Guolun Li

1.1 $h_i = \phi(2(x_4 - x_i)) = \begin{cases} 1, & \text{if } x_4 = x_i \\ 0, & \text{otherwise} \end{cases}$

$y = \phi(2(1 - \Sigma h_i)) = \begin{cases} 1, & \text{if } h_i = 1 \text{ for some } i \in \{1,2,3\} \\ 0, & \text{otherwise} \end{cases}$

Thus, $W^{(1)}\vec{x} + b^{(1)} = 2\begin{pmatrix} x_4 - x_1 \\ x_4 - x_2 \\ x_4 - x_3 \end{pmatrix}$

$\Rightarrow W^{(1)} = 2\begin{pmatrix} -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{pmatrix}, \; b^{(1)} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

and $W^{(2)}\vec{h} + b^{(2)} = 2(1 - \Sigma h_i)$

$\Rightarrow W^{(2)} = -2(1 \;\; 1 \;\; 1), \; b^{(2)} = -2$

1.2 We use a brute force approach (list out all permutations) to achieve the goal.

We'll construct two hidden layers and one output layer.

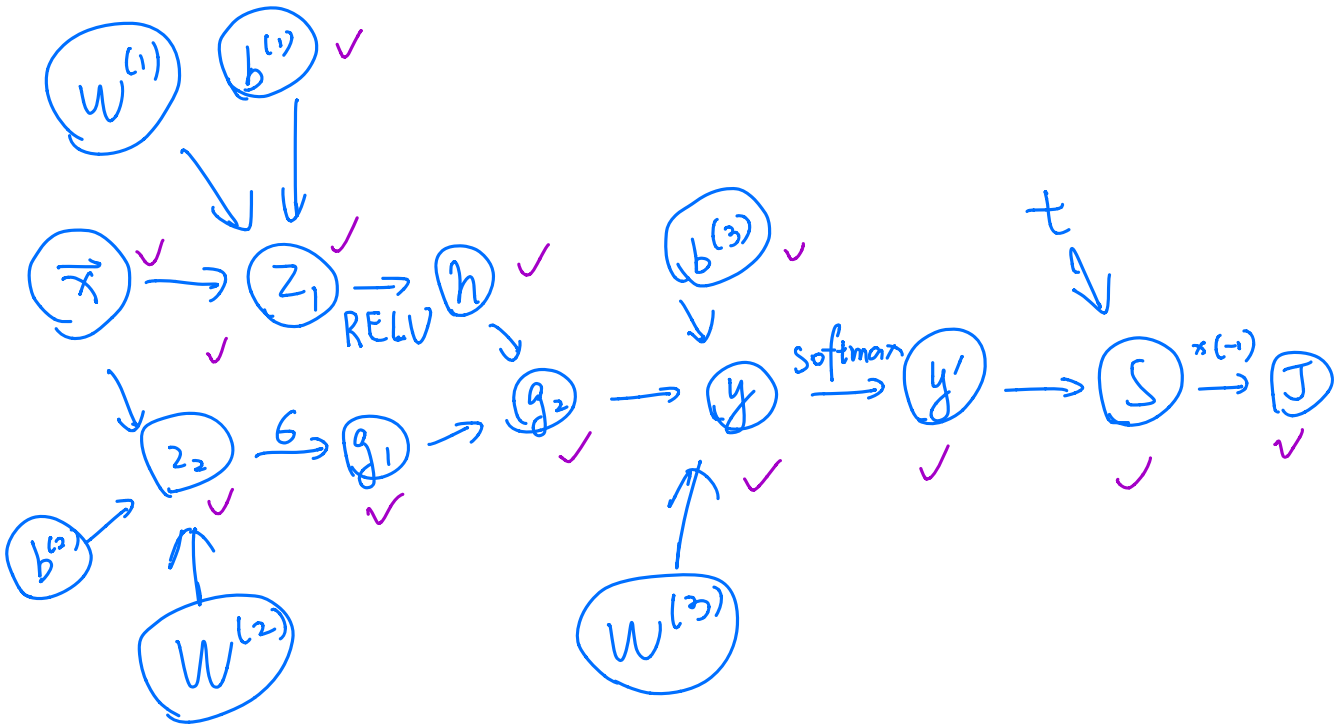
The first hidden layer compares, one by one, the first three nodes to the last three nodes. The second hidden layer enumerates all possibilites that the last three nodes are a permutation of the first three nodes.

