Lec7_Machine_Learning&Deep_Learning

Traditional machine learning vs. deep learning

Traditional: eliminate the variations of illumination, viewpoints, scales, Occlusions, Background Clutter, Intra-class Variations

Gradient decent(optimal loss function)

The line: $y = a * x + b$

The model: $z = a * x - y + b$

output 1 if $z > 0$

output -1 if $z \leq 0$

Loss Function - Initialization: random num a' and b'

$$L(a', b') = \frac{1}{N}\sum_{i=1}^{N}(z'_i - z_i)^2$$

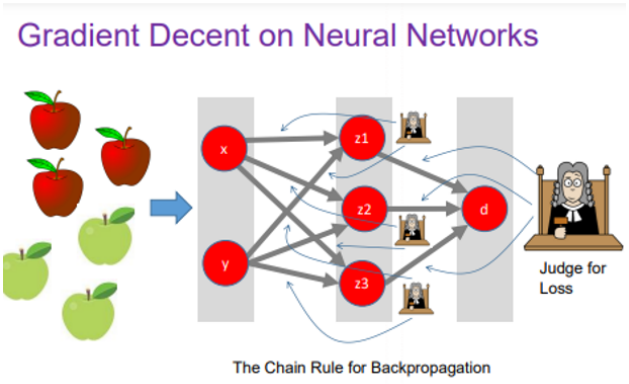
optimal parameter: $\partial L/\partial a \to 0$, $\partial L/\partial b \to 0$ update a', b' by pushing gradients towards 0

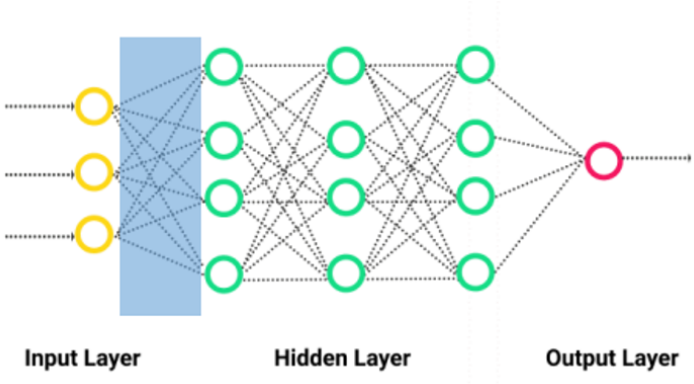
$a' = a' - \partial L/\partial a$, $b' = b' - \partial L/\partial a$ Learning rate: δ

$$a' = a' - \delta\frac{\partial L}{\partial a'}$$

$$b' = b' - \delta\frac{\partial L}{\partial b'}$$

Neural networks





Input layer: Receives data from external sources (data files, images, sensors, etc.) **Hidden layers** : Process data **Output layer**: Provides network-based functions for one or more data points

**Implement with Neural Network

- Convolutional filter:** [[1,0,1][0,1,0][1,0,1]] **-** Neurons are not fully connected, which results in **Local Reception Fields**
- Weights of the filters can be learned by **Backpropagation**

Pooling: max pooling, mean pooling

Activation

$$\text{sgn}(x) = \begin{cases} 1, & x \geqslant 0; \\ 0, & x < 0. \end{cases} \qquad \text{sigmoid}(x) = \frac{1}{1+e^{-x}} \qquad ReLU(x) = \max(0, x)$$

Convolutional Networks Input(kernel) → Feature Maps: polling(Convolution + ReLU) → flatten layer → *classification*: fully connected layer → *Probabilistic Distribution*: output - SoftMax Activation Function

