COMS 4030A

Adaptive Computation and Machine Learning

2.16. **Cross-entropy loss.**

The cross-entropy loss function can be used in the case that a one-hot encoding of the targets is used. It requires the use of new activation function, $softmax$, which is defined as follows.

Suppose a network has k output nodes $n_1, \ldots, n_k$. For each i, let z_{n_i} be the variable for the weighted sum of inputs plus the bias at node n_i (as describe in Lecture 3).
The **softmax activation function** at node n_i is calculated as follows:

$$softmax(n_i) = \frac{e^{z_{n_i}}}{\sum_{j=1}^{k} e^{z_{n_j}}}.$$

Note that the output at a particular output node depends on the z values at all output nodes. The computation of the outputs is done by first calculating $e^{z_{n_1}}, \ldots, e^{z_{n_k}}$, then the sum $T = \sum_{j=1}^{k} e^{z_{n_j}}$ and then $softmax(n_i) = \frac{e^{z_{n_i}}}{T}$ for each i.

If $softmax$ is used as the activation function at the output layer then $0 \leq softmax(n_i) \leq 1$ for each output node n_i and

$$\sum_{i=1}^{k} softmax(n_i) = 1.$$

This means that the output values form a probability distribution over the output nodes and that $softmax(n_i)$ can be interpreted as the probability that node n_i is the correct output.

Example. Suppose a network has three output nodes and for some input into the network the following values are obtained: $z_{n_1} = 1.5$, $z_{n_2} = -0.5$ and $z_{n_3} = 0.75$. Then

$$e^{z_{n_1}} = e^{1.5} = 4.48, \quad e^{z_{n_2}} = e^{-0.5} = 0.61, \quad e^{z_{n_3}} = e^{0.75} = 2.12.$$

so $T = 4.48 + 0.61 + 2.12 = 7.21$, hence

$$softmax(n_1) = \tfrac{4.48}{7.21} = 0.62, \quad softmax(n_2) = \tfrac{0.61}{7.21} = 0.08, \quad softmax(n_3) = \tfrac{2.12}{7.21} = 0.30.$$

Thus, there is a probability of 0.62 that the classification is $(1, 0, 0)$, a probability of 0.08 that it's $(0, 1, 0)$ and a probability of 0.30 that it's $(0, 0, 1)$.

The $softmax$ activation function provides a probability distribution over the output nodes. Since the network uses a one-hot encoding of targets, the targets also form a probability distribution over the output nodes. For example, if the target is $(0, 1, 0)$, then with probability 1 the target is the classification associated with n_2 and with probability 0 it's the classification associated with node n_1 or n_3.

With the outputs and the targets both probability distributions over the output nodes, the cross-entropy function for comparing two probability distributions may be used. Let $\boldsymbol{y} = (y_1, \ldots, y_k)$ be a probability distribution (i.e., $0 \leq y_i \leq 1$ for each i, and $\sum_{j=1}^{k} y_j = 1$), and let $\boldsymbol{t} = (t_1, \ldots, t_k)$ also be a probability distribution. The **cross-entropy** between $\boldsymbol{t}$ and $\boldsymbol{y}$ is

$$H(\boldsymbol{t}, \boldsymbol{y}) = -\sum_{i=1}^{k} t_i \ln(y_i).$$

The value of $H(\boldsymbol{t}, \boldsymbol{y})$ is a measure of similarity between the two probability distributions. It is always the case that $H(\boldsymbol{t}, \boldsymbol{y}) \geq 0$ and the closer that $H(\boldsymbol{t}, \boldsymbol{y})$ is to 0 the more similar the distributions are. (Note that we define $0 \ln 0 = 0$.)

