hw4

October 7, 2021

1 Mathematical Foundations of Deep Neural Network

1.1 Homework 4

1.1.1 2017-11362

1.2 Problem 1: *Finite difference with convolution.*

$$Y_{1,i,j} = X_{i+1,j} - X_{i,j} = w_1 * \begin{bmatrix} x_{i-1,j-1} & x_{i-1,j} & x_{i-1,j+1} \\ x_{i,j-1} & x_{i,j} & x_{i,j+1} \\ x_{i+1,j-1} & x_{i+1,j} & x_{i+1,j+1} \end{bmatrix}$$

(shifted index cause of zero padding of 1)

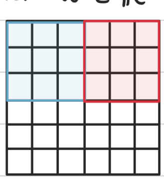
$$= \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{\rightarrow\, w_1} * X[i-1:i+1,\ j-1:j+1]$$

$$Y_{2,i,j} = X_{i,j+1} - X_{i,j} = \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{\rightarrow\, w_2} * X[i-1:i+1,\ j-1:j+1]$$

$$w = [w_1, w_2]$$

1.3 Problem 2: *Average pooling as convolution.*

Filter $w \in \mathbb{R}^{C\times k\times k}$, $w = (w_1, \cdots, w_C)^t$ where $w_h \in \mathbb{R}^{k\times k}$ for $h = 1, \cdots, C$.



$$\rightarrow w_h = \frac{1}{k^2}\underbrace{\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}}_{k} \Bigg\} k, \quad h = 1, \cdots, C.$$

$\therefore$ AvgPool2d is a convolution with a $\overset{w}{\uparrow}$ $C\times k\times k$ filter with stride k and no zero padding.

1.4 Problem 3: *RGB to grayscale mapping with* 1×1 *convolution.*

$W = [0.299, \ 0.587, \ 0.114]$

$\Rightarrow X \mapsto Y$ is a convolution with a filter W.

1.5 Problem 4: *Commutation.*

$\sigma : \mathbb{R} \to \mathbb{R}$: non-decreasing, $\rho : \mathbb{R}^{m \times n} \to \mathbb{R}^{k \times \ell}$: max pooling

For simplicity, assume m and n are divisible by k and ℓ, respectively.

We only have to show that $\sigma(\rho_{11}(X_{11})) = \rho_{11}(\sigma(X_{11}))$ where

$\rho_{11} : \mathbb{R}^{(m/k) \times (n/\ell)} \to \mathbb{R}$, $X_{11} = X[0 : m/k, \ 0 : n/\ell]$

Let $X_{ij} = \rho(X_{11}) = \max\{X_{11}\}$. $(*)$

Assume that $\exists\, i', j'$ s.t $\sigma(X_{i'j'}) = \rho(\sigma(X_{11}))$. i.e, $\sigma(X_{i'j'}) > \sigma(X_{ij})$. $(**)$

Since σ is non-decreasing, $X_{i'j'} < X_{ij}$ $(*)$ $\Rightarrow \sigma(X_{i'j'}) \leq \sigma(X_{ij})$, which contradicts to $(**)$.

$\therefore \sigma(\rho(X)) = \rho(\sigma(X))$.

1.6 Problem 5: *Non-CE loss function.*

[1]:
```
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader

# torchvision: popular datasets, model architectures, and common image␣
↪transformations for computer vision.
from torchvision import datasets
from torchvision.transforms import transforms

from random import randint
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt


'''
Step 1: Prepare dataset
'''
```

```python
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True,␣
 ↪transform=transforms.ToTensor(), download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False,␣
 ↪transform=transforms.ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1

'''
Step 2: Define the neural network class
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=True)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))
```

```
/home/zendo/anaconda3/lib/python3.8/site-
packages/torchvision/datasets/mnist.py:498: UserWarning: The given NumPy array
is not writeable, and PyTorch does not support non-writeable tensors. This means
you can write to the underlying (supposedly non-writeable) NumPy array using the
tensor. You may want to copy the array to protect its data or make it writeable
```

```
before converting it to a tensor. This type of warning will be suppressed for
the rest of this program. (Triggered internally at
../torch/csrc/utils/tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

[2]:
```
'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR()                                     # Define a Neural Network Model

def loss_function(z, y):                                 # Specify loss function␣
 ↪= nn.MSELoss()
    ans = (1 - y) *((1 - torch.sigmoid(-z))**2 + torch.sigmoid(z)**2)
    ans += (1 + y) *((1 - torch.sigmoid(z))**2 + torch.sigmoid(-z)**2)
    return ans

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)   # specify SGD with␣
 ↪learning rate

'''
Step 4: Train model with SGD
'''
import time
start = time.time()
for _ in range(10000) :
    # Sample a random data for training
    ind = randint(0, len(train_set.data)-1)
    image, label = train_set.data[ind], train_set.targets[ind]

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()

    #(This syntax will make more sense once we learn about minibatches)

    # perform SGD step (parameter update)
    optimizer.step()

end = time.time()
print(f"Time ellapsed in training is: {end-start}")

'''
Step 5: Test model (Evaluate the accuracy)
'''
```

```
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Evaluate accuracy using test data
for ind in range(len(test_set.data)) :

    image, label = test_set.data[ind], test_set.targets[ind]

    # evaluate model
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_set.data), correct, len(test_set.data),
        100. * correct / len(test_set.data)))
```

```
Time ellapsed in training is: 2.7893874645233154
[Test set] Average loss: 0.8735, Accuracy: 1554/1991 (78.05%)
```

[3]:

```
'''
Step 3-1: Create the model, specify loss function and optimizer.
'''
model = LR()                                        # Define a Neural Network Model

def logistic_loss(output, target):
    return -torch.nn.functional.logsigmoid(target*output)

loss_function = logistic_loss                                                          ␣
 ↪# Specify loss function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)    # specify SGD with␣
 ↪learning rate

'''
Step 4: Train model with SGD
'''
```

```python
import time
start = time.time()
for _ in range(1000) :
    # Sample a random data for training
    ind = randint(0, len(train_set.data)-1)
    image, label = train_set.data[ind], train_set.targets[ind]

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()

    #(This syntax will make more sense once we learn about minibatches)

    # perform SGD step (parameter update)
    optimizer.step()

end = time.time()
print(f"Time ellapsed in training is: {end-start}")

'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Evaluate accuracy using test data
for ind in range(len(test_set.data)) :

    image, label = test_set.data[ind], test_set.targets[ind]

    # evaluate model
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]
```

```python
# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_set.data), correct, len(test_set.data),
        100. * correct / len(test_set.data)))
```

