COMP5421 Computer Vision

Homework Assignment 4 Neural Networks for Recognition

Hartanto Kwee Jeffrey
jhk@connect.ust.hk — SID: 20851871

1 Theory

Q1.1

With the translated vector as $x + c$, we have

$$\begin{aligned}\text{softmax}((x+c)_i) &= \frac{e^{(x+c)_i}}{\sum_j e^{(x+c)_j}} \\ &= \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} \\ &= \frac{e^{x_i}e^c}{\sum_j e^{x_j}e^c} \\ &= \frac{e^{x_i}}{\sum_k e^{x_j}} \\ &= \text{softmax}(x_i)\end{aligned}$$

If we use $c = -\max x_i$, then $(x+c)_i = x_i - \max x_j \leq 0$, making $0 \leq e^{(x+c)_i} \leq 1$. Restricting the range of values of the exponents to a smaller range makes calculations more stable, since large exponents are difficult to calculate.

Q1.2

- Each element is between 0 to 1. The sum over all elements is 1. (Softmax is also differentiable.)
- "softmax takes an arbitrary real valued vector x and turns it into a probability distribution function"
- The multi-step process allows the calculation to be efficiently executed using vector operations (e.g. in 'numpy'):
  1. $s_i = e^{x_i}$ calculates the exponent of each element, e.g. `s = np.exp(x)`. This amplifies large activations and supresses small activations, creating a greater differences between the activation.
  2. $S = \sum s_i$ calculates the sum of the exponents, e.g. `S = np.sum(s)`, which will become the normalization factor.
  3. $\text{softmax}(x_i) = \frac{1}{S}s_i$ normalizes the vector using S, e.g. `softmax = (1/S)*s`, and this makes the sum across all elements 1.

Q1.3

Without a non-linear activation function (i.e. with a linear activation function, e.g. $f(\mathbf{x}) = A\mathbf{x} + B$ for $A, B \in \mathbb{R}$), we obtain the following expression for relating the activation vectors between layers l and $l-1$:

$$\begin{aligned}\mathbf{a}^{(l)} &= f(\mathbf{z}^{(l)}) \\ &= f(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \\ &= A(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) + B \\ &= A\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \left[A\mathbf{b}^{(l)} + B\right] \\ &= (\text{some matrix})\mathbf{a}^{(l-1)} + (\text{some vector}) \quad \ldots(*)\end{aligned}$$

where $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l-1}\times n_l}$ and $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ are the weights and biases connecting the two layers. Suppose we let $\mathbf{a}^{(l)} = \mathbf{P}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{Q}^{(l)}$. Applying this recurrence relationship repeatedly, we have

$$\begin{aligned}\mathbf{a}^{(N-1)} &= \mathbf{P}^{(N-1)}\mathbf{a}^{(N-2)} + \mathbf{Q}^{(N-1)} \\ &= \mathbf{P}^{(N-1)}(\mathbf{P}^{(N-2)}\mathbf{a}^{(N-3)} + \mathbf{Q}^{(N-2)}) + \mathbf{Q}^{(N-1)} \\ &= \mathbf{P}^{(N-1)}\mathbf{P}^{(N-2)}\mathbf{a}^{(N-3)} + \left(\mathbf{P}^{(N-1)}\mathbf{Q}^{(N-2)} + \mathbf{Q}^{(N-1)}\right) \\ &= \ldots\end{aligned}$$

We eventually find that the outputs $\hat{\mathbf{y}} = \mathbf{a}^{(N-1)}$ and the inputs $\mathbf{x} = \mathbf{a}^{(0)}$ are related by

$$\hat{\mathbf{y}} = \mathbf{P}\mathbf{x} + \mathbf{Q}$$

where $\mathbf{P}$ and $\mathbf{Q}$ results from repeated multiplication and addition of the matrices in $(*)$. We see this is basically the linear regression model.

In the backpropagation step, we try to minimize the sum of square errors $L = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2$, which is also the error we are minimizing in linear regression.

Hence, we conclude that without a non-linear activation function, a multi-layer neural network reduces to linear regression.

Q1.4

The derivation is as follows:

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}} \\ e^{-x} &= \frac{1}{\sigma(x)} - 1\end{aligned}$$

Differentiating both sides with respect to x, we obtain

$$\begin{aligned}\frac{d}{dx}(e^{-x}) &= \frac{d}{dx}\left[\frac{1}{\sigma(x)} - 1\right] \\ -e^{-x} &= -\frac{1}{[\sigma(x)]^2}\cdot\sigma'(x)\end{aligned}$$

We express e^{-x} in terms of $\sigma(x)$, then rearrange to find $\sigma'(x)$:

$$\begin{aligned}\frac{1}{\sigma(x)} - 1 &= \frac{1}{[\sigma(x)]^2}\cdot\sigma'(x) \\ \sigma'(x) &= \sigma(x)(1-\sigma(x))\end{aligned}$$

In terms of x, $\sigma'(x) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{[1+e^{-x}]^2}$.

Q1.5

We are given $y = x^T W + b$, or $y_j = \sum_{i=1}^d x_i W_{ij} + b_j$. Note that J is a function of y_i for $j = 1, 2, ..., k$. By applying chain rule to the scalar form, we have

$$\frac{\partial J}{\partial W_{ij}} = \sum_{j'=1}^{k} \frac{\partial J}{\partial y_{j'}} \frac{\partial y_{j'}}{\partial W_{ij}} = \delta_j \frac{\partial}{\partial W_{ij}} \left(\sum_{i=1}^{d} x_i W_{ij} + b_j \right) = \delta_j x_i$$

$$\frac{\partial J}{\partial x_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{k} \delta_j \frac{\partial}{\partial x_i} \left(\sum_{i=1}^{d} x_i W_{ij} + b_j \right) = \sum_{j=1}^{k} \delta_j W_{ij}$$

$$\frac{\partial J}{\partial b_j} = \sum_{j'=1}^{k} \frac{\partial J}{\partial y_{j'}} \frac{\partial y_{j'}}{\partial b_j} = \delta_j (1) = \delta_j$$

