אוניברסיטת בן-גוריון בנגב

המחלקה להנדסת מערכות תוכנה ומידע

372-2-5214: יישום אלגוריתמים לומדים – תשע"ח, סמסטר א'

צוות הקורס: פרופ' ליאור רוקח, אריאל בר

<u>פרוייקט מסכם</u>

AdaNet: Adaptive Structural Learning of Artificial Neural Networks

מגיש:

דוד אבקסיס 028830099

<u>Reference</u>: Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M. & Yang, S.. (2017). AdaNet: Adaptive Structural Learning of Artificial Neural Networks. Proceedings of the 34th International Conference on Machine Learning, in PMLR 70:874-883

<u>Abstract and motivation for the algorithm</u>:

"We present a new framework for analyzing and learning artificial neural networks. Our approach simultaneously and adaptively learns both the structure of the network as well as its weights. The methodology is based upon and accompanied by strong data-dependent theoretical learning guarantees, so that the final network architecture provably adapts to the complexity of any given problem."

Contents

Preface

This project is aimed to implement an API of AdaNet, using algorithm based on the article: http://proceedings.mlr.press/v70/cortes17a.html

In short this model is building from scratch a neural network according to the data complexity it fits,

This is why it named as adaptive model. For this implementation the problem at hand is always a binary classification.

During fit operation it will build the hidden layers and number of neurons in each layer .

The decision if to go deeper (add hidden layer) or to go wider (add neuron to existing layer),

Or update an existing neuron weight is done in a closed form of calculations

(By using Banach space duality) shown in the article.

Lastly it will optimize the weight of the best neuron (added or existing), update parameters and iterate.

The article talks about several variants of AdaNet, this is the AdaNet.CVX implementation,

Explained on Appendix C - that solves a convex sub problem in each step in a closed form.

Further detailed explanations of this variant is shown in a previous version of the article [v.1]:

All versions: https://arxiv.org/abs/1607.01097

v.1: https://arxiv.org/abs/1607.01097v1

I have used as a reference a MATLAB implementation of the algorithm from: https://github.com/lw394/adanet

This reference included comments from the article author Xavier Gonzalvo, about calculating hyper parameters to improve the model (Ck – was allowed to calculate its value and not set it before as fixed value,
and Λk – was calculated over the layers instead of being fixed hyper parameter) that didn't worked, so I left it out.

Giving back to the community I've shared this project to: https://github.com/davidabek1/adanet

Pseudo code

Because the model is very complex on the details of calculations,
I've decided to leave the pseudo code as close as possible to the original article in favor of its simplicity to understand in high level how the model is working. There are some changes made to emphasis my implementation.

AdaNet.CVX – Building (adapting) the neural network

Input:

$Data = (< x_1, y_1 >, \ldots, < x_m, y_m >)$ - represent the training set which contains m instances, $y_i \in \{-1,1\}$ as binary labels.

l - Max number of allowed hidden layers for the model to build

k – An iterator over number of layers

n_k – Max number of allowed nodes on each hidden layer

Λ_k – Max weight Norms, a hyperparameter of the ℓ_p norms of the weights defining new nodes. In the implemented model each layer has the same value.

$\Gamma_k = \lambda r_k + \beta$ – Complexity penalty as part of the regularization term of the objective loss function on each layer, admits hyperparameters $\lambda \geq 0$ and $\beta \geq 0$

$C_k > 0$ – Max weight margin, a hyperparameter on each layer that is a bound for the hypothesis of each layer.

T – number of epochs (iterations of the model until convergence)

ε – change in objective loss function

```
1. $(w_0, k, j)_{k\in[1,l], j\in[1,n_k]}, D_1, \hat{l}, (\hat{n}_k)_{k\in[1,l], j\in[1,\hat{n}_k]} \leftarrow INIT\left(m, l, (n_k)_{k=1}^{l}\right)$
2. for t ← 1 to $T$
3.     $\left(d_{k,j}\right)_{k\in[1,\hat{l}], j\in[1,\hat{n}_k]} \leftarrow \text{EXISTINGNODES}\left(D_t, (\boldsymbol{h}_k)_{k=1}^{\hat{l}}, (\hat{n}_k, C_k, \Gamma_k)_{k=1}^{l}\right)$
4.     $\left(\tilde{d}_k, \tilde{u}_k\right)_{k=1}^{min(\hat{l}+1,l)} \leftarrow \text{NEWNODES}\left(D_t, (\boldsymbol{h}_k)_{k=1}^{\hat{l}}, (n_k, \hat{n}_k, C_k, \Lambda_k, \Gamma_k)_{k=1}^{l}\right)$
5.     $\left((k,j), \epsilon_t, \hat{l}, (\hat{n}_k)_{k=1}^{l}\right) \leftarrow \text{BESTNODE}\left(\left(d_{k,j}\right)_{k\in[1,\hat{l}], j\in[1,\hat{n}_k]}, \left(\tilde{d}_k\right)_{k\in[1,min(\hat{l}+1,l)]}\right)$
6.     $\boldsymbol{w}_t \leftarrow \text{APPLYSTEP}((k,j), w_{t-1})$
7.     $(D_{t+1}, S_{t+1}) \leftarrow \text{UPDATEDISTRIBUTION}\left(\boldsymbol{w}_t, S, \left(h_{k,j}\right)_{k\in[1,\hat{l}], j\in[1,\hat{n}_k]}\right)$
8.     if $F(w_t) \geq F(w_{t-1}) - \varepsilon$ then
9.         break
```

AdaNet – Classify an instance

Input: $X = (< x_1 >, \ldots, < x_m >)$ – an instance needs to be labeled (with m features)

```
1. $f \leftarrow \sum_{k=0}^{\hat{l}} \sum_{j=1}^{\hat{n}_k} w_{T,k,j}\, h_{k,j}$
```

Alogorithm explanation

<u>For the building of the network (fitting operation)</u>, as Input we've taken sample Data (and not S as this notation will be used later), labeled as binary +1 and -1, and all the described hyperparameters and parameters of the model.

<u>**Line 1**</u> - we initialize the parameters of the model, assigning zero to weights **w** of the future created nodes in the network, and the distribution D1 to be uniform over the data $(\frac{1}{m})$.
the network is starting from input nodes (the features) and one output node (the positive label we will predict)

<u>**Line 2**</u> – start T iterations (or stop by convergence) to adapt the network following sequence of steps described next

<u>**Line 3**</u> – compute the score of updating the weights for existing nodes,
this is done by the ExistingNodes method, in the first epoch the network is empty (only input and output nodes), so this method will not be executed at first epoch.
while in it we will start by scoring all nodes to zero.
calculate the loss for each node, to be used in the next calculations (k for existing layers, j for existing nodes):

article notation-> $\epsilon_{t,k,j} \leftarrow \frac{C_k}{2}\left[1 - \mathbb{E}_{i\sim D_t}\left[\frac{y_i h_{k,j}(x_i)}{C_k}\right]\right]$, math notation -> $\epsilon_{t,k,j} \leftarrow \frac{C_k}{2}\left[1 - \frac{1}{C_k}\left[\left(\overline{y_\iota \mathcal{H}_k^{(p)}}\right)^T D_t\right]\right]$

for current non zero weights calculate the score as:

$$d_{k,j} \leftarrow \left(\epsilon_{t,k,j} - \frac{1}{2}C_k\right) + sgn(w_{t-1,k,j})\frac{\Gamma_k m}{2S_t}$$

for other nodes that **are** zero weight and **not** following the next condition: $\left(\left|\epsilon_{t,k,j} - \frac{1}{2}C_k\right| \leq \frac{\Gamma_k m}{2S_t}\right)$
calculate the score: $d_{k,j} \leftarrow \left(\epsilon_{t,k,j} - \frac{1}{2}C_k\right) - sgn\left(\epsilon_{t,k,j} - \frac{1}{2}C_k\right)\frac{\Gamma_k m}{2S_t}$

ExistingNodes method will return the scores of all current nodes in $(d_{k,j})_{k\in[1,\hat{l}],j\in[1,\hat{n}_k]}$

<u>**Line 4**</u> – compute the score of adding a potential new node to each of the existing layers and also a score for starting a new layer, and putting a node on it. This is done by using "l_p duality to derive a closed-form expression of the node with largest weighted error at each layer".
calculations are done within the NewNodes method.
The article is starting from layer 2, while in the implementation I start from layer 1, handling this condition leading to $\boldsymbol{h}_{k-1}$ as $\boldsymbol{h}_0$ to be the input data X with no activation function.
the hypothesis values of the nodes on layer k at norm p space $\mathcal{H}_k^{(p)}$ is calculated as:

$$\mathcal{H}_1^{(p)} = \{x \mapsto \boldsymbol{u}\cdot\boldsymbol{h}_0(x): \boldsymbol{u}\in\mathbb{R}^{n_0}, \|\boldsymbol{u}\|_p \leq \Lambda_1\}$$

$$\mathcal{H}_k^{(p)} = \left\{x \mapsto \left(\sum_{\boldsymbol{j=1}}^{\boldsymbol{n_{k-1}}} u_j\left(\varphi_{k-1}\circ h_j\right)(x)\right): \boldsymbol{u}\in\mathbb{R}^{n_{k-1}}, \|\boldsymbol{u}\|_p \leq \Lambda_k, h_j \in \mathcal{H}_{k-1}^{(p)}\right\}, (\forall k > 1)$$

where $\Lambda_k > 0$ is a hyperparameter and φ_k is an activation function (as ReLU, sigmoid…)

first for not reached maximum nodes on layer and a new node on a new (allowed) layer calculate the loss:

article notation-> $\tilde{\epsilon}_k \leftarrow \frac{C_k}{2}\left(1 - \frac{\Lambda_k}{C_k}\|Margin(D_t, \boldsymbol{h}_{k-1})\|_q\right)$, math notation->$\tilde{\epsilon}_k \leftarrow \frac{C_k}{2}\left(1 \frac{\Lambda_k}{C_k}\left\|\left(\overline{y_\iota \mathcal{H}_{k-1}^{(p)}}\right)^T \cdot D_t\right\|_q\right)$