```
Time ellapsed in training is: 0.1475691795349121
[Test set] Average loss: 11.4633, Accuracy: 1816/1991 (91.21%)
```

step 3 means MSELoss, and step 3-1 means CE loss.

minimizing KL divergence is better than minimizing MSE.

1.7 Problem 6: *Backprop for MLP.*

(a) $y_L = \begin{bmatrix} A_{L,1,1} & \cdots & A_{L,1,n_{L-1}} \\ \vdots & \ddots & \vdots \\ A_{L,n_L,1} & \cdots & A_{L,n_L,n_{L-1}} \end{bmatrix} \begin{bmatrix} y_{L-1,1} \\ \vdots \\ y_{L-1,n_{L-1}} \end{bmatrix} + \begin{bmatrix} b_{L,1} \\ \vdots \\ b_{L,n_L} \end{bmatrix}$

$y_{L,i} = \sum_{j=1}^{n_{L-1}} A_{L,i,j} \cdot y_{L-1,j} + b_{L,i}$

$\left(\frac{\partial y_L}{\partial b_L}\right)_{i,j} = \frac{\partial y_{L,i}}{\partial b_{L,j}} = \begin{cases} 1 & , i=j \\ 0 & , i \neq j \end{cases} \Rightarrow \frac{\partial y_L}{\partial b_L} = I_{n_L}.$

$\left(\frac{\partial y_L}{\partial y_{L-1}}\right)_{i,j} = \frac{\partial y_{L,i}}{\partial y_{L-1,j}} = A_{L,i,j} \Rightarrow \frac{\partial y_L}{\partial y_{L-1}} = A_L.$

(*) Recall HW#2-6

$f_\theta(x) = u^T \sigma(ax+b) \Rightarrow \nabla_b f_\theta(x) = \mathrm{diag}(\sigma'(ax+b))\, u$

$\nabla_a f_\theta(x) = \mathrm{diag}(\sigma'(ax+b))\, ux.$

$y_{\ell,i} = \sigma\left(\sum_{j=1}^{n_{\ell-1}} A_{\ell,i,j} \cdot y_{\ell-1,j} + b_{\ell,i}\right)$

$\left(\frac{\partial y_\ell}{\partial b_\ell}\right)_{i,j} = \frac{\partial y_{\ell,i}}{\partial b_{\ell,j}} = \begin{cases} \sigma'\left(\sum_{j=1}^{n_{\ell-1}} A_{\ell,i,j} y_{\ell-1,j} + b_{\ell,i}\right) & , i=j \\ 0 & , i \neq j \end{cases} \Rightarrow \frac{\partial y_\ell}{\partial b_\ell} = \mathrm{diag}(\sigma'(A_\ell y_{\ell-1} + b_\ell)).$

$\left(\frac{\partial y_\ell}{\partial y_{\ell-1}}\right)_{i,j} = \frac{\partial y_{\ell,i}}{\partial y_{\ell-1,j}} = \sigma'\left(\sum_{j=1}^{n_{\ell-1}} A_{\ell,i,j} y_{\ell-1,j} + b_{\ell,i}\right) A_{\ell,i,j} \Rightarrow \frac{\partial y_\ell}{\partial y_{\ell-1}} = \mathrm{diag}(\sigma'(A_\ell y_{\ell-1} + b_\ell)) A_\ell$

(b) $y_L = \sum_{j=1}^{n_{L-1}} A_{L,i,j} \cdot y_{L-1,j} + b_L$

$\left(\frac{\partial y_L}{\partial A_L}\right)_{i,j} = \frac{\partial y_L}{\partial A_{L,i,j}} = y_{L-1,j} \Rightarrow \frac{\partial y_L}{\partial A_L} = (y_{L-1,1}, \cdots, y_{L-1,n_{L-1}}) = y_{L-1}^T.$

$y_{\ell,i} = \sigma\left(\sum_{j=1}^{n_{\ell-1}} A_{\ell,i,j} \cdot y_{\ell-1,j} + b_{\ell,i}\right)$

$\left(\frac{\partial y_L}{\partial A_\ell}\right)_{i,j} = \frac{\partial y_L}{\partial A_{\ell,i,j}} = \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{\partial A_{\ell,i,j}} = \sum_{k=1}^{n_\ell} \frac{\partial y_L}{\partial y_{\ell,k}} \sigma'\left(\sum_{j=1}^{n_{\ell-1}} A_{\ell,k,j} \cdot y_{\ell-1,j} + b_{\ell,k}\right) y_{\ell-1,j}$

$\therefore \frac{\partial y_L}{\partial A_\ell} = \mathrm{diag}(\sigma'(A_\ell y_{\ell-1} + b_\ell)) \left(\frac{\partial y_L}{\partial y_\ell}\right)^T y_{\ell-1}^T.$

$\left(\because \left[\mathrm{diag}(\sigma'(A_\ell y_{\ell-1} + b_\ell)) \left(\frac{\partial y_L}{\partial y_\ell}\right)^T y_{\ell-1}^T\right]_{i,j} = \sum_{k=1}^{n_\ell} \sigma'([A_\ell y_{\ell-1}]_k + b_{\ell,k}) \left(\frac{\partial y_L}{\partial y_{\ell,k}}\right) y_{\ell-1,j}\right.$

$\left. = \left(\frac{\partial y_L}{\partial A_\ell}\right)_{i,j} \right)$

1.8 Problem 7: *LeNet5.*

[4]:

```python
import torch
import torch.nn as nn
from torch.optim import Optimizer
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")

