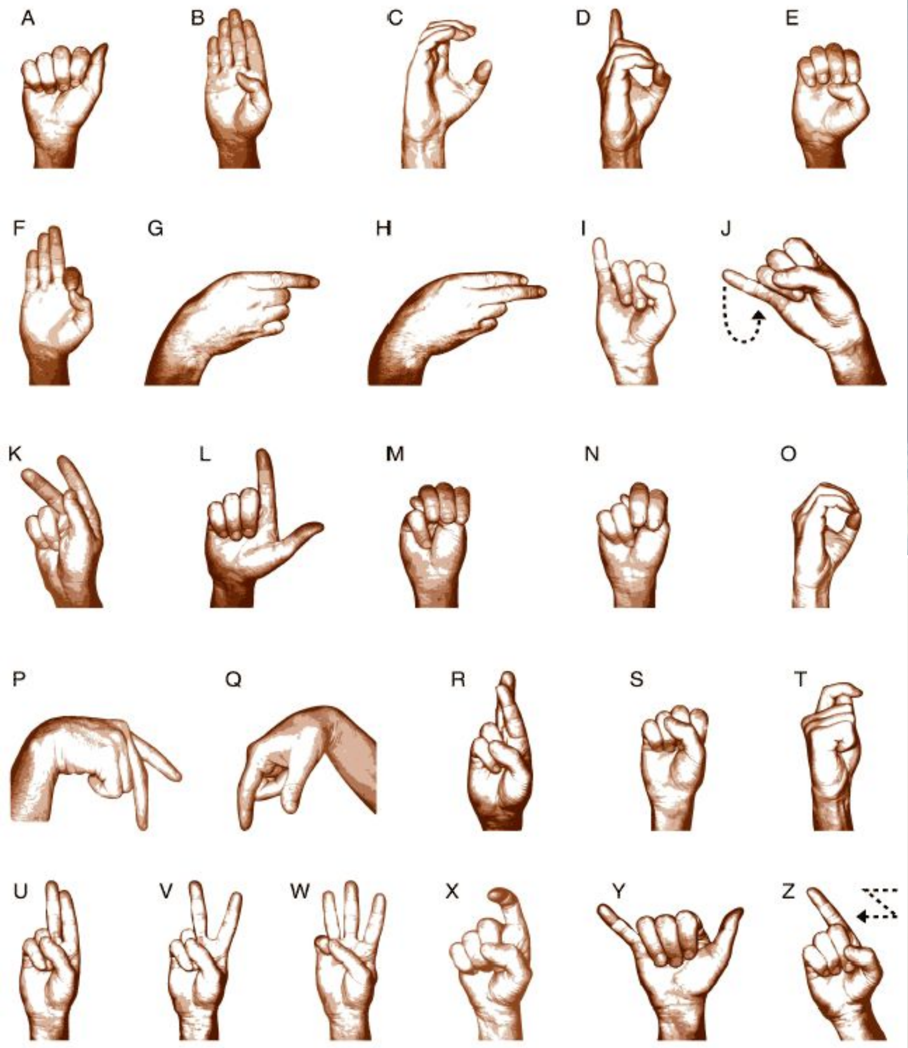
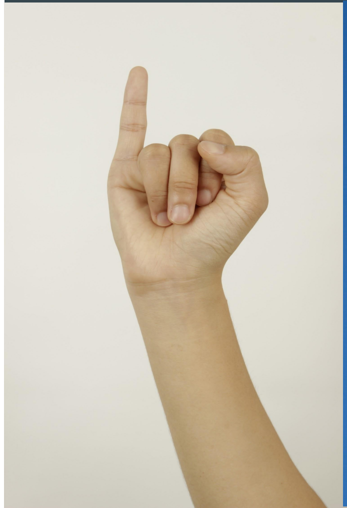
CNN to Classify American Sign Language (ASL)

•••

Group 9
Toufik Bouras, Noah Olsen, Teryn Zmuda


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z



Overview

Problem: Classify American sign language images to letters of the alphabet

Method: Apply CNN with Pytorch, and vary approach with three network types, including VGG-16, Resnet.

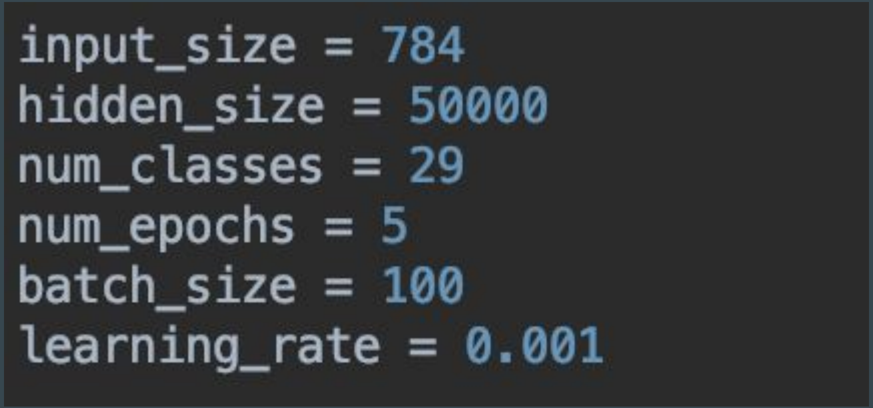
Data: Image data set for the American alphabet sign language, 1 GB

