Tensor Dynamic

-An open source library for dynamically adapting the structure of deep neural networks

by student Daniel Slater

A dissertation submitted in partial fulfilment of the requirements for
the MRes in Computer Science
Department of Computer Science and Information Systems

Birkbeck College, University of London
September 2017

"This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme booklet and the College website. The report may be freely copied and distributed provided the source is explicitly acknowledged."

Abstract

Deep learning techniques have proved successful across a range of complex tasks. When building deep neural networks the structure of the network, the number of layers and nodes per layer must be specified in advance and can have a huge effect on the performance of the model. This thesis looks at approaches to learning the optimum structure as a part of training. We develop a library for adapting structure as part of training, called Tensordynamic. We then look at algorithms that can be applied to learning structure eventually suggesting one approach that can learn good topology as part of training in faster time and with comparable results to a hyper parameter grid search.

Table of contents

Acknowledgments

I would like to thank my wife Judit Kollo and my son David.

Chapter 1 - Introduction

Deep learning techniques have proved incredibly successful at learning a range of complex tasks. The current best performance in computer vision, speech recognition and AI all involve many layered neural networks. Unfortunately these approaches can be incredibly slow to train, and come with an abundance of hyper-parameters that can have a large impact on performance. These hyper parameters include the number of hidden layers, the number of nodes in each layer, the type and magnitude of regularization, the type and parameters for learning method (e.g. RMSProp[1], Adam Optimizer[2], etc), the kinds of activation functions and types of layers.

A common approach to determining the best values of these hyper-parameters is known as grid search[3]. This involves running the network from scratch many times with a range of hyper parameters and then selecting those that produce the best results. The major downside of this is that training successive network is very time consuming and as the number of hyper-parameters grows, the number of networks that must be trained grows rapidly. If we have x parameters that need adjusting, which each can have y different values, we need to run yx simulations.

For a lot of image recognition tasks, training time can be in the hours, which though costly is crazy to run a grid search over the course of a week. But when it comes to sequence learning tasks such as language modelling training time be in the magnitude of weeks. The same is true of reinforcement learning where training can also takes weeks at a time. Another use case is when it comes to running on big data. In my professional career I work for a company called skimlinks which has access to many billions of records around online user behaviour. Each record could potentially be a several thousand dimensional vector, depending on how we encode it. Training until convergence even once, on such a dataset, would take a huge amount of time and be very costly. It might never be an option to run a grid search.

There are currently existing approaches, such as Bayesian optimization, that improve on grid search, but these also suffer from the problem having to re-run the network from scratch and only reduce the number of searches needed as shown in Jasper Snoek et al (2012). It would be desirable to have a method that could adjust its hyper parameters as part of training.

The hyper-parameters around the number of hidden layers and the hidden nodes per layer are together known as network topology or structure. The problems that can arise from a bad choice of network topology are:

1. If a network has too few hidden nodes per layer(also referred to as too narrow), it will have a high error rate for both training and test sets. As the network lacks the capacity to propagate useful signals forward through the network. This is shown in Whitley et al (1991).

2. If a network is too shallow, too few layers, it may find it hard to generalize learning. Also for some simple datasets it may be simply incapable of learning the patterns. For example the XOR pattern requires at least 1 hidden layer and even then may not learn consistently. The two spirals problem, shown in Figure 1, requires at least 2 hidden layers, with 5 nodes each to be learned.

1Tieleman & Hinton (2012)
2Kingma et al (2014)
3Bergstra & Bengio (2012)

3. Too many hidden nodes per layer can result in the network having poor generalization performance. This is also known as overfitting. Computing nodes that are not needed is a waste of cpu/gpu. In the paper Yann le cun et al, Optimal brain damage 1990, show a technique for pruning unneeded hidden nodes from a network. This technique was found to *improve* the accuracy of the network on a test set. Accuracy on the train set did not change much.

4. Too many hidden layers, while having more hidden layers increases the ability to learn complex translation invariant patterns, at certain point more hidden layers results in a worse error rate even on the train set as shown in Srivastava et al Highway networks (2015). Unneeded hidden layers are also a waste of cpu/gpu cycles.

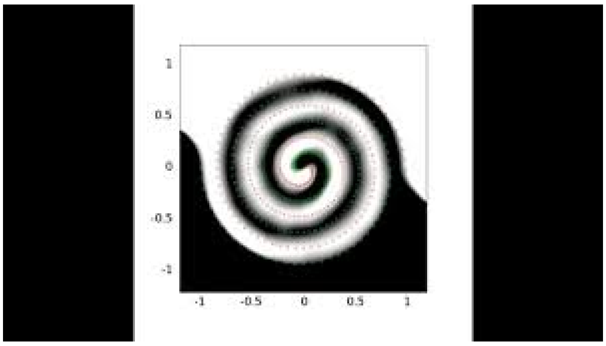


Figure 1.1: Two spirals problem

<https://i.ytimg.com/vi/Y--elMbxVmg/maxresdefault.jpg>

1.1 Aims and objective

The aim of this project is to create a software library that supports adapting the structure and topology of a neural networks and derive a flexible approach to determining the hyper-parameters around the number of layers and the hidden nodes per layer as a part of training.

To achieve this aim I will:

1. Conduct a thorough review of current research papers in this area.
2. Build a software library that better supports functionality around resizing neural networks. The library will include functions to dynamically adapt the network, so that if its structure is too restrictive it increases the number of nodes or layers, and if the structure is unnecessarily complex it can start to remove nodes or layers to simplify itself.
3. Experiment with a mixture of existing and new approaches to achieve the desired goal. To this end, an ideal technique should find these parameters as a part of training, saving the time of having to continually re-run the network from scratch.

1.2 Methodology

1.2.1 Requirements

With these aims a criteria for success evolves naturally. A successful technique should have the following properties. When initialized with a bad choice of nodes and layers for the dataset it is being trained on, it should increase or decrease in size, to finish with an error rate close to that the optimum.

As part of this project the software library must be able to support a neural network resizing. This in some ways a more significant piece of work than the algorithm itself as currently no such libraries exist. The library must support changing numbers of hidden nodes and layers in a network as part of training. It should do so in an intuitive easy to use manor.

1.2.2 Design

I will take an iterative or agile approach to building the library, see Agilemethodology.org. (2015). Rather than trying to precisely define all the requirements classes and functionality before starting, instead the agile approach prescribes building features in a direct step by step manner. Which each step of adding features I will have a better idea of what does and doesn’t work for my requirements and so modify the design as I go. Once a critical mass of features have been reached then I will make decision about how best to refactor the improve the design of the library from the current state.

I will build the library in Python. Python is chosen because it is very well supported as a data science and in particular deep learning environment. It has good trade off between the ease of use of a managed language such as R while having some of performance and portability benefits of lower level languages like C++ or Java. I will attempt to follow standard best practice in building a Python software library. All method and classes that are the interface to the library should be fully documented, including the type of every parameter. I will be using google style docstrings, see https://google.github.io/styleguide/pyguide.html.

I will aim to have reasonable test coverage for all aspects of the library that our appropriate. In general every module should have at least one significant test. I will not attempt to build a deep learning library from scratch, as many successful complete ones exists. So I will instead look into existing libraries and look to build on top of one of them.

1.2.3 Validation

The library functionality will be validated through the use of robust set of unit tests. All functionality will have at least one front to back test associated with it. It appropriate the test will include the training step for the network as well. Software development best practice suggest all tests should run in sub second time, but I will relax this prescription in my library to allow for more real world like tests. Some tests with even involve training against a reduced mnist dataset.

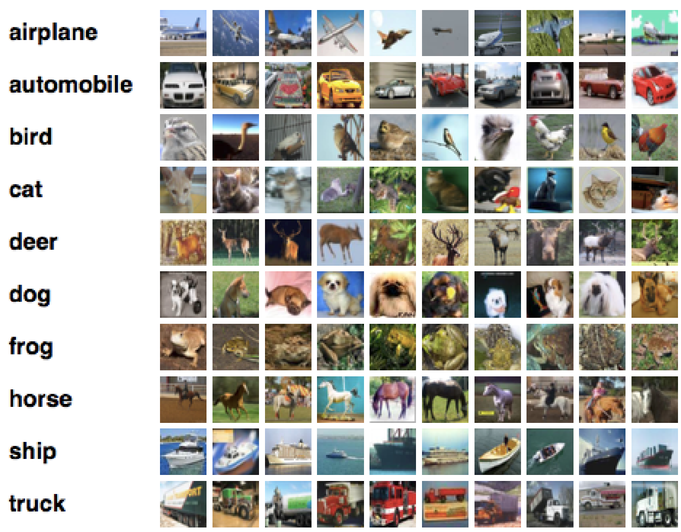
The optimum topology for a neural network can be found by running a grid search across all possible hyper parameters and selecting the parameters with the best performance against the test set of the data set. For the new technique to be worth while it must complete in a faster time it takes to run such a grid search.

1.2.4 Testing

For comparison of this technique against grid search using popular real world datasets, that are complex and nonlinear. The MNIST, contains a collection of 70,000 handwritten digits, each of a number from 0 to 9. The features are a black and white image of the handwritten digit, of dimensions 28 by 28. This dataset though much more complex than the toy data sets, still has limitations. Because most of the images are well centered and a part of a set with few possibilities(how many different ways can you write a 1), it can be solved with a limited topology. Accuracy against a test set of 98% can be achieved with only a single hidden layer containing 100 nodes. We will still bench mark against this dataset, but must bare in mind its simplicity.

For an even more complex dataset we use CIFAR-100. This is a collection of images of 100 different categories of object. For example leopard, cloud, tiger, sea, castle, house, road, etc. Each image is 32 by 32 in size and has 3 channels of color. On this dataset the best result in the world, at time of written is just 75% as shown in Clevert et al (2015). This result was achieved using a network with 15 convolutional layers. See figure 2 for some examples from CIFAR.

A good solution to working out network topology should show reasonable performance across all these



dataset.

Figure 1.1 : Examples from the CIFAR-100 image dataset <https://thkimorgblog.files.wordpress.com/2016/03/e18489e185b3e1848fe185b3e18485e185b5e186abe18489e185a3e186ba-2016-03-12-e1848be185a9e1848ce185a5e186ab-1-02-16.png?w=764>

There are currently no deep learning libraries that support dynamically sizing neural network as part of training. This will necessitate the writing of a new library in order to do verification of any methods. TensorFlow is a performant, well supported, open source computational graph library that has very proven very popular with the deep learning community. I will build a new library on top this, that supports the new features required. The testing, comparisons will be made between performance with this new library using the new dynamic topology learning and using the library in the standard way for grid search.

1.4 Organization of thesis

This thesis will show the development and validation of the Tensordynmaic library. It will also cover other attempts to learn neural network structure through training and present an approach using the Tensordynamic library that appears to have some advantages over grid search. The organization of this thesis will be:

In chapter 2 we review this existing literature. We will start with a recap of the background neural networks and deep learning. We will then look into the existing approaches to learning network topology. Then focus on learning topology as part of training. We will also look at approaches to dealing with the problem of overfitting which will become important later on in the results section. This will help us better understand what features our library will need to support and what areas we might look at to build new algorithms.

In chapter 3 we will look in more depth at the design of the library. What kinds of interfaces might be appropriate and create a plan for how to build a library that can provide the functionality we need. To gain knowledge for this we will also look at interfaces choices for existing deep learning libraries. The chapter concludes with a broad outline of how development of the library will proceed.

In chapter 4 we look at algorithms for learning the network topology as part of training. It covers some early approaches I attempted and some of the reasons they weren't successful. It then concludes with a more promising approach I eventually used, built on top of the Tensordynamic library that proved more successful and will be more completely evaluated in the results chapter.

In chapter 5 we go into much more detail on the eventual implementation of the Tensordynamic library. Here we present annotated code samples from the library and discuss choices in implementation in Python.

In chapter 6 we evaluate how the chosen algorithm described in chapter running in Tensordynamic library. We compare the results achieved to the results when running on grid search. We show that this method has some advantages over grid search in terms of results test accuracy and speed of evaluation, but also suffers from some issues with consistency of performance.

In chapter 7 we summarize the results achieved. Talk about where the research fits we other work in the field and finally discuss interesting ways the work could be extended in the future. In particular applying this approach in recurrent neural networks and reinforcement learning.

1.5 My research contribution

My research contribution will be two fold. First, the development of an open source library for dynamically resizing deep neural network at runtime. Second, an algorithm that can be run on top of this library that allows a neural network to learn the best available(a near optimal)topology faster than applying grid search.

Chapter 2 – Literature review

2.1 What is a neural network

Feed forward neural networks aim to be able to *learn* a set of parameters that can be used to generate a set of outputs from a set of inputs from a data set. They consist of a set of layers each of which contains a set of nodes. Every node in layer k has a connection to every node in layer $k+1$. Connections have a weight W_{ij} associated with them, which is the strength of the connection from node i in layer k to node j in layer $k+1$.

The network is activated on a piece of data one layer at a time. First the input layer is set to the value of that data. Then the next node layer is activated based on the equation

$$l_{k+1}=\phi(l_{k+1}W_{k+1}+b_{k+1})$$

Where l_k is the vector of weights in layer k, W_{k+1} is the matrix of connection weights between layers k and $k+1$, b_{k+1} is a vector of biases for the nodes in layer $k+1$, and ϕ is a function applied to each element of the vector.

The bias b_{k+1} can also be represented as an additional row in the the weight matrix with the input always set to one. This allows us to simplify the equations and language(weights and biases can simply be described as weights) of the network and so will be used from here on.

This function ϕ is known as a non linearity or activation function. It is used to stop each layer simply being a linear combination of weights in the previous layer and we want our network to be able to represent more complex patterns. The sigmoid function is often used $\frac{1}{1+e^{-x}}$ though many other continuous differentiable functions have been shown to work well.

Once all the layers have been activated the activation of the final layer, known as the output layer is considered the output of the network. In order to have parameters for our weights that produce a meaningful output we must train the network on a data set, in this respect there are two classes of network.

- Supervised network are trained on datasets that contains both input data items and also a targets associated with each item of data. Through training the network aims to find a set of weights that result in the smallest difference between the network's output and the targets.
- Unsupervised networks aim to learn the data. There output should be similar for similar pieces of data and different for more distinct data.

This thesis will be mostly looking at supervised learning. In supervised learning the difference of the network output to the target is specified by an error function(also known as loss or cost function). A common error function used is the squared error, this is the error for a single item from the dataset.

$$e=\sum_{i}\frac{1}{2}(t_i-y_i)^2$$

Where t is the target vector y is the vector of the activation of the network. The times by half is there simply so when we calculate the derivative of this function it simplifies to

$$\frac{de}{dy}=t-y$$

Training can also be done in batches by taking the squared error across collections of data items:

$$e=\sum_{j}\left(\sum_{i}\frac{1}{2}(t_{ji}-y_{ji})^2\right)$$

Where t and y are now collections of I samples.

Backpropagation is then used to adjust the weights to minimize the sum of this error. This involves finding the partial derivative of the a given weight with respect to the error function. This can be calculated using the chain rule $\frac{de}{dw}=\frac{de}{dy}\frac{dy}{df}\frac{df}{dw}$ where df is the derivative of the activation function. So for the squared error function above with the sigmoid activation function it would be

$$\frac{de}{dw}=-(t-y)y^2(1-y)$$

The weights are then updated by some delta to follow the negative of the gradient direction. This should result in a decrease of the error over successive iterations.

$$W_{t+1}=W_t-\alpha\frac{de}{dW_t}$$

Here W_t refers to the full weights at time t not the weights in layer t . α is a constant parameter to known as the learning rate.

At first feed forward neural networks were only 3 layers. 1 input, 1 hidden and 1 output. Additional layers could be used but were not found to be helpful. But as weight-initialization, training techniques and computational power have improved multi-layer networks are being found to produce superior performance in many tasks such as image and speech recognition. The term deep learning is applied to networks trained with many layers.

2.1.1 Over-fitting

Because of the high level of complexity and non-linearity deep neural neural network, they can have a big problem with overfitting the data. The first solution to this problem is called early stopping, the data is split into a training set and test set. During training, the training set is used, but after each complete iteration through the training set, the loss is calculated against the test set. When the network is overfitting this will start to go up, even though the training loss is going down the network is setup to stop training once it sees a certain number of iterations without improvement in the error on the test set.

This still leaves a slight problem. Say we had a dataset where the targets were pure random numbers, completely unrelated to the features. If the we were to run many different networks to tune the hyper-parameters, we would expect some of the network out of pure random chance to perform much better on the test set. Giving the false impression of a relationship.

Because of this if early stopping is used the best practice is to divide the data into a training, validation and test set. Early stopping is checked against the validation set. The test is only checked once training is fully completed. This keeps the test set as a pure measure of performance, free from bias. Dealing with overfitting will be something that will need to be watched very closely in this project. In general adding more nodes and layers will reduce error, but this may just be overfitting. If trying out lots of different models, it may just be chance that some model come out ahead against the validation set.

2.1.2 Regularization

Another way to deal with overfitting is to add a penalty for large weights. This is known as regularization. This penalty is added to the loss function of the network. So when getting the gradients of the weights with respect to the loss function, it will tend to pull the weights towards zero.

There are a few different kinds of regularization, the main are two called L1 and L2. In l1 the sum of the weights times some constant value is added to the loss function. In L2 the sum of the square of the weights is used. L2 has tended to be more popular and have better results, in part because L1 can move weights to 0, making them completely ineffective. This may be a desirable property for this project as if all the weights for a node become 0 the node can be safely removed. All of these as well as soft weight sharing are written about in (Nowlan and Hinton, 1992).

2.1.3 Gaussian Noise

This is another approach to improving overfitting. It involves adding gaussian noise to the network input, or to the input to each layer of the network. The gaussian noise always has mean 0, with the standard deviation being in the same range or smaller to the input to the layer.

Because of the noise in each input node, no individual node can be relied upon to provide a good signals from which to predict the targets. But because the sum of many gaussian values will tend towards the true mean, 0, the hidden nodes will tend to learn to take a range of nodes as inputs. This has been shown to reduce overfitting because individual values of particular input nodes are less significant to the network. So it learns patterns not values. This is written about in Yinan et al, (2016)

2.1.4 Dropout

First introduced by Hinton et al (2014) this is another technique for improving overfitting. In dropout inputs nodes are reduced to 0 during training with some random probability x . As with gaussian noise this forces the network to learn general patterns instead of the signals of individual inputs.

Because a percentage of the signals from the network are omitted when training this means when predicting we will have a different distribution. To compensate for this during training the inputs that are kept are scaled by $\frac{1}{x}$ resulting in the training and output distribution having the same variance. Dropout is normally applied to the input layer, but can also be applied before the hidden and output layers. This is shown to further improve generalization ability.

A difference between dropout and gaussian noise is that because an input node is actually set to 0, it is turned off for a given training of the network. Yarin Gal (2015) shows that dropout is actually equivalent to approximate inference in Bayesian neural networks. Another way of thinking about dropout is that because during each activation of the network we are removing the effect of the some sub-set of the nodes, each activation is it’s own sub-network. Dropout trains a large collection of these sub-networks so when it comes to prediction where we run all the nodes, this the equivalent to running an ensemble of networks.

2.2 Batch Normalization

First introduced in by Sergey Ioffe et al (2015), batch normalization was shown to generally improve both the number of iterations required to reach convergence, and the accuracy across the whole length of training for deep neural networks.

Figure 2.1 : An example of performance when using Batch normalization vs normal mini-batching.

<https://shuuki4.files.wordpress.com/2016/01/bn3.png?w=1000>

Before batch normalization was introduced it was already established that normalizing the input data to have mean 0 and unit variance improved the speed and quality of training. In batch normalization normalization of input is applied not just across the whole data set, but per mini-batch, and on the input to each layer of the network.

The reason this helps is because of an effect called covariate shift. Any learning system is heavily affected by the distribution of its input. When it comes to a deep neural network, each layer can be seen as a learning system. Over the course of training the deep layers inputs will be changing significantly as the layers that feed into them change their activation. A Lot of the energy of the deeper layers simply goes into learning to adjust to the new pattern of the previous layer. Batch normalization reduces this problem.

Another problem that neural networks have is called saturation. For activation functions like sigmoid there is a limit on how much signal a node can send to the next layer. In the case of sigmoid the range is only between 0 and 1. This makes it impossible for the network to increase the signal it gets from one of these nodes, once it is already close to maximum. This is one of the reasons that the relu activation function is so successful.

Because of the normalization, Batch normalization could lead to saturation becoming a problem. That is why in addition to normalizing Batch normalization add two new variables, that modify the scale and translation of each input node. This means a layer can learn to set the range of any input value to whatever range is most useful. The final equation for batch normalization looks like this.

$$y_k = \frac{x_k - E(x_k)}{Var(x_k)} \gamma_k + \beta_k$$

Where x_k is the *kth* value passed to the layer, as received from the previous layer. y_k Is the value of the *kth* value passed to the layer after batch normalization. This will then be passed into the weight matrix. $E(x_k)$ Is the mean of the *kth* input value across the mini-batch. $Var(x_k)$ Is the variance of the kth element across the mini-batch. γ_k and β_k are the two trainable parameters for scale and transform of the *kth* element.

2.3 Existing approaches

We have covered the background related to deep neural networks. For this project we are interested in the problem of the determining optimum topology for the network. The existing approaches to learning optimal network topology can be roughly grouped into 4 different categories.

1. Using optimization techniques to re-run multiple networks to find the best number of hyper parameters.
2. Genetic algorithms
3. Pruning networks
4. Growing networks

These will be discussed in order. This project will in particular focus on the growing approaches.

2.3.1 Hyper parameter optimization

Determining network topology is something of a dark art. No good formal method has been found for determining the best numbers of layers and nodes, though there are a number of rules of thumb that exist as described by (Saurabh Karsoliya, 2012):

- The number of output and input nodes is exactly determined by the dimensions of the data and the transformation the task requires.
- If data is linearly separable then no hidden layers are required and thus the number of hidden nodes is 0.
- One hidden layer is enough to learn datasets of lower or medium difficulty. As for the number of hidden nodes there are multiple rules of thumb. It should be:
  - Somewhere between the number of inputs and outputs.
  - The number of inputs * $\frac{2}{3}$
  - It should never be larger than twice the number of inputs.
- Additional hidden layers beyond the first have been shown to increase the accuracy at the expense of increasing the risk of bad local minima. It is recommend to use equal numbers of neurons in the first few hidden layers to reduce the local minima problem

The standard approach data scientists use when actually building a network tends to be something close to trial and error. Varieties of configurations are tried out on training sets and their performance against a test set is compared until a good configuration is found. A slightly more systematic approach was tried by (Jasper Snoek et al, 2012) using bayesian optimization to efficiently search the space of hyper parameters. Genetic algorithms and particle swarm optimization, or any number of algorithms for searching through a large problem space can also be used.

In deep learning the typical approach as stated by (Hinton et al, 2006) is that many unsupervised layers are trained on a dataset sequentially before adding a final supervised layer to produce the prediction. Here it is advised to just keep adding successive layers until you start to see a drop off in validation performance.

There are 2 big downsides to all these approaches, one is that training a neural network can be very slow. (Alex Krizhevsky et al, 2012) state that the training time for their convolutional image classification network was between 5 and 6 days. When talking about time frames of this scale doing multiple runs with different configurations of hyper parameters is not practical. The other downside is that if your neural network is running in an online scenario, where you expect it to react to learn new and novel data over time then the correct number of hidden nodes for initial learning may be very different from those it eventually finds itself dealing with. For these reasons approaches that allow neural network to dynamically adjust the number of nodes and layers they have during training are very desirable.

2.3.2 Genetic algorithms

These can be used simply as a technique for finding the best hyper parameters, in which they would fall into the category of using optimization techniques for hyper parameters search, or they can also be used as a way of determining the actual weights of the network, instead of backprop.

NEAT developed by (Kenneth O. Stanley, 2004) is the most popular example of this. NEAT does not build its network in layers but instead has completely free form configurations of connections. Any node can potentially evolve to be connected to any other node, as long as it doesn't end up with recursive connections(RNN versions also exist). NEAT has 3 unique features that make it different from other approaches.

- Complexification - Starting evolution with small networks that are able to evolve increasingly complex topology.
- Keeping track of which genes match up with others to allow evolutions of topology to be more meaningful.
- Speciating the population - different sets of the population are divided to allow some mutations that may start off leading to an evolutionary disadvantage time to improve. This is needed because most topological mutations will initially reduce fitness so time is needed for these approaches to adjust out of their local minima.

Because there is no backpropagation or training a network is only evaluated by running it once on the dataset and seeing the cost. This being orders of magnitude faster than first training with backpropagation means many individuals can be simulated to eventually find good topology and weights.

NEAT has been shown in experiments to perform very well in tasks such as agents learning to navigate their environment where backprop performs very badly. Unfortunately it performs less well in tasks such as imagine recognition and other tasks where the search space is possibly just too large. For this reason it would be preferable to have technique that could use the benefits of backpropagation while still being able to adapt topology.

2.3.3 Pruning approaches

This involves starting with a large network that we hope has the capacity to learn the data. Then once it has been fully trained we try removing the least useful nodes. The pruning approach has proved to be very successful at improving generalization performance and has been used by the winners of a number AI competitions. Though it should allow a network to find a more optimal number of hidden nodes the pruning approach doesn't have much to say about the number of layers.

The main methods used in the pruning approach are:

Optimal brain damage

Published by (Yann le Cun, 1990) this involves finding which weights if set to 0 would have least impact on the the network's error. The effect on the error is referred to as 'saliency'. The effect of a weight on the error can be calculated by a taylor series that involves a hessian matrix of how each weight affects every other weight. This was considered too computational costly to do in full(for just 1000 weights the hessian would be 10^6 parameters) so a number of simplifying assumptions are made. The equation for the impact of a change in a node on the error is:

$$\delta E = \frac{1}{2}\sum_{i} h_{ii}\,\delta u_i^2$$

Where δE is the change in the error function. i goes through every parameter in the node, which is normally every outgoing weight plus the bias term. δu_i is the change in the parameter ith parameter required to remove it from the network, which is it's value. h is the Hessian matrix with respect to with respect to the parameter. This is calculated by

$$w_{li}^2 \frac{d^2 E}{d a_l^2} + f''(a_i)\frac{dE}{d x_{ii}} f'(a_i)^2 \sum_{l} x_j^2 h_{kk} = \sum_{(i,j)\in V_k}$$

Where V is the set of all pairs of parameters in the network, $a_i = \sum_j w_{ij} x_j$ and $x_i = f(a_i)$ and f is the activation function.

In his paper Yann le Cun trained a network on a dataset of 9300 training and 3350 test example of handwritten digits. An MLP network with 2578 free parameters was trained until the error stopped reducing. It was found that up to 30% of parameters from the network could be removed without a significant increase in error and if the network was retrained after pruning as much as 60% of weights could be pruned without significant reduction in error.

Optimal brain surgeon

By (Hassibi et al, 1994) proposes a variant of optimal brain damage that uses fewer approximations to compute the saliency more accurately but at the cost of a much longer computation time. In (Hassibi et al, 1994) it was shown through experiments that optimal brain surgeon outperformed optimal brain damage in generalization. But it is worth noting that the testing was done on very small networks(sub 100 nodes) in part because optimal brain surgeon is so slow.

Tri-State ReLUs

In contrast to the "optimal brain" techniques, rather than learning the net and then pruning, (Srinivas et al, 2015) comes up with a way of specifying the network complexity as a regularizer in the error function so that pruning is done as part of training. This means reduced network complexity is encouraged but is traded off against the error of the network as it is training.

The activation function used by the networks is a Tri-State Relu

if $i > 0$

$$y = wi$$

else

$$y = wid$$

Where w is the width parameter, d is the depth parameter and i is the input to the function.

The width parameter is per hidden node and is a trainable binary variable. If in training it ends up at 0 then the node has no impact on the calculation and can be pruned. The depth parameter is also binary and shared by all nodes in a layer. If depth is 0 then the activation is a standard relu. But if it gets set to 1 then it becomes the identity function meaning that this layer is just a linear combination of previous activations and so can be easily removed by combining it with the next layer. The width and depth parameters are both trained with the function x(1-x), which is know as a binarizing regularizer and can be shown to push values to either 0 or 1 over the long term.

Experiments were run using this technique on a convolutional network running on the MNIST dataset. The standard LeNet style network with an architecture of 20-50-500-10 had an accuracy of 99.07 while a network trained to optimize the width of the final hidden layer had the same accuracy but with a final architecture of 20-50-61-10. Training of the depth parameter was seen to reduce accuracy more significantly.

This technique is very interesting, but the major downside is that a network must be created at it's max capacity and can only be downsized from there. Also while we are training we have greater than 2 times the number of parameters we need to train and the full capacity must always be trained. For the purposes of this project we would like a technique that allows the network that can grow as well.

2.3.4 Growing approaches

In this we start with a minimal network and gradually increase the size. Adding nodes in response to the error encountered from the data. Hypothetically the growing approach should be faster to train than pruning because in each iteration you are having to compute fewer nodes. But in practice it may take much longer to grow the structure and a lot of training time can be wasted as new additions of nodes make previous learned patterns obsolete.

Cascading neural networks

These were proposed by (Fahlman, 1990). The approach can be summarised as follows:

1. Start with fully connected a network of input and output nodes, with no hidden nodes.
2. Train the network using standard backprop until the error rate stops decreasing.
3. A number of candidate nodes are then created and trained to maximize correlation with the remaining error of the network.

   $$C=\sum_{o}\{\sum_{p}y_{po}(e_{po}-\underline{e_o})\}$$

   Where y_{po} is the output of the network for data point p and output node o and e_{po} is the error of the network as defined by the objective function against data point p for output node o

   The weights for the candidate are then updated via backprop until the candidates correlations with the error stop increasing.
4. The candidate with the best correlation is then selected and added to the network gaining a connection to each output node. The connections from the input to the candidate are frozen but the connections from the input to output and candidate to output are again trained until the error rate stops decreasing.
5. Return to 3 this time the new candidates will have a connection to the previously created candidates. Repeat this until the error rate drops below a threshold or a maximum number of iterations is reached.

Cascade correlation is one of only a few networks that can solve the two spirals problem and in this it has a much better performance than standard MLP. It also is capable of learning the xor and double xor patterns. The problems with cascade correlation are that it tends to badly over fit data and does not parallelize well. This has resulted in it not being widely used.

It has been extend in a few interesting ways, in A Learning Algorithm for Evolving Cascade Neural Networks evolutionary algorithms were used to train the network which was shown to result in faster learning with fewer nodes than through backprop. Fahlman also proposed a variant called Sibling/descendant cascade-correlation which reduced the depth of the networks by giving the option for the newest candidate to be the sibling rather than the descendant of the most recent candidate, if it scored comparatively well in that position.

Dynamic node creation in backpropagation networks (Ash, 1989)
This looks at attempting to find a good number of hidden nodes for a network by starting small and progressively adding single hidden nodes.

1. A one hidden layer network is created with predefined number of hidden nodes.
2. The network is trained, when the error rate stops decreasing over a number of iterations a new hidden node is added.
3. Nodes stop being added either when a certain precision is reached or when the error starts increasing on a validation set

According to their experiments training the network and then freezing existing nodes then adding new unfrozen nodes does not work well. Adding new nodes then retraining the whole network is better. The results of the experiments in the paper show that growing a network according to this technique takes not a significantly longer amount of time than fixed networks and finds near optimal solutions. The remaining questions they want to investigate is how big the new node adding window should be and where nodes should be added in multi layer networks?

Infinite Restricted Boltzmann Machine

By (Alexandre Côté et al, 2015) this is a way to learn the best number of hidden nodes in a restricted Boltzmann machine(RBM). The idea behind it is based on gibbs sampling to choose a number of hidden nodes , z , based on the distribution of visible nodes. Normally left to it's own devices sampling from z you would expect to eventually get to the point where z = the number of visible nodes simply because the identity function is the way to represent the distribution with 0 loss. In order to stop this happening a penalty is applied to the bias of each additional hidden node, that increases as the number of nodes increases.

$$\beta_i = \beta \ln(1+e^{b_i})$$

Where b_i is the bias of the ith hidden unit and β is a global hyper parameter. The penalty β_i is parameterized by the hidden node bias so that it can't simply be compensated for by that. In this way there is a cost to select additional hidden nodes so higher values of z will only be selected if they lead to real improvement in the negative log likelihood.
Here is an example of how it works:

1. Activate the input nodes from some data. Our aim in learning with an RBM is to be able to accurately sample for the distribution of the data. To put it another way, when we turn on our RBM and say give me something random, what it gives us should be indistinguishable from the data set we trained it on.
2. Sample the number of hidden nodes, **z**, from the visible layer:
    1. Get the activation of the hidden_layer: activation = sigmoid(weights*input_nodes+bias)
    2. Apply an energy penalty for each node to discourage successive nodes from being used, the penalty is parameterized by the bias, so it cannot just be compensated for by increasing the bias: energy=activation-softplus(bias)
    3. Factor in the energy of the infinite number of hidden nodes that we could have had: energy = energy.append(log(1-1/softplus(0)))
    4. sample **z** from the energy: **z** = multinomial_sample(exp(energy - sum(energy)))
3. Activate **z** hidden nodes based like a standard feed forward neural network: hidden_nodes[:z] = sigmoid(weights[:z]*input_nodes+bias[:z])

4. Reactivate the inputs_nodes: input_nodes = weights[:z].transpose*hidden_nodes[:z]+back_bias
5. Repeat until the reconstruction error stops decreasing

In the experiments they did the IRBM was shown to have similar performance to a normal RBM and the training time was not significantly longer. This technique has nothing to say about the depth of a network but this approach could be extended to sample the depth as part of the gibbs sampling.

Net2Net: ACCELERATING LEARNING VIA KNOWLEDGE TRANSFER

This thesis by (Chen et al, 2015) is not actually about determining network size, but is included because the technique they use for transferring the trained weights of one net to that of a different size are could easily be used in the growing network approach.
Two techniques are described in the paper.

Net2WiderNet takes a layer of a given size and creates a copy of it with an increased number of nodes that should have close to identical activation of the nodes in the next layer. For every node that has to be added one of the old node is selected randomly and it's weights are used. There should now be 2 copies of this same old node in the new network so the outbound weights of both of these nodes are halved and varied by a small amount of noise.

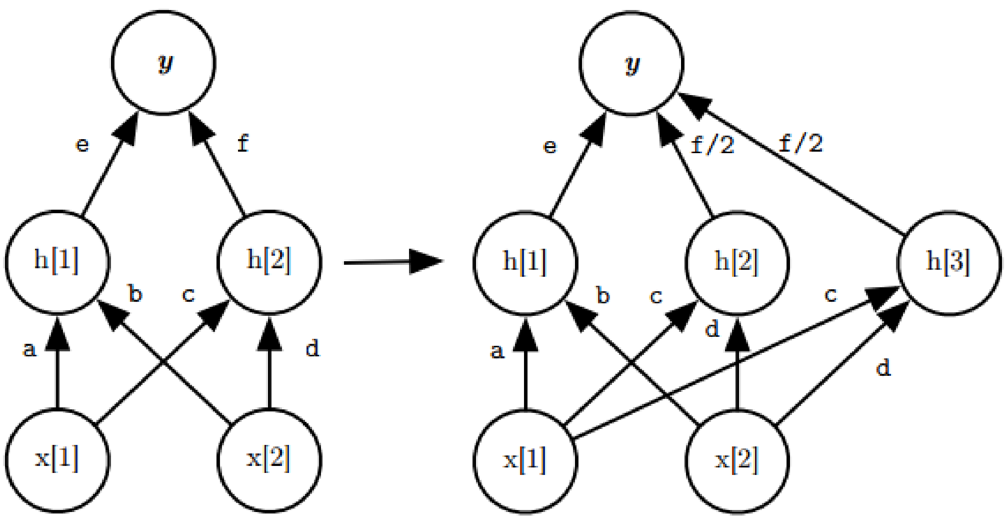


Figure 2.2 : Net 2 wider net example
source: Net2Net, 2015, Tianqi Chen, page 3

Net2DeeperNet takes a single layer and transforms it into 2 layers that have the same activation function as the single layer. This is done by having the newly created layer simply be the identity matrix. Some noise can also be added here to increase divergence.

There are 5 factors that we are interested in for our techniques.

1. Grows width is the ability to increase the number of nodes in a layer.
2. Grows depth is the ability to increase the number of layers in a network.
3. Prunes width is the ability to decrease the number of nodes in a layer.
4. Prunes depth is the ability to decrease the number of layers in a network.
5. Topology determined during training, is weather the network sizing is a part a of training or simply a separate activity. NEAT is set as No for this because the network is never trained as such but simply learning through trial and error on permutations.

The aim of the project is find a technique that can tick all of these boxes.

Technique	Grows width	Grows depth	Prunes width	Prunes depth	Topology determined during training
Cascade correlation	Yes	Yes	No	No	Yes
Dynamic node creation in backprop networks	Yes	No	No	No	Yes
Infinite RBM	Yes	No	Yes	No	Yes
Optimal brain damage	No	No	Yes	No	No
NEAT	Yes	Yes	Yes	Yes	No
Tri-State-Relu	No	No	Yes	Partially	Yes
Net2Net	Yes	Yes	No	No	No

Table 2.1 : List of techniques and their capabilities

Chapter 3 – Design of tensor dynamic python library

Given the lack of research into learning network topology through training, there is a complete lack of support for being able to change network topology during training in popular deep learning libraries. At the time of research I looked into keres, theano, tensor flow, lasagne, and scikit-learn but none of them supported anything like the functionality required. This necessitated the decision to build my own library.

In this chapter I go through the process of designing it and some of the implementation details. The completed library is available on github at https://github.com/DanielSlater/tensordynamic. It has over 6000 lines of code across more than 40 files and more than 100 unit tests. The current feature set includes:

- Build many layered neural networks
- Convolutional layers
- Train networks with a policy for changing structure as part of training
- Batch normalization
- Dropout
- Gaussian noise
- Built in loading of standard data sets such as CIFAR and MNIST
- Support for semi-supervised networks
- Ladder networks
- backwards as well as forwards activation
- Setting custom loss functions
- Custom function for splitting and pruning nodes
- Examples of approaches to building topology as part of training

3.1 Design

For the design of the requirements for the library where as followed:

1. The ability to add or remove nodes from a layer during training
2. The ability to add or remove layers during training
3. An interface for specifying an architecture modification policy to be used during training
4. Good flexibility for setting up signals that can be used as part of the policy.

Point 4 is quite significant. We may want our policy to be changing based many different aspects of the network, such as the second order gradients or the loss function with respect to nodes, or maybe the reconstruction error within layers. If we have the ability to specify policy but we are limited in what we may use as input to our policy it would be a severe limitation to what we can experiment with.

One option for the library would be to start building it from scratch. The appeal of this is being to able to start with a clean slate and have fun working on low level implementation details. But this idea was quickly rejected on the basis that it necessitated a most of the programming time going into building things already well supported and bug tested in many existing libraries. For this project I wanted to optimize as much as possible for experimentation time.

The existing neural network libraries can be roughly divided into two categories, those that provide a high level of abstraction, where the user can specify types of layers and hyper parameters for those. Lasagne and Keras are both examples of this. These libraries take a lot of the complexity away from the user to make usage a lot simpler. The trade off being the user is somewhat locked out from making low level changes to the behavior of their network.

The other category is the computational graph library, such as Theano and Tensor Flow. These both allow the user to set very low level details of how they're neural network is set up. For example operation by operation setting up the mathematical operations for running the network and how the network parameters are updated during training. For comparison here is an example of setting up a neural network with a single hidden layer in Lasagne:

```
import lasagne

l_in = lasagne.layers.InputLayer(shape=(None, 1, 28, 28))

l_hidden = lasagne.layers.DenseLayer(
        l_in, num_units=200,
        nonlinearity=lasagne.nonlinearities.rectify)

l_out = lasagne.layers.DenseLayer(
        l_hidden, num_units=10,
        nonlinearity=lasagne.nonlinearities.softmax)
```

Now here it is in Tensorflow:

```
import tensorflow as tf

input_placeholder = tf.placeholder(tf.float32, shape=(None, 784))

weight_hidden = tf.Variable(tf.truncated_normal((784, 200), stddev=0.1))
bias_hidden = tf.Variable(tf.constant(0.1, shape=(200)))

activation_hidden = tf.nn.relu(tf.matmul(input_placholder,
                                         weight_hidden) + bias_hidden)

weight_output = tf.Variable(tf.truncated_normal((200, 10), stddev=0.1))
bias_output = tf.Variable(tf.constant(0.1, shape=(10)))

activation_output = tf.matmul(activation_hidden, weight_output) + bias_output

target_placeholder = tf.placeholder(tf.float32, shape=[None, output_size])

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
                                labels=target_placeholder,
                                logits=activation_output))

train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

As you can see the Lasagne code is a lot more succinct and allows users to easily set up network by hiding a lot of the low level details, but a lot people still prefer Tensor Flow for the greater control it gives you over the setup of the network. For this project we chose Tensor Flow as the base on which to build the library, because in order to easily modify the number of layers and sizes of layers control of lower level functionality is required.

3.2 Architecture

I considered building something in Tensor Flow that did resizing applied to Tensor Flow variables. But often the conceptual node exists across a few different variables and even layers. To give an example, if we look at the code for creating a hidden layer in Tensor Flow:

```
weight_hidden = tf.Variable(tf.truncated_normal((784, 200), stddev=0.1))
bias_hidden = tf.Variable(tf.constant(0.1, shape=(200)))

activation_hidden = tf.nn.relu(tf.matmul(input_placholder,
                                         weight_hidden) + bias_hidden)
```

If we wanted to say remove a single node from this layer, that would require changes to all of these objects, first the weight_hidden variable would need to be resized, then the bias_hidden. The output shape of the tf.nn.relu operation would also have to be modified. Going further, we would also now need to adjust the properties for the objects consuming from activation_hidden and that adjustment would need to be consistent with the adjustment made in this layer. If we are remove the 13^{th} node, the next layer would also need to remove it 13^{th} input, not simply do a delete of the final input it has. This is all before we start thinking about more complex things, like convolutional layers, or ladder networks, etc. This suggests that in order to build resizing on Tensor Flow I would need to build in structured layers as first class citizens, that operate over the Tensor Flow operations that can then be used in consistent ways.

3.3 Interface design

A recommended way to start when building a new library, is to start with what the ideal method of usage looks like. The main objective in the design of a library, once the functional requirements are satisfied, is the ease of use. There may later turn out to be constraints around what can be done in an optimal way based on cpu and memory issues, but these can in general be hidden behind a well designed easy to use interface.

I began by coming up with some sample approaches to learning structure. I then did some playing around in python to find the simplest way to define an implementation for them. These use cases may be bad ideas and not work at all, but a good interface should allow me to try them all out with ease. Here are a list of possible experiments around size changing and what sample usage in python might look like. Each sample follows from the previous one, so contain the variables from all previous samples:

1. Build a standard flat neural network and train till convergence:

```
import tensorflow as tf

data_set = load_data_set()
input_layer = InputLayer(input_nodes=data_set.feature_shape)
hidden_layer = Layer(input=input_layer, hidden_nodes=200, non_liniarity=tf.nn.relu)
output_layer = CategoricalOutputLayer(output_nodes=data_set.label_shape)
output_layer.train_till_convergence(train_set=data_set.train, test_set=data_set.test)
print(output_layer.get_structure())
# should print out “Layer:200”
```

2. Manually resize a layer of the network:

```
hidden_layer.resize(250)
print(output_layer.get_structure())
# should print out “Layer:250”
```

3. Add a new layer

```
hidden_layer.add_intermediate_layer(Layer, hidden_nodes=200,
non_liniarity=tf.nn.sigmoid)
print(output_layer.get_structure())
# should print out “Layer:250 → Layer:200”
```

4. Learn the best network topology for a given data set

```
output_layer.learn_structure(train_set=data_set.train, test_set=data_set.test)
print(output_layer.get_structure())
# should print out something like: “Layer:242 → Layer:84 → Layer:65”
# depending on what is learned
```

5. Set a random policy of size structure changing

```
import random

def on_convergence(iteration_number, network):
	if iteration_number==200:
		return False # stop training
	layer = random.choice(network.get_resizable_layers())
	layer.resize(random.randint(50, 200))
	return True

output_layer.learn_structure(train_set=data_set.train, test_set=data_set.test,
				on_convergence_func=on_iteration)
```

6. Resize a layer based on a specific policy, such as optimal brain damage to give a score for how important each node is. If we want to resize down the lowest priority nodes will be removed. If we want to increase size, the highest priority nodes split.

```
def node_priority(layer):
	# TODO: add real logic here
	return [ random.randint(1, 100) for _ in range(len(layer.output_nodes))]

layer.resize(145, node_priority_func=node_priority)
```

All these cases give me a good interface for doing operations around learning network size. I built unit test, test cases into my project based on these specifications, over time everything would change, but these were good goals to keep in mind along the way.

3.4 Features

Here we present some of the additional features that the library should support and how the design was approached:

3.4.1 Bactivation

I wanted the library to support was the ability to support relationships between layers beyond the standard forward activation. Newer techniques like the ladder network, involve connection not just forward between layers but also backwards. As well as use forward activation between layers to train in a supervised fashion, a ladder network can also use a reconstructions error that is passed back down the network through multiple layers in order to learn in a unsupervised. It sounded an interesting to look at using this reconstruction signal as a measure of when a certain layer might be over or under capacity and require resizing. In order to support this I built a feature into the layers that I've called “bactivation”, like activation, but backwards. Any layer can optionally trigger a bactivation signal and if available pick it up from the next layer

3.4.2 Lazy prop

Having bactivation raised an interesting new challenge that needed to be addressed. When getting the activation for a layer let's say you have a getter method like so:

```
class MyLayer(BaseLayer):
   def get_activation(self):
      return tf.nn.relu(tf.matmul(self.input_layer.get_activation(),
                                              self._weights) + self._bias)
```

If we took this approach every time we called this method we would create a new set of operations on the Tensor Flow computational graph, which is needlessly inefficient. A simple way around this would be too simply set all these properties up in the constructor and have the method simply return the already created node. But the problem with that, is that now we support bactivation as well as activation we may only know exactly what our operations should be, once may future layers have been attached. This could be fixed by having an initialization method, but that may also have the problem that as layers getting added and removed as part of resizing we need a smart method for tracking changes. A nice solution is to use a python decorator called lazyprop. Lazyprop is a decorator is available as an example on line. For example through this https://pypi.python.org/pypi/lazy-property Pypi library. If it is applied to a function then that function when called the first time has it’s result saved and then saved value is retrieved on subsequent calls. This is discussed in greater detail in the implementation chapter.

Activation_train vs activation_predict

Another thing I wanted to support is the ability to use techniques for improving generalization performance such as dropout or adding gaussian random noise. What's notable about both of these techniques is that they require a different kind of activation when training as opposed to predicting.

This approach can be solved by having a boolean flag set on a layer that tells it which mode it is running in, train or predict. But in Tensor Flow this can be fiddly, here is an example of using dropout in Tensor Flow:

```
keep_prob = tf.placeholder(tf.float32)
dropout_activation = tf.nn.dropout(previous_activation, keep_prob)

# when training
sess.run(optimizer, feed_dict={input_placeholder: features,
                                          output_placeholder: labels,
                                          keep_prob: 0.5})

# when predicting
sess.run(optimizer, feed_dict={input_placeholder: features,
                                          output_placeholder: labels,
                                          keep_prob: 1.0})
```

As you can see this requires a variable be set to a different value depending on the mode we are running in. It also means a hypothetical dropout layer in my library would require defining this placeholder value if the user ever wanted to call any Tensor Flow operation downstream from it. To get around this annoying constraint and allow the design to have a more functional feel I instead took the approach of having every layer have to different activation operations, an activation_predict, and an activation_train. In this most layers activations will be identical, but for a layer which needs to run two different modes, it doesn't make any constraints on future layers behaviour. Have is a code extract from the NoisyInputLayer in my lib. In this report we name it, Tensordynamic:

```
class NoisyInputLayer(InputLayer):
    def __init__(self, input_nodes, session, noise_std, name='NoisyInput'):
        super(NoisyInputLayer, self).__init__(input_nodes, name)
        self._noise_std = noise_std
        self._session = session

    @lazyprop
    def activation_train(self):
        return self.activation + tf.random_normal(tf.shape(self._placeholder),
                                                  stddev=self._noise_std)

    @lazyprop
    def activation_predict(self):
        return self.activation
```

The same activation_predict and activation_train is also applied to the bactivation signal, in the form of bactivation_predict and bactivation_train properties.

3.4.3 Cloning networks

Another useful feature would be the ability to make copies of existing networks to be able to see what how two different modifications to a network might compare. Every layer in the network should contain a clone method, that create a copy of itself connected to a copy of the network leading up to it. In this way a clone on the output layer of the network would create a complete copy of it. But clones on lower layers would act as a nice interface for rebuilding new versions of later parts of the network.

Resizing a variable in Tensor Flow

Tensor Flow is not designed to support resizing of variables and operations once they are on its computational graph, but a bit of experimenting in the framework, showed me that you can assign Tensor Flow variables to have values of a different shape from the ones it was created with. For example the below sample does work:

```
with tf.Session() as session:
     var = tf.Variable(tf.zeros((1,)))
     change_shape_op = tf.assign(var, var, validate_shape=False)
     session.run(change_shape_op)
```

But it will still raise an exception if you attempt to use the variable as part of an operation because of the size mismatch. Luckily this exception is only in the python layer so with a bit more experimentation I found that this recipe allowed me to resize Tensor Flow variables and operations and keep the computational graph running:

```
def tf_resize(session, tensor, new_dimensions=None, new_values=None):
    if new_dimensions is None and new_values is not None:
        new_dimensions = new_values.shape

    if new_values is not None:
        if hasattr(new_values, '__call__'):
            new_values = new_values()

        assign = tf.assign(tensor, new_values, validate_shape=False)
```

```
        session.run(assign)
    elif isinstance(tensor, tf.Variable):
        current_vals = session.run(tensor)
        new_values = np.resize(current_vals, new_dimensions)
        assign = tf.assign(tensor, new_values, validate_shape=False)
        session.run(assign)

    if tuple(tensor.get_shape().as_list()) != new_dimensions:
        new_shape = TensorShape(new_dimensions)
        if hasattr(tensor, '_variable'):
            for i in range(len(new_dimensions)):
                tensor._variable._shape._dims[i]._value = new_dimensions[i]
                tensor._snapshot._shape._dims[i]._value = new_dimensions[i]
                tensor._initial_value._shape._dims[i]._value = new_dimensions[i]
        elif hasattr(tensor, '_shape'):
            for i in range(len(new_dimensions)):
                tensor._shape._dims[i]._value = new_dimensions[i]
        else:
            raise NotImplementedError('unrecognized type %s' % type(tensor))

        for output in tensor.op.outputs:
            output._shape = new_shape
```

This worked as a universal Tensor Flow resizing function, appearing to even work for things like weights in convolutional networks. But not for operations like resize and max_pool so these had to be handled by creating new computational graph elements from scratch.

3.4.4 Testing

Given the library was going to be large project, and the ultimate aim was for it to be open source, I was very rigorous about writing unit tests. Every class in the project has a test class that shadows it and every single layer is put through a range of standard tests checking, that cloning, resize, chaining and other aspects of its behaviour all work.

Also when it comes to the algorithm there a range of automated tests around expectations for how the error rate is affected by changes in network size. For example increase the size of layer by 10 nodes and then decreasing it back to the original size, should not significantly increase the error rate.

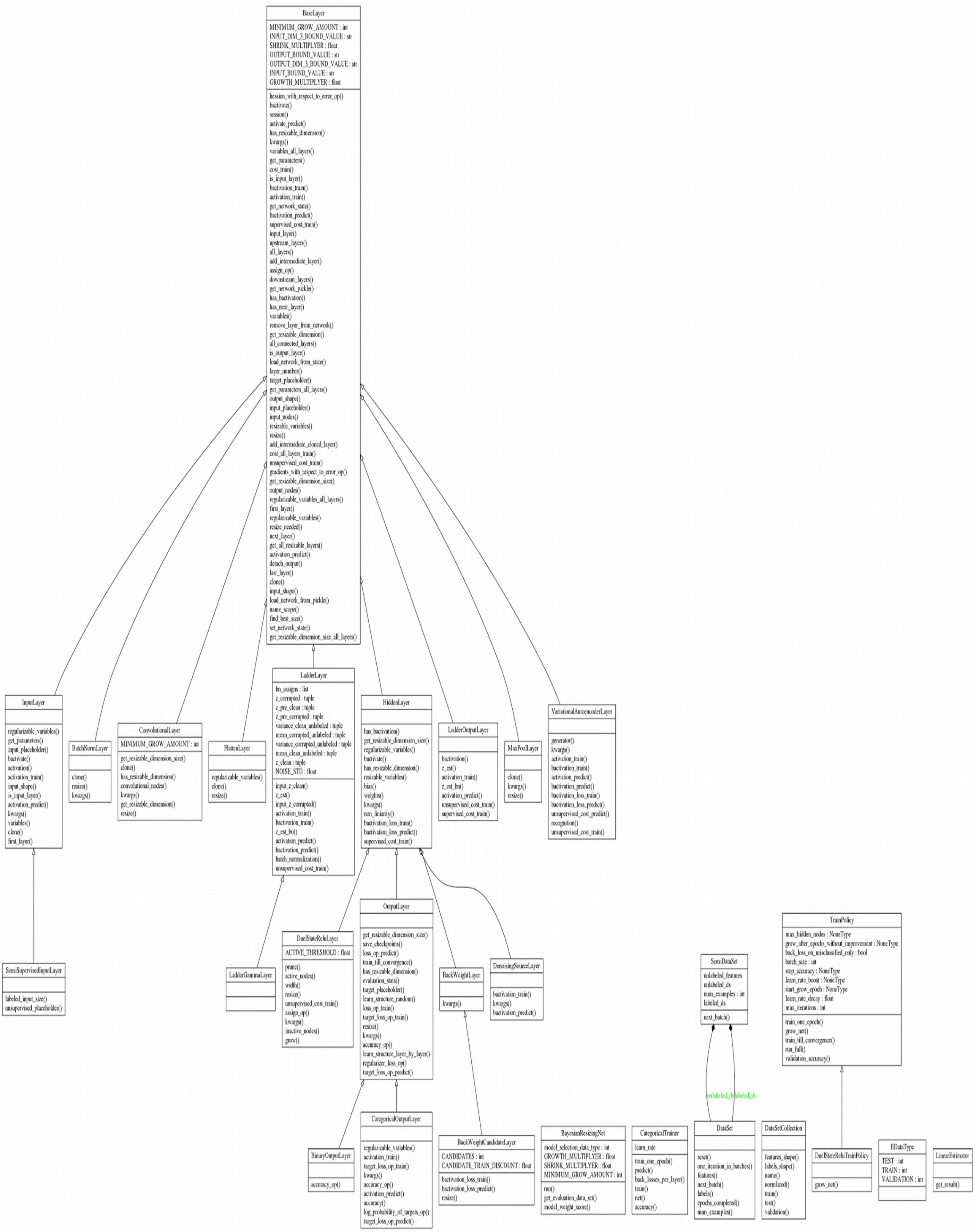
Classes

Figure 3.1 : Class diagram for the tensor dynamic project

Early in the project there was a much larger number of different support classes, but successive reworking eventually simplified things down to the point where most of the work is handled by a single class, the BaseLayer.

3.4.4 BaseLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/base_layer.py



This is the abstract base class from which all other layers inherit.Layers are always constructed with the layer it will come after in the network as the first argument. The BaseLayer contains the logic for connected layers together and navigating between layers in the network, with properties such as *all_connected_layers*, *next_layer* and *last_layer.*

Though the base layer does not itself contain any Tensor Flow variables, it contains methods so that layers classes that inherit from it can bind variables to either it's input or output dimensions. It also contains base the logic for resizing a layer. When a BaseLayer is resized it looks for bound variables in the attached layer to cascade a resize between layers that require it.

A common annoyance of building deep network is passing around lots of default variables that need to be shared between layers. The approach tensor dynamic takes to mitigating this problem is to have all layers look to previous layers for any variables they have not themselves been passed for example:

```
with tf.Session() as session:
    input_layer = Layer(784, session)
    hidden_layer_1 = HiddenLayer(input_layer, 256, non_linearity=tf.nn.relu)
    hidden_layer_2 = HiddenLayer(hiddden_layer_1, 256)
```

When the first hidden layer is constructed it requires a session variable to be passed, but not finding it, it looks to the layers it is attached to, see if they have the required variable. When hidden_layer_2 is constructed it also requires a non_linearity function, so will look to it’s input layer for this. To facilitate this functionality the constructor of the BaseLayer get sets its initial variables like so:

```
self._session = self._get_property_or_default(session, '_session', None)
```

Where session is a variable that is passed to the __init__ method. The _get_property_or_default look like:

```
def _get_property_or_default(self, init_value, property_name, default_value):
  if init_value is not None:
    return  init_value
  if self.input_layer is not None:
    if hasattr(self.input_layer, property_name) and
         getattr(self.input_layer, property_name) is not None:
      return getattr(self.input_layer, property_name)
    else:
      earlier_in_stream_result = \
        self.input_layer._get_property_or_default(init_value,
                                                  property_name,
                                                  default_value)
      if earlier_in_stream_result is not None:
        return earlier_in_stream_result

  return default_value
```

Also in the BaseLayer constructor is the setting up of the cross layer connections. The input is checked for being a valid layer and then assigned to class a variable, the base layer method _attach_layer is then called so the previous layer can have the ability to inspect it’s downstream layers:

```
def _attach_next_layer(self, layer):
  if self.has_next_layer:
    raise Exception(
"Can not attach_next_layer to Layer: %s which already has a next layer" %
```

```
                        self._name)
    if not isinstance(layer, BaseLayer):
        raise TypeError("Attached layer must be of type %s" % BaseLayer)

    self._next_layer = layer
```

This allows us to have helper methods such as last_layer which allows us to grab the output layer from any layer in the network.

