Outline

In this notebook I am going to describe the major steps toward createing a CCN from scratch as well as using transfer leaning to complete the Dog Breed classification project.

1. Initialization ¶

2. Define a basic CNN to start

3. Data Exploration and need for Augmentation

4. Tune CNN and hyper parameters

5. Transfer learning

6. Conclusion

Initialization:

```
The datasets are provided for us and devided into test,train and validation
 each have 113 folders corresponde to 113 dog breads.The label distribution
 label distribution  seems acceptable(
mean 6.272727 std 1.712509 min 3.000000 max 10.000000), although it could be
better considering number of samples.
```

[https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip)

[https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip)

In [69]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

In [70]:
```
dog_files_test = np.array(glob("dogImages/test/*/*"))
dog_files_train = np.array(glob("dogImages/train/*/*"))
dog_files_valid = np.array(glob("dogImages/valid/*/*"))

# print number of images in each dataset
print('There are %d total dog images in test.' % len(dog_files_test))
print('There are %d total dog images in train.' % len(dog_files_train
))
print('There are %d total dog images in valid.' % len(dog_files_valid
))
```

```
There are 836 total dog images in test.
There are 6680 total dog images in train.
There are 835 total dog images in valid.
```

In [71]:
```
import torch
from PIL import Image
import torchvision.transforms as transforms
import json
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [87]:
```
import torchvision.models as models
import torch
import torchvision.models as models
# define VGG16 model
VGG16 = models.vgg16(pretrained=True)
# check if CUDA is available
use_cuda = torch.cuda.is_available()
if not use_cuda:
    device = "cpu"
    print('Training on CPU')
else:
    device = torch.device("cuda:0")
    print('Training on GPU ...')
    print("Using",torch.cuda.get_device_name(device))
# move model to GPU if CUDA is available
#if use_cuda:
   # VGG16 = VGG16.cuda()
VGG16 = VGG16.to(device)
```

