How Neural Networks Learn

by

Khoren Petrosyan

Master, American University of Armenia, 2018

A thesis submitted in partial satisfaction of

the requirements for the degree of

Master of Science

in

Computer & Information Science

in the

COLLEGE OF SCIENCE AND ENGINEERING

of the

AMERICAN UNIVERSITY OF ARMENIA

Supervisor: __Arsen Mamikonyan_____

Signature: _____ Date:_____

Committee Member: _____

Signature: _____ Date:_____

Committee Member: _____

Signature: _____ Date:_____

Committee Member: _____

Abstract

Neural Networks stand behind most of the recent breakthroughs in Machine Learning and Artificial Intelligence. Used form computer vision to game-playing smart programs, these systems are very commonplace yet not fully understood. Neural Networks are mostly trained by Gradient Descent optimization algorithm, which works by updating weights of Neural Networks by calculating gradient over training batches. This makes interpretation of what Neural Networks learn and how they learn it unclear unlike most machine learning algorithms, which creates a reputation of black-box methods for Neural Networks. Attempts have been made to interpret Neural Networks, most of which are summarized in this [1] tutorial. In the methods outlined in [1] analysis and interpretation of already trained Neural Networks are conducted, while in this thesis, although some post-mortem analizes are also present, it mainly explores the learning procedure, not the final result of training Neural Networks. The core of this thesis are several experiments that will be organised in separate sections, with three subsections each, the first subsection describing the motivation behind the experiment, second subsection describing the experiment itself and the third one explaining the results. Trough experiments the thesis illustrates that one of the most prevalent intuitions about neural networks, that they learn to generalize by extracting useful features like edges and textures, is incorrect for Feed-Forward Neural Networks, despite being shown correct on Convolutional Neural Networks. I also provide intuitive explanation as for why is that, that ConvNets learn useful features and generalize, while plain Feed-Forward Networks don't.

Licenses for Software and Content

Software Copyright License (to be distributed with software developed for masters project)

Content Copyright License (to be included with Technical Report)

the sole service of providing access to and/or use of the OC via a network (e.g. the Internet), whether it be via the world wide web, FTP, or any other method.

2. You may modify your copy or copies of the OpenContent or any portion of it, thus forming works based on the Content, and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified content to carry prominent notices stating that you changed it, the exact nature and content of the changes, and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the OC or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License, unless otherwise permitted under applicable Fair Use law.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the OC, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the OC, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Exceptions are made to this requirement to release modified works free of charge under this license only in compliance with Fair Use law where applicable.

3. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, distribute or modify the OC. These actions are prohibited by law if you do not accept this License. Therefore, by distributing or translating the OC, or by deriving works herefrom, you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or translating the OC.

NO WARRANTY

4. BECAUSE THE OPENCONTENT (OC) IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE OC, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE OC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE OF THE OC IS WITH YOU. SHOULD THE OC PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUME THE COST OF ALL NECESSARY REPAIR OR CORRECTION.

5. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MIRROR AND/OR REDISTRIBUTE THE OC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR

CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE OC, EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

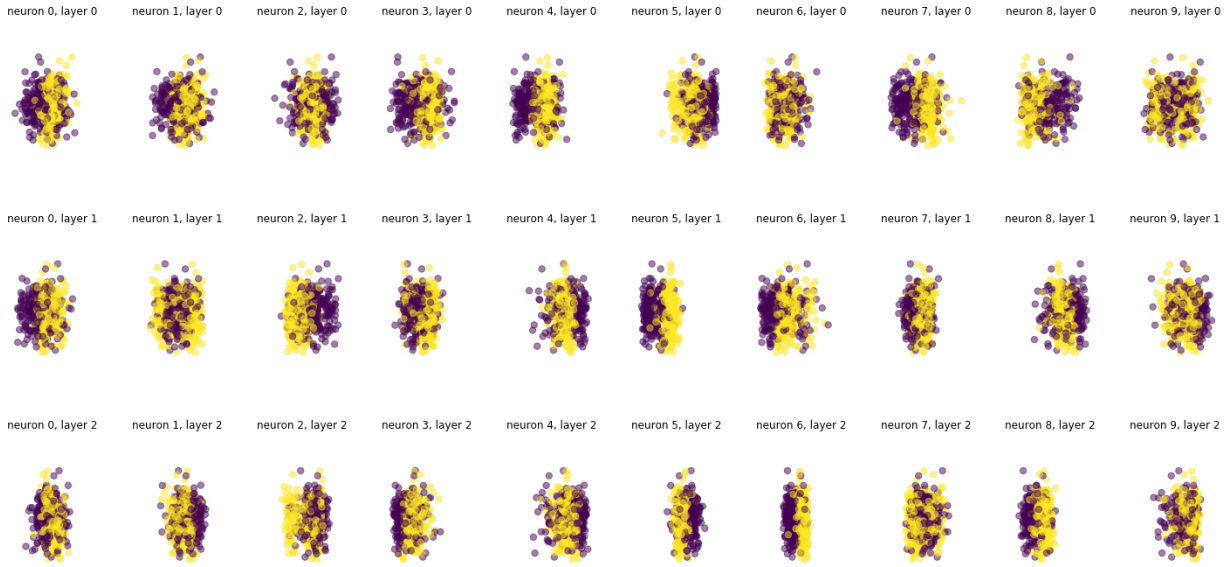
(This license is known as “OpenContent License (OPL)” and can be found at http://opencontent.org/opl.shtml)

Experiment 1.

Motivation: In this experiment on the example of Feed-Forward Neural Networks on classification task I'll try to demonstrate how much information a single neuron carries, and how neurons on the next layer "use" that information. Values neurons can take are limited depending on the activation function
used on that layer. For *sigmoid* it's *(0,1)* for *ReLU* (Rectified Linear Unit) it's *0* or *x*, where *x* is the value of the activation of that neuron, for *tanh* it's (-1,1). So naturally a question arises, what signal does an activation of -1 in tanh for example send to the neurons of the next layer? After all neurons exchange information to be able to make predictions, but how exactly? Depending on this we can also understand how these networks optimize, because optimization is nothing more then learning what signals to send to the next layer for a given input, so the rest of the network can successfully classify the input.

Description: In this experiment I visualize the activations of neurons during training. I use two datasets for this experiment. One is famous MNIST handwritten digits dataset, and the other is fake dataset of random numbers drawn from uniform distribution, and which have random labels as their ground truths. The letter dataset was created to prove that implications made in the experiment hold even for random data.
Above visualization comes from a Feed Forward Multilayer Perceptron with 3 hidden layers with 10 neurons each. Activations of hidden layers are visualized, with upper layer of 10 neurons being closer to the input, and lower layer being closer to the output layers of network.

