50.021 -AI

Alex

Week 05: Fine Tuning aka Retraining

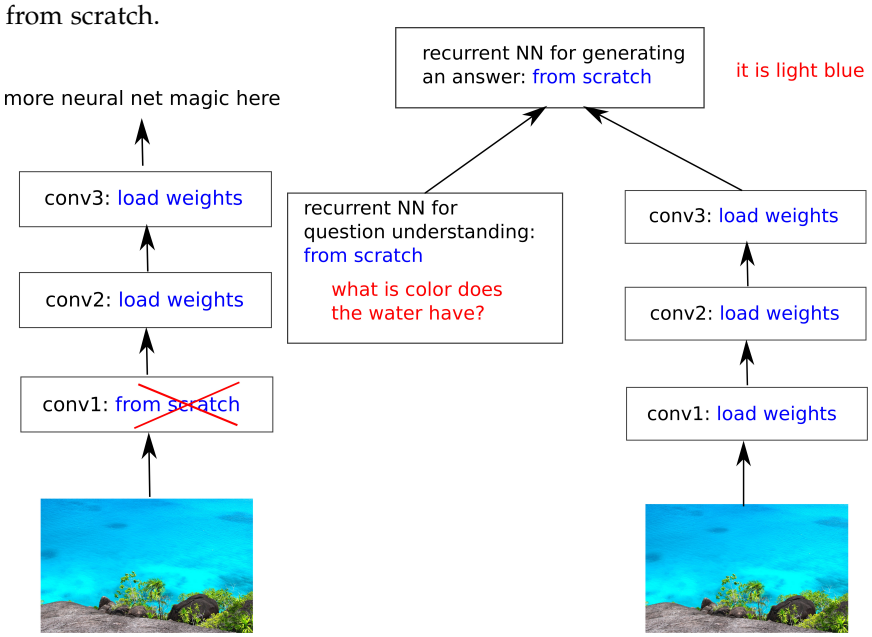
[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

A recap

I hope you have seen the following:

- data augmentation can improve prediction performance at test time – the good side: you can use a classifier without retraining. down side: you are slower at test time.
- it can be useful when preparing samples at training time
- when training a model: reusing weights for the lower layers of a model improves performance drastically
- it makes no sense to load weights for a layer, when one skips loading weights for any layer below. why ? the weights of a layer expect certain output statistics (remember the effect of batch normalization?) from the layer below. If you do not load weights for the layer below, and train then, with high probability you will not get the weights right.

Left: a neural net where finetuning makes NO sense –because we skip a layer. **Right:** a neural net where finetuning makes sense – because we load model weights from the inputs up to a certain layer, other branches (for processing the text) are completely learnt from scratch.



- caveat: most weights are as of today derived/converted from caffe models. caffe assumes a convolutional structure of $N - C - H - W$ that is: batchsize–number of channels–channel height–channel width. Tensorflow default is $N - H - W - C$. You will never find out what error you made if you make that mistake :) .

Why data augmentation and loading model weights does matter so much?

A golden rule in machine learning

The number of parameters and the complexity of the model learn must be in proportion to the number of samples you have for training

- large number of samples – learn a complex model
- small number of samples – learn a simple model – like a linear SVM, but ok to learn over some precomputed, hand-designed features

Why does finetuning help? preset parameters to some values

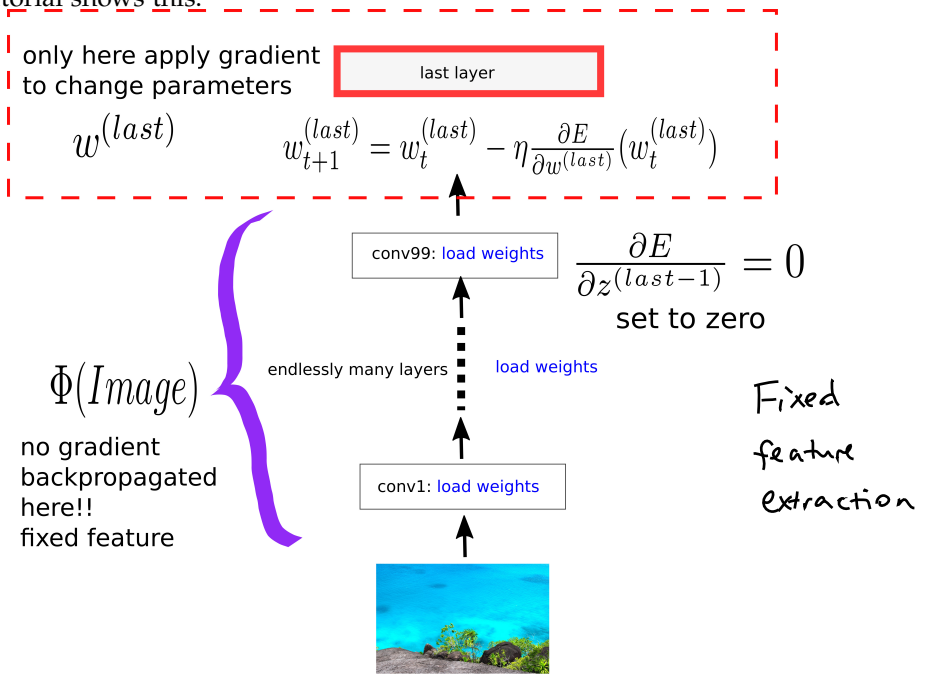
As for loading model weights the answer is: when training a neural network, one always finds some local optimum. The local optimum depends on what detectors you learn in every layer. Remember the wave-like filters from the zeiler paper?

