Mathematical Foundations of Deep Neural Networks, M1407.001200
E. Ryu
Fall 2022

## Homework 6 Solutios

**Problem 1:** *Dropout-ReLU=ReLU-Dropout.* Consider the following layer

```
class myLayer(nn.Module):
  def __init__(self, input_size, output_size):
    super(myLayer, self).__init__()
    self.linear = nn.Linear(input_size,output_size)
    self.sigma = nn.ReLU()
    # self.sigma = nn.Sigmoid()
    # self.sigma = nn.LeakyReLU()
    self.dropout= nn.Dropout(p=0.4)
  def forward(self, x):
    return dropout(sigma(linear))
    # return sigma(dropout(linear))  # Is this is equivalent?
```

In which of the three following cases are the operations linear-dropout-$\sigma$ and linear-$\sigma$-dropout equivalent?

(a) `self.sigma = nn.ReLU()`

(b) `self.sigma = nn.Sigmoid()`

(c) `self.sigma = nn.LeakyReLU()`

**Solution.** The ReLU and LeakyReLU are nonnegative homogeneous, i.e.,

$$\sigma(cx) = c\sigma(x)$$

for any $c \geq 0$ and $x \in \mathbb{R}$. Note that `Dropout(y)` can be expressed as `H*y` where `*` is elementwise multiplication and `H` is random 0-$(1/(1-p))$ mask. Since the elements of `H` are nonnegative, `sigma(H*y)==H*sigma(y)`. So dropout and $\sigma$ commute in cases (a) and (c).

For (b), consider a single-layer neural network whose input and output are both 1-dimensional with weight $a$ and zero bias. If $p = 0.4$ and the neuron is not dropped,

$$\texttt{dropout(sigma(linear))} = \frac{\sigma(ax)}{0.6} = \frac{1}{0.6(1+e^{-ax})}$$

and

$$\texttt{sigma(dropout(linear))} = \sigma\left(\frac{ax}{0.6}\right) = \frac{1}{1+e^{-ax/0.6}},$$

where $x \in \mathbb{R}$ is the input. The two are not equivalent for all $x$.

**Problem 2:** *Default weight initialization.* Consider the multi-layer perceptron

$$\begin{aligned} y_L &= A_L y_{L-1} + b_L \\ y_{L-1} &= \sigma(A_{L-1} y_{L-2} + b_{L-1}) \\ &\vdots \\ y_2 &= \sigma(A_2 y_1 + b_2) \\ y_1 &= \sigma(A_1 x + b_1), \end{aligned}$$

where $x \in \mathbb{R}^{n_0}$, $A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $b_\ell \in \mathbb{R}^{n_\ell}$, and $n_L = 1$. For the sake of simplicity, let

$$\sigma(z) = z.$$

Assume $x_1, \dots, x_{n_0}$ are IID with zero-mean and unit variance. If this network is initialized with the default weight initialization of PyTorch, what will the mean and variance of $y_L$ be?

*Clarification.* For this problem, you are being asked to read the PyTorch source code `https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html` to identify the default initialization behavior and then to perform calculations.

**Solution.** From default initialization of Pytorch, $A_\ell \sim \text{Uniform}\Big[-\frac{1}{\sqrt{n_\ell - 1}}, \frac{1}{\sqrt{n_\ell - 1}}\Big]$, $b_\ell \sim \text{Uniform}\Big[-\frac{1}{\sqrt{n_\ell - 1}}, \frac{1}{\sqrt{n_\ell - 1}}\Big]$. Also, $x \sim \mathcal{N}(0,1)$.
We first show that $\mathbb{E}[y_\ell] = 0$ for all $1 \le \ell \le L$.
The proof of the statement is as below. For base case $\ell = 1$, $\mathbb{E}[y_1] = \mathbb{E}[A_1 x + b_1] = \mathbb{E}[A_1]\mathbb{E}[x] + \mathbb{E}]b_1] = 0$. Assuming that $\mathbb{E}[y_\ell] = 0$ holds, $\mathbb{E}[y_{\ell+1}] = \mathbb{E}[A_\ell y_\ell + b_\ell] = \mathbb{E}[A_\ell]\mathbb{E}[y_\ell] + \mathbb{E}[b_\ell] = 0$. Therefore, by mathematical induction, $\mathbb{E}[y_L] = 0$.

For variance,

$$\begin{aligned}
\text{Var}(y_\ell) &= \text{Var}(y_{\ell i}) \\
&= \text{Var}\Big(\sum_{j=1}^{n_\ell - 1} A_{\ell i j} y_{(\ell-1)j}\Big) + \text{Var}(b_{\ell i}) \\
&= \sum_{j=1}^{n_\ell - 1} \text{Var}(A_{\ell i j} y_{(\ell-1)j}) + \text{Var}(b_{\ell i}) \\
&= \sum_{j=1}^{n_\ell - 1} \Big(\text{Var}(A_{\ell i j})\text{Var}(y_{(\ell-1)j}) + \text{Var}(A_{\ell i j})\mathbb{E}^2[y_{(\ell-1)j}] + \mathbb{E}^2[A_{\ell i j}]\text{Var}(y_{(\ell-1)j})\Big) + \text{Var}(b_{\ell i}) \\
&= \sum_{j=1}^{n_\ell - 1} \text{Var}(A_{\ell i j})\text{Var}(y_{(\ell-1)j}) + \text{Var}(b_{\ell i}) \\
&= n_{\ell-1} \times \frac{1}{3n_{\ell-1}} \text{Var}(y_{\ell-1}) + \frac{1}{3n_{\ell-1}} \\
&= \frac{1}{3}\text{Var}(y_{\ell-1}) + \frac{1}{3n_{\ell-1}}.
\end{aligned}$$

Since $y_L$ is a scalar, by plugging in $\text{Var}(y_0) = \text{Var}(x) = 1$,

$$\text{Var}(y_L) = \frac{1}{3^L} + \sum_{k=0}^{L-1} \frac{1}{3^{L-K} n_k}.$$

This completes the solution. ■

**Problem 3:** *Backprop for MLP with residual connections.* Let $\sigma\colon \mathbb{R} \to \mathbb{R}$ be a differentiable activation function and consider the following MLP with residual connections

$$\begin{aligned} y_L &= A_L y_{L-1} + b_L \\ y_{L-1} &= \sigma(A_{L-1} y_{L-2} + b_{L-1}) + y_{L-2} \\ &\vdots \\ y_3 &= \sigma(A_3 y_2 + b_3) + y_2 \\ y_2 &= \sigma(A_2 y_1 + b_2) + y_1 \\ y_1 &= \sigma(A_1 x + b_1), \end{aligned}$$

where $x \in \mathbb{R}^n$, $A_1 \in \mathbb{R}^{m\times n}$, $b_1 \in \mathbb{R}^m$, $A_\ell \in \mathbb{R}^{m\times m}$, $b_\ell \in \mathbb{R}^m$ for $\ell = 2, \dots, L-1$, and $A_L \in \mathbb{R}^{1\times m}$, $b_L \in \mathbb{R}^1$. (To clarify, $\sigma$ is applied element-wise.) For notational convenience, define $y_0 = x$.

(i) Find formulae for

$$\frac{\partial y_\ell}{\partial y_{\ell-1}}$$

for $\ell = 2, \dots, L$.

(ii) Find formulae for

$$\frac{\partial y_L}{\partial b_\ell}, \qquad \frac{\partial y_L}{\partial A_\ell}$$

for $\ell = 1, \dots, L$.

(iii) The gradients

$$\frac{\partial y_L}{\partial b_i}, \quad \frac{\partial y_L}{\partial A_i}$$

for $i = 1, \dots, \ell$ need not vanish when $[A_j = 0$ for some $j \in \{\ell+1, \dots, L-1\}]$ or $[\sigma'(A_j y_{j-1} + b_j) = 0$ for some $j \in \{\ell+1, \dots, L-1\}]$. Explain why.

**Solution.**

(i) First, in the case $\ell = L$ we have $\frac{\partial y_L}{\partial y_{L-1}} = A_L$.
Next, we have

$$y_\ell = \sigma(A_\ell y_{\ell-1} + b_\ell) + y_{\ell-1}$$

Note that $i$th component of $y_\ell$ is

$$(y_\ell)_i = \sigma((A_\ell y_{\ell-1} + b_\ell)_i) + (y_{\ell-1})_i$$

since $\sigma$ is applied element-wise. Thus

$$\left(\frac{\partial y_\ell}{\partial y_{\ell-1}}\right)_{ij} = \begin{cases} \sigma'((A_\ell y_{\ell-1} + b_\ell)_i)\,(A_\ell)_{ij} + 1 & \text{if } j = i \\ \sigma'((A_\ell y_{\ell-1} + b_\ell)_i)\,(A_\ell)_{ij} & \text{otherwise.} \end{cases}$$

We vectorize this result into

$$\frac{\partial y_\ell}{\partial y_{\ell-1}} = \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right) A_\ell + I$$

for $\ell = 2, \dots, L-1$.

(ii) By the same calculations as in the prior homework assignment, we have

$$\frac{\partial y_L}{\partial b_L} = 1, \qquad \frac{\partial y_L}{\partial A_L} = y_{L-1}^{\intercal}.$$

For $\ell = 1\dots, L-1$, using the chain rule,

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

Since

$$\frac{\partial y_\ell}{\partial b_\ell} = \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right),$$

we have

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

Using the chain rule likewise,

$$\frac{\partial y_L}{(\partial A_\ell)_{ij}} = \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{(\partial A_\ell)_{ij}}$$

Since

$$\frac{\partial y_\ell}{\partial\left(A_\ell\right)_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ \sigma'((A_\ell y_{\ell-1} + b_\ell)_i)\left(y_{\ell-1}\right)_j \\ \vdots \\ 0 \end{bmatrix},$$

we have

$$\left(\frac{\partial y_L}{\partial A_\ell}\right)_{ij} = \frac{\partial y_L}{\partial\left(A_\ell\right)_{ij}} = \left(\frac{\partial y_L}{\partial y_\ell}\right)_i \sigma'((A_\ell y_{\ell-1} + b_\ell)_i)\left(y_{\ell-1}\right)_j.$$

Vectorizing this result gives us

$$\frac{\partial y_L}{\partial A_\ell} = \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right)\left(\frac{\partial y_L}{\partial y_\ell}\right)^{\intercal} y_{\ell-1}^{\intercal}.$$

(iii) The vanishing gradient problem occurs when the matrix product

$$\frac{\partial y_L}{\partial y_\ell} = \frac{\partial y_L}{\partial y_{L-1}}\frac{\partial y_{L-1}}{\partial y_{L-2}}\cdots\frac{\partial y_{\ell+1}}{\partial y_\ell}, \qquad \text{for } \ell = 1\dots, L.$$

vanishes, but the identity matrix in $\frac{\partial y_\ell}{\partial y_{\ell-1}}$ prevents this. It, However, is still possible that

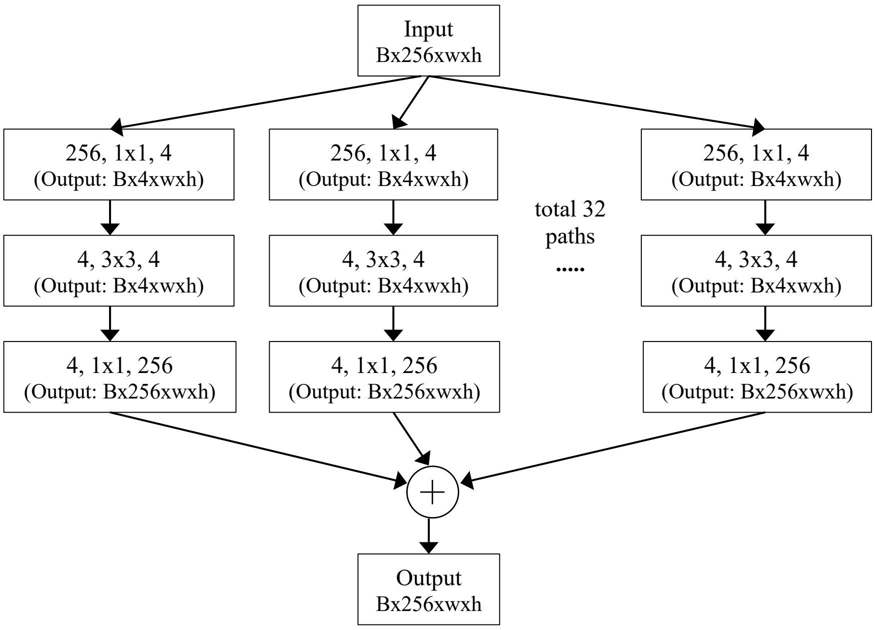
$$\frac{\partial y_L}{\partial b_\ell}, \qquad \frac{\partial y_L}{\partial A_\ell}$$

are small if $\operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right)$ is small for $\ell = 1, \dots, L-1$.

**Problem 4:** *Split-transform-merge convolutions.* Consider a series of $1 \times 1$, $3 \times 3$, $1 \times 1$ conv-ReLU operations with 256–128–128–256 channels:

```
class MyConvLayer(nn.Module):
  def __init__(self):
    super(MyConvLayer, self).__init__()
    self.conv1 = nn.Conv2d(256, 128, 1,)
    self.conv2 = nn.Conv2d(128, 128, 3, padding=1)
    self.conv3 = nn.Conv2d(128, 256, 1)
  def forward(self, x):
    out = torch.nn.functional.relu(self.conv1(x))
    out = torch.nn.functional.relu(self.conv2(out))
    out = torch.nn.functional.relu(self.conv3(out))
    return out
```

An issue with this construction, however, is that it has too many trainable parameters. To reduce the number of trainable parameters, we use the following *split-transform-merge* structure: [apply a series of $1 \times 1$, $3 \times 3$, $1 \times 1$ conv-ReLU operations with 256–4–4–256 channels] a total of 32 times and sum the 32 outputs. The following figure illustrates this construction.



To clarify, all convolutions use biases and the strides are all equal to 1. ReLU is not applied after the sum operation.

(a) How many trainable parameters are present in both constructions?

(b) In the following page, implement this convolution with the split-transform-merge structure.

```
class STMConvLayer(nn.Module):
  def __init__(self):
    super(STMConvLayer, self).__init__()
    #-----------------------------------------
    # Fill in code here



    #-----------------------------------------
  def forward(self, x):
    # [apply 1x1conv with 4 output channels
    #  apply 3x3conv with 4 output channels (with padding=1)
    #  apply 1x1conv with 256 output channels] X 32
    # Add all 32 outputs
    #-----------------------------------------
    # Fill in code here



    #-----------------------------------------

    return out
```

**Solution.** For MyConvLayer,

$$
\begin{aligned}
\#\text{weight of conv1} &= 256 * 128 = 32768 \\
\#\text{bias of conv1} &= 128 \\
\#\text{weight of conv2} &= 128 * 128 * 9 = 147456 \\
\#\text{bias of conv2} &= 128 \\
\#\text{weight of conv3} &= 128 * 256 = 32768 \\
\#\text{bias of conv3} &= 256
\end{aligned}
$$

Hence the total number of trainable parameters of MyConvLayer is 213504.

$$
\begin{aligned}
\#\text{weight of layer1} &= 256 * 4 * 32 = 32768 \\
\#\text{bias of layer1} &= 4 * 32 = 128 \\
\#\text{weight of layer2} &= 4 * 4 * 9 * 32 = 4608 \\
\#\text{bias of layer2} &= 4 * 32 = 128 \\
\#\text{weight of layer3} &= 4 * 256 * 32 = 32768 \\
\#\text{bias of layer3} &= 256 * 32 = 8192.
\end{aligned}
$$

Hence the total number of trainable parameters of STMConv is 78592.
(The ratio is 213054/78592 = 2.71.)

```
class STMConv(nn.Module):
  def __init__(self):
    super(STMConv, self).__init__()
    self.layer1 = nn.ModuleList([
      nn.Conv2d(256, 4, kernel_size=1, stride=1)
      for i in range(32)
    ])
    self.layer2 = nn.ModuleList([
      nn.Conv2d(4, 4, kernel_size=3, stride=1, padding=1)
      for i in range(32)
    ])
    self.layer3 = nn.ModuleList([
      nn.Conv2d(4, 256, kernel_size=1, stride=1)
      for i in range(32)
    ])
  def forward(self, x):
    out = []
    for i in range(32):
      tmp = torch.nn.functional.relu(self.layer1[i](x))
      tmp = torch.nn.functional.relu(self.layer2[i](tmp))
      tmp = torch.nn.functional.relu(self.layer3[i](tmp))
      out = out.append(tmp)
    return sum(out)
```

**Problem 5:** *Regularization can mitigate double descent.* Assume we have labels $Y_1, \dots, Y_{N_{\text{train}}} \in \mathbb{R}$ generated IID as $X_i \sim \mathcal{N}(0, I_d)$ and $Y_i \sim X_i^\intercal \beta^\star + \mathcal{N}(0, \sigma^2)$ for $i = 1, \dots, N_{\text{train}}$, where $\beta^\star \in \mathbb{R}^d$. Use $d = 35$ and $\sigma = 0.5$, and $N_{\text{train}} = 300$. Fit the data with a 2-layer ReLU network $f_{\theta,W}(x) = \theta^\intercal \text{ReLU}(Wx)$ with $\theta \in \mathbb{R}^p$ and $W \in \mathbb{R}^{p \times d}$. Assume $W_{ij} \sim \mathcal{N}(0, 1/p)$ IID. For simplicity, assume $W$ is fixed (not trained) once initialized. Train $\theta$ via

