor image and a negative image N (i.e. a different person), where the goal is:

$$||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2 \tag{24}$$

where $||f(A) - f(P)||^2 = d(A, P)$ and $||f(A) - f(N)||^2 = d(A, N)$. If you shuffle the elements in the stated equations you get:

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq 0 \tag{25}$$

To make sure the neural network doesn't just output zero for all encoding, in other words, to make sure that it doesn't set all the encodings equal to one another we add an additional margin α:

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0 \tag{26}$$

$$||f(A) - f(P)||^2 + \alpha \leq ||f(A) - f(N)||^2 \tag{24}$$

[11] Taigman *et al.*, 2014: DeepFace closing the gap to human level performance
[12] Schroff *et al.*, FaceNet: A unified embedding for face recognition and clustering

Figure 30: Examples of Neural style transfer. Content is denoted with C, the style is denoted with S and the generated image is denoted with G

Now let's formalize equation 24 and define the triplet loss function. Given 3 images, A, P, N we are going to define the loss as follows:

$$L(A, P, N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0) \tag{27}$$

where

$$J = \sum_{i=1}^{m} L(A^{(i)}, P^{(i)}, N^{(i)}) \tag{28}$$

Say you have a training set of $10k$ pictures of $1k$ persons, you have to take the $10k$ pictures and use it to generate, to select triplets and train your learning algorithm using gradient descent. Notice that in order to define this dataset of triplets, you do need some pairs of A and P. So for the purpose of training your dataset you need data with multiple pictures of the same person. Now how do you choose your triplets? One of the problems that arise when A, P and N are chosen randomly, $d(A, P) + \alpha \geq d(A, N)$ is easily satisfied. So, to construct a training set, you want to choose triplet that are "hard" to train on where $d(A, P) \approx d(A, N)$. The effect of this is that it increases the computational efficiency of your learning algorithm.

4.5.5 Face Verification and Binary Classification

The triplet loss function is one good way to learn the parameters of a convolutional network for face recognition. Another way to learn these parameters is by posing face recognition as a binary classification problem, where the output $\hat{y}$ is 1 if it is the same person and 0 if it is a different person. More formally:

$$\hat{y} = \sigma(\sum_{k=1}^{128} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b) \tag{29}$$

Now, you might think of the 128 numbers as features that you feed into logistic regression, where you have additional parameters w, i, and b. You can have variations on this formulation. For instance, you can replace $|f(x^{(i)})_k - f(x^{(j)})_k|$ with $\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$ which is also called the χ^2-similarity. So, the input is pair of images and the output is either 0 or 1. And same as before, you are training a Siamese network meaning parameters are similar.

4.6 Neural Style Transfer

4.6.1 What is neural style transfer?

Examples of neural style transfer are given in Figure 30, where the content, style and generated image are denoted with S, C and G, respectively. In order to implement neural style transfer you need to look at the features extracted from G by a convolutional network at various layers, both shallow and deep.

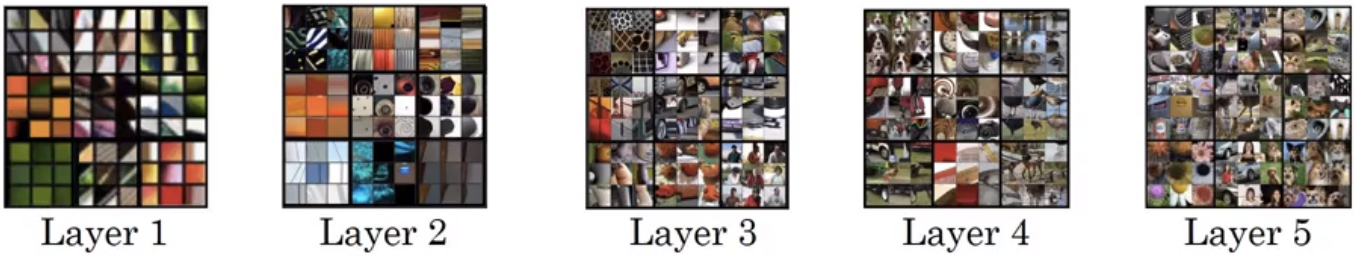


Figure 31: Visualizing deep layers of a ConvNet

4.6.2 What are deep ConvNets learning?

Now what are deep ConvNets actually learning? You can visualize what a deep network is learning by picking a unit in layer 1 and fine the nine image patches that maximize the unit's activation[13]. You can repeat this for other units (see Figure 31). You will see that in the deeper layers a hidden unit will see a larger region of the image where at the extreme each pixel could hypothetically affect the output of these later layers in the network. In other words, the earlier (shallower) layers of a ConvNet tend to detect lower-level features such as edges and simple textures, and the later (deeper) layers tend to detect higher-level features such as more complex textures as well as object classes.

4.6.3 Cost Function

Let's discuss the neural style transfer cost function[14]. By generating the cost function you can generate the image that you want. The cost function J of G:

$$J(G) = J_{Content}\alpha(C, G) + \beta J_{Style}(S, G) \tag{30}$$

where you compare the generated image G with the content C and style S. The way the algorithm would run is as follows:

1. Initiate G randomly, e.g.: $G : 100 \times 100 \times 3$
2. Use gradient descent to minimize $J(G)$ where $G := G - \frac{\alpha}{\alpha G} J(G)$

4.6.4 Content Cost Function

As you recall, the neural style transfer cost function had a content cost component and a style cost component. In this section we will discuss the content cost function $J_{Content}(C, G)$. Say you use hidden layer l to compute content cost. If l is very small (shallow layer), then the algorithm will really force your generated image to pixel values very similar to your content image. On the other hand, when l is very large (deep layer), the pixel values are not so similar to your content image. Ideally, you want to pick a layer in between. What you can do is use a pre-trained ConvNet (e.g. VGG network) and measure given a content image and a generated image how similar they are in content. So let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l in the images. If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content. Now $J_{Content}(C, G)$ is computed as follows:

$$J_{Content}(C, G) = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2 \tag{31}$$

So $J_{Content}(C, G)$ is actually the element-wise sum of squared difference between the activations in layer l between image C and G. And so when you perform gradient descent later on this will incentive the algorithm to find an image G so that these hidden layers activations are similar to what you got for the content image.

4.6.5 Style Cost Function

Now let's discuss the style cost function $J_{Style}(S, G)$. Now what does the style of an image mean? Say you are using layers l's activation to measure "style". What we are going to do is define style as the correlation between activations across channels n_c, which indicated how often high-level features (e.g. vertical texture and orange tint) occur together (high correlation) and don't occur together (low correlation) in different parts of the image. Now let's formalize this intuition. What you do is compute a style matrix which will measure all those different correlations:

[13] Zeiler and Fergus, 2013: Visualizing and understanding convolutional networks
[14] Gatys *et al.*, 2015: A neural algorithm of artistic style.

- Let $a^{[l]}_{i,j,k} =$ *activation at (i,j,k).* $G^{[]l}$ is $n^{[l]}_c \times n^{[l]}_c$ where i indexes height, j indexes width and k index the channel. So $G^{[l]}_{kk'}$ will measure how correlated the activations are in channel k compared to the activations in channel k'. k and k' will range from 1 through n_c.
- Style matrix $G^{[l]}_{kk'} = \sum_i \sum_j a^{[l]}_{ijk}\ a^{[l]}_{ijk'}$, where it iterates over the height i and width j.

Now, you would to this for the style image S and the generated image G:

$$G^{[l](S)}{}_{kk'} = \sum_i \sum_j a^{[l](S)}{}_{ijk}\ a^{[l](S)}{}_{ijk'} \tag{32}$$

$$G^{[l](G)}{}_{kk'} = \sum_i \sum_j a^{[l](G)}{}_{ijk}\ a^{[l](G)}{}_{ijk'} \tag{33}$$

You end up with two style matrices which you combine in a final style cost function:

$$J^{[l]}_{style}(S,G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} ||G^{[l](S)} - G^{[l](G)}||^2_F \tag{34}$$

which equals:

$$J^{[l]}_{style}(S,G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} (G^{[l](S)}{}_{kk'} - G^{[l](G)}{}_{kk'})^2 \tag{35}$$

Finally it turns out that you get visually more satisfying results if you use multiple layers if you use the style cost function from multiple layers:

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S,G) \tag{36}$$

where $\lambda^{[l]}$ are a set of additional hyperparameters.

4.6.6 1D and 3D Generalizations

You have learned a lot about ConvNets. Most of the discussion has focused on 2D data it turns out many of the ideas you have learned also apply to 1D and 3D datasets. One example of a 1-dimensional dataset is an ECG signal (electrocardiogram) which measures heart rhythms to make medical diagnosis. An example of a 3-dimensional dataset is CT-scan (X-ray scan), where you can scan through different slices of the human body (width, height and depth).

