Assignment 1

Q4. Explain what a residual network is, and the basic motivation for using it. Also explain what the main elements of resnet34 are and resnet50. How many layers, how many neurons total, how many weights; and then anything else you want to say.

What problem does ResNet solve?

Most convolutional neural nets demonstrate a high performance on image classification. Research shows that deeper the networks, better the performance. However, deeper networks take longer to train and sometimes don't converge. Additionally, with increasing network depth, accuracy gets saturated and degrades rapidly. This is one of the main reasons that most CNNs use SGD and try to solve for the weights using back propagation.

In 2015, Kaiming He and his colleagues at Microsoft Research Asia ran into similar problems and introduced the solution of residual networks. The key difference between residual neural nets and conventional CNNs is the way in which the output of previous layers connects to the output of new layers.

Why is ResNet popular?

One of the reasons why ResNet gained popularity is because of over fitting. Larger networks can model more complex problems, but the risk of overfitting is higher too. Residual networks, however, are smaller networks with fewer parameters and don't overfit as much. In a residual setup, you would not only pass the output of layer 1 to layer 2 and on, but you would also add up the outputs of layer 1 to the outputs of layer 2. It's convergence property as discussed in the previous paragraph is also another reason for its popularity.

What Is A Plain Neural Network?

The convolutional layers mostly have 3×3 filters and follow two simple design rules:

- For the same output feature map size, the layers have the same number of filters
- If the feature map size is halved, the number of filters is doubled to preserve the time complexity per layer.

Downsampling is performed directly on convolutional layers which have a stride of 2. The network ends with a global average pooling layer and a 1,000-way fully-connected layer with softmax.

How do ResNets work?

The main goal of the residual network is to build a deeper neural network. We can have two intuitions based on this:

- As we keep going deeper into implementing large number of layers, one should make sure not to degrade the accuracy and error rate.
- Keep learning the residuals to match the predicted with the actual

Kaiming He, took the idea of reducing the degradation that happens in deeper networks by using residual networks. In his paper [1] he talks at length about how deeper nets accuracy is saturated, and that this isn't caused by overfitting but adding more layers to deep models which in turn lead to higher training error. The degradation indicates that not all systems are easy to optimize. By adding layers that are "identity mapping", and the other layers are copied from the learned shallower model.

Mathematically, a residual is defined as the error in a result. In He's paper, the residual has been defined as I(x) = H(x) - F(x)

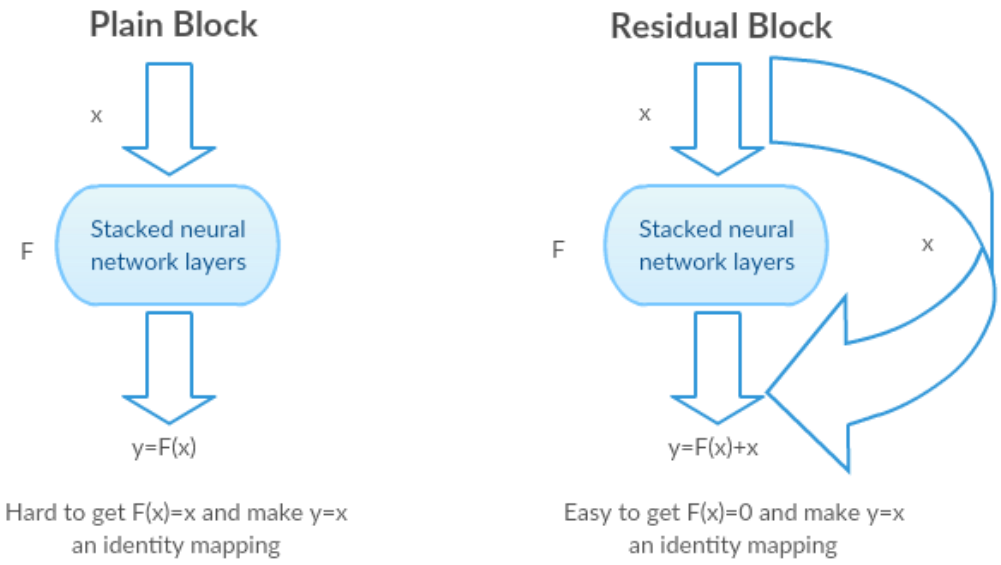
where I(x) is identity mapping which is I(x) =x,

H(x) the desired mapping

F(x) the mapping which network layers where able to achieve according to their inputs.

H(x) = F(x) +I(x) is our objective and only F(x) is changing every time assuming H(x) and I(x) as constants.

A residual network learns from "residuals" as opposed to only learning contributing features. In other words, a ResNet subtracts feature information from the input of a layer to learn about the residuals. (Refer to the image below)



During training stage, the residual network alters the weights until the output is equivalent to the identity function. In the process the outcome of residual function eventually becomes 0 and the input features gets mapped onto the hidden layers. Therefore, the error correction is not required. In turn the identity function helps in building a deeper network. The residual function then maps the identity, weights and biases to fit the actual value.

What are its limitations?

One major limitation of ResNets is the number of samples you can feed to it. Most cloud computing platforms limit the number of input samples to 80K for ResNets and hence other algorithms like VGG16 / VGG-19are more popular when

ResNet34 vs ResNet50

Following is the architecture of ResNet-34 and ResNet-50

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

	ResNet34	ResNet50
Layers	34	50
Neurons	8,746	23,818
Weights	25.6M	98.6 M

Note:

i. Weight and node calculations are done in excel sheet based on architecture shown above. Please refer here for more details.

ii. For calculating neurons, I have assumed that for a CNN, neurons are spread across 3 dimensions – width, height and depth

Q5. Transfer learning using Fast.ai and create_cnn: Please explain how pretrained resnet34 is modified to get the network that the notebook ultimately trains (i.e., explain what are the last layers that are added).