```
Training on GPU ...
Using Tesla V100-SXM2-16GB
```

Define a CNN for classification

    *Network architecture

There are 3 sets of two convlutional layer followed by a max pooling to control computinal complexity, the initial dept is 32 which woulde be doubeled after each set. It is inspired by VGG network architecture. drope out is applied (to avoid over fitting) before send the tresult to a 133 flate output layer(correspond to deg breeds). I am going to use this architecture with arbitary learning rat of 0.01 to see how it performs on a none augmented data, later if it is needed I am going to augment and compare the results. after that I am going to tune the model itself.

[https://arxiv.org/abs/1409.1556 (https://arxiv.org/abs/1409.1556)](https://arxiv.org/abs/1409.1556)

In [88]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    def __init__(self, n_classes, depth_1 = 32):
        super(Net, self).__init__()
        # Keep track of things
        depth_2 = depth_1 * 2
        depth_3 = depth_2 * 2
        # Max pooling layer
        self.pool = nn.MaxPool2d(2,2)
        # Conv set 1
        self.conv1_1 = nn.Conv2d(3,depth_1,3,stride = 1,padding = 1)
        self.conv1_2 = nn.Conv2d(depth_1,depth_1,3,stride = 1,padding
= 1)
        self.bn1_1 = nn.BatchNorm2d(depth_1)
        self.bn1_2 = nn.BatchNorm2d(depth_1)
        # Conv set 2
        self.conv2_1 = nn.Conv2d(depth_1,depth_2,3,stride = 1,padding
= 1)
        self.conv2_2 = nn.Conv2d(depth_2,depth_2,3,stride = 1,padding
= 1)
        self.bn2_1 = nn.BatchNorm2d(depth_2)
        self.bn2_2 = nn.BatchNorm2d(depth_2)
        # Conv set 3
        self.conv3_1 = nn.Conv2d(depth_2,depth_3,3,stride = 1,padding
= 1)
        self.conv3_2 = nn.Conv2d(depth_3,depth_3,3,stride = 1,padding
= 1)
        self.bn3_1 = nn.BatchNorm2d(depth_3)
        self.bn3_2 = nn.BatchNorm2d(depth_3)
        # Output correspond to number of dogs breds
        self.fc_out = nn.Linear(depth_3,n_classes)
        # Initialize weights
        nn.init.kaiming_normal_(self.conv1_1.weight, nonlinearity='re
lu')
        nn.init.kaiming_normal_(self.conv1_2.weight, nonlinearity='re
lu')
        nn.init.kaiming_normal_(self.conv2_1.weight, nonlinearity='re
lu')
        nn.init.kaiming_normal_(self.conv2_2.weight, nonlinearity='re
lu')
        nn.init.kaiming_normal_(self.conv3_1.weight, nonlinearity='re
lu')
        nn.init.kaiming_normal_(self.conv3_2.weight, nonlinearity='re
lu')

    def forward(self, x):
        # Conv 1
        x = F.relu(self.bn1_1(self.conv1_1(x)))
        x = F.relu(self.bn1_2(self.conv1_2(x)))
        x = self.pool(x)
        # Conv 2
        x = F.relu(self.bn2_1(self.conv2_1(x)))
        x = F.relu(self.bn2_2(self.conv2_2(x)))
```

```
        x = self.pool(x)
        # Conv 3
        x = F.relu(self.bn3_1(self.conv3_1(x)))
        x = F.relu(self.bn3_2(self.conv3_2(x)))
        x = self.pool(x)
        # reduce dimention
        x = x.view(x.size(0),x.size(1),-1)
        # And now max global pooling
        x = x.max(2)[0]
        # Output
        x = self.fc_out(x)
        return x
# instantiate the CNN
model_scratch = Net(n_classes)

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

Data Exploration and need for Augmentation

There are images with different size that we need to crop/resize to the same size at first step. As later I am going to cmpare it with pretrained model, I am going to apply the same size and normilization that they are trained with here as well. Now before going furthur, I want to observe how it is the learning/validation error ratio is for 10 epoches. I am going to augment and check it again to see if there is a meanigful improvment on overfitting issue. specially given the fact that we dont have much big of a trainig data set.

In [74]:
```
import os
from torchvision import datasets

###  No Augmentation Perormance review
transform_resize = 224
transform_crop = 224
data_directory = "dogImages"

print("load image data ... ")
# define transforms for the training data and testing data
train_transforms = transforms.Compose([transforms.CenterCrop(transfor
m_crop), transforms.ToTensor()])

test_transforms = transforms.Compose([transforms.CenterCrop(transform
_crop), transforms.ToTensor()])


# pass transforms in here, then run the next cell to see how the tran
sforms look
train_data = datasets.ImageFolder( data_directory + '/train', transfo
rm=train_transforms )
test_data = datasets.ImageFolder( data_directory + '/test', transform
=test_transforms )
valid_data = datasets.ImageFolder( data_directory + '/valid', transfo
rm=test_transforms )

# ---- print out some data stats ----
print('  Number of train images: ', len(train_data))
print('  Number of test images:  ', len(test_data))
print('  Number of valid images: ', len(valid_data))
# ----------------------------------

trainloader = torch.utils.data.DataLoader( train_data, batch_size=32,
shuffle=True )
testloader = torch.utils.data.DataLoader( test_data, batch_size=16 )
validloader = torch.utils.data.DataLoader( valid_data, batch_size=16
)

# create dictionary for all loaders in one
loaders_scratch = {}
loaders_scratch['train'] = trainloader
loaders_scratch['valid'] = validloader
loaders_scratch['test'] = testloader
n_classes = len(train_data.classes)
```

```
load image data ...
  Number of train images:  6680
  Number of test images:   836
  Number of valid images:  835
```

In [89]:
```
import torch.optim as optim
from torch.optim.lr_scheduler import ReduceLROnPlateau

### TODO: select loss function
#criterion_scratch = None

param_learning_rate = 0.01 # I tewaked it manualy
criterion_scratch = nn.CrossEntropyLoss() # Based on pytorch recomand
ation for multi class clasification with high precision and recall

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=param_le
arning_rate)
```

In [76]:
```
# the following import is required for training to be robust to trunc
ated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, s
ave_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation los
s
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordin
gly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (lo
ss.data - train_loss))
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inpu
ts to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respec
t to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (los
s.data - train_loss))
            train_loss += loss.item()*data.size(0)
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            # calculate the batch loss
```

```
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += loss.item() * data.size(0)


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss:
{:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            #print('Validation loss decreased ({:.6f} --> {:.6f}).  S
aving model ...'.format(valid_loss_min, valid_loss))
            print('  Saving model ...')
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
        else:
            print("")

    # return trained model
    return model


# train the model
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_s
cratch,
                      criterion_scratch, use_cuda, 'none_augment/mode
l_noaugmentation_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('none_augment/model_noaugmen
tation_scratch.pt'))
```

```
Epoch: 1        Training Loss: 33439.447720     Validation Loss: 406
8.734119
  Saving model ...
Epoch: 2        Training Loss: 31758.934265     Validation Loss: 395
0.578050
  Saving model ...
Epoch: 3        Training Loss: 30885.299099     Validation Loss: 386
9.568702
  Saving model ...
Epoch: 4        Training Loss: 30106.976276     Validation Loss: 381
4.408498
  Saving model ...
Epoch: 5        Training Loss: 29459.478176     Validation Loss: 379
3.919446
  Saving model ...
Epoch: 6        Training Loss: 28843.910347     Validation Loss: 372
6.464614
  Saving model ...
Epoch: 7        Training Loss: 28305.996410     Validation Loss: 367
7.568949
  Saving model ...
Epoch: 8        Training Loss: 27765.342726     Validation Loss: 363
3.714413
  Saving model ...
Epoch: 9        Training Loss: 27162.707621     Validation Loss: 363
9.395047

Epoch: 10       Training Loss: 26672.503998     Validation Loss: 358
6.197377
  Saving model ...
```

Out[76]:
```
<All keys matched successfully>
```

Note1:

It apears that after epoch 8 validation loss stop decreasing and it would diverges from training loss, to avoid this I came up with an augemtaion like bellow for model to learn invarient representation of a dog, next I am going to test the laerning/validation error ratio for 10 epoches to see if there are improvment. Resize: centerCrop: random Horiziontal and random vertical flip: random rotation: normilization

In [78]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
transform_resize = 224
transform_crop = 224
data_directory = "dogImages"

print("load image data ... ")
# define transforms for the training data and testing data
train_transforms = transforms.Compose([transforms.Resize(transform_re
size),
                                       transforms.CenterCrop(transfor
m_crop),
                                       transforms.RandomHorizontalFli
p(),
                                       transforms.RandomVerticalFlip
(),
                                       transforms.RandomRotation(20),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485,
0.456, 0.406],
                                                            [0.229,
0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(transform_res
ize),
                                      transforms.CenterCrop(transform
_crop),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.
456, 0.406],
                                                           [0.229, 0.
224, 0.225])])


# pass transforms in here, then run the next cell to see how the tran
sforms look
train_data = datasets.ImageFolder( data_directory + '/train', transfo
rm=train_transforms )
test_data = datasets.ImageFolder( data_directory + '/test', transform
=test_transforms )
valid_data = datasets.ImageFolder( data_directory + '/valid', transfo
rm=test_transforms )

# ---- print out some data stats ----
print('  Number of train images: ', len(train_data))
print('  Number of test images:  ', len(test_data))
print('  Number of valid images: ', len(valid_data))
# -----------------------------------

trainloader = torch.utils.data.DataLoader( train_data, batch_size=32,
shuffle=True )
testloader = torch.utils.data.DataLoader( test_data, batch_size=16 )
validloader = torch.utils.data.DataLoader( valid_data, batch_size=16
```

```
)

# create dictionary for all loaders in one
loaders_scratch = {}
loaders_scratch['train'] = trainloader
loaders_scratch['valid'] = validloader
loaders_scratch['test'] = testloader
n_classes = len(train_data.classes)
```

```
load image data ...
  Number of train images:  6680
  Number of test images:   836
  Number of valid images:  835
```