5 Sequence Models

5.1 Recurrent Neural Networks

5.1.1 Why sequence models

Let's start by looking at a few examples of where sequence models can be useful. For instance, in speech recognition you have been given an input audio clip X and asked to map it to a text transcript Y. Both the input and output here are sequence data (the input is an audio file which plays over time and the output is a sequence of words). Other examples are shown in Figure 32. All these examples can be addressed as supervised learning with labelled data (x, y) as training set. However, you can tell from this list of examples that there are a lot of different types of sequence problems. In some both in- and output contain sequenced data (e.g. speech recognition) and others have only sequenced data as output (e.g. music generation).

5.1.2 Notation

For the notation we are going to use an example that is part of a subtask of information extraction, namely Named entity recognition that seeks to locate and classify named entities (i.e. names of persons, organizations, locations, expressions over time, quantities etc.). This is part of a ML branch called Natural Language Processing (NLP). Let's say, we want to extract the names from the following sentence:

"Harry Potter and Hermione Granger invented a spell."

We will use the following notation:

- x is the input ("Harry Potter and Hermione Granger invented a spell."), where $x^{(i)<t>}$ is the label of the t^{th} word in the i^{th} training example,which can be either 0 (i.e. not a name) or 1 (i.e. name). Moreover, $T_x^{(i)}$ indicates the number of words for the i^{th} training example.
- y is the output, where $y^{(i)<t>}$ is the label of the t^{th} word in the i^{th}training example, which can be either 0 (i.e. not a name) or 1 (i.e. name). Moreover, (for now) T_x indicates the number of words for the i^{th} training example (equal to T_x).

Now how would you represent individual words in a sentence? You would have a vocabulary containing a variety of words (e.g. "a", "Aaron", ..., "and", ..., "Harry", ..., "Zulu"). For this example, we will use a dictionary containing 10.000 (most-common used) words which is not so much. In practice, for NLP applications you would use vocabularies of larger sizes. So $x^{(i)}$ would be an t-dimensional one-hot vector containing 0's and a 1 who's index is corresponding to the index in the vocabulary.

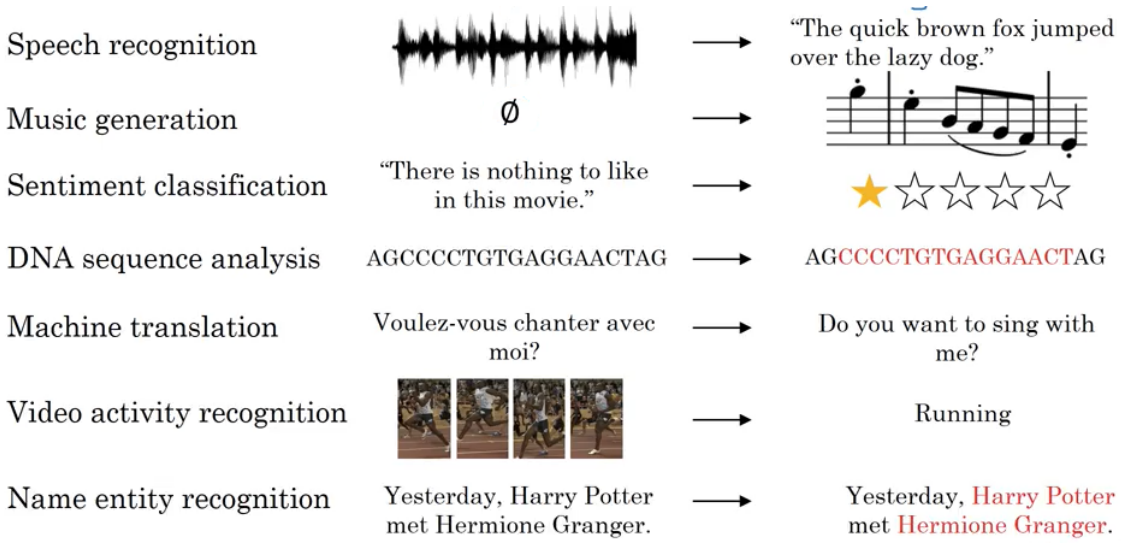
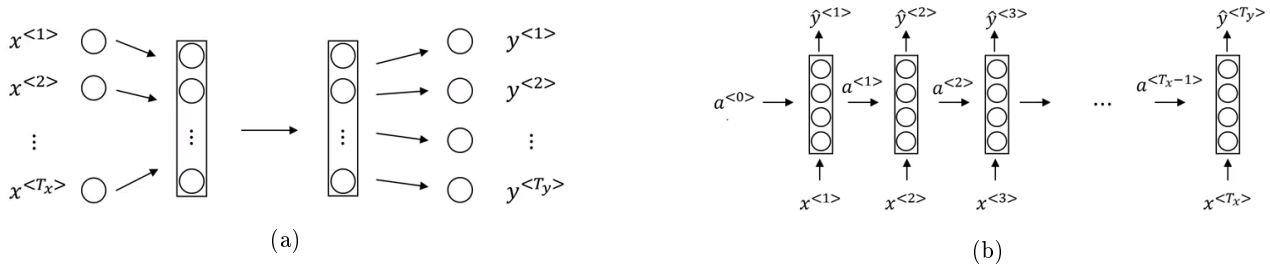


Figure 32: Examples of sequence data



(a) (b)

Figure 33: Forward propagation in a (a) standard neural network and a (b) recurrent neural network.

5.1.3 Recurrent Neural Network Model

In the last section you got familiar with the notation we will use to define sequence learning problems. Now let's discuss how to build a model, to learn the mapping from x to y. One thing you could do is try to use a standard neural network (see Figure 33a). There are two downsides to this approach. One is that the input and output can be of different lengths in different examples. In addition, in a network architecture like this, different features are not learned across different positions of the text. So instead, we propose a different type of neural network, namely a Recurrent neural network. Now what is a recurrent neural network? An example is shown in Figure 33b. What is characteristic of a RNN is that is passes on the activation to the next steps. It scans through the data from left to right. One limitation of using RNNs is that the prediction at a certain time used inputs or uses information from the inputs earlier in the sequence, but not from information later in the sequence, where it parameters it used for each time step are shared (unidirectional RNN). However, we will address this in a later section discussing bidirectional RNNs. We will now discuss forward propagation:

- $a^{<0>} = \overrightarrow{0}$
- $a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$
  $\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y)$

  The activation function that are most-commonly used are $tanh()$ and (to a lesser degree) $ReLU()$. However, the activation function you use, depends on what type of output you have.
- $a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$
  $\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$

  Now let's simplify these two equations:
- $a^{<t>} = g(W_a[a^{<t-1>}, x^t] + b_a)$
  $\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

  The advantage of this notation is that rather carrying around two parameter matrices, W_{aa} and W_{ax}, we can compress them into just one parameter matrix W_a.

5.1.4 Backpropagation through time

Forward propagation runs from left through right (see Figure 33b). In contrast, during backpropagation you run from right to left. Now the loss function will be formulated as follows:

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>}log(\hat{y}^{<t>}) - (1 - y^{<t>})log(1 - \hat{y}^{<t>}) \tag{37}$$

Which equals:

$$L(\hat{y}, y) = \sum_{t=1}^{T_x} L^{<t>}(\hat{y}^{<t>}, y^{<t>}) \tag{38}$$

where you compute the loss per time stamp. Backpropagation now just requires doing computations in the opposite direction, which allows you to compute all the appropriate quantities that let's you take the derivatives and updates the parameters using gradient descent. This is also defines as "Backpropagation through time" where you go from the right to left (i.e. backwards in time).

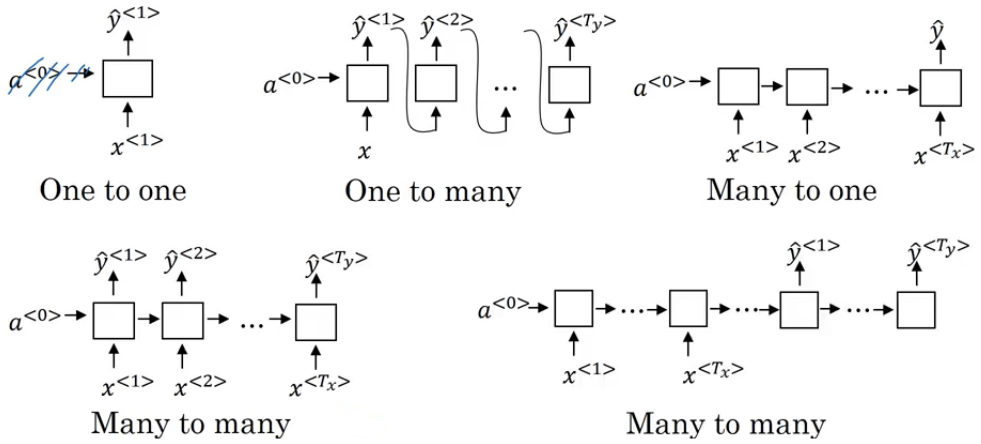


Figure 34: Examples of RNN architectures: One-to-one: standard generic neural network; One-to-many: music generation; Many-to-one: sentiment classification or sequence generation; Many-to-many: name entity recognition and machine translation (with $T_x = T_y$, left or $T_x \neq T_y$, right).