'''
Step 1:
'''

# MNIST dataset
train_dataset = datasets.MNIST(root='./mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./mnist_data/',
                              train=False,
                              transform=transforms.ToTensor())


'''
Step 2: LeNet5
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

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

# Original LeNet uses this layer for C3
class C3_layer(nn.Module):
    def __init__(self):
        super(C3_layer, self).__init__()
        self.ch_in_3 = [[0, 1, 2],
                        [1, 2, 3],
                        [2, 3, 4],
                        [3, 4, 5],
                        [0, 4, 5],
```

```
                        [0, 1, 5]] # filter with 3 subset of input channels
        self.ch_in_4 = [[0, 1, 2, 3],
                        [1, 2, 3, 4],
                        [2, 3, 4, 5],
                        [0, 3, 4, 5],
                        [0, 1, 4, 5],
                        [0, 1, 2, 5],
                        [0, 1, 3, 4],
                        [1, 2, 4, 5],
                        [0, 2, 3, 5]] # filter with 4 subset of input channels
        # put implementation here
        self.c3_in_3 = nn.ModuleList()
        self.c3_in_4 = nn.ModuleList()
        self.c3_in_6 = nn.Conv2d(6, 1, 5)

        for _ in range(6):
            self.c3_in_3.append(nn.Conv2d(3, 1, 5))
        for _ in range(9):
            self.c3_in_4.append(nn.Conv2d(4, 1, 5))

    def forward(self, x):
        # put implementation here
        output = []
        for i in range(6):
            output.append(self.c3_in_3[i](x[:, self.ch_in_3[i],:,:]))
        for i in range(9):
            output.append(self.c3_in_4[i](x[:, self.ch_in_4[i],:,:]))
        output.append(self.c3_in_6(x))
        return torch.cat(output, dim=1)

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                #C3_layer_full(),
                C3_layer(),
                nn.Tanh()
                )
```

```
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output


'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")

'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=100, shuffle=True)

import time
start = time.time()
for epoch in range(10) :
```

```
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        train_loss = loss_function(model(images), labels)
        train_loss.backward()

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))


'''
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
 ↪shuffle=False)

for images, labels in test_loader :
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    test_loss += loss_function(output, labels).item()

    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

```
Total number of trainable parameters: 60806
0th epoch starting.
1th epoch starting.
2th epoch starting.
3th epoch starting.
4th epoch starting.
5th epoch starting.
6th epoch starting.
7th epoch starting.
8th epoch starting.
9th epoch starting.
```

```
Time ellapsed in training is: 80.43484139442444
[Test set] Average loss: 0.0004, Accuracy: 9863/10000 (98.63%)
```

[5]:
```
'''
Step 2: LeNet5 with C3_layer_full
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

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

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                C3_layer_full(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()
```

```
    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output


'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")

'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=100, shuffle=True)

import time
start = time.time()
for epoch in range(10) :
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        train_loss = loss_function(model(images), labels)
        train_loss.backward()

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))


'''
```

```
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
↪shuffle=False)

for images, labels in test_loader :
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    test_loss += loss_function(output, labels).item()

    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

```
Total number of trainable parameters: 61706
0th epoch starting.
1th epoch starting.
2th epoch starting.
3th epoch starting.
4th epoch starting.
5th epoch starting.
6th epoch starting.
7th epoch starting.
8th epoch starting.
9th epoch starting.
Time ellapsed in training is: 37.63770842552185
[Test set] Average loss: 0.0004, Accuracy: 9858/10000 (98.58%)
```

- C3_layer_full: 61706 params vs C3_layer: 60806 params (C3_layer_full > C3_layer)
- C3_layer_full: 98.58% accuracy vs 98.63% accuracy (C3_layer_full < C3_layer)

C3_layer_full has more parameters than C3_layer, but shows low accuracy compared to C3_layer.

(*) Expected number of parameters of C3_layer: (3*6+4*9+6)*25 = 1500

(*) Expected number of parameters of C3_layer_full: 6*16*25 = 2400

Difference between of them are equal to difference between 61706 and 60806.