The first hidden layer has 9 nodes, each node checks if $x_i = x_j$, for some $i \in \{1,2,3\}$, $j \in \{4,5,6\}$. This is similar to Q1.1, and the activation function would be $\phi_1(z) = I(z \in [-1, 1])$.

The second hidden layer has 6 nodes,
each node checks if $(x_1, x_2, x_3) = (x_i, x_j, x_k)$, where (x_i, x_j, x_k) is a permutation of (x_4, x_5, x_6). This can be done by putting weight 1 on the nodes that check $x_1 = x_i$, $x_2 = x_j$, $x_3 = x_k$, and weight 0 on other nodes, then using an activation function $\phi_2(z) = I(z = 3)$.
Then, the output layer checks if any node in the second hidden layer is 1. This is also similar to Q1.1, we simply put weight 1 on each node and use $\phi(z) = I(z = 1)$.

2.1.1 Computational Graph [0pt]

Draw the computation graph relating $\mathbf{x}$, t, $\mathbf{z}_1$, $\mathbf{h}$, $\mathbf{z}_2$, $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{y}$, $\mathbf{y}'$, $\mathcal{S}$ and $\mathcal{J}$.



2.1.2 Backward Pass [1pt]

Derive the backprop equations for computing $\bar{\mathbf{x}} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}}$, one variable at a time, similar to the vectorized backward pass derived in Lec 2.

$\bar{J} = 1$

$\bar{S} = -\bar{J}$

$\bar{y'} = \bar{S} \nabla_{y'} S = -\bar{J} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \frac{1}{y'_t} \\ 0 \\ \vdots \end{pmatrix}$ t^{th} position, since $\frac{\partial S}{\partial y'_k} = \frac{\partial}{\partial y'_k}(\log y'_t) = \begin{cases} \frac{1}{y'_t}, & \text{if } k=t \\ 0, & \text{o.w.} \end{cases}$

$\overline{y} = \left(\frac{\partial y'}{\partial y}\right)^T \overline{y'}$, where $\left(\frac{\partial y'}{\partial y}\right)_{i,j} = I(i=j)\,\text{softmax}(y_i) - \text{softmax}(y_i)\cdot\text{softmax}(y_j)$ (see lemma 1)

$\overline{g_2} = (W^{(2)})^T \overline{y}$,

$\overline{h} = \overline{g_2} \odot g_1$,

$\overline{g_1} = \overline{g_2} \odot h$,

$\overline{z_2} = \sigma'(z_2) \odot \overline{g_1}$, where $\sigma'(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2}$

$\overline{z_1} = \text{RELU}'(z_1) \odot \overline{h}$, where $\text{RELU}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$

(Define $\text{RELU}'(0) = 1$ for computation purpose)

$\overline{x} = W^{(1)T}\overline{z_1} + W^{(2)T}\overline{z_2}$

Lemma 1:

$$\left(\frac{\partial y'}{\partial y}\right)_{i,j} = \frac{\partial y_i'}{\partial y_j} = \frac{\partial}{\partial y_j}\left(\frac{e^{y_i}}{\sum e^{y_k}}\right)$$

$$= \frac{\partial_{y_j} e^{y_i} \sum e_k - e^{y_j}\cdot e^{y_i}}{(\sum e^{y_k})^2}$$

$$= \frac{1}{(\sum e^{y_k})^2}\left[I(i=j)\, e^{y_i}\cdot \sum e^{y_k} - e^{y_i + y_j}\right]$$

$$= I(i=j)\,\text{softmax}(y_i) - \text{softmax}(y_i)\cdot\text{softmax}(y_j)$$

2.2.2 Computation Cost [1pt]

What is the number of scalar multiplications and memory cost of computing the Hessian $\mathbf{H}$ in terms of n?

2.2.2

$L = x^T v v^T x$

$\nabla L = v v^T x$

$H = \nabla^2 L = 2 v v^T$ which requires $2n^2$ multiplications and

$n^2 + n$ memory cost

$\Downarrow$

$O(n^2)$

$\Downarrow$

$O(n^2)$

2.3 Vector-Hessian Products [1pt]

Compute $\mathbf{z} = \mathbf{H}\mathbf{y} = \mathbf{v}\mathbf{v}^\top\mathbf{y}$ where $n = 3$, $\mathbf{v}^\top = [1, 2, 3]$, $\mathbf{y}^\top = [1, 1, 1]$ using two algorithms: reverse-mode and forward-mode autodiff.

In backpropagation (also known as reverse-mode autodiff), you will compute $\mathbf{M} = \mathbf{v}^\top\mathbf{y}$ first, then compute $\mathbf{v}\mathbf{M}$. Whereas, in forward-mode, you will compute $\mathbf{H} = \mathbf{v}\mathbf{v}^\top$ then compute $\mathbf{H}\mathbf{y}$.

Write down the numerical values of $\mathbf{z}^T = [z_1, z_2, z_3]$ for the given $\mathbf{v}$ and $\mathbf{y}$. What is the time and memory cost of evaluating $\mathbf{z}$ with backpropagation (reverse-mode) in terms of n? What about forward-mode?

$n\times 1 \quad 1\times n \quad n\times 1$

For Q 2.3 and Q 2.4, we use the lemma that computing the matrix product MN, where M is $a\times b$, N is $b\times c$, takes in total abc scalar multiplications. This is easy to see, as each entry of MN is the dot product of one row from M and one column from N (takes b steps), and there are ac such entries.

$z^T = (6 \quad 12 \quad 18)$

Back Prop:

Time: $M = v^T y$, $z = vM$

Total time $= n + n = 2n$

Memory: y, v, M, z

Total Memory $= n + n + 1 + n = 3n+1$

Forward:

Time: $H = vv^T$, $z = Hy$

Total time $= n^2 + n^2 = 2n^2$

Memory: v, H, y, z

Total Memory $= n + n^2 + n + n = n^2 + 3n$

$n\times 1\times n \quad n\times 1\times 1\times m$

2.4 Trade-off of Reverse- and Forward-mode Autodiff [1pt] $Z = vv^T y_1 y_2^T$ $H = vv^T$

Consider computing $\mathbf{Z} = \mathbf{H}\mathbf{y}_1\mathbf{y}_2^\top$ where $\mathbf{v} \in \mathbb{R}^{n\times 1}, \mathbf{y}_1 \in \mathbb{R}^{n\times 1}$ and $\mathbf{y}_2 \in \mathbb{R}^{m\times 1}$. What are the time and memory cost of evaluating $\mathbf{Z}$ with reverse-mode in terms of n and m? What about forward-mode? When is forward-mode a better choice? (Hint: Think about the shape of Z, "tall" v.s. "wide".)

Back Prop:

Time: $B_1 = y_1 y_2^T$, $B_2 = v^T B_1$, $Z = v B_2$

Total time = $nm + nm + nm = 3mn$

Memory: y_1, y_2, B_1, v, B_2, Z

Total Memory = $n + m + nm + n + m + nm = 2(m+n+mn)$

Forward:

Time: $A_1 = vv^T$, $A_2 = A_1 y_1$, $Z = A_2 y_2^T$

Total time = $n^2 + n^2 + nm = n(2n+m)$

Memory: v, A_1, y_1, A_2, y_2, Z

Total Memory = $n + n^2 + n + n + m + mn = 3n + m + n^2 + mn$

$\text{BackProp}_{time} - \text{Forward}_{time} = n(2m - 2n)$

$\text{BackProp}_{memory} - \text{Forward}_{memory} = (n+1)(m-n)$

Thus, given the dimension of Z being $n \times m$, using back propagation is better when $m < n$, using forward mode is better when $m > n$.

3.2 Underparameterized Model [1pt]

First consider the underparameterized $d < n$ case. Write down the solution obtained by gradient descent assuming training converges. Show your work. Is the solution unique?

