Deep Learning (IST, 2024-25)

Homework 1

André Martins, Chrysoula Zerva, Mário Figueiredo,
Duarte Alves, Pedro Santos, Duarte Almeida, Francisco Silva, Adrian Herta

Deadline: Monday, December 16, 2024.

Please turn in the answers to the questions below in a PDF file, together with the code you implemented to solve them (when applicable).

IMPORTANT: Please write 1 paragraph indicating clearly what was the contribution of each member of the group in this project. A penalization of 10 points will be applied if this information is missing.

Please submit **a single zip file** in Fenix under your group's name.

Question 1 (35 points)

Landscape classification with linear classifiers and neural networks. In this exercise, you will implement a linear classifier to classify images of landscapes, using a preprocessed version of the Intel Image Classification dataset. The preprocessed dataset contains 17,034 RGB images of size 48x48 of different landscapes, under 6 categories. The set of labels is {0: buildings; 1: forest; 2: glacier; 3: mountain, 4: sea; 5: street}. The dataset was split into training, validation and test sets with sizes 12,630, 1,404, and 3,000 respectively. **Please do not use any machine learning library such as `scikit-learn` or similar for this exercise; just plain linear algebra (the `numpy` library is fine).** The python skeleton code is provided in `hw1-q1.py`.

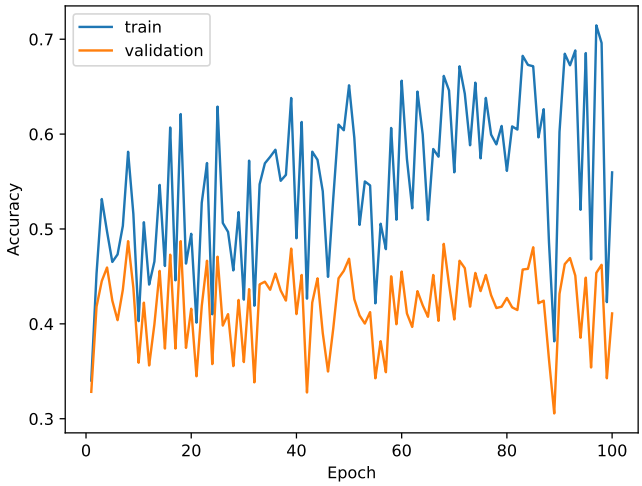
In order to complete this exercise, you will need to download the preprocessed Intel Image Classification dataset that can be found in:

`https://drive.google.com/file/d/16AzJIrmra4qWdY7wU9N1ew4bwadmNt4j`

1. In the first part of the exercise, we will implement the Perceptron algorithm.
   (a) (7 points) Implement the `update_weights` method of the `Perceptron` class in `hw1-q1.py`. Then, train the perceptron for 100 epochs on the training set and report its performance on the training, validation and test sets. Plot the train and validation accuracies as a function of the epoch number. You can do this with the command

   ```
   python hw1-q1.py perceptron -epochs 100
   ```

   Solution: The final training accuracy was 0.56. The final validation accuracy was 0.41. The final test accuracy was 0.39.

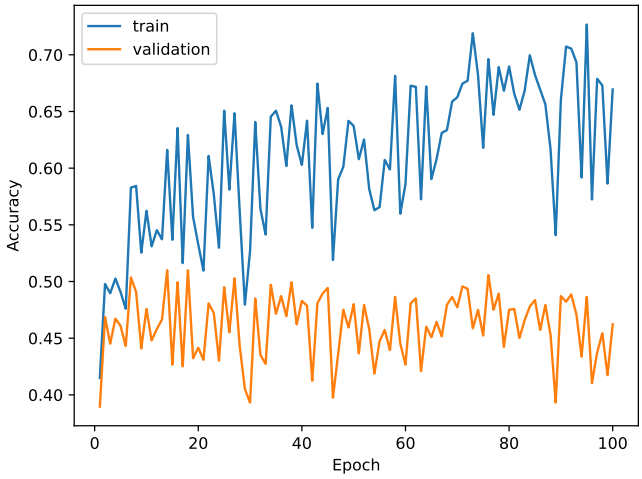


2. In the second part of this exercise, we will implement a logistic regression classifier, using stochastic gradient descent as the training algorithm. In particular, you will implement two versions of the logistic regression classifier: (i) a non-regularized version of logistic regression; and (ii) an ℓ_2-regularized version of logistic regression.

