Mathematical Foundations of Deep Neural Networks, M1407.001200
E. Ryu
Spring 2024

Homework 7
Due 5pm, Monday, April 29, 2024

Problem 1: *The true softmax.* Define

$$\nu_\beta(x) = \frac{1}{\beta} \log \sum_{i=1}^{n} \exp(\beta x_i).$$

Clearly, $\nu_\beta \colon \mathbb{R}^n \to \mathbb{R}$ is differentiable. Show that

(a) $\nu_\beta(x) \to \max\{x_1, \dots, x_n\}$ as $\beta \to \infty$.

(b) $\nabla \nu_1 = \mu$, where μ is the softmax function.

(c) If $i_{\max} = \operatorname{argmax}_{1 \le i \le n} x_i$ is uniquely defined, then $\nabla \nu_\beta(x) \to e_{i_{\max}}$ as $\beta \to \infty$, where $\{e_1, \dots, e_n\}$ is the standard basis of $\mathbb{R}^n$.

Remark. The parameter β is referred to as *inverse temperature*, since $\sum_{i=1}^n \exp(\beta x_i)$ is the "partition function" of statistical physics when $\beta = \frac{1}{k_B T}$, k_B is the Boltzmann constant, and T is the temperature. The regime $\beta \to \infty$ is therefore referred to as the *low-temperature regime.*

Solution.

(a) Using the L'Hospital's rule as $\beta \to \infty$, we get

$$\begin{aligned}
\lim_{\beta\to\infty} \nu_\beta(x) &= \lim_{\beta\to\infty} \frac{\log \sum_{i=1}^n \exp(\beta x_i)}{\beta} \\
&= \lim_{\beta\to\infty} \frac{\left(\sum_{i=1}^n x_i \exp(\beta x_i)\right) / \left(\sum_{i=1}^n \exp(\beta x_i)\right)}{1} \\
&= \lim_{\beta\to\infty} \frac{\sum_{i=1}^n x_i \exp(\beta x_i)}{\sum_{i=1}^n \exp(\beta x_i)} \\
&= \lim_{\beta\to\infty} \frac{\sum_{i=1}^n x_i \exp\left(\beta(x_i - \max_j x_j)\right)}{\sum_{i=1}^n \exp\left(\beta(x_i - \max_j x_j)\right)} \\
&= \frac{\sum_{i=1}^n x_i \mathbf{1}_{\{x_i = \max_j x_j\}}}{\sum_{i=1}^n \mathbf{1}_{\{x_i = \max_j x_j\}}} \\
&= \max\{x_1, \dots, x_n\}.
\end{aligned}$$

(b) From

$$\nu_1(x) = \log\left(\sum_{i=1}^n \exp(x_i)\right),$$

differentiating with respect to each x_j, we get

$$\frac{\partial}{\partial x_j}\nu_1(x) = \frac{\partial}{\partial x_j} \log\left(\sum_{i=1}^n \exp(x_i)\right) = \frac{\exp(x_j)}{\sum_{i=1}^n \exp(x_i)}.$$

Therefore, $\nabla \nu_1 = \mu$ where μ is a softmax function.

(c) We have

$$\left(\nabla \nu_\beta(x)\right)_j = \frac{\exp \beta x_j}{\sum_{i=1}^n \exp(\beta x_i)}$$

Therefore,

$$\begin{aligned}\lim_{\beta\to\infty} \nabla \nu_\beta(x) &= \lim_{\beta\to\infty} \left(\frac{\exp(\beta x_j)}{\sum_{i=1}^n \exp(\beta x_i)} \right)_j \\ &= \lim_{\beta\to\infty} \left(\frac{\exp(\beta(x_j - x_{i_{\max}}))}{\sum_{i=1}^n \exp(\beta(x_i - x_{i_{\max}}))} \right)_j = e_{i_{\max}}.\end{aligned}$$

■

Problem 2: *Are linear layers compute-heavy?* In AlexNet, 96% of trainable parameters are in the final linear layers, and only 4% are in the convolutional layers. How many operations do the linear and convolutional layers require in the forward pass? Only count the additions and multiplications of these layers, and do not count the operations necessary for the activation, pooling, dropout, and softmax layers. Use the version of AlexNet defined in `counting_params.py`. Assume the input image has size $3 \times 227 \times 227$.

Remark. A more complete investigation in the spirit of this problem would count the arithmetic operations of a gradient computation via a backward pass. For the sake of simplicity, we only consider the forward pass.

