Q1 (a)

$K = 4$ , $s = 1$ , $p = 1$

$Op = (W - K + 2P)/s + 1$

where Op is output Dimension

$\Rightarrow$ $W = 50$ $\rightarrow$ Input Image size

$K = 4$

$P = 1$

$s = 1$

$Op = (50 - 4 + 2)/1 + 1$

$= 49$

Output Dimensions $= 49 \times 49$

(b) $K = 8$ , $s = 5$, $p = 0$

$Op = (50 - 8)/5 + 1$

$= \frac{42}{5} + 1$ $\longrightarrow$ We can't use this setup since with a kernel size of 8 & stride of 5 leading to 50 - 8, which is indivisible by 5

(c) Dim $= (W - K + 2P) / S + 1$

where $W$ = Input size
$K$ = kernel size
$S$ = stride
$P$ = Padding

Dim $= \frac{50 + 4 - 10}{2} + 1 = 23$

$23 \times 23 \longrightarrow$ output Dimensions

(d) Dim $= (W + 2P - K) / S + 1$

$\frac{50 - 2}{1} + 1 = 49$

## Q1b

Confusion Matrix:

[[ 973 0 0 0 0 1 1 2 3 0] [ 0 1130 0 2 0 1 0 2 0 0] [ 0 1 1018 6 1 0 0 4 2 0] [ 0 0 0 1006 0 3 0 1 0 0] [ 0 0 2 0 969 0 1 1 1 8] [ 1 0 0 6 0 882 1 1 1 0] [ 2 3 1 1 1 14 935 0 1 0] [ 0 2 5 2 0 0 0 1018 1 0] [ 1 0 1 3 1 2 0 2 962 2] [ 0 2 0 4 1 4 0 4 0 994]]

On the basis of the confusion matrix generated here, the classifier made quite a lot of misclassification errors with 6,7 and 9, 6 was being confused for 5, while 7 was incorrectly predicted as 2, while 9 was being misclassified as 7,5,and 3

## Q2

1. The Principal Component Analysis is a method where we identify vectors or principal componenets that capture the variance in the data. PCA is quite sensitive to variance so the first thing to do is to standardize the data, this is followed by calculating the covariance matrix, and the eigenvectors and eigenvalues of the covariance matrix give us the Principal Components, these eigenvectors are sorted on the basis of the eigenvalues. The two eigen vectors with the biggest eigenvalues can be used to visualize the data in 2 dimensions. Torch.svd performs singular value decomposition to give U,S and V transpose. These can be used to get our principal components.

1. PCA projects the data based on a linear transformation to a new space with basis as prinicipal components that explain the most variance of the data allowing us to reduce dimensionality and discard those principal components that do not capture a lot variance. Autoencoders n be used to achieve dimensionality reduction but they do this using non linear transformations. An autoencoder with a single linear layer is wuite similar to PCA. In both cases one can reconstruct the data back, in case of PCA this is done by multiplying back the SVD matrices while in autoencoders the decoder can reconstruct input from the latent embedding.

1. The linear autoencoder uses a series of linear activation functions for its layers, however these do not capture spatial or temporal dependencies in the input, The convolutional autoencoders however are able to do so using the filters that perform convolutions. Thus using operations such as downsampling or pooling these convolutional autoencoders create simpler representations for complex inputs and then reconstruct them by functions such as upsampling or max unpooling.

### Training the convolutional Autoencoder

In [ ]:
```
from google.colab import drive
drive.mount('/content/mydrive')
```

```
Drive already mounted at /content/mydrive; to attempt to forcibly remount, cal
l drive.mount("/content/mydrive", force_remount=True).
```

In [ ]:
```
!cp mydrive/MyDrive/p2_ae.py .
```

```
In [ ]:  import importlib
         import p2 as autoencoder
         from matplotlib import pyplot as plt
```

```
In [ ]:  importlib.reload(autoencoder)
```

```
In [ ]:  loss_vals,outputs = autoencoder.train()
```

```
loss: 1.0336779356002808         at epoch: 0

loss: 1.0231460332870483         at epoch: 1

loss: 1.0293573141098022         at epoch: 2

loss: 1.0374267101287842         at epoch: 3

loss: 1.035231351852417  at epoch: 4

loss: 1.0142886638641357         at epoch: 5

loss: 1.0323688983917236         at epoch: 6

loss: 1.0372719764709473         at epoch: 7

loss: 1.033547282218933  at epoch: 8

loss: 1.0192219018936157         at epoch: 9

loss: 1.0131200551986694         at epoch: 10

loss: 1.0211453437805176         at epoch: 11

loss: 1.0111045837402344         at epoch: 12

loss: 1.0232096910476685         at epoch: 13

loss: 1.023966908454895  at epoch: 14

loss: 1.0216848850250244         at epoch: 15

loss: 1.0276411771774292         at epoch: 16

loss: 1.0102962255477905         at epoch: 17

loss: 1.0168977975845337         at epoch: 18

loss: 1.016855239868164  at epoch: 19

loss: 1.0216081142425537         at epoch: 20

loss: 1.016607642173767  at epoch: 21

loss: 1.0251773595809937         at epoch: 22

loss: 1.0261647701263428         at epoch: 23

loss: 1.0241814851760864         at epoch: 24

loss: 1.0277217626571655         at epoch: 25

loss: 1.0284432172775269         at epoch: 26

loss: 1.0321084260940552         at epoch: 27

loss: 1.0174095630645752         at epoch: 28

loss: 1.0195969343185425         at epoch: 29
```

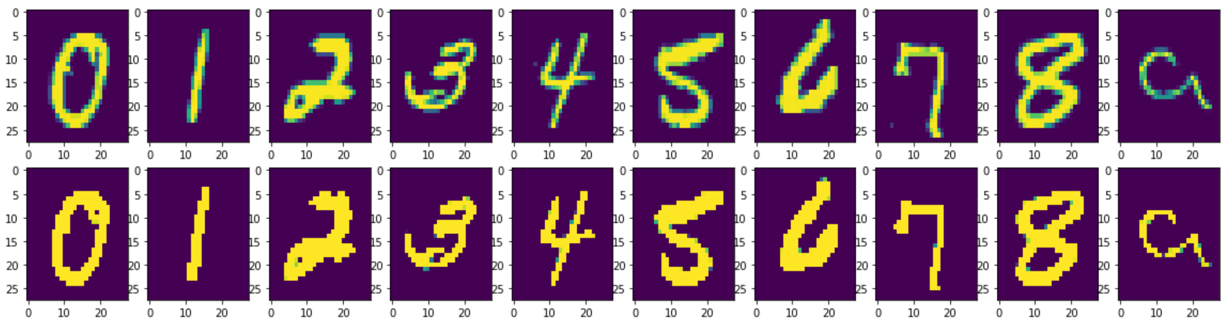
```
In [ ]:  image = outputs[-1][1]
```

```
labels = outputs[-1][3]
reconstructed_image = outputs[-1][2]
image_ = image.reshape(-1, 28, 28)
reconstructed_image_ = reconstructed_image.detach().reshape(-1, 28, 28)
```

In [ ]:

```
f, ax = plt.subplots(2,10,figsize = (20,5))

for i in range(10):
  for j in range(2):
    if(j==0):
      ax[j,i].imshow(image_[labels == i][0],aspect = 'auto')
    else:
      ax[j,i].imshow(reconstructed_image_[labels == i][0],aspect = 'auto')
```

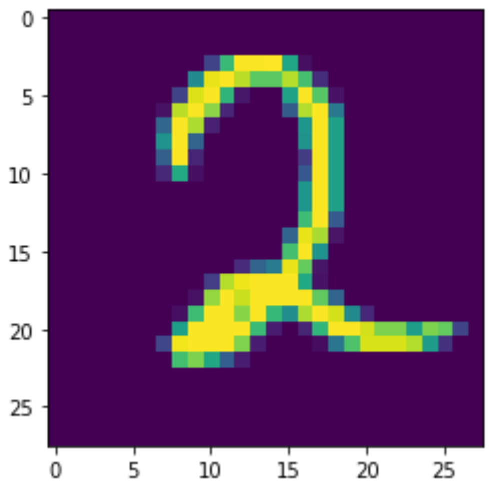


In [ ]:

```
plt.imshow(image_[labels == 2][5])
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f6f1f737dd0>
```