5.1.5 Different types of RNNs

So far, you have seen an RNN architecture where the number of inputs T_x is equal to the number of output T_y. However, it turns out that in other applications T_x may not equal T_y. In this section we will discuss a much richer family of RNN architectures (see Figure 34). Let's take another look at Figure 32 where different type of sequence data are shown. Examples where $T_x \neq T_y$ include music generation, sentiment classification and name entity recognition. In the examples so far $T_x = T_y$ with an architecture similar to the architecture shown in Figure 33b. This type of architecture is called Many-to-many because the input sequence has many inputs and the output sequence has many outputs. Now let's say you want to address sentiment classification where x is going to be a sequence and y a number (e.g. from 1 to 5). This type of architecture is called a Many-to-one architecture. In addition, you also have a One-to-one RNN and a One-to-many RNN. An example of a One-to-many architecture would be music generation. Another interesting example of a Many-to-many RNN architecture is when $T_x \neq T_y$, for instance in machine translation.

5.1.6 Language model and sequence generation

Language modelling is one of the most basic and important tasks in NLP. What a language model does is given any sentence, it gives the probability of that sentence being a particular sentence. To put it more formally, the model receives an input an estimates the probability of a particular sequence of words: $P(y^{<1>}, y^{<2>}, \ldots, y^{<T_y>})$. Now how do you train a language model?

1. Training set: large corpus of English (or any other language) test.
2. Input sentence (e.g. "Cats average 15 hours of sleep a day.")
3. Tokenize: $y^{<1>}, y^{<2>}, \ldots, y^{<T_y>}$. Additional tokens include $< EOS >$ ("End Of Sentence") and $< UNK >$ ("Unknown", word that is not in the dictionary).
4. Build RNN to model the probability of the different sequences
   - Forward propagation: Now the RNN will propagate from left to right where in each step the RNN will predict given a set of preceding words what the next word in the sequence. So, $a^{<1>}$ will predict the first word "cat", consequently, $a^{<2>}$ will predict "average" given the preceding words $P(\_|\textit{"cats"})$, and so on ($a^{<3>} = P(\_|\textit{"cats average"})$).
   - Now the loss softmax function was formulated as follows $L(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} log(\hat{y}_i^{<t>})$, which can also be defined as $L = \sum_t L^{<t>}(\hat{y}^{<t>}, y^{<t>})$.

5.1.7 Sampling novel sequences

After you have trained a sequence model, one of the ways you can informally get a sense of what is learned is to have a sample novel sequences. Now remember that a sequence model models the chance of any particular sequence of words. Now what we are going to do is sample from this distribution to generate novel sequence of words. To train the network you used an architecture as shown in Figure 33b. For sampling, we will use a architecture which is slightly different (see

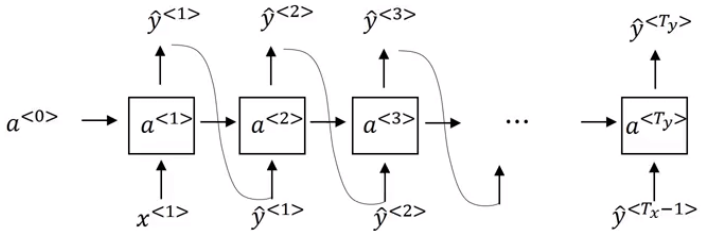


Figure 35: RNN architecture for sampling novel word sequences.

Figure 35), where your algorithm generates tokens given the set of previous generated words. You're vocabulary can have different token-levels:

- Word-level language model Vocabulary = [a, aaron, ..., zulu, $< UNK >$]
- Character-level language model Vocabulary = [a,b,c,...,z, , ,0,9,A,B,C,...,Z]
  - One advantage is that $< UNK >$ elements won't occur any more.
  - Disadvantage is that sequences will have many more tokens (computational expensive and slow).

5.1.8 Vanishing gradients with RNNs

One of the problems with a basic RNN algorithm is that it runs into vanishing gradient problems. The basic RNNs we have discussed are not very good at capturing long-term dependencies. That is, when correlated words are far apart from each other in a sentence. In other words, the basis RNNs have local influences, meaning that output $y^{<i>}$ is mainly influenced by outputs close to $y^{<i>}$. Another concern, although less prominent, is that of exploding gradients, where exponentially large gradients can cause the parameters to become so large that your network is botched. This however can be solved by applying gradient clipping, which looks at your gradient vectors and rescales them when they exceed a certain threshold. Vanishing gradient however is much harder to solve and will be discussed in the next few sections.

5.1.9 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) is a modification to the RNN hidden layer that makes it much better in capturing long-range connections and helps a lot with the vanishing gradient problems. Let's take another look at the formula for computing the activations, where $a^{<t>} = g(W_a[a^{<t-1>}, x^t] + b_a)$. A lot of the ideas regarding GRU are due to two papers[15][16]. As a example sentence we will use "The cat, which already ate ..., was full", where the word "cat" and the verb "was" are related. So as we read this sentence from left to right the GRU unit is going to have a new variable called c, which stand for memory cell. What the memory cell does, it provides a bit of memory to remember for example, whether the cat was singular or plural, so that when it gets much further into the sentence it can still work under consideration whether the subject of the sentence was singular or plural, where at some point in time $c^{<t>} = a^{<t>}$. Now, at every time step we are going to consider overwriting the memory cell with a value $\tilde{c}^{<t>}$, which is a candidate for replacing $c^{<t>}$ where $\tilde{c}^{<t>} = tanh(W_c[c^{<t-1>}, x^t] + b_c)$. The important idea of the GRU is that we are going to have a gate Γ_u ("update") which will have a value between 0 and 1 and where $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^t] + b_u)$. Now to recap:

$$a^{<t>} = c^{<t>} \tag{39}$$

where you have a memory cell $c^{<t>} = a^{<t>}$ and a candidate value:

$$\tilde{c}^{<t>} = tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c) \tag{40}$$

which probability for updating will be controlled by a gate:

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \tag{41}$$

The value for Γ_u is most likely to be either 0 or 1. In other words, the gate will decide whether to update and replace $c^{<t>}$ or not. Now the specific equation we will use for the GRU is as follows:

$$c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>} \tag{42}$$

[15] Cho *et al.*, 2014: On the properties of neural machine translation: Encoder-decoder approaches
[16] Chung *et al.*, 2014: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modelling

One important note is that $c^{<t>}, \tilde{c}^{<t>}$ and Γ_u have the same dimension. Therefore, for proper mathematics you need to apply element-wise multiplications. So far, we have described a simplified GRU unit, now let's describe a more complex one:

$$\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \tag{43}$$

where Γ_r stand for relevance, which tells you how relevant $c^{<t>}$ is, where:

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \tag{44}$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \tag{45}$$

$$c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>} \tag{46}$$

Now why not use a simplified GRU? It turns out that over many years, researchers have experiment with many different possible versions of how to design these units. The GRU however, is a standard one that is just commonly used. Another common version is called a LSTM which stand for Long Short Term Memory which we will talk about in the next section.

5.1.10 Long Short Term Memory (LSTM)

In the last section we spoke about GRU which allows you to learn very long-range connections in a sequence. Another type of unit that allows you to do this very well is LSTM which stand for Long Short Term Memory[17], which is a more general and even more powerful version of the GRU. To recap, these are the formulas used for the GRU unit:

GRU

$$\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \tag{43}$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \tag{44}$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \tag{45}$$

$$c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>} \tag{46}$$

$$a^{<t>} = c^{<t>} \tag{39}$$

LSTM

$$\tilde{c}^{<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \tag{47}$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \tag{48}$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \tag{49}$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \tag{50}$$

$$c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + \Gamma_f \times c^{<t-1>} \tag{51}$$

$$a^{<t>} = \Gamma_o \times c^{<t>} \tag{52}$$

So a LSTM network used a separate update and forget gate, as it has three gates instead of two. You can put multiple LSTM units in a sequence where it is relatively ease to pass values through the network and memorizing values for a longer period of time. As you can imagine, there are also a few variations, where one adds the value $c^{<t-1>}$ to the gate computations also called a peephole connection, meaning that the gates values also depend on the previous memory cell value. The architecture of a GRU and LSTM are visualized in Figure 36.

[17] Hochreiter & Schmidhuber, 1997: Long short-term memory.

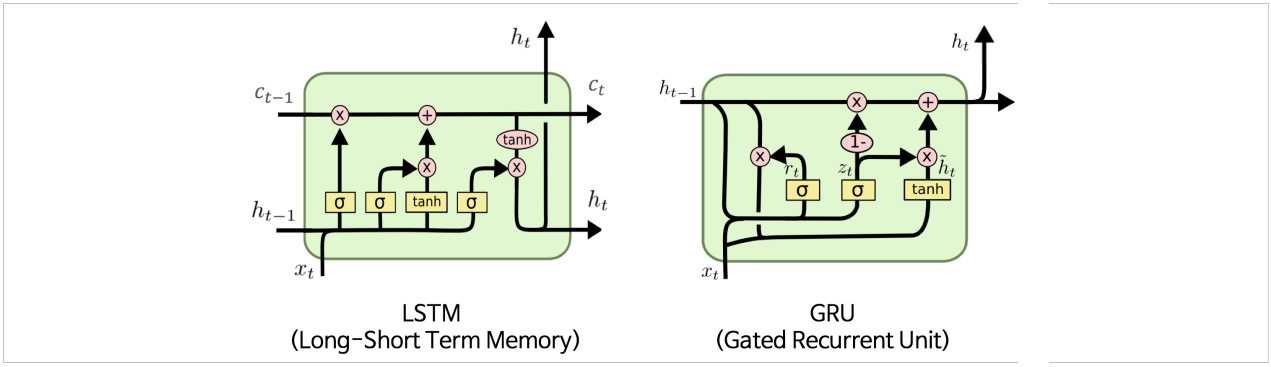


Figure 36: Architectures of a Gate Recurrent Unit and a Long Short Term Memory network.

