Deep Learning - Homework 2

Group 10

95572 Filipa Cotrim

95618 Leonor Barreiros

The entirety of the homework was done via zoom call. So both of us equally contributed to every question.

Question 1

Question 1.1

1.1 (a)

The matrix **z** ($\mathbf{z} = conv(\mathbf{W}, \mathbf{x})$) denotes the result of applying the filter **W** to **x** That is, it's the convolution matrix. So, its dimensions are:

- $width = \frac{x.width - filter.width + 2 \times padding.width}{stride} + 1 = W - N + 1 \equiv W'$
- $height = \frac{x.height - filter.height + 2 \times padding.height}{stride} + 1 = H - M + 1 \equiv H'$

1.1 (b)

From **(a)** we know that $\mathbf{z} \in R^{H' \times W'}$. So, possible vectorizations of **z** and **x** are:

- $\mathbf{z'} = \begin{bmatrix} z_{11} \\ \vdots \\ z_{1W'} \\ \vdots \\ z_{H'1} \\ \vdots \\ z_{H'W'} \end{bmatrix}$ with dimensions $H'W' \times 1$

- $\mathbf{x'} = \begin{bmatrix} x_{11} \\ \vdots \\ x_{1W} \\ \vdots \\ x_{H1} \\ \vdots \\ x_{HW} \end{bmatrix}$ with dimensions $HW \times 1$

Each entry of **z** is a linear combination of **W** and the appropriate slice of **x**. So, we have the following mapping: $z_{ab} = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{ij} x_{(i+a-1)(j+b-1)}$

Our goal in this exercise is to show that there exists a matrix $\mathbf{M} \in R^{H'W' \times HW}$ such that $\mathbf{z'} = \mathbf{M}\mathbf{x'}$. This product expresses the convolution operation (that is, the construction of **z**) as a matrix-vector product.

Each entry of **z'** corresponds to an entry of **z** and will be a linear combination of a line of **M** and **x'**. That means we also have the following mapping: $z_{ab} = \mathbf{m}_{W'(a-1)+b}\mathbf{x'} = \sum_{i=1}^{H} \sum_{j=1}^{W} m_{W'(a-1)+b, W(i-1)+j} x_{ij}$.

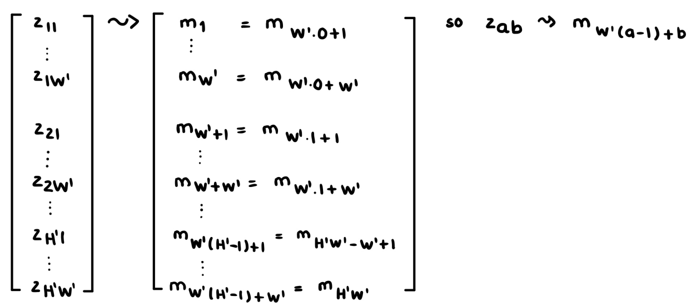

Figure 1: How we defined the mapping from each entry of **z'** to each line of **M**

The result entry of **z'** corresponds to applying the filter to the respective slice of **x** (represented by **x'**). So, the definition of each line of **M** will need to represent that. Bearing in mind that the indices a and b refer to each entry of **z'** , we have:

$$m_{W'(a-1)+b, W(i-1)+j} = \begin{cases} 0 & \text{if } \neg[(a \leq i \leq M + a - 1) \wedge (b \leq j \leq N + b - 1)] \\ w_{ij} & \text{if } otherwise \end{cases}$$

Defining **M** this way ensures we have **z'** = **Mx'** because each line of **M** will contain each w_{ij} in the respective position of each entry of **x'** that's in the $\mathbf{z}_{ab}$ convolution unit. Since all other entries in that line are 0, we guarantee that **Mx'** expresses the convolution operation.

We only defined M for one vectorization of **z** and **x**. However, every two definitions of vectorization (i.e., that put the elements in different orders) are related by $vec(\mathbf{y}) = \mathbf{P}vec'(\mathbf{y})$, where **P** is a permutation matrix. So, having solved the problem with the presented matrix **M** it is trivial to solve it for another pair of vectorizations with **M'** = $\mathbf{P}^T\mathbf{MP}$.

