Machine Learning Exam Sheet - Answers

Candidate: 1001689

April 17, 2016

Question 1

a) We have a binary classifier categorizing mail as *spam* and *non spam*, to be more specific we have a SVM that accomplishes with the separating hyperplane ($\mathbf{w}$, $\mathbf{w}_0$).

We denote as a *false positive* a *spam* datapoint misclassification and conversely as a *false negative* a *non spam* datapoint misclassification.

Our goal is to reduce the number of *false positives* and a colleague proposes a *bias-shifting* approach for accomplishing the goal, with no retraining of the SVM, that works this way:

- Pick a positive α s.t. $\alpha > 0$
- Update $\mathbf{w}'_0 = \mathbf{w}_0 - \alpha$

In practical terms this approach pushes the separating hyperplane in the direction of the *spam* datapoints.

The parameter α can be found via cross-validation such that minimizes the number of *false positives*.

We are asked to come up with two reasons for disagreeing with the proposal of the colleague.

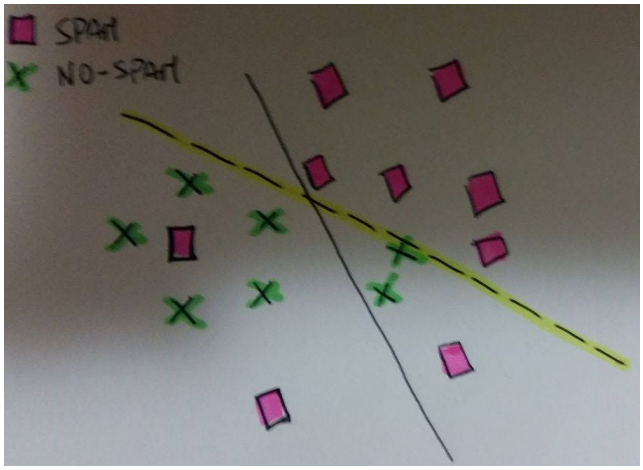
1) The *bias-shifting* approach might accomplish the goal of reducing the number of *false positives* but it does so at the expense of a (very) likely (and probably bigger) increase in the number of *false negatives*.
   We are making harder to classify *spam* as we raised the bar for doing that, meaning that many *spam* emails are being classified as *non-spam*. The reason lies in the change in the detection procedure for *spam* classification from:

$$x \cdot w \geq -w_0$$

   to:

$$x \cdot w \geq -w_0 + \alpha$$

2) The colleague limits our horizon for improvement to modifying the *bias* parameter, it could be argued that sometimes we could do a better job in reducing *false positives* if we could slightly rotate the separating hyperplane (by modifying the **w** slope parameter) rather than just doing a plain shifting of the separating hyperplane, also we could get better results in combining the two.



Imagine that the solid line represents the optimal hyperplane found, in this scenario it is easy to see that if we slightly rotate the hyperplane we can find another one minimizing the rate of *false positives* with a minor increase of the *false negatives* (an example is the highlited line).

P.s. The line, if we were to make a rigorous graphical interpretation, probably does not represent the most optimal hyperplane in that scenario but our goal was to make the point of the second reason we provided to disagree with the colleague.

b) We denote as a *false positive* a *non spam* datapoint misclassification, in this task we are concerned in reducing the number of *false positives*.

A colleague proposes to make 200 copies of every *spam* email so that the *enriched* training set has lot more *spam* emails than before.

Our first observation is that with some tweaks of the original SVM formulation we can reuse the old training set we can get the optimal separating hyperplane that we would obtain with the *enriched* training set. The benefit is that the size of the old training set is smaller than the *enriched* one and the complexity of the SVM solving algorihms depends also on the number of samples.

We proceed by recalling the original SVM formulation:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} & y_i(x_i w + w_0) \geq 1 - \xi_i \\ & \xi_i \geq 0 \qquad\qquad \textbf{for } i = 1, ..., m \end{array}$$

In our quest of a formulation for handling the *enriched* training set by reusing the old training set we also recall that *multiple copies of the same datapoint have the same distance to the margin* (ie. their *slack* variables are going to have the same value), our new formulation is going to exploit this observation.

We are making some changes in notation to make the formulation easier to understand:

1. We will denote with *spam* the indices of the *spam* datapoints.
2. We will denote with $no - spam$ the indices of the *non-spam* datapoints.