5.1.11 Bidirectional RNN

By now you have seen the main building blocks of RNNs. There are just two more ideas that let you build more powerful models, one is bidirectional RNN (BRNN), which lets you take information from both earlier and later elements in the sequence. In a BRNN you will build a backward connection in addition to the forward propagation which will both predict $\hat{y}$:

$$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y) \tag{53}$$

This allows the prediction at a certain time step to take as input both information from the past, as well as information from the present, as well as information from the future. The blocks can be regular RNN, GRU and LSTM units. The second idea we will explore are deep RNNs which combines all the different things you have learned so far about RNNs and will be discussed in the next section.

5.1.12 Deep RNNs

The different versions of RNNs you have seen so far will already work quite well. However, for learning very complex functions it is sometimes useful to stack multiple layers of RNNs together to build even deeper versions of these models . A slight change in notation is applied where we specify the hidden layer l with $a^{[l]<t>}$. Now the following applies:

$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^2)$$

So whereas for standard RNNs you have seen neural networks that are very deep. For RNNs, having three layers is already quite a lot because of the temporal dimension, these networks can already get quite big even if you have just a small handful of layers.

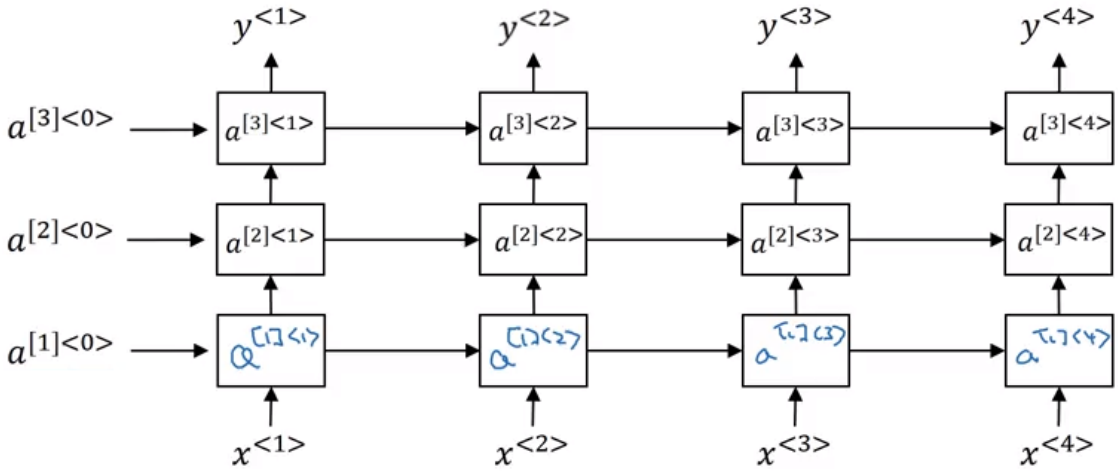


Figure 37: One hidden layer of a deep RNN.

5.2 Introduction to Word Embeddings

5.2.1 Word Representation

In this section you see how RNNs, GRUs and LSTMs can be applied to NLP. One of the key ideas you learn about is word embeddings which is a way of representing words including analogies like man is to woman as king is to queen. So far we have represented words by using a vocabulary [a, aaron, ..., zulu, $< UNK >$] ($|V| = 10.000$) with one-hot vectors where 1 corresponds to the index of the respective word and 0 to the other words. One of the weaknesses of this representation is that it treats each word as a thing onto itself and it doesn't allow an algorithm to easily generalize cross words. Now let's propose to use instead of a one-hot vector presentation a featurized representation where the algorithm could learn a set of features and values for each word (see Figure 38a). You end up with high-dimensional feature vectors, which gives a better representation than a one-hot vector. If we are able to learn a 300-dimensional feature vector or 300 dimensional embeddings of each word, one of the popular things to do is to embed in a two-dimensional space so that you can visualize them (see Figure 38b. A common algorithm doing this is t-SNE algorithm[18].

5.2.2 Using word embeddings

Word embeddings is used in combination with transfer learning as follows:

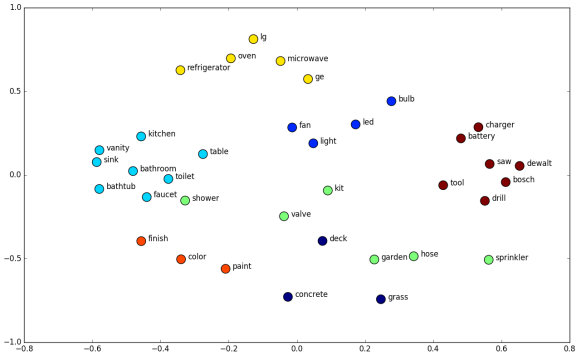
1. Learn word embeddings from large text corpus ($1 - 100B$ words)
    - Say you have two sentences: *"Robert Lin is an apple farmer"* and *"Robert Lin is a durian cultivator"*. Using word embeddings you can learn the words "durian" and "cultivator" from the first sentence by word embeddings. One of the reasons that word embeddings will be able to do this is the the algorithm can examine very large data sets of unlabelled text (or download pre-trained embedding online). And by examining a large amount of unlabelled text, the algorithm can figure out that "orange" and "durian" are similar as well as "farmer" and "cultivator".
2. Transfer embedding to new task with smaller training set ($100k$ words)
3. Optional: Continue to fine-tune the word embeddings with new data

So word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set. This allows you to carry out transfer learning, where you take information you have learned from huge

[18] van der Maaten and Hinton, 2008: Visualizing data using t-SNE

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	−1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

(a)



(b)

Figure 38: Word embeddings (a) in multi-dimensional and (b) two-dimensional space

amounts of unlabelled text and than transfer that knowledge to a task, such as named entity recognition for which you may have a relatively small labelled training set. While this is true for a lot of NLP tasks (e.g. text summarisation, co-reference resolution, parsing), it is less true for others (e.g. machine translation, language modelling). Finally, word embeddings has an interesting relationship with face encoding ideas that you learned about in the previous course. So if you remember, for face recognition, we train a Siamese network architecture that would learn for instance a 128-dimensional representation of different faces. In the literature encoding refers to these multi-dimensional vectors. One difference between face recognition and words embedding is that for face recognition you want to train a neural network that can take as input any face picture, and have a neural network compute an encoding for that new picture. In contrast, what we do when using word embeddings it that we have a fixed vocabulary and we have a vector that just learns a fixed encoding or learns a fixed embedding for each of the words in the vocabulary.

5.2.3 Properties of word embeddings

One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning (e.g. man is to women as king is to queen). Now how do you develop an algorithm which is able to find these analogies automatically. Now let's take another look at Figure 38a. Now what we'll do is subtract the vectors from the different words. So based on the values, the output vectors will be as follows:

$$e_{man} - e_{woman} = \begin{pmatrix} -2 \\ 0 \\ 0 \\ 0 \end{pmatrix} \text{ and } e_{king} - e_{queen} = \begin{pmatrix} -2 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Now, what this subtraction captures is the main difference between king and queen, as represented by these vector, is also to gender. Which is why the main difference between $e_{man} - e_{queen}$ and $e_{king} - e_{queen}$ is approximately the same. So when you try to solve an analogy, what you do is compute the following[19]

$$e_{man} - e_{queen} \approx e_{king} - e_w$$

When representing the 300-dimensional space, the distance between man and women will be approximately the same as the distance between king and queen. To put it more formally, to find word w:

$$arg \max_w \ sim(e_w, e_{king} - e_{man} + e_{woman}) \tag{54}$$

Earlier we spoke about t-SNE. What t-SNE does it takes 300-dimensional data and maps it in a non-linear way to a two-dimensional space. Now let's quickly describe the similarity function which is most commonly used which is called the cosine similarity function. In cosine similarity you define the similarity between two vectors as follows:

$$sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2} \tag{55}$$

What this is is basically the inner-product between u and v, so when the vectors are very similar their inner product will tend to be large. Now one of the remarkable results about word embeddings is the generality of analogy relationships they can learn.