```
    @property
    def last_layer(self):
        if self._next_layer is not None:
            return self._next_layer.last_layer
        return self
```

Full list of public methods and properties for BaseLayer are:

Method name	**Description**
name_scope	When creating new TensorFlow variables, this method gives a consistent name scope for variables within this layer.
activation_train	Returns the TensorFlow variable that is the output activation of this layer used during training. This will generally be the same as activation_predict but with added random elements, e.g. Dropout or gaussian noise.
activition_predict	Returns the TensorFlow variable that is the output activation of this layer used during prediction. This will generally be the same as activation_train but without random elements, e.g. Dropout or gaussian noise.
bactivate	Returns boolean for if this layer has a backwards activation in addition to its forward activation. Bactivation is used in Ladder networks.
is_input_layer	Returns True if this is an input layer to a network. This is currently true only for the InputLayer class.
is_output_layer	Returns True if this is the last layer in its connected network.
output_nodes	Returns a tuple of ints for the output nodes in this layer.
input_nodes	Returns a tuple of ints for the input nodes to this layer. Which will be equal to the output nodes of the previous layer.

session	The TensorFlow Session object within which this classes TensorFlow variables are initialized.
bactivation_train	If this layer also activates backwards then this is the TensorFlow variable for the backwards activation when training. This will generally be the same as bactivation_predict but with added random elements, e.g. Dropout or gaussian noise.
bactivation_predict	If this layer also activates backwards then this is the TensorFlow variable for the backwards activation when predicting. This will generally be the same as bactivation_train but with added random elements, e.g. Dropout or gaussian noise.
output_shape	The shape of the output Tensor from this layer, this will generally be the same as the output nodes with additional None as the first dimension.
input_shape	The shape of the input Tensor to this layer, this will generally be the same as the input nodes with additional None as the first dimension.
next_layer	The next layer in the network from this layer.
input_layer	The previous layer in the network to this layer.
has_next_layer	Returns True if this layer has a next layer.
last_layer	Returns the last connected layer in the network to this layer.
first_layer	Returns the first layer in the network connected to this layer.
input_placeholder	The TensorFlow placeholder variable for this network. The will be the input placeholder for the InputLayer in the network.
target_placeholder	The TensorFlow placeholder variable that is used as the target for the loss function in the network.
downstream_layers	Returns an ordered generator of every layer after this one in the network.
upstream_layers	Returns an ordered generator of every

	layer before this one in the network.
all_layers	Returns a generator of every layer in the network connected to this layer. Ordered by their order in the network.
all_connected_layers	Another name for the method above.
activate_predict	A method that can be passed an input of the t dimension that the input layer for this network takes and return the activation this layer has for it.
layer_number	Returns an int for how many layers are before this one in the network, plus 1.
kwargs	Returns a dictionary of the kwargs needed to reconstruct initialize this layer. This property is used by the clone method
clone	Creates a clone of the current layer and all upstream layers it is connected to.
resize	Used to change the size of the layers output nodes. It takes a range of options for how to do the resize.
remove_layer_from_network	Removes the this layer from its connected network, this layer becomes unusable, its previous layer becomes connected directly to its next layer and the next layer is resized so its dimensions match the previous layer.
detach_output	Removes all layers downstream of this one from the network.
add_intermediate_cloned_layer	Adds a copy of the current layer to the network as its next layer and connects its next layer as the output of this clone.
add_intermediate_layer	Adds a new layer to the network as this layers next layer and connects its next layer as the output of this new layer.
variables	Returns an iterable of all TensorFlow variables used in this layer.
regularizable_variables	Returns an iterable of all TensorFlow variables that should be regularized during training.
variables_all_layers	Returns an iterable of all variables in all layers in the network connected to this layer.

regularizable_variables_all_layers	Returns an iterable of all variables in all layers in the network connected to this layer that should be regularized during training.
get_parameters	Returns an int that is the number of parameters in this layer. e.g. for a hidden layer this will be the weights, input size multiplied by output size, plus the output size for bias.
get_parameters_all_layers	Returns an int for the sum of parameters across all connected layers.
has_resizable_dimension	Returns true if this layer can be resized, Output and Input layer and some others do not have a resizable dimension.
get_resizable_dimension_size	Get the size of the resizable dimension.
get_all_resizable_layers	Returns an iterable of every layer that is resizable in the connected network.
get_resizable_dimension	Returns an int that is the index of the resizable dimension for the network. For a standard hidden layer this will be 0.
get_resizable_dimension_size_all_layers	Returns a list of the resizable dimension sizes for each resizable layer.
find_best_size	A method that given a training and validation DataSet attempts to find the best size for this layer, by doing successive resizes and testing the results.
get_network_state	Returns a dictionary that contains the full state of this network, used for loading and saving.
get_network_pickle	Returns a pickle that contains the full state of this network, used for loading and saving.
load_network_from_state	A static method that takes a dictionary returned by get_network_state and creates a new network from it.
load_network_from_pickle	A static method that takes a pickle returned by get_network_pickle and creates a new network from it.
set_network_state	Modify an existing network to have the same state as a state returned by get_network_state.
resizable_variables	Returns the TensorFlow variables in this

	layer that are changed by resizing.
gradients_with_respect_to_error_op	Returns a TensorFlow op that gets the gradient of each variable with respect to the error.
hessian_with_respect_to_error_op	Returns a TensorFlow op that gets the hessian of each variable with respect to the error.

Table 3.1 : List of method on BaseLayer class

InputLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/input_layer.py

A subclass of BaseLayer, this creates the Tensor Flow placeholder variable used to be the whole graph. It holds the placeholder variable used for triggering the rest of the network. It has overrides of a selection of the methods from the BaseLayer to turn off resizing, which is not possible on an input and bactivation.

HiddenLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/hidden_layer.py

A standard flat, dense neural hidden network layer. It contains TensorFlow variables for one matrix of weights and a vector of bias. It can also take a parameter for bactivate, in which case it also creates a back bias to be used in reverse activation. In addition to the BaseLayer methods it also has:

Method name	**Description**
weights	Returns a numpy array that is the value of the the weight TensorFlow weight variable on the computational graph. Also has a setter that sets the value on the graph.
bias	Returns a numpy array that is the value of the the bias TensorFlow weight variable on the computational graph. Also has a setter that sets the value on the graph.
non_linarity	This is the activation function used to add a non-linearity to the layers output activation. Needs to be a TensorFlow op.
bactivation_loss_train	Returns the TensorFlow op that is the squared difference between this layer's bactivation train and the previous layers activation. Is a measure of how well this layer is able to reconstruct its input.
bactivation_loss_predict	Returns the TensorFlow op that is the squared difference between this layer's bactivation predict and the previous layers activation. Is a measure of how

	well this layer is able to reconstruct its input.

Table 3.2 : List of methods on HiddenLayer class

OutputLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/output_layer.py

The final layer in any network must be an instance of this class or a subclass of it. This layer also creates the target placeholder and loss and regularization functions. Also contains methods for getting stats on training progress and success rate.

Method name	**Description**
target_placeholder	Returns the TensorFlow placeholder variable used as the target during training or prediction.
target_loss_op_train	Returns the TensorFlow op that is used during training to compare the target to the activation, so it does not include regularization aspects of training. The type of loss, cross entropy or MSE is configured elsewhere.
target_loss_op_predict	Returns the TensorFlow op that is used during prediction to compare the target to the activation, so it does not include regularization aspects of training. This means that any random elements are not run, such as dropout or gaussian noise. The type of loss, cross entropy or MSE is configured elsewhere.
loss_op_train	This is the same as target_loss_op_train, but also includes the regularization penalty.
loss_op_predict	This is the same as target_loss_op_predict, but also includes the regularization penalty.
regularizer_loss_op	Returns the TensorFlow operation that is the regularizer term used during training, or prediction.
train_till_convergence	When given a data t this method trains the network until it stops seeing improvement against a validation set for a set number of iterations. This method only modifies weights, it does not change structure.
evaluation_stats	Returns stats on accuracy, loss, and log

	prob for a given data set.
learn_structure_layer_by_layer	Learns network structure by training to convergence and then modifying the size of each layer and checking for improvement. More details in the implementation chapter.
save_checkpoints	Saves the network of a given file path.
learn_structure_random	Learns structure by making random changes to the structure. More details in the implementation chapter.

Table 3.3 : List of methods on OutputLayer class

CategoricalLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/categorical_output_layer.py

Subclass of OutputLayer that is designed for categorical distributions. It uses the softmax activation function, so that probabilities sum to one. Also cross entropy is used instead of mean squared error for the loss.

BinaryOutputLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/binary_output_layer.py

Another subclass of OutputLayer, this one is set up for binary distributions. It is requires that the dataset labels have only a single dimension, which is always set to either 0 and 1.

ConvolutionalLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/convolutional_layer.py

Subclass of BaseLayer that is design for handling 3-D convolutions. Resizing for this layer always takes place along the convolutional output axis, it does not currently support resizing of the along the other dimensions of the convolution window. But given this development work, there is no technical barrier to supporting this. It does not have any extra method in addition to those it inherited from BaseLayer

MaxPool

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/max_pool_layer.py

This layer is used to apply max pool operations to 3-D inputs, most normally ConvolutionalLayers. It could be extended to support other input dimensions in the future. It cannot itself be resized, but instead is only resized as a by product of the ConvolutionalLayer whos input it receives being resized. It does not have any extra method in addition to those it inherited from BaseLayer

FlattenLayer

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/flatten_layer.py

Another subclass of BaseLayer, it reshapes 3-D layers into 1-D layers. Most commonly this is used for feeding a ConvolutionalLayers output into a HiddenLayer or OutputLayer for classification. It does not have any extra method in addition to those it inherited from BaseLayer

DataSet

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/data/data_set.py

This acts as a container for a pair of features and labels. Contains convenience methods such as random reshuffling and iteration through in batches.

Name	**Description**
features	Returns np.Array of features for this dataset, the size of the first dimension should match that of the labels property.
labels	Returns np.Array of labels for this dataset, the size of the first dimension should match that of the features property.
num_examples	Returns int for number of examples in this dataset.
epochs_completed	Returns int for the number of epoch of training we have gone through using either the next_batch or one_iteration in batches methods.
next_batch	Returns a tuple of features and labels for a given batch_size of samples. When it hits the end of the data, it loops round to the beginning of the data and increments the number of epochs_completed.
one_iteration_in_batches	Like next_batch but returns a generator of calls to next_batch that closes after exactly one epoch.
reset	Reset the epoch count and our position in current epoch.

Table 3.4 : List of methods on DataSet class

DataSetCollection

https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/data/data_set_collection.py
This holds 3 DataSets, one for training, testing and validation, as a convenient way of separating them. Paired with this class are a set of methods for creating MNIST, CIFAR-10, CIFAR-100, TwoSpirals and XOR versions of this class.

Name	**Description**
normalized	Returns True if the dataset has been normalized.
train	Returns the DataSet for the train set in this collection.
test	Returns the DataSet for the test set in this collection.
validation	Returns the DataSet for the validation set in this collection.
name	Returns string for the friendly name for this collection.
features_shape	Shape of a single instance of features for the dataset.
labels_shape	Shape of a single instance of labels for the dataset.

Table 3.5 : List of methods on DataSet class

3.4.5 Usage

All the examples from the design portion of this Chapter will work. Here as a more in depth example of creating an implementation of Optimal brain damage on do node pruning/growing:

```
import numpy as np

def node_importance_optimal_brain_damage(layer, data_set_train,
                                         data_set_validation):
        """ Determines node importance based on Optimal brain damage algorithm
        http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf this method can be used to
        determine which nodes should be pruned when reducing the size of a layer, or
        which
```

```
        should be split when increasing the number of nodes

        Args:
                layer (BaseLayer): Subclass of base layer that we are ran
                data_set_train (DataSet): data set used for training
                data_set_validation (DataSet): data set used for validation

        Returns:
                np.array : A 1-d array with the same number of elements as there are
                  output nodes for the layer
        """
        data_set = data_set_train
        weights_hessian_op, bias_hessian_op = \
                        layer.hessien_with_respect_to_error_op

        weights, bias, weights_hessian, bias_hessian = layer.session.run(
        [layer._weights, layer._bias, weights_hessian_op, bias_hessian_op],
        feed_dict={layer.input_placeholder: data_set.features,
                layer.target_placeholder: data_set.labels}
        )

        weights_squared = np.square(weights)
        bias_squared = np.square(bias)

        return np.sum(weights_squared * weights_hessian,
                        axis=0) + bias_squared * bias_hessian
```

This method can then be fed into a created hidden layers within a network, like this:

```
import tensorflow as tf

from tensor_dynamic.node_importance import
node_importance_optimal_brain_damage
from tensor_dynamic.data.mnist_data import get_mnist_data_set_collection

data_set_collection = get_mnist_data_set_collection(validation_ratio=.15)

last_layer = InputLayer(data_set_collection.features_shape)

with tf.Session() as session:
    input_layer = InputLayer(data_set_collection.features_shape)
    hidden_layer = HiddenLayer(input_layer, 250, session,
                        node_importance_func=node_importance_optimal_brain_damage)
    output = CategoricalOutputLayer(hidden_layer,
                            data_set_collection.labels_shape,
                                            session)

    output.train_till_convergence(data_set_collection.train)

    hidden_layer.resize(240)
    # optimal brain damage will be used to select the 10 nodes to remove
```

Chapter 4 - Exploring structure adaption algorithms

In this chapter we look at what kinds of algorithms we can build using the Tensor dynamic library. We look at implementations of some the approaches from the literature review chapter. We then look at some other approaches that I looked into as part of this thesis. Then look at the most successful original approach I found that fulfilled all my requirements. Namely growing and shrinking the number of nodes per layer and adapting the number of layers as part of training.

4.1 Networks without layers

My first attempt at structure learning, long before I even new exactly what my requirements were beyond simply not liking having to specify number of layers and nodes in neural networks, was building a small library in F# that removed the layers concept standard to neural networks. Instead of the network being defined as a series of layers it was instead built on a node by node basis.

Sets of input and output nodes could be created that could then be joined by any arbitrary sets of nodes all with there own individual settings and no constraints on what they could be connected to, except that infinite loops of connections were disallowed. Each individual node could have it's own activation function, could be connected directly to output nodes or to other hidden nodes at any level of depth. They could also have their own individual learning rate or style. Some could use standard gradient descent, others momentum, or even random functions. Backpropagation error could also be linked to individual nodes and collected over iterations.

This seemed like a fun idea to play around with, but had a few problems. The first being that a nice feature of a layer is that each one contains a set of maths operations, normally a matrix multiplication and a bias addition that can be very handily optimized, either run in parallel on a cpu or pushed to a gpu. Removing the grouping of nodes into layers, massively reduced the optimization I could do by running in parallel.

There were still efficiencies to running the network in parallel, achieved by having a threads kick off from each input node, and then block each other if they reach a hidden node that does not have the full complement of it's inputs. But this was a very incremental gain per thread, given all the checking that was required by each.

The next problem that became apparent once the layer structure had been thrown out was that for an approach that aimed to remove a few annoying hyper-parameters, I had created a vast number of parameters. Now that each node could connect to every other node, and have any range of learning rate, approach and activation function, it was very hard to know what a good choice was.

I tried building a network with a simple rule to have a node split itself if it found it was the node with the highest sum of backpropagation error after a fixed number of iterations. But this left a lot of possibilities, which nodes should it now connect to? The same as the node it just split from? But then how much more signal can it process? How would we ever build more depth? We could connect to every other node in the network, in the hope that it might find across every thing a good way to correct its error, but if every node is always connected to every other node, that starts to look like a very inefficient version of a network with layers.

Also errors always seemed to collect at the nodes closest to the output nodes, so if nodes earlier in the network were ever to expand I would need some kind of weighting based on a node's distance from the output, which again seemed like another hyper-parameter in need of tuning.

The final nail in the coffin of this node as first class citizen approach was that a few simple tests seemed to show that having a layer structure worked better. I could not find any research papers that backed my quick and dirty checks, the closest thing is perhaps the cascade correlation network, but it was different enough to suggest that sticking to layers might be the way to go.

4.3 Implementations from literature review

While developing the tensor dynamic library I attempted to implement various techniques from the literature review chapter of this thesis. This was useful both as a personal learning exercise an algorithm exploration exercise, but also as test of the libraries architecture and design. Could it easily support the integration of these techniques.

4.3.1 Cascade correlation network

Because it relates to structure learning, I re-created the cascade correlation network mentioned in the literature review chapter, a full implementation is too long to give here, but can be found here on github https://github.com/DanielSlater/CascadeCorrelation note this is a separate repo from the from Tensor dynamic and currently stands as the only Python implementation of cascade correlation online.

As is mentioned in the literature review cascade correlation has not been shown to be successful beyond trivial toy cases such as the two spirals and my experiences were no different.

4.3.2 Batch normalization, Dropout and Gaussian input noise

Before moving onto more complex structure techniques I implemented batch normalization drop out and Gaussian input noise. Given the huge increase in performance batch normalization provided it seemed natural to always have it available. Dropout and Gaussian noise are also general techniques that can be applied to any layer so it seemed natural to put all 3 together directly into the BaseLayer.

The constructor of the BaseLayer class now takes a parameter `batch_normalize_input`, which if set to true turns on batch normalization for any input to this layer. This means any layer, hidden, convolutional, etc, that inherits from BaseLayer can easily use batch normalization. I also added the parameters `layer_noise_std` and `drop_out_prob` for Gaussian noise and Dropout respectively. Drop_out_prob is a floating point value between 0 and 1 representing the probability with which each input node will be set to zero, if it is set to None or 0 no drop out is turned off for the layer.

Implementation for batch normalization required the follow piece of code to be added to the BaseLayer constructor:

```
if self._batch_normalize_input:
    self._batch_norm_mean_train, self._batch_norm_var_train = (None,
                                                               None)
    self._batch_norm_mean_predict, self._batch_norm_var_predict = (None,
                                                                   None)

    with self.name_scope():
        self._batch_norm_scale = self._create_variable("batch_norm_scale",
                                (self.INPUT_BOUND_VALUE,),
                                batch_norm_scale if
                                batch_norm_scale is not None else
                                tf.ones(self.input_nodes),
                                is_kwarg=True)
```

```
        self._batch_norm_transform =self._create_variable("batch_norm_transform",
                               (self.INPUT_BOUND_VALUE,),
                                                     batch_norm_transform if
                               batch_norm_transform is not None else
                               tf.zeros(self.input_nodes),
                               is_kwarg=True)
        self._normalized_train = None
        self._normalized_predict = None
```

Then the activation_train method needed to be modified as such to support all 3 techniques:

```
def _process_input_activation_train(self, input_tensor):
    if self._batch_normalize_input:
        self._batch_norm_mean_train, self._batch_norm_var_train = \
         tf.nn.moments(self._input_layer.activation_train,
                                      axes=range(len(self.input_nodes)))
        self._normalized_train = ((input_tensor - self._batch_norm_mean_train) /
                          tf.sqrt(self._batch_norm_var_train +
                               tf.constant(1e-10)))
        input_tensor = (self._normalized_train + self._batch_norm_transform) * \
                  self._batch_norm_scale

    if self._drop_out_prob:
        input_tensor = tf.nn.dropout(input_tensor, self._drop_out_prob)

    if self._layer_noise_std is not None:
        input_tensor = input_tensor + tf.random_normal(
                                             tf.shape(self.input_layer.activation_train),
                                             stddev=self._layer_noise_std)

    return input_tensor
```

For prediction activation the following changes were made:

```
def _process_input_activation_predict(self, input_tensor):
    if self._batch_normalize_input:
        self._batch_norm_mean_predict, self._batch_norm_var_predict = \
         tf.nn.moments(self._input_layer.activation_predict,
                              axes=range(len(self.input_nodes)))

        self._normalized_predict = (
                  (input_tensor - self._batch_norm_mean_predict) / tf.sqrt(
                         self._batch_norm_var_predict + tf.constant(1e-10)))
        input_tensor = (self._normalized_predict + self._batch_norm_transform) *\
                  self._batch_norm_scale
    return input_tensor
```

That code allows us to add batch normalization, drop out or gaussian input noise to every subclass of base layer.

4.3.3 Optimal brain damage

An implementation of the weight pruning algorithm optimal brain damage is also given in the previous chapter.

4.3.4 Net 2 wider net

This technique described in the net 2 net section of the literature review, gives a good general technique for adding new hidden nodes to a layer. The below method was used to produce the new weights for the larger layer.

```
def net_2_wider_net(array, vectors_to_extend, noise_std=None,
              halve_extended_vectors=False):
```

```
    """Extends the array arg by the column/row specified in vectors_to_extend
      duplicated

    Examples:
          a = np.array([[0, 1, 0],
                    [0, 1, 0]])
          array_split_extension(a, {1: [1]}) # {1: [1]} means duplicate column, with
                                  # index 1
          # np.array([[0, 1, 0, 1], [0, 1, 0, 1]]))

    Args:
          array (np.array): The array we want to split
          vectors_to_extend ({int:[int]): The keys are the axis we want to split,
                                  0 = rows, 1 = keys, while the values are
                                  which rows/columns along that axis we want
                                  to duplicate.
          noise_std (float): If set some random noise is applied to the extended
                      column and subtracted from the duplicated column. The
                      std of the noise is the value of this column.
          halve_extended_vectors (bool): If True then extended vector and vector
                                 copied from both halved so as to leave the
                                 network activation, relatively unchanged

    Returns:
          np.array : The array passed in as array arg but now extended
    """
          for axis, split_indexes in vectors_to_extend.iteritems():
            for x in split_indexes:
          split_args = [slice(None)] * array.ndim
          split_args[axis] = x
          add_weights = np.copy(array[split_args])
          reshape_args = list(array.shape)
          reshape_args[axis] = 1
          add_weights = add_weights.reshape(reshape_args)

          if halve_extended_vectors:
                    add_weights *= .5
                    array[split_args] *= .5

          if noise_std:
                    random_noise = np.random.normal(scale=noise_std,
                                              size=add_weights.shape)
                    add_weights += random_noise
                    array[split_args] -= np.squeeze(random_noise, axis=[axis])

          array = np.r_[str(axis), array, add_weights]
          return array
```

Here is an implementation of net 2 deeper net:

```
def net_2_deeper_net(bias, noise_std=0.1):
    """This is a similar idea to net 2 deeper net from
      http://arxiv.org/pdf/1511.05641.pdf It assumes that this is a linear layer
      that is being extended and also adds some noise. In the tensordynamic
      code, this assumption is almost always wrong, but it still appears to work
      if the layer is then trained after adding.

    Args:
          bias (numpy.array): The bias for the layer we are adding after
          noise_std (Optional float): The amount of normal noise to add to the
                              layer. If None then no noise is added
                              Default is 0.1
```

```
    Returns:
        (numpy.matrix, numpy.array)
        The first item is the weights for the new layer
        Second item is the bias for the new layer
    """
        new_weights = np.matrix(np.eye(bias.shape[0]))
        new_bias = np.zeros(bias.shape)

        if noise_std:
        new_weights = new_weights + np.random.normal(scale=noise_std,
                                              size=new_weights.shape)
        new_bias = new_bias + np.random.normal(scale=noise_std,
                                        size=new_bias.shape)

        return new_weights.astype(bias.dtype), new_bias.astype(bias.dtype)
```

4.3.5 Tri-state-relu

You can find a class called DuelStateReluLayer in this file on github
https://github.com/DanielSlater/tensordynamic/blob/master/tensor_dynamic/layers/duel_state_relu_layer.py
The DuelStateReluLayer is a simplified version of the TriStateRelu technique described in the literature review section. It has the width learning aspect but not the depth. It is also has additional prune and grow methods. If you the recall from the literature review chapter, in tri-state relu a width parameter is learned for each node that has a loss function that tends to force the the value into a state of either 0 or 1. 0 being an inactive node and 1 being an active node. The prune method removes any nodes set to the off state from the layer, while the grow method increases the number of nodes in the layer if all nodes are set to be in the on state.
This seemed an intuitive approach, but unfortunately the thing I found in experimentation is that this technique never seemed to set all the nodes to be on, except when they had very small numbers of hidden nodes, e.g. 10 or less for MNIST. This meant the technique did not work as signal for growing numbers of hidden nodes beyond trivially small numbers. I tried some amount of fiddling with the hyper parameters for width penalty but could not get this approach to work, so eventually abandoned it.

4.3.6 Backtivation

Another area I experimented with was having the network layers run in a unsupervised as well as supervised manner simultaneously. This was somewhat inspired by the ladder networks approach. The network would be trained on a labeled data set. Each network would in addition to it's standard activation have some weights that would try to reconstruct the networks input activation. Either the original data if it is the first hidden layer, or the output of the previous hidden layer for deeper layers. So each layer is both a standard feedforward layer and an auto-encoder at the same time. The networks loss function, rather than being just the MSE between labels and network activation, would also include each layers unsupervised loss, multiplied by some constant.
My idea was that the difference between a layers input activation and reconstruction of it, would be proxy for how much information that layer was losing in its encoding. Layers that were losing large amounts of information might correspond to ones that needed more capacity, while if the reconstruction were good that might mean that the layer could be safely pruned.
This sounded like a plausible approach but there were two problems with it. First the accuracy of the network got worse in proportion with how strong the unsupervised constant was set to. Even at quite low levels it seemed to negatively affect training performance. Second when I looked at the reconnstruction loss numbers they seemed to always be huge in the first layer and then negligible in every subsequent layer. I did a lot of messing around with different hyperparameters trying to adjust this, but without any success.

4.3.7 Tri-state relu extension

This approach involved using tri-state relu for it's pruning capabilities, but extending it so that if all the nodes in a layer were set to be on, it would add new nodes. Similarly if it's all layers had high scores for depth it would add new layers. This approach sounded plausible, but the issues I encountered were first that layers never seemed to end up with all nodes on unless set to implausibly small numbers of nodes in a layer. Then once new nodes are created they always seemed to go back to being off within a few iterations. This meant that the network never grew in size.

4.4 A new approach to adapting the structure of deep networks

After the various experiments mentioned in the chapter I eventually found one approach that did fulfill with some regularity criteria set up in the introduction. The approach I propose for learning topology is to use a combination of hill climbing, net 2 net and optimal brain damage. The idea is as follows:

1. Build a neural network with some reasonable guess at the topology for the network.
2. Train this network until convergence
3. Starting with the first hidden layer find the best size. If resizing has been tried on each hidden layer and we have seen no improvements, then go on straight to step 4.
    1. First increase the number of nodes by some amount, for all the experiments in this project I use 1.1 as this constant, with a floor of 3, this seemed to work well. When increasing the number of nodes we use some selection algorithm to decide which nodes are the most significant, a number of options are explored for this in the results section. This method is referred to as the node selection function from here on in. Optimal brain damage is a good choice.
    2. Once the nodes are selected split them using net2wider net as described in the literature review, plus some random noise.
    3. Train the new structure until convergence, then test to see if this change has resulted in improvement against a validation set. If yes accept the change and go back to 3.1, keep going until a new change shows no improvement. Then go back to step 3 but for the next layer.
    4. If the first attempt at growing this layer failed then attempt to prune the layer, using the node importance function to decide which nodes can be removed. Then train the network until convergence. If the first pruning is a success then keep pruning until a change shows no improvement. Then go to step 3.
4. If you are here then resizing has been tried on every layer in turn without success. At this point attempt adding a new layer using net 2 deeper net. This layer should be added between the last hidden layer and the output layer. The reasons for this are discussed in the results section. Compare the loss of the network to what it had before this structure change. If it is an improvement accept this change and return to step 3. If it favours the old network, reject the change, structure learning is now complete.
5. Stop when you have made *x* number of random changes with no improvement.

This approach fulfills all the objective, it is able to adapt size as part of training, never needs to start training again from a new network, and should tend towards smaller networks while trading this off against improvements in performance you might get from using a larger network.

There are also many points within it that could be explored further.

- Is optimal brain damage the right way to prune/split nodes?
- Are there better policies for which structure changes to make? Maybe there are signals in the training that suggest when to add a new layer vs increase an existing one.
- How can we deal with problems of overfitting when doing so many successive train to convergence steps?

I will investigate all of these points later on in this project. A python implementation of this method is shown in chapter 5.

Chapter 5 - Library implementation

In this chapter we cover in more detail how the library was implemented. Going into more details on how techniques such as lazy prop were extended. How the library is able to be consistent about resizes between layers. How adding new layers is handled. How cloning is implemented. How saving and loading state is dealt with.

5.1 Lazyprop

As discussed in Chapter 3, it was necessary to have a way to cache the results of operation for later use, lazyprop was chosen as a way to achieve this. This is what my lazyprop implementation looks like:

```
_LAZY_PROP_VALUES = '__lazy_prop_values__'

def lazyprop(fn):
    @property
    def _lazyprop(self):
        if not hasattr(self, _LAZY_PROP_VALUES):
            setattr(self, _LAZY_PROP_VALUES, {})
        lazy_props_dict = self.__dict__[_LAZY_PROP_VALUES]
        if fn.__name__ not in lazy_props_dict:
            lazy_props_dict[fn.__name__] = fn(self)
        return lazy_props_dict[fn.__name__]

    return _lazyprop
```

A lazyprop property can then be used like so:

```
class Example():

    @lazyprop

    def gradients_with_respect_to_error_op(self):

        gradients_ops = []

        for variable in self.resizable_variables:

            gradients_ops.append(tf.gradients(

                        self.last_layer.target_loss_op_predict, variable)[0])


        return gradients_ops
```

Having the lazyprop attribute means that the tf.gradients operation will only be evaluated once for each variable no matter how many times the property is accessed. But there are some methods in Tensorflow, such as the flattening of convolutional layers, or adding new layers, which require rebuilding the entire downstream graph.

In order to support this functionality we need to actually remove the old Tensorflow operation from the graph and create a new operations for everything downstream of that operation. In those cases if we return the cached result from our lazyprop then we will be returning the wrong operations. For these cases we need the ability to clear our lazyprops and then the ability to have a subscriber function called when the lazyprop is cleared. These methods allow us to do that:

```
_LAZY_PROP_SUBSCRIBERS = '__lazy_prop_subscribers__'



def clear_lazyprop(object, property_name):
        """Clear the named lazyprop from this object


        Args:
        object (object):
```

```
        property_name (str):
        """
        assert isinstance(property_name, str)

        if _LAZY_PROP_VALUES in object.__dict__:
            if property_name in object.__dict__[_LAZY_PROP_VALUES]:
         del object.__dict__[_LAZY_PROP_VALUES][property_name]

        if _LAZY_PROP_SUBSCRIBERS in object.__dict__:
           if property_name in object.__dict__[_LAZY_PROP_SUBSCRIBERS]:
        for fn in object.__dict__[_LAZY_PROP_SUBSCRIBERS][property_name]:
                    fn(object)
```

Then this function allows us to set up a subscriber:

```
def subscribe_to_lazy_prop(object, property_name, on_change_func):
        """If the passed in lazyprop is ever cleared the function passed in is called

        Args:
        object (object):
        property_name (str):
        on_change_func (object -> None): function to be called when the lazy prop is cleared,
the
                                    object is passed in as the first arg
        """
        assert isinstance(property_name, str)

        if not hasattr(object, _LAZY_PROP_SUBSCRIBERS):
        setattr(object, _LAZY_PROP_SUBSCRIBERS, defaultdict(lambda: set()))

        object.__dict__[_LAZY_PROP_SUBSCRIBERS][property_name].add(on_change_func)
```

Given that in the majority of cases the correct response to having a layerprop cleared is to clear your own lazyprop, this helper method was created to do this easily:

```
def clear_lazyprop_on_lazyprop_cleared(subscriber_object, subscriber_lazyprop,
                              listen_to_object, listen_to_lazyprop=None):
```

```
"""Clear the lazyprop on the subscriber_object if the listen_to_object property is cleared


Args:
    subscriber_object (object):
    subscriber_lazyprop (str):
    listen_to_object (object):
    listen_to_lazyprop (str):
"""
if listen_to_lazyprop is None:
listen_to_lazyprop = subscriber_lazyprop


assert isinstance(listen_to_lazyprop, str)
assert isinstance(subscriber_lazyprop, str)


subscribe_to_lazy_prop(listen_to_object, listen_to_lazyprop,
                lambda _: clear_lazyprop(subscriber_object, subscriber_lazyprop))
```

A property like the before mentioned gradients_with_respect_to_eror_op can be augmented to:

```
@lazyprop
def gradients_with_respect_to_error_op(self):
    clear_lazyprop_on_lazyprop_cleared(self,
                  "gradients_with_respect_to_error_op",
                   self.last_layer,
                  "target_loss_op_predict")


    gradients_ops = []
    for variable in self.resizable_variables:
  gradients_ops.append(tf.gradients(
       self.last_layer.target_loss_op_predict, variable)[0])


    return gradients_ops
```

5.3 Layer resizing

The key pieces of functionality required is the ability to a resize layers in a consistent way. When the number of nodes changes in the layer being resized, this requires changes to the weights in the next layer. The input dimension of its weights need to be modified accordingly. We also want this functionality to easily generalize across different kinds of layer subclass, e.g. Convolutional, Ladder, etc. The BaseLayer is a natural place for this functionality.

We already have the tf_resize method, described in the previous chapter, that allows us to resize an individual tensorflow variable. But resizing a layer requires the resizing of multiple variables together and some variables from connected layers. e.g. if we want to prune 1 node from the output nodes in a batch normalized HiddenLayer that is connected to another batch normalized HiddenLayer, then we what we need to remove is:

1. one row from the weights variable in the first hidden layer
2. one float from the bias in the first hidden layer
3. one float from gamma in the next hidden layer
4. one float from the beta in the next hidden layer
5. one column from the weights variable in the next hidden layer

And the index of all of these must be consistent for this to be a true prune, not just removing random variables.

To facilitate this I wrote something that looks a bit like data binding. When nodes are created within a layer, the method _create_variable is used, that takes a parameter called bound_dimensions. This parameter is a tuple that defines the dimensions of the variable. The values in the tuple can be either ints, defining the size of that dimension or, a string, either ‘input’ or ‘output’ that binds that variable to the input or output size of the layer respectively. Here is what the method looks like:

```
def _create_variable(self, name, bound_dimensions, default_val, is_kwarg=True,
                     is_trainable=True):
    int_dims = self._bound_dimensions_to_ints(bound_dimensions)


    with self.name_scope():
          if isinstance(default_val, np.ndarray):
                   default_val = self._weight_extender_func(default_val, int_dims)
          elif default_val is None:
                   if len(int_dims) == 1:
                            default_val = self._bias_initializer_func(int_dims[0])
                   else:
                            default_val = self._weight_initializer_func(int_dims)


          var = tf.Variable(default_val,
                    trainable=(not self._freeze) and is_trainable,
                    name=name)
```

```
        self._session.run(tf.variables_initializer([var]))
        self._bound_variables[name] = self._BoundVariable(name, bound_dimensions,
                                                          var, is_kwarg)
        return var
```

The value self._BoundVariable is a NamedTuple defined as:

```
_BoundVariable = namedtuple('_BoundVariable', ['name', 'dimensions', 'variable','is_kwarg'])
```

The is_kwarg property on the _BoundVariable defines if this variable is passed it as a __init__ arg. Keeping track of this will help when we want to clone layers. The method for the clone can look through all the bound variables with is_kwarg = True and pass them into it's clones constructor.

When the input or output size of a layer is changed the layer looks through all the bound variables to see which are bound to the changed dimension and call tf_resize to change them as appropriate. The actual resize method on BaseLayer is bloated with a lot of functionality for edge cases, but here is a simplified version of the method which contains most of what is done:

```
def resize_output(self, new_output_nodes,
                  data_set_train=None,
                  data_set_validation=None):
    """Resize this layer by changing the number of output nodes. Will also
      resize any downstream layers


    Args:
        data_set_validation (DataSet):Data set used for validating this network
        data_set_train (DataSet): Data set used for training this network
        new_output_nodes (int): If passed we change the number of output nodes of
                    this layer to be new_output_nodes
    """
    # choose nodes to split or prune
    output_nodes_to_prune, split_output_nodes = None, None


    if new_output_nodes < self.get_resizable_dimension_size():
        output_nodes_to_prune = self._choose_nodes_to_prune(new_output_nodes,
                                                            data_set_train,
                                                            data_set_validation)
    elif new_output_nodes > self.get_resizable_dimension_size():
        split_output_nodes = self._choose_nodes_to_split(new_output_nodes,
                                                         data_set_train,
                                                         data_set_validation)
```

```
for name, bound_variable in self._bound_variables.iteritems():
    if self._bound_dimensions_contains_output(bound_variable.dimensions):
        self._forget_assign_op(name
        int_dims =self._bound_dimensions_to_ints(bound_variable.dimensions)

        if isinstance(bound_variable.variable, tf.Variable):
            new_values = self._session.run(bound_variable.variable)
            if output_nodes_to_prune or split_output_nodes:
                output_bound_axis = bound_variable.dimensions.index(
                    self.OUTPUT_BOUND_VALUE)
                if output_nodes_to_prune:
                    new_values = np.delete(new_values,
                    output_nodes_to_prune,
                    output_bound_axis)
                else:  # split
                    new_values = array_extend(new_values,
                    {output_bound_axis:
                    split_output_nodes},
                                                   noise_std=.1)

            tf_resize(self._session, bound_variable.variable, int_dims,
                      new_values, self._get_assign_function(name))
        else:
            # this is a tensor, not a variable so has no weights
            tf_resize(self._session, bound_variable.variable, int_dims)

if has_lazyprop(self, 'activation_predict'):
    tf_resize(self._session, self.activation_predict,
        (None,) + self._output_nodes)
if has_lazyprop(self, 'activation_train'):
    tf_resize(self._session, self.activation_train,
        (None,) + self._output_nodes)
```

```
    if self._next_layer and self._next_layer._resize_needed():
        self._next_layer._resize_input(
                                input_nodes_to_prune=output_nodes_to_prune,
                                split_input_nodes=split_output_nodes)
```

Resize input is similar, but rather than take a desired size instead takes the indexes of the nodes that were split or pruned so that those connections are correctly modified.

5.3.1 Adding new layers

As shown in the literature review section, there is a lot of research around the resizing of layers, but a lot less around the effect of adding and removing layers from existing trained networks. We want to be able to support adding and remove layers to already trained deep neural networks. Unlike resizing an existing layer this requires not just changing variables in the graph but building new elements into it. Here as an example of adding a new layer to a network:

```
data_set = load_data_set()
input_layer = InputLayer(input_nodes=data_set.feature_shape)
hidden_layer = Layer(input=input_layer, hidden_nodes=200, non_liniarity=tf.nn.relu)
output_layer = CategoricalOutputLayer(output_nodes=data_set.label_shape)
hidden_layer.add_intermediate_layer(Layer, hidden_nodes=200,
non_liniarity=tf.nn.sigmoid)
```

The first step internally for doing this is a method that disconnects a layer from its subsequent layers. Its must also clear all the lazyprops after disconnecting as these may now be invalid. Here is the implementation:

```
def detach_output(self):
    """Detaches the connect between this layer and the next layer

    Returns:
        BaseLayer : The next layer, now detached from this layer
    """
    if self._next_layer is None:
        raise ValueError("Cannot detach_output if there is no next layer")
    next_layer = self._next_layer

    next_layer._input_layer = None
    clear_all_lazyprops(next_layer)

    self._next_layer = None
    clear_all_lazyprops(self)

    return next_layer
```

The add_intermediate_layer method can now be implemented like:

```
def add_intermediate_layer(self, layer_creation_func, *args, **kwargs):
    """Adds a layer to the network between this layer and the next one.

    Args:
        layer_creation_func (BaseLayer->BaseLayer): Method that creates the
         intermediate layer, takes this layer as a parameter. Any args or
```

```
        kwargs get passed in after passing in this layer
    """
    old_next_layer = self.detach_output()
    new_next_layer = layer_creation_func(self, *args, **kwargs)

    # make sure sizes are correct going forward
    new_next_layer.resize(new_next_layer.get_resizable_dimension_size())

    new_next_layer._next_layer = old_next_layer
    old_next_layer._input_layer = new_next_layer
```

The first argument to the function is layer_creation_func, this function is called with current layer as the first arg and then any `*args` and `*kwargs` to this method passed through this allows us to pass in a layer class `__init__` function to the method, or for something more complex an anonymous function could be used.

5.3.2 Cloning layers

A common task is that we want to simply clone an existing layer and add the clone a new layer after the current one. Because most of the real work is done on the Tensorflow graph doing a python copy or deepcopy will be inadequate. A clone in Tensordynamic instead constructs a new identical copy of the layer, connected to a clone of the input layer, via the class constructor. So that the copy is an exact replica, there is a property kwargs that returns all of the variables needed in the constructor, on the BaseLayer it looks like:

```
@property
def kwargs(self):
    kwargs = {
        'output_nodes': self._output_nodes,
        'weight_extender_func': self._weight_extender_func,
        'layer_noise_std': self._layer_noise_std,
        'drop_out_prob': self._drop_out_prob,
        'batch_normalize_input': self._batch_normalize_input,
        'freeze': self._freeze,
        'name': self._name}
    kwargs.update(self._bound_variables_as_kwargs())
    return kwargs
```

If necessary it can be overridden by the subclass, that have extra kwargs. For the clone method, sometimes you want to clone onto a new Tensorflow session, session is also a parameter to the clone method, it looks like:

```
def clone(self, session=None):
    """Produce a clone of this layer AND all connected upstream layers

    Args:
        session (tensorflow.Session): If passed in the clone will be created with
        all variables initialised in this session If None then the current
        session of this layer is used

    Returns:
        tensorflow_dynamic.BaseLayer: A copy of this layer and all upstream
                layers
    """
    new_self = self.__class__(self.input_layer.clone(session or self.session),
                              # self.output_nodes,
                              session=session or self._session,
                              **self.kwargs)

    return new_self
```

5.3.3 Saving and loading

Tensorflow has a built in check-pointing system for saving variable values, but with structure now being something modifiable this check-pointing system is no longer fit for purpose. We need to build new methods that allow us to get and set an entire neural network. To help us we first need methods for getting the state of a layer. 3 things are required for defining the state of a layer. The class of the layer, the kwargs taken and the size it has been resized to. The _get_layer_state method returns these values.

```
def _get_layer_state(self):
    return self.__class__, self.get_resizable_dimension_size(), self.kwargs
```

For the state of an entire network, the get_network_state method returns the the state of every layer connected to the current one:

```
def get_network_state(self):
    return [layer._get_layer_state() for layer in self.all_connected_layers]
```

This list can then be pickled, or put through some other encoding to be saved to disk. In order to reinstate a saved network the load_network_from_state method can be used:

```
@staticmethod
def load_network_from_state(state, session):
    last_layer = None

    for type, size, kwargs in state:
        if last_layer is None:
            if 'session' in type.__init__.__func__.func_code.co_varnames:
                last_layer = type(session=session, **kwargs)
            else:
                last_layer = type(**kwargs)
        else:
            last_layer = type(last_layer, session=session, **kwargs)

    return last_layer
```

Many layers take the Tensorflow session variable as an input, so this is handled automatically. The type value in the above loop will be the class for the layer, so this will rebuild the a copy of the network.

5.3.4 Train till convergence

Another common task is to run a current structure configuration until convergence. A method that does this is on the output layer it looks like:

```
def train_till_convergence(train_one_epoch_function,
                           continue_epochs=3,max_epochs=10000,
                           on_no_improvement_func=None):
        """Runs the train_one_epoch_function until we go
    continue_epochs without improvement in the best error

        Args:
           on_no_improvement_func (()->()): Called whenever we don't
          see an improvement in training, can be used to change
          the learning rate
           train_one_epoch_function (()->number): Function that when
          called runs one epoch of training returning the error
          from training.
           continue_epochs (int): The number of epochs without
```

```
   improvement before we terminate training, default 3
    max_epochs (int): The max number of epochs we can run for.
    default 10000

Returns:
        int: The error we got for the final training epoch
"""
best_error = train_one_epoch_function()
error = best_error
epochs_since_best_error = 0

for epochs in xrange(1, max_epochs):
    error = train_one_epoch_function()

    if error < best_error:
   best_error = error
   epochs_since_best_error = 0
    else:
   epochs_since_best_error += 1
   if epochs_since_best_error >= continue_epochs:
          break

   if on_no_improvement_func:
          on_no_improvement_func()

return error
```

4.3.5 Find best size

This is a method on BaseLayer that attempts to find the best size by resizing a single layer in isolation. Possibly the absolute best way to run this method would be to try every single layers size, from one hidden node to infinity, or some large number, Then select the single best one. The obvious problem with this being the large computational costs. Also given that we expect there to be some optimum number of hidden nodes for a layer and performance to get progressively worse in either direction from there, I’ve taken a hill climbing approach here.

From the starting size we first attempt to grow the number of hidden nodes in the layer by some multiple, I’ve defaulted this to 1.1. After increasing in size, we re-train until convergence. We then see if our new error is less than what we had before resizing. If we our error has reduced, we accept the change and keep trying to grow the number of hidden nodes in the layer by 1.1 until we stop seeing improvement, at which point we return to the best size we encountered. If the first change is rejected we start to try going down in size by 1/1.1. If we keep going in that direction until we stop seeing an improvement in the error. At that point we return to the structure with the best error rate.

Here is what the full method looks like:

```
def find_best_size(self, data_set_train, data_set_validation,
                                   model_evaluation_function, best_score=None,
                                   initial_learning_rate=0.001,
                   tuning_learning_rate=0.0001):
    """Attempts to resize this layer to minimize the loss against the validation
    dataset by resizing this layer

    Args:
         data_set_train (tensor_dynamic.data.data_set.DataSet):
         data_set_validation (tensor_dynamic.data.data_set.DataSet):
         model_evaluation_function (BaseLayer,
         tensor_dynamic.data.data_set.DataSet -> float): Method for judging
         success of training. We try to maximize this.
         best_score (float): Best score achieved so far, this is purely for
```

```
        optimization. If it is not passed this is calculated in the method
        initial_learning_rate (float): Learning rate to use for first run
        tuning_learning_rate (float): Learning rate to use for subsequent runs,
        normally smaller than initial_learning_rate

Returns:
        (bool, float) : If we resized, the best score we achieved from the
                evaluation function
"""
if not self.has_resizable_dimension():
        raise Exception("Can not resize unresizable layer %s" % (self,))

if best_score is None:
        self.last_layer.train_till_convergence(data_set_train,
                                data_set_validation,
                                                learning_rate=initial_learning_rate)
        best_score = model_evaluation_function(self, data_set_validation)

start_size = self.get_resizable_dimension_size_all_layers()
best_state = self.get_network_state()

resized = False

# try bigger
new_score = self._layer_resize_converge(data_set_train, data_set_validation,
                                                model_evaluation_function,
                        self._get_new_node_count(self.GROWTH_MULTIPLYER),
                                tuning_learning_rate)

# keep getting bigger until we stop improving
while new_score > best_score:
        resized = True
        best_score = new_score
        best_state = self.get_network_state()

        new_score = self._layer_resize_converge(data_set_train,
                                data_set_validation,
                                model_evaluation_function,
                        self._get_new_node_count(self.GROWTH_MULTIPLYER),
                                tuning_learning_rate)
if not resized:
        self.set_network_state(best_state)

        new_score = self._layer_resize_converge(data_set_train,
                                data_set_validation,
                                model_evaluation_function,
                        self._get_new_node_count(self.SHRINK_MULTIPLYER),
                                tuning_learning_rate)

        while new_score > best_score:
                resized = True
                best_score = new_score
                best_state = self.get_network_state()
                new_score = self._layer_resize_converge(data_set_train,
                                        data_set_validation,
                                        model_evaluation_function,
                        self._get_new_node_count(self.SHRINK_MULTIPLYER),
                                        tuning_learning_rate)

# return to the best size we found
self.set_network_state(best_state)
```

```
    return resized, best_score
```

4.3.6 Find best layer structure

This is the method that attempts to find the best structure for the network by making progressive changes to the network until no better change in structure yields a better error rate. From the starting structure, `find_best_size` is called on each layer in turn starting with the first. This process is repeated until we do a full iteration without any of the calls to `find_best_size` resulting in improvements to the error. Once this occurs we attempt to add a new hidden layer between the last hidden layer and the output layer. Again this change in structure is trained until convergence and then evaluated against a validation set. If this results in an improvement in the error, this change is accepted. We then do another iteration of resizing every layer, including the new hidden one until we do a full pass without improvement in the error. We then again attempt to add a new hidden layer.

This process is continued until we have done a whole iteration of resizing every layer and adding a new hidden layer without any action leading to improvement. At this point we return to the structure that had the best result and accept this as our new network structure.

Here is what the full method looks like:

```
def learn_structure_layer_by_layer(self, data_set_train, data_set_validation,
                                   start_learn_rate=0.001,
                                   continue_learn_rate=0.0001,
        model_evaluation_function=bayesian_model_comparison_evaluation,
                                   add_layers=False,
                                   save_checkpoint_path=None):
    self.train_till_convergence(data_set_train, data_set_validation,
                                learning_rate=start_learn_rate)
    best_score = model_evaluation_function(self, data_set_validation)

    if save_checkpoint_path:
        self.save_checkpoints(save_checkpoint_path)

    while True:
        best_score = self._best_sizes_for_current_layer_number(best_score,
                continue_learn_rate, data_set_train,
                data_set_validation, model_evaluation_function,
                                save_checkpoint_path)

        if add_layers:
            state = self.get_network_state()
            self.input_layer.add_intermediate_cloned_layer()
            self.last_layer.train_till_convergence(data_set_train,
        data_set_validation, learning_rate=continue_learn_rate)
            result = model_evaluation_function(self, data_set_validation)
            if result > best_score:
                best_score = result

                if save_checkpoint_path:
                    self.save_checkpoints(save_checkpoint_path)
            else:
                # adding a layer didn't help, so reset
                self.set_network_state(state)
                return
        else:
            return
```

In the next chapter we will attempt to evaluate this approach and also investigate some of the options around it.

Chapter 6 - Evaluation

In order to demonstrate the validity of this approach there are quite a few different issues to look at. The first is when resizing a layer what are the effect of different approaches. This needs to be looked at for both splitting and pruning. To consider pruning first, here were the algorithms tried out to determine which nodes to remove.

- *Random* – remove nodes at random, the baseline for any other approach. In python this looks like:

```
return np.random.normal(size=(layer.get_resizable_dimension_size()))
```

  It if referred to in the test results as random

- *By_dummy_activation_from_input_layer* – activate the network with 3 fake data samples, all input nodes set to one, all input nodes set to zero and all input nodes set to negative one. Prune the nodes with the lowest activation summed across all 3 samples. In python this looks like:

```
shape = (1,) + tuple(int(x) for x in
          layer.input_placeholder.get_shape()[1:])
all_pos_1 = np.ones(shape=shape, dtype=np.float32)
all_zero = np.zeros(shape=shape, dtype=np.float32)
all_neg_1 = -np.ones(shape=shape, dtype=np.float32)

importance = layer._session.run(layer.activation_predict,
              feed_dict={layer.input_placeholder:
               np.append(np.append(all_pos_1,
                           all_zero, axis=0),
                     all_neg_1, axis=0)})

return np.sum(importance, axis=0)
```

- by_real_activation_from_input_layer – activate the network with the actual data features, from either the train or validation set. Prune the nodes with the lowest activation summed across all. In python this looks like:

```
data_set = data_set_train or data_set_validation
importance = layer._session.run(layer.activation_predict,
               feed_dict={layer.input_placeholder:
                           data_set.features})

return np.sum(importance, axis=0)
```

- by_real_activation_from_input_layer_variance – As above but the variance of the activation across all samples is used. In python:

```
data_set = data_set_train or data_set_validation
importance = layer._session.run(layer.activation_predict,
               feed_dict={layer.input_placeholder:
                           data_set.features})
return np.var(importance, axis=0)
```

- by_square_sum – Prune based on the sum of the square value of each parameter in the node. Like Optimal brain damage but ignoring the 2nd derivative term. In python:

```
weights, bias = layer._session.run([layer._weights, layer._bias])
return np.sum(np.square(weights), axis=0) + np.square(bias)
```

- *optimal_brain_damage* - The same described in the literature review chapter, except this is applied to only a single layer.

```
data_set = data_set_train or data_set_validation

weights_hessian_op, bias_hessian_op = \
          layer.hessien_with_respect_to_error_op

weights, bias, weights_hessian, bias_hessian = layer.session.run(
[layer._weights, layer._bias, weights_hessian_op, bias_hessian_op],
```

```
feed_dict={layer.input_placeholder: data_set.features,
        layer.target_placeholder: data_set.labels}
)

weights_squared = np.square(weights)
bias_squared = np.square(bias)

return np.sum(weights_squared * weights_hessian, axis=0) + \
            bias_squared * bias_hessian
```

- *by_removal* – Run the train or validation set with each node in turn having all its parameters set to zero. Prune based on each node's effect on the target loss. This is close to the true loss. If we could test every permutation of every number of nodes that we want to prune, we would know exactly what the best removal set is. But this is far too many combinations to calculate. The simplification of doing one at a time and assuming the combined loss is the same as the sum of individual loss, done here, does not seem unreasonable. But it is still far more slow than any other technique.

```
data_set = data_set_train or data_set_validation
base_error = layer._session.run(layer.last_layer.target_loss_op_predict,
                feed_dict={layer.input_placeholder:
                                data_set.features,
                        layer.target_placeholder:
                                data_set.labels})

weights, bias = layer._session.run([layer._weights, layer._bias])

errors = []
for i in range(layer.get_resizable_dimension_size()):
        # null node
        new_bias = np.copy(bias)
        new_bias[i] = 0.

        new_weights = np.copy(weights)
        new_weights[:, i] = 0.
        layer.weights = new_weights
        layer.bias = new_bias

        error_without_node = \
       layer.session.run(layer.last_layer.target_loss_op_predict,
                    feed_dict={layer.input_placeholder:
                               data_set.features,
                            layer.target_placeholder:
                               data_set.labels})
        errors.append(base_error - error_without_node)

layer.weights = weights
layer.bias = bias

return errors
```

- *full_taylor_series* - Run a full Taylor series against the validation set. One of the assumptions in optimal brain damage is the first term can be ignored because it should be zero, or close verses the train results, the same will not be true of the validation set though. This should make the term good for considering the generalization ability or lack thereof, of the nodes.

```
data_set = data_set_validation

weights_jacobean_op, bias_jacobean_op = \
        layer.gradients_with_respect_to_error_op
weights_hessian_op, bias_hessian_op = \
        layer.hessien_with_respect_to_error_op

weights, bias, weights_jacobean, bias_jacobean, weights_hessian, bias_hessian = \
layer.session.run([layer._weights, layer._bias, weights_jacobean_op,
                bias_jacobean_op, weights_hessian_op, bias_hessian_op],
                 feed_dict={layer.input_placeholder: data_set.features,
                        layer.target_placeholder: data_set.labels}
```

```
                    )

    weights_squared = np.square(weights)
    bias_squared = np.square(bias)

    return np.sum((weights_squared * weights_hessian) * .5 + weights *
        weights_jacobean, axis=0) +
       (bias_squared * bias_hessian) * .5 + bias * bias_jacobean
```

- *error_derrivative* - First order derivative on validation set.