And here is the new formulation that we propose:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|w\|^2 + 200C \sum\limits_{spam} \xi_{spam} + C \sum\limits_{no-spam} \xi_{no-spam} \\ \text{subject to} & y_i(x_i w + w_0) \geq 1 - \xi_i \\ & \xi_i \geq 0 \qquad\qquad \textbf{for } i = 1, ..., m \end{array}$$

We can solve the above problem using a variety of optimization techniques, in our scenario we will recast the above problem (*primal*) to an equivalent *dual* problem that can be efficiently solved, the Lagrangian of the above is:

$$\begin{gathered} \mathcal{L}(w, w_0, \xi; \alpha, \beta) = \\ \frac{1}{2}\|w\|^2 + 200C \sum_{spam} \xi_{spam} + C \sum_{no-spam} \xi_{no-spam} - \sum_{i=1}^{m} \alpha_i [y_i(x_i w + w_0) - \\ 1 + \xi_i] - \sum_{i=1}^{m} \beta_i \xi_i \end{gathered}$$

We then proceed in taking out the derivatives of the Lagrangian w.r.t. to the parameters w, w_0 and ξ_i (we need to be careful in distinguishing the case of the i-th datapoint being *spam* or *non-spam*).

$$\begin{aligned} \frac{\partial w}{\partial \mathcal{L}} &= w - \sum_{i=1}^{m} \alpha_i y_i x_i \\ \frac{\partial w_0}{\partial \mathcal{L}} &= -\sum_{i=1}^{m} \alpha_i y_i \end{aligned}$$

$$\frac{\partial \xi_i}{\partial \mathcal{L}} = \begin{cases} C - \alpha_i - \beta_i & \text{if } i \text{ is } \textit{non-spam} \\ 200C - \alpha_i - \beta_i & \text{if } i \text{ is } \textit{spam} \end{cases}$$

Then we finally set the gradient of the Lagrangian to 0, thus obtaining:

$$w = \sum_{i=1}^{m} \alpha_i y_i x_i$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

$$\begin{array}{ll} \alpha_i + \beta_i = C & \text{if i is } \textit{non-spam} \\ \alpha_i + \beta_i = 200C & \text{if i is } \textit{spam} \end{array}$$

Then by substituting the w of the Lagrangian with $\sum_{i=1}^{m} \alpha_i y_i x_i$ and by exploiting the other equations we obtain the dual problem:

$$\begin{array}{lll} \text{maximize} & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y_i y_j x_i x_j - \sum_{i=1}^{m} \alpha_i & \\ \text{subject to} & \alpha_i + \beta_i = C & \text{if i is } \textit{non-spam} \\ & \alpha_i + \beta_i = 200C & \text{if i is } \textit{spam} \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 & \\ & \alpha_i, \beta_i \geq 0 & \end{array}$$

We can then translate the first two equality constraints in the following inequality constraints:

$$\begin{array}{c} \alpha_i + \beta_i = C \Longrightarrow 0 \leq \alpha_i \leq C \\ \alpha_i + \beta_i = 200C \Longrightarrow 0 \leq \alpha_i \leq 200C \end{array}$$

Coming back to the question, we disagree with the colleague as making 200 copies of each *spam* email could potentially mean that our *enriched* trainset will have *way more spam* emails compared to the *non-spam* ones: in practial terms it means that the *spam* support vectors will be more penalized (in fact they will be 200 times penalized).

The outcome is that the separating hyperplane will be directed towards the side where the *non-spam* datapoints are lying in thus increasing the number of *false positives* and making us not fulfilling our initial goal.

In lieu of the fallacy of the approach proposed by the colleague we propose instead the opposite approach: penalize the *non-spam* datapoints by making 200 copies of each one of them, this approach effectively accomplishes the goal as the separating hyperplane will be moved towards the *spam* datapoints thus reducing the number of *false positives*.

We can also modify our SVM formulation in above to take in account the *enriched* dataset containing 200 copies of every *non-spam* email.

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|w\|^2 + C \sum\limits_{spam} \xi_{spam} + 200C \sum\limits_{no-spam} \xi_{no-spam} & \\ \text{subject to} & y_i(x_i w + w_0) \geq 1 - \xi_i & \\ & \xi_i \geq 0 & \textbf{for } i = 1, ..., m \end{array}$$

c) The task is still concerned with *false positives*, more specifically we have to come up with a way to get 0 *false positives* on the training set while still doing our best for minimizing the *false negatives*.

We are going to provide some intuition regarding the *slack variables* in the SVM Soft-Margin formulation that will be exploited later for an alternative formulation for SVM accomplishing the above task.

