Training ResNet-152 on Imagenet

Vivian Pais

IIT Dharwad

20th August 2023

Content

Click section titles to jump to section.

Introduction to Neural Networks (1/6)

Consider an equation $f(a,b)$ as follows:

$$f(a,b) = ax + by + z$$

Here the variables multiplied with the inputs are called weights and the constant z is called bias.
Now if we know that a certain input tuple (1,9) gives an output 4, how do we learn?
Let us initialise x,y and z here as random values, say 3,2 and -5.
Now we have $f(1,9) = (3 * 1) + (9 * 2) + (-5) = 16$.
Intuitively we can see that reducing the value of x here will get us closer to our goal.
If we change x from 3 to -1 we get $f(1,9) = 12$, which is closer to our goal of 4.
So how do we systematically manipulate our variables to get the output closer to our goal?

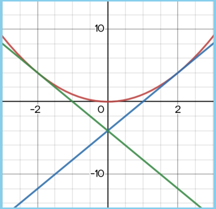
Introduction to Neural Networks (2/6)

We define a loss function L.
Here the loss function can be squared error.

$$L = (f(1,9) - 4)^2$$
$$L = x^2 + 81y^2 + z^2 + 18xy + 18yz + 2xz$$

The value obtained by filling in the proper values in the loss function will be called as loss.
Now we differentiate L with respect to x,y and z to find how they affect the loss function.
$dL/dx = 2x + 18y + 2z = 37$
$dL/dy = 162y + 18x + 18z = 288$
$dL/dz = 2z + 18y + 2x = 32$
These values are called gradients.

Introduction to Neural Networks (3/6)



For this equation, $y = x^2$, there are two tangents at -2 and 2 respectively.
The slope of tangent at -2 is negative and the slope of tangent at 2 is positive.

Thus to minimize the loss function, we can see that we need to subtract the derivative of every variable multiplied by a learning factor from the variable.
Now we update the weights x, y and z using learning factor or learning rate of 0.001.
$x = 3 - 0.001 * 37$
$y = 2 - 0.001 * 288$
$z = -5 - 0.001 * 32$
Now $f(1, 9) = (1 * 2.963) + (9 * 1.712) + (-5.032) = 13.339$
This has improved our result from 16 to 13.339 which is closer to 4.

Introduction to Neural Networks (4/6)

But how do we find gradients for complex equations?
Consider the following:
$x = w^2, y = 2x^3, z = 4y^2$
Now how can we calculate how w, x and y affect z?
$z = 4y^2 = 16x^6 = 16w^{12}$
So $dz/dy = 8y$,$dz/dx = 96x^5$,$dz/dw = 192w^{11}$
But this can be cumbersome when the equations are as complex as the ones we will be using in our neural networks.
Instead we can do the following:
$dz/dy = 8y$,$dz/dx = (dz/dy) * (dy/dx) = 8 * 2x^3 * 6x^2 = 96x^5$,
$dz/dw = (dz/dx) * (dx/dw) = 96 * w^{10} * 2w = 192w^{11}$
Now we use these gradients to manipulate w, x and y to obtain the required z.
This is called backpropagation, i.e. finding how every weight affects the loss and updating the weights appropriately.

Introduction to Neural Networks (5/6)

Training a neural network consists of the following steps:

- Forward pass : Calculating the value of the loss for a certain (input,required output) pair.
- Backward pass: Calculation of the gradients.
- Updating the weights.

A neural network consists of neurons and layers.
A neuron is an equation like the equation given above which takes every output of the previous layer as input, and gives one output.
A layer is a collection of neurons.

Introduction to Neural Networks (6/6)

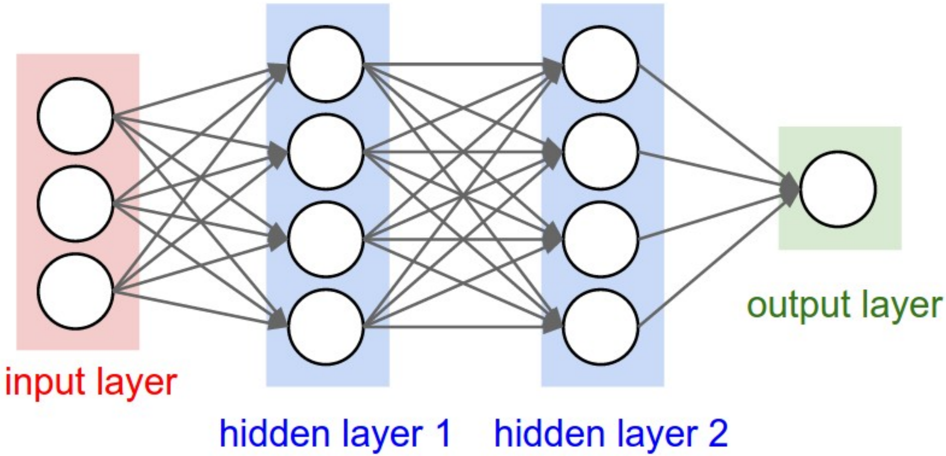


Image credits : `https://cs231n.github.io/convolutional-networks/`

Downloading of Imagenet Dataset (1/1)

Install kaggle using pip3:

```
pip3 install kaggle
```

Create an account on kaggle and login.

Go to account tab on user profile and select 'Create API Token'.

Go to the folder containing downloaded file using terminal and type:

```
mv kaggle.json ~/.kaggle/kaggle.json
```

Now go the location where you want to download the 167.62GB ImageNet dataset and type:

```
kaggle competitions download -c imagenet-object-localization-challenge
```

After the download is completed go to ILSVRC $\rightarrow$ Data $\rightarrow$ CLS-LOC $\rightarrow$ val and type:

```
wget https://raw.githubusercontent.com/soumith/imagenetloader.torch/master/valprep.sh
chmod 755 valprep.sh
./valprep.sh
rm valprep.sh
```

Now your ImageNet dataset is ready for use.

ResNet Model (1/11)

Various operations performed on input matrix

The blocks and layers of ResNet consist of the following smaller elements:

1. Convolutions
2. Average Pooling
3. Max Pooling
4. ReLU and Leaky ReLU
5. Batch Normalization

ResNet Model (2/11)

Convolutions

Kernel:

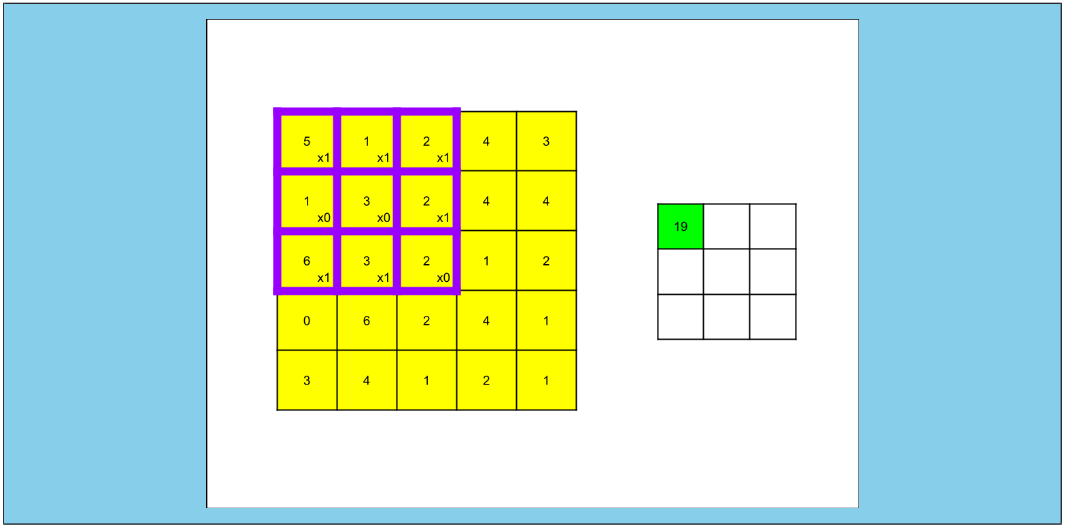
We use small grids of weights, usually of size 3x3 or 5x5, place them on the input matrix, multiply the values of the grid with the corresponding values of the input matrix behind the grid and sum the results. This gives us one value and this forms one cell of the output matrix. Then the kernel is moved to the right and the same process is repeated for the whole row and then for all the rows.