In [79]:

```
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'none_augment/model_augmentation_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('none_augment/model_augmentation_scratch.pt'))
```

```
Epoch: 1        Training Loss: 29126.578182     Validation Loss: 3619.310997
  Saving model ...
Epoch: 2        Training Loss: 28206.389803     Validation Loss: 3512.117984
  Saving model ...
Epoch: 3        Training Loss: 27685.631363     Validation Loss: 3508.756071
  Saving model ...
Epoch: 4        Training Loss: 27081.482113     Validation Loss: 3452.839887
  Saving model ...
Epoch: 5        Training Loss: 26545.053951     Validation Loss: 3385.909094
  Saving model ...
Epoch: 6        Training Loss: 26136.892130     Validation Loss: 3415.109824

Epoch: 7        Training Loss: 25605.508467     Validation Loss: 3310.772789
  Saving model ...
Epoch: 8        Training Loss: 25193.762247     Validation Loss: 3280.700325
  Saving model ...
Epoch: 9        Training Loss: 24664.896971     Validation Loss: 3218.254964
  Saving model ...
Epoch: 10       Training Loss: 24299.343431     Validation Loss: 3181.319473
  Saving model ...
```

Out[79]:

```
<All keys matched successfully>
```

Note2:

A bit better but start overfitting after epoche 9, I am going to change it as bellow. I test it for 50 epoch to see hoe it performs.

In [96]:

```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
transform_resize = 224
transform_crop = 224
data_directory = "dogImages"

print("load image data ... ")
# define transforms for the training data and testing data

train_transforms = transforms.Compose([transforms.Resize(transform_re
size),
                                       transforms.RandomHorizontalFli
p(),
                                       #transforms.CenterCrop(transfo
rm_crop),
                                       transforms.RandomResizedCrop(t
ransform_resize, scale=(0.08,1), ratio=(1,1)),
                                       #transforms.RandomHorizontalFl
ip(),
                                       #transforms.RandomVerticalFlip
(),
                                       #transforms.RandomRotation(2
0),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485,
0.456, 0.406],
                                                            [0.229,
0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(transform_res
ize),
                                      transforms.CenterCrop(transform
_crop),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.
456, 0.406],
                                                           [0.229, 0.
224, 0.225])])


# pass transforms in here, then run the next cell to see how the tran
sforms look
train_data = datasets.ImageFolder( data_directory + '/train', transfo
rm=train_transforms )
test_data = datasets.ImageFolder( data_directory + '/test', transform
=test_transforms )
valid_data = datasets.ImageFolder( data_directory + '/valid', transfo
rm=test_transforms )

# ---- print out some data stats ----
print('  Number of train images: ', len(train_data))
print('  Number of test images:  ', len(test_data))
print('  Number of valid images: ', len(valid_data))
```

```
# ------------------------------------

trainloader = torch.utils.data.DataLoader( train_data, batch_size=32,
shuffle=True )
testloader = torch.utils.data.DataLoader( test_data, batch_size=16 )
validloader = torch.utils.data.DataLoader( valid_data, batch_size=16
)

# create dictionary for all loaders in one
loaders_scratch = {}
loaders_scratch['train'] = trainloader
loaders_scratch['valid'] = validloader
loaders_scratch['test'] = testloader
n_classes = len(train_data.classes)
```

```
load image data ...
  Number of train images:  6680
  Number of test images:   836
  Number of valid images:  835
```

