IMAAGE
CLASIFIER
NETWORRK

Neural Networks, More than you wanted to Show

Exploring toy neural nets under node removal. Section 1.

Introduction

This post is a long and graph heavy exploration of a tiny toy neural network.

Suppose you have some very small neural network. For some inexplicable reason, you want to make it even smaller. Can we understand how the network behaviour changes when some nodes are deleted?

Training code

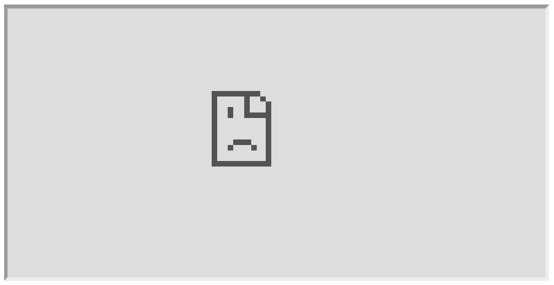
The network, and how it was trained.

```
            #This module trains a small neural network to determine if a point is inside or
outside a circle, and then saves that model to a file.
#First import some libraries.

import tensorflow as tf
import numpy as np
from math import *
import matplotlib.pyplot as plt
from itertools import *

#This function accepts n, and returns a generated dataset of n datapoints. Each individual
datapoint consists of 2 float inputs, and one bool output.
#Actually the bool is stored as the integers 0,1 because the tensorflow library was written
for the general case of catagorization into any number of catagories.
#The x input is uniformly sampled from the -1,1 square.
#The y output is 1 if the point is outside the circle centred on the origin with radius
sqrt(2/pi). This radius was chosen so exactly half the points
# (on average) will lie within the circle
def generate(n):
 x=np.random.uniform(-1,1,size=[n,2])
 y=(np.sum(x*x,1)>sqrt(2/pi)).astype(int)
 return x,y


#The number of hidden layer neurons
N=20

#The keras model accepts an input for x
inputs = tf.keras.Input(shape=(2,))
#In order to make removing some neurons easier, the network structure accepts a list of
on_nodes. These will be 1 for any node considered to be turned on, and 0 for any node
considered off.
#This value will be set to a constant block of ones during training. When the network is
evaluated it is usually on many neurons at once. The input here will generally be all 0's and
1's, despite this
#input accepting any floating point values. The input will also generally consist of
BATCH_SIZE repititions of the same vector. Ie within any one batch evaluation, the set of
neurons that are off is generally fixed.
#Another advantage of this setup is it lets us take gradients of the performance with respect
to these on_nodes.

on_nodes = tf.keras.Input(shape=(N,))

#A single hidden layer containing 20 relu nodes.
hidden_layer = tf.keras.layers.Dense(N, activation=tf.nn.relu)(inputs)
```

```
#multiplying by on_nodes to turn off any nodes we might want off.
hidden_layer_picked=hidden_layer*on_nodes

#an output layer of 2 neurons, containing probabilities assigned to each possibility.
#this goes through a softmax.
outputs = tf.keras.layers.Dense(2, activation=tf.nn.softmax)(x_picked)

model = tf.keras.Model(inputs=[inputs, on_nodes], outputs=outputs)

#compiling the model. The choice of optimiser and loss was because these seemed like
standardish sensible choices, and are often used on bigger nets.
model.compile(
   optimizer=tf.keras.optimizers.Adam(0.01),
   loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
   metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

BATCH=1000

#a whole lot of 1. Guess what this is used for. Thats right on_nodes. All the nodes are on
during training.
ones=np.ones([BATCH,N])

x,y=generate(BATCH)

#why 20 epochs, not more or less. Fiddling about until it seemed to converge in practice,
thats why.
model.fit([x,ones],y,epochs=20)

#keras lets you save your models with a single API call. How convenient.
model.save('circle_test')
```

Video of interactive heatmap

Given that the model only has 2 inputs, we can plot the model to get a good idea of what it is doing. The network was only trained on inputs within the $[-1, 1]$ square, but out of distribution behaviour can be interesting, so it is visualized on the $[-2, 2]$ square. The circle which the network is trained to approximate is shown. In the upper plot, yellow represents the networks certainty that the point is inside the circle, and navy blue, as certainty that a point is outside the circle. The colour scheme shown was chosen by whoever chose the defaults on the matplotlip.pyplot.imshow library.



Key takeaways:

1. With all the nodes on, the network accurately approximates a circle
2. Network performance varies a lot depending on which neurons in particular are removed

3. Flipping one neuron can have a significant effect, but doesn't usually completely change the network behaviour.
4. The network generally performs better with more neurons.

Plotting the Loss

Lets analyse the loss (as measured by sparse categorical cross-entropy)

We can let each node exist independently with a probability p , and plot the probability density function of the resulting.

Generating the data

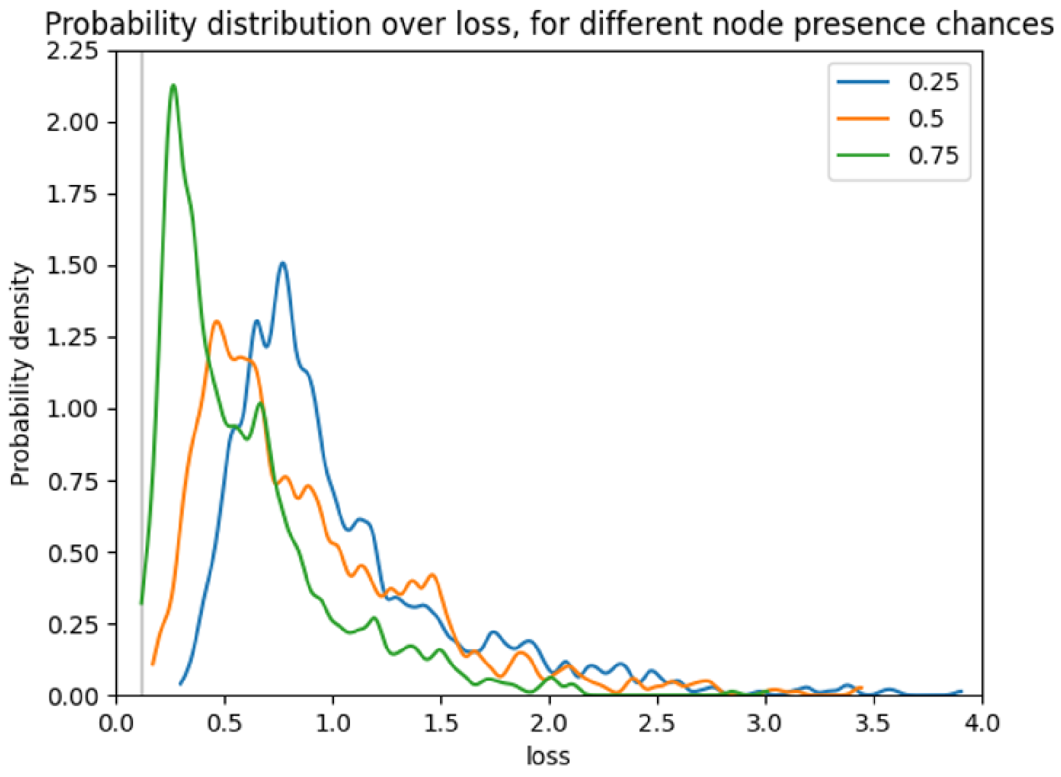
The following code runs the network with a random subset of the nodes removed, and saves the losses to file.

```
            import Network_loader_neat as net
import numpy as np
from random_utils_neat import *
import pickle
#20
N=net.N
#because 17 is the most random number. ;-)
np.random.seed(17)
data={}
for prob in [0.25,0.5,0.75]:
   result_list=[]
   for i in range(1000):
       on_nodes=(np.random.rand(N)<prob).astype(int)
       #uses a fixed random seed and so a fixed test dataset.
       #within the net.test function.
       #while also still allowing on_nodes to be different every time
       with RandomStateHolder(seed=23):
           loss,grad=net.test(on_nodes)
       result_list.append((loss,grad,on_nodes))
   data[prob]=result_list
   print("Finished calculating values for prob=%s"%prob)
with open("generated_data_1",mode="wb") as file:
   pickle.dump(data,file)
```

Probability density functions of loss

And then the result is loaded and plotted to create the plot below. Note that the lines are smoothed using a Gaussian kernel of width 0.03. This value is enough to make sure the graph isn't unreadably squiggly, but leaves the structure of the PDF.

Some small wiggles remain.



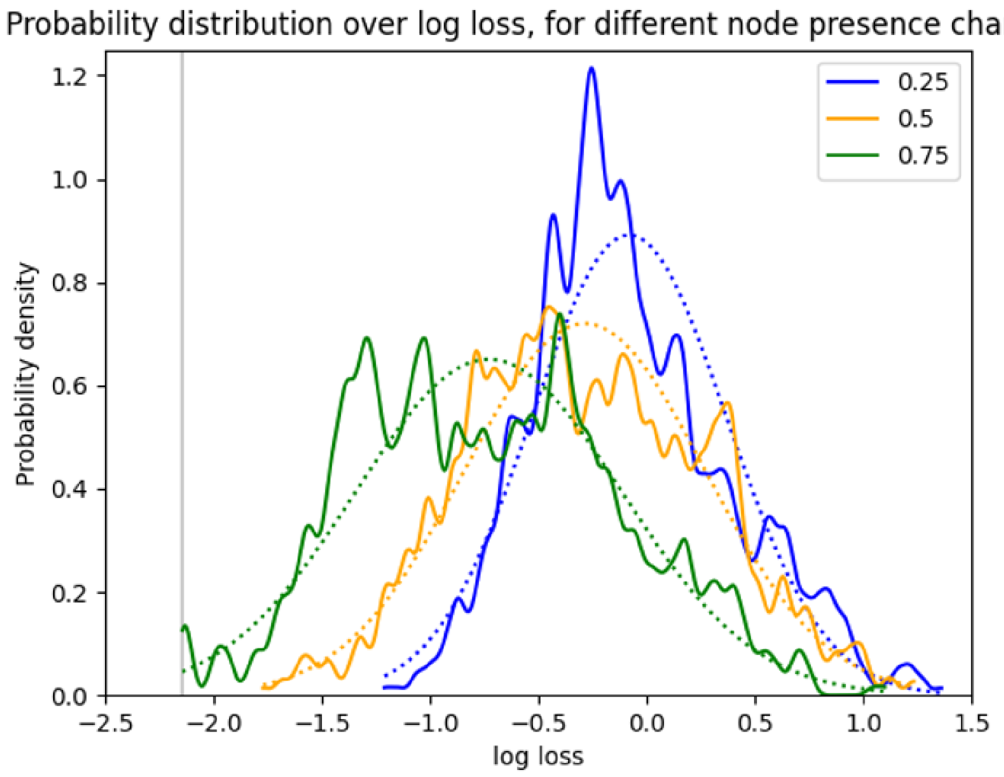
Prob	Mean loss
0.25	1.0277158
0.5	0.87821335
0.75	0.584365