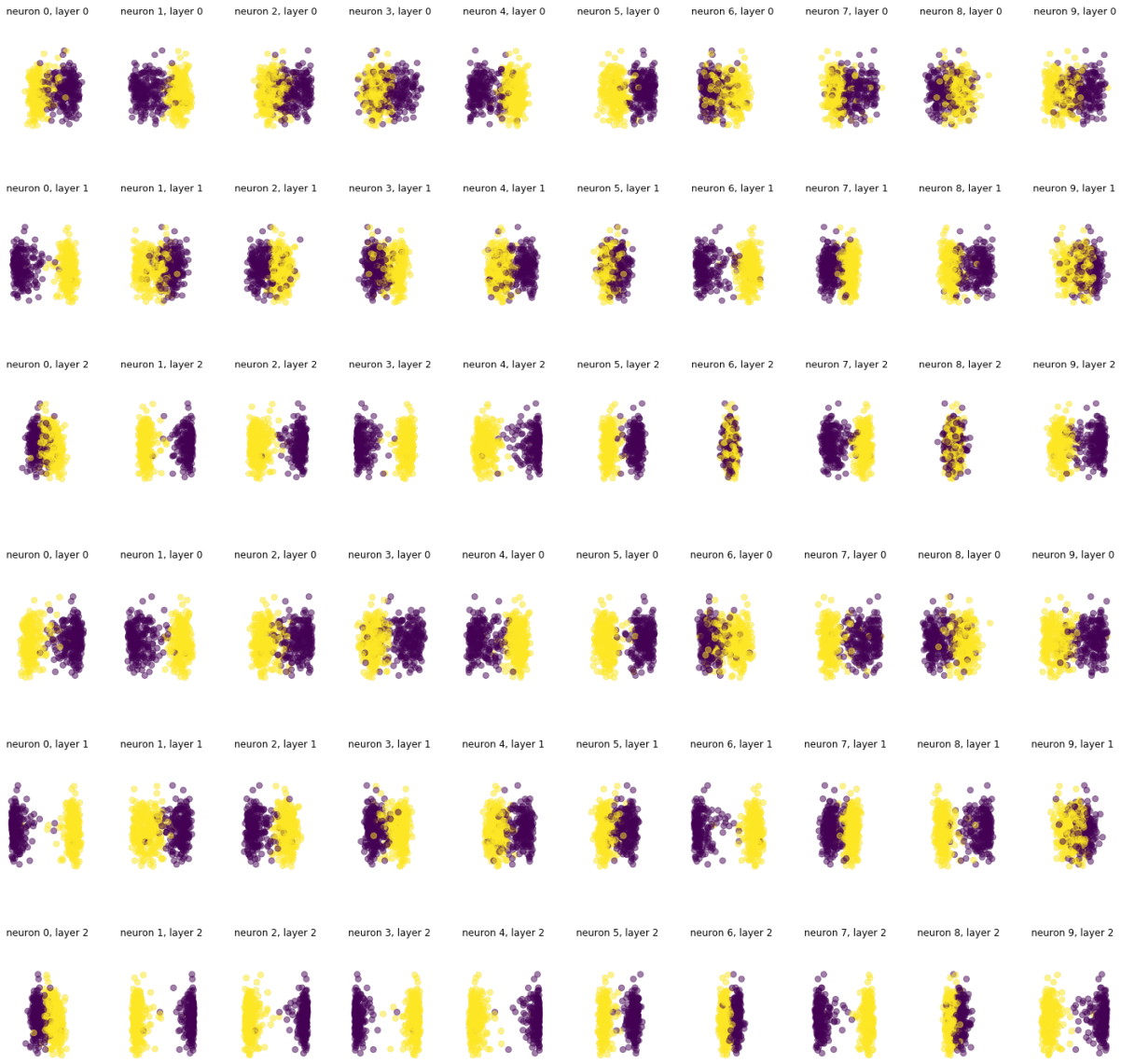
Epoch 1:

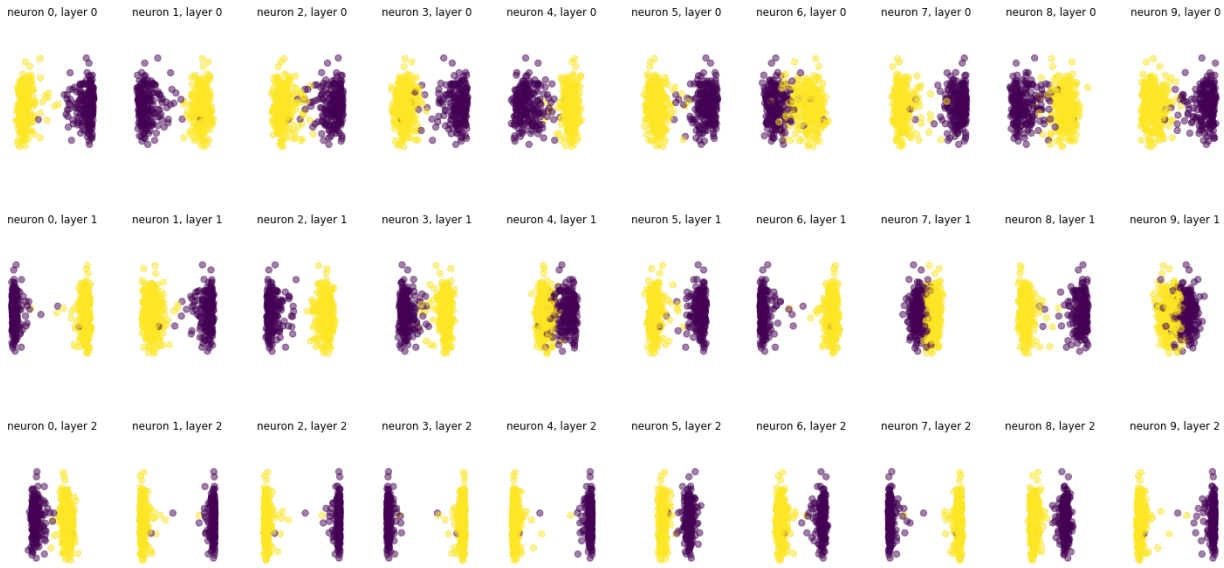


In the first plot a model trained to predict only 2 classes is used (zeros and ones). This is done for clarity purposes. Yellow ones are pictures of zeros and purple ones are pictures of ones. Each dot is a single picture. X coordinate represents the activation value of the neuron when the picture passed through it, and Y coordinate is auxiliary and doesn't mean anything. It is meant to help visualization of scalars (activation values) so that they don't clamp on each other.

Here we can see, that activations are random and overlapping for both classes, as this activations are only influenced by the initial weight initializations. But over time the picture changes. Look what happened in next couple of epochs.

Epoch 3, 5, 15:





Here you can see that activations for two classes start to separate, and in a random directions, for some neurons class of zeros tends towards -1 end of tanh activation and class of ones to 1 (first neuron for example), and for the others it's the other way around (second neuron for example). One more thing to notice is that the third hidden layer activations diverged more than the second layers activations and first layers. This is due to vanishing gradient, as less and less gradient reaches the lower layers of neural networks and forces to separate. This is beneficial, as lower layers have more variance this way, and can capture more diverse poses and appearances. This is one of the reasons additionally layers benefit the performance of neural networks.

The above observation, that neural networks try to separate class activations for different classes, holds for any architecture, depth, width (number of neurons in each layer) and activation function as we verified for all above.

To be sure this is not an anomaly of MNIST dataset, I applied the same visualization for a fake dataset of random numbers drawn from uniform distribution, and obtained the same result.

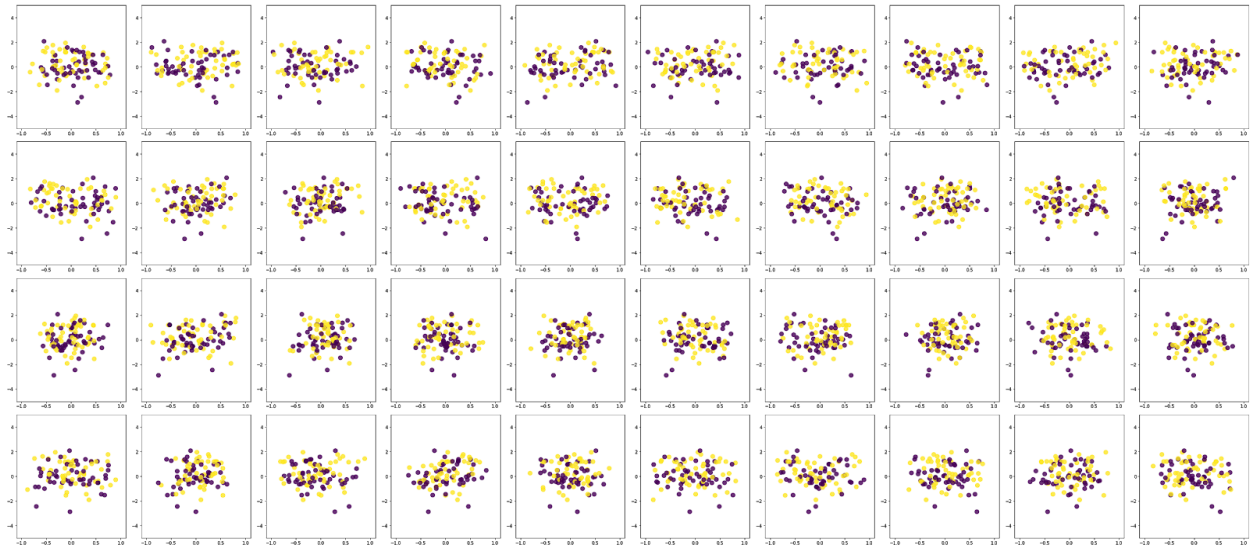
Data has a shape of (100,100) with 10,000 random numbers and 200 random labels (0s and 1s). A five hidden layer neural network was chosen for this data set. Size increase is due to the fact we want to fit a random data, and we need bigger network to overfit to it. Side note, increasing the number of columns at the expense of rows makes it easier for the network to overfit to our "data" as it now has more dimensions to find non existent patterns at.

Although it takes much more epochs, but at the end the same result is obtained, class activations diverge to two different sides of tanh activation (-1 and 1).

