ADVANCED ARTIFICIAL INTELLIGENCE: COMPUTER VISION

(tips and hints for lab work)

Attendance

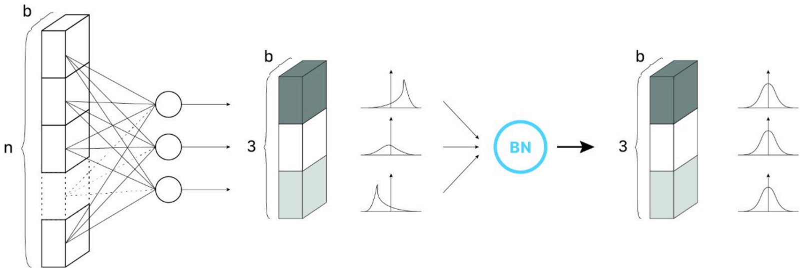
Today's labs

- Lab: add batch normalization to CNN for CIFAR10
- Lab: finetune ResNet18 with CIFAR10 (or MNIST) data
  - retrain the whole network from the pretrained weights
  - freeze part of the network (at the pretrained weights) and train the rest
- Lab: generate MNIST-like images
- Lab: generate CIFAR-like images

Note on image data in custom data loader

- jpg images are 3-channel color images (we see them as grayscale, when R, G, and B channels have the same values)
- image data values stored as `uint8` (values between 0 and 255)
- for display image data must be stored in the format $H \times W \times C$
- but pytorch requires data to be in $C \times H \times W$ and `float32` (typically in the range 0.0–1.0)
- transform from `utin8` HWC to `float` CHW either:
  - explicitly in the `__getitem__` method, or
  - by using `ToTensor()` transformation (`transform=transforms.ToTensor()`)

Batch normalization layer



Batch normalization layer I

- in order to reduce vanishing or exploding gradient problems with SGD, it is often useful to normalize data in every layer
- similar to standardizing input data
- batch normalization: replace activation vector z_n by $\tilde{z}_n$

$$\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} z$$

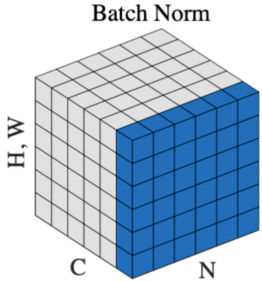
$$\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} (z - \mu_{\mathcal{B}})^2$$

$$\hat{z}_n = \frac{(z - \mu_{\mathcal{B}})}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$\tilde{z}_n = \gamma \hat{z}_n + \beta$$

Batch normalization layer II

- done dynamically in every layer with every batch
  - trainable parameters: one γ and one β per feature map per layer
  - note: mean and variance should be held fixed at inference time (`net.eval()`, `net.train()`)

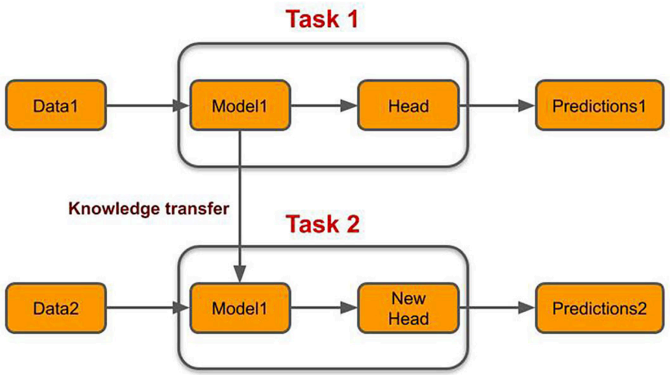


```python
class CNN_bn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.bn1 = nn.BatchNorm2d(6)

        self.conv2 = nn.Conv2d(6, 16, 5)
        self.bn2 = nn.BatchNorm2d(16)

        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.bn3 = nn.BatchNorm1d(120)

        self.fc2 = nn.Linear(120, 84)
        self.bn4 = nn.BatchNorm1d(84)

        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.bn1(self.pool(F.relu(self.conv1(x))))
        x = self.bn2(self.pool(F.relu(self.conv2(x))))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Two modes for forward pass: training vs evaluation

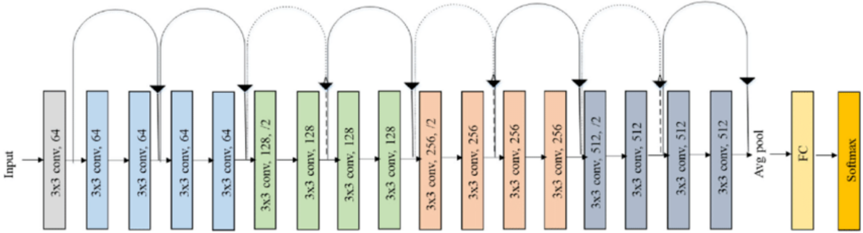
```
net.eval()

correct = 0
total = 0
# no need to compute gradients, since we are not training
with torch.no_grad():
    for data in test_loader:
        images, labels = data

        # run images through the network
        outputs = net(images)

        # class with largest value is our prediction
        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy on test images: {100 * correct // total} %')
```

Transfer learning



Finetuning Resnet18 — starting from a pre-trained network

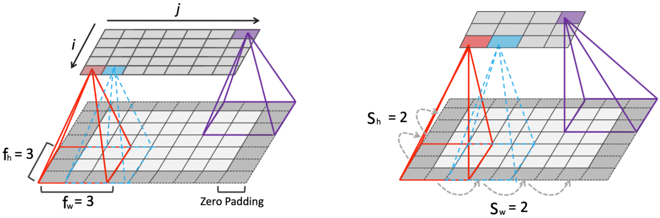
```python
from torchvision.models import resnet18, ResNet18_Weights
weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights, progress=False)

input_features = model.fc.in_features
model.fc = nn.Linear(in_features=input_features, out_features=10, bias=True)

model.conv1 = nn.Conv2d(3, 64, kernel_size=(3, 3), stride=1, padding=1, bias=False)
```



Finetuning Resnet18 — Freezing layers

```python
for param in model.parameters():
  param.requires_grad = False

# model.conv1.requires_grad_(True)
model.layer4.requires_grad_(True)
model.fc.requires_grad_(True)
```

Strided convolution



- with a filter of size $f_h \times f_w$, an image of size $x_h \times x_w$, and padding of size p_h and p_w on each side, the output size is

$$(x_h + 2p_h - f_h + 1) \times (x_w + 2p_w - f_w + 1)$$

- if we perform the convolution every s pixels, output size is

$$\left(\left\lfloor \frac{x_h + 2p_h - f_h}{s_h} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{x_w + 2p_w - f_w}{s_w} \right\rfloor + 1 \right)$$