Using Pre-Training Can Improve Model Robustness and Uncertainty

Reproduced by: Xingchen Liu Rongyu Wang
12/06/2019

Pre-training is widely used for deep convolutional neural networks
  ○ Application: “Pre-train then Tune” paradigm
  ○ Research: create “universal representations”

**Doubts** on Pre-training
  ○ He et al. [1] argued pre-training shows no performance benefits for traditional models or large tasks.
  ○ Lin et al. [2] found that pre-training doesn’t have advantages when giving sufficient time for training, even for the tuning for extremely small datasets.
Motivation

- To demonstrate the effectiveness of pre-training on several aspects.

- To prove that pre-training enhances model uncertainty estimates in
  - Out-of-Distribution detection
  - calibration
Related Work

- **Pre-training**
  - Improves generalization for tasks with small datasets, including transfer learning[3][4], and tasks with significant variation[5].
  - Used in large tasks such as Microsoft COCO[1]
    - but no accuracy gains in performance[2][15]
Related Work

- **Uncertainty**
  - To detect out-of-distribution samples, use the maximum value of a classifier’s softmax distribution [18]
  - Mahalanobis distance-based scores that characterize out-of-distribution samples using hidden features [9]
  - Using GAN [21] to generate out-of-distribution samples
  - Applying non-specific, real, and diverse outlier images or text instead improves out-of-distribution detection performance and calibration [22]
  - Contemporary networks can easily become miscalibrated without additional regularization [23]
Target Task

Show the performance of pre-training in:

- **Uncertainty**
  - Out-of-Distribution Detection
  - Calibration
Data Summary

- Downsampled (64 x 64) ImageNet[10] for pre-training
- CIFAR-10, CIFAR-100 and Tiny ImageNet datasets[24] without 200 overlapping Tiny ImageNet classes from Downsampled ImageNet
Implementation

- Use 40-2 Wide ResNets trained using SGD with Nesterov momentum and a cosine learning rate.

\[
\eta_t = \eta_{min} + \frac{1}{2} (\eta_{max} - \eta_{min}) (1 + \cos(\frac{T_{cur}}{T_i} \pi)),
\]

- Pre-training: 100 epochs on Downsampled ImageNet, fine-tuned for 10 epochs for CIFAR and 20 for Tiny ImageNet without dropout (learning rate of 0.001).
- Baseline: 100 epochs with a dropout rate of 0.3.
class WideResNet(nn.Module):
    def __init__(self, depth, num_classes, widen_factor=1, dropRate=0.0):
        super(WideResNet, self).__init__()
        nChannels = [16, 16 * widen_factor, 32 * widen_factor, 64 * widen_factor]
        assert ((depth - 4) % 6 == 0)
        n = (depth - 4) // 6
        block = BasicBlock
        # 1st conv before any network block
        self.conv1 = nn.Conv2d(3, nChannels[0], kernel_size=3, stride=1,
                               padding=1, bias=False)
        # 1st block
        self.block1 = NetworkBlock(n, nChannels[0], nChannels[1], block, 1, dropRate)
        # 2nd block
        self.block2 = NetworkBlock(n, nChannels[1], nChannels[2], block, 2, dropRate)
        # 3rd block
        self.block3 = NetworkBlock(n, nChannels[2], nChannels[3], block, 2, dropRate)
        # global average pooling and classifier
        self.bn1 = nn.BatchNorm2d(nChannels[3])
        self.relu = nn.ReLU(inplace=True)
        self.fc = nn.Linear(nChannels[3], num_classes)
        self.nChannels = nChannels[3]
Train from scratch

```python
# Train network on CIFAR10
train_data, test_data = load_cifar10_data()
model = WideResNet(40, 10, 2, 0.3)
model.cuda()
total_steps = 100 * len(train_data)
optimizer = torch.optim.SGD(model.parameters(), 0.1, momentum=0.9, nesterov=True)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda= lambda step: cosine_lr(step, total_steps, 1, 10e-5))
save_name = "CIFAR10_baseline.pt"
train_process(model, train_data, test_data, optimizer, scheduler, 100, save_name)
```

Pre-train then tune

```python
# Pre-train on ImageNet Downsample
torch.cuda.manual_seed(1)
train_data, test_data = load_imagenetds_data()
model = WideResNet(40, 10, 2, 0.3)
model.cuda()
total_steps = 100 * len(train_data)
optimizer = torch.optim.SGD(model.parameters(), 0.1, momentum=0.9, nesterov=True)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda= lambda step: cosine_lr(step, total_steps, 1, 10e-5))
save_name = "Tiny_ImageNet_baseline.pt"
train_process(model, train_data, test_data, optimizer, scheduler, 100, save_name)
```

# Tune on CIFAR10

```python
train_data, test_data = load_cifar10_data()
model = WideResNet(40, 1000, 2, 0)
model.cuda()
model.load_state_dict(torch.load("saved_model/pretrained_model.pt"))
total_steps = 100 * len(train_data)
optimizer = torch.optim.SGD(model.parameters(), 0.001, momentum=0.9, nesterov=True)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda= lambda step: cosine_lr(step, total_steps, 1, 10e-5))
save_name = "CIFAR10_pretrained.pt"
train_process(model, train_data, test_data, optimizer, scheduler, 10, save_name)
```
Out-of-Distribution Detection

- In distribution data: test dataset
- Out-of-distribution data: Gaussian noise, textures etc.
- Use the maximum softmax probabilities to score anomalies
- Evaluation metrics: AUROC (the Area Under the Receiver Operating Characteristic curve), AUPR (the Area Under the Precision-Recall Curve)
Calculate AUROC and AUPR

def get_measures(out_score, in_score):
    out_score = np.array(out_score)
in_score = np.array(in_score)
examples = np.concatenate((out_score, in_score), axis=None)
labels = np.zeros(len(examples), dtype=np.int32)
labels[:len(out_score)] += 1

auROC = sk.roc_auc_score(labels, examples)
aupr = sk.average_precision_score(labels, examples)
return auROC, aupr

def get_results(ood_loader, auROC_results, aupr_results):
aurocs, auprs = [], []
for i in range(5):
    out_score = get_scores(ood_loader)
auroc, aupr = get_measures(out_score, in_score)
aurocs.append(auroc)
auprs.append(aupr)

auROC = np.mean(aurocs)
aupr = np.mean(auprs)
auROC_results.append(auroc*100)
aupr_results.append(aupr*100)
print("AUROC: ", 100*auROC)
print("AUPR: ", 100*aupr)
print(" 

")
## Out-of-distribution Detection

<table>
<thead>
<tr>
<th>Dataset</th>
<th>AUROC Normal</th>
<th>AUROC Pre-Train</th>
<th>AUPR Normal</th>
<th>AUPR Pre-Train</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>91.5</td>
<td>94.5</td>
<td>63.4</td>
<td>73.5</td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>69.4</td>
<td>83.1</td>
<td>29.7</td>
<td>52.7</td>
</tr>
<tr>
<td>Tiny ImageNet</td>
<td>71.8</td>
<td>73.9</td>
<td>30.8</td>
<td>31.0</td>
</tr>
</tbody>
</table>
Out-of-distribution Detection for CIFAR10

<table>
<thead>
<tr>
<th>CIFAR10</th>
<th>AUROC</th>
<th>AUPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Pre-training</td>
</tr>
<tr>
<td>Gaussian</td>
<td>89.32</td>
<td>95.44</td>
</tr>
<tr>
<td>Rademacher</td>
<td>89.04</td>
<td>94.61</td>
</tr>
<tr>
<td>Blob</td>
<td>94.56</td>
<td>97.15</td>
</tr>
<tr>
<td>Textures</td>
<td>87.98</td>
<td>93.74</td>
</tr>
<tr>
<td>SVHN</td>
<td>91.77</td>
<td>95.65</td>
</tr>
<tr>
<td>CIFAR100</td>
<td>87.42</td>
<td>90.49</td>
</tr>
<tr>
<td>Mean</td>
<td>90.02</td>
<td>94.49</td>
</tr>
</tbody>
</table>
## Out-of-distribution Detection for CIFAR100

<table>
<thead>
<tr>
<th>CIFAR100</th>
<th>AUROC</th>
<th>AUPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Pre-training</td>
<td>Normal</td>
</tr>
<tr>
<td>Gaussian</td>
<td>49.67</td>
<td>96.61</td>
</tr>
<tr>
<td>Rademacher</td>
<td>46.75</td>
<td>97.66</td>
</tr>
<tr>
<td>Blob</td>
<td>85.89</td>
<td>89.34</td>
</tr>
<tr>
<td>Textures</td>
<td>73.29</td>
<td>79.65</td>
</tr>
<tr>
<td>SVHN</td>
<td>74.51</td>
<td>79.21</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>75.66</td>
<td>75.21</td>
</tr>
<tr>
<td>Mean</td>
<td>67.63</td>
<td>86.28</td>
</tr>
</tbody>
</table>
### Reproduction Results

#### Out-of-distribution Detection for Tiny ImageNet

<table>
<thead>
<tr>
<th>Tiny ImageNet</th>
<th>AUROC</th>
<th>AUPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Pre-training</td>
</tr>
<tr>
<td>Gaussian</td>
<td>64.82</td>
<td>78.01</td>
</tr>
<tr>
<td>Rademacher</td>
<td>67.48</td>
<td>73.42</td>
</tr>
<tr>
<td>Blob</td>
<td><strong>64.84</strong></td>
<td>60.47</td>
</tr>
<tr>
<td>Textures</td>
<td>68.99</td>
<td>72.25</td>
</tr>
<tr>
<td>SVHN</td>
<td>86.66</td>
<td>89.24</td>
</tr>
<tr>
<td>Mean</td>
<td>70.58</td>
<td>74.68</td>
</tr>
</tbody>
</table>
Reproduction Results

AUROC results

- CIFAR10
- CIFAR100
- Tiny ImageNet

Normal
Pre-trained
Reproduction Results

AUPR Results

- **CIFAR10**
  - Normal: Lower
  - Pre-trained: Higher

- **CIFAR100**
  - Normal: Lower
  - Pre-trained: Higher

- **Tiny ImageNet**
  - Normal: Lower
  - Pre-trained: Higher
Calibration

- According to methods of Chawla et al[22], adopt RMS and MAD to measure the calibration of a classifier

\[
\text{RMS: } \sqrt{\frac{1}{n} \sum_{i=1}^{b} \frac{|B_i|}{|B_i|} \left( \sum_{k \in B_i} 1(y_k = \hat{y}_k) - \frac{1}{|B_i|} \sum_{k \in B_i} c_k \right)^2}.
\]

\[
\text{MAD: } \sum_{i=1}^{b} \frac{|B_i|}{n} \left| \frac{1}{|B_i|} \sum_{k \in B_i} 1(y_k = \hat{y}_k) - \frac{1}{|B_i|} \sum_{k \in B_i} c_k \right|
\]

\[
\text{Soft F1 score: } \frac{c_a^T m}{1^T (c_a + m)/2}
\]

- Compare results of using pre-training and not using pre-training.
## In-distribution Calibration

<table>
<thead>
<tr>
<th></th>
<th>RMS Error</th>
<th></th>
<th>MAD Error</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Pre-Train</td>
<td>Normal</td>
<td>Pre-Train</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>6.4</td>
<td>2.9</td>
<td>2.9</td>
<td>1.2</td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>13.3</td>
<td>3.6</td>
<td>10.3</td>
<td>2.5</td>
</tr>
</tbody>
</table>
import torchvision.transforms as trn
import torchvision.datasets as dset
import torch.utils.data as data_util

# SVHN
svhn_transform = trn.Compose([trn.Resize(dim), trn.ToTensor(), trn.Normalize(mean, std)])
svhn_data = SVHN(root="data/SVHN", download=True, split="test", transform=svhn_transform)
svhn_loader = torch.utils.data.DataLoader(svhn_data, batch_size=200, shuffle=True)
print('SVHN Calibration')
get_results(svhn_loader, auroc_results, aupr_results)

# Texture
texture_transform = trn.Compose([trn.Resize(dim), trn.CenterCrop(dim), trn.ToTensor(), trn.Normalize(mean, std)])
texture_data = dset.ImageFolder(root="data/dtd/images", transform=texture_transform)
texture_loader = torch.utils.data.DataLoader(texture_data, batch_size=200, shuffle=True, num_workers=4)
print('Texture Detection')
get_results(texture_loader, auroc_results, aupr_results)

# CIFAR
if name == "CIFAR100":
    cifar_data = dset.CIFAR100('data/CIFAR100', train=False, transform=test_transform)
cifar_loader = torch.utils.data.DataLoader(cifar_data, batch_size=200, shuffle=True)
print('CIFAR Detection')
get_results(cifar_loader, auroc_results, aupr_results)

if name == "CIFAR100":
    cifar_data = dset.CIFAR10('data/CIFAR10', train=False, transform=test_transform)
cifar_loader = torch.utils.data.DataLoader(cifar_data, batch_size=200, shuffle=True)
print('CIFAR Detection')
get_results(cifar_loader, auroc_results, aupr_results)

# Mean Result
print(auroc_results)
auroc_mean = sum(auroc_results) / len(auroc_results)
aupr_mean = sum(aupr_results) / len(aupr_results)
print('Mean Results')
# Gaussian Noise

```python
gaussian_targets = torch.ones(ood_num * 5)
m = normal.Normal(0.5, 0.5)
gaussian_ood = m.sample((ood_num * 5, 3, dim, dim))

gaussian_ood = torch.utils.data.TensorDataset(gaussian_ood, gaussian_targets)
gaussian_loader = torch.utils.data.DataLoader(gaussian_ood, batch_size=200, shuffle=True)

print('Gaussian Noise Detection')
get_results(gaussian_loader, auroc_results, aupr_results)
```

# Rademacher Noise

```python
rademacher_targets = torch.ones(ood_num * 5)
m = bernoulli.Bernoulli(0.5)
rademacher_ood = m.sample((ood_num * 5, 3, dim, dim))
rademacher_data = torch.utils.data.TensorDataset(rademacher_ood, rademacher_targets)
rademacher_loader = torch.utils.data.DataLoader(rademacher_data, batch_size=200, shuffle=True)

print('Rademacher Noise Detection')
get_results(rademacher_loader, auroc_results, aupr_results)
```
```python
def rms_mad_error(confidence, correct, error_name, bin_size):
    index = np.argsort(confidence)
    confidence = confidence[index]
    correct = correct[index]
    num_bins = len(confidence) // bin_size
    bins = [i * bin_size for i in range(num_bins)]

    error_count = 0
    num_samples = len(confidence)
    for i in range(num_bins):
        if i < num_bins - 1:
            start, end = bins[i], bins[i + 1]
        else:
            start, end = bins[i], len(confidence)
        bin_conf = confidence[start:end]
        bin_corr = correct[start:end]
        bin_size = len(bin_conf)

        if bin_size > 0:
            diff = np.abs(np.nanmean(bin_conf) - np.nanmean(bin_corr))
            if error_name == 'rms':
                error_count += bin_size / num_samples * np.square(diff)
            elif error_name == 'mad':
                error_count += bin_size / num_samples * diff

    if error_name == 'rms':
        error_count = np.sqrt(error_count)

    return error_count
```

Calculate RMS, MAD, Sf1
Calculate RMS, MAD, Sf1

def SoftF(confidence, correct):
    recall = 1 - correct
    precision = 1 - confidence
    num = (precision * recall).sum()
    denom = (precision + recall).sum() / 2
    sf1 = num / denom
    return sf1
Reproduction Results

Calibration for CIFAR10

<table>
<thead>
<tr>
<th>CIFAR10</th>
<th>RMS Error</th>
<th></th>
<th>MAD Error</th>
<th></th>
<th>Soft F1 Score</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Pre-training</td>
<td>Normal</td>
<td>Pre-training</td>
<td>Normal</td>
<td>Pre-training</td>
</tr>
<tr>
<td>In-Distribution</td>
<td>6.39</td>
<td>2.88</td>
<td>2.87</td>
<td>1.24</td>
<td>27.83</td>
<td>29.65</td>
</tr>
<tr>
<td>Gaussian</td>
<td>33.02</td>
<td>28.17</td>
<td>18.09</td>
<td>14.06</td>
<td>15.44</td>
<td>34.63</td>
</tr>
<tr>
<td>Rademacher</td>
<td>33.29</td>
<td>29.63</td>
<td>18.20</td>
<td>14.73</td>
<td>14.61</td>
<td>30.12</td>
</tr>
<tr>
<td>Blob</td>
<td>26.76</td>
<td>22.54</td>
<td>14.94</td>
<td>11.73</td>
<td>36.69</td>
<td>48.46</td>
</tr>
<tr>
<td>Textures</td>
<td>24.83</td>
<td>23.18</td>
<td>16.02</td>
<td>13.46</td>
<td>29.88</td>
<td>38.08</td>
</tr>
<tr>
<td>SVHN</td>
<td>25.26</td>
<td>23.30</td>
<td>15.37</td>
<td>12.74</td>
<td>34.01</td>
<td>42.58</td>
</tr>
<tr>
<td>CIFAR100</td>
<td>24.80</td>
<td>22.14</td>
<td>16.26</td>
<td>13.82</td>
<td>28.31</td>
<td>35.69</td>
</tr>
<tr>
<td>Mean</td>
<td>24.91</td>
<td>21.69</td>
<td>14.54</td>
<td>11.69</td>
<td>26.68</td>
<td>37.03</td>
</tr>
</tbody>
</table>
# Reproduction Results

## Calibration for CIFAR100

<table>
<thead>
<tr>
<th>CIFAR100</th>
<th>RMS Error</th>
<th>MAD Error</th>
<th>Soft F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Pre-training</td>
<td>Normal</td>
</tr>
<tr>
<td>In-Distribution</td>
<td>13.32</td>
<td>3.63</td>
<td>10.26</td>
</tr>
<tr>
<td>Gaussian</td>
<td>28.25</td>
<td>8.48</td>
<td>24.30</td>
</tr>
<tr>
<td>Rademacher</td>
<td>28.11</td>
<td>7.49</td>
<td>24.52</td>
</tr>
<tr>
<td>Blob</td>
<td>22.62</td>
<td>10.13</td>
<td>17.95</td>
</tr>
<tr>
<td>Textures</td>
<td>23.40</td>
<td>10.81</td>
<td>20.11</td>
</tr>
<tr>
<td>SVHN</td>
<td>24.06</td>
<td>10.23</td>
<td>24.06</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>23.64</td>
<td>11.48</td>
<td>19.91</td>
</tr>
<tr>
<td>Mean</td>
<td>23.34</td>
<td>8.89</td>
<td>19.61</td>
</tr>
</tbody>
</table>
Reproduction Results

RMS Error

- CIFAR10
- CIFAR100

<table>
<thead>
<tr>
<th></th>
<th>Normal</th>
<th>Pre-trained</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIFAR100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reproduction Results

MAD Error

- **CIFAR10**
  - Normal: ~15 units
  - Pre-trained: ~12 units

- **CIFAR100**
  - Normal: ~22 units
  - Pre-trained: ~8 units
Reproduction Results

Soft F1 Score

- CIFAR10
  - Normal: 25
  - Pre-trained: 35

- CIFAR100
  - Normal: 45
  - Pre-trained: 55
Conclusion

- The benefits of pre-training extend beyond merely quick convergence, as previously thought, since pre-training can improve model uncertainty.
  - Pre-trained representations directly translate to improvements in predictive uncertainty estimates.
  - Training from scratch can only reach the same performance as training with pre-training on unperturbed data.
Future Work

- Figure out the reasons for unexpected lower performance of pre-trained models
  - OOD detection (Tiny-ImageNet) on Blob
- Reproduce the part of evaluation on robustness of models.
- Validate the effects of pre-training on other datasets.
- Compare the effects of different strategies for pre-training.
- Some work could specialize pre-training for these downstream tasks.
Job Split

Xingchen Liu:
Train baseline network from scratch, OOD detection

Clare Wang:
Pre-train network and tune, Calibration
References

References

References