Stride:

Stride determines how much we move the kernel across the input matrix. If stride is 1, the kernel moves one step at a time, if the stride is 2, it moves 2 steps at a time and so on.

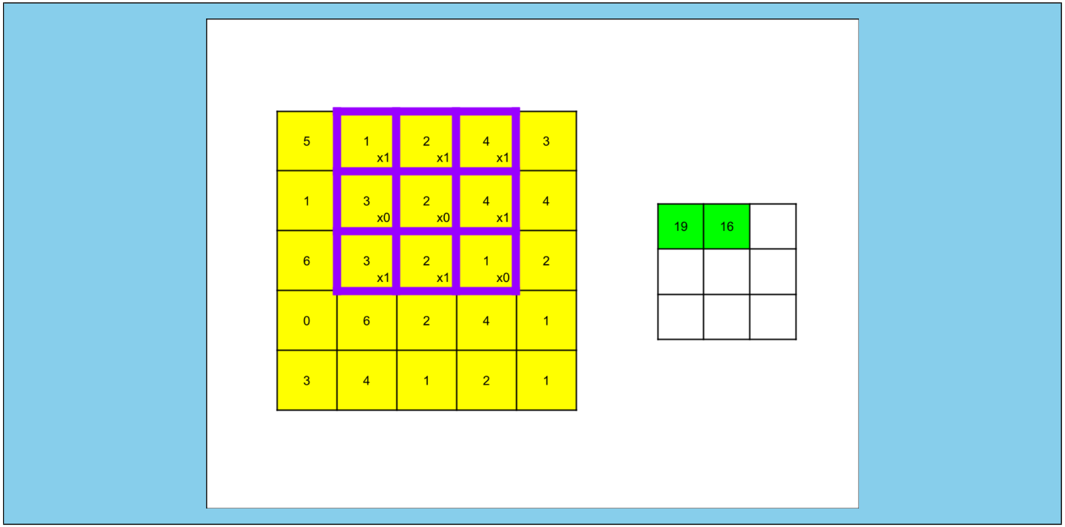
ResNet Model (3/11)

Convolutions



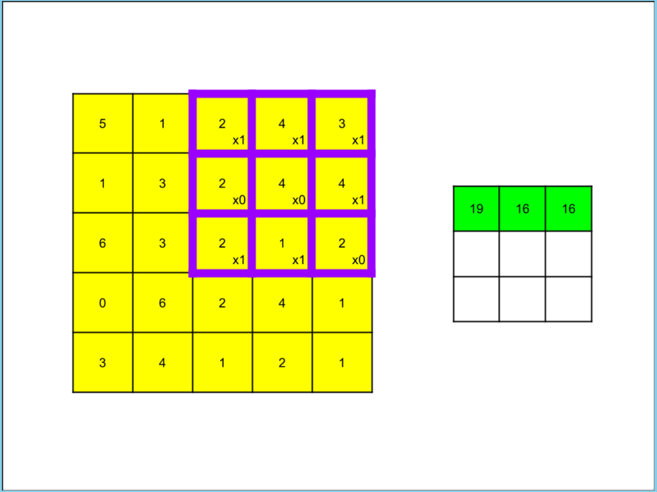
ResNet Model (3/11)

Convolutions



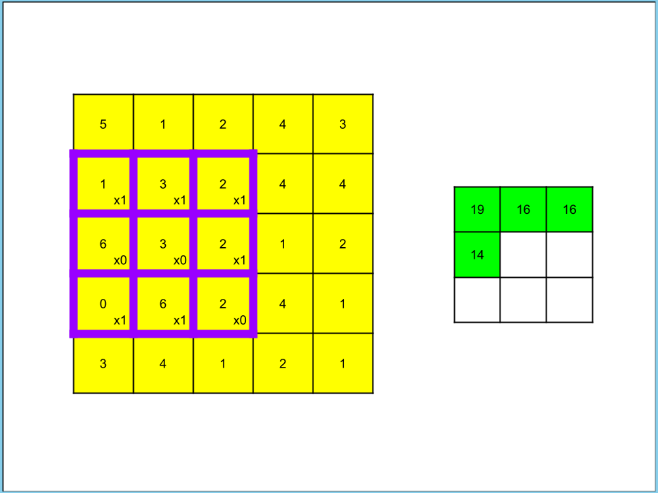
ResNet Model (3/11)

Convolutions



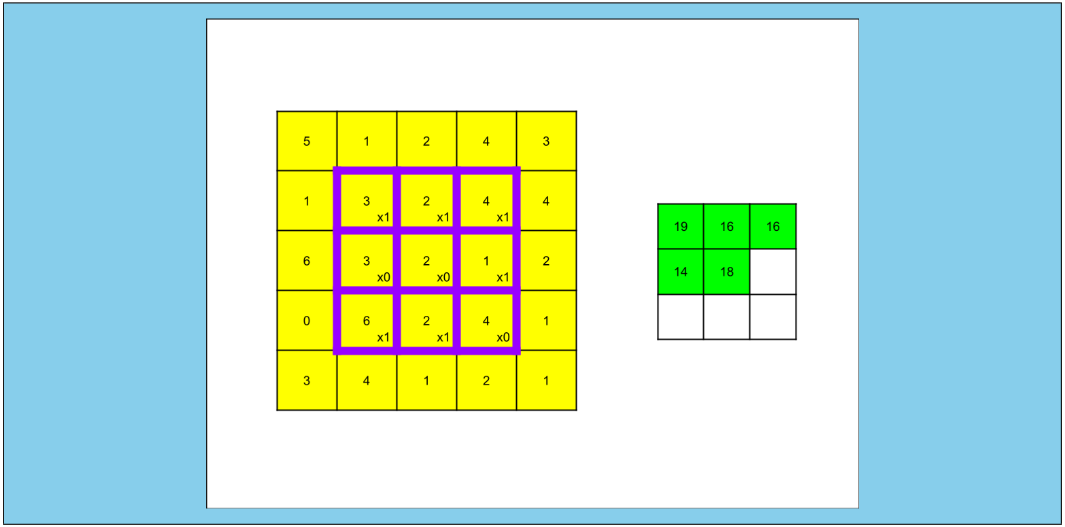
ResNet Model (3/11)

Convolutions



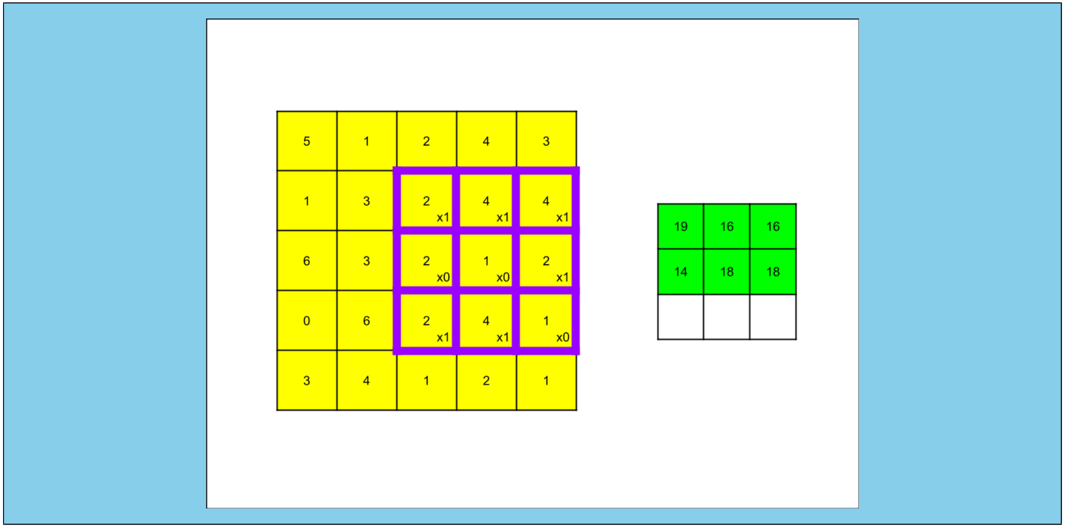
ResNet Model (3/11)