- Training: 87,000 images, 64 x 64 pixels
  - Classify into 29 classes, 26 letters A-Z and 3 for space, delete, and nothing
  - 3,000 files for each class
- Validation: 29 images, 64x64 pixels

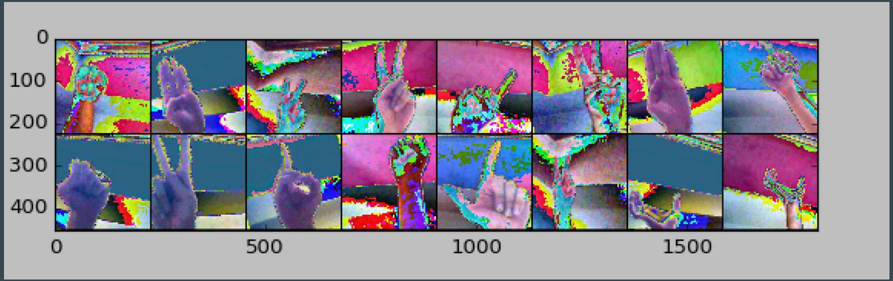
Convolutional Neural Network (CNN):
Multi-layer perceptron

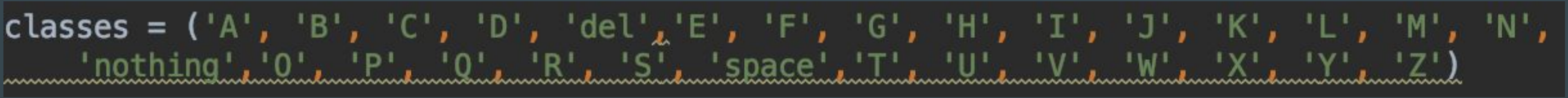
Method Specifics & Theory

- Mini-batch gradient descent
- 29 classes
- Classification organization



```
input_size = 784
hidden_size = 50000
num_classes = 29
num_epochs = 5
batch_size = 100
learning_rate = 0.001
```

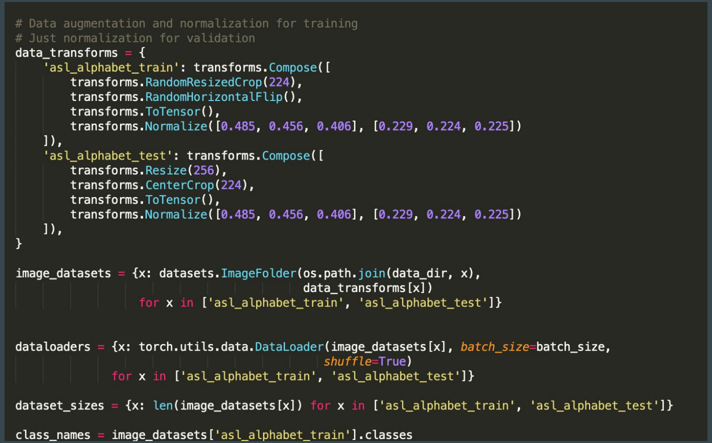




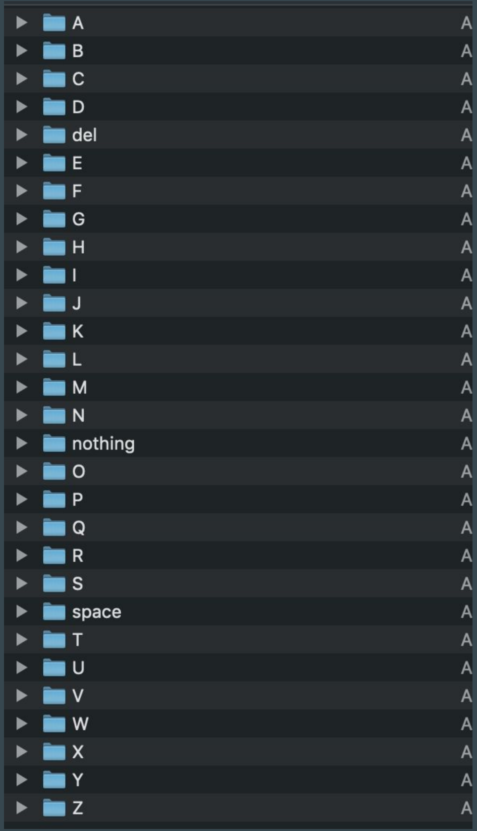
```
classes = ('A', 'B', 'C', 'D', 'del', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
     'nothing', 'O', 'P', 'Q', 'R', 'S', 'space', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z')
```

Data Challenges

- Extremely large dataset.
- The names of the classes are based on the folder names of each image.
- Use the Pytorch function ImageFolder to handle this kind of data.



```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'asl_alphabet_train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'asl_alphabet_test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['asl_alphabet_train', 'asl_alphabet_test']}


dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                              shuffle=True)
              for x in ['asl_alphabet_train', 'asl_alphabet_test']}

dataset_sizes = {x: len(image_datasets[x]) for x in ['asl_alphabet_train', 'asl_alphabet_test']}

class_names = image_datasets['asl_alphabet_train'].classes
```