In [ ]:

```
```

In [95]:
```
model_scratch = train(50, loaders_scratch, model_scratch, optimizer_s
cratch,
                      criterion_scratch, use_cuda, 'none_augment/mode
l_augmentation2_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('none_augment/model_augmenta
tion2_scratch.pt'))
```

```
Epoch: 1 	Training Loss: 33289.013680 	Validation Loss: 406
9.169052
  Saving model ...
Epoch: 2 	Training Loss: 32003.008823 	Validation Loss: 396
8.229279
  Saving model ...
Epoch: 3 	Training Loss: 31424.128082 	Validation Loss: 391
8.502841
  Saving model ...
Epoch: 4 	Training Loss: 30893.075760 	Validation Loss: 387
6.353407
  Saving model ...
Epoch: 5 	Training Loss: 30494.032368 	Validation Loss: 381
6.702356
  Saving model ...
Epoch: 6 	Training Loss: 30068.997829 	Validation Loss: 376
9.562776
  Saving model ...
Epoch: 7 	Training Loss: 29692.055294 	Validation Loss: 373
4.023375
  Saving model ...
Epoch: 8 	Training Loss: 29333.469135 	Validation Loss: 371
2.007081
  Saving model ...
Epoch: 9 	Training Loss: 28963.161345 	Validation Loss: 367
0.662122
  Saving model ...
Epoch: 10 	Training Loss: 28482.317898 	Validation Loss: 367
5.888136

Epoch: 11 	Training Loss: 28217.756271 	Validation Loss: 360
2.349226
  Saving model ...
Epoch: 12 	Training Loss: 27997.603096 	Validation Loss: 356
9.642857
  Saving model ...
Epoch: 13 	Training Loss: 27623.184799 	Validation Loss: 363
2.852955

Epoch: 14 	Training Loss: 27279.235981 	Validation Loss: 344
6.853124
  Saving model ...
Epoch: 15 	Training Loss: 26986.070410 	Validation Loss: 344
2.554840
  Saving model ...
Epoch: 16 	Training Loss: 26823.070984 	Validation Loss: 336
5.308640
  Saving model ...
Epoch: 17 	Training Loss: 26293.918552 	Validation Loss: 336
9.692473

Epoch: 18 	Training Loss: 26070.885666 	Validation Loss: 338
9.864799

Epoch: 19 	Training Loss: 25697.397066 	Validation Loss: 332
7.614166
  Saving model ...
```

```
Epoch: 20 	Training Loss: 25595.160543 	Validation Loss: 334
2.726606

Epoch: 21 	Training Loss: 25270.672350 	Validation Loss: 325
8.532246
  Saving model ...
Epoch: 22 	Training Loss: 25016.625162 	Validation Loss: 324
8.496114
  Saving model ...
Epoch: 23 	Training Loss: 24679.960295 	Validation Loss: 318
6.919896
  Saving model ...
Epoch: 24 	Training Loss: 24533.861128 	Validation Loss: 314
5.723070
  Saving model ...
Epoch: 25 	Training Loss: 24426.135239 	Validation Loss: 316
2.367277

Epoch: 26 	Training Loss: 24135.673895 	Validation Loss: 321
9.893327

Epoch: 27 	Training Loss: 23964.198538 	Validation Loss: 309
7.609233
  Saving model ...
Epoch: 28 	Training Loss: 23745.290079 	Validation Loss: 302
4.265840
  Saving model ...
Epoch: 29 	Training Loss: 23404.061167 	Validation Loss: 309
8.837419

Epoch: 30 	Training Loss: 23170.097200 	Validation Loss: 312
9.051152

Epoch: 31 	Training Loss: 23070.647598 	Validation Loss: 326
9.955036

Epoch: 32 	Training Loss: 22898.501955 	Validation Loss: 317
8.775061

Epoch: 33 	Training Loss: 22752.647522 	Validation Loss: 300
7.862948
  Saving model ...
Epoch: 34 	Training Loss: 22473.109957 	Validation Loss: 292
2.860757
  Saving model ...
Epoch: 35 	Training Loss: 22300.828423 	Validation Loss: 313
1.648133

Epoch: 36 	Training Loss: 22094.458605 	Validation Loss: 290
2.139578
  Saving model ...
Epoch: 37 	Training Loss: 21998.592911 	Validation Loss: 291
2.403932

Epoch: 38 	Training Loss: 21841.727213 	Validation Loss: 290
2.105658
  Saving model ...
```

```
Epoch: 39       Training Loss: 21736.855339     Validation Loss: 285
4.788197
  Saving model ...
Epoch: 40       Training Loss: 21429.024645     Validation Loss: 277
4.516261
  Saving model ...
Epoch: 41       Training Loss: 21196.958555     Validation Loss: 297
7.358753

Epoch: 42       Training Loss: 21341.049385     Validation Loss: 285
9.885207

Epoch: 43       Training Loss: 20857.280237     Validation Loss: 291
0.965622

Epoch: 44       Training Loss: 20829.968506     Validation Loss: 278
5.109583

Epoch: 45       Training Loss: 20580.846634     Validation Loss: 270
9.397722
  Saving model ...
Epoch: 46       Training Loss: 20467.874403     Validation Loss: 289
6.407328

Epoch: 47       Training Loss: 20421.746988     Validation Loss: 282
4.337608

Epoch: 48       Training Loss: 20304.645758     Validation Loss: 273
9.622649

Epoch: 49       Training Loss: 20211.557783     Validation Loss: 273
1.664424

Epoch: 50       Training Loss: 19991.709316     Validation Loss: 273
4.223965
```

```
Out[95]: <All keys matched successfully>
```

