BatchNorm and Residual Networks

Two Important Modern CNN Architecture Concepts

David I. Inouye

Batch Normalization Dynamically Normalizes Each Feature to Have Zero Mean and Unit Variance

- **Basic idea:** Normalize input batch of each layer **during the forward pass**

  1. Input is minibatch of data $X^t \in \mathbb{R}^{m \times d}$ at iteration t
  2. Compute mean and standard deviation for every feature

$$\mu_j^t = \mathbb{E}[x_j^t], \quad \sigma_j^t = \sqrt{\mathbb{E}[(x_j^t - \mu_j^t)^2]}, \quad \forall j \in \{1, \dots, d\}$$

  3. Normalize each feature (note **different for every batch**)

$$\tilde{x}_{i,j}^t = \frac{x_{i,j}^t - \mu_j^t}{\sigma_j^t}$$

  4. Output $\tilde{X}^t$

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In Advances in Neural Information Processing Systems (pp. 2483-2493).

Because BatchNorm Removes Linear Effects, Extra Linear Parameters Are Also Learned

- The form of this final update is:

$$\tilde{x}_{i,j}^t = \frac{x_{i,j}^t - \mu_j^t}{\sigma_j^t} \cdot \gamma_j + \beta_j$$

  - Where γ_j and β_j are learnable parameters
  - While μ_j^t and σ_j^t are computed from the **minibatch**
- But how do we compute μ_j^t and σ_j^t during test time (i.e., no minibatch)?
- Use running average of mean and variance:

$$\mu_{run}^t = \lambda \mu_{run}^{t-1} + (1-\lambda)\mu_{batch}^t$$

$$\sigma^2{}_{run}^t = \lambda \sigma^2{}_{run}^{t-1} + (1-\lambda)\sigma^2{}_{batch}^t$$

For CNNs, the Channel Dimension Is Treated as a “Feature”

- If the input minibatch tensor is $X^t \in \mathbb{R}^{m \times c \times h \times w}$, then the channel dimension c is treated as a feature:

$$\mu_j^t = \mathbb{E}[x_j^t], \quad \sigma_j^t = \sqrt{\mathbb{E}[(x_j^t - \mu_j^t)^2]}, \quad \forall j \in \{1, \dots, c\}$$

  - Where the mean is taken over **both** the batch dimension m **and** the spatial dimensions h and w.
  - Called “Spatial Batch Normalization”.
- Variants: Instance, Group or Layer Normalization.

https://pytorch.org/docs/stable/nn.html#normalization-layers

BatchNorm Can Stabilize and Accelerate Training of Deep Models

- **To use in practice:**
  - Only normalize batches during training (`model.train()`)
  - Turn off after training (`model.eval()`)
    - Uses running average of mean and variance
- Surprisingly effective at stabilizing training, reducing training time, and producing better models
- Not fully understood why it works

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In Advances in Neural Information Processing Systems (pp. 2483-2493).

BatchNorm Demo: Let's create and inspect a batchnorm 2D (i.e., for images) layer

```python
# Demo of batchnorm
import torch
import torch.nn as nn
class BatchNormModel(nn.Module):
    def __init__(self, n_channels):
        super().__init__()
        self.bn = nn.BatchNorm2d(n_channels)

    def forward(self, x):
        x = self.bn(x)
        return x
n_channels = 3 # Each channel is treated as a "feature" for i
bn_model = nn.BatchNorm2d(n_channels)
list(bn_model.named_parameters())
```

```
[('weight',
  Parameter containing:
  tensor([1., 1., 1.], requires_grad=True)),
 ('bias',
  Parameter containing:
  tensor([0., 0., 0.], requires_grad=True))]
```

- Notice that there are weight and bias parameters for each channel.

BatchNorm's behavior during training

```python
def print_mean_std(A, label='unlabeled'):
    print(f'{label}: Mean and standard deviation across chann
    print(torch.mean(A, dim=(0,2,3))) # Sum
    print(torch.std(A, dim=(0,2,3), unbiased=False))
    print()

torch.manual_seed(0)
bn_model.train()
batch1 = 2*torch.randn((100, n_channels, 2, 2)) + torch.arang
batch2 = 3*torch.randn((100, n_channels, 2, 2)) + -5 # (N, C,
out1 = bn_model(batch1)
out2 = bn_model(batch2)

print_mean_std(batch1, 'batch1')
print_mean_std(out1, 'out1')
print_mean_std(batch2, 'batch2')
print_mean_std(out2, 'out2')
```

```
batch1: Mean and standard deviation across channels
tensor([0.0107, 1.0870, 2.0128])
tensor([2.0200, 1.9704, 2.1094])

out1: Mean and standard deviation across channels
tensor([ 1.4901e-08,  6.8545e-09, -3.9041e-08], grad_fn=
<MeanBackward1>)
tensor([1.0000, 1.0000, 1.0000], grad_fn=<StdBackward0>)

batch2: Mean and standard deviation across channels
tensor([-4.9791, -5.2417, -4.8956])
tensor([3.0027, 3.0281, 2.9813])

out2: Mean and standard deviation across channels
tensor([ 3.3081e-08,  7.1824e-08, -8.6427e-08], grad_fn=
<MeanBackward1>)
tensor([1.0000, 1.0000, 1.0000], grad_fn=<StdBackward0>)
```

- Notice that even though distributions of the batches are quite different and different across channels, the output has been renormalized across the channel to always have zero mean and unit variance.