Notes:

1. The small local wiggles are probably just random noise.
2. The grey line represents a loss of 0.117, the networks score with no neurons missing.
3. The order of the green, orange and blue lines show that networks with more nodes missing generally have higher loss.
4. The green line touches the grey line, this is partly because in a 1000 samples, 5 of them happen to contain all the nodes. Not that unexpected given $0.75^{20} \times 1000 = 3.17$.
5. If only the node 14 is missing, then the loss is 0.119, a barely noticeable difference. This occurs 3 times in the 75% on data.
6. Despite what you might think from the above, the data for p = 0.75 sampled 952 distinct points. (Points with an excess of 1's are unusually likely to be sampled repeatedly)
7. The data looks rightward skewed. This makes sense as there are more opportunities to do exceptionally badly than exceptionally well.

Probability density functions of log loss

To consider the hypothesis that the loss is log-normally distributed, lets see the same data, but with the logarithm of the loss.



If the data was perfectly log-normally distributed, the plot above would be a bell curve. The bell curves of best fit are shown for comparison. I would say that these curves look to be pretty close to bell curves, and the variation could well be written off as noise.

Here is a table of the means and standard deviations of those bell curves.

Prob	Mean log loss	STD log loss
0.25	-0.08020699	0.44782004
0.5	-0.28468585	0.5544732
0.75	-0.7286095	0.6133959

Even if the small deviations from the (dotted line) bell curves isn't noise, it looks like assuming a log-normal distribution is a reasonably good approximation, so I will be doing that going forward.

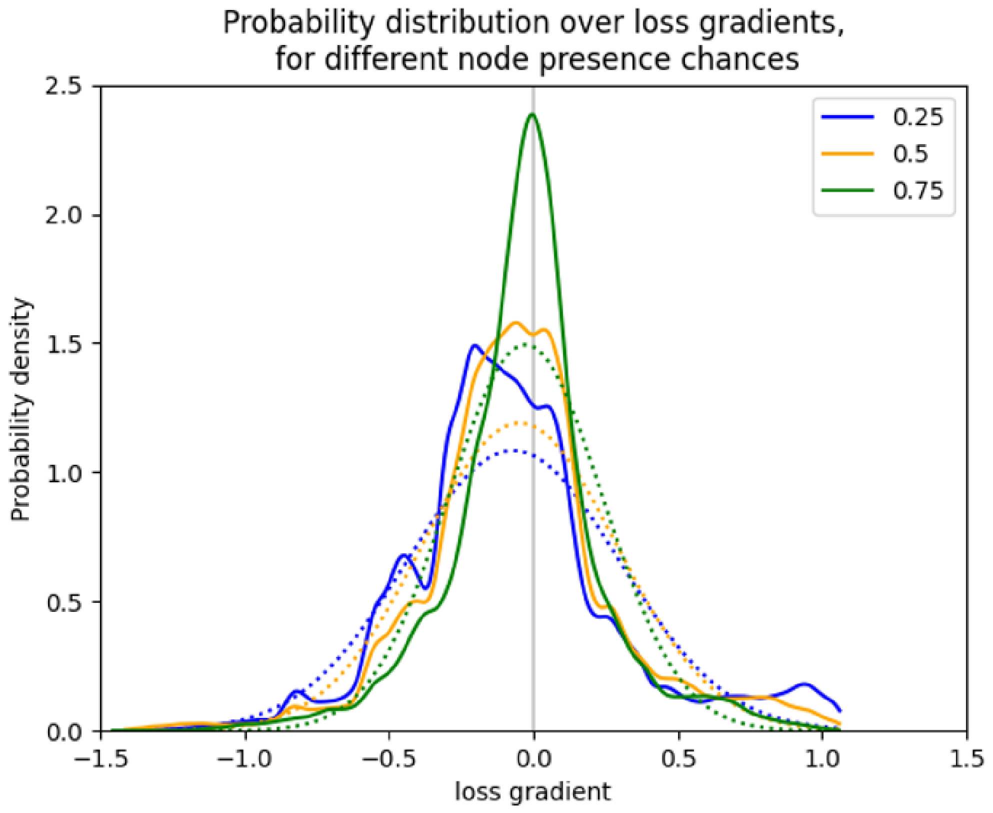
Gradients of the loss

These nodes are being turned on or off by multiplying each node by $b_1, .. b_n \in \{0, 1\}$.

The multiplication happens after the Relu activation function, not that this is any different to the multiplication happening before the activation function.

$$b_i \operatorname{relu}(x_0 w_0 + x_1 w_1) = \operatorname{relu}(b_i (x_0 w_0 + x_1 w_1))$$

Why? Well these gradients can be easily computed by back propagation, and they could give insight into the structure of the network.



We can see that the gradients tend to be more pointy topped and heavy tailed than their bell curves predict.

The table of means and standard deviations

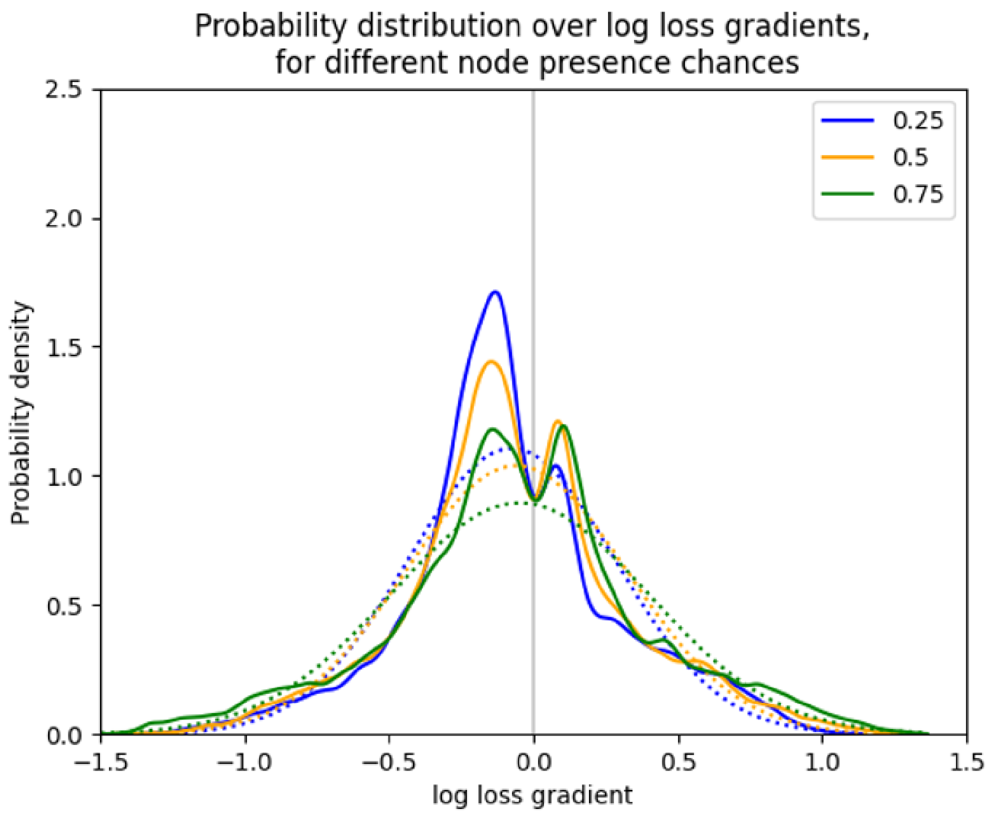
Prob	Mean loss gradient	STD loss gradient
0.25	-0.0693278	0.36804602
0.5	-0.04751453	0.33512998
0.75	-0.027095174	0.26713666

Note that the means are slightly negative, indicating that on average, the score is better with more nodes. On average, the 0.75 samples should contain about 10 nodes more than the 0.25 samples. Taking the mean loss gradient at 0.5, and scaling it by a factor of 10, we get -0.4751453. The mean loss at 0.75 minus the mean loss at 0.25 is 0.584365-1.0277158=-0.4433508. These numbers are reasonably close, which seems like a good sanity check.

In the previous section, we considered that log loss had a nicer distribution. What is the gradient of log loss doing?

$$\frac{\partial}{\partial k_i} \log l = \frac{\partial l}{\partial k_i} \frac{1}{l}$$

This means we can find $\frac{\partial loss}{\partial b_i}$ just by dividing the gradient of the loss by the loss.



Prob	Mean log loss gradient	STD log loss gradient
0.25	-0.0746937	0.36064485
0.5	-0.05837615	0.38386792
0.75	-0.046731997	0.44563505

The only really surprising thing about the above graph is the consistent trough at 0.

Is that node on

One hypothesis we might form is that we are observing the sum of 2 different distributions, each with a single peak.

Lets take the data for Prob=0.5. Each neuron is equally likely to be present or removed.