And calculate the weights coming into the new nodes:

$$\tilde{u}_k \leftarrow \frac{\Lambda_k |Margin(D_t,h_{k-1,j})|^{q-1} sgn\left(Margin(D_t,h_{k-1,j})\right)}{\|Margin(D_t,\boldsymbol{h}_{k-1})\|_q^{\frac{q}{p}}}$$

New nodes that **not** following next condition: $\left(\left|\tilde{\epsilon}_t - \frac{1}{2}C_k\right| \leq \frac{\Gamma_k m}{2S_t}\right)$

Calculate the score: $\tilde{d}_k \leftarrow \left(\tilde{\epsilon}_k - \frac{1}{2}C_k\right) - sgn\left(\tilde{\epsilon}_k - \frac{1}{2}C_k\right)\frac{\Gamma_k m}{2S_t}$

NewNodes method will return the scores of all new potential nodes $\left(\tilde{d}_k, \tilde{u}_k\right)_{k=1}^{min(\hat{l}+1,l)}$

<u>**Line 5**</u> – select the highest score calculated in previous methods and update model parameters based on it.
This is done in the BestNode method where first we compare the scores coming from ExistingNodes, and NewNodes methods, and select the **node** with highest score as the one that has the largest contribution toward decreasing the objective loss function.
we keep the location of this best node, and update model parameters if changed:
for selecting a new node -
* then to the number of nodes on its layer $\hat{n}_k$ will be added one,
* likewise if the node is on a new layer then to the number of layers $\hat{l}$ will be added one.
* calculate the new $\mathcal{H}_k^{(p)}$ based on resulted $\tilde{u}_k$ from NewNodes method
* update the new ϵ_t based on resulted $\tilde{\epsilon}_k$ from NewNodes method

BestNode method will return $\left((k,j), \epsilon_t, \hat{l}, (\hat{n}_k)_{k=1}^{l}\right)$

<u>**Line 6**</u> – will find the step required to update the weight of the best node selected in previous BestNode method.
This is done in the ApplyStep method that will apply the optimal step size by computing it via line search.
an optimizer will minimize the next loss function and find the step required to update the weight of the best selected node:

$$F(w_{t-1} + \eta e_k) = \frac{1}{m}\sum_{i=1}^{m}\Phi\left(1 - y_i f_{t-1}(x_i) - \eta y_i h_k(x_i)\right) + \sum_{j\neq k}\Gamma_j\left|w_{t-1,j}\right| + \Gamma_k\left|w_{t-1,k} + \eta\right|$$

Where $\Phi(-x)$ is a non-increasing convex function like exponential or logistic,
and η is the step size to minimize the loss function
(e_k is the k th unit vector in $\mathbb{R}^n$, $k \in [1, N]$, N is the total nodes in our structured network),
and $f_{t-1} = \sum_{j=1}^{N} w_{t-1,j} h_j$

ApplyStep method will return w_t

<u>**Line 7**</u> – updating the distribution before the next iteration.
This is done in the UpdateDistribution method that will calculate it as the normalized gradient of the objective loss function (without the regularization term).

$$D_t(i) = \frac{\Phi'\left(1 - y_i f_{t-1}(x_i)\right)}{S_t}$$

where S_t is the normalization factor, $S_t = \sum_{i=1}^{m}\Phi'\left(1 - y_i f_{t-1}(x_i)\right)$

UpdateDistribution method will return updated D and S as (D_{t+1}, S_{t+1})

<u>**Line 8**</u> – check for convergence threshold ε, if the updated loss function is less than a threshold, stop the iteration (<u>**line 9**</u>).
in the implementation based on observations I have checked for the average last x epochs against the threshold ε.

<u>**For the classification of an instance (predict operation)**</u>, we are doing a standard feed forward through the built network, to find the output prediction f, that if it is more than zero it will predict the positive class (+1), and if less than the negative class (-1).
<u>**Line 1**</u> - shows that we sum the multiplication of the weights of each node by its hypothesis, while the hypothesis shown above (line 4) is calculated by the weights leading to the node by its previous h activated.

Illustration

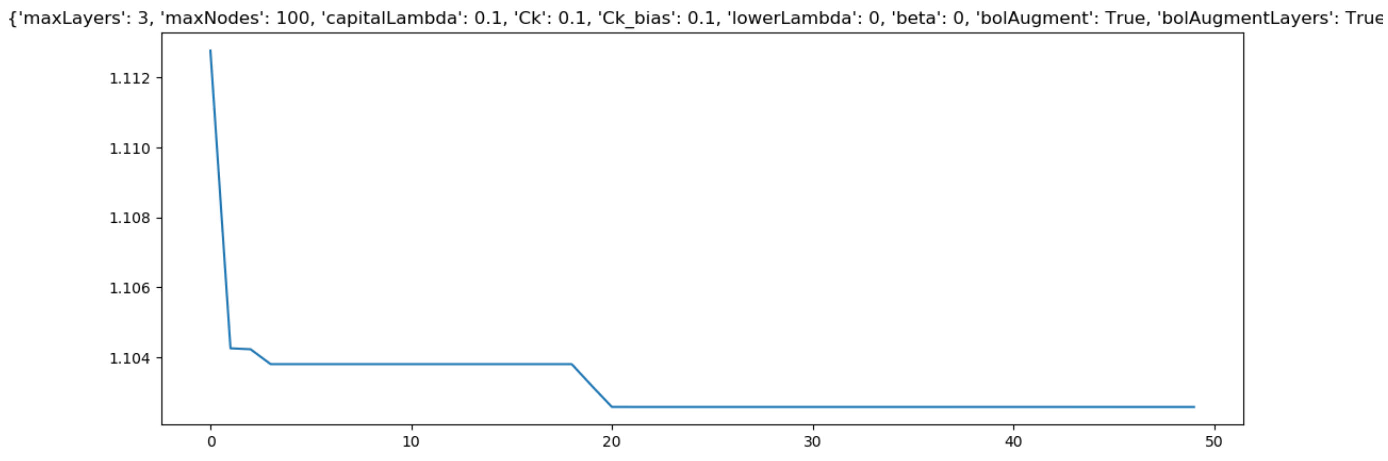
Since the model is highly complex, with a lot of steps, I have focused on understanding the many available parameters and hyperparameters of the model, that will be shown in this section.

Most of the experiments done with a toy dataset of classifying two spirals, the python code to produce this toy dataset is part of the submitted code in twospirals.py.
Added experiments done with the German-IDA dataset.

Configuration parameters of the model, were max layers of 3, and max nodes of 100 on each layer,
50 epochs, all the rest were the parameters to tune the model, an example lines from the results.csv (submitted as part of the project files) shows the experiments data collected:
Ck and Ck_bias are 0.1, augment layers and augment are alternating TRUE and FALSE,
capital lambda set to 0.1, beta set to 0, lower lambda is 0 or 10e-4.
the results are shown as, mean prediction of test dataset ≈ 0.659, fit time in seconds, objective loss function start value ≈1.1, and finish value after epochs ≈ 1.09, the rate of first 3 epochs (average of the change in loss) ≈ 0.004, the rate of the last 3 epochs (this gives information if the model can continue to converge or, no change seen for several epochs). Last 2 columns shows what kind of network was built, here we can see 2 hidden layers with about 9 nodes on first and 1 node on second (for this example no third layer).

numN	numL	lowerLambda	lossLastChange	lossLast	lossFirstChange	lossFirst	capitalLambda	bolAugmentLayers	bolAugment	beta	adanet_mean	adanet_fit_time	T	Ck_bias	Ck
[[9] [1] [0]]	2	0	-1.29E-08	[1.09531534]	-0.004913846	[1.10635802]	0.1	TRUE	TRUE	0	0.659863946	8.720475912	50	0.1	0.1
[[9] [1] [0]]	2	0	0	[1.09510968]	-0.004139051	[1.10480006]	0.1	FALSE	TRUE	0	0.659863946	12.20190287	50	0.1	0.1
[[8] [1] [0]]	2	0	0	[1.09561652]	-0.003986942	[1.10460798]	0.1	TRUE	FALSE	0	0.659863946	13.86191583	50	0.1	0.1
[[9] [1] [0]]	2	0	0	[1.09494415]	-0.004263989	[1.10504186]	0.1	FALSE	FALSE	0	0.659637188	10.37349248	50	0.1	0.1
[[9] [1] [0]]	2	0.0001	0	[1.09511742]	-0.004588202	[1.10568485]	0.1	TRUE	TRUE	0	0.659863946	13.40412211	50	0.1	0.1
[[7] [1] [0]]	2	0.0001	0	[1.09499182]	-0.004438195	[1.1054041]	0.1	FALSE	TRUE	0	0.659637188	10.3324945	50	0.1	0.1
[[9] [1] [0]]	2	0.0001	0	[1.09496677]	-0.004521625	[1.10557394]	0.1	TRUE	FALSE	0	0.659637188	13.63291574	50	0.1	0.1
[[9] [1] [0]]	2	0.0001	-1.30E-14	[1.09529248]	-0.004904518	[1.10633642]	0.1	FALSE	FALSE	0	0.659410431	12.16090488	50	0.1	0.1

For this table we can show a graph showing convergence of the loss function:



This graph had made me think differently on the stopping criteria for a converged model, since it seems not changing for almost 20 epochs, and then changes, that is why I've added a parameter for how many epochs to average for the changing loss value, defaulting to 30. For that naturally added the threshold value, defaulting to 10e-8.