```
    data_set = data_set_validation
    weights_jacobean_op, bias_jacobean_op = layer.gradients_with_respect_to_error_op

    weights, bias, weights_jacobean, bias_jacobean = layer.session.run(
        [layer._weights, layer._bias, weights_jacobean_op, bias_jacobean_op],
        feed_dict={layer.input_placeholder: data_set.features,
                   layer.target_placeholder: data_set.labels}
        )

    return np.sum(weights * weights_jacobean, axis=0) + bias * bias_jacobean
```

- dummy_random_weights - This looks at instead of growing the network by splitting it node, in simply creates new nodes initializing them with random weights. This is not tested for pruning.

6.1 Pruning a layer

I ran all of these on both the MNIST and CIFAR-100 datasets. The results are the average across 5 runs on each. Training a 300 nodes neural network until convergence against a validation set, then pruning the network down to 290 nodes using the specified algorithm. Batch normalization was used on all layers. The regularization coefficient was always 0.01. L2 regularization was used. The loss function was cross entropy. The activation function on the hidden layer was relu. The learning rate was 0.0001, and the optimizer was an AdamOptimizer. Early stopping was used against a validation set, comprising of 15% of the data. The number of continue epochs was 2.

After the network was pruned, the network was again trained until convergence against the validation set. Here are the results before pruning:

train error before growing	validation error before growing	test error before gro
48768.06	15449.36	

Here is a summary of the results these are averaged across 30 runs for each technique, 15 on MNIST, 15 on CIFAR-100:

Pruning hidden layer from 300 to 290 nodes						
method	train error after convergence	validation error after convergence	test error after convergence	train error reduction	validation error reduction	test error reduction
full_taylor_series	44618.50	15448.76	17153.11	4149.55	0.60	-66.13
by_removal	45271.01	15539.81	17210.90	3497.05	-90.46	-123.92

by_real_activation_from_input_layer_variance	45337.93	15484.40	17141.60	3430.13	-35.04	-54.62
by_square_sum	45226.35	15510.16	17182.01	3541.71	-60.81	-95.02
by_dummy_activation_from_input_layer	45051.31	15479.68	17163.27	3716.75	-30.33	-76.29
by_real_activation_from_input_layer	45096.93	15481.31	17151.07	3671.13	-31.96	-64.09
optimal_brain_damage	44720.03	15479.80	17131.90	4048.03	-30.44	-44.92
random	44956.92	15503.43	17163.59	3811.14	-54.08	-76.61
error_derrivative	44897.01	15497.64	17171.67	3871.05	-48.29	-84.69

Table 6.1 : Pruning layers from 300 to 290 nodes

Lower numbers in the error reduction columns are worse results, showing that the error got worse. Because of reduction of nodes we would expect all of these to get worse. Interestingly full Taylor which runs against the validation set actually resulted in an increase on the validation set performance. Because it is actually removing nodes that have overfit to the validation set, but this improvement does not follow through to the test set. Optimal brain damage works out best.

The big difference between accuracy in the test and validation sets suggests that overfitting on this data is a problem. So the above was re-run using mean zero variance one gaussian noise applied to the input to each layer. Again these are averaged across 30 runs for each technique, 15 on MNIST, 15 on CIFAR-100, this resulted in:

train error before growing	**validation error before growing**	**test error before growing**
48139.78	14369.41	15899.82

method	**train error after convergence**	**validation error after convergence**	**test error after convergence**	**train error reduction**	**validation error reduction**	**test error reduction**
dummy_random_weights	46063.44	14334.50	15858.18	2076.34	34.91	41.64
full_taylor_series	46085.16	14296.17	15843.76	2054.62	73.25	56.06
by_removal	45774.56	14358.71	15903.68	2365.22	10.70	-3.86
by_real_activation_fro	46328.	14318.	15857.	1810.9	51.37	42.51

m_input_layer_variance	87	04	31	1		
by_square_sum	46183.21	14324.27	15852.73	1956.57	45.14	47.09
by_dummy_activation_from_input_layer	46075.55	14329.22	15862.82	2064.23	40.19	37.00
by_real_activation_from_input_layer	46484.83	14327.64	15872.83	1654.95	41.77	26.99
optimal_brain_damage	46018.77	14321.32	15855.72	2121.01	48.10	44.10
random	46244.53	14333.25	15863.83	1895.25	36.16	35.99
error_derrivative	46170.45	14322.35	15858.44	1969.33	47.06	41.37

Table 6.2 : Pruning layers from 300 to 290 nodes with

random gaussian noise as input to all layers

Pruning was also tried on a much larger layer where overfitting is likely to be more of a problem, here we go down from 800 nodes to 780:

train error before growing	**validation error before growing**	**test error before growing**
40535.19	14148.22	15644.34

After pruning and then training to convergence we see:

method	**train error after convergence**	**validation error after convergence**	**test error after convergence**	**train error reduction**	**validation error reduction**	**test error reduction**
full_taylor_series	38052.97	14093.99	15594.36	2482.21	54.22	49.98
by_removal	37986.23	14142.74	15647.46	2548.96	5.47	-3.11
by_real_activation_from_input_layer_variance	37986.79	14117.77	15611.39	2548.40	30.45	32.95
by_square_sum	37654.31	14124.65	15611.06	2880.88	23.57	33.29
by_dummy_activation_from_input_layer	37660.59	14120.90	15626.24	2874.60	27.31	18.10
by_real_activation_from_input_layer	37885.00	14130.71	15626.15	2650.18	17.50	18.20

optimal_brain_damage	37735.53	14123.57	15621.41	2799.66	24.65	22.94
random	38170.31	14122.23	15624.19	2364.88	25.99	20.16
error_derrivative	37897.66	14125.74	15624.82	2637.53	22.48	19.52

Again by_removal does best, but all still result in general performance getting worse.

6.2 Growing a layer

I did another set of experiments on how these approaches affected growing the size of a layer. For growing, instead of removing the least important nodes, the most important nodes are selected and split in two using net 2 wider net. Here most important is defined by the result of the node importance function. The idea of Net2WiderNet is to leave the post split activation largely unchanged, but here due to the use of the relu function, activation will change significantly. Here we are interested in how well the layer is able to improve after training with the newly split nodes.

One of the problems we face when creating new nodes in a layer is this is an already learned pattern. If you simple add a new node to an existing trained network with random weights and bias, the node does not perform well. If we think of this in terms of competition, because the existing pattern is already at convergence. The new node, at first, is purely contributing error to the network, so the easiest gradient path is for the node to just learn itself out of existence. It would need a lot of time to become useful, which it doesn't have because it very quickly starts fails against the validation set.

Our ideal method for node splitting has a higher increase in train error after split, but leads to a lowest test error on the test set after training to convergence. In addition to all the methods tested out for pruning, simply setting the new weights randomly, with no splitting was also tried.

The growing tests use all the same hyper parameters as the pruning test, except instead of going from 300 nodes down to 290, we grow from 30 nodes to 33. These are averaged across 30 runs for each technique, 15 on MNIST, 15 on CIFAR-100. The average for training before growing the network was:

train error before growing	**validation error before growing**	**test error before growing**
68286.48	16088.76	17732.84

The results were:

method	**train error after convergence**	**validation error after convergence**	**test error after convergence**	**train error reduction**	**validation error reduction**	**test error reduction**
dummy_random_weights	67285.98	15967.25	17604.68	1000.51	121.50	128.16
full_taylor_series	67118.	15943.9	17575.	1167.	144.8	157.7

	72	1	13	77	5	1
by_removal	67143.48	15924.89	17552.94	1143.00	163.87	179.90
by_real_activation_from_input_layer_variance	66990.87	15916.72	17546.84	1295.61	172.03	186.00
by_square_sum	67082.65	15928.14	17558.17	1203.83	160.62	174.67
by_dummy_activation_from_input_layer	67284.21	15948.99	17580.70	1002.27	139.77	152.14
by_real_activation_from_input_layer	67297.94	15952.68	17582.66	988.54	136.08	150.18
optimal_brain_damage	66995.87	15926.92	17555.57	1290.61	161.84	177.27
random	67068.75	15924.27	17555.63	1217.74	164.49	177.21
error_derrivative	67161.55	15938.98	17569.71	1124.93	149.78	163.13

Table 6.3 : Growing hidden layer from 30 to 33 nodes

What we can see from this is all approaches outperform the dummy_random approach, which involves no splitting but instead just creating random weights for new nodes. The approach that worked out best was by_real_activation_from_input_layer_variance thought it's is only slightly better than simply choosing nodes at random. The fact that adding dummy weights is the worst suggests that maybe a part of the problem with growing is in order to add new nodes to an existing pattern there needs to be some increase in inaccuracy first.

Having already fully trained all the existing nodes we will be sitting at or close to some local minima. The new nodes being created randomly will initially find that the best behavior is for them to learn to do nothing, so as to sit in the existing local minima. All other approaches actually split existing nodes, that are likely significant in the pattern and so potentially move us away from the existing local minima, to hopefully find a deeper one once we have the flexibility of the new nodes.

That test though useful is for an increase in size at a very small level, where overfitting would not be a problem. It is worth also looking at increasing the size of a network at a larger size, where overfitting is more of a possibility. Here are the results for growing from 120 nodes up to 140.

train error before growing	**validation error before growing**	**test error before growing**
56935.07	15993.61	17644.0

method	**train error after convergence**	**validation error after convergence**	**test error after convergence**	**train error reduction**	**validation error reduction**	**test error reduction**
dummy_random_weights	54243.3	16046.9	17703.	2691.7	-53.33	-59.07

	2	4	10	5		
full_taylor_series	54626.15	16165.08	17745.55	2308.92	-171.47	-101.52
by_removal	54977.87	15980.59	17631.58	1957.20	13.02	12.45
by_real_activation_from_input_layer_variance	54776.33	16059.87	17704.84	2158.75	-66.26	-60.81
by_square_sum	54828.00	16025.77	17678.75	2107.07	-32.16	-34.72
by_dummy_activation_from_input_layer	54881.79	16026.49	17679.42	2053.29	-32.88	-35.39
by_real_activation_from_input_layer	54712.41	16062.47	17706.88	2222.67	-68.86	-62.85
optimal_brain_damage	54998.89	16107.28	17738.06	1936.19	-113.67	-94.03
random	54996.49	16038.53	17688.83	1938.59	-44.92	-44.80
error_derrivative	54956.47	16018.14	17671.63	1978.60	-24.53	-27.60

Here again we see what looks like issues from overfitting, with huge increases in training performance and decrease for validation and test. The only exception to this is the by_removal approach which does improve generalization performance.

This is interesting, the by removal approach splits nodes based on how the validation score is affected by removing a given node. These nodes when removed resulted in the biggest drop in validation performance. But after splitting them and retraining actually improved generalization results. This is possibly worthy of further study?

Given these results look to be largely down to generalization issues, here are the results when re-run with added gaussian noise:

this resulted in:

Reduction in error when going from 120-140 nodes, with variance 1.0 gaussian noise per layer			
Method	**Train error reduction**	**Validation error reduction**	**Test error reduction**
dummy_random_weights	47.27	2.47	2.40
by_dummy_activation_from_input_layer	43.94	3.01	3.34
by_real_activation_from_input_layer	48.08	2.60	2.68
by_real_activation_from_input_layer_variance	51.92	2.73	3.01
by_removal	46.54	3.25	3.68
by_square_sum	40.28	2.20	2.35
error_derrivative	58.54	3.63	3.53

full_taylor_series	54.86	2.40	3.39
optimal_brain_damage	47.36	2.68	2.69
random	41.14	2.54	2.75

Here we see all approaches now do improve the score across training, validation and test sets. by_removal comes out best, but there is not much variation between approaches. Across the results run, by_removal appears to perform best, but given its computational cost optimal_brain_damage is used from here on, unless otherwise specified.

6.3 Adding layers

Another question we need to consider is how best to add new layers to the network. Should new layers be added between the input and first hidden layer, between hidden layers or between the last hidden layer and the output. Also how does having a mix of convolutional layers and hidden layers affect error rates?

I ran series of experiments where a network with 3 hidden layers was trained to convergence. Then a new hidden layer was inserted after one each of the existing hidden layer in turn. This was tried on just the CIFAR-100 dataset, the MNIST was left out because it is known to be easy to approximate with just 1 or 2 layers. Once the new layer was created, the network was again trained to convergence to observe the test error.

The CIFAR-100 dataset has 100 classes with 600 samples for each class. For this and all other CIFAR tests I derived the set into 425 training items, 75 validation items and 100 test items per class. No changes were made to the standard CAFAR-100 dataset. The input was normalized through batch normalization applied to the input layer.

Here are the averages across 15 runs:

New layer inserted at index	Train error	Train accuracy	Test error	Test accuracy
Before inserting new layer	77814.85	54.45%	31208.42	26.49%
1	71486.88	57.02%	31486.81	26.65%
2	68714.79	58.73%	31962.07	26.35%
3	65738.1	60.59%	32192.89	26.47%

We can see from these results the issue that we will consistently run into throughout this project, overfitting. Though train error and accuracy improve when adding the new layer into any position, the test error and accuracy get worse. In fact inserting into position 3, just ahead of the output layer gives the greatest improvement against the train set, but the worst degradation of performance against the test set.

To better deal with overfitting I started added mean zero variance one gaussian noise to the input to every layer. With this the results where:

New layer	Train error	Train accuracy	Test error	Test accuracy

inserted at index				
Before inserting new layer	82938.77	48.67%	28955.51	30.21%
1	76925.22	51.19%	29005.24	30.38%
2	75864.71	52.54%	28968.16	30.60%
3	75100.74	52.30%	28957.31	30.62%

As is to be expected adding gaussian noise reduced training accuracy but improved test accuracy. Results like this were consistent across other experiments so as standard I add gaussian noise on all further experiments. The other thing we can see from these results that is consistent with the previous set is that adding new layers in the last position has better results for the training performance and marginally so for testing. A result like this is also consistent with the approach Geoffrey Hinton took when building Deep restricted boltzmann machine of adding new layers on the end of already trained networks. Given this I adopted this the policy of when adding new layers always adding them between last hidden layer and the output layer.

6.4 Learning structure

For evaluating the overall technique I first ran a neural network with 1 hidden layer containing 10 hidden nodes, relu activation function and batch normalization on each layer and gaussian noise of 0. The loss function was softmax and the it used l2 regularization with a coefficient of 0.0001. Here is an example of how the network sized changed over its run on the MNIST data set:

Hidden nodes	10	13	16	19	22	25	28	31	34
Validation error	3159.92	3088.22	3082.83	3080.82	3034.13	2970.9	2901.57	2838.14	2799.41
Best	10	13	16	19	22	25	28	31	34

37	40	44	48	52	57	62	68	74	81
2645.99	2554.23	2375.97	2282.24	2230.94	2100.31	2074.57	2031.4	1963.41	1934.23
37	40	44	48	52	57	62	68	74	81

89	97	89-89	97-89	80-89	72-89	65-89	72-97	72-80	72-72
1804.16	1818.26	1801.82	1804.54	1747.41	1707.65	1793.61	1730.69	1629.31	1499.75
89	89	89-89	89-89	89-89	72-89	72-89	272-89	72-80	72-72

72-65	79-72	65-72	72-72-72
1522.34	1561.51	1603.82	1671.76

72-72	72-72	72-72	72-72

The top row of the above table shows what hidden node configuration is tried at each step. The next row shows the validation error after training with that configuration. If the validation error for that configuration is better than the previous best, that configuration is accepted as the new best. Otherwise the configuration reverts back to the current best. The final structure chosen for the above network was 2 layers each with 72 hidden nodes. After that every change tried resulted in an increase in the validation error. Including the final try of adding a 3rd hidden layer.

It is interesting to see the best configuration found with 1 hidden layer contained 89 nodes, but once the second hidden layer is added, both layers are found to improve performance by reducing down to 72 nodes each.

The final test accuracy was 96.87% which is not too bad. For comparison the test accuracy for the starting configuration containing 10 hidden nodes was around 90%. We can compare that with the result we get when building a network with 2 hidden layers, each size 72 and seeing what the performance is, if we do that we see that the average over 10 runs is 96.15% and the best performance is only 96.35%. This is interesting and also a little hard to explain. One thing I worried about a lot in the project is that this technique might have big problems with overfitting, but here it seems, for at least 1 run that performance actually improved significantly compared to running from scratch.

When we do 30 runs of the same thing we see the different final sizes and test accuracies. In the below table each line is a run of the structure learning network, starting with a single hidden layer with 10 nodes. The final hidden nodes column shows the structure network ended up with. The numbers there are the numbers of hidden nodes in each hidden layer, starting with the layer closest to the input. So 106-97 would be a 2 layers hidden network, where the first layer had 106 nodes and the second 97. The number of parameters is the total numbers of parameters used in that network. Which is the sum of parameters in each layer. The other columns are the error and accuracy for the various division of the data. The error here is the sum of error across all items in the dataset:

Structure learning on MNIST							
train error	**train accuracy**	**validation error**	**validation accuracy**	**test error**	**test accuracy**	**final hidden nodes**	**number of parameters**
1831.05	99.19%	919.45	97.51%	916.85	97.55%	139	112361
1787.36	99.03%	890.92	97.50%	823.28	97.57%	106-97	96543
3440.33	98.01%	1158.46	96.71%	1060.65	96.87%	62	50992
2037.92	98.93%	1002.34	97.35%	911.84	97.60%	167	134677
1768.91	99.04%	978.94	97.50%	869.87	97.68%	68-127	65381
1051.99	99.54%	804.10	97.96%	805.18	97.96%	127-127	119307
2204.86	98.74%	969.11	97.33%	917.07	97.44%	57-127	55327
2364.75	98.75%	991.34	97.37%	916.77	97.59%	97	78887
3540.57	98.02%	1225.72	96.63%	1167.66	96.62%	52	43022
5144.12	97.00%	1501.62	95.86%	1336.21	96.22%	28	23894
2779.92	98.42%	1089.50	96.98%	1023.60	97.11%	68	55774
4871.04	97.17%	1422.96	96.00%	1408.08	96.02%	44	36646
7267.86	95.72%	1849.69	94.89%	1734.58	95.13%	22	19112

6211.22	96.43%	1679.30	95.41%	1605.75	95.52%	31	26285
879.68	99.59%	828.70	97.71%	819.05	97.88%	106-221	111299
7584.71	95.45%	1896.10	94.73%	1720.27	95.06%	16	14330
2696.58	98.46%	1085.48	96.95%	1073.92	97.01%	52-6	46532
1523.53	99.20%	958.62	97.55%	882.41	97.59%	89-89	80699
1846.36	98.98%	932.75	97.48%	919.61	97.53%	81-81	72939
2141.73	98.80%	916.24	97.50%	966.14	97.29%	52-106	49392
991.93	99.49%	805.72	97.76%	765.11	97.81%	139-44	117659
1937.27	98.87%	941.97	97.45%	881.15	97.66%	81-106	75289
1127.34	99.42%	868.65	97.82%	810.33	97.88%	97-243	104647
3642.57	97.83%	1207.37	96.75%	1128.62	96.77%	48	39834
2459.94	98.65%	1105.39	96.98%	998.68	97.36%	116	94030
1914.02	98.93%	933.01	97.51%	921.82	97.52%	57-127	55327
1346.41	99.27%	911.81	97.64%	870.54	97.71%	97-167	96287
3635.30	97.88%	1207.99	96.62%	1235.33	96.76%	68	55774
2241.80	98.68%	949.02	97.33%	901.82	97.50%	49-97	46155
947.64	99.56%	781.45	97.91%	833.13	97.68%	124-116	115058
Averages							
2773.96	98.43%	1093.79	97.02%	1040.84	97.13%		69781

For comparison I ran a grid search on MNIST, varying the numbers of hidden nodes and taking the average of 3 runs of each. Other than the number of hidden nodes, all other hyper parameters were unchanged from those used for the structure learning. The results are presented here, ordered by train accuracy accesending:

Grid search results on MNIST			
Hidden nodes	**Parameters**	**Test accuracy**	**Train accuracy**
50-50-50	46728	98.97%	96.65%
50	41428	98.94%	96.65%
100-50-50	88578	99.28%	96.92%
50-50	44078	99.03%	96.93%
75-75-75	73053	99.27%	96.93%
75-50-50	67653	99.25%	97.00%
75-50	65003	99.17%	97.05%
75-75-50	70853	99.30%	97.06%
75	61353	99.37%	97.07%
100-50	85928	99.45%	97.08%
100-75-75	94603	99.37%	97.08%
100-100	91578	99.41%	97.10%
75-75	67203	99.31%	97.11%
100-100-100	101878	99.35%	97.11%
100-100-50	96228	99.35%	97.14%

100	81278	99.56%	97.14%
125-50-50	109503	99.36%	97.15%
100-75-50	92403	99.43%	97.18%
200-50-50	172278	99.44%	97.18%
125-75-50	113953	99.36%	97.20%
150-150	144078	99.47%	97.20%
100-100-75	99053	99.34%	97.20%
125-75-75	116153	99.46%	97.22%
125-100-50	118403	99.49%	97.22%
125-100-100	124053	99.43%	97.22%
150-125	140003	99.52%	97.24%
200-125-75	194703	99.50%	97.24%
125-125	117203	99.47%	97.24%
150-100-75	143403	99.43%	97.24%
125-100-75	121228	99.51%	97.25%
125-125-50	122853	99.47%	97.27%
125-125-75	126303	99.53%	97.27%
200-75-50	178603	99.54%	97.28%
200-125	185603	99.57%	97.28%
150-100-100	146228	99.48%	97.29%
150-75-75	137703	99.48%	97.30%
100-75	88753	99.46%	97.30%
125-100	113753	99.49%	97.30%
150-50-50	130428	99.44%	97.30%
150-50	127778	99.54%	97.31%
150-125-50	145653	99.49%	97.31%
125-50	106853	99.49%	97.31%
125-125-100	129753	99.47%	97.32%
150-125-100	152553	99.45%	97.32%
200-100	180278	99.56%	97.32%
125-75	110303	99.47%	97.33%
200-150-75	201653	99.57%	97.33%
150-150-75	154803	99.52%	97.33%
150-125-75	149103	99.50%	97.33%
200-100-50	184928	99.54%	97.34%
125	101203	99.60%	97.34%
200-75-75	180803	99.51%	97.35%
150-100	135928	99.52%	97.37%
150-75	131853	99.54%	97.37%
200-150-100	205728	99.52%	97.37%
150-100-50	140578	99.52%	97.37%

150-75-50	135503	99.53%	97.38%
200-150-50	197578	99.57%	97.38%
150-150-50	150728	99.56%	97.39%
150-150-100	158878	99.54%	97.40%
200-100-75	187753	99.56%	97.40%
200-100-100	190578	99.50%	97.41%
200-200	201578	99.55%	97.41%
200-200-100	220878	99.58%	97.41%
200-125-100	198153	99.54%	97.43%
150	121128	99.65%	97.44%
200-200-50	210228	99.62%	97.44%
200-125-50	191253	99.56%	97.46%
200-150	190928	99.56%	97.47%
200-75	174953	99.59%	97.47%
200	160978	99.67%	97.49%
200-200-75	215553	99.58%	97.50%
200-50	169628	99.60%	97.51%

Also here are the best 5 single results, not averages, from the grid search:

Best 5 single results from grid search			
Train error	**Test error**	**Hidden nodes**	**Parameters**
99.66%	97.56%	200-200-50	210228
99.65%	97.56%	200-200-75	215553
99.59%	97.57%	200-75	174953
99.58%	97.60%	200-125-100	198153
99.72%	97.61%	200	160978

From the results of the grid search we can first see that more hidden nodes in the first layer is a big factor in performance. Here are the average train and test accuracies for configurations with different numbers of hidden nodes. So here the 50 is the average of the configurations 50, 50-50 and 50-50-50.

Hidden nodes in first layer	Average train accuracy	Average test accuracy
50	98.98%	96.74%
75	99.28%	97.04%
100	99.40%	97.13%
125	99.47%	97.26%
150	99.51%	97.33%

200	99.56%	97.39%

When it comes to adding layers this seemed to be less significant, here is a chart of the differences between one hidden layer of x nodes and 2 layers each with x nodes.

One layer hidden nodes	Two layers hidden nodes	Train accuracy difference	Test accuracy difference
50	50-50	0.09	0.28
75	75-75	-0.06	0.04
100	100-100	-0.15	-0.04
125	125-125	-0.13	-0.1
150	150-150	-0.18	-0.24
200	200-200	-0.12	-0.08

For MNIST it looks like adding layers actually decreased performance for all but very small numbers of nodes. These results are consistent with what is generally known about the data set.

We can now compare the results from this grid search with the results from the structure learning. The structure learnings best result was a test accuracy of 97.96% with a final configuration of 127-127. Interestingly this is actually better than any of the best single results from the grid search and with quite a few less parameters than the grid. The 2 next best results from structure learning also out perform any grid search results, both scoring 97.88% but with very unusual looking hidden nodes configurations. One had the configuration 97-243, the other 106-221, both having much larger 2nd hidden layers than the first. This is very strange, I don’t recall any literature or reports advising larger layers after smaller ones. In general it is suggested that later layers should be the same size or smaller than previous ones. I have thought that was a failure of structure learning, except for the fact that it has such good results. A sample run of networks with hidden nodes of that configuration did not yield good results so it would seem this is something specific to how parameter learning occurred with structure learning. This requires more investigation.

Unfortunately other structure learning results are not so impressive. The worst is a run that stopped at only 16 hidden nodes, adding only 6 from the original. Here is the path that network took:

Hidden nodes	Validation error
10	2818.26
13	2465.48
16	1896.1
19	1945.25
16-16	1903.24

The move up to 19 nodes from 16 results in a decrease in the validation score, after that adding a layer did not help. An interesting point here is that the validation error of 1896.1 is very low. I sampled 15 networks trained in the same way, starting with 10 nodes then increasing in size. The average score at 16 hidden nodes was 2423.79, the other networks did not get a better validation error than 1896 until around 30 hidden nodes. It seems this configuration was freakishly good at generalizing and did not benefit from then adding more nodes. Indeed it would be interesting to find another configuration like this and see what the saliency map for the image was.

This highlights the unstableness of this approach. Though no other result was as bad, many networks stopped growing with under 50 nodes and test accuracies in the 95% range.

Using by_removal

As mentioned in the growing and pruning layers section, the by_removal option worked out a lot better than any other technique for choosing node splitting and pruning, but it was far more computationally expensive and so was not selected. Here is the results of 30 runs using by_removal instead or optimal_brain_damage.

Train error	Train accuracy	Validation error	Validation accuracy	Test error	Test accuracy	Hidden nodes	Parameters
922.534	99.55%	831.664	97.93%	813.984	97.98%	152-116	140342
1998.26	98.91%	965.265	97.25%	980.152	97.27%	97	78887
1169.45	99.51%	874.579	97.66%	850.073	98.05%	152-106	138692
1326.33	99.35%	900.901	97.66%	936.536	97.76%	183	147429
1306.63	99.36%	966.885	97.54%	1041.4	97.26%	221	177715
1925.19	98.94%	1031	97.30%	980.2	97.51%	106	86060
12252.4	92.70%	2779.2	92.52%	2620.18	92.63%	13	11939
1096.66	99.51%	869.866	97.65%	835.734	97.85%	106-97	96543
675.415	99.74%	789.751	98.15%	809.971	98.09%	242-267	260117
898.084	99.62%	796.955	97.97%	782.621	98.01%	167-221	172787
1953.95	98.94%	1025.48	97.37%	1007.1	97.41%	89	72511
1263.41	99.34%	909.914	97.57%	924.877	97.59%	183	147429
793.211	99.64%	725.346	98.07%	711.818	98.10%	201-136	188869
583.988	99.76%	698.935	98.09%	711.13	98.17%	152-47	128957
1102.38	99.45%	940.551	97.52%	963.996	97.52%	139	112361
579.719	99.75%	726.129	98.25%	748.333	98.23%	167-102	151367
1355.88	99.31%	938.432	97.49%	948.058	97.47%	127	102797
977.283	99.58%	927.548	97.90%	966.564	97.51%	183	147429
1787.04	99.02%	977.814	97.35%	976.546	97.26%	106	86060
1249.48	99.36%	953.326	97.56%	895.266	97.53%	167	134677
1297.53	99.36%	912.75	97.68%	901.315	97.54%	152	122722
1270.6	99.35%	822.088	97.88%	814.745	97.69%	139-60	120091
1633.95	99.11%	961.038	97.52%	926.642	97.54%	127	102797
1258.5	99.38%	944.337	97.63%	943.472	97.69%	183	147429
2319.17	98.62%	1056.02	97.03%	1046.86	97.18%	74	60556
12218	92.88%	2744.53	92.56%	2594.13	92.61%	10	9548
1016.2	99.53%	901.681	97.77%	884.818	97.70%	221	177715
2001.91	98.88%	1025.09	97.21%	960.55	97.44%	97	78887
1905.13	98.84%	992.393	97.27%	899.085	97.67%	62-62	55022
1317.52	99.38%	968.473	97.55%	957.17	97.49%	201	161775

This set of results contains a bunch of runs that ended at upwards of 98% an exceptional score for MNIST. The best of which was a run that ended with a configuration of 167-102 and a score of 98.23%. Far better than anything from the grid search runs. Unfortunately there were two runs that went very badly, one ending at only 10 hiddens nodes and the other 13, both achieving bad accuracy scores. But if you take these two results, the average score is 97.66% better than the best score from grid search. Again this points to this technique being potentially useful, but more needing to be done to deal with the inconsistency.

6.5 CIFAR-100

Here are the results on CIFAR, this is using exactly the same technique as with the MNIST results. Note this is using flat hidden layer, so not a convolutional network:

Train error	Train accuracy	Validation error	Validation accuracy	Test error	Test accuracy	Final hidden nodes	Number of parameters
99547.8	39.20%	28128.7	25.17%	31459.9	24.95%	152-57	488179
79853.1	52.25%	27940.0	27.00%	31059.5	27.20%	427-291	1473499
87113.5	47.47%	27939.8	26.44%	30924.3	27.00%	267-182	894609
78113.8	53.09%	27844.3	27.38%	30925.3	27.53%	322-427	1177869
75648.3	53.63%	27493.5	28.37%	30714.6	28.02%	354-293-354	1339579
99041.1	39.91%	28127.8	25.47%	31255.8	25.86%	183-106	599285
79530.9	52.05%	27699.1	27.90%	30935.4	27.64%	427-322	1489929
98786.5	39.79%	28485.6	25.48%	31698.1	25.37%	183-40	580409
74386.5	55.28%	27867.8	27.51%	30880.6	28.02%	427-427	1545579
90373.6	45.04%	27840.7	26.33%	31066.2	26.66%	243-114	792913
89632.4	46.01%	27836.8	25.83%	30987.7	26.83%	139-293	504575
90102.1	45.51%	27813.3	26.58%	31027.4	26.58%	293-127	957511
83142.9	49.74%	27838.1	26.91%	31016.8	26.88%	322-221	1090319
76130.8	54.88%	27777.1	27.69%	30921.9	27.68%	469-240	1585699
87235.9	46.96%	27669.9	26.71%	30771.7	27.01%	267-152-102	894319
78509.6	52.81%	27898.9	27.56%	31008.2	27.96%	389-293	1346575
76837.8	53.76%	27796.7	27.52%	30897.4	27.91%	469-242	1586843
91674.1	44.08%	27986.1	25.59%	31180.6	25.45%	515-37	1612735
94047.5	42.91%	27912.5	26.08%	31265.6	25.60%	167-95	545419
79862.3	51.98%	27748.2	27.54%	30840.9	28.08%	389-389	1393807
88026.7	48.14%	28272.8	26.66%	31393.2	26.49%	469	1495319
91593.0	44.27%	27747.1	26.37%	31087.3	26.66%	201-139	666575
88347.9	45.98%	27944.3	25.96%	31045.7	26.62%	389-68	1235875
101527.0	38.26%	28053.2	25.36%	31411.6	25.24%	167-57	535159
85889.6	47.49%	28026.3	26.47%	31120.8	26.54%	293-139	962263
84980.2	50.26%	28068.1	26.20%	31111.4	26.70%	515	1641369
97393.7	40.41%	28068.8	25.20%	31264.6	25.64%	267-37	840959
93681.2	42.81%	28016.7	25.44%	31271.3	26.27%	267-57-62	852579
81957.1	50.50%	27899.5	27.13%	31109.6	26.93%	293-354	1047403
83704.8	49.36%	27858.2	27.22%	30941.1	27.20%	354-201	1186651

Averages							
86889.1	47.46%	27920.0	26.57%	31086.5	26.75%		1078793

Here how this compares with a grid search on CIFAR-100. The below table is with all other parameters unchanged from the above run. Also of note this is not the average of 3 runs for each parameter, like the earlier grid search, but instead the result of a single run. The average of more would be preferable but unfortunately it is extremely computationally expensive:

accuracy_train	**error_train**	**accuracy_test**	**error_test**	**dimensions**	**parameters**
40.88%	125235	25.63%	32143.7	300	952000
42.39%	122240	27.45%	31519.1	300-300	1042300
36.78%	131385	26.97%	31390.7	300-300-300	1132600
38.51%	131214	25.81%	32907.3	300-300-300-500	1303100
42.00%	130166	25.10%	33485.2	500	1586600
44.33%	116181	27.06%	31485.9	500-300	1716900
42.11%	123374	27.49%	31999.9	500-300-300	1807200
43.15%	120535	26.81%	32391.4	500-300-300-500	1977700
45.09%	113070	28.37%	30886.7	500-500	1837100
45.74%	111462	27.95%	31075.1	500-500-300	1967400
40.63%	126023	26.57%	32352.5	500-500-300-500	2137900
43.61%	122976	27.38%	32206.5	500-500-500	2087600
40.94%	131692	25.42%	34084.3	500-500-500-500	2338100
45.44%	126294	25.15%	33675.8	1000	3173100
47.82%	116115	26.44%	32596.5	1000-300	3403400
41.71%	122905	26.95%	31931.6	1000-300-300	3493700
43.49%	121315	26.91%	32570	1000-300-300-500	3664200
50.35%	114415	27.19%	32852.5	1000-500	3623600
48.12%	109961	28.57%	31219.7	1000-500-300	3753900
43.40%	118106	26.88%	31946	1000-500-300-500	3924400
41.11%	127851	25.56%	33008.6	1000-500-500	3874100
38.04%	137288	24.43%	33836.4	1000-500-500-500	4124600
47.95%	123389	25.96%	33972.4	1000-1000	4174100
46.77%	106352	28.77%	30362	1000-1000-300	4404400
43.72%	124798	25.93%	33424.2	1000-1000-300-	4574900

				500	
51.16%	112390	27.60%	33201.3	1000-1000-500	4624600
40.95%	122996	26.45%	31943.9	1000-1000-500-500	4875100

The best results for the grid search were these 5 entries:

accuracy_train	error_train	accuracy_test	error_test	dimensions	parameters
46.77%	106352	28.77%	30362	1000-1000-300	4404400
48.12%	109961	28.57%	31219.7	1000-500-300	3753900
45.09%	113070	28.37%	30886.7	500-500	1837100
45.74%	111462	27.95%	31075.1	500-500-300	1967400
51.16%	112390	27.60%	33201.3	1000-1000-500	4624600

Here are the best results using structure learning:

Train error	Train accuracy	Validation error	Validation accuracy	Test error	Test accuracy	Final hidden nodes	Number of parameters
79862.3	51.98%	27748.2	27.54%	30840.9	28.08%	389-389	1393807
75648.3	53.63%	27493.5	28.37%	30714.6	28.02%	354-293-354	1339579
74386.5	55.28%	27867.8	27.51%	30880.6	28.02%	427-427	1545579
78509.6	52.81%	27898.9	27.56%	31008.2	27.96%	389-293	1346575
76837.8	53.76%	27796.7	27.52%	30897.4	27.91%	469-242	1586843

As we can see here the best results come from the grid the search with the range of 2-3 hidden layers. From grid search all the 1 layer networks performed badly. But all the runs of structure learning are at least in the a reasonable range getting getting close to 25% or more. The best scores for the structure learning have very high training scores, all the top 5 were higher than the highest training score for the grid search, 51.6%. This suggest that part of the reason for the underperformance is overfitting.

Chapter 7 - Summary

The technique introduced here appears able to in the best case get better results than grid search and in a much smaller portion of training time. You can start a neural network with a terrible hidden node configuration and the techniques left to it’s own devices can find you a good working configuration.

The downside is, it’s results are very inconsistent, sometimes performing far worse than what you would get from a good configuration and can be prone to overfitting. Given this, if anyone is planning on using this approach “in the wild” I would advise running a batch of samples of using this technique and selecting the best one or then running training from scratch with the best configuration.

7.1 My Contribution

The library I have built, Tensordynamic is I think an interesting addition to the landscape of available deep learning tools. It is very much my hope that other people will make use of it, there are at time of writing 4 stars on github. I will try to promoting it as a successful open source platform for this area of research. This contains unique functionality around resizing network as part of training that currently exists nowhere else. If my library is not itself adopted I would hope some of the functionality is built into existing deep learning libraries.

The technique of learning the structure as part of training as presented in this thesis, is successfull. You can initialize a deep neural network with very bad structure and it will find a more acceptable one, going further than that, it seems to generalize better than that running in a vanilla manor. I believe this is a significant contribution, though one where I would suspect other better work might be able to be built upon it.

7.2 Future work

As stated in the introduction there are two areas in which it would be very interesting to study this approach, the first is reinforcement learning. Can you take an already trained policy gradient agent and easily change its structure to expand capacity without removing its current ability. Training reinforcement learning agents on complex tasks can take weeks which means a technique such as this would be very desirable, but also the agent score can be very sensitive to even small changes in its parameters, so it may be this technique does not work. It would need to be studied extensively.

Other area is of future study is running this on recurrent neural networks. Again these can take huge amounts of time to train, so a technique that adjusts structure as it learns would be very useful.

There are also lots of unanswered question around the implementation of the algorithm as presented here. The main algorithm tries every possibility when looking for improvements in size, but there may be smarter ways to know which layers to grow or prune or when to add new layers. I would love the time to experiment more around what signals might tell you when a layer is due to be increased in size. I experimented with this a little, looking into seeing when the reconstruction error that a layer introduced was higher than other layers. But I did have time to dig into this properly.

Also everything in this thesis suffered from the issue of overfitting, given the specifics of this technique what other things can be done to help with overfitting when changing the size of the network. Maybe the gaussian noise that I went with is not the best approach. Maybe a big difference between training and validation error can be a signal that a network is too large and overfitting and be used to trigger layers reducing in size. Or adding more layers.

Another interesting idea is determining when extra layers might be useful, by trying approximate how well a linear layer can reproduce the output of a nonlinear layer, if a linear layer works as a good approximation, by good here I mean that the training error of the network does not change significantly, that suggests the layer is unnecessary and can easily be removed. If on the other hand the layer is very badly approximated by a linear layer then it has significant non-linearity and could indicate that more layers are needed. I again briefly looked into this but did not have time to fully pursue.

Another things that would be interesting to use is looking at the use of highway networks by Srivastava et al (2015) when adding extra layers. Highway networks deal with the issue of when adding extra layers starts to reduce training error. Highway networks have a gate that depending on the input either does or doesn't applying the activation from the new layer. It could very interesting looking into how this gate signal might be used in structure learning. Possibly the propensity of the gate for letting through signals could also be a good indicator for adding or removing layers.

I would also be interested to see what more could be done with the current Tensordynamic library. This was a first pass at dynamically learning structure and at the moment a lot of the structure learning is very discrete, i.e. have structure, train, modify structure, re-train. It would be interesting to play around with techniques that better integrate structure and parameter learning like infinite restricted boltzmann machines.

Also somewhat inspired by the concept of dropout, what can be done around applying dropout to entire layers, say by randomly skipping the feed forward signal between layers.

In some ways what is most interesting about this project is not what it does in terms of the algorithm, which is functional and works, but is not as successful as I might have hoped, but rather the space of unexplored possibilities it opens up. There are so many ideas that could be built of the back of this project.

Another thing missing from this project is this technique's performance in an unsupervised setting. This is completely supported by the library I have just lacked the time to produce results as I’ve focused on the supervised side. Potentially this technique is a very good fit for unsupervised learning.

Also it would be great to plot out in more detail what the distribution of node weights looks like when learning structure progressively compared to what it looks like if training runs with set dimensions from the beginning given the difference in experimental results observed in this project.

8 - Bibliography

- Tieleman & Hinton et al (2012) Coursera slide 29, Lecture 6
- Kingma et al (2014) Adam: A Method for Stochastic Optimization
- Jasper Snoek et al (2012) PRACTICAL BAYESIAN OPTIMIZATION OF MACHINE LEARNING ALGORITHMS
- Whitley et al (1991) Generalization in feed forward neural networks
- Yann le cun et al (1990) Optimal brain damage
- Srivastava et al (2015) Highway Networks
- Agilemethodology.org. (2015)
- Google python style guide https://google.github.io/styleguide/pyguide.html
- Clevert et al (2015) Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)
- Nowlan and Hinton (1992) Simplifying Neural Networks by Soft Weight-Sharing
- Yinan et al (2016) Whiteout: Gaussian Adaptive Noise Regularization in FeedForward Neural Networks
- Hinton et al (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting
- Yarin Gal (2015) Uncertainty in Deep Learning
- Sergey Ioffe et al (2015) Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- Saurabh Karsoliya (2012) Approximating Number of Hidden layer neurons in Multiple Hidden Layer BPNN Architecture
- Hinton et al (2006) A fast learning algorithm for deep belief nets
- Alex Krizhevsky et al (2012) ImageNet Classification with Deep Convolutional Neural Networks
- Kenneth O. Stanley (2004) Efficient Evolution of Neural Networks Through Complexification
- Hassibi et al (1993) Optimal brain surgeon
- Srinivas et al (2015) LEARNING THE ARCHITECTURE OF DEEP NEURAL NETWORKS
- Fahlman (1990) The Cascade-Correlation Learning Architecture
- Vitaly Schetinin (2003) A Learning Algorithm for Evolving Cascade Neural Networks
- Theorie Labor (1994) Reducing Network Depth in the Cascade-Correlation Learning Architecture
- Timur Ash (1989) Dynamic node creation in backpropagation networks
- Alexandre Côté et al (2015) Infinite Restricted Boltzmann Machine
- Tianqi Chen (2016) Net2Net: ACCELERATING LEARNING VIA KNOWLEDGE TRANSFER
- LazyProp https://pypi.python.org/pypi/lazy-property

9 - Appendix

The full code for library and all tests conducted can be found at
https://github.com/DanielSlater/tensordynamic

tensordynamic/**temp.py**

```
import tensorflow as tf
import numpy as np

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.utils import xavier_init


class MyLayer(BaseLayer):
    def get_activation(self):
        return tf.nn.relu(tf.matmul(self.input_layer.get_activation(), self._weights) + self._bias)


class VariationalAutoencoder(object):
    """ Variation Autoencoder (VAE) with an sklearn-like interface implemented using TensorFlow.

    This implementation uses probabilistic encoders and decoders using Gaussian
    distributions and  realized by multi-layer perceptrons. The VAE can be learned
    end-to-end.

    See "Auto-Encoding Variational Bayes" by Kingma and Welling for more details.
    """
    def __init__(self, network_architecture, transfer_fct=tf.nn.softplus,
                 learning_rate=0.001, batch_size=100):
        self.network_architecture = network_architecture
        self.transfer_fct = transfer_fct
        self.learning_rate = learning_rate
        self.batch_size = batch_size

        # tf Graph input
        self.x = tf.placeholder(tf.float32, [None, network_architecture["n_input"]])

        # Create autoencoder network
        self._create_network()
        # Define loss function based variational upper-bound and
        # corresponding optimizer
        self._create_loss_optimizer()

        # Initializing the tensor flow variables
        init = tf.initialize_all_variables()

        # Launch the session
        self.sess = tf.InteractiveSession()
        self.sess.run(init)

    def _create_network(self):
        # Initialize autoencode network weights and biases
        network_weights = self._initialize_weights(**self.network_architecture)
```

```
        # Use recognition network to determine mean and
        # (log) variance of Gaussian distribution in latent
        # space
        self.z_mean, self.z_log_sigma_sq = \
            self._recognition_network(network_weights["weights_recog"],
                                      network_weights["biases_recog"])

        # Draw one sample z from Gaussian distribution
        n_z = self.network_architecture["n_z"]
        eps = tf.random_normal((self.batch_size, n_z), 0, 1,
                               dtype=tf.float32)
        # z = mu + sigma*epsilon
        self.z = tf.add(self.z_mean,
                        tf.mul(tf.sqrt(tf.exp(self.z_log_sigma_sq)), eps))

        # Use generator to determine mean of
        # Bernoulli distribution of reconstructed input
        self.x_reconstr_mean = \
            self._generator_network(network_weights["weights_gener"],
                                    network_weights["biases_gener"])

    def _initialize_weights(self, n_hidden_recog_1, n_hidden_recog_2,
                            n_hidden_gener_1,  n_hidden_gener_2,
                            n_input, n_z):
        all_weights = dict()
        all_weights['weights_recog'] = {
            'h1': tf.Variable(xavier_init(n_input, n_hidden_recog_1)),
            'h2': tf.Variable(xavier_init(n_hidden_recog_1, n_hidden_recog_2)),
            'out_mean': tf.Variable(xavier_init(n_hidden_recog_2, n_z)),
            'out_log_sigma': tf.Variable(xavier_init(n_hidden_recog_2, n_z))}
        all_weights['biases_recog'] = {
            'b1': tf.Variable(tf.zeros([n_hidden_recog_1], dtype=tf.float32)),
            'b2': tf.Variable(tf.zeros([n_hidden_recog_2], dtype=tf.float32)),
            'out_mean': tf.Variable(tf.zeros([n_z], dtype=tf.float32)),
            'out_log_sigma': tf.Variable(tf.zeros([n_z], dtype=tf.float32))}
        all_weights['weights_gener'] = {
            'h1': tf.Variable(xavier_init(n_z, n_hidden_gener_1)),
            'h2': tf.Variable(xavier_init(n_hidden_gener_1, n_hidden_gener_2)),
            'out_mean': tf.Variable(xavier_init(n_hidden_gener_2, n_input)),
            'out_log_sigma': tf.Variable(xavier_init(n_hidden_gener_2, n_input))}
        all_weights['biases_gener'] = {
            'b1': tf.Variable(tf.zeros([n_hidden_gener_1], dtype=tf.float32)),
            'b2': tf.Variable(tf.zeros([n_hidden_gener_2], dtype=tf.float32)),
            'out_mean': tf.Variable(tf.zeros([n_input], dtype=tf.float32)),
            'out_log_sigma': tf.Variable(tf.zeros([n_input], dtype=tf.float32))}
        return all_weights

    def _recognition_network(self, weights, biases):
        # Generate probabilistic encoder (recognition network), which
        # maps inputs onto a normal distribution in latent space.
        # The transformation is parametrized and can be learned.
        layer_1 = self.transfer_fct(tf.add(tf.matmul(self.x, weights['h1']),
                                           biases['b1']))
        layer_2 = self.transfer_fct(tf.add(tf.matmul(layer_1, weights['h2']),
                                           biases['b2']))
        z_mean = tf.add(tf.matmul(layer_2, weights['out_mean']),
                        biases['out_mean'])
        z_log_sigma_sq = \
            tf.add(tf.matmul(layer_2, weights['out_log_sigma']),
                   biases['out_log_sigma'])
```

```
        return (z_mean, z_log_sigma_sq)

    def _generator_network(self, weights, biases):
        # Generate probabilistic decoder (decoder network), which
        # maps points in latent space onto a Bernoulli distribution in data space.
        # The transformation is parametrized and can be learned.
        layer_1 = self.transfer_fct(tf.add(tf.matmul(self.z, weights['h1']),
                                           biases['b1']))
        layer_2 = self.transfer_fct(tf.add(tf.matmul(layer_1, weights['h2']),
                                           biases['b2']))
        x_reconstr_mean = \
            tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights['out_mean']),
                                 biases['out_mean']))
        return x_reconstr_mean

    def _create_loss_optimizer(self):
        # The loss is composed of two terms:
        # 1.) The reconstruction loss (the negative log probability
        #     of the input under the reconstructed Bernoulli distribution
        #     induced by the decoder in the data space).
        #     This can be interpreted as the number of "nats" required
        #     for reconstructing the input when the activation in latent
        #     is given.
        # Adding 1e-10 to avoid evaluatio of log(0.0)
        reconstr_loss = \
            -tf.reduce_sum(self.x * tf.log(1e-10 + self.x_reconstr_mean)
                           + (1-self.x) * tf.log(1e-10 + 1 - self.x_reconstr_mean),
                           1)
        # 2.) The latent loss, which is defined as the Kullback Leibler divergence
        ##    between the distribution in latent space induced by the encoder on
        #     the data and some prior. This acts as a kind of regularizer.
        #     This can be interpreted as the number of "nats" required
        #     for transmitting the the latent space distribution given
        #     the prior.
        latent_loss = -0.5 * tf.reduce_sum(1 + self.z_log_sigma_sq
                                           - tf.square(self.z_mean)
                                           - tf.exp(self.z_log_sigma_sq), 1)
        self.cost = tf.reduce_mean(reconstr_loss + latent_loss)   # average over batch
        # Use ADAM optimizer
        self.optimizer = \
            tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(self.cost)

    def partial_fit(self, X):
        """Train model based on mini-batch of input data.

        Return cost of mini-batch.
        """
        opt, cost = self.sess.run((self.optimizer, self.cost),
                                  feed_dict={self.x: X})
        return cost

    def transform(self, X):
        """Transform data by mapping it into the latent space."""
        # Note: This maps to mean of distribution, we could alternatively
        # sample from Gaussian distribution
        return self.sess.run(self.z_mean, feed_dict={self.x: X})

    def generate(self, z_mu=None):
        """ Generate data by sampling from latent space.
```

```
        If z_mu is not None, data for this point in latent space is
        generated. Otherwise, z_mu is drawn from prior in latent
        space.
        """
        if z_mu is None:
            z_mu = np.random.normal(size=self.network_architecture["n_z"])
        # Note: This maps to mean of distribution, we could alternatively
        # sample from Gaussian distribution
        return self.sess.run(self.x_reconstr_mean,
                             feed_dict={self.z: z_mu})

    def reconstruct(self, X):
        """ Use VAE to reconstruct given data. """
        return self.sess.run(self.x_reconstr_mean,
                             feed_dict={self.x: X})


def train(network_architecture, learning_rate=0.001,
          batch_size=100, training_epochs=10, display_step=5):
    vae = VariationalAutoencoder(network_architecture,
                                 learning_rate=learning_rate,
                                 batch_size=batch_size)
    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(n_samples / batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, _ = mnist.train.next_batch(batch_size)

            # Fit training using batch data
            cost = vae.partial_fit(batch_xs)
            # Compute average loss
            avg_cost += cost / n_samples * batch_size

        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), \
                  "cost=", "{:.9f}".format(avg_cost)
    return vae
```

tensordynamic/tensor_dynamic/**bayesian_resizing_net.py**

```
import logging
import sys
from math import log

import tensorflow as tf
from enum import Enum

from tensor_dynamic.layers.flatten_layer import FlattenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.output_layer import OutputLayer
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer

logger = logging.getLogger(__name__)


class EDataType(Enum):
  TRAIN = 0
  TEST = 1
  VALIDATION = 2


def create_flat_network(data_set_collection, hidden_layers, session, regularizer_coeff=0.01,
            batch_normalize_input=True,
            activation_func=tf.nn.relu,
            input_noise_std=None):
  """Create a network of connected flat layers with sigmoid activation func

  Args:
    hidden_layers (tuple of int): First int is number of input nodes, then each hidden layer, final is output layer
    session (tf.Session):
    regularizer_coeff (float):

  Returns:
    OutputLayer
  """
  last_layer = InputLayer(data_set_collection.features_shape)

  if len(last_layer.output_nodes) > 1:
```

```
        last_layer = FlattenLayer(last_layer, session)

    for hidden_nodes in hidden_layers:
        last_layer = HiddenLayer(last_layer, hidden_nodes, session, non_liniarity=activation_func,
                                 layer_noise_std=input_noise_std,
                                 batch_normalize_input=batch_normalize_input)

    output = CategoricalOutputLayer(last_layer, data_set_collection.labels_shape, session,
                                    regularizer_weighting=regularizer_coeff,
                                    batch_normalize_input=batch_normalize_input)
    return output


class BayesianResizingNet(object):
    GROWTH_MULTIPLYER = 1.1
    SHRINK_MULTIPLYER = 1. / GROWTH_MULTIPLYER
    MINIMUM_GROW_AMOUNT = 3

    def __init__(self, output_layer, model_selection_data_type=EDataType.TEST):
        if not isinstance(output_layer, OutputLayer):
            raise TypeError("resizable_net must implement AbstractResizableNet")
        self._output_layer = output_layer
        self.model_selection_data_type = model_selection_data_type

    def run(self, data_set_collection, initial_learning_rate=0.01, tuning_learning_rate=0.001):
        """Train the network to find the best size

        Args:
            tuning_learning_rate (float):
            initial_learning_rate (float):
            data_set_collection (tensor_dynamic.data.data_set_collection.DataSetCollection):
        """
        # DataSet must be multi-model for now
        self._output_layer.train_till_convergence(data_set_collection.train,
                                                  self.get_evaluation_data_set(data_set_collection),
                                                  learning_rate=initial_learning_rate)
        best_score = self.model_weight_score(self._output_layer,
self.get_evaluation_data_set(data_set_collection))
        best_dimensions = self._output_layer.get_resizable_dimension_size_all_layers()

        logger.info("starting dim %s score %s", best_score, best_dimensions)

        unresized_layers = list(self._output_layer.get_all_resizable_layers())

        if len(unresized_layers) == 0:
            raise Exception("Found no layers to resize")

        current_resize_target = unresized_layers[0]

        while True:
            resized, new_best_score = current_resize_target.find_best_size(data_set_collection.train,
                                                                           self.get_evaluation_data_set(
                                                                               data_set_collection),
                                                                           self.model_weight_score,
                                                                           best_score=best_score,
                                                                           initial_learning_rate=initial_learning_rate,
                                                                           tuning_learning_rate=tuning_learning_rate)
            if resized:
                best_score = new_best_score
                layers_unsuccessfully_resized = 0
```

```
                if len(unresized_layers) == 1:
                    break
                else:
                    layers_unsuccessfully_resized += 1
                    if layers_unsuccessfully_resized >= len(unresized_layers):
                        # we are done resizing
                        break

            index = unresized_layers.index(current_resize_target) + 1
            if index >= len(unresized_layers):
                unresized_layers[0]
            else:
                unresized_layers[index]

        # TOOD adding layers
        logger.info("Finished with best:%s dims:%s", best_score,
                    self._output_layer.get_resizable_dimension_size_all_layers())

    def get_evaluation_data_set(self, data_set):
        if self.model_selection_data_type == EDataType.TRAIN:
            return data_set.train
        elif self.model_selection_data_type == EDataType.TEST:
            return data_set.test
        elif self.model_selection_data_type == EDataType.VALIDATION:
            return data_set.validation
        else:
            raise Exception("unknown model_selection_data_type %s",
self._output_layer.model_selection_data_type)

    @staticmethod
    def model_weight_score(layer, evaluation_data_set):
        evaluation_features = evaluation_data_set.features
        evaluation_labels = evaluation_data_set.labels

        log_liklihood = log_probability_of_targets_given_weights_multimodal(
            lambda x: layer.last_layer.activate_predict(x),
            evaluation_features,
            evaluation_labels)
        model_parameters = layer.get_parameters_all_layers()
        return bayesian_model_selection(log_liklihood, model_parameters)


def log_probability_of_targets_given_weights_multimodal(network_prediction_function, inputs, targets):
    predictions = network_prediction_function(inputs)

    result = 0.
    for i in range(len(predictions)):
        result += log(sum(predictions[i] * targets[i]))

    return result


def bayesian_model_selection(log_liklihood, number_of_parameters):
    logger.info("log_liklihood %s number_of_parameters %s", log_liklihood, number_of_parameters)
    return log_liklihood - log(number_of_parameters)


if __name__ == '__main__':
    logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
    import tensor_dynamic.data.mnist_data as mnist
```

```
    data_set = mnist.get_mnist_data_set_collection("data/MNIST_data", one_hot=True, limit_train_size=1000)

    with tf.Session() as session:
        brn = BayesianResizingNet(create_flat_network((784, 5, 10), session))
        brn.run(data_set)

        # when running with 10 starting nodes
        # INFO:tensor_dynamic.utils:finished with best error 2630.17977381
        # INFO:__main__:new dim -9335.37930272 score [784, 60, 10]
```

tensordynamic/tensor_dynamic/**categorical_trainer.py**

```
import itertools
import tensorflow as tf

# I think this can be depricated
class CategoricalTrainer(object):
    def __init__(self, net, learn_rate):
        """Sets up an optimizer and various helper methods against a network

        Args:
            net (tensorflow_dynamic.layers.BaseLayer): Net that we will be training against
            learn_rate (float):
        """
        self._net = net
        self._target_placeholder = tf.placeholder(tf.float32, shape=net.output_shape)
        self._learn_rate_placeholder = tf.placeholder("float", shape=[], name="learn_rate")
        self._cost = self._net.cost_all_layers_train(self._target_placeholder)
        self._prediction = tf.argmax(self._net.activation_predict, 1)
        self._correct_prediction = tf.equal(self._prediction, tf.argmax(self._target_placeholder, 1))
        self._accuracy = tf.reduce_mean(tf.cast(self._correct_prediction, "float")) * tf.constant(100.0)
```

```
        optimizer = tf.train.GradientDescentOptimizer(self._learn_rate_placeholder).minimize(self._cost)

        # temp = set(tf.all_variables())
        # optimizer = tf.train.AdamOptimizer()
        # self.net.session.run(tf.initialize_variables(set(tf.all_variables()) - temp))

        assigns = [x.assign_op for x in net.all_layers if x.assign_op is not None]
        if assigns:
            assigns_group = tf.group(*assigns)
            with tf.control_dependencies([optimizer]):
                optimizer = tf.group(assigns_group)

        self._train = optimizer
        self.learn_rate = learn_rate

        # adam_optimizer_variables = itertools.chain(*[x.values() for x in optimizer._slots.values()])
        # self.net.session.run(tf.initialize_variables(adam_optimizer_variables))
        # self.net.session.run(tf.initialize_variables([optimizer._beta1_t, optimizer._beta2_t, optimizer._epsilon_t,
        #                                               optimizer._lr_t]))

    @property
    def net(self):
        return self._net

    def predict(self, input_data):
        return self._net.session.run(self._prediction, feed_dict={self._net.input_placeholder: input_data})

    def accuracy(self, input_data, labels):
        return self._net.session.run([self._accuracy, self._cost],
                                     feed_dict={self._net.input_placeholder: input_data,
                                                self._target_placeholder: labels})

    def train(self, input_data, labels):
        _, cost = self._net.session.run([self._train, self._cost],
                                        feed_dict={self._net.input_placeholder: input_data,
                                                   self._target_placeholder: labels,
                                                   self._learn_rate_placeholder: self.learn_rate})

        return cost

    def train_one_epoch(self, data_set, batch_size):
        start_epoch = data_set.train.epochs_completed
        cost = 0.
        while start_epoch == data_set.train.epochs_completed:
            train_x, train_y = data_set.train.next_batch(batch_size)
            cost += self.train(train_x, train_y)

        return cost

    def back_losses_per_layer(self, input_data, misclassification_only=False, labels=None):
        """The loss per bactivating layer

        Args:
            input_data (np.array):
            misclassification_only (bool): If True back loss is only checked on data that has been misclassified
            labels (np.array): Labels for the input data, required if using misclassification_only

        Returns:
            {tensorflow_dynamic.layers.BaseLayer, float}
        """
```

```
back_losses = [(layer, layer.unsupervised_cost_predict()) for layer in self._net.all_connected_layers if
        layer.bactivate]

if not back_losses:
  return None

if misclassification_only:
  if not labels:
    raise Exception('must supply labels when using misclassification_only')

  predictions = self._net.session.run(self._correct_prediction,
                    feed_dict={self._net.input_placeholder: input_data,
                         self._target_placeholder: labels})

  input_data = [item for item, prediction in zip(input_data, predictions) if prediction <= 0.0]

results = self._net.session.run([a[1] for a in back_losses],
                 feed_dict={self._net.input_placeholder: input_data})
return dict((a[0][0], a[1] / a[0][0].input_nodes) for a in zip(back_losses, results))
```

tensordynamic/tensor_dynamic/**data_functions.py**

```
import random
from collections import defaultdict

import numpy as np


def parity_fn(input_data):
    ones = 0
    for x in input_data:
        if x > 0.5:
            ones += 1
    return [ones % 2]


def symmetry_fn(input_data):
    for x, y in zip(input_data, reversed(input_data)):
        if x != y:
            return [0.0]
    return [1.0]


def last_bit_fn(input_data):
    if input_data[-1]:
        return [1.0, 0.0]
    else:
        return [0.0, 1.0]


def identity_fn(input_data):
    return input_data


def one_fn(input_data):
    return [1.0, 0.0]


def shuffle(in_x, in_y):
    collection = zip(in_x, in_y)
    random.shuffle(collection)
    out_x = np.array([x for x, y in collection])
    out_y = np.array([y for x, y in collection])
    return out_x, out_y


def _create_dataset(fn, input_size, length, even_classes=False):
    X_train = []
    y_train = []
    class_counts = defaultdict(int)
    for i in range(length):
        X = np.random.binomial(1, 0.5, size=input_size)
        y = fn(X)
```

```
        if even_classes:
            y_class = np.argmax(y)
            if i > length / len(y):
                if y_class == max(class_counts, key=class_counts.get):
                    continue
            class_counts[max] += 1

        X_train.append(X)
        y_train.append(y)

    if even_classes:
        collection = zip(X_train, y_train)
        random.shuffle(collection)
        X_train = [x for x, y in collection]
        y_train = [y for x, y in collection]

    return np.array(X_train, dtype='float32') / 1.0, np.array(y_train, dtype='float32') / 1.0


def create_dataset(function, input_size, dataset_size, validation_percent=0.25, test_percent=0.25,
                   even_test_classes=False):
    """Create a dataset from a function

    Args:
        function ([float] -> [float]): function that takes an input of random floats and outputs a transformation of
them,
            can change the size
        input_size (int): length of array that the function should take as an input
        dataset_size (int): total number of rows of data to generate across all data sets
        validation_percent (float): percent of data generated to go into the validation data set
        test_percent (float): percent of data generated to go into the test data set
        even_test_classes (bool): If True then it will garentee that all datasets have equal numbers of classes

    Returns:
        numpy.array, numpy.array, numpy.array, numpy.array, numpy.array, numpy.array
    """
    train_x, train_y = _create_dataset(function, input_size, int(dataset_size * (1.0 - validation_percent -
test_percent)),
                                       even_classes=even_test_classes)
    val_x, val_y = _create_dataset(function, input_size, int(dataset_size * validation_percent))
    test_x, test_y = _create_dataset(function, input_size, int(dataset_size * test_percent))

    return train_x, train_y, val_x, val_y, test_x, test_y

XOR_INPUTS = np.array([[1.0, -1.0],
                       [-1.0, -1.0],
                       [-1.0, 1.0],
                       [1.0, 1.0]], dtype=np.float32)
XOR_TARGETS = np.array([[1.0],
                        [-1.0],
                        [1.0],
                        [-1.0]], dtype=np.float32)

DOUBLE_XOR_INPUTS = np.array([[-1.0, -1.0, -1.0],
                              [-1.0, -1.0, 1.0],
                              [-1.0, 1.0, -1.0],
                              [-1.0, 1.0, 1.0],
                              [1.0, -1.0, -1.0],
                              [1.0, -1.0, 1.0],
                              [1.0, 1.0, -1.0],
```