Convolutions



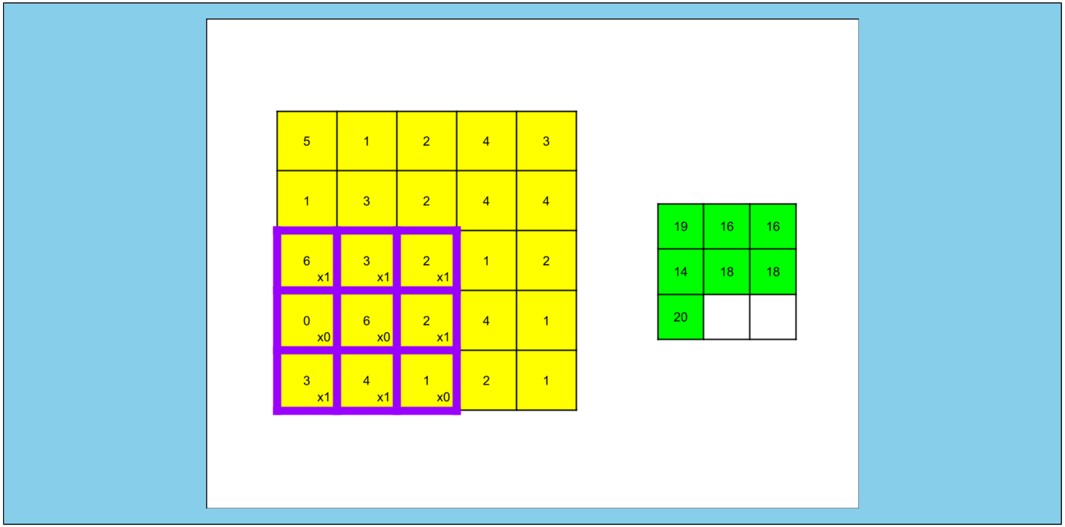
ResNet Model (3/11)

Convolutions



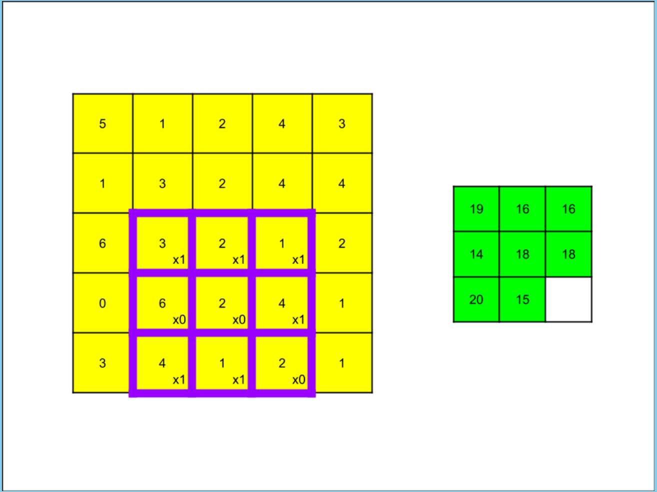
ResNet Model (3/11)

Convolutions

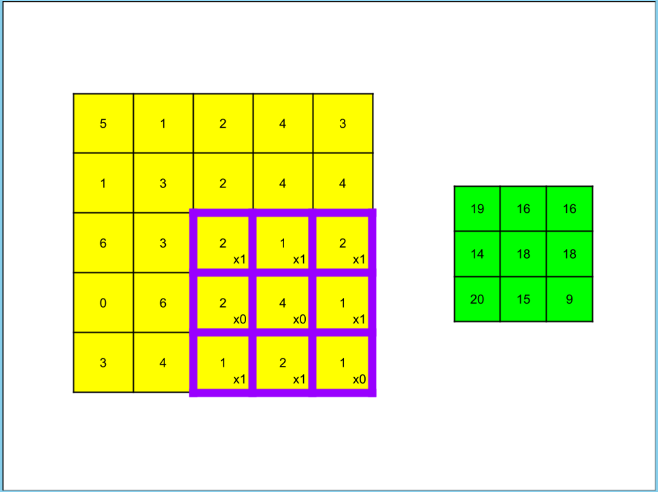


ResNet Model (3/11)

Convolutions

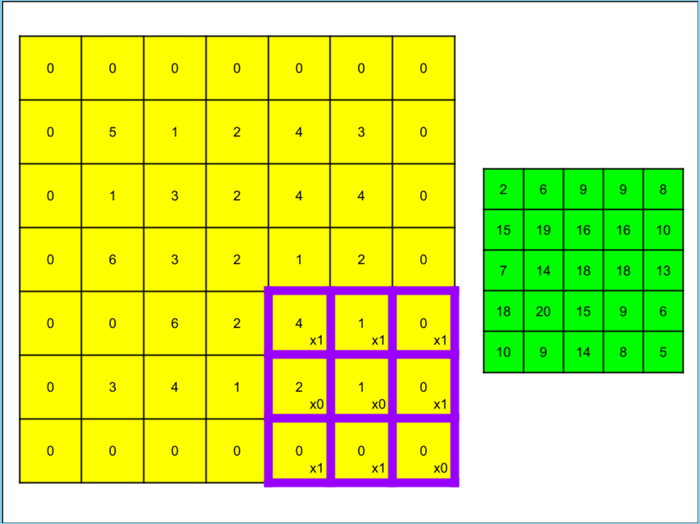


ResNet Model (3/11)
Convolutions



ResNet Model (4/11)

Padding



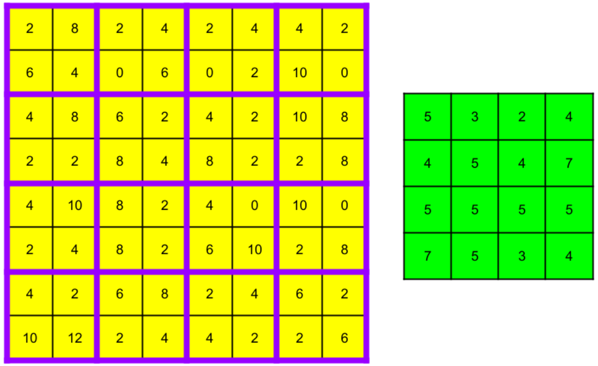
0	0	0	0	0	0	0
0	5	1	2	4	3	0
0	1	3	2	4	4	0
0	6	3	2	1	2	0
0	0	6	2	4 x1	1 x1	0 x1
0	3	4	1	2 x0	1 x0	0 x1
0	0	0	0	0 x1	0 x1	0 x0

2	6	9	9	8
15	19	16	16	10
7	14	18	18	13
18	20	15	9	6
10	9	14	8	5

We saw that applying convolution with kernel of size $k * k$ on image of size $n * n$, with stride=1, gives us output images of size: $(n - (2 * (k - 1)))*$ $(n - (2 * (k - 1)))$. Thus to maintain the size of the output image we extend the input image in each direction by $(k - 1)$.

ResNet Model (5/11)

Average Pooling



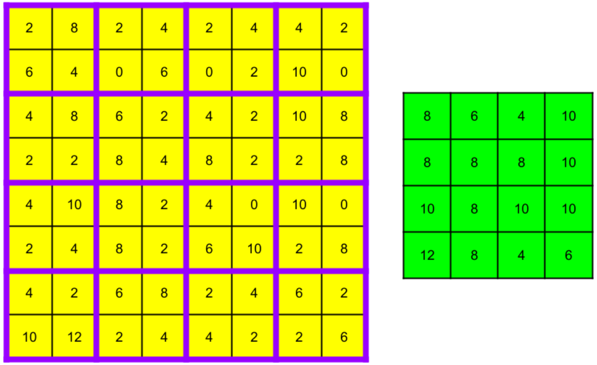
2	8	2	4	2	4	4	2
6	4	0	6	0	2	10	0
4	8	6	2	4	2	10	8
2	2	8	4	8	2	2	8
4	10	8	2	4	0	10	0
2	4	8	2	6	10	2	8
4	2	6	8	2	4	6	2
10	12	2	4	4	2	2	6

5	3	2	4
4	5	4	7
5	5	5	5
7	5	3	4

The average of all the values in the $k * k$ matrix where k is the kernel size passed as parameter to the function gives the value of corresponding cell in output matrix.The stride is equal to the kernel size by default but can be passed as parameter.

ResNet Model (6/11)

Max Pooling



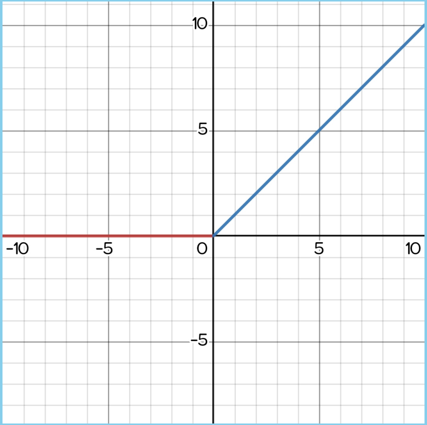
2	8	2	4	2	4	4	2
6	4	0	6	0	2	10	0
4	8	6	2	4	2	10	8
2	2	8	4	8	2	2	8
4	10	8	2	4	0	10	0
2	4	8	2	6	10	2	8
4	2	6	8	2	4	6	2
10	12	2	4	4	2	2	6

8	6	4	10
8	8	8	10
10	8	10	10
12	8	4	6

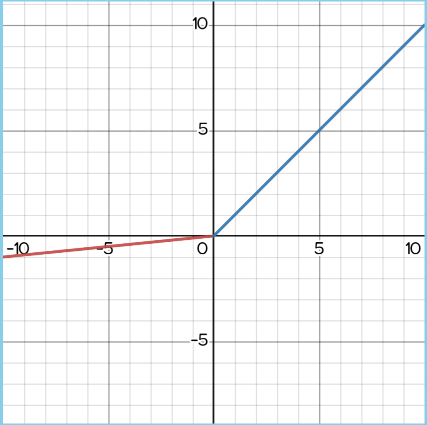
The maximum of all the values in the $k*k$ matrix where k is the kernel size passed as parameter to the function gives the value of corresponding cell in output matrix.The stride is equal to the kernel size by default but can be passed as parameter.

ResNet Model (7/11)

ReLU and Leaky ReLU



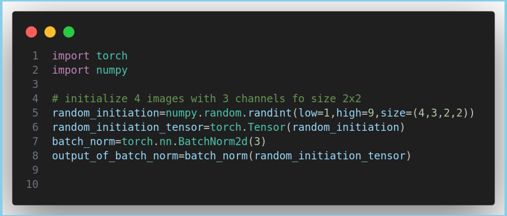
ReLU(x) returns max(0,x).



Leaky ReLU returns max(αx,x) where $\alpha << 1$.

ResNet Model (8/11)

Batch Normalization

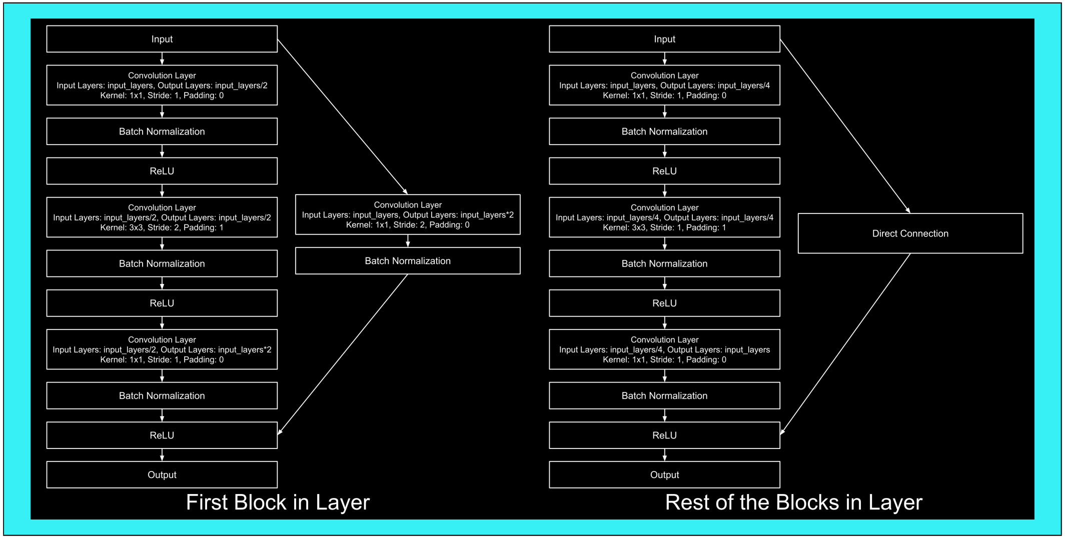


```python
import torch
import numpy

# initialize 4 images with 3 channels fo size 2x2
random_initiation=numpy.random.randint(low=1,high=9,size=(4,3,2,2))
random_initiation_tensor=torch.Tensor(random_initiation)
batch_norm=torch.nn.BatchNorm2d(3)
output_of_batch_norm=batch_norm(random_initiation_tensor)
```

Suppose there are 4 images with 3 channels (R,G,B) of size 2x2 each. Batch Normalization will calculate 6 values (2 for each channel) in this manner:
Mean and standard deviation of red channel = Mean and standard deviation of all the values in the red channel of all images. So it will calculate mean and standard deviation of the 16 values which will be: 4 values in first 2x2 of 3 2x2 matrices in the first image, four corresponding values from the second image, four corresponding values from the third image, four corresponding values from the fourth image.
Then it will subtract the mean from those 16 values then divide those 16 values by the standard deviation.
The batch norm will do this for all the three channels here.
Then after all this has been done, the 16 values in the three output channels of all the images are multiplied with γ of the corresponding channels and β of corresponding channels are added to each of those values. The 3 γ and 3 β are initiated as 1 and 0 when the BatchNorm2d layer is created.

ResNet Model (9/11)

Residual Block


Input
Convolution Layer
Input Layers: input_layers, Output Layers: input_layers/2
Kernel: 1x1, Stride: 1, Padding: 0
Batch Normalization
ReLU
Convolution Layer
Input Layers: input_layers/2, Output Layers: input_layers/2
Kernel: 3x3, Stride: 2, Padding: 1
Batch Normalization
ReLU
Convolution Layer
Input Layers: input_layers/2, Output Layers: input_layers*2
Kernel: 1x1, Stride: 1, Padding: 0
Batch Normalization
ReLU
Output
Convolution Layer
Input Layers: input_layers, Output Layers: input_layers*2
Kernel: 1x1, Stride: 2, Padding: 0
Batch Normalization
First Block in Layer
Input
Convolution Layer
Input Layers: input_layers, Output Layers: input_layers/4
Kernel: 1x1, Stride: 1, Padding: 0
Batch Normalization
ReLU
Convolution Layer
Input Layers: input_layers/4, Output Layers: input_layers/4
Kernel: 3x3, Stride: 1, Padding: 1
Batch Normalization
ReLU
Convolution Layer
Input Layers: input_layers/4, Output Layers: input_layers
Kernel: 1x1, Stride: 1, Padding: 0
Batch Normalization
ReLU
Output
Direct Connection
Rest of the Blocks in Layer

ResNet Model (10/11)

Downsampling the direct connection

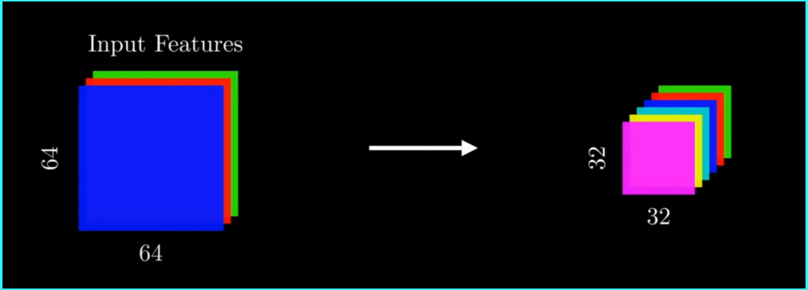
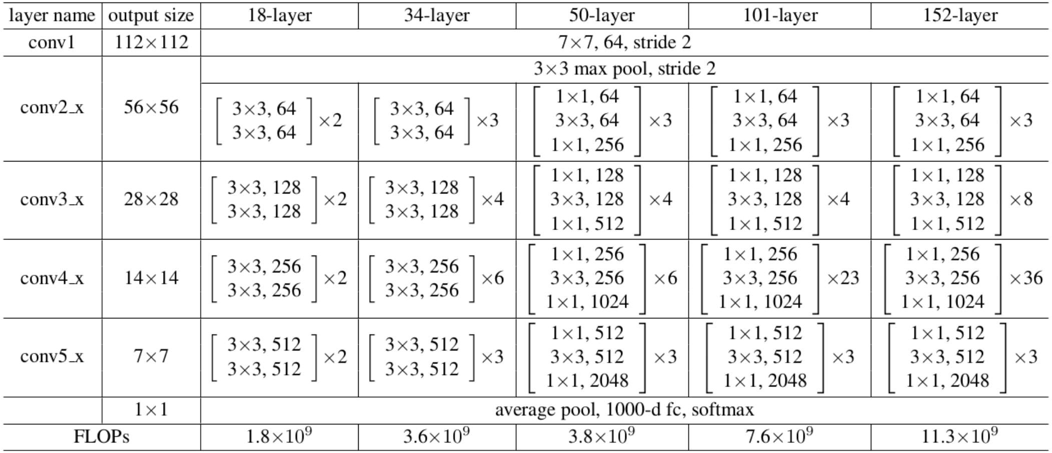