AlexNet: (Convolutions→ subsampling) → full connections → Gaussian connections

VGG16: convolution + ReLU, max pooling, fully connected + ReLU

ResNet

Tutorial

```
**# Simple NN -** inhert torch.nn.Module class.
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()

        # input_size corresponds to input feature dimension.
        # output_size coressponds to class number.
        self.linear = nn.Linear(input_size, output_size)

        #Then, we use sigmoid activation function to gain the probability.
        # when probability is larger than 0.5, we treat it as positive 1. Else, we
treat it as negative 0.
        self.sigmoid = nn.Sigmoid()

    # forward function is inherted from parent's class. x denotes the input
feature.
    def forward(self, x):
        y_pred = self.linear(x)
        y_pred = self.sigmoid(y_pred)
        return y_pred
model = SimpleNN()
criterion = nn.BCELoss() # create loss function. BCE is binary cross entropy loss
# create optimizer. 1st parameter: the parameters will be optimized; 2nd: learning
rate
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
# training - set training flag
model.train()
num_epochs = 50000
for epoch in range(num_epochs):
    if torch.cuda.is_available():
        inputs = Variable(x_train).cuda()
        target = Variable(y_train).cuda()
    else:
        inputs = Variable(x_train)
        target = Variable(y_train)
    out = model(inputs) # forward() function
    loss = criterion(out, target)     # calculate loss
    optimizer.zero_grad()    # clear gradient
    loss.backward() # backward propagation
    optimizer.step() # Updating parameters via SGD
    if (epoch+1) % 1000 == 0:
        print('Epoch[{}/{}], loss: {:.6f}'
              .format(epoch+1, num_epochs, loss.item()))
# testing
model.eval()
results = model(Variable(x_test))
# display the results
for sample, result in zip(samples['val'], results):
    sample.pred = 1 if result > 0.5 else 0
    print(sample.fname, 'with label', sample.label, 'is predicted as',
sample.pred)


# **3-layer NN**
class SimpleNN_3layer(nn.Module):
    def __init__(self):
        super(SimpleNN_3layer, self).__init__()
        self.linear1 = nn.Linear(input_size, int(input_size/2))
        self.act1 = nn.ReLU()
        self.linear2 = nn.Linear(int(input_size/2), output_size)
        self.act2 = nn.Sigmoid()
    def forward(self, x):
        x = self.linear1(x)
        x = self.act1(x)
        x = self.linear2(x)
        y_pred = self.act2(x)
        return y_pred


**# Deep Neural Networks**
norm_mean = [0.485, 0.456, 0.406]
norm_std = [0.229, 0.224, 0.225]
inference_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])
def img_transform(img_rgb, transform=None):
    # img_rgb: PIL Image, transform: torchvision.transform, return: tensor
```

```
    if transform is None:
        raise ValueError("there is no transform")
    img_t = transform(Image.fromarray(img_rgb))
    return img_t
train_imgs = [img_transform(sample.img, inference_transform) for sample in
samples['train']]
train_imgs = torch.stack(train_imgs, dim=0)
test_imgs = [img_transform(sample.img, inference_transform) for sample in
samples['val']]
test_imgs = torch.stack(test_imgs, dim=0) # load data

class classification_head(nn.Module):
    def __init__(self,in_ch,num_classes):
        super(classification_head,self).__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(in_ch,num_classes)
    def forward(self, x):
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

# define ResNet model.
# For original ResNet, its final layer will output 1000 class number.
class Net(nn.Module):
    def __init__(self, num_class,pretrained=True):
        super(Net,self).__init__()
        model = models.resnet50(pretrained=pretrained)
        self.backbone =  nn.Sequential(*list(model.children())[:-2]) #remove the
last Avgpool and Fully Connected Layer
        self.classification_head = classification_head(2048, num_class)

    def forward(self,x):
        x = self.backbone(x)
        output = self.classification_head(x)
        return output

model = Net(1)
# fix the weights of ResNet except the last layer. This is because the training
set is small.
for p in model.backbone.parameters():
  p.requires_grad = False
model.to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(filter(lambda p: p.requires_grad, model.parameters()),
lr=0.001)
model.train()
```