```
                           [1.0, 1.0, 1.0], ], dtype=np.float32)
DOUBLE_XOR_TARGETS = np.array([[1.0],
                           [1.0],
                           [1.0],
                           [-1.0],
                           [-1.0],
                           [1.0],
                           [1.0],
                           [-1.0]], dtype=np.float32)


def xor_sig_ds():
    return XOR_INPUTS / 2.0 + 0.5, XOR_TARGETS / 2.0 + 0.5


def double_xor_sig_ds():
    return DOUBLE_XOR_INPUTS / 2.0 + 0.5, DOUBLE_XOR_TARGETS / 2.0 + 0.5


def xor_tan_ds():
    return XOR_INPUTS, XOR_TARGETS


def double_xor_tan_ds():
    return DOUBLE_XOR_INPUTS, DOUBLE_XOR_TARGETS


def k_nearest_eculidian_dist(main, others, k=1):
    dists = []

    for item in others:
        diff = main-item
        dist = np.sum(diff*diff)
        dists.append(dist)

    dists.sort()

    return sum(dists[:k])


def pearson_correlation_1vsMany(main, others):
    """data1 & data2 should be numpy arrays."""
    result = []
    main_mean = main.mean()
    main_std = main.std()
    if main_std == 0.0 or main_mean == 0.0:
        return [1000.0] * len(others)

    for data in others:
        mean2 = data.mean()
        std2 = data.std()
        corr = ((main*data).mean()-main_mean*mean2)/(main_std*std2)
        result.append(corr)

    return result


def one_hot(data):
    min_col = min([row[0] for row in data])
    max_col = max([row[0] for row in data])
```

```
    range = max_col-min_col
    results = []
    for row in data:
        one_hot_row = [0.0]*(range+1)
        one_hot_row[row[0]-min_col] = 1.0
        results.append(one_hot_row)
    return results


def normalize(data):
    """Normalize a dataset so the values in all rows are between 0 and 1

    Args:
        data ([float]):

    Returns:
        [float]
    """
    minmax = []
    for i in range(len(data[0])):
        min_col = min([row[i] for row in data])
        max_col = max([row[i] for row in data])
        minmax.append((min_col, max_col, max_col-min_col))

    normalized = []

    for row in data:
        norm_row = [(r-m[0])/m[2] for r, m in zip(row, minmax)]
        normalized.append(norm_row)

    return normalized

if __name__ == '__main__':
    train_x, train_y, _, _, _, _ = create_dataset(symmetry_fn, 10, 100, even_test_classes=True)
    print np.mean(train_y)
```

tensordynamic/tensor_dynamic/**lazyprop.py**

```
from collections import defaultdict

_LAZY_PROP_VALUES = '__lazy_prop_values__'
_LAZY_PROP_SUBSCRIBERS = '__lazy_prop_subscribers__'


def lazyprop(fn):
    """A Python Property that will be evaluated once when first called. The result of this is cached and then
returned on
    each subsequent call

    Examples:
        class A:
            @lazyprop
            def do_thing(self):
                return fib(2000)

        Because of lazyprop if you call do_thing twice the first time the value will be cached, then subsequent calls
        Will return the cached version saving having to compute fib(2000) again

    Args:
        fn (class method): Will be made into a lazy prop

    Returns:
        (method as lazy prop)
    """

    @property
    def _lazyprop(self):
        if not hasattr(self, _LAZY_PROP_VALUES):
            setattr(self, _LAZY_PROP_VALUES, {})
        lazy_props_dict = self.__dict__[_LAZY_PROP_VALUES]
        if fn.__name__ not in lazy_props_dict:
            lazy_props_dict[fn.__name__] = fn(self)
        return lazy_props_dict[fn.__name__]

    return _lazyprop


def has_lazyprop(object, property_name):
    """Returns True if this lazyprop has been instanstiated
```

```
    Args:
        object (object):
        property_name (str):

    Returns:
        bool
    """
    if hasattr(object, _LAZY_PROP_VALUES):
        return property_name in object.__dict__[_LAZY_PROP_VALUES]
    return False


def subscribe_to_lazy_prop(object, property_name, on_change_func):
    """If the passed in lazyprop is ever cleared the function passed in is called

    Args:
        object (object):
        property_name (str):
        on_change_func (object -> None): function to be called when the lazy prop is cleared, the object is passed
in
            as the first arg
    """
    assert isinstance(property_name, str)

    if not hasattr(object, _LAZY_PROP_SUBSCRIBERS):
        setattr(object, _LAZY_PROP_SUBSCRIBERS, defaultdict(lambda: set()))

    object.__dict__[_LAZY_PROP_SUBSCRIBERS][property_name].add(on_change_func)


def unsubscribe_from_lazy_prop(object, property_name, on_change_func):
    """Stop the function from being called if the lazyprop is cleared

    Args:
        object (object):
        property_name (str):
        on_change_func (object -> None): function to cancel when the lazy prop is cleared, the object is passed in
            as the first arg
    """
    assert isinstance(property_name, str)

    if hasattr(object, _LAZY_PROP_SUBSCRIBERS):
        object.__dict__[_LAZY_PROP_SUBSCRIBERS][property_name].remove(on_change_func)


def clear_lazyprop(object, property_name):
    """Clear the named lazyprop from this object

    Args:
        object (object):
        property_name (str):
    """
    assert isinstance(property_name, str)

    if _LAZY_PROP_VALUES in object.__dict__:
        if property_name in object.__dict__[_LAZY_PROP_VALUES]:
            del object.__dict__[_LAZY_PROP_VALUES][property_name]

    if _LAZY_PROP_SUBSCRIBERS in object.__dict__:
```

```
        if property_name in object.__dict__[_LAZY_PROP_SUBSCRIBERS]:
            for fn in object.__dict__[_LAZY_PROP_SUBSCRIBERS][property_name]:
                fn(object)


def clear_all_lazyprops(object):
    """Clears all lazy prop from an object. This means they will be re-evaluated next time they are run

    Args:
        object (object): The object we want to clear the lazy props from
    """
    if _LAZY_PROP_VALUES in object.__dict__:
        del object.__dict__[_LAZY_PROP_VALUES]

    if _LAZY_PROP_SUBSCRIBERS in object.__dict__:
        for subscribers in object.__dict__[_LAZY_PROP_SUBSCRIBERS].values():
            for fn in subscribers:
                fn(object)


def clear_lazyprop_on_lazyprop_cleared(subscriber_object, subscriber_lazyprop,
                                       listen_to_object, listen_to_lazyprop=None):
    """Clear the lazyprop on the subscriber_object if the listen_to_object property is cleared

    Args:
        subscriber_object (object):
        subscriber_lazyprop (str):
        listen_to_object (object):
        listen_to_lazyprop (str):
    """
    if listen_to_lazyprop is None:
        listen_to_lazyprop = subscriber_lazyprop

    assert isinstance(listen_to_lazyprop, str)
    assert isinstance(subscriber_lazyprop, str)

    subscribe_to_lazy_prop(listen_to_object, listen_to_lazyprop,
                           lambda _: clear_lazyprop(subscriber_object, subscriber_lazyprop))
```

tensordynamic/tensor_dynamic/**node_importance.py**

```
import numpy as np


def node_importance_by_dummy_activation_from_input_layer(layer, data_set_train, data_set_validation):
    shape = (1,) + tuple(int(x) for x in layer.input_placeholder.get_shape()[1:])
    all_pos_1 = np.ones(shape=shape, dtype=np.float32)

    all_zero = np.zeros(shape=shape, dtype=np.float32)

    all_neg_1 = -np.ones(shape=shape, dtype=np.float32)

    importance = layer._session.run(layer.activation_predict,
```

```
                                feed_dict={layer.input_placeholder:
                                    np.append(np.append(all_pos_1, all_zero, axis=0), all_neg_1,
                                        axis=0)})

    return np.sum(importance, axis=0)


def node_importance_by_real_activation_from_input_layer(layer, data_set_train, data_set_validation):
    data_set = data_set_train or data_set_validation
    if data_set is not None:
        importance = layer._session.run(layer.activation_predict,
                                feed_dict={layer.input_placeholder:
                                    data_set.features})

        return np.sum(importance, axis=0)
    else:
        return node_importance_random(layer, data_set, data_set_validation)


def node_importance_by_real_activation_from_input_layer_variance(layer, data_set_train, data_set_validation):
    data_set = data_set_train or data_set_validation
    if data_set is not None:
        importance = layer._session.run(layer.activation_predict,
                                feed_dict={layer.input_placeholder:
                                    data_set.features})
        return np.var(importance, axis=0)
    else:
        return node_importance_random(layer, data_set, data_set_validation)


def node_importance_by_square_sum(layer, data_set_train, data_set_validation):
    data_set = data_set_train or data_set_validation
    # TODO by bound variable
    weights, bias = layer._session.run([layer._weights, layer._bias])

    return np.sum(np.square(weights), axis=0) + np.square(bias)


def node_importance_random(layer, data_set_train, data_set_validation):
    return np.random.normal(size=(layer.get_resizable_dimension_size()))


def node_importance_by_removal(layer, data_set_train, data_set_validation):
    data_set = data_set_train or data_set_validation

    # TODO by bound variable
    if data_set is None:
        return node_importance_random(layer, data_set)

    base_error = layer._session.run(layer.last_layer.target_loss_op_predict,
                                feed_dict={layer.input_placeholder:
                                    data_set.features,
                                  layer.target_placeholder:
                                    data_set.labels})

    weights, bias = layer._session.run([layer._weights, layer._bias])

    errors = []
    for i in range(layer.get_resizable_dimension_size()):
        # null node
```

```
        new_bias = np.copy(bias)
        new_bias[i] = 0.

        new_weights = np.copy(weights)
        new_weights[:, i] = 0.
        layer.weights = new_weights
        layer.bias = new_bias

        # layer._session.run([tf.assign(layer._weights, new_weights), tf.assign(layer._bias, new_bias)])
        error_without_node = layer.session.run(layer.last_layer.target_loss_op_predict,
                                               feed_dict={layer.input_placeholder:
                                                              data_set.features,
                                                          layer.target_placeholder:
                                                              data_set.labels})
        errors.append(base_error - error_without_node)

    layer.weights = weights
    layer.bias = bias

    return errors


def node_importance_optimal_brain_damage(layer, data_set_train, data_set_validation):
    """ Determines node importance based on Optimal brain damage algorithm
http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf this
    method can be used to determine which nodes should be pruned when reducing the size of a layer, or which
should be split when increasing
    the number of nodes

    Args:
        layer (BaseLayer): Subclass of base layer that we are ran
        data_set_train (DataSet): data set used for training
        data_set_validation (DataSet): data set used for validation

    Returns:
        np.array : A 1-d array with the same number of elements as there are output nodes for the layer
    """
    data_set = data_set_train or data_set_validation

    if data_set is None:
        return node_importance_random(layer, data_set, data_set_validation)

    weights_hessian_op, bias_hessian_op = layer.hessien_with_respect_to_error_op

    weights, bias, weights_hessian, bias_hessian = layer.session.run(
        [layer._weights, layer._bias, weights_hessian_op, bias_hessian_op],
        feed_dict={layer.input_placeholder: data_set.features,
                   layer.target_placeholder: data_set.labels}
    )

    weights_squared = np.square(weights)
    bias_squared = np.square(bias)

    return np.sum(weights_squared * weights_hessian, axis=0) + bias_squared * bias_hessian


def node_importance_full_taylor_series(layer, data_set_train, data_set_validation):
    data_set = data_set_validation

    if data_set is None:
```

```
        return node_importance_random(layer, data_set, data_set_validation)

    weights_jacobean_op, bias_jacobean_op = layer.gradients_with_respect_to_error_op
    weights_hessian_op, bias_hessian_op = layer.hessien_with_respect_to_error_op

    weights, bias, weights_jacobean, bias_jacobean, weights_hessian, bias_hessian = layer.session.run(
        [layer._weights, layer._bias, weights_jacobean_op, bias_jacobean_op, weights_hessian_op,
bias_hessian_op],
        feed_dict={layer.input_placeholder: data_set.features,
                   layer.target_placeholder: data_set.labels}
    )

    weights_squared = np.square(weights)
    bias_squared = np.square(bias)

    return np.sum((weights_squared * weights_hessian) * .5 + weights * weights_jacobean,
                  axis=0) + (bias_squared * bias_hessian) * .5 + bias * bias_jacobean


def node_importance_error_derrivative(layer, data_set_train, data_set_validation):
    data_set = data_set_validation

    if data_set is None:
        return node_importance_random(layer, data_set, data_set_validation)

    weights_jacobean_op, bias_jacobean_op = layer.gradients_with_respect_to_error_op

    weights, bias, weights_jacobean, bias_jacobean = layer.session.run(
        [layer._weights, layer._bias, weights_jacobean_op, bias_jacobean_op],
        feed_dict={layer.input_placeholder: data_set.features,
                   layer.target_placeholder: data_set.labels}
    )

    return np.sum(weights * weights_jacobean, axis=0) + bias * bias_jacobean
```

tensordynamic/tensor_dynamic/**tf_loss_functions.py**

```
import tensorflow as tf


def squared_loss(x, y):
    return tf.reduce_mean(tf.square(x - y))


def cross_entropy_loss(x, y):
    return -tf.reduce_mean((x * tf.log(y) + (1 - x) * tf.log(1 - y)))
```

tensordynamic/tensor_dynamic/**train_policy.py**

```
import operator

import sys
```

```
from tensor_dynamic.layers.duel_state_relu_layer import DuelStateReluLayer
from utils import train_till_convergence


class TrainPolicy(object):
    def __init__(self, trainer, data_set, batch_size=100,
                 max_iterations=10000,
                 max_hidden_nodes=None,
                 stop_accuracy=None,
                 grow_after_turns_without_improvement=None,
                 start_grow_epoch=None,
                 learn_rate_decay=1.,
                 learn_rate_boost=None,
                 back_loss_on_misclassified_only=False):
        """Class for training networks

        Args:
            trainer (tensor_dynamic.CategoricalTrainer):
            data_set:
            batch_size (int):
            max_iterations (int):
            max_hidden_nodes:
            stop_accuracy:
            grow_after_turns_without_improvement:
            start_grow_epoch:
            learn_rate_decay (float):
            learn_rate_boost (float):
            back_loss_on_misclassified_only (bool):

        Returns:

        """
        self.learn_rate_decay = learn_rate_decay
        self.batch_size = batch_size
        self._data_set = data_set
        self._trainer = trainer
        self.max_iterations = max_iterations
        self.max_hidden_nodes = max_hidden_nodes
        self.stop_accuracy = stop_accuracy
        self.grow_after_epochs_without_improvement = grow_after_turns_without_improvement
        self.start_grow_epoch = start_grow_epoch
        self.learn_rate_boost = learn_rate_boost
        self.back_loss_on_misclassified_only = back_loss_on_misclassified_only

    def train_one_epoch(self):
        cost = self._trainer.train_one_epoch(self._data_set, self.batch_size)
        self._trainer.learn_rate *= self.learn_rate_decay

        return cost

    def train_till_convergence(self, continue_epochs=3, use_validation=True, max_epochs=10000):
        if use_validation:
            def train_one_epoch_validation():
                self.train_one_epoch()
                _, validation_loss = self._trainer.accuracy(self._data_set.validation)
                return validation_loss

            train_till_convergence(train_one_epoch_validation, continue_epochs=continue_epochs,
max_epochs=max_epochs)
        else:
```

```
            train_till_convergence(self.train_one_epoch, continue_epochs=continue_epochs,
max_epochs=max_epochs)

    @property
    def validation_accuracy(self):
        self._trainer.predict(self._data_set.validation.features, self._data_set.validation.labels)

    def run_full(self, verbose=True):
        best_validation_loss = sys.float_info.max
        epochs_since_validation_improvement = 0

        if self.start_grow_epoch:
            for _ in range(self.start_grow_epoch):
                train_loss = self.train_one_epoch()
                print("burn in train loss %s" % train_loss)

        while True:
            train_loss = self.train_one_epoch()
            validation_accuracy, validation_loss = self._trainer.accuracy(self._data_set.validation)
            if verbose:
                print(self._data_set.train.epochs_completed, train_loss, validation_accuracy, validation_loss)

            if self.stop_accuracy and validation_accuracy >= self.stop_accuracy:
                print("hit stopping accuracy with validation score of %s" % validation_accuracy)
                return

            if self._data_set.train.epochs_completed >= self.max_iterations:
                print("hit max iterations %s" % self.max_iterations)
                return

            if best_validation_loss > validation_loss:
                print("new best validation loss %s" % validation_loss)
                best_validation_loss = validation_loss
                epochs_since_validation_improvement = 0
            elif self.grow_after_epochs_without_improvement and epochs_since_validation_improvement >
self.grow_after_epochs_without_improvement:
                if self.max_hidden_nodes and sum(
                        [x.output_nodes for x in self._trainer.net.all_layers]) > self.max_hidden_nodes:
                    print("hit stopping number of hidden nodes %s" % self.max_hidden_nodes)
                    return

                if not self.grow_net():
                    print("stopped because we did not grow")
                    return

                epochs_since_validation_improvement = 0
                best_validation_loss = 10000000.
                if self.learn_rate_boost:
                    self._trainer.learn_rate += self.learn_rate_boost
            else:
                epochs_since_validation_improvement += 1

    def grow_net(self):
        # find layer with highest reconstruction error
        if self.back_loss_on_misclassified_only:
            back_losses_per_layer = self._trainer.back_losses_per_layer(self._data_set.train.features)
        else:
            back_losses_per_layer = self._trainer.back_losses_per_layer(self._data_set.train.features,
                                                                        misclassification_only=True,
                                                                        labels=self._data_set.train.labels)
```

```
        print("back_losses %s" % [(k.layer_number, v) for k, v in back_losses_per_layer.iteritems()])
        max_layer = max(back_losses_per_layer.iteritems(), key=operator.itemgetter(1))[0]
        print("adding node to layer %s", max_layer.layer_number)
        # TODO: reset best validation loss?
        max_layer.resize(max_layer.output_nodes + 1)
        print("New shape = %s", [x.output_nodes for x in self._trainer.net.all_layers])

        return True


class DuelStateReluTrainPolicy(TrainPolicy):
    def __init__(self, trainer, data_set, batch_size,
                 max_iterations=10000,
                 max_hidden_nodes=None,
                 stop_accuracy=None,
                 grow_after_turns_without_improvement=None,
                 start_grow_epoch=None,
                 learn_rate_decay=1.,
                 learn_rate_boost=None,
                 # back_loss_on_misclassified_only=False
                 ):
        super(DuelStateReluTrainPolicy, self).__init__(trainer, data_set, batch_size,
                                                       max_iterations, max_hidden_nodes, stop_accuracy,
                                                       grow_after_turns_without_improvement,
                                                       start_grow_epoch,
                                                       learn_rate_decay,
                                                       learn_rate_boost)

    def grow_net(self):
        duel_state_relu_layers = [x for x in self._trainer.net.all_layers if isinstance(x, DuelStateReluLayer)]
        grew = False
        for layer in duel_state_relu_layers:
            if layer.inactive_nodes() == 0:
                # this layer has no inactive nodes so add one grow
                layer.resize(layer.output_nodes + 1)
                grew = True
                print("New shape = %s", [x.output_nodes for x in self._trainer.net.all_layers])
            else:
                print("Have inactive nodes")

        return grew
```

tensordynamic/tensor_dynamic/**utils.py**

```
import logging

import itertools
import numpy as np
import tensorflow as tf
from collections import Iterable
from tensorflow.python.framework.tensor_shape import TensorShape

logger = logging.getLogger(__name__)


def xavier_init(fan_in, fan_out, constant=1.0):
    """ Xavier initialization of network weights
    https://stackoverflow.com/questions/33640581/how-to-do-xavier-initialization-on-tensorflow

    Args:
        fan_in (int | tuple of ints): Number of input connections to this matrix
        fan_out (int | tuple of ints): Number of output connections from this matrix
        constant (float32): Scales the output

    Returns:
        tensorflow.Tensor: A tensor of the specified shape filled with random uniform values.
    """
    if isinstance(fan_in, Iterable):
        fan_in = get_product_of_iterable(fan_in)
    if isinstance(fan_out, Iterable):
        fan_out = get_product_of_iterable(fan_out)

    low = -constant * np.sqrt(6.0 / (fan_in + fan_out))
    high = constant * np.sqrt(6.0 / (fan_in + fan_out))
    return tf.random_uniform((fan_in, fan_out),
                             minval=low, maxval=high,
                             dtype=tf.float32)


def weight_init(shape, constant=1.0):
    fan_in = get_product_of_iterable(shape[:-1])
    fan_out = get_product_of_iterable(shape[-1:])

    low = -constant * np.sqrt(1.0 / (fan_in + fan_out))
    high = constant * np.sqrt(1.0 / (fan_in + fan_out))
    return tf.random_uniform(shape,
                             minval=low, maxval=high,
```

```
                  dtype=tf.float32)


def bias_init(shape, constant=0.1):
    if isinstance(shape, int):
        shape = (shape,)
    return tf.constant(constant, shape=shape, dtype=tf.float32)


def get_product_of_iterable(iterable):
    """Product of the items in the input e.g. [1,2,3,4] => 24

    Args:
        iterable (iterable of ints):

    Returns:
        int
    """
    product = 1
    for x in iterable:
        product *= x
    return product


def tf_resize(session, tensor, new_dimensions=None, new_values=None, assign_function=None):
    """Resize a tensor or variable

    Args:
        assign_function (tensorflow.Operation): Operation for assigning this variable, this is to stop the graph
            getting overloaded
        session (tensorflow.Session): The session within which this variable resides
        tensor (tensorflow.Tensor or tensorflow.Variable): The variable or tensor we wish to resize
        new_dimensions ([int]): The dimensions we want the tensor transformed to. If None will be set to the dims
of the new_values array
        new_values (numpy.array): If passed then these values are given to the resized tensor
    """
    if new_values is not None and new_dimensions is not None:
        if tuple(new_dimensions) != new_values.shape:
            raise ValueError("new_dimsensions and new_values, if set, must have the same shape")

    if new_dimensions is None and new_values is not None:
        new_dimensions = new_values.shape

    if new_values is not None:
        if hasattr(new_values, '__call__'):
            new_values = new_values()

        if assign_function is None:
            assign = tf.assign(tensor, new_values, validate_shape=False)
            session.run(assign)
        else:
            assign_function(new_values)
    elif isinstance(tensor, tf.Variable):
        current_vals = session.run(tensor)
        new_values = np.resize(current_vals, new_dimensions)

        if assign_function is None:
            assign = tf.assign(tensor, new_values, validate_shape=False)
            session.run(assign)
        else:
```

```
            assign_function(new_values)

    old_dimensions = tuple(tensor.get_shape().as_list())

    if old_dimensions != new_dimensions:
        if hasattr(tensor, '_variable'):
            modify_shape(tensor._variable._shape, new_dimensions)
            modify_shape(tensor._snapshot._shape, new_dimensions)
            modify_shape(tensor._initial_value._shape, new_dimensions)

            # new_shape = TensorShape(new_dimensions)
            #
            # tensor._snapshot._shape = new_shape
            # tensor._variable._shape = new_shape
            # tensor._initial_value._shape = new_shape
        elif hasattr(tensor, '_shape'):
            modify_shape(tensor._shape, new_dimensions)
        else:
            raise NotImplementedError('unrecognized type %s' % type(tensor))

        for output in tensor.op.outputs:
            modify_shape(output._shape, new_dimensions)

        _chain_modify_inputs(tensor, new_dimensions, old_dimensions)


def _chain_modify_inputs(tensor, new_dimensions, old_dimensions):
    for input in tensor.op.inputs:
        if input.op.type == 'Placeholder':
            continue

        if len(input._shape) == len(new_dimensions):
            if modify_shape(input._shape, new_dimensions, old_dimensions):
                _chain_modify_inputs(input, new_dimensions, old_dimensions)
        elif len(input._shape) == len(new_dimensions) + 1:
            if modify_shape(input._shape, (input._shape[0]._value,) + new_dimensions, (input._shape[0]._value,) +
old_dimensions):
                _chain_modify_inputs(input, new_dimensions, old_dimensions)
        elif len(input._shape) == 0:
            pass
        elif len(input._shape) + 1 == len(new_dimensions) and new_dimensions[0] is None:
            if modify_shape(input._shape, new_dimensions[1:], old_dimensions[1:]):
                _chain_modify_inputs(input, new_dimensions, old_dimensions)
        else:
            raise Exception("could not deal with this input")


def modify_shape(shape, new_dimensions, old_dimensions=None):
    changed = False
    assert isinstance(shape, TensorShape)
    assert len(shape) == len(new_dimensions)

    for i in range(len(new_dimensions)):
        if shape._dims[i]._value != new_dimensions[i] and (old_dimensions is None or shape._dims[i]._value ==
old_dimensions[i]):
            changed = True
            shape._dims[i]._value = new_dimensions[i]

    return changed
```

```
def tf_resize_cascading(session, variable, new_values):
    # raise NotImplementedError()
    tf_resize(session, variable, tuple(new_values.shape), new_values)
    consumers = variable._as_graph_element().consumers()
    for consumer in consumers:
        for output in consumer.outputs:
            print output


def train_till_convergence(train_one_epoch_function, continue_epochs=3, max_epochs=10000,
                           log=False,
                           on_no_improvement_func=None):
    """Runs the train_one_epoch_function until we go continue_epochs without improvement in the best error

    Args:
        on_no_improvement_func (()->()): Called whenever we don't see an improvement in training, can be used
to change
            the learning rate
        train_one_epoch_function (()->number): Function that when called runs one epoch of training returning the
error
            from training.
        continue_epochs (int): The number of epochs without improvement before we terminate training, default 3
        max_epochs (int): The max number of epochs we can run for. default 10000
        log (bool): If true print result of each epoch

    Returns:
        int: The error we got for the final training epoch
    """
    best_error = train_one_epoch_function()
    error = best_error
    epochs_since_best_error = 0

    for epochs in xrange(1, max_epochs):
        error = train_one_epoch_function()
        if log:
            logger.info("epochs %s error %s", epochs, error)

        if error < best_error:
            best_error = error
            epochs_since_best_error = 0
        else:
            epochs_since_best_error += 1
            if epochs_since_best_error >= continue_epochs:
                if log:
                    logger.info("finished with best error %s", best_error)
                break

            if on_no_improvement_func:
                on_no_improvement_func()

    return error


def get_tf_optimizer_variables(optimizer):
    """Get all the tensorflow variables in an optimzier, for use in initialization

    Args:
        optimizer (tf.train.Optimizer): Some kind of tensorflow optimizer
```

```
    Returns:
        Iterable of tf.Variable
    """
    if isinstance(optimizer, tf.train.AdamOptimizer):
        for var in _get_optimzer_slot_variables(optimizer):
            yield var
        yield optimizer._beta1_power
        yield optimizer._beta2_power
    elif isinstance(optimizer, tf.train.RMSPropOptimizer):
        for var in _get_optimzer_slot_variables(optimizer):
            yield var
    elif isinstance(optimizer, tf.train.GradientDescentOptimizer):
        pass
    else:
        raise TypeError("Unsupported optimizer %s" % (type(optimizer),))


def _get_optimzer_slot_variables(optimizer):
    count = 0
    for slot_values in optimizer._slots.values():
        for value in slot_values.values():
            count += 1
            yield value

    if count == 0:
        raise Exception("Found no variables in optimizer, you may need to call minimize on this optimizer before
calling this method")


def _iterate_coords(tensor):
    if len(tensor.get_shape()) == 1:
        for i in range(tensor.get_shape()[0]):
            yield (i,), (1,)
    else:
        for i in range(tensor.get_shape()[0]):
            for j in range(tensor.get_shape()[1]):
                yield (i, j), (1, 1)


def _variable_size(variable):
    size = 1
    for dim in variable.get_shape():
        size *= int(dim)
    return size


def create_hessian_op(tensor_op, variables, session):
    mat = []
    for v1 in variables:
        for v2 in variables:
            temp = []
            # computing derivative twice, first w.r.t v2 and then w.r.t v1
            first_derivative = tf.gradients(tensor_op, v2)[0]
            for begin, size in _iterate_coords(v2):
                temp.append(tf.gradients(tf.slice(first_derivative, begin=begin, size=size), v1)[0])
            # tensorflow returns None when there is no gradient, so we replace None with, maybe we should just
fail...
            # temp = [0. if t is None else t for t in temp]

            derivatives = tf.concat(0, temp)
```

```
            mat.append(temp)

        raise NotImplementedError()

        return mat


def get_first_two_derivatives_op(loss_op, tensor):
    """Given a loss function get the 2nd derivatives of all variables with respect to the loss function

    Args:
        loss_op:
        tensor:

    Returns:

    """
    # computing derivative twice, first w.r.t v2 and then w.r.t v1
    first_derivative = tf.gradients(loss_op, tensor)[0]
    second_derivative = tf.gradients(first_derivative, tensor)[0]

    return first_derivative, second_derivative


def create_hessian_variable_op(loss_op, tensor):
    """Given a loss function get the 2nd derivatives of all variables with respect to the loss function

    Args:
        loss_op:
        tensor:

    Returns:

    """
    # computing derivative twice, first w.r.t v2 and then w.r.t v1
    first_derivative = tf.gradients(loss_op, tensor)[0]
    second_derivative = tf.gradients(first_derivative, tensor)[0]

    return second_derivative
```

tensordynamic/tensor_dynamic/**weight_functions.py**

```
import math

import numpy as np
import tensorflow as tf


def noise_weight_extender(array, new_dimensions, mean=0.0, var=None):
    """Extends a numpy array to have a new dimension, new values are filled in using random gaussian noise

    Args:
        array (np.array): The array we want to resize
        new_dimensions ([int]): The size to extend the array to, must be larger than the current array
        mean (float):
        var (float): How much random noise to add when changing layer size

    Returns:
        np.array : Array will be of size new_dims
    """
    assert len(array.shape) == len(new_dimensions)

    if any(x <= 0 for x in new_dimensions):
        raise ValueError("new_dimensions must all be greater than 0 was %s" % (new_dimensions,))
    new_values = array

    for index in range(len(new_dimensions)):
        if new_dimensions[index] > array.shape[index]:
            append_size = tuple(
                new_dimensions[index] - new_values.shape[index] if index == i else new_values.shape[i] for i in
                range(len(new_dimensions)))
            new_values = np.append(new_values,
                                   np.random.normal(scale=var or (1.0 / math.sqrt(float(new_dimensions[index]))),
                                                    loc=mean,
                                                    size=append_size)
                                   .astype(array.dtype),
                                   axis=index)
        elif new_dimensions[index] < new_values.shape[index]:
            trim_amount = new_dimensions[index] - new_values.shape[index]
            new_values = np.delete(new_values, np.s_[trim_amount:], index)

    assert new_values.shape == new_dimensions
    return new_values


def array_extend(array, vectors_to_extend, noise_std=None, halve_extended_vectors=False):
    """Extends the array arg by the column/row specified in vectors_to_extend duplicated

    Examples:
        a = np.array([[0, 1, 0],
                      [0, 1, 0]])
        array_split_extension(a, {1: [1]}) # {1: [1]} means duplicate column, with index 1
        # np.array([[0, 1, 0, 1], [0, 1, 0, 1]])

    Args:
        array (np.array): The array we want to split
```

```
        vectors_to_extend ({int:[int]): The keys are the axis we want to split, 0 = rows, 1 = keys,
            while the values are which rows/columns along that axis we want to duplicate
        noise_std (float): If set some random noise is applied to the extended column and subtracted from the
            duplicated column. The std of the noise is the value of this column
        halve_extended_vectors (bool): If True then extended vector and vector copied from both halved so as to
leave
            the network activation, relatively unchanged

    Returns:
        np.array : The array passed in as array arg but now extended
    """
    for axis, split_indexes in vectors_to_extend.iteritems():
        for x in split_indexes:
            split_args = [slice(None)] * array.ndim
            split_args[axis] = x
            add_weights = np.copy(array[split_args])
            reshape_args = list(array.shape)
            reshape_args[axis] = 1
            add_weights = add_weights.reshape(reshape_args)

            if halve_extended_vectors:
                add_weights *= .5
                array[split_args] *= .5

            if noise_std:
                random_noise = np.random.normal(scale=noise_std, size=add_weights.shape)
                add_weights += random_noise
                array[split_args] -= np.squeeze(random_noise, axis=[axis])

            array = np.r_[str(axis), array, add_weights]
    return array


def net_2_deeper_net(bias, noise_std=0.1):
    """
    This is a similar idea to net 2 deeper net from http://arxiv.org/pdf/1511.05641.pdf
    Assumes that this is a linear layer that is being extended and also adds some noise

    Args:
        bias (numpy.array): The bias for the layer we are adding after
        noise_std (Optional float): The amount of normal noise to add to the layer.
            If None then no noise is added
            Default is 0.1
    Returns:
        (numpy.matrix, numpy.array)
        The first item is the weights for the new layer
        Second item is the bias for the new layer
    """
    new_weights = np.matrix(np.eye(bias.shape[0]))
    new_bias = np.zeros(bias.shape)

    if noise_std:
        new_weights = new_weights + np.random.normal(scale=noise_std, size=new_weights.shape)
        new_bias = new_bias + np.random.normal(scale=noise_std, size=new_bias.shape)

    return new_weights.astype(bias.dtype), new_bias.astype(bias.dtype)
```

tensordynamic/tensor_dynamic/scripts/**basic_train_policy_resizing_mnist.py**

```
import os

import tensorflow as tf

import tensor_dynamic.data.mnist_data as mnist
from tensor_dynamic.layers.batch_norm_layer import BatchNormLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.train_policy import TrainPolicy
from tensor_dynamic.categorical_trainer import CategoricalTrainer

# load data
batch_size = 100
initail_learning_rate = 0.15
resize_learning_rate = 0.05
minimal_model_training_epochs = 50
learn_rate_decay = 0.96
hidden_layers = [200, 100, 50, 10]
checkpoint_path = 'resizeing_results'
SAVE = True

data = mnist.get_mnist_data_set_collection("../data/MNIST_data", one_hot=True, validation_size=5000)


def create_network(sess, hidden_layers):
```

```
    inputs = tf.placeholder(tf.float32, shape=(None, 784))
    bactivate = True
    noise_std = 0.3
    non_lin = tf.nn.relu
    input_layer = InputLayer(inputs)
    last = BatchNormLayer(input_layer, sess)
    for hidden_nodes in hidden_layers:
        last = HiddenLayer(last, hidden_nodes, sess, bactivate=bactivate, non_liniarity=non_lin,
unsupervised_cost=.1,
                           noise_std=noise_std)
        last = BatchNormLayer(last, sess)

    outputs = HiddenLayer(last, 10, sess, non_liniarity=tf.sigmoid, bactivate=False, supervised_cost=1.)

    trainer = CategoricalTrainer(outputs, initail_learning_rate)

    return outputs, trainer


with tf.Session() as sess:
    net, trainer = create_network(sess, hidden_layers)

    # train minimal model on mnist/load checkpoints
    if not os.path.exists(checkpoint_path):
        os.mkdir(checkpoint_path)
    saver = tf.train.Saver()
    checkpoints = tf.train.get_checkpoint_state(checkpoint_path)

    if checkpoints:
        saver.restore(sess, checkpoints.model_checkpoint_path)
        print("Loaded checkpoints %s" % checkpoints.model_checkpoint_path)
    else:
        print("retraining network")
        tp = TrainPolicy(trainer, data, batch_size, learn_rate_decay=learn_rate_decay)
        tp.train_till_convergence()

        if SAVE:
            saver.save(sess, checkpoint_path + "/network")

    # get train error
    print("train error ", trainer.accuracy(data.validation.features, data.validation.labels))

    # get reconstruction errors
    print trainer.back_losses_per_layer(data.train.features)

    # get error just on miss-classifications
    print trainer.back_losses_per_layer(data.train.features, misclassification_only=True, labels=data.train.labels)

    results = {}

    # try each different resize, see how it does
    for x in range(len(hidden_layers)):
        print("resizing layer ", x)
        cloned = net.clone()
        hidden_layers = [layer for layer in cloned.all_connected_layers if type(layer) == HiddenLayer]
        hidden_layers[x].resize()  # add 1 node
        new_trainer = CategoricalTrainer(net, resize_learning_rate)
        new_tp = TrainPolicy(new_trainer, data, batch_size, learn_rate_decay=learn_rate_decay)
        new_tp.train_till_convergence()
        acc, cost = trainer.accuracy(data.validation.features, data.validation.labels)
```

```
        print("train error ", acc, cost)
        results[x] = (acc, cost)

print results
```

tensordynamic/tensor_dynamic/scripts/**bayesian_neural_network.py**

```
import logging
import sys
from math import log, exp

import tensor_dynamic.data.mnist_data as mnist
import tensorflow as tf

from tensor_dynamic.utils import train_till_convergence

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

checkpoint_path = "bayesian_neural_network_hidden_40"

restore = False

data = mnist.get_mnist_data_set_collection("../data/MNIST_data", one_hot=True)

input_nodes = 784
output_nodes = 10
hidden_nodes = 40
num_weights = input_nodes * hidden_nodes + hidden_nodes + hidden_nodes * output_nodes + output_nodes
alpha = 300. / num_weights
beta = 1.

input_placeholder = tf.placeholder(tf.float32, shape=(None, input_nodes))
target_placeholder = tf.placeholder(tf.float32, shape=(None, output_nodes))

weights_hidden = tf.Variable(tf.random_normal(shape=(input_nodes, hidden_nodes), mean=0., stddev=1. /
input_nodes))
bias_hidden = tf.Variable(tf.zeros((hidden_nodes,)))

hidden_layer = tf.sigmoid(tf.matmul(input_placeholder, weights_hidden) + bias_hidden)

weights_output = tf.Variable(tf.random_normal(shape=(hidden_nodes, output_nodes), mean=0.,
                             stddev=1. / hidden_nodes))
bias_output = tf.Variable(tf.zeros((output_nodes,)))

output_layer = tf.nn.softmax(tf.matmul(hidden_layer, weights_output) + bias_output)

cross_entropy = -tf.reduce_sum(
  tf.reduce_mean(target_placeholder * tf.log(output_layer) + (1. - target_placeholder) * tf.log(1. - output_layer),
          1))
target_loss = beta * cross_entropy
weights_squared = tf.reduce_sum(tf.square(weights_hidden)) + tf.reduce_sum(tf.square(bias_hidden)) + \
          tf.reduce_sum(tf.square(weights_output)) + tf.reduce_sum(tf.square(bias_output))
```

```
regularization_loss = weights_squared * .5 * alpha

loss_op = target_loss + regularization_loss

correct_op = tf.nn.in_top_k(output_layer, tf.argmax(target_placeholder, 1), 1)


def log_probability_of_targets_given_weights(network_prediction, data, targets):
    predictions = network_prediction(data)

    result = 0.
    for i in range(len(predictions)):
        result += log(sum(predictions[i] * targets[i]))

    return result


def log_prior_probability_of_weights(weights_squared, alpha, min_weight_squared, max_weight_squared):
    numerator = exp(-alpha * .5 * weights_squared)

    integral = lambda x: -2 * exp(-.5 * alpha * x) / alpha
    return log(numerator / (integral(max_weight_squared) - integral(min_weight_squared)))


with tf.Session() as session:
    train_op = tf.train.AdamOptimizer().minimize(loss_op)

    session.run(tf.initialize_all_variables())


    def train():
        error = 0.
        current_epoch = data.train.epochs_completed
        while data.train.epochs_completed == current_epoch:
            images, labels = data.train.next_batch(100)
            _, batch_error = session.run([train_op, loss_op],
                                         feed_dict={input_placeholder: images, target_placeholder: labels})
            error += batch_error
        return error


    saver = tf.train.Saver()

    if restore:
        saver.restore(session, checkpoint_path)
    else:
        final_error = train_till_convergence(train, log=True)
        saver.save(session, checkpoint_path)

    accuracy = session.run(tf.reduce_mean(tf.cast(correct_op, tf.float32)),
                           feed_dict={input_placeholder: data.train.features, target_placeholder: data.train.labels})

    print("accuracy = %s " % accuracy)

    ln_prob_weights = log_prior_probability_of_weights(session.run(weights_squared), alpha, 0., 16. *
num_weights)

    ln_prob_labels_given_weights = log_probability_of_targets_given_weights(
        lambda x: session.run(output_layer, feed_dict={input_placeholder: x}), data.train.features, data.train.labels)
```

```
    # ln p(y) = sum(log(p(y_i))
    prob_of_labels = len(data.train.labels) * log(0.1)

    ln_prob_weights_given_labels_given_data = ln_prob_labels_given_weights + ln_prob_weights -
prob_of_labels

    print(ln_prob_weights_given_labels_given_data, ln_prob_labels_given_weights, ln_prob_weights,
prob_of_labels)
```

tensordynamic/tensor_dynamic/scripts/**ladder_network_mnist.py**

```
"""
Runs a Ladder network implemented via TensorDynamic on the mnist data set
"""

import tensorflow as tf

import tensor_dynamic.data.mnist_data as mnist
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.ladder_layer import LadderLayer, LadderGammaLayer
from tensor_dynamic.layers.ladder_output_layer import LadderOutputLayer

num_labeled = 100
data = mnist.get_mnist_data_set_collection("../data/MNIST_data", number_labeled_examples=num_labeled,
one_hot=True)

NOISE_STD = 0.3
batch_size = 100
num_epochs = 1
num_examples = 60000
```

```
num_iter = (num_examples/batch_size) * num_epochs
learning_rate = 0.1
inputs = tf.placeholder(tf.float32, shape=(None, 784))
targets = tf.placeholder(tf.float32)

with tf.Session() as s:
    s.as_default()
    i = InputLayer(inputs, layer_noise_std=NOISE_STD)
    l1 = LadderLayer(i, 500, 1000.0, s)
    l2 = LadderGammaLayer(l1, 10, 10.0, s)
    ladder = LadderOutputLayer(l2, 0.1, s)
    l3 = ladder

    assert int(i.z.get_shape()[-1]) == 784
    assert int(l1.z_corrupted.get_shape()[-1]) == 500
    assert int(l2.z_corrupted.get_shape()[-1]) == 10

    assert int(l3.z_est.get_shape()[-1]) == 10
    assert int(l2.z_est.get_shape()[-1]) == 500
    assert int(l1.z_est.get_shape()[-1]) == 784

    assert int(l1.mean_corrupted_unlabeled.get_shape()[0]) == 500
    assert int(l2.mean_corrupted_unlabeled.get_shape()[0]) == 10

    loss = ladder.cost_all_layers_train(targets)
    train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)
    pred_cost = -tf.reduce_mean(tf.reduce_sum(targets * tf.log(ladder.activation), 1))  # cost used for prediction

    correct_prediction = tf.equal(tf.argmax(ladder.activation, 1), tf.argmax(targets, 1))  # no of correct predictions
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float")) * tf.constant(100.0)

    s.run(tf.initialize_all_variables())

    ladder.set_all_deterministic(True)

    print "acc", s.run([accuracy], feed_dict={inputs: data.test.features, targets: data.test.labels})

    ladder.set_all_deterministic(False)

    for i in range(num_iter):
        images, labels = data.train.next_batch(batch_size)
        _, loss_val = s.run([train_step, loss], feed_dict={inputs: images, targets: labels})

        print(i, loss_val)

        # if i % 50 == 0:
        #     print "acc" + str(net.catagorical_accurasy(train_x, train_y))

        print "acc", s.run([accuracy], feed_dict={inputs: data.test.features, targets: data.test.labels})

    ladder.set_all_deterministic(True)

    print "acc", s.run([accuracy], feed_dict={inputs: data.test.features, targets: data.test.labels})
```

tensordynamic/tensor_dynamic/scripts/**servo.py**

```
import sys

import itertools

from tensor_dynamic.data.servo import get_data
from tensor_dynamic.data_functions import shuffle
from tensor_dynamic.net import Net
import tensorflow as tf
import numpy as np


train_x, train_y = get_data('../data/')
INPUT_DIM, OUTPUT_DIM = len(train_x[0]), len(train_y[0])
# train_x = tf.constant(train_x)
# train_y = tf.constant(train_y)
max_iterations = 5000
no_back_iterations = 0
batch_size = 5

def withGrowth():
    global train_x, train_y, max_iterations, batch_size
    with tf.Session() as session:
        net = Net(session, INPUT_DIM, OUTPUT_DIM)
        net = net.add_hidden_layer(session, 1, bactivate=True, non_liniarity=tf.nn.sigmoid)
        net = net.add_hidden_layer(session, 1, bactivate=True, non_liniarity=tf.nn.sigmoid)
        last_loss = 1000000000.0
        loss_counts = 0
        for i in range(max_iterations - no_back_iterations):
            train_x, train_y = shuffle(train_x, train_y)
            loss = net.train(train_x, train_y, batch_size=batch_size)
            print(i, loss)
            if loss > last_loss:
                if loss_counts > 9:
                    print "adding new nodes"
                    back_loss = net.get_reconstruction_error_per_hidden_layer(train_x, train_y)
                    print "Back loss %s" % (back_loss,)
                    layer_with_greatest_back_loss = back_loss.index(max(back_loss))
                    net = net.add_node_to_hidden_layer(session, layer_with_greatest_back_loss)
                    print net.hidden_nodes
                    last_loss = 10000000000.0
                    loss_counts = 0
```

```
                else:
                    last_loss = loss
                    loss_counts += 1
            else:
                last_loss = loss

        net.use_bactivate = False
        for j in range(no_back_iterations):
            i = j + max_iterations - no_back_iterations
            train_x, train_y = shuffle(train_x, train_y)
            loss = net.train(train_x, train_y, batch_size=batch_size)
            print(i, loss)


    print "final loss %s, %s" % (i, loss)
    print "nodes %s" % (net.hidden_nodes, )

def noGrowth(layers, bactivate=False):
    global train_x, train_y, max_iterations, batch_size
    with tf.Session() as session:
        net = Net(session, INPUT_DIM, OUTPUT_DIM)

        for l in layers:
            net = net.add_hidden_layer(session, l, bactivate=bactivate)

        for i in range(max_iterations):
            train_x, train_y = shuffle(train_x, train_y)
            loss = net.train(train_x, train_y, batch_size=batch_size)
            print(i, loss)

    print "final loss %s, %s" % (i, loss)
    return loss


MAX_LAYERS = 3
MAX_NODES_PER_LAYER = 14

noGrowth([10], bactivate=True)

# methods = [lambda x:noGrowth(x, bactivate=False), lambda x:noGrowth(x, bactivate=True)]
#
# with open('results.csv', 'a') as f:
#     for m in range(len(methods)):
#         for layers in range(1, MAX_LAYERS+1):
#             x = []
#             for arrangement in itertools.product(*[range(1, MAX_NODES_PER_LAYER+1)]*layers):
#                 res1 = methods[m](arrangement)
#
#                 f.write("%s, %s, %s\n" % (m, arrangement, res1))


# 10x10 no backivate = 0.18
# 10x10 backivate = 0.11
# growth 1x1 backivate = 0.11 fin 7 x 7, 0.12 fin 7 x 5, 0.14 fin 7 x 6
```

tensordynamic/tensor_dynamic/scripts/**standard_2_layer_mnist_example.py**

```
# Import MINST data
import tensor_dynamic.data.mnist_data as input_data
mnist = input_data.get_mnist_data_set_collection("../data/MNIST_data", one_hot=True)

import tensorflow as tf

# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 256 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Construct model
layer_1 = tf.nn.relu(tf.matmul(x, weights['h1']) + biases['b1']) #Hidden layer with RELU activation
layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])) #Hidden layer with RELU activation
pred = tf.matmul(layer_2, weights['out']) + biases['out']

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y)) # Softmax loss
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost) # Adam Optimizer

# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
```

```
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batch_ys})/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

    print "Optimization Finished!"

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print "Accuracy:", accuracy.eval({x: mnist.test.features, y: mnist.test.labels})
```

tensordynamic/tensor_dynamic/scripts/**train_policy.py**

```
import tensorflow as tf

import tensor_dynamic.data.mnist_data as mnist
from tensor_dynamic.layers.back_weight_layer import BackWeightLayer
from tensor_dynamic.layers.batch_norm_layer import BatchNormLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.train_policy import TrainPolicy
from tensor_dynamic.categorical_trainer import CategoricalTrainer

batch_size = 100

data = mnist.get_mnist_data_set_collection("../data/MNIST_data", one_hot=True, validation_size=5000)

with tf.Session() as sess:
    inputs = tf.placeholder(tf.float32, shape=(None, 784))

    bactivate = True
    noise_std = 0.3
    beta = 0.5
    gamma = 0.5
```

```
    non_lin = tf.nn.sigmoid
    input_layer = InputLayer(inputs)
    bn1 = BatchNormLayer(input_layer, sess, beta=beta, gamma=gamma)
    net1 = HiddenLayer(bn1, 1, sess, non_liniarity=non_lin, bactivate=bactivate, unsupervised_cost=.001,
noise_std=noise_std)
    bn2 = BatchNormLayer(net1, sess, beta=beta, gamma=gamma)
    net2 = HiddenLayer(bn2, 1, sess, non_liniarity=non_lin, bactivate=bactivate, unsupervised_cost=.001,
noise_std=noise_std)
    bn3 = BatchNormLayer(net2, sess, beta=beta, gamma=gamma)
    net3 = HiddenLayer(bn3, 1, sess, non_liniarity=non_lin, bactivate=bactivate, unsupervised_cost=.001,
noise_std=noise_std)
    bn4 = BatchNormLayer(net3, sess, beta=beta, gamma=gamma)
    net4 = HiddenLayer(bn4, 1, sess, non_liniarity=non_lin, bactivate=bactivate, unsupervised_cost=.001,
noise_std=noise_std)
    bn5 = BatchNormLayer(net4, sess, beta=beta, gamma=gamma)
    outputNet = HiddenLayer(bn5, 10, sess, non_liniarity=tf.sigmoid, bactivate=False, supervised_cost=1.)

    trainer = CategoricalTrainer(outputNet, 0.15)
    trainPolicy = TrainPolicy(trainer, data, batch_size, max_iterations=3000,
                              grow_after_turns_without_improvement=2,
                              start_grow_epoch=1,
                              learn_rate_decay=0.99,
                              learn_rate_boost=0.01,
                              back_loss_on_misclassified_only=True)

    trainPolicy.run_full()

    print trainer.accuracy(data.test.features, data.test.labels)
```

tensordynamic/tensor_dynamic/scripts/**xor.py**

```
import tensorflow as tf

from tensor_dynamic.data_functions import XOR_INPUTS, XOR_TARGETS
from tensor_dynamic.net import Net

train_x = XOR_INPUTS
train_y = XOR_TARGETS
max_iterations = 2000
batch_size = 1

with tf.Session() as session:
    net = Net(session, 2, 1)
    net = net.add_hidden_layer(session, 40)
    net = net.add_hidden_layer(session, 10)
    net = net.add_hidden_layer(session, 4)

    for i in range(max_iterations):
        #train_x, train_y = shuffle(train_x, train_y)
        loss = net.train(train_x, train_y, batch_size=batch_size)
        print(loss)
```

tensordynamic/tensor_dynamic/layers/**back_weight_candidate_layer.py**

```
import tensorflow as tf
import numpy as np

from tensor_dynamic.layers.back_weight_layer import BackWeightLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.tf_loss_functions import squared_loss
from tensor_dynamic.utils import xavier_init, tf_resize
from tensor_dynamic.weight_functions import noise_weight_extender


class BackWeightCandidateLayer(BackWeightLayer):
    CANDIDATES = 1
    CANDIDATE_TRAIN_DISCOUNT = 0.1

    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_weights=None,
                 back_bias=None,
                 freeze=False,
                 non_liniarity=tf.nn.relu,
                 weight_extender_func=noise_weight_extender,
                 bactivation_loss_func=squared_loss,
                 unsupervised_cost=1.,
                 supervised_cost=1.,
                 noise_std=None,
                 name='BackWeightLayer'):
        super(BackWeightCandidateLayer, self).__init__(input_layer, output_nodes,
                                                       session=session,
                                                       bias=bias,
                                                       weights=weights,
                                                       back_weights=back_weights,
                                                       back_bias=back_bias,
                                                       freeze=freeze,
                                                       bactivation_loss_func=bactivation_loss_func,
                                                       non_liniarity=non_liniarity,
                                                       weight_extender_func=weight_extender_func,
                                                       unsupervised_cost=unsupervised_cost,
                                                       supervised_cost=supervised_cost,
                                                       noise_std=noise_std,
                                                       name=name)

        self._candidate_bias = self._create_variable("candidate_bias",
                                                     (self.CANDIDATES,),
                                                     np.zeros(self.CANDIDATES, dtype=np.float32),
                                                     is_kwarg=False)
        self._candidate_weights = self._create_variable("candidate_weights",
                                                        (self.INPUT_BOUND_VALUE, self.CANDIDATES),
                                                        xavier_init(
                                                            self.input_nodes,
```

```
                                                   1),
                                                 is_kwarg=False)
        self._candidate_back_bias = self._create_variable("candidate_back_bias",
                                                  (self.CANDIDATES,),
                                                  np.zeros(self.CANDIDATES, dtype=np.float32),
                                                  is_kwarg=False)
        self._candidate_back_weights = self._create_variable("candidate_back_weights",
                                                     (self.CANDIDATES, self.INPUT_BOUND_VALUE),
                                                     xavier_init(
                                                         1,
                                                         self.input_nodes),
                                                     is_kwarg=False)

        self._candidate_bactivation_predict = self._non_liniarity(
            tf.matmul(
                self._non_liniarity(
                    tf.matmul(self.input_layer.activation_predict, self._candidate_weights) + self._candidate_bias),
                self._candidate_back_weights) + self._candidate_back_bias)

        self._candidate_bactivation_train = self._non_liniarity(
            tf.matmul(
                self._non_liniarity(
                    tf.matmul(self.input_layer.activation_train, self._candidate_weights) + self._candidate_bias),
                self._candidate_back_weights) + self._candidate_back_bias)

        self.session.run(tf.initialize_variables([self._candidate_weights, self._candidate_bias,
                                              self._candidate_back_bias, self._candidate_back_weights]))

    @lazyprop
    def bactivation_loss_train(self):
        return tf.reduce_mean(tf.square(
            (self.bactivation_train + self._candidate_bactivation_train * self.CANDIDATE_TRAIN_DISCOUNT)
            - self.input_layer.activation_train))

    @lazyprop
    def bactivation_loss_predict(self):
        return tf.reduce_mean(tf.square(
            (self.bactivation_predict + self._candidate_bactivation_predict * self.CANDIDATE_TRAIN_DISCOUNT)
            - self.input_layer.activation_predict))

    def resize(self, new_output_nodes=None):
        if new_output_nodes is None or new_output_nodes > self.output_nodes:
            # promote the candidate
            tf_resize(self._session, self._weights,
                      new_values=np.append(self.session.run(self._weights),
self.session.run(self._candidate_weights),
                                           axis=1).astype(np.float32))

            tf_resize(self._session, self._back_weights,
                      new_values=np.append(self.session.run(self._back_weights),
                                           self.session.run(self._candidate_back_weights), axis=0).astype(np.float32), )

            tf_resize(self._session, self._bias,
                      new_values=np.append(self.session.run(self._bias),
self.session.run(self._candidate_bias)).astype(
                          np.float32))

            tf_resize(self._session, self._back_bias,
                      new_values=np.append(self.session.run(self._back_bias),
                                           self.session.run(self._candidate_back_bias)).astype(np.float32))
```

```
        super(BackWeightCandidateLayer, self).resize(new_output_nodes=new_output_nodes)
```

tensordynamic/tensor_dynamic/layers/**back_weight_layer.py**

```
import tensorflow as tf
from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.tf_loss_functions import squared_loss
from tensor_dynamic.utils import xavier_init
from tensor_dynamic.weight_functions import noise_weight_extender


class BackWeightLayer(HiddenLayer):
    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_weights=None,
                 back_bias=None,
                 freeze=False,
                 non_liniarity=tf.nn.relu,
                 bactivation_loss_func=squared_loss,
                 weight_extender_func=noise_weight_extender,
                 unsupervised_cost=1.,
                 supervised_cost=1.,
```

```
              noise_std=None,
              name='BackWeightLayer'):
    super(BackWeightLayer, self).__init__(input_layer, output_nodes,
                                          session=session,
                                          bias=bias,
                                          weights=weights,
                                          back_bias=back_bias,
                                          bactivate=True,
                                          freeze=freeze,
                                          non_liniarity=non_liniarity,
                                          weight_extender_func=weight_extender_func,
                                          bactivation_loss_func=bactivation_loss_func,
                                          unsupervised_cost=unsupervised_cost,
                                          supervised_cost=supervised_cost,
                                          noise_std=noise_std,
                                          name=name)
    self._back_weights = self._create_variable("back_weights",
                                               (BaseLayer.OUTPUT_BOUND_VALUE, BaseLayer.INPUT_BOUND_VALUE),
                                               back_weights if back_weights is not None else xavier_init(
                                                   self._output_nodes,
                                                   self._input_nodes))

def _layer_bactivation(self, activation):
    return self._non_liniarity(
        tf.matmul(activation, self._back_weights) + self._back_bias)

@property
def kwargs(self):
    kwargs = super(BackWeightLayer, self).kwargs

    # bactivate is not optional for these layers
    del kwargs['bactivate']

    return kwargs
```

tensordynamic/tensor_dynamic/layers/**base_layer.py**

```
import functools
import logging
import pickle
from abc import ABCMeta, abstractmethod
from collections import namedtuple

import numpy as np
import operator

import sys
import tensorflow as tf

from tensor_dynamic.lazyprop import clear_all_lazyprops, lazyprop, clear_lazyprop_on_lazyprop_cleared,
has_lazyprop
from tensor_dynamic.utils import tf_resize, bias_init, weight_init
from tensor_dynamic.weight_functions import noise_weight_extender, array_extend

# fix for older version of tensorflow
if not hasattr(tf, 'variables_initializer'):
    setattr(tf, 'variables_initializer', tf.initialize_variables)


logger = logging.getLogger(__name__)


class BaseLayer(object):
    __metaclass__ = ABCMeta

    GROWTH_MULTIPLYER = 1.1
    SHRINK_MULTIPLYER = 1. / GROWTH_MULTIPLYER
    MINIMUM_GROW_AMOUNT = 3

    OUTPUT_BOUND_VALUE = 'output'
    INPUT_BOUND_VALUE = 'input'
    INPUT_DIM_3_BOUND_VALUE = 'input_3'
    OUTPUT_DIM_3_BOUND_VALUE = 'output_3'

    _BoundVariable = namedtuple('_BoundVariable', ['name', 'dimensions', 'variable', 'is_kwarg'])

    def __init__(self,
                 input_layer,
                 output_nodes,
```

```
             session=None,
             weight_extender_func=None,
             weight_initializer_func=None,
             bias_initializer_func=None,
             layer_noise_std=None,
             drop_out_prob=None,
             batch_normalize_input=False,
             batch_norm_transform=None,
             batch_norm_scale=None,
             name=None,
             freeze=False):
    """Base class from which all layers will inherit. This is an abstract class

    Args:
        weight_initializer_func ((int)->weights): function that creates initial values for weights for this layer
        bias_initializer_func (int->weights): function that creates initial values for weights for this layer
        layer_noise_std (float): If not None gaussian noise with mean 0 and this std is applied to the input of this
            layer
        input_layer (tensor_dynamic.base_layer.BaseLayer): This layer will work on the activation of the
input_layer
        output_nodes (int | tuple of ints): Number of output nodes for this layer, can be a tuple of multi
dimensional output, e.g. convolutional network
        session (tensorflow.Session): The session within which all these variables should be created
        weight_extender_func (func): Method that extends the size of matrix or vectors
        name (str): Used for identifying the layer and when initializing tensorflow variables
        freeze (bool):If True then weights in this layer are not trainable
    """
    if not isinstance(input_layer, BaseLayer):
        raise TypeError("input_layer must be of type %s" % BaseLayer)

    assert isinstance(output_nodes, (int, tuple))
    assert isinstance(input_layer, BaseLayer)
    self._bound_variable_assign_data = {}
    self._input_layer = input_layer
    self._layer_noise_std = layer_noise_std
    self._drop_out_prob = drop_out_prob
    self._batch_normalize_input = batch_normalize_input
    self._name = name
    self._output_nodes = (output_nodes,) if type(output_nodes) == int else output_nodes
    self._input_nodes = self._input_layer._output_nodes
    self._next_layer = None

    self._session = self._get_property_or_default(session, '_session',
                                                  None)
    self._weight_extender_func = self._get_property_or_default(weight_extender_func,
'_weight_extender_func',
                                                               noise_weight_extender)

    self._weight_initializer_func = self._get_property_or_default(weight_initializer_func,
                                                                  '_weight_initializer_func',
                                                                  weight_init)
    self._bias_initializer_func = self._get_property_or_default(bias_initializer_func,
                                                                '_bias_initializer_func',
                                                                bias_init)
    self._freeze = freeze
    self._bound_variables = {}
    input_layer._attach_next_layer(self)

    if self._batch_normalize_input:
        self._batch_norm_mean_train, self._batch_norm_var_train = (None, None)
```

```
        self._batch_norm_mean_predict, self._batch_norm_var_predict = (None, None)

        with self.name_scope():
            self._batch_norm_scale = self._create_variable("batch_norm_scale", (self.INPUT_BOUND_VALUE,),
                                                           batch_norm_scale if batch_norm_scale is not None else tf.ones(
                                                               self.input_nodes), is_kwarg=True)
            self._batch_norm_transform = self._create_variable("batch_norm_transform",
(self.INPUT_BOUND_VALUE,),
                                                               batch_norm_transform if batch_norm_transform is not None else
tf.zeros(
                                                                   self.input_nodes), is_kwarg=True)
            self._normalized_train = None
            self._normalized_predict = None

    def _get_property_or_default(self, init_value, property_name, default_value):
        if init_value is not None:
            return init_value
        if self.input_layer is not None:
            if hasattr(self.input_layer, property_name) and getattr(self.input_layer, property_name) is not None:
                return getattr(self.input_layer, property_name)
            else:
                earlier_in_stream_result = self.input_layer._get_property_or_default(init_value, property_name,
                                                                                     default_value)
                if earlier_in_stream_result is not None:
                    return earlier_in_stream_result

        return default_value

    def name_scope(self, is_train=False, is_predict=False):
        """Used for naming variables associated with this layer in TensorFlow in a consistent way

        Format = "{layer_number}_{layer_name}"

        Examples:
            with self.name_scope():
                my_new_variable = tf.Variable(default_val, name="name")

        Args:
            is_train (bool): Set for parts of tensorflow graph just for training
            is_predict (bool): Set for parts of tensorflow graph just for predicting

        Returns:
            A context manager that installs `name` as a new name scope in the
            default graph.
        """
        name = str(self.layer_number) + "_" + self._name

        if is_train and not is_predict:
            name += "_train"
        elif is_predict:
            name += "_predict"

        return tf.name_scope(name)

    @lazyprop
    def activation_train(self):
        """The activation used for training this layer, this will often be the same as prediction except with dropout or
        random noise applied.