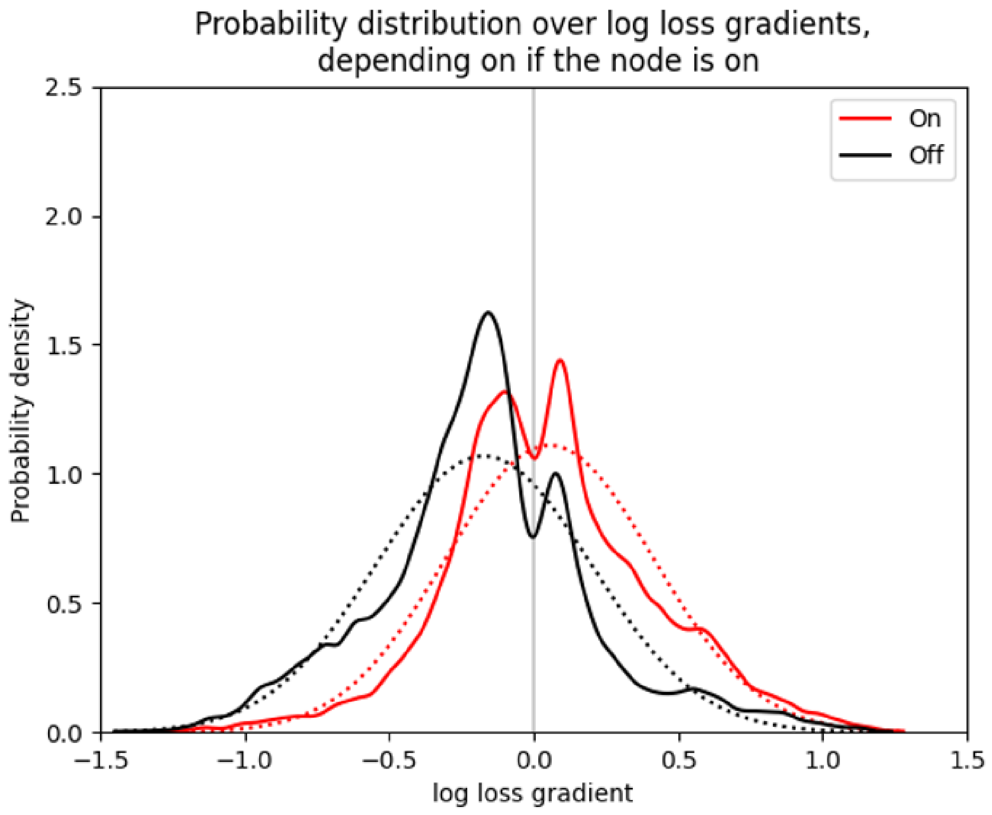
If f is a function that takes a list of 1's and 0's, representing the presence or absence of each neuron, and returns the log loss, then the i th component of the gradient is

$$\lim_{\epsilon \to 0} \frac{f(b_1, b_2, \ldots b_i + \epsilon, \ldots b_n) - f(b_1, b_2, \ldots b_i, \ldots}{\epsilon}$$

An obvious distinction to make here is if b_i is 0 or 1.

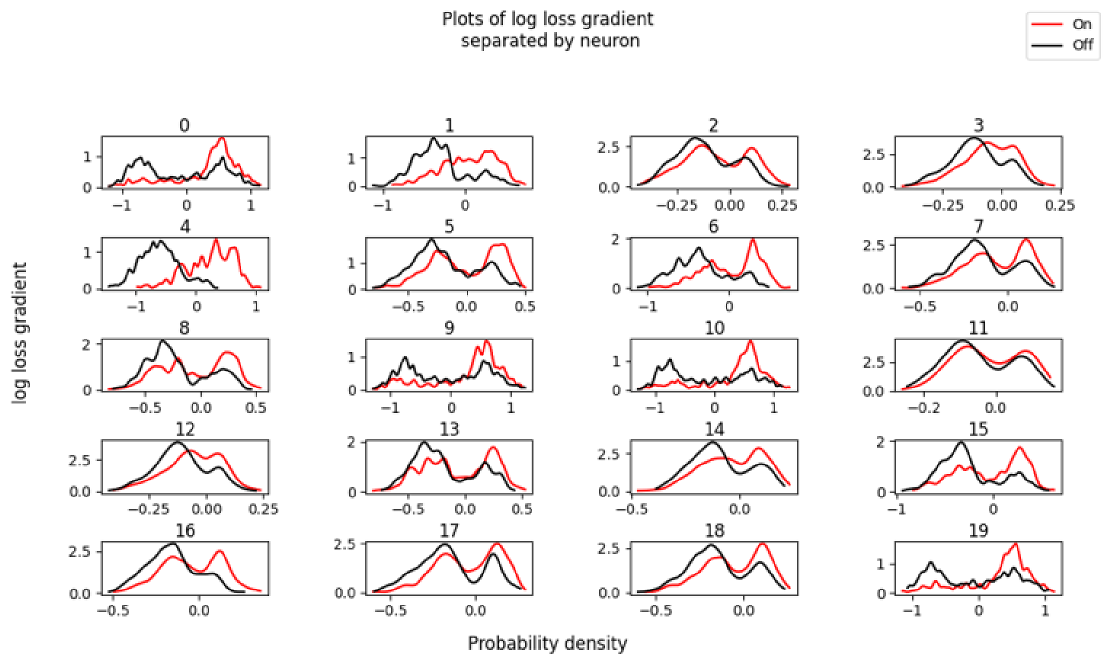
$b_i = 0$ corresponds to looking at a node operating at 0% (ie turned off) and asking if the network would do better if the node was at 1% instead.

$b_i = 1$ corresponds to looking at a node operating at 100% (ie turned on) and asking if the network would do better if the node was at 101% instead.



Node is	Mean log loss gradient	STD log loss gradient
On	0.055790696	0.35940725
Off	-0.17263436	0.37343097

Ok. That wasn't quite what I expected. Lets split this up by neuron and see how that affects the picture.

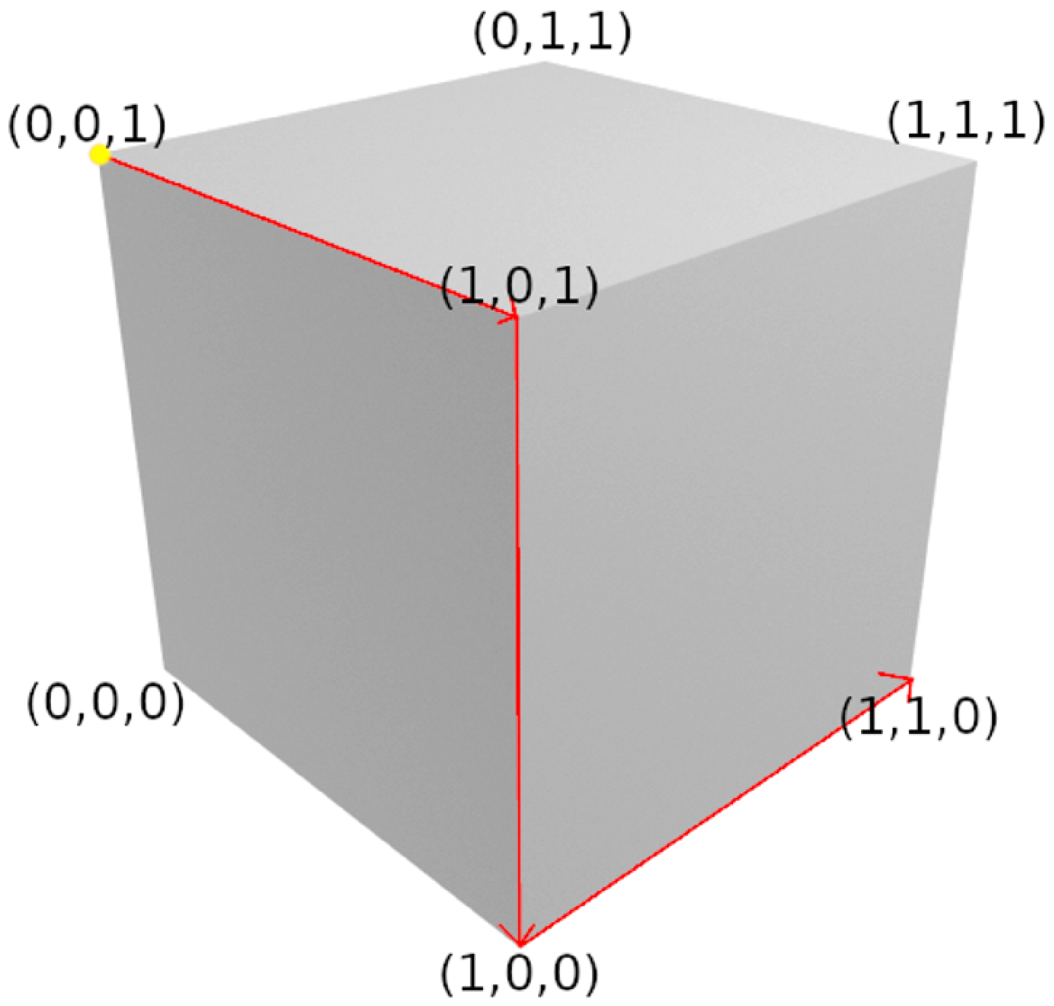


At a glance, some of these plots look smoother, and others look more jagged. For example, neuron 11 has smooth looking curves, and neuron 4 has more jagged curves. Actually, both curves are still being smoothed by a Gaussian kernel of width 0.03. Inspecting the axis, we see that the gradients on neuron 11 are actually much smaller. Some neurons just don't make as much of a difference.

How much does N flips matter

Consider taking a random starting position (independent Bernoulli, p = 0.5) and a random permutation of the neurons. Flip each neuron in the random order until every neuron that was on is now off, and every neuron that was off is now on.

We can visualize this process.



Here is a cube with edges labelled with coordinates. Each corner of the cube has a list of 1's and 0's which represents a way some nodes could be missing. (Of course, the network being visualized has 20 nodes, not just 3, so imagine this cube, but 20 dimensional.) We start by picking a random corner, and drawing the red line, which goes once in each direction.

There are N=20 red arrows, meaning 21 values for the loss at the endpoints.

