50.039 – Theory and Practice of Deep Learning

Alex

Week 05: Fine Tuning of neural networks

There are two big important things which you are not going to learn when trying to get experience on MNIST and CIFAR, but which matter a lot in practice: data augmentation and fine tuning.

The goal of this lecture is to give an introduction to fine tuning as a starter to training a deep neural net.

- you will get an introduction to using amazon AWS for using deep learning. Its branded? Well, yeah, but ...
- understanding goal: if you have a small sample size for training, then it is state of the art to perform finetuning as a special case of transfer learning

Takeaway for this lesson:

- finetuning means you load weights from another model as much as you can
- finetuning can improve performance when training with small sample sizes greatly as compared to training from scratch with a random initialization.
- finetuning can be used for models with different types of inputs, e.g. image and text – but always bottom up: from one of the inputs until the first layer where weights cannot be loaded anymore (bcs one changed the network design at this point to either a completely different layer, or due to shape mismatch). after one such blocker-layer, it makes no sense to load weights further above
- finetuning has two flavours: train all layers, train only the top layer
- using pytorch transforms

1 Finetuning

The goal of this lesson is to understand about finetuning when training a neural net.

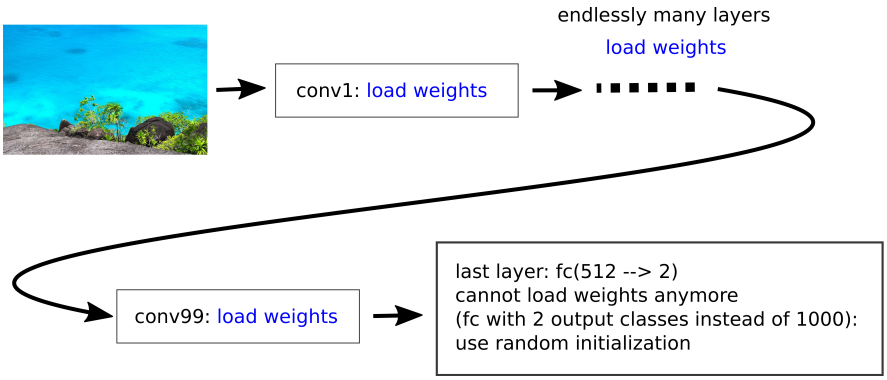
One does not train deep neural from scratch. Always reuse weights from similar tasks as initialization, except your data is in the order of hundred thousands and more.

Finetuning is a general principle. Reuse weights obtained from a similar task. Initialize a neural network before training from the input layer until the first layer where parameter sizes do not match anymore . You want to use a complicated neural net that does something totally else than classification or detection? Finetuning is still applicable for all those layers from the input on , until the first layer where one cannot load weights anymore. Reasons to be unable to load weights can be a layer type mismatch or layer shape mismatch (, e.g. use a convolution layer but with different kernel parameters).

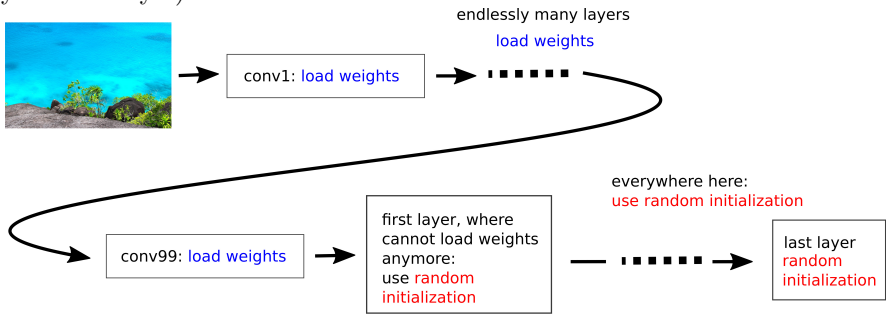
Mnist and Cifar-10 work well without finetuning, however note the simplicity of the tasks: images with 28×28, or 32×32 have limited variability and complexity compared to 300×500 images! Mnist and Cifar-10 are very useful for testing small ideas, but they are outliers within deep learning tasks.

In the practice session you will be asked to take a deep network (resnet or mobilenet or whatever you like), initialize it with weights from a 1000 class imagenet task, and then retrain it for either 102 flowers classes or for 2 ants versus bee classes. Is it surprising that one can re-use weights from 1000 object classes that are mostly things and animals for flowers? The low level filters likely will be very similar.

What needs to be changed? Well the last layer for sure: one has now 102 or 2 output classes instead of 1000.