        Returns:
```

```
            tensorflow.Tensor
        """
        clear_lazyprop_on_lazyprop_cleared(self, 'activation_train', self.input_layer)
        input_tensor = self.input_layer.activation_train

        with self.name_scope(is_train=True):
            input_tensor = self._process_input_activation_train(input_tensor)

            return self._layer_activation(input_tensor, True)

    def _process_input_activation_train(self, input_tensor):
        if self._batch_normalize_input:
            self._batch_norm_mean_train, self._batch_norm_var_train =
tf.nn.moments(self._input_layer.activation_train,
                                                                axes=range(len(self.input_nodes)))
            self._normalized_train = (
                (input_tensor - self._batch_norm_mean_train) / tf.sqrt(self._batch_norm_var_train + tf.constant(1e-
10)))
            input_tensor = (self._normalized_train + self._batch_norm_transform) * self._batch_norm_scale

        if self._drop_out_prob:
            input_tensor = tf.nn.dropout(input_tensor, self._drop_out_prob)

        if self._layer_noise_std is not None:
            input_tensor = input_tensor + tf.random_normal(tf.shape(self.input_layer.activation_train),
                                                           stddev=self._layer_noise_std)

        return input_tensor

    @lazyprop
    def activation_predict(self):
        """The activation used for predictions from this layer, this will often be the same as training except without
        dropout or random noise applied.

        Returns:
            tensorflow.Tensor
        """
        clear_lazyprop_on_lazyprop_cleared(self, 'activation_predict', self.input_layer)
        input_tensor = self.input_layer.activation_predict

        with self.name_scope(is_predict=True):
            input_tensor = self._process_input_activation_predict(input_tensor)
            return self._layer_activation(input_tensor, False)

    def _process_input_activation_predict(self, input_tensor):
        if self._batch_normalize_input:
            self._batch_norm_mean_predict, self._batch_norm_var_predict = tf.nn.moments(
                self._input_layer.activation_predict,
                axes=range(len(self.input_nodes)))

            # TODO: Note this is the WRONG way to apply this, will result in bad results for prediction sizes
            # that do not equal the batch_size...
            self._normalized_predict = (
                (input_tensor - self._batch_norm_mean_predict) / tf.sqrt(
                    self._batch_norm_var_predict + tf.constant(1e-10)))
            input_tensor = (self._normalized_predict + self._batch_norm_transform) * self._batch_norm_scale
        return input_tensor

    @abstractmethod
    def _layer_activation(self, input_tensor, is_train):
```

```
        """The activation for this layer

        Args:
            input_tensor (tensorflow.Tensor):
            is_train (bool): If true this is activation for training, if false for prediction

        Returns:
            tensorflow.Tensor
        """
        raise NotImplementedError()

    @property
    def bactivate(self):
        """All layers have output activation, some unsupervised layer activate backwards as well.

        e.g. Layers in ladder networks, Denoising Autoencoders

        Returns:
            bool
        """
        return False

    @property
    def is_input_layer(self):
        """Are we the input layer

        Returns:
            bool
        """
        return False

    @property
    def is_output_layer(self):
        """Is this the final layer of the network

        Returns:
            bool
        """
        return self._next_layer is None

    @property
    def output_nodes(self):
        """The number of output nodes

        Returns:
            tuple of ints
        """
        return self._output_nodes

    @property
    def input_nodes(self):
        """The number of input nodes to this layer

        Returns:
            tuple of ints
        """
        return self._input_nodes

    @property
    def session(self):
```

```
        """Session used to create the variables in this layer

        Returns:
            tensorflow.Session
        """
        return self._session

    @lazyprop
    def bactivation_train(self):
        """The activation used for training this layer, this will often be the same as prediction except with dropout or
        random noise applied.

        Returns:
            tensorflow.Tensor
        """
        clear_lazyprop_on_lazyprop_cleared(self, 'bactivation_train', self, 'activation_train')
        return self._layer_bactivation(self.activation_train, True)

    @lazyprop
    def bactivation_predict(self):
        """The activation used for predictions from this layer, this will often be the same as training except without
        dropout or random noise applied.

        Returns:
            tensorflow.Tensor
        """
        clear_lazyprop_on_lazyprop_cleared(self, 'bactivation_predict', self, 'activation_predict')
        return self._layer_bactivation(self.activation_predict, False)

    def _layer_bactivation(self, input_tensor, is_train):
        """The bactivation for this layer

        Args:
            input_tensor (tensorflow.Tensor):
            is_train (bool): If true this is activation for training, if false for prediction

        Returns:
            tensorflow.Tensor
        """
        raise NotImplementedError()

    @property
    def output_shape(self):
        return self.activation_predict.get_shape().as_list()

    @property
    def input_shape(self):
        return self._input_layer.output_shape

    @property
    def next_layer(self):
        return self._next_layer

    @property
    def input_layer(self):
        return self._input_layer

    @property
    def has_next_layer(self):
        return self.next_layer
```

```
def _attach_next_layer(self, layer):
    if self.has_next_layer:
        raise Exception("Can not attach_next_layer to Layer: %s which already has a next layer" % self._name)
    if not isinstance(layer, BaseLayer):
        raise TypeError("Attached layer must be of type %s" % BaseLayer)

    self._next_layer = layer

@property
def last_layer(self):
    if self._next_layer is not None:
        return self._next_layer.last_layer
    return self

@property
def first_layer(self):
    return self.input_layer.first_layer

@property
def input_placeholder(self):
    return self.first_layer.input_placeholder

@property
def target_placeholder(self):
    return self.last_layer.target_placeholder

@property
def downstream_layers(self):
    if self._next_layer:
        yield self._next_layer
        for d in self._next_layer.downstream_layers:
            yield d

@property
def upstream_layers(self):
    if self._input_layer:
        yield self._input_layer
        for d in self._input_layer.upstream_layers:
            yield d

@property
def all_layers(self):
    return self.all_connected_layers

@property
def all_connected_layers(self):
    for u in list(reversed(list(self.upstream_layers))):
        yield u
    yield self
    for d in self.downstream_layers:
        yield d

def activate_predict(self, data_set):
    """Get the prediction activation of this network given the data_set as input

    Args:
        data_set (np.array): np.array or Array matching the dimensions of the input placeholder

    Returns:
```

```
            np.array: prediction activation of the network
        """
        return self.session.run(self.activation_predict, feed_dict={self.input_placeholder: data_set})

    def supervised_cost_train(self, targets):
        return None

    def unsupervised_cost_train(self):
        return None

    def cost_train(self, targets):
        supervised_cost = self.supervised_cost_train(targets)
        unsupervised_cost = self.unsupervised_cost_train()

        if supervised_cost is not None:
            if unsupervised_cost is not None:
                return supervised_cost + unsupervised_cost
            return supervised_cost
        if unsupervised_cost is not None:
            return unsupervised_cost

        return None

    def cost_all_layers_train(self, targets):
        all_costs = [x.cost_train(targets) for x in self.all_connected_layers]
        all_costs = filter(lambda v: v is not None, all_costs)
        return tf.add_n(all_costs, name="cost_all_layers")

    @property
    def layer_number(self):
        return len(list(self.upstream_layers))

    @property
    def kwargs(self):
        kwargs = {
            'output_nodes': self._output_nodes,
            'weight_extender_func': self._weight_extender_func,
            'layer_noise_std': self._layer_noise_std,
            'drop_out_prob': self._drop_out_prob,
            'batch_normalize_input': self._batch_normalize_input,
            'freeze': self._freeze,
            'name': self._name}
        kwargs.update(self._bound_variables_as_kwargs())
        return kwargs

    def _bound_variables_as_kwargs(self):
        kwarg_dict = {}
        for name, bound_variable in self._bound_variables.iteritems():
            if bound_variable.is_kwarg:
                kwarg_dict[name] = self.session.run(bound_variable.variable)

        return kwarg_dict

    def clone(self, session=None):
        """Produce a clone of this layer AND all connected upstream layers

        Args:
            session (tensorflow.Session): If passed in the clone will be created with all variables initialised in this
session
                                If None then the current session of this layer is used
```

```
        Returns:
            tensorflow_dynamic.BaseLayer: A copy of this layer and all upstream layers
        """
        new_self = self.__class__(self.input_layer.clone(session or self.session),
                                  # self.output_nodes,
                                  session=session or self._session,
                                  **self.kwargs)

        return new_self

    def _resize_needed(self):
        """ If there is a mismatch between the input size of this layer and the output size of it's previous layer will
        return True

        Returns:
            bool
        """
        if self._input_layer.output_nodes != self.input_nodes:
            return True
        return False

    def resize(self, new_output_nodes=None,
               output_nodes_to_prune=None,
               input_nodes_to_prune=None,
               split_output_nodes=None,
               split_input_nodes=None,
               data_set_train=None,
               data_set_validation=None,
               no_splitting_or_pruning=False,
               split_nodes_noise_std=.1):
        """Resize this layer by changing the number of output nodes. Will also resize any downstream layers

        Args:
            data_set_validation (DataSet):Data set used for validating this network
            data_set_train (DataSet): Data set used for training this network
            no_splitting_or_pruning (bool): If set to true then noise is just added randomly rather than splitting nodes
            new_output_nodes (int | tuple of ints): If passed we change the number of output nodes of this layer to
be new_output_nodes
            output_nodes_to_prune ([int]): list of indexes of the output nodes we want pruned e.g. [1, 3] would
remove
                the 1st and 3rd output node from this layer
            input_nodes_to_prune ([int]): list of indexes of the input nodes we want pruned e.g. [1, 3] would remove
the
                1st and 3rd input node from this layer
            split_output_nodes ([int]): list of indexes of nodes to split. This is for growing the layer
            split_input_nodes: ([int]): list of indexes of nodes that where split in the previous layer.
            split_nodes_noise_std (float): standard deviation of noise to add when splitting a node
        """
        if isinstance(new_output_nodes, tuple):
            new_output_nodes = new_output_nodes[self.get_resizable_dimension()]
        elif new_output_nodes is not None and not isinstance(new_output_nodes, int):
            raise ValueError("new_output_nodes must be tuple of int %s" % (new_output_nodes,))

        if not no_splitting_or_pruning:
            # choose nodes to split or prune
            if new_output_nodes is not None:
                if output_nodes_to_prune is None and split_output_nodes is None:
                    if new_output_nodes < self.get_resizable_dimension_size():
                        output_nodes_to_prune = self._choose_nodes_to_prune(new_output_nodes, data_set_train,
```

```
                                                    data_set_validation)
            elif new_output_nodes > self.get_resizable_dimension_size():
                split_output_nodes = self._choose_nodes_to_split(new_output_nodes, data_set_train,
                                                    data_set_validation)
        elif self.has_resizable_dimension():
            new_output_nodes = self.get_resizable_dimension_size()
            if output_nodes_to_prune:
                new_output_nodes -= len(output_nodes_to_prune)
            if split_output_nodes:
                new_output_nodes += len(split_output_nodes)

    new_input_nodes = self.input_layer.output_nodes
    input_nodes_changed = new_input_nodes != self._input_nodes

    if self.has_resizable_dimension() and new_output_nodes is not None:
        output_nodes_changed = new_output_nodes != self.get_resizable_dimension_size()
        temp_output_nodes = list(self._output_nodes)
        temp_output_nodes[self.get_resizable_dimension()] = new_output_nodes

        self._output_nodes = tuple(temp_output_nodes)
    else:
        output_nodes_changed = False

    self._input_nodes = new_input_nodes

    for name, bound_variable in self._bound_variables.iteritems():
        if input_nodes_changed and self._bound_dimensions_contains_input(bound_variable.dimensions) or \
                output_nodes_changed and
self._bound_dimensions_contains_output(bound_variable.dimensions):

            self._forget_assign_op(name)

            int_dims = self._bound_dimensions_to_ints(bound_variable.dimensions)

            if isinstance(bound_variable.variable, tf.Variable):
                old_values = self._session.run(bound_variable.variable)
                if output_nodes_to_prune or split_output_nodes:
                    output_bound_axis = bound_variable.dimensions.index(self.OUTPUT_BOUND_VALUE)
                    if output_nodes_to_prune:
                        old_values = np.delete(old_values, output_nodes_to_prune, output_bound_axis)
                    else:  # split
                        old_values = array_extend(old_values, {output_bound_axis: split_output_nodes},
                                                  noise_std=split_nodes_noise_std)
                if input_nodes_to_prune or split_input_nodes:
                    input_bound_axis = bound_variable.dimensions.index(self.INPUT_BOUND_VALUE)
                    if input_nodes_to_prune:
                        old_values = np.delete(old_values, input_nodes_to_prune, input_bound_axis)
                    else:  # split
                        old_values = array_extend(old_values, {input_bound_axis: split_input_nodes},
                                                  halve_extended_vectors=True)
                if no_splitting_or_pruning:
                    new_values = self._weight_extender_func(old_values, int_dims)
                else:
                    new_values = old_values

                tf_resize(self._session, bound_variable.variable, int_dims,
                          new_values, self._get_assign_function(name))
            else:
                # this is a tensor, not a variable so has no weights
                tf_resize(self._session, bound_variable.variable, int_dims)
```

```
        if input_nodes_changed and self._batch_normalize_input:
            if self._batch_norm_mean_train is not None:
                tf_resize(self._session, self._batch_norm_mean_train, self._input_nodes)
                tf_resize(self._session, self._batch_norm_var_train, self._input_nodes)
            if self._batch_norm_mean_predict is not None:
                tf_resize(self._session, self._batch_norm_mean_predict, self._input_nodes)
                tf_resize(self._session, self._batch_norm_var_predict, self._input_nodes)
            if self._normalized_train is not None:
                tf_resize(self._session, self._normalized_train, (None,) + self._input_nodes)
            if self._normalized_predict is not None:
                tf_resize(self._session, self._normalized_predict, (None,) + self._input_nodes)

            # This line fixed the issue, this is all very hacky...
            # self._mat_mul.op.inputs[0]._shape = TensorShape((None,) + self._input_nodes)
            from tensorflow.python.framework.tensor_shape import TensorShape

            if '_mat_mul_is_train_equal_' + str(True) in self.__dict__:
                self.__dict__['_mat_mul_is_train_equal_' + str(True)].op.inputs[0]._shape = TensorShape(
                    (None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(True)].op.inputs[0].op.inputs[0]._shape = TensorShape(
                    (None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(True)].op.inputs[0].op.inputs[0].op.inputs[
                    0]._shape = TensorShape((None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(True)].op.inputs[0].op.inputs[0].op.inputs[0].op.inputs[
                    0]._shape = TensorShape((None,) + self._input_nodes)
                # tf_resize(self._session, self.__dict__['_mat_mul_is_train_equal_' + str(True)], (None,) +
self._input_nodes)
            if '_mat_mul_is_train_equal_' + str(False) in self.__dict__:
                self.__dict__['_mat_mul_is_train_equal_' + str(False)].op.inputs[0]._shape = TensorShape(
                    (None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(False)].op.inputs[0].op.inputs[0]._shape =
TensorShape(
                    (None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(False)].op.inputs[0].op.inputs[0].op.inputs[
                    0]._shape = TensorShape((None,) + self._input_nodes)
                self.__dict__['_mat_mul_is_train_equal_' + str(False)].op.inputs[0].op.inputs[0].op.inputs[0].op.inputs[
                    0]._shape = TensorShape((None,) + self._input_nodes)
                # tf_resize(self._session, self.__dict__['_mat_mul_is_train_equal_' + str(False)], (None,) +
self._input_nodes)

        if output_nodes_changed:
            if has_lazyprop(self, 'activation_predict'):
                tf_resize(self._session, self.activation_predict, (None,) + self._output_nodes)
            if has_lazyprop(self, 'activation_train'):
                tf_resize(self._session, self.activation_train, (None,) + self._output_nodes)

        if input_nodes_changed and self.bactivate:
            if has_lazyprop(self, 'bactivation_train'):
                tf_resize(self._session, self.bactivation_train, (None,) + self._input_nodes)
            if has_lazyprop(self, 'bactivation_predict'):
                tf_resize(self._session, self.bactivation_predict, (None,) + self._input_nodes)

        if self._next_layer and self._next_layer._resize_needed():
            self._next_layer.resize(input_nodes_to_prune=output_nodes_to_prune,
split_input_nodes=split_output_nodes,
                                    no_splitting_or_pruning=no_splitting_or_pruning)

    def _forget_assign_op(self, name):
        if name in self._bound_variable_assign_data:
```

```
            del self._bound_variable_assign_data[name]

    def _bound_dimensions_to_ints(self, bound_dims):
        int_dims = ()
        for x in bound_dims:
            if isinstance(x, int):
                if x == -1:
                    int_dims += (None,)
                else:
                    int_dims += (x,)
            elif x == self.OUTPUT_BOUND_VALUE:
                int_dims += (self._output_nodes[0],)
            elif x == self.INPUT_BOUND_VALUE:
                int_dims += (self._input_nodes[0],)
            elif x == self.INPUT_DIM_3_BOUND_VALUE:
                assert len(self._input_nodes) == 3, "must have 3 input dimensions"
                int_dims += (self._input_nodes[2],)
            elif x == self.OUTPUT_DIM_3_BOUND_VALUE:
                assert len(self._input_nodes) == 3, "must have 3 output dimensions"
                int_dims += (self._output_nodes[2],)
            elif x is None:
                int_dims += (None,)
            else:
                raise Exception("bound dimension must be either int or 'input' or 'output' or None found %s" % (x,))
        return int_dims

    def _create_variable(self, name, bound_dimensions, default_val, is_kwarg=True, is_trainable=True):
        int_dims = self._bound_dimensions_to_ints(bound_dimensions)

        with self.name_scope():
            if isinstance(default_val, np.ndarray):
                default_val = self._weight_extender_func(default_val, int_dims)
            elif default_val is None:
                if len(int_dims) == 1:
                    default_val = self._bias_initializer_func(int_dims[0])
                else:
                    default_val = self._weight_initializer_func(int_dims)

            var = tf.Variable(default_val, trainable=(not self._freeze) and is_trainable, name=name)

            self._session.run(tf.variables_initializer([var]))
            self._bound_variables[name] = self._BoundVariable(name, bound_dimensions, var, is_kwarg)
            return var

    def _register_tensor(self, name, bound_dimensions, variable, is_constructor_variable=True):
        """Register a variable that will need to be resized with this layer

        Args:
            name (str): Name used for displaying errors and debuging
            bound_dimensions (tuple of (self.OUTPUT_BOUND_VALUE or self.INPUT_BOUND_VALUE or int)):
            variable (tf.Tensor): The variable to bind
            is_constructor_variable (bool): If true this variable is passed as an arg to the constructor of this class if it
is cloned
        """
        int_dims = self._bound_dimensions_to_ints(bound_dimensions)
        assert tuple(variable.get_shape().as_list()) == tuple(int_dims)
        self._bound_variables[name] = self._BoundVariable(name, bound_dimensions, variable,
is_constructor_variable)

    def _bound_dimensions_contains_input(self, bound_dimensions):
```

```
        return any(x for x in bound_dimensions if x == self.INPUT_BOUND_VALUE or x ==
self.INPUT_DIM_3_BOUND_VALUE)

    def _bound_dimensions_contains_output(self, bound_dimensions):
        return any(x for x in bound_dimensions if x == self.OUTPUT_BOUND_VALUE or x ==
self.OUTPUT_DIM_3_BOUND_VALUE)

    def _get_assign_function(self, name):
        bound_variable = self._bound_variables[name]

        if name not in self._bound_variable_assign_data:
            with self.name_scope():
                placeholder = tf.placeholder(bound_variable.variable.dtype.base_dtype,
                                             shape=self._bound_dimensions_to_ints(bound_variable.dimensions))
                assign_op = tf.assign(bound_variable.variable, placeholder, validate_shape=False)
                self._bound_variable_assign_data[name] = (assign_op, placeholder)

        assign_op, placeholder = self._bound_variable_assign_data[name]
        return lambda value: self.session.run(assign_op, feed_dict={placeholder: value})

    def remove_layer_from_network(self):
        """Attempt to remove this layer from the network, may resize the input layer of the next, when
        removing
        """
        input_layer = self.input_layer
        next_layer = self.next_layer
        if next_layer.input_nodes != input_layer.output_nodes:
            # need to resize layer so there is a match when we cut
            self.resize(input_layer.output_nodes,
                        no_splitting_or_pruning=True)

        self.detach_output()
        input_layer.detach_output()

        input_layer._next_layer = next_layer
        next_layer._input_layer = input_layer

    def detach_output(self):
        """Detaches the connect between this layer and the next layer

        Returns:
            BaseLayer : The next layer, now detached from this layer
        """
        if self._next_layer is None:
            raise ValueError("Cannot detach_output if there is no next layer")
        next_layer = self._next_layer

        next_layer._input_layer = None
        clear_all_lazyprops(next_layer)

        self._next_layer = None
        clear_all_lazyprops(self)

        return next_layer

    def _get_deeper_net_kwargs(self):
        raise NotImplemented()

    def add_intermediate_cloned_layer(self):
        """Add a layer after the current one, that is an exact clone of this layer, but with net 2 deeper net weight
```

```
        initlialization"""
        kwargs = self._get_deeper_net_kwargs()
        if self._batch_normalize_input:
            kwargs['batch_norm_transform'] = np.zeros(shape=kwargs['batch_norm_transform'].shape,
                                                      dtype=kwargs['batch_norm_transform'].dtype)
            kwargs['batch_norm_scale'] = np.ones(shape=kwargs['batch_norm_scale'].shape,
                                                 dtype=kwargs['batch_norm_scale'].dtype)

        self.add_intermediate_layer(lambda x: self.__class__(self, session=self.session, **kwargs))

    def add_intermediate_layer(self, layer_creation_func, *args, **kwargs):
        """Adds a layer to the network between this layer and the next one.

        Args:
            layer_creation_func (BaseLayer->BaseLayer): Method that creates the intermediate layer, takes this
layer as a
                parameter. Any args or kwargs get passed in after passing in this layer
        """
        old_next_layer = self.detach_output()
        new_next_layer = layer_creation_func(self, *args, **kwargs)

        # make sure sizes are correct going forward
        new_next_layer.resize(new_next_layer.get_resizable_dimension_size())

        new_next_layer._next_layer = old_next_layer
        old_next_layer._input_layer = new_next_layer

    @property
    def assign_op(self):
        """Optional tensor flow op that will be set as a dependency of the train step. Useful for things like clamping
        variables, or setting mean/var in batch normalization layer

        Returns:
            tensorflow.Operation or None
        """
        return None

    @property
    def variables(self):
        """Get all the tensorflow variables used in this layer, useful for weight regularization

        Returns:
            Iterable of tf.Variable:
        """
        for bound_variable in self._bound_variables.values():
            yield bound_variable.variable

    @property
    def regularizable_variables(self):
        """variables that can be regularized on this layer"""
        raise NotImplementedError()

    @property
    def variables_all_layers(self):
        """Get all the tensorflow variables used in all connected layers, useful for weight regularization

        Returns:
            Iterable of tf.Variable:
        """
        for layer in self.all_layers:
```

```
            for variable in layer.variables:
                yield variable

    @property
    def regularizable_variables_all_layers(self):
        """variables that can be regularized on all connected layers"""
        for layer in self.all_layers:
            for variable in layer.regularizable_variables:
                yield variable

    def get_parameters_all_layers(self):
        """The number of parameters in this layer

        Returns:
            int
        """
        total = 0

        for layer in self.all_layers:
            total += layer.get_parameters()

        return total

    def get_parameters(self):
        """The number of parameters in this layer

        Returns:
            int
        """
        total = 0

        for bound_variable in self._bound_variables.values():
            total += int(functools.reduce(operator.mul, bound_variable.variable.get_shape()))

        return total

    def has_resizable_dimension(self):
        """True if this layer can be resized, otherwise false

        Returns:
            bool
        """
        return False

    def get_resizable_dimension_size(self):
        """Get the size of the dimension that is resized by the resize method.
        In the future may support multiple of these for conv layers

        Returns:
            int
        """
        return None

    def get_all_resizable_layers(self):
        """Yields all layers connected to this one that are resiable, orders them by how close
        to the input layer they are

        Returns:
            Generator of BaseLayer
        """
```

```
        for layer in self.all_connected_layers:
            if layer.has_resizable_dimension():
                yield layer

    def get_resizable_dimension(self):
        return 0

    def get_resizable_dimension_size_all_layers(self):
        """

        Returns:
            (int,): Tuple of each layer size for each layer that is resizable in the network
        """
        return tuple(layer.get_resizable_dimension_size() for layer in self.get_all_resizable_layers())

    def _get_new_node_count(self, size_multiplier, from_size=None):
        if not self.has_resizable_dimension():
            raise Exception("Can not resize this dimension")

        from_size = from_size or self.get_resizable_dimension_size()

        new_size = int(from_size * size_multiplier)
        # in case the multiplier is too small to changes values
        if abs(new_size - from_size) < self.MINIMUM_GROW_AMOUNT:
            if size_multiplier > 1.:
                new_size = from_size + self.MINIMUM_GROW_AMOUNT
            else:
                new_size = from_size - self.MINIMUM_GROW_AMOUNT

        return new_size

    def _layer_resize_converge(self, data_set_train, data_set_validation,
                               model_evaluation_function,
                               new_size,
                               learning_rate):
        if new_size <= 0:
            logger.info("layer too small stopping downsize")
            return -sys.float_info.max

        self.resize(new_output_nodes=new_size,
                    data_set_train=data_set_train,
                    data_set_validation=data_set_validation)

        self.last_layer.train_till_convergence(data_set_train, data_set_validation,
                                               learning_rate=learning_rate)
        result = model_evaluation_function(self, data_set_validation)
        logger.info("layer resize converge for dim: %s result: %s", self.get_resizable_dimension_size_all_layers(),
                    result)
        return result

    def find_best_size(self, data_set_train, data_set_validation,
                       model_evaluation_function, best_score=None,
                       initial_learning_rate=0.001, tuning_learning_rate=0.0001,
                       grow_only=False, prune_only=False):
        """Attempts to resize this layer to minimize the loss against the validation dataset by resizing this layer

        Args:
            data_set_train (tensor_dynamic.data.data_set.DataSet):
            data_set_validation (tensor_dynamic.data.data_set.DataSet):
            model_evaluation_function (BaseLayer, tensor_dynamic.data.data_set.DataSet -> float): Method for
```

```
judging
            success of training. We try to maximize this
         best_score (float): Best score achieved so far, this is purly for optimization. If it is not passed this is
            calculated in the method
         initial_learning_rate (float): Learning rate to use for first run
         tuning_learning_rate (float): Learning rate to use for subsequent runs, normally smaller than
            initial_learning_rate

      Returns:
         (bool, float) : if we resized, the best score we achieved from the evaluation function
      """
      if not self.has_resizable_dimension():
         raise Exception("Can not resize unresizable layer %s" % (self,))

      if best_score is None:
         self.last_layer.train_till_convergence(data_set_train, data_set_validation,
                                                learning_rate=initial_learning_rate)
         best_score = model_evaluation_function(self, data_set_validation)

      start_size = self.get_resizable_dimension_size_all_layers()
      best_state = self.get_network_state()

      resized = False

      # keep getting bigger until we stop improving
      if not prune_only:
         # try bigger
         new_score = self._layer_resize_converge(data_set_train, data_set_validation,
                                                 model_evaluation_function,
                                                 self._get_new_node_count(self.GROWTH_MULTIPLYER),
                                                 tuning_learning_rate)

         while new_score > best_score:
            resized = True
            best_score = new_score
            best_state = self.get_network_state()

            new_score = self._layer_resize_converge(data_set_train, data_set_validation,
                                                    model_evaluation_function,
                                                    self._get_new_node_count(self.GROWTH_MULTIPLYER),
                                                    tuning_learning_rate)
      if not resized and not grow_only:
         logger.info("From start_size %s Bigger failed, trying smaller", start_size)
         self.set_network_state(best_state)

         new_score = self._layer_resize_converge(data_set_train, data_set_validation,
                                                 model_evaluation_function,
                                                 self._get_new_node_count(self.SHRINK_MULTIPLYER),
                                                 tuning_learning_rate)

         while new_score > best_score:
            resized = True
            best_score = new_score
            best_state = self.get_network_state()
            new_score = self._layer_resize_converge(data_set_train, data_set_validation,
                                                    model_evaluation_function,
                                                    self._get_new_node_count(self.SHRINK_MULTIPLYER),
                                                    tuning_learning_rate)

      # return to the best size we found
```

```
        self.set_network_state(best_state)

        logger.info("From start_size %s Found best was %s", start_size, self.get_resizable_dimension_size())

        return resized, best_score

    def _choose_nodes_to_split(self, desired_size, data_set_train, data_set_validation):
        assert isinstance(desired_size, int)

        current_size = self.get_resizable_dimension_size()

        if desired_size <= current_size:
            raise ValueError("Can't split to get smaller than we are")

        importance = self._get_node_importance(data_set_train, data_set_validation)

        to_split = set()

        while desired_size > current_size + len(to_split):
            max_node = np.argmax(importance)
            importance[max_node] = -sys.float_info.max
            to_split.add(max_node)

        return list(to_split)

    def _choose_nodes_to_prune(self, desired_size, data_set_train, data_set_validation):
        assert isinstance(desired_size, int)

        current_size = self.get_resizable_dimension_size()

        if desired_size >= current_size:
            raise ValueError("Can't prune to size larger than we are")

        importance = self._get_node_importance(data_set_train, data_set_validation)

        to_prune = set()

        while desired_size < current_size - len(to_prune):
            min_node = np.argmin(importance)
            importance[min_node] = sys.float_info.max
            to_prune.add(min_node)

        return list(to_prune)

    def _get_layer_state(self):
        """Returns an object that can be used to set this layer to it's current state and size

        Returns:
            object
        """
        return self.__class__, self.get_resizable_dimension_size(), self.kwargs

    def _set_layer_state(self, state):
        """Set this to the state passed, may cause resizing

        Args:
            state ((type, int, dict)): Object create by self.get_layer_state
        """
        class_type, size, kwargs = state
        assert class_type == type(self)
```

```
        if not hasattr(self, '_bound_variables'):
            return

        if self.get_resizable_dimension_size() != size:
            self.resize(size, no_splitting_or_pruning=True)

        for name, value in kwargs.iteritems():
            assert hasattr(self, '_' + name), 'expected to have property with name %s' % ('_' + name,)

            attribute = getattr(self, '_' + name)

            if isinstance(attribute, tf.Variable):
                self._get_assign_function(name)(value)
            elif type(attribute) == type(value) or isinstance(attribute, type(value)) or isinstance(value,
                                                                                                    type(attribute))\
                    or attribute is None:
                setattr(self, '_' + name, value)
            else:
                raise Exception("Mismatch variable type for %s, existing type was %s new type was %s" %
                                (name, type(attribute), type(value)))

    def get_network_state(self):
        return [layer._get_layer_state() for layer in self.all_connected_layers]

    def get_network_pickle(self):
        return pickle.dumps(self.get_network_state())

    @staticmethod
    def load_network_from_state(state, session):
        last_layer = None

        for type, size, kwargs in state:
            if last_layer is None:
                if 'session' in type.__init__.__func__.func_code.co_varnames:
                    last_layer = type(session=session, **kwargs)
                else:
                    last_layer = type(**kwargs)
            else:
                last_layer = type(last_layer, session=session, **kwargs)

        return last_layer

    @staticmethod
    def load_network_from_pickle(data, session):
        state = pickle.loads(data)

        return BaseLayer.load_network_from_state(state, session)

    def set_network_state(self, state):
        all_current_layers = list(self.all_connected_layers)

        current_layers_iter = iter(all_current_layers)
        state_iter = iter(state)

        while True:
            try:
                next_state = state_iter.next()
            except StopIteration:
                return
```

```
        next_current_layer = current_layers_iter.next()
        class_type, size, kwargs = next_state

        if class_type == type(next_current_layer):
            next_current_layer._set_layer_state(next_state)
        else:  # next layer needs to be removed
            layer_after = current_layers_iter.next()
            assert class_type == type(layer_after)

            next_current_layer.remove_layer_from_network()
            layer_after._set_layer_state(next_state)

@property
def resizable_variables(self):
    raise NotImplementedError()

@lazyprop
def gradients_with_respect_to_error_op(self):
    clear_lazyprop_on_lazyprop_cleared(self, "gradients_with_respect_to_error_op",
                                       self.last_layer, "target_loss_op_predict")

    gradients_ops = []
    for variable in self.resizable_variables:
        gradients_ops.append(tf.gradients(self.last_layer.target_loss_op_predict, variable)[0])

    return gradients_ops

@lazyprop
def hessien_with_respect_to_error_op(self):
    clear_lazyprop_on_lazyprop_cleared(self, "hessien_with_respect_to_error_op",
                                       self, "gradients_with_respect_to_error_op")

    hessian_ops = []
    for variable, gradients in zip(self.resizable_variables, self.gradients_with_respect_to_error_op):
        hessian_ops.append(tf.gradients(gradients, variable)[0])

    # TODO: use tf.hessian in tensorflow 1. also use tf.diag_part
    return hessian_ops
```

tensordynamic/tensor_dynamic/layers/**batch_norm_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer


class BatchNormLayer(BaseLayer):
    def __init__(self, input_layer,
                 session=None,
                 name='BatchNormLayer',
                 running_mean=None,
                 running_var=None,
                 ewma_running_mean=None,
                 ewma_running_var=None,
                 beta=None,
                 gamma=None):
        super(BatchNormLayer, self).__init__(input_layer,
                                             input_layer.output_nodes,
                                             session,
                                             name=name)
        # self._running_mean = self._create_variable(
        #     "running_mean",
        #     (BaseLayer.INPUT_BOUND_VALUE,),
        #     running_mean if running_mean is not None else tf.zeros((self.input_nodes,)))
        # self._running_var = self._create_variable(
        #     "running_var",
        #     (BaseLayer.INPUT_BOUND_VALUE,),
        #     running_var if running_var is not None else tf.ones((self.input_nodes,)))
        #
        # self._ewma = tf.train.ExponentialMovingAverage(decay=.99)
        #
        # self._batch_norm_beta = self._create_variable("beta", (self.INPUT_BOUND_VALUE,),
tf.zeros((self.input_nodes,)),
        #                                               is_kwarg=False, is_trainable=False)
        # self._batch_norm_gamma = self._create_variable("gamma", (self.INPUT_BOUND_VALUE,),
        #                                                tf.ones((self.input_nodes,)),
        #                                                is_kwarg=False,
        #                                                is_trainable=False)

        # self._activation_train = self._update_batch_norm(ewma_running_var=ewma_running_var,
        #                                                  ewma_running_mean=ewma_running_mean)
        # self._activation_predict = self._evaluate()

        self._beta = beta
        self._gamma = gamma

        self._mean, self._var = tf.nn.moments(self._input_layer.activation_train, axes=[0])
        self._register_tensor("mean", (self.INPUT_BOUND_VALUE,), self._mean)
        self._register_tensor("var", (self.INPUT_BOUND_VALUE,), self._var)

        #self._register_variable()

    def _layer_activation(self, input_tensor, is_train):
        return self._batch_normalize(input_tensor, self._mean, self._var)

    # @property
    # def assign_op(self):
    #     return self._assign_op
```

```
    def clone(self, session=None):
        return self.__class__(self.input_layer.clone(session or self._session),
                              session=session or self._session,
                              name=self._name)

    def _update_batch_norm(self, ewma_running_mean=None, ewma_running_var=None):
        "batch normalize + update average mean and variance of layer"
        mean, var = tf.nn.moments(self._input_layer.activation_train, axes=[0])
        assign_mean = self._running_mean.assign(mean)
        assign_var = self._running_var.assign(var)
        self._assign_op = self._ewma.apply([self._running_mean, self._running_var])

        # need to init the variables the ewma creates
        self._session.run(tf.initialize_variables(self._ewma._averages.values()))

        ewma_running_mean_variable = self._ewma.average(self._running_mean)
        if ewma_running_mean is not None:
            self.session.run(tf.assign(ewma_running_mean_variable, ewma_running_mean))

        self._register_tensor('ewma_running_mean', (self.OUTPUT_BOUND_VALUE,),
ewma_running_mean_variable)

        ewma_running_var_variable = self._ewma.average(self._running_var)
        if ewma_running_var is not None:
            self.session.run(tf.assign(ewma_running_var_variable, ewma_running_var))

        self._register_tensor('ewma_running_var', (self.OUTPUT_BOUND_VALUE,), ewma_running_var_variable)

        with tf.control_dependencies([assign_mean, assign_var]):
            return self._batch_normalize_no_resize(self._input_layer.activation_train, mean, var)

    def _evaluate(self):
        mean = self._ewma.average(self._running_mean)
        var = self._ewma.average(self._running_var)
        return self._batch_normalize_no_resize(self._input_layer.activation_predict, mean, var)

    def _batch_normalize_no_resize(self, batch, mean, var):
        # the tf.nn.batch_norm_with_global_normalization only supports convolutional networks so for now we
have to
        # reshape to a convolution normalize then shape back... but can't be resized then... :(

        reshape_to_conv = tf.reshape(batch, [-1, 1, 1, self.input_nodes], name="reshape_to_conv")
        self._r1 = reshape_to_conv
        self._register_tensor("reshape_to_conv", (-1, 1, 1, self.INPUT_BOUND_VALUE), reshape_to_conv,
is_constructor_variable=False)
        batch_normalized = tf.nn.batch_norm_with_global_normalization(reshape_to_conv, mean, var,
                                                                      self._batch_norm_scale,
                                                                      self._batch_norm_transform,
                                                                      0.00001,
                                                                      False)
        reshape_from_conv = tf.reshape(batch_normalized, [-1, self.input_nodes], name="reshape_from_conv")
        self._r2 = reshape_from_conv
        self._register_tensor("reshape_from_conv", (-1, self.INPUT_BOUND_VALUE), reshape_from_conv,
is_constructor_variable=False)

        return reshape_from_conv

    def _batch_normalize(self, batch, mean, var):
        # this version doesn't produce correct numbers when running predict after training
        normalized = ((batch - mean) / tf.sqrt(var + tf.constant(1e-10)))
```

```
        if self._gamma:
            normalized = normalized * self._gamma
        if self._beta:
            normalized = normalized + self._beta
        return normalized

    def resize(self, new_output_nodes=None, input_nodes_to_prune=None, output_nodes_to_prune=None,
               split_output_nodes=None,
               split_input_nodes=None,
               split_nodes_noise_std=None):
        new_output_nodes = new_output_nodes or self.input_layer.output_nodes

        super(BatchNormLayer, self).resize(new_output_nodes=new_output_nodes,
                                           input_nodes_to_prune=input_nodes_to_prune,
                                           output_nodes_to_prune=output_nodes_to_prune,
                                           split_output_nodes=split_output_nodes,
                                           split_input_nodes=split_input_nodes,
                                           split_nodes_noise_std=split_nodes_noise_std)

    @property
    def kwargs(self):
        kwargs = super(BatchNormLayer, self).kwargs
        kwargs['beta'] = self._beta
        kwargs['gamma'] = self._gamma

        return kwargs
```

tensordynamic/tensor_dynamic/layers/**binary_output_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.output_layer import OutputLayer
from tensor_dynamic.lazyprop import lazyprop


class BinaryOutputLayer(OutputLayer):
    def __init__(self, input_layer,
                 session=None,
                 bias=None,
                 weights=None,
                 back_bias=None,
                 freeze=False,
                 weight_extender_func=None,
                 layer_noise_std=None,
                 regularizer_weighting=0.01,
                 name='BinaryOutputLayer'):
        super(BinaryOutputLayer, self).__init__(input_layer, (1,),
                                                session=session,
                                                bias=bias,
                                                weights=weights,
                                                back_bias=back_bias,
                                                freeze=freeze,
                                                weight_extender_func=weight_extender_func,
                                                layer_noise_std=layer_noise_std,
                                                regularizer_weighting=regularizer_weighting,
                                                name=name)

    @lazyprop
    def accuracy_op(self):
```

```
        correct_prediction = tf.equal(
            tf.round(tf.abs(self.activation_predict - self.target_placeholder)), 0)

        return tf.reduce_mean(tf.cast(correct_prediction,
                                      tf.float32))
```

tensordynamic/tensor_dynamic/layers/**categorical_output_layer.py**

```
import tensorflow as tf

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.layers.output_layer import OutputLayer
from tensor_dynamic.lazyprop import lazyprop, clear_lazyprop_on_lazyprop_cleared


class CategoricalOutputLayer(OutputLayer):
    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_bias=None,
                 freeze=False,
                 weight_extender_func=None,
                 layer_noise_std=None,
                 drop_out_prob=None,
                 batch_normalize_input=None,
                 batch_norm_transform=None,
                 batch_norm_scale=None,
                 regularizer_weighting=0.01,
                 regularizer_op=tf.nn.l2_loss,
                 loss_cross_entropy_or_log_prob=True,
                 save_checkpoints=0,
                 name='CategoricalOutputLayer'):
        super(CategoricalOutputLayer, self).__init__(input_layer, output_nodes,
                                                      session=session,
                                                      bias=bias,
                                                      weights=weights,
                                                      back_bias=back_bias,
                                                      freeze=freeze,
                                                      weight_extender_func=weight_extender_func,
                                                      layer_noise_std=layer_noise_std,
                                                      drop_out_prob=drop_out_prob,
                                                      batch_normalize_input=batch_normalize_input,
                                                      batch_norm_transform=batch_norm_transform,
                                                      batch_norm_scale=batch_norm_scale,
                                                      regularizer_weighting=regularizer_weighting,
                                                      regularizer_op=regularizer_op,
```

```
                                         save_checkpoints=save_checkpoints,
                                         name=name)
        self._loss_cross_entropy_or_log_prob = loss_cross_entropy_or_log_prob

    @lazyprop
    def _pre_softmax_activation_predict(self):
        clear_lazyprop_on_lazyprop_cleared(self, '_pre_softmax_activation_predict', self.input_layer,
                                           'activation_predict')
        with self.name_scope(is_predict=True):
            input_activation = self._process_input_activation_predict(self.input_layer.activation_predict)
            return self._layer_activation(input_activation, False)

    @lazyprop
    def _pre_softmax_activation_train(self):
        clear_lazyprop_on_lazyprop_cleared(self, '_pre_softmax_activation_train', self.input_layer,
                                           'activation_train')
        with self.name_scope(is_train=True):
            input_activation = self._process_input_activation_train(self.input_layer.activation_train)
            return self._layer_activation(input_activation, True)

    @lazyprop
    def activation_predict(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'activation_predict', self.input_layer, 'activation_predict')
        with self.name_scope(is_predict=True):
            return tf.nn.softmax(self._pre_softmax_activation_predict)

    @lazyprop
    def activation_train(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'activation_train', self.input_layer, 'activation_train')
        with self.name_scope(is_train=True):
            return tf.nn.softmax(self._pre_softmax_activation_train)

    @lazyprop
    def target_loss_op_train(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'target_loss_op_train', self.input_layer)
        with self.name_scope(is_train=True):
            return self._target_loss_op(self._pre_softmax_activation_train)

    @lazyprop
    def target_loss_op_predict(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'target_loss_op_predict', self.input_layer)
        with self.name_scope(is_predict=True):
            return self._target_loss_op(self._pre_softmax_activation_predict)

    def _target_loss_op(self, input_tensor):
        if self._loss_cross_entropy_or_log_prob:
            loss = tf.nn.softmax_cross_entropy_with_logits(logits=input_tensor, labels=self._target_placeholder)
        else:
            loss = -tf.log(tf.reduce_sum(tf.nn.softmax(input_tensor) * self.target_placeholder, 1))

        return tf.reduce_sum(loss)

    @lazyprop
    def accuracy_op(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'accuracy_op', self.input_layer)
        return tf.reduce_mean(tf.cast(tf.nn.in_top_k(self._pre_softmax_activation_predict,
tf.argmax(self.target_placeholder, 1), 1),
                                      tf.float32))

    def accuracy(self, data_set):
```

```
        """Get the accuracy of our predictions against the real targets, returns a value in the range 0. to 1.

        Args:
            data_set (DataSet):

        Returns:
            float: accuracy in the range 0. to 1.
        """
        assert isinstance(data_set, DataSet)

        return self.session.run(self.accuracy_op,
                                feed_dict={self.input_placeholder: data_set.features,
                                           self._target_placeholder: data_set.labels})

    @lazyprop
    def log_probability_of_targets_op(self):
        clear_lazyprop_on_lazyprop_cleared(self, 'log_probability_of_targets_op', self.input_layer)
        return tf.reduce_sum(tf.log(tf.reduce_sum(tf.nn.softmax(self.activation_predict) * self.target_placeholder,
1)))

    @property
    def regularizable_variables(self):
        yield self._weights

    @property
    def kwargs(self):
        kwargs = super(OutputLayer, self).kwargs

        del kwargs['bactivate']
        del kwargs['bactivation_loss_func']
        del kwargs['non_liniarity']

        kwargs['loss_cross_entropy_or_log_prob'] = self._loss_cross_entropy_or_log_prob

        return kwargs
```

tensordynamic/tensor_dynamic/layers/**convolutional_layer.py**

```
import numpy as np

import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.node_importance import node_importance_optimal_brain_damage


class ConvolutionalLayer(BaseLayer):
```

```
    MINIMUM_GROW_AMOUNT = 1

    def __init__(self,
                 input_layer,
                 convolution_dimensions,  # 3d (width, height, convolutions)
                 stride=(1, 1, 1),
                 padding='SAME',  # only same currently supported
                 session=None,
                 weights=None,
                 bias=None,
                 weight_extender_func=None,
                 weight_initializer_func=None,
                 bias_initializer_func=None,
                 node_importance_func=None,
                 name='ConvolutionalLayer',
                 freeze=False,
                 layer_noise_std=None,
                 drop_out_prob=None,
                 batch_normalize_input=False,
                 non_liniarity=tf.nn.relu):
        assert len(input_layer.output_nodes) == 3, "expected input to have 3 dimensions"
        assert len(convolution_dimensions) == 3, "expected output to have 3 dimensions"
        self._convolution_dimensions = convolution_dimensions

        output_nodes = (input_layer.output_nodes[0] / stride[0],
                        input_layer.output_nodes[1] / stride[1],
                        convolution_dimensions[2] / stride[2])

        super(ConvolutionalLayer, self).__init__(input_layer,
                                                 output_nodes,
                                                 session=session,
                                                 weight_extender_func=weight_extender_func,
                                                 weight_initializer_func=weight_initializer_func,
                                                 bias_initializer_func=bias_initializer_func,
                                                 freeze=freeze,
                                                 layer_noise_std=layer_noise_std,
                                                 drop_out_prob=drop_out_prob,
                                                 batch_normalize_input=batch_normalize_input,
                                                 name=name)

        self._weights = self._create_variable("weights",
                                              (convolution_dimensions[0], convolution_dimensions[1],
                                               BaseLayer.INPUT_DIM_3_BOUND_VALUE,
BaseLayer.OUTPUT_DIM_3_BOUND_VALUE),
                                              weights)
        self._bias = self._create_variable("bias",
                                           (BaseLayer.OUTPUT_DIM_3_BOUND_VALUE,),
                                           bias)
        self._node_importance_func = self._get_property_or_default(node_importance_func,
                                                                   '_node_importance_func',
                                                                   node_importance_optimal_brain_damage)
        self._stride = stride
        self._padding = padding
        self._non_liniarity = non_liniarity

    @property
    def convolutional_nodes(self):
        return self._convolution_dimensions

    def _layer_activation(self, input_activation, is_train):
```

```
        x = tf.nn.conv2d(input_activation, self._weights, strides=(1,) + self._stride,
                         padding=self._padding)
        x = tf.nn.bias_add(x, self._bias)
        return self._non_liniarity(x)

    @property
    def regularizable_variables(self):
        yield self._weights

    @property
    def resizable_variables(self):
        yield self._weights
        yield self._bias

    def resize(self, new_output_nodes=None,
               output_nodes_to_prune=None,
               input_nodes_to_prune=None,
               split_output_nodes=None,
               split_input_nodes=None,
               data_set_train=None,
               data_set_validation=None,
               no_splitting_or_pruning=False,
               split_nodes_noise_std=.1):
        if isinstance(new_output_nodes, int):
            temp = list(self.output_nodes)
            temp[2] = new_output_nodes / self._stride[2]
            new_output_nodes = tuple(temp)

        super(ConvolutionalLayer, self).resize(new_output_nodes,
                                               output_nodes_to_prune=output_nodes_to_prune,
                                               input_nodes_to_prune=input_nodes_to_prune,
                                               split_output_nodes=split_output_nodes,
                                               split_input_nodes=split_input_nodes,
                                               split_nodes_noise_std=split_nodes_noise_std,
                                               data_set_train=data_set_train,
                                               data_set_validation=data_set_validation,
                                               no_splitting_or_pruning=no_splitting_or_pruning)

    def clone(self, session=None):
        """Produce a clone of this layer AND all connected upstream layers

        Args:
            session (tensorflow.Session): If passed in the clone will be created with all variables initialised in this
session
                                          If None then the current session of this layer is used

        Returns:
            tensorflow_dynamic.BaseLayer: A copy of this layer and all upstream layers
        """
        new_self = self.__class__(self.input_layer.clone(session or self.session),
                                  self.convolutional_nodes,
                                  session=session or self._session,
                                  **self.kwargs)

        return new_self

    def has_resizable_dimension(self):
        return True

    def get_resizable_dimension_size(self):
```

```
        return self.convolutional_nodes[2]

    def _get_node_importance(self):
        # simplest way to do this just sum the weights in each convolution
        weights = self.session.run(self._weights)
        bias = self.session.run(self._bias)