Image Credits: `https://www.youtube.com/watch?v=o_3mboe1jYI`

The size of the direct connection(number of channels and the size of those channels) in every block must be equal to the size of the output obtained after forward loop through the block, so that the output of the block and the direct connection can be added and passed on as the input of the next block. Thus the direct connection is passed though a convolutional layer with 1x1 kernel, stride 2 and output channels doubled whenever the size of the input is changed by the block across which the direct connection exists.

ResNet Model (11/11)

Various Models in ResNet

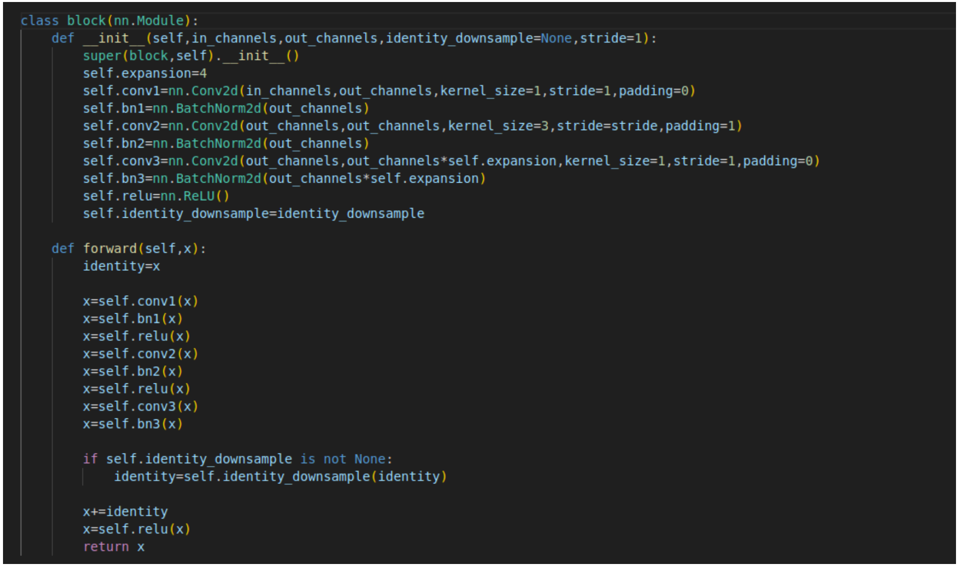


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
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

Image Credits: `https://arxiv.org/pdf/1512.03385.pdf`

ResNet Code Implementation (1/3)

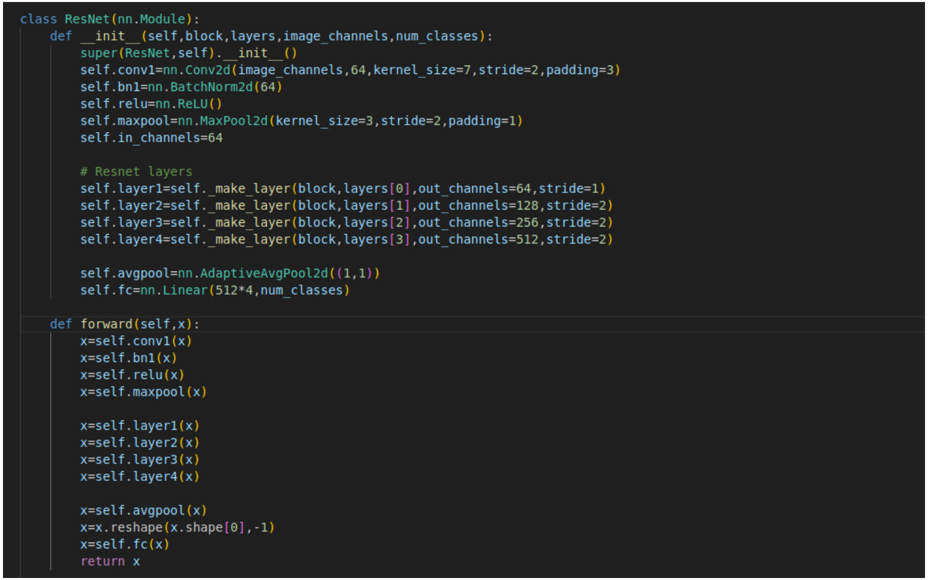
Code Implementation of Residual Block



```python
class block(nn.Module):
    def __init__(self,in_channels,out_channels,identity_downsample=None,stride=1):
        super(block,self).__init__()
        self.expansion=4
        self.conv1=nn.Conv2d(in_channels,out_channels,kernel_size=1,stride=1,padding=0)
        self.bn1=nn.BatchNorm2d(out_channels)
        self.conv2=nn.Conv2d(out_channels,out_channels,kernel_size=3,stride=stride,padding=1)
        self.bn2=nn.BatchNorm2d(out_channels)
        self.conv3=nn.Conv2d(out_channels,out_channels*self.expansion,kernel_size=1,stride=1,padding=0)
        self.bn3=nn.BatchNorm2d(out_channels*self.expansion)
        self.relu=nn.ReLU()
        self.identity_downsample=identity_downsample

    def forward(self,x):
        identity=x

        x=self.conv1(x)
        x=self.bn1(x)
        x=self.relu(x)
        x=self.conv2(x)
        x=self.bn2(x)
        x=self.relu(x)
        x=self.conv3(x)
        x=self.bn3(x)

        if self.identity_downsample is not None:
            identity=self.identity_downsample(identity)

        x+=identity
        x=self.relu(x)
        return x
```

ResNet Code Implementation (2/3)

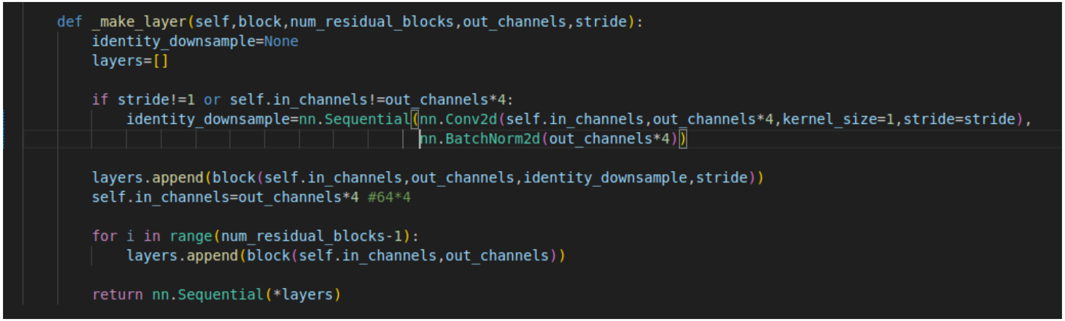
Init function and forward loop of ResNet