Solution. For the convolutional layer, suppose the input has shape $c_{\text{in}} \times h_{\text{in}}^2$, kernel size is k, and the output has shape $c_{\text{out}} \times h_{\text{out}}^2$. The number of multiplications and additions are (number of output positions)$\times$(operations per output position), which is

$$(c_{\text{out}} \times h_{\text{out}}^2) \times (c_{\text{in}} \times k^2).$$

For the linear layer, suppose the input has length c_{in} and the output has length c_{out}. The number of multiplications and additions are

$$c_{\text{in}} \times c_{\text{out}}.$$

The number of multiplications and additions for each layer are:

layer	input size		kernel size	output size		#operations	
	c_{in}	h_{in}		c_{out}	h_{out}		
conv1	3	227	11	64	55	$(64 \times 55^2) \times (3 \times 11^2)$	= 70,276,800
pool1	64	55	3	64	27		
conv2	64	27	5	192	27	$(192 \times 27^2) \times (64 \times 5^2)$	= 223,948,800
pool2	192	27	3	192	13		
conv3	192	13	3	384	13	$(384 \times 13^2) \times (192 \times 3^2)$	= 112,140,288
conv4	384	13	3	256	13	$(256 \times 13^2) \times (384 \times 3^2)$	= 149,520,384
conv5	256	13	3	256	13	$(256 \times 13^2) \times (256 \times 3^2)$	= 99,680,256
pool5	256	13	3	256	6		
linear1	9,216			4,096		$9,216 \times 4,096$	= 37,748,736
linear2	4,096			4,096		$4,096 \times 4,096$	= 16,777,216
linear3	4,096			1,000		$4,096 \times 1,000$	= 4,096,000

The convolutional layers require 655,566,528 multiplications and additions, which consume 91.79% of total operations. The linear layers require 58,621,952 multiplications and additions, which consume 8.21% of total operations. ■

Problem 3: *Removing BN after training.* During training, the addition of batch norm adds additional operations that were otherwise not present and therefore increases the computational cost per iteration. During testing, however, the effect of batch normalization can be combined with the preceding convolutional or linear layer so that no additional computational cost is incurred. Download the starter code `bn_remove.py` and the save file `smallNetSaved` and carry out the removal of the batchnorm layers. Specifically, load the pre-trained `smallNetTrain` model and set the weights and parameters of `smallNetTest` so that the two models produce exactly the same outputs on the test set.

Solution. See `bn_remove_sol.py`. ■

Problem 4: *Backprop with convolutions.* Consider 1D convolutions with single input and output channels, stride 1, and padding 0. Let $w_1, \dots, w_L$ be convolutional filters with sizes $f_1, \dots, f_L$. Let $A_{w_\ell} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, where $n_\ell = n_{\ell-1} - f_\ell + 1$, be the matrix representing convolution with w_ℓ, i.e., multiplication by A_{w_ℓ} is equivalent to convolution with w_ℓ, for $\ell = 1, \dots, L$. Let $\sigma \colon \mathbb{R} \to \mathbb{R}$ be a differentiable activation function. Consider the convolutional neural network

$$\begin{aligned} y_L &= A_{w_L} y_{L-1} + b_L \mathbf{1}_{n_L} \\ y_{L-1} &= \sigma(A_{w_{L-1}} y_{L-2} + b_{L-1} \mathbf{1}_{n_{L-1}}) \\ &\vdots \\ y_2 &= \sigma(A_{w_2} y_1 + b_2 \mathbf{1}_{n_2}) \\ y_1 &= \sigma(A_{w_1} x + b_1 \mathbf{1}_{n_1}), \end{aligned}$$

where $x \in \mathbb{R}^{n_0}$, $b_\ell \in \mathbb{R}$, $\mathbf{1}_{n_\ell} \in \mathbb{R}^{n_\ell}$ is the vector with all entries being 1, and $n_L = 1$. For notational convenience, define $y_0 = x$.

(a) Define

$$v_L = 1, \qquad v_\ell = \frac{\partial y_L}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) \qquad \text{for } \ell = 1, \dots, L-1.$$

Let $\mathcal{C}_{v_\ell^\intercal}$ be the 1D convolutional operator defined by interpreting $v_\ell^\intercal \in \mathbb{R}^{n_\ell}$ as a convolutional filter for $\ell = 1, \dots, L$. Show that

$$\begin{aligned} &\frac{\partial y_L}{\partial y_{L-1}} = A_{w_L}, \qquad \frac{\partial y_\ell}{\partial y_{\ell-1}} = \operatorname{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) A_{w_\ell} \qquad \text{for } \ell = 2, \dots, L-1 \\ &\frac{\partial y_L}{\partial w_\ell} = (\mathcal{C}_{v_\ell^\intercal} y_{\ell-1})^\intercal \qquad \text{for } \ell = 1, \dots, L \\ &\frac{\partial y_L}{\partial b_\ell} = v_\ell \mathbf{1}_{n_\ell} \qquad \text{for } \ell = 1, \dots, L. \end{aligned}$$

(b) As discussed in homework 1, forming the full matrix A_{w_ℓ} is wasteful and should be avoided. Describe how matrix-vector or vector-matrix products with respect to A_{w_i} or $A_{w_i}^\intercal$ should be used in the forward pass and backpropagation.

Clarification. A matrix-vector product $A_{w_i} v$ should be computed by performing convolution. A vector-matrix product $u^\intercal A_{w_i} = (A_{w_i}^\intercal u)^\intercal$ should be computed by performing transpose-convolution, which was discussed in homework 1.

Solution.

(a) Since the convolution with w_ℓ is equal to the multiplication by A_{w_ℓ}, differentiation by y_ℓ and b_ℓ are essentially the same as in the case of problem 6, homework 4.

$$\begin{aligned} \frac{\partial y_L}{\partial y_{L-1}} &= A_{w_L} \\ \frac{\partial y_\ell}{\partial y_{\ell-1}} &= \operatorname{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) A_{w_\ell} \end{aligned}$$

for all $\ell = 1, \dots, L$ and

$$\frac{\partial y_\ell}{\partial b_\ell} = \frac{\partial y_\ell}{\partial (b_\ell \mathbf{1}_{n_l})} \frac{\partial (b_\ell \mathbf{1}_{n_l})}{\partial b_\ell} = \operatorname{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) \mathbf{1}_{n_\ell}$$

for all $\ell = 1, \dots, L-1$, so

$$\frac{\partial y_L}{\partial b_\ell} = \frac{\partial y_L}{\partial y_\ell} \operatorname{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) \mathbf{1}_{n_\ell} = v_\ell \mathbf{1}_{n_\ell}.$$

It remains to compute $\frac{\partial y_\ell}{\partial w_\ell}$, which is, by the chain rule,

$$\frac{\partial y_L}{\partial w_\ell} = \frac{\partial y_L}{\partial y_\ell} \frac{\partial y_\ell}{\partial w_\ell}.$$

From

$$\begin{aligned} y_\ell &= \sigma(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell}) \\ &= \begin{bmatrix} \vdots \\ \sigma\left((A_{w_\ell} y_{\ell-1})_i + b_\ell\right) \\ \vdots \end{bmatrix} \\ &= \begin{bmatrix} \vdots \\ \sigma\left(\Sigma_{j=i+1}^{i+f_\ell} (w_\ell)_{j-i} (y_{\ell-1})_{j-1} + b_\ell\right) \\ \vdots \end{bmatrix} \end{aligned} \tag{1}$$

we get

$$\begin{aligned} \frac{\partial y_\ell}{\partial (w_\ell)_k} &= \begin{bmatrix} \sigma'((A_{w_\ell} y_{\ell-1})_1 + b_\ell) \cdot (y_{\ell-1})_k \\ \vdots \\ \sigma'((A_{w_\ell} y_{\ell-1})_i + b_\ell) \cdot (y_{\ell-1})_{i+k-1} \\ \vdots \\ \sigma'((A_{w_\ell} y_{\ell-1})_{n_\ell} + b_\ell) \cdot (y_{\ell-1})_{n_\ell+k-1} \end{bmatrix} \\ &= \operatorname{diag}\left(\sigma'((A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell}))\right) \begin{bmatrix} (y_{\ell-1})_k \\ \vdots \\ (y_{\ell-1})_{i+k-1} \\ \vdots \\ (y_{\ell-1})_{n_\ell+k-1} \end{bmatrix} \end{aligned} \tag{2}$$

Thus,

$$\frac{\partial y_\ell}{\partial w_\ell} = \operatorname{diag}\left(\sigma'((A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell}))\right) \begin{bmatrix} (y_{\ell-1})_1 & (y_{\ell-1})_2 & \cdots & (y_{\ell-1})_{f_\ell} \\ \vdots & & & \\ (y_{\ell-1})_i & (y_{\ell-1})_{i+1} & \cdots & (y_{\ell-1})_{i+f_\ell-1} \\ \vdots & & & \\ (y_{\ell-1})_{n_\ell} & (y_{\ell-1})_{n_\ell+1} & \cdots & (y_{\ell-1})_{n_\ell+f_\ell-1} \end{bmatrix}$$

$$\frac{\partial y_L}{\partial w_\ell} = v_\ell \begin{bmatrix} (y_{\ell-1})_1 & (y_{\ell-1})_2 & \cdots & (y_{\ell-1})_{f_\ell} \\ \vdots & & & \\ (y_{\ell-1})_i & (y_{\ell-1})_{i+1} & \cdots & (y_{\ell-1})_{i+f_\ell-1} \\ \vdots & & & \\ (y_{\ell-1})_{n_\ell} & (y_{\ell-1})_{n_\ell+1} & \cdots & (y_{\ell-1})_{n_\ell+f_\ell-1} \end{bmatrix}$$

Notice that this is exactly the transpose of the convolution of $y_{\ell-1}$ with $v_\ell^\intercal$ as a convolutional filter.