Datasets, in the real world, often are not *linearly separable* meaning that it is not possible to come up with a separating hyperplane that perfectly separates the two classes of inputs.

When we are faced with a *not linearly separable* dataset the best we can ask for is to minimize the distance of the points to their respective goal boundary. *Slack variables* (denoted with ξ) model this notion of distance to the goal boundary of each point.

This is the classic SVM Soft-Margin formulation.

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i & \\ \text{subject to} & y_i(x_i w + w_0) \geq 1 - \xi_i & \\ & \xi_i \geq 0 & \textbf{for } i = 1, ..., m \end{array}$$

The key idea is that each point pays a price (determined by **C**) depending on how further is from the goal boundary, our goal is to minimize the overall price.

The two observations in below apply:

1. A point is classified correctly if its slack variable is in the range $\in [0, 1]$
2. A point is misclassified if its slack variable has a value > 1

From the first observation we can obtain the following statement:

> Obtaining 0 *false positives* on the training set $\Longrightarrow$ Every *non-spam* datapoint's *slack variable* has value $\in [0, 1]$

From the second observation it follows that:

> Minimizing *false negatives* $\Longrightarrow$ Squeeze as much as you can the *slack variables* of the *spam* datapoints by penalizing them in the objective.

Here is our final SVM formulation incorporating the observations in above ($no - spam$ denotes the indices of the *non-spam* datapoints and *spam*).

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|w\|^2 + C\sum\limits_{spam}\xi_{spam} + \sum\limits_{no-spam}\xi_{no-spam} & \\ \text{subject to} & y_i(x_i w + w_0) \geq 1 - \xi_i & \textbf{for } i = 1, ..., m \\ & \xi_i \geq 0 & \textbf{for } i = 1, ..., m \\ & \xi_{no-spam} \leq 1 & \end{array}$$

A minor observation, we shall note that a *non-spam* datapoint having its *slack variable* $= 1$ is going to be classified as *spam* as the classifier decision procedure is going to yield 0, to address this we suggest to modify the classifier decision procedure by changing the $\geq$ sign to $>$.

Another approach we could use for the above problem is setting the maximum allowed distance of a *non-spam* datapoint to the boundary a number slightly below 1 (ie. $1 - 10^{-3}$) and this would not require us to modify the classifier decision procedure.

Question 2

a) Our primary task is to find out the likelihood of observing the data $\langle((a : b, c), y_i)\rangle_{i=1}^{m}$ given the parameters p_{bc}^{a}.

The quest for a good notation has proved to be a bit challenging so we felt necessary to provide some remarks before providing you the likelihood:

1. p is the set of probabilities of all the possible triplets $(a : b, c)$ in the observed data with the given constraints
2. The p_{bc}^{a} inside the body of the product is the probability of the i-th triplet $(a : b, c)$ with $b \prec c$

The likelihood is then given by:

$$\mathcal{L}(\langle((a : b, c), y_i)\rangle_{i=1}^{m}|p) = \prod_{i=1}^{m}(p_{bc}^{a})^{y_i}(1 - p_{bc}^{a})^{1-y_i}$$

Our next task is concerned with the estimation of the number of free parameters in the model given n fruits.

We are going to exploit two observations:

1. The elements a, b and c in the triplet $(a : b, c)$ are *distinct*.
2. $p_{cb}^{a} = 1 - p_{bc}^{a}$, meaning that we just need to find a way to regard cb and bc as the same.

The idea is that for each item we count the number of possible 2-combinations you can build with the remaining items, thus we get that the number of *free* parameters is:

$$n\binom{n-1}{2}$$

b) Let's start by recalling the likelihood formula of the previous question:

$$\mathcal{L}(\langle((a : b, c), y_i)\rangle_{i=1}^{m}|p) = \prod_{i=1}^{m}(p_{bc}^{a})^{y_i}(1 - p_{bc}^{a})^{1-y_i}$$

Again p^a_{bc} refers to all the possible pairs of fruits

In this setting we have p^a_{bc} being the SoftMax function with $x = 1$:

$$p^a_{bc} = \frac{exp(M_{ab})}{exp(M_{ab}) + exp(M_{ac})}$$

Let's plug the above in the previously obtained likelihood function:

$$\mathcal{L}(\langle((a:b,c),y_i)\rangle_{i=1}^m|M) = \prod_{i=1}^m \left(\frac{exp(M_{ab})}{exp(M_{ab}) + exp(M_{ac})}\right)^{y_i} \left(1 - \frac{exp(M_{ab})}{exp(M_{ab}) + exp(M_{ac})}\right)^{1-y_i}$$