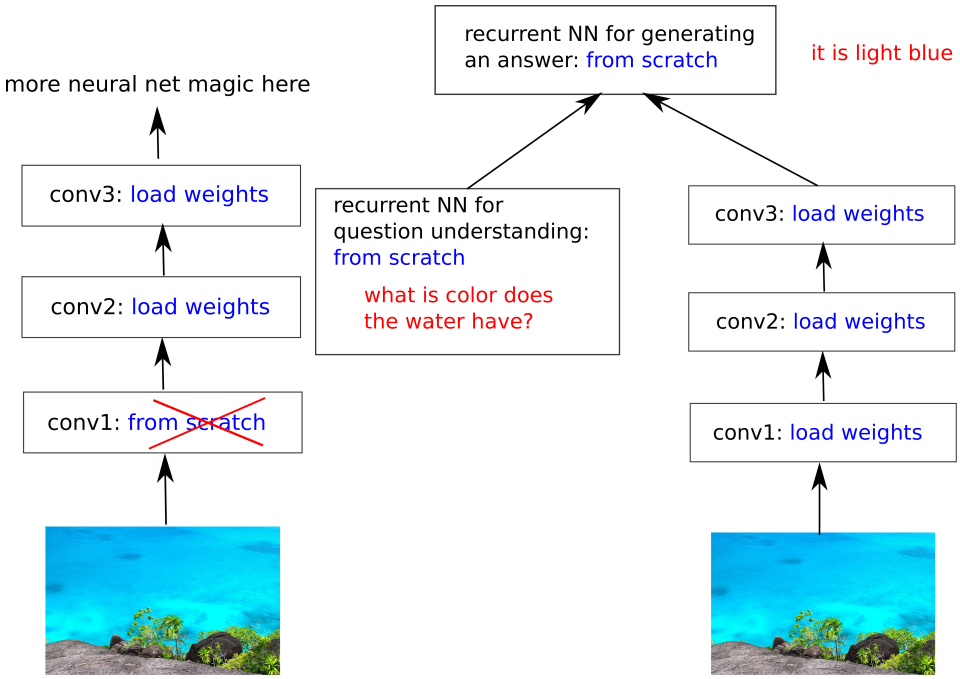


A more general case for a more complex network (where one does not change only the last layer) looks like that:



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 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 output statistics 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.



Example: suppose you want to tell stories from video frames. so you want to process a sequence of images, and then output a sequence of text.

You will consider to do the textual outputs by a recurrent neural net, but you will need to input feature vectors computed from every frame of the video. If you dont want to use 3D-convolutions, then you can process every image frame by a neural net trained for classification, and input as features the activations of neurons from some higher layer of this neural net!

Why does loading model weights does matter so much?

1.1 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, and train it over some precomputed, hand-designed features

1.2 Why does finetuning help?

Consider training a neural network:

then, 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?

These filters have high dimensionality of their parameters. `https://mxnet.incubator.apache.org/api/python/gluon/model_zoo.html` . Training 15 million parameters with 1000 samples violates the golden rule. You will overfit for sure.

Alias	Network	# Parameters	Top-1 Accuracy	Top-5 Accuracy	Origin
alexnet	AlexNet	61,100,840	0.5492	0.7803	Converted from pytorch vision
densenet121	DenseNet-121	8,062,504	0.7497	0.9225	Converted from pytorch vision
densenet161	DenseNet-161	28,900,936	0.7770	0.9380	Converted from pytorch vision
densenet169	DenseNet-169	14,307,880	0.7617	0.9317	Converted from pytorch vision
densenet201	DenseNet-201	20,242,984	0.7732	0.9362	Converted from pytorch vision
inceptionv3	Inception V3 299x299	23,869,000	0.7755	0.9364	Converted from pytorch vision
mobilenet0.25	MobileNet 0.25	475,544	0.5185	0.7608	Trained with script
mobilenet0.5	MobileNet 0.5	1,342,536	0.6307	0.8475	Trained with script
mobilenet0.75	MobileNet 0.75	2,601,976	0.6738	0.8782	Trained with script
mobilenet1.0	MobileNet 1.0	4,253,864	0.7105	0.9006	Trained with script
mobilenetv2_1.0	MobileNetV2 1.0	3,539,136	0.7192	0.9056	Trained with script
mobilenetv2_0.75	MobileNetV2 0.75	2,653,864	0.6961	0.8895	Trained with script
mobilenetv2_0.5	MobileNetV2 0.5	1,983,104	0.6449	0.8547	Trained with script
mobilenetv2_0.25	MobileNetV2 0.25	1,526,856	0.5074	0.7456	Trained with script

You can learn filters well only when you have enough training samples, 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 – loading weights does precisely that.

- Finetuning preinitializes your network to some features which were good on another task.
- empirical evidence: low-level features in deep networks learnt over wide and general tasks can be reused for many other tasks, even with strange color distributions or geometrical tasks (see Pennys paper on mouse authentication)

