Lab 4

Welcome to the fourth lab. In this lab, we will derive the backpropagation equations, code the training procedure, and test it on the XOR problem. Additionally, there will be a couple of theoretical questions about weight decay (or L2 regularization) that should give you some intuition on how it works.

Exercise 1

Derive the back-propagation algorithm. You might find the results of the second exercise of the previous lab a useful reference.

- Consider a neural network with L layers and a loss function $\mathcal{L}$ composed of a generic error term $\mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})$ and weight decay term $\mathcal{R}_\lambda(\mathbf{W})$. Call the output of the ℓ-th layer $\mathbf{y}^\ell = \phi_\ell(\mathbf{z}^\ell)$ with $\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{y}^{\ell-1} + \mathbf{b}^\ell$ its pre-activation output. Finally, consider a vector $\delta^\ell = \nabla_{\mathbf{z}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ containing the gradient of the loss with respect to the pre-activation outputs of layer ℓ.
- Compute δ^L.
- Compute $\nabla_{\mathbf{W}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ in terms of $\mathbf{y}^{\ell-1}$ and δ^ℓ.
- Compute $\nabla_{\mathbf{b}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ in terms of $\mathbf{y}^{\ell-1}$ and δ^ℓ.
- Compute $\delta^{\ell-1}$ from δ^ℓ.
- Use vectorized operations (i.e. operations with vectors and matrices): they will make your code in the next exercise *much* faster!
- Optional: Extend the vectorized operations to handle data in batches. Call $\mathbf{Y}^\ell$ and Δ^ℓ the matrices whose i-th rows contains the activations $\mathbf{y}^\ell$ and deltas δ^ℓ of the i-th training example in the batch.
- Hint: make sure that the results have the right shape. The deltas should be vectors, and the gradients should have the same shape as the respective parameters.

Solution

Note that backpropagation is not needed on the regularization term, since it can be differentiated directly with respect to the weights.

Note that $\delta_i^\ell = \partial \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}}) / \partial z_i^\ell$. By applying the chain rule, we have, for the last layer:

$$\delta_i^L = \frac{\partial \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial y_i^L} \cdot \frac{\partial y_i^L}{\partial z_i^L} \tag{1}$$

$$= \frac{\partial \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial y_i^L} \cdot \phi_L'(y_i^L) \tag{2}$$

Where the first term depends on the loss function. By the same reasoning, the derivatives of the parameters of a generic layer ℓ are:

$$\frac{\partial \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial w_{ij}^\ell} = \frac{\partial \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial z_i^\ell} \cdot \frac{\partial z_i^\ell}{\partial w_{ij}^\ell} + \frac{\partial \mathcal{R}(w_{ij}^\ell)}{\partial w_{ij}^\ell} = \delta_i^\ell \cdot y_j^{\ell-1} + \frac{\lambda}{2} w_{ij}^\ell \tag{3}$$

$$\frac{\partial \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial b_j^\ell} = \frac{\partial \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial z_i^\ell} \cdot \frac{\partial z_i^\ell}{\partial b_j^\ell} = \delta_i^\ell \tag{4}$$

As for the deltas:

$$\delta_i^{\ell-1} = \sum_j \frac{\partial \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial y_i^{\ell-1}} \cdot \frac{\partial y_i^{\ell-1}}{\partial z_i^{\ell-1}} = \phi_{\ell-1}'(y_i^{\ell-1}) \cdot \sum_j \delta_j^\ell \cdot w_{ji}^\ell$$

In vectorized form:

$$\delta^L = \nabla_{\mathbf{y}^L} \mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}}) \odot \phi'_L(\mathbf{y}^L) \tag{5}$$

$$\delta^{\ell-1} = \left(\delta^\ell \mathbf{W}^\ell\right) \odot \phi'_{\ell-1}(\mathbf{y}^{\ell-1}) \tag{6}$$

$$\nabla_{\mathbf{W}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}}) = \left(\mathbf{y}^{\ell-1}\right) \otimes \delta^{\ell^T} + \frac{\lambda}{2}\mathbf{W}^\ell \tag{7}$$

$$\nabla_{\mathbf{b}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}}) = \delta^\ell \tag{8}$$

with $\odot$ denoting the element-wise product of two vectors and $\otimes$ the outer product. For two vectors $\mathbf{a}$ and $\mathbf{b}$, if $\mathbf{c} = \mathbf{a} \odot \mathbf{b}$ and $\mathbf{D} = \mathbf{a} \otimes \mathbf{b}^T$, then $c_i = a_i \cdot b_i$ and $D_{ij} = a_i \cdot b_j$.