Following the different changes of the parameters lead to the conclusion that the most influencing parameter is capital Lambda, as the other parameters influenced slightly on the convergence but the for the test prediction did not change drastically, what did happen with the change of capital Lambda, as shown below:
we can see that the prediction mean is ≈ 0.73, as opposed to the 0.65 before.

numN	numL	lowerLambda	lossLastChange	lossLast	lossFirstChange	lossFirst	capitalLambda	bolAugmentLayers	bolAugment	beta	adanet_mean	adanet_fit_time	T	Ck_bias	Ck
[[24] [22] [1]]	3	0	-0.001538675	[0.96018201]	-0.004506606	[1.10553429]	1	TRUE	TRUE	0	0.698412698	43.23122168	50	0.1	0.1
[[23] [22] [2]]	3	0	-0.001504749	[0.93214464]	-0.004266754	[1.10504618]	1	FALSE	TRUE	0	0.738321995	40.07264757	50	0.1	0.1
[[26] [23] [0]]	2	0	-0.000708518	[0.95717841]	-0.005009985	[1.10653153]	1	TRUE	FALSE	0	0.765986395	39.81350827	50	0.1	0.1
[[25] [22] [1]]	3	0	-0.001224811	[0.90397878]	-0.004030694	[1.10457787]	1	FALSE	FALSE	0	0.799546485	37.90759778	50	0.1	0.1
[[27] [22] [0]]	2	0.0001	-0.004040987	[0.9313953]	-0.003697486	[1.10392687]	1	TRUE	TRUE	0	0.76984127	38.70897937	50	0.1	0.1
[[28] [21] [0]]	2	0.0001	-0.007775728	[0.97453978]	-0.004381286	[1.10528067]	1	FALSE	TRUE	0	0.688662132	41.71098042	50	0.1	0.1
[[27] [22] [0]]	2	0.0001	-0.00072829	[0.96005211]	-0.004431333	[1.1053941]	1	TRUE	FALSE	0	0.679591837	39.90307379	50	0.1	0.1
[[25] [23] [2]]	3	0.0001	-0.000749573	[0.9520086]	-0.004566399	[1.10563779]	1	FALSE	FALSE	0	0.689795918	40.14832449	50	0.1	0.1

This values and the understanding that the model didn't converge yet, has lead me to increase the number of epochs, while knowing it will stop if converged.
So new data added to the table of when it converged or stopped (as sometimes there is an optimization problem that the model is not finding optimal solution, instead of exception the model is gracefully going back to its last best known epoch).
The experiments can be shown below:

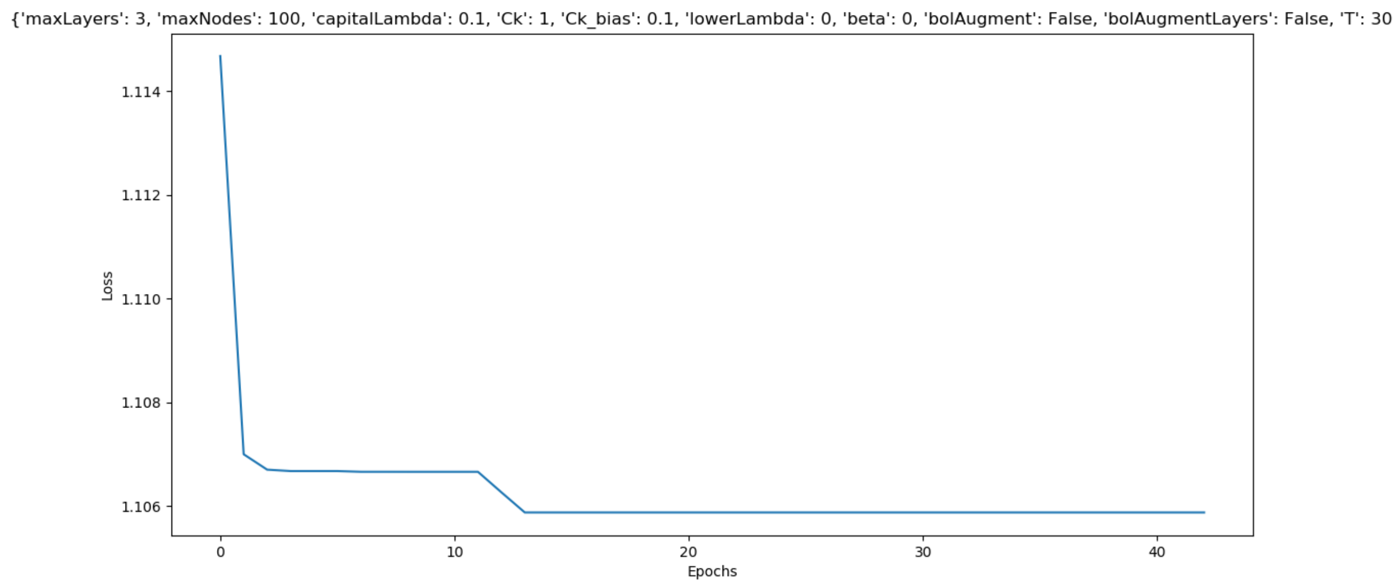
numN	numL	maxNodes	maxLayers	lowerLambda	lossLastChange	lossLast	lossFirstChange	lossFirst	dataset	capitalLambda	bolAugmentLayers	bolAugment	beta	adanet_std	adanet_mean	adanet_fit_time	actual_epochs	T	Ck_bias	Ck
[[26] [22] [2]]	3	100	3	0	-0.000720435	0.911345751	-0.004154124	1.104450876	twospirals	1	FALSE	FALSE	0	0	0.763492063	28.36062193	50	50	0.1	1
[[29] [19] [2]]	3	100	3	0	-0.0007405	0.982813895	-0.004547921	1.105127057	twospirals	10	FALSE	FALSE	0	0	0.722902494	25.87247968	50	50	0.1	1
[[17] [11] [0]]	2	100	3	0	-0.003989904	1.019920437	-0.004755099	1.105584499	twospirals	100	FALSE	FALSE	0	0	0.640136054	10.5796051	28	50	0.1	1
[[46] [2] [0]]	2	100	3	0	-0.000560573	0.85236432	-0.111754481	1.094251667	german-ida	1	FALSE	FALSE	0	0	0.72	1.531087637	50	50	0.1	1
[[28] [19] [3]]	3	100	3	0	-0.000979184	1.004617308	-0.009474112	1.092081274	german-ida	10	FALSE	FALSE	0	0	0.72	1.360077858	50	50	0.1	1
[[27] [21] [2]]	3	100	3	0	-0.000972758	0.999397228	-0.013853333	1.106369106	german-ida	100	FALSE	FALSE	0	0	0.71	1.390079498	50	50	0.1	1

numN	numL	maxNodes	maxLayers	lowerLambda	lossLastChange	lossLast	lossFirstChange	lossFirst	dataset	capitalLambda	bolAugmentLayers	bolAugment	beta	adanet_std	adanet_mean	adanet_fit_time	actual_epochs	T	Ck_bias	Ck
[[6] [1] [0]]	2	100	3	0	0	1.108942529	-0.004082202	1.119072262	twospirals	0.1	FALSE	FALSE	0	0	0.661451247	58.74936032	300	300	0.1	1
[[46] [53] [19]]	3	100	3	0	-0.000495769	0.865508376	-0.004351765	1.119605409	twospirals	1	FALSE	FALSE	0	0	0.782086168	157.4400051	118	300	0.1	1
[[99] [100] [100]]	3	100	3	0	0	0.968453099	-0.003771375	1.118425715	twospirals	10	FALSE	FALSE	0	0	0.784126984	705.8805914	300	300	0.1	1

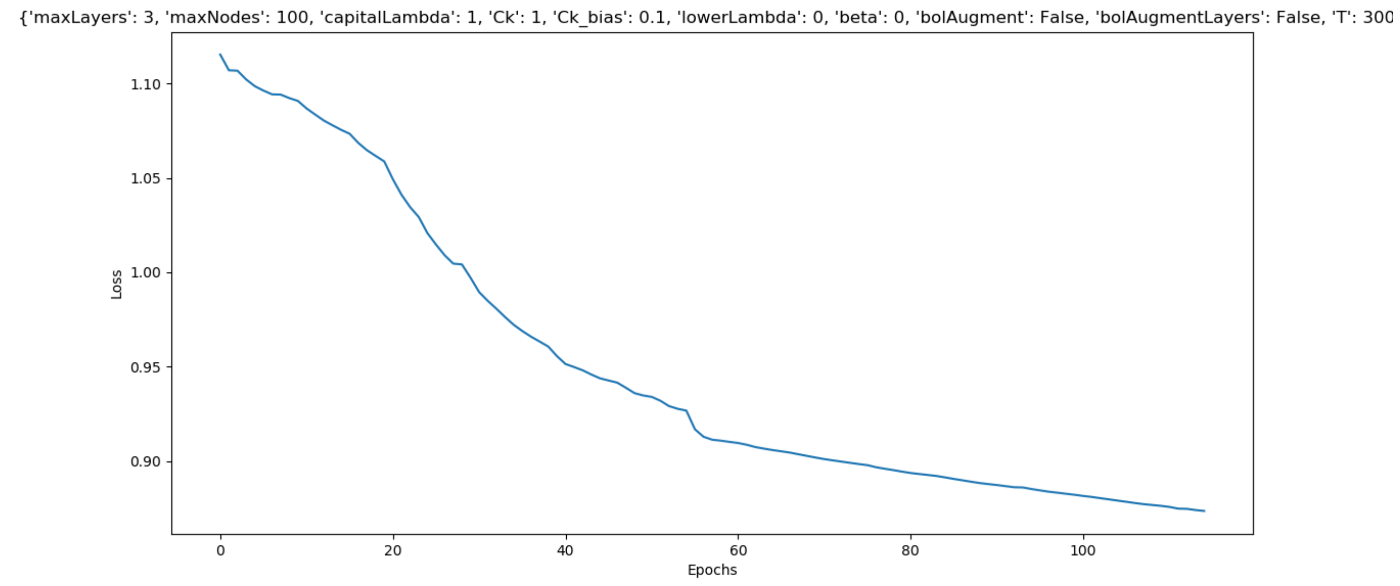
numN	numL	maxNodes	maxLayers	lowerLambda	lossLastChange	lossLast	lossFirstChange	lossFirst	dataset	capitalLambda	bolAugmentLayers	bolAugment	beta	adanet_std	adanet_mean	adanet_fit_time	actual_epochs	T	Ck_bias	Ck
[[6] [1] [0]]	2	100	3	0	0	1.105878196	-0.00398532	1.11467398	twospirals	0.1	FALSE	FALSE	0	0	0.664172336	11.6423583	43	300	0.1	1
[[37] [54] [24]]	3	100	3	0	-0.000560431	0.873651134	-0.004277532	1.115247519	twospirals	1	FALSE	FALSE	0	0	0.798185941	176.8210795	115	300	0.1	1
[[35] [30] [2]]	3	100	3	0	-0.000566219	0.979837383	-0.004208647	1.115125847	twospirals	10	FALSE	FALSE	0	0	0.745578231	49.52905297	68	300	0.1	1
[[38] [26] [4]]	3	100	3	0	-0.000534079	0.986158919	-0.004411161	1.115565191	twospirals	100	FALSE	FALSE	0	0	0.73537415	46.69863725	68	300	0.1	1
[[100] [0] [0]]	1	100	3	0	0	0.852246251	-0.114047901	1.109525076	german-ida	0.1	FALSE	FALSE	0	0	0.8	12.32051778	300	300	0.1	1
[[99] [100] [100]]	3	100	3	0	-5.79E-06	0.813562673	-0.11238177	1.102878107	german-ida	1	FALSE	FALSE	0	0	0.72	23.92255378	300	300	0.1	1
[[100] [100] [100]]	3	100	3	0	-4.39E-06	0.902521847	-0.005964826	1.082849403	german-ida	10	FALSE	FALSE	0	0	0.74	22.66435742	300	300	0.1	1
[[100] [100] [100]]	3	100	3	0	0	0.891579394	-0.010503139	1.092213873	german-ida	100	FALSE	FALSE	0	0	0.78	20.30693841	300	300	0.1	1