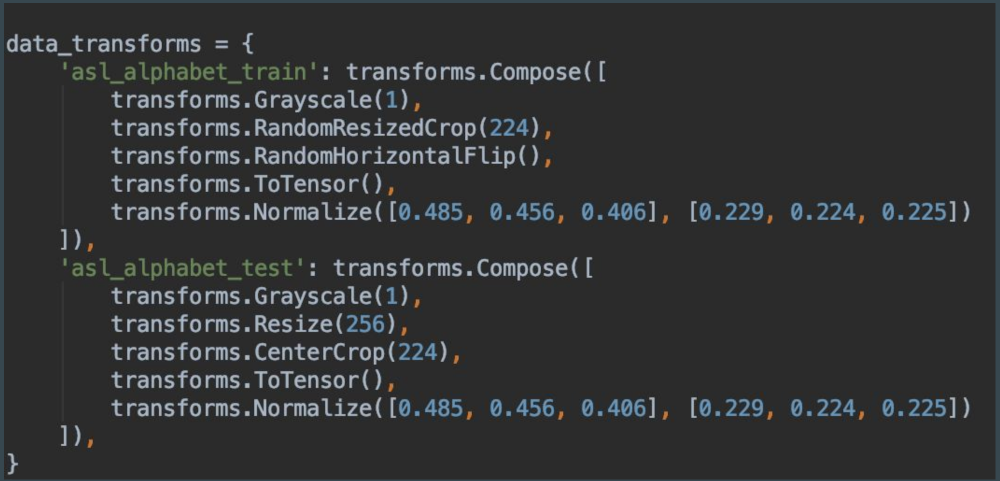


```
▶ A        A
▶ B        A
▶ C        A
▶ D        A
▶ del      A
▶ E        A
▶ F        A
▶ G        A
▶ H        A
▶ I        A
▶ J        A
▶ K        A
▶ L        A
▶ M        A
▶ N        A
▶ nothing  A
▶ O        A
▶ P        A
▶ Q        A
▶ R        A
▶ S        A
▶ space    A
▶ T        A
▶ U        A
▶ V        A
▶ W        A
▶ X        A
▶ Y        A
▶ Z        A
```

Training Images for “A”

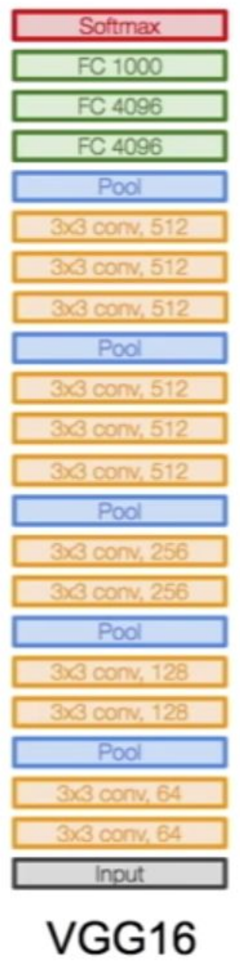
Data augmentation and normalization for training

Normalization for validation



```
data_transforms = {
    'asl_alphabet_train': transforms.Compose([
        transforms.Grayscale(1),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'asl_alphabet_test': transforms.Compose([
        transforms.Grayscale(1),
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

Understanding the different networks

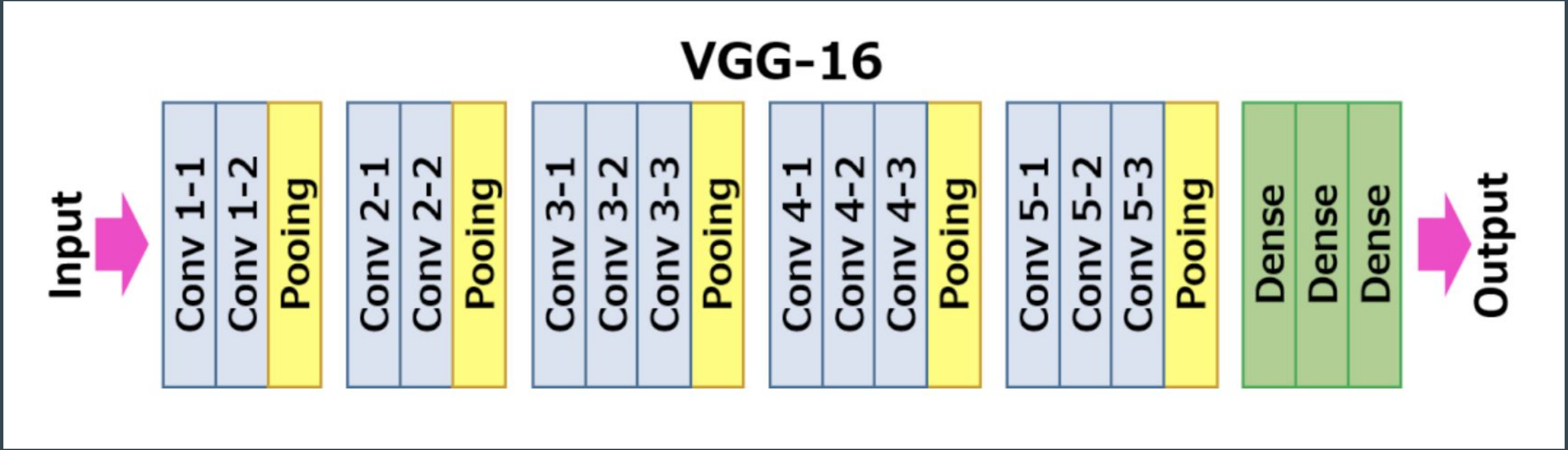


VGG-16

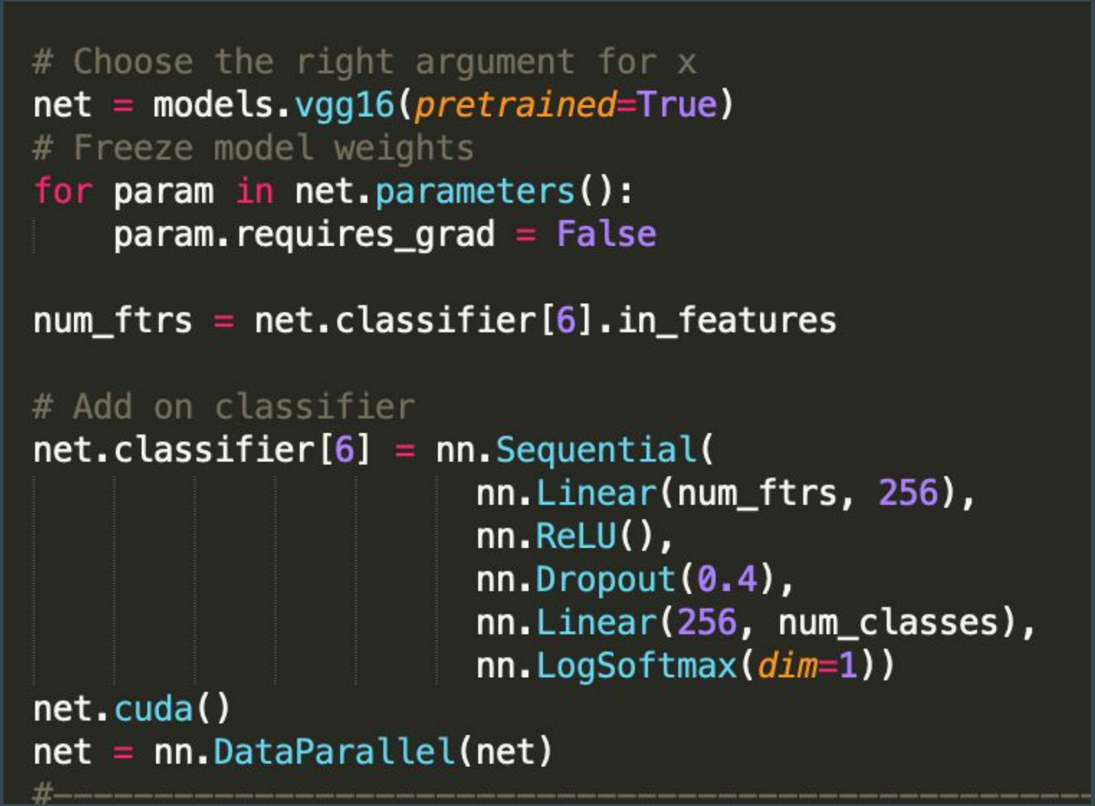
VGG-16:

- 2014 genesis, build from AlexNet, Oxford
- Competitor: GoogleNet
- Deeper but smaller filters, only 3 x 3 CONV all the way: 8 layers (AlexNet) to 16 layers
- Total memory = 100 MB/Image (forward only); 138M parameters

VGG-16 Network Diagram

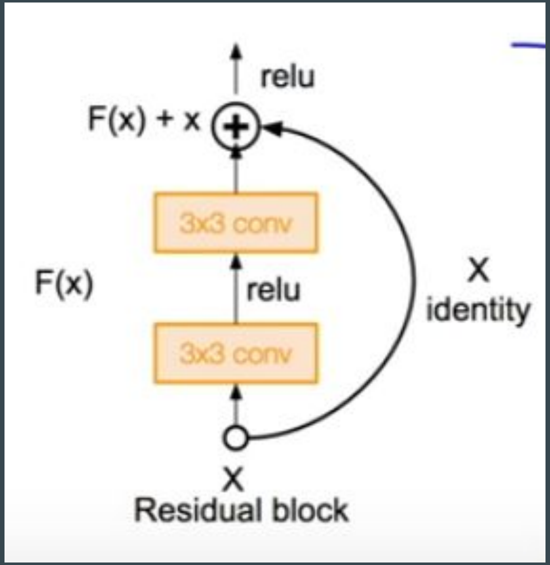


VGG-16 Network Code



```python
# Choose the right argument for x
net = models.vgg16(pretrained=True)
# Freeze model weights
for param in net.parameters():
    param.requires_grad = False

num_ftrs = net.classifier[6].in_features

# Add on classifier
net.classifier[6] = nn.Sequential(
                      nn.Linear(num_ftrs, 256),
                      nn.ReLU(),
                      nn.Dropout(0.4),
                      nn.Linear(256, num_classes),
                      nn.LogSoftmax(dim=1))
net.cuda()
net = nn.DataParallel(net)
#------------------------------------------------
```

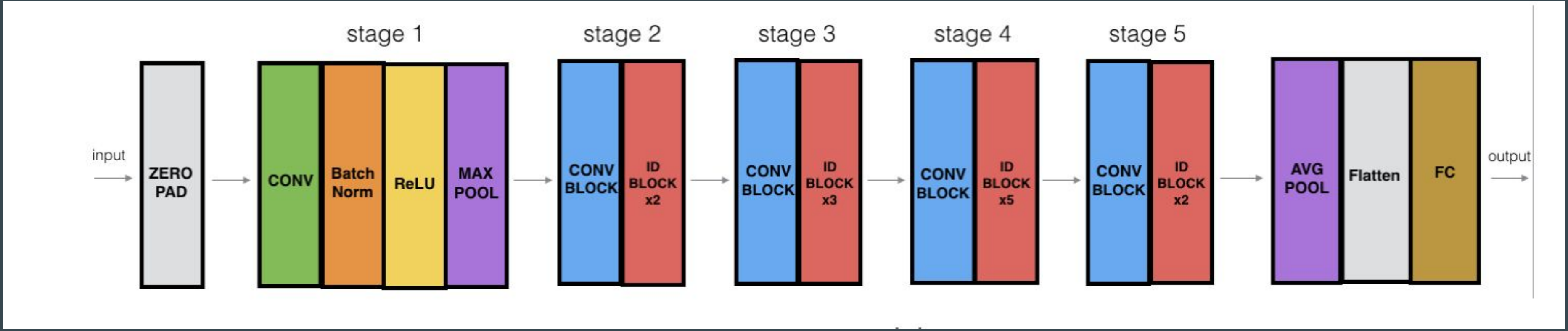
Resnet



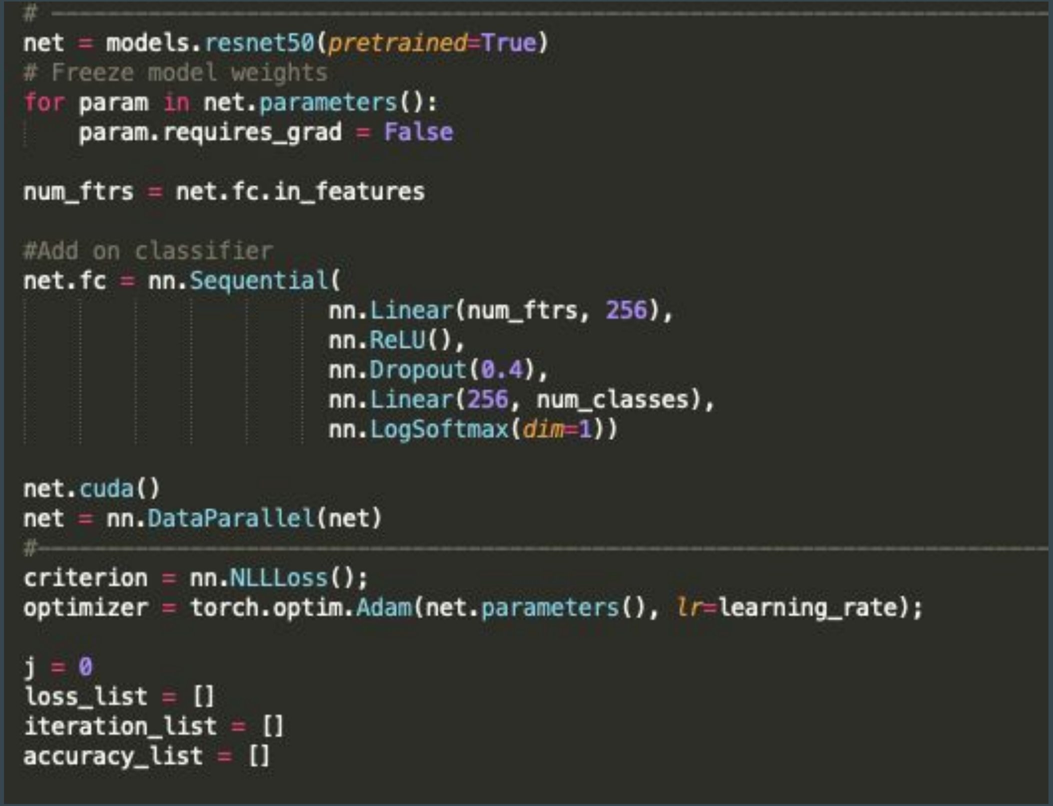
Resnet:

- Follow up on Alexnet
- Stack resenet blocks
- Batch normalization after each layer
- Deeper than previous networks
- Residual connections
- Can train thousands of layers and still achieve good performance.

Resnet Network Diagram



Resnet Network Code