Let's now take the negative log likelihood instead, minimizing the negative log likelihood is equivalent as to maximizing the likelihood.

$$NLL(\langle((a:b,c),y_i)\rangle_{i=1}^m|M) =$$

$$= -\sum_{i=1}^m y_i log\left(\frac{exp(M_{ab})}{exp(M_{ab}) + exp(M_{ac})}\right) + (1-y_i) log\left(\frac{exp(M_{ac})}{exp(M_{ab}) + exp(M_{ac})}\right)$$

$$= -\sum_{i=1}^m y_i(M_{ab} - log(exp(M_{ab}) + exp(M_{ac}))) + (1-y_i)(M_{ac} - log(exp(M_{ab}) + exp(M_{ac})))$$

$$= -\sum_{i=1}^m y_i M_{ab} + M_{ac} - log(exp(M_{ab}) + exp(M_{ac})) - y_i M_{ac}$$

$$= \sum_{i=1}^m -y_i M_{ab} - M_{ac} + log(exp(M_{ab}) + exp(M_{ac})) + y_i M_{ac}$$

The above negative-log-likelihood is also dubbed as SoftMax logistic regression variant which is convex [1], we can then proceed in finding the optimal parameters M_{ab} for every pair of items, thus we have n^2 parameters.

The NLL of the logistic regression is also known as *cross-entropy*[2].

M_{ab} refers to every possible pair of the items, this means that there could be some entries in the sum where M_{ab} is not present implying that the derivative w.r.t. M_{ab} in these entries is 0, for this reason we felt more cautious taking

[1] Rennie, Jason DM. "Regularized logistic regression is strictly convex." Unpublished manuscript. URL people.csail.mit.edu/jrennie/writing/convexLR.pdf (2005).

[2] Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012. Ch. 2.8.2

the derivative w.r.t. M_{ab} and M_{ac} for a single sample containing one of these two parameters:

$$\nabla^i_{M_{ab}} = -y_i + \frac{exp(M_{ab})}{exp(M_{ab}) + exp(M_{ac})}$$

$$\nabla^i_{M_{ac}} = -1 + \frac{exp(M_{ac})}{exp(M_{ab}) + exp(M_{ac})} + y_i$$

Again, if a parameter M_{ab} is not present in a given entry of the sum the derivative w.r.t. M_{ab} is 0 unless M_{ba} is present.

We suggest to reinforce a lexicographic order to rewrite every instance of M_{ba} into M_{ab} to make the matter easier for the optimization algorithm. Thus we can say that the total number of parameters is not the number of every pair of the fruits (n^2) but instead is $\binom{n}{2}$.

Thus we can simply say that:

$$\nabla_{M_{ab}}(NLL) = \sum_{i=1}^{m} \nabla^i_{M_{ab}}$$

We can then find the optimal parameters M_{ab} with the Steepest Descent algorithm [3] that iteratively updates the weights for each sample in the training set (η is the learning-rate parameter, we need some care in choosing a good step-size):

$$M'_{ab} = M_{ab} - \eta \nabla^i_{M_{ab}}$$

c) This question is about *clustering* fruits using the square similarity matrix M of the previous exercise.

Let's describe some features of the M matrix:

(a) It is a square matrix of size $N \times N$ with N being the number of fruits.

(b) Given a, b being two fruits and $a \prec b$ the entry in the a-th row and b-th column $M_{a,b}$ denotes its similarity measure.

(c) It is symmetric so we have $M_{a,b} = M_{b,a}$.

(d) The diagonal of the matrix is 1 meaning that comparing every fruit with itself will yield score 1.

(e) *Assumption.* As we use the matrix M for denoting similarity between a pair of fruits we assume that all the entries are positive so every entry of the matrix is $\in [0, 1]$.

[3] Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012. Ch. 8.3.2

There are many ways to use the above matrix for clustering purposes.

We could proceed by starting creating a distance function between two pair of fruits a and b:

$$d(a, b) = 1 - M_{a,b}$$

The idea is: Higher similarity score $\implies$ Shorter distance.

The above distance function rules out the use of the popular k-Means clustering algorithm because it is not designed for clustering *structured objects*, we can though use a variant of the k-Means algorithm with the clusters fixed a-priori called k-Medoids [4].