1.1 (c)

In our network we have the following parameters:

- In the convolutional layer we have MN parameters (the size of the filter, which is what we learn)
- In the max pooling layer we have no more parameters, because we do a fixed computation on the convolution matrix. However, we obtain a matrix h_2 with the following dimensions (we introduced the floor function for the case when the division yields a non-integer value, which isn't possible when talking about dimensions of a matrix):
  - $width = \lfloor \frac{z.width}{stride} \rfloor = \lfloor \frac{W-N+1}{2} \rfloor$
  - $height = \lfloor \frac{z.height}{stride} \rfloor = \lfloor \frac{H-M+1}{2} \rfloor$
- In the fully connected layer we learn a weight matrix that takes as input a vectorization of h_2 and has an output layer with 3 units (corresponding to each of the 3 possible classes): $W_{FC} \in R^{3\times(h_2.width \times h_2.height)}$. That's $3(\lfloor \frac{W-N+1}{2} \rfloor)(\lfloor \frac{H-M+1}{2} \rfloor)$ parameters

So, in total we have $MN + 3(\lfloor \frac{W-N+1}{2} \rfloor)(\lfloor \frac{H-M+1}{2} \rfloor)$ parameters in our network.

Now let's consider a network where we replace the convolution and max pooling layers with another fully connected layer (yielding an output with the same dimensions as h_2). We would have the following parameters:

- A weight matrix that takes as input a vectorization of **x** and has as output layer with $h_2.width + h_2.height$ units: $W'_{FC} \in R^{(h_2.width \times h_2.height)\times(HW)}$. That's $(\lfloor \frac{W-N+1}{2} \rfloor \times \lfloor \frac{H-M+1}{2} \rfloor) \times (HW)$ parameters
- The same $3(\lfloor \frac{W-N+1}{2} \rfloor)(\lfloor \frac{H-M+1}{2} \rfloor)$ in the following fully connected layer our network also has

So, in total we have $(\lfloor \frac{W-N+1}{2} \rfloor)(\lfloor \frac{H-M+1}{2} \rfloor)HW + 3(\lfloor \frac{W-N+1}{2} \rfloor)(\lfloor \frac{H-M+1}{2} \rfloor)$ parameters in this network.

Notice that, in the CNN, the number of parameters is quadratic in relation to the dimension of **x**. Notice also that, in the FFNN, the number of parameters is in the fourth degree in relation to the dimension of **x**. So, with increasing H and W, the number of parameters in the latter will become larget than in the former.

But how smaller is the number of parameters in our network? Let's see an example similar to the one we saw in Practical 8 with $H = W = 227$ and $M = N = 11$

- In our network we have $121 + 34992$ parameters
- In the other one we have $601034256 + 34992$ parameters

The other network has around 17118 times more parameters than ours. As we move to more realistic applications, where images are bigger and kernels are smaller, this difference becomes even clearer. In conclusion, because of CNN's parameter sharing, they end up having much fewer parameters to learn.

Question 1.2

We begin by defining the input matrix $\mathbf{X}$ whose rows contain each element of our sequence (i.e., each element of $\mathbf{x'}$). We trivially realize that (because each element of the sequence has size 1) $\mathbf{X} = \mathbf{x'}$.

To compute single-head self-attention on this input, we firstly do:

- Query matrix: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q = \mathbf{X} = \mathbf{x'}$
- Key matrix: $\mathbf{K} = \mathbf{X}\mathbf{W}_K = \mathbf{X} = \mathbf{x'}$
- Value matrix: $\mathbf{V} = \mathbf{X}\mathbf{W}_V = \mathbf{X} = \mathbf{x'}$

The attention probabilities $\mathbf{P}$ are giving by computing scaled dot-product attention and applying softmax row-wise. That is, $\mathbf{P} = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{1}) = softmax(\mathbf{x'} \cdot \mathbf{x'}^T)$.

The attention output is thus given by $\mathbf{Z} = \mathbf{P}\mathbf{V} = softmax(\mathbf{x'} \cdot \mathbf{x'}^T) \cdot \mathbf{x'}$

Question 2

Question 2.1

In a fully-connected network, each neuron applies a linear transformation to the input vector through a weights matrix. In this way, each neuron in a layer is connected to every neuron in the next layer, meaning all inputs affect all outputs. This type of neural network requires a large number of weights, since a different weight is used for every input feature. On the other hand, Convolutional Neural Networks are trained to identify and extract the best features from the images with fewer parameters. This extraction is done by using kernels (a type of filter) that scan and process the input and are shared across all inputs, thus making CNNs require fewer weights.

Question 2.2

CNNs are able to develop an internal representation of a 2D image, where the filter is shifted around at every convolutional step, allowing it to find patterns and match those patterns wherever their location is in the image (location invariance). This weights sharing and location invariance are very useful for object detection. Even if a pattern is tilted/rotated/cropped, the padding step of a CNN makes it able to recognize it, by down sampling the existing feature maps (invariance to local translation), making CNNs robust to changes. Additionally, the deeper the layer the more complex patterns are recognized, since each layer learns from the previous one (hierarchy). All these attributes allow CNNs to learn progressively more complex and abstract data, making it effective for image recognition.
A FCN does not preserve the image structure, images are pre-processed/flatten so that input pixels are received as a flat vector. There are no spatial patterns for correlations, since pixels have no relation to the image post-processing. As a consequence, there is no pattern recognition/matching. An image is processed like any other input, making FCNs not as effective as CNNs.

Question 2.3

As explained before, CNNs benefit significantly from their location invariance, from the fact that they take advantage of the spatial structure of the data in order to identify and match patterns and overall learning of useful features. In the case of independent sensors, the CNN loses this important ability and could make a FCN a better option, since this type of neural network can handle any input and no

assumption is made when learning the data.
Generally speaking, we agree a FCN will achieve a better generalization. However, this could change depending on the data and problem at hand.

Question 2.4

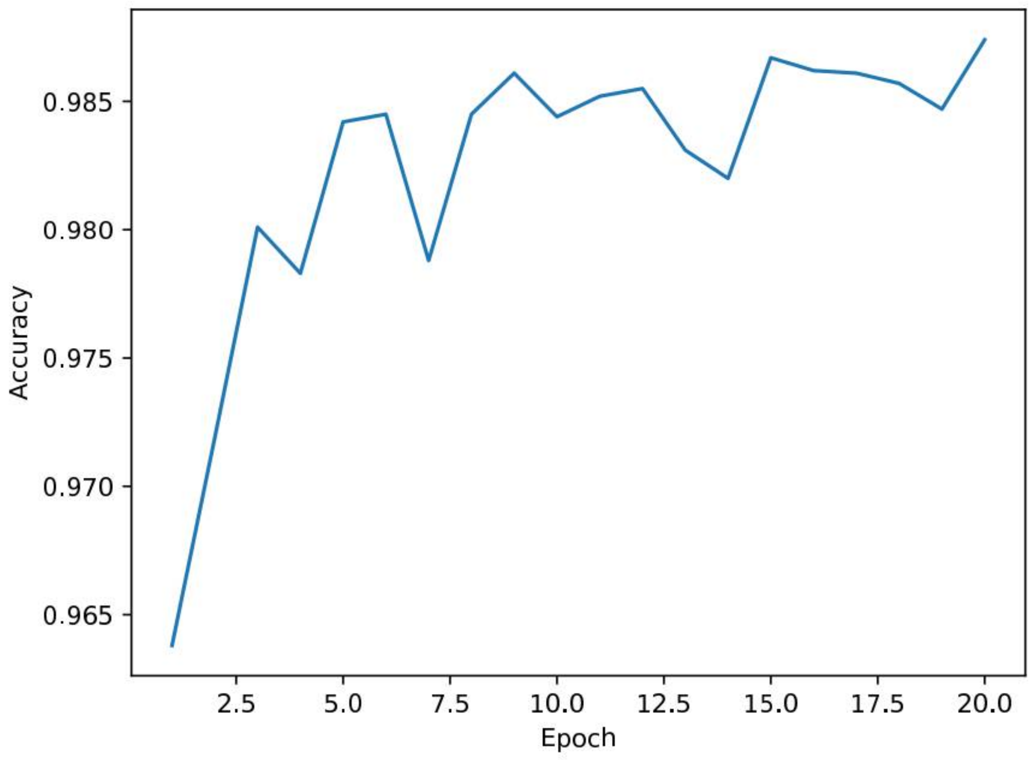

Figure 2: Validation accuracy of Adam optimizer with learning rate of 0.0005 and dropout of 0.3 as a function of the epoch number.

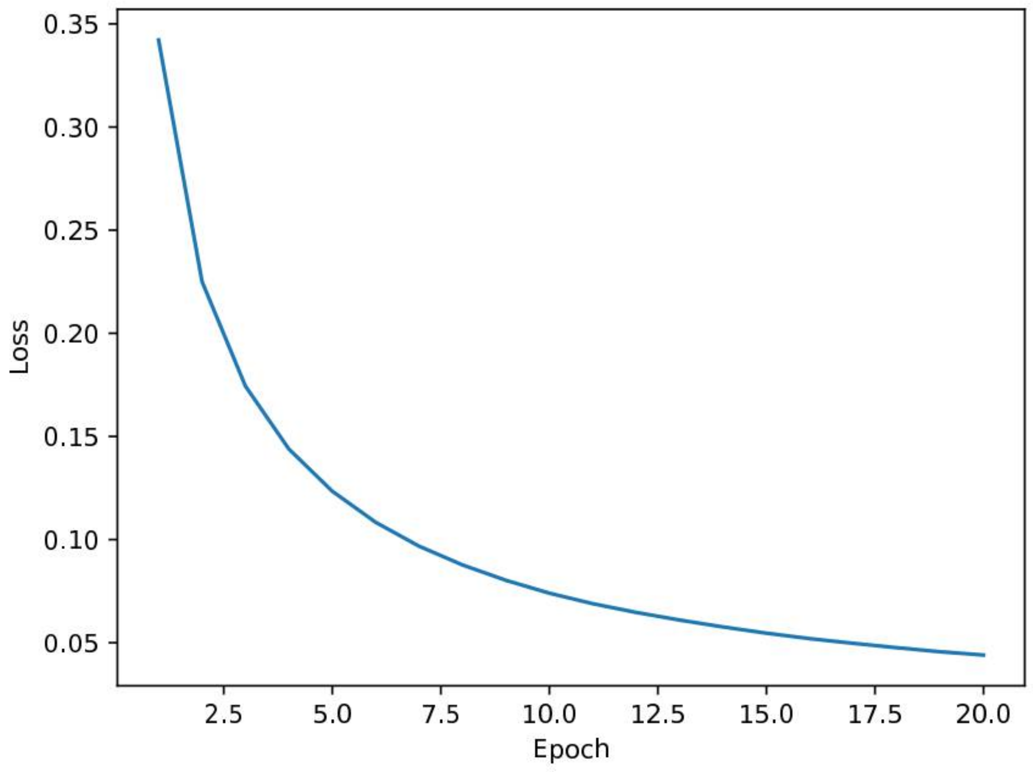

Figure 3: Training loss of Adam optimizer with learning rate of 0.0005 and dropout of 0.3 as a function of the epoch number.

Question 2.5

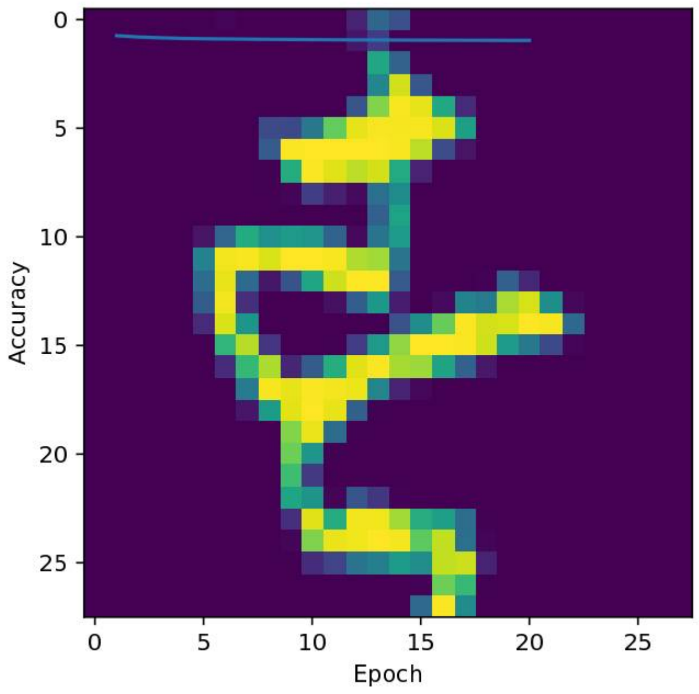

Figure 4: Original training example.

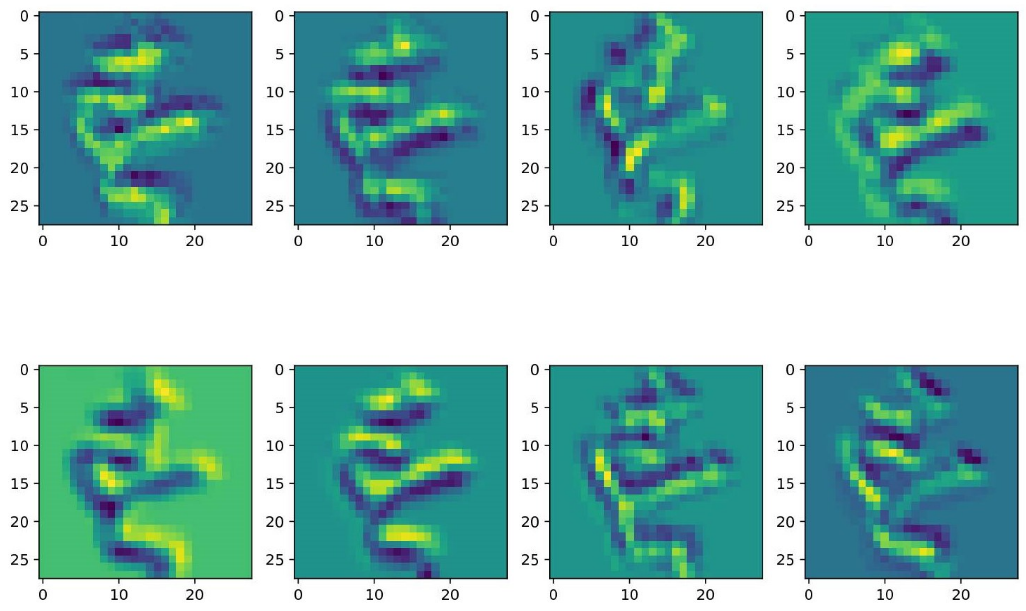

Figure 5: Activation map of first convolutional layer.
What appears to be highlighted in the activation maps are the pixels that form the character.

Question 3

Question 3.1

3.1 (a)

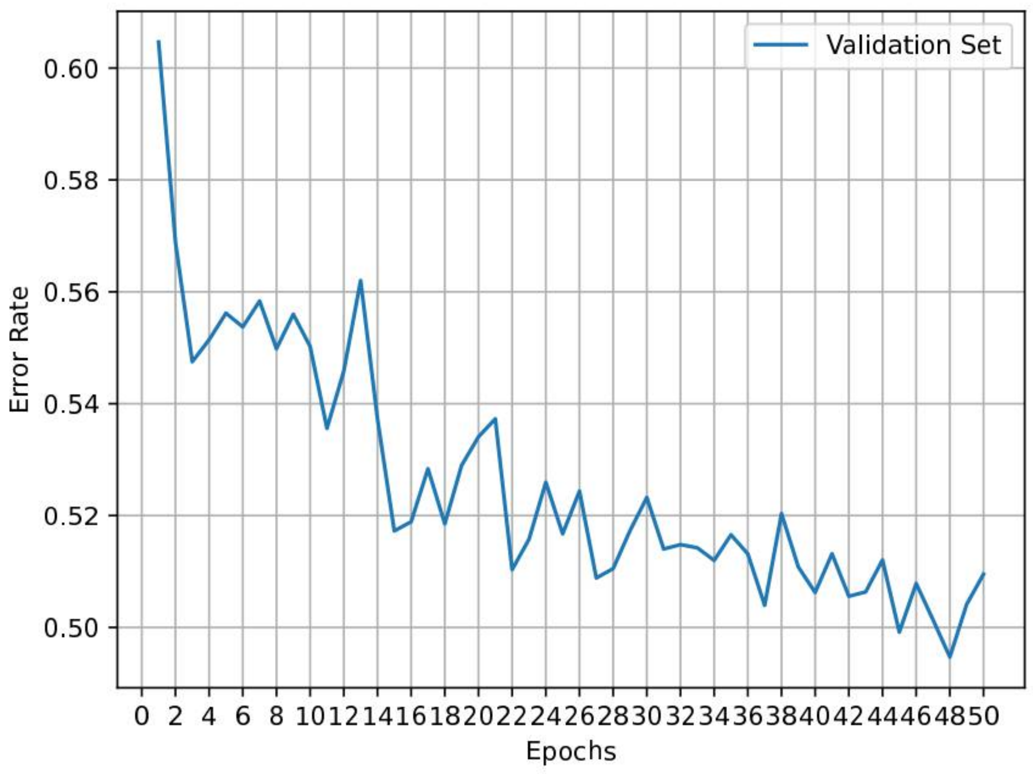

Figure 6: LSTM Validation error rate over epochs.

The final error rate in the test set was 50.23%.

3.1 (b)

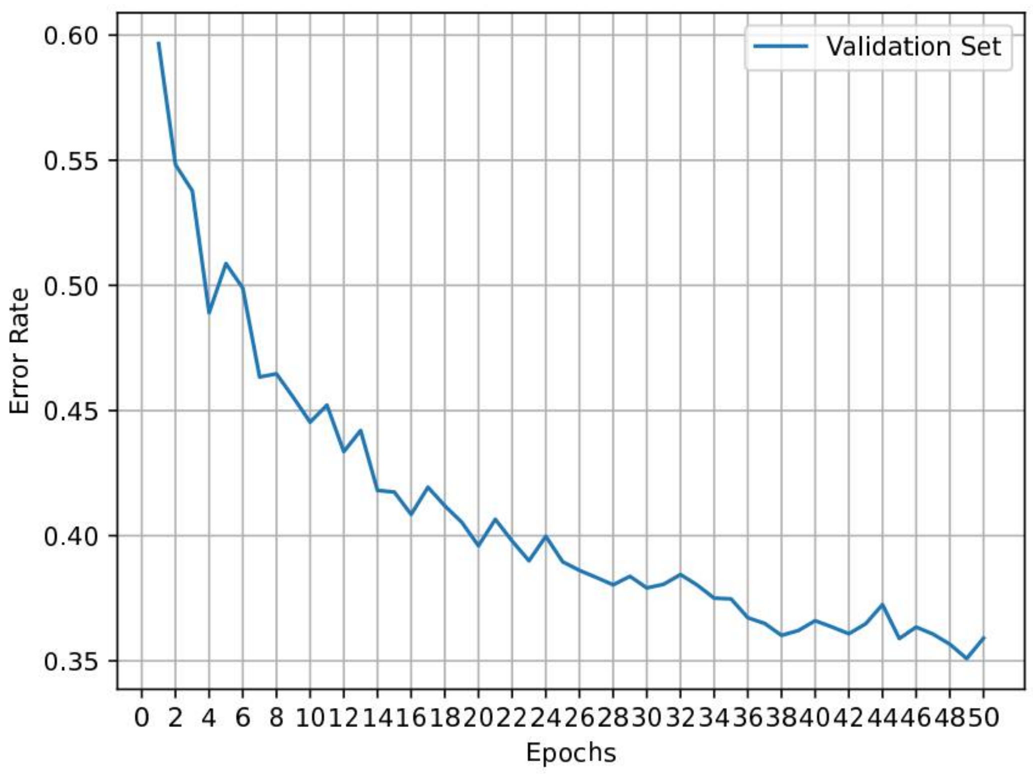

Figure 7: LSTM with attention mechanism validation error rate over epochs.

The final error rate in the test set was 38.08%.

3.1 (c)

During test time, the decoder does a while loop and, on each iteration (until it sees the stop symbol or until it reaches the maximum decoding length), predicts another word for the output.

That means that our model makes a definitive decision on each iteration, that is, it does a **greedy search**, which leads to **error propagation** - if word t is wrong, word $t+1$ will be conditioned by it and will probably also be wrong.

What we would change is to implement **beam search**, which would maintain the most probable partial translations, instead of only sticking with the best one at each step. It combines efficiency, by not exploring all possible outputs, only the most likely ones, with accuracy, by not sticking to choices that seem locally good but that are wrong in the long run.

Furthermore, we could also use an **ensemble model**, by training diverse, independent models and combining their decisions in a vote.