Consider a neural network that uses $softmax$ activation function at the output layer and a dataset that uses one-hot encoding of targets. For a datapoint $\boldsymbol{x}$ that has corresponding target $\boldsymbol{t}$ and produces an output of $\boldsymbol{y}$ from the network, the **cross-entropy loss** is defined as

$$L_{CE}(\boldsymbol{t}, \boldsymbol{y}) = -\sum_{i=1}^{k} t_i \ln(y_i).$$

EXERCISES:

(1) Suppose a neural network has five output nodes $n_1, \ldots, n_5$.
For each i, compute $softmax(n_i)$ if the z values at the output nodes are:
(a) $z_{n_1} = 3$, $z_{n_2} = 5$, $z_{n_3} = 0.5$, $z_{n_4} = -2$ and $z_{n_5} = 1.7$;
(b) $z_{n_1} = -2$, $z_{n_2} = -5$, $z_{n_3} = -0.5$, $z_{n_4} = 0.1$ and $z_{n_5} = -1.5$;
(c) $z_{n_1} = 0$, $z_{n_2} = 0.2$, $z_{n_3} = -0.1$, $z_{n_4} = 0.1$ and $z_{n_5} = -0.7$.

(2) Compute $L_{CE}(\boldsymbol{t}, \boldsymbol{y})$ for the following probability distributions:
(i) $\boldsymbol{y} = (0.5, 0.3, 0.2)$ and $\boldsymbol{t} = (1, 0, 0)$;
(ii) $\boldsymbol{y} = (0.5, 0.3, 0.2)$ and $\boldsymbol{t} = (0, 1, 0)$;
(iii) $\boldsymbol{y} = (0.5, 0.3, 0.2)$ and $\boldsymbol{t} = (0, 0, 1)$;
(iv) $\boldsymbol{y} = (0.1, 0.2, 0.7)$ and $\boldsymbol{t} = (0.3, 0.3, 0.4)$;

2.17. **Gradient descent with cross-entropy loss.**

We consider the case of the Neural Network Training Algorithm in which the cross-entropy loss function is used.

When using the cross-entropy loss function, the $softmax$ activation function is used at the output layer, and the targets must be in one-hot encoding. The reason for these restrictions is that both the outputs and the targets should form a probability distribution over the output nodes.

Recall that the choice of loss function directly affects the calculation of δ values at the output nodes. In the following calculations the δ values for output nodes of a network are obtained for the cross-entropy loss. The combination of $softmax$ activation function and cross-entropy loss function give a nice and simple expression for the δ values at the output nodes.

As before, let $\boldsymbol{W}$ be the list of all current weights in the network. Suppose that $\boldsymbol{x}$ is an input to the network and the feedforward step with input $\boldsymbol{x}$ has been completed and output $\boldsymbol{y}$ has been obtained.

The cross-entropy loss is given by the following expression

$$L_{CE}(\boldsymbol{t}, \boldsymbol{y}) = -\sum_{\ell} t_\ell \ln(y_\ell),$$

where ℓ ranges over all output nodes.

Since the $softmax$ activation function is used at the output layer, for any output node ℓ we have that

$$y_\ell = softmax(z_\ell) = \frac{e^{z_\ell}}{\sum_m e^{z_m}},$$

where m ranges over all output nodes.

Thus, $L_{CE}(\boldsymbol{t}, \boldsymbol{y})$ can be written as:

$$L_{CE}(\boldsymbol{t}, \boldsymbol{y}) = -\sum_{\ell} t_\ell \ln\left(\frac{e^{z_\ell}}{\sum_m e^{z_m}}\right)$$

Then $L_{CE}(\boldsymbol{t}, \boldsymbol{y})$ can be simplified as follows:

$$
\begin{aligned}
L_{CE}(\boldsymbol{t}, \boldsymbol{y}) &= -\sum_{\ell} t_\ell \left(\ln e^{z_\ell} - \ln \sum_m e^{z_m} \right) \\
&= -\sum_{\ell} t_\ell \ln e^{z_\ell} + \sum_{\ell} t_\ell \ln \sum_m e^{z_m} \\
&= -\sum_{\ell} t_\ell z_\ell + \sum_{\ell} t_\ell \ln \sum_m e^{z_m} \\
&= -\sum_{\ell} t_\ell z_\ell + \left(\sum_{\ell} t_\ell \right) \left(\ln \sum_m e^{z_m} \right) \\
&= -\sum_{\ell} t_\ell z_\ell + \ln \sum_m e^{z_m} \quad (\text{since } \sum_{\ell} t_\ell = 1).
\end{aligned}
$$

The following derivation is used to calculate $\delta_n = \frac{\partial L_{CE}}{\partial z_n}\big|_{\boldsymbol{xtW}}$ for a fixed output node n.

In the summations, ℓ and m range over all output nodes, and n is one of those output nodes.

$$
\begin{aligned}
\tfrac{\partial L_{CE}}{\partial z_n} &= \tfrac{\partial}{\partial z_n} \left(-\sum_{\ell} t_\ell z_\ell + \ln \sum_m e^{z_m} \right) \\
&= -t_n + \tfrac{\partial}{\partial z_n} \left(\ln \sum_m e^{z_m} \right) \\
&= -t_n + \frac{1}{\sum_m e^{z_m}} \tfrac{\partial}{\partial z_n} \sum_m e^{z_m} \\
&= -t_n + \frac{e^{z_n}}{\sum_m e^{z_m}} \\
&= -t_n + y_n \quad (\text{since } y_n = softmax(z_n)).
\end{aligned}
$$

It follows that that $\frac{\partial L_{CE}}{\partial z_n}\big|_{\boldsymbol{xtW}} = (y_n - t_n)\big|_{\boldsymbol{xtW}} = a_n - t_n$ and, therefore, $\delta_n = a_n - t_n$.

EXERCISES

(1) Rewrite the pseudocode for the Neural Network Training Algorithm (with three layers) in such a way that the cross-entropy loss function is used.
At the hidden layer, you can use the σ activation function.

(2) Try the first exercise again, but use *relu* at the hidden layer; then again with *tanh*.

(3) Consider a network with 2 input nodes, one hidden layer with 2 nodes, and 2 output nodes. The weights and the bias values are given by W_1, W_2, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$:

$$W_1 = \begin{bmatrix} -2 & -1 \\ 3 & 0 \end{bmatrix} \quad W_2 = \begin{bmatrix} 2 & 3 \\ -1 & -2 \end{bmatrix} \quad \boldsymbol{b}_1 = (0.5, 1.5) \quad \boldsymbol{b}_2 = (2, -1).$$

The activation function in the hidden layer is sigmoid (or $relu$, or $tanh$).

The output layer uses $softmax$ and the targets are one-hot encoded.

Using cross-entropy loss with input $\boldsymbol{x} = (-1, 1)$ and target $\boldsymbol{t} = (1, 0)$, do the following:

(a) First feed the input into the network to get the output and compute the loss.

(b) Perform one iteration of backpropagation training with $\eta = 0.1$.

(c) Feed the input into the network again and see if the loss has decreased.

2.18. **Learning rate.**

The learning rate η is set by the user. It is an example of a **hyperparameter** (as opposed to the weights of the network, which are parameters).

The value of η is either fixed throughout the training process as a small value (usually between 0.001 and 0.01) or is set to **decay** during training. That is, after every epoch, η can be decreased slightly. For example, suppose the max number of epochs is set to K; then in epoch k, where $0 \leq k \leq K$, the learning rate can be set as

$$\eta_k = 0.01 - 0.009\tfrac{k}{K}.$$

In the above example, the learning rate decays linearly from 0.01 to 0.001. Any similar decaying function could be used. We can think of a decaying learning rate as allowing large changes to the weights initially to adjust them in the right direction and then, as training proceeds, allowing smaller changes to fine-tune the weights.

2.19. **Batch gradient descent.**

The neural network training algorithm described earlier performs the gradient descent weight update after a single input from the dataset is fed through the network. This method is sometimes called **stochastic gradient descent**, or **online gradient descent**.

In batch gradient descent a batch of inputs are fed through the network before any gradient descent update is done; that is, before any weights are changed.

If the batch used is the whole dataset, this is referred to as **batch gradient descent**.

If batches of size M are used, where M is less than the size of the dataset, this is referred to as **mini-batch gradient descent**. (So stochastic gradient descent is mini-batch with $M = 1$.)

The changes to the neural network training algorithm for batch gradient descent are described below.

Suppose the chosen batch size is M. (M is another hyperparameter.)
Randomize the inputs in your dataset and select the first M inputs.
For each input, feed it through the network and, once the output is obtained, compute the gradient for each edge weight $\frac{\partial L}{\partial w_{mn}}\big|_{\boldsymbol{xtW}}$ and the gradient for each bias $\frac{\partial L}{\partial b_n}\big|_{\boldsymbol{xtW}}$. This is done as in the case of stochastic gradient descent, using backpropagation to compute delta values and then computing:

$$\frac{\partial L}{\partial w_{mn}}\Big|_{\boldsymbol{xtW}} = a_m \delta_n \quad \text{and} \quad \frac{\partial L}{\partial b_n}\Big|_{\boldsymbol{xtW}} = \delta_n.$$

However, the updates to the weights are not done.
Rather, the gradient for each weight is stored. Then, after all M inputs have been fed through the network, the average of the gradients is computed, i.e., for each edge weight w_{mn}, compute

$$\text{AveGrad}\, w_{mn} = \frac{1}{M} \sum_{i=1}^{M} \frac{\partial L}{\partial w_{mn}}\Big|_{\boldsymbol{xtW}}^{i}$$

where $\frac{\partial L}{\partial w_{mn}}\big|_{\boldsymbol{xtW}}^{i}$ is the gradient computed for the i^{th} input. Similarly, for each bias b_n, compute

$$\text{AveGrad}\, b_n = \frac{1}{M} \sum_{i=1}^{M} \frac{\partial L}{\partial b_n}\Big|_{\boldsymbol{xtW}}^{i}.$$

Only then are the weights updated as follows:

$$\underline{w}_{mn} \leftarrow \underline{w}_{mn} - \eta\, \text{AveGrad}\, w_{mn} \quad \text{and} \quad \underline{b}_n \leftarrow \underline{b}_n - \eta\, \text{AveGrad}\, b_n.$$

Thereafter, select the next batch of M inputs and repeat the process.
An epoch is complete when all datapoints have been used as part of a batch.

Batch gradient descent is more computationally efficient than stochastic gradient descent and can have a more stable convergence since the updates are done less frequently. However, for the same reason, a greater number of epochs may be required for convergence, so convergence may be slower.

2.20. L_1- and L_2-Regularization.

In L_1- and L_2-regularization, the objective is to prevent individual weights from becoming excessively large (in absolute value).

For L_2-regularization, this is done by adding an extra term to whichever loss function L is used, as follows:

$$L_2(\boldsymbol{t}, \boldsymbol{y}) = L(\boldsymbol{t}, \boldsymbol{y}) + \lambda \tfrac{1}{2} \sum_{\text{all } w_{mn}} w_{mn}^2.$$

The constant λ is included to balance the two terms of the loss function L_2. If there are many edges in the network, then the sum of all the squares of edge weights may dominate the L term. To avoid this, the term is multiplied by some value λ, which is another example of a hyperparameter. Finding a suitable value for λ may require some **hyperparameter tuning** (which is essentially just experimenting with a range of possible values to see what works best).

The update rule for edge weight $\underline{w}_{mn}$ is

$$\underline{w}_{mn} \leftarrow \underline{w}_{mn} - \eta \left(\tfrac{\partial L_2}{\partial w_{mn}} \big|_{\boldsymbol{xtW}} \right)$$

where the partial derivative $\frac{\partial L_2}{\partial w_{mn}}$ is calculated as follows:

$$\begin{aligned} \tfrac{\partial L_2}{\partial w_{mn}} &= \tfrac{\partial L}{\partial w_{mn}} + \tfrac{\partial}{\partial w_{mn}} \lambda \tfrac{1}{2} \textstyle\sum_{\text{all } w_{mn}} w_{mn}^2 \\ &= \tfrac{\partial L}{\partial w_{mn}} + \lambda \tfrac{1}{2} \tfrac{\partial}{\partial w_{mn}} w_{mn}^2 \\ &= \tfrac{\partial L}{\partial w_{mn}} + \lambda \, w_{mn}. \end{aligned}$$

The update rule for edge weight $\underline{w}_{mn}$ is then

$$\underline{w}_{mn} \leftarrow \underline{w}_{mn} - \eta \left(\tfrac{\partial L}{\partial w_{mn}} \big|_{\boldsymbol{xtW}} \right) - \eta \lambda \, \underline{w}_{mn}$$

and the computation of $\frac{\partial L}{\partial w_{mn}} \big|_{\boldsymbol{xtW}}$ is done as before using the delta values.

Thus, the only change to the update rule for edge weights in the case of L_2-regularization is to subtract a small value from the weight if it is positive and to add a small value to the weight if it is negative. This prevents the weight values from drifting too far from 0. The bias values could also be included in the summation if required and a similar small change to the bias update is then also made.

Note that when calculating the L_2 loss for a given input with respect to its target, it is necessary to compute the sum $\sum_{\text{all } w_{mn}} w_{mn}^2$.

For L_1-regularization, the following term is added to the loss function L:

$$L_1(\boldsymbol{t}, \boldsymbol{y}) = L(\boldsymbol{t}, \boldsymbol{y}) + \lambda \sum_{\text{all } w_{mn}} |w_{mn}|.$$

The partial derivative $\frac{\partial L_1}{\partial w_{mn}}$ is calculated as follows:

$$\begin{aligned} \tfrac{\partial L_1}{\partial w_{mn}} &= \tfrac{\partial L}{\partial w_{mn}} + \tfrac{\partial}{\partial w_{mn}} \lambda \textstyle\sum_{\text{all } w_{mn}} |w_{mn}| \\ &= \tfrac{\partial L}{\partial w_{mn}} + \lambda \tfrac{\partial}{\partial w_{mn}} |w_{mn}| \\ &= \tfrac{\partial L}{\partial w_{mn}} + \lambda \, sign(w_{mn}) \end{aligned}$$

where $sign(w_{mn}) = 1$ if $w_{mn} > 0$, $sign(w_{mn}) = -1$ if $w_{mn} < 0$, and $sign(w_{mn}) = 0$ if $w_{mn} = 0$. (Recall that $\frac{d}{dx}|x| = 1$ if $x > 0$ and $\frac{d}{dx}|x| = -1$ if $x < 0$. The function $|x|$ is non-differentiable if $x = 0$, but we can just set it to 0 in this case.)

The update rule for edge weight $\underline{w}_{mn}$ is then

$$\underline{w}_{mn} \leftarrow \underline{w}_{mn} - \eta \left(\tfrac{\partial L}{\partial w_{mn}} \big|_{\boldsymbol{xtW}} \right) - \eta\lambda \, sign(\underline{w}_{mn})$$

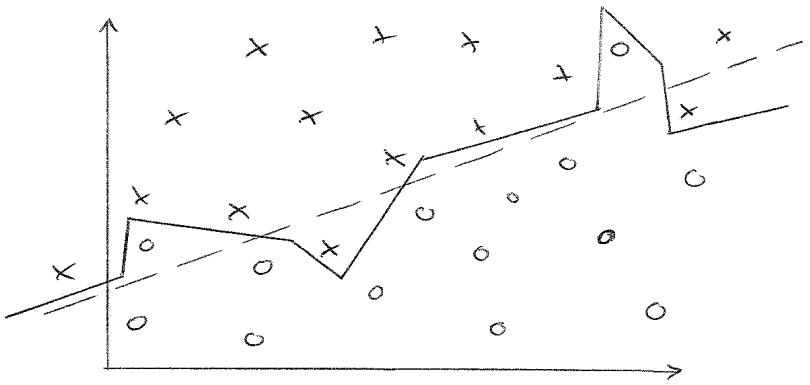
and the computation of $\frac{\partial L}{\partial w_{mn}} \big|_{\boldsymbol{xtW}}$ is done as before using the delta values.

As in L_2-regularization, the weights are pushed back towards 0 in each update step.
In the L_1 case, the weights are adjusted by a fixed value (i.e., by $\pm\eta\lambda$), whereas in the L_2 case, the weight adjustment is scaled according to the existing value of the weight (i.e., by $\eta\lambda\underline{w}_{mn}$).

2.21. **Avoiding Overtraining.**

Overtraining, or **overfitting**, of a model on a dataset means that the training algorithm has created a model that matches the data in the dataset too closely. If this happens, it can mean that the classifier is less accurate when used to classify previously unseen datapoints, i.e., datapoints that were not used for training but come from the same classification problem. In this case, we say that the model does not **generalise** well.

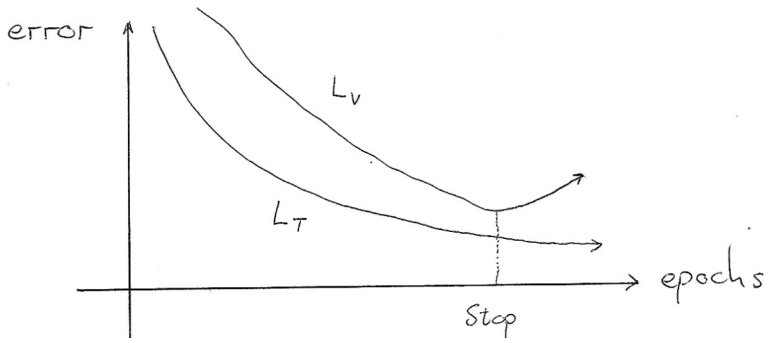
For example, in the following diagram, there is a natural separation of $\circ$'s from $\times$'s by a straight line, which has only a few misclassifications. However, if a neural network with many layers is used then, after training, the network could result in a classifier with a decision boundary that looks like the jagged line. As can be seen, the jagged line matches the training data too well, including all the 'outliers', and will misclassify many unseen points.



A way to avoid overtraining is through the following approach. From the original dataset, separate off a portion of the datapoints (say about 25%), called the **validation dataset**. Keep the remaining datapoints as the **training dataset** which is then used to train the weights of the neural network (or whatever model is being used). During training, after every epoch, calculate the loss of the network on both the training dataset and the validation dataset using the same loss function L that is used for training.

Let L_T denote the loss on the training dataset.
Let L_V denote the loss on the validation dataset.

If we plot L_T and L_V over the course of the training, the graphs should look something like:



Initially both L_T and L_V will decrease during training, but overtraining starts when L_V starts to increase while L_T continues to decrease. This means that the network is matching the training dataset too well and is starting to misclassify points in the validation dataset. Thus, to avoid overtraining we should stop training at the point when L_V starts to increase.

At this point it is useful to be able to test how good your network (or your model) is. In order to do this, another dataset of test datapoints that have not been used up to now is required. Thus, at the start, one should split the dataset into 3 sets:

training dataset (for training the weights)
validation dataset (to avoid overtraining and to tune hyperparameters)
test dataset (to test the effectiveness of the neural network/model).

A rough guide for splitting the data is: 50% training, 25% validation and 25% testing.
The datapoints in each dataset should be randomly chosen from the initial dataset.

2.22. **Data Normalisation.**

Consider a dataset that has attributes with very different ranges, such as:

$$\begin{matrix} x_1 & x_2 & x_3 & x_4 \end{matrix} \qquad t$$
$$\begin{bmatrix} 29 & 0 & 100000 & 8 \\ 31 & 1 & 500000 & 6 \\ 24 & 1 & 250000 & 3 \\ \vdots & \vdots & \cdots & \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix}$$

If the first datapoint from this dataset is input into a network then the activation value at a node in the first hidden layer would look like:

$$g(29w_1 + 0w_2 + 100000w_3 + 8w_4)$$

where g is the activation function. In the sum, the term $100000w_3$ dominates the other terms and the activation value at this node does not rely on the other input values. During training, it should happen eventually that w_3 adjusts to a small number and the other weights grow to large numbers so that all inputs contribute to the activation value. This can increase the training time substantially. To avoid this, it is useful to preprocess the data before starting training. One way to do this is to rescale the data so that each column's values have a similar range. This is called **normalising** or **standardising** the data. There are two commonly-used methods for doing this, which we describe below.

1) **max-min normalisation**:

Take the values in a column of data corresponding to some attribute, say $\begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix}$.

Let max be the maximum value in $\{d_1, \ldots, d_N\}$ and min the minimum value in $\{d_1, \ldots, d_N\}$, and define the function:

$$f(d) = \frac{d - min}{max - min}.$$

Apply the above function to every d_i in the column to get $\begin{bmatrix} f(d_1) \\ \vdots \\ f(d_N) \end{bmatrix}$.

Note that every value in the column now lies in the interval $[0, 1]$.

Also note that you can recover the original data values by applying the function

$$f^{-1}(x) = (max - min)x + min.$$

2) **mean-standard deviation normalisation**:

Consider a column $\begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix}$ again.

First calculate the **mean**, denoted μ, of the values in $\{d_1, \ldots, d_N\}$:

$$\text{mean} = \mu = \frac{1}{N} \sum_{i=1}^{N} d_i$$

and then calculate the **variance** σ^2:

$$\text{variance} = \sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (d_i - \mu)^2 = \frac{1}{N} \left(\sum_{i=1}^{N} d_i^2 \right) - \mu^2$$

and, finally, calculate the **standard deviation**: $\sigma = \sqrt{\sigma^2}$.
Then define the function:

$$f(d) = \frac{d - \mu}{\sigma}.$$

Apply the above function to every d_i in the column to get $\begin{bmatrix} f(d_1) \\ \vdots \\ f(d_N) \end{bmatrix}$.

Note that after applying the function f to the column, the new values in the column have a mean of 0 and a standard deviation of 1 (please check this).

Also, the original values can be recovered using the function: $f^{-1}(x) = \sigma x + \mu$.

2.23. **Categorical Data.**

Given a dataset, each column (i.e., each attribute of the data) can be normalised using one of the methods from above if the values in the column are numerical. However, there may be attributes that do not use numerical values. For example, an attribute may have True or False values in it, or may have a number of different categories, such as A_1, A_2, ..., A_n. (Think of a column with "place of birth" entries.) These entries need to be converted into numerical values.

In the case of True/False, or if there are only two categories A_1 and A_2, then 1 and 0 can be used as the numerical values, or 1 and -1 if that is more natural.

If there are n categories, say $A_1, \ldots, A_n$, then a range of values can be used, such as:

0 for A_1,
$\frac{1}{n-1}$ for A_2,
$\vdots$
$\frac{i-1}{n-1}$ for A_i, for $1 \leq i \leq n$.

The above approach is only really appropriate if the categories $A_1, \ldots, A_n$ have a natural rank order. For example, the categories may correspond to grades for a course, as in F, E, D, C, B, A, in which case it makes sense that A has a greater value than B, which has value greater then C, and so on. Data of this type is called **ordinal data**.

If the categories do not have a natural rank order, then a better approach is to use a one-hot encoding of the categories. This means that the one column of data with the A_i's in it is replaced by n columns of data with 0's and 1's in it. An A_1 gets replaced by a 1 in the

first column and 0's elsewhere and, in general, an A_i gets replaced by a 1 in column i and 0's elsewhere.

For example, with 3 classes A_1, A_2 and A_3, the column $\begin{bmatrix} A_1 \\ A_3 \\ A_2 \\ A_1 \\ A_3 \end{bmatrix}$ is replaced by $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Note that a single column in the dataset is replaced by multiple columns so the size of the input vector is changed accordingly.

EXERCISES

(1) Suppose you have the following dataset with only 5 datapoints:

$$\begin{bmatrix} 6 & -24 & 125,000 & A & -0.001 & T \\ 9 & -31 & 175,000 & C & 0.0023 & T \\ 3 & -7 & 95,000 & A & -0.004 & F \\ 11 & -17 & 300,000 & B & 0.0045 & F \\ 4 & -11 & 250,000 & B & 0.003 & T \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ X \\ Z \\ Y \end{bmatrix}$$

Do the preprocessing of the data for both the inputs and the targets (using one-hot encoding). For the input values, try both methods of normalisation described above.

(2) A hospital manager wants to predict how many beds will be needed in the geriatric ward in a given week. He asks you to design a neural network method for making this prediction. He has data for the last five years that cover:

The number of people in the geriatric ward each week.

The weather (average day and night temperatures) for each day.

The season of the year (spring, summer, autumn, winter).

Whether or not there was an epidemic on.

Describe how you would set up the input and output data for this problem.

2.24. **A Note on Information Theory.**

Given a probability distribution over a set of events, the **self-information** of a certain event, say x, is defined as:

$$I(x) = -\log P(x)$$

where $P(x)$ is the probability of event x occurring. As $P(x)$ tends to 1, the value of $I(x)$ tends to 0, and if $P(x)$ tends to 0, then $I(x)$ grows large.
(The log can be of any base, but is usually $\log_2$ or ln.)

The intuition behind the above definition is that rare events, i.e., those with low probability, give more information than common events, i.e., those with high probability.

For example, suppose you are interested in which team won a certain soccer match - the blue or the red team. If you are told that the following event occurred: 'The blue team's player passed the ball to his teammate in the 85th minute', then the information acquired about who won the game is very low. However, if you are told that the following event occurred: 'The blue team's player scored a goal in the 85th minute', then the information acquired about who won the game is greater. The event of a player passing the ball is very common and has a high probability, while the event of scoring a goal has a low probability and gives much more information.

The **Shannon entropy** of a probability distribution P is defined as

$$H(P) = \mathbb{E}_{x\sim P}[I(x)]$$

which gives the expected amount of information obtained from an event drawn from the probability distribution P.

If P is a finite probability distribution, say it has k events $x_1, \ldots, x_k$, with $P(x_i) = p_i$ for each i, then

$$H(P) = \sum_{i=1}^{k} p_i I(x_i) = -\sum_{i=1}^{k} p_i \log p_i.$$

Cross-entropy is a quantity closely related to Shannon entropy, defined for two probability distributions P and Q as

$$H(P, Q) = \mathbb{E}_{x\sim P}[I_Q(x)].$$

The cross-entropy of P and Q is a measure of the expected amount of information about probability distribution Q obtained from an event drawn from probability distribution P. The

above expression simplifies to

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x).$$

If P and Q are finite distributions on a set of k events, say P is the probability distribution $(p_1, \ldots, p_k)$ and Q is the probability distribution $(q_1, \ldots, q_k)$, then

$$H(P, Q) = -\sum_{i=1}^{k} p_i \log(q_i)$$

which is the cross entropy used in Section 2.16.

Another metric for comparing two probability distributions P and Q is the **Kullback-Leibler divergence**, or **KL divergence**:

$$D_{KL}(P||Q) = \mathbb{E}_{x \sim P} \left[\log \frac{P(x)}{Q(x)} \right]$$

The value of $D_{KL}(P||Q)$ is zero if P and Q are the same distribution. The greater the value of $D_{KL}(P||Q)$, the more dissimilar P and Q are. The KL divergence has a number of applications in machine learning. The formula for KL divergence comes from the following:

$$D_{KL}(P||Q) = \mathbb{E}_{x \sim P} \left[I_Q(x) - I_P(x) \right]$$

which is the expected difference between the amount of information about probability distribution Q and the amount of information about probability distribution P, obtained from an event drawn from distribution P.