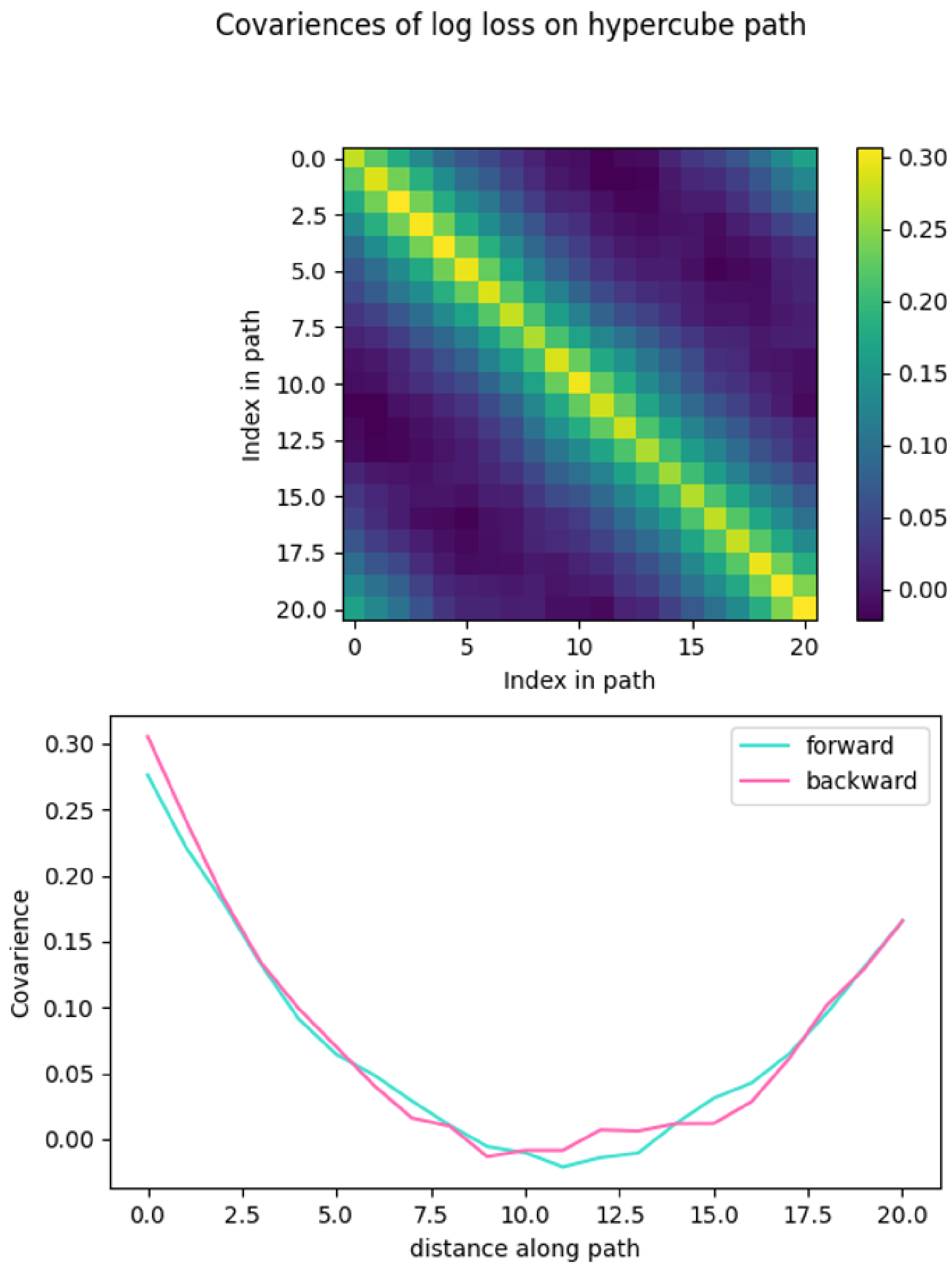
The distribution over paths is symmetric about reversal.

The heat map show the covariance matrix between log losses along the paths.

The turquoise line on the graph beneath shows the covariance between the log loss at start of the path, and the log loss along the path. (In distance from start).

The pink line shows the covariance of log loss at the end of the path, and log loss along the path, measured in distance from the end.

These should be identical under symmetry by path reversal. And indeed the lines look close enough that any difference can reasonably be attributed to sampling error.



This graph shows that it takes around 5 or 6 node-flip operations before the correlations decay into insignificance.

The surprising thing shown is the significant positive correlation between log loss of a network, and log loss of the reverse. This means if you take a pruned network that scores well, and reverse it, turning on all the nodes that were off, and turning off all the nodes that were on, the result is usually still a well scoring network.

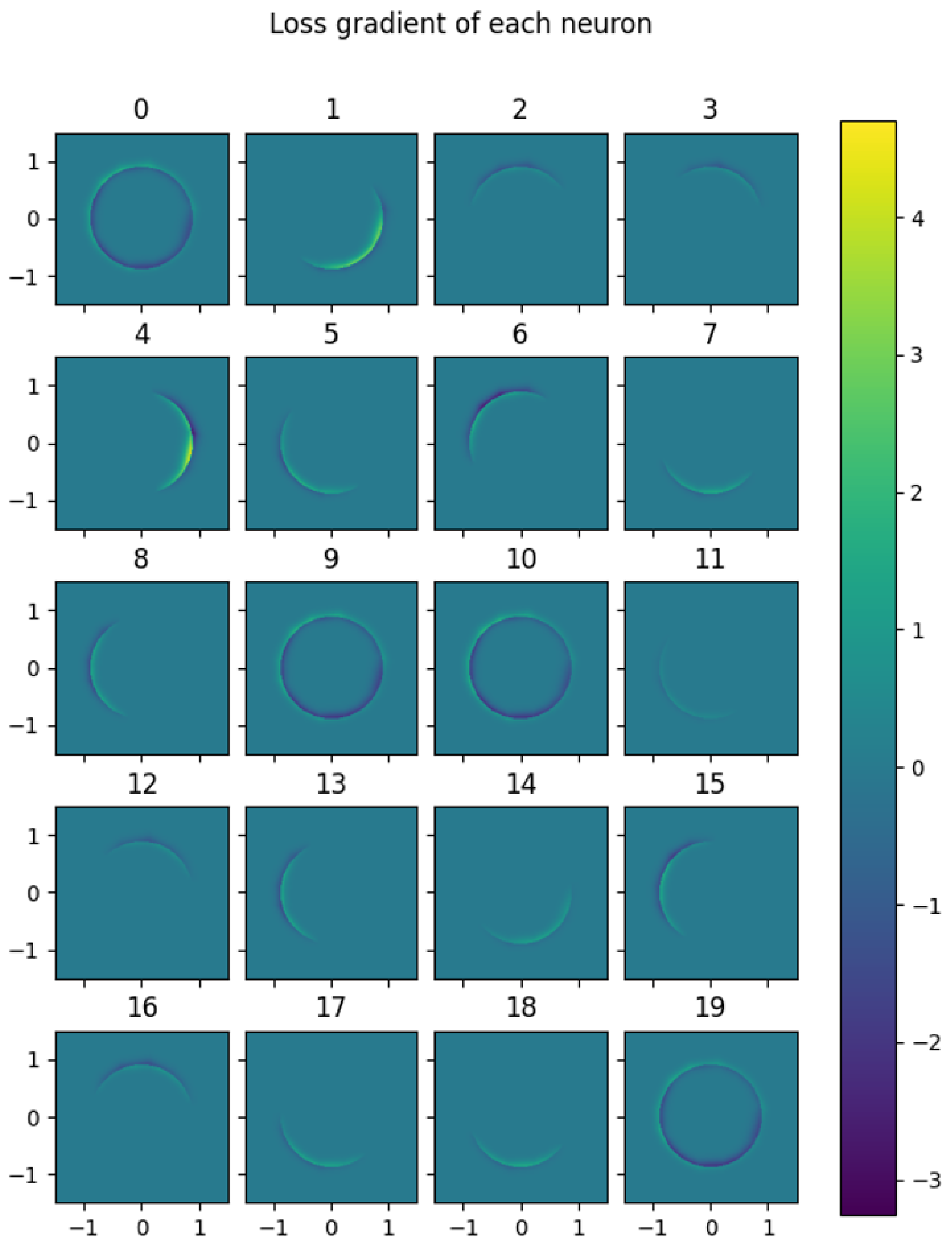
I suspect that a pruned network scores well when its nodes balance out, their being equally many nodes focussing on the top, bottom, left and right of the network.

As the full network is well balanced, this makes the nodes of the reversal also well balanced.

Heatmap of Gradient

So far, all considerations of the gradients are averaged over a test sample. But the gradient of the loss is well defined at every point.

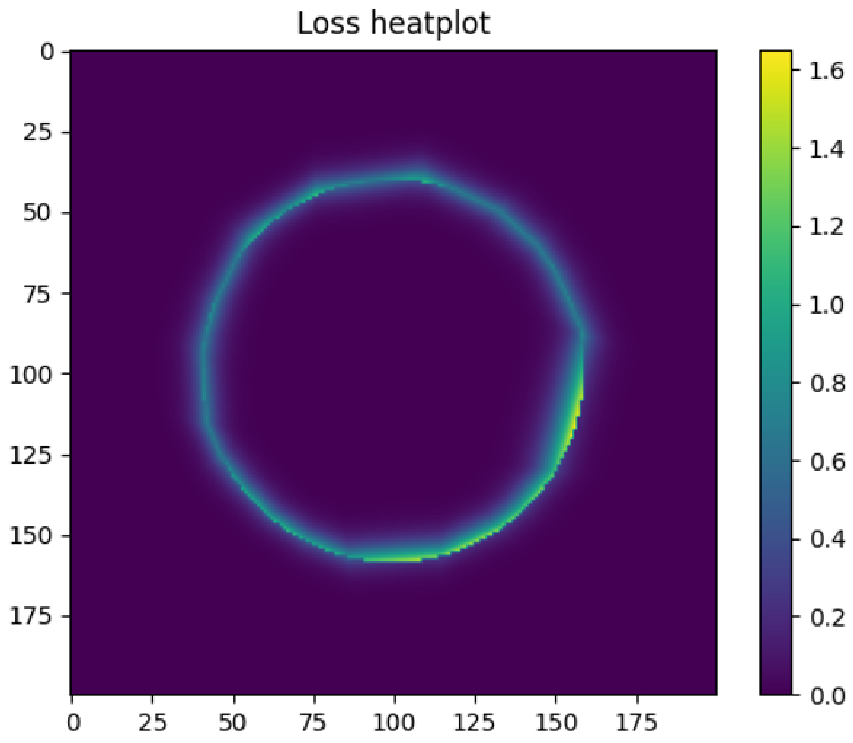
Here is a heatmap of gradient of loss for each node. (Without pruning)



Blue represents a part of the solution space where a node is lowering the loss. If the network is confidant in its predictions one way or the other, then a marginal change to the neurons has a negligible effect on loss. The places where the network is uncertain are the annulus around the boundary circle. Thus the mid blue/green of 0 is seen away from the boundary circle.

Nodes 0, 9, 10 and 19 are acting as a bias. They ignore their input, assigning all points to within the circle. Hence blue (improved predictions) inside the circle, and yellow (worse predictions) just outside the circle. The other nodes slice a part of the space away at the edges, saying that every point sufficiently far in some direction is outside the circle. Slicing off an edge makes the part outside the circle bluer, and the part inside that gets caught on this edge somewhat yellow.

Here is a plot of the loss, showing most of the predictive loss occurs around the edge of the circle.



Visualizing Neural networks, how to blame the bias

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

Background

This post is strongly based on this paper, which it calls the LRP algorithm, https://arxiv.org/pdf/1509.06321v1.pdf [1]

I later learned of the existence of this paper, which is even more similar to the ideas discussed here. https://arxiv.org/pdf/1704.02685.pdf [2]

In it, I examine 2 of the methods for neural network visualization, and show that they have structural similarities. I show that these algorithms only differ by a difference in how they treat the biases, and (possibly a difference in getting started)

The second algorithm obeys conservation laws, it tries to parcel credit and blame for a decision up to the input neurons.

Intro

The task we want is to assign importance to different inputs of a neural network in production of an output. So for example, in the case of a trained image classifier, the visualization method would take in a particular image, and highlight the parts of the image that the network thought were important.

A general method for visualizing neural networks is back-propagation. First evaluate the network forwards. Then work backwards through the network by using some rule about how to reverse each individual layer.

One example of this is differentiation. Finding the rate of change of the output, with respect to each input. But there are others.

Firstly, lets pretend biases in the network don't exist. We are allowed non-linearities, so long as they satisfy the equation $f(0) = 0$.

Lets look at various layer types and the different back-propagation rules.

Maximum

Most often found in the form of max pooling.

$$b = \max_i a_i$$

Gradient

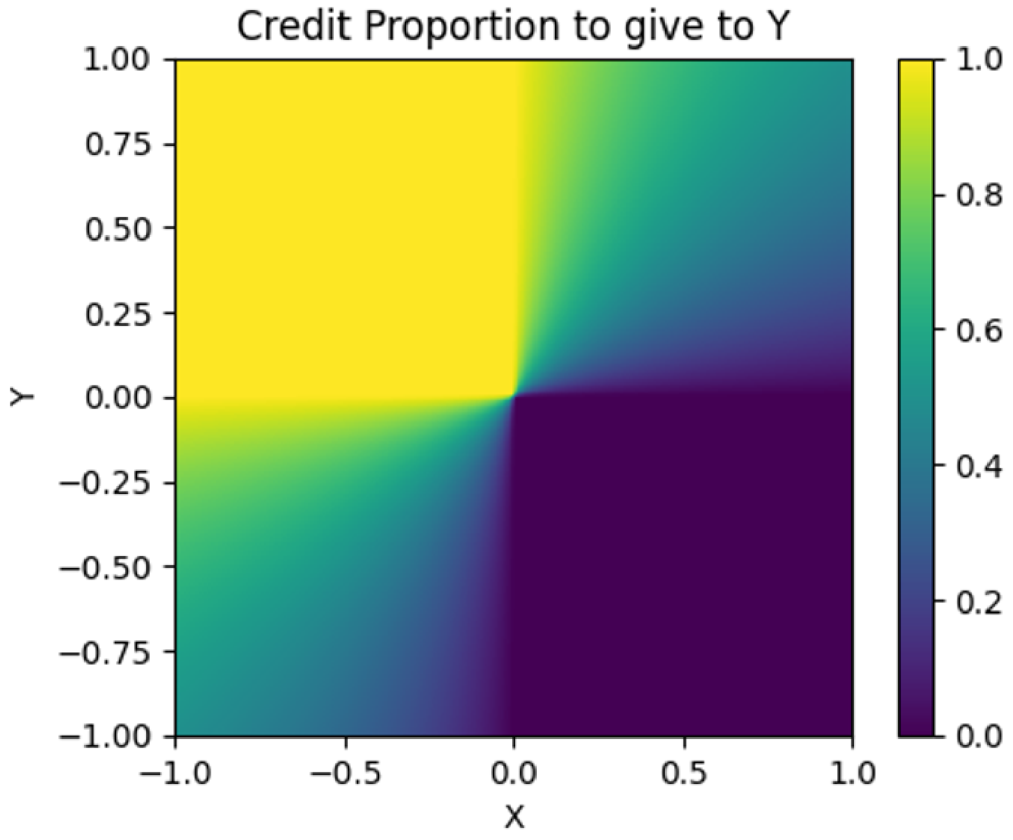
The rule used in gradient descent, and I think the only rule used in the paper above for back propagating maximum. (Notation note. R here isn't exactly a function. Its more like $\frac{d}{dx}$, its output is related to the context in which the input occurs. Think of every number in the forward net having an associated number)

$$R(a_i) = \begin{cases} R(b) & a_i = b \\ 0 & \text{else} \end{cases}$$

Ignoring the case of an exact tie. Exact ties, and what to do on that kind of singular point will be ignored in general, because they are fiddly and unimportant.

Radial.

Treats 0 as a special point.



$$R(a_i) = s_i R(b)$$

Where the plot above shows s_i and the s_i are non-negitive and sum to 1.

Case. $a_i < 0, \quad b > 0 \implies s_i = 0$. Relative to 0, a negative value contributes nothing to a positive maximum.

Case, $a_i > 0 \implies s_i = \frac{a_i}{\sum_{a_i>0} a_i}$ (In particular $s_i = 1$ if i is the only value with positive a_i

Case all negative. $s_i = \frac{1}{a_i \sum_i \frac{1}{a_i}}$.

Matrix multiplication

A common operation in neural networks. Even convolutions can be expressed as a matrix multiplication. The matrix just happens to be sparse, and contain repetitions.

$$b_j = \sum_i a_i w_{ij}$$

Gradient

Yet again a popular choice.

$$R(a_i) = \sum_j w_{ij} R(b_j)$$

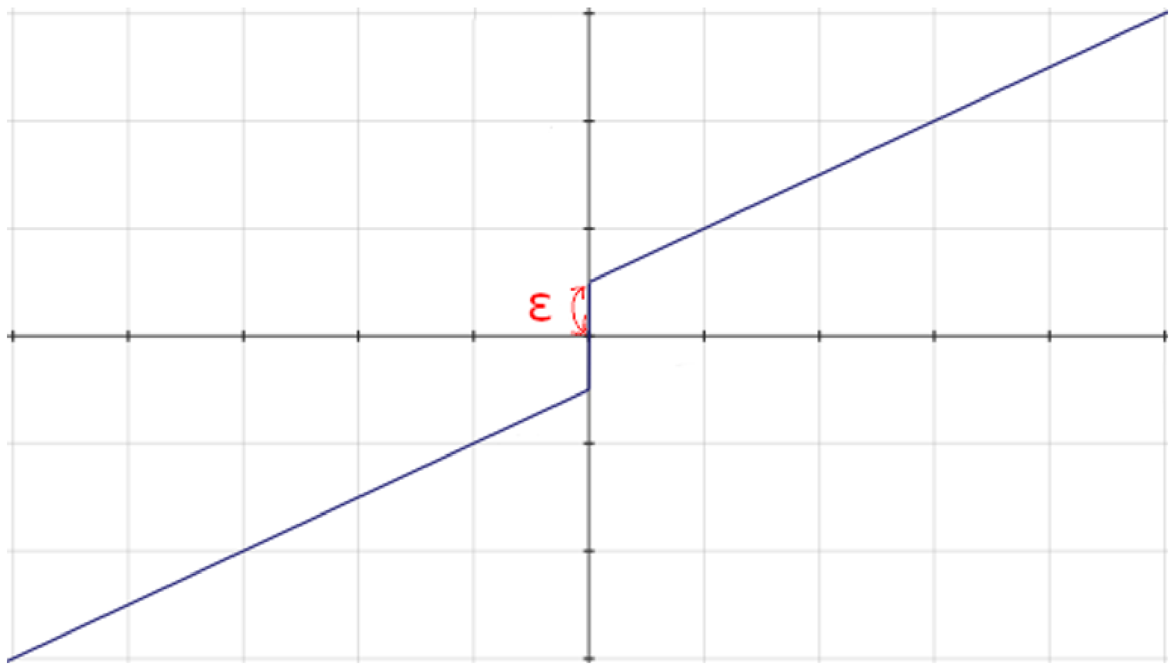
Normalized

What the LRP algorithm uses is

$$R(a_i) = \sum_j \frac{a_i w_{ij}}{\text{blip}(b_j)} R(b_j)$$

(This is deduced from equation 6 in [1])

Where blip is this function



A straight line of gradient 1, except for a little vertical jump to avoid 0.

Why are they using this blip? Because they want to divide by b_j here to get an interesting theoretical property (conservation of total) but if they don't add this little jump, they get numerical instability caused by dividing by something too close to 0.

Nonlinearity

There are several mechanisms proposed to deal with an arbitrary potentially non-linear function b = f(a), applied elementwise. We will impose the condition $f(0)=0$ for now.

Ignore

Very simple R(a) = R(b)

Gradient

$R(a) = f'(a)R(b)$ standard rule of calculus.

Slope

$R(a) = \frac{a}{b} R(b)$

Repeat

$R(a) = f(R(b))$

Downside: Is nonlinear in R(b), unlike every other method here.

Consistency rules

You can't just pick any option from each of these lists. Well you could. But there are some nice mathematical consistency properties it would be nice to have.

Scaling equivalence

Suppose you want to multiply all values in the network by a constant α, there are 2 ways you could do this. You could see the constant as a scaled identity matrix, and use the matrix multiplication rules. Or you could see the constant multiplication as a special case of the arbitrary elementwise function.

Matrix Rule	Nonlinearity Rule	Scale
Gradient	Gradient Slope Repeat	α
Normalized	Ignore	1

The other condition we might reasonably impose is that the nonlinearity formed by taking the maximum with a set of constants is treated the same way by the nonlinearity and maximum rules. Actually the constants must all be ≤ 0 because of the $f(0) = 0$ condition.

Gradient matches gradient. (Unsurprising)

In general its very hard to get anything else to work because $\max(a, c) = \max(a, \max(c, c)) = \max(a, c, c) = \max(\max(a, c), c)$ which ruins most kinds of credit splitting. Still, I suspect that radial maximum matches slope nonlinearity in the special case of a maximum of 2 objects. (Based on visually similar graphs.)

The Algorithms in [1]

The paper mentions gradients as existing work.

It also mentions work on the Deconvolution Method by Zeiler and Fergus, (2014) that roughly amounts to.

Layer type	**Back propagation rule used**
Maximum	Gradient
Matrix Multiply	Gradient
Nonlinearity	Repeat [3]

The LRP Algorithm by (Bach et al., 2015) has this form.

Layer type	Back propagation rule used
Maximum	Gradient
Matrix Multiply	Normalized[4]
Nonlinearity	Ignore

Simplifying the LRP

Firstly let $\epsilon = 0$ in the Normalized matrix multiply backpropagation rule.

Let R(a) be the backpropagation rule based on the table above. (Gradient Maximum, Normalized Matrix Multiply, Ignore Nonlinearity)

As the authors of [1] show, these rules conserve $\sum_i R(a_i)$ across network layers.

Consider another measure of the importance of a node $Q(a) = R(a)/a$. Can we rephrase all the equations to be in terms of Q(a) instead?

Rephrasing Maximum

$$b = \max_i (a_i)$$

$$R(a_i) = \begin{cases} R(b) & a_i = b \\ 0 & \text{else} \end{cases}$$

$$Q(a_i) = \frac{R(a_i)}{a_i} = \begin{cases} \frac{R(b)}{a_i} & a_i = b \\ \frac{0}{a_i} & \text{else} \end{cases} = \begin{cases} Q(b) & a_i = b \\ 0 & \text{else} \end{cases}$$

So this transformation turns the gradient maximum rule into the gradient maximum rule.

Rephrasing Matrix Multiplication

The $\epsilon = 0$ rule means $\text{blip}(b_j) = b_j$.

$$b_j = \sum_i a_i w_{ij}$$

$$R(a_i) = \sum_j \frac{a_i w_{ij}}{\mathrm{blip}(b_j)} R(b_j)$$

$$Q(a_i) = \frac{R(a_i)}{a_i} = \sum_j \frac{w_{ij}}{b_j} R(b_j) = \sum_j w_{ij} Q(b_j)$$

So this transformation turns the fancy normalization rule into the gradient again. All the problems where division means a risk of divide by 0, or more realistically dividing by almost 0 and getting a crazy huge number, have vanished.

Rephrasing Nonlinearity

Here is the place it gets a little more complicated.

$$b = f(a)$$

$$R(a) = R(b)$$

$$Q(a) = \frac{R(a)}{a} = \frac{R(b)}{b}\frac{b}{a} = \frac{b}{a} Q(b)$$

Under rephrasing, the trivial Ignore rule turns into the nontrivial slope rule.

Layer type	**In terms of** R	**In terms of** Q
Maximum	Gradient	Gradient
Matrix Multiply	Normalized	Gradient
Nonlinearity	Ignore	Slope

This slope rule works well so long as $f(0) = 0$. I mean technically there are lots of fiddly analysis details here. Lets just say. f is continuous, and that $\exists \delta > 0 : f'(x)$ is defined and bounded on $(-\delta, 0) \cup (0, \delta)$.

But if you haven't done real analysis, all you really need to know is that there are lots of slightly different definitions of reasonably nice functions, and so long as $f(0) = 0$ and there is nothing like the sharp cusp of a square-root going on at 0, the result is nice enough.

Also note that Slope and Gradient are the same method if the only nonlinearity used is Relu.

So all that is left is to deal with the biases.

Rephrasing conservation

This is simple. $\sum_i a_i Q(a_i)$ is now the conserved quantity.

End conditions

In order to make these whole procedures exactly the same, we need to make sure the end conditions match too. At the start of the forward process, where backpropagation ends, you can just multiply Q(input) by input. At the end of the forward process, where backpropagation starts, you can divide. The work by Bach et al started the reverse process using the output of the network, which was considered to be a positive scalar representing the likelihood assigned by the network to some particular classification.

This means that the reformulated version, the backpropagation is started with $Q(\text{output}) = \frac{\text{output}}{\text{output}} = 1$

What the actual problem was

Sometimes in a neural net, some fragment of decision comes down to a bias.

Sometimes the answer to "why is this number so large" would almost entirely be a large bias. The method bach et al proposed would focus on the tiny inputs or tiny weights rescaling them up in an attempt to explain the value.

This produces numerical instability by scaling up negligibly tiny weights and inputs. Which is exactly why bach et al added the blip function.

Solution 1: Blame the Bias

Augment the input data of the network with a list of 1's with the number of 1's equalling the number of biases used in the network. (Here a nonlinearity with $f(0) \neq 0$ can be considered to have a bias added to it.) These propagate through the network unaffected until each 1 is used up by scaling it by its corresponding bias, and adding it to where the bias should go.

You have now converted a network with biases into a network without biases. This means the techniques I propose work fine without numerical instability.

However, when propagating relevance backwards, some of that relevance goes to the list of 1's that correspond to the biases. So you get a heatmap of the relevant parts of the image. But also extra data about the relevance of various biases.

Instead of using lots of distinct 1's, you could use a single 1 everywhere. This would just add up the relevances of all individual nodes into a single total relevance.

Solution 2: Compare to baseline

This solution involves picking some "baseline" input. This could be an all blank image, or the average over the whole dataset. Run this through the network as well as the image you want to analyze. This gives $b = f(a)$ from the image, and $\bar{b} = f(\bar{a})$ from the baseline. We can now define the nonlinearity step as

$$R(a) = \frac{b - \bar{b}}{a - \bar{a}} R(b)$$

This is symmetric in the 2 images.

There are several different approaches to taking the maximum. One approach is to consider $m = \exp(\frac{1}{\epsilon}\sum \log(\epsilon x_i))$, the softmax function with softness ϵ. This function can be composed of the basic building blocks described above. So the relevancy can be calculated for any fixed ϵ. Then just take the limit $\lim \epsilon \to 0$.

Another way of computing the maximum is to use $\max(a, b) = a + \mathrm{relu}(b - a)$. Unfortunately these 2 computations suggest different relevancies.

Running code.

If you want to play around with this, look here. Unzip simple_model1.zip and point the link in My_Net_explain2.ipynb to the right location.

What I was hoping for regarding alignment

There is an old apocryphal story about someone training neural networks to distinguish dogs from wolves, and all the dogs being on grass, and all the wolves being on snow, so the network learned to distinguish grass from snow instead.

So suppose you have carefully highlighted all the dogs and wolves in the training data. But you won't have that extra data at run time. My idea was to optimize the net to make sure that the region of the image containing the animal was marked relevant. (The whole relevancy calculation is differentiable, so can be optimized during training) The end result of this training procedure should be a single perfectly normal neural net that can be shown a wolf on grass for the first time, and correctly classify it.

Unfortunately I haven't got this to work. All I got was the network rampantly goodhearting the relevancy metric. (Well I was trying this on a smaller MNIST based problem, but it was conceptually the same)

1. ^

   Evaluating the visualization of what a
   Deep Neural Network has learned
   Wojciech Samek Member, IEEE, Alexander Binder, Gr´egoire Montavon, Sebastian Bach, and Klaus-Robert https://arxiv.org/pdf/1509.06321v1.pdf

2. ^

   Learning Important Features Through Propagating Activation Differences
   Avanti Shrikumar Peyton Greenside Anshul Kundaje

https://arxiv.org/pdf/1704.02685.pdf

3. ^

   Relu only, other nonlinearities are not considered in this work.

4. ^

   Later in the paper they propose another option here. This other option will not be discussed further.

Train first VS prune first in neural networks.

Crossposted from the AI Alignment Forum. May contain more technical jargon than usual.

This post aims to answer a simple question about neural nets, at least on a small toy dataset. Does it matter if you train a network, and then prune some nodes, or if you prune the network, and then train the smaller net.

What exactly is pruning.

The simplest way to remove a node from a neural net is to just delete it. Let $y_j = f(\sum_i x_i w_{ij} + b_j)$ be the function from one layer of the network to the next.

Given I and J as the set of indicies that aren't being pruned, this method is just

$$\tilde{w}_{ij} = \begin{cases} w_{ij} & i \in I \text{ and } j \in J \\ \text{nothing} & \text{else} \end{cases}$$

$$\tilde{b}_j = \begin{cases} b_j & j \in J \\ \text{nothing} & \text{else} \end{cases}$$

however, a slightly more sophisticated pruning algorithm adjusts the biases based on the mean value of x_i in the training data. This means that removing any node carrying a constant value doesn't change the networks behavior.

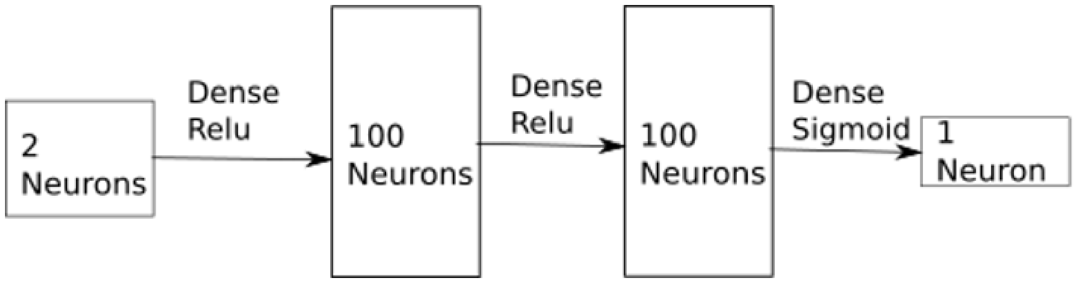
The formula for bias with this approach is

$$\tilde{b}_j = \begin{cases} b_j + \sum_{i \notin I} \bar{x}_i w_{ij} & j \in J \\ \text{nothing} & \text{else} \end{cases}$$

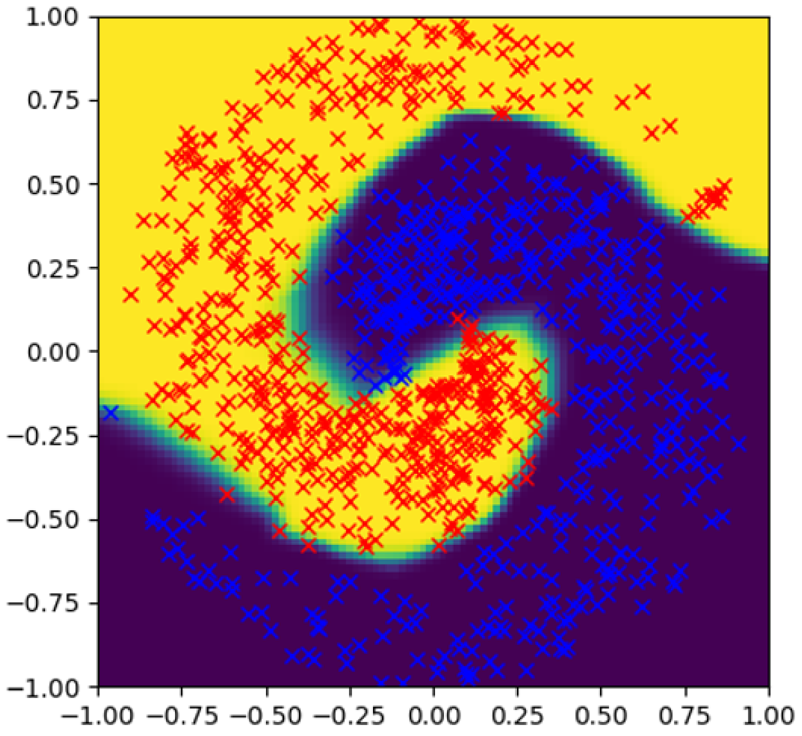
This approach to network pruning will be used throughout the rest of this post.

Random Pruning

What does random pruning do to a network. Well here is a plot showing the behavior of a toy net trained on spiral data. The architecture is

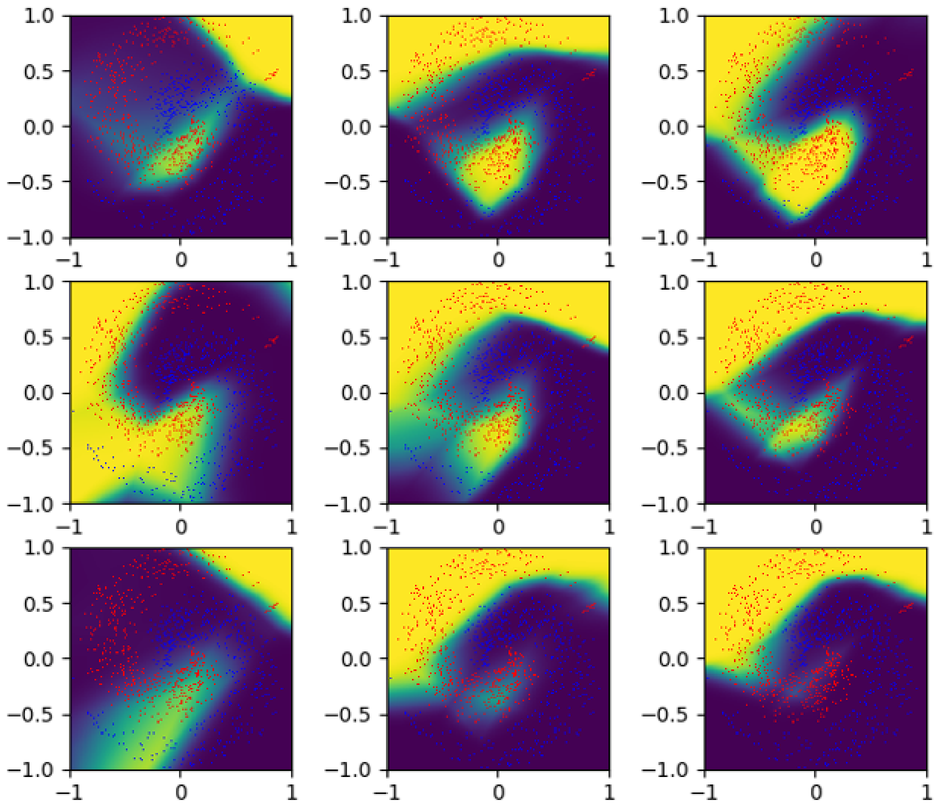


And this produces an image like



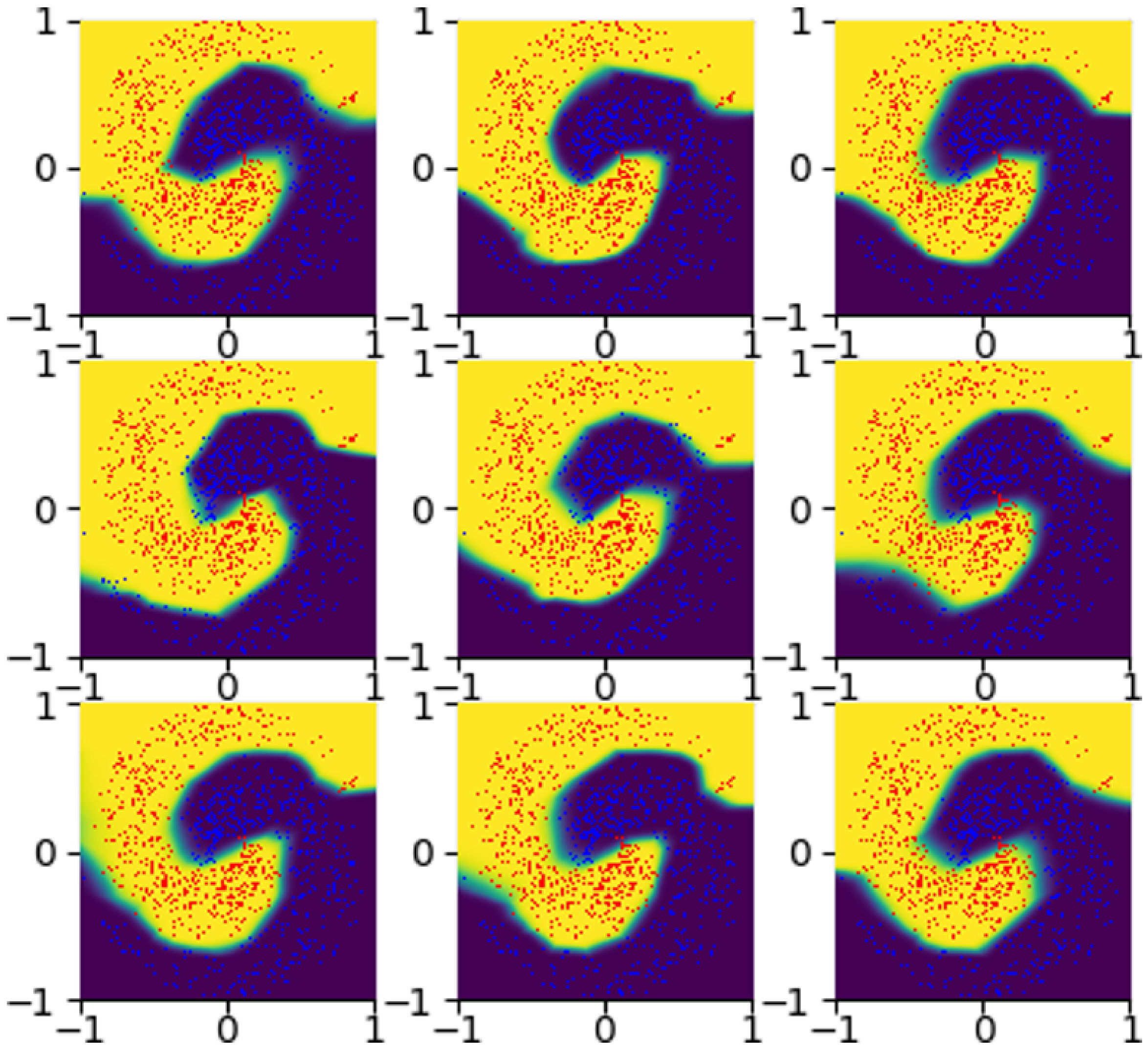
In this image, points are colored based on the network output. The training data is also shown. This shows the network making confidant correct predictions for almost all points. If you want to watch what this looks like during training, look here https://www.youtube.com/watch?v=6uMmB2NPv1M

When half of the nodes are pruned from both intermediate layers, adjusting the bias appropriately, the result looks like this.