(b) Notice that the vector-matrix product with respect to $A_{w_i}^\intercal$ is absent in neither the forward pass nor the backpropagation, according to (a). Only the matrix-vector product with respect to A_{w_i} remains.

For the forward pass, we only need to store the convolutional filter w_i's and process the convolution with w_i's when necessary. For the backpropagation, we need to additionally store the vectors v_ℓ in reverse order starting from $\ell = L$ during the back propagation, to calculate the gradients $\frac{\partial y_L}{\partial w_\ell}$ and $\frac{\partial y_L}{\partial b_\ell}$ from $\ell = L, \ldots, 1$.

■

Problem 5: *Larger network in network.* Consider the convolutional neural network `Net1` designed to classify the CIFAR10 dataset. Use the starter code `LNiN.py`.

```
class Net1(nn.Module):
  def __init__(self, num_classes=10):
    super(Net1, self).__init__()
    self.features = nn.Sequential(
      nn.Conv2d(3, 64, kernel_size=7, stride=1),
      nn.ReLU(),
      nn.Conv2d(64, 192, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(192, 384, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(384, 256, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(256, 256, kernel_size=3, stride=1),
    )
    self.classifier = nn.Sequential(
      nn.Linear(256 * 18 * 18, 4096),
      nn.ReLU(),
      nn.Linear(4096, 4096),
      nn.ReLU(),
      nn.Linear(4096, num_classes)
    )

  def forward(self, x):
    x = self.features(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

(a) Consider `Net2`, which replaces the fully-connected layers of `Net1` with convolutional layers. Implement `Net2` and the weight initialization function so that `Net1` and `Net2` are equivalent in the following sense: When the parameters of `Net1` are appropriately copied over, `Net2` produces exactly the same output as `Net1` for inputs of size $B \times 3 \times 32 \times 32$.

```
class Net2(nn.Module):
  def __init__(self, num_classes=10):
    super(Net2, self).__init__()
    self.features = nn.Sequential(
      nn.Conv2d(3, 64, kernel_size=7, stride=1),
      nn.ReLU(),
      nn.Conv2d(64, 192, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(192, 384, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(384, 256, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.Conv2d(256, 256, kernel_size=3, stride=1),
    )
  ##########################################################
  ### TODO: Complete initialization of self.classifier    ###
  ###        by filling in the ...                         ###
  ##########################################################
    self.classifier = nn.Sequential(
      nn.Conv2d(...),
      nn.ReLU(),
      nn.Conv2d(...),
      nn.ReLU(),
      nn.Conv2d(...)
    )

  def copy_weights_from(self, net1):
    with torch.no_grad():
      for i in range(0, len(self.features), 2):
        self.features[i].weight.copy_(net1.features[i].weight)
        self.features[i].bias.copy_(net1.features[i].bias)

      for i in range(len(self.classifier)):
        ####################################################
        ### TO DO: Correctly transfer weight of Net1     ###
        ####################################################

  def forward(self, x):
    x = self.features(x)
    x = self.classifier(x)
    return x

model1 = Net1() # model1 randomly initialized
model2 = Net2()
model2.copy_weights_from(model1)

test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    transform=torchvision.transforms.ToTensor()
)
test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10
)

imgs, _ = next(iter(test_loader))
diff = torch.mean((model1(imgs)-model2(imgs).squeeze())**2)
print(f"Average Pixel Diff: {diff.item()}") # should be small
```

(b) Let `X` be a tensor of size $B \times 3 \times h \times w$ with $h > 32$ and $w > 32$. While `Net2` can take `X` as input, `Net1` cannot. By appropriately filling in `...`, describe how `Net2` applied to `X` is equivalent to `Net1` applied to patches of `X`.

```
# Continues from code of (a)
test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.Resize((36, 38)),
        torchvision.transforms.ToTensor()
        ]),
    download=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10,
    shuffle=False
)

images, _ = next(iter(test_loader))
b, w, h = images.shape[0], images.shape[-1], images.shape[-2]
out1 = torch.empty((b, 10, h - 31, w - 31))
for i in range(h - 31):
  for j in range(w - 31):
    ############################################################
    ### TO DO: fill in ... to make out1 and out2 equal      ###
    ############################################################
    out1[:, :, i, j] = model1(...)
out2 = model2(images)
diff = torch.mean((out1-out2)**2)

print(f"Average Pixel Diff: {diff.item()}")
```

Solution. See `LNiN_sol.py`. ■