Q1

What you want to construct is an OR gate based on previous and current values. It is sufficient to consider the base case at $t0 \rightarrow t1$ and the case for $t1 \rightarrow t2$. The rest follows by induction. At time step 1 we must satisfy two equations:

$$\begin{aligned}
&h_1 = f(w_1 + b_2) \quad for\ x_1 = 1\\
&y_1 = 1 = x_3 h_1 + b_3\\
&h_1 = f(b_2)]; \quad for\ x_1 = 0\\
&y_1 = 0 = x_3 h_1 + b_3
\end{aligned} \tag{1}$$

For this case, we can choose $b_2 > 0$ or $b_2 \leq 0$. In the latter case, this creates a contradiction where $w_1 + b_2 > 0$ and $w_1 + b_2 \leq 0$ simultaneously. Therefore we choose $b_2 > 0$. This implies the following:

$$\begin{aligned}
b_3 &= 1\\
x_3 &= -1\\
w_1 + b_2 &\leq 0\\
b_2 &> 0
\end{aligned} \tag{2}$$

Working on the system of equations for $t1 \rightarrow t2$ given that $h_1 = 0$ for $x_1 = 1$ and $h_1 = 1$ for $x_1 = 0$:

$$\begin{aligned}
h_2 &= f(b_2) \quad for\ x_1 = 1, x_2 = 0\\
y_2 = 0 &= x_3 h_2 + b_3\\
h_2 &= f(w_1 + b_2) = 0 \quad for\ x_1 = 1, x_2 = 1\\
y_2 = 1 &= b_3 = 1\\
h_2 &= f(w_2 + b_2) \quad for\ x_1 = 0, x_2 = 0\\
y_2 = 0 &= x_3 h_2 + b_3\\
h_2 &= f(w_1 + w_2 + b_2) \quad for\ x_1 = 0, x_2 = 1\\
y_2 = 0 &= x_3 h_2 + b_3
\end{aligned} \tag{3}$$

This creates the following implications:

$$\begin{aligned}
b_2 &> 0\\
w_1 + b_2 &\leq 0\\
w_2 + b_2 &> 0\\
w_1 + w_2 + b_2 &> 0\\
b_3 &= 1\\
x_3 &= -1\\
b_2 &= 1\\
w_1 &= -1\\
w_2 &= 1
\end{aligned} \tag{4}$$

Q2

Using the same reasoning as above, you can establish similar set of constraints:

$$\begin{aligned} b_2 &\leq 0 \\ w_1 + b_2 &> 0 \\ w_2 + b_2 &\leq 0 \\ w_1 + w_2 + b_2 &> 0 \end{aligned} \tag{5}$$

The solution for these constraints are:

$$\begin{aligned} b_3 &= 1 \\ x_3 &= -1 \\ b_2 &= 0 \\ w_1 &= 1 \\ w_2 &= 0 \end{aligned} \tag{6}$$

Which interestingly removes the concurrent feedback path in the RNN.

Q3

Highest weighted unigrams (-):

('too', tensor(0.4642)), ('bad', tensor(0.4078)), ("n't", tensor(0.3311)), ('no', tensor(0.3101)), ('dull', tensor(0.2750)), ('worst', tensor(0.2291)), ('minutes', tensor(0.2126)), ('plot', tensor(0.2024)), ('mess', tensor(0.1895)), ('flat', tensor(0.1803))

Least weighted unigrams (-):

('funny', tensor(-1.3329)), ('most', tensor(-1.1824)), ('best', tensor(-1.1650)), ('love', tensor(-1.1536)), ('story', tensor(-1.0664)), ('about', tensor(-1.0500)), ('performances', tensor(-0.9468)), ('life', tensor(-0.8979)), ('who', tensor(-0.8322)), ('work', tensor(-0.8290))

Highest weighted unigrams (+):

('best', tensor(0.4150)), ('fun', tensor(0.3577)), ('love', tensor(0.3464)), ('heart', tensor(0.3055)), ('entertaining', tensor(0.2991)), ('fascinating', tensor(0.2743)), ('world', tensor(0.2416)), ('solid', tensor(0.2404)), ('documentary', tensor(0.2351)), ('us', tensor(0.2328))

Least weighted unigrams (+):

("n't", tensor(-3.2279)), ('too', tensor(-2.1348)), ('or', tensor(-1.7833)), ('like', tensor(-1.7680)), ('bad', tensor(-1.5712)), ('no', tensor(-1.3889)), ('just', tensor(-1.3437)), ('more', tensor(-1.1376)), ('much', tensor(-1.0718)), ('does', tensor(-1.0701))

Epoch	**Training Accuracy**	**Dev Accuracy**
0	0.68916	0.7236
1	0.7459	0.7201
2	0.7579	0.7190
3	0.7610	0.7202
4	0.7638	0.7201
5	0.7653	0.7202
6	0.7658	0.7202
7	0.7657	0.7202
8	0.7660	0.7202
9	0.7666	0.7202
10	0.7664	0.7202
11	0.7673.	0.7202
12	0.7676	0.7202
13	0.7680	0.7202
14	0.7675	0.7202
15	0.7680	0.7202
16	0.7670	0.7202
17	0.7682	0.7202
18	0.7674	0.7202
19	0.7683	0.7202

Table 1: Unigram Model - Epoch 20, Learning Rate 0.001

Q4

I do not think with greater epochs the dev accuracy will improve. It seems, since my unigram model does not contain prefix or suffix features, it cannot generalize at all to unseen words. Further there are certain unigram words that require context to decide whether it is (+) or (-) which overfits in the training data set. Also more than a few epochs doesn't seem to help as the parameter updates are purely additive or subtractive on the same training set, which only accentuates the overfitting as time goes on, especially against unseen data in the dev set. The table of the epoch is shown in Table 1.

Q5

Too lazy to matplotlib or table-fy the result in latex, so here it is, you can see the performance dramatically improves in both training and dev with bigram and trigram models that incorporates <UNK> token. I also added L2 regularization.

epoch 0: train acc = 0.6885838150289018, dev acc = 0.7419724770642202
epoch 1: train acc = 0.8197976878612717, dev acc = 0.7603211009174312
epoch 2: train acc = 0.875578034682081, dev acc = 0.7626146788990825
epoch 3: train acc = 0.903757225433526, dev acc = 0.7660550458715596
epoch 4: train acc = 0.9215317919075144, dev acc = 0.7706422018348624
epoch 5: train acc = 0.9338150289017341, dev acc = 0.7729357798165137
epoch 6: train acc = 0.9442196531791908, dev acc = 0.7672018348623854
epoch 7: train acc = 0.95, dev acc = 0.7752293577981652
epoch 8: train acc = 0.9578034682080925, dev acc = 0.7821100917431193
epoch 9: train acc = 0.9627167630057804, dev acc = 0.783256880733945
epoch 10: train acc = 0.9661849710982658, dev acc = 0.786697247706422

```
epoch 11: train acc = 0.9684971098265895, dev acc = 0.7935779816513762
epoch 12: train acc = 0.9703757225433526, dev acc = 0.7958715596330275
epoch 13: train acc = 0.9738439306358382, dev acc = 0.7981651376146789
epoch 14: train acc = 0.9761560693641619, dev acc = 0.7993119266055045
epoch 15: train acc = 0.977456647398844, dev acc = 0.7970183486238532
epoch 16: train acc = 0.9787572254335261, dev acc = 0.7970183486238532
epoch 17: train acc = 0.9799132947976879, dev acc = 0.7970183486238532
epoch 18: train acc = 0.982514450867052, dev acc = 0.7970183486238532
epoch 19: train acc = 0.9832369942196532, dev acc = 0.7981651376146789
```

Q6

A basic feed-forward linear neural network was implemented. It seems that some learning is happening and it does perform better than the unigram model.

```
epoch 0: train acc = 0.7270231213872832, dev acc = 0.7534403669724771
epoch 1: train acc = 0.7614161849710983, dev acc = 0.7729357798165137
epoch 2: train acc = 0.7705202312138728, dev acc = 0.7752293577981652
epoch 3: train acc = 0.7697976878612717, dev acc = 0.7603211009174312
epoch 4: train acc = 0.7721098265895954, dev acc = 0.7637614678899083
epoch 5: train acc = 0.7797687861271676, dev acc = 0.75
epoch 6: train acc = 0.7791907514450868, dev acc = 0.7603211009174312
epoch 7: train acc = 0.7807803468208092, dev acc = 0.75
epoch 8: train acc = 0.7771676300578034, dev acc = 0.6961009174311926
epoch 9: train acc = 0.7836705202312139, dev acc = 0.7557339449541285
```

Q7

The improvement over the unigram model is significant and it seems to even surpass the hand-tuned ngram features of the perceptron classifier. It does see to be overfitting, so adding some dropout (0.2) to the RNN could be useful (currently 0).

```
epoch 0: train acc = 0.7732658959537573, dev acc = 0.805045871559633
epoch 1: train acc = 0.8543352601156069, dev acc = 0.819954128440367
epoch 2: train acc = 0.8947976878612717, dev acc = 0.8142201834862385
epoch 3: train acc = 0.9290462427745665, dev acc = 0.8188073394495413
epoch 4: train acc = 0.9606936416184971, dev acc = 0.8188073394495413
epoch 5: train acc = 0.9773121387283237, dev acc = 0.8061926605504587
epoch 6: train acc = 0.9851156069364162, dev acc = 0.8084862385321101
epoch 7: train acc = 0.9891618497109826, dev acc = 0.8038990825688074
epoch 8: train acc = 0.9920520231213873, dev acc = 0.8119266055045872
epoch 9: train acc = 0.9927745664739884, dev acc = 0.7855504587155964
```

Q8

Skipped. I assume you can figure out how to implement this with a multi-layered transformer where the input are your GloVe vectors (with different sentence length padded). On the output, you just look at a single sigmoid-ed output. Just have to figure out how to do the loss with tensorflow or pytorch, which will require reading the documentation and look up examples on the Google.

Alternatively you can improve the RNN, see the comments in the code for *MyNNClassifier* in *model.py*.