Compressing images inside a neural network

References:

Github: <https://github.com/marmust/NETMEM>

Gmail: [daniel.varkin@gmail.com](mailto:daniel.varkin@gmail.com)

Introduction:

Why even compress images:

Image compression is quite an important technology, since image files become very large, and their size increases exponentially when increasing resolution linearly, high quality image compression algorithms are needed in many cases, for example: youtube, who are handling thousands of GBs of image data per hour, and other video streaming platforms like twitch, and netflix.

The problem of image compression:

Current image compression algorithms usually have 1 of two downsides:

1: images have to be decoded from the algorithm to be used, for example zip.

2: lowering the amount of detail in the images by simplifying unimportant areas of the image such as: blurry background, darkness, etc, which can make the result

look blocky, and reduce the quality of fast moving objects: such as confetti. For example youtube video compression.

Proposed solution:

Solution:

Neural networks remember things by having different “activation patterns” coming from different inputs provided, and those weight less than the final image

Reconstructed, and the second advantage of neural network memory, is that

When neurons are by themselves, they can remember as much data as they

Weight themselves, but when they are together, each neuron can remember a

couple of things at once, letting the network remember more than it weighs.

Optimizing the solution and bringing it into code:

Here are my findings from my experiments conducted:

**Part 1: building the network:**

Optimal network architecture is the following:

1: at the beginning the network should be the same weight as the

images themselves.

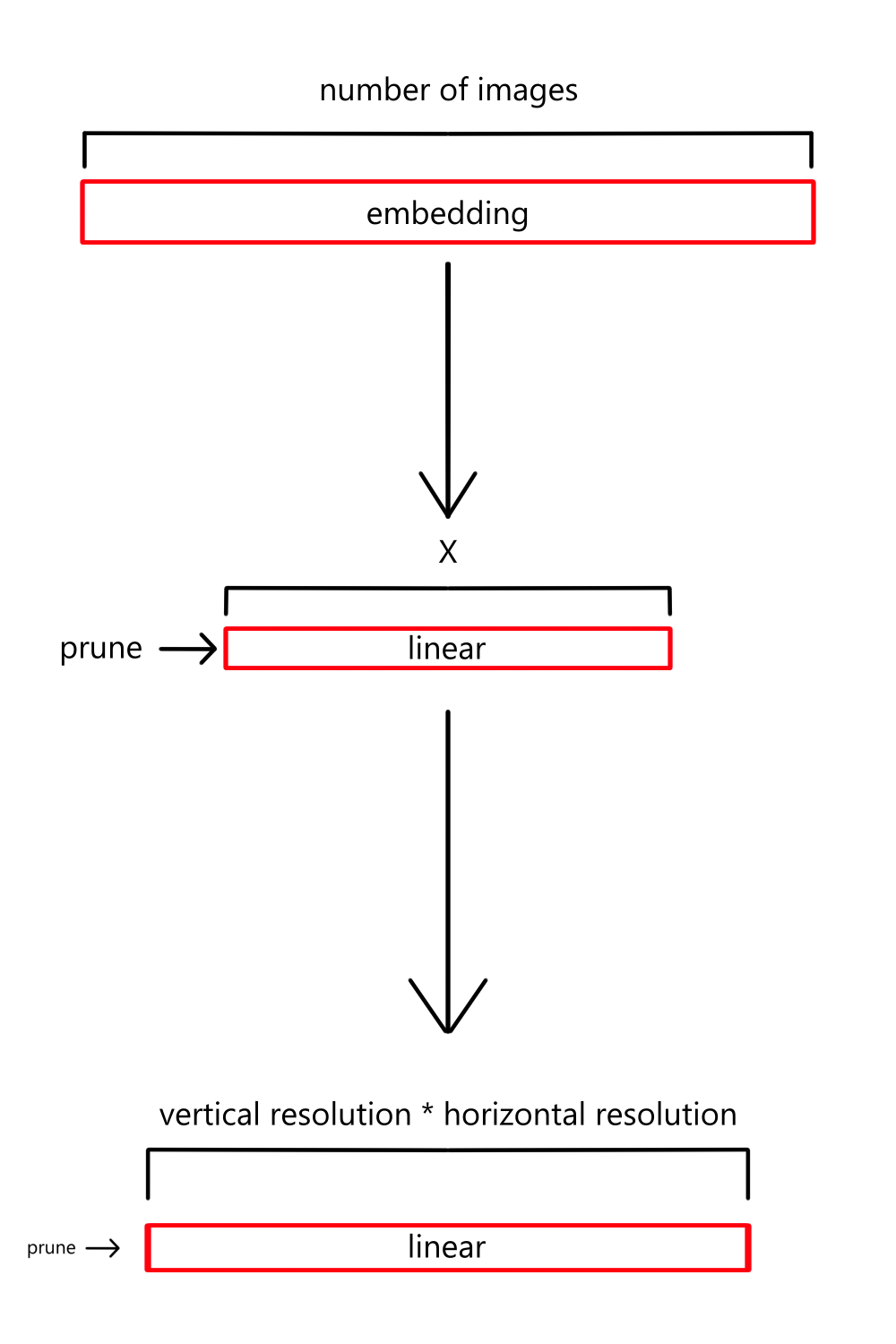
2: input layer should be an embedding layer, with an input size