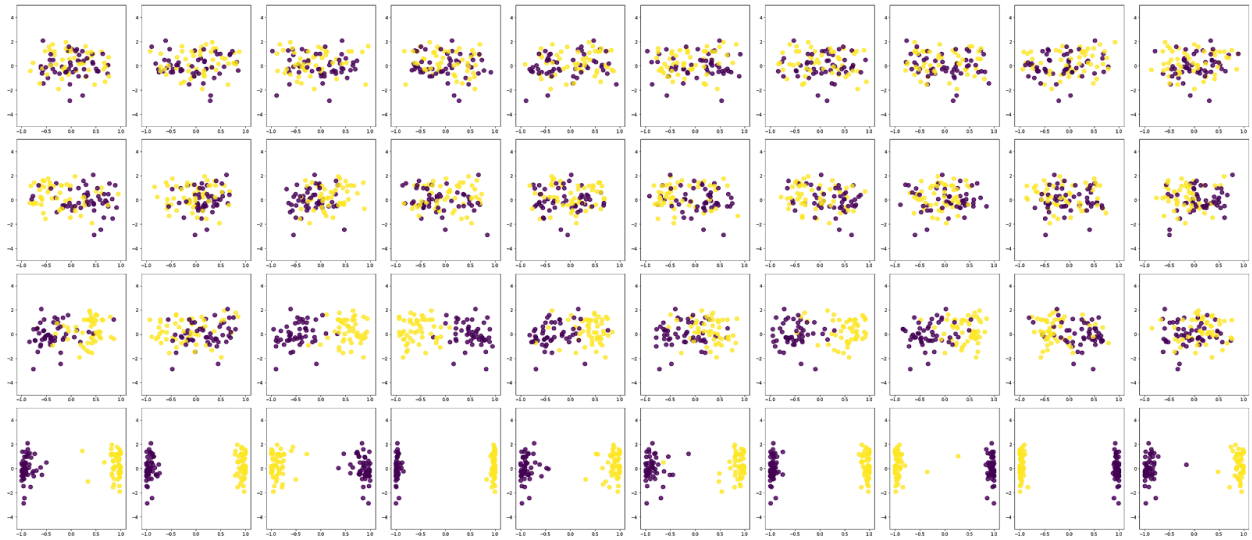
Above are the same visualizations for random data

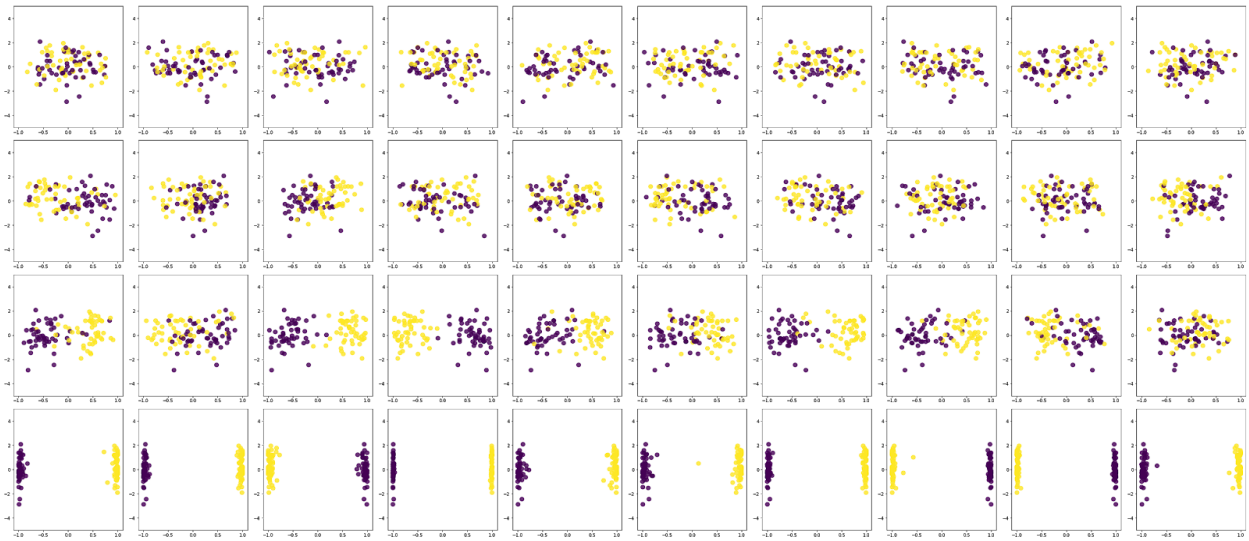
On random data

Epochs 1, 100, 200:



-



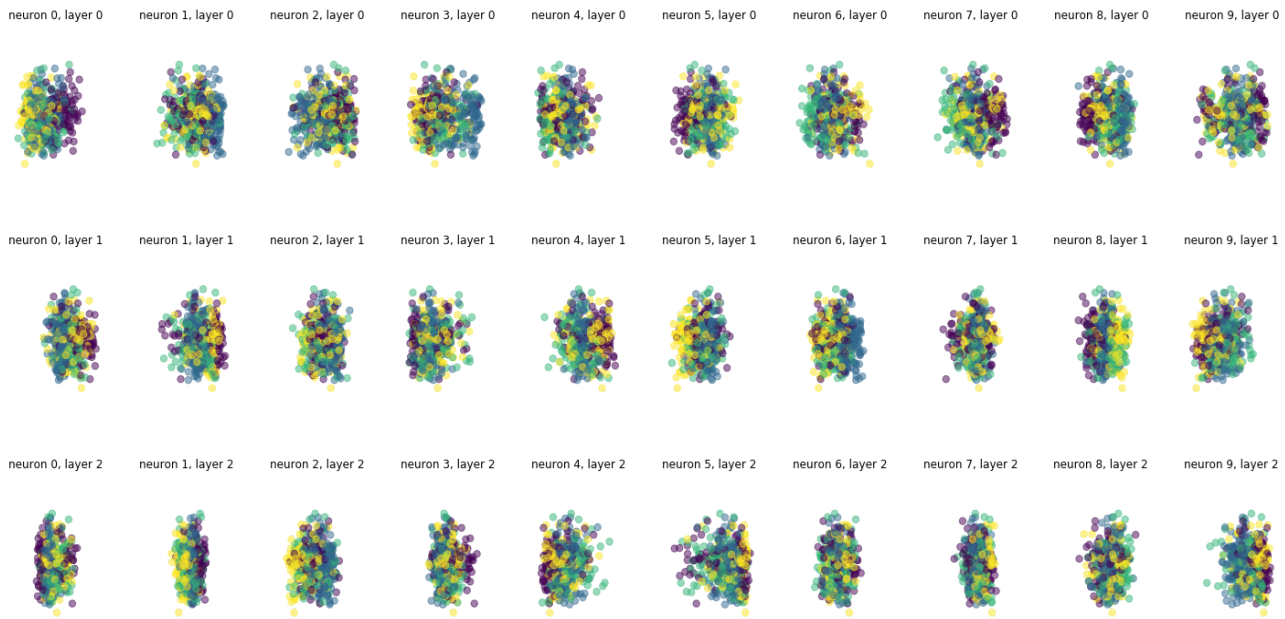


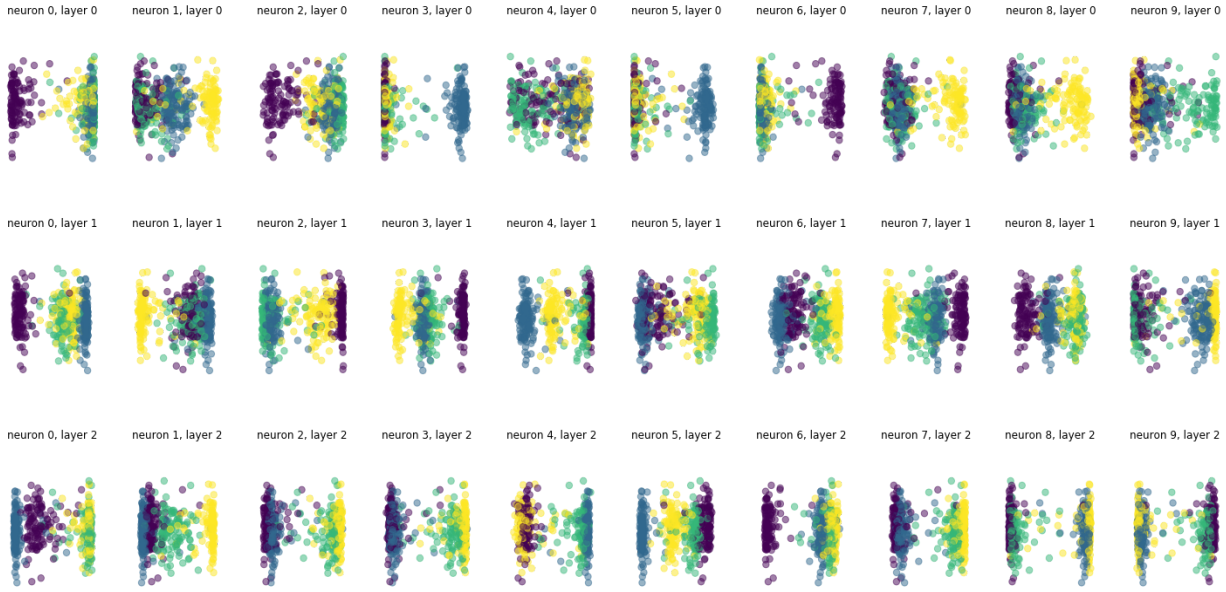
As with MNIST dataset, class activations diverge over epochs, at the beginning starting with random activations around 0 and slowly diverging to -1 and 1.