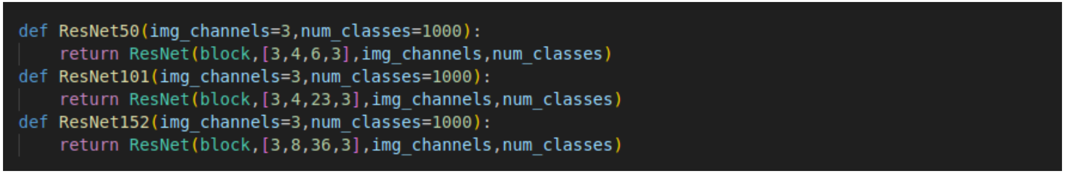
```python
class ResNet(nn.Module):
    def __init__(self,block,layers,image_channels,num_classes):
        super(ResNet,self).__init__()
        self.conv1=nn.Conv2d(image_channels,64,kernel_size=7,stride=2,padding=3)
        self.bn1=nn.BatchNorm2d(64)
        self.relu=nn.ReLU()
        self.maxpool=nn.MaxPool2d(kernel_size=3,stride=2,padding=1)
        self.in_channels=64

        # Resnet layers
        self.layer1=self._make_layer(block,layers[0],out_channels=64,stride=1)
        self.layer2=self._make_layer(block,layers[1],out_channels=128,stride=2)
        self.layer3=self._make_layer(block,layers[2],out_channels=256,stride=2)
        self.layer4=self._make_layer(block,layers[3],out_channels=512,stride=2)

        self.avgpool=nn.AdaptiveAvgPool2d((1,1))
        self.fc=nn.Linear(512*4,num_classes)

    def forward(self,x):
        x=self.conv1(x)
        x=self.bn1(x)
        x=self.relu(x)
        x=self.maxpool(x)

        x=self.layer1(x)
        x=self.layer2(x)
        x=self.layer3(x)
        x=self.layer4(x)

        x=self.avgpool(x)
        x=x.reshape(x.shape[0],-1)
        x=self.fc(x)
        return x
```

ResNet Code Implementation (3/3)

Definition of make_layer and definition of various ResNet Models



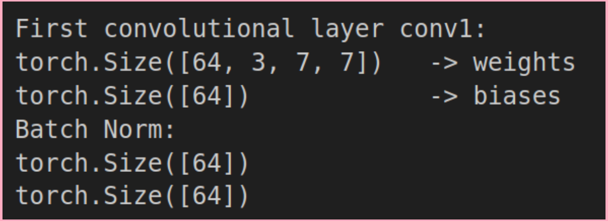
```python
    def _make_layer(self,block,num_residual_blocks,out_channels,stride):
        identity_downsample=None
        layers=[]

        if stride!=1 or self.in_channels!=out_channels*4:
            identity_downsample=nn.Sequential(nn.Conv2d(self.in_channels,out_channels*4,kernel_size=1,stride=stride),
                                              nn.BatchNorm2d(out_channels*4))

        layers.append(block(self.in_channels,out_channels,identity_downsample,stride))
        self.in_channels=out_channels*4 #64*4

        for i in range(num_residual_blocks-1):
            layers.append(block(self.in_channels,out_channels))

        return nn.Sequential(*layers)
```



```python
def ResNet50(img_channels=3,num_classes=1000):
    return ResNet(block,[3,4,6,3],img_channels,num_classes)
def ResNet101(img_channels=3,num_classes=1000):
    return ResNet(block,[3,4,23,3],img_channels,num_classes)
def ResNet152(img_channels=3,num_classes=1000):
    return ResNet(block,[3,8,36,3],img_channels,num_classes)
```

ResNet-152 Model Visualised through Model Parameters (1/6)

1st Layer



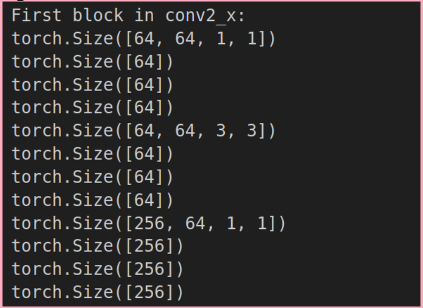
```
First convolutional layer conv1:
torch.Size([64, 3, 7, 7])    -> weights
torch.Size([64])             -> biases
Batch Norm:
torch.Size([64])
torch.Size([64])
```

The first convolutional layer takes in an image of 3 channels with size 224 x 224, uses stride 2 and gives output of 64 channels of size 112 x 112.

ResNet-152 Model Visualised through Model Parameters (2/6)

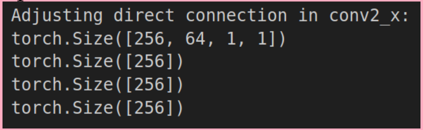
2nd Layer

Step 1:



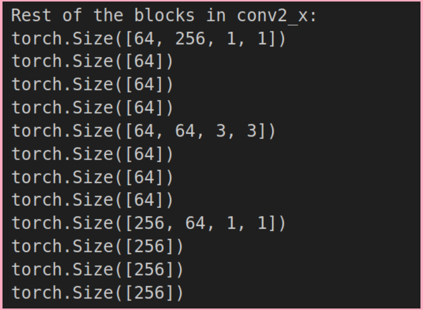
```
First block in conv2_x:
torch.Size([64, 64, 1, 1])
torch.Size([64])
torch.Size([64])
torch.Size([64])
torch.Size([64, 64, 3, 3])
torch.Size([64])
torch.Size([64])
torch.Size([64])
torch.Size([256, 64, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256])
```

Step 2:



```
Adjusting direct connection in conv2_x:
torch.Size([256, 64, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256])
```

Step 3:



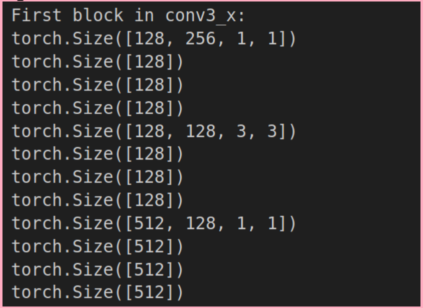
```
Rest of the blocks in conv2_x:
torch.Size([64, 256, 1, 1])
torch.Size([64])
torch.Size([64])
torch.Size([64])
torch.Size([64, 64, 3, 3])
torch.Size([64])
torch.Size([64])
torch.Size([64])
torch.Size([256, 64, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256])
```

This block appears 2 times and the output of this layer is 256 channels of size 56 x 56. The size is reduced by half because of maxpooling with stride 2 before this layer.

ResNet-152 Model Visualised through Model Parameters (3/6)

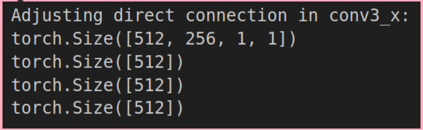
3rd Layer

Step 1:



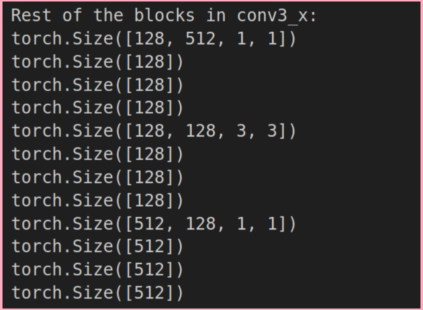
```
First block in conv3_x:
torch.Size([128, 256, 1, 1])
torch.Size([128])
torch.Size([128])
torch.Size([128])
torch.Size([128, 128, 3, 3])
torch.Size([128])
torch.Size([128])
torch.Size([128])
torch.Size([512, 128, 1, 1])
torch.Size([512])
torch.Size([512])
torch.Size([512])
```

Step 2:



```
Adjusting direct connection in conv3_x:
torch.Size([512, 256, 1, 1])
torch.Size([512])
torch.Size([512])
torch.Size([512])
```

Step 3:



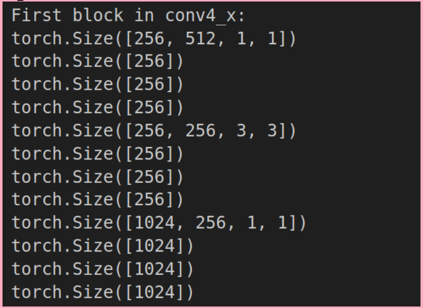
```
Rest of the blocks in conv3_x:
torch.Size([128, 512, 1, 1])
torch.Size([128])
torch.Size([128])
torch.Size([128])
torch.Size([128, 128, 3, 3])
torch.Size([128])
torch.Size([128])
torch.Size([128])
torch.Size([512, 128, 1, 1])
torch.Size([512])
torch.Size([512])
torch.Size([512])
```

This block appears 7 times and the output of this layer after Step 3 is 512 channels of size 28 x 28. The size is reduced by half because the convolution layer in Step 1 with kernel 3x3 is applied with stride 2.