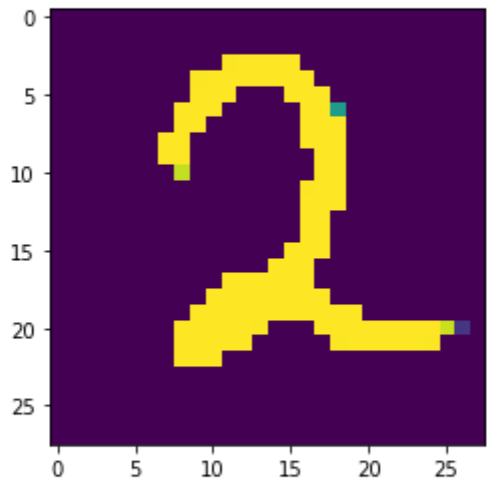


In [ ]:

```
plt.imshow(reconstructed_image_[labels == 2][5])
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f6f30f05bd0>
```



1. A denoising autoencoder is one where the input is first corrupted using functions such as gausian noise and the autoencoder is trained to reconstruct the original input, thus the loss is calculated with the original input while the autoencoder is fed the corrupted version. A Variational Autoencoder is one which can be used for data generation. Rather than creating a latent embedding the VAE generates the mean and standard deviations for a latent distribution. In the loss function in addition to an mse term between the reconstructed and original input a KL divergence term is calculated between a standard normal and the normal distribution returned by the encoder in order to regularize and avoid overfitting. This also enables continuity and interpolation, allowing us to sample from the latent space and generate meaningful features. In a denoising autoencoder such interpolation would not result in meaningful outputs.

## Q3

In [ ]:

```python
import rnn as lstmae
import importlib
importlib.reload(lstmae)
```

```
vocab size: 658
```

Out[ ]:

```
<module 'rnn' from '/Users/venugopalbhatia/Documents/Deep Learning Theory and
Applications/Assignment 4/rnn.py'>
```