```
# ----------------------------------------------------------------------------
net = models.resnet50(pretrained=True)
# Freeze model weights
for param in net.parameters():
    param.requires_grad = False

num_ftrs = net.fc.in_features

#Add on classifier
net.fc = nn.Sequential(
                      nn.Linear(num_ftrs, 256),
                      nn.ReLU(),
                      nn.Dropout(0.4),
                      nn.Linear(256, num_classes),
                      nn.LogSoftmax(dim=1))

net.cuda()
net = nn.DataParallel(net)
#-----------------------------------------------------------------------------
criterion = nn.NLLLoss();
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate);

j = 0
loss_list = []
iteration_list = []
accuracy_list = []
```

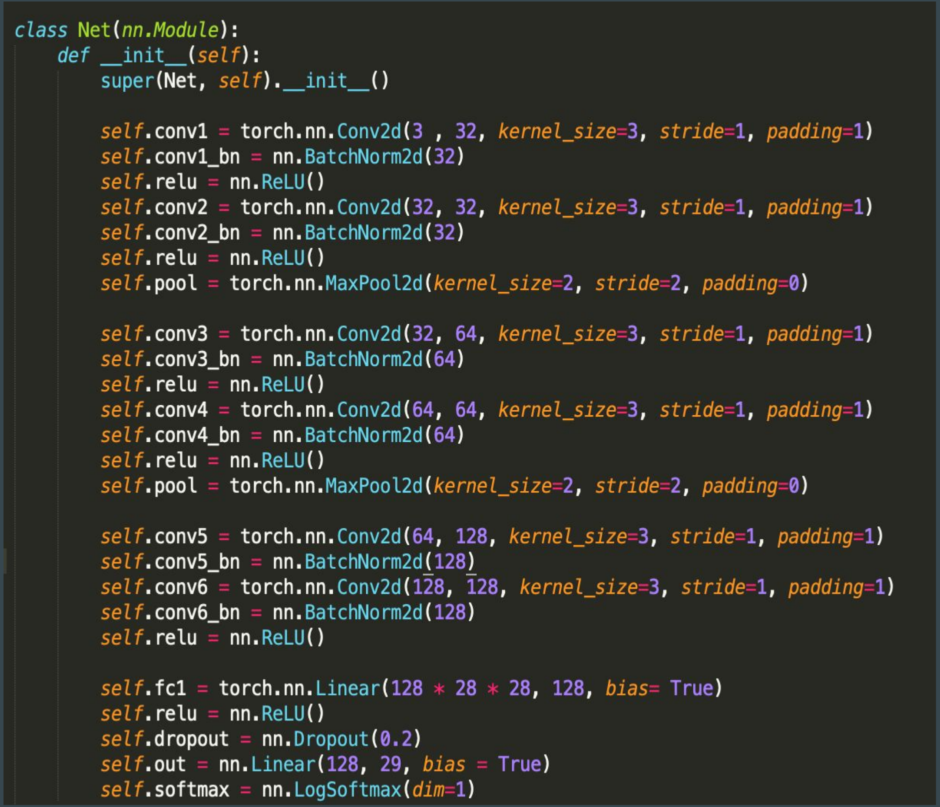
Our Model

Our attempt at a more custom fitted model. The initial inspiration was Resnet.

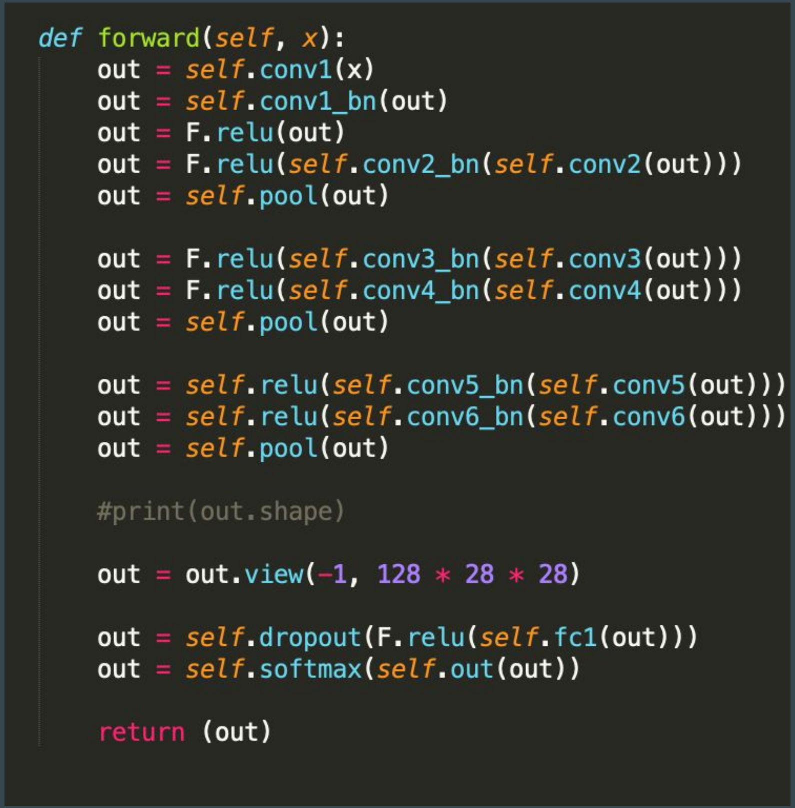
Features:

- Deeper than previous networks
- Residual connections

Code



```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = torch.nn.Conv2d(3 , 32, kernel_size=3, stride=1, padding=1)
        self.conv1_bn = nn.BatchNorm2d(32)
        self.relu = nn.ReLU()
        self.conv2 = torch.nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1)
        self.conv2_bn = nn.BatchNorm2d(32)
        self.relu = nn.ReLU()
        self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        self.conv3 = torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3_bn = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.conv4 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.conv4_bn = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        self.conv5 = torch.nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv5_bn = nn.BatchNorm2d(128)
        self.conv6 = torch.nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1)
        self.conv6_bn = nn.BatchNorm2d(128)
        self.relu = nn.ReLU()

        self.fc1 = torch.nn.Linear(128 * 28 * 28, 128, bias= True)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)
        self.out = nn.Linear(128, 29, bias = True)
        self.softmax = nn.LogSoftmax(dim=1)
```

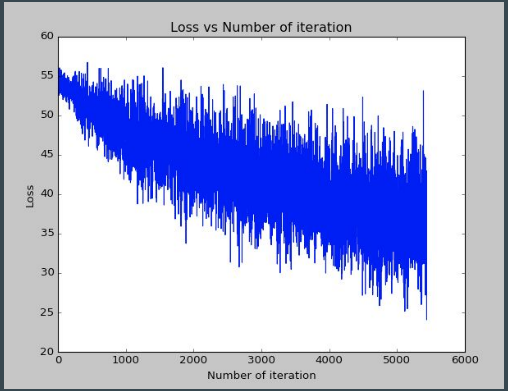


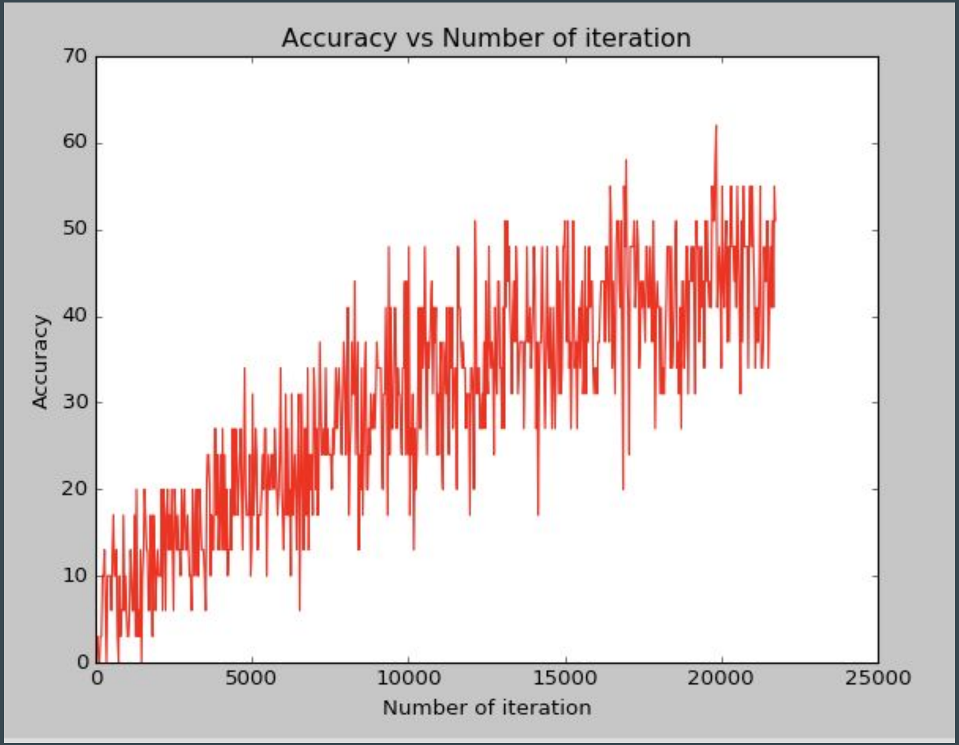
```python
def forward(self, x):
    out = self.conv1(x)
    out = self.conv1_bn(out)
    out = F.relu(out)
    out = F.relu(self.conv2_bn(self.conv2(out)))
    out = self.pool(out)

    out = F.relu(self.conv3_bn(self.conv3(out)))
    out = F.relu(self.conv4_bn(self.conv4(out)))
    out = self.pool(out)

    out = self.relu(self.conv5_bn(self.conv5(out)))
    out = self.relu(self.conv6_bn(self.conv6(out)))
    out = self.pool(out)

    #print(out.shape)

    out = out.view(-1, 128 * 28 * 28)

    out = self.dropout(F.relu(self.fc1(out)))
    out = self.softmax(self.out(out))

    return (out)
```

Accuracy and Loss Over Time for our Model

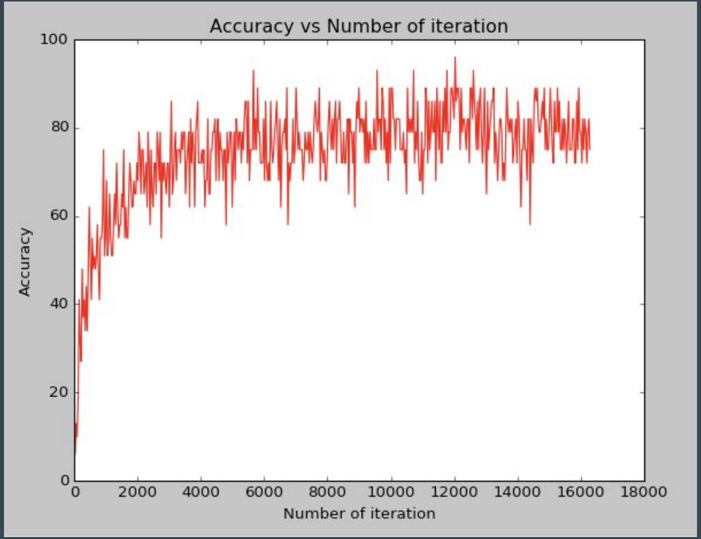
Results:

- Started off with very low accuracy
- Learned significantly over the lifetime of the model.
- Loss improved over time.
- Very slow performance



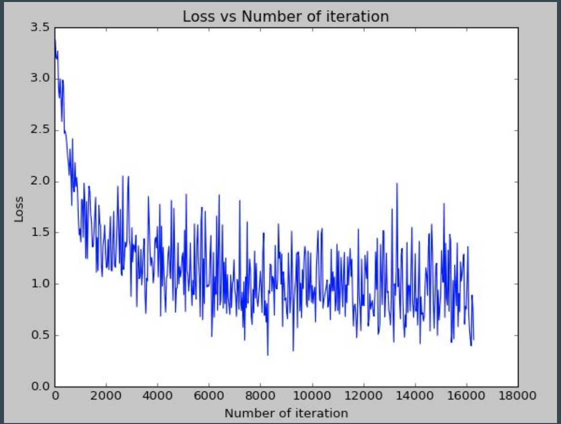


Accuracy and Loss Over Time for adapted VGG16



Results:

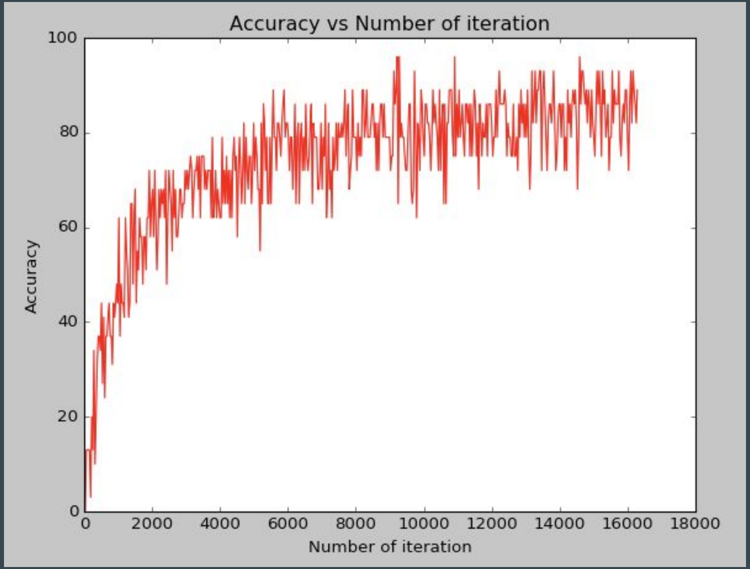
- Very slow.
- Performance is quite strong.
- Finished with around 80% accuracy.

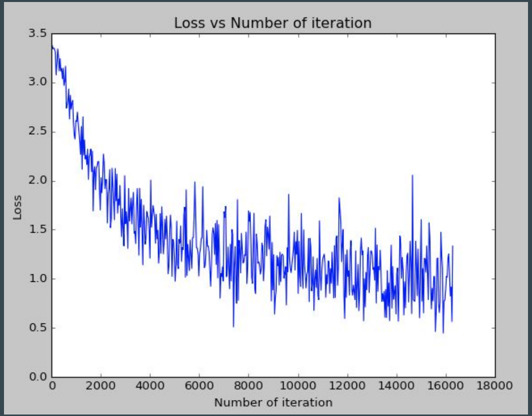


Accuracy and Loss Over Time for adapted Resnet50

Results:

- Started off with very low accuracy
- Scaled rapidly
- Impressively quick performance
- Finished with around 90% accuracy





Questions?

Sources

- https://neurohive.io/en/popular-networks/vgg16/
- https://arxiv.org/abs/1512.03385
- **Akash. (2018, April 22). ASL Alphabet. Retrieved from https://www.kaggle.com/grassknoted/asl-alphabet**
- **Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. CoRR, abs/1409.1556 . Retrieved from http://arxiv.org/abs/ 1409.1556**
- **Stanford School of Engineering, CNN Architecture Lecture. Aug. 11, 2017. Retrieved from https://www.youtube.com/watch?v=DAOcjicFr1**