ResNet-152 Model Visualised through Model Parameters (4/6)

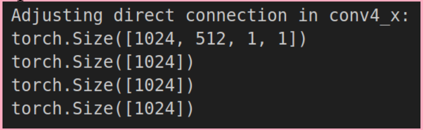
4th Layer

Step 1:



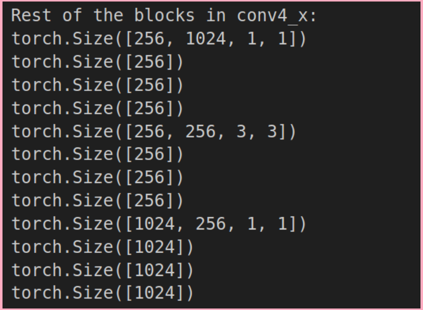
```
First block in conv4_x:
torch.Size([256, 512, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([256])
torch.Size([256])
torch.Size([1024, 256, 1, 1])
torch.Size([1024])
torch.Size([1024])
torch.Size([1024])
```

Step 2:



```
Adjusting direct connection in conv4_x:
torch.Size([1024, 512, 1, 1])
torch.Size([1024])
torch.Size([1024])
torch.Size([1024])
```

Step 3:



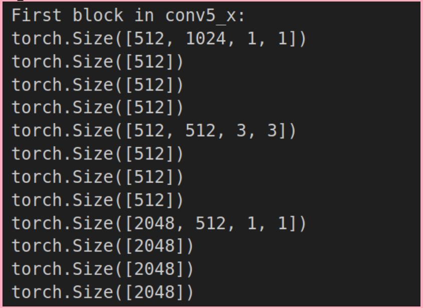
```
Rest of the blocks in conv4_x:
torch.Size([256, 1024, 1, 1])
torch.Size([256])
torch.Size([256])
torch.Size([256])
torch.Size([256, 256, 3, 3])
torch.Size([256])
torch.Size([256])
torch.Size([256])
torch.Size([1024, 256, 1, 1])
torch.Size([1024])
torch.Size([1024])
torch.Size([1024])
```

This block appears 35 times and the output of this layer after Step 3 is 1024 channels of size 14 x 14. The size is reduced by half because the convolution layer in Step 1 with kernel 3x3 is applied with stride 2.

ResNet-152 Model Visualised through Model Parameters (5/6)

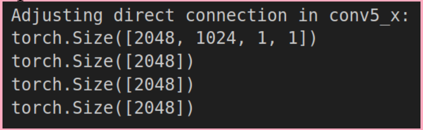
5th Layer

Step 1:



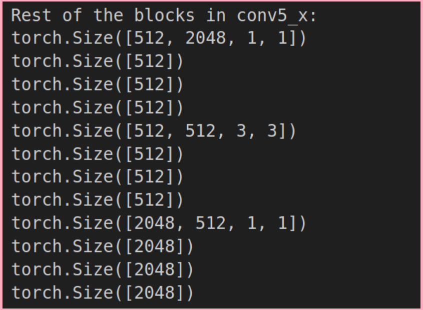
```
First block in conv5_x:
torch.Size([512, 1024, 1, 1])
torch.Size([512])
torch.Size([512])
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([512])
torch.Size([512])
torch.Size([2048, 512, 1, 1])
torch.Size([2048])
torch.Size([2048])
torch.Size([2048])
```

Step 2:



```
Adjusting direct connection in conv5_x:
torch.Size([2048, 1024, 1, 1])
torch.Size([2048])
torch.Size([2048])
torch.Size([2048])
```

Step 3:

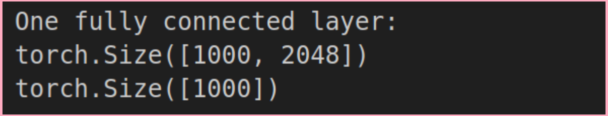


```
Rest of the blocks in conv5_x:
torch.Size([512, 2048, 1, 1])
torch.Size([512])
torch.Size([512])
torch.Size([512])
torch.Size([512, 512, 3, 3])
torch.Size([512])
torch.Size([512])
torch.Size([512])
torch.Size([2048, 512, 1, 1])
torch.Size([2048])
torch.Size([2048])
torch.Size([2048])
```

This block appears 2 times and the output of this layer after Step 3 is 2048 channels of size 7 x 7. The size is reduced by half because the convolution layer in Step 1 with kernel 3x3 is applied with stride 2.

ResNet-152 Model Visualised through Model Parameters (6/6)

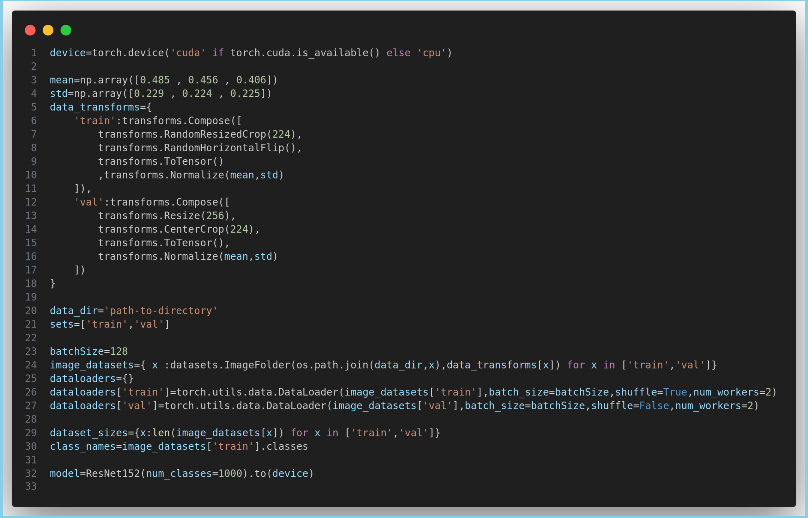
Fully Connected Layer



```
One fully connected layer:
torch.Size([1000, 2048])
torch.Size([1000])
```

The 2048 channels of size 7 x 7 are passed through adaptive average pooling layer with output size 1 x 1 and then the 2048 values are passed as input to a fully connected layer to get 1000 values corresponding to the 1000 output classes.

Data Processing (1/1)



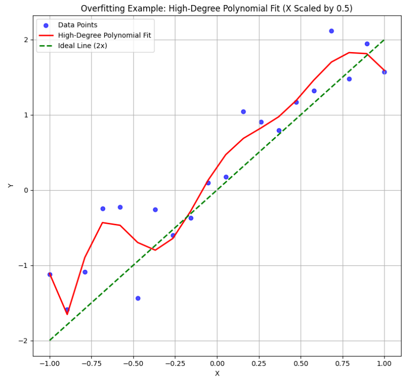
```python
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')

mean=np.array([0.485 , 0.456 , 0.406])
std=np.array([0.229 , 0.224 , 0.225])
data_transforms={
    'train':transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor()
        ,transforms.Normalize(mean,std)
    ]),
    'val':transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean,std)
    ])
}

data_dir='path-to-directory'
sets=['train','val']

batchSize=128
image_datasets={ x :datasets.ImageFolder(os.path.join(data_dir,x),data_transforms[x]) for x in ['train','val']}
dataloaders={}
dataloaders['train']=torch.utils.data.DataLoader(image_datasets['train'],batch_size=batchSize,shuffle=True,num_workers=2)
dataloaders['val']=torch.utils.data.DataLoader(image_datasets['val'],batch_size=batchSize,shuffle=False,num_workers=2)

dataset_sizes={x:len(image_datasets[x]) for x in ['train','val']}
class_names=image_datasets['train'].classes

model=ResNet152(num_classes=1000).to(device)

```

The mean and std values are the mean and standard deviation of the ImageNet Dataset.

Accuracy Enhancement Techniques (1/3)

Since the objective of ResNet-152 is image classification, we use torch.nn.CrossEntropyLoss() as the loss function.
However the depth of the neural network can cause some overfitting, so we need to introduce some measures to simplify the network, while maintaining accuracy.
Some methods include:

1. Regularisation
2. Drop-out



Overfitting: When the model learns the input data points perfectly, so that new points for which we need to predict the output give a high amount of loss.