We can see how for large capital lambda (1,10,100), the model will structure deeper complex network, allowing slower convergence of the model, but it is not necessarily influencing the same for the accuracy of prediction.
For example capital Lambda of 1 leading to highest prediction of 0.798 in twospirals dataset, as opposed to capital Lambda of 0.1 leading to highest and simplest network for german-ida dataset(0.8 accuracy and 1 layer with 100 nodes).

Some of the graphs correspond to the table above:


{'maxLayers': 3, 'maxNodes': 100, 'capitalLambda': 0.1, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0, 'beta': 0, 'bolAugment': False, 'bolAugmentLayers': False, 'T': 30

Actual of 47 epochs, stopped by convergence


{'maxLayers': 3, 'maxNodes': 100, 'capitalLambda': 1, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0, 'beta': 0, 'bolAugment': False, 'bolAugmentLayers': False, 'T': 300

Actual of 115 epochs, stopped by optimizer not finding optimal solution,
we can see how higher capital Lambda leading to a more slowly converged model.

Another experiment done using the 'BFGS' solver as opposed to 'Nelder-Mead', the difference is that 'BFGS' optimizer is using also a gradient to find the optimal solution while 'Nelder-Mead' is using a numerical approximate of the gradient evaluations to find the minimum.
I learned that even though logically having also the gradient as a more precise approach didn't lead to better results, on the contrary regarding time that the model worked 6-8 times more than 'NM', convergence seemed the same and the accuracy shown the almost the same also.

All of the graphs and results.csv are submitted in the folder named tuning.

Strengths

1. "Accepting the general structure of a neural network as an effective parameterized model for supervised learning we introduce a framework for training neural networks that adapts the structure and complexity of the network to the difficulty of the particular problem at hand, with no pre-defined architecture."
2. "Incredibly, our method will also turn out to be convex and hence more stable than the current methodologies employed."
3. "Our techniques are general and can be applied to other neural network architectures, including CNNs and LSTMs as well as to other learning settings such as multi-class classification and regression."
4. The results this method brings, are better than the baseline models, even if treated the same with cross validation to find the hyperparameters values.

5. Not only better results but usually simpler network is learned than the other baseline models.

Drawbacks

In the paper no drawbacks are outlined, but having 3 versions of the same article, bringing so many variations of models like: ADANET.CVX (current), ADANET, ADANET.SD, ADANET.R, ADANET.P, ADANET.D, ADANET+ shows a probable drawback of complexity, they have compared each variant, but we know that it could be that for different problems one model will be better than the other, leading again to the step of searching and deciding for the best model to implement.

Experiments Results

In this section I will introduce the scores reported during the iterations.

Measures?

The measures ones will find next are:
mean accuracy, standard deviation of mean accuracy, t-test significance, mean recall score, mean f1 score, mean roc-auc score, mean fit execution time, mean prediction execution time.

Hyper-parameters and cross validation?

An approach taken in the article to evaluate the model was used here also, but in smaller scale to speed up results.
I'm iterating through a **repeated** KFold iterator, that during each repeat setting aside a test fold for future testing results, and then with the remaining folds running training and comparing to a validation fold.
then selecting the best validation scores for the hyper-parameters best combination, and using this parameters on the test folds the evaluate the models.
I have used for speeding up 3 times repeating of 3 folds leading to created 9 folds of which:
3 are set aside for the testing results.
Remaining repeated 2 times of 3 folds are used as 2 for training and 1 for validation score.
Locating the combination of hyper-parameters that led for the best validation score.
then using this best combination on the set aside test folds to create all the test result measures and significance comparison.
The data gathered during the cross validation phase and the test data were saved into 4 csv files: 3 for each of the classifiers cross validation phase and 4th for the test and comparison data.

Baselines?

I have used the same baselines as in the article:
- Multi-Layer Perceptron (MLP) or the feed forward neural network, is a straight forward baseline to compare to the adanet model scores, to compare an adaptive learned network structure to a predefined (selected by cross validation) neural network.
- Logistic Regression (LR), since we're dealing with binary classification problem, it is reasonable to select as baseline the simplest classifier to see how it handles the classification problems at hand.

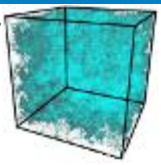
Datasets?

I've selected 10 datasets of the binary classification, from different sources: toy set randomized, mldata.org, sklearn dataset (breast cancer), UCI repository, tensorflow dataset (CIFAR-10).
The datasets are:
two spirals – a toy dataset, created by math randomization of 2 spirals data that are hard to classify as one or the other, I've attached the python file to create it. The created data set includes 44,100 samples of 2 features each to assigned label -1.0 or +1.0 (one spiral or the other).
During the tuning phase I've used this dataset and in some comparison results from the next german dataset.

german-ida –

German IDA - submitted by sonne 1623 views, 7725 downloads, 0 comments
last edited by sonne **-** Sep 14, 2011, 15:17 CETRating

Summary: The German data set from the IDA Benchmark repository.



Data Shape: 20 attributes, 1000 instances ()

License: PDDL

Tags: IDA_Benchmark_Repository

Tasks / Methods / Challenges: 1 tasks, 0 methods, 0 challenges

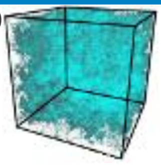
Download: HDF5 (174.6 KB) XML CSV ARFF LibSVM Matlab Octave

This dataset is fetched from mldata.org, as a ready for classifier training, since the data is preprocessed, with no missing values, and all categorical labels were converted to numerical values.

diabetes_scale –

diabetes_scale - submitted by mldata 5673 views, 9449 downloads, 0 comments
last edited by cong **-** Sep 14, 2011, 16:22 CETRating

Summary: PIMA indian diabetes data (scaled to [-1,1])



Data Shape: 9 attributes, 768 instances ()

License: unknown (from LibSVMTools repository)

Tags: libsvm LibSVMTools slurped

Tasks / Methods / Challenges: 2 tasks, 3 methods, 1 challenges

Download: HDF5 (64.5 KB) XML CSV ARFF LibSVM Matlab Octave

This dataset is fetched from mldata.org, as a ready for classifier training, since the data is preprocessed, with no missing values, and all categorical labels were converted to numerical values.
I only converted the target value from (-1,1) labels into (-1.0,+1.0) float binary values.

Wisconsin –

```
sklearn.datasets.load_breast_cancer(return_X_y=False)[source]
```

Load and return the breast cancer wisconsin dataset (classification).

The breast cancer dataset is a classic and very easy binary classification dataset.

Classes	2
Samples per class	212(M),357(B)
Samples total	569
Dimensionality	30
Features	real, positive

No need to preprocess the data, except for transforming the target labels from (0,1) to (-1.0,+1.0)

tic tac toe -

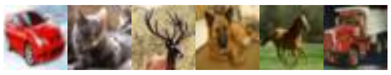
Name	Data Types	Default Task	Attribute Types	# Instances	# Attributes	Year
Tic-Tac-Toe Endgame	Multivariate	Classification	Categorical	958	9	1991

Previous datasets were fetched directly from the internet, this one from the UCI website, needed to be downloaded as a csv file (in tic-tac-toe folder), and then read and processed for the classifiers.
Preprocess included converting the features categories of 'x' (x player move), 'o' (o player move), and 'b'(blank no move) into 1.0, 2.0, 3.0 respectively.
This is using the method of "A simple approach to ordinal classification" article.
The target labels again converted from 'positive', 'negative' to +1.0, -1.0 respectively.

CIFAR 10 – (5 datasets)

The CIFAR-10 data (https://www.cs.toronto.edu/~kriz/cifar.html) contains 60,000 32x32 color images of 10 classes.

In order to convert the problem from n_classes classification I'm selecting pairs of images producing a binary problem. 5 pairs of images selected: dog_horse, deer_horse, deer_truck, automobile_truck, cat_dog.



Using built in functions in TensorFlow to access this dataset, producing 60,000 samples with 1024 features, like the article approach I have used 2 images manipulations to reduce the number of features to 155.
first is the color histogram, that divide the color information into 27 features (bins of 3 by 3 by 3),
second is the Histogram of Oriented Gradients (HOG), that finds the "edges" of the image, this produced 128 features based on the skimage package to calculate it.