$L = \frac{1}{n}|A|^2$, $A = X\hat{w} - t$.

$\Rightarrow \overline{A} = \frac{2}{n}A$

$\Rightarrow \overline{\hat{w}} = X^T\overline{A} = X^T \cdot \frac{2}{n}A = \frac{2}{n}X^T(X\hat{w} - t)$

Gradient Descent will result in $\overline{\hat{w}} = 0 \Rightarrow X^TX\hat{w} = X^Tt$

$\Rightarrow \hat{w}^* = (X^TX)^{-1}X^Tt$.

Since $d < n$, then X^TX is invertible, so $\hat{w}^*$ has an explicit formula so it is unique.

3.3.2 [1pt]

Now, let's generalize the previous 2D case to the general $d > n$. Show that gradient descent from zero initialization i.e. $\hat{\mathbf{w}}(0) = 0$ finds a unique minimizer if it converges. Write down the solution and show your work.

See next page.

Solution: $\hat{w} = X^T(XX^T)^{-1}t$

Proof:

$L = \frac{1}{n}|A|^2$, $A = X\hat{w} - t$.

$\Rightarrow \overline{A} = \frac{2}{n}A \Rightarrow \overline{\hat{w}} = X^T\overline{A} = X^T \cdot \frac{2}{n}A = \frac{2}{n}X^T(X\hat{w} - t)$ i.e. gradient

Thus, $\hat{w}_{k+1} = \hat{w}_k - \eta \cdot \frac{2}{n}X^T(X\hat{w}_k - t)$ i.e. gradient descent where $\eta > 0$ is learning rate

We'll show two things:

1) Given $\hat{w}_0 = \vec{0}$, $\forall k \in \mathbb{N}$, $\hat{w}_k = X^T c_k$ for some $c_k \in \mathbb{R}^d$.

2) Given that gradient descent converges, there will be $X\hat{w} - t = 0$

We'll show 1) first by induction:

- Obviously $\hat{w}_0 = X^T\vec{0}$ satisfies the requirement
- Assume that for $k \in \mathbb{N}$, $\hat{w}_k = X^T c_k$ for some $c_k \in \mathbb{R}^n$.

Then $\hat{w}_{k+1} = X^T c_k - \eta \cdot \frac{2}{n}X^T(X\hat{w}_k - t)$

$= X^T\underbrace{\left(c_k - \eta \cdot \frac{2}{n}X\hat{w}_k + t\right)}_{= c_{k+1} \in \mathbb{R}^n}$

So 1) has been shown.

For 2), since gradient descent converges, then

$\lim_{k\to\infty} \hat{w}_{k+1} = \lim_{k\to\infty} \hat{w}_k = \hat{w}$, where $\hat{w}$ is the solution.

Thus, $\lim_{k\to\infty} \hat{w}_{k+1} = \lim_{k\to\infty} \hat{w}_k - \eta \cdot \frac{2}{n} X^T (X \lim_{k\to\infty} \hat{w}_k - t)$

$$\Rightarrow \hat{w} = \hat{w} - \eta \cdot \frac{2}{n} X^T (X\hat{w} - t)$$

$$\Rightarrow X^T (X\hat{w} - t) = 0$$

$$\Rightarrow X X^T (X\hat{w} - t) = 0$$

In the case of $d > n$, XX^T is $n \times n$ thus invertible.

$$\Rightarrow (XX^T)^{-1}(XX^T)(X\hat{w} - t) = (XX^T)^{-1}\vec{0}$$

$$\Rightarrow X\hat{w} - t = 0$$

Combining 1) and 2): Let $\hat{w} = X^T c$, where $\hat{w} = \lim_{k\to\infty} \hat{w}_k$ and $c = \lim_{k\to\infty} c_k$, then:

$$X(X^T c) - t = 0$$

$$\Rightarrow XX^T c = t$$

$$\Rightarrow c = (XX^T)^{-1} t$$

$$\Rightarrow \hat{w} = X^T (XX^T)^{-1} t \text{ is unique}$$

3.3.3 [1pt]

Visualize and compare underparameterized with overparameterized polynomial regression: `https://colab.research.google.com/drive/1Atkk9hjUaXV-bDttCxAv9WiMsB6EJTKU`. Include your code snippets for the `fit_poly` function in the write-up. Does overparameterization (higher degree polynomial) always lead to overfitting, i.e. larger test error?