In [ ]:

```python
lstmae.train()
```

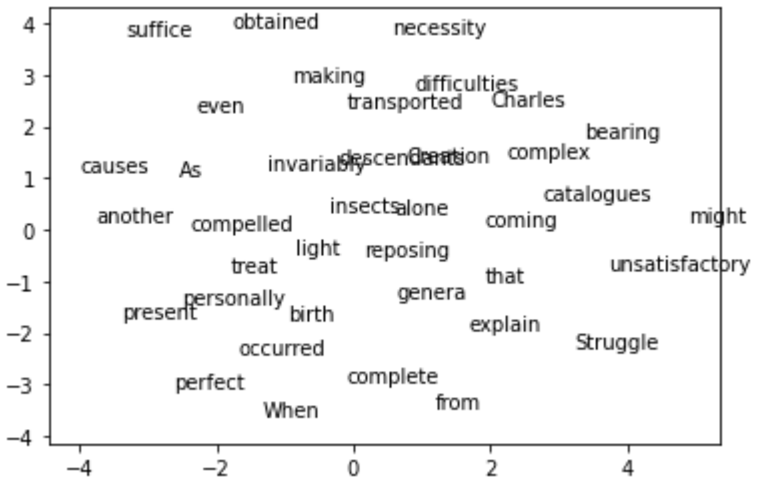
```
/Users/venugopalbhatia/Documents/Deep Learning Theory and Applications/Assignm
ent 4/rnn.py:95: UserWarning: Implicit dimension choice for softmax has been d
eprecated. Change the call to include dim=X as an argument.
  prediction = F.softmax(prediction)
Iteration:  0 loss:  6.489196300506592 perplexity: 657.9943209457862
Iteration:  10 loss:  6.460655212402344 perplexity: 639.4799145051444
Iteration:  20 loss:  6.443840980529785 perplexity: 628.8174427859442
Iteration:  30 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  40 loss:  6.491812705993652 perplexity: 659.7181550386665
Iteration:  50 loss:  6.491813659667969 perplexity: 659.718784195227
Iteration:  60 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  70 loss:  6.443813323974609 perplexity: 628.8000521021265
Iteration:  80 loss:  6.423813343048096 perplexity: 616.348988601647
Iteration:  90 loss:  6.42181396484375 perplexity: 615.1179049785542
Iteration:  100 loss:  6.451813220977783 perplexity: 633.8505626011029
Iteration:  110 loss:  6.4318132400512695 perplexity: 621.2994923866887
Iteration:  120 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  130 loss:  6.4918131828308105 perplexity: 659.7184696168717
```

```
Iteration:  140 loss:  6.491812705993652 perplexity: 659.7181550386665
Iteration:  150 loss:  6.473813056945801 perplexity: 647.9496916175866
Iteration:  160 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  170 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  180 loss:  6.439813137054443 perplexity: 626.2897585274395
Iteration:  190 loss:  6.431812763214111 perplexity: 621.299196128075
Iteration:  200 loss:  6.467813491821289 perplexity: 644.0739133629853
Iteration:  210 loss:  6.451813220977783 perplexity: 633.8505626011029
Iteration:  220 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  230 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  240 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  250 loss:  6.449812889099121 perplexity: 632.5839183911748
Iteration:  260 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  270 loss:  6.469812870025635 perplexity: 645.362948912731
Iteration:  280 loss:  6.457812786102295 perplexity: 637.6648208348649
Iteration:  290 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  300 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  310 loss:  6.453813076019287 perplexity: 635.1194402070215
Iteration:  320 loss:  6.491812705993652 perplexity: 659.7181550386665
Iteration:  330 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  340 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  350 loss:  6.423813343048096 perplexity: 616.348988601647
Iteration:  360 loss:  6.455812931060791 perplexity: 636.3908579232971
Iteration:  370 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  380 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  390 loss:  6.489812850952148 perplexity: 658.4001327264009
Iteration:  400 loss:  6.447813510894775 perplexity: 631.3204074313693
Iteration:  410 loss:  6.411813259124756 perplexity: 608.9969497792877
Iteration:  420 loss:  6.403813362121582 perplexity: 604.144472444436
Iteration:  430 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  440 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  450 loss:  6.461813449859619 perplexity: 640.221013196582
Iteration:  460 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  470 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  480 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  490 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  500 loss:  6.4238128662109375 perplexity: 616.3486947036168
Iteration:  510 loss:  6.439813137054443 perplexity: 626.2897585274395
Iteration:  520 loss:  6.453813076019287 perplexity: 635.1194402070215
Iteration:  530 loss:  6.455813407897949 perplexity: 636.3911613781777
Iteration:  540 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  550 loss:  6.465813159942627 perplexity: 642.7868394984783
Iteration:  560 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  570 loss:  6.431812286376953 perplexity: 621.2988998696026
Iteration:  580 loss:  6.463812351226807 perplexity: 641.502031743488
Iteration:  590 loss:  6.451813220977783 perplexity: 633.8505626011029
Iteration:  600 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  610 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  620 loss:  6.451813220977783 perplexity: 633.8505626011029
Iteration:  630 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  640 loss:  6.491812705993652 perplexity: 659.7181550386665
Iteration:  650 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  660 loss:  6.471813201904297 perplexity: 646.6551810083894
Iteration:  670 loss:  6.461813449859619 perplexity: 640.221013196582
Iteration:  680 loss:  6.429813385009766 perplexity: 620.058225055616
Iteration:  690 loss:  6.4378132820129395 perplexity: 625.0385213599036
Iteration:  700 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  710 loss:  6.439813613891602 perplexity: 626.2900571657394
Iteration:  720 loss:  6.465813636779785 perplexity: 642.7871460032013
Iteration:  730 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  740 loss:  6.4458136558532715 perplexity: 630.0591197483975
Iteration:  750 loss:  6.403813362121582 perplexity: 604.144472444436
Iteration:  760 loss:  6.487813472747803 perplexity: 657.0850569562913
Iteration:  770 loss:  6.461813449859619 perplexity: 640.221013196582
Iteration:  780 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  790 loss:  6.483813285827637 perplexity: 654.4618440748899
Iteration:  800 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  810 loss:  6.4318132400512695 perplexity: 621.2994923866887
Iteration:  820 loss:  6.491812705993652 perplexity: 659.7181550386665
```

```
Iteration:  830 loss:  6.4318132400512695 perplexity: 621.2994923866887
Iteration:  840 loss:  6.4318132400512695 perplexity: 621.2994923866887
Iteration:  850 loss:  6.443813323974609 perplexity: 628.8000521021265
Iteration:  860 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  870 loss:  6.491813659667969 perplexity: 659.718784195227
Iteration:  880 loss:  6.491813659667969 perplexity: 659.718784195227
Iteration:  890 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  900 loss:  6.443813323974609 perplexity: 628.8000521021265
Iteration:  910 loss:  6.423813343048096 perplexity: 616.348988601647
Iteration:  920 loss:  6.42181396484375 perplexity: 615.1179049785542
Iteration:  930 loss:  6.451813697814941 perplexity: 633.8508648446759
Iteration:  940 loss:  6.4318132400512695 perplexity: 621.2994923866887
Iteration:  950 loss:  6.471813678741455 perplexity: 646.6554893576816
Iteration:  960 loss:  6.4918131828308105 perplexity: 659.7184696168717
Iteration:  970 loss:  6.491812705993652 perplexity: 659.7181550386665
Iteration:  980 loss:  6.473813056945801 perplexity: 647.9496916175866
Iteration:  990 loss:  6.4918131828308105 perplexity: 659.7184696168717
```

Here are the embeddings visualization

In [ ]:
```
lstmae.getVisualization_2(3)
```

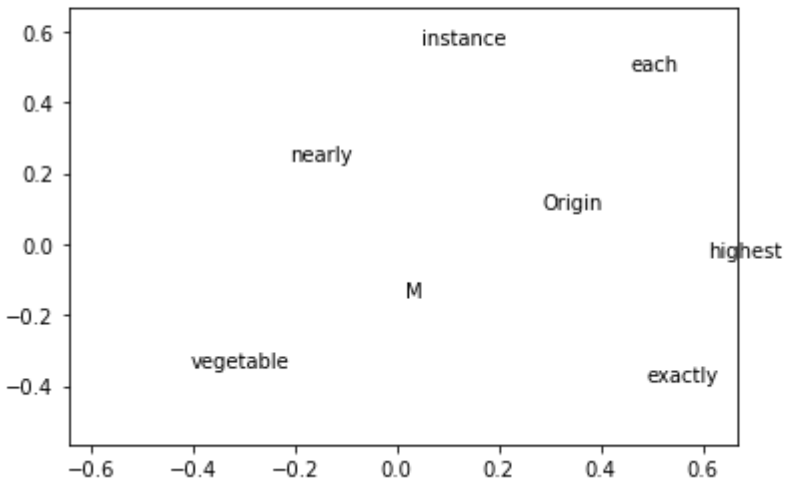


The embeddings seem to be clustered together on the basis of sequence length and the starting letter of the word. This could possibly be due to the position of the word in the one hot encoded vector of each of the words

The getVisualization() code is to just create my own set of sentences, train the LSTMae and and visualize the embeddings

In [ ]:
```
lstmae.getVisualization()
```

```
torch.Size([1, 1663, 64])
```



## Q4

1. The GCN model of Kipf and Welling scales linearly in the number of graph edges and learns hidden representations that encode both the local graph structure and features of nodes. They have addressed the problem of classifying nodes in a graph where only a small subset of nodes are labeled, known as graph based semi supervised classification. They have encoded the graph structure using a variation of a convolutional neural network and trained it on supervised targets for all nodes with the labels, avoiding explicit graph based regularization in the loss function. Their model uses and efficient layer wise propagation rule that is based on a first order approximation of spectral convolutions on graphs. A neural network model based on graph convolutions can therefore be built by stacking multiple convolutional layers. An issue with this could be that this approach is limited to undirected graphs, an equal importance to self connections and edges to neigboring nodes and large memory requirements for large datasets that may require additional approximations.

1. The Model with the BetterGCNConv was achieving an accuracy of about 68%, while the original GCNConv class was giving accuracies of about 80%

1. For the reddit data the BetterGCNConv gives about 71.1% accuracy. Our Originalconv Method gives an accuracy of only about 49.4%, worse than guessing. The OriginalGCN is unable to distinguish the graph structure here and thus is not able to generalize.