Bilkent University
EEE443 - Neural Networks
Mini Project Report
Fall 2022-2023

Name Surname - ID
Ahmet Faruk Ulutaş - 21803717

Instructor
Asst. Prof. Tolga Çukur

Teaching Assistant
Onat Dalmaz

Question 1.

a) For data management and visualization, a number of libraries are imported (h5py, math, matplotlib.pyplot, numpy, random). There are four specified assistance functions:

- Using the typical weights for each color channel, the rgb_to_grayscale(images) function transforms RGB photos to grayscale.
- With the help of the function normalize_data(data), the input data are adjusted to fall between 0.1 and 0.9.
- Using the function normalize_to_zero_one_range(pictures), the input images are normalized to fall between 0 and 1.
- In either grayscale or RGB, a selection of the photographs is shown via the function display_images(images, title, is_grayscale=True, n=200). The number of photos to display is specified by an optional parameter.

The h5py file "/content/drive/MyDrive/mini-project/data1.h5" is loaded, and the dataset is then transformed to a numpy array. The rgb_to_grayscale function is used to convert RGB data to grayscale. Each grayscale image's mean is determined, then it is subtracted from the corresponding image. The goal of this operation is to center the data around zero and eliminate the mean. The mean-removed data's standard deviation is calculated. The data that has had the mean removed is then trimmed to be three standard deviations or less from the mean. To eliminate outliers and lessen the impact of extreme numbers, this is done. Using the normalize_data function, the clipped grayscale data is normalized to fall between the values of 0.1 and 0.9. Using the normalize_to_zero_one_range function, the original RGB data is normalized to fall inside the range of 0 to 1. 200 indices are chosen at random from a variety of data sets. The script displays the chosen RGB and grayscale picture samples using the display_images function. While the grayscale photos are presented in their converted state, the RGB images are displayed without any grayscale conversion.

Comments on Results

The RGB photos display a broad range of hues and patterns due to their random sampling of different natural images. However, I couldn't see anything meaningful in the photos.

On the other hand, grayscale patterns are shorter and more pointless. Additionally, they are trimmed at 3 standard deviations and normalized, which makes them appropriate for input into a neural network. These photos may have somewhat less contrast than the original RGB images after the normalization and scaling procedure, however this is required to make sure the neural network performs at its best.

A typical pre-processing step in machine learning involves transforming RGB photos to grayscale and then normalizing them; it is anticipated that the grayscale images would have less visual information than the original RGB images. However, they still preserve the pictures' crucial structural data, which is what the neural network will use to train itself.


200 Random Sample Patches - RGB Format


200 Random Sample Patches - Normalized Grayscale

3.1

b,c,d) There are three defined functions:

- With the help of the function initialize_weights_and_biases(Lin, Lhid), a neural network with a hidden layer may have its weights and biases set up. The "Xavier initialization" approach is used to initialize the weights, and it divides the total number of input and hidden layer nodes by the square root of 6 to calculate the range of the uniform distribution. Biases start out as zeros.
- The network's activation function is called sigmoid(x). It accepts a value and returns the value's sigmoid.
- This function, aeCost(We, data, params), calculates the cost and gradients for an autoencoder. A neural network that attempts to reproduce its input at its output is called an autoencoder in this context.

Three elements make up the cost function: a weight decay term, a term for the Kullback-Leibler divergence that forces the hidden layer to remain sparse, and a quadratic term for the difference between the input and output. The gradient descent approach is used by the gradient_descent(data, params, max_iter=200, learning_rate=0.1) function to reduce the autoencoder's cost function. A single vector is created by concatenating the initial weights and biases. The aeCost function is used to compute the cost and gradients for each iteration, and the obtained gradients are used to update the weights. Each row in the data represents a sample, and each column represents a pixel, flattening the data into a two-dimensional array. The number of hidden layer nodes is set to 64, and the number of input nodes is set to the number of pixels in the flattened data. Lambda_, a weight decay parameter, is also set. The gradient descent approach is used to figure out the weights and biases of the autoencoder for each combination of the beta and rho parameters. Rho is the sparsity parameter, while beta is a parameter that regulates the weight of the sparsity penalty term. For each combination, the final weights and biases are determined. There is a specified function named display_features(W, title). The buried layer of the autoencoder's learnt features are shown with this function. Each row of the weight matrix is reconfigured into a 2D array, which is then shown as an image. The displayed grid's rows and columns are chosen such that they have a square or nearly square form. The final weights acquired by the gradient descent process are used to extract the weights for the first layer, which connects the input layer to the hidden layer. The display_features function is then used to show these weights. The pixel intensities of each picture correspond to the weights applied to each input when identifying the "feature" that the autoencoder has learnt to recognize. These characteristics can be viewed as the underlying patterns or structures that the autoencoder has discovered in the input data.

Comments on Results

b) For all possible combinations of and, we can see that the cost function typically falls with the number of iterations. This shows that the gradient descent approach is operating as intended, progressively adjusting the network's weights and biases to reduce the cost function.

The performance of various combinations of and, however, varies. The cost function's rate of decline during its iterative cycle and the value it reaches at the end are clear indicators of this.

The cost function declines the least quickly, in particular, when 0.1. The cost is still significantly high in comparison to other options even after 200 cycles. This shows that a lower value would not be the best choice for this specific activity because it can make learning more difficult.

We observe a significant improvement in performance when it is 0.5. It appears that a greater value causes learning to occur more quickly since the cost function decrements at a much faster pace. The fastest drop in cost among the runs with = 0.5 indicates that learning occurs when 0.2.

Finally, the outcomes differ when 1. The cost starts out quite high but rapidly drops off for = 0.05. The cost starts at a reasonable amount for = 0.1 and likewise falls off fast. When = 0.2, the cost starts out quite modest and declines more slowly than other values. This suggests that when it is 1, there may be a trade-off between the initial cost and the rate of learning.

For β = 0.1 and ρ = 0.05:
Iteration 0/200, cost: 3.8296094346782867
Iteration 10/200, cost: 2.233638607491719
Iteration 20/200, cost: 1.5592964285319368
Iteration 30/200, cost: 1.1925442913632014
Iteration 40/200, cost: 0.9699210327500595
Iteration 50/200, cost: 0.8251244834009179
Iteration 60/200, cost: 0.7260626288339715
Iteration 70/200, cost: 0.6555997599115397
Iteration 80/200, cost: 0.6039112473123661
Iteration 90/200, cost: 0.5650388520621534
Iteration 100/200, cost: 0.5352000126083161
Iteration 110/200, cost: 0.5119004876175886
Iteration 120/200, cost: 0.4934421833953021
Iteration 130/200, cost: 0.47863716002701234
Iteration 140/200, cost: 0.4666346993779102
Iteration 150/200, cost: 0.45681308268485893
Iteration 160/200, cost: 0.4487098052918495
Iteration 170/200, cost: 0.4419753775220418
Iteration 180/200, cost: 0.4363420199039012
Iteration 190/200, cost: 0.4316020067171414

For β = 0.1 and ρ = 0.1:
Iteration 0/200, cost: 3.3416821308071754
Iteration 10/200, cost: 1.8645878284950368
Iteration 20/200, cost: 1.262908801091014
Iteration 30/200, cost: 0.9457024131250669
Iteration 40/200, cost: 0.761527029675063
Iteration 50/200, cost: 0.6478423140973586
Iteration 60/200, cost: 0.574357289669285
Iteration 70/200, cost: 0.5251091533151
Iteration 80/200, cost: 0.4911345420598026
Iteration 90/200, cost: 0.4671350584193848
Iteration 100/200, cost: 0.44984346642411127
Iteration 110/200, cost: 0.43717303228421

Iteration 120/200, cost: 0.42775120896148366
Iteration 130/200, cost: 0.42065249691351525
Iteration 140/200, cost: 0.4152394967921812
Iteration 150/200, cost: 0.41106518571784145
Iteration 160/200, cost: 0.40781109865805804
Iteration 170/200, cost: 0.405247237084899
Iteration 180/200, cost: 0.40320549380629483
Iteration 190/200, cost: 0.4015616942376844