Testing parameters

LR_best_params	MLP_best_params	adanet_best_params	adanet_numN	danet_numL	dataset
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[2] [1] [0]]	**2**	**automobile_truck**
{'tol': 0.001, 'C': 1.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[1] [0] [0]]	**1**	**cat_dog**
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[1] [0] [0]]	**1**	**deer_horse**
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[2] [1] [0]]	**2**	**deer_truck**
{'tol': 0.001, 'C': 1.0}	{'max_iter': 10, 'hidden_layer_sizes': (10,), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.045, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[12] [12] [22]]	**3**	**diabetes_scale**
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[1] [0] [0]]	**1**	**dog_horse**
{'tol': 0.001, 'C': 1.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[47] [3] [0]]	**2**	**german-ida**
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.0, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[35] [2] [0]]	**2**	**tic_tac_toe**
{'tol': 0.001, 'C': 1.0}	{'max_iter': 10, 'hidden_layer_sizes': (10, 10, 10), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.045, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[18] [16] [0]]	**2**	**twospirals**
{'tol': 0.001, 'C': 10.0}	{'max_iter': 10, 'hidden_layer_sizes': (10,), 'alpha': 0.001, 'learning_rate_init': 0.001}	**{'maxLayers': 3, 'maxNodes': 2000, 'capitalLambda': 1.045, 'Ck': 1, 'Ck_bias': 0.1, 'lowerLambda': 0.001, 'beta': 0.001, 'bolAugment': True, 'bolAugmentLayers': True, 'T': 50, 'optMethod': 'Nelder-Mead', 'optIsGrad': None}**	[[27] [0] [0]]	**1**	**wisconsin**

We can see that for all of the CIFAR10 datasets AdaNet built an un-reasonable network of one, two or three nodes. This is due to an immediate stop of the model, since for first several epochs there is no change in the loss function value leading to an assumption the model is converged so it stops iterating. I have tried several other parameters with no success of maintaining several epochs until it structure a learned network to classify.

Results for Accuracy AdaNet – MLP

מקסימום של AdaNet-MLP_ttest	מקסימום של AdaNet-MLP_tprob	מקסימום של MLP_accuracy_std	מקסימום של MLP_accuracy	מקסימום של adanet_accuracy_std	מקסימום של adanet_accuracy	תוויות שורה
-3974.15%	0.06%	0.54%	74.02%	0.43%	50.47%	automobile_truck
-1952.76%	0.26%	0.50%	62.42%	0.86%	49.95%	cat_dog
-6663.20%	0.02%	0.62%	75.40%	0.27%	49.90%	deer_horse
-7128.06%	0.02%	0.46%	87.08%	0.36%	49.41%	deer_truck
197.62%	18.68%	15.78%	53.13%	1.29%	74.87%	diabetes_scale
-8672.39%	0.01%	0.27%	73.66%	0.47%	49.70%	dog_horse
435.52%	4.89%	2.82%	67.66%	0.42%	76.05%	german-ida
147.76%	27.76%	6.44%	59.69%	1.70%	66.35%	tic_tac_toe
-631.22%	2.42%	0.23%	99.59%	6.16%	71.95%	twospirals
2408.87%	0.17%	3.28%	36.14%	0.99%	91.40%	wisconsin
2408.87%	**27.76%**	**15.78%**	**99.59%**	**6.16%**	**91.40%**	**סכום כולל**

First the resulting 50% accuracy of AdaNet on all the CIFAR10 datasets are due to stopped model not learning the problem, I've described above in testing parameters.

Diabetes and tic_tac_toe are better accuracy of AdaNet over MLP but without significance due to high variance of MLP results.
German and Wisconsin are better accuracy of AdaNet over MLP with significance (less than 5%, 4.9% and 0.2% accordingly), For German AdaNet learnt 47 nodes on first layer and 3 on second (using all 50 epochs to learn new node on each round) as opposed to exploiting most complex allowed structure of selected 3 layers of 10 nodes (CV selection from (10,), (10,10), (10,10,10)),
For Wisconsin AdaNet learnt only 27 nodes on first layer resulting to 91.4% of accuracy, interesting to see that MLP selected the simplest structure of only 1 layer 10 nodes, resulting to poor results of 36%.

For the toy dataset MLP is exceling with 99.6% as opposed to 72% of AdaNet, and with significance (2.4%), nevertheless we can see that AdaNet learnt on first layer 18 nodes, and on second 16 nodes, while MLP used the most complex structure of 3 layers 10 nodes each, looking in the validation results we can learn that AdaNet stopped the learning prematurely since here we see 34 learned nodes and the best validation result is of 49 nodes. So AdaNet shows that when the problem is more complex letting it enough epochs to learn the problem will result to best suit structured network.

Results for Accuracy AdaNet – LR

סכום של AdaNet-LR_ttest	סכום של AdaNet-LR_tprob	סכום של LR_accuracy_std	סכום של LR_accuracy	מקסימום של adanet_accuracy_std	מקסימום של adanet_accuracy	תוויות שורה
-5265.64%	0.04%	0.49%	74.72%	0.43%	50.47%	automobile_truck
-1794.64%	0.31%	0.27%	62.93%	0.86%	49.95%	cat_dog
-9787.26%	0.01%	0.23%	76.57%	0.27%	49.90%	deer_horse
-9087.25%	0.01%	0.36%	88.33%	0.36%	49.41%	deer_truck
-221.88%	15.67%	1.21%	75.91%	1.29%	74.87%	diabetes_scale
-4998.94%	0.04%	0.46%	73.96%	0.47%	49.70%	dog_horse
-151.19%	26.97%	0.62%	76.45%	0.42%	76.05%	german-ida
-2500.00%	0.16%	1.56%	68.96%	1.70%	66.35%	tic_tac_toe
121.11%	34.95%	0.26%	66.46%	6.16%	71.95%	twospirals
-523.72%	3.46%	0.25%	95.61%	0.99%	91.40%	wisconsin
-34209.42%	**81.62%**	**5.70%**	**759.89%**	**6.16%**	**91.40%**	**סכום כולל**

Leaving out the CIFAR10 datasets due to the AdaNet learning issue, it's interesting to see that LR is doing better on all of the datasets, but it is reasonable since LR is looking for the optimal solution it can without boundaries of time or epochs, so it's doing its best, as opposed to AdaNet that is limited to 50 epochs, we can see that the results are better but of no significance except for Wisconsin with 3.5% significance.

Results for f1score,recall,AUC AdaNet – MLP

מקסימום של MLP_auc	מקסימום של adanet_auc	מקסימום של MLP_recall	מקסימום של adanet_recall	מקסימום של MLP_f1score	מקסימום של adanet_f1score	תוויות שורה
18.05%	47.85%	71.13%	100.00%	73.36%	67.09%	automobile_truck
32.52%	48.93%	60.65%	100.00%	61.70%	66.62%	cat_dog
16.66%	67.58%	71.32%	100.00%	74.31%	66.58%	deer_horse
5.66%	71.86%	84.44%	100.00%	86.59%	66.14%	deer_truck
43.70%	81.20%	66.67%	87.76%	52.12%	82.11%	diabetes_scale
18.98%	50.30%	66.95%	100.00%	71.63%	66.40%	dog_horse
39.10%	77.81%	37.73%	45.53%	37.33%	53.20%	german-ida
48.85%	62.17%	79.51%	96.97%	71.31%	79.22%	tic_tac_toe
0.02%	80.75%	99.64%	72.16%	99.59%	71.90%	twospirals
21.82%	89.50%	0.00%	96.94%	0.00%	93.50%	wisconsin
48.85%	**89.50%**	**99.64%**	**100.00%**	**99.59%**	**93.50%**	**סכום כולל**

The f1score, recall and AUC are following the same path as the accuracy score explained above in relevant section, so besides CIFAR10, AdaNet is doing better than MLP except for twospirals dataset.

Results for f1score,recall,AUC AdaNet – LR

מקסימום של LR_auc	מקסימום של adanet_auc	מקסימום של LR_recall	מקסימום של adanet_recall	מקסימום של LR_f1score	מקסימום של adanet_f1score	תוויות שורה
17.47%	47.85%	74.88%	100.00%	74.93%	67.09%	automobile_truck
32.24%	48.93%	61.97%	100.00%	62.54%	66.62%	cat_dog
16.02%	67.58%	75.49%	100.00%	76.27%	66.58%	deer_horse
4.86%	71.86%	87.32%	100.00%	88.08%	66.14%	deer_truck
18.62%	81.20%	88.41%	87.76%	82.82%	82.11%	diabetes_scale
18.80%	50.30%	73.11%	100.00%	73.62%	66.40%	dog_horse
22.11%	77.81%	47.52%	45.53%	54.78%	53.20%	german-ida
37.65%	62.17%	92.45%	96.97%	79.76%	79.22%	tic_tac_toe
26.99%	80.75%	66.95%	72.16%	66.48%	71.90%	twospirals
0.50%	89.50%	96.13%	96.94%	96.55%	93.50%	wisconsin
37.65%	**89.50%**	**96.13%**	**100.00%**	**96.55%**	**93.50%**	**סכום כולל**

Same here it is following the pattern shown in accuracy, that LR is doing a little bit better than AdaNet.

