



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, $\nu_\beta: \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable. Show that

- (a) $\nu_\beta(x) \rightarrow \max\{x_1, \dots, x_n\}$ as $\beta \rightarrow \infty$.
- (b) $\nabla \nu_1 = \mu$, where μ is the softmax function.
- (c) If $i_{\max} = \operatorname{argmax}_{1 \leq i \leq n} x_i$ is uniquely defined, then $\nabla \nu_\beta(x) \rightarrow e_{i_{\max}}$ as $\beta \rightarrow \infty$, 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 $\beta \rightarrow \infty$ is therefore referred to as the *low-temperature regime*.

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.

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.

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 $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ 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

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

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

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

(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 $A_{w_i}^\top$ 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 $u^\top A_{w_i} = (A_{w_i}^\top u)^\top$ should be computed by performing transpose-convolution, which was discussed in homework 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 \mathbf{X} be a tensor of size $B \times 3 \times h \times w$ with $h > 32$ and $w > 32$. While Net2 can take \mathbf{X} as input, Net1 cannot. By appropriately filling in \dots , describe how Net2 applied to \mathbf{X} is equivalent to Net1 applied to patches of \mathbf{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()}")
```