For β = 0.1 and ρ = 0.2:
Iteration 0/200, cost: 2.0531667421301774
Iteration 10/200, cost: 1.1282932116089779
Iteration 20/200, cost: 0.7834675551407349
Iteration 30/200, cost: 0.6144749732996344
Iteration 40/200, cost: 0.5227787059019761
Iteration 50/200, cost: 0.47038690414229983
Iteration 60/200, cost: 0.4393190182865594
Iteration 70/200, cost: 0.42031971533558626
Iteration 80/200, cost: 0.40837276506265574
Iteration 90/200, cost: 0.4006521514110091
Iteration 100/200, cost: 0.39551604763142756
Iteration 110/200, cost: 0.39198656438814444
Iteration 120/200, cost: 0.38946927082216604
Iteration 130/200, cost: 0.3875966816634771
Iteration 140/200, cost: 0.38613837412530494
Iteration 150/200, cost: 0.38494811382289634
Iteration 160/200, cost: 0.38393211803953936
Iteration 170/200, cost: 0.3830296801288295
Iteration 180/200, cost: 0.38220116918020214
Iteration 190/200, cost: 0.3814205058157185

For β = 0.5 and ρ = 0.05:
Iteration 0/200, cost: 17.482059206287087
Iteration 10/200, cost: 2.4612839857708697
Iteration 20/200, cost: 1.0279328125776352
Iteration 30/200, cost: 0.6575572057332331
Iteration 40/200, cost: 0.5241888415221618
Iteration 50/200, cost: 0.46769328918150804
Iteration 60/200, cost: 0.4411996962832057
Iteration 70/200, cost: 0.4277696116835182
Iteration 80/200, cost: 0.42046012838585345
Iteration 90/200, cost: 0.4161821409479655
Iteration 100/200, cost: 0.41348044275662066
Iteration 110/200, cost: 0.4116395369845877
Iteration 120/200, cost: 0.41029491249678157
Iteration 130/200, cost: 0.4092545263591472
Iteration 140/200, cost: 0.40841346509726906
Iteration 150/200, cost: 0.4077119105459707

Iteration 160/200, cost: 0.40711395327693795
Iteration 170/200, cost: 0.4065967109431407
Iteration 180/200, cost: 0.4061446434843669
Iteration 190/200, cost: 0.4057465306294434

For β = 0.5 and ρ = 0.1:
Iteration 0/200, cost: 14.139804415445353
Iteration 10/200, cost: 1.5716906113949374
Iteration 20/200, cost: 0.6375134095338324
Iteration 30/200, cost: 0.46912536119210646
Iteration 40/200, cost: 0.4275057715553712
Iteration 50/200, cost: 0.4148812751799822
Iteration 60/200, cost: 0.41002448077273224
Iteration 70/200, cost: 0.4075280489778983
Iteration 80/200, cost: 0.4058802298788961
Iteration 90/200, cost: 0.4046210191950512
Iteration 100/200, cost: 0.4035914346195544
Iteration 110/200, cost: 0.4027237993040512
Iteration 120/200, cost: 0.4019809130964709
Iteration 130/200, cost: 0.401337689751658
Iteration 140/200, cost: 0.40077520460736044
Iteration 150/200, cost: 0.4002784696023187
Iteration 160/200, cost: 0.3998353877533647
Iteration 170/200, cost: 0.3994361256532648
Iteration 180/200, cost: 0.39907266772541533
Iteration 190/200, cost: 0.3987384721857696

For β = 0.5 and ρ = 0.2:
Iteration 0/200, cost: 8.89633001369869
Iteration 10/200, cost: 0.8560169040220151
Iteration 20/200, cost: 0.47150904664866183
Iteration 30/200, cost: 0.4281976470660043
Iteration 40/200, cost: 0.4165577058671923
Iteration 50/200, cost: 0.41042921679720656
Iteration 60/200, cost: 0.40643417456405506
Iteration 70/200, cost: 0.40368108656708307
Iteration 80/200, cost: 0.4017194001289714
Iteration 90/200, cost: 0.40026884449143824
Iteration 100/200, cost: 0.39914900576020407
Iteration 110/200, cost: 0.39824338113540686
Iteration 120/200, cost: 0.39747674319841314
Iteration 130/200, cost: 0.39680048946232965
Iteration 140/200, cost: 0.396183175360042
Iteration 150/200, cost: 0.3956044137816839
Iteration 160/200, cost: 0.3950509483477502
Iteration 170/200, cost: 0.3945141255937962
Iteration 180/200, cost: 0.3939882664810277
Iteration 190/200, cost: 0.3934696162026385

For β = 1 and ρ = 0.05:
Iteration 0/200, cost: 33.545324789489214
Iteration 10/200, cost: 1.505329250805224
Iteration 20/200, cost: 0.6106166440616885
Iteration 30/200, cost: 0.4676367486362191
Iteration 40/200, cost: 0.4342340050997027
Iteration 50/200, cost: 0.4241067896503914
Iteration 60/200, cost: 0.41986287972775915
Iteration 70/200, cost: 0.4173508504038423
Iteration 80/200, cost: 0.41548088767169344
Iteration 90/200, cost: 0.41393924792579817
Iteration 100/200, cost: 0.41262012199243087
Iteration 110/200, cost: 0.41147661388733164
Iteration 120/200, cost: 0.41048033734120193
Iteration 130/200, cost: 0.40961008802454657
Iteration 140/200, cost: 0.4088484550192477
Iteration 150/200, cost: 0.4081806433359088
Iteration 160/200, cost: 0.4075939367223026
Iteration 170/200, cost: 0.4070773651996471
Iteration 180/200, cost: 0.4066214538118453
Iteration 190/200, cost: 0.4062180153480401

