(Better to update the answer than leave it wrong with a comment chain)

1)a)i)

Number of channels \* raw pixels + 1(bias)

1)a)ii)

L1 Regularization

Adds the sum of the absolute values of the coefficients as a penalty to the loss, favors few non-zero coefficients.

L2 Regularization

Adds the squared value of the coefficients as a penalty to the loss, favors all small coefficients.

1)a)iii)

The validation and test sets allow us to evaluate our model on data that was not used to train it. This gives us an unbiased estimate of the performance.

We further use the validation set to select our hyperparameters. Since the validation set was used in parameter tuning it is no longer unbiased and so we use the separate test set to give us a true unbiased model performance.

1)a)iv)

* Reduce the number of features in the model
* Increase size of the training set
* Increase the size of the regularization penalty!
* Early stopping
* Data augmentation: add noise to mitigate overfitting
* Add dropout or batch normalization layers for regularization
* Decrease flexibility



1)b)i)

The partial volume effect is the loss of apparent activity in small objects or regions because of the limited resolution of the imaging system. (More than one class/segment occupies the same pixel)

1)b)ii)



F1 = 2\*(precision\*sensitivity)/(precision+sensitivity)

A = 2\*(0.7\*0.8)/(0.7+0.8)=0.746666

B = 2\*(0.85\*0.65)/(0.85+0.65)=0.736666

Note: You could argue that either one of the models is better – one has better recall, the other one better precision. The two F1 scores are so close to one another, that it’s not really a good measure for a definite ranking of the two.

A

1)b)iii)

Precision = 0.8

Recall = 0.75

New = 2\*(0.8\*0.75)/(0.8+0.75) = 0.77419

Better than both

Note: Again, by raw F1 score this model would be better, but taking a closer look it is only definitely better than model A (as it beats this in all metrics we are given). Model B has a bette r recall, so could be considered better if that is what you want to optimise for.

1)b)iv)

Accuracy = (TP+TN)/(TP+FN+TN+FP)

When there is a class imbalance within the image, the algorithm will receive a very high score by simply assigning the entire image to the dominant class

1)b)v) (Not examinable anymore)

f1(Xa,Xb)f2(Xb,Xc)f3(Xb,Xc,Xd)f4(Xc,Xd,Xe)f5(Xd,Xf)f6(Xg)

2)a)i)

x1 = 1x64x64 -> 1x62x62 -> 1x31x31

x2 = 1x64x64 -> 1x66x66 -> 1x66x66

x3 = 1x64x64 -> 1x62x62 -> 1x64x64

x4 = 1x64x64 -> 1x61x61 -> 1x30x30

x5 = 1x64x64 -> 1x32x32 -> 1x30x30 -> 1x30x30

x6 = 1x64x64 -> 1x64x64 -> 1x61x61-> 1x59x59

------

I wrote a function in python for this that might be useful:

def conv\_output(height, width, kernel\_size, padding, stride, dilation):

height\_out = int(np.floor(((height + 2 \* padding - dilation \* (kernel\_size - 1) - 1) / stride) + 1))

width\_out = int(np.floor(((width + 2 \* padding - dilation \* (kernel\_size - 1) - 1) / stride) + 1))

print(height\_out, width\_out)

return height\_out, width\_out

Eg input:

h = 64

w = 64

h, w = conv\_output(h, w, kernel\_size=3, padding=0, stride=1, dilation=1)

h, w = conv\_output(h, w, kernel\_size=2, padding=0, stride=2, dilation=1)

Output:

62 62

31 31

------

2)a)ii)

c1 = Conv2d(in=1, out=2, kernel=2, stride=2, pad=0) 2x32x32

c2 = Conv2d(in=2, out=4, kernel=2, stride=2, pad=0) 4x16x16

c3 = Conv2d(in=4, out=8, kernel=3, stride=1, pad=0) 8x14x14

c4 = Conv2d(in=8, out=16, kernel=3, stride=1, pad=0) 16x12x12

c5 = Conv2d(in=16, out=32, kernel=3, stride=1, pad=0) 32x10x10

ct1 = ConvTranspose2d(in=32, out=16, kernel=3, stride=1, pad=0) 16x12x12

ct2 = ConvTranspose2d(in=16, out=10, kernel=3, stride=1, pad=0) 10x14x14

ct3 = ConvTranspose2d(in=10, out=10, kernel=3, stride=1, pad=0) 10x16x16

ct4 = ConvTranspose2d(in=10, out=10, kernel=2, stride=2, pad=0) 10x32x32

ct5 = ConvTranspose2d(in=10, out=10, kernel=2, stride=2, pad=0) 10x64x64

2)a)iii)

(Usually in exam = flipping needed)\*that’s cap he said can do either:

2 \* 3 + 3 \* -2 + 4 \* 0 + 3 \* -1 = -3

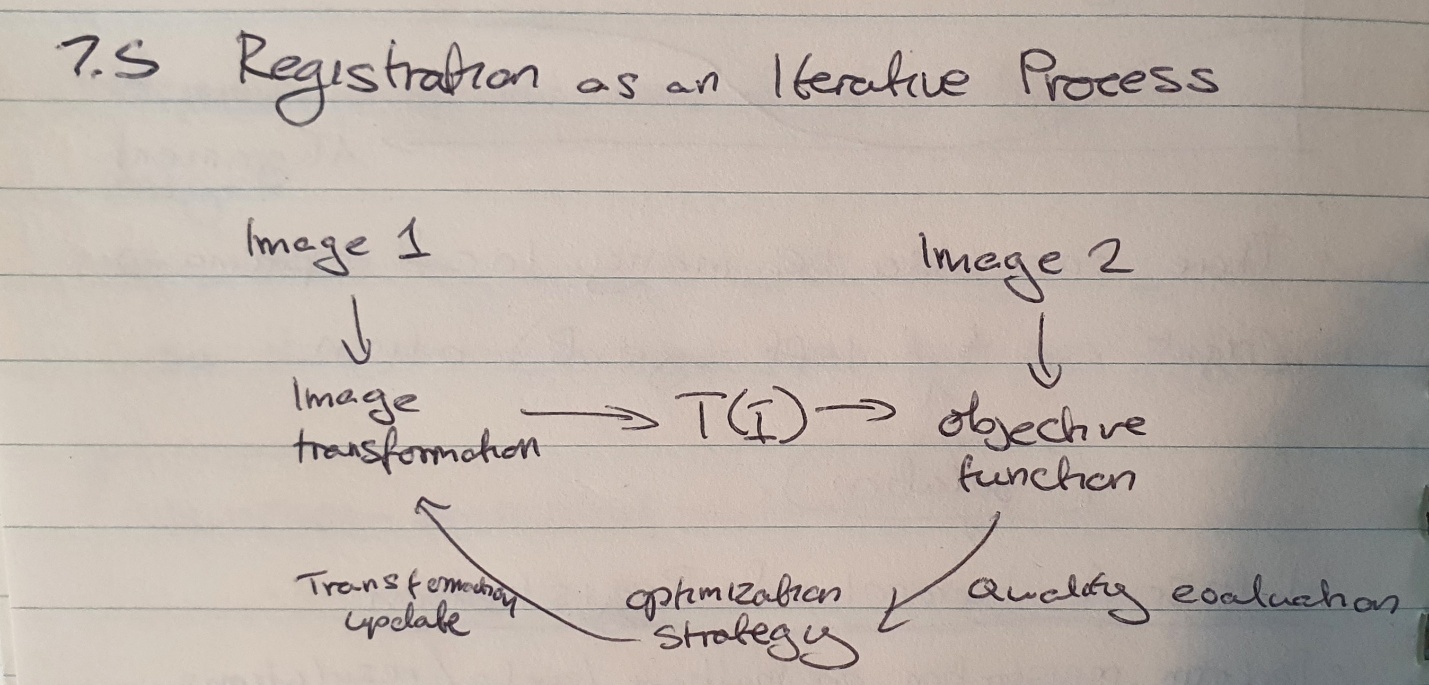
3 \* 3 + 2 \* -2 + 3 \* 0 + 2 \* -1 = 3

|  |  |
| --- | --- |
| -3 | 3 |

With padding:

|  |  |  |  |
| --- | --- | --- | --- |
| -2 | -3 | -2 | 0 |
| -8 | -3 | 3 | 6 |
| -8 | 6 | 5 | 6 |

2)b)i)



* Transform image 1
* Compare transformed image 1 and 2
* Update transformation method based on differences

2)b)ii)

A spatial transformer network allows a neural network to learn how to perform spatial transformations on the input image to enhance the geometric invariance of the model. E.g in the Mnist dataset it would try to make all the images of each number look as similar as possible by scaling, shearing and rotating them.

2)b)iii)

Conv2d(in=1, out=8, kernel=5) 8x60x60

MaxPool2d(kernel=2, stride=2) 8x30x30

ReLU()

Conv2d(in=8, out=10, kernel=5) 10x26x26

MaxPool2d(kernel=2, stride=2) 10x13x13

ReLU()

Linear(10x13x13,32)

ReLU()

Linear(32,3)

Rigid 2D has 3d of freedom

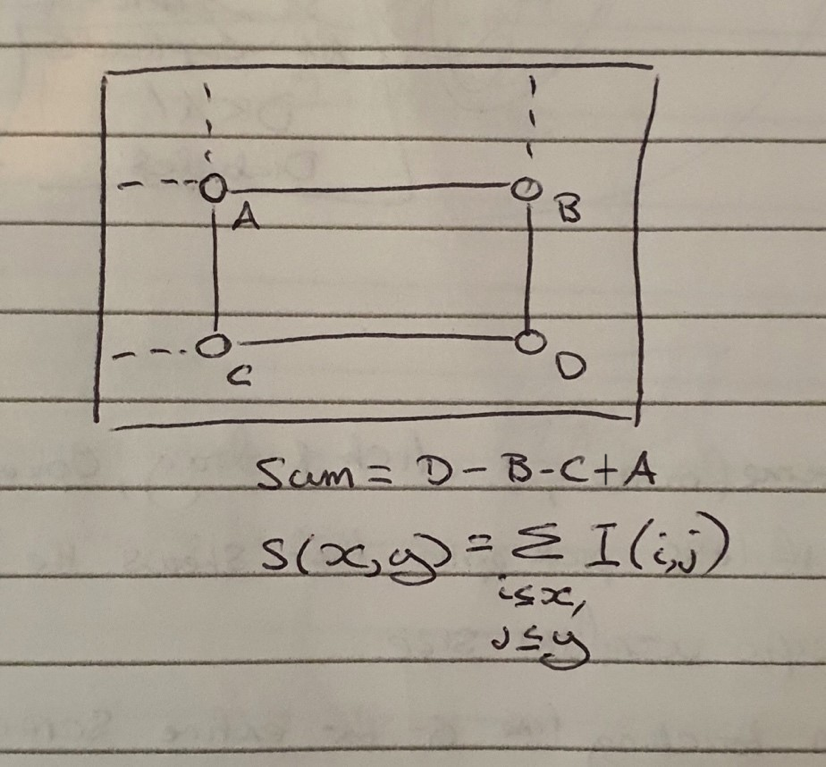
2)b)iv)