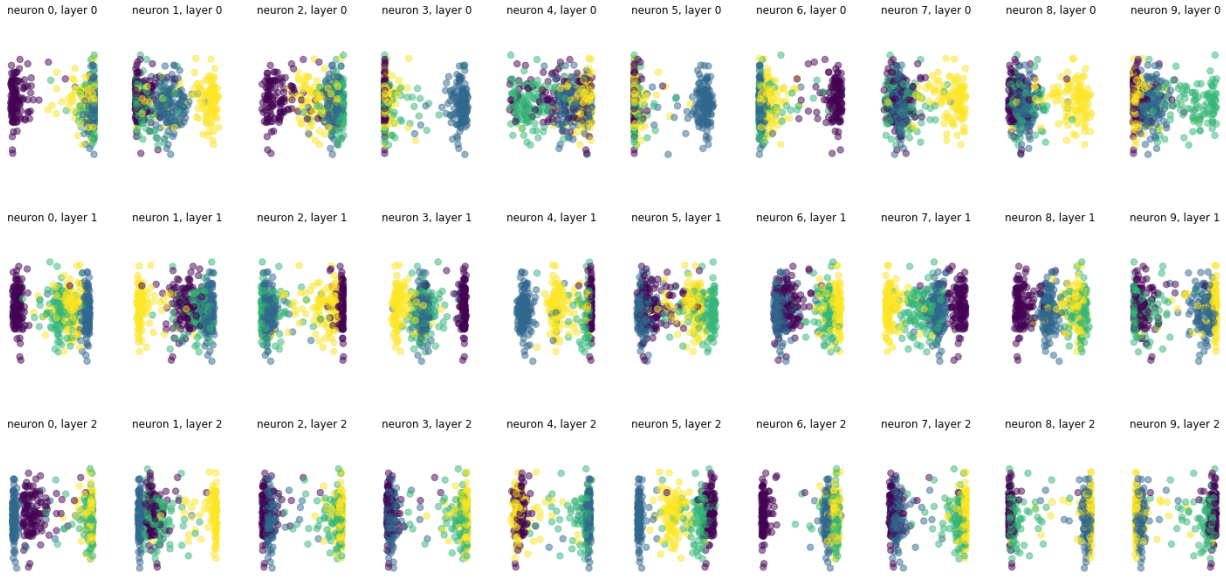
With more than two classes

Question arises, what will happen if we have more than 2 classes? The same will happen, but now half of the classes will tend to one side of the activation and the rest to the other side. Let's do this for 4 classes. Ten classes would be hard to see visually, that's why I restrain with only 4.
Above are the corresponding plots for epochs 1 (not trained yet), 20, 40, 60

Epochs 1, 100, 200:



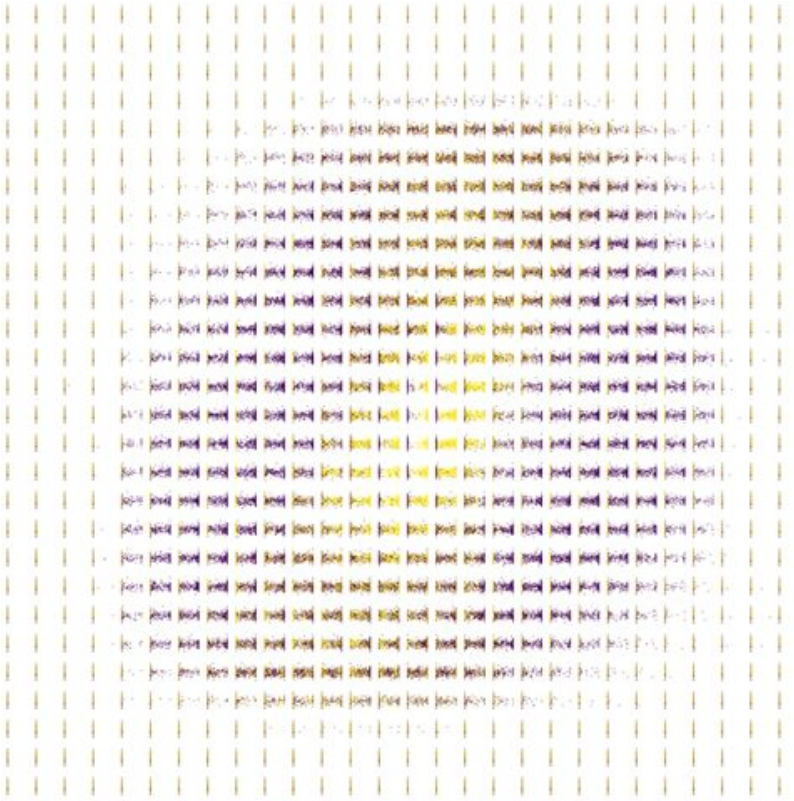




Same phenomenon can be observed, but now more then one class activations tend to one side.
General idea can be expressed as follows: when learning, each individual neuron in neural networks tries to split class activations into two, throwing activations of one set of classes on one side of activation , and the rest on the other. For tanh those sidess are -1 and 1, for sigmoid 0 and 1, for ReLU 0 and x. Note that there are 2^num_class - 2

possible ways to split class activations so that half will fall on one side and the other half on the other.

Explanation: To understand why Neural Nets behave like it's useful to think how much information can one neuron express. Let's consider a neuron with tanh (Hyperbolic function) activation. Its an activation function that takes values in range (-1,1) where tanh(0) = 0, and it monotonically grows towards bigger positive values and asymptotically approaches 1, and decreases approaching -1 for negatives. Because one neuron's activation is only one scalar, and because as we've seen above, neurons split classes into two subgroups and push the activation values of one group of classes to one side of activation function, and the other half of classes to the other side, activation value of a neuron is an evidence that the input to the network belongs to one of the classes that activate on that side of tanh. For example if one neuron "splits" class activations so that classes $(c_1, c_2, \ldots, c_5)$ activate closer to 1 and classes $(c_6, c_7, \ldots, c_{10})$ activate closer to -1, then for a given input, if the neuron activated with a value 0.98 for example, it indicates that most likely the input belongs to one of the classes $(c_1, c_2, \ldots, c_5)$. Because different neurons "split" classes differently, activations of intermediate neuron will differ, but a given activation of each neuron at a given layer is an evidence that the input to image belongs to one of the classes that activate like that, and because of this different splittings (and the number of neurons), next layers get enough information to process it further and make component conclusions about the class of the input. To illustrate the point let's look at the pixel values of MNIST digits, but only images of 0s and 1s.