Results for fit time, predict time AdaNet – MLP - LR

מקסימום של LR_pred_time	מקסימום של MLP_pred_time	מקסימום של adanet_pred_time	מקסימום של LR_fit_time	מקסימום של MLP_fit_time	מקסימום של adanet_fit_time	תוויות שורה
0.007	0.010	0.102	2.011	1.047	1.760	automobile_truck
0.005	0.010	0.092	1.057	1.505	1.505	cat_dog
0.010	0.006	0.105	1.433	1.086	1.449	deer_horse
0.010	0.005	0.099	1.567	1.230	1.594	deer_truck
0.001	0.001	0.005	0.005	0.081	1.513	diabetes_scale
0.006	0.005	0.114	1.961	1.266	1.728	dog_horse
0.001	0.000	0.005	0.007	0.212	1.761	german-ida
0.001	0.006	0.005	0.006	0.111	1.214	tic_tac_toe
0.000	0.010	0.104	0.055	3.557	26.174	twospirals
0.000	0.000	0.000	0.011	0.047	0.958	wisconsin
0.010	**0.010**	**0.114**	**2.011**	**3.557**	**26.174**	**סכום כולל**

First we can see that all 3 classifiers behave the same, more time on fitting a model, and only a fraction to predict. Second even though AdaNet is searching for the solution while building the network, it is using almost the same time as the MLP. LR is much faster than them both.
twospirals dataset is an exception, that should be investigated as for AdaNet it takes more time, but still it is doing poorly compared to MLP.

Conclusion

The purpose of this project was to confront an article and to be able to implement it.
I must admit this article is very complex with high math (for me at least), and in order to fully be able to grasp the model, it was required to first learn deeply several math topics related to the article.
Nevertheless the exposure to this frontier model and knowledge is priceless and fascinating.
During the work of the project I was exposed to several new interesting tools and areas like CIFAR10 dataset and classification, that required to figure out how to reduce raw image features into color histograms and histogram of Oriented Gradients (HOG), in the article they have done the same thing but figuring out how to do it is interesting.
Another thing I thought will be required was to use the TensorFlow package to build the network, so I've built one for the LR and one for the MLP, but not for AdaNet, as I saw the code is suffice for that, so I've turned back and used the sklearn implementation of LR and MLP instead (mainly because the ease of use of different searched parameters)

Even though the model implementation is not highly stable and consistent, the reported potential is shown immediately, and this ability to adapt a network based on different problems, overcoming so many issues with today's pre-processing of neural networks of how to build one that solves ones problem is amazing.
When someone is approaching a problem to solve with a neural network, the first question is how wide and how deep? Searching for several architectures will not promise the maximum available results, so even by getting good results, it is not saying that there does not exist either a simpler solution with same results, or more complex solution with better results.
Adanet is doing just that, finding the best architecture solving the problem, but doing it while updating the network weights as well.

Suggestions for further developments and analysis:

1. First thing is to follow the math requirements and stabilize the model (it does throw warnings during operation, regarding overflow arithmetic or using exponential function incorrectly)

2. Second to implement the last v.3 AdaNet model that is not adding one node at a time but admitting another parameter B (number of nodes on adding a new piece of layer), this version is suggesting great results on any problem given.

3. The article itself is noting that the theory used for this model can be used to implement different kinds of networks, like CNN or LSTM, but also for exotic choices (like Huang et al., 2016), so it is definitely worth to explore implementation of CNN or LSTM AdaNet model.

Citations

From researchgate site, while writing the report there were 15 citations:
https://www.researchgate.net/publication/304857984_AdaNet_Adaptive_Structural_Learning_of_Artificial_Neural_Networks/citations

See the red highlighted sentences as alternatives to AdaNet.

... This model was shown to outperform non-incremental denoising autoencoders in classification tasks with the MNIST (LeCun et al., 1998) and the CIFAR-10 (Krizhevsky, 2009) benchmark datasets. Cortes et al. (2016)proposed to adapt both the structure of the network and its weights by balancing the model complexity and empirical risk minimization. In contrast to enforcing a pre-defined architecture, the algorithm learns the required model complexity in an adaptive fashion. ...

Continual Lifelong Learning with Neural Networks: A Review

... Architecture search is not limited to using RL or evolution. Alternative approaches explored include cascade-correlation (Fahlman & Lebiere, 1990), boosting (Cortes et al., 2016;Huang et al., 2017), hill-climbing (Elsken et al., 2017), MCTS (Negrinho & Gordon, 2017), SMBO (Mendoza et al., 2016;Liu et al., 2017a), random search (Bergstra & Bengio, 2012) and grid search (Zagoruyko & Komodakis, 2016). Moreover, meta-learning also encompasses the discovery of activation functions (Ramachandran et al., 2017) and optimizers (Bello et al., 2017), for example. ...

Regularized Evolution for Image Classifier Architecture Search

... We believe this algorithm might be able to go deeper without losing performance by partially overcoming the vanishing gradient problem, learning "mapping" filters to maintain the features sparseness, and learn a bigger set of high level features. In addition, the Deep Cascade Learning has the potential to find the number of layers required to fit a certain problem (adaptive architecture), similarly to the Cascade Correlation [1], Infinite Restricted Boltzmann Machine [33], and AdaNet [34]. ...

Deep Cascade Learning

... Several groups have performed architecture search by incrementally increasing the complexity of the model: Stanley & Miikkulainen (2002) used a similar approach in the context of evolutionary algorithms, used a schedule of increasing number of layers, and Grosse et al. (2012) used a similar approach to systematically search through the space of latent factor models specified by a grammar. Finally, Cortes et al. (2017);Huang et al. (2017a) Image grow CNNs sequentially using boosting; however, their results do not improve the accuracy over manual designs. ...

Progressive Neural Architecture Search

... Genetic algorithms are also explored in learning network structures [40]. In [7], the AdaNet was proposed to learn directed acyclic network structures using a theoretical framework with some guarantee. The proposed AOG building block is potentially useful as a better heuristic in the search or to facilitate theoretical analyses by taking advantage of the simple production rule in structure composition. ...

AOGNets: Deep AND-OR Grammar Networks for Visual Recognition

... This is caused by the greedy construction of the model layer by layer with finer granularity in the total number of nodes in each layer. Our framework is independent of the type of input features so it can be used for other purposes as well (e.g. as a weak learner in ADANET [Cortes et al., 2016a]). Algorithm 1 Constructs a network where γ is the average growth of a layer on each training step and Γ is a threshold. ...

Solution Description

This document is accompanied with 4 code files implemented with Python 3.6,
which are extensively documented closely as Google style guide,
http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html :

AdaNet_CVX.py – The API code, implements the AdaNet.CVX algorithm

```
AdaNet: Adaptive Structural Learning of Artificial Neural Networks
This API implements using algorithm based on the article:
http://proceedings.mlr.press/v70/cortes17a.html

In short this model is building from scratch a neural network according to the data
complexity it fits,
this is why it named as adaptive model. The problem at hand is always a binary
classification.
During fit operation it will build the hidden layers and number of neurons in each layer.
The decision if to go deeper (add hidden layer) or to go wider (add neuron to existing
layer),
or update an existing neuron weight is done in a closed form of calculations
(by using Banach space duality) shown in the article.
Lastly it will optimize the weight of the best neuron (added or existing), update
parameters and iterate.

The article talks about several variants of AdaNet, this is the AdaNet.CVX implementation,
explained on Appendix C - that solves a convex sub problem in each step in a closed form.

Further detailed explanations of this variant is shown in a previous version of the
article [v.1]:
all versions: https://arxiv.org/abs/1607.01097
v.1: https://arxiv.org/abs/1607.01097v1
```

test_adanet.py – The helper module to load, initialize and score the classifiers (AdaNet.CVX, MLP as FFNN, LR)

```
This helper module is used to implement and test the AdaNet_CVX.py API

- First part is loading 10 datasets for comparisons
- Second part is defining which classifiers will be used as baselines to test AdaNet (MLP-
FFNN, LR)
- Third part is for scoring results and includes several functions to iterate and produce
relevant scores
```

AdaNet_CIFAR_10_feature_extraction.py – a helper module to create the CIFAR10 dataset
I have implemented to save the extracted features in a compressed numpy array to disk, so it will be fast for testing, if ones is iterating and starting over.

```
CIFAR-10 Image Category Dataset
The CIFAR-10 data ( https://www.cs.toronto.edu/~kriz/cifar.html ) contains 60,000 32x32
color images of 10 classes.
It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
Alex Krizhevsky maintains the page referenced here.
This is such a common dataset, that there are built in functions in TensorFlow to access
this data.

Running this command requires an internet connection and a few minutes to download all the
images.
```

twospirals.py– a helper module to create a toy dataset of two spirals to classify correctly

```
Generate "two spirals" dataset with N instances.
degrees controls the length of the spirals
start determines how far from the origin the spirals start, in degrees
noise displaces the instances from the spiral.
 0 is no noise, at 1 the spirals will start overlapping
```

How the API implemented:

__init__ method

The class has the init method to initialize the class

```
class AdaNetCVX(object):
    def 
__init__(self,maxLayers=5,maxNodes=50,capitalLambda=10,pnorm=2.0,Ck=10.0,Ck_bias=10.0,\

lowerLambda=0.001,beta=0.1,bolAugment=True,bolAugmentLayers=True,activationFunction='tanh'
, T=50,lossFunction='binary',surrogateLoss='logistic', numStableEpoch=30,
minAvgLossChange=1e-8, optMethod='Nelder-Mead', optIsGrad=None):
        """ init of AdanNetCVX class as the classifier instance

        Args:
            maxLayers: int, default=5, max hidden layers that AdaNet can extend to
            maxNodes: int, default=50, max neurons that AdaNet can use on each hidden 
layer
            capitalLambda: float, default=10, hyper parameter of the lp norms of the 
weights defining new nodes
                           each layer has the same value, will be used to define 
cfg['maxWeightNorms'] param.
            pnorm: float, default=2, lp norm used to calculate the weights of new nodes
            Ck: float, default=10, upper bounds on the nodes in each layer
            Ck_bias: int, default=10, upper bounds intercept bias on the nodes in each 
layer
            lowerLambda: float, default=0.001, part of calculation of complexity penalties 
(regularizer) capital Gamma
                            used in reg method as Gamma = lambda*rj + beta.
                            The regularization term is a weighted-l1 penalty 
(Gamma*abs(w))
            beta: float, default=0.1, part of calculation of complexity penalties 
(regularizer) capital Gamma
                            used in reg method as Gamma = lambda*rj + beta
                            The regularization term is a weighted-l1 penalty 
(Gamma*abs(w))
            bolAugment: boolean, default=True
            bolAugmentLayers: boolean, default=True
            activationFunction: str, default='tanh' (hyperbolic tan), activation function 
used at each node
                                      for the hypothesis value
            T:          int, default=50, number of ephocs the model will run for 
convergence
            lossFunction: str, default='binary', the model is used as a binary 
classification problem,
                              but this foundation is to be ready to implement future 
option of regression problem
                              will be used as 'MSE' value to calcualte mean squared error.
            surrogateLoss: str, default='logistic', the surrogate loss function is defined 
in the article
                           as the capital phi, that is activated on the difference of the 
zero/one loss problem (1-y*(Sig(w*h)))
                           this is in order to be sure the sub problem is convex for 
optimization.
```

```
                            activation function is the logistic as exp(x)/(1+exp(x)).
                            alternative is the 'exp' function as , exp(x)
            numStableEpoch: int, default=30, over this number of epochs will be calculated
the average of loss change,
                             if it will show to be less than a threshold, a convergence
will be assumed
                             and the fitting iterations will stop.
            minAvgLossChange: float, default=1e-8, this is the threshold of the average
loss change,
                              over a stable number of epochs. if the average is not higher
than this value,
                              convergence will be assumed and the fitting iterations will
stop.
            optMethod: str, default='Nelder-Mead' the method that the optimizer will use,
                             for the step search of best node weight
            optIsGrad: boolean, default=None, if to ask for the gradient from the loss
function,
                             if required by some optimizers methods, if yes a True value is
required,
                             None value otherwise.

        """
```