        # group everything by convolutional output node
        weights = weights.transpose(3, 0, 1, 2).reshape(self.convolutional_nodes[2], -1)
        importance = [np.sum(weights[i]) + bias[i] for i in range(self.convolutional_nodes[2])]
        return importance

    def get_resizable_dimension(self):
        return 2

    @property
    def kwargs(self):
        kwargs = super(ConvolutionalLayer, self).kwargs

        kwargs['stride'] = self._stride
        kwargs['padding'] = self._padding
        kwargs['non_liniarity'] = self._non_liniarity

        return kwargs
```

tensordynamic/tensor_dynamic/layers/**denoising_source_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.tf_loss_functions import squared_loss
from tensor_dynamic.weight_functions import noise_weight_extender


# CURRENTLY BROKEN
class DenoisingSourceLayer(HiddenLayer):
    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_bias=None,
                 back_weights=None,
                 freeze=False,
                 a1=None,
                 a2=None,
                 a3=None,
                 a4=None,
                 a5=None,
                 a6=None,
                 a7=None,
                 a8=None,
                 a9=None,
                 a10=None,
                 non_liniarity=tf.nn.relu,
                 bactivation_loss_func=squared_loss,
                 weight_extender_func=noise_weight_extender,
                 unsupervised_cost=1.,
                 supervised_cost=1.,
                 noise_std=None,
                 name='BackWeightLayer'):
        super(DenoisingSourceLayer, self).__init__(input_layer, output_nodes,
                                                   session=session,
                                                   bias=bias,
                                                   weights=weights,
                                                   bactivate=True,
                                                   freeze=freeze,
                                                   non_liniarity=non_liniarity,
                                                   weight_extender_func=weight_extender_func,
                                                   bactivation_loss_func=bactivation_loss_func,
                                                   unsupervised_cost=unsupervised_cost,
                                                   supervised_cost=supervised_cost,
                                                   noise_std=noise_std,
                                                   name=name)
        assert len(self.input_nodes) == 1
        assert len(self.output_nodes) == 1
```

```
        self._a1 = self._create_variable('a1', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a1 if a1 is not None else tf.zeros(self.input_nodes))
        self._a2 = self._create_variable('a2', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a2 if a2 is not None else tf.ones(self.input_nodes))
        self._a3 = self._create_variable('a3', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a3 if a3 is not None else tf.zeros(self.input_nodes))
        self._a4 = self._create_variable('a4', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a4 if a4 is not None else tf.zeros(self.input_nodes))
        self._a5 = self._create_variable('a5', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a5 if a5 is not None else tf.zeros(self.input_nodes))
        self._a6 = self._create_variable('a6', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a6 if a6 is not None else tf.zeros(self.input_nodes))
        self._a7 = self._create_variable('a7', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a7 if a7 is not None else tf.ones(self.input_nodes))
        self._a8 = self._create_variable('a8', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a8 if a8 is not None else tf.zeros(self.input_nodes))
        self._a9 = self._create_variable('a9', (BaseLayer.INPUT_BOUND_VALUE,),
                                         a9 if a9 is not None else tf.zeros(self.input_nodes))
        self._a10 = self._create_variable('a10', (BaseLayer.INPUT_BOUND_VALUE,),
                                          a10 if a10 is not None else tf.zeros(self.input_nodes))

    def _gaussian_denoise(self, input_corrupted, activation):
        mu = self._a1 * tf.sigmoid(self._a2 * activation + self._a3) + self._a4 * activation + self._a5
        v = self._a6 * tf.sigmoid(self._a7 * activation + self._a8) + self._a9 * activation + self._a10

        z_est = (input_corrupted - mu) * v + mu

        return z_est

    @lazyprop
    def bactivation_train(self):
        return self._gaussian_denoise(self.activation_corrupted, self.activation_train)

    @lazyprop
    def bactivation_predict(self):
        return self._gaussian_denoise(self.activation_predict, self.activation_predict)

    @property
    def kwargs(self):
        kwargs = super(DenoisingSourceLayer, self).kwargs

        # bactivate is not optional for these layers
        del kwargs['bactivate']

        return kwargs
```

tensordynamic/tensor_dynamic/layers/**duel_state_relu_layer.py**

```
import math

import operator
import tensorflow as tf
import numpy as np
from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.weight_functions import noise_weight_extender
```

```
class DuelStateReluLayer(HiddenLayer):
    ACTIVE_THRESHOLD = 0.25  # 0.2

    def __init__(self,
                 input_layer,
                 output_nodes,
                 width_binarizer_constant=1e-4,
                 width_regularizer_constant=1e-2,
                 inactive_nodes_to_leave=3,
                 session=None,
                 weights=None,
                 bias=None,
                 width=None,
                 noise_std=None,
                 non_liniarity=tf.nn.relu,
                 supervised_cost=1.,
                 unsupervised_cost=1.,
                 weight_extender_func=noise_weight_extender,
                 name='DuelStateReluLayer',
                 freeze=False):
        super(DuelStateReluLayer, self).__init__(input_layer, output_nodes,
                                                 session=session, weight_extender_func=weight_extender_func,
                                                 weights=weights,
                                                 bias=bias,
                                                 bactivate=False,
                                                 noise_std=noise_std,
                                                 supervised_cost=supervised_cost,
                                                 unsupervised_cost=unsupervised_cost,
                                                 non_liniarity=non_liniarity,
                                                 name=name,
                                                 freeze=freeze)
        self._width = self._create_variable("width",
                                            (BaseLayer.OUTPUT_BOUND_VALUE,),
                                            width if width is not None else np.ones(self.output_nodes,
                                                                                    dtype=np.float32))
        self._width_regularizer_constant = width_regularizer_constant
        self._width_binarizer_constant = width_binarizer_constant
        self._inactive_nodes_to_leave = inactive_nodes_to_leave

    def _layer_activation(self, input_activation):
        activation = super(DuelStateReluLayer, self)._layer_activation(input_activation)
        return activation * self._width

    def unsupervised_cost_train(self):
        return tf.reduce_sum(self._width * (1 - self._width)) * self._width_binarizer_constant + \
            tf.reduce_sum(self._width) * self._width_regularizer_constant

    @property
    def kwargs(self):
        kwargs = super(DuelStateReluLayer, self).kwargs

        # bactivate is not optional for these layers
        del kwargs['bactivate']
        del kwargs['bactivation_loss_func']

        kwargs['width_regularizer_constant'] = self._width_regularizer_constant
        kwargs['width_binarizer_constant'] = self._width_binarizer_constant
        kwargs['inactive_nodes_to_leave'] = self._inactive_nodes_to_leave
```

```
        return kwargs

    def width(self):
        """
        return a 1D array of the widths used for the layer
        """
        return self.session.run(self._width)

    def active_nodes(self):
        return len([x for x in np.abs(self.width()) if x > self.ACTIVE_THRESHOLD])

    def inactive_nodes(self):
        return self._output_nodes - self.active_nodes()

    def prune(self, inactive_nodes_to_leave=3):
        """
        Removes inactive nodes from the layer

        Parameters
        ----------
        inactive_nodes_to_leave: int
            Number of inactive nodes we want left after pruning

        Returns
        -------
        bool: True we we pruned nodes, otherwise False
        """
        # may need to validate we aren't the output layer...
        active_nodes = self.active_nodes()
        nodes_to_prune = self.output_nodes - (active_nodes + inactive_nodes_to_leave)
        if nodes_to_prune <= 0:
            # no need to prune if we have only 1 inactive node
            # TODO resize so as to leave 1 inactive node?
            return False

        # find the nodes_to_prune least active nodes
        width = self.width()
        width_abs = np.abs(width)
        width_sorted = sorted(width_abs, reverse=False)
        prune_below_width = width_sorted[nodes_to_prune - 1]
        prune_indexes = [i for i, x in enumerate(width_abs) if x <= prune_below_width]

        self.resize(output_nodes_to_prune=prune_indexes)

        print("layer %s pruned node size now %s" % (self.layer_number, self.output_nodes))

        return True

    def grow(self, inactive_nodes_to_leave=3):
        active_nodes = len([x for x in np.abs(self.width()) if x > self.active_nodes()])
        inactive_nodes = self.output_nodes - active_nodes
        if inactive_nodes >= inactive_nodes_to_leave:
            return False

        width = self.width()
        width_abs = np.abs(width)
        max_index = max(enumerate(width_abs), key=operator.itemgetter(1))[0]

        # add some nodes
        self.resize(split_output_nodes=[max_index])
```

```
        print("layer %s added node size now %s" % (self.layer_number, self.output_nodes))

        # set newly created node to active
        width = np.append(width, 1.0)
        self._session.run(self._width.assign(width))

    def resize(self, new_output_nodes=None, output_nodes_to_prune=None, input_nodes_to_prune=None,
               split_output_nodes=None,
               split_input_nodes=None,
               split_nodes_noise_std=.01):
        width = self.width()
        output_nodes_increase = (new_output_nodes or self._output_nodes) - self._output_nodes

        super(DuelStateReluLayer, self).resize(new_output_nodes, output_nodes_to_prune,
input_nodes_to_prune,
                                               split_output_nodes, split_input_nodes, split_nodes_noise_std)

        if output_nodes_increase > 0:
            # set newly created node to active
            width = np.append(width, [1.0]*output_nodes_increase)
            self.session.run(tf.assign(self._width, width, validate_shape=False))

    @property
    def assign_op(self):
        return self._width.assign(tf.clip_by_value(self._width, 0.01, 0.99))
```

tensordynamic/tensor_dynamic/layers/**flatten_layer.py**

```
import functools

import operator

from tensor_dynamic.layers.base_layer import BaseLayer
import tensorflow as tf

from tensor_dynamic.lazyprop import lazyprop, clear_all_lazyprops


class FlattenLayer(BaseLayer):
    def __init__(self,
                 input_layer,
                 session=None,
                 name='FlattenLayer'):
        assert len(input_layer.output_nodes) > 1, "expected multiple input dims"
        output_nodes = functools.reduce(operator.mul, input_layer.output_nodes)

        super(FlattenLayer, self).__init__(input_layer,
                                           (output_nodes,),
                                           session=session,
                                           name=name)

    def _layer_activation(self, input_activation, is_train):
        # TODO can this be done using tf.shape? like input noise?
        return tf.reshape(input_activation, [-1, self.output_nodes[0]])

    def resize(self, new_output_nodes=None,
               output_nodes_to_prune=None,
               input_nodes_to_prune=None,
               split_output_nodes=None,
               split_input_nodes=None, split_nodes_noise_std=.1):
        output_nodes = (functools.reduce(operator.mul, self.input_layer.output_nodes),)

        if self.output_nodes != output_nodes:
            self._output_nodes = output_nodes

            # can't resize the tf.reshape so just regen everything
            clear_all_lazyprops(self)
            for layer in self.downstream_layers:
                clear_all_lazyprops(layer)

            if self.next_layer is not None and self.next_layer._resize_needed():
                # TODO: D.S make sure resize is consistant, i.e new nodes are not just created on the end...
                # Must do this at some point
                self._next_layer.resize(input_nodes_to_prune=output_nodes_to_prune,
split_input_nodes=split_output_nodes)

    def clone(self, session=None):
        """Produce a clone of this layer AND all connected upstream layers
```

```
        Args:
            session (tensorflow.Session): If passed in the clone will be created with all variables initialised in this
session
                                          If None then the current session of this layer is used

        Returns:
            tensorflow_dynamic.BaseLayer: A copy of this layer and all upstream layers
        """
        new_self = self.__class__(self.input_layer.clone(session or self.session),
                                  session=session or self._session,
                                  name=self._name)

        return new_self

    @property
    def regularizable_variables(self):
        return
        yield
```

tensordynamic/tensor_dynamic/layers/**hidden_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.node_importance import node_importance_by_square_sum
from tensor_dynamic.tf_loss_functions import squared_loss
from tensor_dynamic.weight_functions import net_2_deeper_net
```

```
class HiddenLayer(BaseLayer):
    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_bias=None,
                 bactivate=False,
                 freeze=False,
                 non_liniarity=None,
                 weight_extender_func=None,
                 weight_initializer_func=None,
                 bias_initializer_func=None,
                 layer_noise_std=None,
                 drop_out_prob=None,
                 bactivation_loss_func=None,
                 node_importance_func=None,
                 batch_normalize_input=None,
                 batch_norm_transform=None,
                 batch_norm_scale=None,
                 name='Layer'):
        super(HiddenLayer, self).__init__(input_layer,
                                          output_nodes,
                                          session=session,
                                          weight_extender_func=weight_extender_func,
                                          weight_initializer_func=weight_initializer_func,
                                          bias_initializer_func=bias_initializer_func,
                                          layer_noise_std=layer_noise_std,
                                          drop_out_prob=drop_out_prob,
                                          batch_normalize_input=batch_normalize_input,
                                          batch_norm_transform=batch_norm_transform,
                                          batch_norm_scale=batch_norm_scale,
                                          freeze=freeze,
                                          name=name)
        self._non_liniarity = self._get_property_or_default(non_liniarity, '_non_liniarity', tf.nn.sigmoid)
        self._bactivate = bactivate
        self._bactivation_loss_func = self._get_property_or_default(bactivation_loss_func,
'_bactivation_loss_func',
                                                                    squared_loss)
        self._node_importance_func = self._get_property_or_default(node_importance_func,
'_node_importance_func',
                                                                   node_importance_by_square_sum)

        self._weights = self._create_variable("weights",
                                              (BaseLayer.INPUT_BOUND_VALUE, BaseLayer.OUTPUT_BOUND_VALUE),
                                              weights)

        self._bias = self._create_variable("bias",
                                           (BaseLayer.OUTPUT_BOUND_VALUE,),
                                           bias)

        if self.bactivate:
            self._back_bias = self._create_variable("back_bias",
                                                    (BaseLayer.INPUT_BOUND_VALUE,),
                                                    back_bias)
        else:
            self._back_bias = None

    @property
    def weights(self):
        return self._weights.eval(self.session)
```

```
@weights.setter
def weights(self, value):
    self._get_assign_function('weights')(value)

@property
def bias(self):
    return self._bias.eval(self.session)

@weights.setter
def bias(self, value):
    self._get_assign_function('bias')(value)

@property
def bactivate(self):
    return self._bactivate

@property
def kwargs(self):
    kwargs = super(HiddenLayer, self).kwargs

    kwargs['bactivate'] = self.bactivate
    kwargs['bactivation_loss_func'] = self._bactivation_loss_func
    kwargs['non_liniarity'] = self._non_liniarity

    return kwargs

@property
def has_bactivation(self):
    return self.bactivate

def _layer_activation(self, input_activation, is_train):
    name = '_mat_mul_is_train_equal_' + str(is_train)
    mat_mul = tf.matmul(input_activation, self._weights)

    # self._register_tensor(name, (None, BaseLayer.INPUT_BOUND_VALUE), mat_mul)
    # this is a bit hacky... but the above commented out line is not working...
    self.__dict__[name] = mat_mul
    return self._non_liniarity(mat_mul + self._bias)

def _layer_bactivation(self, activation, is_train):
    if self.bactivate:
        return self._non_liniarity(
            tf.matmul(activation, tf.transpose(self._weights)) + self._back_bias)

@property
def non_liniarity(self):
    return self._non_liniarity

def supervised_cost_train(self, targets):
    if not self.next_layer:
        return tf.reduce_mean(tf.reduce_sum(tf.square(self.activation_train - targets), 1))
    else:
        return None

@lazyprop
def bactivation_loss_train(self):
    return tf.reduce_mean(tf.reduce_sum(tf.square(self.bactivation_train - self.input_layer.activation_train), 1))

@lazyprop
```

```
    def bactivation_loss_predict(self):
        return tf.reduce_mean(
            tf.reduce_sum(tf.square(self.bactivation_predict - self.input_layer.activation_predict), 1))

    def has_resizable_dimension(self):
        return True

    def get_resizable_dimension_size(self):
        return self.output_nodes[0]

    def _get_node_importance(self, data_set_train, data_set_validation):
        return self._node_importance_func(self, data_set_train, data_set_validation)

    def _get_deeper_net_kwargs(self):
        kwargs = self.kwargs
        weights, bias = net_2_deeper_net(kwargs['bias'])
        kwargs['bias'] = bias
        kwargs['weights'] = weights
        return kwargs

    @property
    def regularizable_variables(self):
        yield self._weights

    @property
    def resizable_variables(self):
        yield self._weights
        yield self._bias


if __name__ == '__main__':
    with tf.Session() as session:
        input_p = tf.placeholder("float", (None, 10))
        layer = HiddenLayer(input_p, 20, session=session)
        layer.activation.get_shape()
```

tensordynamic/tensor_dynamic/layers/**highway_layer.py judal**

```
# from tensor_dynamic.layers.base_layer import BaseLayer
# from tensor_dynamic.lazyprop import lazyprop
#
# # TODO...
# class HighwayLayer(BaseLayer):
#     def __init__(self, first_layer, second_layer=None, session=None, name=None):
#         assert first_layer.output_nodes == second_layer.output_nodes
#         super(HighwayLayer, self).__init__(second_layer, first_layer.output_nodes, session=session,
name=name)
#         self._carry_layer = first_layer
#         self._gate = # this should be a layer itself with strong negative bias
self._create_variable((BaseLayer.INPUT_BOUND_VALUE,), self)
#
#     @lazyprop
#     def activation(self):
#         return self._gate * self._carry_layer.activation + ((1 - self._gate) * self.input_layer.activation)
#
#     def resize(self):
#         raise NotImplementedError()
```

tensordynamic/tensor_dynamic/layers/**input_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.lazyprop import lazyprop


class InputLayer(BaseLayer):
    def __init__(self, input_nodes, session=None, layer_noise_std=None, drop_out_prob=None, name='Input'):
        """Input layer to a neural network

        Args:
            input_nodes (tensorflow.placeholder or (int) or int): If an int then a tensorflow.placeholder is created
                with dimensions (None, placholder) if a tuple a placeholder is created of that dimension
            name(str): the name for this layer
        """
        if isinstance(input_nodes, int):
            input_nodes = (input_nodes,)
        if isinstance(input_nodes, (tuple, list)):
            input_nodes = tf.placeholder('float', (None,) + input_nodes)
            self._output_nodes = tuple(int(x) for x in input_nodes.get_shape()[1:])
        elif isinstance(input_nodes, tf.Tensor):
            # assume it's a placeholder
            self._output_nodes = tuple(int(x) for x in input_nodes.get_shape()[1:])
        else:
            raise TypeError("Expected input_nodes to be int or tuple")

        self._name = name
        self._placeholder = input_nodes
        self._next_layer = None
        self._input_layer = None
        self._layer_noise_std = layer_noise_std
        self._drop_out_prob = drop_out_prob
        self._session = session

    @property
    def activation(self):
        return self._placeholder

    @property
    def activation_train(self):
        tensor = self._placeholder
        if self._drop_out_prob:
            tensor = tf.nn.dropout(tensor, self._drop_out_prob)

        if self._layer_noise_std is not None:
            tensor = tensor + tf.random_normal(tf.shape(tensor),
                                               stddev=self._layer_noise_std)
        return tensor

    @property
    def activation_predict(self):
        return self._placeholder

    @property
    def first_layer(self):
        return self

    @property
    def bactivate(self):
        return False
```

```
    @property
    def input_shape(self):
        raise Exception("Input layer has no input shape")

    @property
    def input_placeholder(self):
        return self._placeholder

    @property
    def is_input_layer(self):
        return True

    def clone(self, session=None):
        return self.__class__(**self.kwargs)

    @property
    def variables(self):
        return ()

    def _layer_activation(self, _1, _2):
        pass

    def get_parameters(self):
        return 0

    @property
    def kwargs(self):
        kwargs = {
            'input_nodes': self.output_nodes,
            'name': self._name,
            'layer_noise_std': self._layer_noise_std,
            'drop_out_prob': self._drop_out_prob}
        return kwargs

    @property
    def regularizable_variables(self):
        return
        yield


# Not currently working...
class SemiSupervisedInputLayer(InputLayer):
    def __init__(self, input_dim, name='Input'):
        if isinstance(input_dim, tuple):
            supervised = tf.placeholder('float', input_dim)
            unsupervised = tf.placeholder('float', input_dim)
        elif isinstance(input_dim, int):
            supervised = tf.placeholder('float', (None, input_dim))
            unsupervised = tf.placeholder('float', (None, input_dim))

        super(SemiSupervisedInputLayer, self).__init__(supervised, name=name)
        self._unsupervised_placeholder = unsupervised

    @property
    def unsupervised_placeholder(self):
        return self._unsupervised_placeholder

    @lazyprop
    def labeled_input_size(self):
```

```
        return tf.shape(self._placeholder)[1]
```

tensordynamic/tensor_dynamic/layers/**ladder_layer.py**

```
import math

import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.input_layer import SemiSupervisedInputLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.weight_functions import noise_weight_extender

join = lambda l, u: tf.concat(0, [l, u], name="join")
labeled = lambda x, labeled_size: tf.slice(x, [0, 0], [labeled_size, -1], name="slice_unlabeled") if x is not None
else x
unlabeled = lambda x, labeled_size: tf.slice(x, [labeled_size, 0], [-1, -1], name="slice_labeled") if x is not None
else x
split_lu = lambda x, labeled_size: (labeled(x, labeled_size), unlabeled(x, labeled_size))


class LadderLayer(BaseLayer):
    NOISE_STD = 0.3

    def __init__(self, input_layer,
                 output_nodes,
                 denoising_cost=1.,
                 session=None,
                 beta=None,
                 weights=None,
                 back_weights=None,
                 non_liniarity=tf.nn.relu,
                 weight_extender_func=noise_weight_extender,
                 freeze=False,
                 name="ladder"):
```

```
        super(LadderLayer, self).__init__(input_layer,
                                          output_nodes,
                                          session=session,
                                          weight_extender_func=weight_extender_func,
                                          freeze=freeze,
                                          name=name)
        if not isinstance(self.first_layer, SemiSupervisedInputLayer):
            raise Exception("To use a ladder network you must have a
tensor_dynamic.layers.input_layer.SemiSupervisedInputLayer as the input layer to the network")

        self._denoising_cost = denoising_cost
        self._non_liniarity = non_liniarity

        self._weights = self._create_variable("weights",
                                              (BaseLayer.INPUT_BOUND_VALUE, BaseLayer.OUTPUT_BOUND_VALUE),
                                              weights if weights is not None else tf.random_normal(
                                                  (self.input_nodes, self.output_nodes),
                                                  stddev=1. / math.sqrt(self.input_nodes)))

        self._back_weights = self._create_variable("back_weights",
                                                   (BaseLayer.OUTPUT_BOUND_VALUE, BaseLayer.INPUT_BOUND_VALUE),
                                                   back_weights if back_weights is not None else tf.random_normal(
                                                       (self.output_nodes, self.input_nodes),
                                                       stddev=1. / math.sqrt(self.output_nodes)))

        self._beta = self._create_variable("beta",
                                           (BaseLayer.OUTPUT_BOUND_VALUE,),
                                           beta if beta is not None else tf.zeros([self.output_nodes]))
        """values generating mean of output"""

        self.bn_assigns = []

        with self.name_scope():
            # self._running_mean = tf.Variable(tf.constant(0.0, shape=[self.output_nodes]), trainable=False,
            #                                  name="running_mean")
            # self._running_var = tf.Variable(tf.constant(1.0, shape=[self.output_nodes]), trainable=False,
            #                                 name="running_var")
            # self._ewma = tf.train.ExponentialMovingAverage(decay=0.99)

            self.z_pre_corrupted = tf.matmul(self._input_corrupted, self._weights, name="z_pre_corrupted")

            z_pre_corrupted_labeled, z_pre_corrupted_unlabeled = split_lu(self.z_pre_corrupted,
self.first_layer.labeled_input_size)

            self.z_pre_clean = tf.matmul(self.input_layer.activation_predict, self._weights, name="z_pre_clean")

            z_pre_clean_labeled, z_pre_clean_unlabeled = split_lu(self.z_pre_clean,
self.first_layer.labeled_input_size)

            self.mean_corrupted_unlabeled, self.variance_corrupted_unlabeled =
tf.nn.moments(z_pre_corrupted_unlabeled,
                                                                                                axes=[0])

            self.mean_clean_unlabeled, self.variance_clean_unlabeled = tf.nn.moments(z_pre_clean_unlabeled,
axes=[0])

            self.z_corrupted = join(self.batch_normalization(z_pre_corrupted_labeled),
                                    self.batch_normalization(z_pre_corrupted_unlabeled, self.mean_corrupted_unlabeled,
                                                             self.variance_corrupted_unlabeled)) + \
                               tf.random_normal(tf.shape(self.z_pre_corrupted),
```

```
                                         stddev=self.NOISE_STD)

        self.z_clean = join(self._update_batch_normalization(z_pre_clean_labeled),
                            self.batch_normalization(z_pre_clean_unlabeled, self.mean_clean_unlabeled,
                                                     self.variance_clean_unlabeled))

        # if session:
        #     session.run(tf.initialize_variables([self._running_mean,
        #                                          self._running_var]))

    @lazyprop
    def _input_corrupted(self):
        if isinstance(self.input_layer, LadderLayer):
            return self.input_layer.activation_train
        else:
            return self.input_layer.activation_predict + tf.random_normal(tf.shape(self.input_layer.activation_predict),
                                                                           stddev=self.NOISE_STD)

    @lazyprop
    def activation_train(self):
        print "Corrupt Act ", self.layer_number, ": ", self.input_nodes, " -> ", self.output_nodes
        return self._activation_method(self.z_corrupted)

    @lazyprop
    def activation_predict(self):
        print "Clean Act ", self.layer_number, ": ", self.input_nodes, " -> ", self.output_nodes
        # z = self.update_batch_normalization(self)
        # TODO: add back in update batch norm
        return self._activation_method(self.z_clean)

    @lazyprop
    def z_est(self):
        print "Layer ", self.layer_number, ": ", self.output_nodes, " -> ", self.input_nodes, ", denoising cost: ",
self._denoising_cost

        u = tf.matmul(self.next_layer.z_est, self._back_weights, name="u")
        u = self.batch_normalization(u)
        # self._input_corrupted ?? this is changed?
        return self._g_gauss(unlabeled(self.input_z_corrupted, self.first_layer.labeled_input_size), u)

    @lazyprop
    def z_est_bn(self):
        if isinstance(self.input_layer, LadderLayer):
            return (self.z_est - self.input_layer.mean_clean_unlabeled) / self.input_layer.variance_clean_unlabeled
        else:
            # no norm that layer
            return self.z_est / 1 - 1e-10

    @property
    def bactivation_train(self):
        return self.z_est

    @property
    def bactivation_predict(self):
        #maybe this should be from an uncorrupted forward pass?
        return self.z_est

    @staticmethod
    def batch_normalization(batch, mean=None, var=None):
        if mean is None or var is None:
```

```
            mean, var = tf.nn.moments(batch, axes=[0], name="batch_normalization")
        return (batch - mean) / tf.sqrt(var + 1e-10)

    def _update_batch_normalization(self, batch):
        "batch normalize + update average mean and variance of layer"
        # mean, var = tf.nn.moments(batch, axes=[0])
        # assign_mean = self._running_mean.assign(mean)
        # assign_var = self._running_var.assign(var)

        # self.bn_assigns.append(self._ewma.apply([self._running_mean, self._running_var]))
        # with tf.control_dependencies([assign_mean, assign_var]):
        #     return (batch - mean) / tf.sqrt(var + 1e-10)
        return self.batch_normalization(batch)

    # @property
    # def assign_op(self):
    #     return self.bn_assigns

    @property
    def input_z_clean(self):
        if isinstance(self.input_layer, LadderLayer):
            return self.input_layer.z_clean
        else:
            return self.input_layer.activation

    @property
    def input_z_corrupted(self):
        if isinstance(self.input_layer, LadderLayer):
            return self.input_layer.z_corrupted
        else:
            return self.input_layer.activation_predict + tf.random_normal(tf.shape(self.input_layer.activation_predict),
                                                                          stddev=self.NOISE_STD)

    def unsupervised_cost_train(self):
        mean = tf.reduce_mean(tf.reduce_sum(tf.square(self.z_est_bn - unlabeled(self.input_z_clean,
self.first_layer.labeled_input_size)), 1))
        # TODO: input_nodes may change...
        return (mean / self.input_nodes) * self._denoising_cost

    def _g_gauss(self, z_c, u):
        """gaussian denoising function proposed in the original paper"""
        a1 = self._create_variable('a1', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a2 = self._create_variable('a2', (BaseLayer.INPUT_BOUND_VALUE,), tf.ones([self.input_nodes]))
        a3 = self._create_variable('a3', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a4 = self._create_variable('a4', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a5 = self._create_variable('a5', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a6 = self._create_variable('a6', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a7 = self._create_variable('a7', (BaseLayer.INPUT_BOUND_VALUE,), tf.ones([self.input_nodes]))
        a8 = self._create_variable('a8', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a9 = self._create_variable('a9', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a10 = self._create_variable('a10', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))

        mu = a1 * tf.sigmoid(a2 * u + a3) + a4 * u + a5
        v = a6 * tf.sigmoid(a7 * u + a8) + a9 * u + a10

        z_est = (z_c - mu) * v + mu

        return z_est

    def _activation_method(self, z):
```

```
        with self.name_scope():
            return self._non_liniarity(z + self._beta, name="activation")


class LadderGammaLayer(LadderLayer):
    def __init__(self, input_layer, output_nodes,
                 denoising_cost,
                 session=None,
                 beta=None,
                 gamma=None,
                 weights=None,
                 back_weights=None,
                 freeze=False,
                 non_liniarity=tf.nn.softmax,
                 weight_extender_func=noise_weight_extender,
                 name="ladder_gamma_layer"):
        super(LadderGammaLayer, self).__init__(input_layer, output_nodes,
                                               denoising_cost,
                                               session=session,
                                               beta=beta,
                                               weights=weights,
                                               back_weights=back_weights,
                                               freeze=freeze,
                                               non_liniarity=non_liniarity,
                                               weight_extender_func=weight_extender_func,
                                               name=name)
        self._gamma = self._create_variable("gamma",
                                            (BaseLayer.OUTPUT_BOUND_VALUE,),
                                            gamma if gamma is not None else tf.ones([self.output_nodes]))
        """values for generating std dev of output"""

    def _activation_method(self, z):
        with self.name_scope():
            return self._non_liniarity(self._gamma * (z + self._beta), name="activation_gamma")
```

tensordynamic/tensor_dynamic/layers/**ladder_output_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.ladder_layer import LadderLayer, unlabeled, labeled
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.weight_functions import noise_weight_extender


class LadderOutputLayer(BaseLayer):
  def __init__(self, input_layer,
         denoising_cost,
         session=None,
         freeze=False,
         weight_extender_func=noise_weight_extender,
         name="ladder_output"):
    super(LadderOutputLayer, self).__init__(input_layer,
                        input_layer.output_nodes,
                        session=session,
                        freeze=freeze,
                        weight_extender_func=weight_extender_func,
                        name=name)
    self._denoising_cost = denoising_cost

  @property
  def activation_predict(self):
    return labeled(self.input_layer.activation_predict)

  @property
  def activation_train(self):
    return labeled(self.input_layer.activation_predict)

  @property
  def bactivation(self):
    print "Layer ", self.layer_number, ": ", None, " -> ", self.input_nodes, ", denoising cost: ",
self._denoising_cost
```

```
        return self.z_est_bn

    @lazyprop
    def z_est(self):
        print "Layer ", self.layer_number, ": ", self.output_nodes, " -> ", self.input_nodes, ", denoising cost: ",
self._denoising_cost
        u = unlabeled(self.input_layer.activation_train)
        u = LadderLayer.batch_normalization(u, self.input_layer.mean_clean_unlabeled,
self.input_layer.variance_clean_unlabeled)
        return self._g_gauss(unlabeled(self.input_layer.z_corrupted), u)

    @lazyprop
    def z_est_bn(self):
        return (self.z_est - self.input_layer.mean_clean_unlabeled) / self.input_layer.variance_clean_unlabeled

    def unsupervised_cost_train(self):
        cost = tf.reduce_mean(tf.reduce_sum(tf.square(self.z_est_bn - unlabeled(self.input_layer.z_clean)), 1))
        # TODO: input_nodes may change...
        return (cost / self.input_nodes) * self._denoising_cost

    def supervised_cost_train(self, targets):
        # todo may have to do something more around the labelled vs unlabelled data

        labeled_activations_corrupted = labeled(self.input_layer.activation_train) #tf.slice(self.activation, [0, 0],
tf.shape(targets))
        return -tf.reduce_mean(tf.reduce_sum(targets * tf.log(labeled_activations_corrupted), 1))

    # def train(self, unlabeled_input, labeled_input, labeled_targets):
    #
    # def prediction(self):
    #

    def _g_gauss(self, z_c, u):
        """gaussian denoising function proposed in the original paper"""
        a1 = self._create_variable('a1', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a2 = self._create_variable('a2', (BaseLayer.INPUT_BOUND_VALUE,), tf.ones([self.input_nodes]))
        a3 = self._create_variable('a3', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a4 = self._create_variable('a4', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a5 = self._create_variable('a5', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a6 = self._create_variable('a6', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a7 = self._create_variable('a7', (BaseLayer.INPUT_BOUND_VALUE,), tf.ones([self.input_nodes]))
        a8 = self._create_variable('a8', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a9 = self._create_variable('a9', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))
        a10 = self._create_variable('a10', (BaseLayer.INPUT_BOUND_VALUE,), tf.zeros([self.input_nodes]))

        mu = a1 * tf.sigmoid(a2 * u + a3) + a4 * u + a5
        v = a6 * tf.sigmoid(a7 * u + a8) + a9 * u + a10

        z_est = (z_c - mu) * v + mu

        return z_est
```

tensordynamic/tensor_dynamic/layers/**max_pool_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.lazyprop import clear_all_lazyprops
import math


class MaxPoolLayer(BaseLayer):
    def __init__(self,
                 input_layer,
                 ksize=(2, 2, 1),
                 strides=(2, 2, 1),
                 padding="SAME",
                 layer_noise_std=None,
                 session=None,
                 name='MaxPoolLayer'):
        assert len(input_layer.output_nodes) == 3, "expected 3 output dimensions"
        assert len(ksize) == 3, "expected 3 ksize dimensions"
        assert len(strides) == 3, "expected 3 strides dimensions"

        output_nodes = self._calculate_output_nodes(input_layer, strides)

        super(MaxPoolLayer, self).__init__(input_layer,
                                           output_nodes,
                                           layer_noise_std=layer_noise_std,
                                           session=session,
                                           name=name)

        self._strides = strides
        self._ksize = ksize
        self._padding = padding

    @property
    def regularizable_variables(self):
        return
        yield None

    @property
    def resizable_variables(self):
        return
        yield None

    @staticmethod
    def _calculate_output_nodes(input_layer, strides):
        return (int(math.ceil(input_layer.output_nodes[0] / float(strides[0]))),
                int(math.ceil(input_layer.output_nodes[1] / float(strides[1]))),
                int(math.ceil(input_layer.output_nodes[2] / float(strides[2]))))

    def _layer_activation(self, input_tensor, is_train):
        return tf.nn.max_pool(input_tensor, ksize=(1,) + self._strides,
                              strides=(1,) + self._ksize,
                              padding=self._padding)

    def resize(self, new_output_nodes=None,
               output_nodes_to_prune=None,
```

```
            input_nodes_to_prune=None,
            split_output_nodes=None,
            split_input_nodes=None, split_nodes_noise_std=.1):
        output_nodes = self._calculate_output_nodes(self.input_layer, self._strides)

        if self.output_nodes != output_nodes:
            self._output_nodes = output_nodes

            clear_all_lazyprops(self)
            for layer in self.downstream_layers:
                clear_all_lazyprops(layer)

            if self.next_layer is not None and self.next_layer._resize_needed():
                # TODO: D.S make sure resize is consistant, i.e new nodes are not just created on the end...
                # Must do this at some point
                self._next_layer.resize(input_nodes_to_prune=output_nodes_to_prune,
                                        split_input_nodes=split_output_nodes)

    def clone(self, session=None):
        """Produce a clone of this layer AND all connected upstream layers

        Args:
            session (tensorflow.Session): If passed in the clone will be created with all variables initialised in this
session
                                          If None then the current session of this layer is used

        Returns:
            tensorflow_dynamic.BaseLayer: A copy of this layer and all upstream layers
        """
        new_self = self.__class__(self.input_layer.clone(session or self.session),
                                  session=session or self._session,
                                  name=self._name)

        return new_self

    @property
    def kwargs(self):
        kwargs = super(MaxPoolLayer, self).kwargs

        kwargs['strides'] = self._strides
        kwargs['padding'] = self._padding
        kwargs['ksize'] = self._ksize

        return kwargs
```

tensordynamic/tensor_dynamic/layers/**output_layer.py**

```
import logging
import math
import random

import tensorflow as tf

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.utils import get_tf_optimizer_variables, train_till_convergence

logger = logging.getLogger(__name__)


def bayesian_model_comparison_evaluation(model, data_set):
    """Use bayesian model comparison to evaluate a trained model

    Args:
        model (OutputLayer): Trained model to evaluate
        data_set (DataSet): data set this model was trained on, tends to be test set, but can be train if set up so

    Returns:
        float : log_probability_og_model_generating_data - log(number_of_parameters)
    """
    log_prob, _, _ = model.last_layer.evaluation_stats(data_set)
    param = model.get_parameters_all_layers()
```

```
    score = log_prob - math.log(param)
    print (model.get_resizable_dimension_size(), score, log_prob, param)
    return score


class OutputLayer(HiddenLayer):
    def __init__(self, input_layer, output_nodes,
                 session=None,
                 bias=None,
                 weights=None,
                 back_bias=None,
                 freeze=False,
                 non_liniarity=None,
                 bactivate=False,
                 bactivation_loss_func=None,
                 weight_extender_func=None,
                 layer_noise_std=None,
                 drop_out_prob=None,
                 batch_normalize_input=None,
                 batch_norm_transform=None,
                 batch_norm_scale=None,
                 regularizer_weighting=0.01,
                 regularizer_op=tf.nn.l2_loss,
                 save_checkpoints=0,
                 name='OutputLayer'):
        super(OutputLayer, self).__init__(input_layer, output_nodes,
                                          session=session,
                                          bias=bias,
                                          weights=weights,
                                          back_bias=back_bias,
                                          bactivate=bactivate,
                                          freeze=freeze,
                                          non_liniarity=non_liniarity,
                                          weight_extender_func=weight_extender_func,
                                          bactivation_loss_func=bactivation_loss_func,
                                          layer_noise_std=layer_noise_std,
                                          drop_out_prob=drop_out_prob,
                                          batch_normalize_input=batch_normalize_input,
                                          batch_norm_transform=batch_norm_transform,
                                          batch_norm_scale=batch_norm_scale,
                                          name=name)
        self._regularizer_weighting = regularizer_weighting
        self._regularizer_op = regularizer_op
        self._save_checkpoints = save_checkpoints

        with self.name_scope():
            self._target_placeholder = tf.placeholder('float', shape=(None,) + self.output_nodes, name='target')

    @property
    def target_placeholder(self):
        return self._target_placeholder

    def _target_loss_op(self, input_tensor):
        return tf.reduce_mean(tf.reduce_sum(tf.square(input_tensor - self._target_placeholder), 1))

    @lazyprop
    def target_loss_op_train(self):
        with self.name_scope(is_train=True):
            return self._target_loss_op(self.activation_train)
```

```
@lazyprop
def target_loss_op_predict(self):
    with self.name_scope(is_predict=True):
        return self._target_loss_op(self.activation_predict)

@lazyprop
def loss_op_train(self):
    if self._regularizer_weighting > 0.:
        return self.target_loss_op_train * (1. - self._regularizer_weighting) + \
                self.regularizer_loss_op * self._regularizer_weighting
    else:
        return self.target_loss_op_train

@lazyprop
def loss_op_predict(self):
    if self._regularizer_weighting > 0.:
        return self.target_loss_op_predict * (1. - self._regularizer_weighting) + \
                self.regularizer_loss_op * self._regularizer_weighting
    else:
        return self.target_loss_op_train

@lazyprop
def regularizer_loss_op(self):
    with self.name_scope():
        weights_squared = [self._regularizer_op(variable) for variable in self.regularizable_variables_all_layers]

        # TODO improve
        chain_weights_squared = weights_squared[0]
        for x in weights_squared[1:]:
            chain_weights_squared = chain_weights_squared + x

        return tf.reduce_mean(chain_weights_squared)

@lazyprop
def accuracy_op(self):
    return self.target_loss_op_predict  # TODO accuracy doesn't make sense here...

def resize(self, **kwargs):
    assert kwargs.get('new_output_nodes') is None, "Can't change output nodes for Output layer"
    assert kwargs.get('split_output_nodes') is None, "Can't change output nodes for Output layer"
    super(OutputLayer, self).resize(**kwargs)

def has_resizable_dimension(self):
    return False

def get_resizable_dimension_size(self):
    return None

def train_till_convergence(self, data_set_train, data_set_validation=None, mini_batch_size=100,
                           continue_epochs=2, learning_rate=0.0001,
                           optimizer=tf.train.AdamOptimizer,
                           on_iteration_complete_func=None,
                           on_mini_batch_complete_func=None):
    """Train this network until we stopping seeing an improvement in the error of the validation set

    Args:
        optimizer (tf.train.Optimizer): Type of optimizer to use, e.g. Adam or RMSProp
        learning_rate (float): Learning rate to be used in adam optimizer
        continue_epochs (int): Number of epochs without improvement to go before stopping
        mini_batch_size (int): Number of items per mini-batch
```

```
            data_set_train (tensor_dynamic.data.data_set.DataSet): Used for training
            data_set_validation (tensor_dynamic.data.data_set.DataSet): If passed used for checking error rate
each
                iteration

        Returns:
            float: Error/Accuracy we finally converged on
        """
        assert isinstance(data_set_train, DataSet)
        if data_set_validation is not None:
            assert isinstance(data_set_validation, DataSet)

        optimizer_instance = optimizer(learning_rate,
                                       name="prop_for_%s" % (str(self.get_resizable_dimension_size_all_layers())
                                                              .replace('(', '_').replace(')', '_')
                                                              .replace('[', '_').replace(']', '_')
                                                              .replace(',', '_').replace(' ', '_'),))
        train_op = optimizer_instance.minimize(self.loss_op_train)

        self._session.run(tf.variables_initializer(list(get_tf_optimizer_variables(optimizer_instance))))
        print(optimizer_instance._name)

        iterations = [0]

        validation_size = data_set_validation.num_examples if data_set_validation is not None else
data_set_train.num_examples

        validation_part_size = validation_size / int(math.ceil(validation_size / 1000.))

        def train():
            iterations[0] += 1
            train_error = 0.
            test_error = None

            for features, labels in data_set_train.one_iteration_in_batches(mini_batch_size):
                _, batch_error = self._session.run([train_op, self.loss_op_train],
                                                   feed_dict={self.input_placeholder: features,
                                                              self.target_placeholder: labels})

                if on_mini_batch_complete_func is not None:
                    on_mini_batch_complete_func(self, iterations[0], batch_error)

                train_error += batch_error

            if data_set_validation is not None and data_set_validation is not data_set_train:
                # we may have to break this into equal parts
                test_error, acc = 0., 0.
                parts = 0
                for features, labels in data_set_validation.one_iteration_in_batches(validation_part_size):
                    parts += 1
                    batch_error, batch_acc = self._session.run([self.loss_op_predict, self.accuracy_op],
                                                               feed_dict={
                                                                   self.input_placeholder: features,
                                                                   self.target_placeholder: labels})
                    test_error += batch_error
                    acc += batch_acc

                test_error /= parts
                print(train_error, test_error, acc / parts)
```

```
            if on_iteration_complete_func is not None:
                on_iteration_complete_func(self, iterations[0], train_error=train_error, test_error=test_error)

            return test_error or train_error

        error = train_till_convergence(train, log=False, continue_epochs=continue_epochs)

        logger.info("iterations = %s error = %s", iterations[0], error)

        return error, iterations[0]

    def evaluation_stats(self, dataset):
        """Returns stats related to run

        Args:
            dataset (DataSet):

        Returns:
            (float, float, float): log_probability of the targets given the data, accuracy, target_loss
        """
        log_prob, accuracy, target_loss = self.session.run([self.log_probability_of_targets_op,
                                                            self.accuracy_op,
                                                            self.target_loss_op_predict],
                                                           feed_dict={self.input_placeholder: dataset.features,
                                                                      self._target_placeholder: dataset.labels})

        return log_prob, accuracy, target_loss

    @property
    def kwargs(self):
        kwargs = super(OutputLayer, self).kwargs

        kwargs['regularizer_weighting'] = self._regularizer_weighting
        kwargs['regularizer_op'] = self._regularizer_op
        kwargs['save_checkpoints'] = self._save_checkpoints

        return kwargs

    def learn_structure_layer_by_layer(self, data_set_train, data_set_validation, start_learn_rate=0.001,
                                       continue_learn_rate=0.0001,
                                       model_evaluation_function=bayesian_model_comparison_evaluation,
                                       add_layers=False,
                                       save_checkpoint_path=None,
                                       grow_only=False):
        self.train_till_convergence(data_set_train, data_set_validation, learning_rate=start_learn_rate)
        best_score = model_evaluation_function(self, data_set_validation)

        if save_checkpoint_path:
            self.save_checkpoints(save_checkpoint_path)

        while True:
            best_score = self._best_sizes_for_current_layer_number(best_score, continue_learn_rate,
data_set_train,
                                                                   data_set_validation, model_evaluation_function,
                                                                   save_checkpoint_path,
                                                                   grow_only=grow_only,
                                                                   prune_only=False)

            if add_layers:
                state = self.get_network_state()
```

```
                self.input_layer.add_intermediate_cloned_layer()
                self.last_layer.train_till_convergence(data_set_train, data_set_validation,
                                                       learning_rate=continue_learn_rate)
                result = model_evaluation_function(self, data_set_validation)
                if result > best_score:
                    best_score = result

                    if save_checkpoint_path:
                        self.save_checkpoints(save_checkpoint_path)
                else:
                    # adding a layer didn't help, so reset
                    self.set_network_state(state)
                    return
            else:
                return

    def learn_structure_layer_by_layer_grow_vs_prune(self, data_set_train, data_set_validation,
start_learn_rate=0.001,
                                                     continue_learn_rate=0.0001,
                                                     model_evaluation_function=bayesian_model_comparison_evaluation,
                                                     save_checkpoint_path=None):
        # grow phase
        self.learn_structure_layer_by_layer(data_set_train,
                                            data_set_train,
                                            start_learn_rate=start_learn_rate,
                                            continue_learn_rate=continue_learn_rate,
                                            model_evaluation_function=model_evaluation_function,
                                            add_layers=True,
                                            save_checkpoint_path=save_checkpoint_path,
                                            grow_only=True)

        # prune phase
        best_score = model_evaluation_function(self, data_set_validation)

        if save_checkpoint_path:
            self.save_checkpoints(save_checkpoint_path)

        # while True:
        best_score = self._best_sizes_for_current_layer_number(best_score, continue_learn_rate, data_set_train,
                                                               data_set_validation, model_evaluation_function,
                                                               save_checkpoint_path, prune_only=True)

    def _best_sizes_for_current_layer_number(self, best_score, continue_learn_rate, data_set_train,
data_set_validation,
                                             model_evaluation_function,
                                             save_checkpoint_path,
                                             grow_only=False,
                                             prune_only=False):
        if grow_only and prune_only:
            raise Exception()

        layers = list(self.get_all_resizable_layers())
        index = 0
        attempts_with_out_resize = 0
        while attempts_with_out_resize < len(layers):
            resized, best_score = layers[index].find_best_size(data_set_train, data_set_validation,
                                                               model_evaluation_function=model_evaluation_function,
                                                               best_score=best_score,
                                                               tuning_learning_rate=continue_learn_rate,
                                                               grow_only=grow_only, prune_only=prune_only)
```

```
            if resized:
                attempts_with_out_resize = 1
                if save_checkpoint_path:
                    self.save_checkpoints(save_checkpoint_path)
            else:
                attempts_with_out_resize += 1
            index += 1
            if index == len(layers):
                index = 0
        return best_score

    def save_checkpoints(self, checkpoint_path):
        self._save_checkpoints += 1
        with open(checkpoint_path + "_" + str(self._save_checkpoints) + ".tdc", "w") as f:
            pkl = self.get_network_pickle()
            f.write(pkl)

    def learn_structure_random(self, data_set_train, data_set_validate, start_learn_rate=0.01,
                               continue_learn_rate=0.0001,
                               evaluation_method=bayesian_model_comparison_evaluation,
                               save_checkpoint_path=None):
        rejected_changes = 0
        self.train_till_convergence(data_set_train, data_set_validate, learning_rate=start_learn_rate)

        if save_checkpoint_path:
            self.save_checkpoints(save_checkpoint_path)

        number_of_convergences = 1

        best_model_weight = evaluation_method(self, data_set_validate)
        last_change_was_success = False
        layer_to_resize = None
        node_change = None

        # make random change
        while rejected_changes <= 4:
            network_start_state = self.get_network_state()

            if not last_change_was_success:
                # Only make a new random choice if the last choice was a failure, and if so choose a different layer
                layer_to_resize = random.choice(
                    list(x for x in self.get_all_resizable_layers() if x != layer_to_resize))
                node_change = random.choice([self.GROWTH_MULTIPLYER, self.SHRINK_MULTIPLYER])

            start_size = layer_to_resize.get_resizable_dimension_size()
            new_node_count = layer_to_resize._get_new_node_count(node_change)
            layer_to_resize.resize(new_node_count)

            self.train_till_convergence(data_set_train, data_set_validate, learning_rate=continue_learn_rate)

            number_of_convergences += 1

            # did it work?
            new_model_weight = evaluation_method(self, data_set_validate)

            if new_model_weight <= best_model_weight:
                rejected_changes += 1
                print("REJECTED change of layer %s" % (layer_to_resize.layer_number,))
                print("from size:%s param:%s score:%s" % (start_size, best_param,
                                                          best_model_weight))
```

```
        print("To size:%s param:%s score:%s score change" % (new_node_count,
                                                             self.get_parameters_all_layers(),
                                                             new_model_weight))
        self.set_network_state(network_start_state)
        last_change_was_success = False
    else:
        rejected_changes = 0
        print("ACCEPTED change of layer %s" % (layer_to_resize.layer_number,))
        print("from size:%s param:%s score:%s" % (start_size, best_param,
                                                  best_model_weight))
        print("To size:%s param:%s score:%s score change" % (new_node_count,
                                                             self.get_parameters_all_layers(),
                                                             new_model_weight))
        best_param = self.get_parameters_all_layers()
        best_model_weight = new_model_weight

        last_change_was_success = True

        if save_checkpoint_path:
            self.save_checkpoints(save_checkpoint_path)
```

```
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.lazyprop import lazyprop
from tensor_dynamic.utils import xavier_init
from tensor_dynamic.weight_functions import noise_weight_extender


class VariationalAutoencoderLayer(BaseLayer):
  def __init__(self, input_layer, output_nodes,
               hidden_recog_nodes_1,
               hidden_recog_nodes_2,
               hidden_generation_nodes_1,
               hidden_generation_nodes_2,
               session=None,
               hidden_recog_weights_1=None,
               hidden_recog_weights_2=None,
               hidden_recog_bias_1=None,
               hidden_recog_bias_2=None,
               hidden_generation_weights_1=None,
               hidden_generation_weights_2=None,
               hidden_generation_bias_1=None,
               hidden_generation_bias_2=None,
               output_mean_weights=None,
               output_mean_bias=None,
               output_var_weights=None,
               output_var_bias=None,
               reconstruction_mean_weights=None,
               reconstruction_mean_bias=None,
               freeze=False,
               non_liniarity=tf.nn.softplus,
               weight_extender_func=noise_weight_extender,
               unsupervised_cost=1.,
               supervised_cost=1.,
               name='VariationalAutoencoderLayer'):
    super(VariationalAutoencoderLayer, self).__init__(input_layer, output_nodes,
                                                      session=session,
                                                      freeze=freeze,
                                                      weight_extender_func=weight_extender_func,
                                                      name=name)
    self._unsupervised_cost = unsupervised_cost
    self._non_linarity = non_liniarity
    self._hidden_recog_nodes_1 = hidden_recog_nodes_1
    self._hidden_recog_nodes_2 = hidden_recog_nodes_2
    self._hidden_generation_nodes_1 = hidden_generation_nodes_1
    self._hidden_generation_nodes_2 = hidden_generation_nodes_2

    self._hidden_recog_weights_1 = self._create_variable("hidden_recog_weights_1",
                                                         (BaseLayer.INPUT_BOUND_VALUE, self._hidden_recog_nodes_1),
                                                         hidden_recog_weights_1 if hidden_recog_weights_1 is not None else
xavier_init(
                                                             self._input_nodes,
                                                             self._hidden_recog_nodes_1))
    self._hidden_recog_bias_1 = self._create_variable("hidden_recog_bias_1",
                                                      (self._hidden_recog_nodes_1,),
                                                      hidden_recog_bias_1 if hidden_recog_bias_1 is not None else tf.zeros(
                                                          (self._hidden_recog_nodes_1,)))
    self._hidden_recog_weights_2 = self._create_variable("hidden_recog_weights_2",
```

```
                                                         (self._hidden_recog_nodes_1, self._hidden_recog_nodes_2),
                                                         hidden_recog_weights_2 if hidden_recog_weights_2 is not None else
xavier_init(
                                                             self._hidden_recog_nodes_1,
                                                             self._hidden_recog_nodes_2))
        self._hidden_recog_bias_2 = self._create_variable("hidden_recog_bias_2",
                                                      (self._hidden_recog_nodes_2,),
                                                      hidden_recog_bias_2 if hidden_recog_bias_2 is not None else tf.zeros(
                                                          (self._hidden_recog_nodes_2,)))
        self._hidden_generation_weights_1 = self._create_variable("hidden_generation_weights_1",
                                                              (BaseLayer.OUTPUT_BOUND_VALUE,
                                                               self._hidden_generation_nodes_1),
                                                              hidden_generation_weights_1 if hidden_generation_weights_1 is not
None else xavier_init(
                                                                  self._output_nodes,
                                                                  self._hidden_generation_nodes_1))
        self._hidden_generation_bias_1 = self._create_variable("hidden_generation_bias_1",
                                                           (self._hidden_generation_nodes_1,),
                                                           hidden_generation_bias_1 if hidden_generation_bias_1 is not None
else tf.zeros(
                                                               (self._hidden_generation_nodes_1,)))
        self._hidden_generation_weights_2 = self._create_variable("hidden_generation_weights_2",
                                                              (self._hidden_generation_nodes_1,
                                                               self._hidden_generation_nodes_2),
                                                              hidden_generation_weights_2 if hidden_generation_weights_2 is not
None else xavier_init(
                                                                  self._hidden_generation_nodes_1,
                                                                  self._hidden_generation_nodes_2))
        self._hidden_generation_bias_2 = self._create_variable("hidden_generation_bias_2",
                                                           (self._hidden_generation_nodes_2,),
                                                           hidden_generation_bias_2 if hidden_generation_bias_2 is not None
else tf.zeros(
                                                               (self._hidden_generation_nodes_2,)))
        self._output_mean_weights = self._create_variable("output_mean_weights",
                                                      (self._hidden_recog_nodes_2, BaseLayer.OUTPUT_BOUND_VALUE),
                                                      output_mean_weights if output_mean_weights is not None else xavier_init(
                                                          self._hidden_recog_nodes_2,
                                                          self._output_nodes))
        self._output_mean_bias = self._create_variable("output_mean_bias",
                                                   (BaseLayer.OUTPUT_BOUND_VALUE,),
                                                   output_mean_bias if output_mean_bias is not None else tf.zeros(
                                                       (self._output_nodes,)))
        self._output_var_weights = self._create_variable("output_var_weights",
                                                     (self._hidden_recog_nodes_2, BaseLayer.OUTPUT_BOUND_VALUE),
                                                     output_var_weights if output_var_weights is not None else xavier_init(
                                                         self._hidden_recog_nodes_2,
                                                         self._output_nodes))
        self._output_var_bias = self._create_variable("output_var_bias",
                                                  (BaseLayer.OUTPUT_BOUND_VALUE,),
                                                  output_var_bias if output_var_bias is not None else tf.zeros(
                                                      (self._output_nodes,)))
        self._reconstruction_mean_weights = self._create_variable("reconstruction_mean_weights",
                                                              (
                                                                  self._hidden_generation_nodes_2,
                                                                  BaseLayer.INPUT_BOUND_VALUE),
                                                              reconstruction_mean_weights if reconstruction_mean_weights is not
None else xavier_init(
                                                                  self._hidden_generation_nodes_2,
                                                                  self._input_nodes))
        self._reconstruction_mean_bias = self._create_variable("reconstruction_mean_bias",
```

```
                                    (BaseLayer.INPUT_BOUND_VALUE,),
                                    reconstruction_mean_bias if reconstruction_mean_bias is not None else
tf.zeros(
                                      (self._input_nodes,)))

    self._z_mean_train, self._z_log_sigma_sq_train = self.recognition(self.input_layer.activation_train)
    self._z_mean_predict, self._z_log_sigma_sq_predict = self.recognition(self.input_layer.activation_predict)

    eps = tf.random_normal(tf.shape(self._z_mean_train), 0, 1,
                dtype=tf.float32)
    # z = mu + sigma*epsilon
    self._z_train = tf.add(self._z_mean_train,
                tf.mul(tf.sqrt(tf.exp(self._z_log_sigma_sq_train)), eps))

    self._x_reconstruction_train = self.generator(self._z_train)
    self._x_reconstruction_predict = self.generator(self._z_mean_predict)

  def recognition(self, input):
    layer_1 = self._non_linarity(tf.add(tf.matmul(input, self._hidden_recog_weights_1),
                      self._hidden_recog_bias_1))
    layer_2 = self._non_linarity(tf.add(tf.matmul(layer_1, self._hidden_generation_weights_2),
                      self._hidden_recog_bias_2))
    z_mean = tf.add(tf.matmul(layer_2, self._output_mean_weights),
            self._output_mean_bias)
    z_log_sigma_sq = \
      tf.add(tf.matmul(layer_2, self._output_var_weights),
          self._output_var_bias)
    return z_mean, z_log_sigma_sq

  def generator(self, input):
    layer_1 = self._non_linarity(tf.add(tf.matmul(input, self._hidden_generation_weights_1),
                      self._hidden_generation_bias_1))
    layer_2 = self._non_linarity(tf.add(tf.matmul(layer_1, self._hidden_generation_weights_2),
                      self._hidden_generation_bias_2))
    x_reconstr_mean = \
      tf.nn.sigmoid(tf.add(tf.matmul(layer_2, self._reconstruction_mean_weights),
                self._reconstruction_mean_bias))
    return x_reconstr_mean

  @lazyprop
  def activation_train(self):
    return self._z_train

  @lazyprop
  def activation_predict(self):
    return self._z_mean_predict

  @lazyprop
  def bactivation_train(self):
    return self._x_reconstruction_train

  @lazyprop
  def bactivation_predict(self):
    return self._x_reconstruction_predict

  @lazyprop
  def bactivation_loss_train(self):
    # 1.) The reconstruction loss (the negative log probability
    #    of the input under the reconstructed Bernoulli distribution
    #    induced by the decoder in the data space).
```

```
        #     This can be interpreted as the number of "nats" required
        #     for reconstructing the input when the activation in latent
        #     is given.
        # Adding 1e-10 to avoid evaluatio of log(0.0)
        reconstr_loss = \
            -tf.reduce_sum(self.input_layer.activation_train * tf.log(1e-10 + self.bactivation_train)
                           + (1 - self.input_layer.activation_train) * tf.log(1e-10 + 1 - self.bactivation_train),
                           1)
        # 2.) The latent loss, which is defined as the Kullback Leibler divergence
        #     between the distribution in latent space induced by the encoder on
        #     the data and some prior. This acts as a kind of regularizer.
        #     This can be interpreted as the number of "nats" required
        #     for transmitting the the latent space distribution given
        #     the prior.
        latent_loss = -0.5 * tf.reduce_sum(1 + self._z_log_sigma_sq_train
                                           - tf.square(self._z_mean_train)
                                           - tf.exp(self._z_log_sigma_sq_train), 1)
        return tf.reduce_mean(reconstr_loss + latent_loss)
        #return reconstr_loss + latent_loss

    @lazyprop
    def bactivation_loss_predict(self):
        reconstr_loss = \
            -tf.reduce_sum(self.input_layer.activation_predict * tf.log(1e-10 + self.bactivation_predict)
                           + (1 - self.input_layer.activation_predict) * tf.log(1e-10 + 1 - self.bactivation_predict),
                           1)
        latent_loss = -0.5 * tf.reduce_sum(1 + self._z_log_sigma_sq_predict
                                           - tf.square(self._z_mean_predict)
                                           - tf.exp(self._z_log_sigma_sq_predict), 1)
        return tf.reduce_mean(reconstr_loss + latent_loss)
        #return reconstr_loss + latent_loss

    def unsupervised_cost_train(self):
        return self.bactivation_loss_train * self._unsupervised_cost

    def unsupervised_cost_predict(self):
        return self.bactivation_loss_predict * self._unsupervised_cost

    @property
    def kwargs(self):
        kwargs = super(VariationalAutoencoderLayer, self).kwargs

        return kwargs
```