In matrix form (denominator-layout notation),

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \frac{\partial J}{\partial W_{12}} & \cdots & \frac{\partial J}{\partial W_{1k}} \\ \frac{\partial J}{\partial W_{21}} & \frac{\partial J}{\partial W_{22}} & \cdots & \frac{\partial J}{\partial W_{2k}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{d1}} & \frac{\partial J}{\partial W_{d2}} & \cdots & \frac{\partial J}{\partial W_{dk}} \end{bmatrix} = \begin{bmatrix} \delta_1 x_1 & \delta_2 x_1 & \dots & \delta_k x_1 \\ \delta_1 x_2 & \delta_2 x_2 & \dots & \delta_k x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \delta_1 x_d & \delta_2 x_d & \dots & \delta_k x_d \end{bmatrix} = x\delta^T$$

$$\frac{\partial J}{\partial x} = W\delta$$

$$\frac{\partial J}{\partial b} = \delta$$

Q1.6

1. Observe the plotted gradient of the sigmoid function:

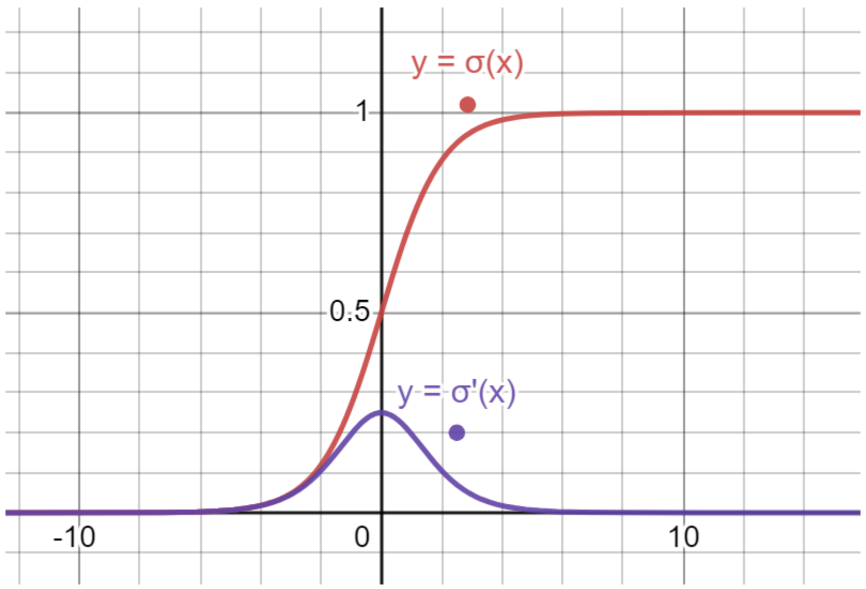


Figure 1: Graph of $y = \sigma'(x)$

We can see that the range of values of $\sigma'(x)$ is very small (≤ 0.25). During backpropogation, the gradients for progressively early layers will decrease exponentially due to the repeated multiplication of $\sigma'(x)$ in chain rule. This significantly slows down the updating of weights in earlier layers, causing a "vanishing gradient".

2. The output range of sigmoid is $(0, 1)$, while that of tanh is $(-1, 1)$. We prefer tanh because the range of output is stretched to $(-1, 1)$ and in backpropagation, it makes the gradient ($\nabla_{W_i ij} J \propto \delta_j(y)$) greater which leads to quicker convergence.

3. This is because the gradient of tanh is steeper: its maximum gradient is 1, 4 times greater than that of sigmoid. This alleviates the problem of vanishing gradients. We also take the opportunity to emphasize the gradients near 1 is important: since sigmoid has at most 0.25 gradient, every extra layer reduce its

gradient by a factor of 0.25, and for three layers that amounts to $0.25^3 = 0.015625 \approx 0.016$ which is a large reduction. However, repeated multiplications of 1 will yield 1, so it makes vanishing gradients much less proabable in multi-layer neural networks.

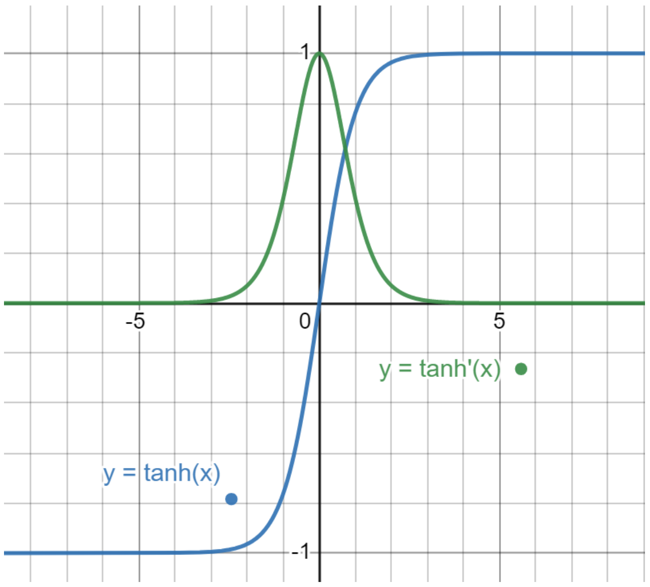


Figure 2: Graph of $y = \tanh'(x)$

4. Since $e^{-x} = \dfrac{1-\sigma(x)}{\sigma(x)} \implies e^{-2x} = \dfrac{1-\sigma(2x)}{\sigma(2x)}$, we have

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = \frac{1-\dfrac{1-\sigma(2x)}{\sigma(2x)}}{1+\dfrac{1-\sigma(2x)}{\sigma(2x)}} = 2\sigma(2x) - 1$$

2 Fully Connected Network

Q2.1

Q2.1.1

If we initialize the network with zero weights and biases, for ReLu/linear/tanh activation functions where $f(0) = 0$, the network will only be able to output zeros since the hidden layers evaluate to zero, and backpropagation gradient will also be zero because the weights and outputs are zero (refer to the equations derived in Q1.5 and Q2.3). The network, as a result, is unable to train.

For single-output neural networks, even if the outputs are not zero, the gradient will be the same for all weights, so all of the weights are updated the same way, causing inherent symmetry within weights and hence has poor accuracy.

For sigmoid networks (non-zero output at 0) like the one we implemented, the convergence will be slower because normally the neuron weights at loss function minima will be far from all 0, and it will take more backpropagation steps to reach a local minima.

Q2.1.3

We use random initialization to "break symmetry", solving the problem with zero/constant-initialization and increasing accuracy. Also, running the program multiple times with randomly initialized weights can give different and perhaps better results.

Scaling the initialization controls the sizes of the gradients and hence the sizes. If we view the update rules, the gradient fed to a layer involves summing up the weights of the previous layer. Scaling the weights according to layer size controls the size of this sum, and prevents vanishing or exploding gradients. This helps stablize and speed up the gradient descent.