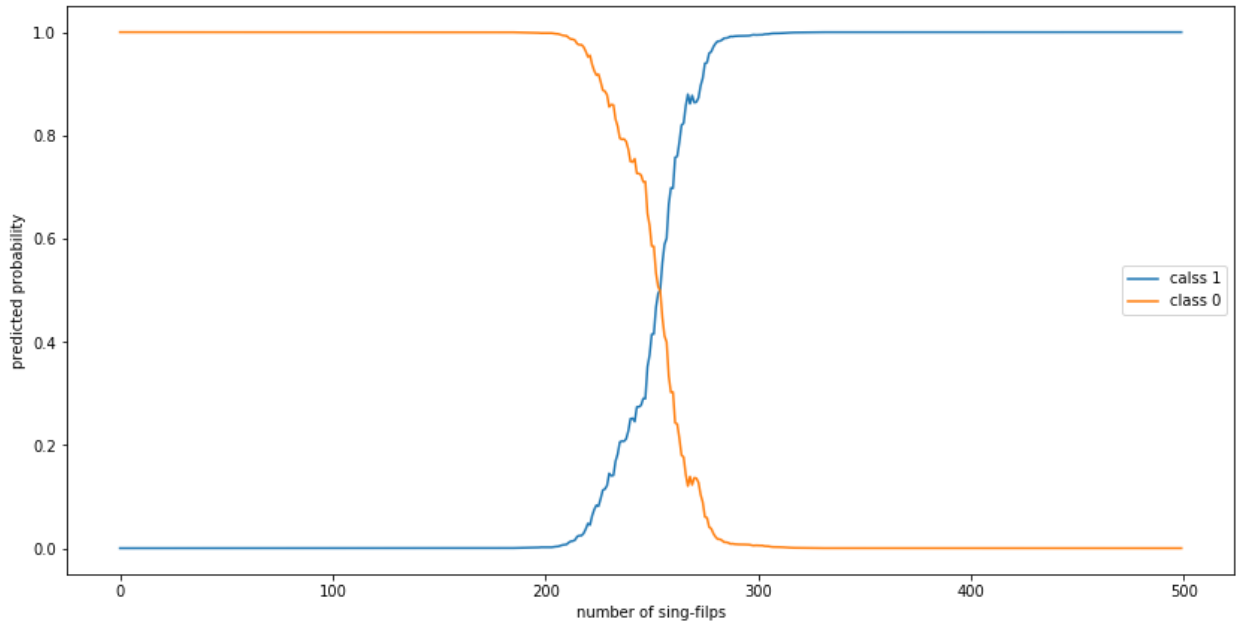


The plot is constructed similarly to the plot above, but now instead of hidden unit activations, each individual pixel is represented as a separate plot (or you can think it's a separate hidden unit on it's own, like from previous plots). Pixels on the corners have value of 0, so they're clamped to the left, but central pixels are active.
One can easily outline a zero in purple and a one in yellow in this image. This is because some pixels in data are only active in the pictures of zeros and the others only in pictures of ones. So the first hidden layer picks up on this class dependent differences in pixel values, and quickly learns to predict zero when one set of pixels is active, and one when another set of pixels are. Hidden layers work the same way. In fact, as far as the neural network is concerned, input layer is another hidden layer for it that it takes as an input, calculates the dot product with its weights, and passes to the next hidden layer. And pixel values in this analogy can be thought as activation values of "the first hidden layer" (the input layer).
Neural networks this way try to make meaningful conclusions about the class of the image by its activation value. Splitting class activationsIt's built in them through Backpropagation algorithm too. When a training sample makes a forward passes through the network, it produces some output on the last layer that would hopefully be as close to the ground-truth one-hot vector as possible. For example in case of MNIST, if we pass forward a picture of a digit through a trained network, we want to see as output a one hot vector of length 10 where all but the position corresponding to our picture class contain 0s, and at the corresponding class position we have a 1. But if final result differs from this ideal scenario (practically it always does), an error is calculated like this $(y - \hat{y})$,where y is the ground truth label and $\hat{y}$ is the prediction of the network, and Backpropagation algorithm adjusts the weights. Network does so by tweaking the weights to better match the output of the previous layer, so it makes a small move towards the direction of the steepest descent of the loss.
A useful way to think about this is to imagine that network creates a pattern of activation on each layer for each class, and these patterns must be different so that network doesn't confuse two classes. When activation values have a certain pattern, next layers of the network know the class of the input. So Gradient Descent algorithm adjusts the weights of a given layer to match that pattern of activation on the previous layer, in a direction depending on the way a given neuron splitted class activations.
Important to note, that if a given neuron does not split class activations effectively (for inputs from a fixed class, activation value jumps all around 0) and doesn't tend to either side, Backpropagation algorithm, trying to adjust for class activation, will push the weight of that neuron to 0, and the neron will become as what's known as a "dead neuron". That neuron will no longer have any significant influence over the network predictions.
Also important to note, that the variance of the activation pattern decreases over time, as it's built in the architecture, with the first layer variance being the result of differences one image of a given class can have with another one. On the last layer we want no variance in activation values for images of the same class, as we want the network to always predict the same when given an image of the same class. So over hidden layers, the variance decreases.

Experiment 2.

Motivation: Let's confirm the idea that for neural network, class is just a pattern of activation that decreases its variance over layers closer to the output.

Description: In the following experiment I train a network on MNIST dataset with 3 hidden layer of 500 neurons each. Then throw away all but last layer. If our assumption was correct, we can calculate the activation pattern for one class at the last hidden layer, and when that pattern is given to the last layer of the network as the activation of its previous layer, it will predict with high confidence the class. Also, if we change the pattern more and more, the prediction confidence must decrease. For this experiment I choose to use only two classes form MNIST, calculate activation for one class, and then change that activation more and more. As the activation function used to train the model is tanh (activates between -1 and 1), then changing the pattern of activation means simply multiplying the activation value by -1. Pattern for class zero was extracted (by simply computing the activation value of the last hidden layer) and prediction was made giving that activation pattern to the last layer of the network. Then one-by-one the sign of activation values was filmed and predictions made.



Predicted class probabilities. X axis represents the number of sing flips, Y axis is the predicted probability of the class. Zeroes with yellow, ones with blue.

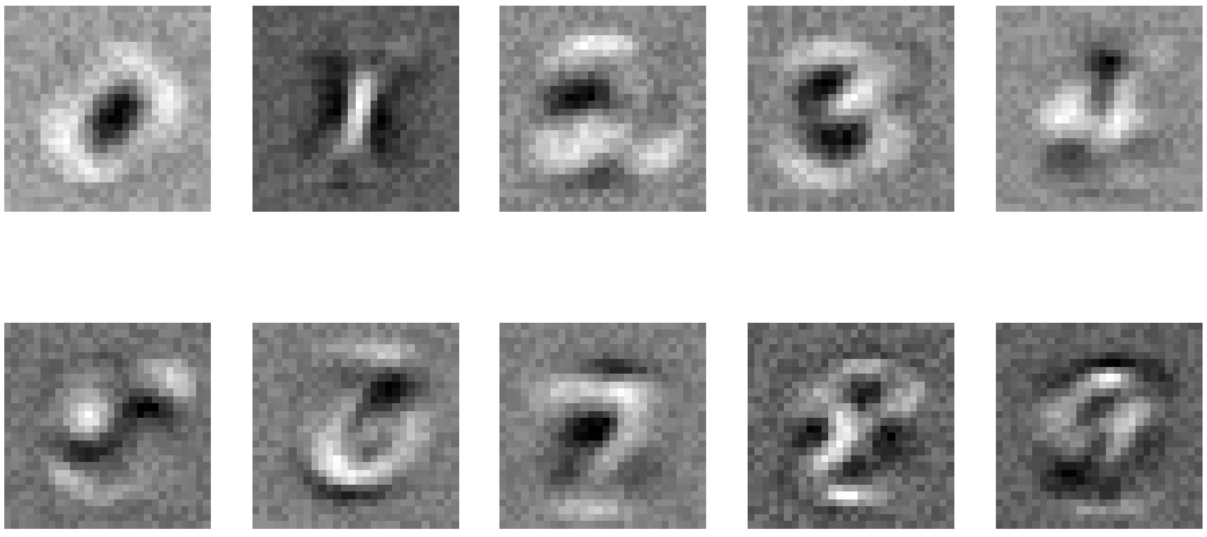
As we can see, network predicts the class to be zero with very high probability until around 200 out of 500 neuron activations are flipped, then it decreases reaching 0.5 probability (uncertainty level) when around half the activations are flipped.

Explanation: This is a nice illustration of the claims made earlier : that for the network a class is just an activation pattern. As the plot suggests, the decision boundary for belonging to one class or the other is at about 500/2 filed signes. So as we change the signs in from the "ideal" case for one class, neural network gradually starts to increase the probability of the other class. Network has some tolerance for small deviations too, because to be misclassified, image must activate less than half of neurons on the last hidden layer.

Experiment 3.

Motivation: A lot became clear about neural networks from previous paragraphs, but saying that neural networks using Backpropagation algorithm optimize their weights by converging to the local vector that splits class activation is not an intuitive explanation. In this and next experiment I try to give more and better visual clues to understand how exactly everything happens in a more intuitive manner.