tensordynamic/tensor_dynamic/data/**cifar_data.py**

```
"""Functions for downloading and reading MNIST data."""
from __future__ import print_function

import cPickle as pickle
import numpy as np
import os

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.data_set_collection import DataSetCollection
from tensor_dynamic.data.mnist_data import dense_to_one_hot

CIFAR_DATA_DIR = os.path.dirname(__file__) + "/CIFAR_data"


def _load_CIFAR_batch(filename):
  """ load single batch of cifar """
  with open(filename, 'r') as f:
    datadict = pickle.load(f)
    X = datadict['data']
    Y = datadict['labels']
    X = X.reshape(10000, 3, 32, 32).transpose(0, 2, 3, 1).astype("float")
    Y = np.array(Y)
    return X, Y


def _load(ROOT):
  """ load all of cifar """
  xs = []
  ys = []
  for b in range(1, 6):
    f = os.path.join(ROOT, 'data_batch_%d' % (b,))
    X, Y = _load_CIFAR_batch(f)
    xs.append(X)
    ys.append(Y)
  Xtr = np.concatenate(xs)
  Ytr = np.concatenate(ys)
  del X, Y
  Xte, Yte = _load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
  return Xtr, Ytr, Xte, Yte


def get_cifar_10_data_set_collection(root_path=CIFAR_DATA_DIR, one_hot=True,
                                     validation_size=0,
                                     validation_ratio=None):
  """Get the cifar 100 data set requires files to be downloaded and extracted into cifar-10-batches-py
  directory within root path

  Args:
    root_path (str):
    one_hot (bool): If True converts sparse labels to one hot encoding
```

```
    Returns:
        DataSetCollection
    """
    root_path += "/cifar-10-batches-py"

    features_train, labels_train, features_test, labels_test = _load(root_path)

    if one_hot:
        labels_train = dense_to_one_hot(labels_train)
        labels_test = dense_to_one_hot(labels_test)

    if not validation_size and validation_ratio:
        validation_size = int((len(labels_train) + len(labels_test)) * validation_ratio)

    if validation_size:
        features_validation = features_train[validation_size:]
        labels_validation = labels_train[validation_size:]

        features_train = features_train[validation_size:]
        labels_train = labels_train[validation_size:]
        validation = DataSet(features_validation, labels_validation, to_binary=True)
    else:
        validation = None

    train = DataSet(features_train, labels_train, to_binary=True)

    test = DataSet(features_test, labels_test, to_binary=True)

    collection = DataSetCollection('CIFAR-10', train, test, validation=validation, normalize=True)

    return collection


def get_cifar_100_data_set_collection(root_path=CIFAR_DATA_DIR, one_hot=True, use_fine_labels=True,
                                      validation_size=0,
                                      validation_ratio=None):
    """Get the cifar 100 data set requires files to be downloaded and extracted into cifar-100-python
    directory within root path

    Args:
        root_path (str):
        one_hot (bool): If True converts sparse labels to one hot encoding
        use_fine_labels (bool): If true use full 100 labels, if False use 10 categories

    Returns:
        DataSetCollection
    """
    root_path = root_path + "/cifar-100-python"

    features_train, labels_train = _load_cifar_100_set(root_path + "/train", use_fine_labels)
    features_test, labels_test = _load_cifar_100_set(root_path + "/test", use_fine_labels)

    if one_hot:
        num_classes = 100 if use_fine_labels else 10
        labels_train = dense_to_one_hot(labels_train, num_classes)
        labels_test = dense_to_one_hot(labels_test, num_classes)

    if not validation_size and validation_ratio:
        validation_size = int((len(labels_train) + len(labels_test)) * validation_ratio)
```

```
    if validation_size:
        features_validation = features_train[:validation_size]
        labels_validation = labels_train[:validation_size]

        features_train = features_train[validation_size:]
        labels_train = labels_train[validation_size:]
        validation = DataSet(features_validation, labels_validation, to_binary=True)
    else:
        validation = None

    train = DataSet(features_train, labels_train, to_binary=True)

    test = DataSet(features_test, labels_test, to_binary=True)

    collection = DataSetCollection('CIFAR-100' + ('-fine' if use_fine_labels else '-coarse'),
                                   train, test, validation=validation, normalize=True)

    return collection


def _load_cifar_100_set(filepath, use_fine_labels):
    with open(filepath, 'rb') as file:
        data = pickle.load(file)

    features = data['data'].reshape(data['data'].shape[0],
                                    3, 32, 32)

    # change from channel, width, height to width, height, channel
    features = features.transpose(0, 2, 3, 1)

    features = features.astype(np.float32)

    if use_fine_labels:
        labels = np.array(data['fine_labels'],
                          dtype=np.uint8)
    else:
        labels = np.array(data['coarse_labels'],
                          dtype=np.uint8)

    return features, labels


# TODO: Fix and maybe use this in the future
def _maybe_download_and_extract(data_dir):
    """Download and extract the tarball from Alex's website."""
    import sys
    import urllib
    import tarfile

    DATA_URL_10 = 'http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz'
    DATA_URL_100 = 'http://www.cs.toronto.edu/~kriz/cifar-100-binary.tar.gz'

    dest_directory = data_dir
    if not os.path.exists(dest_directory):
        os.makedirs(dest_directory)
    filename = DATA_URL_10.split('/')[-1]
    filepath = os.path.join(dest_directory, filename)
    if not os.path.exists(filepath):
        def _progress(count, block_size, total_size):
            sys.stdout.write('\r>> Downloading %s %.1f%%' % (filename,
```

```
                                    float(count * block_size) / float(total_size) * 100.0))
            sys.stdout.flush()

        filepath, _ = urllib.request.urlretrieve(DATA_URL_10, filepath, _progress)
        print()
        statinfo = os.stat(filepath)
        print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
    extracted_dir_path = os.path.join(dest_directory, 'cifar-10-batches-bin')
    if not os.path.exists(extracted_dir_path):
        tarfile.open(filepath, 'r:gz').extractall(dest_directory)


if __name__ == '__main__':
    # data_set = _get_CIFAR10_data("CIFAR_data/cifar-10-batches-py")
    data_set = get_cifar_100_data_set_collection(CIFAR_DATA_DIR, one_hot=True, validation_ratio=.2)
    data_set = get_cifar_10_data_set_collection(CIFAR_DATA_DIR, one_hot=True, validation_ratio=.2)
    print(data_set.name)
```

tensordynamic/tensor_dynamic/data/**data_set.py**

```
import numpy


class DataSet(object):
    def __init__(self, features, labels, fake_data=False,
                 flatten=False,
                 to_binary=False):
        if fake_data:
            self._num_examples = 10000
        else:
            assert features.shape[0] == labels.shape[0], (
                "images.shape: %s labels.shape: %s" % (features.shape,
                                                       labels.shape))
            self._num_examples = features.shape[0]

            # Convert shape from [num examples, rows, columns, depth]
            # to [num examples, rows*columns]
            if flatten:
                assert features.shape[3] == 1
                features = features.reshape(features.shape[0],
                                            features.shape[1] * features.shape[2])

            if to_binary:
                # Convert from [0, 255] -> [0.0, 1.0].
                features = features.astype(numpy.float32)
                features = numpy.multiply(features, 1.0 / 255.0)

        self._features = features
        self._labels = labels
        self._epochs_completed = 0
        self._index_in_epoch = 0
```

```
@property
def features(self):
    """Returns np.Array of features for this dataset, the size of the first dimension should match that of the
    labels property"""
    return self._features

@property
def labels(self):
    """Returns np.Array of labels for this dataset, the size of the first dimension should match that of the
    features property"""
    return self._labels

@property
def num_examples(self):
    """Returns int for number of examples in this dataset"""
    return self._num_examples

@property
def epochs_completed(self):
    """Returns int for the number of epoch of training we have gone through using either the next_batch or
    one_iteration in batches methods"""
    return self._epochs_completed

def next_batch(self, batch_size):
    """Return the next `batch_size` examples from this data set.

    Args:
        batch_size (int):

    Returns:
        tuple of feautres and labels for each batch
    """
    assert batch_size <= self._num_examples

    if self._index_in_epoch == 0 and self._epochs_completed > 0:
        # Shuffle the data
        perm = numpy.arange(self._num_examples)
        numpy.random.shuffle(perm)
        self._features = self._features[perm]
        self._labels = self._labels[perm]

    start = self._index_in_epoch
    self._index_in_epoch += batch_size
    if self._index_in_epoch >= self._num_examples:
        end = None

        # Finished epoch
        self._epochs_completed += 1
        self._index_in_epoch = 0
    else:
        end = self._index_in_epoch

        # we will overrun next run
        if end + batch_size > self._num_examples:
            self._epochs_completed += 1
            self._index_in_epoch = 0

    return self._features[start:end], self._labels[start:end]
```

```
    def one_iteration_in_batches(self, batch_size):
        """ This uses the next_batch method, but in contrast to that method this returns a genertor that will
terminate
        after exactly one epoch of batches.

        Args:
            batch_size (int):

        Returns:
            Generator of tuple of feautres and labels for each batch
        """
        self._index_in_epoch = 0
        starting_epoch = self._epochs_completed

        while starting_epoch == self._epochs_completed:
            yield self.next_batch(batch_size)

    def reset(self):
        """Reset the epoch count and our position in current epoch"""
        self._index_in_epoch = 0
        self._epochs_completed = 0
```

tensordynamic/tensor_dynamic/data/**data_set_collection.py**

```
import numpy as np

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.semi_data_set import SemiDataSet


class DataSetCollection(object):
    def __init__(self, name, train, test, validation=None, normalize=True):
        """Collects data for doing full training and validation of a model

        Args:
            name (str): name for this data set, used for reporting results
            normalize (bool): If True data is normalized, by taking the mean and std of the training set
                and applying it to all other data sets
            train (DataSet): features and labels used for training
            test  (DataSet): features and labels used for testing
            validation (DataSet): optional features and labels used for validation
        """
        assert isinstance(name, str)
        assert isinstance(train, (DataSet, SemiDataSet))
        assert isinstance(test, (DataSet, SemiDataSet))
        assert train.features.shape[1:] == test.features.shape[1:]
        assert train.labels.shape[1:] == test.labels.shape[1:]
        if validation is not None:
            assert isinstance(validation, (DataSet, SemiDataSet))
            assert train.features.shape[1:] == validation.features.shape[1:]
            assert train.labels.shape[1:] == validation.labels.shape[1:]

        if normalize:
            mean_image = np.mean(train.features, axis=0)
            std = np.std(train.features, axis=0)
            std += 1e-10

            train._features -= mean_image
            test._features -= mean_image
            train._features /= std
            test._features /= std

            if validation:
                validation._features -= mean_image
                validation._features /= std

        self._train = train
        self._test = test
        self._validation = validation
        self._name = name
        self._normalized = normalize

    @property
    def normlized(self):
        return self._normalized

    @property
    def train(self):
        return self._train

    @property
    def test(self):
        return self._test
```

```
    @property
    def validation(self):
        return self._validation

    @property
    def name(self):
        return self._name

    @property
    def features_shape(self):
        """Shape of a single instance of features for the dataset, ignores batch dimension

        Returns:
            (int)
        """
        return self._train.features.shape[1:]

    @property
    def labels_shape(self):
        """Shape of a single instance of labels for the dataset, ignores batch dimension

        Returns:
            (int)
        """
        return self._train.labels.shape[1:]
```

tensordynamic/tensor_dynamic/data/**mnist_data.py**

```
"""Functions for downloading and reading MNIST data."""
from __future__ import print_function

import gzip
import os
import urllib

import numpy

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.data_set_collection import DataSetCollection
from tensor_dynamic.data.semi_data_set import SemiDataSet

SOURCE_URL = 'http://yann.lecun.com/exdb/mnist/'


def _maybe_download(filename, work_directory):
    """Download the data from Yann's website, unless it's already here."""
    if not os.path.exists(work_directory):
        os.mkdir(work_directory)
    filepath = os.path.join(work_directory, filename)
```

```
    if not os.path.exists(filepath):
        filepath, _ = urllib.urlretrieve(SOURCE_URL + filename, filepath)
        statinfo = os.stat(filepath)
        print('Succesfully downloaded', filename, statinfo.st_size, 'bytes.')
    return filepath


def _read32(bytestream):
    dt = numpy.dtype(numpy.uint32).newbyteorder('>')
    return numpy.frombuffer(bytestream.read(4), dtype=dt)


def _extract_images(filename):
    """Extract the images into a 4D uint8 numpy array [index, y, x, depth]."""
    print('Extracting', filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2051:
            raise ValueError(
                'Invalid magic number %d in MNIST image file: %s' %
                (magic, filename))
        num_images = _read32(bytestream)
        rows = _read32(bytestream)
        cols = _read32(bytestream)
        buf = bytestream.read(rows * cols * num_images)
        data = numpy.frombuffer(buf, dtype=numpy.uint8)
        data = data.reshape(num_images, rows, cols, 1)
        return data

def dense_to_one_hot(labels_dense, num_classes=10):
    """Convert class labels from scalars to one-hot vectors."""
    num_labels = labels_dense.shape[0]
    index_offset = numpy.arange(num_labels) * num_classes
    labels_one_hot = numpy.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
    return labels_one_hot


def _extract_labels(filename, one_hot=False):
    """Extract the labels into a 1D uint8 numpy array [index]."""
    print('Extracting', filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2049:
            raise ValueError(
                'Invalid magic number %d in MNIST label file: %s' %
                (magic, filename))
        num_items = _read32(bytestream)
        buf = bytestream.read(num_items)
        labels = numpy.frombuffer(buf, dtype=numpy.uint8)
        if one_hot:
            return dense_to_one_hot(labels)
        return labels


def get_mnist_data_set_collection(train_dir=os.path.dirname(__file__) + "/MNIST_data",
                                  number_labeled_examples=None,
                                  one_hot=True,
                                  validation_size=0,
                                  validation_ratio=None,
```

```
                        limit_train_size=None,
                        flatten=True):
    """Load mnist data

    Args:
        train_dir (str): directory to store the downloaded data, or to where it has previously been downloaded
        number_labeled_examples (int): For semi supervised learning, how many labels to use, if None we use
supervised
            learning
        one_hot (bool): If True labels will be one hot vectors, not ints
        validation_size (int): Number of items to move to validation set
        limit_train_size (int): If set limit number of training items to this
        flatten (bool): If true data set is flattened to simply be array of image values, not 3d array of
            [width, height, depth]

    Returns:
        DataSetCollection
    """
    TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'
    TRAIN_LABELS = 'train-labels-idx1-ubyte.gz'
    TEST_IMAGES = 't10k-images-idx3-ubyte.gz'
    TEST_LABELS = 't10k-labels-idx1-ubyte.gz'

    local_file = _maybe_download(TRAIN_IMAGES, train_dir)
    train_images = _extract_images(local_file)

    local_file = _maybe_download(TRAIN_LABELS, train_dir)
    train_labels = _extract_labels(local_file, one_hot=one_hot)

    local_file = _maybe_download(TEST_IMAGES, train_dir)
    test_images = _extract_images(local_file)

    local_file = _maybe_download(TEST_LABELS, train_dir)
    test_labels = _extract_labels(local_file, one_hot=one_hot)

    if not validation_size and validation_ratio:
        validation_size = int((len(train_labels) + len(test_labels)) * validation_ratio)

    validation_images = train_images[:validation_size]
    validation_labels = train_labels[:validation_size]

    train_images = train_images[validation_size:]
    train_labels = train_labels[validation_size:]

    if limit_train_size:
        train_images = train_images[:limit_train_size]
        train_labels = train_labels[:limit_train_size]

    if number_labeled_examples is None:
        train = DataSet(train_images, train_labels, flatten=flatten, to_binary=True)
    else:
        train = SemiDataSet(train_images, train_labels, number_labeled_examples)

    test = DataSet(test_images, test_labels, flatten=flatten, to_binary=True)

    if validation_size:
        validation = DataSet(validation_images, validation_labels, flatten=flatten, to_binary=True)
    else:
        validation = None
```

```
    return DataSetCollection('MNIST', train, test, validation)


if __name__ == '__main__':
    mnist = get_mnist_data_set_collection("MNIST_data", one_hot=True, validation_ratio=.2)
    print(mnist.name)
```

tensordynamic/tensor_dynamic/data/**semi_data_set.py**

```
import numpy

from tensor_dynamic.data.data_set import DataSet


# TODO: Fix this it's broken
class SemiDataSet(object):
    def __init__(self, features, labels, unlabeled_features):
        self.unlabeled_features = unlabeled_features

        # Unlabled DataSet
        self.unlabeled_ds = DataSet(features, labels)

        # Labeled DataSet
        self.num_examples = self.unlabeled_ds.num_examples
        indices = numpy.arange(self.num_examples)
        shuffled_indices = numpy.random.permutation(indices)
        features = features[shuffled_indices]
        labels = labels[shuffled_indices]
        y = numpy.array([numpy.arange(10)[l == 1][0] for l in labels])
        idx = indices[y == 0][:5]
        n_classes = y.max() + 1
        n_from_each_class = unlabeled_features / n_classes
        i_labeled = []
        for c in range(n_classes):
            i = indices[y == c][:n_from_each_class]
            i_labeled += list(i)
        l_images = features[i_labeled]
        l_labels = labels[i_labeled]
        self.labeled_ds = DataSet(l_images, l_labels)

    def next_batch(self, batch_size):
        unlabeled_images, _ = self.unlabeled_ds.next_batch(batch_size)
        if batch_size > self.unlabeled_features:
            labeled_images, labels = self.labeled_ds.next_batch(self.unlabeled_features)
        else:
            labeled_images, labels = self.labeled_ds.next_batch(batch_size)
        images = numpy.vstack([labeled_images, unlabeled_images])
        return images, labels
```

tensordynamic/tensor_dynamic/data/**servo.py**

```
from tensor_dynamic.data_functions import normalize
import numpy as np

INPUT_COLUMNS = ['']


def num_to_one_hot(char):
    if char == 'A':
        return [1.0, 0.0, 0.0, 0.0, 0.0]
    elif char == 'B':
        return [0.0, 1.0, 0.0, 0.0, 0.0]
    elif char == 'C':
        return [0.0, 0.0, 1.0, 0.0, 0.0]
    elif char == 'D':
        return [0.0, 0.0, 0.0, 1.0, 0.0]
    else:
        return [0.0, 0.0, 0.0, 0.0, 1.0]


def get_data(data_dir):
    inputs = []
    outputs = []
    with open(data_dir + 'servo.data') as f:
        for line in f:
            items = line.split(' ')
            input_cols = items[0].split(',')[:-1]
            inputs.append(num_to_one_hot(input_cols[0])+num_to_one_hot(input_cols[1])+
[float(input_cols[2]),float(input_cols[2])])
            outputs.append([float(items[1])])
    return np.array(normalize(inputs), dtype=np.float64), np.array(normalize(outputs), dtype=np.float64)


if __name__ == '__main__':
    x, y = get_data('')
    print x
    print y
```

tensordynamic/tensor_dynamic/data/two_spirals.py

```
import numpy as np

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.data_set_collection import DataSetCollection


def two_spirals(number_of_points, noise=.5):
    """
     Returns the two spirals dataset.

    Args:
```

```
        noise (float):
        number_of_points (int):
    """
    points_per_class = number_of_points / 2
    n = np.sqrt(np.random.rand(points_per_class, 1)) * 780 * (2 * np.pi) / 360
    d1x = -np.cos(n) * n + np.random.rand(points_per_class, 1) * noise
    d1y = np.sin(n) * n + np.random.rand(points_per_class, 1) * noise
    return (np.vstack((np.hstack((d1x, d1y)), np.hstack((-d1x, -d1y)))),
            np.hstack((np.zeros(points_per_class), np.ones(points_per_class))).reshape(number_of_points, 1))


def get_two_spirals_data_set_collection():
    train_features, train_labels = two_spirals(2000)
    test_features, test_labels = two_spirals(1000)
    train = DataSet(train_features, train_labels)
    test = DataSet(test_features, test_labels)

    return DataSetCollection("two spirals", train, test, normalize=False)


if __name__ == '__main__':
    dsc = get_two_spirals_data_set_collection()
    print(dsc.name)
```

tensordynamic/tensor_dynamic/data/xor.py

```
import numpy as np

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.data_set_collection import DataSetCollection


def xor():
    """
     Returns the two spirals dataset.

    Args:
        noise (float):
        number_of_points (int):
    """
    features = np.array([[1., 0.],
                         [0., 0.],
                         [0., 1.],
                         [1., 1.]])

    labels = np.array([[1.],
                       [0.],
```

```
                [0.],
                [1.]])

    return features, labels


def get_xor_data_set_collection():
    features, labels = xor()
    train = DataSet(features, labels)
    test = DataSet(features, labels)

    return DataSetCollection("xor", train, test, normalize=False)


if __name__ == '__main__':
    dsc = get_two_spirals_data_set_collection()
    print(dsc.name)
```

tensordynamic/tests/**base_tf_testcase.py**

```
import logging
import os

import numpy as np
from unittest import TestCase

import sys
import tensorflow as tf

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)


def get_mnist_data(limit_size=None, flatten=True):
    import tensor_dynamic.data.mnist_data as mnist
    import tensor_dynamic.data.data_set as ds
    import os
    return mnist.get_mnist_data_set_collection(os.path.dirname(ds.__file__) +
BaseTfTestCase.MNIST_DATA_DIR, one_hot=True,
                                             flatten=flatten,
                                             limit_train_size=limit_size)

class BaseTfTestCase(TestCase):
    MNIST_DATA = None
    MNIST_INPUT_NODES = 784
    MNIST_OUTPUT_NODES = 10
    MNIST_LIMIT_TEST_DATA_SIZE = 1000
    MNIST_DATA_DIR = "/MNIST_data"

    def setUp(self):
        self.session = tf.Session()
        self.session.__enter__()
```

```
        self.session.as_default().__enter__()

    def tearDown(self):
        self.session.__exit__(None, None, None)

    @property
    def mnist_data(self):
        if self.MNIST_DATA is None:
            self.MNIST_DATA = get_mnist_data(limit_size=self.MNIST_LIMIT_TEST_DATA_SIZE)
        return self.MNIST_DATA

    def data_sum_of_gaussians(self, num_guassians, data_width, data_count):
        gauss_to_data = np.random.uniform(-1., 1., size=(num_guassians, data_width))
        data = []
        for i in range(data_count):
            guassians = np.random.normal(size=num_guassians)
            data_item = np.matmul(guassians, gauss_to_data)
            data.append(data_item)

        return data
```

tensordynamic/tests/**test_bayesian_resizing_net.py**

```
from tensor_dynamic.bayesian_resizing_net import BayesianResizingNet, EDataType, \
    create_flat_network
from tests.base_tf_testcase import BaseTfTestCase


class TestBayesianResizingNet(BaseTfTestCase):
    MNIST_LIMIT_TEST_DATA_SIZE = 3000

    def _create_resizing_net(self, data_set_collection, dimensions):
        outer_net = BayesianResizingNet(create_flat_network(data_set_collection, dimensions, self.session),
                                        model_selection_data_type=EDataType.TRAIN)
        return outer_net

    def test_shrink_from_too_big(self):
        net = self._create_resizing_net(self.mnist_data, (2000, ))
        net.run(self.mnist_data)

        print net._output_layer.get_resizable_dimension_size_all_layers()

        self.assertLess(net._output_layer.get_resizable_dimension_size_all_layers()[0], 2000)

    def test_grow_from_too_small(self):
        # does not always pass
        net = self._create_resizing_net(self.mnist_data, (5, ))
        net.run(self.mnist_data)

        print net._output_layer.get_resizable_dimension_size_all_layers()

        self.assertGreater(net._output_layer.get_resizable_dimension_size_all_layers()[0], 10)

    def test_resizing_net_grow(self):
        dimensions = (20, )
        inner_net = create_flat_network(self.mnist_data, dimensions, self.session)
        inner_net.train_till_convergence(self.mnist_data.train)
        next(iter(inner_net.get_all_resizable_layers())).resize(25)
        inner_net.train_till_convergence(self.mnist_data.train)

    def test_resizing_net_shrink(self):
        dimensions = (20, )
        inner_net = create_flat_network(self.mnist_data, dimensions, self.session)
        inner_net.train_till_convergence(self.mnist_data.train)
        next(iter(inner_net.get_all_resizable_layers())).resize(15)
        inner_net.train_till_convergence(self.mnist_data.train)

    def test_resizing_net_shrink_twice(self):
        dimensions = (20, )
        inner_net = create_flat_network(self.mnist_data, dimensions, self.session)
        inner_net.train_till_convergence(self.mnist_data.train)
        next(iter(inner_net.get_all_resizable_layers())).resize(15)
        inner_net.train_till_convergence(self.mnist_data.train)
        next(iter(inner_net.get_all_resizable_layers())).resize(10)
```

```
        inner_net.train_till_convergence(self.mnist_data.train)

    def test_loss_does_not_decrease_when_returning_to_old_size_from_small(self):
        dimensions = (20, )
        inner_net = create_flat_network(self.mnist_data, dimensions, self.session)
        start_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("start_loss: ", start_loss)

        next(iter(inner_net.get_all_resizable_layers())).resize(5)
        small_size_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("small_size_loss: ", small_size_loss)

        next(iter(inner_net.get_all_resizable_layers())).resize(20)
        return_to_old_size_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("return_to_old_size_loss: ", return_to_old_size_loss)

        self.assertAlmostEqual(start_loss, return_to_old_size_loss, delta=20)

    def test_loss_does_not_decrease_when_returning_to_old_size_from_big(self):
        dimensions = (10, )
        inner_net = create_flat_network(self.mnist_data, dimensions, self.session)
        start_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("start_loss: ", start_loss)

        next(iter(inner_net.get_all_resizable_layers())).resize(50)
        small_size_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("small_size_loss: ", small_size_loss)

        next(iter(inner_net.get_all_resizable_layers())).resize(10)
        return_to_old_size_loss = inner_net.train_till_convergence(self.mnist_data.train, learning_rate=0.01)
        print("return_to_old_size_loss: ", return_to_old_size_loss)

        self.assertAlmostEqual(start_loss, return_to_old_size_loss, delta=20)
```

tensordynamic/tests/**test_lazyprop.py**

```
from unittest import TestCase

from tensor_dynamic.lazyprop import lazyprop, clear_all_lazyprops, subscribe_to_lazy_prop,
unsubscribe_from_lazy_prop, \
    clear_lazyprop_on_lazyprop_cleared, clear_lazyprop


class _PropClass(object):
    STATIC_VAL = None

    @lazyprop
```

```
    def lazyprop(self):
        return self.STATIC_VAL


class TestLazyprop(TestCase):
    def test_clear_all(self):
        prop_class = _PropClass()
        prop_class.STATIC_VAL = 1
        self.assertEquals(prop_class.lazyprop, 1)

        prop_class.STATIC_VAL = 2
        self.assertEquals(prop_class.lazyprop, 1)

        clear_all_lazyprops(prop_class)
        self.assertEquals(prop_class.lazyprop, 2)

    def test_subscribe_lazy_prop_change(self):
        prop_class = _PropClass()
        checker = []
        subscribe_to_lazy_prop(prop_class, 'lazyprop',
                               lambda _: checker.append(1))

        clear_all_lazyprops(prop_class)

        self.assertEqual(checker, [1])

    def test_unsubscribe_lazy_prop_change(self):
        prop_class = _PropClass()
        checker = []
        func = lambda _: checker.append(1)
        subscribe_to_lazy_prop(prop_class, 'lazyprop', func)

        clear_all_lazyprops(prop_class)

        self.assertEqual(len(checker), 1)

        unsubscribe_from_lazy_prop(prop_class, 'lazyprop', func)

        clear_all_lazyprops(prop_class)

        self.assertEqual(len(checker), 1)

    def test_clear_lazyprop_on_lazyprop_cleared(self):
        prop_class_1 = _PropClass()
        prop_class_2 = _PropClass()

        clear_lazyprop_on_lazyprop_cleared(prop_class_2, 'lazyprop',
                                           prop_class_1, 'lazyprop')

        prop_class_1.STATIC_VAL = 1
        prop_class_2.STATIC_VAL = 2

        self.assertEqual(prop_class_1.lazyprop, 1)
        self.assertEqual(prop_class_2.lazyprop, 2)

        prop_class_1.STATIC_VAL = 3
        prop_class_2.STATIC_VAL = 4

        clear_lazyprop(prop_class_1, 'lazyprop')
```

```
        self.assertEqual(prop_class_1.lazyprop, 3)
        self.assertEqual(prop_class_2.lazyprop, 4)
```

tensordynamic/tests/**test_net.py**

```
import pickle
import unittest

import numpy as np
import tensorflow as tf

from tensor_dynamic.layers.base_layer import BaseLayer
from tensor_dynamic.layers.batch_norm_layer import BatchNormLayer
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.base_tf_testcase import BaseTfTestCase


class TestNet(BaseTfTestCase):
    @unittest.skip('Failing because BatchNormLayer is not resizing')
    def test_resize_shallow(self):
        bactivate = True
        net1 = InputLayer(784)
        net2 = HiddenLayer(net1, 10, self.session, bactivate=bactivate)
        bn1 = BatchNormLayer(net2, self.session)
        output_net = HiddenLayer(bn1, 10, self.session, bactivate=False)

        print(self.session.run(output_net.activation_predict, feed_dict={net1.input_placeholder: np.zeros(shape=(1,
784))}))

        net2.resize(net2.output_nodes + 1)

        print(self.session.run(output_net.activation_predict, feed_dict={net1.input_placeholder: np.zeros(shape=(1,
784))}))
```

```
    @unittest.skip('Failing because BatchNormLayer is not resizing')
    def test_resize_deep(self):
        bactivate = True
        net1 = InputLayer(784)
        bn1 = BatchNormLayer(net1, self.session)
        net2 = HiddenLayer(bn1, 8, self.session, bactivate=bactivate)
        bn2 = BatchNormLayer(net2, self.session)
        net2 = HiddenLayer(bn2, 6, self.session, bactivate=bactivate)
        bn3 = BatchNormLayer(net2, self.session)
        net3 = HiddenLayer(bn3, 4, self.session, bactivate=bactivate)
        output_net = HiddenLayer(net3, 2, self.session, bactivate=False)

        print(self.session.run(output_net.activation_predict, feed_dict={net1.input_placeholder: np.zeros(shape=(1,
784))}))

        net2.resize(net2.output_nodes + 1)

        print(self.session.run(output_net.activation_predict, feed_dict={net1.input_placeholder: np.zeros(shape=(1,
784))}))

    def test_layers_with_noise(self):
        input_layer = InputLayer(784)
        bn1 = BatchNormLayer(input_layer, self.session)
        net1 = HiddenLayer(bn1, 70, bactivate=True, layer_noise_std=1.)
        output_net = HiddenLayer(net1, 10, bactivate=False, non_liniarity=tf.identity)

        print(self.session.run(output_net.activation_train, feed_dict={
            input_layer.input_placeholder: np.zeros(shape=(1, 784))}))

    def test_clone(self):
        net1 = InputLayer(784)
        bn1 = BatchNormLayer(net1, self.session)
        net2 = HiddenLayer(bn1, 8, self.session)
        bn2 = BatchNormLayer(net2, self.session)
        net2 = HiddenLayer(bn2, 6, self.session)
        bn3 = BatchNormLayer(net2, self.session)
        net3 = HiddenLayer(bn3, 4, self.session)
        output_net = HiddenLayer(net3, 2, self.session)

        cloned_net = output_net.clone(self.session)

        self.assertNotEquals(cloned_net, output_net)
        self.assertNotEquals(cloned_net.input_layer, output_net.input_layer)
        self.assertEqual(len(list(cloned_net.all_layers)), len(list(output_net.all_layers)))

    def test_accuracy_bug(self):
        import tensor_dynamic.data.mnist_data as mnist
        import tensor_dynamic.data.data_set as ds
        import os

        data = mnist.get_mnist_data_set_collection(os.path.dirname(ds.__file__) + "/MNIST_data", one_hot=True)

        input_layer = InputLayer(data.features_shape)
        outputs = CategoricalOutputLayer(input_layer, data.labels_shape, self.session)

        outputs.train_till_convergence(data.test,
                                       learning_rate=0.2, continue_epochs=1)

        # this was throwing an exception
        accuracy = outputs.accuracy(data.test)
```

```
        self.assertLessEqual(accuracy, 100.)
        self.assertGreaterEqual(accuracy, 0.)

    def test_save_load_network(self):
        net1 = InputLayer(784)
        net2 = HiddenLayer(net1, 20, self.session)
        output_net = CategoricalOutputLayer(net2, 10, self.session)

        data = output_net.get_network_pickle()

        new_net = BaseLayer.load_network_from_pickle(data, self.session)

        print new_net

    def test_save_load_network_to_disk(self):
        net1 = InputLayer(784)
        net2 = HiddenLayer(net1, 20, self.session)
        output_net = CategoricalOutputLayer(net2, 10, self.session)

        data = output_net.get_network_pickle()

        with open("temp", "w") as f:
            f.write(data)

        new_data = pickle.load(open("temp", "r"))

        new_net = BaseLayer.load_network_from_state(new_data, self.session)

        print new_net
```

tensordynamic/tests/**test_tensorflow_features.py**

```
import unittest
```

```
import numpy as np
import tensorflow as tf

from tensor_dynamic.utils import tf_resize_cascading, tf_resize
from tests.base_tf_testcase import BaseTfTestCase


class TestTensorflowFeatures(BaseTfTestCase):
    def test_extend_dim(self):
        var = tf.Variable(tf.zeros((1,)))
        activation = tf.square(var)

        new_value = tf.zeros((2,))
        change_shape_op = tf.assign(var, new_value, validate_shape=False)

        self.session.run(change_shape_op)  # Changes the shape of `var` to new_value's shape.

        new_var = self.session.run(var)
        new_activation_var = self.session.run(activation)
        self.assertSequenceEqual(new_var.shape, (2,))
        self.assertSequenceEqual(new_activation_var.shape, (2,))

    def test_tf_resize_new_values(self):
        var = tf.Variable(range(20))
        self.session.run(tf.initialize_variables([var]))

        tf_resize(self.session, var, new_values=np.array(range(10)))

        self.assertEqual(len(self.session.run(var)), 10)

    @unittest.skip('functionality not implemented yet')
    def test_cascading_resize(self):
        a = tf.Variable(tf.zeros((1, 2)), name="a")
        b = tf.sigmoid(a, name="b")
        matrix = tf.Variable(tf.zeros((2, 4)), name="matrix")
        y = tf.matmul(b, matrix)

        tf_resize_cascading(self.session, a, np.zeros((1, 3)))
        self.assertSequenceEqual(a.get_shape().as_list(), (3,))
        self.assertSequenceEqual(b.get_shape().as_list(), (3,))
        self.assertSequenceEqual(matrix.get_shape().as_list(), (3, 4))
        self.assertSequenceEqual(y.get_shape().as_list(), (4,))

    def test_resize_convolution_convolution_dimension(self):
        input_var = tf.Variable(np.random.normal(0., 1., (1, 4, 4, 1)).astype(np.float32))
        weights = tf.Variable(np.ones((2, 2, 1, 32), dtype=np.float32))
        output = tf.nn.relu(
            tf.nn.conv2d(input_var, weights, strides=[1, 1, 1, 1],
                         padding="SAME"))

        self.session.run(tf.initialize_variables([input_var, weights]))

        result1 = self.session.run(output)

        tf_resize(self.session, weights, new_values=np.ones([2, 2, 1, 16]))

        result2 = self.session.run(output)
        assert result2.shape == (1, 4, 4, 16)

    def test_resize_convolution_intput_layer(self):
```

```
        # resize input layer
        input_var = tf.Variable(np.random.normal(0., 1., (1, 4, 4, 1)).astype(np.float32))
        weights = tf.Variable(np.ones((2, 2, 1, 32), dtype=np.float32))
        output = tf.nn.relu(
            tf.nn.conv2d(input_var, weights, strides=[1, 1, 1, 1],
                         padding="SAME"))

        self.session.run(tf.initialize_variables([input_var, weights]))
        result1 = self.session.run(output)

        tf_resize(self.session, input_var, new_values=np.ones([1, 5, 5, 1]))
        result3 = self.session.run(output)
        print(result3)
        assert result3.shape == (1, 5, 5, 32)

    # COULD NOT FIND A WAY TO MAKE WORK...
    # def test_resize_reshape_func(self):
    #     input_var = tf.Variable(np.random.normal(0., 1., (1, 2, 2, 3)).astype(np.float32))
    #     reshaper = tf.reshape(input_var, (-1, 2*2*3))
    #
    #     self.session.run(tf.initialize_variables([input_var]))
    #
    #     result1 = self.session.run(reshaper)
    #
    #     tf_resize(self.session, input_var, new_dims=(1, 2, 2, 4))
    #     tf_resize(self.session, reshaper, new_dims=(1, 2*2*4))
    #
    #     result2 = self.session.run(reshaper)
    #
    #
    #     print(result1)
    #     print(result2)

    def test_gradient_vs_hessian(self):
        var = tf.placeholder('float', shape=(4))
        reshaped = tf.reshape(var, (2, 2,))
        operation = tf.reduce_sum(tf.pow(reshaped, 3))

        jacob = tf.gradients(operation, reshaped)[0]
        hessian_b = tf.gradients(jacob, reshaped)[0]

        hessian_a = tf.hessians(operation, var)[0]
        hessian_a_diag = tf.diag_part(hessian_a)

        print hessian_a
        print hessian_a_diag
        print hessian_b

        input = np.array([0., 1., 2., 3.])

        print self.session.run(hessian_a, feed_dict={var: input})
        print self.session.run(hessian_a_diag, feed_dict={var: input})
        print self.session.run(hessian_b, feed_dict={var: input})
```

tensordynamic/tests/**test_utils.py**

```
import unittest

import numpy as np

import tensorflow as tf

from tensor_dynamic.utils import train_till_convergence, create_hessian_op, tf_resize, \
    get_tf_optimizer_variables
from tests.base_tf_testcase import BaseTfTestCase


class TestUtils(BaseTfTestCase):
    def test_train_till_convergence(self):
        FINAL_ERROR = 3
        errors = [5, 4, 3, 2, 2, 1, 2, 2, FINAL_ERROR]
        errors_iter = iter(errors)

        final_error = train_till_convergence(lambda: next(errors_iter), continue_epochs=3)

        self.assertEqual(final_error, FINAL_ERROR)

    @unittest.skip('functionality not implemented yet')
    def test_compute_hessian(self):
        # this currently fails because I can't get the method to work, tensorflow does not support gradients after
        # doing a reshape/slice op
        n_input = 3
        n_hidden = 2
        n_output = 1
        x_input = tf.placeholder(tf.float32, shape=[None, n_input])
        y_target = tf.placeholder(tf.float32, shape=[None, n_output])

        hidden_weights = tf.Variable(initial_value=tf.truncated_normal([n_input, n_hidden]))
```

```
        hidden_biases = tf.Variable(tf.truncated_normal([n_hidden]))
        hidden = tf.sigmoid(tf.matmul(x_input, hidden_weights) + hidden_biases)

        output_weights = tf.Variable(initial_value=tf.truncated_normal([n_hidden, n_output]))
        output_biases = tf.Variable(tf.truncated_normal([n_output]))
        output = tf.nn.softmax(tf.matmul(hidden, output_weights) + output_biases)
        # Define cross entropy loss
        loss = -tf.reduce_sum(y_target * tf.log(output))

        self.session.run(tf.initialize_variables([hidden_weights, hidden_biases, output_weights, output_biases]))
        hessian_op = create_hessian_op(loss, [hidden_weights, hidden_biases, output_weights, output_biases],
                                       self.session)
        result = self.session.run(hessian_op, feed_dict={x_input: np.random.normal(size=(1, n_input)),
                                                         y_target: np.random.normal(size=(1, n_output))})

        print(result)

    @unittest.skip('functionality not implemented yet')
    def test_compute_hessian_1_variable(self):
        # this currently fails because I can't get the method to work, tensorflow does not support gradients after
        # doing a reshape/slice op
        n_input = 2
        n_output = 2

        x_input = tf.placeholder(tf.float32, shape=[None, n_input])
        y_target = tf.placeholder(tf.float32, shape=[None, n_output])

        weights = tf.Variable(initial_value=[[-2., -1.], [1., 2.]])

        output = tf.nn.softmax(tf.matmul(x_input, weights))
        # Define cross entropy loss
        loss = -tf.reduce_sum(y_target * tf.log(output))

        self.session.run(tf.initialize_variables([weights]))
        hessian_op = create_hessian_op(loss, [weights], self.session)
        result = self.session.run(hessian_op, feed_dict={x_input: np.ones((1, n_input)),
                                                         y_target: np.ones((1, n_output))})

        # TODO D.S find simple hessian example + numbers

        print(result)

    def test_gradient_through_reshape(self):
        input = tf.Variable(initial_value=tf.zeros([2, 2]))
        after_reshape = tf.reshape(input, [-1])
        target = tf.square(1. - tf.reduce_sum(after_reshape))

        train_op = tf.train.GradientDescentOptimizer(0.5).minimize(target)

        self.session.run(tf.initialize_variables([input]))

        print self.session.run(input)

        self.session.run(train_op)

        print self.session.run(input)

        print self.session.run(tf.gradients(target, input))

    def test_tf_resize_shrink(self):
```

```
        zeros = tf.zeros((6,))
        var = tf.Variable(initial_value=zeros)
        self.session.run(tf.initialize_variables([var]))

        tf_resize(self.session, var, new_dimensions=(4,))

        self.assertEqual(self.session.run(var).shape, (4,))

    def test_tf_resize_shrink_twice(self):
        zeros = tf.zeros((6,))
        var = tf.Variable(initial_value=zeros)
        self.session.run(tf.initialize_variables([var]))

        tf_resize(self.session, var, new_dimensions=(4,))

        tf.train.GradientDescentOptimizer(0.1).minimize(var)

        tf_resize(self.session, var, new_dimensions=(2,))

        self.assertEqual(self.session.run(var).shape, (2,))

        # this was causing an exception
        tf.train.GradientDescentOptimizer(0.1).minimize(var)

    def test_tf_resize_grow(self):
        zeros = tf.zeros((3,))
        var = tf.Variable(initial_value=zeros)
        self.session.run(tf.initialize_variables([var]))

        tf_resize(self.session, var, new_dimensions=(6,))

        self.assertEqual(self.session.run(var).shape, (6,))

    def test_tf_resize(self):
        zeros = tf.zeros((4,))
        var = tf.Variable(initial_value=zeros)
        loss = tf.square(1 - tf.reduce_sum(var))
        self.session.run(tf.initialize_variables([var]))

        optimizer_1 = tf.train.RMSPropOptimizer(0.01)
        train_1 = optimizer_1.minimize(loss)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer_1))))

        self.session.run(train_1)

        tf_resize(self.session, var, new_dimensions=(6,))

        optimizer_2 = tf.train.RMSPropOptimizer(0.01)
        train_2 = optimizer_2.minimize(loss)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer_2))))

        self.session.run(train_2)
```

tensordynamic/tests/**test_weight_functions.py**

```
import numpy as np

from unittest import TestCase
```

```
from tensor_dynamic.weight_functions import array_extend, noise_weight_extender


class TestWeightFunctions(TestCase):
    def test_array_split_extention_axis_1(self):
        a = np.array([[1, 2, 3],
                      [4, 5, 6]])

        split_extended = array_extend(a, {1: [1]})

        np.testing.assert_array_almost_equal(split_extended, np.array([[1, 2, 3, 2], [4, 5, 6, 5]]))

    def test_array_split_extention_axis_2(self):
        a = np.array([[1, 2, 3],
                      [4, 5, 6]])

        split_extended = array_extend(a, {0: [0]})

        np.testing.assert_array_almost_equal(split_extended, np.array([[1, 2, 3],
                                                                       [4, 5, 6],
                                                                       [1, 2, 3]]))

    def test_array_split_extention_axis_3(self):
        a = np.array([[[1, 2], [3, 4]],
                      [[5, 6], [7, 8]]])

        split_extended = array_extend(a, {2: [0]})

        np.testing.assert_array_almost_equal(split_extended, np.array([[[1, 2, 1], [3, 4, 3]],
                                                                       [[5, 6, 5], [7, 8, 7]]]))

    def test_array_split_extention_vector(self):
        a = np.array([1, 2, 3])

        split_extended = array_extend(a, {0: [0]})

        np.testing.assert_array_almost_equal(split_extended, np.array([1, 2, 3, 1]))

    def test_array_split_extention_halve_splits(self):
        a = np.array([[2., 4., 8.],
                      [1., 2., 3.]])

        split_extended = array_extend(a, {0: [0]}, halve_extended_vectors=True)

        np.testing.assert_array_almost_equal(split_extended, np.array([[1., 2., 4.],
                                                                       [1., 2., 3.],
                                                                       [1., 2., 4.]]))

    def test_noise_weight_extender_shrink(self):
        a = np.array([[2., 4., 8.],
                      [1., 2., 3.]])

        b = noise_weight_extender(a, (2, 2))

        self.assertEqual(b.shape, (2, 2))

    def test_noise_weight_extender_4_dim(self):
        a = np.random.normal(size=(5, 4, 3, 2))

        new_dimensions = (5, 4, 3, 3)
```

```
        b = noise_weight_extender(a, new_dimensions)

        self.assertEqual(b.shape, new_dimensions)

    def test_noise_weight_extender_4_dim_2(self):
        a = np.random.normal(size=(5, 4, 3, 3))

        new_dimensions = (5, 4, 3, 1)
        b = noise_weight_extender(a, new_dimensions)

        self.assertEqual(b.shape, new_dimensions)

    def test_noise_weight_extender_4_dim_3(self):
        a = np.random.normal(size=(5, 4, 3, 2))

        new_dimensions = (2, 3, 4, 5)
        b = noise_weight_extender(a, new_dimensions)

        self.assertEqual(b.shape, new_dimensions)

    def test_noise_weight_extender_1_dim(self):
        a = np.random.normal(size=(5,))

        new_dimensions = (10,)
        b = noise_weight_extender(a, new_dimensions)

        self.assertEqual(b.shape, new_dimensions)
```

tensordynamic/tests/data/**test_data_set.py**

```
import numpy as np
from unittest import TestCase

from tensor_dynamic.data.data_set import DataSet
from tensor_dynamic.data.data_set_collection import DataSetCollection


class TestDataSet(TestCase):
    def test_num_examples(self):
        data_set = DataSet(np.random.normal(size=(100, 10)), np.random.normal(size=(100, 1)))

        self.assertEqual(data_set.num_examples, 100)

    def test_one_batch_iteration_exact_batch(self):
        batch_size = 10
        data_set = DataSet(np.random.normal(size=(20, 10)), np.random.normal(size=(20, 1)))

        results = list(data_set.one_iteration_in_batches(batch_size))

        self.assertEqual(len(results), 2)
        self.assertEqual(len(results[0][0]), batch_size)
        self.assertEqual(len(results[0][1]), batch_size)
        self.assertEqual(len(results[-1][0]), batch_size)
        self.assertEqual(len(results[-1][1]), batch_size)
```

```
    def test_one_batch_iteration_exact_partial_batch(self):
        batch_size = 10
        data_set = DataSet(np.random.normal(size=(25, 10)), np.random.normal(size=(25, 1)))

        results = list(data_set.one_iteration_in_batches(batch_size))

        self.assertEqual(len(results), 2)
        self.assertEqual(len(results[0][0]), batch_size)
        self.assertEqual(len(results[0][1]), batch_size)
        self.assertEqual(len(results[-1][0]), batch_size)
        self.assertEqual(len(results[-1][1]), batch_size)
```

tensordynamic/tests/experiments/**grid_search.py**

```
import functools

import tensorflow as tf

from tensor_dynamic.bayesian_resizing_net import create_flat_network
from tensor_dynamic.data.cifar_data import get_cifar_100_data_set_collection
from tests.base_tf_testcase import get_mnist_data


def do_grid_search(data_set_collection, model_functions, file_name, learning_rate=0.001,
                   continue_epochs=4, **extra_parameters):
    """Run a grid search and write all results to csv file

    Args:
        continue_epochs (int):
        data_set_collection (tensor_dynamic.data.data_set_collection.DataSetCollection):
        model_functions: Function that returns an iterator of functions that when given a tensorflow session create
the
            model we want to run for each element in the search
        extra_parameters (dict):
        learning_rate (float):
    """
    extra_parameters['learning_rate'] = learning_rate
    extra_parameters['continue_epochs'] = continue_epochs
    write_parameters_file(data_set_collection, file_name, **extra_parameters)
    with open(file_name, 'w') as result_file:
        result_file.write(
            'log_prob_train, error_train, accuracy_train, error_test, accuracy_test, log_prob_test, dimensions,
parameters\n')

        for model_function in model_functions(data_set_collection):
            tf.reset_default_graph()
            with tf.Session() as session:
                model = model_function(session)
```

```
            model.train_till_convergence(data_set_collection.train, data_set_collection.test,
                                         learning_rate=learning_rate, continue_epochs=continue_epochs,
                                         optimizer=extra_parameters['optimizer'])

            train_log_prob, train_error, train_acc = model.evaluation_stats(data_set_collection.train)

            test_log_prob, test_error, test_acc = model.evaluation_stats(data_set_collection.test)

            result_file.write("%s,%s,%s,%s,%s,%s,%s,%s\n" % (train_log_prob, train_error, train_acc,
                                                             test_log_prob, test_error, test_acc,
                                                             str(model.get_resizable_dimension_size_all_layers())
                                                             .replace(',', '-'),
                                                             model.get_parameters_all_layers()))


def write_parameters_file(data_set_collection, file_name, **kwargs):
    with open(file_name + '.txt', 'w') as param_file:
        param_file.write("data_set=%s\n" % (data_set_collection.name,))
        param_file.write("data_set_normalized=%s\n" % (data_set_collection.normlized,))
        for key, value in kwargs.iteritems():
            param_file.write("%s=%s\n" % (key, value))


def flat_model_functions(data_set_collection, regularizer, activation_func, input_layer_noise_std,
input_noise_std):
    def get_model(session, parameters):
        return create_flat_network(data_set_collection, parameters, session, regularizer_coeff=regularizer,
                                   activation_func=activation_func,
                                   input_layer_noise_std=input_layer_noise_std,
                                   input_noise_std=input_noise_std)
    yield functools.partial(get_model, parameters=(1000, 1000, 1000, 1000, 1000,))
    # 1 layer
    # for layer_1 in [300, 500, 1000]:
    #     yield functools.partial(get_model, parameters=(layer_1,))
    #
    #     for layer_2 in [300, 500, 1000]:
    #         if layer_2 <= layer_1:
    #             yield functools.partial(get_model, parameters=(layer_1, layer_2))
    #
    #             for layer_3 in [300, 500]:
    #
    #                 if layer_3 <= layer_2:
    #                     yield functools.partial(get_model, parameters=(layer_1, layer_2, layer_3))
    #
    #                     for layer_4 in [500]:
    #
    #                         if layer_4 <= layer_4:
    #                             yield functools.partial(get_model, parameters=(layer_1, layer_2, layer_3, layer_4))


if __name__ == '__main__':
    regularizer = 0.0

    data_set_collection = get_cifar_100_data_set_collection()
    input_layer_noise_std = 1.0
    input_noise_std = 1.0
    do_grid_search(data_set_collection,
                   functools.partial(flat_model_functions,
                                     regularizer=regularizer,
                                     activation_func=tf.nn.relu,
```

```
                    input_layer_noise_std=input_layer_noise_std,
                    input_noise_std=input_noise_std),
'cifar-100_noise-all-layer-another.csv',
regularizer=regularizer,
learning_rate=0.00001,
input_layer_noise_std=input_layer_noise_std,
input_noise_std=input_noise_std,
activation_func=tf.nn.relu,
network='flat',
optimizer=tf.train.AdamOptimizer)
```

tensordynamic/tests/experiments/**show_adding_random_nodes_is_bad.py**

```
from tensor_dynamic.data.mnist_data import get_mnist_data_set_collection
import tensorflow as tf
```

```
# set up network
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer

data_set_collection = get_mnist_data_set_collection()

START_SIZE = 30
END_SIZE = 31

with tf.Session() as session:
    non_liniarity = tf.nn.relu

    regularizer_coeff = 0.001
    last_layer = InputLayer(data_set_collection.features_shape,
                            # drop_out_prob=.5,
                            # layer_noise_std=1.
                            )

    for _ in range(1):
        last_layer = HiddenLayer(last_layer, START_SIZE, session, non_liniarity=non_liniarity,
                                 batch_normalize_input=True)

    output = CategoricalOutputLayer(last_layer, data_set_collection.labels_shape, session,
                                    regularizer_weighting=regularizer_coeff,
                                    batch_normalize_input=True)

    # train network to convergence

    output.train_till_convergence(data_set_collection.train, data_set_collection.test, learning_rate=0.0001,
                                  continue_epochs=2)

    # report stats

    train_log_prob, train_acc, train_error = output.evaluation_stats(data_set_collection.train)

    test_log_prob, test_acc, test_error = output.evaluation_stats(data_set_collection.test)

    print("%s,%s,%s,%s,%s,%s,%s,%s\n" % (train_log_prob, train_error, train_acc,
                                         test_log_prob, test_error, test_acc,
                                         str(output.get_resizable_dimension_size_all_layers())
                                         .replace(',', '-'),
                                         output.get_parameters_all_layers()))

    # add x nodes with random values to the trained network

    last_layer.resize(END_SIZE, no_splitting_or_pruning=True)

    # train till convergence

    output.train_till_convergence(data_set_collection.train, data_set_collection.test, learning_rate=0.0001,
                                  continue_epochs=2)

    # report stats

    train_log_prob, train_acc, train_error = output.evaluation_stats(data_set_collection.train)

    test_log_prob, test_acc, test_error = output.evaluation_stats(data_set_collection.test)

    print("%s,%s,%s,%s,%s,%s,%s,%s\n" % (train_log_prob, train_error, train_acc,
```

```
                                   test_log_prob, test_error, test_acc,
                                   str(output.get_resizable_dimension_size_all_layers())
                                   .replace(',', '-'),
                                   output.get_parameters_all_layers()))

    # remove new node, don't train till convergence

    last_layer.resize(START_SIZE, no_splitting_or_pruning=True)

    # report stats

    train_log_prob, train_acc, train_error = output.evaluation_stats(data_set_collection.train)

    test_log_prob, test_acc, test_error = output.evaluation_stats(data_set_collection.test)

    print("%s,%s,%s,%s,%s,%s,%s,%s\n" % (train_log_prob, train_error, train_acc,
                                   test_log_prob, test_error, test_acc,
                                   str(output.get_resizable_dimension_size_all_layers())
                                   .replace(',', '-'),
                                   output.get_parameters_all_layers()))

    # contrast above with
```

ensordynamic/tests/experiments/**single_run.py**

```
import tensorflow as tf

from tensor_dynamic.data.mnist_data import get_mnist_data_set_collection
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer

# data_set_collection = get_cifar_100_data_set_collection()
from tensor_dynamic.node_importance import node_importance_optimal_brain_damage

data_set_collection = get_mnist_data_set_collection(validation_ratio=.15)

with tf.Session() as session:
    non_liniarity = tf.nn.relu

    regularizer_coeff = 0.01
    last_layer = InputLayer(data_set_collection.features_shape,
                            # drop_out_prob=.5,
                            # layer_noise_std=1.
                            )

    # last_layer = FlattenLayer(last_layer, session)

    for _ in range(1):
```

```
    last_layer = HiddenLayer(last_layer, 10, session, non_liniarity=non_liniarity,
                             node_importance_func=node_importance_optimal_brain_damage,
                             batch_normalize_input=True)

  # last_layer = ConvolutionalLayer(last_layer, (5, 5, 32), stride=(1, 1, 1), session=session,
  #                                 non_liniarity=non_liniarity)
  #
  # last_layer = MaxPoolLayer(last_layer, session=session)
  #
  # last_layer = ConvolutionalLayer(last_layer, (5, 5, 64), stride=(1, 1, 1), session=session,
  #                                 non_liniarity=non_liniarity)
  #
  # last_layer = MaxPoolLayer(last_layer, session=session)
  #
  # last_layer = FlattenLayer(last_layer, session=session)
  #
  # last_layer = HiddenLayer(last_layer, 1024, session, non_liniarity=non_liniarity)
  #
  # last_layer = HiddenLayer(last_layer, 512, session, non_liniarity=non_liniarity)
  #
  # last_layer = HiddenLayer(last_layer, 512, session, non_liniarity=non_liniarity)

  output = CategoricalOutputLayer(last_layer, data_set_collection.labels_shape, session,
                                  batch_normalize_input=True,
                                  loss_cross_entropy_or_log_prob=False,
                                  regularizer_weighting=regularizer_coeff)

  # output.train_till_convergence(data_set_collection.train, data_set_collection.test, learning_rate=0.00001,
  #                               continue_epochs=2)

  output.learn_structure_layer_by_layer(data_set_collection.train, data_set_collection.validation,
                                        start_learn_rate=0.0001, continue_learn_rate=0.0001,
                                        add_layers=True)

  train_log_prob, train_acc, train_error = output.evaluation_stats(data_set_collection.train)

  val_log_prob, val_acc, val_error = output.evaluation_stats(data_set_collection.validation)

  test_log_prob, test_acc, test_error = output.evaluation_stats(data_set_collection.test)

  print("%s,%s,%s,%s,%s,%s,%s,%s\n" % (train_log_prob, train_error, train_acc,
                                       test_log_prob, test_error, test_acc,
                                       str(output.get_resizable_dimension_size_all_layers())
                                       .replace(',', '-'),
                                       output.get_parameters_all_layers()))

  # (7508.6528, 0.97310001)
  # INFO:tensor_dynamic.layers.output_layer:iterations = 23 error = 7508.65
```

tensordynamic/tests/experiments/**test_growing_layers.py**

```
import tensorflow as tf

from tensor_dynamic.data.cifar_data import get_cifar_100_data_set_collection
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.flatten_layer import FlattenLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer

data_set_collection = get_cifar_100_data_set_collection(validation_ratio=.15)
ITERATIONS = 10


def print_stats(data_set_collection, model, layer_num):
    train_log_prob, train_acc, train_error = model.evaluation_stats(data_set_collection.train)
    val_log_prob, val_acc, val_error = model.evaluation_stats(data_set_collection.validation)
    test_log_prob, test_acc, test_error = model.evaluation_stats(data_set_collection.test)
    text = "%s,%s,%s%s,%s,%s,%s,%s,%s,%s,%s, %s\n" % (train_log_prob, train_error, train_acc,
                                                val_log_prob, val_error, val_acc,
                                                test_log_prob, test_error, test_acc,
                                                str(model.get_resizable_dimension_size_all_layers())
                                                .replace(',', '-'),
                                                model.get_parameters_all_layers(), layer_num)
    print(text)
    with open('adding_layers.csv', "w") as file_avg:
        file_avg.write(text)


def try_intermediate_layer(layer_num):
    print "add layer at pos " + str(layer_num)
    list(output.all_connected_layers)[layer_num].add_intermediate_layer(
        lambda x: HiddenLayer(x, nodes_per_layer, session,
                              non_liniarity=non_liniarity,
                              batch_normalize_input=True))
    output.train_till_convergence(data_set_collection.train, data_set_collection.validation,
                                  learning_rate=0.0001)
    output.save_checkpoints('cifar-100-layers')
    print_stats(data_set_collection, output, layer_num)
    output.set_network_state(state)


for _ in range(ITERATIONS):
    with tf.Session() as session:
        non_liniarity = tf.nn.relu
        nodes_per_layer = 400
```

```
regularizer_coeff = 0.01
last_layer = InputLayer(data_set_collection.features_shape,
                        # drop_out_prob=.5,
                        # layer_noise_std=1.
                        )

last_layer = FlattenLayer(last_layer, session)

for _ in range(3):
    last_layer = HiddenLayer(last_layer, nodes_per_layer, session, non_liniarity=non_liniarity,
                             batch_normalize_input=True)

output = CategoricalOutputLayer(last_layer, data_set_collection.labels_shape, session,
                                batch_normalize_input=True,
                                loss_cross_entropy_or_log_prob=True,
                                regularizer_weighting=regularizer_coeff)

output.train_till_convergence(data_set_collection.train, data_set_collection.validation,
                              learning_rate=0.0001)

state = output.get_network_state()

output.save_checkpoints('cifar-100-layers')

print_stats(data_set_collection, output, -1)

for i in range(3):
    try_intermediate_layer(4 - i)

    # (7508.6528, 0.97310001)
    # INFO:tensor_dynamic.layers.output_layer:iterations = 23 error = 7508.65
```

tensordynamic/tests/experiments/**test_growing_width_single_flat.py**

```
import tensorflow as tf
from tensor_dynamic.bayesian_resizing_net import create_flat_network
from tests.base_tf_testcase import get_mnist_data
```

```
def main(data_set_collection):
    results = []

    with tf.Session() as session:
        net = create_flat_network((data_set_collection.features_shape[0],
                                   20,
                                   data_set_collection.labels_shape[0]), session)

        error = net.train_till_convergence(data_set_collection.train, data_set_collection.test,
                                           learning_rate=0.001)
        parameters = net.get_parameters_all_layers()
        results.append((net.get_resizable_dimensions()[0], parameters, error))

        while net.get_resizable_dimensions()[0] <= 500:
            net.get_all_resizable_layers()[0].resize(net.get_resizable_dimensions()[0] + 10)
            error = net.train_till_convergence(data_set_collection.train, data_set_collection.test,
                                               learning_rate=0.0001)
            parameters = net.get_parameters_all_layers()
            results.append((net.get_resizable_dimensions()[0], parameters, error))

    print results


if __name__ == '__main__':
    data_set_collection = get_mnist_data()
    main(data_set_collection)
```

