bonus

November 17, 2020

[13]:
```
import pandas as pd
```

1 CS 156a Extra Credit

Victoria Liu

November 20, 2020

1.0.1 Problem 1

1.0.2 Problem 2

Let's train the original dense net model, which has two layers; 200 in the first layer and 100 in the second layer. I'll show the full process of using command line stuff in the first model, but in the future I'll just report the results (otherwise it'll get too crowded).

[4]:
```
!python3 train.py -m dense_2layers_200_100 -d
```

```
2020-11-17 19:22:22.011218: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN)to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2020-11-17 19:22:22.027704: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x7fb4d2c69d80 initialized for platform Host (this does not
guarantee that XLA will be used). Devices:
2020-11-17 19:22:22.027722: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (0): Host, Default Version
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 200)               157000
_________________________________________________________________
activation (Activation)      (None, 200)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               20100
```

```
_________________________________________________________________
activation_1 (Activation)    (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 178,110
Trainable params: 178,110
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2832 -
accuracy: 0.9202 - val_loss: 0.1394 - val_accuracy: 0.9582
Epoch 2/10
469/469 [==============================] - 1s 1ms/step - loss: 0.1134 -
accuracy: 0.9669 - val_loss: 0.1002 - val_accuracy: 0.9679
Epoch 3/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0753 -
accuracy: 0.9775 - val_loss: 0.0844 - val_accuracy: 0.9732
Epoch 4/10
469/469 [==============================] - 1s 1ms/step - loss: 0.0553 -
accuracy: 0.9836 - val_loss: 0.0662 - val_accuracy: 0.9782
Epoch 5/10
469/469 [==============================] - 1s 1ms/step - loss: 0.0425 -
accuracy: 0.9867 - val_loss: 0.0684 - val_accuracy: 0.9786
Epoch 6/10
469/469 [==============================] - 1s 1ms/step - loss: 0.0328 -
accuracy: 0.9899 - val_loss: 0.0753 - val_accuracy: 0.9781
Epoch 7/10
469/469 [==============================] - 1s 1ms/step - loss: 0.0276 -
accuracy: 0.9915 - val_loss: 0.0743 - val_accuracy: 0.9799
Epoch 8/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0217 -
accuracy: 0.9931 - val_loss: 0.0746 - val_accuracy: 0.9782
Epoch 9/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0171 -
accuracy: 0.9945 - val_loss: 0.0826 - val_accuracy: 0.9761
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0148 -
accuracy: 0.9955 - val_loss: 0.0692 - val_accuracy: 0.9808
Figure(640x480)
```

Here are the learning curves, and there are 178,110 trainable parameters.

What about the out-of-sample accuracy?

[6]:
```
!python3 evaluate.py -m dense_2layers_200_100
```

```
2020-11-17 19:34:35.953041: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN)to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2020-11-17 19:34:35.964498: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x7fee01811510 initialized for platform Host (this does not
guarantee that XLA will be used). Devices:
2020-11-17 19:34:35.964516: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (0): Host, Default Version

Training loss: 0.009355693124234676
Training accuracy: 0.9973166584968567

Validation loss: 0.07394441962242126
Validation accuracy: 0.9811999797821045
Figure(640x480)
```

Ok, looks like we have above 98% validation accuracy, which is pretty awesome! Now, we'll try some other values for the number of neurons, and we capture our findings in the following dataframe and learning curves:

[21]:
```
layer1 = [40, 40, 40, 40, 50, 50, 50, 50, 55, 55, 55, 55, 70, 70, 70, 70, 200,␣
 ↪400, 200, 41]
layer2 = [25, 30, 35, 40, 25, 30, 35, 40, 25, 30, 35, 40, 25, 30, 35, 40, 400,␣
 ↪800, 100, 41]
accuracy = [.9668, 0.9675, 0.9699, 0.9699, 0.9684, 0.9705, 0.9713, 0.9718, 0.
 ↪9704, 0.9689, 0.9704, 0.9732, 0.9712, 0.9753, 0.9731, 0.9742, 0.9774, 0.
 ↪9815, 0.9811, 0.9706]
parameters = [32685, 32940, 33195, 33450, 40785, 41090, 41395, 41700, 44835,␣
 ↪45165, 45495, 45825, 56985, 57390, 57795, 58200, 241410, 642810, 178110,␣
 ↪34327]
df = pd.DataFrame(
    {
    'layer1' : layer1,
    'layer2' : layer2,
    'accuracy' : accuracy,
    'parameters' : parameters
    }
    )
df.sort_values(by=['parameters'])
```

[21]:

	layer1	layer2	accuracy	parameters
0	40	25	0.9668	32685
1	40	30	0.9675	32940
2	40	35	0.9699	33195
3	40	40	0.9699	33450

19	41	41	0.9706	34327
4	50	25	0.9684	40785
5	50	30	0.9705	41090
6	50	35	0.9713	41395
7	50	40	0.9718	41700
8	55	25	0.9704	44835
9	55	30	0.9689	45165
10	55	35	0.9704	45495
11	55	40	0.9732	45825
12	70	25	0.9712	56985
13	70	30	0.9753	57390
14	70	35	0.9731	57795
15	70	40	0.9742	58200
18	200	100	0.9811	178110
16	200	400	0.9774	241410
17	400	800	0.9815	642810

[1]:
```
#to delete before submission to clear up notebook
layerone = []
layertwo = []
for layer1 in [40, 50, 55, 70]:
    for layer2 in [25, 30, 35, 40]:
        html_call = f'<img src="dense_2layers_{layer1}_{layer2}_learn.jpg"␣
 ↪width="500"/>'
        print(html_call)
print(f'<img src="dense_2layers_200_400_learn.png" width="500"/>')
print(f'<img src="dense_2layers_400_800_learn.png" width="500"/>')
print(f'<img src="dense_2layers_41_41_learn.png" width="500"/>')
```

```
<img src="dense_2layers_40_25_learn.jpg" width="500"/>
<img src="dense_2layers_40_30_learn.jpg" width="500"/>
<img src="dense_2layers_40_35_learn.jpg" width="500"/>
<img src="dense_2layers_40_40_learn.jpg" width="500"/>
<img src="dense_2layers_50_25_learn.jpg" width="500"/>
<img src="dense_2layers_50_30_learn.jpg" width="500"/>
<img src="dense_2layers_50_35_learn.jpg" width="500"/>
<img src="dense_2layers_50_40_learn.jpg" width="500"/>
<img src="dense_2layers_55_25_learn.jpg" width="500"/>
<img src="dense_2layers_55_30_learn.jpg" width="500"/>
<img src="dense_2layers_55_35_learn.jpg" width="500"/>
<img src="dense_2layers_55_40_learn.jpg" width="500"/>
<img src="dense_2layers_70_25_learn.jpg" width="500"/>
<img src="dense_2layers_70_30_learn.jpg" width="500"/>
<img src="dense_2layers_70_35_learn.jpg" width="500"/>
<img src="dense_2layers_70_40_learn.jpg" width="500"/>
<img src="dense_2layers_200_400_learn.png" width="500"/>
<img src="dense_2layers_400_800_learn.png" width="500"/>
```

```
<img src="dense_2layers_41_41_learn.png" width="500"/>
```

The models are titled as following: "dense_2layers_*layer1*_*layer2*_learn.jpg"

Overall, it seems like the more neurons we add to either layer, the better the accuracy gets. For example, if we keep the number of neurons in the first layer constant and add more neurons in the second layer, the validation accuracy will always increase, and vice versa for keeping the second layer constant. Interestingly, it seems like it might be a good idea to keep the number of neurons in the first and second layers similar; otherwise, the number of parameters will skyrocket. For example, when we have 41 neurons in both layers, we get 0.9706 accuracy and 34327 parameters. When we have a comparable number of total neurons but distributed in 50 vs 30, we get 0.970 accuracy but 41090 parameters. Also, the smallest number of parameters for which I could get over 97 accuracy was around 34327 parameters.