equal to the number of images, and an output size equal to X

(we find X later)

3: middle layer should be a linear layer of input size equal to X,

And an output size also equal to X.

4: and the last layer should be a linear layer with input size of X,

And output size of the image resolution.

5: the activation functions that i found best are the following:

Embedding = none

Linear1 = sigmoid

Linear2 = relu

6: and for the loss function i found that both MSELoss and

And BCELoss are good, but BCE results in less artifacts

When checking the network.

7: i have not tried different optimizers, but it seems that Adam

works Good enough.

To match the network weight exactly to the image weight we can

Do the following:

I = number of images

V = vertical resolution of an image

H = horizontal resolution of an image

Image weight = IHV

Network weight:

IX + XHV + I + X + HV

Equation:

IX + XHV + I + X + HV = IHV

Solve for X (middle layer size to match image weight):

X = (IHV - I - HV) / (I + HV + 1)

**Part 2: training and optimizing:**

This part still requires research, but here are some important things to

keep note of:

1: low learning rate: a high learning rate will cause images to merge with

each other, and a way too small learning rate will train the network too

slowly.

I use a learning rate of ~0.0005 when training on 100 images of size

128\*128

2: a high number of runs over the dataset:

In the previous example I went over the dataset 2000 times.

3: pruning: after training prune the network on both linear layers, a good  
 Pruning value that i found is 0.15, it does result in some noise, but

Nothing serious. If you are planning on pruning the network even more,

Make sure to use some kind of denoiser on the images so they come

Out cleaner.

Experiments:

The following experiments have been done on a network that:

* Remembered 250 images
* Image resolution: 256x256
* Architecture of:

Memorizer(

(em): Embedding(250, 248)

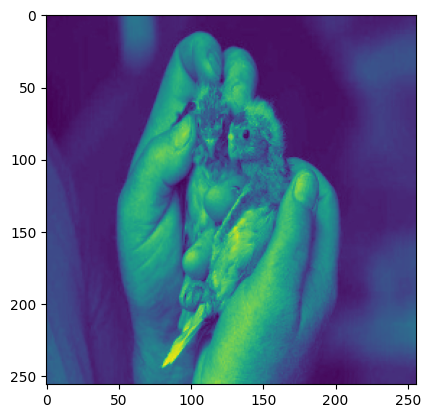
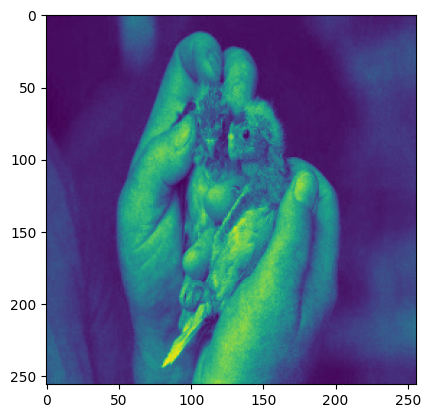
(fc1): Linear(in\_features=248, out\_features=248, bias=True)