You can learn filters well only when you have enough training samplesm, often one needs hundreds of thousands. When having only a few thousand samples it is best to preset the filters, and start from such a good initialization.

Finetuning can in theory destroy such a good initialization (when trained too long) – in practice backpropagating gradients changes the highest level weights most, so that – at the beginning of training – the weights in the update layer adapt faster to what one wants to learn – and the overfitting by changing lower layer weights to bad optima sets in later.

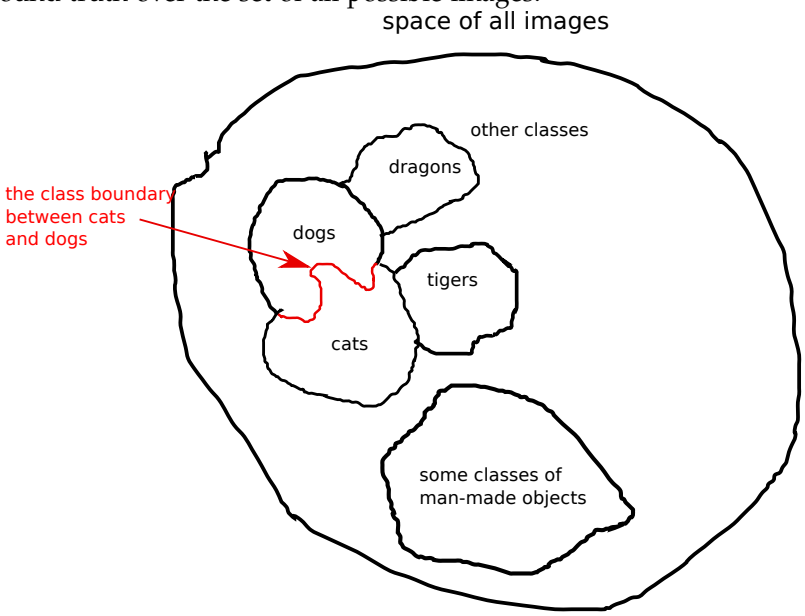
One can bring the idea of finetuning only the output layers to the extreme - and this works best when the number of training samples is very small:

You train a neural network with finetuning, but you stop back-propagating gradients beyond the highest layer. That means: we update weights only at the very last layer. The google finetuning tutorial shows this.



Why does data augmentation help?

As for **data augmentation**, one answer is: a neural network is not a perfect predictor. For classification it learns in the end boundaries between classes in image space. Given that one trains with a finite number of data, the learnt boundaries are not perfect aligned to ground truth over the set of all possible images.



The naive view of class boundaries is shown in above graphic. There are object classes, and boundaries between them. This naive view of smooth boundaries seem not to hold for neural networks.

In fact one can show: very similar images may have totally different labels. One can take one image of a cat, change very little in this image, and it gets a totally different label, like: mango, durian, chocolate sauce, toothbrush

Why does that happen? We take images, and process them in a neural network. Layer by layer activations get computed. If a layer has K neurons, then the space of all possible activation vectors is a K-dimensional vector space.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

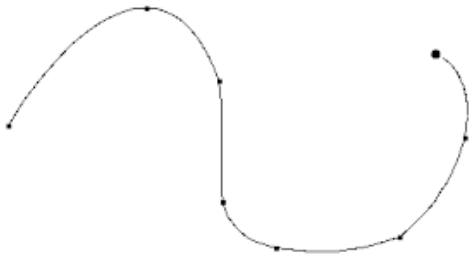
Table 1: GoogLeNet incarnation of the Inception architecture

For one image the activations of all neurons of one layer are one single point in a high-dimensional space (dimensionality being equal to the number of all output neurons in this layer!) – for above googlenet/inception model: inception (4a) has 100352 dimensions.