_init_cfg method

```
    def _init_cfg(self):
        """inner helper to initialize the configuration dictionary (cfg)
        based upon the model is working
        """
```

__str__ method

```
    def __str__(self):
        """Implementing the str method to return classifier name
        """
```

get_params method

This method will show its current parameters, either by initialize or if updated by setting new parameters

```
    def get_params(self, cfgout=False):
        """Get parameters for this estimator.
        Args:
            cfgout (boolean): if to return AdaNet init configuration based on input params
                            as another dictionary default to False
        Returns:
            params (dict): mapping of current input parameter names mapped to their
values.
                            if set_params() not used, will return the parameters the
classifier was initialized
            cfg (dict): based on input parameters a configuration dictionary is created
                         based upon the model is working
        """
```

set_params method

This method will set new classifier parameters

```
    def set_params(self, **params):
        """Set the parameters of this estimator.
        Args:
            **params (kwargs): a dictionary like list of parameters and their values,
                                e.g. T=1000 will set the number of epochs the model will run
        Returns:
                self
        """
```

_adanet method

This method is the main method that orchestrate all the operation of the model

```
    def _adanet(self, Xdata,ydata, cfg):
        '''Main method for AdaNet.CVX
        This method encapsulating all parts needed to learning the structure of the
adapted neural network.
        It is using inner methods, in order to simplify input and output arguments,
        while those inner methods are used only during the fit operation.
        Inner methods include:
            - ExistingNodes() to calc the value of updating weights of existing nodes
            - NewNode() to calc the value of adding a new node to each of existing hidden
layers
                or a starting a new one.
            - BestNode() - to find from the above methods what should be next step in
building the network,
                either fine tune existing weights, or adding a neuron (node)
                either on one of existing layers or open a new hidden layer
            - ApplyStep() - for the best node selected in BestNode() calc the weight based
on line search optimization
            - UpdateDistribution() - this method updates the distribution over the sample
as input for the next iteration
        Args:
            Xdata: Ndarray [m_samples, n_features]
                    input train data
            ydata: Ndarray [m_samples,1]
                    target train labels in [-1,1]
            cfg:   dict, base configuration hyper parameters of the model
        Returns:
            adaParams: dict, learned architecture and parameters of the network,
                          will be used during predict and score functions.
            history: dict, saved snapshots of learned network during fit operation,
                           for use of debuging
        '''
```

ExistingNodes method

This method is inner to the main _adanet method, and will score existing nodes.

```
        def ExistingNodes():
            ''' Existing-nodes function
            This method calculate the value of updating weights of existing nodes d_kj
(also intercept value d_bias),
            this value will be compared to new nodes method values (in the next method
NewNodes()),
            and decide best approach minimizing loss function in the BestNode() method.
            Returns:
            d_kj: dict, each key represent existing hidden layer {0...numLayers-1}
                        and the dict value is a NDarray [numNodes,1] representing for each
node the score value,
                        which is higher is better for comapring alternatives of using new
nodes or existing ones.
            d_bias: NDarray [1,1], the intercept value (bias) for the d_kj above
            '''
```

NewNodes method

This method is inner to the main _adanet method, and will score new nodes.

```
        def NewNodes():
            ''' New nodes function
            This method calculate the score value (dn_k) for adding new nodes on each of
the existing layers
            (while max nodes on the layer has not yet reached),
            and also what is the score value of adding a node on a possible new hidden
layer
            (again while max hidden layer has not been reached).
            This values including the previous calculations of existing nodes,
            will be compared in the next function of BestNode()
            Returns:
            dn_k: dict, each key represent existing hidden layer + 1 optional new hidden
layer {0...numLayers}
                        and the dict value is a NDarray [1,1] representing for each new
node on relevant hidden layer
                        the score value, which is higher is better for comapring
alternatives of using new nodes or existing ones.
            un_k: dict, each key represent existing hidden layer + 1 optional new hidden
layer {0...numLayers}
                        and the dict value is a NDarray [numNodes(k-1),1] representing the
weights coming from previous layer (k-1)
            '''
```

BestNode method

This method is inner to the main _adanet method, and will find the highest score of node based upon decision of how to build the network.

```
        def BestNode():
            ''' Best node function
            This method examine the values coming from ExistingNodes() and NewNodes(),
            and select the best approach for this iteration, based on this decision,
            if decide to add a new node to the network, it will calculate relevant
information of the new node
            as updated loss value, what is the state of network regarding of number of
hidden layers,
            and number of nodes on each hidden layer.
            if decide to stay with existing nodes it will only point to best node in a
layer as jk_best
            Returns:
            jk_best: str ('bias') if selected best approach is only update bias,
                    or Ndarray [1,1] => [location of best node, on which layer this node
resides]
            e_t:  dict, each key represent existing hidden layer + 1 optional new hidden
layer {0...numLayers}
                    the value at each key is Ndarray of [Number of Nodes on this layer,1],
                    while each value represent the error loss of this node
            numLayers: int, current number of layers in the structured network,
                       will be updated, if by best approach the new node will be selected
on a new hidden layer
            numNodes: Ndarray [maxNumLayers,1], current number of nodes on each of hidden
layers,
                       will be updated, if by best approach a new will be selected, either
on existing or new layer,
                       number of nodes will be updated.
                       the gap from existing number of layers to max allowed layers, is
carying 0 as number of nodes.
            '''
```

ApplyStep method

This method is inner to the main _adanet method, and will find the step size it should take to update the weight of the found best node to update.

```
        def ApplyStep(jk_best, surrogateLoss):
            ''' Apply step
            This method will for the best node selected in BestNode() calculate
            the new weight of the selected node based on line search optimization
            Args:
            jk_best: str ('bias') if selected best approach is only update bias,
                    or Ndarray [1,1] => [location of best node, on which layer this node
resides]
            surrogateLoss: str, 'logistic' or 'exp' used as the capital phi(-x) in the
article to impose
                         a non-increasing convex function upper bounding the 0/1 loss
w.r.t W (weight).
                         can be exponential phi(x)=exp(x), or logistic function
phi(x)=log(1+exp(x))
            Returns:
```

```
            Wt: dict, each key represent existing hidden layer {0...numLayers-1}
                        and the dict value is a NDarray [numNodes,1] representing for each
node the weight value,
                        considered as the weight of the node hypothesis.

            TODO:
            Optimizer is using Nelder-Mead solver using an approximate gradient
numerically
                      without the use of a gradient (jacobian),
                      while testing with BFGS solver using the gradient in the optimizer,
                      after certain amount of epochs the solver throws failure to
optimize,
                      looking for solution suggest maybe the values of the gradient goes
negative,
                      i couldn't figure out to resolve, so went for the Nelder-Mead solver
without a gradient.

            Same phenomenon is happening with Nelder-Mead, although less frequent,
                    my workaround is to reverse to last known good data, stop iterations
of convergence,
                    and return with that results.

            Comment on MATLAB reference code to Normalize sum of weights to 1??
                As this is part of a proof in the paper (which I couldn't find),
                As the refernce was corresponding with one of the paper's authors,
                it could come from that.
            '''
```

updateDistribution method

This method is inner to the main _adanet method, and will update the distribution over the sample.

```
        def updateDistribution(Wt):
            ''' Update Distributions
            This method will update the maintained distibution on ephoc t,
            over the sample (Xdata is a sample of the distribution D).
            The distribution is a gradient of the objective loss function,
            and normalized over the sample (overhaul sum of sample grads).
            Args:
            Wt: dict, each key represent existing hidden layer {0...numLayers-1}
                        and the dict value is a NDarray [numNodes,1] representing for each
node the weight value,
                        considered as the weight of the node hypothesis.
            Returns:
            Dnew: Ndarray [m_samples,1], the maintained distribution over the sampled data
for the current ephoc t,
                  our dataset is a sample from the distribution D, that we try to
estimate.
                  calculated as the gradient of the objective loss function.
            Snew: Ndarray [1,1], the normalization factor of the distribution,
                  as the overhaul sum of each distribution value of the sampled dataset
            '''
```

_activation method

Inner helper method to activate "crush" functions on the nodes of the built network.

```
    def _activation(self,h,afunc):
        ''' Activation function
        The node activation function used in the network over the hypothesis value
        Args:
            h: Ndarray[m_samples, n_nodes] hypothesis value to activate upon the crushing
function
            afunc: str, the activation function either 'relu', 'tanh' or 'sigmoid'
        Returns:
            a: Ndarray[m_samples, n_nodes] the h value after activation
        '''
```