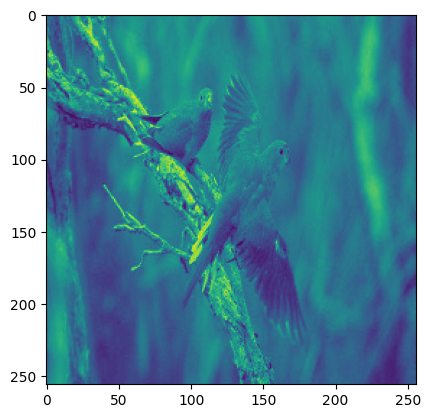
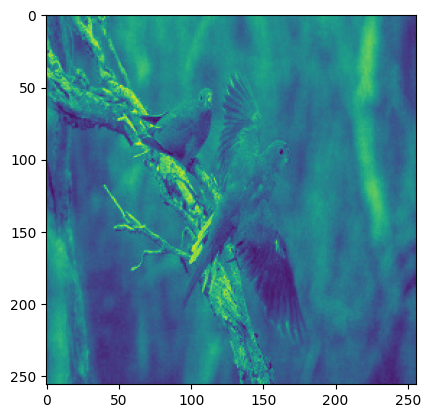
(fc2): Linear(in\_features=248, out\_features=65536, bias=True)

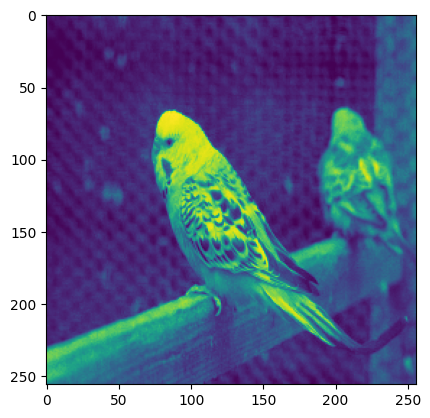
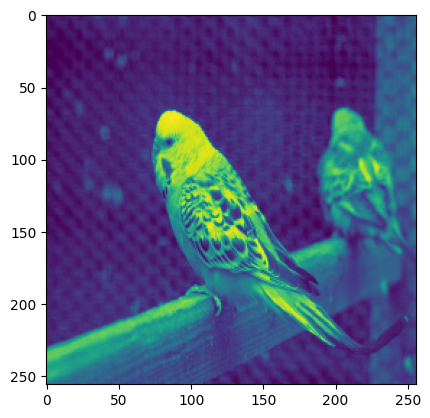
)

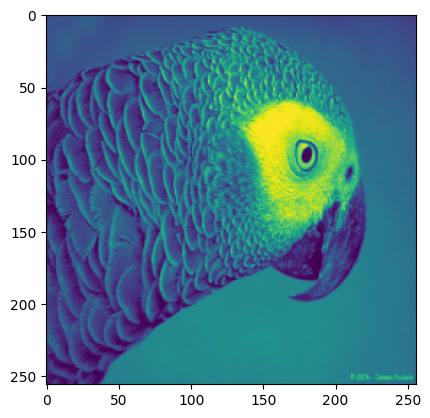
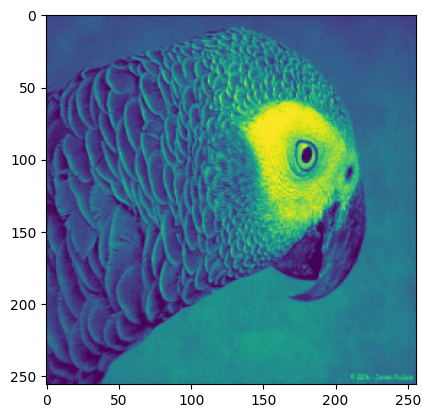
* The images weigh: 62.50 MB.
* The clean network weighs: 62.42 MB.
* It has been pruned by 0.15.
* The pruned weight is 53.14 MB.

Samples:

Network original







Conclusion:

This method of image compression is only good for specialized use cases, when it is

Better to trade storage space for computing power to run the network.

However in cases where a fast image by index lookup is needed, and the dataset of

Images does not change, this can be a pretty powerful algorithm.

Upsides:

- has minimal quality reduction while still compressing images to a smaller size.

- more neural network modules can be added to for example: search for images,

Classify them, etc…

- can sometimes be faster than conventional database structures that involve a

Massive amount of images, at a low resolution.

Downsides:

- the network requires pretty heavy GPU’s to train and run especially at larger

Sizes.

- every time the dataset needs to be changed, images added or removed, the

network has to be retrained.

- this is lossy compression, so images still come out somewhat corrupted.