We now extend these formulas to handle batched data. $\nabla_{\mathbf{Y}^L} \mathcal{L}(\mathbf{Y}^L, \hat{\mathbf{Y}})$ and $\phi'_L(\mathbf{Y}^L)$ are both matrices of the same size, and the element-wise product still works as intended:

$$\Delta^L = \nabla_{\mathbf{Y}^L} \mathcal{L}(\mathbf{Y}^L, \hat{\mathbf{Y}}) \odot \phi'_L(\mathbf{Y}^L)$$

As for $\Delta^{\ell-1}$, its i-th row $\delta_i^{\ell-1}$ is computed using the i-th row of Δ^ℓ, δ_i^ℓ: we are trying to transform a matrix of size $N \times n^\ell$ into a matrix of size $N \times n^{\ell-1}$. Since $\mathbf{W}^\ell$ is of size $n^\ell \times n^{\ell-1}$, dimensional analysis suggests we can just swap the order of the matrix multiplication (as we did for the forward pass in dense layers), and keep the element-wise product as before:

$$\Delta^{\ell-1} = \left(\left(\mathbf{W}^\ell\right)^T \Delta^\ell\right) \odot \phi'_{\ell-1}(\mathbf{Y}^{\ell-1})$$

Indeed, the i-th row of the result is the dot-product of the i-th row of Δ^ℓ with each column of $\mathbf{W}^\ell$, followed by the element-wise multiplication with $\phi'_{\ell-1}$ computed on the i-th row of $\mathbf{Y}^\ell$, exactly as before.

The gradient of $\mathbf{W}^\ell$ is a bit more involved to compute, as it results in a three-dimensional tensor: the first dimension is for the samples, the second dimension is for the neurons of the ℓ-th layer, and the third dimension for the neurons of the $(\ell-1)$-th layer. In other words, the element indexed by i, j, k is the derivative of the loss of the i-th sample in the batch with respect to w_{jk}:

$$\left(\nabla_{\mathbf{W}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})\right)_{ijk} = \frac{\partial \mathcal{L}(\mathbf{Y}_i^L, \hat{\mathbf{Y}}_i)}{\partial w_{jk}^\ell} = \delta_{ij}^\ell \cdot \mathbf{Y}_{ik}^{\ell-1} + \frac{\lambda}{2} w_{jk}^\ell$$

Which is the outer product of the i-th row of Δ^ℓ with the (transposed) i-th row of $\mathbf{Y}^{\ell-1}$. This vectorized operation is not exactly standard, and cannot be done in plain R, but other languages/frameworks allow this type of operations.

Finally, the biases are straightforward:

$$\nabla_{\mathbf{b}^\ell} \mathcal{L}(\mathbf{Y}^L, \hat{\mathbf{Y}}) = \Delta^\ell$$

Note that we have to sum over the first dimension of the gradients for $\mathbf{W}^\ell$ and $\mathbf{b}^\ell$ to obtain the average of the gradient over the batch.

Exercise 2

In this exercise, we will code the backpropagation algorithm and apply it to a very simple example: the XOR problem. We will use reference classes to keep the code modular and organized.

1. Create a class that computes the binary cross entropy and its derivative.

2. Create a class that computes the ReLU activation and its derivative.
3. Create a class that computes the sigmoid activation and its derivative.
4. Create a class that computes the tanh activation and its derivative (which is $1 - \tanh(x)^2$)
5. Create a class that performs one step of gradient descent.
6. Write a function for the forward pass and for backpropagation.
7. Create a neural network with a suitable architecture and train it on the XOR problem.
8. Visualize the weights and biases of the trained network. Can you explain how the network makes predictions?
9. Test different activation functions, learning rates, initializations, number of layers and their sizes.
10. Optional: implement the softmax activation, the categorical cross entropy loss function, and train a network on the MNIST dataset.

Hint: you can check that the gradients you compute are correct by comparing them with the "empirical" gradients obtained through the finite differences method.

Hint: If you do this exercise in a R script, you will be able to debug the code and see where things are not going according to plan.