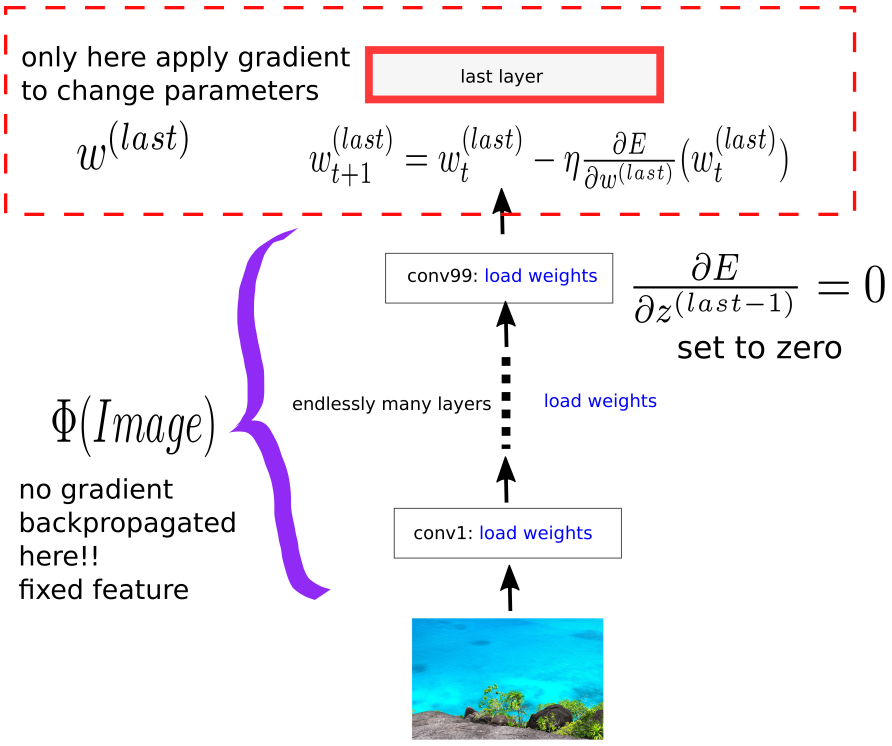
In theory a good initialization from finetuning can be destroyed (when trained too long with too little samples) – but in practice backpropagating gradients changes the highest level weights most (due to vanishing gradients), so that – at the beginning of training – the weights in the update layer adapt faster to what one wants to learn – and the overfitting by changing lower layer weights to bad optima sets in only later.

Finetuning has a special case: when the number of training data is very small, then one may want to retrain only the last 1-2 layers. This can be achieved by forcibly set $dE/dz = 0$ for activations below the layers that one wants to retrain. This has the advantage, that one can precompute the features for each image for the last layer for which one does not change weights. However this is only possible, if one does not use random image augmentations. So here one has a time vs efficiency tradeoff, I tried to let you see the more general case

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 backpropagating gradients beyond the highest layer. That means: we update weights only at the very last layer. The google finetuning tutorial shows this.

Here an example when you retrain only the last layer as an extreme case of finetuning. This is often shown in tutorials, but you need to check, in practice training all layers is often better, than training only the last layer.



2 Fine Tuning in practice

One way is to load the model fully, then delete/overwrite all parts which you do not need. See `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`

```
model_ft = models.resnet18(pretrained=True) #loads model
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2) #overwrites fully connected layer model_ft.fc to predic
```

The layer names are model-specific.

Another way is to only load what you really need:
`https://pytorch.org/tutorials/beginner/saving_loading_models.html#warmstarting-model-using-parameters-from-a-different-model`
`https://pytorch.org/docs/stable/_modules/torch/nn/modules/module.html#Module.load_state_dict`
or

```
class yourresnet(models.ResNet):
```

```
    def load_state_dict2(self, loaded_state_dict, ignored_keys=[]):
        """Copies parameters and buffers from :attr:‘state_dict‘ into
        this module and its descendants. The keys of :attr:‘state_dict‘ must
        exactly match the keys returned by this module’s :func:‘state_dict()‘
        function.
        Arguments:
            state_dict (dict): A dict containing parameters and
                persistent buffers.
        """
        print(’here’)
        own_state = self.state_dict()

        for name, param in loaded_state_dict.items():
            #print(’at name’,name)
            if name not in ignored_keys:       # changed here
                if name not in own_state:
                    raise KeyError(’unexpected key "{}" in loaded_state_dict’
                                   .format(name))
                if isinstance(param, torch.nn.Parameter):
                    # backwards compatibility for serialized parameters
                    param = param.data
                own_state[name].copy_(param)
         #  changed here
        missing = set(own_state.keys()) - set(loaded_state_dict.keys()) - set(ignored_keys)
        if len(missing) > 0:
            print(’missing keys in loaded_state_dict: "{}"’.format(missing))
```

Note that this routine may need to be adapted to some special cases, see example the densenet, where some patterns are renamed.

If you want to train only the last layer of a resnet:

```
model = models.resnet18(pretrained=True) #loads model

#
#many lines of code here
#
# assume model.fc is the last layer
#
  optimizer_ft = optim.SGD(model.fc.parameters(), lr=0.001, momentum=0.9)
```

If you want to train all last layers of a resnet:

```
model = models.resnet18(pretrained=True) #loads model
```

```
#
#many lines of code here
#
# assume model.fc is the last layer
#
  optimizer_ft = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

if only some parameters of any network, use `model.named_parameters()` to list and filter them:

```
  params_to_update = []
  print("Params to learn:")

  for name,param in model.named_parameters():
    if  (param.requires_grad == True) and ( yourwhatevercondition(name,param) ):
      params_to_update.append(param)
      print("\t",name)

  optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```