Note3:

That is great! both are going down. I also observed that both start at lower error values now comparing previouse trainings . It was a bit trickiear than what I thought at firts and time consuming too! next I am going to tweak the learning rate and find a semi optimal one.

Tune CNN and hyper parameters:

I am going to train/test the model with a range of learning rates that seems resonable to me... I am going to do ot manually but I am sure there should be better ways like gridsearch.. I do it for just 10 epoches to get a sence , probably it is a good idea to do it for more epoches....

In [97]:
```
param_learning_rate = 0.001 # I tewaked it manualy
criterion_scratch = nn.CrossEntropyLoss() # Based on pytorch recomand
ation for multi class clasification with high precision and recall

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=param_le
arning_rate)
```

In [98]:
```
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_s
cratch,
                      criterion_scratch, use_cuda, 'learning_rate/mod
el_scratch_001.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('learning_rate/model_scratch
_001.pt'))
```

```
Epoch: 1        Training Loss: 19419.441353     Validation Loss: 257
1.991577
  Saving model ...
Epoch: 2        Training Loss: 19154.762239     Validation Loss: 256
2.179347
  Saving model ...
Epoch: 3        Training Loss: 19072.971766     Validation Loss: 255
3.935386
  Saving model ...
Epoch: 4        Training Loss: 19076.901560     Validation Loss: 254
6.110691
  Saving model ...
Epoch: 5        Training Loss: 19132.224756     Validation Loss: 255
5.562618

Epoch: 6        Training Loss: 18983.615808     Validation Loss: 258
1.055560

Epoch: 7        Training Loss: 18937.071451     Validation Loss: 254
2.892222
  Saving model ...
Epoch: 8        Training Loss: 18976.418844     Validation Loss: 253
5.450424
  Saving model ...
Epoch: 9        Training Loss: 19064.935646     Validation Loss: 256
3.680194

Epoch: 10       Training Loss: 18820.238550     Validation Loss: 255
7.209637
```

Out[98]:
```
<All keys matched successfully>
```

It looks like now it learn faster, I am going to descrease learning rate even more.