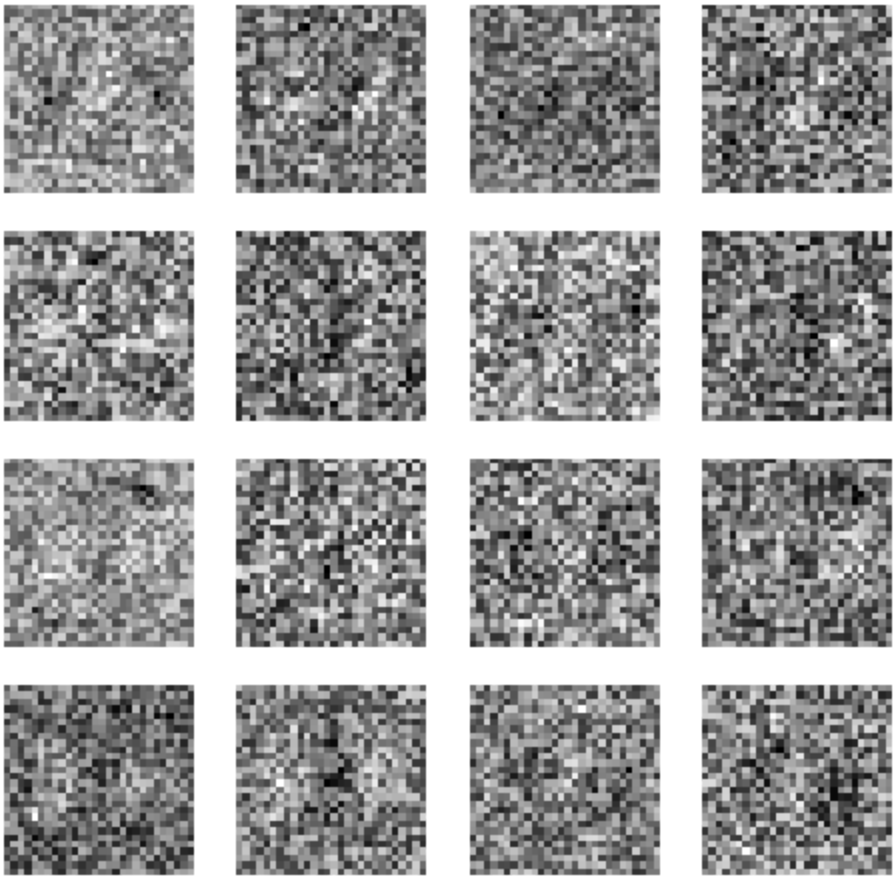
Description: It has long been known that when one applies logistic regression (which is in fact a one layer Feed Forward Neural Network - without hidden layer) to MNIST and then visualize weights corresponding to each class, the plots look roughly like a template visually very similar to the pictures in that class.



Visualization of weights logistic-regression learned on MNIST

One can clearly (or less so) see contours of digits, as if logistic regression learned a template for each class. But this change when we do the same with neural networks.

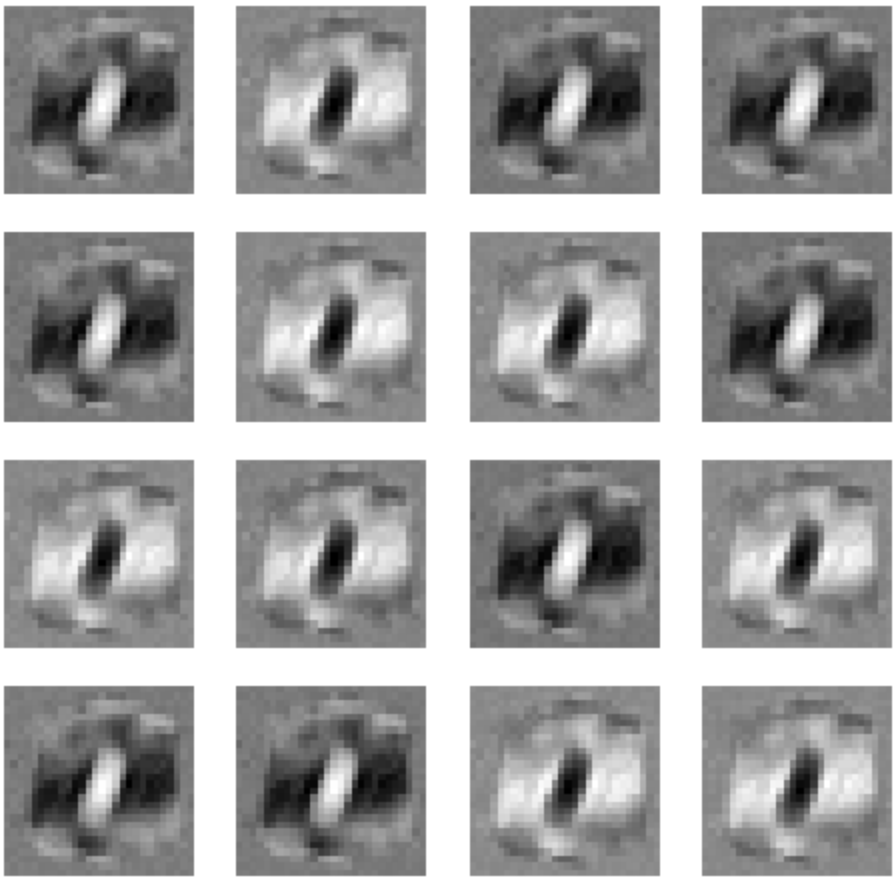
For this experiment I used a several different architectures with differing number of layers, activation functions and hidden units in each layer, and the results hold for all of those.
or clarity of visualization let's restrict ourselves with 2 classes (pictures of 0's and 1's), though this holds for any number of classes. After the model reaches maximum accuracy, meaning it’s properly trained, let’s plot the weights of the first layer which has 16 hidden units.



Visualization of weights of the first layer in neural network trained on MNIST dataset. First hidden layer, from where the weights were taken, has 16 hidden units.

Weights here look like a random noise, so maybe Feed Forward Neural Networks are doing something smart and incomprehensible, right? No, they don't. It's just that the pixels that are always inactive (<200 out of 784 pixels in all MNIST images have values different from 0) don't contribute to the loss and their weights don't get updated. On the other hand, little changes in the important weights get optimized with very little modifications, so little it's not visible. To prove this let's repeat the same experiment, but

make pixel values in images very small (by dividing pixel values by 10000 for example), so the weights have to adapt and get bigger (so that the corresponding dot product activations will get sufficiently far from 0 in case of tanh and 0.5 in case of sigmoid for example). The results will differ. Below is the same plot, but after dividing the data to 100000.



As you can see it just created a "joint template" for both classes. This looks very similar to the scatter plot of pixel values above. It's easy to notice that weight matrices differ by sign and (7 with one sign and 9 with another, roughly equal number).

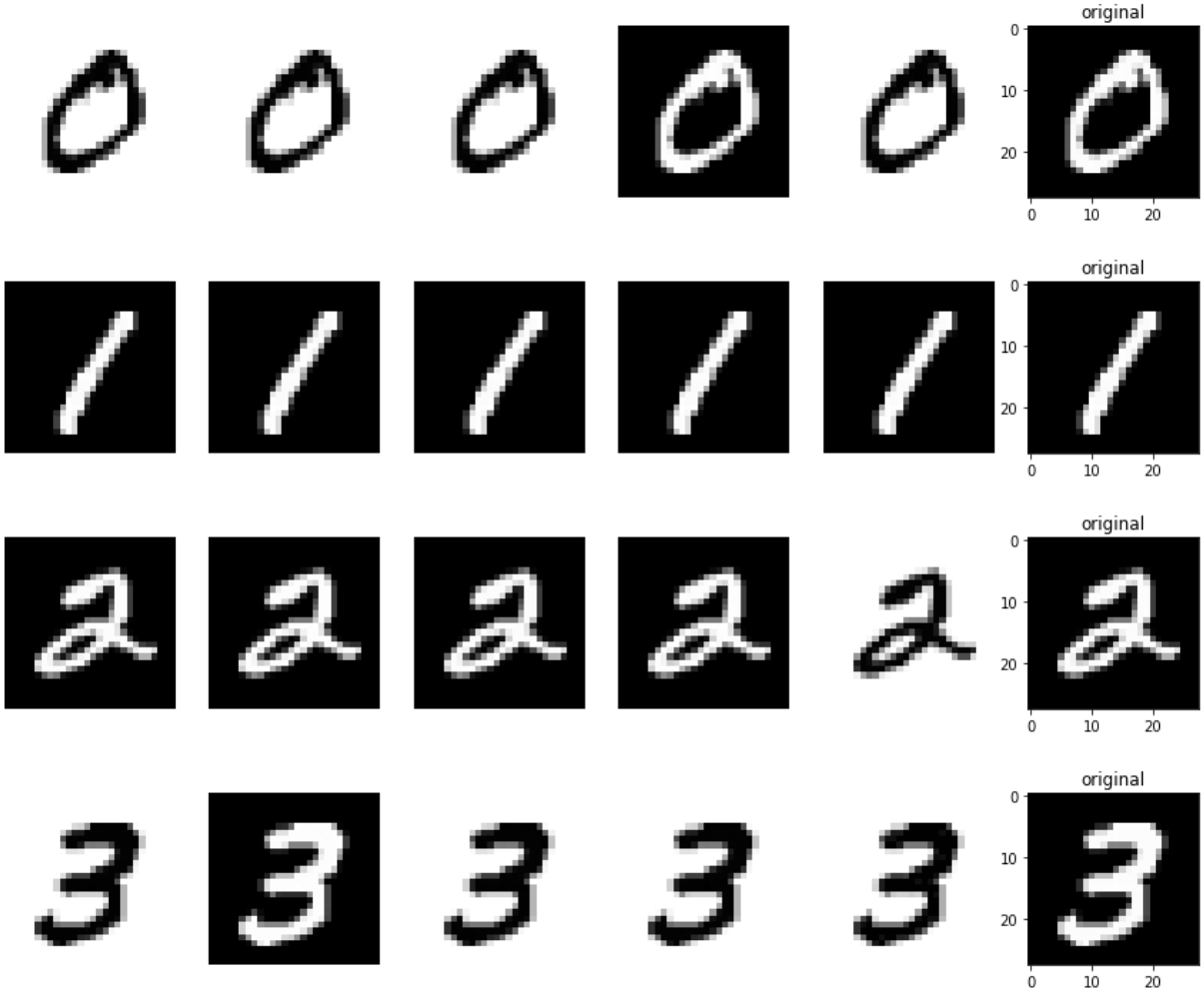
Explanation: What happened is that the network learned which pixels are most likely to belong to which class, and created a joint template for those, but as we've seen from *Experiment 1* network tries to separate class conditional activations, so in a given template, if the regions that belong to 0s are drawn with black (have negative values), then the regions belonging to 1s will always have the opposite color. The first template for example has the regions belonging to 0s with black - negative, and 1s with white - positive. This means that this neuron learned to split data so that pictures of 0s will activate closer to -1 and pictures of one closer to 1. This is a clearer way to see how

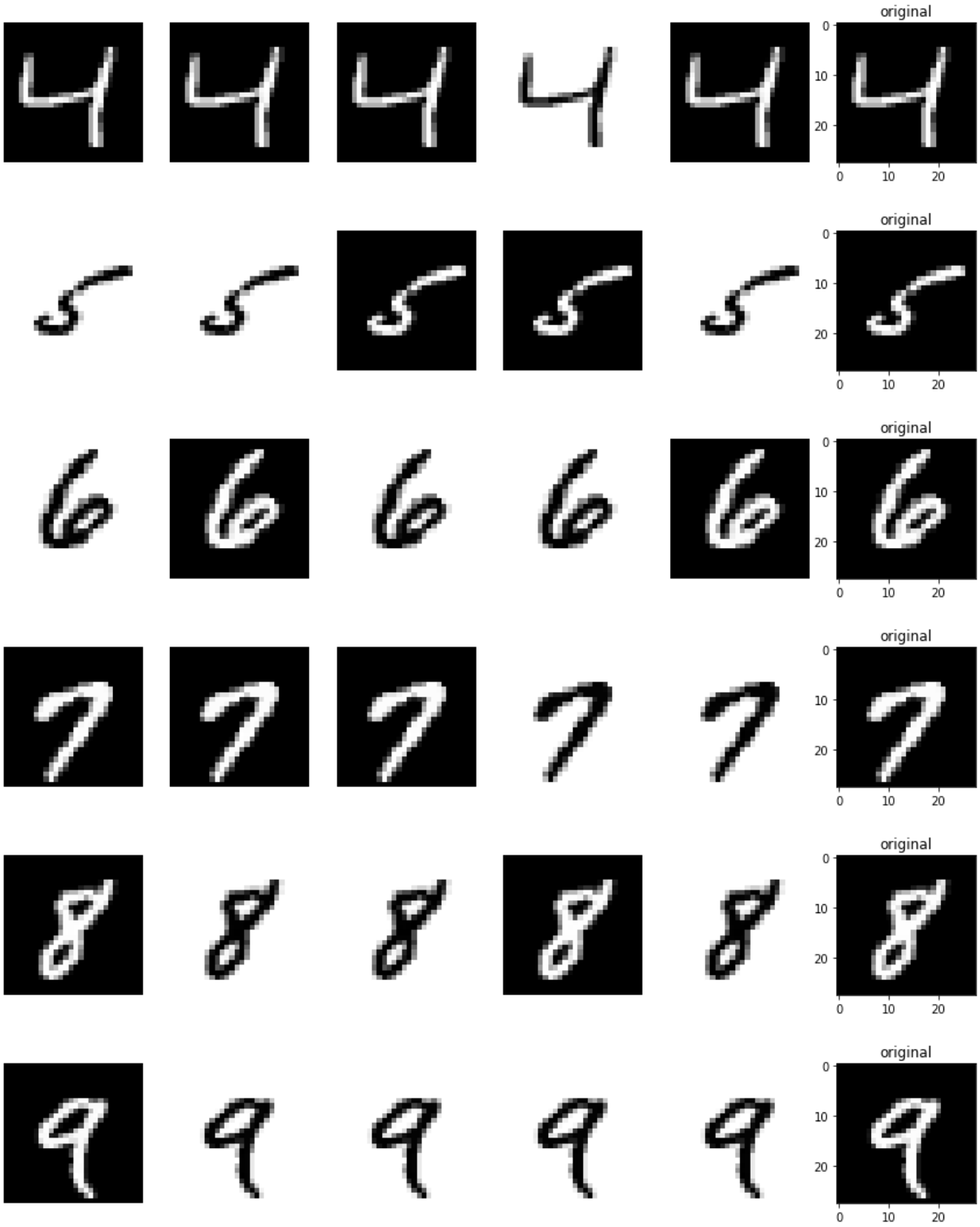
exactly neurons split class conditional activations. Important to notice the difference between templates created by the logistic regression and neural network. These two differ in a way that neural networks learn *joint* templates for all classes, which is one of the reasons they're so much more powerful than logistic regression.

Experiment 4.

Motivation: So how do these templates get constructed? Can we get even more detailed explanation?