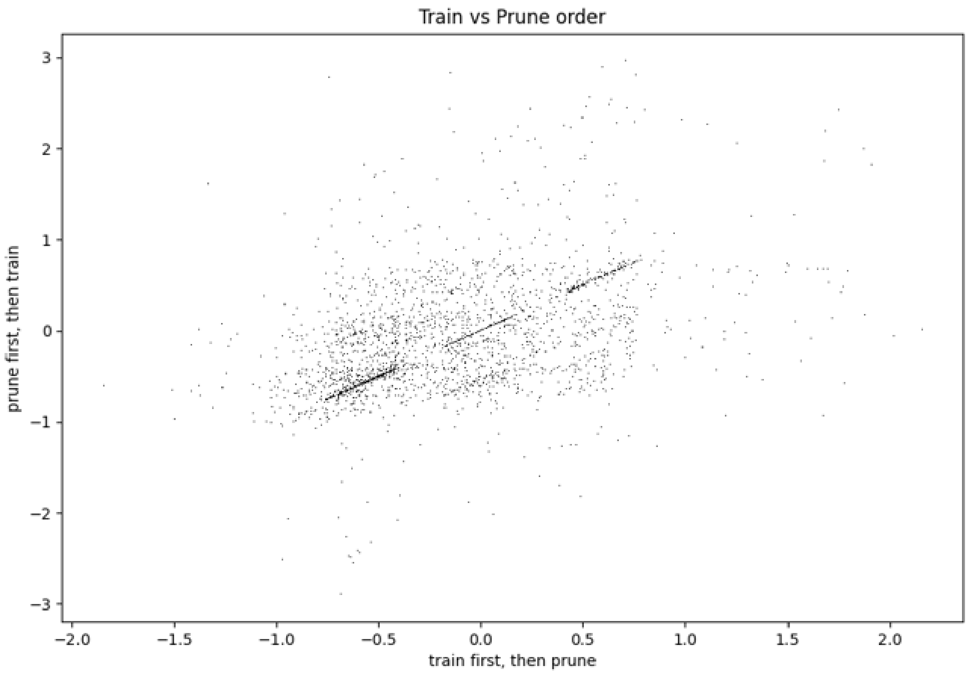


If you fine tune those images to the training data, it looks like this.
https://youtu.be/qYKsM29GSEE

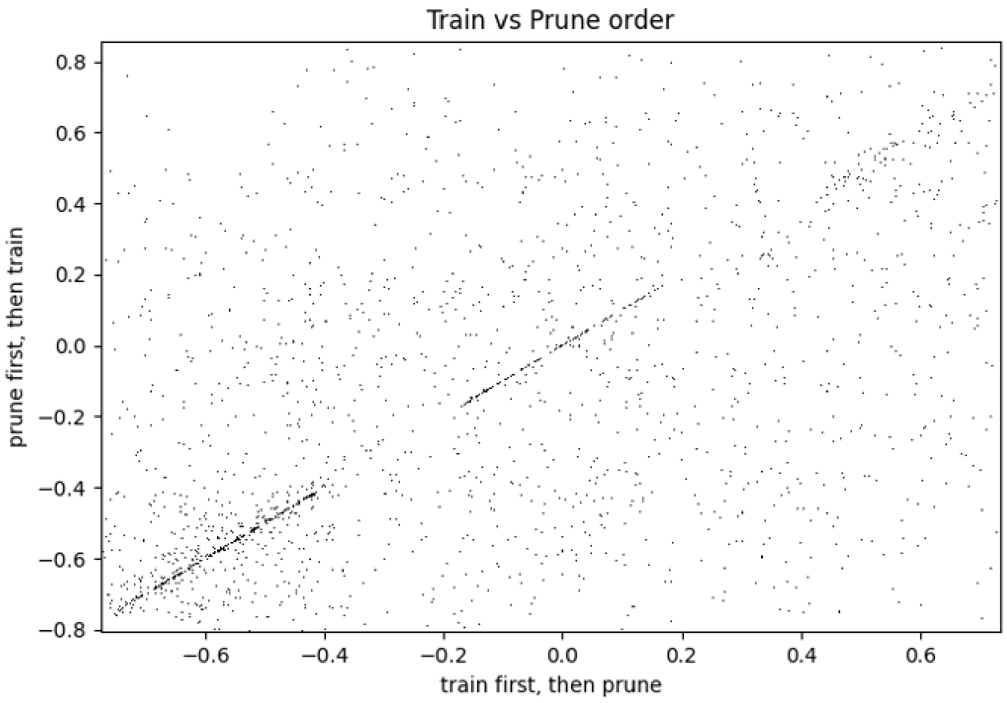
If you take the untrained network, and train it, the result looks like this.
https://www.youtube.com/watch?v=AymwqNmlPpg



Ok. Well this shows that pruning and training don't commute with random pruning. This is kind of as expected. The pruned then trained networks are functional high scoring nets. The others just aren't. If you prune half the nodes at random, often a large chunk of space is categorized wrongly.   This shows that the networks aren't that similar. This is kind of to be expected. However, these networks do have some interesting correlations. Taking the main 50x50 weight matrixes from each network and plotting them against each other reveals.



Same plot, but zoomed in to show detail.



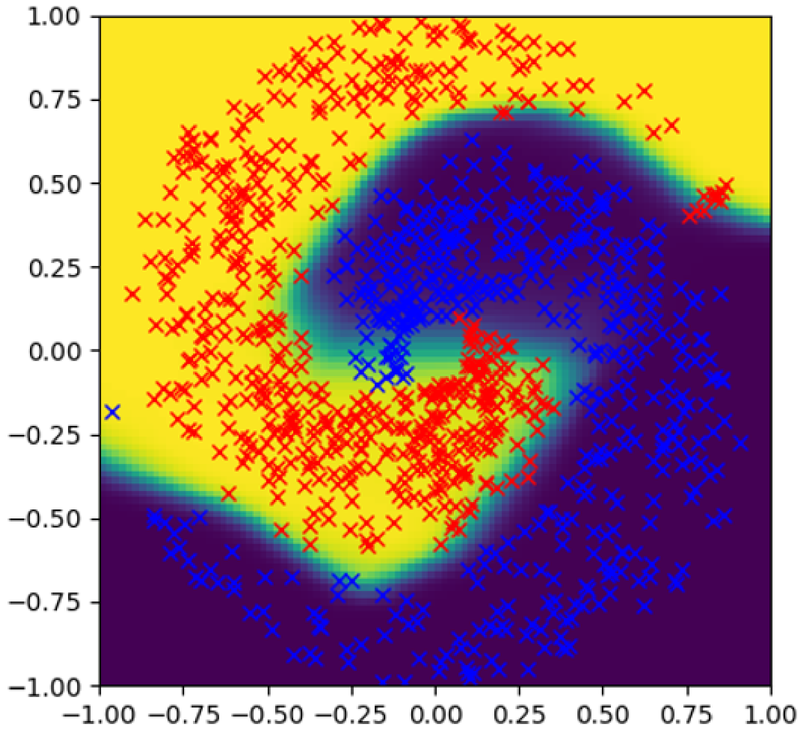
Notice the dashed diagonal line. The middle piece of this line is on the diagonal to machine precision. It consists of the points that were never updated during training at all. The uniform distribution with sharp cutoffs is simply due to that being the initialization distribution. The upper and lower sections consist of points moved about ±0.59 by the training process. For some reason, the training process likes to change values by about that amount.

Nonrandom Pruning

A reasonable hypothesis about neural networks is that a significant fraction of neurons aren't doing much, so that if those neurons are removed then the network will have much the same structure with or without training. Lets test that by pruning the nodes with the smallest standard deviation.

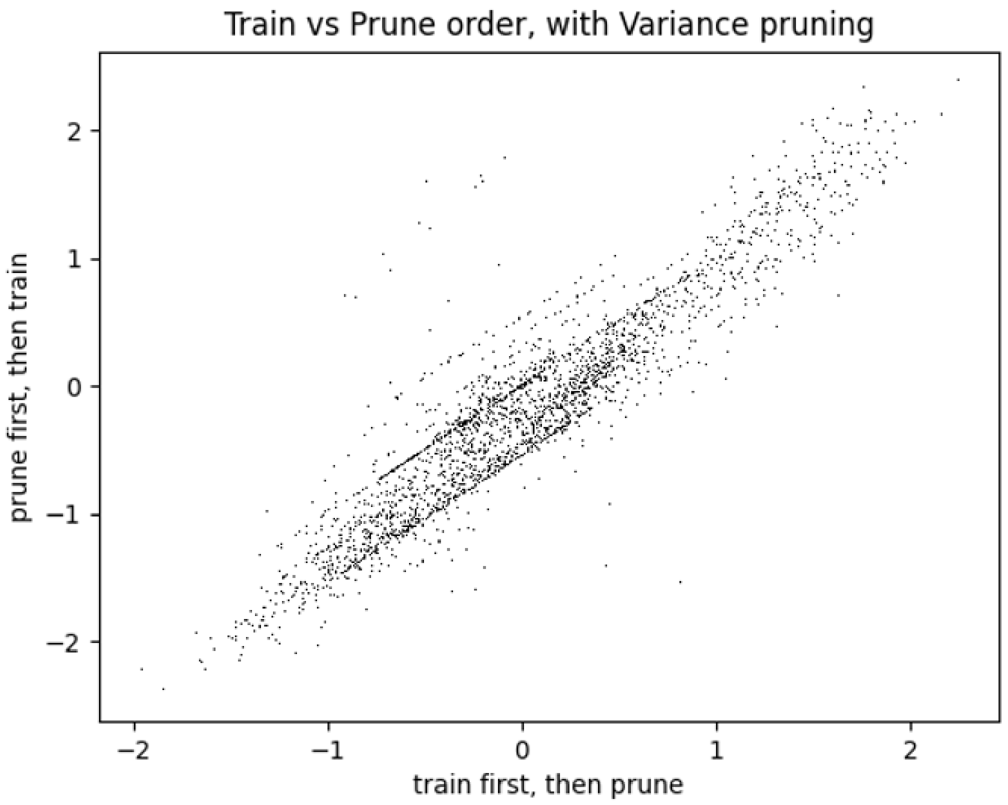
This pruning left an image visually indistinguishable from the original. entirely consistent with the hypothesis that these nodes weren't doing anything.

When those same nodes are removed first, and the model is then trained, the result looks like this.



Similar to the trained and then pruned (see top of document), but slightly different.

Plotting the kernels against each other reveals



This shows a significant correlation, but still some difference in results. This suggests that some neurons in a neural net aren't doing anything. No small change will make them helpful, so the best they can do is keep out the way.

It also suggests that if you remove those unhelpful neurons from the start, and train without them, the remaining neurons often end up in similar roles.