Accuracy Enhancement Techniques (2/3)

Regularisation

Regularisation involves adding an element that depends on low powers of the weights to the loss function, so that the model is simplified.

L1 Regularisation:

L1 regularisation involves adding the sum of weights multiplied with some appropriate alpha to the loss function.
Loss += α * (Sum of the absolute value of weights)

L2 Regularisation:

L2 Regularisation involves adding the sum of squares of weights multiplied with some appropriate alpha to the loss function.
Loss += $\frac{\alpha}{2}$ * (Sum of squares of the weights)

This can be visualised as: If $a = b^2 + c^2$, $b = 1$ and $c = 9$, minimising a will lead to b and c getting closer. $b = 4$ and $c = 5$ will give $a = 41$ instead of the original $a = 82$. Thus when loss contains a factor containing low power of the weights, the weights are updated to be regularised instead of being at the extremes.

Accuracy Enhancement Techniques (3/3)

Drop-out

Drop-out involves turning off certain neurons to reduce complexity of the network.

During Training:

At each training step, the neurons are turned off with a probability p, that is, their output is multiplied by zero with a probability p before passing their output forward. Thus a random fixed-size batch of neurons is active at each training step. This is only done during training and not testing. The random selection occurs independently for each neuron and each batch and p is usually $0.2 - 0.5$.

During Testing:

All the neurons are used, but the outputs of all the neurons are multiplied by (1-p) so that the final sum is of the same order as if p fraction of neurons are turned off.

Drop-out is usually done in the fully connected layers towards the end of a neural network, in the layers connected to the final layer which calculates logits for all the output classes. However drop-out is not encouraged in networks with batch normalization (source: `https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/` under heading "Don't Use With Dropout")

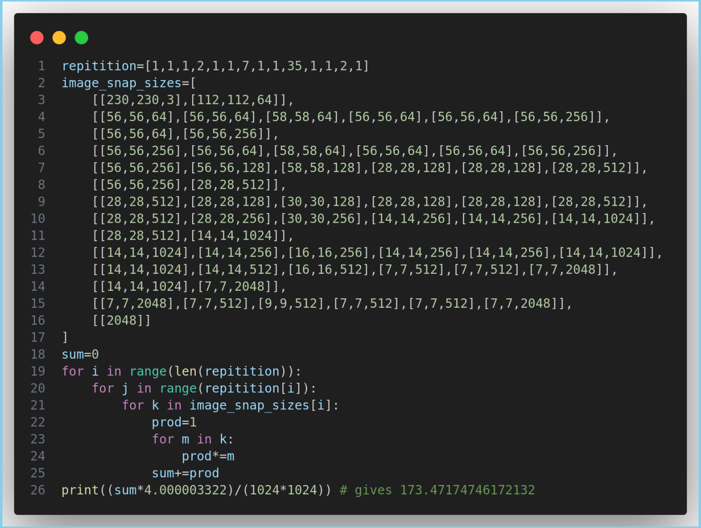
Usage of Resources (1/3)

For GPU: NVIDIA RTX A6000

Batch Size	GPU Memory Used	Time for training epoch
1	2460 MiB	—
2	2638 MiB	—
36	8576 MiB	2h 16m
48	10930 MiB	2h 12m
64	13854 MiB	2h 06m
80	18166 MiB	2h 01m
96	20558 MiB	2h 02m
120	23884 MiB	1h 58m
128	25118 MiB	1h 48m
256	49140 MiB	1h 48m

Usage of Resources (2/3)

GPU Memory Usage Per Image



```python
repitition=[1,1,1,2,1,1,7,1,1,35,1,1,2,1]
image_snap_sizes=[
    [[230,230,3],[112,112,64]],
    [[56,56,64],[56,56,64],[58,58,64],[56,56,64],[56,56,64],[56,56,256]],
    [[56,56,64],[56,56,256]],
    [[56,56,256],[56,56,64],[58,58,64],[56,56,64],[56,56,64],[56,56,256]],
    [[56,56,256],[56,56,128],[58,58,128],[28,28,128],[28,28,128],[28,28,512]],
    [[56,56,256],[28,28,512]],
    [[28,28,512],[28,28,128],[30,30,128],[28,28,128],[28,28,128],[28,28,512]],
    [[28,28,512],[28,28,256],[30,30,256],[14,14,256],[14,14,256],[14,14,1024]],
    [[28,28,512],[14,14,1024]],
    [[14,14,1024],[14,14,256],[16,16,256],[14,14,256],[14,14,256],[14,14,1024]],
    [[14,14,1024],[14,14,512],[16,16,512],[7,7,512],[7,7,512],[7,7,2048]],
    [[14,14,1024],[7,7,2048]],
    [[7,7,2048],[7,7,512],[9,9,512],[7,7,512],[7,7,512],[7,7,2048]],
    [[2048]]
]
sum=0
for i in range(len(repitition)):
    for j in range(repitition[i]):
        for k in image_snap_sizes[i]:
            prod=1
            for m in k:
                prod*=m
            sum+=prod
print((sum*4.000003322)/(1024*1024)) # gives 173.47174746172132
```

Layers like convolution and batch normalization operate on input matrices and hence the input matrices are required for gradient calculation. So we calculate the GPU Memory required per image as shown. Each value stored is a torch.float32 and its storage use varies slightly, so the storage required per image fluctuates a little. The CUDA context however allocates a fixed memory greater than 173MiB to account for fluctuating storage usage.

Usage of Resources (3/3)

GPU Memory Usage Per Image

The number of parameters in our model: 60,268,520.
Since our model is saved on the GPU, it will require storage for:
Saving 60,268,520 parameters.
Saving gradient values of 60,268,520 parameters.
Saving copies of the 60,268,520 parameters to update the parameters during backpropagation.
Each parameter uses around 4KiB.
Thus we need $\frac{(60{,}268{,}520*3*4)}{1024*1024} = 689.71$MiB.

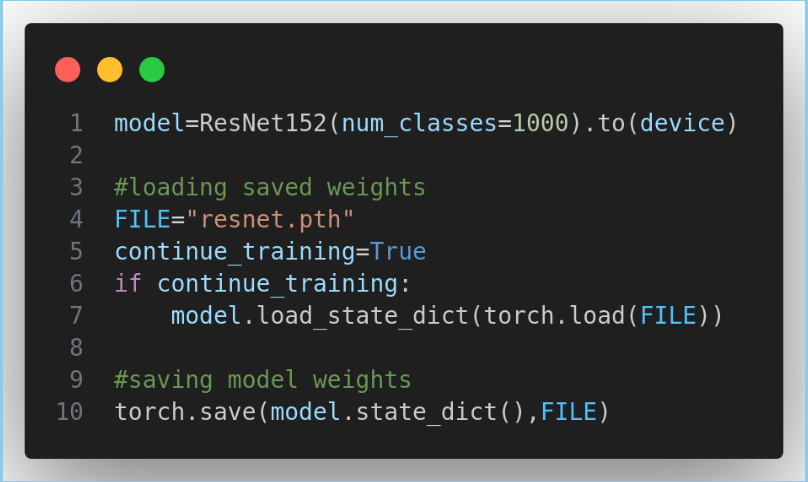
On running the resnet model with batch size of 1, and using torch.cuda.memory_allocated() to check the memory, it returned a value that fluctuated around 0.862GiB which can be confirmed to be 689.71MiB used by the model and 173.47MiB used by the image.

The torch method torch.cuda.memory_reserved() returns memory reserved, which is the memory used by the tensors plus the extra memory allocated to account for the fluctuation.

Running the command **nvidia-smi** however, usually shows extra memory usage of around 1-2 GiB. This is the GPU memory used by the CUDA context.

Thus the total GPU Memory Usage is around ((173+reserved_memory)*batch_size)MiB + 2−4GiB.

Saving our model (1/1)



```
model=ResNet152(num_classes=1000).to(device)

#loading saved weights
FILE="resnet.pth"
continue_training=True
if continue_training:
    model.load_state_dict(torch.load(FILE))

#saving model weights
torch.save(model.state_dict(),FILE)
```

The 60,268,520 parameter values are saved in a pth format file at the location specified by the FILE variable. Storage used = 231 MiB.