RademacherComplexity

```
    def RademacherComplexity(self,H):
        '''Rademacher Complexity function
        This method computes the Rademacher complexity over the hypothesis H,
        Rademacher is a distribution with equally probability of 0.5 for -1 and +1 values.
        in order to add regularization to the objective loss function to become a coercive
convex function.
        Args:
            H: Ndarray[m_samples, n_nodes] the hypothesis value to add Rademacher noise to
it
        Returns:
            R: Ndarray[m_samples, n_nodes] the hypothesis value after the Rademacher noise
added to it
        '''
```

GaussianComplexity method

An alternative to Rademacher complexity.

```
    def GaussianComplexity(self,H):
        '''Gaussian Complexity function
        This function computes the Gaussian complexity over the hypothesis H,
        The Gaussian distribution is the normal distribution with mean=0 and sigma=1
        in order to add regularization to the objective loss function to become a coercive
convex function.
        Args:
            H: Ndarray[m_samples, n_nodes] the hypothesis value to add Gaussian noise to
it
        Returns:
            R: Ndarray[m_samples, n_nodes] the hypothesis value after the Gaussian noise
added to it
        '''
```

reg method

The Gamma part of the regularizer term of the objective loss function.

```
    def reg(self,R,lam,beta):
        '''Regularization parameter
        This function calculates the Capital Gamma of the object loss function, as the
regularization parameter
        Args:
            R: Ndarray[1, ] the hypothesis value after a noise distribution added to it
                the H value sent to this function is the overhaul value of H on the k
layer
            lam: float, lower lambda as a hyper parameter for the regularization term,
                it will multiply r value (noised h) with the lambda parameter
        beta: Ndarray[1, ] as a hyper parameter for the regularization term,
                it will be added to the multiplied term to construct gamma value
        Returns:
            gamma: Ndarray[1, ] the capital gamma value as the multiplier of the
regularization term with the weights
                weighted l1 penalty (abs of weights).
                Capital_GAMMA = lowerLambda*rj+beta
        '''
```

_adanet_init method

Inner initialization method will be executed as a first step of fit method.

```
    def _adanet_init(self,numExamples,numInputNodes,cfg):
        ''' Initialization function
        implementation of Init method, Figure 5,  pg 15
        This method
        Args:
            numExamples: int, number of samples (rows) provided by the input parameter X
in fit() method
            numInputNodes: int, number of features (columns) provided by the input
parameter X in fit() method
            cfg: dict, configuration dictionary based on input parameters,
                        or during set_parmas() method the configuration parameters can be
set,
                        based upon the model is working.
        Returns:
            AdaInit: dict, extended initialized configuration dictionary based on the
specific data provided,
                        during the fit operation, the values in the dictionary will be
updated,
                        according to the model.
                        examples: numNodes Ndarray [maxNodes,1] is initialized to zero
                                  on each possible hidden layer (maxNodes).
                                  during fit, added nodes will update the number of nodes
                                  on each of created layers by the model.
        '''
```

_loss_function method

The objective loss function of the model

```
    def _loss_function(self,w_k, h_k, reg_k, y, loss_notk,reg_notk, surrloss,lossfunc):
        ''' Loss Function
        implementation of Coordinate descent described on pg 6
        This method calculate the objective loss function of the model,
        supporting the coordinate descent operation (optimizer) for the line search
optimization
        to update the weight of the best node,
        that will maximize convergence of the model (minimizing the loss function)
        Args:
            w_k: float, the current weight of the best node selected to find a step update
(coordinate descent)
                 that will minimize the loss function
            h_k: Ndarray [m_samples,1], the hypothesis values of the best node by each of
the samples.
            reg_k:  Ndarray[1,1], the Gamma value of the best node layer
            y: Ndarray[m_samples,1], the target values
            loss_notk: Ndarray[m_samples,1] the "loss" of all but the best node,
                        implement yi*ft-1(xi) part of the loss function
            reg_notk: Ndarray[1,1], the sum of regularizer term of all but the best node
layer,
                                    implemented as all but best node => Sigma(Gamma*W)

            surrloss: str, default='logistic', the surrogate loss function is defined in
the article
                           as the capital phi, that is activated on the difference of the
zero/one loss problem (1-y*(Sig(w*h)))
                           this is in order to be sure the sub problem is convex for
optimization.
                           activation function is the logistic as exp(x)/(1+exp(x)).
                           alternative is the 'exp' function as , exp(x)
            lossfunc: str, default='binary', the model is used as a binary classification
problem,
                            but this foundation is to be ready to implement future option
of regression problem
                            will be used as 'MSE' value to calcualte mean squared error.
        Returns:
            loss: objective loss function,
                  this value will be used by the optimizer to find the w_k (weight of best
node),
                  that best minimizing the function
            grad: gradient of the loss function,
                    can be used by some of the optimizers to faster and accurate
                    find the w_k that best minimizing lost function
        '''
```

_slfunc method

An inner helper method of the surrogate activation of the loss function, denoted as phi in the model.

```
    def _slfunc(self,x, func):
        ''' surrogate loss function
        the surrogate loss function is defined in the article
        as the capital phi, that is activated on the difference of the zero/one loss
problem (1-y*(Sig(w*h)))
        this is in order to be sure the sub problem is convex for optimization.
        Args:
            x: Ndarray, can be in different input shapes, will return same shape as input
               the value to activate upon, the surrogate loss function
            func: str, either 'logistic' or 'exp' to activate relevant function on the
input value
        Returns:
            val: Ndarray, same shape as input x,
                 the resulted value by the surrogate activation function used on the input
value
        '''
```

_slgrad method

An inner helper method of the gradient of the surrogate activation of the loss function, denoted as phi' in the model.

```
    def _slgrad(self,x, func):
        '''gradient of surrogate loss function
        Args:
            x: Ndarray, can be in different input shapes, will return same shape as input
               the value to activate upon, the gradient of surrogate loss function
            func: str, either 'logistic' or 'exp' to activate relevant gradient function
on the input value
        Returns:
            valG: Ndarray, same shape as input x,
                  the resulted value by the gradient surrogate activation function used on
the input value
        '''
```

_adanet_predict method:

```
    def _adanet_predict(self,params, Xdata):
        '''inner method of predict implementation of AdaNet
        This method is feeding the input data across the learned AdaNet network,
        and evaluate for each input the probability of target value on scale -1 to 1.
        Args:
            params: dict, learned architecture and parameters of the network
            Xdata: Ndarray[m_samples,n_features], test samples to evaluate target values
        Returns:
            pred: Ndarray [m_samples,1]
                  probability of target value on scale -1 to 1
        '''
```

fit method:

```
    def fit(self, X, y):
        """ Fit method of the AdaNet classifier
        Args:
            X: Ndarray[m_samples,n_features], train samples to learn the optimal neural
network,
                that is required by this specific problem. complex problems will result
deep and wide network,
                while simple ones will result simple architecure of the network
            y: Ndarray[m_samples,1], train target labels in [-1,1], as a binary problem
        """
```

predict method:

```
    def predict(self, X):
        '''Predict implementation of AdaNet
        This method calling the inner predict method that
        is feeding the input data across the learned AdaNet network,
        and evaluate for each input the target value.
        Args:
            X: Ndarray[m_samples,n_features], test samples to evaluate target values
        Returns:
            y_pred: Ndarray [m_samples,1]
                    target evaluated labels in [-1,1]
        '''
```

predict_proba method

```
    def predict_proba(self, X):
        '''Probability estimates of AdaNet
        This method provides the probabilities of predicted lables based on test dataset.
        Args:
            X: Ndarray[m_samples,n_features], test samples to evaluate target values
probabilities
        Returns:
            predict_proba(): Ndarray [m_samples,1]
                            based on the binary problem evaluate the positive probability
                            of target label in range 0 to 1.
        '''
```

score method

This method will compute the accuracy based on input data and true labels.

```
    def score(self, X, y):
        """Score implementation of AdaNet
        used to measure accuracy score (mean) of given test data and labels.
        Args:
            X,y test data and true labels, in order to predict and compare with true
labels,
              measuring the accuracy score
        Returns:
            score(): float, accuracy score
        """
```

Using the API:

- first import the module class:

```
from AdaNet_CVX import AdaNetCVX
```

- Second initialize the class with relevant parameters, all parameters have default values.

```
adanet_clf = AdaNetCVX(**params_adanet)
```

- Third, ones can start using the fit and predict standard methods, just like with sklearn package.

```
adanet_clf.fit(X_train, y_train)
```

After fit operation the classifier provides attributes that can be queried to understand and debug the model operation: adanet_clf.adaParams, adanet_clf.history

both are dictionaries with keys indicating the operation of the model.

```
adanet_clf.predict(X_test)
```

Using test_adanet to evaluate AdaNet_CVX API

As noted this helper module is divided into 3 parts: load data, initialize classifiers, evaluation report

Just running this helper will produce 4 csv files with the results of the cross validation of each of the classifiers, and the testing results including comparing them for significance.

First part is loading 10 datasets as requested, from different sources: toy set randomized, mldata.org, sklearn dataset, UCI repository, tensorflow dataset.
the datasets will be furthered explained later, are: twospirals, german-ida, diabetes_scale from mldata.org, Wisconsin breast cancer from sklearn, tic tac toe from UCI repository, 5 pairs of photos (to convert the problem from n_classes classification to binary) from the CIFAR10 dataset dog_horse, deer_horse, deer_truck, automobile_truck, cat_dog.

Second part initializing 3 classifiers AdaNet to evaluate and MLP (Multi-Layer Perceptron or FFNN Feed Forward Neural Network) and LR (Logistic Regression) as compared baselines.

Third part is iterating the different datasets and different classifiers, then for each first iterate a **repeating** KFold split (used 3 folds, 3 times => from the created 9 folds, 3 will be set aside as test folds, and then twice take each 3 folds split into two train folds and one validation fold) , this allow us to first cross validate the best experimented hyperparameters, and then with a several set aside test folds to evaluate test scores. Lastly average the resulted scores and comparing t-test for significance into a csv file.