   (a) (7 points) We will start with the logistic regression **without ℓ_2 regularization**. For this, implement the `update_weights` method of the `LogisticRegression` class. In this exercise, you can ignore the `l2_penalty` argument. Train the classifier for 100 epochs with a learning rate of 0.001 using the command.

   ```
   python hw1-q1.py logistic_regression -epochs 100
   ```

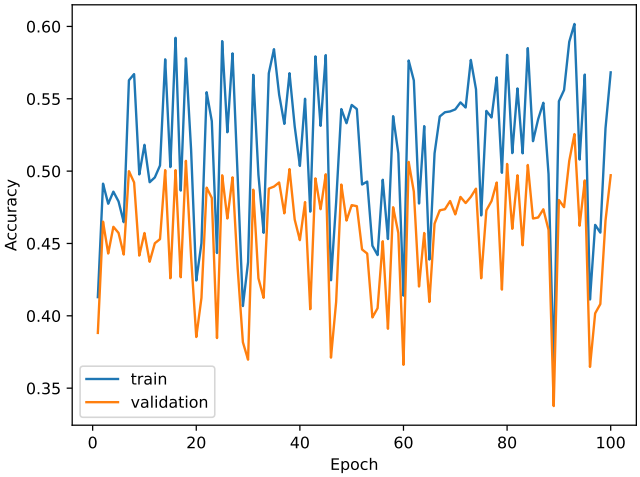
   Solution: The final test accuracy was 0.46. The train and validation accuracies are displayed below.



   (b) (5 points) Now, modify the **update_weights** function to support ℓ_2 regularization, where the strength of the regularization is defined by the `l2_penalty` argument. Note that when `l2_penalty` is zero we should recover the non-regularized version of the classifier. Train the model with a `l2_penalty` of 0.01 and report the final test accuracies as well as plot the train and validation accuracies over the epochs. Comment on the differences on the train and validation accuracies obtained with and without regularization. For training the classifier, you can use the command:

   ```
   python hw1-q1.py logistic_regression -epochs 100 -l2_penalty 0.01
   ```

Solution:



The final test accuracy was 0.51, with the training and validation losses plotted above. The model without regularization achieved a higher training accuracy but a lower validation accuracy. Additionally, its validation accuracy is reducing, suggesting the model is over-fitting to the training data. For the regularized classifier, the training accuracy is lower but the validation accuracy is higher (and still increasing), which is also reflected in a higher final test accuracy.

(c) (3 points) Finally, we take a closer look at how regularization impacts the values of the weights of the logistic regression classifier. For this, report the ℓ_2-norm of the weights (i.e., $\|\boldsymbol{W}\|_F = \sqrt{\sum_{i,j} W_{ij}^2}$, where $\boldsymbol{W}$ denotes the weights of the logistic regression classifier) of both the non-regularized and regularized versions of the logistic regression classifiers as a function of the number of epochs and comment on the obtained values.

Solution: As shown in the Figure below, the weights of the regularized version are kept significantly smaller than those of the non-regularized version of the classifier. This is expected since the ℓ_2 regularization component in the loss function of the classifier encourages smaller, more evenly distributed weights.

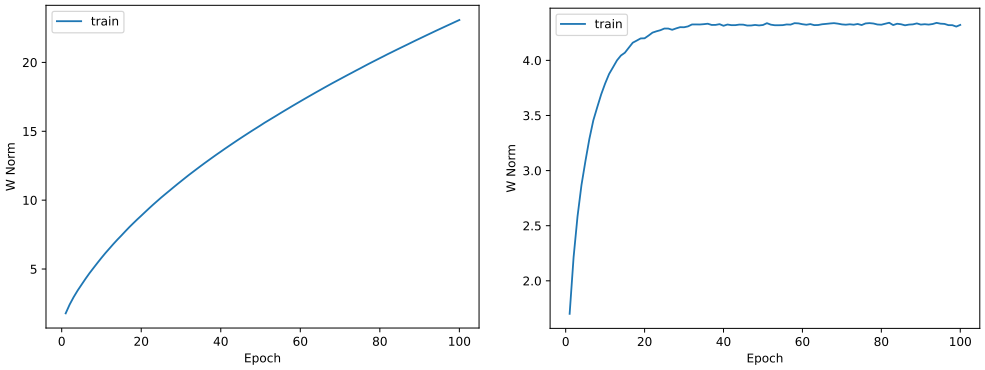


Figure 1: Logistic regression ℓ_2-norm of the weights across epochs for `l2_penalty`=0.0 (left) and `l2_penalty`=0.01 (right).

(d) (3 points) What would you expect to be different in the values of the weights at the end of training if you were to use ℓ_1 regularization instead of ℓ_2 regularization? You do **not** need to implement a logistic regression classifier with ℓ_1 regularization, just briefly comment on the expected differences.

Solution:

If we were to use ℓ_1 regularization instead of ℓ_2 regularization we would expect the number of weights that are exactly zero-valued to increase (as ℓ_1 regularization encourages sparsity).

3. Now, you will implement a multi-layer perceptron (a feed-forward neural network), again using as input the original feature representation (i.e., simple independent pixel values).

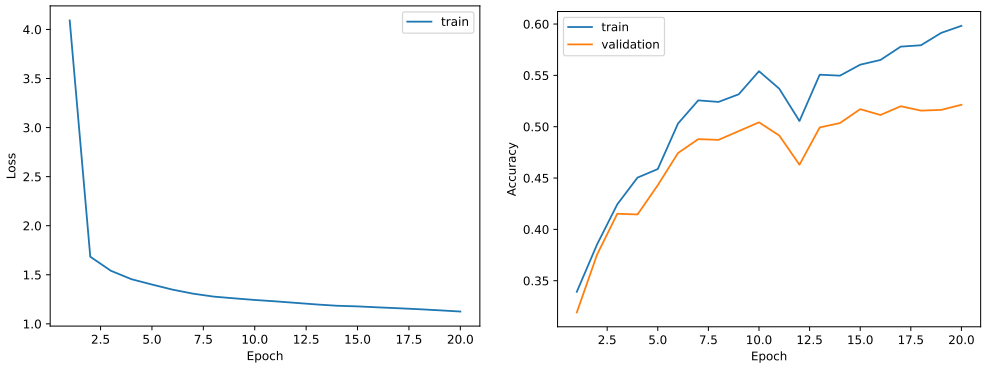
   (a) (10 points) **Without using any neural network toolkit,** implement a multi-layer perceptron with a single hidden layer to solve this problem, including the gradient backpropagation algorithm needed to train the model. Use 100 hidden units, a `relu` activation function for the hidden layers, and a multinomial logistic loss (also called cross-entropy) in the output layer. Do not forget to include bias terms in your hidden units. Train the model for 20 epochs with stochastic gradient descent with a learning rate of 0.001. Initialize biases with zero vectors and values in weight matrices with $w_{ij} \sim \mathcal{N}(\mu, \sigma^2)$ with $\mu = 0.1$ and $\sigma^2 = 0.1^2$ (hint: use `numpy.random.normal`). Run your code with the base command, adding the necessary arguments

   ```
   python hw1-q1.py mlp
   ```

   Report the final test accuracy and include the plots of the train loss and train and validation accuracies as a function of the epoch number.

   Solution: The final test accuracy was 0.54.

   The train loss and train and validation accuracies as a function of the epoch number are shown below.



Question 2 (35 points)

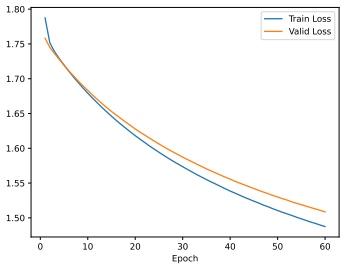
Landscape classification with an automatic differentiation toolkit. In Question 1, you implemented gradient backpropagation manually. Now, you will implement the same classification system using a deep learning framework with automatic differentiation. Skeleton code for PyTorch is provided in `hw1-q2.py`. If you prefer a different framework, you may use it instead.

1. (10 points) Implement a linear model using logistic regression and stochastic gradient descent for training. Use a `batch_size` of 32 and set the ℓ_2 regularization (`l2_decay`) parameter to 0.01. Train your model for 100 `epochs` and tune the `learning_rate` by evaluating the following values: $\{0.00001, 0.001, 0.1\}$.

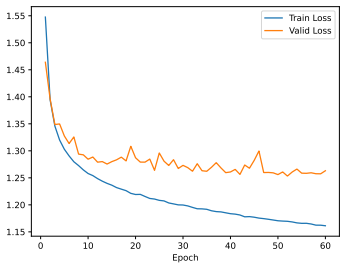
   Identify the learning rate that achieved the highest validation accuracy, reporting corresponding test and validation accuracy. Plot the training and validation losses for each configuration, and compare the differences between the validation loss curves, explaining these differences by relating to the learning rate values.

In the provided skeleton code, you will need to implement the `train_batch()` function as well as the `__init__()` and `forward()` methods of the `LogisticRegression` class.

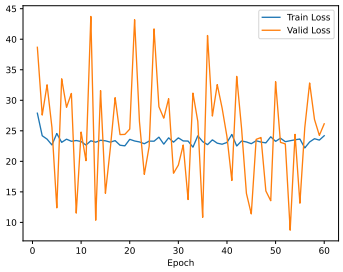
Solution: The best configuration was achieved with a learning rate of 0.001, resulting in a validation accuracy of 0.5313 and a test accuracy of 0.5263.



lr = 0.00001



lr = 0.001



lr = 0.1

Using lr = 0.00001 results in a smoother loss curve due to the smaller step size. However, this choice slows down convergence, resulting in a higher loss value after 100 epochs than that attained with lr = 0.001.

For lr = 0.1, the loss curve is very oscillating and presents no downward trend, as the high learning rate causes each update to move too far in the direction of the gradient.

The best results are attained by lr = 0.001, which provides the best tradeoff between stability in the training process and convergence speed.

2. (25 points) In this exercise, you will implement a feed-forward neural network. Your implementation should follow the hyperparameters and training/model design choices detailed in Table 1, which should be used by default.

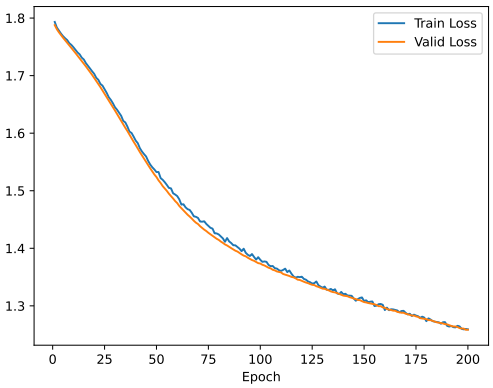
Number of Epochs	200
Learning Rate	0.002
Hidden Size	200
Number of Layers	2
Dropout	0.3
Batch Size	64
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 1: Default hyperparameters.

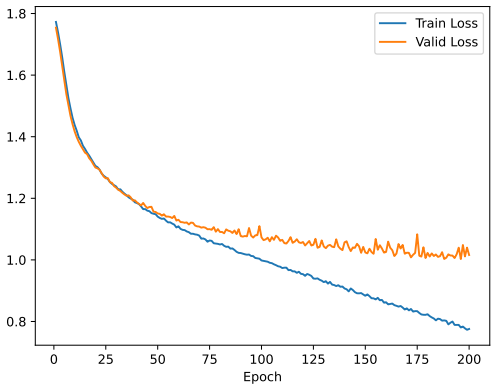
In the skeleton code, you will need to implement the class `FeedforwardNetwork`'s `__init__()` and `forward()` methods.

(a) (8 points) Train 2 models: one using the default hyperparameters and another with `batch_size` 512 and the remaining hyperparameters at their default value. Plot the train and validation losses for both, report the test and validation accuracy for both and explain the differences in both performance and time of execution.

Solution:



`batch size` = 512



`batch size` = 64

For the default model, the test accuracy was 0.6003 and the validation accuracy was 0.6061.

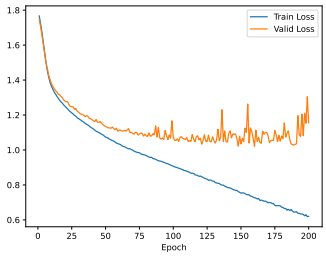
For the model with batch size 512, the test accuracy was 0.5203 and the validation accuracy was 0.4979.

Training time is over two times higher for the default model (batch size 64). The model with batches of size 64 does 8 times more updates than the one with batch size 512. Using a batch size of 512 leads to the faster computation time but results in lower performance for the same number of epochs since the model converges slower.

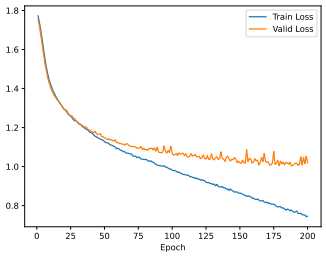
(b) (9 points) Train the model setting `dropout` to each value in $\{0.01, 0.25, 0.5\}$ while keeping all other hyperparameters at their default values. Report the final validation and test accuracies and plot the training and validation losses for the three configurations. Analyze and explain the results.

Solution:

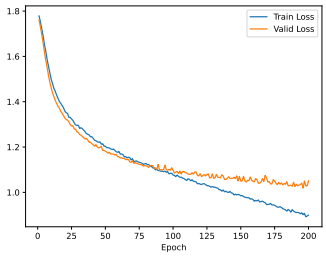
For `dropout` = 0.01, the test accuracy was 0.5777 and the validation accuracy was 0.5776. For `dropout` = 0.25, the test accuracy was 0.6063 and the validation accuracy was 0.6125. For a `dropout` = 0.50, the the test accuracy was 0.5863 and the validation accuracy was 0.5990.



`dropout` = 0.01



`dropout` = 0.25



`dropout` = 0.5

By using dropout regularization, we are randomly selecting a percentage of the neurons not to be used.

A higher value for dropout, represents a higher percentage of the neurons not being used.

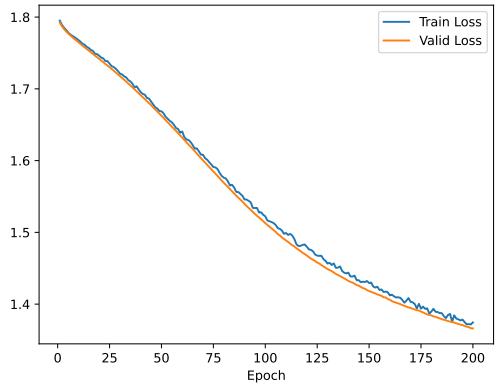
A dropout rate of 0.01 shows more overfitting than the other two configurations, as indicated by the bigger gap between the validation and test loss curves, since only 1

A dropout rate of 0.5 provides the stronger regularization effect of the three configurations, yielding the smallest gap between the validation and test losses. However, this regularizing effect seems to be excessive, as both validation and test accuracies are lower than those achieved with a smaller dropout value of 0.25.

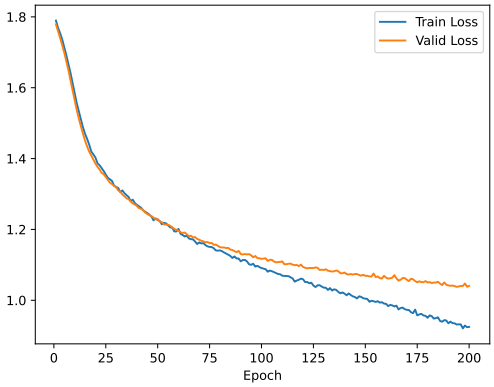
The intermediate value of 0.25 achieves the optimal balance between model regularization and capacity, leading to the best validation and test accuracies.

(c) (8 points) Using a `batch_size` of 1024, train the default model while setting the `momentum` parameter to each value in {0.0; 0.9} (use the `-momentum` flag). For the two configurations, plot the train and validation losses and report the test and validation accuracies. Explain the differences in performance.

Solution:



`momentum` = 0.0



`momentum` = 0.9

For a momentum of 0.9, the test accuracy was 0.5950 and the validation accuracy was 0.5990. For a momentum of 0.0, the test accuracy was 0.4780 and the validation accuracy was 0.4644.

Momentum takes into account the update that was made in the previous step and combines it into the computation of the current update.

Momentum reduces the update in directions with changing gradients (correct descent direction if the new gradient direction was different from the one in the previous update) and increases the update in directions with stable gradient (reinforces the descent update, if the new gradient has the same direction as the one in the previous update). This creates a smoother descent through the loss landscape, leading to faster convergence - reaching the same values of loss with less training epochs.

Inspecting the plots, we note that, after 200 epochs, the model with 0.0 momentum has reached loss values that the model with 0.9 momentum reaches after only around 25 epochs, confirming that the model with 0.9 momentum is indeed converging faster.

After the same number of training epochs, the model with faster convergence naturally reaches lower values for loss and higher values for accuracy, which is corroborated by the plots.

Question 3 (30 points)

Multi-layer perceptron with activations $g(z) = z(1-z)$. In this exercise, we will consider a feed-forward neural network with a single hidden layer and the activation function $g(z) = z(1-z)$. We will see that, under some assumptions, this choice of activation, unlike other popular activation functions such as tanh, sigmoid, or relu, can be tackled as a linear model via a reparametrization.

We assume a univariate regression task, where the predicted output $\hat{y} \in \mathbb{R}$ is given by $\hat{y} = \boldsymbol{v}^\top \boldsymbol{h} + v_0$, where $\boldsymbol{h} \in \mathbb{R}^K$ are internal representations, given by $\boldsymbol{h} = \boldsymbol{g}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, $\boldsymbol{x} \in \mathbb{R}^D$ is a vector of input variables, and $\Theta = (\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{v}, v_0) \in \mathbb{R}^{K\times D} \times \mathbb{R}^K \times \mathbb{R}^K \times \mathbb{R}$ are the model parameters.

1. (10 points) Show that we can write $\boldsymbol{h} = \boldsymbol{A}_\Theta \boldsymbol{\phi}(\boldsymbol{x})$ for a certain feature transformation $\boldsymbol{\phi} : \mathbb{R}^D \to \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ independent of Θ, where $\phi_1(\boldsymbol{x}) = 1$ (i.e., the first feature is a constant feature), and $\boldsymbol{A}_\Theta \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}}$. That is, $\boldsymbol{h}$ is a **linear transformation** of $\boldsymbol{\phi}(\boldsymbol{x})$. Determine the mapping $\boldsymbol{\phi}$ and the matrix $\boldsymbol{A}_\Theta$.

Solution: Let $\boldsymbol{w}_i^\top$ be the ith row of $\boldsymbol{W}$ and b_i the ith entry of $\boldsymbol{b}$. We have

$$\begin{aligned} h_i &= g(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i) = \boldsymbol{w}_i^\top \boldsymbol{x} + b_i - (\boldsymbol{w}_i^\top \boldsymbol{x} + b_i)^2 \\ &= \boldsymbol{w}_i^\top \boldsymbol{x} + b_i - \boldsymbol{x}^\top \boldsymbol{w}_i \boldsymbol{w}_i^\top \boldsymbol{x} - 2 b_i \boldsymbol{w}_i^\top \boldsymbol{x} - b_i^2 \\ &= (1 - 2b_i) \boldsymbol{w}_i^\top \boldsymbol{x} - \langle\!\langle \boldsymbol{w}_i \boldsymbol{w}_i^\top, \boldsymbol{x}\boldsymbol{x}^\top \rangle\!\rangle + b_i - b_i^2, \end{aligned} \tag{1}$$

where $\langle\!\langle\rangle\!\rangle$ denotes the Frobenius inner product $\langle\!\langle \boldsymbol{A}, \boldsymbol{B} \rangle\!\rangle = \mathrm{vec}(\boldsymbol{A})^\top \mathrm{vec}(\boldsymbol{B}) = \mathrm{Tr}(\boldsymbol{A}^\top \boldsymbol{B})$. Let

$$\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_D \\ x_1^2 \\ \vdots \\ x_D^2 \\ 2x_1x_2 \\ \vdots \\ 2x_{D-1}x_D \end{bmatrix}, \quad \boldsymbol{a}_i = \begin{bmatrix} b_i - b_i^2 \\ w_{i1}(1-2b_i) \\ \vdots \\ w_{iD}(1-2b_i) \\ -w_{i1}^2 \\ \vdots \\ -w_{iD}^2 \\ -w_{i1}w_{i2} \\ \vdots \\ -w_{i(D-1)}w_{iD} \end{bmatrix}, \quad \boldsymbol{A}_\Theta = \begin{bmatrix} \boldsymbol{a}_1^\top \\ \vdots \\ \boldsymbol{a}_K^\top \end{bmatrix}. \tag{2}$$

We then have $h_i = \boldsymbol{a}_i^\top \boldsymbol{\phi}(\boldsymbol{x})$ and $\boldsymbol{h} = \boldsymbol{A}_\Theta \boldsymbol{\phi}(\boldsymbol{x})$. Note that the dimensionality of $\boldsymbol{\phi}(\boldsymbol{x})$ is $1 + D + D + D(D-1)/2 = \frac{D^2+3D+2}{2} = \frac{(D+1)(D+2)}{2}$.

2. (5 points) Based on the previous claim, show that $\hat{y}$ is a linear transformation of $\boldsymbol{\phi}(\boldsymbol{x})$, i.e., we can write $\hat{y}(\boldsymbol{x}; \boldsymbol{c}_\Theta) = \boldsymbol{c}_\Theta^\top \boldsymbol{\phi}(\boldsymbol{x})$ for some $\boldsymbol{c}_\Theta \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$. Determine $\boldsymbol{c}_\Theta$. Does this mean this is a linear model in terms of the original parameters Θ? (Note: you can answer this question even if you have not solved the previous exercise.)

Solution: We have $\hat{y} = \boldsymbol{v}^\top \boldsymbol{h} + v_0 = \boldsymbol{v}^\top \boldsymbol{A}_\Theta \boldsymbol{\phi}(\boldsymbol{x}) + v_0 = \boldsymbol{c}_\Theta^\top \boldsymbol{\phi}(\boldsymbol{x})$, with $\boldsymbol{c}_\Theta = \boldsymbol{A}_\Theta^\top \boldsymbol{v} + v_0 \boldsymbol{e}_1$, where $\boldsymbol{e}_1 = [1, 0, ..., 0]$. The resulting model is **not** linear in Θ, because $\boldsymbol{c}_\Theta$ is not a linear function of Θ. Some entries of $\boldsymbol{A}_\Theta$ are **quadratic** – not linear – in terms of $\boldsymbol{W}$.

3. (10 points) Assume $K \geq D$. Show that for any real vector $\boldsymbol{c} \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ and any $\epsilon > 0$ there is a choice of the original parameters $\Theta = (\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{v}, v_0)$ such that $\|\boldsymbol{c}_\Theta - \boldsymbol{c}\| < \epsilon$. That is, up to ϵ-precision **we can equivalently parametrize the model with $\boldsymbol{c}$ instead of Θ.** Describe a procedure to obtain Θ from $\boldsymbol{c}$. (Hint: use orthogonal decomposition of a certain matrix assumed non-singular. Use the fact that any matrix is ϵ-close to a non-singular matrix.)

Solution: Let $\mathrm{vech}(\boldsymbol{M})$ denote the half-vectorization of a matrix $\boldsymbol{M}$, i.e., the vector formed by collecting the elements in the lower triangular part of $\boldsymbol{M}$. We can write

$$\boldsymbol{c}_\Theta = \boldsymbol{A}_\Theta^\top \boldsymbol{v} + v_0 \boldsymbol{e}_1 = \begin{bmatrix} c_1 \\ \boldsymbol{c}_2 \\ \boldsymbol{c}_3 \end{bmatrix}, \tag{3}$$

where

$$
\begin{aligned}
c_1 &= [\boldsymbol{b} \odot (\mathbf{1} - \boldsymbol{b})]^\top \boldsymbol{v} + v_0 \in \mathbb{R}, \\
\boldsymbol{c}_2 &= \boldsymbol{W}^\top [(\mathbf{1} - 2\boldsymbol{b}) \odot \boldsymbol{v}] \in \mathbb{R}^D, \\
\boldsymbol{c}_3 &= \text{vech}(\boldsymbol{W}^\top \text{Diag}(\boldsymbol{v}) \boldsymbol{W}) \in \mathbb{R}^{\frac{D(D+1)}{2}}.
\end{aligned} \tag{4}
$$

We now develop a procedure to obtain Θ from $\boldsymbol{c}$. Given $\boldsymbol{c}_3$, we form the symmetric matrix $\boldsymbol{M} \in \mathbb{R}^{D \times D}$ such that $\boldsymbol{c}_3 = \text{vech}(\boldsymbol{M})$. We can assume $\boldsymbol{M}$ is non-singular since any singular matrix is ϵ-close to a non-singular one. Let $\boldsymbol{M} = \boldsymbol{Q}\text{Diag}(\boldsymbol{\lambda})\boldsymbol{Q}^\top$ be an eigendecomposition of $\boldsymbol{M}$, where $\boldsymbol{\lambda}$ is the vector of eigenvalues (all of them nonzero, since $\boldsymbol{M}$ is non-singular) and $\boldsymbol{Q}$ an orthogonal matrix with the corresponding eigenvectors as columns. Then we can "pad $\boldsymbol{Q}$ and $\boldsymbol{\lambda}$ with zeros" and take $\boldsymbol{W} = \begin{bmatrix} \boldsymbol{Q}^\top \\ \mathbf{0}_{K-D,D} \end{bmatrix}$ and $\boldsymbol{v} = \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{0}_{K-D} \end{bmatrix}$.

We now use the equation for $\boldsymbol{c}_2$ to determine $\boldsymbol{b}$. Since $\boldsymbol{c}_2 = \boldsymbol{W}^\top [(\mathbf{1} - 2\boldsymbol{b}) \odot \boldsymbol{v}] \in \mathbb{R}^D$, we obtain

$$
\boldsymbol{b} = \begin{bmatrix} \frac{1}{2}[\mathbf{1} - \boldsymbol{\lambda}^{-1} \odot \boldsymbol{Q}^\top \boldsymbol{c}_2] \\ \mathbf{0}_{K-D} \end{bmatrix}. \tag{5}
$$

Finally, we use the equation for c_1 to obtain v_0.

An equivalent parametrization might not exist if $K < D$, since in that case $\hat{\boldsymbol{c}}_\Theta$ may yield a matrix $\boldsymbol{M}$ with rank greater than K, from which one cannot obtain $\boldsymbol{W}$ and $\boldsymbol{v}$ as desired.

4. (5 points) Suppose we are given training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$ with $N > \frac{(D+1)(D+2)}{2}$ and where all input vectors $\{\boldsymbol{x}_n\}$ are linearly independent, and that we want to minimize the squared loss

$$
L(\boldsymbol{c}_\Theta; \mathcal{D}) = \frac{1}{2} \sum_{n=1}^{N} (\hat{y}(\boldsymbol{x}_n; \boldsymbol{c}_\Theta) - y_n)^2.
$$

Can we find a closed form solution $\hat{\boldsymbol{c}}_\Theta$? Comment on the fact that global minimization is usually intractable for feedforward neural networks – what makes our problem special? (Note: you can answer this question even if you have not solved the previous exercises.)

Solution: Let $\boldsymbol{X} \in \mathbb{R}^{N \times \frac{(D+1)(D+2)}{2}}$ have $\boldsymbol{\phi}(\boldsymbol{x}_n)$ as rows, and define $\boldsymbol{y} = (y_1, \ldots, y_N)$. We want to minimize $\frac{1}{2}\|\boldsymbol{X}\boldsymbol{c}_\Theta - \boldsymbol{y}\|_F^2$ with respect to $\boldsymbol{c}_\Theta$. This is a least squares problem whose solution is $\hat{\boldsymbol{c}}_\Theta = \boldsymbol{X}^+ \boldsymbol{y} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$. This is a global optimum. What makes our problem special is the assumption that the activation functions are quadratic. This approach would not work in general for other non-linear activations.