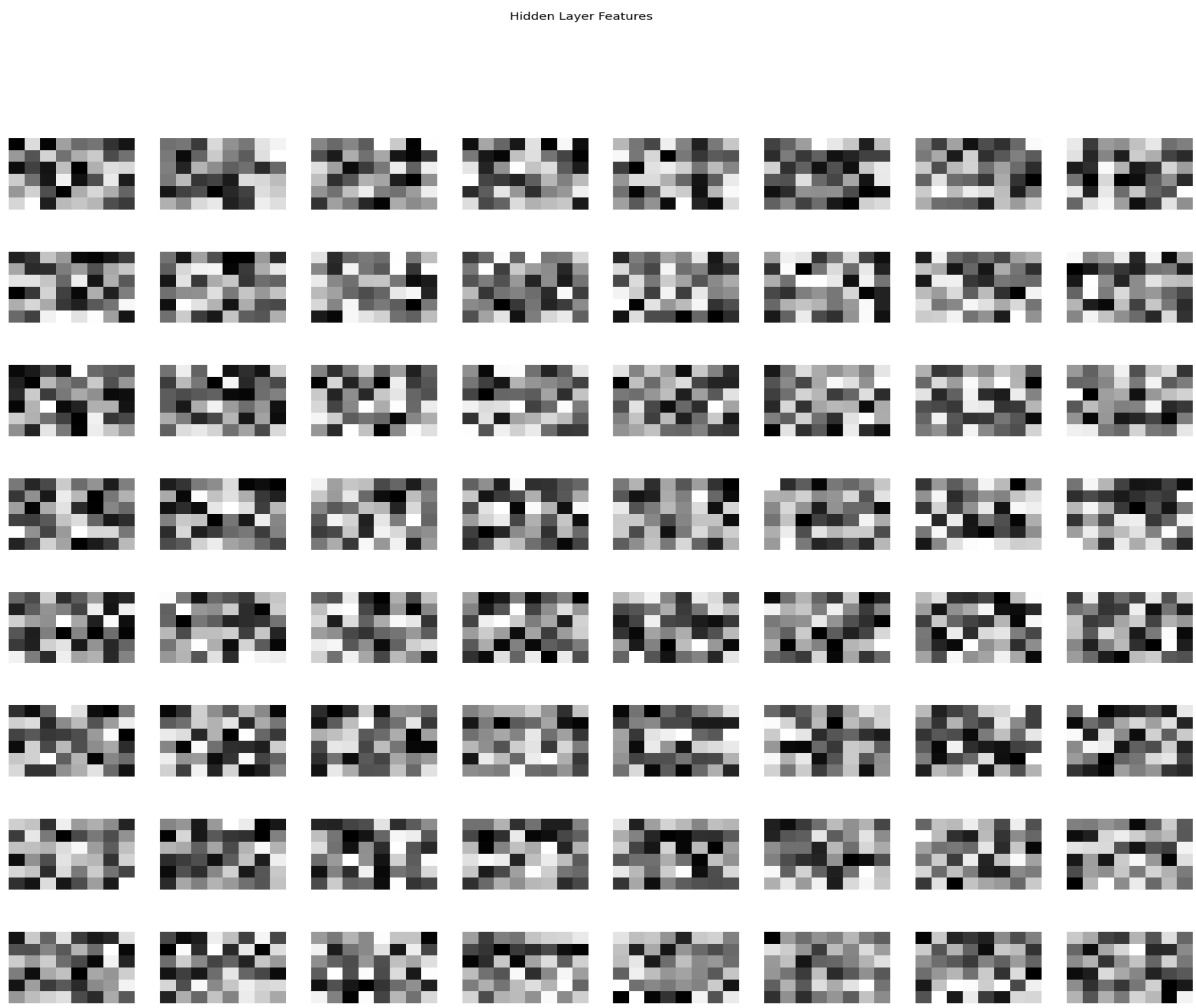
For β = 1 and ρ = 0.1:
Iteration 0/200, cost: 28.179752103868626
Iteration 10/200, cost: 0.8013951813551039
Iteration 20/200, cost: 0.4552021330899315
Iteration 30/200, cost: 0.4288477384338485
Iteration 40/200, cost: 0.42240063960529944
Iteration 50/200, cost: 0.4182392359797669
Iteration 60/200, cost: 0.4149018072004527
Iteration 70/200, cost: 0.4121546263705743
Iteration 80/200, cost: 0.40988246153351043
Iteration 90/200, cost: 0.40799705421339005
Iteration 100/200, cost: 0.40642674170626675
Iteration 110/200, cost: 0.4051130698306228
Iteration 120/200, cost: 0.40400838241656184
Iteration 130/200, cost: 0.4030738532444136
Iteration 140/200, cost: 0.4022778649457668
Iteration 150/200, cost: 0.401594677736326
Iteration 160/200, cost: 0.4010033394890209
Iteration 170/200, cost: 0.40048679543371535
Iteration 180/200, cost: 0.4000311621256612
Iteration 190/200, cost: 0.39962513607085637

For β = 1 and ρ = 0.2:
Iteration 0/200, cost: 12.960411203638959
Iteration 10/200, cost: 0.4726697386826236

Iteration 20/200, cost: 0.4206516532476602
Iteration 30/200, cost: 0.41243380083912334
Iteration 40/200, cost: 0.4071683770815875
Iteration 50/200, cost: 0.40360253070912444
Iteration 60/200, cost: 0.4011314745974828
Iteration 70/200, cost: 0.3993649806071342
Iteration 80/200, cost: 0.39805135551481435
Iteration 90/200, cost: 0.3970286720885329
Iteration 100/200, cost: 0.396192871592482
Iteration 110/200, cost: 0.3954770961203594
Iteration 120/200, cost: 0.3948383695972728
Iteration 130/200, cost: 0.39424903925619154
Iteration 140/200, cost: 0.39369128372709794
Iteration 150/200, cost: 0.3931535907647697
Iteration 160/200, cost: 0.3926284982891182
Iteration 170/200, cost: 0.3921111453094993
Iteration 180/200, cost: 0.39159834205487376
Iteration 190/200, cost: 0.3910879730495996

c) The hidden layer's features are displayed in a range of tones from white to black, showing the network's ability to distinguish between different intensities in the input data. The depiction of these aspects, however, does not resemble any normal natural image component in any appreciable way. There are no obvious patterns, features, or structures that could be related to elements of these photos that could be recognized, such as edges, corners, or certain textures.

Although the learnt features are abstract, it's vital to keep in mind that human capacity to visually understand these traits does not always correspond to the network's ability to accomplish its duty efficiently. Even if the visualizations don't accurately depict the elements of real pictures, the network may be picking up useful, albeit abstract, representations that help it with its goal.

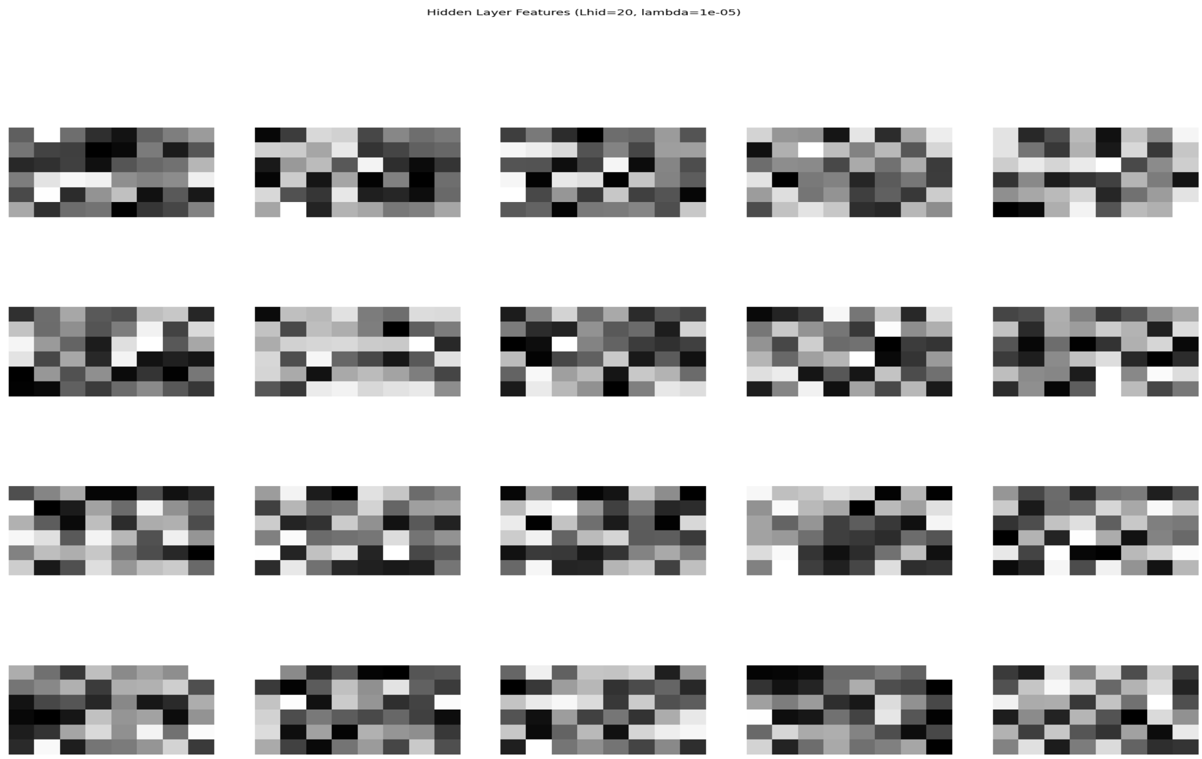


d) First off, the model's initial cost grows as the Lhid parameter, which determines the hidden layer's size, increases. This shows that the model's complexity is increasing and that additional parameters need to be optimized. This circumstance also necessitates extra computing time and resources. However, because more hidden neurons can better capture the properties of the data, this increase in complexity might improve the model's performance. This is especially evident in the cost reduction between Lhid=50 and Lhid=80. Though the danger of overfitting rises along with the model's complexity, this can be detrimental to the model's capacity to generalize.
To avoid overfitting, the Lambda parameter serves as a regularization term. The complexity of the model is constrained as the lambda value rises, preventing overfitting. The initial cost lowers as Lambda rises, as seen by the results for Lambda=1e-5, Lambda=1e-4, and Lambda=1e-3. This suggests that larger Lambda values restrict the model's complexity and speed up the process of reaching the optimal value. A highly high Lambda value, however, can prohibit the model from understanding the dataset's complicated properties, resulting in a decline in model performance.