5.2.4 Embedding matrix

Let's start to formalize the problem of learning a good word embedding. When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix. Say you have a vocabulary [a, aaron, ..., zulu, $< UNK >$] ($|V| = 10.000$) and 300 features, you end up with a embedding matrix of size 300×10.000. To compute a embedding vector you do the following. Say you have vocabulary $E = \begin{pmatrix} a & aaron & \dots & orange & \dots & zulu & < UNK > \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$ where the rows correspond to the different features (300). Say orange has index 6257 in the vocabulary, it's one-hot matrix looks

[19] Mikolov *et al.*, 2013: Linguistic regularities in continuous space word representations.

as follows: $o_{6257} = \begin{pmatrix} 0 \\ \dots \\ 0 \\ \dots \\ 1 \\ \dots \\ 0 \end{pmatrix}$ with length 10.000. Now the embedding vector is computed by $E \times o_{6257} = e_{6257}$ with sizes of 300×10.000, 10.000×1 and 300×1. To put it more formally:

$$\textit{embedding for word j: } e_j = E \times o_j \tag{56}$$

The thing to remember is that the goal is to learn an embedding vector e_j, where you initialize e randomly and learn the different features using gradient descent. When you are implementing this, it is not efficient to use one-hot vectors because you are multiplying a lot of values with 0. Alternatively, you would usually use a specialized function.

5.3 Learning Word Embeddings: Word2vec & GloVe

5.3.1 Learning word embeddings

We will now discuss some concrete work embedding algorithms. We'll start with more complex algorithms after which we will discuss the more simpler ones. Now say u want to develop a neural language model using word embeddings[20] where you want to predict a word in a sequence (e.g. "I want a glass of orange _"). Now let's propose we compute the 300-dimensional embedding matrices (see Figure 39). What we can do is fill these in a 1-hidden layer neural network which feeds the output to a softmax function, which has its own parameters. The softmax classifies among the 10.000 possible outputs in the vocabulary for the final word we want to predict. Now the hidden layer and softmax layer will have their own parameters ($W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$). Now as input we have 6 words with 300 dimension. Consequently, the input layer will be an ($300 \times 6 =$) 1800-dimensional vector. Alternatively, you can also look at a number of previous words when dealing with longer sentences. Now you can use gradient descent to maximise the likelihood of your predictions. It turns out this algorithm will learn pretty decent word embeddings. Now let's generalize this and see how we can derive even simpler algorithms. Let's say you have the sentence "I want a glass of orange _ to go along with my cereal", you can end up with different context/target pairs (here the target word is "juice"):

- Preceding words: preceding four words as context ("a glass of orange")
- Surrounding words: the words on the left and the right of the target word as context ("a glass of orange" and "to go along with")
- Last 1 word: the last one word as context ("orange")
- Nearby 1 word ("skip gram" model): a word that is nearby ("glass")

[20] Bengio *et al.*, 2003: A neural probabilistic language model

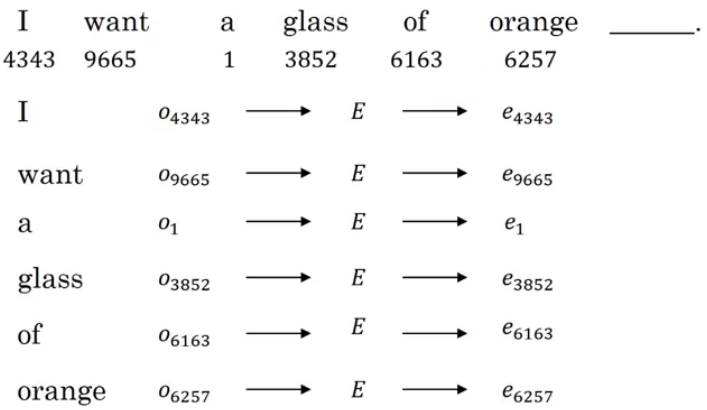


Figure 39: Computation of embedding matrices for a Neural Language Model. o refers to the one-hot vector and E to the vocabulary matrix.

5.3.2 Word2Vec

The Word2Vec algorithm is simpler and includes a more efficient way to learn word embeddings. Let's say you are given a sentence in the training set "*I want a glass of orange juice to go along with my cereal.*". Skip-grams include taking context/target pairs to create our supervised learning problem. Now, rather than having the context being the words preceding the target word, we will randomly pick the target word based on a fixed context word. The goal of this supervised learning problem is not to do well on this problem necessarily but to learn good word embeddings. Now let's propose we have a vocabulary size of 10.000 words, where we want to learn a mapping from some input context x (e.g. "orange") to some output target y (e.g. "juice", "glass"). We will have an one-hot vector e_c that will be multiplied by some embedding matrix E yielding the embedding vector e_c for the input context word. We will now feed the embedding vector e_c to a softmax unit which will output $\hat{y}$. Now the softmax model does the following:

$$\textit{softmax: } p(t|c) = \frac{e^{\Theta_t^T e_c}}{\sum_{j=1}^{10.000} e^{\Theta_j^T e_c}} \tag{57}$$

where Θ_t is the parameter associated with the object word t. Finally the loss function for the softmax will be as follows:

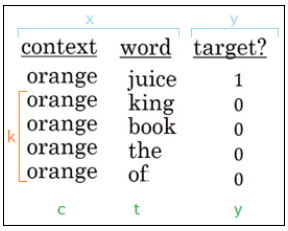
$$L(\hat{y}, y) = -\sum_{i=1}^{10.000} y_i log(\hat{y}_i) \tag{58}$$

where y is a one-hot vector. This model is called the ski-gram modelled where a skip is taken from the context to the target word (earlier or later in the sentence). One problem with using this model is the computational speed. Every time you run this model, you have to compute the sum over all the words in your vocabulary. When your vocabulary takes on larger sizes the algorithm can become really slow. There are a few solutions to this. One is using a hierarchical softmax. One quick topic we'll discuss is how to sample the context c? One way is to sample uniformly from your training corpus, however, you will find that some words are chosen really frequently (e.g. the, a, of, and, to, etc.). To compute embedding for less frequent words you can apply different heuristics to balance out sampling from the common and less-common words.

5.3.3 Negative Sampling

One major problem when using the softmax function is that you have to sum over you entire vocabulary every time you run you algorithm which can become computationally very expensive. In order to resolve this, you can use a modified algorithm called negative sampling which does something similar to the skip-gram model, however much more efficient[21]. What this algorithm does it takes a sentence (e.g. "*I want a glass of orange juice to go along with my ceral*") and make pairs of contexts and words randomly picked from the dictionary labelling them as positive (target equals 1, e.g. "orange" and "juice") or negative 0 (target equals 0, "orange" and "king"). To summarize:

1. We will take a context and a target word which will yield a positive example (= 1)
2. We will take the same context word and pair it with a word randomly chosen from the dictionary k times and label all of those as negative examples (= 0)
3. We are going to create supervised learning problem where the algorithm inputs x (i.e. pairs of words) where after it has to predict the output y.



This is how you generate the training set. Now how do you choose k? Thumb of rule, if you have a large dataset, choose k to be between 5 and 20. When dealing with a larger training set you set k to a lower value say between 2 and 5. We will now describe how the supervised learning model maps from x to y. The softmax function we spoke of in the earlier section had the following formula:

$$\textit{softmax: } p(t|c) = \frac{e^{\Theta_t^T e_c}}{\sum_{j=1}^{10.000} e^{\Theta_j^T e_c}} \tag{57}$$

If you look in the table, the context c and target word t will be input x and the target will be output y. The softmax function will look as follows:

$$P(y = 1|c, t) = \sigma(\Theta_t^T e_c) \tag{59}$$

So if you have k examples, you can think of a $1 : k$ ratio of negative to positive examples. In other words, for every positive examples, you have k negative examples with which to train this logistic regression-like model. Now to draw this

[21] Mikolove *et al.*, 2013: Distributed representation of words and phrases and their compositionality.

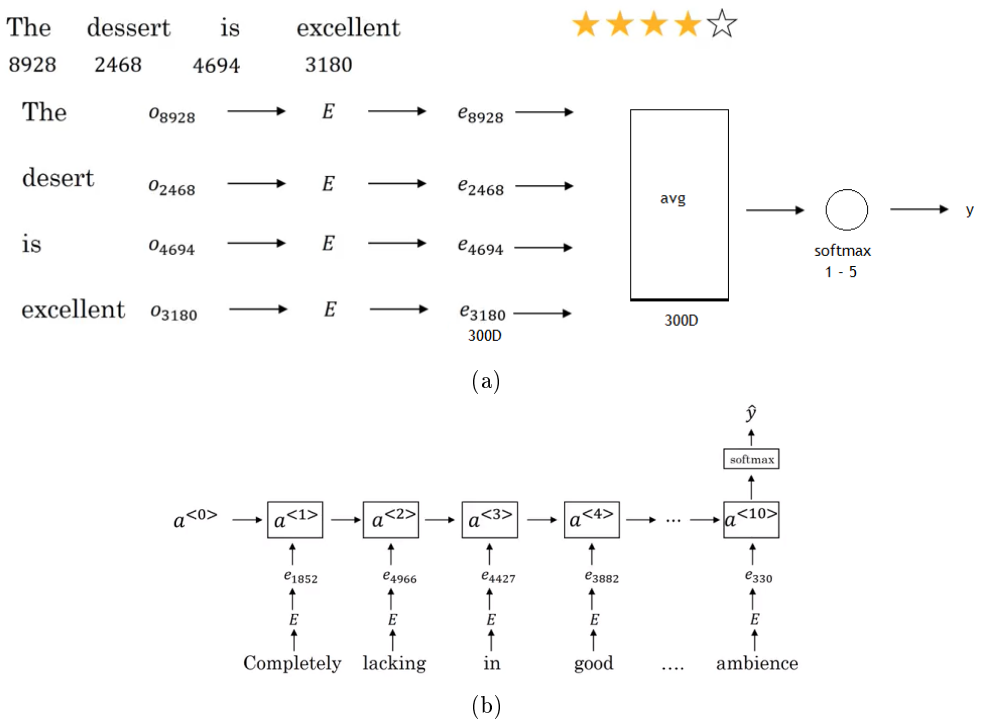


Figure 40: Sentiment classification using as (a) simple neural network and a (b) RNN.