Q2.3

The derivations for the gradients are listed below.
Definition of the neural network:

$$a^{(t)} = W^{(t)}h^{(t-1)} + b^{(t)}$$

$$h^{(t)} = g\left(a^{(t)}\right),\ h^{(T)} = o\left(a^{(T)}\right)$$

The dimensions of the variables are $W^{(t)} \in R^{n^{(t-1)} \times n^{(t)}}$ and $h^{(t)}, a^{(t)},\ b^{(T)} \in R^{n^{(t)} \times 1}$, where T is the number of layers in the network and n_i is the number of neurons in the i-th layer. The gradient of cross-entropy(softmax(x)):

$$\frac{\partial J}{\partial a_i^{(T)}} = h_i^{(T)} - y_i \Longrightarrow \frac{\partial J}{\partial a^{(T)}} = h^{(T)} - y$$

The gradients of the output layer (should be consistent with Q1.5) are

$$\frac{\partial J}{\partial W_{ij}^{(T)}} = \sum_{j'=1}^{n^{(T)}} \frac{\partial J}{\partial a_{j'}^{(T)}} \frac{\partial a_{j'}^{(T)}}{\partial W_{ij}^{(T)}} = \frac{\partial J}{\partial a_j^{(T)}} \frac{\partial a_j^{(T)}}{\partial W_{ij}^{(T)}} = \left[h_j^{(T)} - y_j\right] h_i^{(T-1)}$$

$$\Longrightarrow \frac{\partial J}{\partial W^{(T)}} = h^{(T-1)} \left(\left[h^{(T)} - y\right]\right)^T$$

$$\frac{\partial J}{\partial b_j^{(T)}} = \sum_{j'=1}^{n^{(T)}} \frac{\partial J}{\partial a_{j'}^{(T)}} \frac{\partial a_{j'}^{(T)}}{\partial b_j^{(T)}} = \frac{\partial J}{\partial a_j^{(T)}} \frac{\partial a_j^{(T)}}{\partial b_j^{(T)}} = \left[h_j^{(T)} - y_j\right]$$

$$\Longrightarrow \frac{\partial J}{\partial b^{(T)}} = h^{(T)} - y$$

$$\frac{\partial J}{\partial h_i^{(T-1)}} = \sum_{j'=1}^{n^{(T)}} \frac{\partial J}{\partial a_{j'}^{(T)}} \frac{\partial a_{j'}^{(T)}}{\partial h_i^{(T-1)}} = \sum_{j'=1}^{n^{(T)}} \left[h^{(T)} - y\right]_{j'} \frac{\partial a_{j'}^{(T)}}{\partial h_i^{(T-1)}} = \sum_{j'=1}^{n^{(T)}} \left[h^{(T)} - y\right]_{j'} W_{ij'}^{(T)}$$

$$\Longrightarrow \frac{\partial J}{\partial h^{(T-1)}} = W^{(T)} \left[h^{(T)} - y\right]$$

The gradients of the hidden layers are

$$\frac{\partial J}{\partial W_{ij}^{(T-1)}} = \sum_{j'=1}^{n^{(T-1)}} \frac{\partial J}{\partial h_{j'}^{(T-1)}} \frac{\partial h_{j'}^{(T-1)}}{\partial a_{j'}^{(T-1)}} \frac{\partial a_{j'}^{(T-1)}}{\partial W_{ij}^{(T-1)}} = \frac{\partial J}{\partial h_j^{(T-1)}} g'\left(a_j^{(T-1)}\right) h_i^{(T-2)}$$

$$\Longrightarrow \frac{\partial J}{\partial W^{(T-1)}} = h^{(T-2)} \left[\frac{\partial J}{\partial h^{(T-1)}} \odot g'\left(a^{(T-1)}\right)\right]^T$$

$$\frac{\partial J}{\partial b_j^{(T-1)}} = \sum_{j'=1}^{n^{(T-1)}} \frac{\partial J}{\partial h_{j'}^{(T-1)}} \frac{\partial h_{j'}^{(T-1)}}{\partial a_{j'}^{(T-1)}} \frac{\partial a_{j'}^{(T-1)}}{\partial b_j^{(T-1)}} = \frac{\partial J}{\partial h_j^{(T-1)}} g'\left(a_j^{(T-1)}\right)$$

$$\Longrightarrow \frac{\partial J}{\partial b^{(T-1)}} = \frac{\partial J}{\partial h^{(T-1)}} \odot g'\left(a^{(T-1)}\right)$$

$$\frac{\partial J}{\partial h_i^{(T-2)}} = \sum_{j'=1}^{n^{(T-1)}} \frac{\partial J}{\partial h_{j'}^{(T-1)}} \frac{\partial h_{j'}^{(T-1)}}{\partial a_{j'}^{(T-1)}} \frac{\partial a_{j'}^{(T-1)}}{\partial h_i^{(T-2)}} = \sum_{j'=1}^{n^{(T-1)}} \frac{\partial J}{\partial h_{j'}^{(T-1)}} g'\left(a_{j'}^{(T-1)}\right) W_{ij'}^{(T-1)}$$

$$\Longrightarrow \frac{\partial J}{\partial h^{(T-2)}} = W^{(T-1)} \left[\frac{\partial J}{\partial h^{(T-1)}} \odot g'\left(a^{(T-1)}\right)\right]$$

Note that since our network only has two layers, $h^{(T-2)} = x$. Also note that in our implementation, h, b and a are one-dimensional. Transposition may be needed at times.

3 Training Models

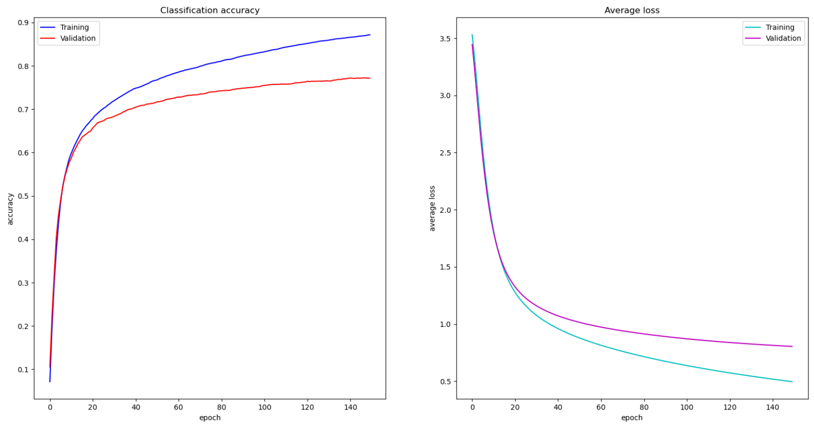
Q3.1

Q3.1.2

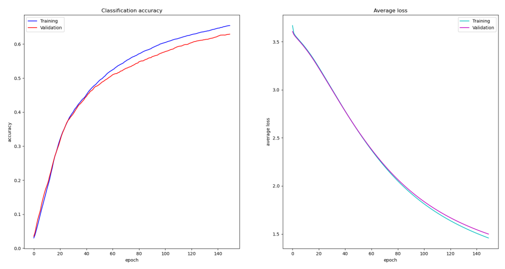
The parameters for the best model is

```
max_iters = 150
batch_size = 50
learning_rate = 8e-4
```

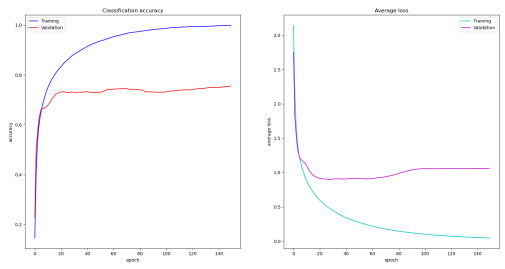
Figure 3 shows the loss and accuracy plot for the three different settings.