We now describe very briefly the k-Medoids algorithm:

1. Pick a number k representing the number of desired clusters.
2. From the training set pick k random samples, these samples will be the centroid of the newly initialised clusters.
3. For each sample:
    1. Find the centroid that best minimizes the distance function created in above. We denote with C the cluster in which it has been assigned.
    2. Update the centroid of the cluster C by choosing a sample in the cluster C that best minimizes the sum of the distance with all the other samples belonging to the same cluster C.

We note that similarity matrix can be used by techniques falling under the umbrella of *spectral clustering*, the word *spectral* is used as it exploits insights about *eigenvalues* and *eigenvectors*, *spectral clustering* is a very rich family of algorithms that really would deserve a lengthy explanation but for the scope of the question we will limit ourselves in describing a way to use the matrix M using some insights about the *Graph Laplacian* matrix which is one of the most important tools for *spectral clustering*.

Our aim is just to give a practical approach so we will be reusing some known results without giving mathematical proofs that are beyond the scope of the question[5].

Let $G = (V, E)$ be an undirected graph.

Let W be an adjacency matrix where:

(a) if there exists an edge connecting vertices V_i and $V_j \implies W_{ij} \geq 0$

(b) if there exists no edge connecting vertices V_i and $V_j \implies W_{ij} = 0$

[4] Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012. Ch. 14.4.2

[5] Von Luxburg, Ulrike. "A tutorial on spectral clustering." Statistics and computing 17.4 (2007): 395-416.

Let D be a diagonal matrix in which every entry represents the degree of a given vertex:

$$D_i = \sum_{j=1}^{n} w_{ij}$$

We denote with L the *Graph Laplacian* matrix:

$$L = D - W$$

It turns out that this matrix has special properties and we are most interested in the following result:

L matrix has the eigenvalue 0 has multiplicity K equivalent to the number of connected component of the graph [6].

Each corresponding eigenvector of the eigenvalue 0 tells us which vertices belong to the given component, in fact the eigenvector is a vector with some zero entries as the vertices not belonging to the given connected component with the other entries being 1.

Let's denote with x the matrix with each column being an eigenvector of 0, this is going to be a $N \times K$ matrix.

We can think of the i-th row of the X matrix as being coordinates of the i-th vertex, in a nutshell we built a mapping from G to $\mathbb{R}^{n \times k}$.

We can then use a classical clustering algorithm like k-Mean with the X matrix representing the coordinates of every vertex.

Now you will argue but how can we really use the similarity matrix M, you talked about graphs but in the truth you did not give us a practical way to create the graph from the similarity matrix M.

The truth is there is no unique approach to do that, you can unleash you creativity there, in literature some of the suggested approach are:

1. k-NN graph: simply create an edge between a given vertex and k most similar ones, in this case the matrix M will give us the similarity score between two points.
2. ϵ-NN graph: Fix an ϵ and create and edge between every two pairs having similarity score higher to ϵ.

[6] Von Luxburg, Ulrike. "A tutorial on spectral clustering." Statistics and computing 17.4 (2007): 395-416.

Question 3

a) In the last years many breakthroughs have been achieved that led to a near-human (or even better) perfomance in the MNIST, CIFAR and ImageNet datasets.

The common denominator behind these breakthroughs has been an always-increasing *depth* of the convolutional neural networks, in *academia* the on-going trend is about *deep networks*.

While the *depth* of a network plays a crucial role there have been problems surrounding *deep networks*, like the *vanishing gradient* problem, the authors of the paper pinpoint another problem: *degradation*.

Degradation is a surprising phenomena affecting *deep networks*, namely as we increase the number of layers of the network the *training accuracy* starts to stagnate and then degrades in a turbolent way, the authors note that surprisingly *degradation* is not caused by *overfitting*.

We are faced with the question if we would expect a similar *degradation* problem to happen when performing linear regression with polynomial basis expansion as we increase the degree of the *polynomial basis*, more specifically the question is: *does increasing the degree of the polynomial basis eventually results in a worse training error*?

The short answer is *no* because increasing the degree of the *polynomial basis* makes easier to fit the model to the data incurring in *overfitting*, so we are going to have an always increasing *accuracy* of the model.

Now the point is *how does the degradation problem happen* in the setting described in the paper?

It turns out that it has something to do with the optimizers having a hard time with the *deeper networks* hinting that every network is a different beast and some of them are inherently hard to optimize.

A possible explaination is the following: it is harder to optimize a network having multiple (and perhaps non-linear) transformations of the input.