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad \sum_{i=1}^{N_{\text{train}}} \frac{1}{2}(f_{\theta,W}(X_i) - Y_i)^2 + \frac{\lambda}{2}\|\theta\|^2$$

with $\lambda > 0$. Using the notation $\tilde{X}_i = \text{ReLU}(WX_i)$ for $i = 1, \dots, N_{\text{train}}$ and

$$\tilde{X} = \begin{bmatrix} \tilde{X}_1^\intercal \\ \vdots \\ \tilde{X}_{N_{\text{train}}}^\intercal \end{bmatrix} \in \mathbb{R}^{N_{\text{train}} \times p}, \qquad Y = \begin{bmatrix} Y_1 \\ \vdots \\ Y_{N_{\text{train}}} \end{bmatrix} \in \mathbb{R}^{N_{\text{train}}},$$

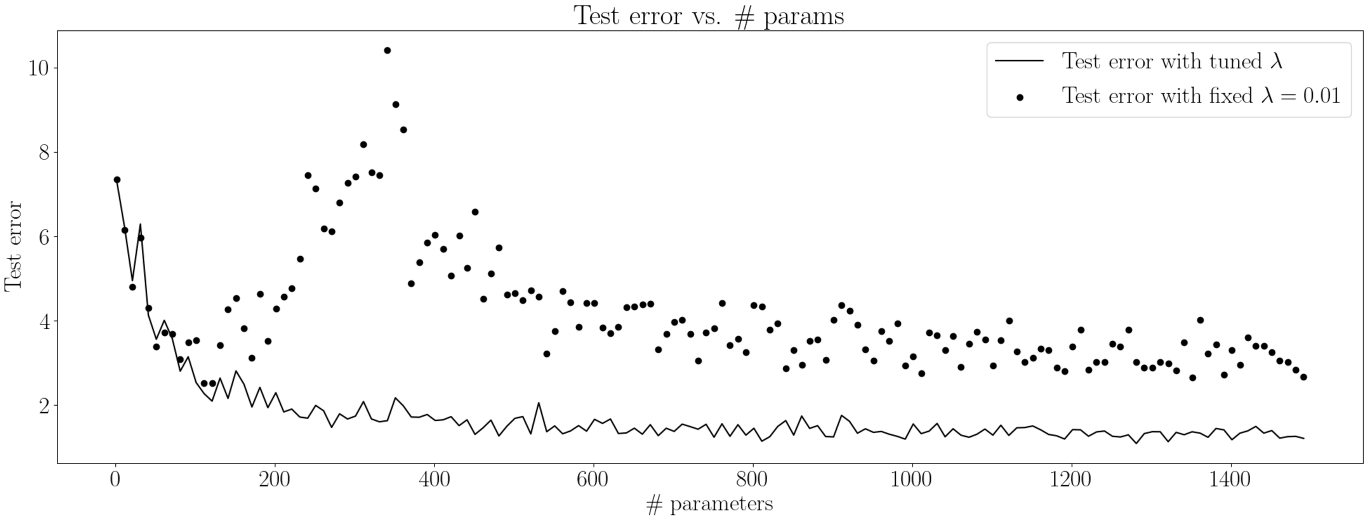
we can equivalently express the optimization problem as

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \quad \frac{1}{2}\|\tilde{X}\theta - Y\|^2 + \frac{\lambda}{2}\|\theta\|^2.$$

Train (compute the global minimum) by using linear algebra to solve the least-squares problem. With the fixed regularization parameter $\lambda = 0.01$, we indeed observe the double descent phenomenon when we plot the test error against the number of parameters $p$. Show that the double descent phenomenon vanishes if $\lambda$ is tuned. Specifically, use the training dataset to (precisely) compute $\theta$ and use the validation dataset of size $N_{\text{validation}} = 60$ to (roughly) tune for $\lambda \in [10^{-2}, 10^2]$. (You should separately tune $\lambda$ for each $p$, as you would do in practice.) Then, use the test dataset of size $N_{\text{test}} = 30$ to plot the test error for each $p$ and its corresponding optimal $\lambda$. Use the starter code `ddescent.py`.

*Remark.* This problem was inspired by [1].

*Hint.* The results should look something like:



**Solution.** See `ddescent_sol.py`. ■

## References

[1] P. Nakkiran, P. Venkat, S. Kakade, and T. Ma. Optimal regularization can mitigate double descent, *ICLR*, 2021.