Note: all these functions should process the data in batches, i.e. matrices where every row is a different sample of the same batch.

```
loss_function = setRefClass(  # base class for loss functions
  "loss_function",
  methods = list(
    forward = function(y_true, y_pred) NA,
    backward = function(y_true, y_pred) NA
  )
)

binary_crossentropy = setRefClass(
  "binary_crossentropy",
  contains = "loss_function",
  methods = list(
    forward = function(y_true, y_pred) {
      # we add a small constant to avoid having a NaN result from log(0)
      -sum(
        y_true * log(y_pred + 1e-9) + (1 - y_true) * log(1 - y_pred + 1e-9)
      ) / nrow(y_true)
    },
    backward = function(y_true, y_pred) {
      # we add a small constant to avoid dividing by zero
      -(
        y_true / (y_pred + 1e-9) - (1 - y_true) / (1 - y_pred + 1e-9)
      ) / nrow(y_true)
    }
  )
)

y_true = matrix(c(0, 0, 0, 1, 1, 0), ncol=3)
y_pred = matrix(c(0.2, 0.5, 0.3, 0.8, 0.1, 0.1), ncol=3)
loss = binary_crossentropy()

loss$forward(y_true, y_pred)
```

```
## [1] 1.952027
```

```
loss$backward(y_true, y_pred)
```

```
##       [,1]       [,2]       [,3]
## [1,] 0.625  0.7142857 -5.0000000
## [2,] 1.000 -0.6250000  0.5555556
```

```
activation = setRefClass(  # base class for activation functions
  "activation",
  methods = list(
    forward = function(x) NA,
    backward = function(x) NA
  )
)


relu = setRefClass(
  "relu",
  contains = "activation",
  methods = list(
    forward = function(x) {
      ifelse(x > 0, x, 0)
    },
    backward = function(x) {
      ifelse(x > 0, 1, 0)
    }
  )
)

x = matrix(c(-0.1, 0.3, 0.7, 0.5, -1.0, 0.7), ncol=3)
act = relu()

act$forward(x)
```

```
##      [,1] [,2] [,3]
## [1,]  0.0  0.7  0.0
## [2,]  0.3  0.5  0.7
```

```
act$backward(x)
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    0
## [2,]    1    1    1
```

```
sigmoid = setRefClass(
  "sigmoid",
  contains = "activation",
  methods = list(
    forward = function(x) {
      1 / (1 + exp(-x))
    },
    backward = function(x) {
      exp(-x) / (1 + exp(-x))^2
    }
  )
)
```

```
x = matrix(c(2, 0, -2, 0.5, -0.25, 0.25), ncol=3)
act = sigmoid()

act$forward(x)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.8807971 0.1192029 0.4378235
## [2,] 0.5000000 0.6224593 0.5621765
```

```
act$backward(x)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.1049936 0.1049936 0.2461341
## [2,] 0.2500000 0.2350037 0.2461341
```

```
htan = setRefClass(
  "htan",
  contains = "activation",
  methods = list(
    forward = function(x) {
      tanh(x)
    },
    backward = function(x) {
      1 - tanh(x)^2
    }
  )
)

act = htan()

act$forward(x)
```

```
##           [,1]       [,2]       [,3]
## [1,] 0.9640276 -0.9640276 -0.2449187
## [2,] 0.0000000  0.4621172  0.2449187
```

```
act$backward(x)
```

```
##            [,1]       [,2]      [,3]
## [1,] 0.07065082 0.07065082 0.9400148
## [2,] 1.00000000 0.78644773 0.9400148
```

```
gradient_descent_optimizer = setRefClass(
  "gradient_descent_optimizer",
  fields = list(
    learning_rate = "numeric"
  ),
  methods = list(
    step = function(x, gradient) {
      x - learning_rate * gradient
    }
  )
)

opt = gradient_descent_optimizer(learning_rate = 0.1)
opt$step(10, 10)
```

```
## [1] 9
```

```
dense_neural_network = setRefClass(
  "dense_neural_network",
  fields = list(
    weights = "list",
    biases = "list",
    activations = "list",
    loss = "loss_function",
    optimizer = "gradient_descent_optimizer"
  ),
  methods = list(

    predict = function(batch_x) {
      result = batch_x

      for(i in 1:length(weights)) {
        bias_matrix = matrix(
          rep(biases[[i]], nrow(batch_x)),
          nrow = nrow(batch_x), byrow = TRUE
        )

        result = result %*% weights[[i]] + bias_matrix
        result = activations[[i]]$forward(result)
      }

      result
    },

    train_on_batch = function(batch_x, batch_y, iter, lrate) {
      intermediate_activations = list(batch_x)

      result = batch_x
      for(i in 1:length(weights)) {
        bias_matrix = matrix(
          rep(biases[[i]], nrow(batch_x)),
          nrow = nrow(batch_x), byrow = TRUE
        )

        result = result %*% weights[[i]] + bias_matrix
        result = activations[[i]]$forward(result)
        intermediate_activations[[i + 1]] = result
      }

      batch_loss = loss$forward(
        batch_y,
        intermediate_activations[[length(intermediate_activations)]]
      )

      weight_gradients = list()
      bias_gradients = list()

      output = intermediate_activations[[length(intermediate_activations)]]
      for(i in 1:nrow(batch_x)) {
```

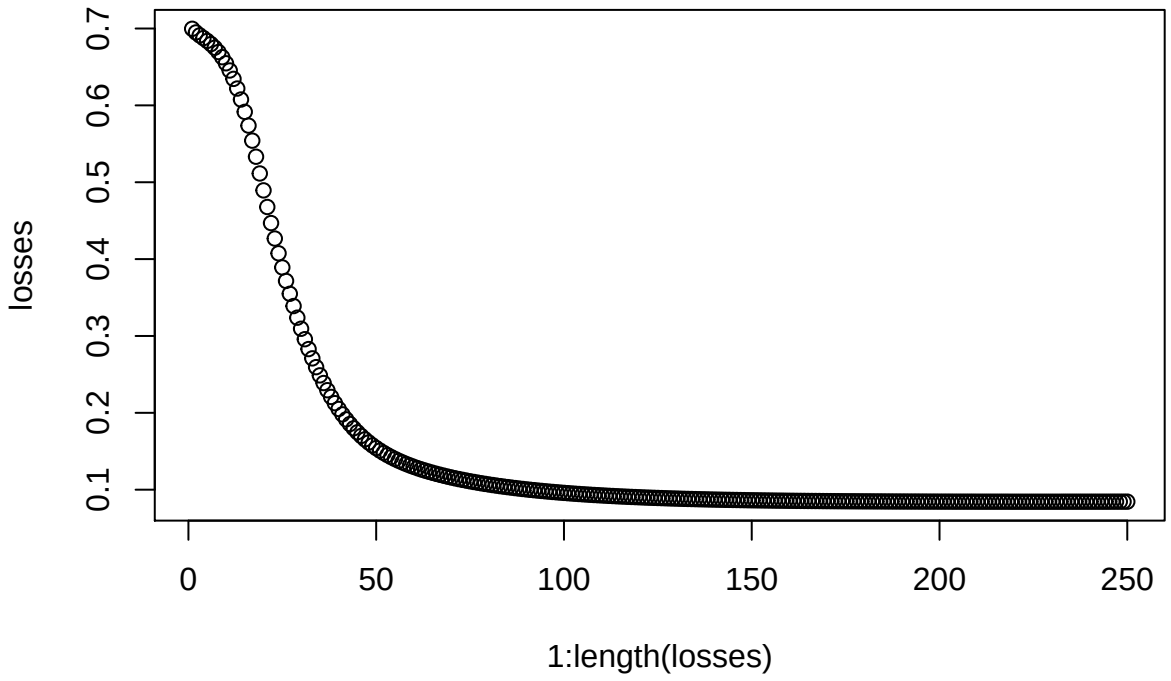
```
        deltas = (loss$backward(
          batch_y[i,,drop=FALSE],
          output[i,,drop=FALSE]
        ) * activations[[length(activations)]]$backward(
          output[i,,drop=FALSE]
        ))[1,]

        for(j in length(activations):1) {
          stopifnot(any(!is.na(deltas)))

          wg = intermediate_activations[[j]][i,] %o% deltas
          bg = deltas

          if(length(weight_gradients) > 0 && !is.null(weight_gradients[[j]])) {
            weight_gradients[[j]] = weight_gradients[[j]] + wg
            bias_gradients[[j]] = bias_gradients[[j]] + bg
          }
          else {
            weight_gradients[[j]] = wg
            bias_gradients[[j]] = bg
          }

          if(j > 1) {
            deltas = (
              weights[[j]] %*% deltas *
                activations[[j - 1]]$backward(intermediate_activations[[j]][i,])
            )

            # keep as vector, not matrix, so that the outer product
            # will be a matrix instead of 3d tensor
            deltas = deltas[,1]
          }
        }
      }

      for(i in 1:length(activations)) {
          weights[[i]] <<- optimizer$step(weights[[i]], weight_gradients[[i]])
          biases[[i]] <<- optimizer$step(biases[[i]], bias_gradients[[i]])
      }

      batch_loss
    }
  )
)


# just an utility function to create networks
build_dense_neural_network = function(input_size, layers, loss, optimizer) {
  weights = list()
  biases = list()
  activations = list()

  last_layer_size = input_size
```

```
  for(i in 1:length(layers)) {
    if(class(layers[[i]]) == "numeric") {
      sd = sqrt(2 / (last_layer_size + layers[[i]]))
      vals = rnorm(n = last_layer_size * layers[[i]], mean = 0, sd = sd)
      vals = ifelse(vals > 2 * sd, 2 * sd, vals)
      vals = ifelse(vals < -2 * sd, -2 * sd, vals)
      weights[[length(weights) + 1]] = matrix(
        vals, ncol = layers[[i]], nrow = last_layer_size
      )
      biases[[length(biases) + 1]] = rep(0, layers[[i]])
      last_layer_size = layers[[i]]
    }
    else {
      activations[[length(activations) + 1]] = layers[[i]]
    }
  }

  dense_neural_network(
    weights = weights,
    biases = biases,
    activations = activations,
    loss = loss,
    optimizer = optimizer
  )
}


data.x = matrix(c(
  -1, -1,
  -1, 1,
  1, -1,
  1, 1
), nrow = 4, ncol = 2, byrow = TRUE)


data.y = matrix(c(
  0, 1, 1, 0
), nrow = 4, ncol = 1)


network = build_dense_neural_network(
  input_size = 2,
  layers = list(2, htan(), 1, sigmoid()),
  loss = binary_crossentropy(),
  optimizer = gradient_descent_optimizer(learning_rate = 0.25)
)


losses = lapply(1:250, function(i) network$train_on_batch(data.x, data.y))
network$predict(data.x)
```

```
##              [,1]
## [1,] 6.863932e-24
## [2,] 8.446034e-01
```

```
## [3,] 8.446042e-01
## [4,] 6.863932e-24
```

```
plot(1:length(losses), losses)
```




```
network$weights
```

```
## [[1]]
##           [,1]     [,2]
## [1,] -3.142662 -3.14223
## [2,]  3.142662  3.14223
##
## [[2]]
##           [,1]
## [1,]  27.66006
## [2,] -27.66012
```

```
network$biases
```

```
## [[1]]
## [1] -3.290147  3.289672
##
## [[2]]
## [1] 1.831051
```

Exercise 3

This exercise should improve your understanding of weight decay (or L2 regularization).

1. Consider a quadratic error function $E(\mathbf{w}) = E_0 + \mathbf{b}^T\mathbf{w} + 1/2 \cdot \mathbf{w}^T\mathbf{H}\mathbf{w}$ and its regularized counterpart $E'(\mathbf{w}) = E(\mathbf{w}) + \tau/2 \cdot \mathbf{w}^T\mathbf{w}$, and let $\mathbf{w}^*$ and $\tilde{\mathbf{w}}$ be the minimizers of E and E' respectively. We want to find a formula to express $\tilde{\mathbf{w}}$ as a function of $\mathbf{w}^*$, i.e. find the displacement introduced by weight decay.
    - Find the gradients of E and E'. Note that, at the global minimum, we have $\nabla E(\mathbf{w}^*) = \nabla E'(\hat{\mathbf{w}}) = 0$.
    - In the equality above, express $\mathbf{w}^*$ and $\tilde{\mathbf{w}}$ as a linear combination of the eigenvectors of $\mathbf{H}$.

- Through algebraic manipulation, obtain $\tilde{\mathbf{w}}_i$ as a function of $\mathbf{w}_i^*$.
- Interpret this result geometrically.
- Note: $\mathbf{H}$ is square, symmetric, and positive definite, which means that its eigenvectors are perpendicular, and its eigenvalues are positive.

2. Consider a linear network of the form $y = \mathbf{w}^T\mathbf{x}$ and the mean squared error as a loss function. Assume that every observation is corrupted with Gaussian noise $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$. Compute the expectation of the gradient under ϵ and, show that adding gaussian noise to the inputs has the same effect of weight decay.

Solution

Question 1

The error is computed as:

$$E(\mathbf{w}) = E_0 + \sum_i w_i b_i + \frac{1}{2}\sum_i\sum_j w_i w_j h_{ij}$$

The derivative with respect to w_i is, then:

$$\frac{\partial E}{\partial w_i} = b_i + \sum_j w_j h_{ij}$$

Where the factor 1/2 was removed since the pair w_i and w_j is multiplied together twice, and $h_{ij} = h_{ji}$. In vector form:

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w}$$

The same reasoning applied to E' yields:

$$\nabla_{\mathbf{w}} E'(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w} + \tau\mathbf{w}$$

Now let $\mathbf{u}_i$ and λ_i be the eigenvectors and eigenvalues of $\mathbf{H}$, so that $\mathbf{H}\mathbf{u}_i = \lambda_i\mathbf{u}_i$. Any vector $\mathbf{v}$ can then be expressed as $\mathbf{v} = \sum_i \gamma_i\mathbf{u}_i$. Now, note that

$$\mathbf{H}\mathbf{v} = \sum_i \gamma_i \mathbf{H}\mathbf{u}_i = \sum_i \gamma_i\lambda_i\mathbf{u}_i$$

Moreover, at the global minimum, both gradients equal zero, hence:

$$\mathbf{b} + \underbrace{\sum_i \alpha_i\lambda_i\mathbf{u}_i}_{\mathbf{H}\mathbf{w}^*} = \mathbf{b} + \underbrace{\sum_i \beta_i\lambda_i\mathbf{u}_i}_{\mathbf{H}\hat{\mathbf{w}}} + \tau\underbrace{\sum_i \beta_i\mathbf{u}_i}_{\hat{\mathbf{w}}} \iff \sum_i \left(\alpha_i\lambda_i - \beta_i\lambda_i - \tau\beta_i\right)\mathbf{u}_i = \mathbf{0}$$

Since the eigenvectors are pairwise orthogonal, the above expression is zero only when each term inside the sum is zero, i.e.

$$\alpha_i\lambda_i - \beta_i\lambda_i - \tau\beta_i = 0 \iff \beta_i = \frac{\lambda_i}{\lambda_i + \tau}\alpha_i$$

Now, by replacing this into the expression for $\hat{\mathbf{w}}$, we get:

$$\tilde{\mathbf{w}} = \sum_i \beta_i \mathbf{u}_i = \sum_i \frac{\lambda_i}{\lambda_i + \tau} \alpha_i \mathbf{u}_i = \mathbf{d} \odot \mathbf{w}^*$$

With $\odot$ being the element-wise product, and $d_i = \lambda_i(\lambda_i + \tau)^{-1}$.

The eigenvalues of $\mathbf{H}$ indicate how much the error changes by moving in the direction of the corresponding eigenvector, with larger changes associated to smaller eigenvalues. In light of this, the formula above is saying that the largest changes are applied to the weights that have little influence on the error, while "important" weights are not perturbed much.

Question 2

The prediction for $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon$ is:

$$\tilde{y} = \mathbf{w}^T (\mathbf{x} + \epsilon) = \mathbf{w}^T \mathbf{x} + \mathbf{w}^T \epsilon$$

The error of this sample is

$$\tilde{E} = \frac{1}{2} \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right)^2$$

And its gradient with respect to a single weight is

$$\begin{aligned} \frac{\partial \tilde{E}}{\partial w_i} &= \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right) (-x_i - \epsilon_i) \\ &= -x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right) - \epsilon_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right) \end{aligned}$$

The expectation with respect to ϵ is

$$\begin{aligned} \mathbb{E}\left[\frac{\partial \tilde{E}}{\partial w_i} \right] &= \mathbb{E}\left[-x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right) \right] + \mathbb{E}\left[-\epsilon_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon \right) \right] \\ &= -x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} \right) + \mathbb{E}\left[-\epsilon_i \hat{y} + \epsilon_i \mathbf{w}^T \mathbf{x} + \epsilon_i \mathbf{w}^T \epsilon \right] \\ &\overset{*}{=} \frac{\partial E}{\partial w_i} + \sum_j w_j \mathbb{E}\left[\epsilon_i \epsilon_j \right] \\ &= \frac{\partial E}{\partial w_i} + w_i \sigma^2 \end{aligned}$$

Where we used $\partial E / \partial w_i$ to denote the gradient of the error of the de-noised sample, and the step marked with $*$ follows because $\mathbb{E}[\epsilon_i \epsilon_j] = \mathrm{Cov}\left[\epsilon_i, \epsilon_j\right] = \delta_{ij} \sigma^2$.

Clearly, the gradient is the same that results from weight decay.