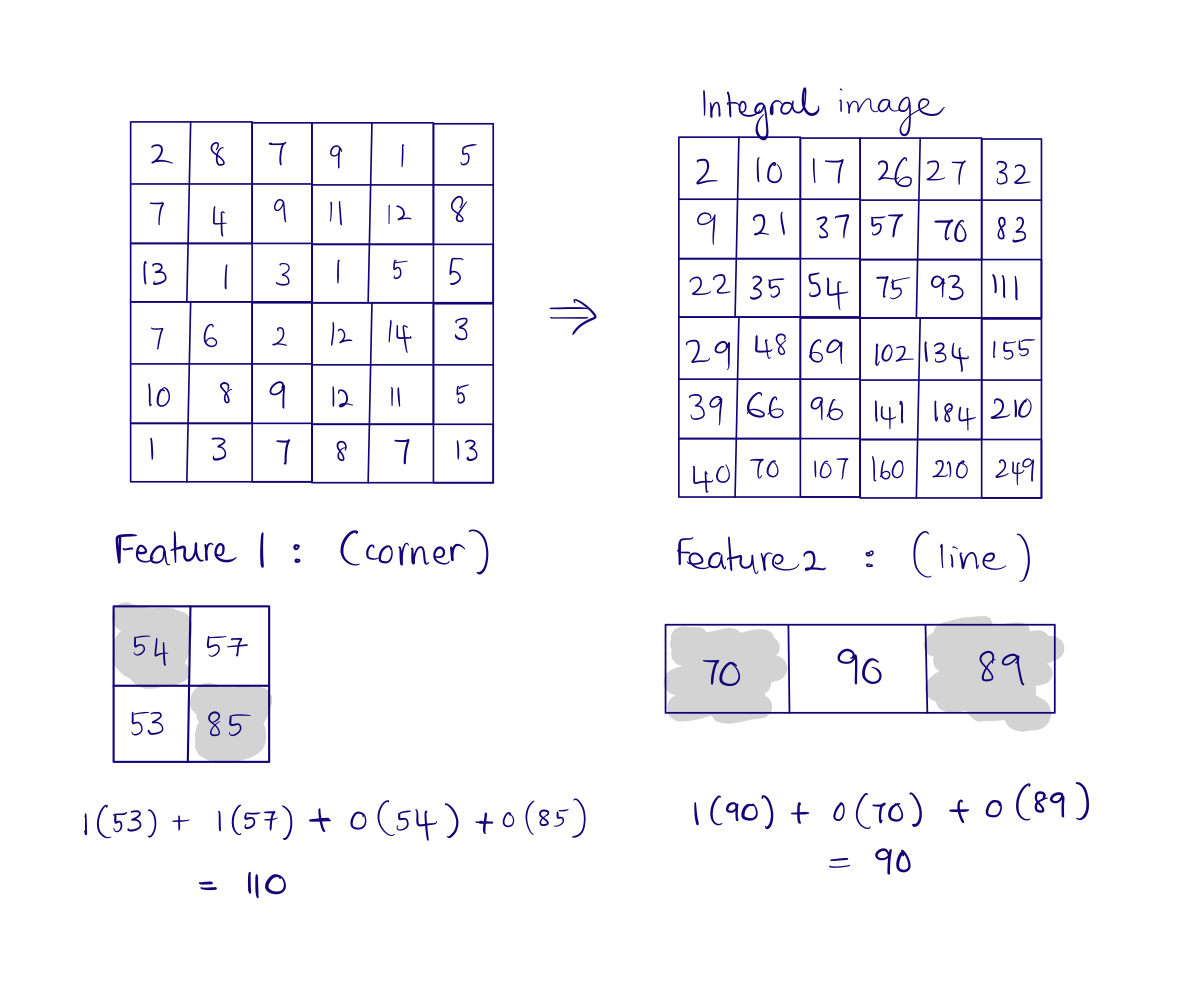
* Sum of squared differences
* Sum of absolute differences
* Correlation Coefficient

3)a)i)

By using Integral Images.

Calculate the sum of intensities from the top left corner to different points in the image. Then we can calculate the intensities in specific regions of the image using the following method

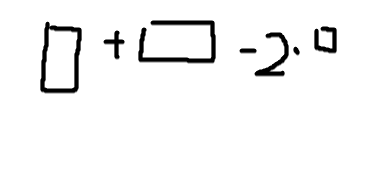


3)a)ii)

My take:

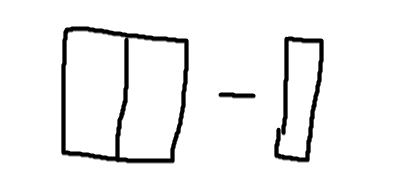
Since they want use to use integral images, we can only use the sums.

For Feature 1:



107 + 111 – 2 \* 54 = 110

For feature 2:



160 – 70 = 90

Same answers but using the area sums and not the individual .

3)b)i)

Only the negative will be weighted (class all as positive) (according to piazza)

3)b)ii)

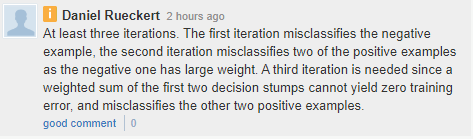
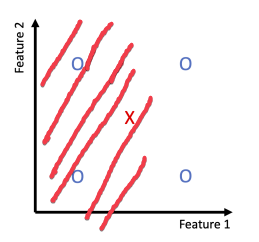
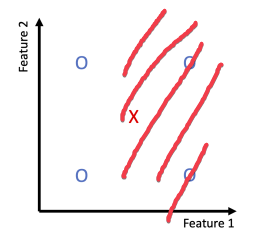
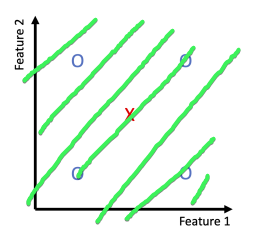


Illustration for the piazza answer:



(All answers below are variations of only classifying the left two first, may be incorrect)

3 stumps.

1st stump separates 2 positives (TL and BL) from the other 3

2nd stump separates the other 2 positives (TR and BR) from the other 3

3rd stump assigns every positive point (TL BL TR BR) and the negative point to positive

Each positive point is assigned as positive 2/3 stumps

Each negative point is assigned as positive 1/3 stumps

\_\_\_\_\_\_\_\_\_\_\_\_\_

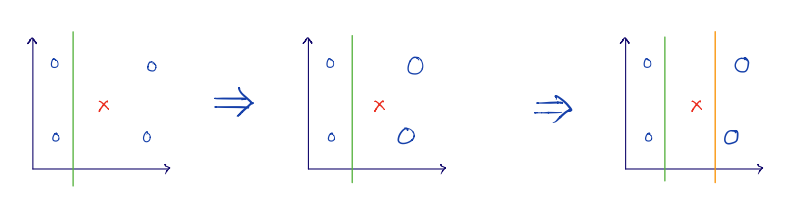
(I’m not sure if the following is correct – please comment accordingly):

First stump (green) correctly classifies 3 of the points, misclassifies the 2 positives on the right. These 2 are therefore assigned higher weights.

Second stump (orange) correctly classifies those as well, achieving zero training loss.

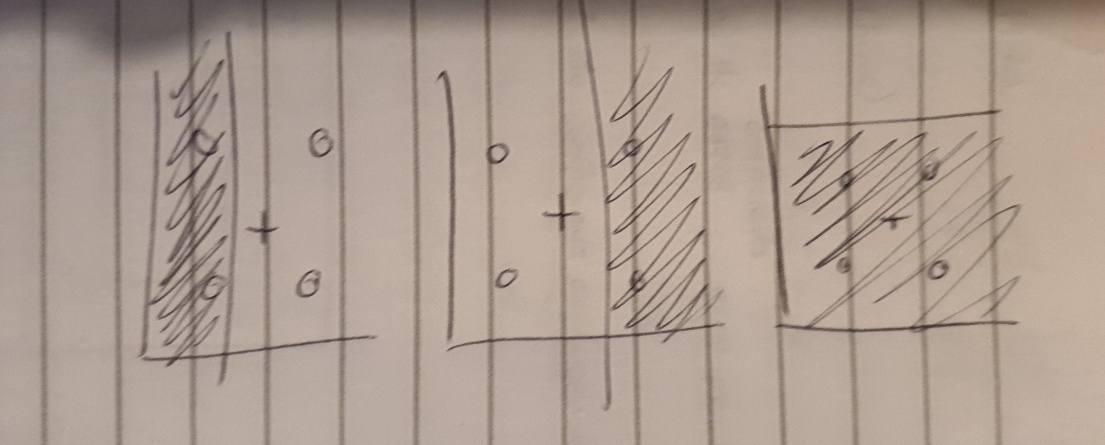
The ensemble of these 2 weak learners (therefore, 2 iterations) should achieve zero training loss.

Horizontal lines should work as well, I believe.

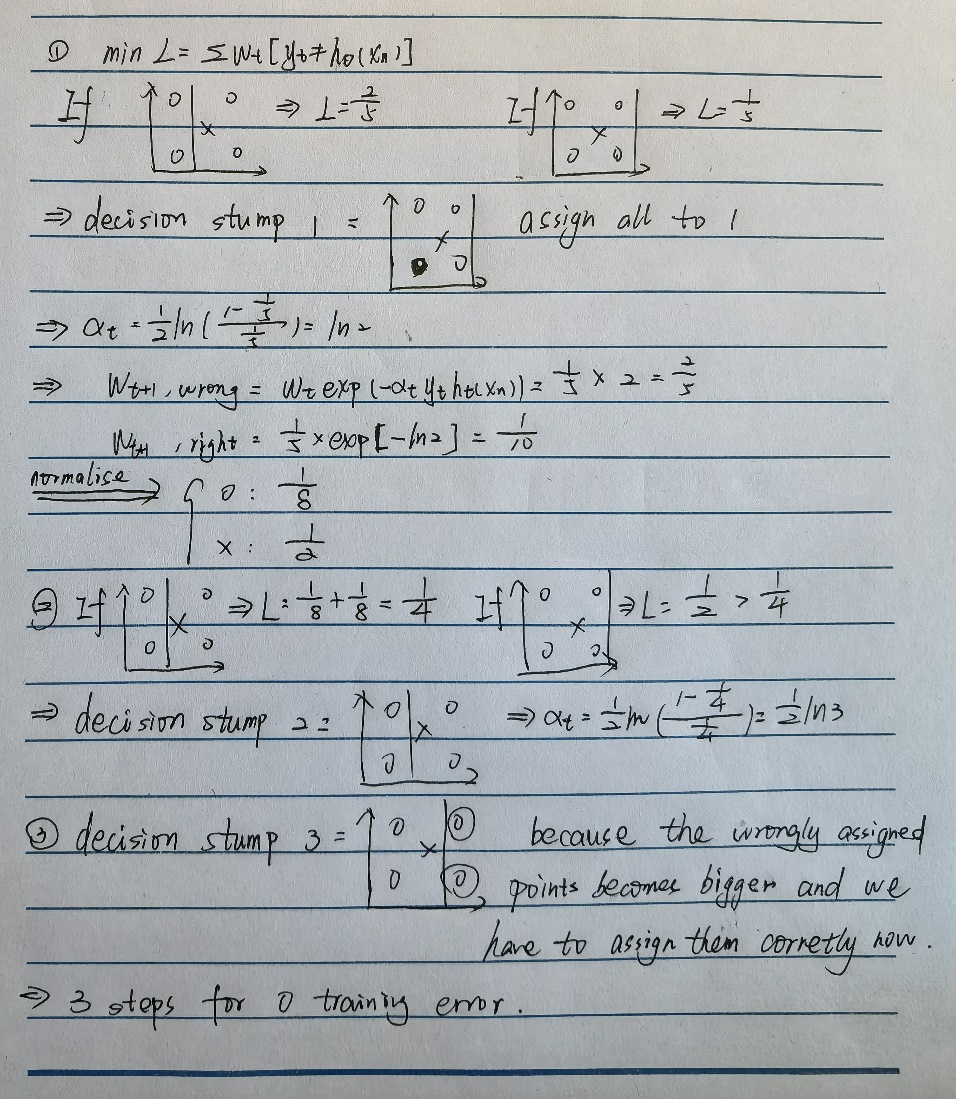


The 2nd classifier has a higher weighting than the 1st (2/3 of the points the 2nd classifier gets correct were wrong in the 1st classifier). This means the 2nd classifier has a higher rating than the 1st and since there are only 2 classifiers the 2nd classifier decides the sign of each point. Therefore the 2 points the 2nd classifier gets wrong are assigned incorrectly

This following scenario covers this issue, the shaded regions are classified as positive



I agree with the above answer but I think the logic should be shown below. Please let me know if I'm wrong, thank you!



3)c)i)

* Extract region proposals (max 2000) from image using a proposal method (selective search)
  + Selective search means we segment the image based on several factors (pixel intensities, colours)
* Warp each region proposal to a standardized size
* For each warped region compute the CNN features
* Classify features with a support vector machine
* Update bounding boxes using regression

3)c)ii)

* Slow to train & slow inference time because the CNN features must be computed for each region one at a time.
* Selective search is a fixed algorithm, therefore no learning happens at that stage
* ad-hoc training objectives: SVM and bbox regressors can’t be trained until each convnet has finished (bottleneck)

3)c)iii)

* Instead of feeding the region proposals to the CNN we feed the input image in and generate a convolutional feature map. We then identify regions of proposals and warp them into squares and use a RoI pooling layer to warp them into squares.
* This is fed into a fully connected layer which outputs a classifier and bounding box for each region of interest

The reason “Fast R-CNN” is faster than R-CNN is because you don’t have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.

3)c)iv)

Mask R-CNN -> Add FCN that acts on region proposals and trains FCN to perform segmentation using annotated masks.

4)a)i)

ReLU is useful since it stops us from activating all the neurons at the same time. If neurons are not important in the computation (give a value less than 0) we essentially turn them off.  
This is useful during backpropagation since we can train only the neurons that are “turned on”.

This makes the network far more computationally efficient.

Binary step function is not suited for building neural networks. The gradient of the step function is always 0 which means it is not possible to perform gradient descent. The derivate is always 0.

(A more common answer)

ReLU has gradient 1 if x > 0 which means the gradient is not diminished along the back path if some neurons are > 0.

Tanh/sigmoid would be susceptible to vanishing gradients as they have very low gradients at most values

4)a)ii)

They add non-linearity to the network which makes it much more powerful. The network can learn much more complex patterns from the data.

Note that without these activations, even a very deep network would be equivalent to a linear combination of the inputs, which would severely restrict the class of functions that could be represented by the model.

4)b)

|  |  |
| --- | --- |
| 128x128x3 | n/a |
| 120x120x32 | ((9 \* 9 \* 3) + 1) \* 32 = 7808 |
| 60x60x32 (Pool) | 0 |
| 56x56x64 | (5 \* 5 \* 32 + 1) \* 64 = 51264 |
| 28x28x64 (Pool) | 0 |
| 24x24x64 | (5 \* 5 \* 64 + 1) \* 64 = 102464 |
| 12x12x64 (Pool) | 0 |
| 4 | 12\*12\*64\*4 + 4 = 36868 |

4)c)i)

(answers using log10, not log2)

-(log(0.29)) = 0.5376

4)c)ii)

-(log(0.45)) = 0.3468

4)c)iii)

The model predicted a higher rating for the correct answer. Cross Entropy loss is 0 for all classes except for the actual class so it only bases the loss on the higher rating of the correct answer.

Initially, the network was untrained and very unsure, so predicted an almost equal probability for all classes. As it trains more, it outputs higher-probability predictions, even if they are occasionally wrong.

My take: correct prediction: 0.25 0.26 0.25 0.24 (correct class is 2), loss= 1.34

Incorrect prediction: 0 0.3 0.7 0 loss = 1.2

So an incorrect prediction can have a lower loss than a correct one in situations where the network assigns the correct class a higher probability, even if it assigns another, wrong class the highest probability.

4)c)iv)

(How would you have a differentiable accuracy loss?)

If there is class imbalance the network may train itself to guess every object as the class with the highest number of entries

Furthermore, accuracy provides a less useful learning signal, since you are just saying whether the network guessed correctly or incorrectly for each element, rather than comparing probability distributions. This results in a less smooth learning process.

4)d)i)

multidimensional vector that contains each type (car, sports, electric, truck, plane, ship) where 1 for every class it belongs to and 0 for those it does not.

E.g sports-car would be (1,1,0,0,0,0)

4)d)ii)

We now have multiple positive classes. Softmax averages the weight of all classes to 1 and therefore if there were 2 correct classes the max it could guess would be 0.5 for each. Since we are now using Sigmoid with cross entropy the network could achieve a loss of 0 by simply outputting 1 for every class

Softmax encourages one of the output elements to be very high and the rest to be very low. This is appropriate when there is only one correct output label but will not let the network easily learn to output multiple labels for the same image.

4)d)iii)

Use sigmoid in final layer

Set output to 1 if sigmoid > 0.5 or other value

Use Binary Cross Entropy Loss [BCELoss — PyTorch 1.8.0 documentation](https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html)

Multiple sigmoids in final layer? One for each label (would need to add neurons [to (c) architecture] in output layer so one for each label)

^ I mean sigmoid is applied element wise, no? Say you had a 1x6 vector, a single sigmoid is enough

Followed by x[x > threshold] = 1