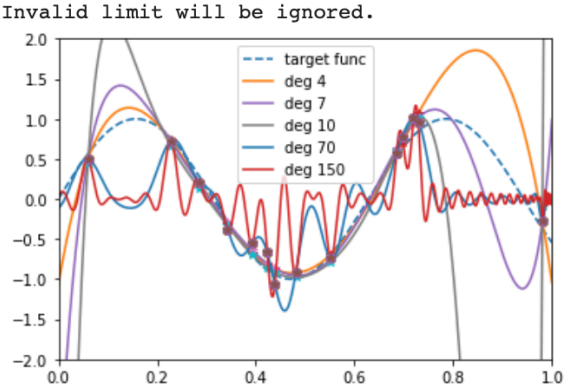
Function Fill In for fit_poly():

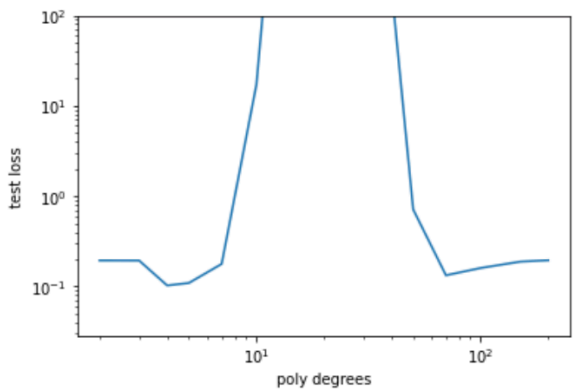
```
def fit_poly(X, d,t):
  X_expand = poly_expand(X, d=d, poly_type = poly_type)
  n = X.shape[0]
  if d > n:
    W = X_expand.T.dot(np.linalg.inv(X_expand.dot(X_expand.T))).dot(t)
  else:
    W = np.linalg.inv(X_expand.T.dot(X_expand)).dot(X_expand.T).dot(t)
  return W
```

Loss Values Output:

```
poly_type = 'chebyshev' # try legendre or chebyshev
loss_val_list = []

poly_degrees = [2, 3, 4, 5, 7, 10, 15, 20,30,50,70,100,150,200]
plot_poly_degress = [4, 7, 10, 70, 150] ## only plot these polynomials
##
plot_target_func()

for d in poly_degrees:
  W = fit_poly(X, d,t)
  plot_flag = True if d in plot_poly_degress else False
  loss_val = plot_prediction(X, W, d, domain, plot_flag)
  loss_val_list.append(loss_val)
plt.legend()

plot_val_loss(poly_degrees, loss_val_list)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:43: UserWarning: Attempted to set no
Invalid limit will be ignored.
```





```
[7]  for i in range(len(poly_degrees)):
       print(f"degree = {poly_degrees[i]}, loss = {loss_val_list[i]}")
```

```
degree = 2, loss = 0.19386411705081336
degree = 3, loss = 0.193432748835684
degree = 4, loss = 0.10255222991025262
degree = 5, loss = 0.10915309162008865
degree = 7, loss = 0.177356881829107
degree = 10, loss = 17.094862480309498
degree = 15, loss = 1323020.060699293
degree = 20, loss = 15596274997.211277
degree = 30, loss = 317604.6227527557
degree = 50, loss = 0.7120003515525691
degree = 70, loss = 0.13281043237871068
degree = 100, loss = 0.1599430277105052
degree = 150, loss = 0.18800282052817618
degree = 200, loss = 0.19468029146824442
```

As can be seen, the loss function increases with polynomial degree first, then decreases back to the original level. By comparing degree 7 and degree 70, we see that degree 70 has a smaller loss. This shows that overparametrization ($n=14$, $d=71$) doesn't necessary cause overfitting.