

Algorithms:

[1] - fashionMNIST_fcAE_training.py

We used the provided autoencoder code on Canvas as our starting point for testing different model training parameters. While evaluating different model training schemes, we would examine one variable (e.g. learning rule or loss function) at a time and hold all others equal.

Architecture

Since it was mentioned to not worry about the size of the machine, we largely left the number of layers and dimension sizes alone. The existing architecture contained three fully connected layers for the encoder and decoder, respectively. The dimensionality through the encoder started at 784 (corresponding to our 28 x 28 fashionMNIST images), then halved for the second layer input, and halved again for the third layer input. The third layer projected the image to the latent space cardinality. Then, the decoder reversed this process, resulting in an output dimension that matched the input of 784.

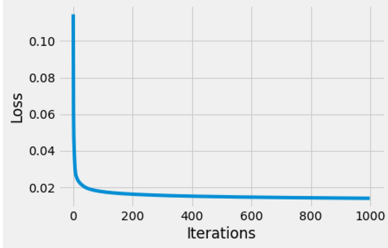
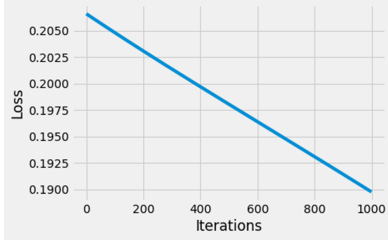
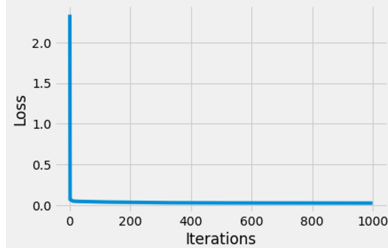
Overall, a larger machine would have better changes of fitting to training data; however, it'd also be at a higher risk of overfitting to training data (we can consider the extreme case where the machine is as large as the training set: the model would use its weights for compression rather than the hidden layer, resulting in good results for the training set but poor generalization to the test set)

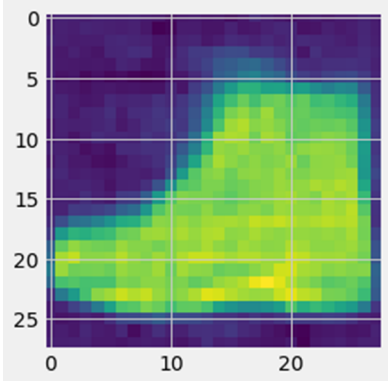
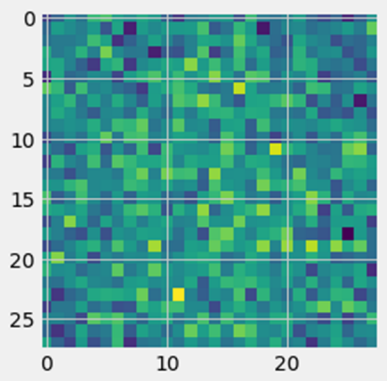
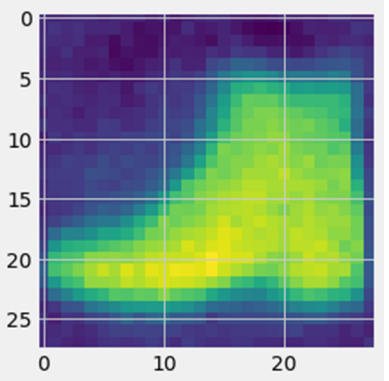
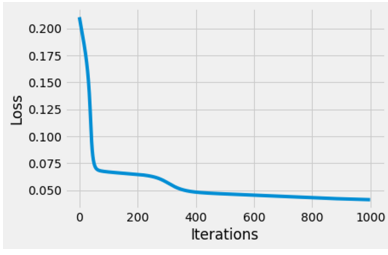
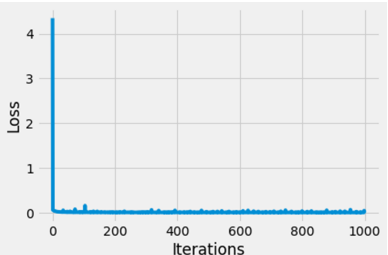
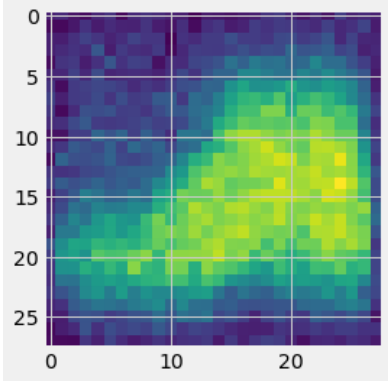
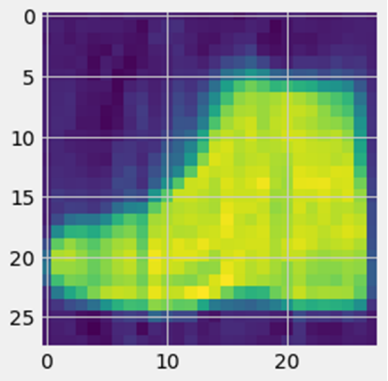
Learning rules