into a neural network, say you have the word "orange" which has 6257 as index in the vocabulary. You now multiply the one-hot vector o_{6257} by the embedding matrix E yielding an embedding vector e_{6257}. Now what you have is a 10.000 possible logistic regression classification problem. However, instead of training all 10.0000 on every iteration you used train k randomly chosen negative examples, which is far less computationally expensive. Last final thing is how to choose the negative examples? One thing you can do is sampling the words according to the empirical frequency (i.e. how often different words appear). The problem is you end up only with common-used words, which is not a good representation of your vocabulary. Alternatively, you can take a heuristic value which lies between the two extremes of sampling from empirical frequencies to the uniform distribution.

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{i=1}^{10.000} f(w_j^{\frac{3}{4}})} \tag{60}$$

where w_i is the observed frequency of a word.

5.3.4 GloVe word vectors

The GloVe algorithm is also a simple algorithm although less used than the Word2Vec or the skip-gram models. GloVe is an abbreviation of "global vectors for word representation"[22]. Again we have a sentence (e.g. "*I want a glass of orange juice to go along with my cereal.*"). In the GloVe algorithm you also have a context c and target t word as well as X_{ij} which is the number of times i appears in context of j. You can think of i and j as c and t, respectively where X_{ij} can be equal to X_{ji}. Now a GloVe model optimizes the following:

$$minimize \sum_{i=1}^{10.000} \sum_{j=1}^{10.000} f(X_{ij})(\Theta_i^T e_j + b_i + b_j - logX_{ij})^2 \ where\ i = t\ and\ j = c \tag{61}$$

where $f(X_{ij})$ is a weighting term which will be equal to 0 if $X_{ij} = 0$. Also, this weighting parameter is also a function giving more weight to common-used words compared to less common-used words.

5.4 Applications using Word Embeddings

5.4.1 Sentiment Classification

Sentiment classification is one of the major building blocks of NLP and includes determining the sentiment of a text (an example is shown in Figure 40a). You can train a system to map from x to y based on a label data set, where after you can

[22] Pennington *et al.*, 2014: GloVe: Global vectors for word representation

use this to monitor people's comments. As an example, let's take a look at the first sentence "*The dessert is excellence*". Say you retrieve the embedding vectors, average them and feed them into an softmax function yielding $\hat{y}$. One problem of this algorithm is that it ignores word-order (e.g. "not good" is seen as positive). A more sophisticated model used RNN for sentiment classification (see Figure 40b). This is an example of a many-to-one RNN architecture. With an algorithm as this, it is much easier to take word sequence into account unlike the previous algorithm which simply sums of the different vectors.

5.4.2 Debiasing word embeddings

Bias in word embeddings discussed in this video includes gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model[23]. Addressing bias in word embeddings is done as follows (for more detailed information how every step is performed, you can look at the article referred to in the footnote)(see Figure 41):

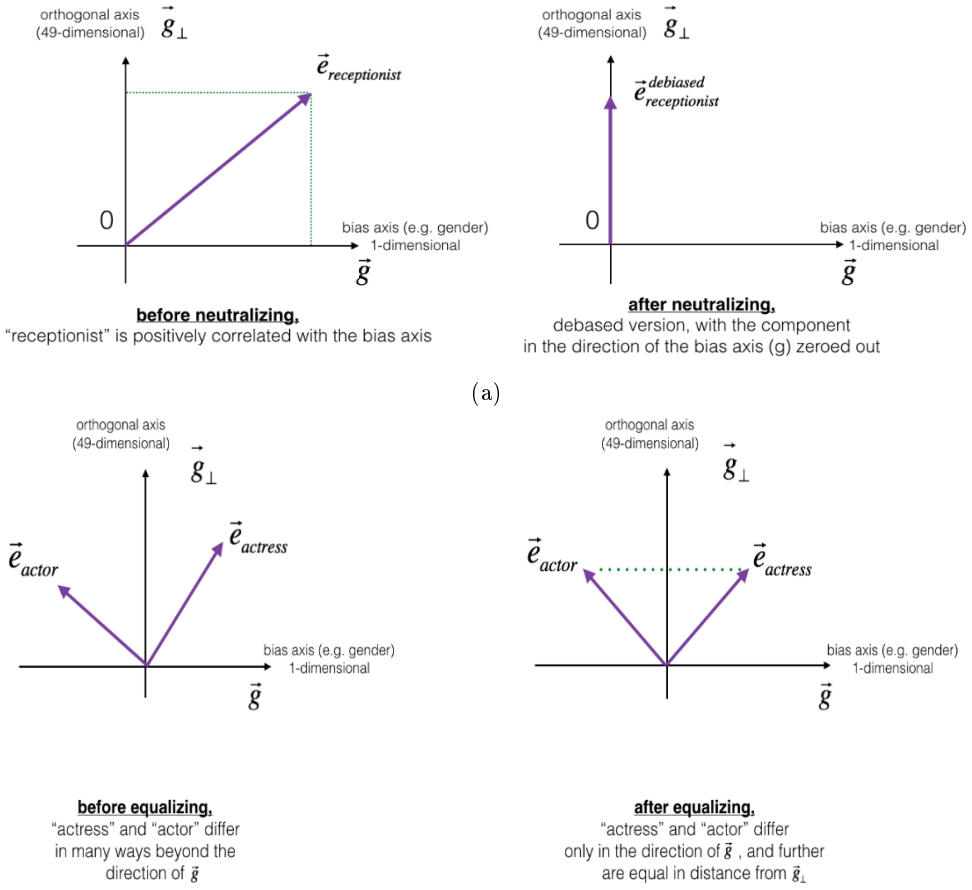
1. Identify bias direction.
2. Neutralize (Figure 41a) for every word that is not definitional ("docter", "babysitter"), project to get rid of bias.
3. Equalize pairs (Figure 41b).

5.5 Various sequence to sequence architectures

5.5.1 Basic Models

In the following sections we will discuss sequence to sequence models which are useful for machine translation to speech recognition. As an example we have an input sentence "*Jane visite l'Afrique en septembre*" $(x^{<1>}, x^{<2>}, x^{<3>}, x^{<4>}, x^{<5>})$

[23] Bolukbasi *et al.*, 2016: Man is to computer programmer as woman is to homemaker? Debiasing word embeddings



(a)

(b)

Figure 41: Debiasing word embeddings by (a) neutralizing and (b) equalizing word pairs.

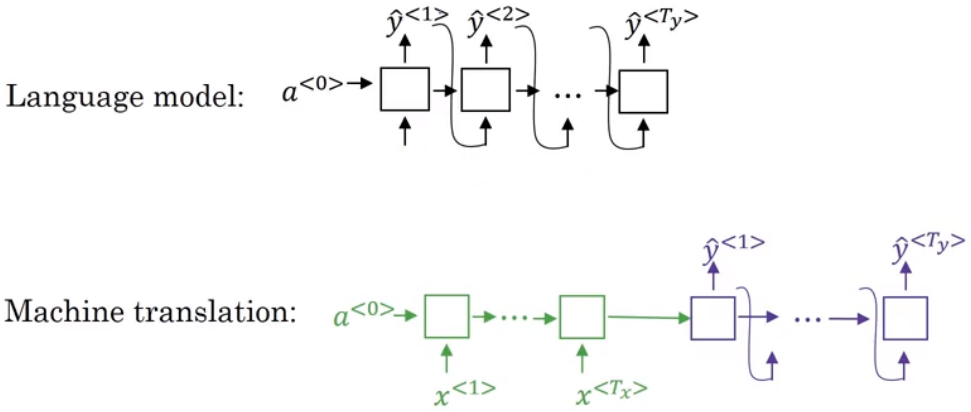


Figure 42: Language model (upper) and machine translation model (lower), where green represents the encoding RNN and purple the decoding RNN.

with the translated sentence as output, namely "*Jane is visiting Africa in September*" ($y^{<1>}, y^{<2>}, y^{<3>}, y^{<4>}, y^{<5>}, y^{<6>}$). The ideas that we spoke of in this section are mainly derived from two publications[24][25]. Now we have one encoder network which is an RNN which we feed the input one word at a time (see green RNN in Figure 42). After ingesting the input sequence the RNN outputs a vector which represents the input sentence. After this, you can build a decoder network which takes as input the encoder output and outputs the translation also one word at a time (see purple RNN in Figure 42). Eventually the algorithm outputs the sentence tokens. Remarkably, this model works well by simply using an encoder network which finds an encoding of the input and then uses a decoder network to generate the corresponding translation. A similar architecture also works really well for image captioning, which inputs an image and outputs a caption corresponding to the image. For this you can use a pre-trained Alexnet convolutional network which can be the encoder, where you feed the encoder output to a decoder network which outputs the caption one word at the time.