In conclusion, testing the model's performance on the validation set and experimenting with different parameter combinations are frequently the best ways to identify the ideal training parameters. By doing this, you can be confident that the model retains its capacity to generalize to new data while still learning the training data well. Higher Lhid and lower

Lambda values in this situation seem to typically offer greater performance, however it's vital to take the danger of overfitting into account. This danger is managed using the lambda regularization option.

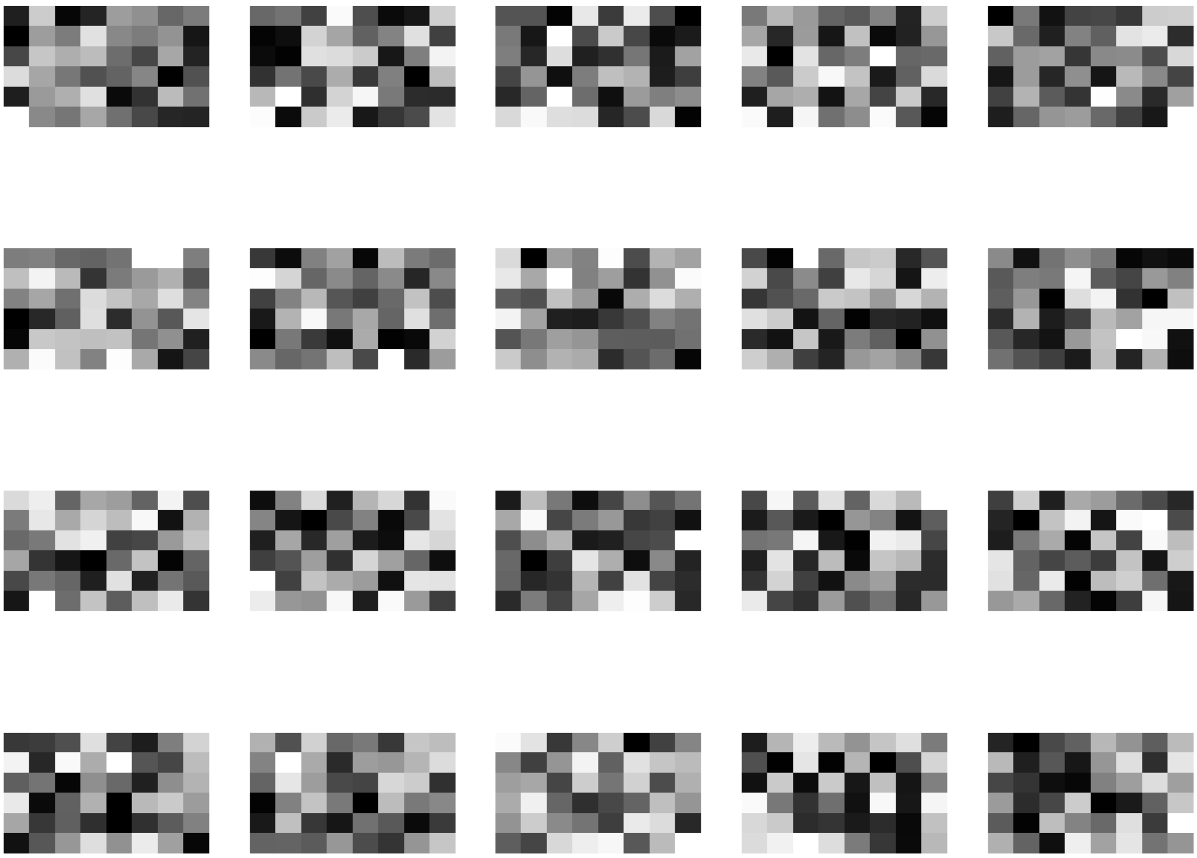
Hidden Layer Features (Lhid = 20, lambda=1e-05)
Iteration 0/200, cost: 1.4880975308989715
Iteration 10/200, cost: 0.943611052793218
Iteration 20/200, cost: 0.7131228904244059
Iteration 30/200, cost: 0.5984369805273756
Iteration 40/200, cost: 0.5328591637126263
Iteration 50/200, cost: 0.4915089156524807
Iteration 60/200, cost: 0.4635981460951479
Iteration 70/200, cost: 0.4438058890498421
Iteration 80/200, cost: 0.42923855168516134
Iteration 90/200, cost: 0.4182015191604664
Iteration 100/200, cost: 0.40964337390615885
Iteration 110/200, cost: 0.4028809431546497
Iteration 120/200, cost: 0.3974532361816003
Iteration 130/200, cost: 0.39303923140813873
Iteration 140/200, cost: 0.38940933372907394
Iteration 150/200, cost: 0.38639553497919105
Iteration 160/200, cost: 0.38387243007593147
Iteration 170/200, cost: 0.38174477541524415
Iteration 180/200, cost: 0.379939120510197
Iteration 190/200, cost: 0.3783980489631394



Hidden Layer Features (Lhid = 20, lambda=0.0001)

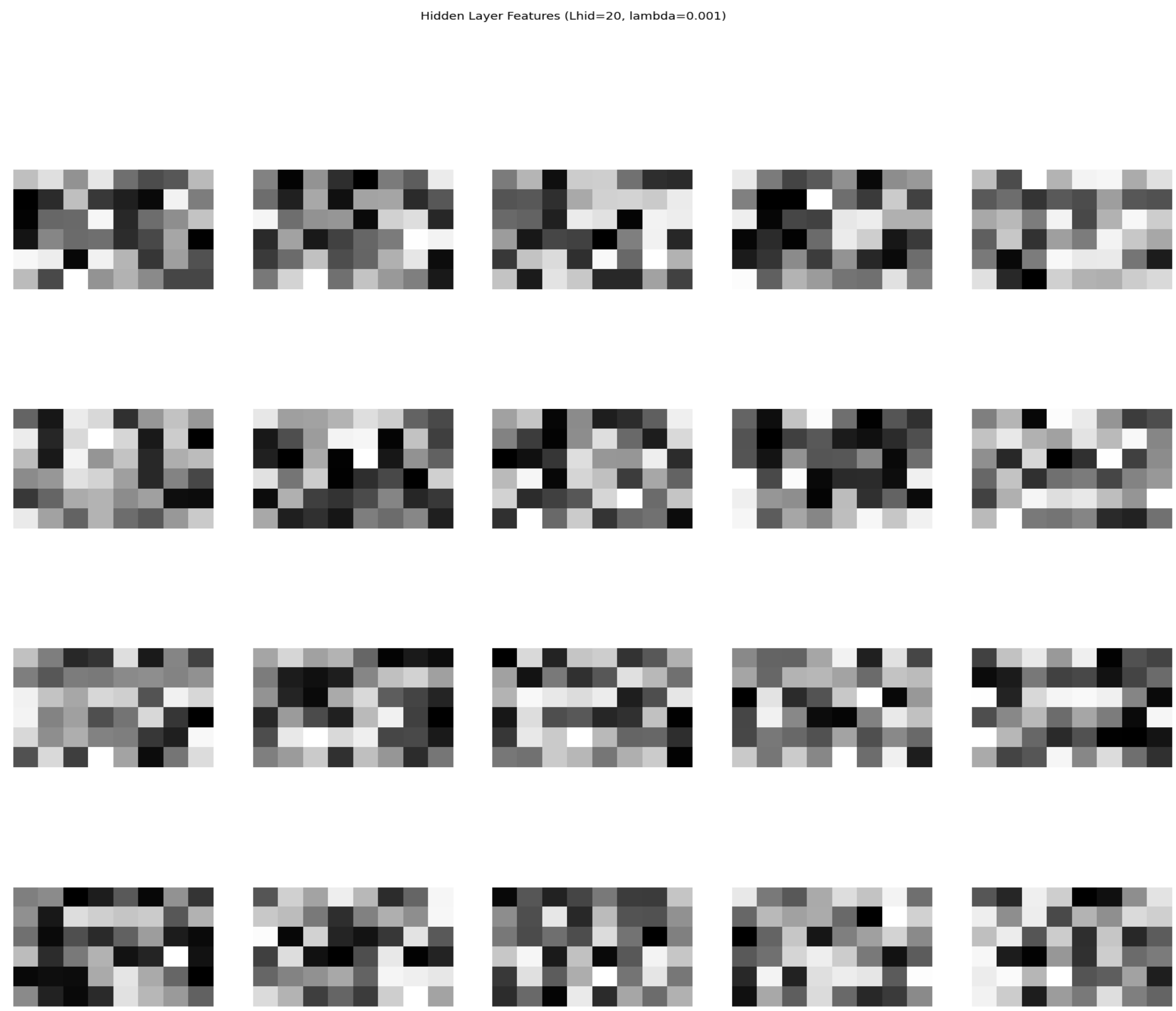
Iteration 0/200, cost: 1.4001733546696538
Iteration 10/200, cost: 0.9247851516118122
Iteration 20/200, cost: 0.7106312795102065
Iteration 30/200, cost: 0.6009838266493106
Iteration 40/200, cost: 0.5374216488975635
Iteration 50/200, cost: 0.49700424110639485
Iteration 60/200, cost: 0.46954581346748886
Iteration 70/200, cost: 0.44995979159523514
Iteration 80/200, cost: 0.43545967612658465
Iteration 90/200, cost: 0.4244060662099607
Iteration 100/200, cost: 0.41577885713006674
Iteration 110/200, cost: 0.40891402197934723
Iteration 120/200, cost: 0.4033629715561303
Iteration 130/200, cost: 0.398813063201117
Iteration 140/200, cost: 0.39504052848996263
Iteration 150/200, cost: 0.3918814826831054
Iteration 160/200, cost: 0.3892134612433189
Iteration 170/200, cost: 0.38694331436553786
Iteration 180/200, cost: 0.3849990656694981
Iteration 190/200, cost: 0.3833243123200788