What about BatchNorm's behavior during test time?

Let's set simulate two simple batches and then apply at test time

```
torch.manual_seed(0)
batch1 = torch.randn((100, n_channels, 2, 2)) + torch.arange(
batch2 = torch.randn((100, n_channels, 2, 2)) + 5 # (N, C, H,

bn_model.train()
out1 = bn_model(batch1)
out2 = bn_model(batch2)

bn_model.eval()  # Turn OFF dynamic normalization
print('Running mean and standard devaiation')
print(bn_model.running_mean)
print(torch.sqrt(bn_model.running_var))
print()

out1 = bn_model(batch1)
out2 = bn_model(batch2)
print_mean_std(batch1, 'batch1')
print_mean_std(out1, 'out1')
print_mean_std(batch2, 'batch2')
print_mean_std(out2, 'out2')
```

```
Running mean and standard devaiation
tensor([0.0987, 0.2405, 0.4342])
tensor([1.3707, 1.3690, 1.3793])

batch1: Mean and standard deviation across channels
tensor([0.0054, 1.0435, 2.0064])
tensor([1.0100, 0.9852, 1.0547])

out1: Mean and standard deviation across channels
tensor([-0.0681,  0.5865,  1.1398], grad_fn=<MeanBackward1>)
tensor([0.7368, 0.7197, 0.7647], grad_fn=<StdBackward0>)

batch2: Mean and standard deviation across channels
tensor([5.0070, 4.9194, 5.0348])
tensor([1.0009, 1.0094, 0.9938])

out2: Mean and standard deviation across channels
tensor([3.5808, 3.4178, 3.3355], grad_fn=<MeanBackward1>)
tensor([0.7302, 0.7373, 0.7205], grad_fn=<StdBackward0>)
```

- Notice that the running mean and running standard deviation are used for normalization during test time rather than the batch.

Residual Networks Add the Input to the Output of the CNN

- Most deep model layers have the form:

$$y = f(x)$$

  - Where f could be any function including a convolutional layer like $f(x) = \sigma(\text{Conv}(\sigma(\text{Conv}(x))))$

- Residual layers add back in the input:

$$y = f(x) + x$$

- Notice that $f(x)$ models the difference between x and y (hence the name **residual**).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

A Residual Network Enables Deeper Networks Because Gradient Information Can Flow Between Layers

- A data flow diagram shows the "shortcut" connections.
- Consider composing 2 residual layers:
  - $z^{(1)} = f_1(x) + x$
  - $z^{(2)} = f_2(z^{(1)}) + z^{(1)}$
- Or, equivalently:
  - $z^{(2)} = f_2(f_1(x) + x) + f_1(x) + x$
- If the residuals $= 0$, then this is merely the identity function.

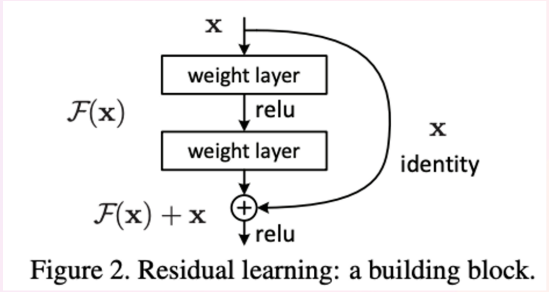
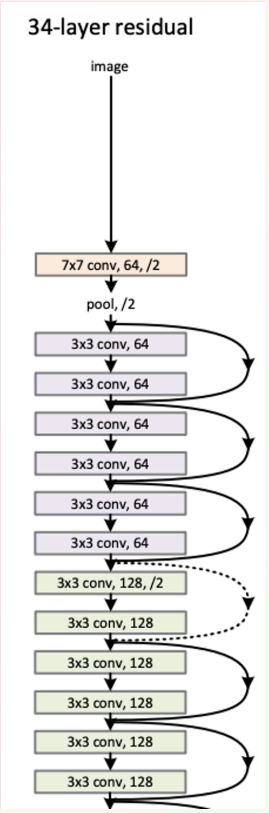


Figure 2. Residual learning: a building block.