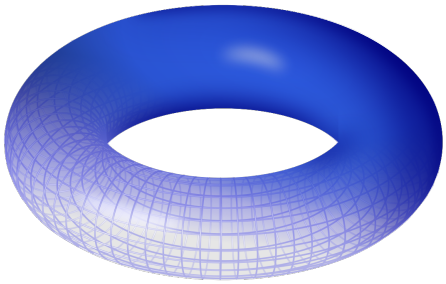
One can ask: when we compute activation vectors for many images, how will they be distributed in this space of all neuron activations? Usually they do not fill up the high-dimensional space with equal density.

Often they are approximately distributed in a small thin region of the high dimensional space of all possible activations. Such a thin region (of lower dimensionality) is called a manifold.

What are manifolds? curves are 1-dim manifolds, a curved 2-dim hyperplane, like a torus, would be a 2-dim manifold.



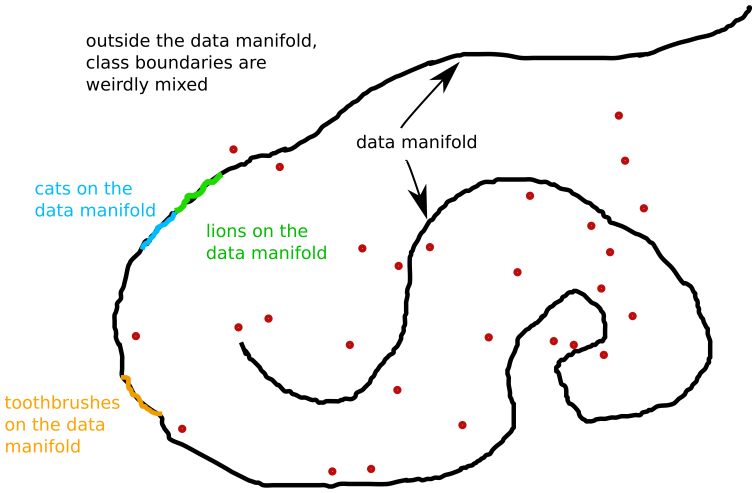
deformation of line (2D)



deformation of plane (3D)

Higher order manifolds cannot be drawn trivially, but defined mathematically.

The hypothesis is: the images – mapped into neuron activations of one layer– lie the space of neuron activations on a thin manyfold with some outliers (the red dots).



This hypothesis has one big consequence: the neural network was never trained with a sufficient number of images in regions outside of such manifolds. But: one can only predict reliably, when one has seen a similar sample!

Therefore: the boundaries between classes in these outlier zones (outside of the data manifolds) are very random (due to lack of similar samples in these zone), and: the predictions in these outlier zones are weird and almost random!

As a consequence: One can change an image very little by walking outside of the data manifold, and the prediction changes from one class to a totally different class – a green snake can become a red fire truck – regarding the prediction of a neural network. Lets explore this in the in class task.

Data augmentation **at training time** helps to fill a larger region

of the space of all activations with well-defined training samples. As said, one can only predict reliably, when one has seen a similar sample. Data augmentation at training time helps to cover more of the space with samples. Thats why it helps

Data augmentation **at test time** with many views helps in the following sense: looking at many views and averaging predictions can help as a buffer when one view is an outlier (outside of the data manifold) and falls into the outlier zone, and its prediction would be quasi-random.

These explanations are intuitions. Data manifold is a thinking concept to describe that the mapped training samples fill the space of all activations of a neuron layer very unequally. There are always a few examples nearby :). Intuitions are always a bit wrong and a bit blur.

In class task

Goal is:

- take an image, and create an image that looks very similar but has a totally nonsense prediction for it! This is fooling of neural networks.
- you will start with simple prediction code and code a method that starts with a given input image, and creates a very similar image for which the neural network outputs a different prediction label - a label that you have chosen beforehand.

We want to show along the lines of:
`https://arxiv.org/pdf/1412.1897.pdf`

that many images have very similarly looking images with totally nonsense labels.

consider `alexnet_test_centercrop.py` . It takes an image and computes a prediction. Business as usual.

- the overall goal: take as inputs: an image, a desired class label, and update image pixels until the prediction of the image outputs the desired class label as the highest scoring class.
- How? Compute the gradient in the space of image red-green-blue subpixels. Apply the gradient with a stepsize to perform a gradient ascent.
- You can start with above code. Modify the code to return not only the output predictions in a tensor, but also the gradient of a particular class label of output predictions. you may need to define at least one additional placeholder for the desired class to optimize.

  It is one possible way to define a gradient tensor, and fetch the combined prediction and gradient when doing a sess.run on gradient and prediction tensor. you can change image pixel values in a while loop which repeatedly calls sess.run to fetch prediction and gradient, then updates the image.

  tf.slice helps to extract the component of a tensor for a particular index (you want to maximize the output prediction for a particular class, no?) [1]. tf.gradients is the way to compute a gradient.

- recommended initial stepsize: 5 to 20.
- you can compute a gradient for either a simple or a more complex criterion. The simple criterion is the prediction score of the output of the neural network for a chosen class. It should be maximized. A more complex criterion is: above prediction score for the chosen class minus the prediction score of the highest scoring class which is not the chosen class. Maximizing this does two things: it increases the prediction score for the chosen class, and it decreases the prediction score of the highest scoring class which is not the chosen class.

Lets program a first version: You can do the ascent in a while loop, until the predicted label (in the sense that the predicted score is highest for that label) is equal to the desired label.

This still may not work when saving the image:

[1] tf.argmax helps to return a tensor containing the index of the highest scoring class – in case that you want to maximize the difference between the output of the current class and the output of the highest scoring other class

After saving the image still might have a different label. Why? Each pixel is stored as three RGB-subpixels - for red, green and blue channel.

A the image subpixel values can be out of bounds – relative to what image toolboxes that save images expect. Typically it should be in $[0, 255]$. Above optimization does not care for such boundaries.

B After optimization, our values for subpixels are floats . When saved the image subpixel values get rounded to integers. PIL usually expects inputs as integers in $\{0, \ldots, 255\}$. The rounding may change the prediction!!

C Image saving might introduce additional biases, e.g. changes in subpixel values due to lossy compression, for example when saving as jpg. Save images as png without lossy compression.

Here are the proposed solutions for A and B.

A:
perform a gradient descent in a suitable subspace: in the while loop run sess.run, fetch the gradient, compute for your fixed stepsize a temporary new image (of shape (1,227,227,3)). This is nothing new, except that you do not overwrite the image from the last iteration.

Now: Find the subpixels for which the updated values (with dataset mean added back!) would be outside of the interval $[1, 254]$. Set the gradient to zero for all those indices. Now update the image with this modified gradient.

Why does this work well? Imagine you would compute the gradient only for some set of subpixels, and perform gradient ascent – in this case the prediction for the desired label would still go up. Computing the gradient only for some set of subpixels, and perform gradient ascent only for those is same as: not changing the other pixels. This is same as applying a zero gradient in the other pixels. Thats why zeroing gradient in some subpixels (for those where after applying the gradient values would be out of bounds) works! It ensures that every subpixel stays in $[0, 255]$

This method has a theoretical non-convergence risk (namely if the gradient would be set to zero in every subpixel.)

B:
have two termination conditions in the while loop. the first is: terminate when the predicted label is equal to the desired label – as we had before. But now you add the dataset mean back, and round the image. test if the rounded image still has the desired label. if not,

then continue the gradient ascent. If it does, then save the rounded image to disk.

...

validate that the saved image has indeed the final silly label by putting it into `alexnet_test_nocrop.py` and running it there .

Hint for numerical stabilization: if the idea with testing the rounded image does not work for you, you can save, re-load the disk-saved image and test it. If even this does not work, change the convergence condition to: the fc8-layer prediction of our desired class must be at 5 larger than the score of the highest-scoring other class. Or you compute a softmax over the fc8 scores $p(c) = \frac{\exp(s(c))}{\sum_{c'} \exp(s(c))}$ – you know the max-value-subtraction trick when computing a softmax, dont you???

- See the off manifold instability: try to classify the image using `alexnet_test_centercrop.py` and different resizing options than 250 set there.
- visualize the differences between original and modified image.
- the idea works in principle with all kinds of neural networks.

Outlook: other topics which combines image or video inputs

- Visual question answering `https://arxiv.org/abs/1606.01847`, `https://arxiv.org/pdf/1612.00837.pdf`,
- video captioning `http://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Pan_Hierarchical_Recurrent_Neural_CVPR_2016_paper.pdf`,
- video story telling `https://sites.google.com/site/describingmovies/`.

Another golden rule in machine learning

You dont play with your data? dont visualize it? dont ask questions to your data, and dont try to answer them by computing weird customized statistics? Then you will be able to solve only the easiest problems. Intuitions, but not definitions can roughly guide you WHAT to compute in order to answer questions that you have to your data.