We tested the following optimization strategies:

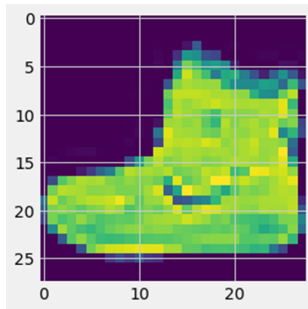
1. Adam (original)
2. Stochastic gradient descent (SGD)
3. Adagrad
4. Adadelata
5. RMSprop

Results:

| Adam NRMSE train: [0.11268099] NRMSE test: [0.11385359] | Stochastic gradient descent NRMSE train: [0.39528444] NRMSE test: [0.39538896] | Adagrad NRMSE train: [0.14601283] NRMSE test: [0.14604554] |
|--|---|--|
| MSE over training epochs:  | MSE over training epochs:  | MSE over training epochs:  |

| | | |
|---|--|--|
| <p>Example model output:</p>  | <p>Example model output:</p>  | <p>Example model output:</p>  |
| <p>Adadelata</p> <p>NRMSE train: [0.19134974]</p> <p>NRMSE test: [0.19091569]</p> | <p>RMSprop</p> <p>NRMSE train: [0.11553478]</p> <p>NRMSE test: [0.11665033]</p> | |
| <p>MSE over training epochs:</p>  | <p>MSE over training epochs:</p>  | |
| <p>Example model outputs:</p>  | <p>Example model output:</p>  | |

* For reference, here is the original image that corresponds to the example model output:



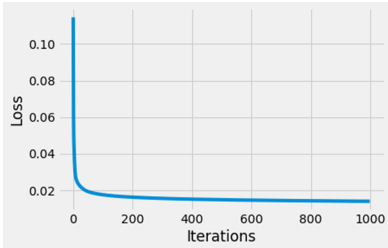
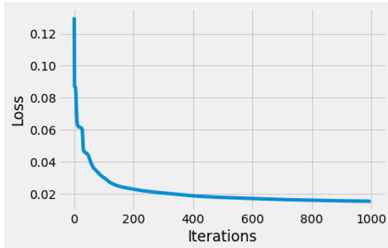
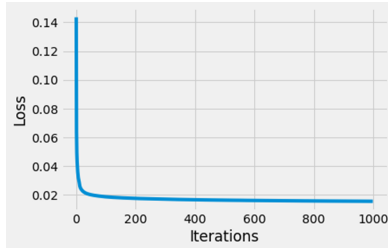
The worst result by far belonged to stochastic gradient descent. The loss function did not really move over the epochs as shown by the loss plot, where the y-axis spanned 0.19 to 0.2050. The rest of the optimization strategies were able to converge after 1000 training epochs. Adam showed the best results for NRMSE and MSE loss (although it was closely followed by RMSprop), so we stuck with it.

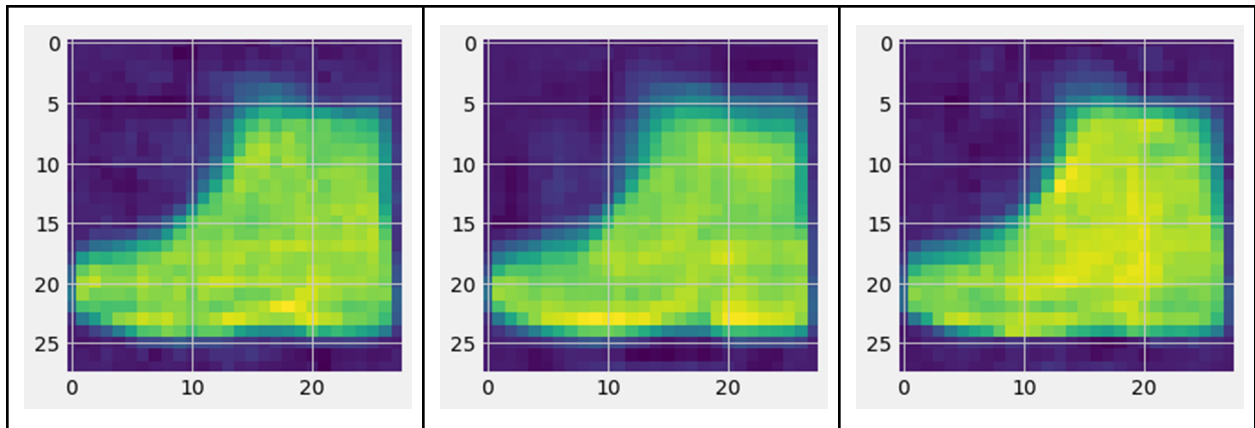
Nonlinearities

We tested the following nonlinearities:

1. Leaky ReLU (original)
2. Sigmoid
3. ReLU

Results:

| Leaky ReLU NRMSE train: [0.11268099] NRMSE test: [0.11385359] | Sigmoid NRMSE train: [0.11883657] NRMSE test: [0.1196512] | ReLU NRMSE train: [0.11940407] NRMSE test: [0.12207762] |
|--|---|--|
| MSE over training epochs:  | MSE over training epochs:  | MSE over training epochs:  |
| Example model output: | Example model output: | Example model output: |



All three loss functions performed similarly, but the leaky ReLU had the lowest NRMSE values, so we stuck with it. We also tested the leaky ReLU with a negative slope of 0.1 instead of the given initial 0.5, but the difference was negligible.

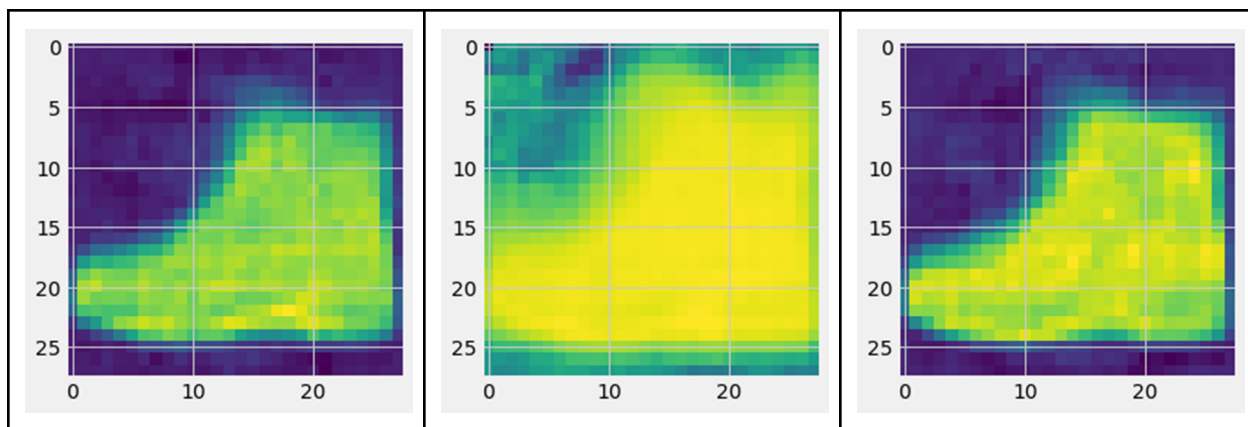
Loss functions

We tested the following loss functions:

1. MSE (original)
2. Binary cross entropy
3. KL divergence

Results:

| | | |
|---|--|---|
| MSE NRMSE train: [0.11268099] NRMSE test: [0.11385359] | Binary cross entropy NRMSE train: [12.898898] NRMSE test: [12.803504] | KL divergence NRMSE train: [0.24288267] NRMSE test: [0.24332926] |
| Loss over training epochs: | Loss over training epochs: | Loss over training epochs: |
| Example model output: | Example model output: | Example model output: |



Visually, it was immediately apparent that binary cross entropy performed the worst out of the three. Additionally, although the KL divergence example image appeared quite good, its NRMSE values were significantly higher than those belonging to MSE. We therefore continued to use MSE for training.

Free parameters

The size of the latent space seems to be dependent on storage specifications (how much compression are we trying to achieve). Bigger latent spaces would have less compression and therefore yield better results, but would require more storage space. For the purposes of trying to improve the architecture, we just constrained the latent space to a dimension of 8 (and this would change anyways when calculating results with different compression ratios).

We kept the number of epochs to 1000. This was enough for the loss functions to converge, and further training would see diminishing returns in improvements.

In the end, the original model training scheme (using Adam, the leaky ReLU, and MSE) performed the best.

[2] - fashionMNIST_fcAE_lstsq_on_decoder.py

We started with the output at the decoder and reversed the decoding pipeline by using least squares to get back to the latent space. This consisted of two main parts:

1. Inverting the nonlinearity: to invert the leaky ReLU, we defined a function “activation_inv” that took in an input x and returned it if x was nonnegative. If x was negative, we returned $2*x$ to “undo” the initial leaky ReLU transformation.
2. Reversing the fully connected linear layer:
 - the forward process followed the equation $x_{out} = x_{in} @ W^T + b$
 - to reverse this and get the input, we subtracted the bias and then multiplied by the inverse of W^T : $x_{in} = (x_{out} - b) @ W^{T^{-1}}$

Once we had these two parts, we just had to implement them in the reverse order of the forward decoding process.

Finally, the Frobenius norm between the latent space discovered by the autoencoder (forward process) and the latent space obtained using the least-squares approach was computed for the training and test sets. The smaller the norm, the closer the two matrices are to each other. The goal of this step was to validate the effectiveness of the autoencoder model in compressing and decompressing the input data. Theoretically, there should have been no difference between the two. Our Frobenius norms reported as 0.0153 for the training set and 0.0062 for the test set. This disparity could have been due to slight differences in the autoencoder and least-squares approach. For example, our least-squares approach made use of the pseudoinverse, which could have offset the results a little.

```

116 x_train_std_max = x_train_std_max()
117 x_train_std_min = x_train_std_min()
118 x_train_std_norm = (x_train_std - x_train_std_min)/(x_train_std_max-x_train_std_min)
119 x_test_std_norm = (x_test_std - x_train_std_min)/(x_train_std_max-x_train_std_min)
120
121 latent_dim = 8
122 model = Autoencoder(latent_dim)
123 model.load_state_dict(torch.load("./results/latent_{}_best_parameters_std_norm.pt".format(latent_dim)))
124
125 # Forward process
126 with torch.no_grad():
127     result_train = model(x_train_std_norm).detach().cpu()
128     result_test = model(x_test_std_norm).detach().cpu()
129
130 h_train = model.encode(x_train_std_norm)
131 h_test = model.encode(x_test_std_norm)
132
133 # Reverse the decoder process
134 # Activation function is LeakyRelu(0.5)(x)
135 # Inverse is x*x when x>0 and x*-5x otherwise
136 activation_inv = lambda x: torch.where(x >= 0, x, 2*x)
137
138 # Start with decoder-1c3
139 weight = model.decoder1.weight # shape is (784, 392)
140 bias = model.decoder1.bias # shape is (784)
141
142 # Reverse self.decoder1(x)
143 # Forward process was x = (x @ weight.T) + bias
144 # so we need to subtract bias and multiply by inverse of 0.5
145 dec1_input_train = (result_train - bias) @ torch.linalg.pinv(weight.T)
146 dec1_input_test = (result_test - bias) @ torch.linalg.pinv(weight.T)
147
148 # Reverse self.decoder2(x)
149 dec2_output_train = activation_inv(dec1_input_train)
150 dec2_output_test = activation_inv(dec1_input_test)
151
152 # Reverse self.decoder3(x)
153 weight = model.decoder2.weight
154 bias = model.decoder2.bias
155 dec2_input_train = (dec2_output_train - bias) @ torch.linalg.pinv(weight.T)
156 dec2_input_test = (dec2_output_test - bias) @ torch.linalg.pinv(weight.T)
157
158 # Reverse self.decoder4(x)
159 dec1_output_train = activation_inv(dec2_input_train)
160 dec1_output_test = activation_inv(dec2_input_test)
161
162 # Reverse self.decoder5(x)
163 weight = model.decoder1.weight
164 bias = model.decoder1.bias
165 dec1_input_train = (dec1_output_train - bias) @ torch.linalg.pinv(weight.T)
166 dec1_input_test = (dec1_output_test - bias) @ torch.linalg.pinv(weight.T)
167
168 h_train_from_output = dec1_input_train
169 h_test_from_output = dec1_input_test
170
171 print("Frobenius norm between latent space discovered by Autoencoder and Least-squares approach:")
172 print("Training set:")
173     torch.norm(h_train-h_train_from_output)
174 print("Test set:")
175     torch.norm(h_test-h_test_from_output)
176
177 Run current cell (Use IPython to create cells)
  
```

| Name | Type | Size | Value |
|--------------------|------------------------|--------------|---|
| bias | nn.parameter.Parameter | (196,) | Parameter object of torch.nn.parameter module |
| dec1_input_test | Tensor | (10000, 8) | Tensor object of torch module |
| dec1_input_train | Tensor | (60000, 8) | Tensor object of torch module |
| dec1_output_test | Tensor | (10000, 196) | Tensor object of torch module |
| dec1_output_train | Tensor | (60000, 196) | Tensor object of torch module |
| dec2_input_test | Tensor | (10000, 196) | Tensor object of torch module |
| dec2_input_train | Tensor | (60000, 196) | Tensor object of torch module |
| dec2_output_test | Tensor | (10000, 392) | Tensor object of torch module |
| dec2_output_train | Tensor | (60000, 392) | Tensor object of torch module |
| dec3_input_test | Tensor | (10000, 392) | Tensor object of torch module |
| dec3_input_train | Tensor | (60000, 392) | Tensor object of torch module |
| device | str | 4 | cuda |
| h_test | Tensor | (10000, 8) | Tensor object of torch module |
| h_test_from_output | Tensor | (10000, 8) | Tensor object of torch module |
| h_train | Tensor | (60000, 8) | Tensor object of torch module |

```

Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license()" for more information.

Python 7.11.1 -- An enhanced interactive Python.
Restarting kernel...

Removing all variables...

In [1]: runfile('C:/Users/user/Documents/fashionMNIST_AE_2/fashionMNIST_fcAE_stdq_on_decoder.py', wdir='C:/Users/user/Documents/fashionMNIST_AE_2')
Using device: cuda
Frobenius norm between latent space discovered by Autoencoder and least-squares approach:
training set: tensor(0.0153)
test set: tensor(0.0062)

In [2]:
  
```

Code structure:

This algorithm is for a basic implementation of an autoencoder, and it uses PyTorch to create and provides insight for the model training process. It consists of two parts: an encoder that compresses the input data into a lower-dimensional representation, and a decoder that reconstructs the original data from the compressed representation. The dataset is split into training and test sets, and the input data is converted to PyTorch tensors. Next, an autoencoder neural network is defined using the nn.Module class of PyTorch. The autoencoder has three encoder and three decoder layers, each of which is a fully connected (dense) layer. The encoder layers reduce the dimensionality of the input data to the specified latent dimension, while the decoder layers increase the dimensionality back to the original input shape. The activation function used for each layer is a leaky ReLU function with a slope of 0.5. After defining the autoencoder model, the training and test data are normalized using standardization, and the resulting tensors are normalized to the range [0, 1]. The autoencoder model is then loaded from a

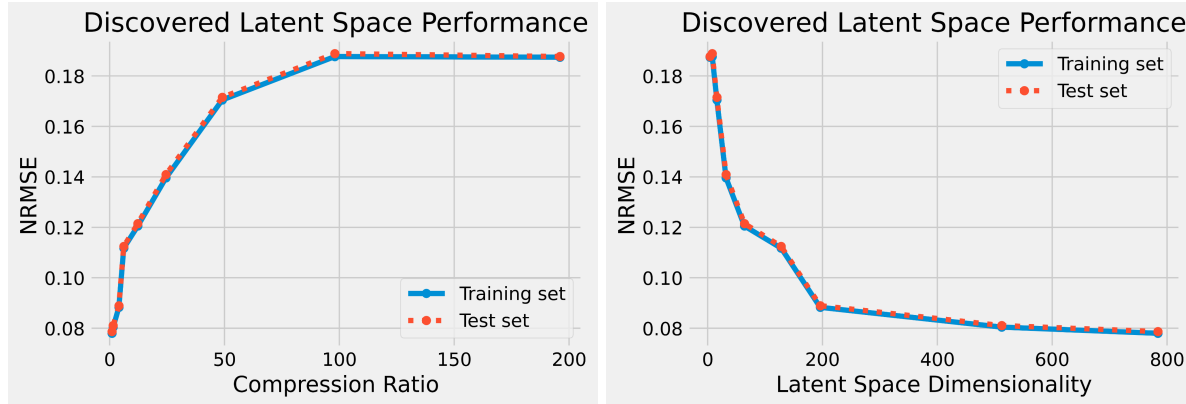
saved file, and the encoder and decoder processes are reversed to obtain the compressed representation (latent space) of the input data. This is done using the least-squares approach, which is a method of finding a solution to an overdetermined system of equations.

[3] - fashionMNIST_fcAE_lstsq_true_instances_on_decoder.py

Similarly to algorithm 2, this code performed reverse reconstruction starting at the output layer of the autoencoder and working backwards through each decoder layer. We followed the same process of applying the inverse operation of each layer's activation function and linear transformation. However, this time we started with the target (a training/test set instance) instead of the decoder output. From here, we computed the latent space before taking it back through the forward process of the decoder and getting another output.

We computed the Frobenius norm in both the latent space and top layer between the autoencoder outputs and our least-squares approach. This provided a measure for us to compare decoding based on the autoencoder's computed latent space and our target latent space. We were able to check how close the reconstructed latent vectors were to the "true" ones. Our code reported the following results:

- Frobenius norm between latent space discovered by Autoencoder and least-squares approach with true instances:
 - training set= tensor(1724.5399)
 - test set= tensor(709.8581)
- Frobenius norm at top layer between outputs and targets:
 - training set= tensor(1083.3223)
 - test set= tensor(447.2425)



We were initially surprised that the NRMSE showed worse results for the discovered latent space, since we thought that starting from the target output might yield higher fidelity results. However, after further thought, it did seem to make sense that since the decoder weights were tuned to the autoencoder latent space and not the one we discovered by considering the target as the decoder output, the autoencoder latent space would have more optimized performance.

The autoencoder output and latent space did seem to follow the same general trend of increased performance when lowering the compression ratio (increasing the latent space dimensionality). This makes intuitive sense because we would expect more information to be retained from the original input with less compression.

Code structure

This algorithm evaluates the performance of an autoencoder model on the fashionMNIST dataset. The first few lines of code standardize the input data by subtracting the mean and dividing by the standard deviation, and then normalize it to have values between 0 and 1. The code then defines a list of latent dimensions to use in the autoencoder, and iterates through each latent dimension to evaluate the performance of the autoencoder using the selected latent dimension.

Within the loop, the code loads the pre-trained model for the current latent dimension, and applies it to both the training and test data to obtain the reconstructed output. It then calculates the normalized root-mean-squared error (NRMSE) between the reconstructed output and the input data for both the training and test data. The code also calculates the compression ratio (the ratio of the original input dimension to the latent dimension) for each evaluation. This is also done for the discovered latent space (where we begin at the decoder output with original training and test images). The code generates four plots: two for the autoencoder's performance on the training and test data based on compression ratio and latent space dimensionality, and two more for discovered latent space performance based on latent space dimensionality and compression ratio. The performance is measured using the NRMSE metric, with lower values indicating better performance.