5.5.2 Picking the most likely sentence

There are some similarities between the sequence to sequence machine translation and the language models you have worked with earlier in the course. However, there are some significant differences as well. In Figure 42 you see a language model which predicts the probability of a sentence $P(y^{<1>}, \dots, y^{<T_y>})$ and a machine translation model with a encoder and decoder network $P(y^{<1>}, \dots, y^{<T_y>|x^{<1>}}, \dots, x^{<T_x>})$. What you may notice is that the decoder network is similar to network of the language model. However, while the language model always start with a vector of 0's $a^{<0>}$, the machine translation has an encoder network that computes some representation for the input sentence. Due to this, we call this a "Conditioned language model". In machine translation, you want to do the following:

$$arg \max_{y^{<1>},\dots,y^{<T_y>}} P(y^{<1>}, \dots, y^{<T_y>}|x) \tag{62}$$

One algorithm which can do this is called beam search, which will be discussed later. Alternatively, you can choose for a greedy search which takes the words with the highest probability consecutively. Now let's propose you have the following translations of "*Jane visite l'Afrique en septembre*":

- *Jane is visiting Africa in September*
- *Jane is going to be visiting Africa in September*

The greedy search will choose the second translation as output because "going" is more frequently used that "visiting". However, you could say that the first translation is most optimal. The greedy algorithm may therefore not find the most optimal translations.

5.5.3 Beam Search

We will now discuss the beam search algorithm. Now as you know, the goal of machine translation is to find the most optimal translation of an input sentence. In the first step of beam search, you use a network fragment to try to evaluate what the probability is of the first output y given input sentence x $P(y^{<t>}|x)$. Whereas greedy search will choose the most likely word an moves on, beam search can consider multiple alternatives, where B is used to denote the beam width

[24] Sutskever *et al.*, 2014: Sequence to sequence learning with neural networks.
[25] Cho *et al.*, 2014: Learning phrase representations using RNN encoder-decoder for statistical machine translation.

(i.e. the number of alternatives the algorithm remembers). In step two of the beam search, it will choose the second word for every B alternatives $P(y^{<2>}|x, y^{<1>})$. When we have a beam width of 3, at every step you instantiate three copies of the network to evaluate the partial sentence fragments of the output. In the third and last step of the beam search algorithm you predict the third word given the first and second word $P(y^{<3>}|x, y^{<2>}, y^{<1>})$. Again beam search will pick the three most likely words. The outcome of this process where one word is added at the time is the translated sentence. Now when B is set to 1, the algorithm essentially becomes a greedy search which was discussed in the previous section.

5.5.4 Refinements to Beam Search

Now we will make some minor changes that result in a better working beam search algorithm. One refinement is length normalization. Beam search maximizes the following probability:

$$arg\max_{y} \Pi_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>}) \tag{63}$$

These probability are values smaller than 1 which can result in numerical underflow, meaning that the value is too small for the floating part representation in your computer to store the value accurately. So in practice, instead of maximizing this product, we will use a sum of log to yield a more stable algorithm that is less prone to numerical rounding errors/underflow:

$$arg\max_{y} \sum_{t=1}^{T_y} logP(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>}) \tag{64}$$

There is one additional change that will make the beam search algorithm run better. If you have a long sentence the probability of that sentence will be very small. Therefore, the current algorithm will probably prefer shorter sentences over long sentences. To solve this we will change the function again slightly by adding a heuristic:

$$arg\max_{y} \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} logP(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>}) \tag{65}$$

which significantly reduces to penalty for computing long sentences. In addition, the hyperparameter α which softens the normalization function, where if $\alpha = 1$ the function is completely normalized by length and if $\alpha = 0$ no normalization is applied. A common-used value for α is 0.7. To wrap up, as you run beam search you end up with a lot of different possible translations. Now what you do is score the different output sentences by the log probability objective and choose the one with the highest value as your final translation. Finally, a few implementation details regarding the beam width. If the beam width is very large you consider a lot of different possibilities, therefore you end up with a good translation. However, your algorithm will become quite slow and computationally expensive in terms of the memory requirements. On the other hand, when the beam width is very small, the result will become worse with a faster and computationally less expensive algorithm. A final note, namely unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact minimum for $arg\max_{y} P(y|x)$.

5.5.5 Error Analysis in Beam Search

As you have seen earlier error analysis can help you show where you should focus your time on. Beam search is an approximate search algorithm (heuristic) meaning that it doesn't find the optimal solution guaranteed. Say you have an input sentence "*Jane visite l'Afrique en septembre*", on which your algorithm gives the following translation $\hat{y}$: "*Jane visited Africa last September*" while the optimal solution y^* is "*Jane visits Africa in September*". Now your model has two main components, namely the sequence to sequence RNN model (with an encoder and decoder network) and a beam search algorithm with fixed beam width B. Ideally, you would want to attribute this error to one of the two components, either the RNN or beam search algorithm. Now how do you decide what to do? The RNN computes $P(y|x)$. It turns out that the most useful thing to do is compute $P(y^*|x)$ and $P(\hat{y}|x)$ to see which of these to values is bigger:

- Case 1: $P(y^*|x) > P(\hat{y}|x)$. In this case, the beam search algorithm chose $\hat{y}$, but y^* attains a higher $P(y|x)$. You can therefore conclude that beam search is at fault.
- Case 2: $P(y^*|x) \leq P(\hat{y}|x)$. In this case, t$y^*$ is a better translation than $\hat{y}$, but the RNN predicted $P(y^*|x) < P(\hat{y}|x)$. You can therefore conclude that the RNN is at fault.

5.5.6 Blue Score

One of the challenges of machine translation is, given an input sentence, there could be multiple translations that are equally good. How do you evaluate this? The way this is done conventionally is by a Bleu Score (abbreviation of "Bilingual

	count	clip
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

Table 2: Computation of modified bigram precision, which is the sum of the count clipped divided by the total number of bigrams $\frac{4}{6}$. The $Count_{clip}$ gives credit only up to the maximum number of times abigram appears in the references

evaluation understudy")[26]. Say you have an input sentence "*Le chat est sur le tapis*" which has the following, equally good, translations in English:

- Reference 1: *The cat is on the mat.*
- Reference 2: *There is a cat on the mat*

Now what the bleu score does is assigning a score to the different translations, where you have a high bleu score when the machine generated translation is pretty close to any of the references provided by humans. One way to measure how good the machine translation output is, is by looking at each word in the output sentence and see if it appears in the references, this is also called a precision of the machine translation output. For instance, when you have the machine translation output "*The the the the the the the*", you would have a precision of $\frac{7}{7}$ because all of the words in the MT output are present in the references. However, it is safe to say the translation is very poor. So to resolve this we will use a modified precision in which we will give each word credit only up to the maximum number of times it appears in the references. Consequently, the MT output will now have a precision of $\frac{2}{7}$. Additionally, you would also like to look at more than only isolated words (i.e. unigrams), like word pairs of two (i.e. bigrams) or three (i.e. trigrams) word appearing next to each other. Let's say we have an alternative MT output, namely "*The cat the cat on the mat*". Possible bigrams include "the cat", "cat the", "cat on", "on the" and "the mat". The modified precision is computed as shown in Table 2. Let's formalize this a bit further for $n-grams$:

$$p_n = \frac{\sum_{n-grams \in \hat{y}} Count_{clip}(n-gram)}{\sum_{n-grams \in \hat{y}} Count(n-gram)} \tag{66}$$

Finally let's put this together to form the bleu score. Say you have p_n = bleu score on n-grams only where you have p_1, p_2, p_3 and p_4. The combined Bleu score is as follows:

$$BPexp(\frac{1}{4}\sum_{n=1}^{4} p_n) \tag{67}$$

where BP stand for the brevity penalty which considers and adjusts for the length of the vector where BP is defined as follows:

$$BP = \begin{cases} 1 \text{ if } MT\_output\_length > reference\_output\_length \\ exp(1 - MT\_output\_length/reference\_output\_length) \text{ otherwise} \end{cases} \tag{68}$$

5.5.7 Attention Model Intuition

The attention model is an modification of the encoder/decoder model making the algorithm work much better[27]. Given really long sentences the encoder reads in the sentence and memorizes its activations, after which it is feed to the decoder network which will generate the translation. What you see however is when sentences get longer (or are really short), the blue score decreases. To resolve this, the attention model looks at smaller parts of sentences at a time, resulting in an constant bleu score across sentence length. Say we have the usual example sentence "*Jane visite l'Afrique en septembre*" ($x^{<1>}, x^{<2>}, x^{<3>}, x^{<4>}, x^{<5>}$) feed into a bidirectional RNN (see Figure 43). Now what the attention model does is, when predicting a word, it computes a set of attention weights $\alpha^{<t,t'>}$ which represents how much attention you must pay for these words in order to predict the target word. In other words, by setting the weight to certain values, you create a context on which the prediction of the target word is based.