tensordynamic/tests/experiments/**two_spirals.py**

```
import tensorflow as tf

from tensor_dynamic.data.cifar_data import get_cifar_100_data_set_collection
from tensor_dynamic.data.two_spirals import get_two_spirals_data_set_collection
from tensor_dynamic.layers.convolutional_layer import ConvolutionalLayer
from tensor_dynamic.layers.flatten_layer import FlattenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.max_pool_layer import MaxPoolLayer
from tensor_dynamic.layers.output_layer import OutputLayer
from tensor_dynamic.layers.binary_output_layer import BinaryOutputLayer
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
```

```
data_set_collection = get_two_spirals_data_set_collection()

with tf.Session() as session:
    non_liniarity = tf.nn.tanh

    regularizer_coeff = 0.001
    last_layer = InputLayer(data_set_collection.features_shape, session)

    last_layer = HiddenLayer(last_layer, 5, session, non_liniarity=non_liniarity)

    last_layer = HiddenLayer(last_layer, 5, session, non_liniarity=non_liniarity)

    last_layer = HiddenLayer(last_layer, 5, session, non_liniarity=non_liniarity)
    #
    # last_layer = Layer(last_layer, 300, session, non_liniarity=non_liniarity)
    #
    # last_layer = Layer(last_layer, 300, session, non_liniarity=non_liniarity)
    #
    # last_layer = Layer(last_layer, 300, session, non_liniarity=non_liniarity)

    output = BinaryOutputLayer(last_layer, session, regularizer_weighting=regularizer_coeff)

    output.train_till_convergence(data_set_collection.train, data_set_collection.test, learning_rate=0.0001,
                                  continue_epochs=3)


    # (7508.6528, 0.97310001)
    # INFO:tensor_dynamic.layers.output_layer:iterations = 23 error = 7508.65
```

tensordynamic/tests/experiments/**weight_pruning_tests.py**

```
from collections import defaultdict

import tensorflow as tf
from tensor_dynamic.data.cifar_data import get_cifar_100_data_set_collection
from tensor_dynamic.data.mnist_data import get_mnist_data_set_collection
from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.flatten_layer import FlattenLayer
from tensor_dynamic.layers.hidden_layer import node_importance_by_square_sum, HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.node_importance import node_importance_by_dummy_activation_from_input_layer,
node_importance_random, \
    node_importance_optimal_brain_damage, node_importance_full_taylor_series, \
    node_importance_by_real_activation_from_input_layer_variance, node_importance_by_removal, \
    node_importance_error_derrivative
from tensor_dynamic.node_importance import node_importance_by_real_activation_from_input_layer

NUM_TRIES = 15


def dummy_random_weights():
    raise Exception()

start = 400
end = 380
```

```
def main(file_name_all="pruning_tests%s-%s-%s.csv" % ('_noise=.5', start, end),
file_name_avg="pruning_tests%s-%s-%s.csv" % ('_noise=.5', start, end)):
    data_set_collections = [get_mnist_data_set_collection(validation_ratio=.15),
                            get_cifar_100_data_set_collection(validation_ratio=.15)]
    methods = [node_importance_by_dummy_activation_from_input_layer,
               node_importance_by_real_activation_from_input_layer,
               node_importance_by_square_sum,
               node_importance_by_removal,
               node_importance_random,
               node_importance_optimal_brain_damage,
               node_importance_full_taylor_series,
               node_importance_by_real_activation_from_input_layer_variance,
               node_importance_error_derrivative,
               dummy_random_weights
               ]

    final_dict = defaultdict(lambda: [])

    with open(file_name_all, 'w') as result_file:
        result_file.write(
            'method, data_set, before_prune_train, before_prune_validation, before_prune_trest, after_prune_train,
after_prune_validataion, after_prune_test, after_converge_train, after_converge_validataion,
after_converge_test, converge_iterations\n')
        for data in data_set_collections:
            for _ in range(NUM_TRIES):
                tf.reset_default_graph()
                with tf.Session() as session:
                    input_layer = InputLayer(data.features_shape)

                    if len(data.features_shape) > 1:
                        input_layer = FlattenLayer(input_layer)

                    layer = HiddenLayer(input_layer, start, session=session,
                                        layer_noise_std=.5,
                                        node_importance_func=None,
                                        non_liniarity=tf.nn.relu,
                                        batch_normalize_input=True)
                    output = CategoricalOutputLayer(layer, data.labels_shape,
                                                    batch_normalize_input=True,
                                                    regularizer_weighting=0.01,
                                                    layer_noise_std=.5
                                                    )

                    output.train_till_convergence(data.train, data.validation, learning_rate=0.0001)

                    state = output.get_network_state()

                    for method in methods:
                        output.set_network_state(state)
                        layer._node_importance_func = method

                        _, _, target_loss_test_before_resize_test = output.evaluation_stats(data.test)
                        _, _, target_loss_test_before_resize_validation = output.evaluation_stats(data.validation)
                        _, _, target_loss_test_before_resize_train = output.evaluation_stats(data.train)

                        no_splitting_or_pruning = method == dummy_random_weights

                        layer.resize(end, data_set_train=data.train,
```

```
                    data_set_validation=data.validation,
                    no_splitting_or_pruning=no_splitting_or_pruning)

            _, _, target_loss_test_after_resize_test = output.evaluation_stats(data.test)
            _, _, target_loss_test_after_resize_validation = output.evaluation_stats(data.validation)
            _, _, target_loss_test_after_resize_train = output.evaluation_stats(data.train)

            error, iterations = output.train_till_convergence(data.train, data.validation,
                                                              learning_rate=0.0001)

            _, _, after_converge_test = output.evaluation_stats(data.test)
            _, _, after_converge_validation = output.evaluation_stats(data.validation)
            _, _, after_converge_train = output.evaluation_stats(data.train)

            final_dict[method.__name__].append((target_loss_test_before_resize_train,
                                                target_loss_test_before_resize_validation,
                                                target_loss_test_before_resize_test,
                                                target_loss_test_after_resize_train,
                                                target_loss_test_after_resize_validation,
                                                target_loss_test_after_resize_test,
                                                after_converge_train,
                                                after_converge_validation,
                                                after_converge_test))

            result_file.write('%s,%s,%s,%s,%s,%s,%s,%s, %s, %s, %s, %s\n' % (
                method.__name__, data.name, target_loss_test_before_resize_train,
                target_loss_test_before_resize_validation,
                target_loss_test_before_resize_test,
                target_loss_test_after_resize_train,
                target_loss_test_after_resize_validation,
                target_loss_test_after_resize_test,
                after_converge_train,
                after_converge_validation,
                after_converge_test,
                iterations))
            result_file.flush()

    with open(file_name_avg, "w") as file_avg:
        file_avg.write(
            'method, before_prune_train, before_prune_validataion, before_prune_trest, after_prune_train,
after_prune_validataion, after_prune_test, after_converge_train, after_converge_validataion, after_convert_test,
test_diff\n')
        for name, values in final_dict.iteritems():
            v_len = float(len(values))
            averages = tuple(sum(x[i] for x in values) / v_len for i in range(len(values[0])))
            averages = averages + (averages[2] - averages[-2],)
            file_avg.write('%s,%s,%s,%s,%s,%s,%s,%s, %s, %s, %s\n' % ((name,) + averages))


if __name__ == '__main__':
    main()
```

tensordynamic/tests/layers/**base_layer_testcase.py**

```
import numpy as np

from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.base_tf_testcase import BaseTfTestCase


class BaseLayerWrapper(object):
    MNIST_DATA_DIR = "../../tensor_dynamic/data/MNIST_data/"

    def __init__(self):
        """
        This class only exists so base tests aren't run on there own
        """
        pass

    class BaseLayerTestCase(BaseTfTestCase):
        INPUT_NODES = (10,)
        OUTPUT_NODES = (6, )

        def setUp(self):
            super(BaseLayerWrapper.BaseLayerTestCase, self).setUp()
            self._input_layer = InputLayer(self.INPUT_NODES)

        def _create_layer_for_test(self):
            raise NotImplementedError('Override in sub class to return a new instance of the layer to be tested')

        def test_clone(self):
            layer = self._create_layer_for_test()
            clone = layer.clone()
            input_values = np.random.normal(size=(1,) + layer.input_nodes)
            layer_activation = self.session.run(layer.activation_predict,
                                                feed_dict={layer.input_placeholder: input_values})
            clone_activation = self.session.run(clone.activation_predict,
                                                feed_dict={clone.input_placeholder: input_values})

            np.testing.assert_array_almost_equal(
                layer_activation, clone_activation,
                err_msg="Expect activation to be unchanged after cloning, but found difference")

            if layer.bactivate:
                layer_bactivation = self.session.run(layer.bactivation_predict,
                                                     feed_dict={layer.input_placeholder: input_values})
                clone_bactivation = self.session.run(clone.bactivation_predict,
                                                     feed_dict={layer.input_placeholder: input_values})

                np.testing.assert_array_almost_equal(
                    layer_bactivation, clone_bactivation,
                    err_msg="Expect bactivation to be unchanged after cloning, but found difference")

        def test_resize(self):
            layer = self._create_layer_for_test()

            input_noise = np.random.normal(size=[1, layer.input_nodes])
```

```
        layer_activation = self.session.run(layer.activation_predict,
                                            feed_dict={layer.input_placeholder: input_noise})

        layer.resize(layer.output_nodes+1)

        layer_activation_post_resize = self.session.run(layer.activation_predict,
                                                        feed_dict={layer.input_placeholder: input_noise})

        self.assertEqual(layer_activation.shape[-1]+1, layer_activation_post_resize.shape[-1])

    def test_downstream_layers(self):
        layer = self._create_layer_for_test()

        layer2 = HiddenLayer(layer, 2, session=self.session)
        layer3 = HiddenLayer(layer2, 3, session=self.session)

        self.assertEquals(list(layer.downstream_layers), [layer2, layer3])
```

tensordynamic/tests/layers/**test_back_weight_candidate_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.back_weight_candidate_layer import BackWeightCandidateLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestBackWeightCandidateLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return BackWeightCandidateLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_more_nodes_improves_reconstruction_loss_mnist(self):
        data = self.mnist_data.train.features
        recon_1 = self.reconstruction_loss_for(1, data)
```

```
        recon_2 = self.reconstruction_loss_for(2, data)
        recon_5 = self.reconstruction_loss_for(5, data)
        recon_20 = self.reconstruction_loss_for(20, data)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)

    def test_more_nodes_improves_reconstruction_loss_gauss(self):
        data = self.data_sum_of_gaussians(5, 40, 500)
        recon_1 = self.reconstruction_loss_for(1, data)
        recon_2 = self.reconstruction_loss_for(2, data)
        recon_5 = self.reconstruction_loss_for(5, data)
        recon_20 = self.reconstruction_loss_for(20, data)
        recon_50 = self.reconstruction_loss_for(50, data)
        recon_100 = self.reconstruction_loss_for(100, data)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)
        self.assertLess(recon_100, recon_20)

    def reconstruction_loss_for(self, output_nodes, data):
        bw_layer1 = BackWeightCandidateLayer(InputLayer(len(data[0])), output_nodes,
non_liniarity=tf.nn.sigmoid,
                                             session=self.session,
                                             noise_std=0.3)

        cost = bw_layer1.unsupervised_cost_train()
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)

        self.session.run(tf.initialize_all_variables())

        for i in range(100):
            for j in range(0, len(data) - 100, 100):
                self.session.run(optimizer, feed_dict={bw_layer1.input_placeholder: data[j:j + 100]})

        result = self.session.run(bw_layer1.unsupervised_cost_predict(),
                                  feed_dict={bw_layer1.input_placeholder: data})
        print("denoising with %s hidden layer had cost %s" % (output_nodes, result))
        return result
```

tensordynamic/tests/layers/**test_back_weight_layer.py**

```
import numpy as np
import tensorflow as tf

from tensor_dynamic.layers.back_weight_layer import BackWeightLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestBackWeightLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return BackWeightLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_more_nodes_improves_reconstruction_loss(self):
        recon_1 = self.reconstruction_loss_for(1)
        recon_2 = self.reconstruction_loss_for(2)
        recon_5 = self.reconstruction_loss_for(5)
        recon_20 = self.reconstruction_loss_for(20)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)

    def reconstruction_loss_for(self, output_nodes):
        data = self.mnist_data
        bw_layer1 = BackWeightLayer(InputLayer(784), output_nodes, non_liniarity=tf.nn.sigmoid,
session=self.session,
                                    noise_std=0.3)

        cost = bw_layer1.unsupervised_cost_train()
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)

        self.session.run(tf.initialize_all_variables())

        end_epoch = data.train.epochs_completed + 5
```

```
        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            self.session.run(optimizer, feed_dict={bw_layer1.input_placeholder: train_x})

        result = self.session.run(bw_layer1.unsupervised_cost_predict(),
                                  feed_dict={bw_layer1.input_placeholder: data.train.features})
        print("denoising with %s hidden nodes had cost %s" % (output_nodes, result))
        return result

    def test_reconstruction_of_single_input(self):
        input_layer = InputLayer(1)
        layer = BackWeightLayer(input_layer, 1, non_liniarity=tf.nn.sigmoid, session=self.session, noise_std=0.3)

        cost = layer.unsupervised_cost_train()
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)

        self.session.run(tf.initialize_all_variables())

        data = np.random.normal(0.5, 0.5, size=[200, 1])

        for x in range(100):
            self.session.run([optimizer], feed_dict={input_layer.input_placeholder: data})

        result = self.session.run([cost], feed_dict={input_layer.input_placeholder: data})
        print result
```

tensordynamic/tests/layers/**test_batch_norm_layer.py**

```
import numpy as np
import tensorflow as tf

from tensor_dynamic.layers.batch_norm_layer import BatchNormLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestBatchNormLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return BatchNormLayer(self._input_layer, self.session)

    def test_normalize(self):
        samples = 20
        input_nodes = 20
        input = InputLayer(input_nodes)
        batchLayer = BatchNormLayer(input, self.session)

        data = np.random.normal(200., 100., size=(samples, input_nodes))

        result = self.session.run(batchLayer.activation_train,
                                  feed_dict={batchLayer.input_placeholder:
                                                 data})

        self.assertAlmostEqual(result.mean(), 0., 3)
        self.assertAlmostEqual(result.var(), 1., 3)

    def test_predict_after_training(self):
        samples = 200
        input_nodes = 2
        input = InputLayer(input_nodes)
        batch_norma_layer = BatchNormLayer(input, self.session)

        # add the updates of batch normalization statistics to train_step
        train = batch_norma_layer.activation_train
        # with tf.control_dependencies([train]):
        #     train = tf.group(batch_norma_layer.assign_op)

        for i in range(200):
            data = np.random.normal(200., 10., size=(samples, input_nodes))
            self.session.run(train,
                             feed_dict={batch_norma_layer.input_placeholder:
                                            data})

        data2 = np.random.normal(200., 10., size=(samples, input_nodes))

        result = self.session.run(batch_norma_layer.activation_predict,
                                  feed_dict={batch_norma_layer.input_placeholder:
                                                 data2})

        self.assertAlmostEqual(result.mean(), 0., delta=10.)
        self.assertAlmostEqual(result.var(), 1., delta=1.)

    def test_resize(self):
```

```
        # batch norm layer is resized based only on it's input layer
        input_nodes = 2
        input = InputLayer(input_nodes)
        layer = HiddenLayer(input, 2, self.session)
        batchLayer = BatchNormLayer(layer, self.session)

        RESIZE_NODES = 3
        layer.resize(RESIZE_NODES)

        self.assertEqual(batchLayer.output_nodes, (RESIZE_NODES, ))

        self.session.run(batchLayer.activation_predict, feed_dict={batchLayer.input_placeholder: [np.ones(2)]})

    def test_predict_vs_train_bn(self):
        data = self.mnist_data
        bn = BatchNormLayer(InputLayer(784), session=self.session)

        optimizer = bn.activation_train

        # add the updates of batch normalization statistics to train_step
        # with tf.control_dependencies([optimizer]):
        #     optimizer = tf.group(bn.assign_op)

        self.session.run(tf.initialize_all_variables())

        end_epoch = data.train.epochs_completed + 3

        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            self.session.run(optimizer, feed_dict={bn.input_placeholder: train_x})

        result_predict = self.session.run(bn.activation_predict,
                                          feed_dict={bn.input_placeholder: data.test.features})

        result_train = self.session.run(bn.activation_train,
                                        feed_dict={bn.input_placeholder: data.test.features})

        self.assertAlmostEqual(result_train.mean(), result_predict.mean(), delta=0.2)
        self.assertAlmostEqual(result_train.var(), result_predict.var(), delta=0.2)

    def test_predict_vs_train_similar_activation(self):
        data = self.mnist_data
        bn = BatchNormLayer(InputLayer(784), session=self.session)
        layer = HiddenLayer(bn, 5, session=self.session, bactivate=True)

        cost = layer.unsupervised_cost_train()
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)

        # add the updates of batch normalization statistics to train_step
        # with tf.control_dependencies([optimizer]):
        #     optimizer = tf.group(bn._assign_op)

        self.session.run(tf.initialize_all_variables())

        end_epoch = data.train.epochs_completed + 3

        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            self.session.run(optimizer, feed_dict={layer.input_placeholder: train_x})
```

```
        result_train = self.session.run(layer.unsupervised_cost_train(),
                                        feed_dict={layer.input_placeholder: data.test.features})
        result_predict = self.session.run(layer.unsupervised_cost_predict(),
                                          feed_dict={layer.input_placeholder: data.test.features})

        self.assertAlmostEqual(result_train, result_predict, delta=0.2)
```

tensordynamic/tests/layers/**test_convolutional_layer.py**

```
import numpy as np
import tensorflow as tf

from tensor_dynamic.layers.convolutional_layer import ConvolutionalLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestConvolutionalLayer(BaseLayerWrapper.BaseLayerTestCase):
    INPUT_NODES = (10, 10, 1)
    OUTPUT_NODES = (3, 3, 16)

    def _create_layer_for_test(self):
        return ConvolutionalLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_create_layer(self):
        convolution_dimensions = (5, 5, 4)
        input_p = tf.placeholder("float", (None, 10, 10, 1))
        layer = ConvolutionalLayer(InputLayer(input_p), convolution_dimensions, session=self.session)

        self.assertEqual(layer.activation_predict.get_shape().as_list(), [None, 10, 10, 4])

    def test_create_extra_weight_dimensions(self):
        convolutional_nodes = 3, 3, 16
        layer = ConvolutionalLayer(InputLayer((10, 10, 2)), convolutional_nodes, session=self.session,
```

```
                           weights=np.array([[[[100.0]]]], dtype=np.float32))

        self.assertEqual(layer._weights.get_shape().as_list(), [3, 3, 2, 16])

    def test_reshape(self):
        convolution_nodes = (4, 4, 8)
        input_vals = np.random.normal(size=(1, 20, 20, 3)).astype(np.float32)
        layer = ConvolutionalLayer(InputLayer((20, 20, 3)), convolution_nodes, session=self.session)

        result1 = self.session.run(layer.activation_predict, feed_dict={layer.input_placeholder: input_vals})

        layer.resize(9)
        result2 = self.session.run(layer.activation_predict, feed_dict={layer.input_placeholder: input_vals})

        print(result1)
        print(result2)

        self.assertEquals(result2.shape[3], 9)

    def test_create_extra_weight_dimensions_fail_case(self):
        layer = ConvolutionalLayer(InputLayer((10, 10, 3)), (2, 2, 4), session=self.session,
                           weights=np.random.normal(size=(2, 2, 1, 1)).astype(np.float32))

        self.assertEqual(layer._weights.get_shape().as_list(), [2, 2, 3, 4])

    def test_resize(self):
        convolution_nodes = (4, 4, 8)
        input_p = tf.placeholder("float", (None, 20, 20, 3))
        layer = ConvolutionalLayer(InputLayer(input_p), convolution_nodes, session=self.session)
        layer.resize(9)

        print layer._bias.get_shape()

        self.assertEqual(layer.activation_predict.get_shape().as_list(), [None, 20, 20, 9])
        self.assertEquals(layer.output_nodes, (20, 20, 9))

    def test_get_output_layer_activation(self):
        input_p = tf.placeholder("float", (None, 10, 10, 3))
        layer = ConvolutionalLayer(InputLayer(input_p), (4, 4, 4), session=self.session)
        layer2 = ConvolutionalLayer(layer, (2, 2, 8), session=self.session)
        layer3 = ConvolutionalLayer(layer2, (2, 2, 16), session=self.session)

        self.assertEquals(layer.last_layer.activation_predict, layer3.activation_predict)

    # def test_layer_noisy_input_activation(self):
    #     input_size = 100
    #     noise_std = 1.
    #     input_p = tf.placeholder("float", (None, input_size))
    #     layer = ConvolutionalLayer(InputLayer(input_p), input_size,
    #                              weights=np.diag(np.ones(input_size, dtype=np.float32)),
    #                              bias=np.zeros(input_size, dtype=np.float32),
    #                              session=self.session,
    #                              non_liniarity=tf.identity)
    #
    #     result_noisy = self.session.run(layer.activation_train,
    #                                    feed_dict={
    #                                        input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})
    #
    #     self.assertAlmostEqual(result_noisy.std(), noise_std, delta=noise_std / 5.,
    #                            msg="the result std should be the noise_std")
```

```
#
#     layer.predict = True
#
#     result_clean = self.session.run(layer.activation_predict, feed_dict={
#         input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})
#
#     self.assertAlmostEqual(result_clean.std(), 0., places=7,
#                            msg="There should be no noise in the activation")
```

tensordynamic/tests/layers/**test_denoising_source_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.back_weight_candidate_layer import BackWeightCandidateLayer
from tensor_dynamic.layers.denoising_source_layer import DenoisingSourceLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper

# CURRENTLY BROKEN
class TestDenoisingSourceLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return DenoisingSourceLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_more_nodes_improves_reconstruction_loss_mnist(self):
        data = self.mnist_data.train.features
        recon_1 = self.reconstruction_loss_for(1, data)
        recon_2 = self.reconstruction_loss_for(2, data)
        recon_5 = self.reconstruction_loss_for(5, data)
        recon_20 = self.reconstruction_loss_for(20, data)
        recon_100 = self.reconstruction_loss_for(100, data)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)
        self.assertLess(recon_100, recon_20)

    def test_more_nodes_improves_reconstruction_loss_gauss(self):
        data = self.data_sum_of_gaussians(5, 40, 500)
        recon_1 = self.reconstruction_loss_for(1, data)
        recon_2 = self.reconstruction_loss_for(2, data)
        recon_5 = self.reconstruction_loss_for(5, data)
        recon_20 = self.reconstruction_loss_for(20, data)
        recon_100 = self.reconstruction_loss_for(100, data)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)
        self.assertLess(recon_100, recon_20)

    def reconstruction_loss_for(self, output_nodes, data):
        bw_layer1 = BackWeightCandidateLayer(InputLayer(len(data[0])), output_nodes,
non_liniarity=tf.nn.sigmoid,
                                             session=self.session,
                                             noise_std=0.3)

        cost = bw_layer1.unsupervised_cost_train()
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)
```

```
        self.session.run(tf.initialize_all_variables())

        for i in range(5):
            for j in range(0, len(data) - 100, 100):
                self.session.run(optimizer, feed_dict={bw_layer1.input_placeholder: data[j:j + 100]})

        result = self.session.run(bw_layer1.unsupervised_cost_predict(),
                                  feed_dict={bw_layer1.input_placeholder: data})
        print("denoising with %s hidden layer had cost %s" % (output_nodes, result))
        return result
```

tensordynamic/tests/layers/**test_duel_state_relu_layer.py**

```
import numpy as np

from tensor_dynamic.categorical_trainer import CategoricalTrainer
from tensor_dynamic.layers.batch_norm_layer import BatchNormLayer
```

```
from tensor_dynamic.layers.duel_state_relu_layer import DuelStateReluLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.train_policy import DuelStateReluTrainPolicy
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestDuelStateReluLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return DuelStateReluLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_mnist_start_large(self):
        data = self.mnist_data

        input_layer = InputLayer(784)
        hidden_1 = DuelStateReluLayer(input_layer, 200, session=self.session, inactive_nodes_to_leave=200)
        output = HiddenLayer(hidden_1, self.MNIST_OUTPUT_NODES, session=self.session)
        trainer = CategoricalTrainer(output, 0.1)

        end_epoch = data.train.epochs_completed + 5

        print(trainer.accuracy(data.test.features, data.test.labels))

        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            trainer.train(train_x, train_y)

        accuracy, cost = trainer.accuracy(data.test.features, data.test.labels)
        print(accuracy, cost)
        # print(output.active_nodes())
        print(hidden_1.active_nodes())

        self.assertGreater(accuracy, 90.)
        # self.assertEqual(output.active_nodes(), self.MNIST_OUTPUT_NODES, msg='expect all output nodes to
be active')
        self.assertLess(hidden_1.active_nodes(), hidden_1.output_nodes, msg='expect not all hidden nodes to be
active')

    def test_prune_layer(self):
        # create layer and active in such a way that all but 1 output node is useless
        self.INPUT_NODES = 3
        self.OUTPUT_NODES = 1
        x = np.zeros((self.INPUT_NODES, self.OUTPUT_NODES), np.float32)
        for i in range(self.OUTPUT_NODES):
            x[0, i - 1] = 1.0
        y = np.zeros((self.OUTPUT_NODES, self.OUTPUT_NODES), np.float32)
        np.fill_diagonal(y, 1.)
        layer_1 = DuelStateReluLayer(InputLayer(self.INPUT_NODES), self.INPUT_NODES,
session=self.session, weights=x,
                                    width_regularizer_constant=1e-2)
        layer_2 = HiddenLayer(layer_1, self.OUTPUT_NODES, weights=y, freeze=True)
        trainer = CategoricalTrainer(layer_2, 0.1)

        data_1 = [1.0] * self.INPUT_NODES
        data_2 = [0.0] * self.INPUT_NODES
        label_1 = [1.0] + [0.0] * (self.OUTPUT_NODES - 1)  # only the first node is correlated with the input
        label_2 = [0.0] * self.OUTPUT_NODES

        inputs = [data_1, data_2]
        labels = [label_1, label_2]
```

```
        for i in range(500):
            self.session.run([trainer._train],
                             feed_dict={layer_2.input_placeholder: inputs[:1],
                                        trainer._target_placeholder: labels[:1],
                                        trainer._learn_rate_placeholder: 0.05})
            self.session.run([trainer._train],
                             feed_dict={layer_2.input_placeholder: inputs[1:],
                                        trainer._target_placeholder: labels[1:],
                                        trainer._learn_rate_placeholder: 0.05})

        # layer should only have 1 active node
        self.assertGreater(layer_1.width()[0], DuelStateReluLayer.ACTIVE_THRESHOLD)
        self.assertEqual(layer_1.active_nodes(), 1)

        activation_pre_prune = self.session.run([layer_2.activation_predict],
                                                feed_dict={layer_1.input_placeholder: inputs})

        # after pruning layer should have 2 nodes
        layer_1.prune(inactive_nodes_to_leave=1)

        self.assertEqual(layer_1.output_nodes, 2)

        activation_post_prune = self.session.run([layer_2.activation_predict],
                                                 feed_dict={layer_1.input_placeholder: inputs})

        np.testing.assert_array_almost_equal(activation_pre_prune, activation_post_prune, decimal=2)

    def test_mnist(self):
        data = self.mnist_data
        input = InputLayer(self.MNIST_INPUT_NODES)
        d_1 = DuelStateReluLayer(input, 3, width_regularizer_constant=1e-7, width_binarizer_constant=1e-10,
                                 session=self.session)
        # when we add in batch norm layers we find that no active nodes are created, width is always less than
0.5?
        # bn_1 = BatchNormLayer(d_1)
        d_2 = DuelStateReluLayer(d_1, 3, width_regularizer_constant=1e-7, width_binarizer_constant=1e-10, )
        # bn_2 = BatchNormLayer(d_2)
        output = HiddenLayer(d_2, self.MNIST_OUTPUT_NODES)

        trainer = CategoricalTrainer(output, 0.1)
        end_epoch = data.train.epochs_completed + 20

        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            trainer.train(train_x, train_y)

        accuracy, cost = trainer.accuracy(data.test.features, data.test.labels)
        print(accuracy, cost)
        print("active nodes ", d_1.active_nodes())
        self.assertGreater(accuracy, 70.)

    def test_with_train_policy(self):
        data = self.mnist_data
        input = InputLayer(self.MNIST_INPUT_NODES)
        d_1 = DuelStateReluLayer(input, 1, width_regularizer_constant=1e-7, width_binarizer_constant=1e-9,
                                 session=self.session)
        bn_1 = BatchNormLayer(d_1)
        d_2 = DuelStateReluLayer(bn_1, 1, width_regularizer_constant=1e-7, width_binarizer_constant=1e-9, )
        bn_2 = BatchNormLayer(d_2)
```

```
        output = HiddenLayer(bn_2, self.MNIST_OUTPUT_NODES)
        trainer = CategoricalTrainer(output, 0.1)
        trainer = DuelStateReluTrainPolicy(trainer, data, 100, max_iterations=300, stop_accuracy=90.,
                                           grow_after_turns_without_improvement=1,)
        trainer.run_full(True)
```

tensordynamic/tests/layers/**test_flatten_layer.py**

```
import numpy as np
import tensorflow as tf

from tensor_dynamic.layers.convolutional_layer import ConvolutionalLayer
from tensor_dynamic.layers.flatten_layer import FlattenLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper

class TestFlattenLayer(BaseLayerWrapper.BaseLayerTestCase):
    INPUT_NODES = (10, 10, 4)

    def _create_layer_for_test(self):
        return FlattenLayer(self._input_layer, session=self.session)

    def test_create_layer(self):
        input_p = tf.placeholder("float", (None, 10, 10, 4))
        layer = FlattenLayer(InputLayer(input_p), session=self.session)

        self.assertEqual(layer.activation_predict.get_shape().as_list(), [None, 10 * 10 * 4])

    def test_reshape(self):
        input_vals = np.random.normal(size=(1, 2, 2, 1)).astype(np.float32)

        convolution_nodes = (2, 2, 2)
        input_p = tf.placeholder("float", (None, 2, 2, 1))
        layer = ConvolutionalLayer(InputLayer(input_p), convolution_nodes, session=self.session)
        flatten = FlattenLayer(layer, session=self.session)
        result1 = self.session.run(layer.activation_predict, feed_dict={flatten.input_placeholder: input_vals})

        layer.resize(3)
        result2 = self.session.run(layer.activation_predict, feed_dict={flatten.input_placeholder: input_vals})

        print(result1)
        print(result2)
```

```
        self.assertEquals(result2.shape[3], 3)

    def test_resize(self):
        convolution_nodes = (2, 2, 2)
        input_p = tf.placeholder("float", (None, 2, 2, 1))
        layer = ConvolutionalLayer(InputLayer(input_p), convolution_nodes, session=self.session)
        flatten = FlattenLayer(layer, session=self.session)

        self.assertEqual(flatten.output_nodes[0], 2 * 2 * 2)

        layer.resize(3)
        self.assertEqual(flatten.output_nodes[0], 2 * 2 * 3)
```

tensordynamic/tests/layers/**test_hidden_layer.py**

```
import numpy as np
import tensorflow as tf
from math import log

from tensor_dynamic.layers.categorical_output_layer import CategoricalOutputLayer
from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.hidden_layer import HiddenLayer
from tensor_dynamic.node_importance import node_importance_optimal_brain_damage,
node_importance_by_removal, \
    node_importance_by_real_activation_from_input_layer_variance,
node_importance_full_taylor_series
from tensor_dynamic.utils import get_tf_optimizer_variables
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestHiddenLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return HiddenLayer(self._input_layer, self.OUTPUT_NODES, session=self.session)

    def test_create_layer(self):
        output_nodes = 20
        input_p = tf.placeholder("float", (None, 10))
        layer = HiddenLayer(InputLayer(input_p), output_nodes, session=self.session)

        self.assertEqual(layer.activation_predict.get_shape().as_list(), [None, output_nodes])

    def test_create_extra_weight_dimensions(self):
        output_nodes = 2
        input_p = tf.placeholder("float", (None, 2))
        layer = HiddenLayer(InputLayer(input_p), output_nodes, session=self.session,
                            weights=np.array([[100.0]], dtype=np.float32))

        self.assertEqual(layer._weights.get_shape().as_list(), [2, 2])

    def test_reshape(self):
        output_nodes = 2
        input_p = tf.placeholder("float", (None, 2))
        layer = HiddenLayer(InputLayer(input_p), output_nodes, session=self.session,
                            weights=np.array([[100.0]], dtype=np.float32))

        result1 = self.session.run(layer.activation_predict, feed_dict={layer.input_placeholder: [[1., 1.]]})

        layer.resize(3)
        result2 = self.session.run(layer.activation_predict, feed_dict={layer.input_placeholder: [[1., 1.]]})
```

```
        print(result1)
        print(result2)

        self.assertEquals(len(result2[0]), 3)

    def test_create_extra_weight_dimensions_fail_case(self):
        output_nodes = 2
        input_p = tf.placeholder("float", (None, 4))
        layer = HiddenLayer(InputLayer(input_p), output_nodes, session=self.session,
                            weights=np.array([[10., 10.],
                                              [10., 10.],
                                              [10., 10.]], dtype=np.float32))

        self.assertEqual(layer._weights.get_shape().as_list(), [4, 2])

    def test_resize(self):
        output_nodes = 10
        input_p = tf.placeholder("float", (None, 10))
        layer = HiddenLayer(InputLayer(input_p), output_nodes, session=self.session)
        layer.resize(output_nodes + 1)

        print layer._bias.get_shape()

        self.assertEqual(layer.activation_predict.get_shape().as_list(), [None, output_nodes + 1])
        self.assertEquals(layer.output_nodes, (output_nodes + 1,))

    def test_get_output_layer_activation(self):
        input_p = tf.placeholder("float", (None, 10))
        layer = HiddenLayer(InputLayer(input_p), 1, session=self.session)
        layer2 = HiddenLayer(layer, 2, session=self.session)
        layer3 = HiddenLayer(layer2, 3, session=self.session)

        self.assertEquals(layer.last_layer.activation_predict, layer3.activation_predict)

    def test_layer_noisy_input_activation(self):
        input_size = 100
        noise_std = 1.
        input_p = tf.placeholder("float", (None, input_size))
        layer = HiddenLayer(InputLayer(input_p), input_size,
                            weights=np.diag(np.ones(input_size, dtype=np.float32)),
                            bias=np.zeros(input_size, dtype=np.float32),
                            session=self.session,
                            non_liniarity=tf.identity,
                            layer_noise_std=noise_std)

        result_noisy = self.session.run(layer.activation_train,
                                        feed_dict={
                                            input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})

        self.assertAlmostEqual(result_noisy.std(), noise_std, delta=noise_std / 5.,
                               msg="the result std should be the noise_std")

        result_clean = self.session.run(layer.activation_predict, feed_dict={
            input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})

        self.assertAlmostEqual(result_clean.std(), 0., places=7,
                               msg="There should be no noise in the activation")

    def test_layer_noisy_input_bactivation(self):
        input_size = 100
```

```
        noise_std = 1.
        input_p = tf.placeholder("float", (None, input_size))
        layer = HiddenLayer(InputLayer(input_p), input_size,
                            weights=np.diag(np.ones(input_size, dtype=np.float32)),
                            bias=np.zeros(input_size, dtype=np.float32),
                            back_bias=np.zeros(input_size, dtype=np.float32),
                            session=self.session,
                            bactivate=True,
                            non_liniarity=tf.identity,
                            layer_noise_std=noise_std)

        result_noisy = self.session.run(layer.bactivation_train,
                                        feed_dict={
                                            input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})

        self.assertAlmostEqual(result_noisy.std(), noise_std, delta=noise_std / 4.,
                               msg="the result std should be the noise_std")

        result_clean = self.session.run(layer.bactivation_predict, feed_dict={
            input_p: np.ones(input_size, dtype=np.float32).reshape((1, input_size))})

        self.assertAlmostEqual(result_clean.std(), 0., delta=0.1,
                               msg="When running in prediction mode there should be no noise in the
bactivation")

    def test_more_nodes_improves_reconstruction_loss(self):
        recon_1 = self.reconstruction_loss_for(1)
        recon_2 = self.reconstruction_loss_for(2)
        self.assertLess(recon_2, recon_1)
        recon_5 = self.reconstruction_loss_for(5)
        self.assertLess(recon_5, recon_2)
        recon_20 = self.reconstruction_loss_for(20)
        self.assertLess(recon_20, recon_5)
        recon_500 = self.reconstruction_loss_for(500)
        self.assertLess(recon_500, recon_20)

    def reconstruction_loss_for(self, output_nodes):
        data = self.mnist_data
        input_layer = InputLayer(784)
        bw_layer1 = HiddenLayer(input_layer, output_nodes, session=self.session,
                                layer_noise_std=1.0, bactivate=True)

        cost_train = tf.reduce_mean(
            tf.reduce_sum(tf.square(bw_layer1.bactivation_train - input_layer.activation_train), 1))
        cost_predict = tf.reduce_mean(
            tf.reduce_sum(tf.square(bw_layer1.bactivation_predict - input_layer.activation_predict), 1))
        optimizer = tf.train.AdamOptimizer(0.0001).minimize(cost_train)

        self.session.run(tf.initialize_all_variables())

        end_epoch = data.train.epochs_completed + 5

        while data.train.epochs_completed <= end_epoch:
            train_x, train_y = data.train.next_batch(100)
            _, tr = self.session.run([optimizer, cost_train], feed_dict={bw_layer1.input_placeholder: train_x})
            # print(tr)

        result = self.session.run(cost_predict,
                                  feed_dict={bw_layer1.input_placeholder: data.train.features})
        print("denoising with %s hidden layer had cost %s" % (output_nodes, result))
```

```
        return result

    def test_reconstruction_of_single_input(self):
        input_layer = InputLayer(1)
        layer = HiddenLayer(input_layer, 1, bactivate=True, session=self.session, layer_noise_std=0.3)

        cost_train = tf.reduce_mean(
            tf.reduce_sum(tf.square(layer.bactivation_train - input_layer.activation_train), 1))
        cost_predict = tf.reduce_mean(
            tf.reduce_sum(tf.square(layer.bactivation_predict - input_layer.activation_predict), 1))
        optimizer = tf.train.AdamOptimizer(0.1).minimize(cost_train)

        self.session.run(tf.initialize_all_variables())

        data = np.random.normal(0.5, 0.5, size=[200, 1])

        for x in range(100):
            self.session.run([optimizer], feed_dict={input_layer.input_placeholder: data})

        result = self.session.run([cost_predict], feed_dict={input_layer.input_placeholder: data})
        print result

    def test_noise_reconstruction(self):
        INPUT_DIM = 10
        HIDDEN_NODES = 1
        input_layer = InputLayer(INPUT_DIM)
        bw_layer1 = HiddenLayer(input_layer, HIDDEN_NODES, session=self.session,
layer_noise_std=1.0,
                                bactivate=True)

        # single cluster reconstruct
        data = []
        for i in range(10):
            data.append([i * .1] * INPUT_DIM)

        cost_train = tf.reduce_mean(
            tf.reduce_sum(tf.square(bw_layer1.bactivation_train - input_layer.activation_train), 1))
        cost_predict = tf.reduce_mean(
            tf.reduce_sum(tf.square(bw_layer1.bactivation_predict - input_layer.activation_predict), 1))
        optimizer = tf.train.AdamOptimizer(0.01).minimize(cost_train)

        self.session.run(tf.initialize_all_variables())

        for j in range(200):
            self.session.run(optimizer, feed_dict={bw_layer1.input_placeholder: data})

        result = self.session.run(cost_predict,
                                  feed_dict={bw_layer1.input_placeholder: data})

        print("denoising with %s hidden layer had cost %s" % (HIDDEN_NODES, result))

    def test_find_best_layer_size(self):
        data = self.mnist_data
        input_layer = InputLayer(data.features_shape)
        layer = HiddenLayer(input_layer, 10, session=self.session, layer_noise_std=1.0, bactivate=False)
        output = CategoricalOutputLayer(layer, data.labels_shape)

        layer.find_best_size(data.train, data.test,
                             lambda m, d: output.evaluation_stats(d)[0] - log(output.get_parameters_all_layers()),
                             initial_learning_rate=0.1, tuning_learning_rate=0.1)
```

```
        assert layer.get_resizable_dimension_size() > 10

    # TODO: Move to categorical output layer
    # def test_learn_struture(self):
    #     data = self.mnist_data
    #     input_layer = InputLayer(data.features_shape)
    #     layer = HiddenLayer(input_layer, 10, session=self.session, input_noise_std=1.0,
bactivate=False)
    #     output = CategoricalOutputLayer(layer, data.labels_shape)
    #
    #     output.learn_structure_random(data.train, data.test)
    #
    #     assert layer.get_resizable_dimension_size() > 10

    def test_remove_unimportant_nodes_does_not_affect_test_error(self):
        data = self.mnist_data
        batch_normalize = False
        input_layer = InputLayer(data.features_shape, drop_out_prob=None)
        layer = HiddenLayer(input_layer, 800, session=self.session,
                            batch_normalize_input=batch_normalize,
                            # D.S TODO TEST
                            node_importance_func=node_importance_optimal_brain_damage)
        output = CategoricalOutputLayer(layer, data.labels_shape,
batch_normalize_input=batch_normalize)

        output.train_till_convergence(data.train, data.test, learning_rate=0.001)

        _, _, target_loss_before_resize = output.evaluation_stats(data.test)  # Should this be on test or
train?

        print(target_loss_before_resize)

        layer.resize(795, data_set_validation=data.test)

        _, _, target_loss_after_resize = output.evaluation_stats(data.test)

        print(target_loss_after_resize)

        self.assertAlmostEqual(target_loss_before_resize, target_loss_after_resize, delta=10.0)

    def test_get_and_set_state(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 50, session=self.session,
                            node_importance_func=node_importance_optimal_brain_damage)
        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape,
regularizer_weighting=0.0001)

        acitvation = self.session.run(output.activation_predict, feed_dict={output.input_placeholder:
                                                                           self.mnist_data.train.features[:1]})

        weights_hidden = layer._weights.eval()
        bias_hidden = layer._bias.eval()
        weights_output = output._weights.eval()
        bias_output = output._bias.eval()

        state = layer.get_network_state()

        layer.resize(10)
```

```
        layer.set_network_state(state)

        restored_acitvation = self.session.run(output.activation_predict,
                                               feed_dict={output.input_placeholder: self.mnist_data.train.features[:1]})

        new_weights_hidden = layer._weights.eval()
        new_bias_hidden = layer._bias.eval()
        new_weights_output = output._weights.eval()
        new_bias_output = output._bias.eval()

        np.testing.assert_almost_equal(new_weights_hidden, weights_hidden)
        np.testing.assert_almost_equal(new_bias_hidden, bias_hidden)
        np.testing.assert_almost_equal(new_weights_output, weights_output)
        np.testing.assert_almost_equal(new_bias_output, bias_output)
        np.testing.assert_almost_equal(restored_acitvation, acitvation)

    def test_weights_getter_and_setter(self):
        weights_value = np.random.normal(size=(self.mnist_data.features_shape[0], 1))
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 1, session=self.session, weights=weights_value)

        np.testing.assert_almost_equal(weights_value, layer.weights)

        new_weights_value = np.random.normal(size=(self.mnist_data.features_shape[0], 1))

        layer.weights = new_weights_value

        np.testing.assert_almost_equal(new_weights_value, layer.weights)

    def test_growing(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 1, session=self.session,
                            node_importance_func=node_importance_optimal_brain_damage)
        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape,
regularizer_weighting=0.0001)

        weights_hidden = layer._weights.eval()
        bias_hidden = layer._bias.eval()
        weights_output = output._weights.eval()

        layer.resize(2)

        new_weights_hidden = layer._weights.eval()
        new_bias_hidden = layer._bias.eval()
        new_weights_output = output._weights.eval()

        np.testing.assert_almost_equal(new_weights_output[0], weights_output[0] / 2)

    def test_remove_layer_from_network(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 10, session=self.session,
                            node_importance_func=node_importance_optimal_brain_damage)
        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape,
regularizer_weighting=0.0001)

        activation = self.session.run(output.activation_predict,
                                      feed_dict={output.input_placeholder: self.mnist_data.train.features[:1]})

        layer.remove_layer_from_network()
```

```
        activation = self.session.run(output.activation_predict,
                                      feed_dict={output.input_placeholder: self.mnist_data.train.features[:1]})

        self.assertEqual(output.layer_number, 1)
        self.assertEqual(output.input_nodes, (784,))

    def test_use_state_to_remove_layer(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 10, session=self.session,
                            node_importance_func=node_importance_optimal_brain_damage)
        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape,
regularizer_weighting=0.0001)

        initial_activation = self.session.run(output.activation_predict,
                                              feed_dict={output.input_placeholder: self.mnist_data.train.features[:1]})

        state = output.get_network_state()

        layer.add_intermediate_cloned_layer()

        with_extra_layer_activation = self.session.run(output.activation_predict,
                                                       feed_dict={
                                                           output.input_placeholder: self.mnist_data.train.features[
                                                                                     :1]})

        self.assertNotEqual(tuple(with_extra_layer_activation[0]), tuple(initial_activation[0]))

        output.set_network_state(state)
        restored_activation = self.session.run(output.activation_predict,
                                               feed_dict={output.input_placeholder: self.mnist_data.train.features[:1]})

        np.testing.assert_almost_equal(restored_activation, initial_activation)

    def test_resize_with_batch_norm_and_2_layers_resize_2(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer1 = HiddenLayer(input_layer, 2, session=self.session, batch_normalize_input=True)
        layer2 = HiddenLayer(layer1, 2, session=self.session, batch_normalize_input=True)
        output = CategoricalOutputLayer(layer2, self.mnist_data.labels_shape,
batch_normalize_input=False)

        output.train_till_convergence(self.mnist_data.train, learning_rate=0.1)

        layer2.resize(3)

        output.train_till_convergence(self.mnist_data.train, learning_rate=0.1)

    def test_resize_with_batch_norm_and_2_layers_resize_1(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer1 = HiddenLayer(input_layer, 5, session=self.session, batch_normalize_input=True)
        layer2 = HiddenLayer(layer1, 5, session=self.session, batch_normalize_input=True)
        output = CategoricalOutputLayer(layer2, self.mnist_data.labels_shape,
batch_normalize_input=False)

        # output.train_till_convergence(self.mnist_data.train, learning_rate=0.1)
        optimizer = tf.train.AdamOptimizer()
        loss = optimizer.minimize(output.target_loss_op_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer))))
        self.session.run(loss,
                         feed_dict={output.input_placeholder: self.mnist_data.train.features[:3],
                                    output.target_placeholder: self.mnist_data.train.labels[:3]})
```

```
        layer1.resize(6)

        optimizer2 = tf.train.AdamOptimizer()
        loss2 = optimizer2.minimize(output.target_loss_op_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer2))))
        self.session.run(loss2,
                         feed_dict={output.input_placeholder: self.mnist_data.train.features[:3],
                                    output.target_placeholder: self.mnist_data.train.labels[:3]})

    def test_resize_with_batch_norm_and_2_layers_resize_3(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer1 = HiddenLayer(input_layer, 2, session=self.session, batch_normalize_input=True)
        layer2 = HiddenLayer(layer1, 3, session=self.session, batch_normalize_input=True)

        optimizer = tf.train.AdamOptimizer()
        loss = optimizer.minimize(layer2.activation_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer))))
        self.session.run(loss,
                         feed_dict={input_layer.input_placeholder: self.mnist_data.train.features[:3],
                                    })

        layer1.resize(4)

        optimizer2 = tf.train.AdamOptimizer()
        loss2 = optimizer2.minimize(layer2.activation_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer2))))
        self.session.run(loss2,
                         feed_dict={input_layer.input_placeholder: self.mnist_data.train.features[:3],
                                    })

    def test_resize_with_batch_norm_resize(self):
        input_layer = InputLayer(self.mnist_data.features_shape)
        layer = HiddenLayer(input_layer, 2, session=self.session, batch_normalize_input=True)
        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape,
batch_normalize_input=False)

        # output.train_till_convergence(self.mnist_data.train, learning_rate=0.1)
        optimizer = tf.train.AdamOptimizer()
        loss = optimizer.minimize(output.activation_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer))))
        self.session.run(loss,
                         feed_dict={output.input_placeholder: self.mnist_data.train.features[:3],
                                    output.target_placeholder: self.mnist_data.train.labels[:3]})

        layer.resize(3)

        optimizer2 = tf.train.AdamOptimizer()
        loss2 = optimizer2.minimize(output.activation_predict)
        self.session.run(tf.initialize_variables(list(get_tf_optimizer_variables(optimizer2))))
        self.session.run(loss2,
                         feed_dict={output.input_placeholder: self.mnist_data.train.features[:3],
                                    output.target_placeholder: self.mnist_data.train.labels[:3]})

    def test_bug_issue(self):
        non_liniarity = tf.nn.relu
        regularizer_coeff = 0.01
        last_layer = InputLayer(self.mnist_data.features_shape,
                                # drop_out_prob=.5,
                                layer_noise_std=1.
```

```
                        )

            last_layer = HiddenLayer(last_layer, 100, self.session, non_liniarity=non_liniarity,
                                     batch_normalize_input=True)

            output = CategoricalOutputLayer(last_layer, self.mnist_data.labels_shape, self.session,
                                            batch_normalize_input=True,
                                            regularizer_weighting=regularizer_coeff)

            output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                          learning_rate=.1)

            last_layer.resize(110)

            output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                          learning_rate=.1)

            last_layer.resize(90)

            output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                          learning_rate=.1)

        def test_adding_hidden_layer_with_resize(self):
            non_liniarity = tf.nn.relu
            regularizer_coeff = None
            layer = InputLayer(self.mnist_data.features_shape)

            layer = HiddenLayer(layer, 100, self.session, non_liniarity=non_liniarity,
                                batch_normalize_input=False)

            output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape, self.session,
                                            batch_normalize_input=True,
                                            regularizer_weighting=regularizer_coeff)

            output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                          learning_rate=.1)

            layer.add_intermediate_cloned_layer()
            layer.resize(110)

            self.session.run(output.activation_predict,
                             feed_dict={output.input_placeholder: self.mnist_data.train.features[:3],
                                        output.target_placeholder: self.mnist_data.train.labels[:3]})

        def test_bug_issue_with_state(self):
            non_liniarity = tf.nn.relu
            regularizer_coeff = 0.01
            layer = InputLayer(self.mnist_data.features_shape, layer_noise_std=1.)

            layer = HiddenLayer(layer, 6, self.session, non_liniarity=non_liniarity,
                                batch_normalize_input=True)

            output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape, self.session,
                                            batch_normalize_input=True,
                                            regularizer_weighting=regularizer_coeff)

            state = output.get_network_state()

            layer.resize(10)
```

```
        output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                      learning_rate=.1)

        output.set_network_state(state)

        output.train_till_convergence(self.mnist_data.train, self.mnist_data.validation,
                                      learning_rate=.1)

    def test_hessian(self):
        layer = InputLayer(self.mnist_data.features_shape, layer_noise_std=1.)

        layer = HiddenLayer(layer, 6, self.session,
                            batch_normalize_input=True)

        output = CategoricalOutputLayer(layer, self.mnist_data.labels_shape, self.session,
                                        batch_normalize_input=True)

        hession_op = layer.hessien_with_respect_to_error_op

        result = self.session.run(hession_op,
feed_dict={output.input_placeholder:self.mnist_data.train.features,
                                             output.target_placeholder: self.mnist_data.train.labels})
        print result
```

tensordynamic/tests/layers/**test_ladder_layer.py**

```
import unittest

import numpy as np
import tensorflow as tf
from tests.layers.base_layer_testcase import BaseLayerWrapper

from tensor_dynamic.layers.input_layer import InputLayer, SemiSupervisedInputLayer
from tensor_dynamic.layers.ladder_layer import LadderLayer, LadderGammaLayer
from tensor_dynamic.layers.ladder_output_layer import LadderOutputLayer


class TestLadderLayer(BaseLayerWrapper.BaseLayerTestCase):

    def _create_layer_for_test(self):
        return LadderLayer(SemiSupervisedInputLayer(self.INPUT_NODES), self.OUTPUT_NODES,
session=self.session)

    def test_batch_normalize(self):
        inputs = tf.placeholder("float", (None, 2))
        batch_norm_op = LadderLayer.batch_normalization(inputs)

        self.assertTrue(np.array_equal(self.session.run(batch_norm_op, feed_dict={inputs: [[1.0, 1.0]]}), [[0.0,
0.0]]))
        self.assertTrue(np.array_equal(self.session.run(batch_norm_op, feed_dict={inputs: [[1.0, 1.0], [0.0, -1.0]]}),
                                       [[1., 1.], [-1., -1.]]))

    def test_bactivation(self):
        placeholder = tf.placeholder("float", (None, 4))
        input = InputLayer(placeholder, self.session)
        layer = LadderLayer(input, 2, 0.1, self.session)
        LadderOutputLayer(layer, 0.1, self.session)
        self.assertEquals([None, 4], layer.bactivation_predict.get_shape().as_list())
        self.assertEquals([None, 4], layer.bactivation_train.get_shape().as_list())

    @unittest.skip('Need to fix batch sizing for ladder networks')
    def test_train_xor(self):
        train_x = [[0.0, 1.0, -1.0, 0.0],
                   [1.0, 0.0, -1.0, 1.0],
                   [0.0, 1.0, -1.0, -1.0],
                   [-1.0, 0.5, 1.0, 0.0]]
        train_y = [[-1.0, 0.0],
                   [1.0, 1.0],
                   [0., -1.0],
                   [-1.0, 0.0]]
        targets = tf.placeholder('float', (None, 2))

        ladder = InputLayer(len(train_x[0]), self.session)
        ladder = LadderLayer(ladder, 6, 1000., self.session)
        ladder = LadderLayer(ladder, 6, 10., self.session)
        ladder = LadderGammaLayer(ladder, 2, 0.1, self.session)
        ladder = LadderOutputLayer(ladder, 0.1, self.session)

        cost = ladder.cost_all_layers_train(targets)
        train = tf.train.AdamOptimizer(0.1).minimize(cost)
```

```
        self.session.run(tf.initialize_all_variables())
        _, cost1 = self.session.run([train, cost], feed_dict={ladder.input_placeholder:train_x, targets:train_y})

        print self.session.run([train, cost], feed_dict={ladder.input_placeholder:train_x, targets:train_y})
        print self.session.run([train, cost], feed_dict={ladder.input_placeholder:train_x, targets:train_y})
        print self.session.run([train, cost], feed_dict={ladder.input_placeholder:train_x, targets:train_y})

        _, cost2 = self.session.run([train, cost], feed_dict={ladder.input_placeholder:train_x, targets:train_y})

        self.assertGreater(cost1, cost2, msg="Expected loss to reduce")

    def test_mnist(self):
        import tensor_dynamic.data.mnist_data as mnist

        num_labeled = 100
        data = mnist.get_mnist_data_set_collection("../data/MNIST_data",
number_labeled_examples=num_labeled, one_hot=True)

        batch_size = 100
        num_epochs = 1
        num_examples = 60000
        num_iter = (num_examples/batch_size) * num_epochs
        starter_learning_rate = 0.02
        inputs = tf.placeholder(tf.float32, shape=(None, 784))
        targets = tf.placeholder(tf.float32)

        with tf.Session() as s:
            s.as_default()
            i = InputLayer(inputs)
            l1 = LadderLayer(i, 500, 1000.0, s)
            l2 = LadderGammaLayer(l1, 10, 10.0, s)
            ladder = LadderOutputLayer(l2, 0.1, s)

            loss = ladder.cost_all_layers_train(targets)
            learning_rate = tf.Variable(starter_learning_rate, trainable=False)
            train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)

            bn_updates = tf.group(*(l1.bn_assigns + l2.bn_assigns))
            with tf.control_dependencies([train_step]):
                train_step = tf.group(bn_updates)
            pred_cost = -tf.reduce_mean(tf.reduce_sum(targets * tf.log(tf.clip_by_value(ladder.activation_predict,
1e-10, 1.0)), 1))  # cost used for prediction

            correct_prediction = tf.equal(tf.argmax(ladder.activation_predict, 1), tf.argmax(targets, 1))  # no of
correct predictions
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float")) * tf.constant(100.0)

            s.run(tf.initialize_all_variables())

            #print "init accuracy", s.run([accuracy], feed_dict={inputs: data.test.images, targets: data.test.labels})

            min_loss = 100000.

            writer = tf.train.SummaryWriter("/tmp/td", s.graph_def)
            writer.add_graph(s.graph_def)

            for i in range(num_iter):
                images, labels = data.train.next_batch(batch_size)
                _, loss_val = s.run([train_step, loss], feed_dict={inputs: images, targets: labels})
```

```
            if loss_val < min_loss:
                min_loss = loss_val
            print(i, loss_val)

            # print "acc", s.run([accuracy], feed_dict={inputs: data.test.images, targets: data.test.labels})

        #acc = s.run(accuracy, feed_dict={inputs: data.test.images, targets: data.test.labels})
        print "min loss", min_loss
        #print "final accuracy ", acc
        self.assertLess(min_loss, 20.0)
        #self.assertGreater(acc, 70.0)
```

tensordynamic/tests/layers/**test_variational_autoencoder_layer.py**

```
import tensorflow as tf

from tensor_dynamic.layers.input_layer import InputLayer
from tensor_dynamic.layers.variational_autoencoder_layer import VariationalAutoencoderLayer
from tests.layers.base_layer_testcase import BaseLayerWrapper


class TestVariationalAutoencoderLayer(BaseLayerWrapper.BaseLayerTestCase):
    def _create_layer_for_test(self):
        return VariationalAutoencoderLayer(self._input_layer, self.OUTPUT_NODES, 10, 10, 10, 10,
session=self.session)

    def test_more_nodes_improves_reconstruction_loss(self):
        recon_1 = self.reconstruction_loss_for(1)
        recon_2 = self.reconstruction_loss_for(2)
        recon_5 = self.reconstruction_loss_for(5)
        recon_20 = self.reconstruction_loss_for(20)
        self.assertLess(recon_2, recon_1)
        self.assertLess(recon_5, recon_2)
        self.assertLess(recon_20, recon_5)

    def reconstruction_loss_for(self, output_nodes):
        data = self.mnist_data
        bw_layer1 = VariationalAutoencoderLayer(InputLayer(784), output_nodes,
                                                10, 10, 10, 10,
                                                session=self.session)
```

```
cost = bw_layer1.unsupervised_cost_train()
optimizer = tf.train.AdamOptimizer(0.1).minimize(cost)

self.session.run(tf.initialize_all_variables())

end_epoch = data.train.epochs_completed + 3

while data.train.epochs_completed <= end_epoch:
    train_x, train_y = data.train.next_batch(100)
    self.session.run(optimizer, feed_dict={bw_layer1.input_placeholder: train_x})

result = self.session.run(bw_layer1.unsupervised_cost_predict(),
                          feed_dict={bw_layer1.input_placeholder: data.train.features})
print("denoising with %s hidden layer had cost %s" % (output_nodes, result))
return result
```