Images from: He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

Detail: If the Dimensionality Is Not the Same, Then Use Either Fully Connected Layer or Convolution Layer to Match

- In the 1D case, suppose $f(x) : \mathbb{R}^d \to \mathbb{R}^m$, then we need to multiply x by linear operator to match the dimension:

$$y = f(x) + Wx, \quad \text{where } W \in \mathbb{R}^{m \times d}$$

- Similarly, for images, if $f(x) : \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c' \times h' \times w'}$, we can apply a convolution layer to match the dimensions:

$$y = f(x) + \text{conv}(x), \quad \text{where } \text{conv}(\cdot) : \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c' \times h' \times w'}$$

Residual Network Demo: Very simple residual network in PyTorch

(See https://towardsdatascience.com/residual-network-implementing-resnet-a7da63c7b278 for a tutorial on the real ResNet architectures from https://arxiv.org/abs/1512.03385)

- Code below simply loads CIFAR10 dataset like before.

▶ Code

Residual Network Demo: Very simple residual network in PyTorch

```
import torch.nn as nn
import torch.nn.functional as F

class ResidualNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 16, 5, padding=2)
        self.fc1 = nn.Linear(16 * 7 * 7, 120)
        self.fc2 = nn.Linear(120, 120)
        self.fc3 = nn.Linear(120, 10)

    def forward(self, x):
        # Input is (N, 3, 32, 32)
        x = self.pool(F.relu(self.conv1(x)))  # (N, 16, 14, 1
        # RESIDUAL LAYER
        x = self.pool(F.relu(self.conv2(x)) + x)  # (N, 16, 7
        x = x.view(-1, 16 * 7 * 7) #
        x = F.relu(self.fc1(x))  # (N, 120)
        # RESIDUAL LAYER
        x = F.relu(self.fc2(x)) + x  # (N, 84)
        x = self.fc3(x)  # (N, 10)
        return x

net = ResidualNet()
```

Let's train our very simple residual network

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(
    net.parameters(), lr=0.001, momentum=0.9)
for epoch in range(2):  # loop over dataset
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batch
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')
```

```
[1,  2000] loss: 1.825
[1,  4000] loss: 1.509
[1,  6000] loss: 1.395
[1,  8000] loss: 1.365
[1, 10000] loss: 1.281
[1, 12000] loss: 1.239
[2,  2000] loss: 1.158
[2,  4000] loss: 1.149
[2,  6000] loss: 1.100
[2,  8000] loss: 1.101
[2, 10000] loss: 1.088
[2, 12000] loss: 1.059
Finished Training
```

Evaluate our Residual Network

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the non-residual CNN on the 10000 test ima
print('Accuracy of the network on the 10000 test images: %d %
    100 * correct / total))
```

```
Accuracy of the non-residual CNN on the 10000 test images: 53
%
Accuracy of the network on the 10000 test images: 61 %
```

U-Nets Have an Autoencoder Structure With Skip Connections for **Semantic Segmentation** Task

- Concatenation + convolution rather than residual skip connections
- **Any** (pretrained) classification backbone can be used for encoder
- State-of-the-art semantic segmentation are based on this idea

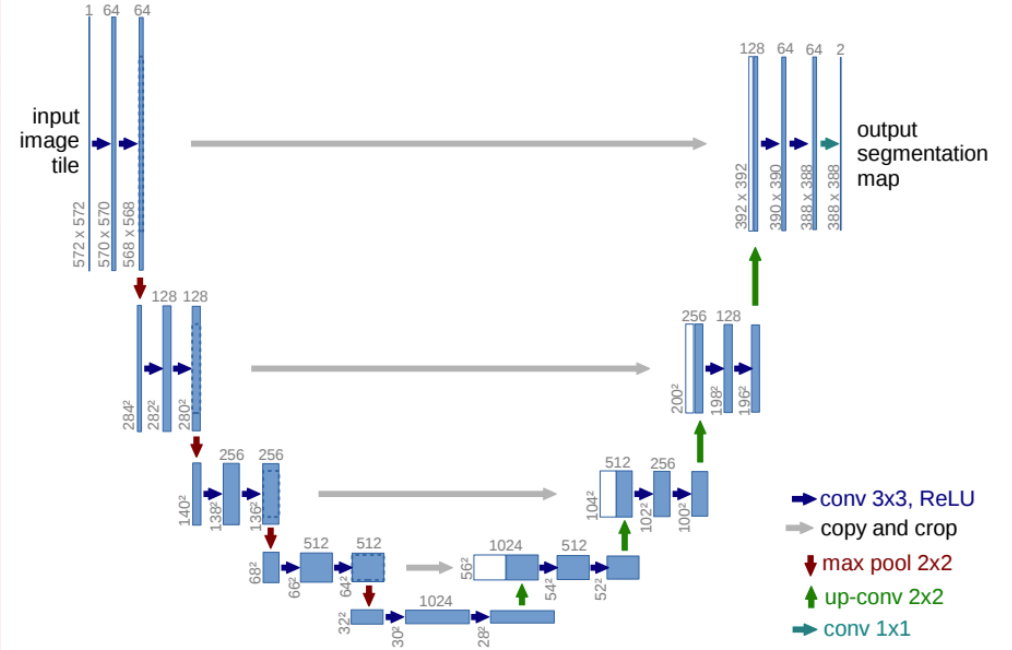


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Figure from: Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham.