Mathematical Foundations of Deep Neural Networks, M1407.001200
E. Ryu
Spring 2021

Final Exam

Thursday, December 16, 2021, 3:30–7:30 pm
4 hours, 6 questions, 100 points, 16 pages

This exam is open-book in the sense that you may use any non-electronic resource.
While we don't expect you will need more space than provided,
you may continue on the back of the pages.

Name: ______________________________

1. (15 points) *Removing BN after Conv-ReLU.* Assume we have trained the model `ConvReLUBN`, which applies 2D convolution, ReLU, and then batch normalization as specified in the code in the following page. Let $\rho\colon \mathbb{R} \to \mathbb{R}$ denote the ReLU activation function. Let

$$\mathrm{BN}_{\gamma,\beta,\mu,\sigma,\varepsilon}(\rho(\mathrm{Conv2D}_{w,b}(X)))$$

with parameters $w \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times f_1\times f_2}$, $b \in \mathbb{R}^{C_{\text{out}}}$, $\gamma \in \mathbb{R}^{C_{\text{out}}}$, $\beta \in \mathbb{R}^{C_{\text{out}}}$, $\mu \in \mathbb{R}^{C_{\text{out}}}$, $\sigma \in \mathbb{R}^{C_{\text{out}}}$, and $\varepsilon > 0$ denote the Conv-ReLU-BN network applied to a 4D input tensor X. Here, μ and σ denote the running mean and standard deviation of the BN layer. We now wish to construct a network without batch normalization that is equivalent to the original network in `eval()` mode.

(a) Show that it may not be possible to find $\tilde{w} \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times f_1\times f_2}$ and $\tilde{b} \in \mathbb{R}^{C_{\text{out}}}$ such that

$$\mathrm{BN}_{\gamma,\beta,\mu,\sigma,\varepsilon}(\rho(\mathrm{Conv2D}_{w,b}(X))) = \rho(\mathrm{Conv2D}_{\tilde{w},\tilde{b}}(X))$$

for all inputs X.

(b) Implement code that finds $\tilde{w} \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times f_1\times f_2}$, $\tilde{b} \in \mathbb{R}^{C_{\text{out}}}$, $S \in \{-1,+1\}^{C_{\text{out}}}$, and $B \in \mathbb{R}^{C_{\text{out}}}$ such that

$$\mathrm{BN}_{\gamma,\beta,\mu,\sigma,\varepsilon}(\rho(\mathrm{Conv2D}_{w,b}(X))) = S \odot \rho(\mathrm{Conv2D}_{\tilde{w},\tilde{b}}(X)) \oplus B$$

for all inputs X, where the $\odot$ and $\oplus$ operations are defined in the following broadcasted sense:

$$(S \odot Y \oplus B)_{b,c,i,j} = (S_c Y_{b,c,i,j} + B_c)$$

for $b = 1,\ldots,B$, $c = 1,\ldots,C$, and all spatial dimensions i and j. To clarify, S is a length C_{out} 1D tensor with entries -1 or $+1$. Your implementation should fill in the 4 instances of `...` in the following page.

(c) Can this problem be extended to the case where ρ is the leaky ReLU?

Clarification. The precise meaning of $\mathrm{BN}_{\gamma,\beta,\mu,\sigma,\varepsilon}(\rho(\mathrm{Conv2D}_{w,b}(X)))$ and $S \odot \rho(\mathrm{Conv2D}_{\tilde{w},\tilde{b}}(X)) \oplus B$ is clarified by the code in the following page.

```
import torch
import torch.nn as nn

### Conv-ReLU-BatchNorm ##
class ConvReLUBN(nn.Module):
  def __init__(self, in_chan, out_chan, f):
    super(ConvReLUBN, self).__init__()
    self.conv = nn.Conv2d(in_channels=in_chan,
                out_channels=out_chan, kernel_size=f)
    self.bn = nn.BatchNorm2d(out_chan)

  def forward(self, x):
    return self.bn(torch.relu(self.conv(x)))

### Conv-ReLU-Sign-Bias ###
class ConvReLUSignBias(nn.Module):
  def __init__(self, in_chan, out_chan, f):
    super(ConvReLUSignBias, self).__init__()
    self.conv = nn.Conv2d(in_channels=in_chan,
                out_channels=out_chan, kernel_size=f)
    self.S = torch.ones(out_chan)
    self.B = nn.parameter.Parameter(torch.zeros(out_chan))

  def init_from_conv_relu_bn(self, init_from):
    gamma, beta = init_from.bn.weight, init_from.bn.bias
    mean, var = init_from.bn.running_mean, init_from.bn.running_var
    eps = init_from.bn.eps
    with torch.no_grad():
      self.S = ...
      self.conv.weight = ...
      self.conv.bias = ...
      self.B = ...

  def forward(self, x):
    S = self.S.view(1, -1, 1, 1)
    B = self.B.view(1, -1, 1, 1)
    return S*torch.relu(self.conv(x)) + B

### Check equivalence of two models ###
model = ConvReLUBN(3, 4, 3) # Original model
model(torch.normal(torch.zeros((5, 3, 10, 10)), 1)) #compute BN stats
model_test = ConvReLUSignBias(3, 4, 3) # Equivalent model
model_test.init_from_conv_relu_bn(model)

# Compare two models on random 5x3x10x10 input
x = torch.normal(torch.zeros((5, 3, 10, 10)), 1)
model.eval()
y, y_test = model(x), model_test(x)
print(f"MSE between two inferences: {torch.mean((y-y_test)**2)}")
```

2. (20 points) *Larger network in network.* Consider the convolutional neural network `Net1` designed to classify the CIFAR10 dataset.

```
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),
    )

  ##############################################################
  ### TO DO: Complete the 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(len(self.features)):
        if i % 2 == 0:
          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()
...
train model1
...

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,
)

images, _ = next(iter(test_loader))
diff = torch.mean((model1(images)-model2(images).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 the same ###
    #########################################################
    out1[:, :, i, j] = model1(...)
out2 = model2(images)
diff = torch.mean((out1-out2)**2)

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

3. (15 points) *Anomaly detection via flow models.* Assume we have a trained flow model that we use to evaluate the likelihood function p_θ. In this problem, you will use this trained flow model to perform anomaly detection between the MNIST and KMNIST datasets. In step 1, load the MNIST and KMNIST datasets, and split the MNIST test dataset into "validation" and "test" sets. In step 2, define the flow model. In step 3, load the trained flow model. In step 4, calculate the mean and standard deviation of

$$\{\log p_\theta(Y_i)\}_{i=1}^{M},$$

where $Y_1, \ldots, Y_M$ are the validation data. Define a threshold to be mean $-$ 3 standard deviations, and define inputs with log likelihood below this threshold to be anomalies. In step 5, check how many of the MNIST images within the test set are classified as anomalies and report the type I error rate. In step 6, check how many of the KMNIST images are classified as non-anomalies and report the type II error rate. The following starter code provides the implementation of steps 1–3. Complete the implementation of steps 4–6.

Remark. In this problem, we split the test data into validation and test sets because the entire training set was already used to train the flow model. If we were to train the flow model from scratch, it would be better to split the training set into the training and validation sets to set aside the validation data for step 3.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 128

'''
Step 1:
'''
test_val_dataset = datasets.MNIST(root='./mnist_data/',
                train=False,
                transform=transforms.ToTensor())

test_dataset, validation_dataset = \
    torch.utils.data.random_split(test_val_dataset, [5000, 5000])

# KMNIST dataset, only need test dataset
anomaly_dataset = datasets.KMNIST(root='./kmnist_data/',
                train=False,
                transform=transforms.ToTensor(),
                download=True)

'''
Step 2:
'''
# Define prior distribution
class Logistic(torch.distributions.Distribution):
  def __init__(self):
    super(Logistic, self).__init__()

  def log_prob(self, x):
    return -(F.softplus(x) + F.softplus(-x))

  def sample(self, size):
    z = torch.distributions.Uniform(0., 1.).sample(size).to(device)
    return torch.log(z) - torch.log(1. - z)

# Implement coupling layer
class Coupling(nn.Module):
  def __init__(self, in_out_dim, mid_dim, hidden, mask_config):
    super(Coupling, self).__init__()
    self.mask_config = mask_config

    self.in_block = \
        nn.Sequential(nn.Linear(in_out_dim//2, mid_dim), nn.ReLU())
    self.mid_block = nn.ModuleList(
        [nn.Sequential(nn.Linear(mid_dim, mid_dim), nn.ReLU())
          for _ in range(hidden - 1)])
    self.out_block = nn.Linear(mid_dim, in_out_dim//2)

  def forward(self, x, reverse=False):
    [B, W] = list(x.size())
    x = x.reshape((B, W//2, 2))
    if self.mask_config:
      on, off = x[:, :, 0], x[:, :, 1]
    else:
      off, on = x[:, :, 0], x[:, :, 1]

    off_ = self.in_block(off)
    for i in range(len(self.mid_block)):
      off_ = self.mid_block[i](off_)
    shift = self.out_block(off_)

    if reverse:
      on = on - shift
    else:
      on = on + shift

    if self.mask_config:
      x = torch.stack((on, off), dim=2)
    else:
      x = torch.stack((off, on), dim=2)
    return x.reshape((B, W))

class Scaling(nn.Module):
  def __init__(self, dim):
    super(Scaling, self).__init__()
    self.scale = nn.Parameter(torch.zeros((1, dim)))

  def forward(self, x, reverse=False):
    log_det_J = torch.sum(self.scale)
    if reverse:
      x = x * torch.exp(-self.scale)
    else:
      x = x * torch.exp(self.scale)
    return x, log_det_J

class NICE(nn.Module):
  def __init__(self,in_out_dim, mid_dim, hidden,
        mask_config=1.0, coupling=4):
    super(NICE, self).__init__()
    self.prior = Logistic()
    self.in_out_dim = in_out_dim

    self.coupling = nn.ModuleList([
      Coupling(in_out_dim=in_out_dim,
            mid_dim=mid_dim,
            hidden=hidden,
            mask_config=(mask_config+i)%2) \
      for i in range(coupling)])

    self.scaling = Scaling(in_out_dim)

  def g(self, z):
    x, _ = self.scaling(z, reverse=True)
    for i in reversed(range(len(self.coupling))):
      x = self.coupling[i](x, reverse=True)
    return x

  def f(self, x):
    for i in range(len(self.coupling)):
      x = self.coupling[i](x)
    z, log_det_J = self.scaling(x)
    return z, log_det_J

  def log_prob(self, x):
    z, log_det_J = self.f(x)
    log_ll = torch.sum(self.prior.log_prob(z), dim=1)
    return log_ll + log_det_J

  def sample(self, size):
    z = self.prior.sample((size, self.in_out_dim)).to(device)
    return self.g(z)

  def forward(self, x):
    return self.log_prob(x)

'''
Step 3: Load the pretrained model
'''
nice = NICE(in_out_dim=784, mid_dim=1000, hidden=5).to(device)
nice.load_state_dict(torch.load('nice.pt', map_location=device))


'''
Step 4: Calculate standard deviation by using validation set
'''
validation_loader = torch.utils.data.DataLoader(
    dataset=validation_dataset, batch_size=batch_size)
...


'''
Step 5: Anomaly detection (mnist)
'''
test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset, batch_size=batch_size)
...
print(f'{count} type I errors among {len(test_dataset)} data')
```

```
'''
Step 6: Anomaly detection (kmnist)
'''
anomaly_loader = torch.utils.data.DataLoader(
    dataset=anomaly_dataset, batch_size=batch_size)
...
print(f'{count} type II errors among {len(anomaly_dataset)} data')
```

4. (20 points) *Ingredients of Glow.* Let

$$A = PL(U + \mathrm{diag}(s)) \in \mathbb{R}^{C\times C},$$

where $P \in \mathbb{R}^{C\times C}$ is a permutation matrix, $L \in \mathbb{R}^{C\times C}$ is a lower triangular matrix with unit diagonals, $U \in \mathbb{R}^{C\times C}$ is upper triangular with zero diagonals, and $s \in \mathbb{R}^C$. To clarify, $L_{ii} = 1$ for $i = 1, \dots, C$, $L_{ij} = 0$ for $1 \le i < j \le C$, and $U_{ij} = 0$ for $1 \le j \le i \le C$.

(a) Let $f_1(x) = Ax$. Show

$$\log \left| \frac{\partial f_1}{\partial x} \right| = \sum_{i=1}^{C} \log |s_i|.$$

(b) Given $h\colon \mathbb{R}^{a\times b\times c} \to \mathbb{R}^{a\times b\times c}$, define

$$\left| \frac{\partial h(X)}{\partial X} \right| = \left| \frac{\partial (h(X).\mathrm{reshape}(abc))}{\partial (X.\mathrm{reshape}(abc))} \right|,$$

i.e., we define the absolute value of the Jacobian determinant with the input and output tensors vectorized. Note that the reshape operation, which maps elements from the tensor in $\mathbb{R}^{a\times b\times c}$ to the elements of the vector in $\mathbb{R}^{abc}$, is not unique. Show that the definition of $\left| \frac{\partial h(X)}{\partial X} \right|$ does not depend on the specific choice of reshape.

(c) Let $f_2(X \mid P, L, U, s)$ be the 1×1 convolution from $\mathbb{R}^{C\times m\times n}$ to $\mathbb{R}^{C\times m\times n}$ with filter $w \in \mathbb{R}^{C\times C\times 1\times 1}$ defined as

$$w_{i,j,1,1} = A_{i,j}, \qquad \text{for } i = 1, \dots, C,\ j = 1, \dots, C.$$

So $X \in \mathbb{R}^{C\times m\times n}$ and $f_2(X \mid P, L, U, s) \in \mathbb{R}^{C\times m\times n}$. (Assume the batch size is 1.) Show

$$\log \left| \frac{\partial f_2(X \mid P, L, U, s)}{\partial X} \right| = mn \sum_{i=1}^{C} \log |s_i|.$$

(d) Consider the following coupling layer from $X \in \mathbb{R}^{2C\times m\times n}$ to $Z \in \mathbb{R}^{2C\times m\times n}$:

$$\begin{aligned} Z_{1:C,:,:} &= X_{1:C,:,:} \\ Z_{C+1:2C,:,:} &= f_2(X_{C+1:2C,:,:} | P, L(X_{1:C,:,:}), U(X_{1:C,:,:}), s(X_{1:C,:,:})), \end{aligned}$$

where P is a fixed permutation matrix, $L(\cdot)$ outputs lower triangular matrices with unit diagonals in $\mathbb{R}^{C\times C}$, $U(\cdot)$ outputs upper triangular matrices with zero diagonals in $\mathbb{R}^{C\times C}$, and $s(\cdot) \in \mathbb{R}^C$. Show

$$\log \left| \frac{\partial Z}{\partial X} \right| = mn \sum_{i=1}^{C} \log |s_i|.$$

Remark. Given any $A \in \mathbb{R}^{n\times n}$, a decomposition $A = PL(U + \mathrm{diag}(s))$ can be computed via the so-called PLU factorization, which performs steps analogous to Gaussian elimination.

5. (15 points) *GAN with non-uniform weights.* Consider the variant of the GAN with non-uniform weights on type I and type II errors:

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad \underset{\phi\in\mathbb{R}^p}{\text{maximize}} \quad \mathbb{E}_{X\sim p_{\text{true}}}[\log D_\phi(X)] + \lambda\mathbb{E}_{\tilde{X}\sim p_\theta}[\log(1-D_\phi(\tilde{X}))].$$

Here, $\lambda > 0$ represents the relative significance of a type II error over a type I error. Assuming the discriminator network D_ϕ is infinitely expressive, i.e., assuming $D_\phi\colon \mathbb{R}^n \to (0,1)$ can represent any function from $\mathbb{R}^n$ to $(0,1)$, show that the stated minimax problem is equivalent to

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad D_f(p_{\text{true}}\|p_\theta)$$

with

$$f(u) = \begin{cases} u\log\frac{u}{u+\lambda} + \lambda\log\frac{\lambda}{\lambda+u} + (1+\lambda)\log(1+\lambda) - \lambda\log\lambda & u \geq 0 \\ \infty & \text{otherwise.} \end{cases}$$

You may use the following facts without proof: $f(1) = 0$ and f is convex.

6. (15 points) In this problem, we consider the setup of training a VAE with a trainable prior. Specifically, we assume $Z \sim r_\lambda(z)$, where λ is a trainable parameter, and $X \sim p_\theta(x \mid Z)$. Let $q_\phi(z \mid X)$ be the approximate posterior. Let

$$\mathrm{VLB}_{\theta,\phi,\lambda}(X_i) = \mathbb{E}_{Z \sim q_\phi(z|X_i)} \left[\log \left(\frac{p_\theta(X_i \mid Z) r_\lambda(Z)}{q_\phi(Z \mid X_i)} \right) \right].$$

   (a) Show that $\log p_\theta(X_i) \geq \mathrm{VLB}_{\theta,\phi,\lambda}(X_i)$.

   (b) Describe how to evaluate stochastic gradients of $\mathrm{VLB}_{\theta,\phi,\lambda}(X_i)$ using the log-derivative trick.

   (c) Assume $r_\lambda = \mathcal{N}(\lambda_1, \mathrm{diag}(\lambda_2))$, where $\lambda_1, \lambda_2 \in \mathbb{R}^k$, $q_\phi(z \mid X_i) = \mathcal{N}(\mu_\phi(X_i), \Sigma_\phi(X_i))$ with diagonal Σ_ϕ, and $p_\theta(X_i \mid z) = \mathcal{N}(f_\theta(z), \sigma^2 I)$. Describe how to evaluate stochastic gradients of $\mathrm{VLB}_{\theta,\phi,\lambda}(X_i)$ using the reparameterization trick.

Clarification. For (b) and (c), we are looking for expressions (equations) that can be directly implemented in PyTorch, as were the expressions derived in class.