Let's imagine a neural network trained to accomplish a task and another one with some layers added on the top of the first, mentally it is easy to see that there exists a solution for the latter to accomplish the same *training error*: the added layers simply copy the input (*identity mapping*).

The goal of the authors of the paper has been in making *deep networks* easier to optimize exploiting the intuition about *identity mappings* described in above.

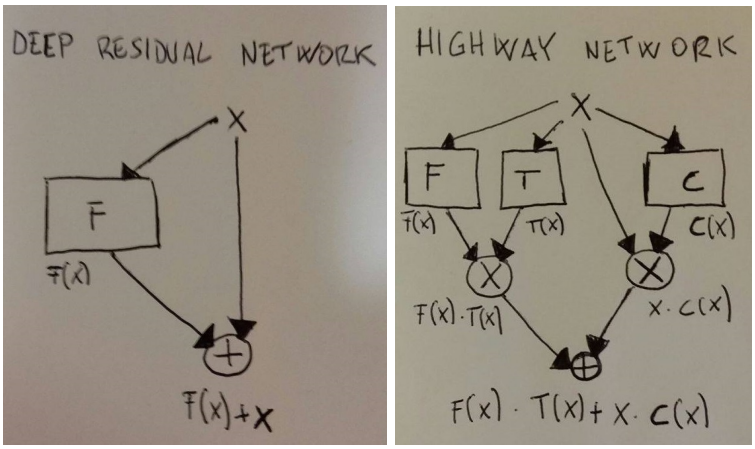
b) We denote with $F(x)$ some transformation of the input, $F(x)$ can be even a network spanning multiple layers, w will denote the weights of the unit.

A *highway network* computes, with T being the *Transform Gate* function and C the *Carry Gate* function, the gating functions give direction about how the input is transformed:

$$y = F_w(x) \cdot T_w(x) + x \cdot C_w(x)$$

A *residual network* computes the residual of a function being:

$$y = F_w(x) + x$$



From the formula and their topology depicted in above we can evince that a *residual network* is basically a *highway network* with simply *no gates functions* or if we want to be more precise with $T(x) = 1$ and $C(x) = 1$, thus *residual networks are a subcase of highway networks*, we can also equivalently say that *residual networks* are *highway networks* with a parameter-less and never-closing *carry gate*.

These two networks share many similarities that we are going to discuss.

First, both the networks ease the life of optimizers even if they are *very deep*, we can say that in a sense both of them enable the training of *very deep networks*.

Second, given an already trained network there exists a *deeper counterpart solution with both residual network and highway network* with the same *training accuracy*, the added layers should be set up in a way that they merely copy the input.

As a consequence both of them make easier approximating *identity mappings*: just drive to 0 the weight of $F(x)$ for the *residual network* and in the case of *highway networks* just close the *Transform gate* and let the *Carry gate* $= 1$ ($T(x) = 0$, $C(x) = 1$).

Third, both of them use *shortcut connections*, they help the information vehiculate in a preserving-way across many layers.

c) *Lesioning* is a technique that allows us to inspect the contribute of each layer in computing the output by forcing a single layer at time to simply copy the input.

Let's recall what *highway networks* compute:

$$y = F_w(x) \cdot T_w(x) + x \cdot C_w(x)$$

In the setting of the second paper for simplicity the authors set:

$$C_w(x) = (1 - T_w(x))$$

Thus making *highway network* compute:

$$y = F_w(x) \cdot T_w(x) + x \cdot (1 - T_w(x))$$

The idea is that T controls the routing of the information, let's say we want to close the *transform gate* we just let $T_w(x) = 0$ and the layer will be simply copying the input, thus we effectively closed the *transform gate*.

The authors of the paper suggests to bias at the beginning the network towards *carry behaviour* (by setting a negative weights on T in the above formula) to make easier inspecting whether the training changes that by making the *transform gate* doing meaningful work with the input rather than just merely copying that, in general we aim for every layer in the network contributing to the result, we do not want idle layers.

The amazing idea behind *lesioning* is that by letting a single layer not doing any transformation to the input we can evaluate its importance, ie. if we should find out that by silencing a layer the error rate does not get worse we can follow that we can probably achieve the same task with a network having a lower *depth*, viceversa if the error rate gets dramatically worse we can infer that the layer effectively contributes to the result.

We can depict the *lesioning* technique as a *layer-level debugging* facility for neural networks, this kind of *debugging* is usually not that simple with *plain networks*.