Hidden Layer Features (Lhid=20, lambda=0.0001)



Hidden Layer Features (Lhid = 20, lambda=0.001)

Iteration 0/200, cost: 1.8443952143802558
Iteration 10/200, cost: 1.1328960544559004
Iteration 20/200, cost: 0.8286261933114731
Iteration 30/200, cost: 0.6804267790266123
Iteration 40/200, cost: 0.5976566525441729
Iteration 50/200, cost: 0.5464938826255079
Iteration 60/200, cost: 0.5125115790561408
Iteration 70/200, cost: 0.48872669949927683
Iteration 80/200, cost: 0.471407128488841
Iteration 90/200, cost: 0.4584002531149526
Iteration 100/200, cost: 0.4483881872417091
Iteration 110/200, cost: 0.44052461286375966
Iteration 120/200, cost: 0.43424437529839094
Iteration 130/200, cost: 0.4291575695627094
Iteration 140/200, cost: 0.4249876659888395
Iteration 150/200, cost: 0.4215338416062049
Iteration 160/200, cost: 0.4186472264284792
Iteration 170/200, cost: 0.4162154619841397
Iteration 180/200, cost: 0.41415239313584346
Iteration 190/200, cost: 0.41239102355319973

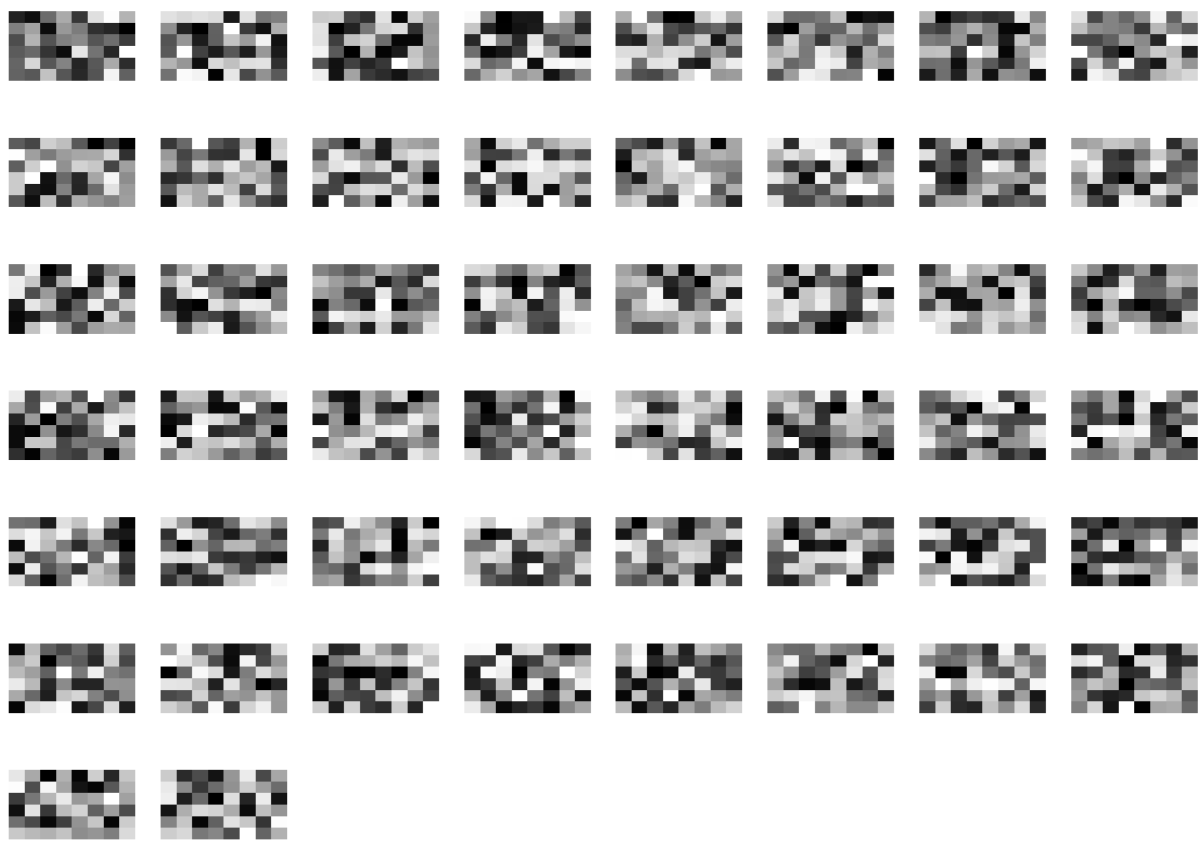


Hidden Layer Features (Lhid = 50, lambda=1e-05)

Iteration 0/200, cost: 3.508602003019861
Iteration 10/200, cost: 1.9822357433033466
Iteration 20/200, cost: 1.3650881696766586
Iteration 30/200, cost: 1.0407965668441415
Iteration 40/200, cost: 0.847958506419402
Iteration 50/200, cost: 0.724173262262043
Iteration 60/200, cost: 0.6402273998372205
Iteration 70/200, cost: 0.5808697939941039
Iteration 80/200, cost: 0.5374977620178052
Iteration 90/200, cost: 0.5049585690821494
Iteration 100/200, cost: 0.4800128926924247
Iteration 110/200, cost: 0.4605414409733565
Iteration 120/200, cost: 0.44511057262103393
Iteration 130/200, cost: 0.4327225544618944
Iteration 140/200, cost: 0.4226658903322917
Iteration 150/200, cost: 0.41442236555975337
Iteration 160/200, cost: 0.4076074991171765
Iteration 170/200, cost: 0.4019313473071975
Iteration 180/200, cost: 0.3971720775490463
Iteration 190/200, cost: 0.39315776788456125

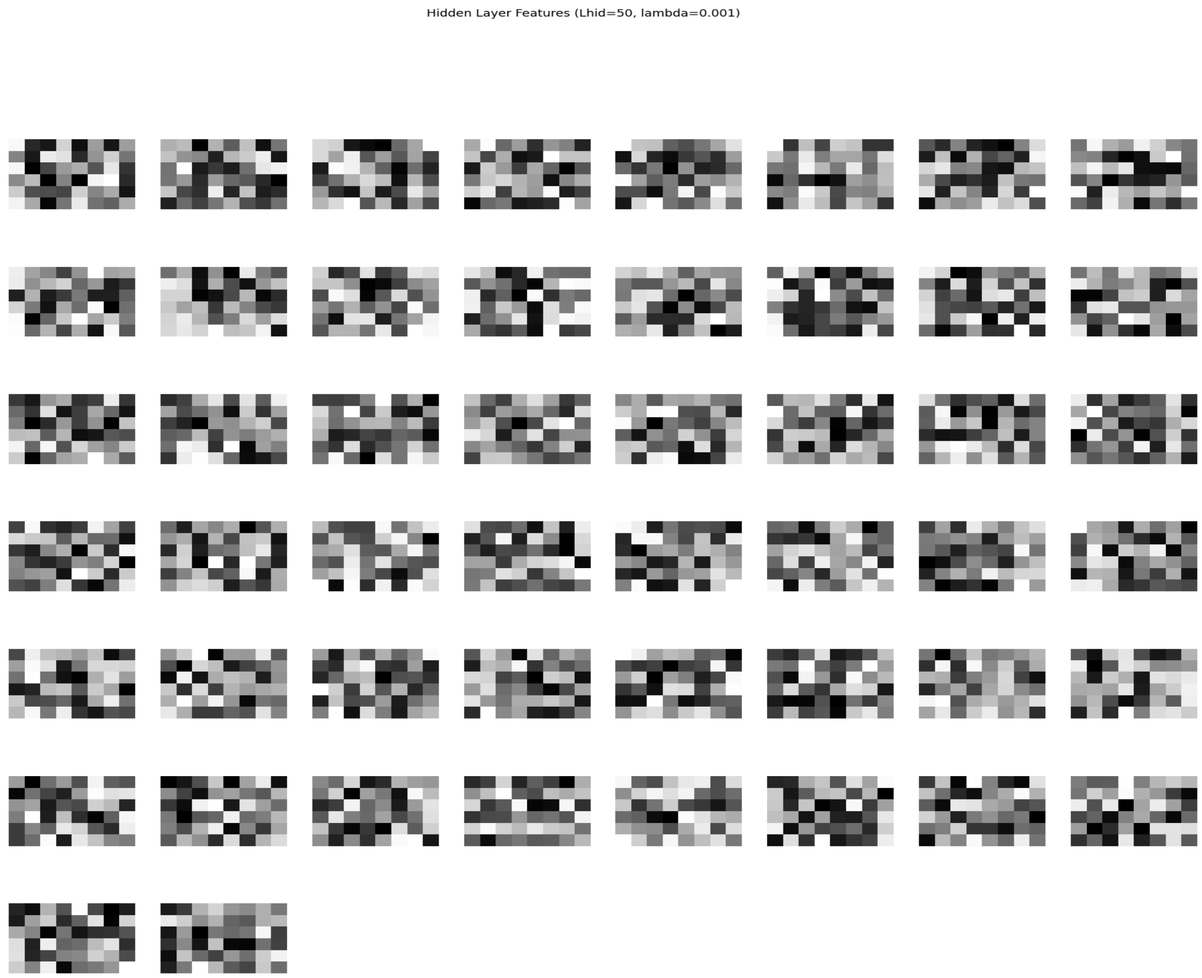
Hidden Layer Features (Lhid = 50, lambda=0.0001)
Iteration 0/200, cost: 3.708438072821951
Iteration 10/200, cost: 2.0531941497077355
Iteration 20/200, cost: 1.4003525797898235
Iteration 30/200, cost: 1.0621890047898541
Iteration 40/200, cost: 0.8628791253744617
Iteration 50/200, cost: 0.7357849443273747
Iteration 60/200, cost: 0.6500348675663992
Iteration 70/200, cost: 0.5896411881338939
Iteration 80/200, cost: 0.5456471866619206
Iteration 90/200, cost: 0.5127193649239482
Iteration 100/200, cost: 0.48752144301628375
Iteration 110/200, cost: 0.46787995177902614
Iteration 120/200, cost: 0.4523299257321893
Iteration 130/200, cost: 0.4398549955963722
Iteration 140/200, cost: 0.4297323083672227
Iteration 150/200, cost: 0.42143658685573804
Iteration 160/200, cost: 0.41457888168891693
Iteration 170/200, cost: 0.40886637976177403
Iteration 180/200, cost: 0.4040753814720688
Iteration 190/200, cost: 0.4000327352552474

Hidden Layer Features (Lhid=50, lambda=0.0001)



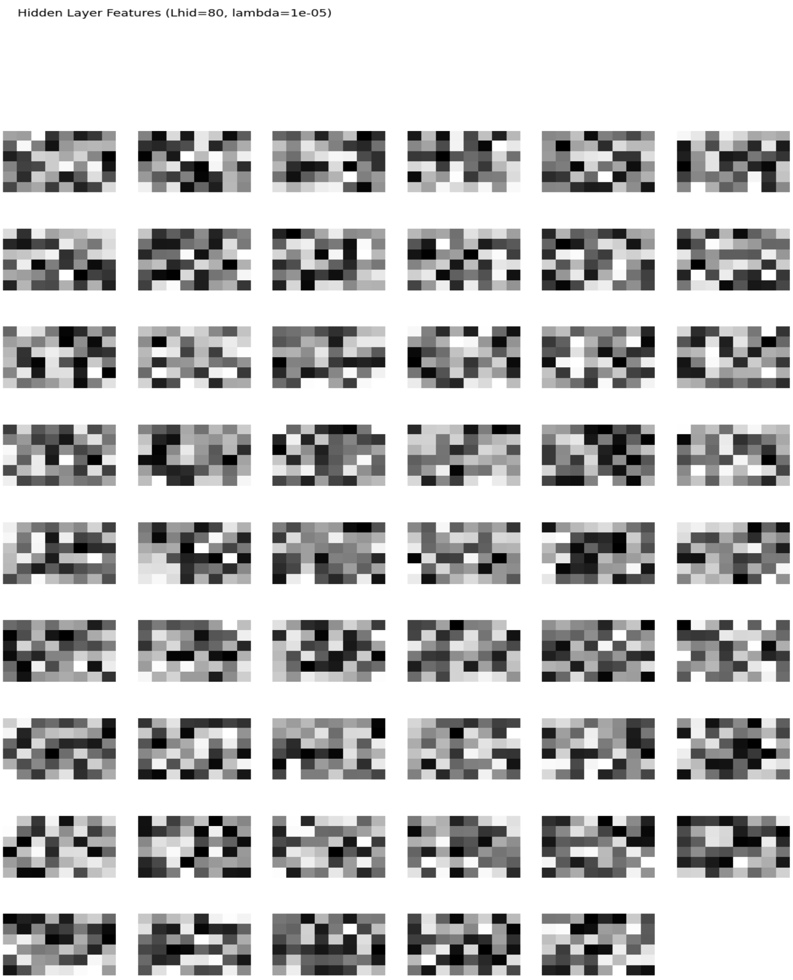
Hidden Layer Features (Lhid = 50, lambda=0.001)

Iteration 0/200, cost: 3.5248837007379663
Iteration 10/200, cost: 2.035325956600947
Iteration 20/200, cost: 1.4181283109936118
Iteration 30/200, cost: 1.09363802391564
Iteration 40/200, cost: 0.901430948097
Iteration 50/200, cost: 0.7785517469793224
Iteration 60/200, cost: 0.6955234294710846
Iteration 70/200, cost: 0.6370017502239494
Iteration 80/200, cost: 0.5943600804301069
Iteration 90/200, cost: 0.5624474275199407
Iteration 100/200, cost: 0.5380347157727561
Iteration 110/200, cost: 0.5190147807416492
Iteration 120/200, cost: 0.5039655488967036
Iteration 130/200, cost: 0.49189951497714374
Iteration 140/200, cost: 0.48211394006700725
Iteration 150/200, cost: 0.4740980041927877
Iteration 160/200, cost: 0.46747343858372736
Iteration 170/200, cost: 0.4619555153341461
Iteration 180/200, cost: 0.4573267890587397
Iteration 190/200, cost: 0.45341903963765956



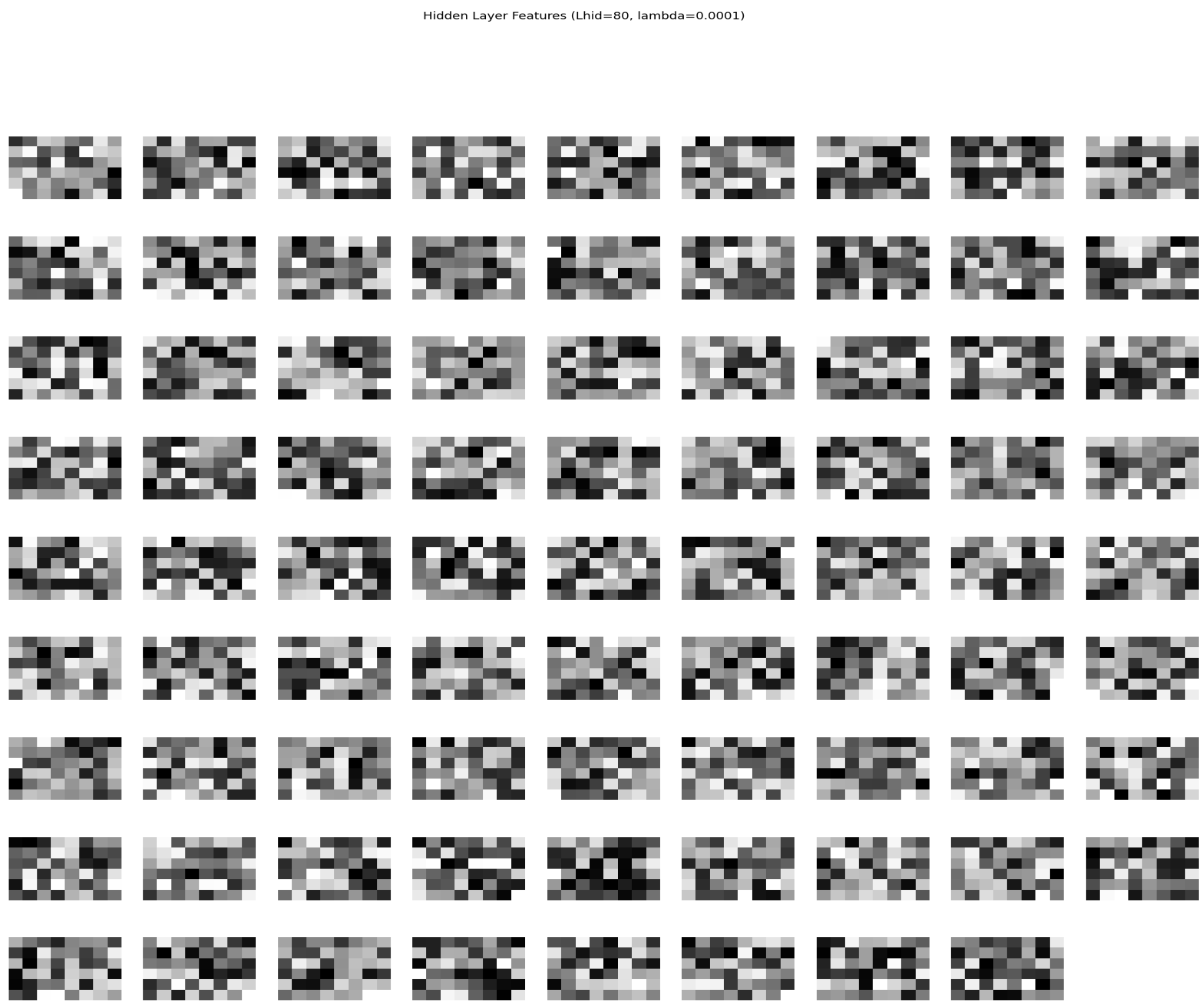
Hidden Layer Features (Lhid = 80, lambda=1e-05)

Iteration 0/200, cost: 5.456909551206997
Iteration 10/200, cost: 3.007481899491826
Iteration 20/200, cost: 2.018255275211757
Iteration 30/200, cost: 1.4817903579296927
Iteration 40/200, cost: 1.1595983495643745
Iteration 50/200, cost: 0.952536272415394
Iteration 60/200, cost: 0.8123968968285522
Iteration 70/200, cost: 0.7136270133166587
Iteration 80/200, cost: 0.6417310174034649
Iteration 90/200, cost: 0.5880125742432651
Iteration 100/200, cost: 0.5470057838663984
Iteration 110/200, cost: 0.5151384463675668
Iteration 120/200, cost: 0.48999768634070906
Iteration 130/200, cost: 0.46990720237762973
Iteration 140/200, cost: 0.45367387715836227
Iteration 150/200, cost: 0.4404304753881304
Iteration 160/200, cost: 0.4295349499620755
Iteration 170/200, cost: 0.42050422136018556
Iteration 180/200, cost: 0.41296957503591264
Iteration 190/200, cost: 0.40664597345178644



Hidden Layer Features (Lhid = 80, lambda=0.0001)

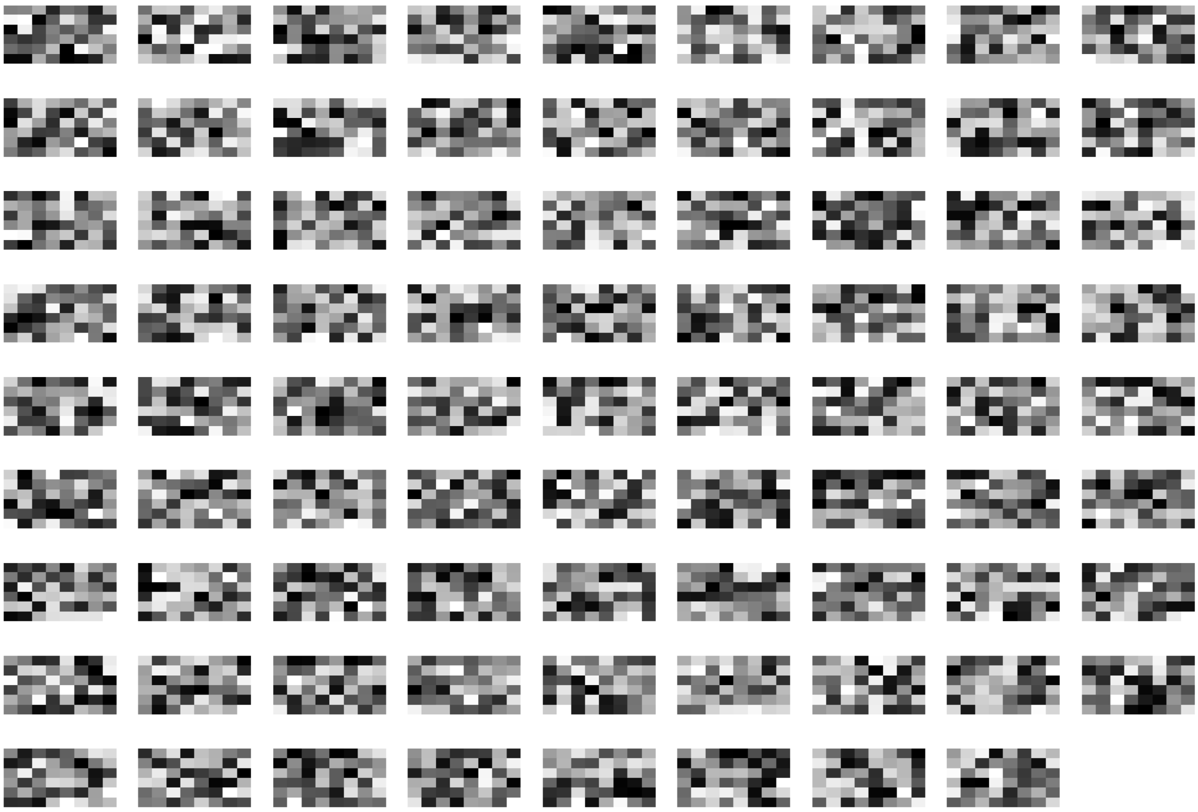
Iteration 0/200, cost: 5.094494797469144
Iteration 10/200, cost: 2.827861788227721
Iteration 20/200, cost: 1.9202967579179502
Iteration 30/200, cost: 1.425587157626858
Iteration 40/200, cost: 1.1258709466783638
Iteration 50/200, cost: 0.9317516559986834
Iteration 60/200, cost: 0.799539455315551
Iteration 70/200, cost: 0.7058842994980695
Iteration 80/200, cost: 0.6374342235250681
Iteration 90/200, cost: 0.5861225227173785
Iteration 100/200, cost: 0.5468483410234758
Iteration 110/200, cost: 0.5162607701208481
Iteration 120/200, cost: 0.4920863262888315
Iteration 130/200, cost: 0.47273946938382183
Iteration 140/200, cost: 0.45708790889872636
Iteration 150/200, cost: 0.44430622063108305
Iteration 160/200, cost: 0.43378174949313186
Iteration 170/200, cost: 0.42505249560677566
Iteration 180/200, cost: 0.417765137288631
Iteration 190/200, cost: 0.4116460621451797



Hidden Layer Features (Lhid = 80, lambda=0.001)

Iteration 0/200, cost: 5.187240606523807
Iteration 10/200, cost: 2.885489579137908
Iteration 20/200, cost: 1.9776408264445728
Iteration 30/200, cost: 1.4852458857300723
Iteration 40/200, cost: 1.1876299824918692
Iteration 50/200, cost: 0.9952340422107953
Iteration 60/200, cost: 0.8644390893160525
Iteration 70/200, cost: 0.7719651107725736
Iteration 80/200, cost: 0.7045105526435724
Iteration 90/200, cost: 0.6540444482396341
Iteration 100/200, cost: 0.61549214386897
Iteration 110/200, cost: 0.5855226882498198
Iteration 120/200, cost: 0.5618782914592123
Iteration 130/200, cost: 0.5429860614398387
Iteration 140/200, cost: 0.5277240910006183
Iteration 150/200, cost: 0.5152755998995899
Iteration 160/200, cost: 0.5050352071486058
Iteration 170/200, cost: 0.4965470869417961
Iteration 180/200, cost: 0.4894631948461482
Iteration 190/200, cost: 0.48351445572772433

Hidden Layer Features (Lhid=80, lambda=0.001)



Question 2

- **Initialize_weights**: Random values chosen at random from a normal distribution are used to initialize the network's weights and biases.
- Applying the softmax activation function, **stable_softmax** normalizes the output of the network into probabilities. It frequently appears in multi-class classification issues.
- Calculates the network's forward pass using the **forward_pass** function. It spreads through each layer while doing activation operations.
- **Backpropagation**: Calculates the network's backpropagation. In order to update the weights and biases, it computes the difference between the output of the network and the target.
- Network training with **train_network**. It updates the weights and biases by going back and forth across the training data for a predetermined number of epochs. Additionally, it keeps track of the training and validation results and halts training when the latter grows.

Here, I could not test for all scenarios as the model took more than a few hours to run. However, the outputs I get when D=32, P=256 are as follows. Here I made a top10 listing for randomly selected trigrams. It did not produce significant results. For example, in the first line, the trigram suggested the highest "and" even though it ended with "and". On the other hand, it is partially meaningful that he makes the most "house" proposition after "including" in the 3rd line. Words such as "be", "on", "after" that come after "team" in line 4 are also partially meaningful.

a,b) D, P = 32, 256

[b'were', b'we', b'and']
[b'and', b'well', b'part', b'in', b'these', b'then', b'would', b'those', b'family', b'-']

[b'not', b'year', b'or']
[b'each', b'his', b'well', b'--', b'times', b'on', b"'s", b'united', b'new', b'any']

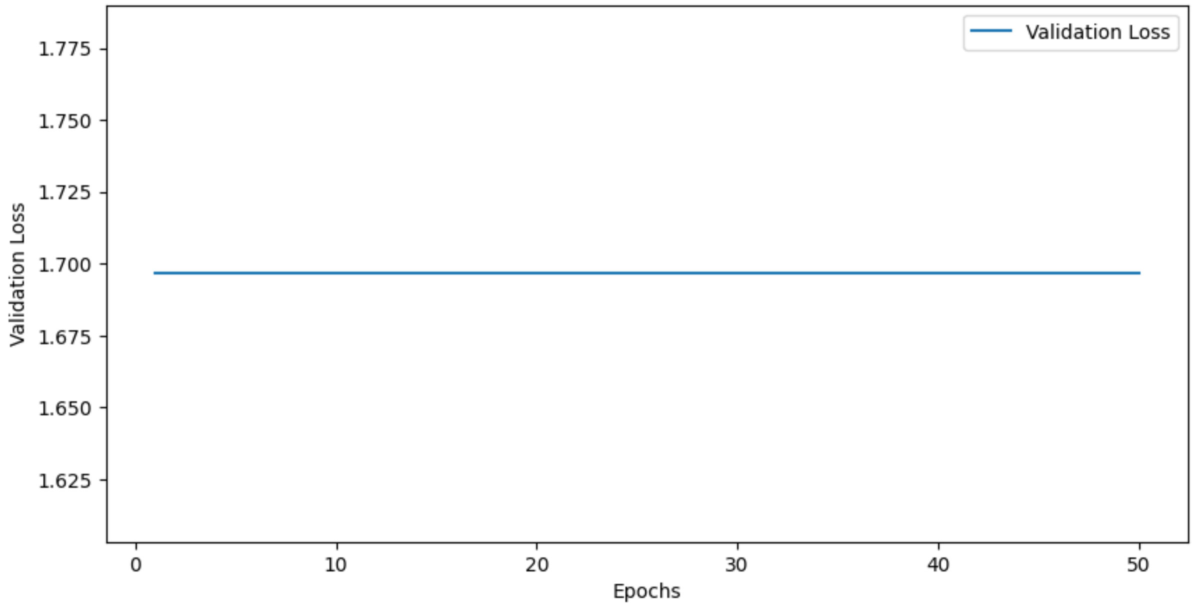
[b'year', b'or', b'including']
[b'house', b'play', b'case', b'former', b'out', b'to', b'university', b'under', b'going', b'each']

[b'or', b'including', b'team']
[b'be', b'on', b'after', b'your', b'now', b'-', b'like', b'general', b'team', b'every']

[b'including', b'team', b'for']
[b'be', b'your', b'officials', b'years', b'several', b'former', b'members', b'police', b'if', b'united']

Question 3

a) According to the examined measures, the model performs rather poorly. As long as the validation error is persistently large, the model's ability to perform better with each passing epoch cannot be assumed. At 16.67%, the test's accuracy is likewise quite low. The confusion matrices for the training and test data demonstrate that the model is unable to correctly predict any of the classes. These findings imply that the model has not generalized well and has not learned from the training data.



The model is not doing well in terms of classification, as seen by the high validation error and poor test accuracy. It struggles to distinguish between several classes and is unable to produce accurate predictions. The fact that the validation loss is constant throughout all epochs shows that the model is not adapting to the training input and is not learning from it. This lack of progress and poor performance can be ascribed to a number of things, including a model design that is insufficient and insufficient data.

b) I got exactly the same results as the RNN in option A. However, LSTM's epochs ran much slower. That's why the test took longer.

Index of comments

3.1 figures are not clear -2

8.1 you can just put the figures of cost is decreasing instead of writing all costs. Try to visualize your results -2

20.1 no comparision of d and p values -10