A CNN usually has the following layers:

1. Input layer—a single raw image is given as an input. For a RGB image its dimension will be AxBx3, where 3 represents the colours Red, Green and Blue
2. A convolution layer - a convolution layer is a matrix of dimension smaller than the input matrix. It performs a convolution operation with a small part of the input matrix having same dimension. The sum of the products of the corresponding elements is the output of this layer.
3. ReLU or Rectified Linear Unit—ReLU is mathematically expressed as max(0,x). It means that any number below 0 is converted to 0 while any positive number is allowed to pass as it is.
4. Maxpool—Maxpool passes the maximum value from amongst a small collection of elements of the incoming matrix to the output. Usually it is a square matrix.
5. Fully connected layer—The final output layer is a normal fully-connected neural network layer, which gives the output.

Usually the convolution layers, ReLUs and Maxpool layers are repeated number of times to form a network with multiple hidden layer commonly known as deep neural network.

The fast ai `create_cnn` factory method helps you to automatically get a pretrained model from a given architecture with a custom head that is suitable for your data.

create_cnn architecture

```
create_cnn(data:DataBunch,

arch:Callable,

cut:Union[int, Callable]=None,

pretrained:bool=True,

lin_ftrs:Optional[Collection[int]]=None,

ps:Floats=0.5,

custom_head:Optional[Module]=None,

split_on:Union[Callable, Collection[ModuleList], NoneType]=None,

bn_final:bool=False,

**learn_kwargs:Any) → Learner
```

The create_cnn method creates a `Learner` object from the `data` object and model inferred from it with the backbone given in `arch`. In our case we explore arch=resnet34 and resnet50.

In the code we explored, since we didn't define any "cut" value, this function automatically cuts off the last convolutional layer and add the following layers:

1. **an `AdaptiveConcatPool2d` layer,**
   AdaptiveConcatPool2d(sz:Optional[int]=None) :: Module

   Layer that concats AdaptiveAvgPool2d and AdaptiveMaxPool2d.

   The output will be 2*sz, or just 2 if sz is None.

   The AdaptiveConcatPool2d object uses adaptive average pooling and adaptive max pooling and concatenates them both. We use this because it provides the model with the information of both methods and improves performance. This technique is called adaptive because it allows us to decide on what output dimensions we want, instead of choosing the input's dimensions to fit a desired output size.

2. **a `Flatten` layer,**
   Flatten(full:bool=False) :: Module

   Flatten x to a single dimension, often used at the end of a model. full for rank-1 tensor

3. **blocks of [`nn.BatchNorm1d`, `nn.Dropout`, `nn.Linear`, `nn.ReLU`] layers.**
   These are layers created by pytorch as the final activation layer the output of which is the classes of the defined function

   The blocks while defining create_cnn are defined by the `lin_ftrs` and `ps` arguments.

   Specifically, the first block will have a number of inputs inferred from the backbone `arch` and the last one will have a number of outputs equal to `data.c` (which contains the number of classes of the data) and the intermediate blocks have a number of inputs/outputs determined by `lin_frts` (of course a block has a number of inputs equal to the number of outputs of the previous block).

The default is to have an intermediate hidden size of 512 (which makes two blocks `model_activation` -> 512 -> `n_classes`). If you pass a float then the final dropout layer will have the value `ps`, and the remaining will be `ps/2`. If you pass a list then the values are used for dropout probabilities directly.

The very last block doesn't have a `nn.ReLU` activation, to allow you to use any final activation you want (generally included in the loss function in pytorch). Also, the backbone will be frozen if you choose `pretrained=True` (so only the head will train if you call `fit`) so that you can immediately start phase one of training as described above.

Alternatively, you can define your own `custom_head` to put on top of the backbone. If you want to specify where to split `arch` you should so in the argument `cut` which can either be the index of a

specific layer (the result will not include that layer) or a function that, when passed the model, will return the backbone you want.

The final model obtained by stacking the backbone and the head (custom or defined as we saw) is then separated in groups for gradual unfreezing or differential learning rates.

Freezing and Unfreezing in Fastai:

- Fastai's freezing and unfreezing functions, help us leverage precomputed weights (from imagenet) and train the fully connected layer for the current problem.
- If a net is frozen, only the weights for the last layer(s) are updated (learned). The "core" is not changed. So, if you use ImageNet as a base and learn cats/dogs on top of it, only the cat/dog relevant layer(s) gets updated.
- Unfreezing allows the network to make changes to the weights of the layers at all.

Model training

- By default, the pretrained part (the body or "backbone") is frozen and we only train (change the weights) of the last layers (the custom head) that we added.
- Then we unfreeze the body (enabling any training of that at all)
- The way the changes to the weights are made (tiny, medium, large as you put it) are controlled by the learning rate. So, the learning rate is set to very small for the early layers (no need to change much), medium for the middle, full learning rate for the head - by choice.
- For resnet34 by default the last 8 layers are trained with new weights for the current data. Those are the avg pooling and the last Fully connected layers. We can view that in fastai by running the following code:

```
r = resnet34()

list(r.children())[:8]
```

Q6. Download a NOT pre-trained resnet34, and then by playing with the number of epochs and learning rates (possibly different learning rates across layers), see how low you can get the error. Can you get below 20%?

Please refer to notebook here

Q7. And for the main part of this HW: download (and label) your own data set of your choice, create a classification problem, and then use the main tools/ideas of this notebook to build a classifier. It does not need to be a multi-label classifier.

Classifying landmarks. 3 classes - Colosseum, UT Tower and Eiffel tower.

Please refer to notebook here