5.5.8 Attention Model

In the last section we have discussed the Attention model which allows a neural network to pay attention to only a part of an input sentence while it is generating a translation. In this section we will formalize this. As shown earlier in Figure

[26] Papineni *et al.*, 2002: A method for automatic evaluation of machine translation

[27] Bahdanau *et al.*, 2014: Neural machine translation by jointly learning to align and translate.

42 we have a bidirectional RNN which at each time step has a forward activation $\overrightarrow{a}^{<t>}$ and a backward activation $\overleftarrow{a}^{<t>}$, where $a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$. Next we have a forward only RNN which will have at each time step $S^{<t>}$ as input a context c and as output the predicted word $y^{<t>}$. The context c is computed based on the attention weights $\alpha^{<t,t'>}$. More formally:

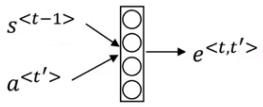
$$\sum_{t'} \alpha^{<1,t'>} = 1 \tag{69}$$

$$c^{<1>} = \sum_{<t'>} \alpha^{<1,t'>} a^{<t'>} \tag{70}$$

In other words, $\alpha^{<t,t'>}$ = amount of "attention" $y^{<t>}$ should pay to $a^{<t'>}$. Now how do we compute the attention weights? The formulas we will need are as follows:

$$\alpha^{<t,t'>} = \frac{exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} exp(e^{<t,t'>})} \tag{71}$$

which is basically a softmax function which will be 1 when al values are summed. In order to compute $e^{<t,t'>}$, we use $s^{<t-1>}$ and $a^{<t'>}$ (see figure to the right). However, we don't know the function. One thing you could do is just train a very small neural network and trust backpropagation as well as gradient descent to learn the right function. We've discussed the Attention model in the light of machine translation. However, this algorithm also works on other problems such as the image captioning problem[28].



5.6 Speech recognition - Audio data

5.6.1 Speech recognition

We will now discuss the speech recognition problem where you have an audio clip as input x and a transcript as output y. One characteristic of sound is that is has phonemes which was used in speech recognition in the early days ("quick" → "kwik, "the" → "de"). Nowadays however, an transcript can be directly derived from an audio clip. What made this possible is the size of the datasets currently used which are quite large (training duration can comprise 100.000 hours). One of the model that seems to work pretty well is the CTC cost for speech recognition which is an abbreviation of "Connectionist Temporal Classification"[29]. Say we have an audio clip saying "the quick brown fox" which is fed into a basic RNN where $T_x = T_y = 1000$. This is quite large, in fact, in speech recognition usually the number of input steps is much bigger than the number of time steps. For instance, the output could of the example sentence "the quick brown fox" could be as follows:

$$\text{ttt\_h\_eee\_\_\_space\_\_\_qqq\_\_}$$

which is considered a correct translation of the first part of the example sentence. The basic rule for the CTC cost function is to collapse repeated characters not separated by "blank" (i.e. underscore). This allows the neural network to have 1000 outputs and still ending up with a much shorter output text transcript.

5.6.2 Trigger Word Detection

Trigger word detection systems are applications of speech recognition and include systems that can be activated by your voice (e.g. Amazon Echo, Baidu DuerOS, Apple Siri, GoogleHome). Trigger words include the name of the systems to be activated, where if this is the case the target label will be 1. One slight disadvantage is that the training set will be unbalanced where you have many 0's and only a few 1's. One way to resolve this is by making a few outputs for a fixed time period equal to 1.

[28] Xu *et al.*, 2015: Show, attend and tell: Neural image caption generation with visual attention
[29] Graves *et al.*, 2006: Connectionist Temporal Classification: Labelling unsegmented data with recurrent neural networks

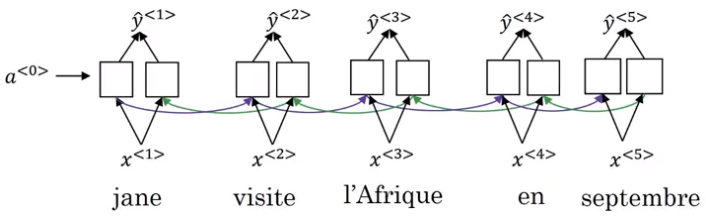


Figure 43: Bidirectional RNN.

6 Stand notation for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

6.1 Neural Networks Notation

General comments

- superscript (i) will denote the i^{th} training example while superscript $[i]$ will denote the i^{th} layer.

Sizes

- m : number of examples in the dataset
- n_x : input size
- n_y : output size (or number of classes)
- $n_h^{[l]}$: number of hidden units of the l^{th} layer.
  In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h^{[number\ of\ layers\ +1]}$.
- L : number of layers in the network

Objects

- $X \in \mathbb{R}^{n_x \times m}$ is the input matrix
- $x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th} example represented as a column vector
- $Y \in \mathbb{R}^{n_y \times m}$ is the label matrix
- $y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example
- $W^{[l]} \in \mathbb{R}^{\#\ units\ in\ next\ layer\ *\#\ units\ in\ the\ previous\ layer}$ is the weight matrix, superscript $[l]$ indicates the layer
- $b^{[l]} \in \mathbb{R}^{number\ of\ units\ in\ next\ layer}$ is the bias vector in the l^{th} layer
- $\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be noted as $a^{[L]}$ where L is the number of layers in the network

Common forward propagation equation examples

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function.

$\hat{y}^{(i)} = softmax(W_h h + b_2)$.

- General Activation Formula: $a_j^{[l]} = g^{[l]}(\Sigma_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} = g^{[l]}(z_j^{[l]})$.
- $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function

- $J_{CE}(\hat{y}, y) = -\Sigma_{i=0}^{m} y^{(i)} log \hat{y}^{(i)}$
- $J_1(\hat{y}, y) = \Sigma_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}|$

7 Deep Learning representations

For representation

- nodes represent inputs, activations or outputs
- edges represent weights or biases

Here are several examples of Standard deep learning representations:

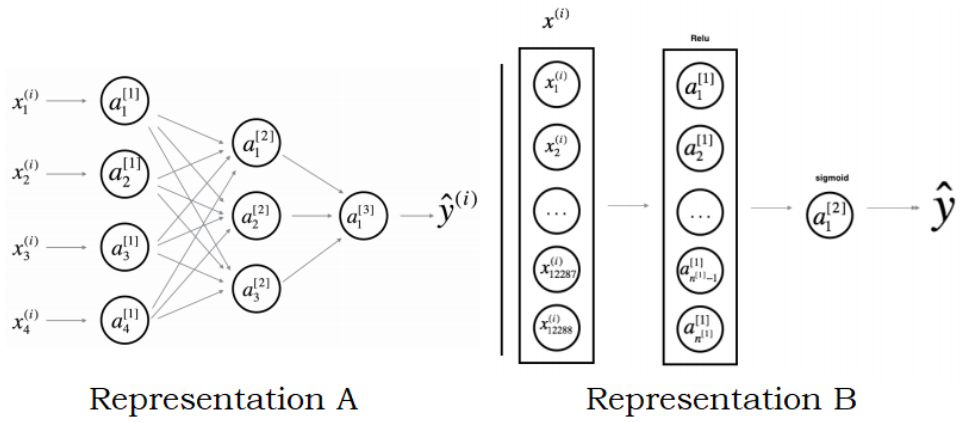


Figure 44: Examples of deep learning representations. (A) Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc ...) that should appear on the edges. (B) Simplified Network: a simpler representation of a two layer neural network, both are equivalent.

8 Python packages and functions

Packages

- *numpy* is a fundamental package for scientific computing with Python
- *h5py* is a common package to interact with a dataset that is stored on an H5 file
- *matplotlib* is a famous library to plot graphs in Python
- *Pil* and *scipy* are used here to test your model with your own picture at the end.
- *sklearn* provides simple and efficient tools for data mining and data analysis.
- *testCases* provides some test examples to assess the correctness of your functions.
- $planar - utils$ provide various useful functions.
- dnn_utils provides some necessary functions for this notebook
- *np.random.seed*(1) is used to keep all the random functions calls consistent. It will help us grade your work. Please don't change the seed.

Numpy package

- $np.exp(x)$ (works for any $np.array(x)$ and applies exponential function to every coordinate)
- *np.reshape*
- *np.multiply*
- $np.maximum$ / $np.minimum$

Other important notes

- Vectorization is very important in deep learning. It provides computational efficiency and clarity
- *image2vector* is widely used for computing row and column factors

Common steps for pre-processing a new dataset are:

- Figure out the dimensions and shapes of the problem (m_{train}, m_{test}, num_{px}, ...)
- Reshape the datasets such that each example is now a vector of size $(num_px * num_px * 3, 1)$
- "Standardize" the data

Index