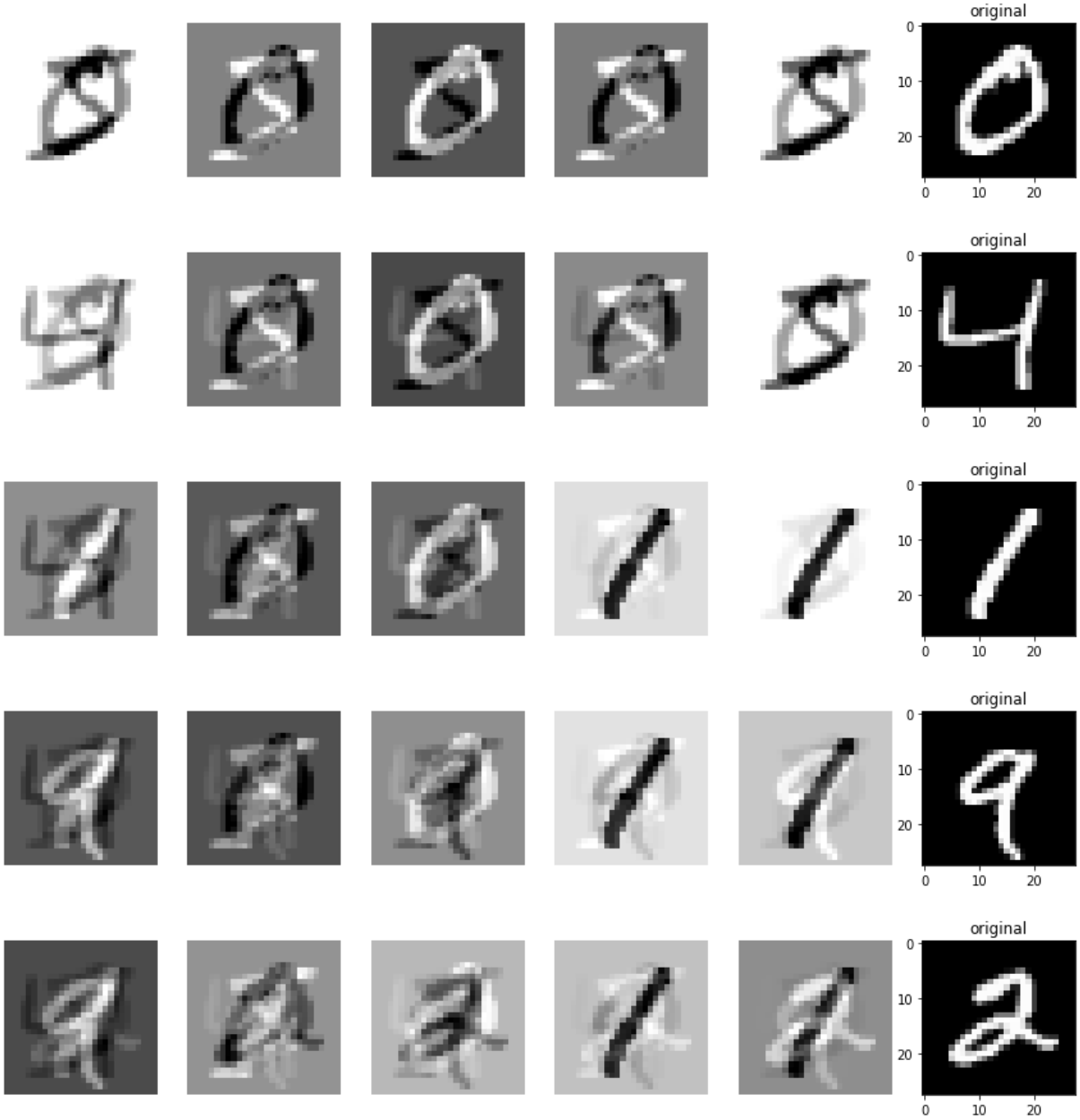
Description: Yes we can! And now we can visualise for all 10 classes! Experiment is fairly simple, it tries to show what is the network learning on each iteration. For that I train a network with only 1 image at a time, each epoch after forward and backward passes and when the weights are updated, I subtract currant weights form the weights at the previous iteration. This will give us exactly the gradient for 1 image. Below are visualizations of the gradient of the first hidden layer with 5 hidden units.





Visualization of the gradients calculated for images from MNIST

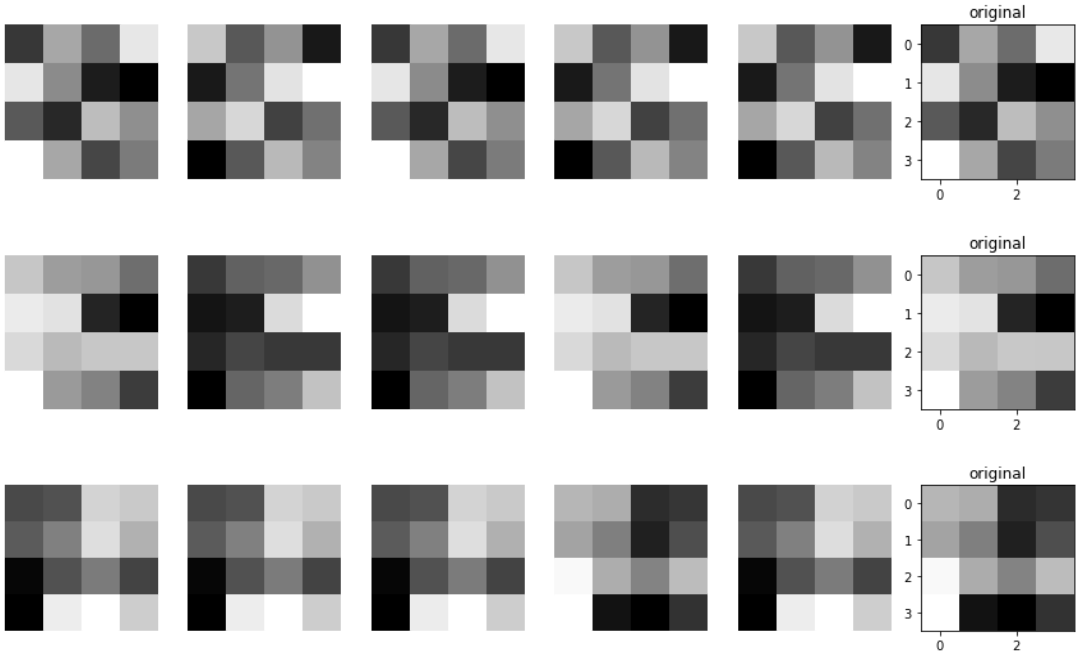
Explanation: These visualizations are not tampletes anymore, these are concrete images form the data set, and as we can see, the gradient replicates the image perfectly. Implications of this are huge, for one it means that potentially we don't have to wait for full forward and backward pass to compute the gradient and make weight updates, and can speed up the training by factors. Notice that for the same picture, gradients are either black on white or the opposite. This provides the effect discussed in experiments 1 and 3, that network learns how to split class conditional activations. Also this explains how we get the templates in experiment 3. Interesting to notice, that you can observe the effect of adding a momentum term. Then a single image persists for several iterations and slowly fades away.



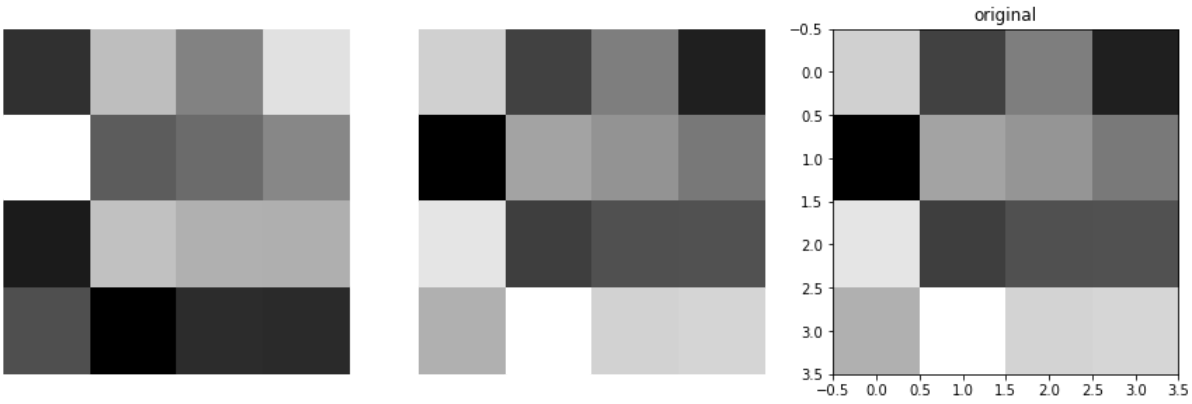
What about the intermediate hidden layers?

Question arises, does this phenomenon typical only to the first layer or all the layers? Answer is, as far as neural network is concerned, it doesn't differentiate between first and hidden layer, and for the network, the input values are like the activation value of its previous layer. This is important because this highlights the importance of normalizing the inputs, so it falls roughly at the range of the activation values of the activation function of the network.

Gradient of the hidden layers



First 5 are the gradients of 5 neurons, the last one is the activation vector. These 3 plots were constructed using for 3 different inputs.



A closer look. Notice that gradient values at corresponding positions match each other or are the opposite of each other by sign. This is in direct analogy with the first layer gradient visualizations.

As you can see, as with the example of the input images, the gradient repeats the pattern. This is as I said earlier, because as far as the network is concerned, our input is just like the activation values of its previous layer. So here the role of the input image plays the activation vector of the previous layer.

We can make an important generalization : each layer of the neural network, including the first one, changes its weights in the direction of or in the opposite direction of its activation vector, depending on the class of the input but consistent for a fixed class.

Results

This thesis showed that Feed Forward Neural Networks, unlike Convolutional Neural Networks, do NOT learn general features like edges and loops, but instead create joint templates of its inputs. Thesis also provided new approaches to visualizing understanding Neural Network, that could potentially lead to some interesting research. One important generalization was made : each layer of the neural network, including the first one, changes its weights in the direction of or in the opposite direction of its activation vector, depending on the class of the input but consistent for a fixed class. This means that neurons on all layers do the same thing, and don’t specialise in any clever way.

References

[1]
Methods for Interpreting and Understanding Deep Neural Networks : Grégoire Montavon
arXiv:1706.07979

[2]
Decoupled Neural Interfaces using Synthetic Gradients : Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, Koray Kavukcuoglu

arXiv:1608.05343

[3]
[LeCun et al., 1998a]
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.