(a) Best learning rate



(b) ×1/10 learning rate



(c) ×10 learning rate

Figure 3: Loss and accuracy curves for three learning rate settings.

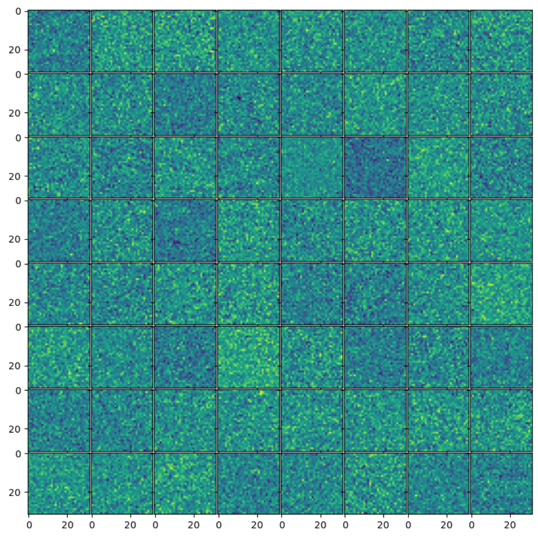
Obviously, deviation from most optimal learning rate will decrease accuracy.

For a smaller learning rate, gradient descent step size is reduced, leading to slower convergence.

For a larger learning rate, gradient descent doesn't happen significantly faster, but may cause the loss to increase and fluctuate due to errors induced by large gradient steps (in a similar fashion as the erratic behaviour of large step size in Euler method). Simply said, it is easier to overshoot a local minima with a large step size.

For the best model,
Training Accuracy: 0.8717592592592597
Validation accuracy: 0.7716666666666666
Test Accuracy: 0.7844444444444445

Q3.1.3


(a) Random weights

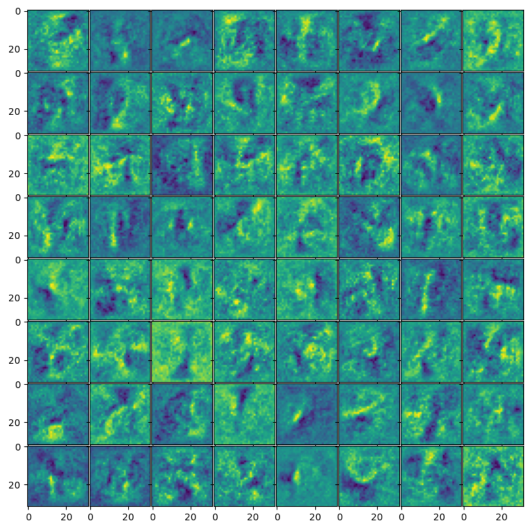

(b) Trained weights

Figure 4: Visualized first layer weights of our best model.

Figure 4 shows the Visualized first layer weights.

There is no observable pattern in the initial weights, but there are large bright and dark patches in the trained weights. This may signify that the network classifies the characters by seeing if different regions in an image are filled in.

There are also layers with random weights. These random weights probably failed to train, and their contribution to the final output (i.e. weight to the output layer) is probably small.

Q3.1.4

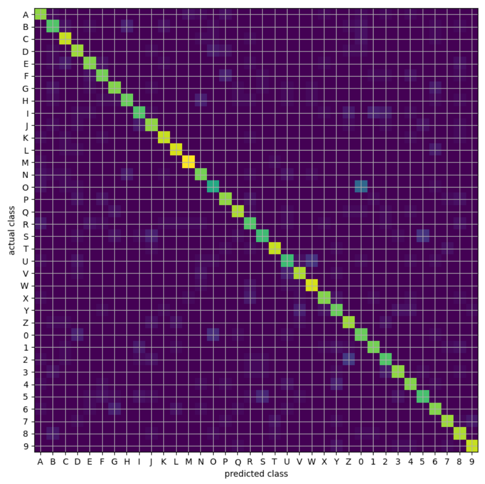


Figure 5: Confusion matrix of our best model.

Figure 5 shows the confusion matrix. The pairs the are most easily confused in this specific network are [0, O], [S, 5], [2, Z], [W, U]. These pairs are structurally similar, and only differ in whether they have sharp corners or not (and corners often become rounded in handwriting) and hence are difficult to differentiate. These letter are usually confused by humans too, but are usually correctable by understanding the context in which the letter is used. However, it would be difficult for the network to differentiate between them.

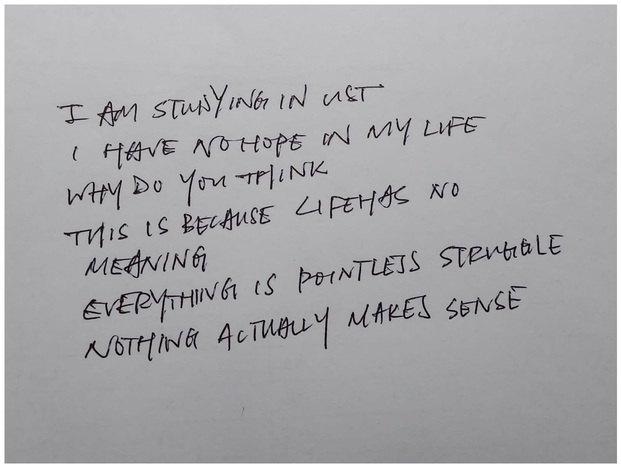
4 Extract Text from Images

Q4.1

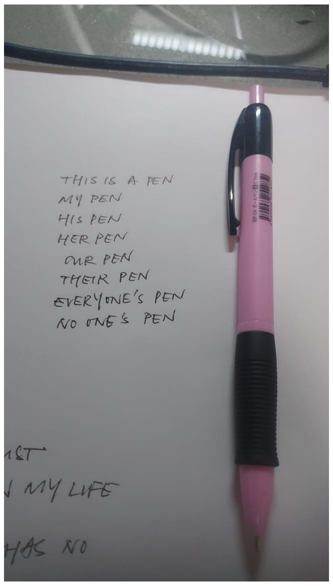
Assumptions:

1. Individual characters are clearly written apart and are not connected together by lazy strokes/compact writing. (miss valid letters)
2. The whole image contains only the piece of paper that the alphabet is written on, and does not include other objects such as the tabletop, cups, pens, etc. All dark pixels detected will be text and not other stuff. (respond to non-letters)

Refer to Figure 6 for the images in which the detection might fail.


I AM STUDYING IN UST
I HAVE NO HOPE IN MY LIFE
WHY DO YOU THINK
THIS IS BECAUSE LIFE HAS NO
MEANING
EVERYTHING IS POINTLESS STRUGGLE
NOTHING ACTUALLY MAKES SENSE

(a) Connected letters


THIS IS A PEN
MY PEN
HIS PEN
HER PEN
OUR PEN
THEIR PEN
EVERYONE'S PEN
NO ONE'S PEN
UST
MY LIFE
HAS NO

(b) Non-letter objects

Figure 6: Example images where detection will fail.

Q4.3

See Figure 7.

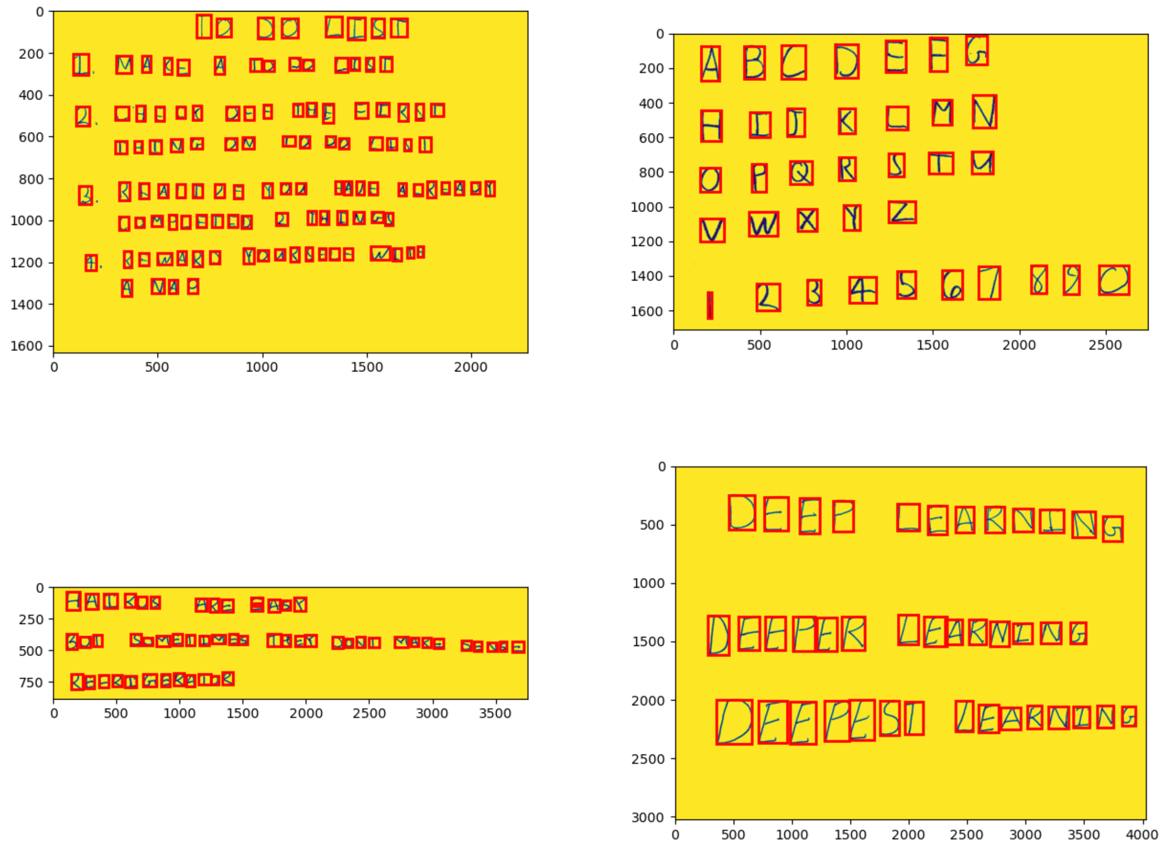


Figure 7: Located bounding boxes of letters in each sample image.

Q4.4

	Extracted text	Ground Truth

1	`TQDQLIST`
	`IHAK6ATDDGLIST`
	`2LHFCKDFFTHEFIRST`
	`THINGQNTQDQLIST`
	`3RFALIZEYOUHAVBALR6ADT`
	`GQMPLFTLD2THINGS`
	`9RGWARDYQWR5BLFWITB`
	`ANAP`
		`TODOLIST`
		`1MAKEATODOLIST`
		`2CHECKOFFTHEFIRST`
		`THINGONTODOLIST`
		`3REALIZEYOUHAVEALREADY`
		`COMPLETED2THINGS`
		`4REWARDYOURSELFWITH`
		`ANAP`

2	`ABLDBFG`
	`HIIKLMN`
	`QPQRSTW`
	`VWXTZ`
	`1Z3GSG789Q`
		`ABCDEFG`
		`HIJKLMN`
		`OPQRSTU`
		`VWXYZ`
		`1234567890`

3	`HAIKUSARGHMAGT`
	`BUTSQMBTIMBGTRETDDNTMAKGBHMQE`
	`RBGRIGBRATQR`
		`HAIKUSAREEASY`
		`BUTSOMETIMESTHEYDONTMAKESENSE`
		`REFRIGERATOR`

4	`DEBPLEARMING`
	`DBBPERLEARHING`
	`DEEPESPLEARNING`
		`DEEPLEARNING`
		`DEEPERLEARNING`
		`DEEPESTLEARNING`

5 Image Compression with Autoencoders

Q5.2

See Figure 8 for the curve.

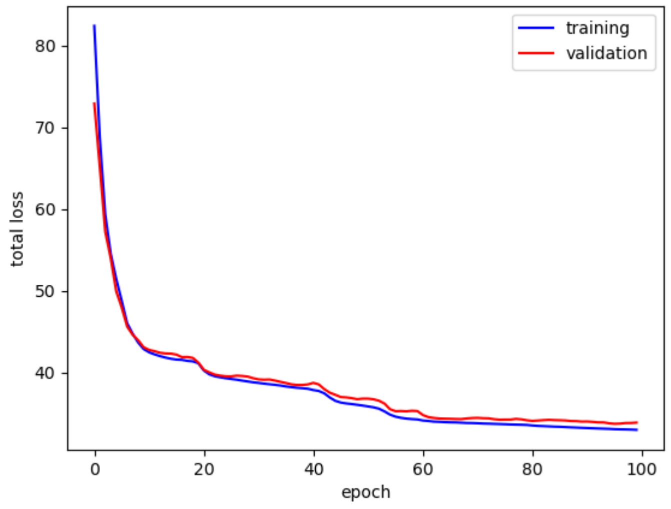


Figure 8: Loss and accuracy curve for autoencoder training.

It can be observed that the training loss curve decreases rapidly at the start, but the decrease quickly dies down at around 8 epochs. After that the loss curve decreases very slowly.

Q5.3

Q5.3.1

Refer to Figure 9 for the images.

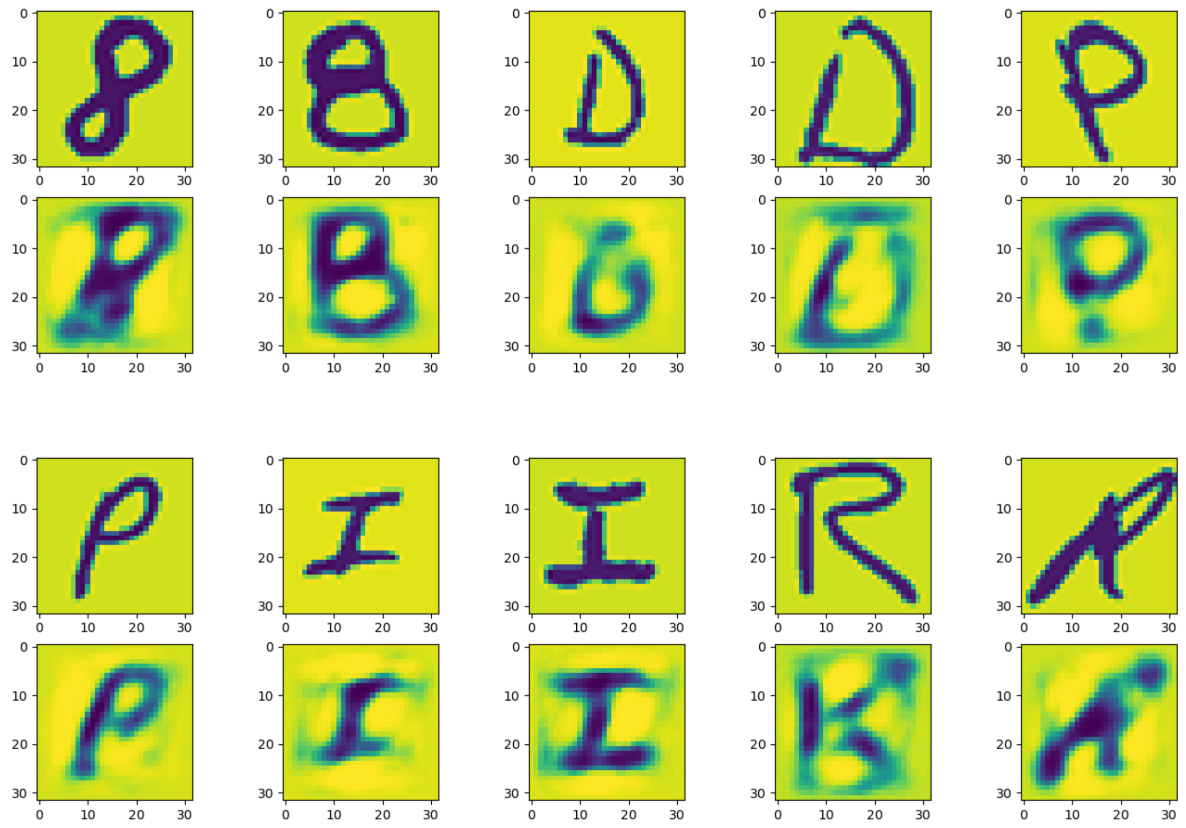


Figure 9: Reconstruction of validation images using the autoencoder.

Compared to the original image, the autoencoded images are blurry, the colors are more spreaded and diffuse, and often the letters are slightly deformed or show extra lines (e.g. the first image of R has extra lines that look like a D in the background.)

Q5.3.2

The PSNR is 15.608972404223165.

6 Compare against PCA

Q6.1

The size of the projection matrix is 32×1024, and its rank is 32.

Q6.2

Refer to Figure 10 for the images.

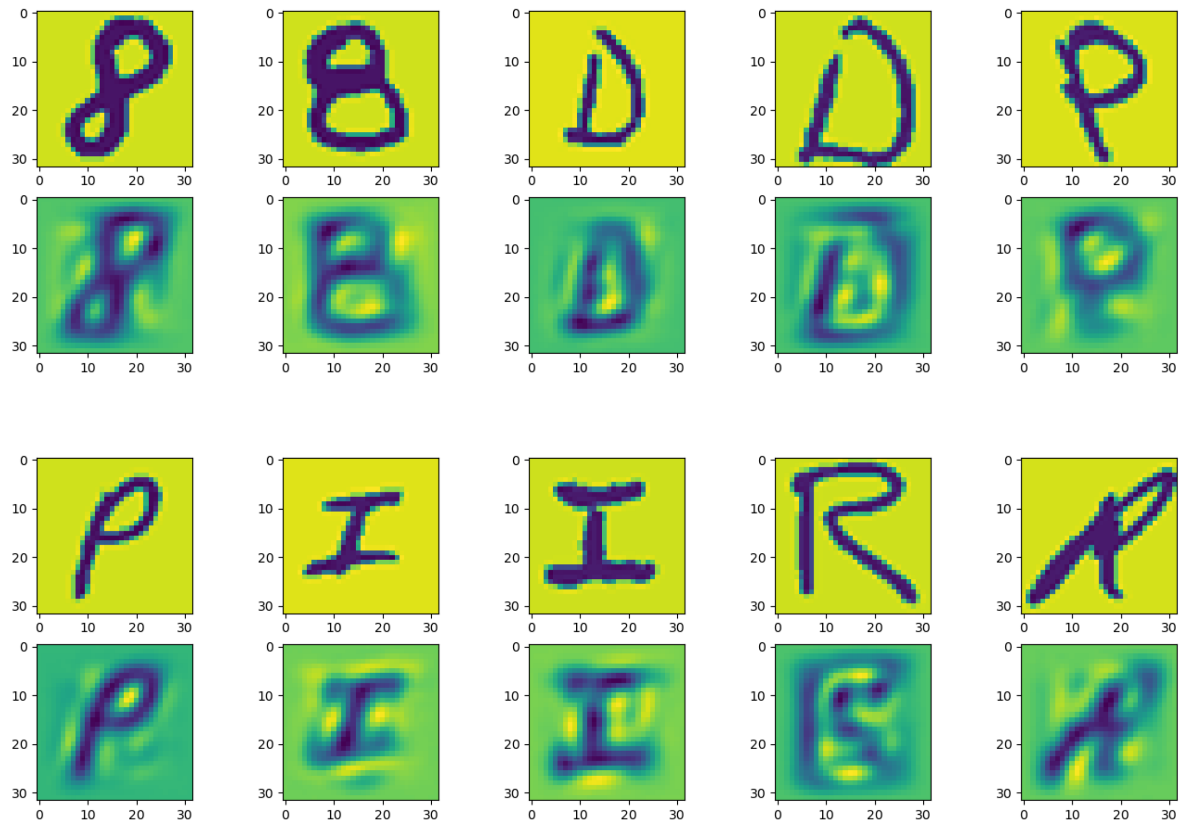


Figure 10: Reconstruction of validation images using PCA.

Compared to the original images, the images are also blurry and have artifacts that look like other characters behind them. It also has a different background color/value from the original image. When compared with autoencoders, PCA tends to have more background noise, but retains the original shape of the characters better.

Q6.3

The average PSNR is 16.348422786979377. This is slightly better than the autoencoder.

PCA is Mathematically optimized to retain the greatest amount of feature using its limited range of learning parameters/within its model (a GLOBAL minimum of MSE, its also mentioned in the question that PCA is the "best low-rank approximation"). However, autoencoders are neural networks, which uses only gradient descent and may only be able to reach LOCAL minimas. (Under limited training epochs the solution may not even be optimal!) Therefore, PCA is more robust.

Q6.4

Number of learned parameters for the autoencoder $= 1024 \times 32 + 32 + 32^2 + 32 + 32^2 + 32 + 32 \times 1024 + 1024 = 68704$ (weights and biases), while that for the PCA model is simply $1024 \times 32 = 32768$. The reason for the different in performance is explained in Q6.3, with the added note that PCA, even with less parameters, learn better than the autoencoder because of its ability to achieve deterministic global minima of MSE.

Out of curiousity, I ran PCA with 64 dimensions which will have similar learned parameters as the autoencoder, and the PSNR score is 19.00511850049286. We see that under the same number of parameters, PCA outperforms autoencoders by a large margin.

7 PyTorch

Q7.1

Q7.1.1

The network is implemented in `run_q71.py`.
Training Accuracy: 0.8758333333333339
Validation accuracy: 0.7766666666666666

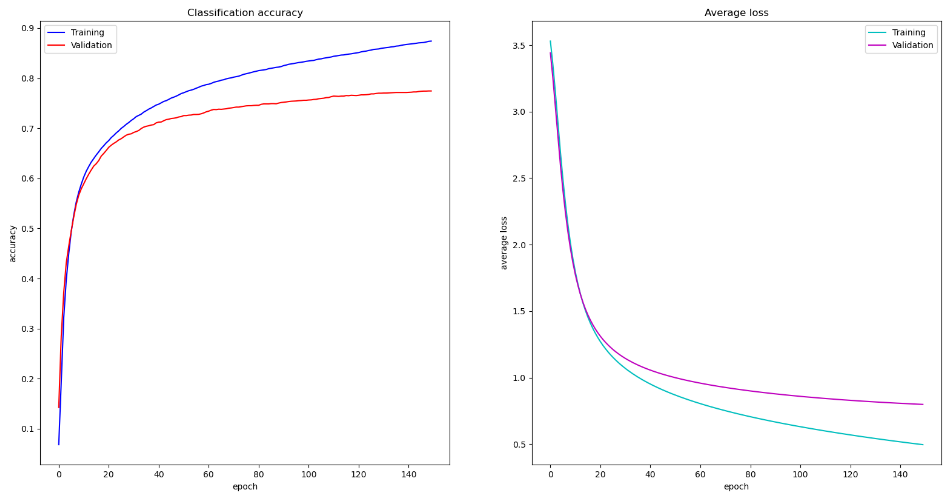


Figure 11: Q7.1.1 accuracy and loss curve.

Q7.1.2

The network is implemented in `run_q72.py`.
Training accuracy: 0.9632666666666667
Testing accuracy: 0.9762

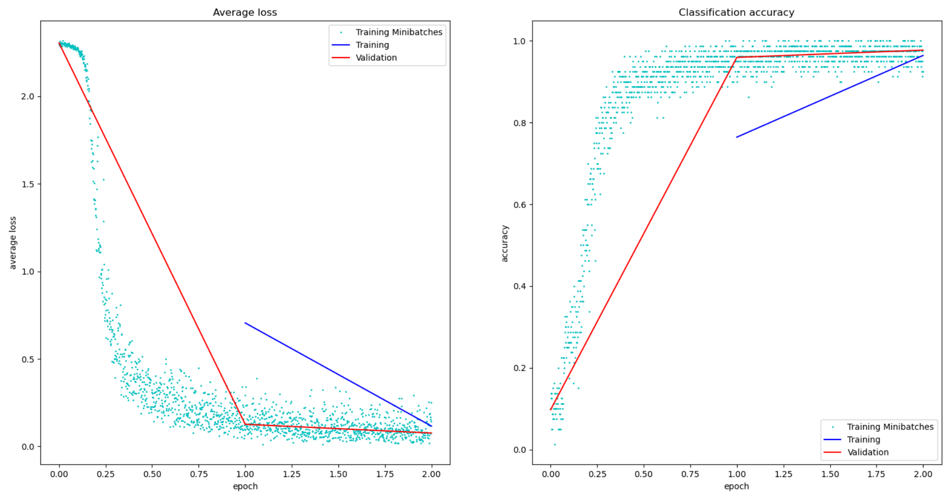


Figure 12: Q7.1.2 accuracy and loss curve. Note the loss and accuracy for each minibatch is also plotted using cyan dots.

Q7.1.3

The network is implemented in `run_q73.py`.
Training accuracy: 0.9747222222222223
Testing accuracy: 0.8905555555555555

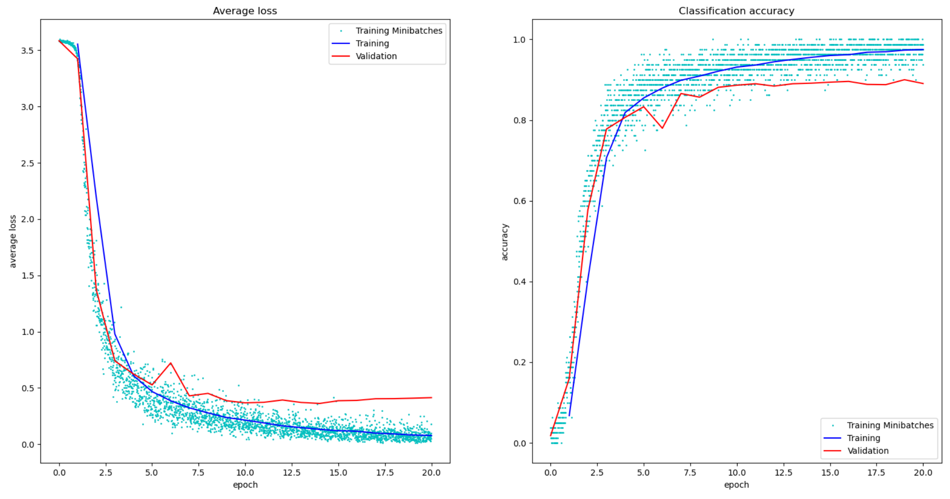


Figure 13: Q7.1.3 accuracy and loss curve.

Q7.1.4

The network is implemented in `run_q74.py`.
Training accuracy: 0.9002836879432624
Testing accuracy: 0.8759574468085106

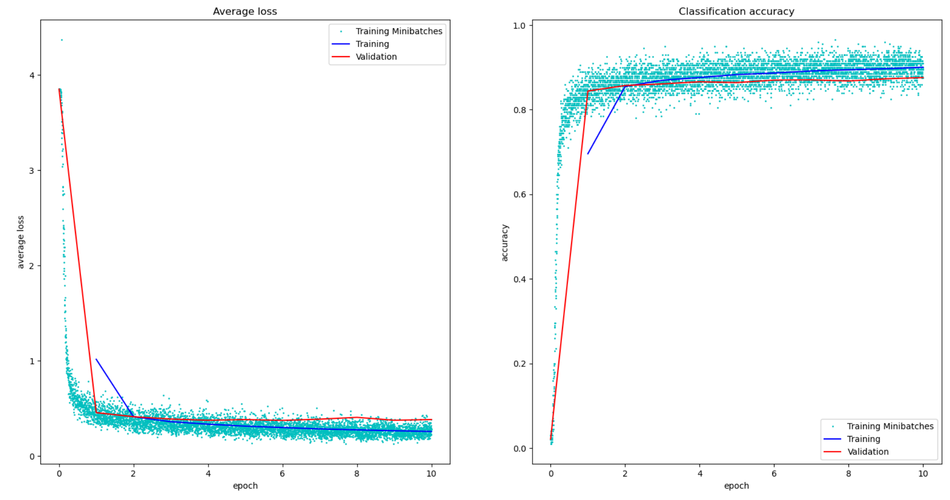


Figure 14: Q7.1.4 accuracy and loss curve.

	Extracted text	Ground Truth

1	`TODQLI5T`
	`LMAKEATDDQLIST`
	`2CHECKDFETHEFIR5T`
	`THINGQNTQDQLf5T`
	`BREALIZEYQUHAVEALREADT`
	`CQMPLETEDZTHINGS`
	`qREWARDYQUR5ELFWITH`
	`ANAP`
		`TODOLIST`
		`1MAKEATODOLIST`
		`2CHECKOFFTHEFIRST`
		`THINGONTODOLIST`
		`3REALIZEYOUHAVEALREADY`
		`COMPLETED2THINGS`
		`4REWARDYOURSELFWITH`
		`ANAP`

2	`ABCDEFG`
	`HIJKLMN`
	`QPQRSTU`
	`VWXYZ`
	`L2345678g2`
		`ABCDEFG`
		`HIJKLMN`
		`OPQRSTU`
		`VWXYZ`
		`1234567890`

3	`HAIKUSAREGMA5Y`
	`BUISQMETIMESTKEYDQNTMAKESEN5E`
	`REFRIGERATQR`
		`HAIKUSAREEASY`
		`BUTSOMETIMESTHEYDONTMAKESENSE`
		`REFRIGERATOR`

4	`DEEPLEARNING`
	`DEEPERLEAKNING`
	`DEEPE5ILEARNING`
		`DEEPLEARNING`
		`DEEPERLEARNING`
		`DEEPESTLEARNING`

Q7.2

It can be seen that the fine-tuned SqueezeNet outperforms the network trained from scratch.

Fine-tuned SqueezeNet

The network is implemented in `run_q7part2_squeeze.py`.
Training Accuracy: 1.0 Validation Accuracy: 0.9382352941176471

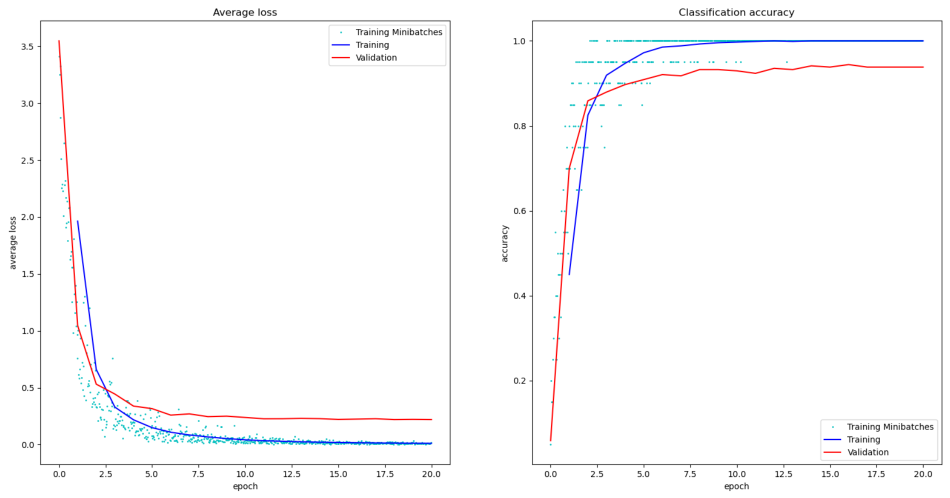


Figure 15: Q7.2 fine-tuned SqueezeNet accuracy and loss curve.

Trained from scratch

The network is implemented in `run_q7part2_scratch.py`.
Training Accuracy: 0.7647058823529411
Validation Accuracy: 0.7

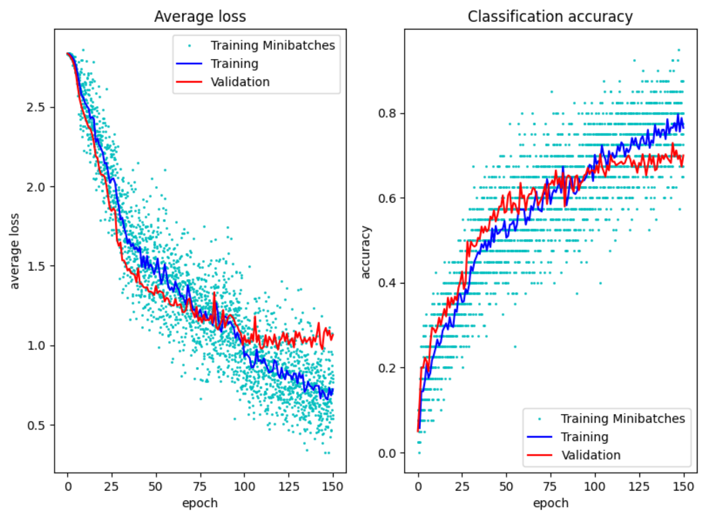


Figure 16: Q7.2 accuracy and loss curve of CNN trained from scratch.