In [ ]:
```
param_learning_rate = 0.0005 # I tewaked it manualy
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_s
cratch,
                      criterion_scratch, use_cuda, 'learning_rate/mod
el_scratch_0005.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('learning_rate/model_scratch
_0005.pt'))
```

```
Epoch: 1        Training Loss: 18849.856522     Validation Loss: 255
8.956687
  Saving model ...
Epoch: 2        Training Loss: 18743.703451     Validation Loss: 254
6.099226
  Saving model ...
Epoch: 3        Training Loss: 18931.380140     Validation Loss: 255
0.586272

Epoch: 4        Training Loss: 18761.624598     Validation Loss: 255
5.475226

Epoch: 5        Training Loss: 18753.853256     Validation Loss: 253
1.223555
  Saving model ...
Epoch: 6        Training Loss: 18978.795713     Validation Loss: 253
1.353685

Epoch: 7        Training Loss: 18639.263281     Validation Loss: 254
4.865511

Epoch: 8        Training Loss: 18841.711714     Validation Loss: 256
2.131870

Epoch: 9        Training Loss: 18755.938637     Validation Loss: 252
3.797336
  Saving model ...
Epoch: 10       Training Loss: 18749.533445     Validation Loss: 253
1.421881
```

Out[ ]:
```
<All keys matched successfully>
```

Note 4 :

Slightly better but not huge improvment. I am going to stope looking for more optimim learning rate and use 0.0005 as lr.

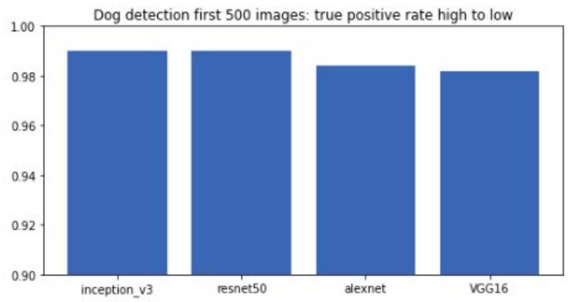
Adding more convolutional layer looks like a bit overkill, however replacing max global pooling layer with average global pooling could be consider.

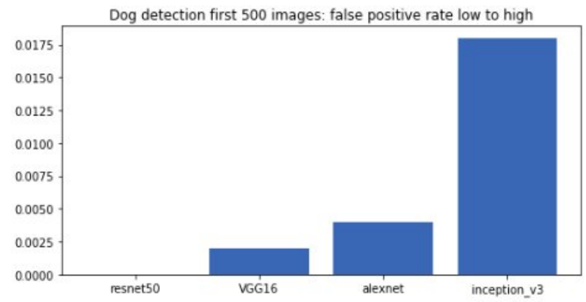
Transfer learning:

At the beginig of notebook. I have evaluated 4 models model_names = ["alexnet","VGG16","resnet50","inception_v3"] at turn out that resnet is doing a great job. I am going to use it as transfer learning model.All I need to do is to replace final linear layer.(model_transfer.fc = nn.Linear(model_transfer.fc.in_features,n_classes))

In [18]:
```
idx_tpr = np.argsort(tpr)[::-1]
idx_fpr = np.argsort(fpr)

fig,ax = plt.subplots(1,2,figsize = (18,4))
ax[0].bar([model_names[i] for i in idx_tpr],tpr[idx_tpr])
ax[0].set_title(f"Dog detection first 500 images: true positive rate high to low")
ax[0].set_ylim((0.9,1))
ax[1].bar([model_names[i] for i in idx_fpr],fpr[idx_fpr])
ax[1].set_title("Dog detection first 500 images: false positive rate low to high")
plt.show()
```





Answer: Resnet50 outperforms other considering low fp and high tp.

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%.

Conclusion:

- As the dataset is small, it apears that data augmentation is needed to provide more training instance and avoid over firring. Fining a good set/order of transformers was time consumng and tricky.
- Althoufg resnet50 did the best job compare to others at my early test, I figured alexnet do actually detect the dogs better.
- I am sure there are better way to tune the hyper parameter as well as design the CNN classifier. I am still learning and hoping to get better undestanding on deeper networks.

In [ ]: