# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 7

November 22nd, 2022

- so far, we encountered two types of data: tabular data and image data
- for the latter, we designed specialized layers to take advantage of the regularity in them
- in other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content
- most importantly, so far, we assumed that our data are all drawn from some distribution, and all the examples are independently and identically distributed (i.i.d.)

- unfortunately, this is not true for most data
- for instance, the words in this sentence are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly
- likewise, image frames in a video, the audio signal in a conversation, and the browsing behavior on a website, all follow a *sequential* order
- it is thus reasonable to assume that specialized models for such data will do better at describing them

- another issue arises from the fact that we might not only receive a sequence as an input, but rather might be expected to continue the sequence
- for instance, the task could be to continue the series $2, 4, 6, 8, 10, \ldots$
- this is quite common in time series analysis, to predict the stock market, the fever curve of a patient, or the acceleration needed for a race car
- again, we want to have models that can handle such data
- in short, while CNNs can efficiently process spatial information, *recurrent neural networks* (*RNNs*) are designed to better handle *sequential* information
- RNNs introduce state variables to store past information, together with the current inputs, to determine the current outputs

# 4.1 Sequence models

- we need statistical tools and new deep neural network architectures to deal with sequence data
- to keep things simple, we use the stock price (FTSE 100 index) illustrated in Figure 1, as an example

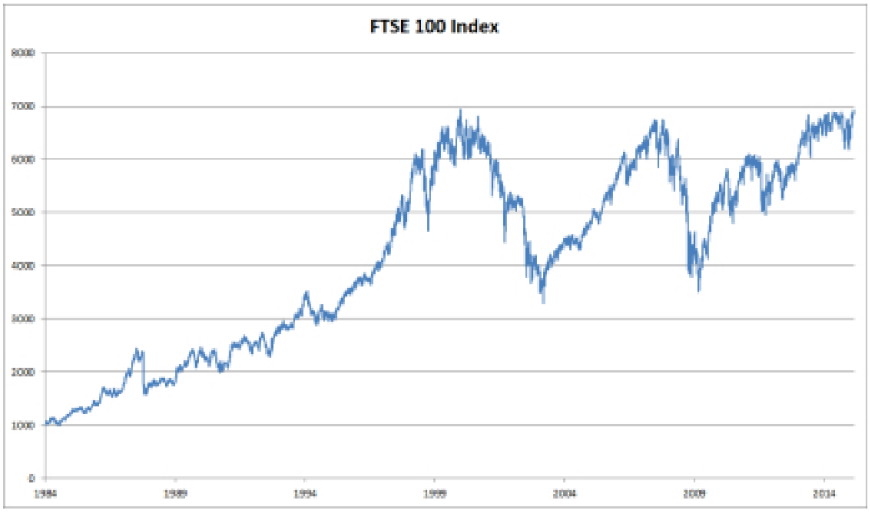


Figure 1: FTSE 100 index over about 30 years.

# 4.1 Sequence models

- let us denote the prices by $x_t$, i.e., at *time step* $t \in \mathbb{Z}^+$ we observe price $x_t$
- note that, for sequences in this chapter, $t$ will typically be discrete and vary over integers or its subset
- suppose that a trader who wants to do well in the stock market on day $t$ predicts $x_t$ via:

$$x_t \sim P(x_t | x_{t-1}, \ldots, x_1).$$

- in order to achieve this, our trader could use a regression model such as the one that we discussed in Chapter 1

- there is just one major problem: the number of inputs $x_{t-1}, \ldots, x_1$ varies, depending on $t$
- that is to say, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable
- much of what follows in this chapter will revolve around how to estimate $P(x_t|x_{t-1}, \ldots, x_1)$ efficiently
- in a nutshell, it boils down to two strategies, as follows

- first, assume that the potentially rather long sequence $x_{t-1}, \ldots, x_1$ is not really necessary
- in this case, it might be sufficient to use some timespan of length $\tau$, and only use the $x_{t-1}, \ldots, x_{t-\tau}$ observations
- the immediate benefit is that, now, the number of arguments is always the same, at least for $t > \tau$
- this allows us to train a deep network, as indicated above
- such models will be called *autoregressive models*, as they literally perform regression on themselves

- the second strategy, shown in Figure 2, is to keep some summary $h_t$ of the past observations, and, at the same time, update $h_t$ in addition to the prediction $\hat{x}_t$
- this leads to models that estimate $x_t$ with $\hat{x}_t = P(x_t|h_t)$ and, moreover, updates of the form $h_t = g(h_{t-1}, x_{t-1})$
- since $h_t$ is not observed in the data, these models are also called *latent autoregressive models*

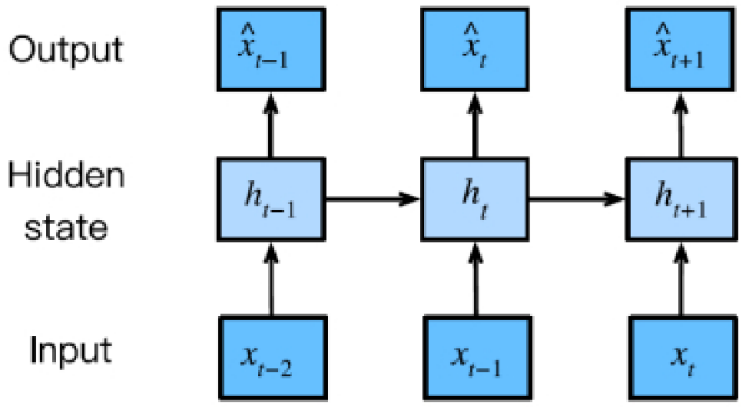


Figure 2: A latent autoregressive model.

- both cases raise the obvious question of how to generate training data
- we typically use historical observations to predict the next observation, given the ones up to right now
- obviously, we do not expect time to stand still; however, a common assumption is that, while the specific values of $x_t$ might change, at least the dynamics of the sequence itself will not
- this is reasonable, since novel dynamics are just that, novel, and thus not predictable using data that we have so far
- statisticians call dynamics that do not change *stationary*
- regardless of what we do, we will thus get an estimate of the entire sequence via:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t | x_{t-1}, \ldots, x_1).$$

- note that the above considerations still hold if we deal with discrete objects, such as words, rather than continuous numbers
- the only difference is that, in such a situation, we need to use a classifier rather than a regression model to estimate $P(x_t|x_{t-1}, \ldots, x_1)$
- recall the approximation that, in an autoregressive model, we use only $x_{t-1}, \ldots, x_{t-\tau}$ instead of $x_{t-1}, \ldots, x_1$ to estimate $x_t$
- whenever this approximation is accurate, we say that the sequence satisfies the *Markov condition*
- in particular, if $\tau = 1$, we have a *first-order Markov model*, and $P(x)$ is given by:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t|x_{t-1}) \text{ where } P(x_1|x_0) = P(x_1).$$

- such models are particularly useful whenever $x_t$ assumes only a discrete value, since, in this case, dynamic programming can be used to compute values along the chain exactly
- for instance, we can compute $P(x_{t+1}|x_{t-1})$ efficiently:

$$\begin{aligned} P(x_{t+1}|x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1}|x_t, x_{t-1})P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1}|x_t)P(x_t|x_{t-1}), \end{aligned}$$

  by using the fact that we only need to take into account a very short history of past observations: $P(x_{t+1}|x_t, x_{t-1}) = P(x_{t+1}|x_t)$
- going into details of dynamic programming is beyond the scope of this section
- control and reinforcement learning algorithms use such tools extensively

- in principle, there is nothing wrong with unfolding $P(x_1, \ldots, x_T)$ in reverse order
- after all, by conditioning, we can always write it via:

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t | x_{t+1}, \ldots, x_T).$$

- in fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too
- in many cases, however, there exists a natural direction for the data, namely going forward in time
- it is clear that future events cannot influence the past

- hence, if we change $x_t$, we may be able to influence what happens for $x_{t+1}$ going forward, but not the converse
- that is, if we change $x_t$, the distribution over past events will not change
- consequently, it should to be easier to explain $P(x_{t+1}|x_t)$ rather than $P(x_t|x_{t+1})$
- for instance, it has been shown that, in some cases, we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise $\epsilon$, whereas the converse is not true
- this is great news, since it is typically the forward direction that we are interested in estimating

# 4.1 Sequence models

- we have reviewed statistical tools and prediction challenges for sequence data; such data can take many forms
- specifically, text is one of the most popular examples of sequence data
- for example, an article can be simply viewed as a sequence of words, or even a sequence of characters
- to facilitate our future experiments with sequence data, we will explain common preprocessing steps for text
- usually, these steps are:
  1. Load text as strings into memory.
  2. Split strings into *tokens* (e.g., words and characters).
  3. Build a *table* of vocabulary to map the split tokens to numerical indices.
  4. Convert text into sequences of numerical indices so they can be manipulated by models easily.

# 4.1 Sequence models

- each text sequence is split into a list of tokens
- a *token* is the basic unit in text; in the end, a list of token lists are obtained, where each token is a string
- the string type of the token is inconvenient to be used by models, which take numerical inputs
- now, we build a dictionary, often called a *vocabulary* as well, to map string tokens into numerical indices starting from 0

- to do so, we first count the unique tokens in all the documents from the training set, namely a *corpus*, and then assign a numerical index to each unique token according to its frequency
- rarely appeared tokens are often removed to reduce the complexity
- any token that does not exist in the corpus, or has been removed, is mapped into a special unknown token “<unk>”
- we can also add a list of reserved tokens, such as “<pad>” for padding, “<bos>” to represent the beginning of a sequence, and “<eos>” for the end of a sequence

# 4.1 Sequence models

- so, we saw how to map text data into tokens, where these tokens can be viewed as a sequence of discrete observations, such as words or characters
- assume that the tokens in a text sequence of length $T$ are $x_1, x_2, \ldots, x_T$
- then, in the text sequence, $x_t$ ($1 \leq t \leq T$) can be considered as the observation or label at time step $t$
- given such a text sequence, the goal of a *language model* is to estimate the joint probability of the sequence $P(x_1, x_2, \ldots, x_T)$

# 4.1 Sequence models

- language models are incredibly useful
- for instance, an ideal language model would be able to generate natural text just on its own, simply by drawing one token at a time $x_t \sim P(x_t|x_{t-1}, \ldots, x_1)$
- all text emerging from such a model would pass as natural language, e.g., English text
- furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments
- clearly, we are still very far from designing such a system, since it would need to *understand* the text, rather than just generate grammatically correct content

- let us now apply Markov models to language modeling
- a distribution over sequences satisfies the Markov property of first order if $P(x_{t+1}|x_t, \ldots, x_1) = P(x_{t+1}|x_t)$
- higher orders correspond to longer dependencies
- this leads to a number of approximations that we could apply to model a sequence:

$$\begin{aligned} P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2|x_1)P(x_3|x_2)P(x_4|x_3), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)P(x_4|x_2, x_3). \end{aligned}$$

- the probability formulas that involve one, two, three, and, in general, $n$ variables are typically referred to as *unigram*, *bigram*, *trigram*, and *n-gram* models, respectively

- in $n$-gram models, the conditional probability of word $x_t$ at time step $t$ only depends on the $n-1$ previous words
- if we want to incorporate the possible effect of words earlier than time step $t-(n-1)$ on $x_t$, we need to increase $n$
- however, the number of model parameters would also increase exponentially with it, as we need to store $|\mathcal{V}|^n$ numbers for a vocabulary set $\mathcal{V}$
- hence, rather than modeling $P(x_t|x_{t-1},\ldots,x_{t-n+1})$, it is preferable to use a latent variable model:

$$P(x_t|x_{t-1},\ldots,x_1) \approx P(x_t|h_{t-1}),$$

where $h_{t-1}$ is a *hidden state* (also known as a *hidden variable*) that stores the sequence information up to time step $t-1$

- in general, the hidden state at any time step $t$ could be computed based on both the current input $x_t$, and the previous hidden state $h_{t-1}$:

$$h_t = f(x_t, h_{t-1}). \tag{1}$$

- for a sufficiently powerful function $f$ in (1), the latent variable model is not an approximation
- after all, $h_t$ may simply store all the data it has observed so far
- however, it could potentially make both computation and storage expensive
- recall that we have discussed hidden layers with hidden units in Chapter 2
- it is noteworthy that hidden layers and hidden states refer to two very different concepts
- hidden layers are, as explained, layers that are hidden from view, on the path from input to output
- hidden states are, technically speaking, *inputs* to whatever we do at a given step, and they can only be computed by looking at data at previous time steps

- *recurrent neural networks* (*RNNs*) are neural networks with hidden states
- before introducing the RNN model, we first revisit the MLP model introduced in Chapter 2
- let us take a look at an MLP with a single hidden layer
- let the hidden layer's activation function be $\phi$
- given a mini-batch of examples $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ with batch size $n$ and $d$ inputs, the hidden layer's output $\boldsymbol{H} \in \mathbb{R}^{n \times h}$ is calculated as:

$$\boldsymbol{H} = \phi(\boldsymbol{X}\boldsymbol{W}_{xh} + \boldsymbol{b}_h). \tag{2}$$

- in (2), we have the weight parameter $\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$, the bias parameter $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$, and the number of hidden units $h$, for the hidden layer
- thus, broadcasting is applied during the summation

- next, the hidden variable $\boldsymbol{H}$ is used as the input of the output layer
- the output layer is given by:

$$\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

  where $\boldsymbol{O} \in \mathbb{R}^{n \times q}$ is the output variable, $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$ is the weight parameter, and $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$ is the bias parameter of the output layer
- if it is a classification problem, we can use softmax($\boldsymbol{O}$) to compute the probability distribution of the output categories
- this is entirely analogous to the regression problem we discussed previously in Section 4.1
- things are entirely different when we have hidden states
- let us look at the structure in some more detail

- assume that we have a mini-batch of inputs $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ at time step $t$
- in other words, for a mini-batch of $n$ sequence examples, each row of $\boldsymbol{X}_t$ corresponds to one example at time step $t$ from the sequence
- next, denote by $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ the hidden variable of time step $t$
- unlike the MLP, here we save the hidden variable $\boldsymbol{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ to describe how to use the hidden variable of the previous time step in the current time step
- specifically, the calculation of the hidden variable of the current time step is determined by the input of the current time step, together with the hidden variable of the previous time step:

$$\boldsymbol{H}_t = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hh} + \boldsymbol{b}_h). \tag{3}$$

- compared with (2), (3) adds one more term $\boldsymbol{H}_{t-1}\boldsymbol{W}_{hh}$, and thus instantiates (1)
- from the relationship between hidden variables $\boldsymbol{H}_t$ and $\boldsymbol{H}_{t-1}$ of adjacent time steps, we know that these variables captured and retained the sequence's historical information up to their current time step, just like the state or memory of the neural network's current time step
- therefore, such a hidden variable is called a *hidden state*
- since the hidden state uses the same definition of the previous time step in the current time step, the computation of (3) is *recurrent*
- hence, neural networks with hidden states based on recurrent computation are named *recurrent neural networks* (*RNNs*)
- layers that perform the computation of (3) in *RNNs* are called *recurrent layers*

- there are many different ways for constructing RNNs; RNNs with a hidden state defined by (3) are very common
- for time step $t$, the output of the output layer is similar to the computation in the MLP:

$$\boldsymbol{O}_t = \boldsymbol{H}_t \boldsymbol{W}_{hq} + \boldsymbol{b}_q.$$

- parameters of the RNN include the weights $\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$, $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$, and the bias $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$ of the hidden layer, together with the weights $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$, and the bias $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$ of the output layer
- it is worth mentioning that, even at different time steps, RNNs always use these same model parameters
- therefore, the parameterization cost of an RNN does not grow as the number of time steps increases

- Figure 3 illustrates the computational logic of an RNN at three adjacent time steps
- at any time step $t$, the computation of the hidden state can be treated as:
  - concatenating the input $\boldsymbol{X}_t$ at the current time step $t$ and the hidden state $\boldsymbol{H}_{t-1}$ at the previous time step $t-1$
  - feeding the concatenation result into a fully-connected layer with the activation function $\phi$
- the output of such a fully-connected layer is the hidden state $\boldsymbol{H}_t$ of the current time step $t$

- in this case, the model parameters are the concatenation of $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hh}$, and a bias of $\boldsymbol{b}_h$, all from (3)
- the hidden state of the current time step $t$, $\boldsymbol{H}_t$, will participate in computing the hidden state $\boldsymbol{H}_{t+1}$ of the next time step $t+1$
- what is more, $\boldsymbol{H}_t$ will also be fed into the fully-connected output layer to compute the output $\boldsymbol{O}_t$ of the current time step $t$
- so, the calculation of $\boldsymbol{X}_t \boldsymbol{W}_{xh} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hh}$ for the hidden state is equivalent to matrix multiplication between the concatenation of $\boldsymbol{X}_t$ and $\boldsymbol{H}_{t-1}$ and the concatenation of $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hh}$

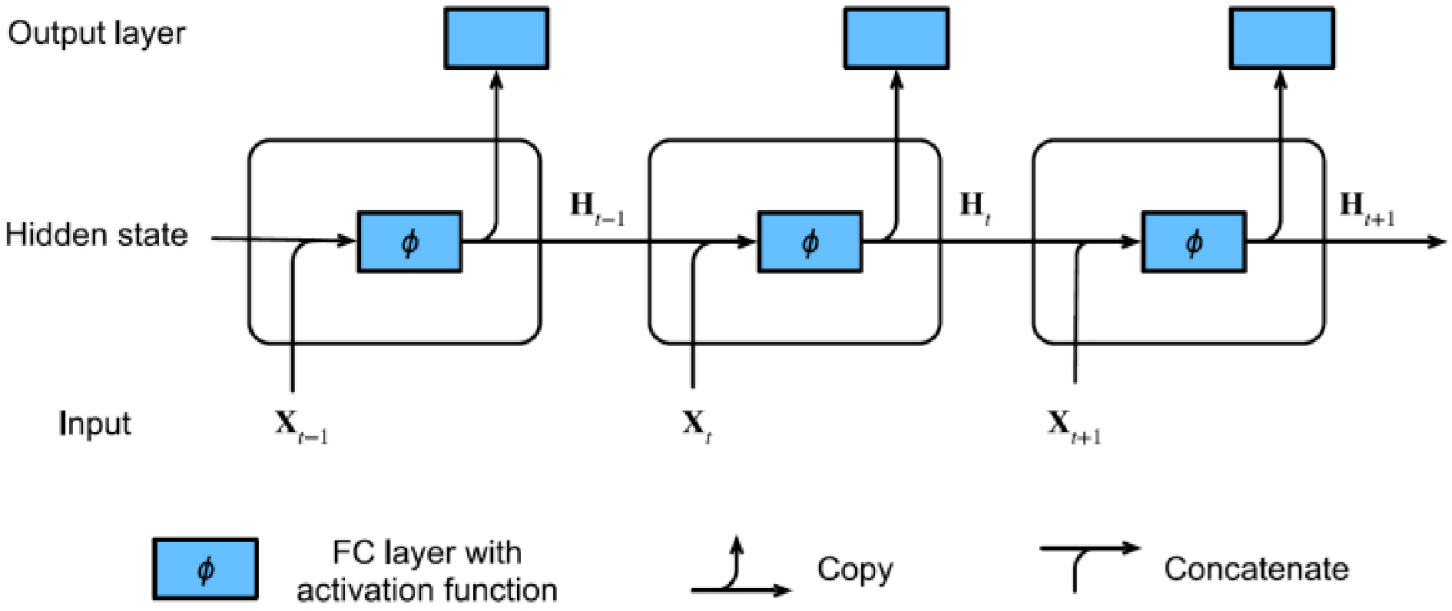


Figure 3: An RNN with a hidden state.

- for a sequence of length $T$, we will compute the gradients over $T$ time steps in an iteration, which results in a chain of matrix products with length $\mathcal{O}(T)$, during backpropagation
- as mentioned in Chapter 2, this might result in numerical instability, e.g., the gradients may either explode or vanish, when $T$ is large
- therefore, RNN models often need extra help to stabilize the training

- generally speaking, when solving an optimization problem, we take update steps for the model parameter, say in the vector form $\boldsymbol{x}$, in the direction of the negative gradient $\boldsymbol{g}$, on a mini-batch
- for example, with $\eta > 0$ as the learning rate, in one iteration, we update $\boldsymbol{x}$ as $\boldsymbol{x} - \eta \boldsymbol{g}$
- let us further assume that the objective function $f$ is well behaved, say, *Lipschitz continuous* with constant $L$
- that is to say, for any $\boldsymbol{x}$ and $\boldsymbol{y}$, we have:

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq L||\boldsymbol{x} - \boldsymbol{y}||.$$

- in this case, we can safely assume that, if we update the parameter vector by $\boldsymbol{g}$, then:

$$|f(\boldsymbol{x}) - f(\boldsymbol{x} - \eta\boldsymbol{g})| \leq L\eta||\boldsymbol{g}||,$$

  which means that we will not observe a change by more than $L\eta||\boldsymbol{g}||$
- this is both a curse and a blessing
- on the curse side, it limits the speed of making progress; whereas on the blessing side, it limits the extent to which things can go wrong if we move in the wrong direction

- sometimes, the gradients can be quite large, and the optimization algorithm may fail to converge
- we could address this by reducing the learning rate $\eta$
- but what if we only *rarely* get large gradients?
- in this case, such an approach may appear entirely unnecessary
- one popular alternative is to *clip* the *gradient* $\boldsymbol{g}$ by projecting it back to a ball of a given radius, say $\theta$, via:

$$\boldsymbol{g} \leftarrow \min\left(1, \frac{\theta}{||\boldsymbol{g}||}\right)\boldsymbol{g}.$$

- by doing so, we know that the gradient norm never exceeds $\theta$, and that the updated gradient is entirely aligned with the original direction of $\boldsymbol{g}$
- it also has the desirable side-effect of limiting the influence any given mini-batch (and within it, any given sample) can exert on the parameter vector
- this gives a certain degree of *robustness* to the model
- gradient clipping provides a quick fix to the gradient exploding
- while it does not entirely solve the problem, it is one of the many techniques to alleviate it

- recall that, for language modeling, in Section 4.1, we aim to predict the next token based on the current and past tokens, thus we shift the original sequence by one token as the labels
- Bengio et al. first proposed to use a neural network for language modeling, in 2003
- in the following, we illustrate how RNNs can be used to build a language model
- let the mini-batch size be 1, and the sequence of the text be “machine”
- to simplify things, we tokenize text into characters, rather than words, and consider a *character-level language model*
- Figure 4 demonstrates how to predict the next character based on the current and previous characters, via an RNN for character-level language modeling

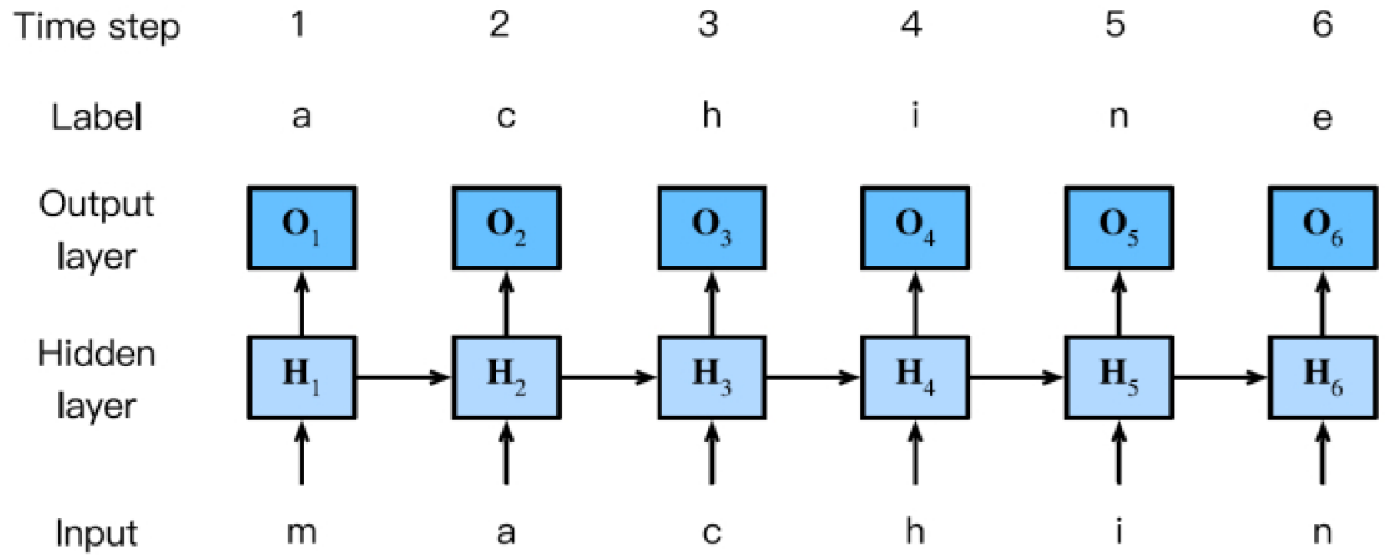


Figure 4: A character-level language model based on the RNN. The input and label sequences are "machin" and "achine", respectively.

- during the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss to compute the error between the model output and the label
- due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3 in Figure 4, $\boldsymbol{O}_3$, is determined by the text sequence “m”, “a”, and “c”
- since the next character of the sequence in the training data is “h”, the loss of time step 3 will depend on the probability distribution of the next character generated based on the feature sequence “m”, “a”, “c”, and the label “h” of this time step

- in practice, each token is represented by a $d$-dimensional vector, and we use a batch size $n > 1$
- therefore, the input $\boldsymbol{X}_t$ at time step $t$ will be a $n \times d$ matrix, which is identical to what we discussed before
- lastly, let us discuss about how to measure the language model quality, which will be used to evaluate RNN-based models
- one way is to check how *surprising* the text is
- a good language model is able to predict with high accuracy tokens that we will see next
- consider the following continuations of the phrase "It is raining", as proposed by different language models:
  1. "It is raining outside"
  2. "It is raining banana tree"
  3. "It is raining piouw;kcj pwepoiut"

- in terms of quality, example 1 is clearly the best; the words are logically coherent
- while it might not quite accurately reflect which word follows semantically (“in San Francisco” and “in winter” would have been perfectly reasonable extensions), the model is able to capture which kind of word follows
- example 2 is considerably worse, by producing a nonsensical extension
- nonetheless, at least the model has learned how to spell words, and some degree of correlation between words
- lastly, example 3 indicates a poorly trained model that does not fit the data properly

- we might measure the quality of the model by computing the likelihood of the sequence
- unfortunately, this is a number that is hard to understand and difficult to compare
- after all, shorter sequences are much more likely to occur than the longer ones, hence evaluating the model on a long novel will inevitably produce a much smaller likelihood than on a short novella
- what is missing is the equivalent of an average

- information theory is useful here; we have defined entropy and cross-entropy when we introduced the softmax regression, in Chapter 1
- if we want to compress text, we can ask about predicting the next token, given the current set of tokens
- a better language model should allow us to predict the next token more accurately
- thus, it should allow us to spend fewer bits in compressing the sequence

- so, we can measure it by the cross-entropy loss averaged over all the $n$ tokens of a sequence:

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t|x_{t-1},\ldots,x_1), \tag{4}$$

  where $P$ is given by a language model and $x_t$ is the actual token observed at time step $t$ from the sequence
- this makes the performance on documents of different lengths comparable
- for historical reasons, scientists in natural language processing prefer to use a quantity called *perplexity*
- in a nutshell, it is the exponential of (4):

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t|x_{t-1},\ldots,x_1)\right).$$

- perplexity can be best understood as the harmonic mean of the number of real choices that we have when deciding which token to pick next
- let us look at a number of cases:
  - In the best case scenario, the model always perfectly estimates the probability of the label token as 1. In this case, the perplexity of the model is 1.
  - In the worst case scenario, the model always predicts the probability of the label token as 0. In this situation, the perplexity is positive infinity.
  - At the baseline, the model predicts a uniform distribution over all the available tokens of the vocabulary. In this case, the perplexity equals the number of unique tokens of the vocabulary. In fact, if we were to store the sequence without any compression, this would be the best we could do to encode it. Hence, this provides a nontrivial upper bound that any useful model must beat.

Thank you!